MAHAVEER INSTITUTE OF SCIENCE AND TECHNOLOGY

(AN UGC AUTONOMOUS INSTITUTION)

Approved by AICTE, Affiliated to JNTUH, Accredited by NAAC with 'A' Grade
Recognized Under Section 2(f) of UGC Act 1956, ISO 9001:2015 Certified
Vyasapuri, Bandlaguda, Post: Keshavgiri, Hyderabad- 500 005, Telangana, India.
https://www.mist.ac.in E-mail:principal.mahaveer@gmail.com, Mobile: 8978380692

Department of Computer Science and Engineering (AIML)

(R18) AUTOMATA THEORY AND COMPILER DESIGN

Lecture Notes

B. Tech II YEAR – II SEM

Prepared by

SANGYAM SOUNDARYA
(Assistant Professor)
Dept.CSE(AIML)

PART-I

INTRODUCTION TO FINITE AUTOMATA

STRUCTURAL REPRESENTATIONS

- An automaton with a finite number of states is called finite automaton or finite state machine.
- The block diagram of finite automaton consist of falling three major components

  - *Input tape*
  - *Read/write head*
  - *Finite control*

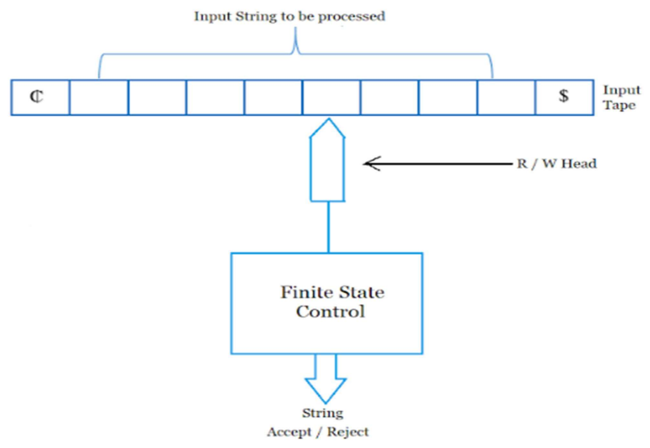


Fig: Block Diagram of Finite Automata

I] Input Tape:

- The input tape is divided into squares, each square contains a single symbol from input alphabet ∑
- The end squares of each tape contain end markers ¢ at left end and $ at right end.
- Absence of end markers indicates that tape is of infinite length.
- The left-to-right sequence of symbols between end markers is the input string to be processed.

II] Read/Write Head:

- The R/W head examines only one square at a time and can move one square either to the left or the right.
- For further analysis, we restrict the movement of R/W head only to the right side.

III] Finite State Control:

- The finite state control is responsible for controlling total functioning of finite automata machine.
- It will decide the which input symbol is read next and where to move either to the left or right.
- The input to the finite control will be usually
    - Input symbol from input tape
    - Present state of machine
- The output may be
    - Movement of R/W head along the tape to the next square or to null move
    - The next state/new state of FSM

AUTOMATA AND COMPLEXITY

Automata theory is a branch of computer science that deals with designing abstract self-propelled computing devices that follow a predetermined sequence of operations automatically. An automaton with a finite number of states is called a Finite Automaton.

- Set of Input symbols
- Configuration states
- Output

Formal definition of a Finite Automaton

An automaton can be represented by a 5-tuple (Q, ∑, δ, q_0, F), where −

- **Q** is a finite set of states.
- ∑ is a finite set of symbols, called the **alphabet** of the automaton.
- **δ** is the transition function.
- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

- **Context-Free Grammars (CFGs):** They are more powerful abstract models than FA and are essentially used in the programming languages and natural language research work.
- **Turing Machines:** They are abstract models for real computers having an infinite memory (in the form of a tape) and a reading head. They form much more powerful computation models than FA, CFGs, and Regular Expressions.

COMPLEXITY THEORY

Studying the cost of solving problems while focusing on resources (time & space) needed as the metric. The running time of an algorithm varies with the inputs and usually grows with the size of the inputs.

MEASURING COMPLEXITY

Measuring complexity involves an algorithm analysis to determine how much time it takes while solving a problem (time complexity). To evaluate an algorithm, a focus is made on relative rates of growth as the size of the input grows.

Since the exact running time of an algorithm often is a complex expression, we usually just estimate it. We measure an algorithm's time requirement as a function of the input size (n) when determining the time complexity of an algorithm.

As T(n), the time complexity is expressed using the Big O notation where only the highest order term in the algebraic expressions are considered while ignoring constant values.

The common running times when analyzing algorithms are:

- O(1) - Constant time or constant space regardless of the input size.
- O(n) - Linear time or linear space, where the requirement increases uniformly with the size of the input.
- O(log n) - Logarithmic time, where the requirement increases in a logarthimic nature.
- O(n^2) - Quadratic time, where the requirement increases in a quadratic nature.

This analysis is based on 2 bounds that can be used to define the cost of each algorithm.

They are:

1. Upper (*Worst Case Scenario*)
2. Lower (*Best Case Scenario*)

The major classifications of complexities include:

- *Class P:* The class P consists of those problems that are solvable in polynomial time. These are problems that can be solved in time O(n^k) for some constant k where n is the input size to the problem. It is devised to capture the notion of efficient computation.
- *Class NP:* It forms the class of all problems whose solution can be achieved in polynomial time by non-deterministic Turing machine. NP is a complexity class used to classify decision problems.

A major contributor to the complexity theory is the complexity of the algorithm used to solve the problem. Among several algorithms used in solving computational problems are those whose complexity can range from fairly complex to very complex.

The more complex an algorithm, the more computational complexity will be in a given problem.

THE CENTRAL CONCEPTS OF AUTOMATA THEORY

1. Symbols:

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

Example: **1, a, b, #**

2. Alphabets:

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

Examples:

1. ∑ = {a, b}

2. ∑ = {A, B, C, D}
3. ∑ = {0, 1, 2}
4. ∑ = {0, 1, ..., 5]
5. ∑ = {#, β, Δ}

3.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

∑ = {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:

1. w = 010
2. Number of Sting |w| = 3

3. Language:

A language is a collection of appropriate string. A language which is formed over Σ can be **finite** or **infinite**.

Example: 1 (Finite Language)

L1 = {Set of string of length 2}

= {aa, bb, ba, bb}

Example: 2 (Infinite Language)

L2 = {Set of all strings starts with 'a'}

= {a, aa, aaa, abb, abbb, ababb}

PART- II

NON-DETERMINISTIC FINITE AUTOMATA

FORMAL DEFINATION

- The concepts of non-deterministic finite automata is exactly reverse of deterministic finite automata (DFA).
- The finite automata is called NFA when there exists many paths for a specific input from current state to next state.
- Thus it is not fixed or determined that with particular input where to go next. Hence NDFA
- There can be multiple final states.
- NFA are more flexible and easier to use than DFA.
- Every DFA is NFA but every NFA is not DFA.
- As it has finite number of states, the machine is called non-deterministic finite machine or non-deterministic finite automaton.
- Examples: War, weather, politics, lottery, gambling, sports etc there are various example in real life that can be related to NFA.

Graphical Representation:

- NFA is represented by digraphs called state diagram or transition diagram
- The vertices represent the states
- The arcs/ edges labelled with an input symbols shows the transitions
- Initial state is denoted by single circle and arrow pointing towards it(i.e. incoming arrow)
- The final state is denoted by two concentric circles

Formal definition of NFA:

➢ NFA can be represented by 5-tuple or Quintuple machine

$M = (Q, \Sigma, \delta, q_0, F)$, where –

- **Q** is a finite set of states.
- **∑** is a finite set of symbols, called the **input alphabet** of the automaton.
- **δ** is the transition function/mapping function which maps

  $\mathbf{Q \times \Sigma \rightarrow 2^Q}$ **(Here x is cartesian Product)**

  Above mapping is usually represented by

  1. Transition function/Mapping function
  2. Transition diagram / Transition graph / Transition State diagram/ State diagram
  3. Transition table/Transition state table/Mapping table

  (Here power set of Q has been taken because in case of NFA ,from current state of machine, the transition can occur to any combination of Q states)

- $\mathbf{q_0}$ is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

➢ Example : Let a NFA can be Q={A,B,C}, ∑={0,1}, q_0={A},F={C} and δ is given by the table

Present State	Next State	
	Input 0	Input 1
→A	{A,B}	{B}
B	{C}	{A,C}
*C	{B,C}	{C}

SOLUTION:

1)Transition function:

By using the below formula we can easily write the transition functions

δ(Present State, Input Symbol)=Next State

δ(A, 0)= {A,B} ,δ (A,1)= {B}

δ(B, 0)= {C} ,δ (B,1)= {A,C}

δ(C, 0)= {B,C} ,δ (C,1)= {C}

2)Transition diagram:

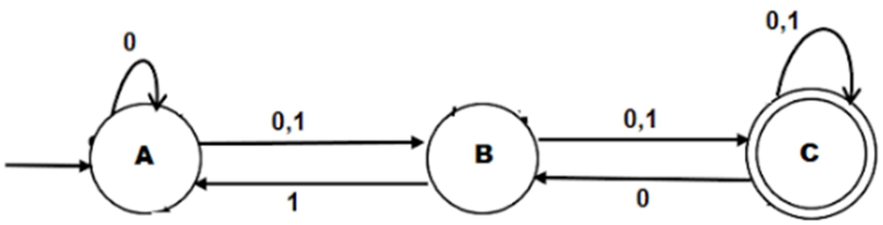


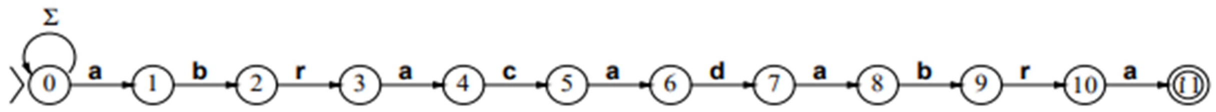
Fig: Transition diagram

APPLICATIONS OF NFA

- NFAs used to reduce the complexity of the mathematical work required to establish many important properties in the theory of computation.
- NFAs are highly applicable to prove closure properties of regular languages in much easier way compare to DFAs.
- Finite State Programming
- Event Driven Finite State Machine (FSM)
- Virtual FSM
- DFA based text filter in Java
- Acceptors and Recognizers
- Transducers
- UML state diagram
- Hardware Applications

TEXT SEARCH:

Consider pattern P = "abracadabra". Figure 2 shows a Nondeterministic Finite Automaton (NFA) that recognizes the language Σ∗P, that is, strings finishing with P. As it can be seen, the automaton has a very regular structure.



In above mentioned figure an NFA to search for "abracadabra". The initial state is marked with “i” and the final state is double-circled. This NFA can be used for text searching as follows: feed it with the characters of T. Each time it recognizes a word, it means that we have read a string in the set Σ∗P, or which is the same, we have found the pattern in the text. Then we can report every occurrence of P in T. Since the NFA has m + 1 states, its simulation over n text characters takes O(mn) time, in the worst and average case. The reason is that, in principle, each NFA state can be active or inactive and we have to update them all. This complexity is not appealing if we compare it against the naive algorithm. A clear option to reduce search time is to make the automaton deterministic (DFA). In this case, there will be only one active state at the time, and processing the automaton over the text will take O(n) search time. Figure 3 shows the DFA for the same pattern. As it can be seen, its regularity has been lost. To the previous “forward” arrows, a number of “backward” arrows have been added, which handle the case where the next text character is not the one we expect to match P.

FINITE AUTOMATA WITH EPSILON(Ɛ) TRANSITIONS

Epsilon NFA (∈-NFA) is similar to the NFA but have just a minor difference which is epsilon (∈) move. The transitions without consuming an input symbol are called ∈-transitions.

- In the state diagrams, they are usually labeled with the Greek letter ∈ also called lamda.
- ∈-transitions provide a convenient way of modeling the systems
- Due to empty move, the first string of language may be an empty or epsilon.

***Note**: ∈ab∈a = aba, where ∈ is empty.*

Formal Definition of Epsilon-NFA

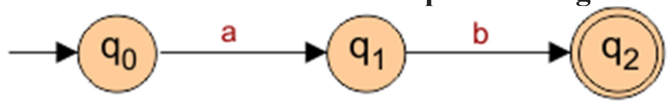
The formal definition of ∈-NFA is represented through 5-tuple (Q, ∑, δ, q_0, F) where,

- **Q** is a finite set of all states ($q_0, q_1, q_2, \dots q_n$) where n is finite number
- ∑ is a finite set of symbols called the alphabet. i.e. {0, 1},
- **δ : Q x (∑ U ∈) → 2^Q is a total function called as transition function**
- **q_0** is the initial state from where any input is processed (q_0 ∈ Q).

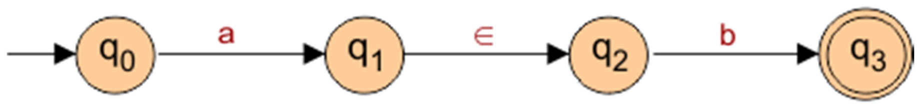
- **F** is a set of final state/states where F will be subset (⊆) of Q

Examples of Epsilon-NFA
There are various examples of epsilon-NFA. let explain some of them.

Example 01
Draw a Finite Automata which accept the string "ab".

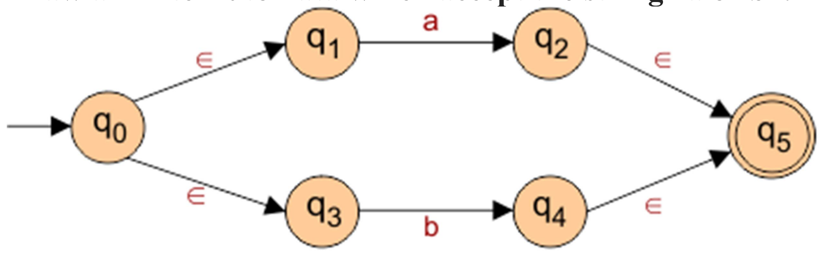


NFA



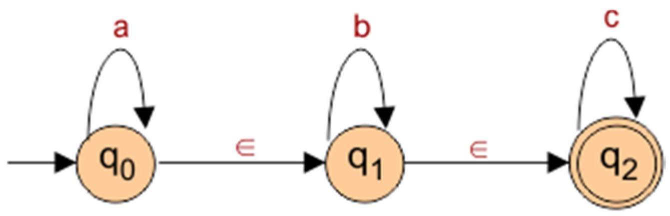
∈-NFA

Example 02
Draw a Finite Automata which accept the string "a or b".



∈-NFA

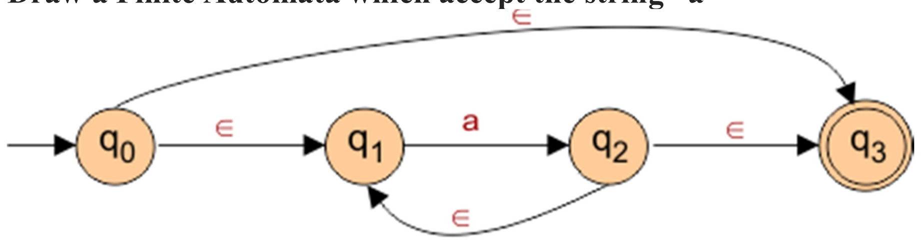
Example 03
Draw a Finite Automata which can accept the string "a or b or c".



∈-NFA

Example 04

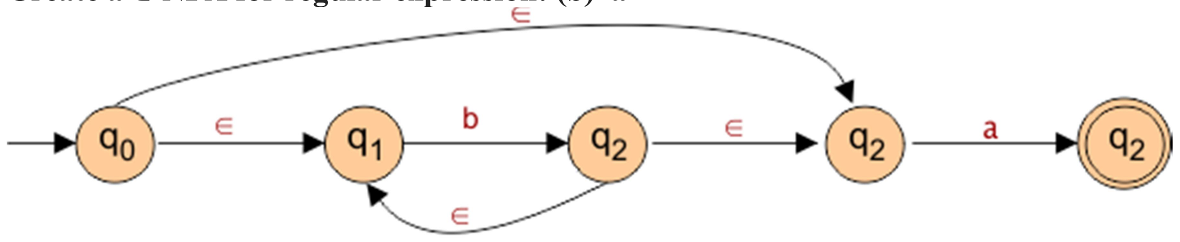
Draw a Finite Automata which accept the string “a*”



∈-NFA

Example 05

Create a ∈-NFA for regular expression: (b)*a



∈-NFA

UNIT I

PART III

DIFFERENCE BETWEEN NFA AND DFA

S.NO	DFA	NFA
1.	DFA stands for Deterministic Finite Automata.	NFA stands for Non-Deterministic Finite Automata.
2.	When processing a string in DFA, there is always a unique state to go next when each character is read. It is because, for each state in DFA, there is exactly one state that corresponds to each character being read.	In NFA several choices may exist for the next state. Can move to more than one state.
3.	DFA cannot use empty string transition.	NFA can use empty string transition.
4.	In DFA we cannot move from one state to another without consuming a symbol.	NFA allows € (null) as the second argument of the transition function. This means that the NFA can make a transition without consuming an input symbol.
5.	For every symbol of the alphabet, there is only one state transition in DFA.	We do not need to specify how does the NFA react according to some symbol.
6.	DFA can be understood as one machine.	NFA can be understood as multiple title machines computing at the same time.
7.	DFA will reject the string if it ends at other than accepting the state	If all the branches of NFA die or reject the string, we can say that NFA rejects the string.
8.	It is more difficult to construct DFA.	NFA is easier to construct.
9.	DFA requires more space.	NFA requires less space.
10.	For every input and output, we can construct a DFA machine.	It is not possible to construct an NFA machine for every input and output.
11.	All DFA is NFA.	Not all NFA is DFA.
12.	Backtracking is allowed in DFA	backtracking is not allowed in NFA

DEFINATION OF DFA

- The finite automata is called deterministic finite automata, if there is only one path for specific input from current state to next state.
- In DFA, for each input symbol, one can determine the state which machine will move.
- As it has finite number of states, the machine is called deterministic finite machine or deterministic finite automata

Graphical Representation:

- A DFA is represented by digraphs called state diagram or transition diagram the vertices represent the states
- The arcs/ edges labelled with an input symbols shows the transitions
- Initial state is denoted by single circle and arrow pointing towards it(i.e. incoming arrow)
- The final state is denoted by two concentric circles

Formal definition of DFA:

- A DFA can be represented by 5-tuple or Quintuple machine

$M = (Q, \Sigma, \delta, q_0, F)$, where –

- **Q** is a finite set of states.
- **∑** is a finite set of symbols, called the **input alphabet** of the automaton.
- **δ** is the transition function/mapping function which maps

  $Q \times \Sigma \rightarrow Q$ (Here x is cartesian Product)

  Above mapping is usually represented by

  1. Transition function/Mapping function
  2. Transition diagram / Transition graph / Transition State diagram/ State diagram
  3. Transition table/Transition state table/Mapping table

- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

POINTS TO REMEMBER:

To design FA ,we need to consider following points

1) **No. of states= Length of string +1 ==> Applicable only for DFA**

2)A string is said to be accepted by FA,if and only if,after applying that string if we are able to reach from initial state to final state.Otherwise it is rejected.

3)If language is finite language then state diagram of DFA will contain Dead state/Trap state

4)If language is infinite language then state diagram of DFA will contain dead state and loops as well.

- Example1 : Let a DFA can be Q={A,B,C}, ∑={0,1}, q_0={A},F={C} and δ is given by the table

Present State	**Next State**	
	Input 0	Input 1
→A	A	B
B	C	A
*C	B	C

SOLUTION:

Given Q={A,B,C}, ∑={0,1}, q_0={A},F={C} and δ is given by the table

Present State	**Next State**	
	Input 0	Input 1
→A	A	B
B	C	A
*C	B	C

1)Transition function:

By using the below formula we can easily write the transition functions

δ(Present State, Input Symbol)=Next State

δ(A, 0)= A ,δ (A,1)= B

δ(B, 0)= C ,δ (B,1)= A

δ(C, 0)= B ,δ (C,1)= C

2)Transition diagram:

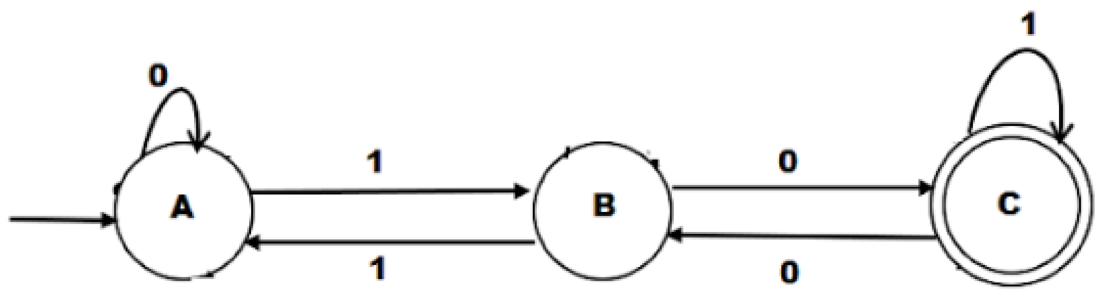


Fig: Transition diagram

HOW A DFA PROCESS STRINGS

Acceptors, Classifiers, and Transducers

Acceptor (Recognizer)

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

Classifier

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

Transducer

An automaton that produces outputs based on current input and/or previous state is called a transducer.

Transducers can be of two types –

• Mealy Machine − The output depends both on the current state and the current input.

• Moore Machine − The output depends only on the current state.

Acceptability by DFA and NDFA

A string is accepted by a DFA/NDFA iff the DFA/NDFA starting at the initial state ends in an accepting state (any of the final states) after reading the string wholly.

- **A string S is accepted by a DFA/NDFA (Q, ∑, δ, q0, F), iff**

δ*(q0, S) ∈ F

- **The language L accepted by DFA/NDFA is**

{S | S ∈ ∑* and δ*(q0, S) ∈ F}

- **A string S′ is not accepted by a DFA/NDFA (Q, ∑, δ, q0, F), iff**

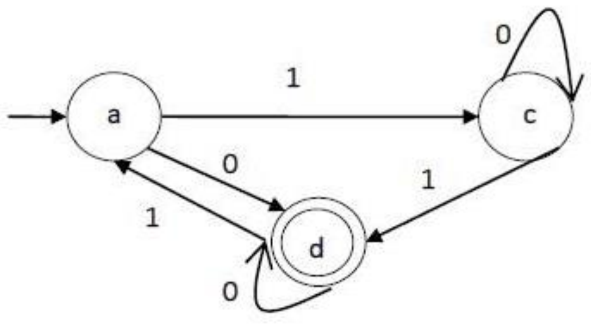
δ*(q0, S′) ∉ F

- **The language L′ not accepted by DFA/NDFA (Complement of accepted language L) is**

{S | S ∈ ∑* and δ*(q0, S) ∉ F}

Example

Let us consider the DFA shown in Figure 1.3. From the DFA, the acceptable strings can be derived



Strings accepted by the above DFA: {0, 00, 11, 010, 101, ...}

Strings not accepted by the above DFA: {1, 011, 111, ...}

TH LANGUAGE OF DFA

A DFA is a five-tuple:

M = (Q, Σ, δ, q0, F)

Q A finite set of states

Σ A finite input alphabet

q0 The initial/starting state, q0 is in Q

F A set of final/accepting states, which is a subset of Q

δ A transition function, which is a total function from Q x Σ to Q

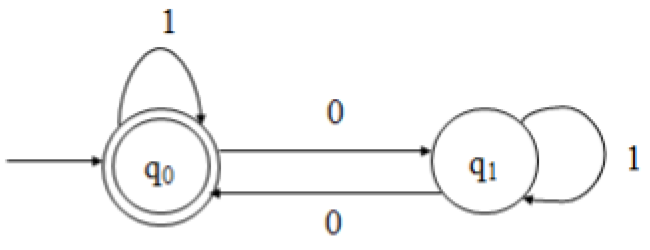
δ: (Q x Σ) –> Q

δ is defined for any q in Q and s in Σ, and δ(q,s) = q' is equal to another state q' in Q.

Intuitively, δ(q,s) is the state entered by M after reading symbol s while in state q

For Example #1:

$Q = \{q_0, q_1\}$
$\Sigma = \{0, 1\}$
Start state is q_0
$F = \{q_0\}$



δ:

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

• **Let M = (Q, Σ, δ, q , F) be a DFA and let w be in Σ*.** Then w is accepted by M iff

δ(q ,w) = p for some state p in F.

• **Let M = (Q, Σ, δ, q , F) be a DFA.** Then the language accepted by M is the set:

0 L(M) = {w | w is in Σ* and δ(q ,w) is in F}

• Another equivalent definition: **L(M) = {w | w is in Σ* and w is accepted by M}**

• Let L be a language. Then L is a regular language iff there exists a DFA M such that **L = L(M).**

• **Notes:**

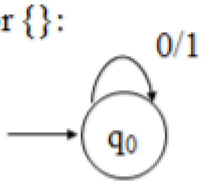
– A DFA M = (Q, Σ, δ,q0,F) partitions the set Σ* into two sets: L(M) and Σ* - L(M).

– If L = L(M) then L is a subset of L(M) and L(M) is a subset of L.

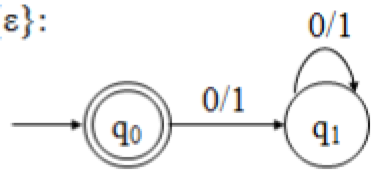
– Similarly, if L(M1) = L(M2) then L(M1) is a subset of L(M2) and L(M2) is a subset of L(M1)

- Let $\Sigma = \{0, 1\}$. Give DFAs for $\{\}$, $\{\varepsilon\}$, Σ^*, and Σ^+.

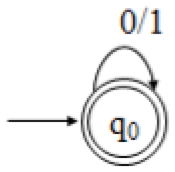
For $\{\}$:



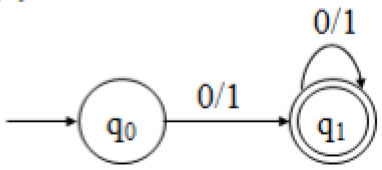
For $\{\varepsilon\}$:



For Σ^*:

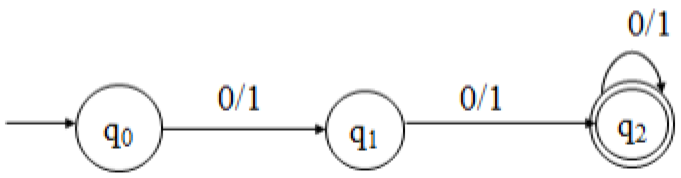


For Σ^+:

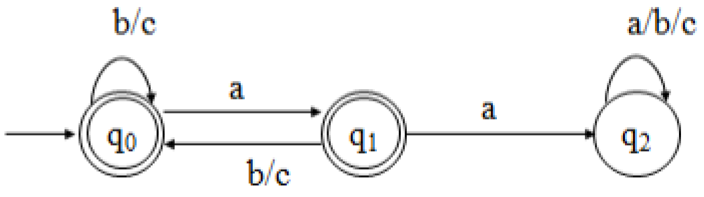


Give a DFA M such that:

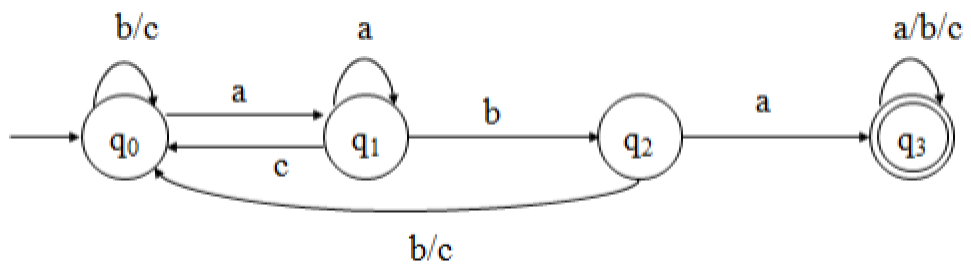
L(M) = {x | x is a string of 0's and 1's and |x| >= 2}



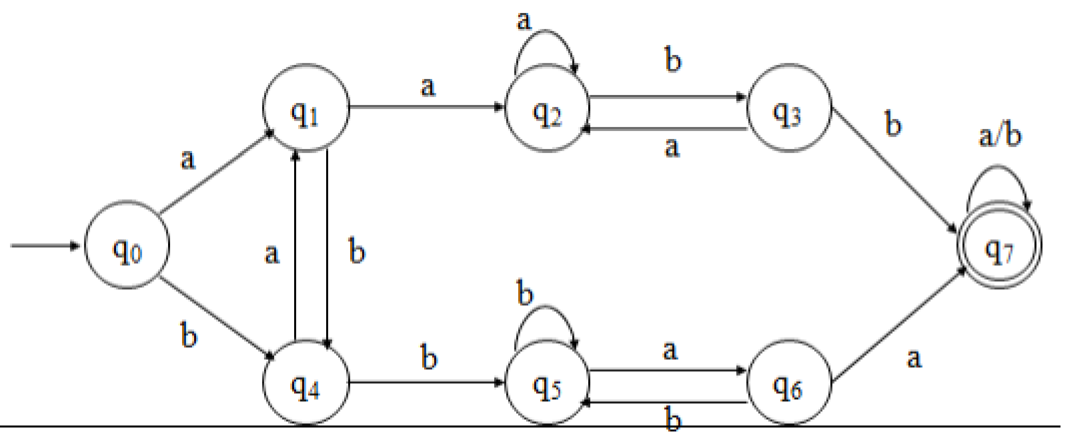
L(M) = {x | x is a string of (zero or more) a's, b's and c's such that x does not contain the substring *aa*}



L(M) = {x | x is a string of a's, b's and c's such that x contains the substring *aba*}



L(M) = {x | x is a string of a's and b's such that x contains both *aa* and *bb*}

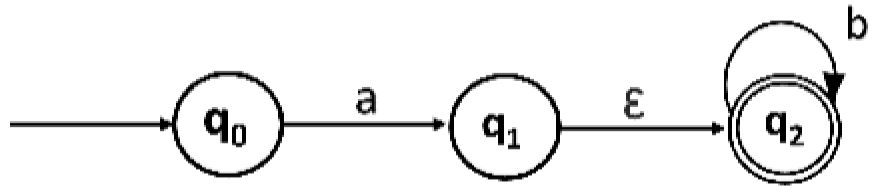


CONVERSION OF NFA WITH Ɛ-TRANSITIONS TO NFA WITHOUT Ɛ-TRANSITIONS

NFA with ε can be converted to NFA without ε, and this NFA without ε can be converted to DFA. To do this, we will use a method, which can remove all the ε transition from given NFA. The method will be:

1. Find out all the ε transitions from each state from Q. That will be called as εclosure{q1} where qi ∈ Q. 2. Then δ' transitions can be obtained. The δ' transitions mean a ε-closure on δ moves.
3. Repeat Step-2 for each input symbol and each state of given NFA.
4. Using the resultant states, the transition table for equivalent NFA without ε can be built.

EXAMPLE
Convert the following NFA with ε to NFA without ε.



Solutions: We will first obtain ε-closures of q0, q1 and q2 as follows:

ε-closure(q0) = {q0}
ε-closure(q1) = {q1, q2}
ε-closure(q2) = {q2}
Now the δ' transition on each input symbol is obtained as:

δ'(q0, a) = ε-closure(δ(δ^(q0, ε),a))
= ε-closure(δ(ε-closure(q0),a))
= ε-closure(δ(q0, a))
= ε-closure(q1)
= {q1, q2}

δ'(q0, b) = ε-closure(δ(δ^(q0, ε),b))
= ε-closure(δ(ε-closure(q0),b))
= ε-closure(δ(q0, b))
= Φ
Now the δ' transition on q1 is obtained as:
δ'(q1, a) = ε-closure(δ(δ^(q1, ε),a))
= ε-closure(δ(ε-closure(q1),a))
= ε-closure(δ(q1, q2), a)
= ε-closure(δ(q1, a) ∪ δ(q2, a))
= ε-closure(Φ ∪ Φ)
= Φ
δ'(q1, b) = ε-closure(δ(δ^(q1, ε),b))
= ε-closure(δ(ε-closure(q1),b))
= ε-closure(δ(q1, q2), b)
= ε-closure(δ(q1, b) ∪ δ(q2, b))
= ε-closure(Φ ∪ q2)
= {q2}

The δ' transition on q2 is obtained as:

δ'(q2, a) = ε-closure(δ(δ^(q2, ε),a))
= ε-closure(δ(ε-closure(q2),a))
= ε-closure(δ(q2, a))
= ε-closure(Φ)
= Φ

δ'(q2, b) = ε-closure(δ(δ^(q2, ε),b))

= ε-closure(δ(ε-closure(q2),b))
= ε-closure(δ(q2, b))
= ε-closure(q2)
= {q2}

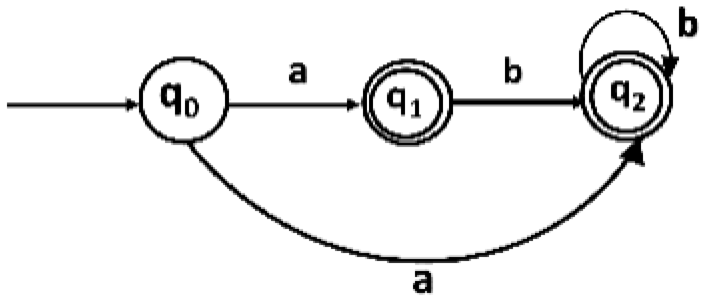
Now we will summarize all the computed δ' transitions:

δ'(q0, a) = {q0, q1}
δ'(q0, b) = Φ
δ'(q1, a) = Φ
δ'(q1, b) = {q2}
δ'(q2, a) = Φ
δ'(q2, b) = {q2}

The transition table can be:

States	a	b
→q0	{q1, q2}	Φ
*q1	Φ	{q2}
*q2	Φ	{q2}

State q1 and q2 become the final state as ε-closure of q1 and q2 contain the final state q2. The NFA can be shown by the following transition diagram:

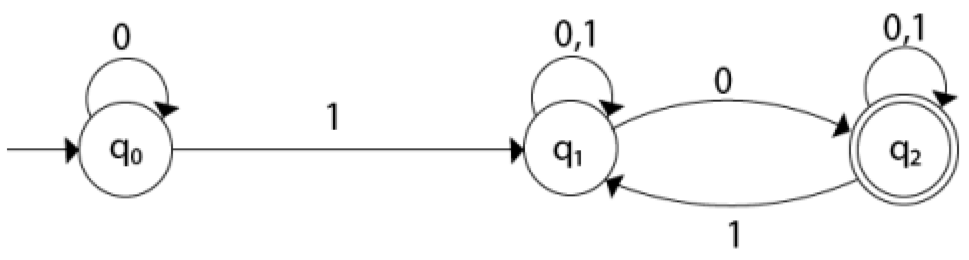


CONVERSION OF NFA TO DFA

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 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)

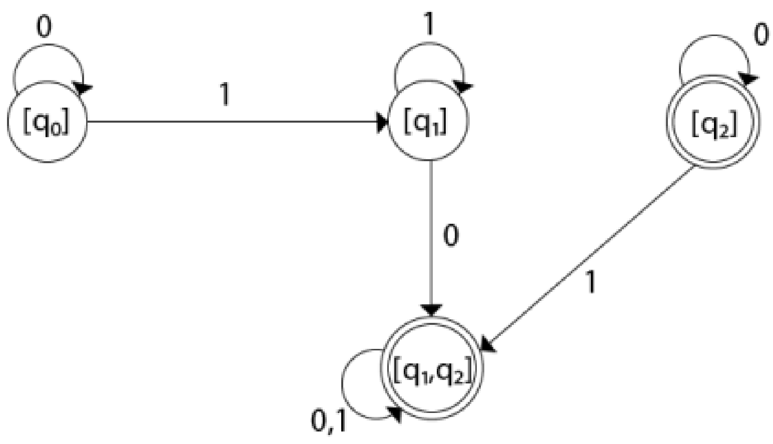
$\delta'([q1], 1) = [q1]$

The δ' transition for state q2 is obtained as:
$\delta'([q2], 0) = [q2]$
$\delta'([q2], 1) = [q1, q2]$

Now we will obtain δ' transition on [q1, q2].
$\delta'([q1, q2], 0) = \delta(q1, 0) \cup \delta(q2, 0)$
$= \{q1, q2\} \cup \{q2\}$
$= [q1, q2]$
$\delta'([q1, q2], 1) = \delta(q1, 1) \cup \delta(q2, 1)$
$= \{q1\} \cup \{q1, q2\}$
$= \{q1, q2\}$
$= [q1, q2]$
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.

MOORE AND MELAY MACHINES

Introduction

A finite automaton (FA) is a simple idealized machine used to recognize patterns within input taken from some character set. It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. There are two types of Finite State Automata-

Moore Machine (Output on State)
Mealy Machine (Output on Transition)

A **Moore Machine** is a finite-state machine whose output depends only on a state, i.e., the current state. It is efficient in simplifying a given behaviour. In a State Transition Diagram, each state is labelled with an output value. The name 'Moore' came from 'Edward F. Moore'.

Formal Definition of Moore Machine

Moore and mealy machines accept all regular languages. The output depends only on the current state of the machine. The output is represented by current state separated by “/”.
Moore machine can be described by 6 tuples (Q, q0, ∑, O, δ, λ) where,

- Q: finite set of states
- q0: initial state of machine
- ∑: finite set of input symbols
- O: output alphabet
- δ: transition function where $Q \times \sum \rightarrow Q$
- λ: output function where $Q \rightarrow O$

A **Mealy Machine** is a finite-state machine whose output depends on present input and a state, i.e., the current state. It is efficient in reducing the number of states. In a State Transition Diagram, each transition is labeled with an output value. In this, every transition for a particular input has a fixed output.

Tuples of Mealy Machine

Mealy Machine can be described by 6 tuples (Q, q0, ∑, O, δ, λ'). Let explain above Mealy Machine where

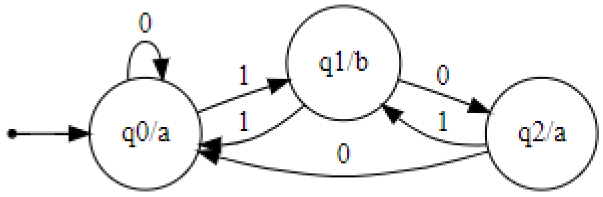
- q0 and q1 are states (Q).
- q0 is the initial state.
- Input alphabets (∑) are "0" and "1".
- Output alphabets (O) are "a" and "b".
- Transition function (Q × ∑ →Q) and output function (Q × ∑ →O)

Conversion of moore to mealay with examples

EXAMPLE 1

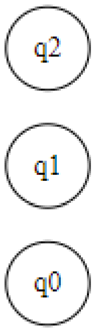
The steps involved in this method are:

1. To redraw the original states without the output. The starting state remains the same.
2. Draw the transitions as they are on the original machine.
3. Add output to the transitions based on their destination's output symbol in the original machine.



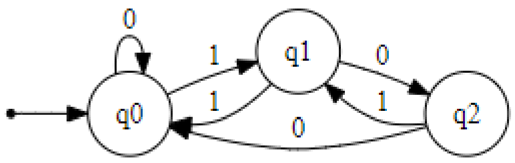
An example Moore machine

1. We redraw the states without the output:



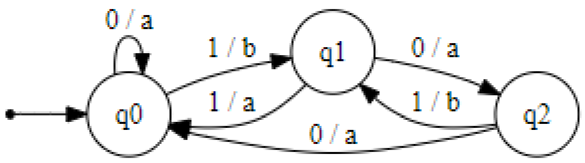
Step 1: Redraw the states

2. Draw transitions:



Step 2: Add transitions

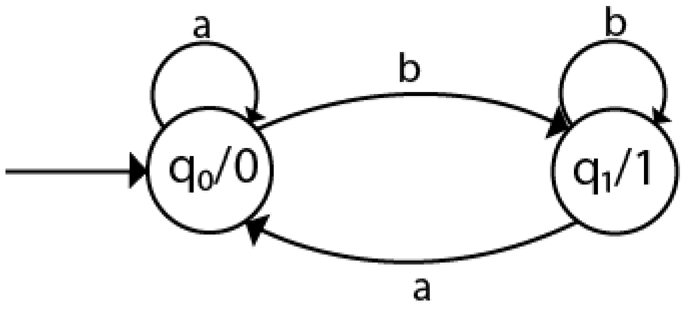
3. Now we add output to the transitions based on the transition's destination:



Step 3 : Add output to transitions. This is the final Mealy machine.

Note: The output of the transition $q0 \rightarrow q1$ is b, because the transition lands on $q1$ which had an output b in the original Moore machine.

EXAMPLE 2



The transition table of given Moore machine is as follows:

Q	A	b	Output(λ)
q0	q0	q1	0
q1	q0	q1	1

The equivalent Mealy machine can be obtained as follows:

λ' (q0, a) = λ(δ(q0, a))

= λ(q0)

= 0

λ' (q0, b) = λ(δ(q0, b))

= λ(q1)

= 1

The λ for state q1 is as follows:

λ' (q1, a) = λ(δ(q1, a))

= λ(q0)

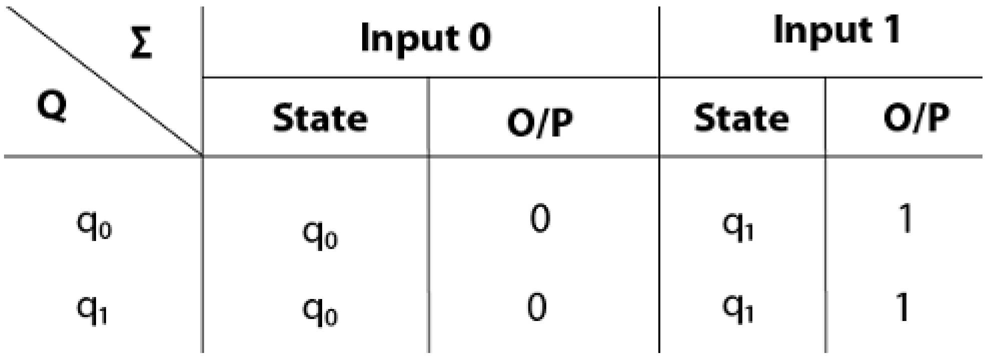
= 0

λ' (q1, b) = λ(δ(q1, b))

= λ(q1)

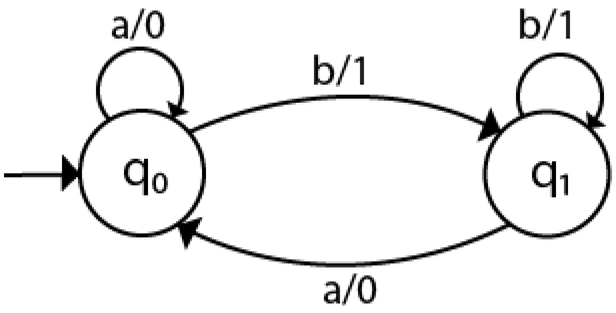
= 1

Hence the transition table for the Mealy machine can be drawn as follows:



Σ / Q	Input 0		Input 1	
	State	O/P	State	O/P
q_0	q_0	0	q_1	1
q_1	q_0	0	q_1	1

The equivalent Mealy machine will be,

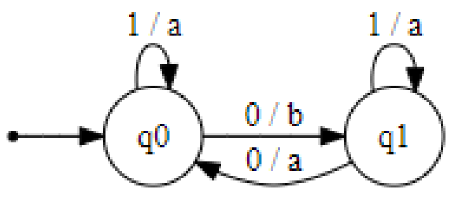


Conversion of mealay to moore with examples

The steps involved in this method are:

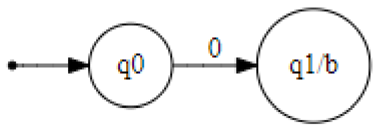
1. Select the initial state and draw the transitions by adding the output symbol of that transition to the destination state.
2. Repeat the process for every single transition.
3. If a conflict occurs, that is, if a state requires to give two different outputs, make a duplicate state.
4. Add transitions for all duplicate states on all input symbols.

Example 1



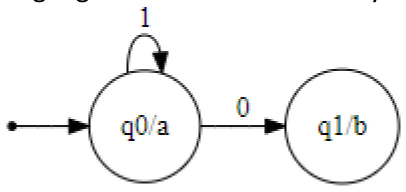
An example Mealy machine

1. We take the initial state and draw its first transition:



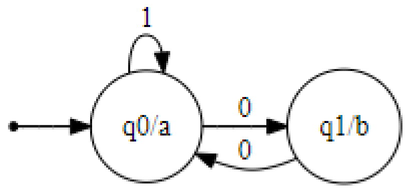
Step 1: Draw first transition of the initial state

The transition had an output b , that is now transferred to our destination node $q1$. Now, for the second transition of state $q0$:



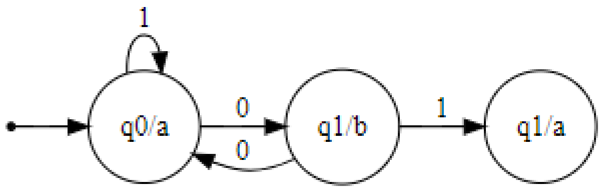
Step 1: Draw remaining transitions of the initial state

2. We repeat the process:



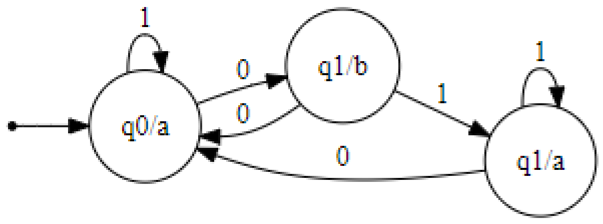
Step 2: Repeat the process

3. We encounter a conflict on the transition $q1 \to q1$ as it has an output a , but the state $q1$ already has an input b. So we make a duplicate state:



Step 3: Add duplicate states when conflict occurs

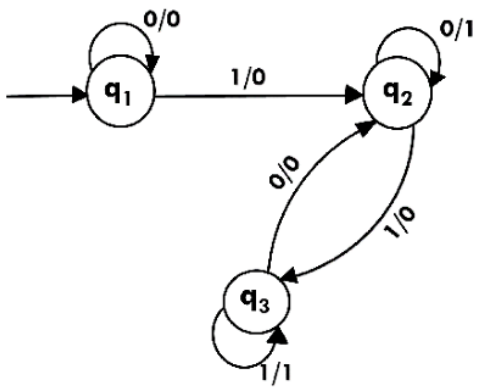
4. Now we add transitions to the new state:



Step 4: Add transitions to the new state

<u>Note:</u> we know from the Mealy machine that $q1$ at 0 goes to $q0$ and gives an output a, so we use the same logic to form the new transition.

Example 2



Transition table for above Mealy machine is as follows:

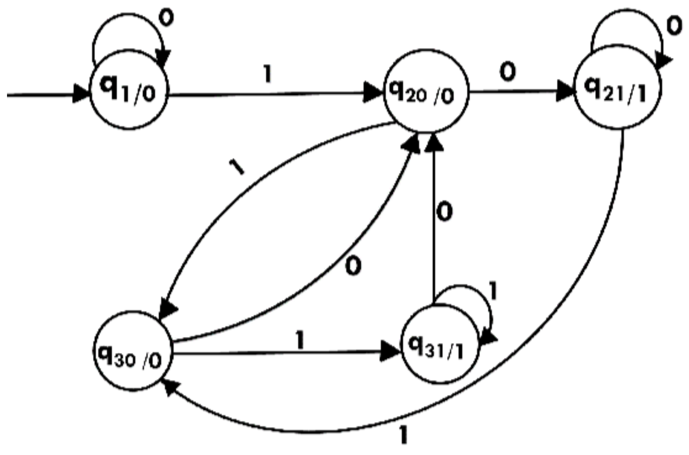
Present State	Next State 0		Next State 1	
	State	O/P	State	O/P
q_1	q_1	0	q_2	0
q_2	q_2	1	q_3	0
q_3	q_2	0	q_3	1

The state q1 has only one output. The state q2 and q3 have both output 0 and 1. So we will create two states for these states. For q2, two states will be q20(with output 0) and q21(with output 1). Similarly, for q3 two states will be q30(with output 0) and q31(with output 1).

Transition table for Moore machine will be:

Present State	Next State 0	Next State 1	O/P
q_1	q_1	q_{20}	0
q_{20}	q_{21}	q_{30}	0
q_{21}	q_{21}	q_{30}	1
q_{30}	q_{20}	q_{31}	0
q_{31}	q_{20}	q_{31}	1

Transition diagram for Moore machine will be:



UNIT -II
PART –I REGULAR EXPRESSION

FINITE AUTOMATA AND REGULAR EXPRESSION

Regular expressions are used to represent the regular languages in Automata. It is also used in compiler designing. Regular expressions are just like arithmetic, logic, and Boolean expressions.
Operations on Regular Language
There may be various operations in regular languages. Let "R" be a Regular expression over the alphabet Sigma if R is

1**. If regular expression (R) is equal to Epsilon (ε), then the language of Regular expression (R) will represent the epsilon set, i.e. { ε}.** A mathematical equation is given below

$$\text{If } R = \varepsilon \text{ Then } L(R) = \{\varepsilon\}$$

2. If regular expression (R) is equal to Φ, then the language of Regular expression (R) will represent the empty set, i.e. { }. The mathematical equation is given below,

$$\text{If } R = \Phi \text{ Then } L(R) = \{\ \}$$

3. **If regular expression (R) is equal to an input symbol "a," which belongs to sigma, then the language of Regular expression (R) will represent the set which has "a" alphabet, i.e. {a}. A** mathematical equation is given below,

$$\text{If } R = a \text{ Then } L(R) = \{a\}$$

4. The union **of two Two Regular Expressions will always produce a regular language**.
Suppose R1 and R2 are two regular expressions. IF R1= a, R2=b then R1 U R2 =a+b So L(R1 U R2) = {a,b}, still string "a,b" is a regular language.

If
R1 = a
R2 = b
Then
R1 U R2 = {a} U {b} = {a+b}
L(R1 U R2) = {a,b}

Hence, the above equation shows that {a,b} is also a regular language.

Note: The intersection of two Two Regular Expressions will always produce a regular language.

5. Concatenation of two Two Regular Expressions will always produce a regular language.
IF R1= a, R2=b then R1.R2 =a.b So L(R1.R2) = {ab}, still string "ab" is a regular language

If
R1 = a
R2 = b

Then

R1 ▪ R2 = {a} ▪ {b} = {ab}
L(R1 ▪ R2) = {ab}

Hence, the above equation shows {ab} is also a regular language.

6. **Kleene closure of Regular Expression (RE) is also a regular language**
If R1 = x and (R1)* is still a regular language
In a regular expression, **x*** means zero or more occurrences of x.
It can generate { ε, x, xx, xxx, xxxx, …..}
In a regular expression**, x**$^+$ means one or more occurrence of x.
It can generate {x, xx, xxx, xxxx, …..}
7. If R is regular, (R) is also a regular language
Note: Only the above mentioned 7 rules are used for regular expressions. By the combination of above 7-rules more regular Expressions can be created.

Types of Regular Expressions

As we know, regular languages are either finite or infinite. So, Regular expressions can be written for both finite and infinite languages. So, types of Regular expressions are of two types

I. Finite Regular Expressions

Finite Regular expressions are used to represent the finite regular languages. So, the length of finite regular expressions is always limited.
For example,
Write the regular expression for the finite language, which accepts all the strings, having the length exactly two over ∑ = {a, b}.
Solution:
Language for the given example is given below
L = {aa, ab, ba, bb} // only 4 strings are possible for given condition
Regular expression for the above language is given below
L(R) = {aa + ab +ba+bb}

II. Infinite Regular Expressions

Infinite regular expressions are used to represent the infinite regular languages. So, the length of infinite regular expressions is always unlimited.

For example,
Write the regular expression for the language which accepts all the strings, having the first symbol should be "b" and the last symbol should be "a" over ∑ = {a, b}.
Solution:
Language for given example is given below

L = {ba, baa, baba, bbaa, baaaa, babbbba………. } // unlimited strings are possible for given condition
Regular expression for above language is given below
L(R) = b (a+b)* a

Examples of Finite Regular Expressions

Following are the various examples of finite regular expressions

Example 01
Write the regular expression for the language **which has no string.**
Solution:
Language for the given example is given below
L = { } //empty string
Regular expression for the above language is given below
L(R) = Φ

Example 02
Write the regular expression for the language **with a string with zero length.**
Solution
Language for the given example is given below
L = { ε } //single string
Regular expression for the above language is given below
L(R) = ε

Example 03
Write the regular expression for the language **with a string with a single length over** ∑ = {a, b}.
Solution
Language for the given example is given below
L = { a,b } //string with single length
A regular expression for the above language is given below
L(R) = (a+b)

Example 04
Write the regular expression for the language **with a string with the length of two over** ∑ = {a, b}.
Solution
Language for the given example is given below
L = (a,b).(a,b) = { aa,ab,ba,bb } //string with length of two
A regular expression for the above language is given below
L(R) = (aa+ab+ba+bb)
Example 05
Write the regular expression for the language with a string **of three over** ∑ = {a, b}.
Solution
Language for the given example is given below

L = (a,b).(a,b). (a,b) = { aaa,abb,aab,baa……..,bbb } //string with length of three
Regular expression for the above language is given below
L(R) = (aaa + abb + aab + baa+……..+ bbb)

Example 06
Write the regular expression for the language **with strings with an at-most length of 2 over** ∑ = {a, b}.
Language for the given example is given below
L = (ε,a,b).(ε,a,b) = { ε, a ,b, aa,ba……..,bb } // **strings with at-most length of 2**
Regular expression for the above language is given below
L(R) = (ε+a+b).(ε+a+b)

Example 07
Write the regular expression for the language **with strings "Not More than two a's and one b" over** ∑ = {a, b}.
Solution
Language for the given example is given below
L = { ε, a ,b, aa,aba, aab…….. } // **strings with Not More than two a's and one b**
Regular expression for the above language is given below
L(R) = { ε + a + b + aa +aba + aab+…….. }

Examples of Infinite Regular Expressions
Infinite Regular expressions are used to represent the infinite regular languages. So, the length of infinite regular expressions is always Unlimited.

Example 01
Write the regular expression for the language that accepts all the strings having all combinations of a's over ∑ = {a}.
Solution
Language for the given example is given below
L = { ε, a, aa, aaa, aaaa, aaaaa, ……….. }
A regular expression for the above language is given below
L(R) = a*
Example 02
Write the regular expression for the language that accepts all the strings having all combinations of a's except the null string over ∑ = {a}.
Solution
Language for the given example is given below
L = {a, aa, aaa, aaaa, aaaaa, ……….. }
A regular expression for the above language is given below
L(R) = a^+

Example 03
Write the regular expression for the language that accepts all the strings, which has **any number of a's and b's** over **∑ = {a, b}.**

Solution
Language for the given example is given below
L = {a, b, ab, ba, aab, aba, aaba ……….. }
Regular expression for the above language is given below
L(R) = (a+b)*

Example 04
Write the regular expression for the language that accepts all the strings, having only one "b" and any numbers of "a" over ∑ **= {a, b}.**
Solution
Language for the given example is given below
L = {b, ab, aba, abaa, abaaaa, aaaaabaaaaaaa……….. } // all strings having only one "b"
Regular expression for the above language is given below
L(R) = a*ba*

Example 05
Write the regular expression for the language that accepts all the strings, which **are starting with 1 and ending with 0, over ∑ = {0, 1}.**
Solution
Language for the given example is given below
L = {10, 100, 11010, 1110, 1111000, ……….. } // starting with 1 and ending with 0
Regular expression for the above language is given below
L(R) = 1(0+1)*0

Example 06
Write the regular expression for the language that accepts all the strings having even length, over ∑ = {0}.
Solution
Language for the given example is given below
L = { ε, 00, 0000, 000000, 00000000, ……….. }
Regular expression for the above language is given below
L(R) = (00)*

It is defined as representing a regular expression in terms of regular language.

- ϕ is regular expression then regular language is { }.
- ϵϵ is a regular expression then regular language is {ϵ}.
- r is regular expression then regular language {r}.
- Let R and S be regular expression then regular language be LR and LS.
- R + S or R|S be regular expression then regular language is LR U LS.
- R . S be regular expression then regular language is LR.LS.
- R* be regular language then regular expression is LR*.
- R++ be regular language then regular expression is L+R+.
- where ***** means 0 or more occurrence ,**+** means 1 or more occurrence.
-

Example:

Regular Expression	Regular language
a	{a}
B	{b}
a+b	{a,b}
a··b	{ab}
a^*	{ϵϵ,a,aa,aaa,aaaa,......}
b^+	{b,bb,bbb,bbbb,.......}

APPLICATIONS OF REGULAR EXPRESSION:

1. Regular expressions are useful in a wide variety of text processing tasks, and more generally string processing, where the data need not be textual. Common applications include data validation, data scraping (especially web scraping), data wrangling, simple parsing, the production of syntax highlighting systems, and many other tasks.
2. While regular expression would be useful on Internet search engines, processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regex.

Applications of Regular Expression:

1. Text Search and Manipulation
2. Input Validation
3. Data Extraction and Parsing
4. Programming and Code Parsing
5. Web Development
6. Network Security
7. Command-Line Operations
8. Database Operations

1. Text Search and Manipulation: Regular expressions are widely recognized for their prowess in text search and manipulation, offering a robust mechanism for finding and replacing specific patterns or substrings within a given text. This application is particularly valuable for tasks such as document editing, codebase maintenance, and data cleaning. In the context of text search, a regular expression allows users to define a pattern, specifying the sequence of characters they want to locate. This pattern may include literal characters, metacharacters, and quantifiers, providing a flexible and expressive syntax.

For example, a simple regular expression like `\b\d{3}\b` can be used to find three-digit numbers in a document, where `\b` denotes a word boundary, and `\d{3}` indicates exactly three digits. This enables users to identify and manipulate numerical data efficiently, whether for formatting purposes or numerical analysis.

Moreover, regular expressions excel in the manipulation of text by enabling the replacement of identified patterns with new values. This find-and-replace functionality proves immensely useful for tasks such as code refactoring, where developers can efficiently update variable names, function calls, or other code elements across multiple files simultaneously.

2. **Input Validation:** Regular expressions play a crucial role in input validation, ensuring that user-provided data adheres to specified formats. In web development, for instance, forms often require users to enter information in a particular structure, such as email addresses, phone numbers, or dates. Regular expressions offer a powerful means to validate and enforce these formats, preventing the submission of incorrect or potentially harmful data.

Consider a scenario where a website prompts users to enter an email address. A regular expression like `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$` can be employed to validate the input against the standard email format. This regex checks for the presence of a valid username, domain, and top-level domain, ensuring that the entered data conforms to the expected email structure.

By incorporating regular expressions into form validation processes, developers enhance the overall security and reliability of applications, mitigating the risks associated with malformed or malicious input.

3. **Data Extraction and Parsing:** Regular expressions play a pivotal role in data extraction and parsing, particularly in scenarios involving log analysis and the processing of structured or semi-structured data. In log files, which often contain vast amounts of information, regular expressions provide a systematic way to extract relevant details based on predefined patterns.

For instance, consider a log file containing entries with timestamps, IP addresses, and error messages. A well-crafted regular expression can be designed to capture and isolate each of these components. This facilitates the extraction of specific information, enabling analysts to identify trends, troubleshoot issues, and gain valuable insights from the log data.

In the broader context of data parsing, regular expressions are instrumental in breaking down complex data structures into manageable components. Whether dealing with CSV files, XML documents, or other formatted data, regex patterns can be tailored to extract meaningful information from the raw data, contributing to more effective data analysis and interpretation.

4. **Programming and Code Parsing:** Regular expressions find extensive application in the realm of programming, offering developers a powerful toolset for tasks such as code search, refactoring, and syntax highlighting in code editors. When working with large codebases, the ability to efficiently locate specific code patterns or elements is crucial for navigation, debugging, and code maintenance.

For instance, a regular expression can be employed to search for all occurrences of a particular function or variable name within a project. This enables developers to quickly assess the scope of a change, identify potential issues, and streamline the codebase.

Moreover, in code editors, regular expressions contribute to syntax highlighting by identifying and applying distinct styles to different code elements. This visual representation enhances code readability and helps developers grasp the structure of the code at a glance.

5. **Web Development:** Regular expressions play a pivotal role in web development, where they are employed for tasks ranging from validating user inputs to parsing and manipulating web-related data. In the context of web forms and user input, regular expressions are instrumental in enforcing data format standards. For example, when users submit URLs or dates through a form, developers can utilize regex patterns to validate and ensure the correctness of the provided data.
Consider a scenario where a web application requires users to enter a valid URL. A regular expression like `^(https?|ftp)://[^\s/$.?#].[^\s]*$` can be used to validate the format of the URL, ensuring that it starts with either "http," "https," or "ftp" and follows the standard structure of a web address.
Beyond input validation, regular expressions are employed in parsing HTML documents. While it's generally advised to use dedicated HTML parsers for complex scenarios, regular expressions can be handy for simpler tasks such as extracting specific tags, attributes, or content. This versatility makes regular expressions a valuable tool in the development of web applications, contributing to both functionality and data integrity.

6. **Network Security:** In the realm of network security, regular expressions contribute significantly to pattern matching within network traffic logs, aiding in security analysis and the validation of IP addresses. Security professionals utilize regular expressions to identify specific patterns indicative of security threats, such as unusual network behavior or known attack signatures.
For instance, a regular expression might be crafted to detect a common type of malicious payload in network traffic. The ability to recognize and flag such patterns allows security analysts to respond promptly to potential threats, strengthening the overall cybersecurity posture.
Regular expressions also play a role in the validation of IP addresses. By defining patterns that conform to valid IP address formats, network administrators can ensure that the data being processed aligns with expected standards. This application contributes to the accuracy of security analyses and helps prevent issues arising from incorrectly formatted or manipulated IP addresses.

7**. Command-Line Operations:** Regular expressions streamline file operations in command-line environments, providing a powerful means to search for and manipulate files efficiently. Command-line tools often support regular expressions, allowing users to perform complex file-related tasks with concise and expressive patterns.

For example, the `grep` command, a widely used tool for searching through text, supports regular expressions to filter and extract specific lines from files. This capability is particularly useful in scenarios where users need to sift through large log files or codebases to find occurrences of particular patterns.
Regular expressions in command-line operations empower users to perform tasks such as bulk renaming of files, searching for specific content within files, and filtering files based on predefined criteria. This flexibility enhances the efficiency of file-related workflows, making regular expressions an indispensable tool for command-line enthusiasts and system administrators.

8. **Database Operations:** In the domain of databases, regular expressions are applied to queries for filtering and retrieving data based on specific patterns. This capability is valuable for tasks such as data cleaning, where patterns need to be identified and modified systematically.
Consider a database containing customer information, where phone numbers are stored in varying formats. A regular expression can be employed in a query to retrieve all records with phone numbers adhering to a standardized format, facilitating consistent data presentation.
Moreover, regular expressions in databases extend beyond simple pattern matching. They can be used for more complex tasks, such as identifying and extracting information from unstructured data fields. This flexibility makes regular expressions a powerful tool for data analysts and database administrators dealing with diverse datasets and data quality challenges.

ALGEBRAIC LAWS FOR REGULAR EXPRESSIONS

1. **Associativity Laws for Regular Expressions RegEx**

A + (B + C) = (A + B) + C and A.(B.C) = (A.B).C.

2. **Commutativity for Regular Expressions RegEx**

A + B = B + A. However, A.B 6= B.A in general.

3. **Identity for Regular Expressions RegEx**

∅ + A = A + ∅ = A and ε.A = A.ε = A

4. **Annihilator for Regular Expressions RegEx**

∅.A = A.∅ = ∅

5. **Distributivity for Regular Expressions RegEx**

Left distributivity: A.(B + C) = A.B + A.C.
Right distributivity: (B + C).A = B.A + C.A.
Idempotent A + A = A.

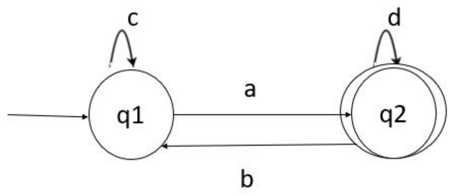
6. **Closure Laws for Regular Expressions RegEx**

(A*)* = A*, ∅* = ε, ε* = ε, A+ = AA* = A*A, and A* = A + + ε.

7. **DeMorgan Type Law for Regular Expressions RegEx**

(L + B)* = (L*B*)*

CONVERSION OF FINITE AUTOMATA TO REGULAR EXPRESSIONS.



There are two popular methods for converting a DFA to its regular expression −

- **Arden's Method**
- **State elimination method**

Let's consider the state elimination method to convert FA to RE.

Rules: The rules for state elimination method are as follows −

Rule 1

The initial state of DFA must not have any incoming edge.

If there is any incoming edge to the initial edge, then create a new initial state having no incoming edge to it.

Rule 2

There must exist only one final state in DFA.

If there exist multiple final states, then convert all the final states into non-final states and create a new single final state.

Rule 3

The final state of DFA must not have any outgoing edge.

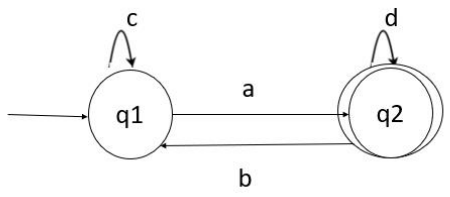
If this exists, then create a new final state having no outgoing edge from it.

Rule 4

Eliminate all intermediate states one by one.

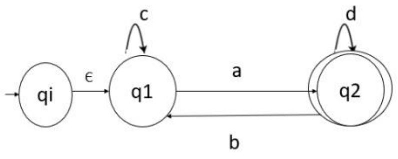
Now, apply these rules to convert the FA to RE easily.

The given FA is as follows −



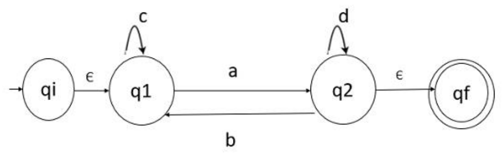
Step 1

Initial state q1 has an incoming edge so create a new initial state qi.

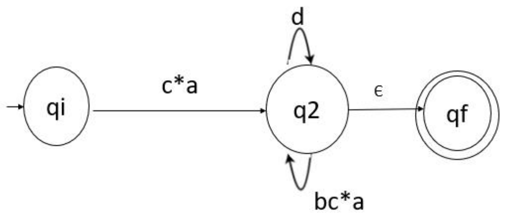


Step 2

Final state q2 has an outgoing edge. So, create a new final state qf.



Step 3
Start eliminating intermediate states
First eliminate q1
There is a path going from qi to q2 via q1. So, after eliminating q1 we can connect a direct path from qi to q2 having cost.
εc*a=c*a
There is a loop on q2 using state qi. So, after eliminating q1 we put a direct loop to q2 having cost.
b.c*.a=bc*a
After eliminating q1, the FA looks like following −

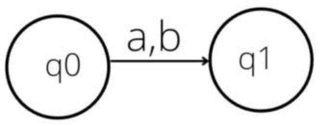


Second eliminate q2
There is a direct path from qi to qf so, we can directly eliminate q2 having cost
C*a(d+bc*a)* ε = c*a(d+bc*a)*
This is our final regular expression for given finite automata.

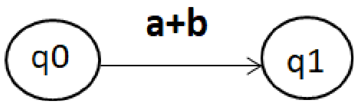
CONVERSION OF REGULAR EXPRESSIONS TO FINITE AUTOMATA.

The steps in this method are given below:-
Step 1: Make a transition diagram for a given regular expression, using NFA with ε moves.
Step 2: Then, Convert this NFA with ε to NFA without ε.
Step 3: Finally, Convert the obtained NFA to equivalent DFA.

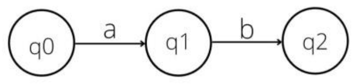
Some standard rules help in the conversion of RE to NFA are:-
1.) If RE is in the form a+b, it can be represented as:



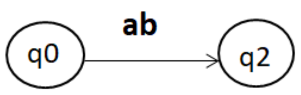
Can represent in



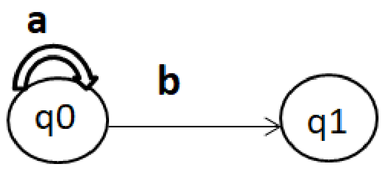
2.) If RE is in the form **ab,** it can be represented as:



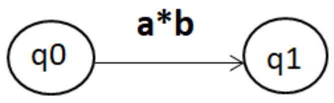
Can represent in



3.) If RE is in the form of **a*,** it can be represented as:



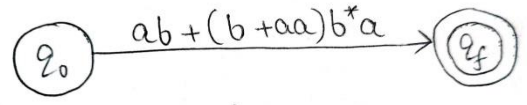
Can represent in



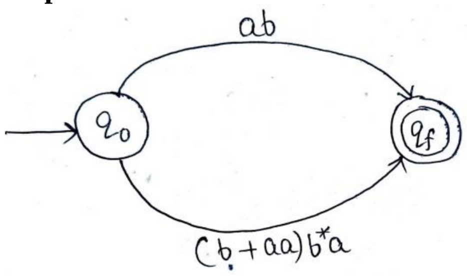
Example 1: Design a Finite Automata from the given RE [ab + (b + aa)b* a].

Solution. At first, we will design the Transition diagram for the given expression.

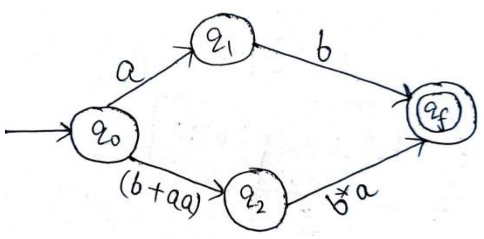
Step 1



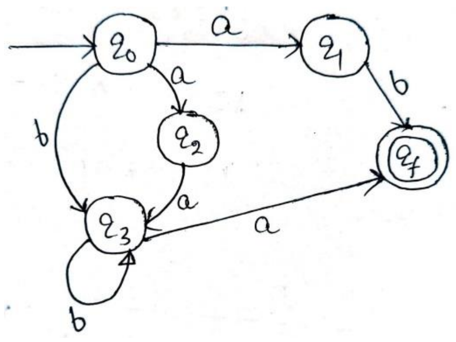
Step 2



Step 3



Step 4



After step 5, we have got NFA without ε. Now we will convert it into the required DFA; we will first write a transition table for this NFA.

Transition Table For NFA:

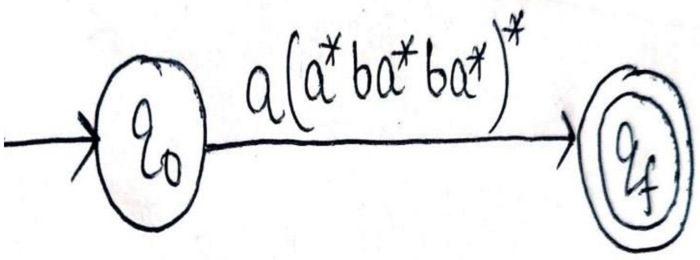
State	a	b
→q0	{q1,q2}	q3
q1	ϕ	qf
q2	q3	ϕ
q3	qf	q3
*qf	ϕ	ϕ

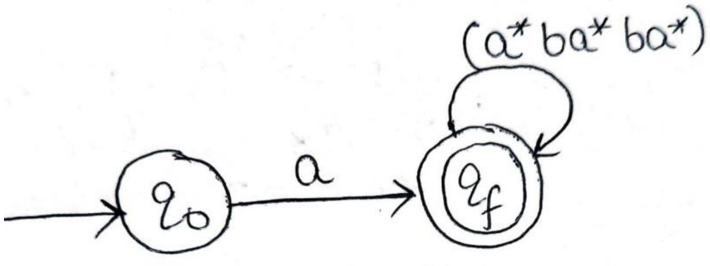
The Corresponding transition table for DFA is:

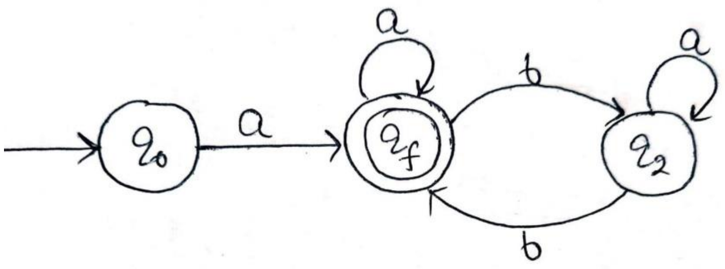
State	a	b
→q0	[q1,q2]	q3
q1	ϕ	qf
q2	q3	ϕ
q3	qf	q3
[q1,q2]	qf	qf
*qf	ϕ	ϕ

Example 2: Design an NFA from the given RE [a (a* ba* ba*)*].

Solution. The Transition diagram that represents the NFA of the above expression is as follow:
Step 1:







We have successfully converted the given RE to an NFA.

PART-II
PUMPING LEMMA FOR REGULAR LANGUAGES:

Pumping Lemma for Regular Languages

Theorem: **If A is a Regular Language, then A has a Pumping Length 'P' such that any string 'S' where |S ≥ P may be divided into three parts S = xyz such that the following conditions must be true**:
1.) $xy^iz \in A$ for every $i \geq 0$
2.) $|y| > 0$
3.) $|xy| \leq P$
In simple words, if a string **y** is 'pumped' or insert any number of times, the resultant string still remains in A.
Pumping Lemma is used as proof of the **irregularity** of a language. It means, that if a language is regular, it always satisfies the pumping lemma. If at least one string is made from pumping, not in language A, then A is not regular.

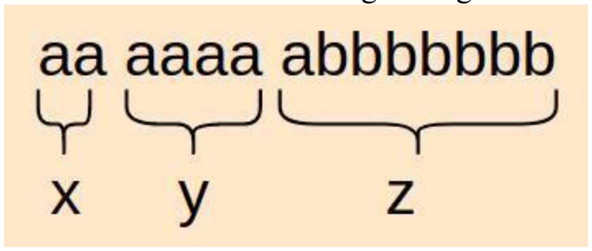
We use the **CONTRADICTION** method to prove that a language is not Regular.

Steps to prove that a language is not Regular using Pumping Lemma:
Step 1: Assume that Language A is Regular.
Step 2: It has to have a Pumping Length (say P).
Step 3: All strings longer than P can be pumped $|S| \geq P$.
Step 4: Now, find a string 'S' in A such that $|S| \geq P$.
Step 5: Divide S into x y z strings.
Step 6: Show that $xy^iz \notin A$ for some i.
Step 7: Then consider how S can be divided into x y z.
Step 8: Show that none of the above strings satisfies all three pumping conditions simultaneously.
Step 9: S cannot be pumped == CONTRADICTION.

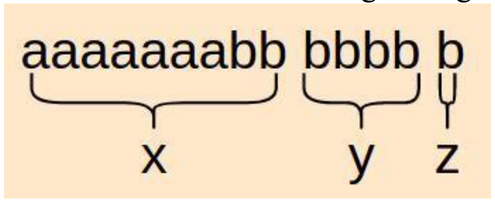
Let's apply these above steps to check whether a Language is not Regular with the help of Pumping Lemma.

Implementation of Pumping lemma for regular languages
Example: Using Pumping Lemma, prove that the language **$A = \{a^nb^n \mid n \geq 0\}$ is Not Regular**.
Solution: We will follow the steps we have learned above to prove this.
Assume that A is Regular and has a Pumping length = P.
Let a string **$S = a^Pb^P$.**
Now divide the S into the parts, x y z.
To divide the S, let's take the value of P = 7.
Therefore, S = aaaaaaabbbbbbb (by putting P=7 in $S = a^Pb^P$).

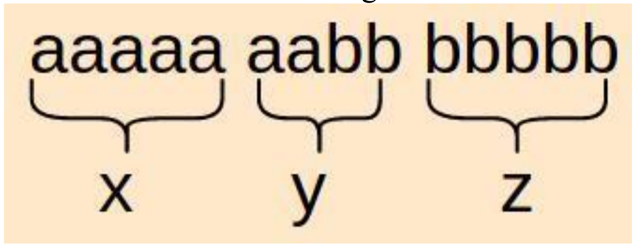
Case 1: Y consists of a string having the letter only ‘a’.



Case 2: Y consists of a string having the letter only ‘b’.



Case 3: Y consists of a string with the letters ‘a’ and ‘b’.



For all the above cases, we need to show $xy^iz \notin A$ for some i.
Let the value of i = 2. **$xy^iz \Rightarrow xy^2z$**

In Case 1. **xy^2z = aa aaaa aaaa abbbbbbb**
No of ‘a’ = 11, No. of ‘b’ = 7.
Since the **No of ‘a’ != No. of ‘b’**, but the original language has an equal number of ‘a’ and ‘b’; therefore, this string will not lie in our language.

In Case 2. **xy^2z = aaaaaaabb bbbb bbbb b**
No of ‘a’ = 7, No. of ‘b’ = 11.
Since the **No of ‘a’ != No. of ‘b’**, but the original language has an equal number of ‘a’ and ‘b’; therefore, this string will not lie in our language.

In Case 3. **xy^2z = aaaa aabb aabb bbbbb**
No of ‘a’ = 8, No. of ‘b’ = 9.

Since the **No of ‘a’ != No. of ‘b’**, but the original language has an equal number of ‘a’ and ‘b’, and also, this string did not follow the a^nb^n pattern; therefore, this string will not lie in our language.

We can see at i = 2 all the above three strings do not lie in the language $A = \{a^n b^n \mid n \geq 0\}$.
Therefore, the language $A = \{a^n b^n \mid n \geq 0\}$ is not Regular.

Application of pumping lemma

Pumping lemma is to be applied to show that certain languages are not regular.
It should never be used to show a language is regular.

- If L is regular, it satisfies the Pumping lemma.
- If L does not satisfy the Pumping Lemma, it is not regular.

PART-III
CONTEXT FREE GRAMMAR

Definition of Context-Free Grammars

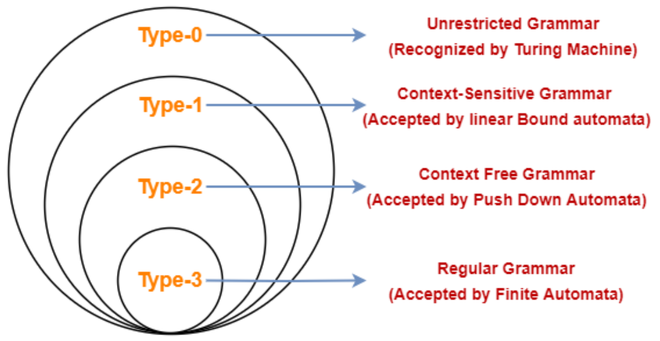
A context free grammar (CFG) is a forma grammar which is used to generate all the possible patterns of strings in a given formal language.
It is defined as four tuples −
G=(V,T,P,S)

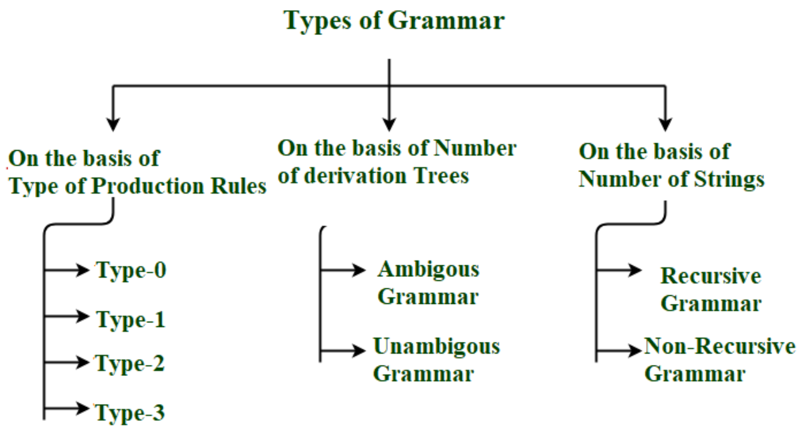
- G is a grammar, which consists of a set of production rules. It is used to generate the strings of a language.
- T is the final set of terminal symbols. It is denoted by lower case letters.
- V is the final set of non-terminal symbols. It is denoted by capital letters
- P is a set of production rules, which is used for replacing non-terminal symbols (on the left side of production) in a string with other terminals (on the right side of production).
- S is the start symbol used to derive the string

Example
Construct CFG for the language having any number of a's over the set ∑={a}
Solution
Regular Expression= a*
Production rule for the Regular Expression is as follows −
S->aS rule 1
S-> ε rule 2
Now if we want to derive a string "aaaaaa" we can start with start symbol
Start with start symbol:

s	rule
aS	1
aaS	1
aaaS	1
aaaaS	1
aaaaaS	1
aaaaaaS	1
aaaaaa	2

The regular expression=a* can generate a set of strings { ε,a,aa,aaa,...}
We can have a null string because S is a start symbol and rule 2 gives S-> ε


Type-0
Unrestricted Grammar
(Recognized by Turing Machine)
Type-1
Context-Sensitive Grammar
(Accepted by linear Bound automata)
Type-2
Context Free Grammar
(Accepted by Push Down Automata)
Type-3
Regular Grammar
(Accepted by Finite Automata)


Types of Grammar
On the basis of Type of Production Rules
Type-0
Type-1
Type-2
Type-3
On the basis of Number of derivation Trees
Ambigous Grammar
Unambigous Grammar
On the basis of Number of Strings
Recursive Grammar
Non-Recursive Grammar

Production Rules and Strings

As we know, every grammar generates a specific language. Each grammar holds some rules, which are called Production Rules. Strings are generated by using production rules. Let's explain both Production Rules and Strings.

What is a string?

The string is a part of the language. Grammar produces strings through its production rules. The length of a string is denoted as |w|, where w is the string. There may be multiple strings in a language, including an empty string represented by epsilon (ε) or lambda (λ).
The following language contains four **strings**
L(G) = {a, bb, abc, b}
Representation: (w|w€ L)
Number of Strings
If Σ{a, b}
The number of Strings (length 2) will be {aa,ab, ba, bb}.Length of String |w| = 2.Possible Number of Strings = 4.
Conclusion: For alphabet {a, b} with length **n**, the number of strings can be generated = $\mathbf{2^n}$.

Production Rule

A production rule is represented by the letter P. Here are some examples of production rules:
P: S → aSb / ∈
The above production rule can be written as
S → aSb
S → ∈
The Production Rule (P: S → aSb / ∈) can generate the following strings

∈, ab, aabb, aaabbb, …….
So, all the above symbols are strings generated through production rules.
The language with its grammar produces strings using production rules that are represented below.
L(G) = { ∈, ab, aabb, aaabbb…….} = L = { a^nb^n , n>=0 }

The deriving a string from grammar is called a derivation in Automata. The Representation of derivation in the form of a tree is called a derivation tree.

Purpose of Derivation

A derivation tree is useful when a string and grammar (production rules) are given, and we must check whether the string belongs to grammar.

Example of Derivation

The following production rules are given

S → xB

B → xS| ε

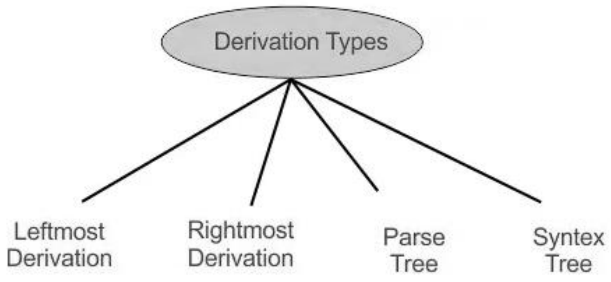
And suppose a **string W= xxx.**

A derivation and its tree are possible if a given string is derived from given production rules.

Types of Derivation

There are four basic types of derivations in the theory of Automata

- Leftmost derivation
- Rightmost derivation
- Parse tree
- Syntax tree



However, the first two types of derivations are mainly used.

- Leftmost and Rightmost derivation, parse, and syntax trees will be similar for unambiguous grammars.
- Leftmost and Rightmost derivation, parse, and syntax trees will not be similar. For ambiguous grammar,

Leftmost Derivation in Automata

Getting a string by expanding the leftmost nonterminal at each step is called a leftmost derivation in Automata.

- The representation of leftmost derivation in a tree is called a leftmost derivation tree.
- We read the given input string (W) from left to right in the leftmost derivation tree.

Example

S → aS / ∈

For generating strings 'aaa'.

S → aS

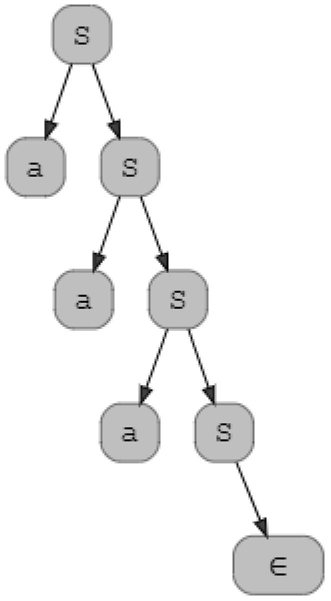
→ aaS (Using S → aS)

→ aaaS (Using S → aS)

→ aaa∈ (Using S → ∈)

→ aaa

In the example above, we try to change the leftmost non-terminal to get the desired output everytime.



Leftmost Derivation

We expand the tree using the leftmost derivation in the diagram above.

First, we expand S, the leftmost non-terminal in the given example. Then, we expand the tree in the same fashion by first reducing the leftmost non-terminal to get the given input string.

Therefore, in the tree above, we expand the nodes from the leftmost non-terminal and successfully get the given input string.

Rightmost Derivation: This is the process of deriving the input string by expanding the rightmost non-terminal.

In the rightmost derivation, we derive the string by changing the rightmost non-terminal to get the input string.

Example

S → aSS / ∈

For generating strings ‘aaa’.

S →aSS

Note: Here, we try to add production rules to the rightmost non-terminal(i.e., S). This continues for the next steps as well.

We try to change the s in the example everytime because it is the rightmost non-terminal.

→aSs (Using S → aS)

→aSas (Using S → aS)

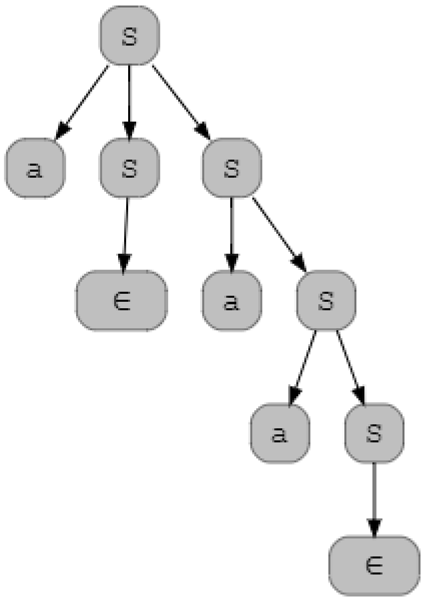
→aSaas (Using S → aS)

→aSaas (Using S → ∈)

→aSaa (Using S → ∈)

→aaa (Using S → ∈)

In the example above, we try to change the rightmost non-terminal to get the desired output.



In the diagram, we expand the tree using the rightmost derivation.

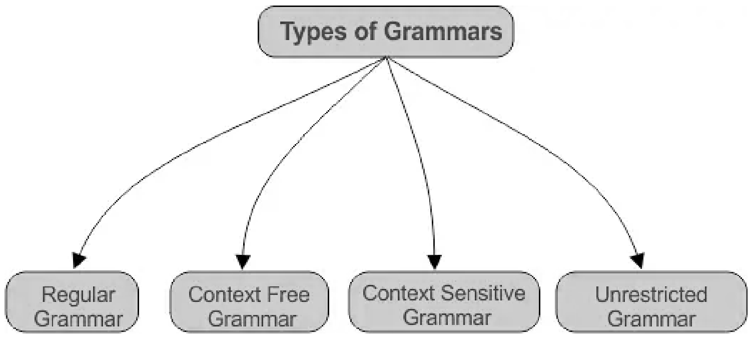
First, we expand the rightmost non-terminal, S, in the given example. Then, we expand the tree in the same fashion by first reducing the rightmost non-terminal to get the given input string.

Therefore, in the tree above, we expand the nodes from the rightmost non-terminal and successfully get the given input string.

THE LANGUAGE OF A GRAMMAR

There are four major types of grammars in the theory of Automata.

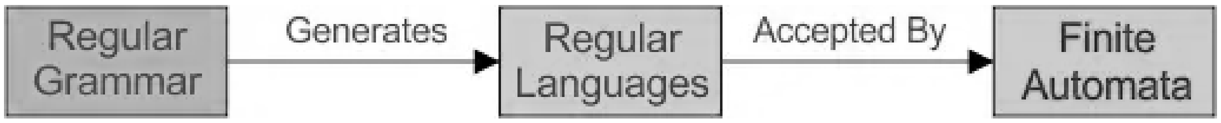
- Regular Grammar
- Context Free Grammar
- Context Sensitive Grammar
- Unrestricted Grammar



Let's explain all the above types of Grammar in detail.

1. Regular Grammar (RL)

Regular Grammar generates the Regular Languages, and the Finite Automata Machine accepts all regular languages. Regular Grammar is also called Type-3 grammar.



Regular languages are of two types.

I. Finite Regular Languages
II. Infinite Regular Languages

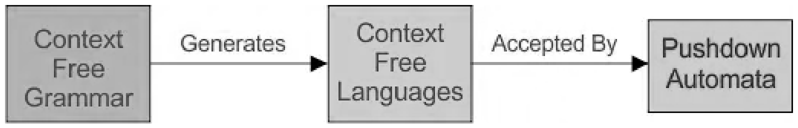
Finite Automata Machine is also of two types

I. Deterministic Finite Automata (DFA)
II. Non-Deterministic Finite Automata (NDFA)

All regular languages, either finite or infinite, are accepted by NDFA. However, all regular languages and some infinite languages are accepted by DFA.

2. Context Free Grammar (CFG)

Context-free Grammar generates Context-free Languages, and the Pushdown Automata Machine accepts all context-free languages. Context-free Grammar is also called Type-2 grammar.



Context Free languages are of two types

I. Deterministic Context-Free Languages
II. Deterministic Context-Free Languages

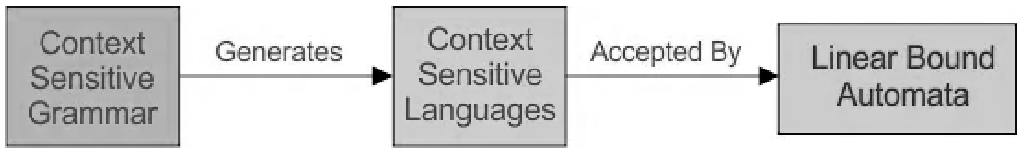
Pushdown Automata Machine is also of two types

I. Deterministic Pushdown Automata (DPDA)
II. Non-Deterministic Pushdown Automata (NPDA)

All Deterministic Context-Free languages are accepted by Deterministic Pushdown Automata (DPDA). But all Non-Deterministic Context-Free languages are accepted by Non-Deterministic Pushdown Automata (NPDA).

3. Context Sensitive Grammar (CSG)

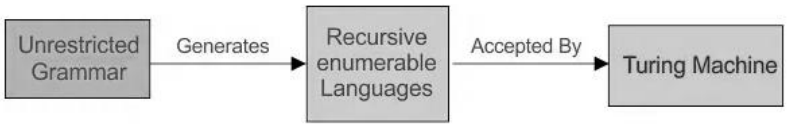
Context-sensitive Grammar generates context-sensitive languages, and linear bound automata machines accept all context-sensitive languages. Context-free Grammar is also called Type-1 grammar.



Note: Context-sensitive languages and their machines are not deterministic or non-deterministic.

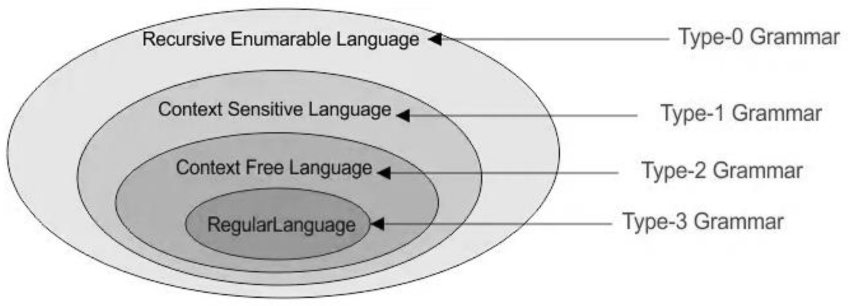
4. Unrestricted Grammar

Unrestricted Grammar generates the Recursive enumerable Languages, and the Turing Machine accepts all Recursive enumerable languages. Unrestricted Grammar is also called Type-0 grammar.



Chomsky Hierarchy

An explanation of all four types of Grammar is known as the Chomsky hierarchy. A diagram of Chomsky's hierarchy is given below.



Necessary: Type-0 Accepter has the Highest Power, and Type-3 accepter has the lowest Power. It means type -3 accepted, which means Turing Machine can accept all languages, whether regular, context-free, or context-sensitive. However, the Type-3 accepter, Finite Automata, can accept only regular languages.

PARSE TREES

The process of deriving a string from a given grammar is called **derivation**. The **geometrical** representation of a derivation is known as a derivation **tree or parse tree**.

In **Parse Tree**, the deepest sub-tree from leftmost is traversed first by following the rule of In-order traversal. In this way, the original input string is obtained, but

- All leaf nodes must be terminals.
- All interior nodes must be Non-terminals.

Example of Parse Tree

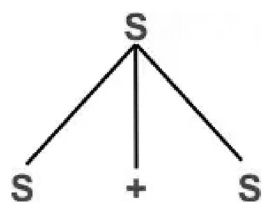
Suppose the following Production rules of a Grammar (G)

S → S + S | S * S

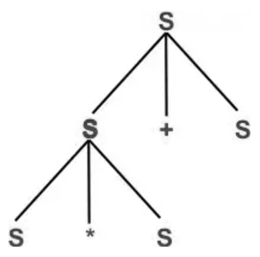
S → x|y|z

and Input is (x * y + Z)

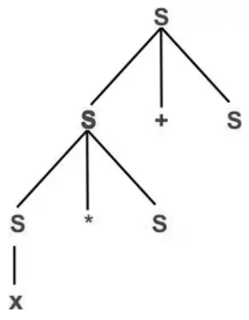
Step 1



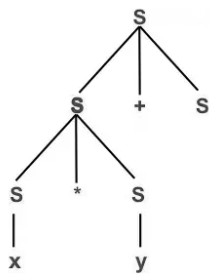
Step 2



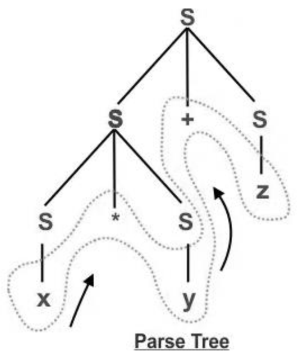
Step 3



STEP 4



STEP 5



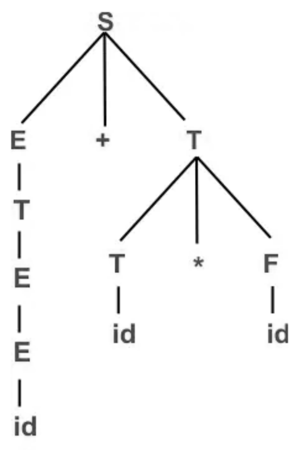
Parse Tree

Syntax Tree In Automata

When constructing a parse tree in Automata, it may contain unnecessary details. So, it is very difficult for a compiler to execute unnecessary information. That’s why a syntax tree, which holds just the necessary information, is used.

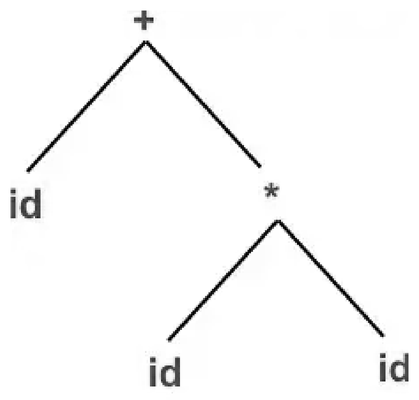
Example of Syntax Tree

Consider the following Parse tree.



Parse Tree

The above parse tree provides the string "id + id * id." Let us eliminate the extra information from the given parse tree to get the same string.

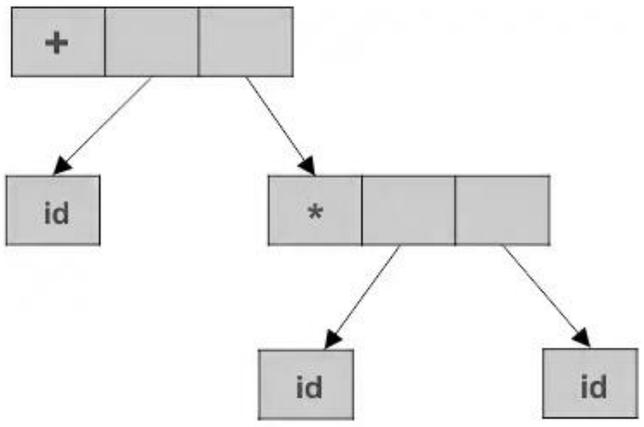


Syntax Tree

Hence, A String "id + id * id" is derived by eliminating unnecessary information. It is called a syntax tree.

Abstract syntax tree

Abstract syntax tree can be represented as follows



Abstract Syntax Tree

In compiler, the Abstract syntax trees are important because the abstract syntax tree contains the least unnecessary information.

A CFG is said to be ambiguous if and only if it contains more than one derivation trees for same string.

1. **Definition of Ambiguous Grammar**: Let G = (N,T, P, S) be a CFG. A string w ∈ L(G) is said to be "ambiguously derivable "if there are two or more different derivation trees for that string in G.
2. **Definition of Ambiguous Grammar:** A CFG given by G = (N, T, P, S) is said to be "ambiguous" if there exists at least one string in L(G) which is ambiguously derivable. Otherwise it is unambiguous.

Ambiguity is a property of a grammar, and it is usually, but not always possible to find an equivalent unambiguous grammar.

An "inherantly ambiguous language" is a language for which no unambiguous grammar exists.

Solved Example:

Show that the grammar G with production

S → a | aAb | abSb
A → aAAb | bS
is ambiguous.

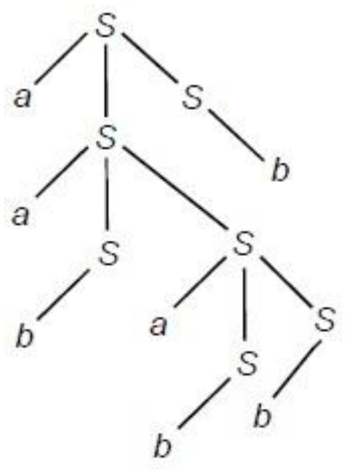
Solution:

S ⟹ abSb (∵ S → abSb)
⟹ abab (∵ S → a)
Similarly,
S ⟹ aAb (∵ S → aAb)
⟹ abSb (∵ A → bS)
⟹ abab (∵ S → a)
Since 'abab' has two different derivations, the grammar G is ambiguous.

Proof of CFG is ambiguous or not by using parse tree solved example:

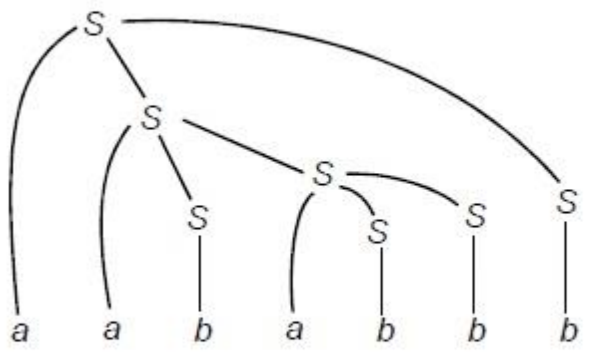
Consider the grammar G with production:
S → aSS
S → b
The parse trees are as follows.

Top down Parsing: Sequence of rules are applied in a leftmost derivation in Top down parsing.



Top down Parsing Tree

Bottom-up Parsing: Sequence of rules are applied in a rightmost derivation in Bottom-up parsing.



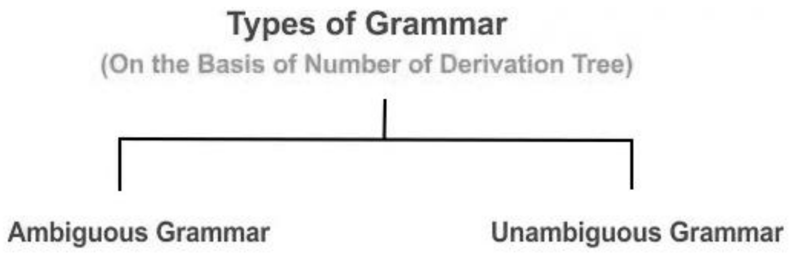
Bottom-up Parsing Tree

Hence it is ambiguous because it consist more than one parse tree for a string w of L(G). To understand the parsing, A grammar can be used in two ways:

1. Using the grammar to generate strings of the language.
2. Using the grammar to recognize the strings.

“Parsing” a string is finding a derivation (or a derivation tree) for that string.

Parsing a string is like recognizing a string. The only realistic way to recognize a string of a context-free grammar is to parse it.



Ambiguous Grammar

A grammar is said to be **ambiguous grammar** if any string generated by it produces more than one.
Parse tree
Or syntax tree
Or leftmost derivation
Or rightmost derivation

Examples of Ambiguous Grammar

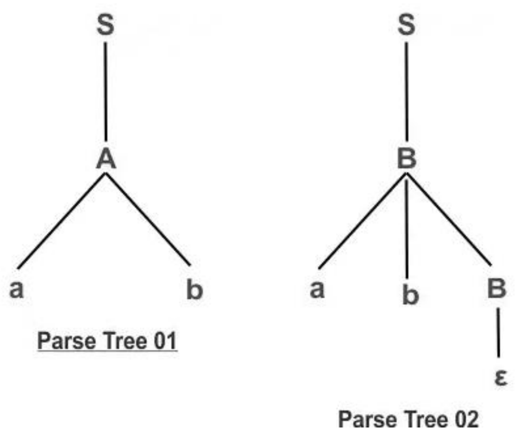
Example 01

Check whether the following grammar is ambiguous or not for string w = ab

S → A / B
A → aAb / ab
B → abB / ∈

Solution

Now, we draw more than one parse tree to get the string w = ab.



The original string (w =ab) can derived through two different parse trees. So, the given grammar is ambiguous.

Unambiguous Grammar

A grammar is said to be unambiguous grammar if every string generated by it produces exactly one.
Parse tree
Or syntax tree
Or leftmost derivation
Or rightmost derivation
Note: So, If we try to derive more than one tree of unambiguous grammar, then all trees will be similar

Examples of Unambiguous Grammar

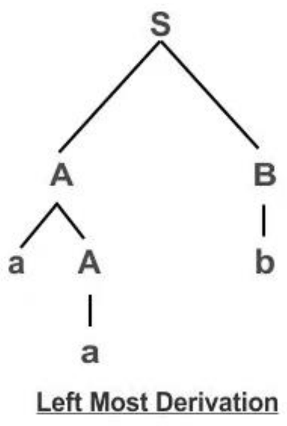
Example 01

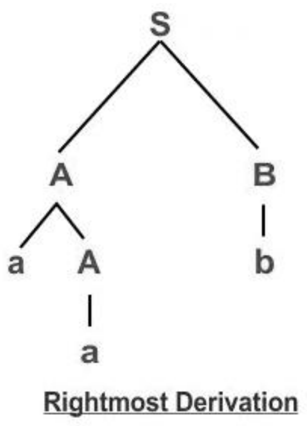
For string "aab" the following grammar is unambiguous

S → AB
A → Aa | a
B → b

Solution

Let's draw the leftmost and rightmost derivations of the above grammar to get the string "aab."





Because all parse trees, syntax trees, and left or right derivations will be similar for the above grammar of string "aab." So, the above grammar is unambiguous.

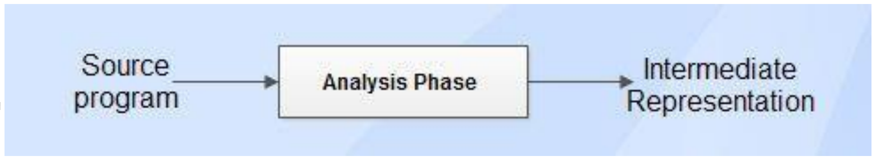
The Structure of a compiler (Phases of Compiler / Compiler Design)

The structure of compiler consists of two parts:

1. **Analysis part(also known as Front end)**
2. **Synthesis part(also known as Back end)**

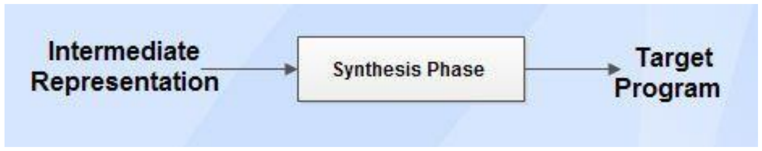
Analysis part

- Analysis part breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses this structure to create an intermediate representation of the source program.
- Information about the source program is collected and stored in a data structure called symbol table.



Synthesis part

• Synthesis part takes the intermediate representation as input and transforms it to the target program.



The design of compiler can be decomposed into several phases, each of which converts one form of source program into another.

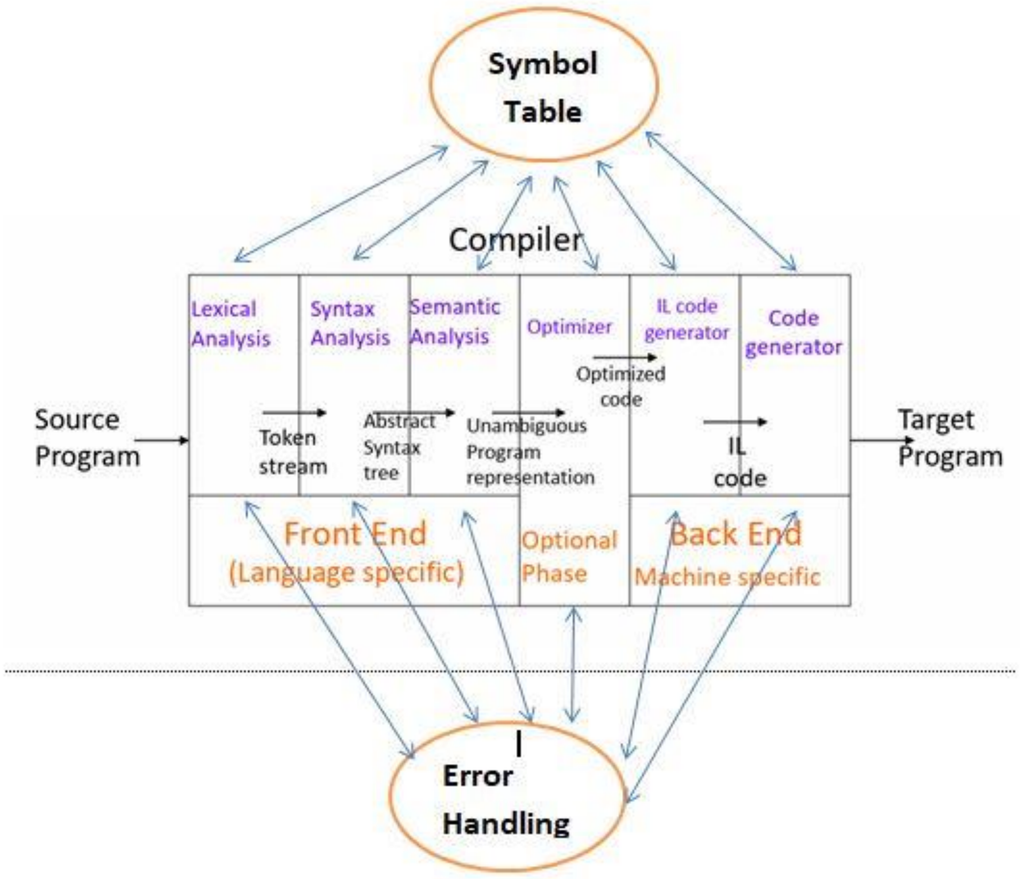
The different phases of compiler are as follows:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation
5. Code optimization
6. Code generation

All of the mentioned phases involve the following tasks:

• Symbol table management.

• Error handling.



1.Lexical Analysis

• Lexical analysis is the first phase of compiler which is also termed as scanning.

• Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.

• **Token:** Token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.

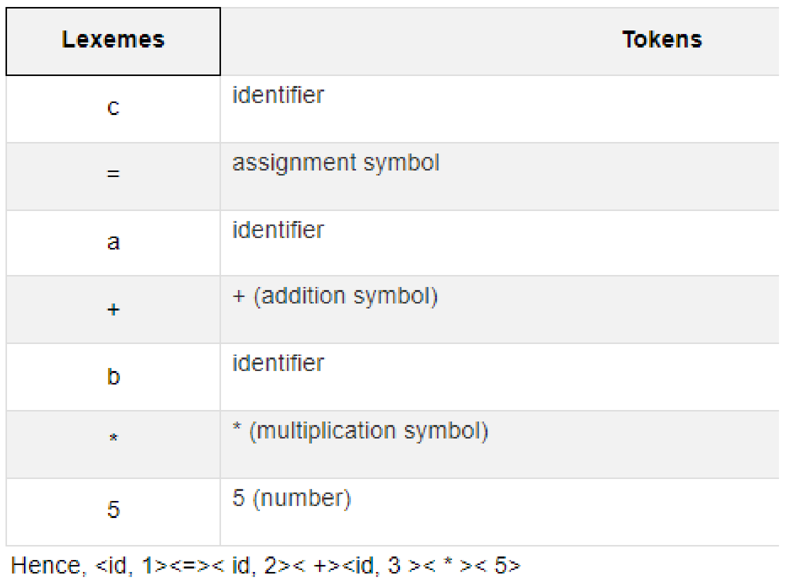
• **Lexeme:** Lexeme is instance of a token i.e., group of characters forming a token. ,

• **Pattern:** Pattern describes the rule that the lexemes of a token takes. It is the structure that must be matched by strings.

• Once a token is generated the corresponding entry is made in the symbol table.

Input: stream of characters
Output: Token
Token Template: <token-name, attribute-value>
(eg.) c=a+b*5;

Lexemes and tokens

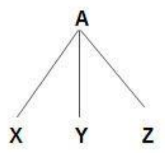


Lexemes	Tokens
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

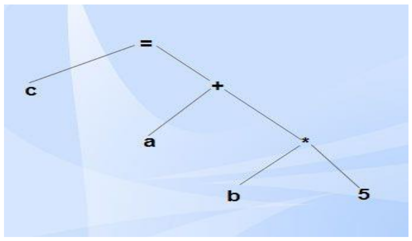
2. Syntax Analysis

• Syntax analysis is the second phase of compiler which is also called as parsing.
• Parser converts the tokens produced by lexical analyzer into a tree like representation called parse tree.
• A parse tree describes the syntactic structure of the input.



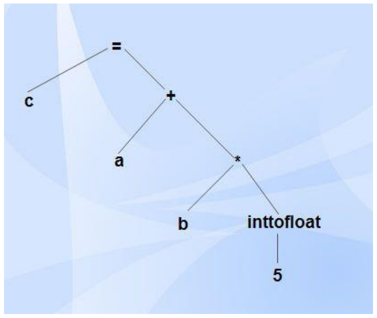
Syntax tree is a compressed representation of the parse tree in which the operators appear as interior nodes and the operands of the operator are the children of the node for that operator.
Input: Tokens

Output: Syntax tree



3. Semantic Analysis

• Semantic analysis is the third phase of compiler.
• It checks for the semantic consistency.
• Type information is gathered and stored in symbol table or in syntax tree.
• Performs type checking.



4. Intermediate Code Generation

• ***Intermediate code generation produces intermediate representations for the source program which are of the following forms:***

1. Postfix notation
2. Three address code
3. Syntax tree

Most commonly used form is the **three address code**

t1 = inttofloat (5)
t2 = id3* tl
t3 = id2 + t2
id1 = t3

Properties of intermediate code

• It should be easy to produce.
• It should be easy to translate into target program.

5. Code Optimization

•Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output.
• It results in faster running machine code.
• It can be done by reducing the number of lines of code for a program.
• This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
• During the code optimization, the result of the program is not affected.

To improve the code generation, the optimization involves

1. Deduction and removal of dead code (unreachable code).
2. Calculation of constants in expressions and terms.
3. Collapsing of repeated expression into temporary string.
4. Loop unrolling.
5. Moving code outside the loop.
6. Removal of unwanted temporary variables.

t1 = id3* 5.0
id1 = id2 + t1

6. Code Generation

• Code generation is the final phase of a compiler.
• It gets input from code optimization phase and produces the target code or object code as result.
• Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
• ***The code generation involves***

1. Allocation of register.php and memory.
2. Generation of correct references.
3. Generation of correct data types.
4. Generation of missing code.

```
LDF R2, id3
MULF R2, # 5.0
LDF R1, id2
```

```
ADDF R1, R2
STF id1, R1
```

Symbol Table Management

• Symbol table is used to store all the information about identifiers used in the program.
• It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
• It allows finding the record for each identifier quickly and to store or retrieve data from that record.
• Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

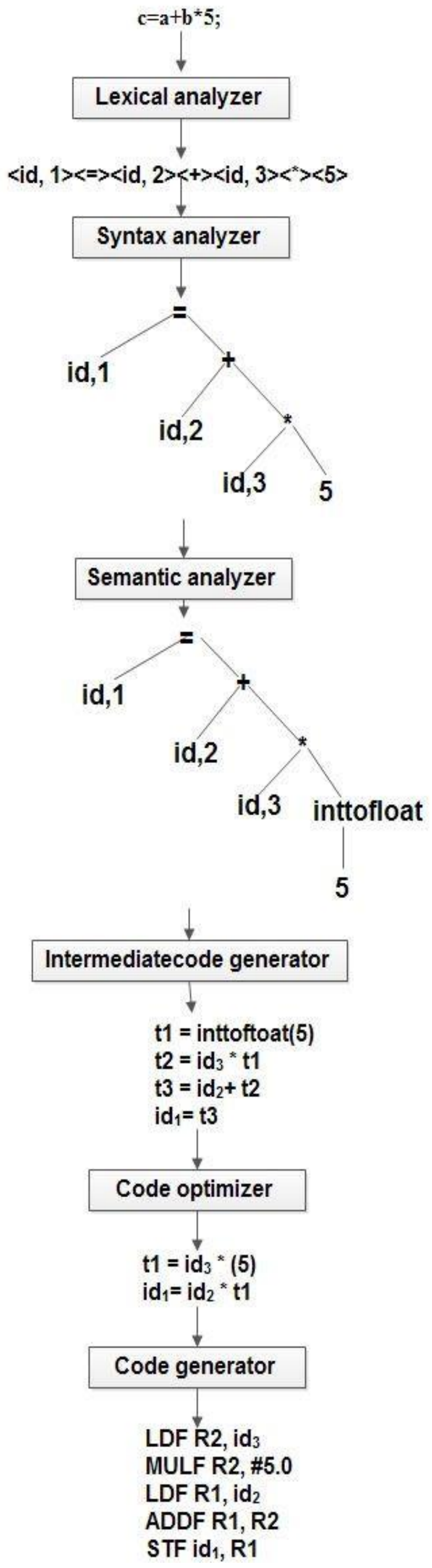
Example: **int a, b; float c; char z;**

Symbol name	Type	Address
a	Int	1000
b	Int	1002
c	Float	1004
z	char	1008

Example

1. **extern double test (double x);**
2. **double sample (int count) {**
3. **double sum= 0.0;**
4. **for (int i = 1; i &lt; = count; i++)**
5. **sum+= test((double) i);**
6. **return sum;**
7. **}**

Symbol name	Type	Scope
test	function, double	extern
x	double	function parameter
sample	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement



```
t1 = inttoftoat(5)
t2 = id3 * t1
t3 = id2+ t2
id1= t3
```

```
t1 = id3 * (5)
id1= id2 * t1
```

```
LDF R2, id3
MULF R2, #5.0
LDF R1, id2
ADDF R1, R2
STF id1, R1
```

Error Handling

• Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.

• In lexical analysis, errors occur in separation of tokens.

• In syntax analysis, errors occur during construction of syntax tree.

• In semantic analysis, errors may occur at the following cases:

(i) When the compiler detects constructs that have right syntactic structure but no meaning

(ii) During type conversion.

• In code optimization, errors occur when the result is affected by the optimization. In code generation, it shows error when code is missing etc.

Figure illustrates the translation of source code through each phase, considering the statement

c =a+ b * 5.

Error Encountered in Different Phases

Each phase can encounter errors. After detecting an error, a phase must somehow deal with the error, so that compilation can proceed.

A program may have the following kinds of errors at various stages:

Lexical Errors

It includes incorrect or misspelled name of some identifier i.e., identifiers typed incorrectly.

Syntactical Errors

- It includes missing semicolon or unbalanced parenthesis. Syntactic errors are handled by syntax analyzer (parser).
- When an error is detected, it must be handled by parser to enable the parsing of the rest of the input. In general, errors may be expected at various stages of compilation but most of the errors are syntactic errors and hence the parser should be able to detect and report those errors in the program.

The goals of error handler in parser are:

• Report the presence of errors clearly and accurately.

• Recover from each error quickly enough to detect subsequent errors.

• Add minimal overhead to the processing of correcting programs.

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

1. Panic mode.
2. Statement level.
3. Error productions.
4. Global correction

Semantical Errors

These errors are a result of incompatible value assignment. The semantic errors that the semantic analyzer is expected to recognize are:

• Type mismatch.

• Undeclared variable.

• Reserved identifier misuse.

• Multiple declaration of variable in a scope.

• Accessing an out of scope variable.

• Actual and formal parameter mismatch.

Logical errors

These errors occur due to not reachable code-infinite loop.

The Science of building a compiler

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is **infinite** and any program can be **very large**, consisting of possibly

millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the **meaning of the program being compiled**. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

Modeling in compiler design and implementation

The study of compilers is mainly a study of how we design the **right mathematical models** and choose the **right algorithms**, while balancing the need for generality and power against **simplicity and efficiency**.

Code optimization

The term "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code. **"Optimization"** is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

Finally, a compiler is a complex system; we must keep the system simple to assure that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, Implementing only those that lead to the greatest benefits on source programs encountered in practice.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of **solving complex and open-ended problems**. The approach used in compiler development involves both **theory and experimentation**. We normally start by formulating the problem based on our intuitions on what the important issues are.

Programming Language Basics

To design an efficient and structured compiler, we require some basics in programming languages.

The basic concepts that are present in the study of programming languages are:

1. **Difference between static and dynamic policy**

The **Static policy** lets the compiler decide when faced with an issue at compile time.
The policy that allows the compiler to take the decision during runtime is called as **Dynamic policy.**

2. **Meaning of environment and state**

t is important to ensure that the language that is being designed has the ability to map the data names with the respective data values before we start designing the language.

For example, the execution of an assignment such as **a = b + 2,** changes the value present in the variable x. In other words, the assignment changes the value in whatever location x is present in.

There are two stages of mapping

- **Environment**

It is a mapping wherein the data names are mapped to their respective location in the memory.

- **State**

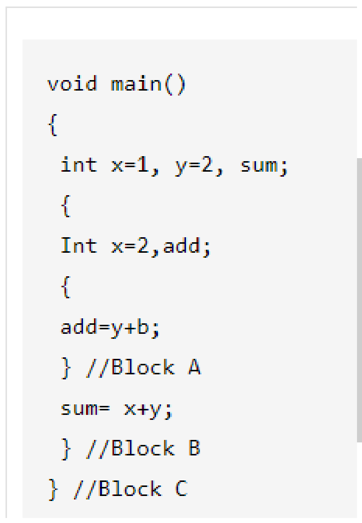
It is a mapping where the locations are mapped to their respective data values. This two-stage mapping is Dynamic in nature

3. Significance of static scope and block structures

Static scoping decides the declaration of a name by examining the program text only. Pascal and C are few languages that use static scoping.

The blocks in the program help to find out the scope of a variable.

Example:



```
void main()
{
 int x=1, y=2, sum;
 {
 Int x=2,add;
 {
 add=y+b;
 } //Block A
 sum= x+y;
 } //Block B
} //Block C
```

In the above example program. The scope of the variable ‘add’ is only until Block A, the scope of variable ‘sum’ is in Block B and the scope of variables x and y are in Block C.

We also see that the value of x varies in two different blocks. Here, the scope of x with value 2 is only in block A, whereas, the scope of x with value 1 is global, or scope is in block C.

4. Meaning of Dynamic Scope

Dynamic scoping determines the declaration of a name time with the help of current active variables. Snobol uses dynamic scoping

Example:

```
#define p(d+1)
void a()
{
 printf("%d", p);
}
void b()
{
 int d=0;
 printf("%d", p);
}
void main()
{
x();
y();
}
```

In the above example, the value of the variable p is determined at runtime during the execution of the functions. a() and b(). When the function a is called, the value of d remains the same as declared globally.

5. Methods of passing parameters

Call by Value

calling a function by value will cause the program to copy the contents of an object passed into a function. To implement this in C, a function declaration has the following form: [return type] functionName([type][parameter name],...).

Call by Value Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int x, int y){
   int temp = x;
   x = y;
   y = temp;
}
int main(){
   int x = 10;
   int y = 11;
   printf("Values before swap: x = %d, y = %d\n", x,y);
   swap(x,y);
   printf("Values after swap: x = %d, y = %d", x,y);
}
```

Output:

```
Values before swap: x = 10, y = 11
Values after swap: x = 10, y = 11
```

Call by reference

We can observe that even when we change the content of x and y in the scope of the swap function, these changes do not reflect on x and y variables defined in the scope of main. This is because we call swap() by value and it will get separate memory for x and y so the changes made in swap() will not reflect in main().

Calling a function by reference will give function parameter the address of original parameter due to which they will point to same memory location and any changes made in the function parameter will also reflect in original parameters. To implement this in C, a function declaration has the following form: [return type] functionName([type]* [parameter name],...).

Call by Reference Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int *x, int *y){
   int temp = *x;
   *x = *y;
   *y = temp;
}
int main(){
   int x = 10;
   int y = 11;
   printf("Values before swap: x = %d, y = %d\n", x,y);
   swap(&x,&y);
   printf("Values after swap: x = %d, y = %d", x,y);
}
```

Output:

```
Values before swap: x = 10, y = 11
```

Values after swap: x = 11, y = 10

We can observe in function parameters instead of using int x,int y we used int *x,int *y and in function call instead of giving x,y, we give &x,&y this methodology is call by reference as we used pointers as function parameter which will get original parameters' address instead of their value. & operator is used to give address of the variables and * is used to access the memory location that pointer is pointing. As the function variable is pointing to the same memory location as the original parameters, the changes made in swap() reflect in main() which we can see in the above output.

6. Aliasing

There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value. It is possible that two formal parameters can refer to the same location; such variables are said to be aliases of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

Example 1. 9 : Suppose a is an array belonging to a procedure p, and p calls another procedure q(x,y) with a call q(a,a). Suppose also that parameters are passed by value, but that array names are really references to the location where the array is stored, as in C or similar languages. Now, x and y have become aliases of each other. The important point is that if within q there is an assignment x[10] = 2, then the value of y[10] also becomes 2.

7. Explicit access control

- Classes and structures introduce a new scope for their members. If p is an object of a class with a field (member) x, then the use of x in p.x refers to field x in the class definition. In analogy with block structure, the scope of a member declaration x in a class C extends to any subclass C, except if C has a local declaration of the same name x.

- Through the use of keywords like public, private, and protected, object-oriented languages such as C + + or Java provide explicit control over access to member names in a superclass. These keywords support encapsulation by restricting access. Thus, private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C + + term). Protected names are accessible to subclasses. Public names are accessible from outside the class.

- In C + +, a class definition may be separated from the definitions of some or all of its methods. Therefore, a name x associated with the class C may have a region of the code that is outside its scope, followed by another region (a method definition) that is within its scope. In fact, regions inside and outside the scope may alternate, until all the methods have been defined.

PART-II

LEXICAL ANALYSIS

Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.

A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.

Lexical analysis consists of two stages of processing which are as follows:

- Scanning
- Tokenization

Token, Pattern and Lexeme

Token: Token is a valid sequence of characters which are given by lexeme. In a programming language

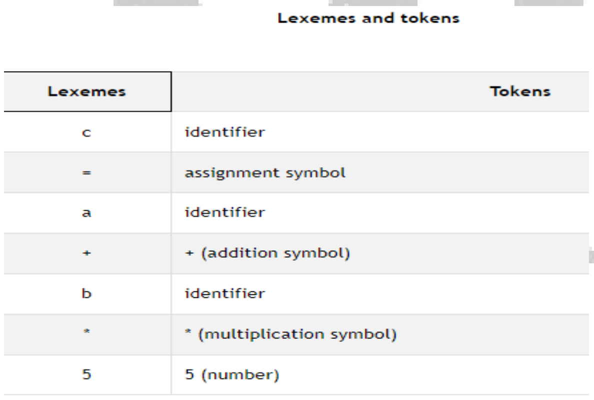
- keywords,
- constant,
- identifiers,
- numbers,
- operators and
- punctuations symbols

are possible tokens to be identified.

Pattern: Pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token. It can be defined by regular expressions or grammar rules.

Lexeme : Lexeme is a sequence of characters that matches the pattern for a token i.e., instance of a token.

(eg.) c=a+b*5;


Lexemes and tokens

Lexemes	Tokens
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

The sequence of tokens produced by lexical analyzer helps the parser in analyzing the syntax of programming languages.

Role of Lexical Analyzer

Lexical analyzer performs the following tasks:

• Reads the source program, scans the input characters, group them into lexemes and produce the token as output.
• Enters the identified token into the symbol table.
• Strips out white spaces and comments from source program.
• Correlates error messages with the source program i.e., displays error message with its occurrence by specifying the line number.

• Expands the macros if it is found in the source program.

Tasks of lexical analyzer can be divided into two processes:

Scanning: Performs reading of input characters, removal of white spaces and comments.

Lexical Analysis: Produce tokens as the output.

Need of Lexical Analyzer

- Simplicity of design of compiler The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.
- Compiler efficiency is improved Specialized buffering techniques for reading characters speed up the compiler process.
- Compiler portability is enhanced

Issues in Lexical Analysis

Lexical analysis is the process of producing tokens from the source program. It has the following issues:
• Lookahead
• Ambiguities
Lookahead is required to decide when one token will end and the next token will begin. The simple example which has lookahead issues are i vs. if, = vs. ==. Therefore a way to describe the lexemes of each token is required.

A way needed to resolve ambiguities

• Is if it is two variables i and f or if?
• Is == is two equal signs =, = or ==?
• arr(5, 4) vs. fn(5, 4) II in Ada (as array reference syntax and function call syntax are similar.

Hence, the number of lookahead to be considered and a way to describe the lexemes of each token is also needed.
Regular expressions are one of the most popular ways of representing tokens.

Ambiguities

The lexical analysis programs written with lex accept ambiguous specifications and choose the longest match possible at each input point. Lex can handle ambiguous specifications. When more than one expression can match the current input, lex chooses as follows:

• The longest match is preferred.
• Among rules which matched the same number of characters, the rule given first is preferred.

Lexical Errors

• A character sequence that cannot be scanned into any valid token is a lexical error.
• Lexical errors are uncommon, but they still must be handled by a scanner.
• Misspelling of identifiers, keyword, or operators are considered as lexical errors.
Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Error Recovery Schemes

1. Panic mode recovery

In panic mode recovery, unmatched patterns are deleted from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

(eg.) For instance 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 f iis a misspelling of the keyword if or an undeclared function identifier.
Since f i is a valid lexeme for the token id, the lexical analyzer will return the token id to the parser.

2. Local correction

- Source text is changed around the error point in order to get a correct text.
- Analyzer will be restarted with the resultant new text as input.

Local correction performs deletion/insertion and/or replacement of any number of symbols in the error detection point.
(eg.) In Pascal, c[i] '='; the scanner deletes the first quote because it cannot legally follow the closing bracket and the parser replaces the resulting'=' by an assignment statement.
Most of the errors are corrected by local correction.

(eg.) The effects of lexical error recovery might well create a later syntax error, handled by the parser. Consider

· · · for $tnight · · ·
The $ terminates scanning of for. Since no valid token begins with $, it is deleted. Then **tnight** is scanned as an identifier.

In effect it results,
· · · fortnight · · ·
Which will cause a syntax error? Such false errors are unavoidable, though a syntactic error-repair may help.

3. Global correction

- It is an enhanced panic mode recovery.
- Preferred when local correction fails.

Lexical error handling approaches

Lexical errors can be handled by the following actions:

• Deleting one character from the remaining input.
• Inserting a missing character into the remaining input.
• Replacing a character by another character.
• Transposing two adjacent characters.

Input Buffering

1. To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.
2. Hence a two-buffer scheme is introduced to handle large look a heads safely.
3. Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer end have been adopted. Input Buffering

There are three general approaches for the implementation of a lexical analyzer:

(i) By using a lexical-analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.

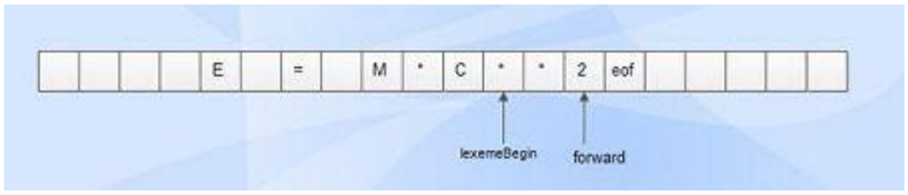
(ii) By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.

(iii) By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Buffer Pair and Sentinels

Because of large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Fig shows the buffer pairs which are used to hold the input data.



Scheme

• Consists of two buffers, each consists of N-character size which are reloaded alternatively.
• N-Number of characters on one disk block, e.g., 4096.
• N characters are read from the input file to the buffer using one system read command.
• eof is inserted at the end if the number of characters is less than N.

Pointers

Two pointers lexemeBegin and forward are maintained.

lexeme Begin points to the beginning of the current lexeme which is yet to be found.
forward scans ahead until a match for a pattern is found.

• Once a lexeme is found, lexemebegin is set to the character immediately after the lexeme which is just found and forward is set to the character at its right end.
• Current lexeme is the set of characters between two pointers.

Disadvantages of this scheme

• This scheme works well most of the time, but the amount of lookahead is limited.
• This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.
(eg.) DECLARE (ARGl, ARG2, . . . , ARGn) in PL/1 program;

• It cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

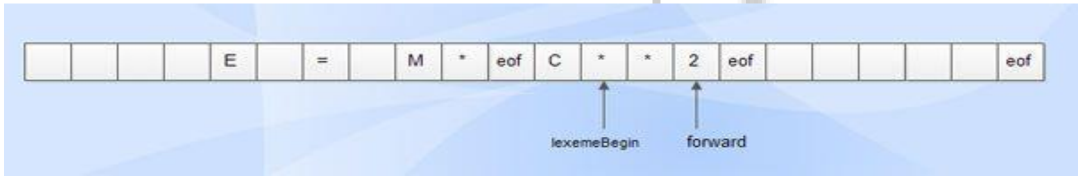
Sentinels

• In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.

• Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.
Test 1: For end of buffer.
Test 2: To determine what character is read.

• The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.

• The sentinel is a special character that cannot be part of the source program. (eof character is used as sentinel).



Advantages

• Most of the time, It performs only one test to see whether forward pointer points to an eof.
• Only when it reaches the end of the buffer half or eof, it performs more tests.
• Since N input characters are encountered between eofs, the average number of tests per input character is very close to 1.

Recognition of Tokens

Using token type and Token value:

- For a programming language, there are various types of tokens such as identifier, Keywords, constants and operators and so on.
- The token is usually represented by a pair of token type and token value.
  Token type Token value
- The token type tells us the category of token and token value gives us the Information regarding token. The token value is also called token attribute.
- The token value can be a pointer to symbol table in case of identifier and constants.
- The lexical analyzer reads the input program and uses symbol table for tokens.
- For example, consider the following symbol table,

Token	Code	Value
if	1	-
else	2	-
while	3	-
for	4	-
identifier	5	ptr to symbol table
constant	6	ptr to symbol table
<	7	1
< =	7	2
>	7	3
>=	7	4
!=	7	5
(	8	1
)	8	2
+	9	1
-	9	2
=	10	-

Consider a program code as,

```
if ( a < 10 )
 i = i + 2;
else
 i = i – 2;
```

The corresponding symbol table for identifiers and constants will be,

Location counter	Type	Value
100	identifier	a
…	…	…
105	constant	10
…	…	…

107	**identifier**	**b**
…	**…**	**…**
110	**constant**	**2**

Our lexical analyzer will generate the following token stream
1, (8, 1), (5, 100), (7, 1), (6 , 105), (8, 2), (5, 107), 10, (5, 107), (9, 1) (6, 110),
2 , (5, 107) , 10, (5, 107), (9, 2), (6, 110)

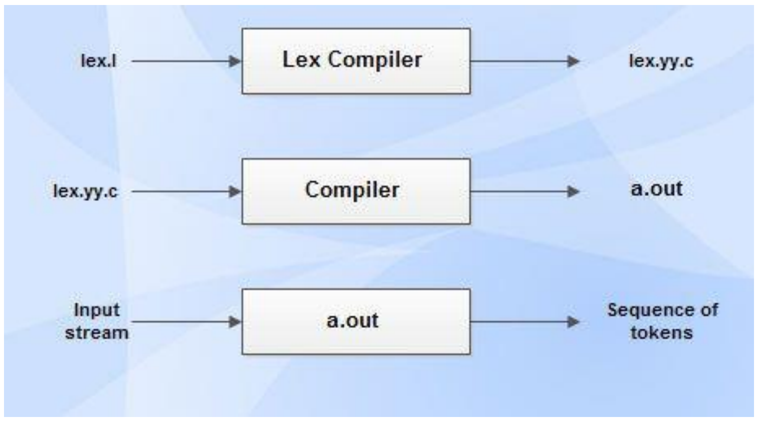
Using transition diagram

- The transition diagram is a directed graph with states drawn as circles (nodes) and edges representing transitions on input symbols. Edges are the arrows connecting the states.
- The transition diagram has a start state which indicates the beginning of token and final state which indicates the end of token.
- The fptr scans the input character by character from left to right.
- **For example, Write a regular expression for identifier and keyword and design a transition diagram for it.**
  Ans: RE = letter (letter + digit) *
- The installID ( ) checks if lexeme is already in table. If it is not present then it will install it. The gettoken( ) examines the lexeme and returns the token name as id or a reserve keyword.

The Lexical-Analyzer Generator Lex

Lex is a tool in lexical analysis phase to recognize tokens using regular expression.

• Lex tool itself is a lex compiler.



• lex.l is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms lex.l to a C program known as lex.yy.c.
• lex.yy.c is compiled by the C compiler to a file called a.out.
• The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.
• yylval is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.

The attribute value can be numeric code, pointer to symbol table or nothing.
Another tool for lexical analyzer generation is Flex.

Structure of Lex Programs

Lex program will be in following form
declarations
%%
translation rules
%%
auxiliary functions

Declarations This section includes declaration of variables, constants and regular definitions.
Translation rules It contains regular expressions and code segments.
Form : Pattern {Action}
Pattern is a regular expression or regular definition.
Action refers to segments of code.
Auxiliary functions This section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer.

Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found.
Once a match is found, the associated action takes place to produce token.
The token is then given to parser for further processing.

Conflict Resolution in Lex

Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following:

• Always prefer a longer prefix than a shorter prefix.
• If two or more patterns are matched for the longest prefix, then the first pattern listed in lex program is preferred

Lookahead Operator

• Lookahead operator is the additional operator that is read by lex in order to distinguish additional pattern for a token.
• Lexical analyzer is used to read one character ahead of valid lexeme and then retracts to produce token.
• At times, it is needed to have certain characters at the end of input to match with a pattern. In such cases, slash (/) is used to indicate end of part of pattern that matches the lexeme.
(eg.) In some languages keywords are not reserved. So the statements

IF (I, J) = 5 and IF(condition) THEN

results in conflict whether to produce IF as an array name or a keyword. To resolve this the lex rule for keyword IF can be written as,
IF/\ (.* \) {
letter }

Finite automata

Finite automata is a finite state machine that acts as a recognizer for a language. When it is provided with an input string it accepts or rejects the string based on whether the string is from the language or not.

FA recognize the regular expression that is a set of strings and accepts it if it represents a regular language else it rejects it. When finite automata accept the regular expression, it compiles it to form a transition diagram.

In finite automata, there is a finite state for each input. FAcan be categorized in two ways deterministic finite automata and non-deterministic finite automata

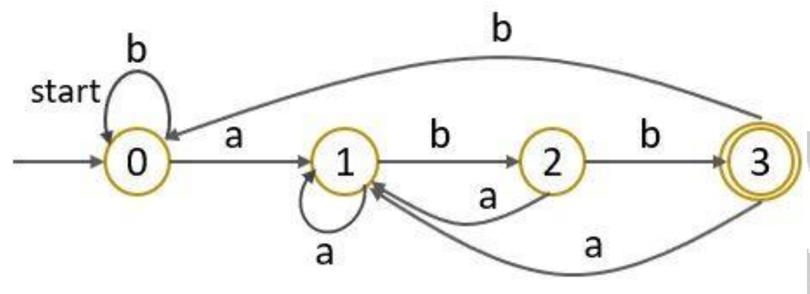
Types of FA: There are two kind of FA:

Deterministic Finite Automata (DFA)
Non-Deterministic Finite Automata (NFA)
Deterministic FA
In deterministic finite automata in response to a single input alphabet, the state transits to only one state. In DFA, no null moves are accepted.

The DFA have five tuples as discussed below:

{Q, Σ, q0, F, δ}

Q = set of all states
Σ = Set of all input alphabets
q0 = initial state
F = Set of Final State
δ = Transition function
The DFA can be represented by the transition graph. In the transition graph, the nodes of the graph represent the states and the edges represent the transition of one state to another. Let us take an example of DFA:



Deterministic Finite Automata

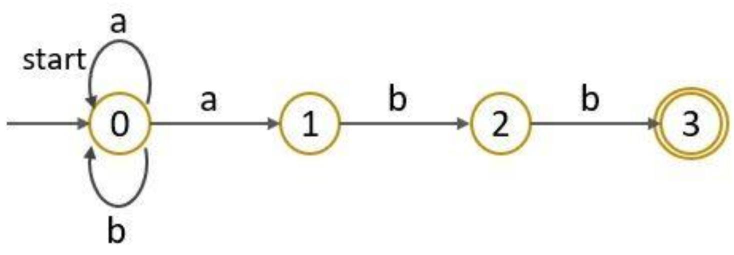
In the above example of DFA, you can notice that with a single input 'a' state '0' transit t only one state '1'. The transition table for the above DFA is as below:

State	a	b
0	{1}	{0}
1	{1}	{2}
2	{1}	{3}
3	{1}	{0}

Transition Table of DFA

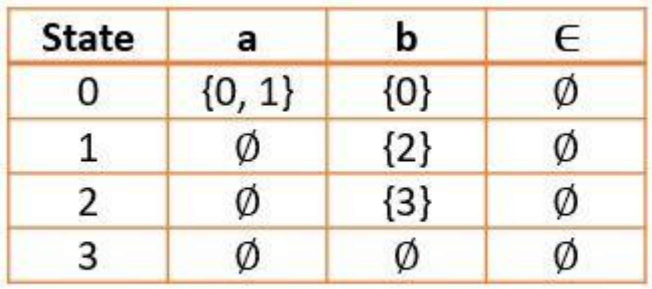
Non-Deterministic FA

In non-deterministic finite automata, in response to a single input alphabet, a state may transit to more than one state. The NFA also accept the NULL move.



Non-Deterministic Finite Automata

In the above example of NFA, you can notice that the state '0' with the input symbol 'a' can either transit to itself or to state '1'. Now let us create a transition table for the NFA above



State	a	b	∈
0	{0, 1}	{0}	Ø
1	Ø	{2}	Ø
2	Ø	{3}	Ø
3	Ø	Ø	Ø

Transition Table of NFA

With the transition table, it becomes easy to identify the transition on a given state with the given input. Although it takes a lot of space if there are a lot of input alphabets as most of the state do not perform any transition on most of the input symbols.

Regular Expression

A regular expression is a string that represents a regular language. An expression is said to be regular if the string is represented as:

a*-> it means zero or more occurrence of a
{∈, a, aa, aaa, aaaa…} // here * is a Kleene closure
a+-> it means one or more occurrences of a

{ a, aa, aaa, aaaa…} // here + is a positive closure
a.b -> it means the concatenation i.e. a will be followed by b
a|b -> it means the union i.e. either 'a' will appear or 'b'
So, this is the way in which the regular expression can represent a regular language. Like we follow the precedence of operators in mathematic like **BODMAS**.

Here also we have precedence of operators in sequence:

- Kleene Closure {a*}
- Positive Closure {a+}
- Concatenation {a.b}
- Union {a|b}

A regular expression is said to be valid we are able to derive the expression using primitive regular expressions using the finite application of rules that are a*, a+, a.b, a|b.

Finite Automata to Regular Expression

There are two methods to convert finite automata to the regular expression:

Ardern's Method
State Elimination

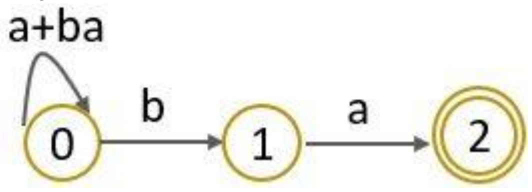
STATE ELIMINATION METHOD

The State elimination method follows the following general set of rules:

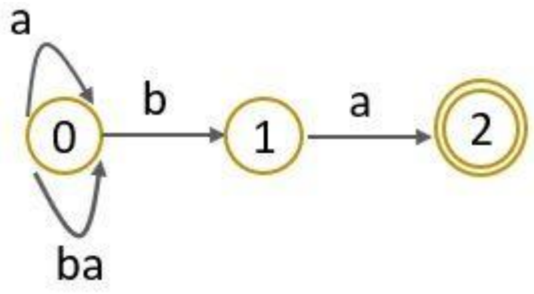
- Add a new initial state **(I).** Make a null transition from the old initial state to the new initial state.
- Add a new final state **(F).** Make null transition(s) to the new final state.
- Eliminate all states, except **I** and **F**, in the given finite automaton. Perform the elimination of states by checking the in-degrees and out-degrees of states and taking a cross product.
- After steps 1 and 2, the **I** state will not have any inward transitions, and the state **F** state will not have any outward transitions.

Example : Regular Expression (a+ba)*ba to Automata?

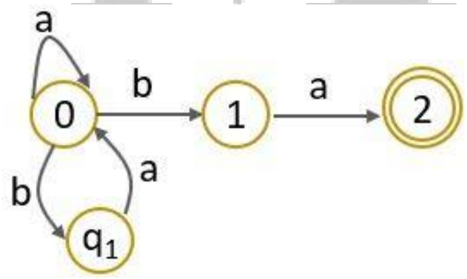
Step 1



Step 2



Step 3

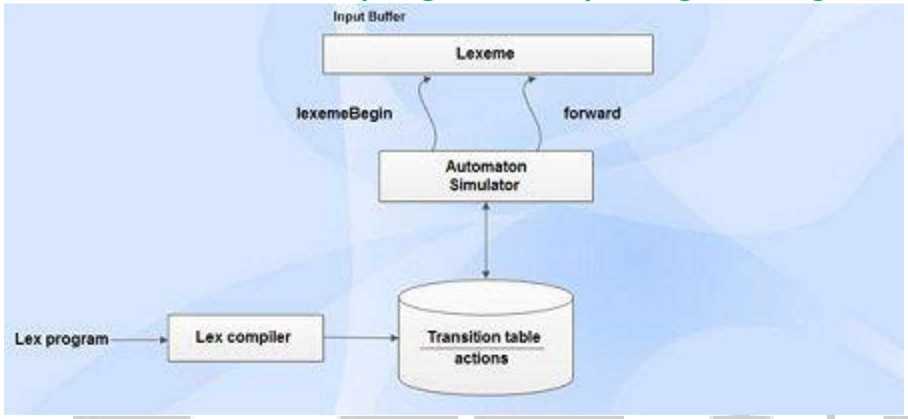


Design of Lexical Analyzer Generator

- Lexical analyzer can either be generated by NFA or by DFA.
- DFA is preferable in the implementation of lex.

Structure of Generated Analyzer

Architecture of lexical analyzer generated by lex is given in Fig.



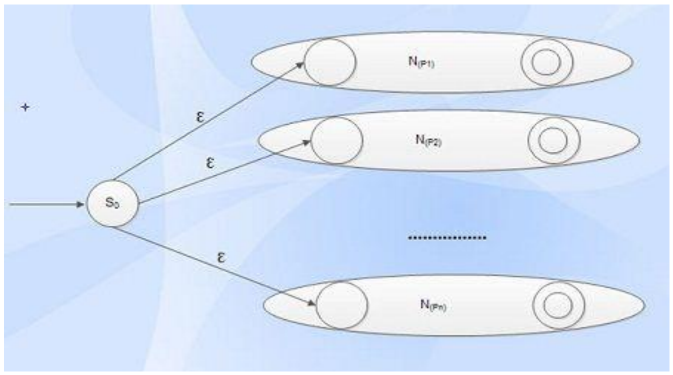
Lexical analyzer program includes:

- Program to simulate automata
- Components created from lex program by lex itself which are listed as follows:

1. A transition table for automaton.
2. Functions that are passed directly through lex to the output.
3. Actions from input program (fragments of code) which are invoked by automaton simulator when needed.

Steps to construct automaton

Step 1: Convert each regular expression into NFA either by Thompson’s subset construction or Direct Method.
Step 2: Combine all NFAs into one by introducing new start state with s-transitions to each of start states of NFAs Ni for pattern Pi·
Step 2: is needed as the objective is to construct single automaton to recognize lexemes that matches with any of the patterns.

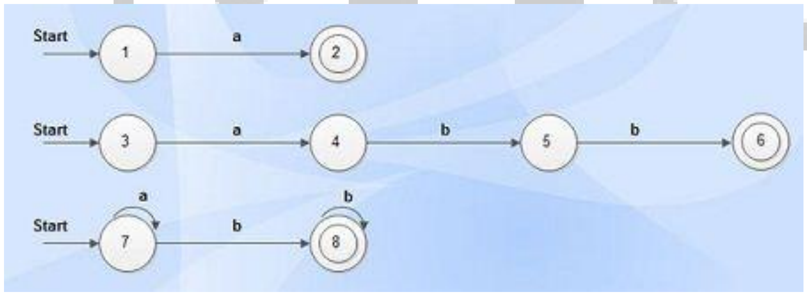


Example: a {action A1 for pattern Pl}
abb { action A2 for pattern P2 }
a*b+ { action A3 for pattern P3 }

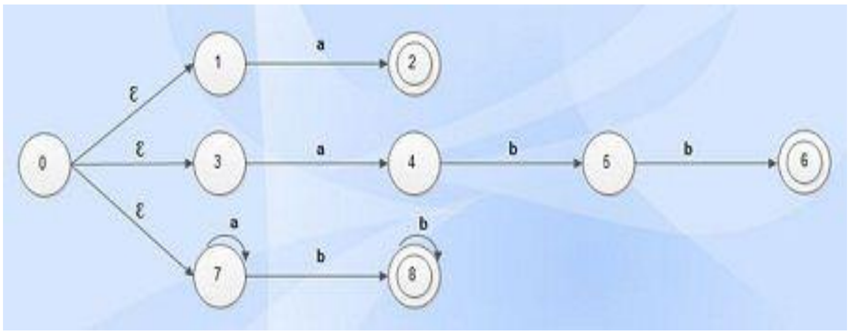
For string obb, pattern P2 and pattern p3 matches. But the pattern P2 will be taken into account as it was listed first in lex program.
For string aabbb · · · , matches pattern p3 as it has many prefixes.

Fig. Shows NFAs for recognizing the above mentioned three patterns.

The combined NFA for all three given patterns is shown in Fig.



S.SOUNDARYA(Assistant Professor)



Optimization of DFA-Based Pattern Matchers.

Optimization of DFA (Deterministic Finite Automaton) based pattern matchers is an important task to improve the efficiency and performance of pattern matching algorithms. Here are some key techniques for optimizing DFA pattern matchers:

1. **DFA Minimization:** The DFA can be minimized by removing unreachable states and merging equivalent states. This reduces the number of states and transitions, resulting in a more compact DFA and faster matching.

2. **State Compression:** The DFA can be compressed by grouping states with similar transitions into a single state. This reduces the memory footprint and improves cache efficiency during matching.

3. **Transition Table Optimization:** The transition table can be optimized to use a more efficient data structure, such as a sparse matrix or a compressed representation. This reduces the memory usage and speeds up the transition lookups.

4. **Transition Function Memoization:** The results of transition function lookups can be memoized to avoid redundant computations. This is particularly useful when the DFA has a large number of states and transitions.

5. **Preprocessing and Precomputation:** Some computations, such as character class lookups or pattern preprocessing, can be done offline or during the DFA construction phase to speed up the matching process.

6. **Parallelization:** The DFA matching process can be parallelized by dividing the input text into chunks and matching them concurrently using multiple threads or processes. This can significantly improve the matching speed on multi-core systems.

7**. Hybrid Approaches:** Hybrid approaches combine multiple matching algorithms, such as DFA and NFA (Non-deterministic Finite Automaton), to optimize performance based on specific patterns or input characteristics. This allows for efficient matching in different scenarios.

These optimization techniques aim to reduce the time and space complexity of DFA pattern matching algorithms, enabling faster and more efficient pattern matching in various applications such as text processing, string searching, lexical analysis, and network intrusion detection**.**

UNIT-III
Push Down Automata:

Context Free Language

Context-free grammar has production rules that are used to generate context-free language. Context-free languages are accepted by push-down automata (PDA).

- **PDA accepts all Regular languages and CFL.**
- For all context-free languages, there must be one comparison in terminals of string e L= {$a^n b^n$ n>=0}

A context-free Grammar (CFG) is a 4-tuple such that

where

- **V** = Finite set of variables / non-terminal symbols. These are denoted by capital letters.
- **T** = Finite set of terminal symbols. These are used to replace the variables. These are denoted by lowercase letters.
- **P** = Production rules are used to generate CFL. i.e. $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$. Remember that the left side must contain only one variable, but the right side ($(V \cup T)^*$) may contain more than one terminal and non-terminal.
- **S** = Start symbol

Example Of Context-Free Language

Consider a grammar G = (V, T, P, S) where-

- V = { S }
- T = { a , b }
- P = { $S \to aSbS$, $S \to bSaS$, $S \to \in$ }
- S = { S }

The above production rules generate the strings having an equal number of a's and b's. It is an example of context-free language.

DEFINITION OF THE PUSHDOWN AUTOMATON

A pushdown automaton (PDA) is a way to implement a context-free grammar (CFG) in a similar way we design Finite Automata (FA) for a regular grammar (RG). A DFA has limited memory, but a PDA can hold unlimited memory.

Basically, a Pushdown Automaton is

"Finite Automata Machine" + "a stack", where

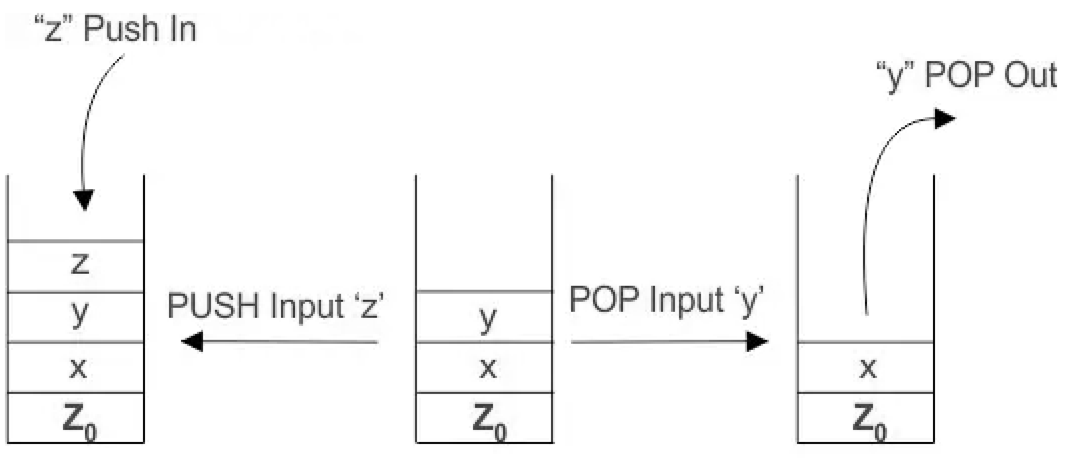
- FA represents a finite memory for a finite number of states
- A stack is an extra memory with infinite size to PUSH and POP values.

Operation of Pushdown Automata

There are two basic Operations in Pushdown Automata.

1. PUSH: It means inserting a new input symbol into the stack.

2. POP: It means removing an input symbol from the stack.



Stack

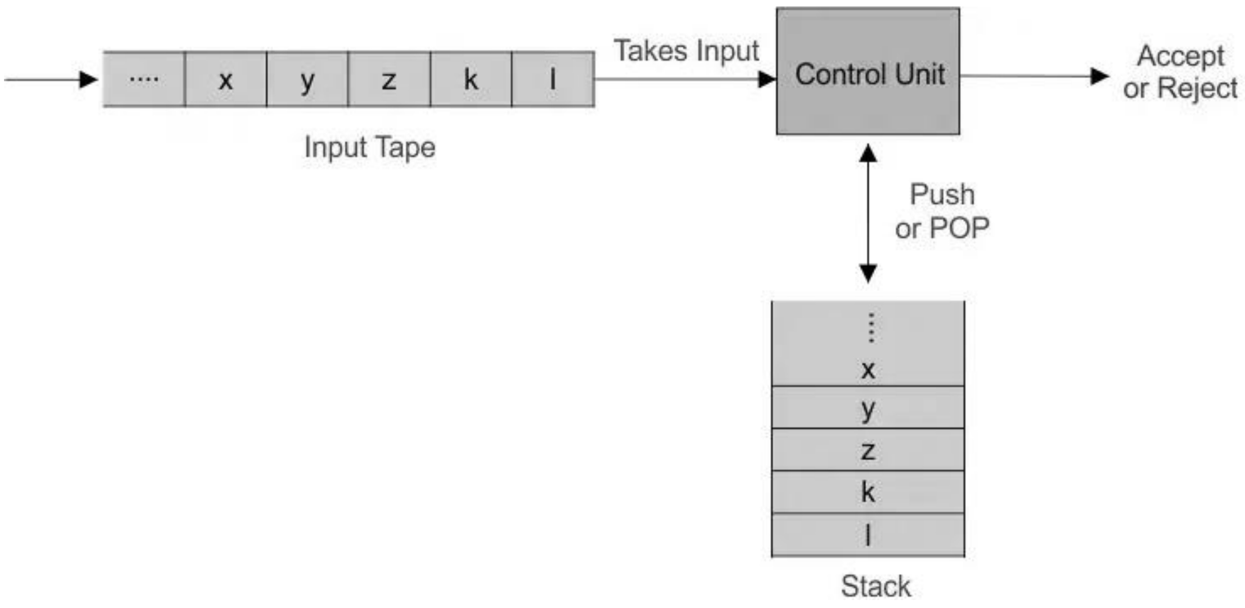
Note: PUSH and POP operations are always performed on the top of a stack

Components of Pushdown Automata

A pushdown automaton has 3 basic components

- **Input tape:** It contains the input string.
- **Control unit:** It takes an input and forwards it to the stack for PUSH or POP.
- **Stack:** It is the memory of infinite size.

The following diagram explains the above components.



Note: A PDA may or may not PUSH or POP an input symbol in every transition. But PDA has to read the top of the stack in every transition.

7-Tuples of Pushdown Automata

Pushdown Automata (PDA) can be formally described by 7-tuples (Q, ∑, Γ, δ, q_0, Z_0, F)

- **Q**is, the finite number of states,
- ∑is the input alphabet, which is PUSH or POP from Stack.

- Γis stack symbols.
- **Transition function:** Q × (∑ ∪ {ε}) x Γ à Q × Γ * // where Q is any state ∑ is the input symbol, ε is epsilon, Γ is the top of the stack value, and Γ * represents all symbols existing in the stack.
- **q_0** is the start state (q_0 ∈ Q)
- **Z_0 is** the default value of the stack. It is used to represent the Null value of the stack. We can say Z_0 is an epsilon.
- **F**is a set of Final states (F ∈ Q). Final states may or may not exist in PDA.

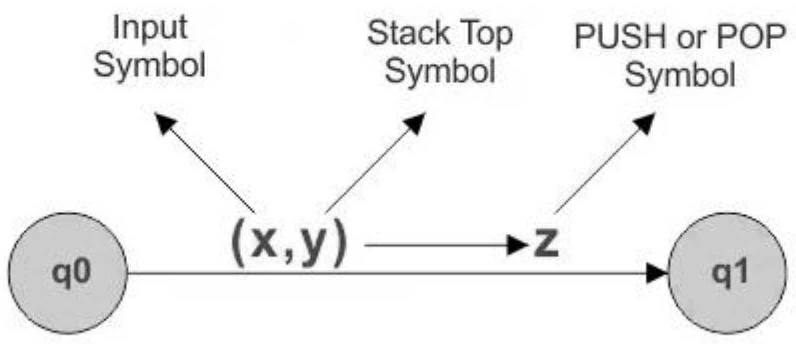
Transition Function in PDA

The transition function depends on the current state and three symbols

- Input symbol
- Stack Top Symbol
- Push Symbol

The Value of any symbol (Input stack or push) can be epsilon (∈).

Let us see the following diagram, which shows a transition in a PDA from a state q_0 to state q_1, labeled as x,y → z



This means at state q0, if we consume an input string 'x' and the top symbol of the stack is 'y', then we pop 'y' and push 'z' on top of the stack and move to the next state q1.

There are three essential cases in Pushdown Automata (PDA) to understand the functionality of the transition function.

As we know, the transition function is Q × (∑ ∪ {ε}) x Γ à Q × Γ *. Where

- Γ represents the **symbol which is at the Top of the Stack** before the transition function
- Γ* represents all the symbols that are present in the Stack after the Transition function.

Example of PDA

Design a PDA for accepting a language $\{0^n1^n \mid n>=1\}$.

According to the given language, consider an Input string with an equal number of 0s and 1s as “0011∈”. Whenever the epsilon (∈) occurs as input, then that transition is the last.

So, we will apply a very simple logic is that for every single ‘1’ only one ‘0’ should get POP out from the stack.

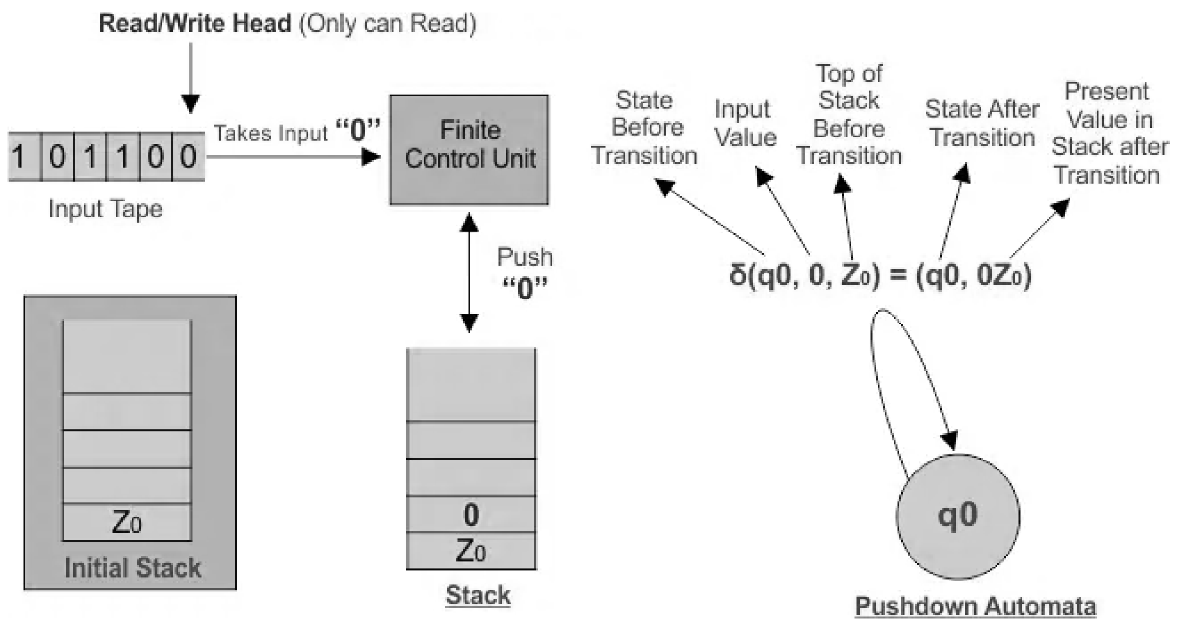
Explanation of Transition Function

Let’s understand all three cases of transition function with a given input string.

Case 01: If the symbols of Γ and Γ* are different then **PUSH** the symbol of Γ in stack.

To perform the PUSH Operation for the **first input “0”,** the transition function and detail diagram are given below

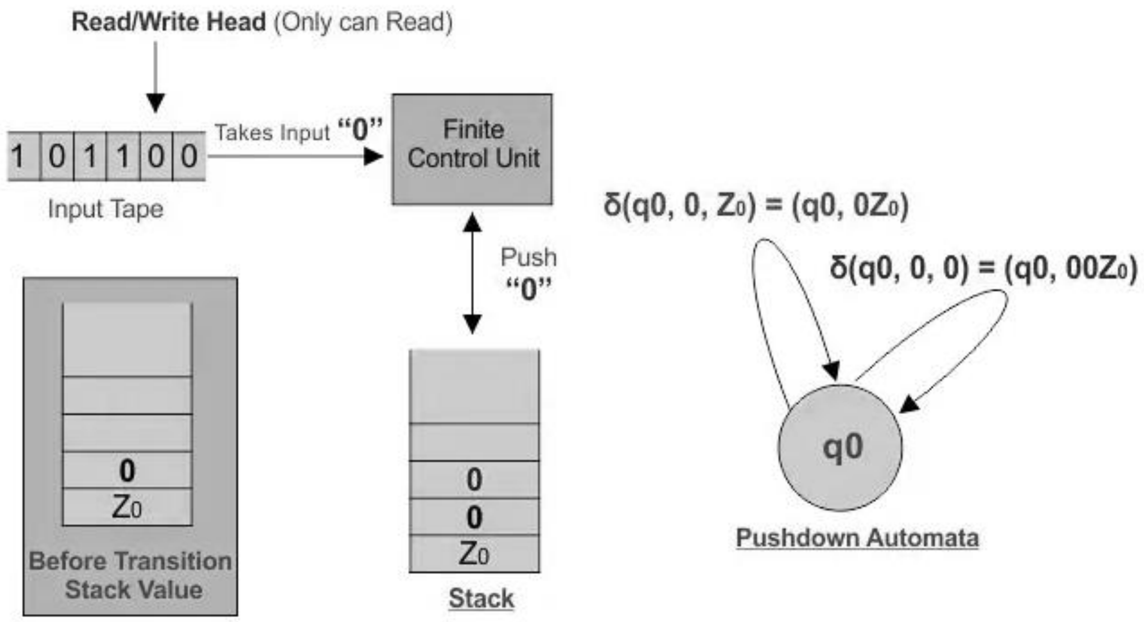
δ(q0, 0, Z0) = (q0, 0Z0)



Note: Z0 is the stack default value, which is also known as epsilon.

To perform the PUSH Operation for the **second input "0",** the transition function and detail diagram are given below.

δ (q0, 0, 0) = (q0, $00Z_0$)



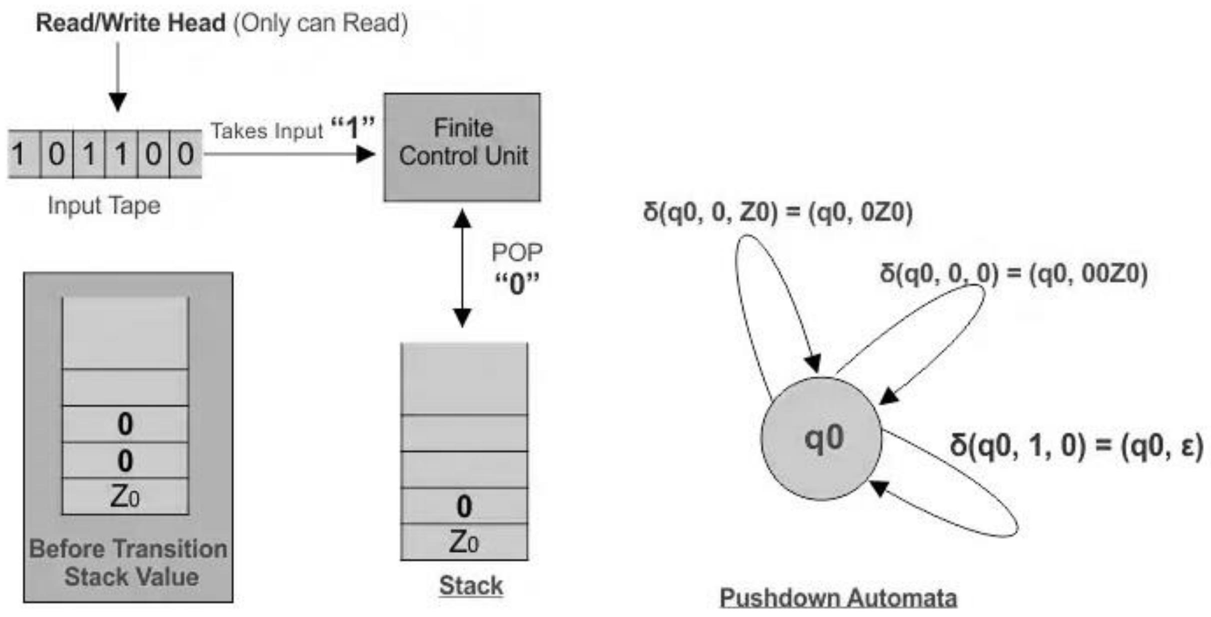
Note: After each transition, the state can either change or remain the same.

ase 02: If the value of Γ is non-empty but the value of Γ* is epsilon (empty)

then **POP** the value of Γ from stack. Let see the following transition function example

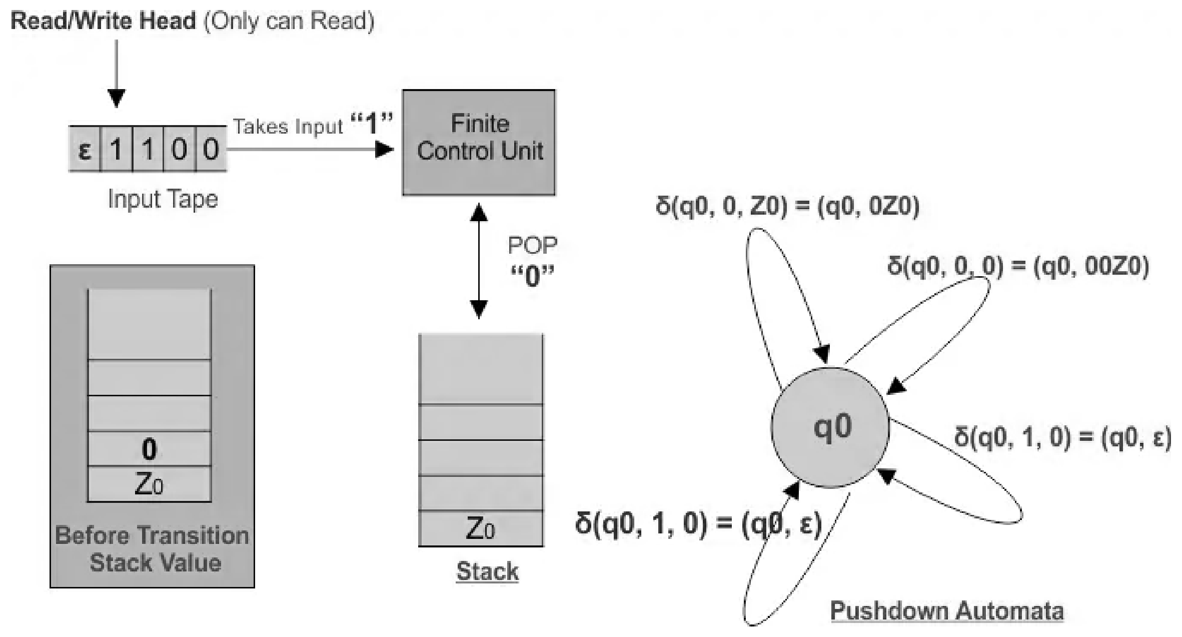
To perform the POP Operation for the **third input "1",** the transition function and detail diagram are given below.

δ(q0, 1, 0) = (q0, ∈) // Now stack hold value "$0Z_0$"



To perform the POP Operation for the **fourth input "1",** the transition function and detail diagram are given below

δ(q0, 1, 0) = (q0, ∈) // Now stack hold value "Z_0"

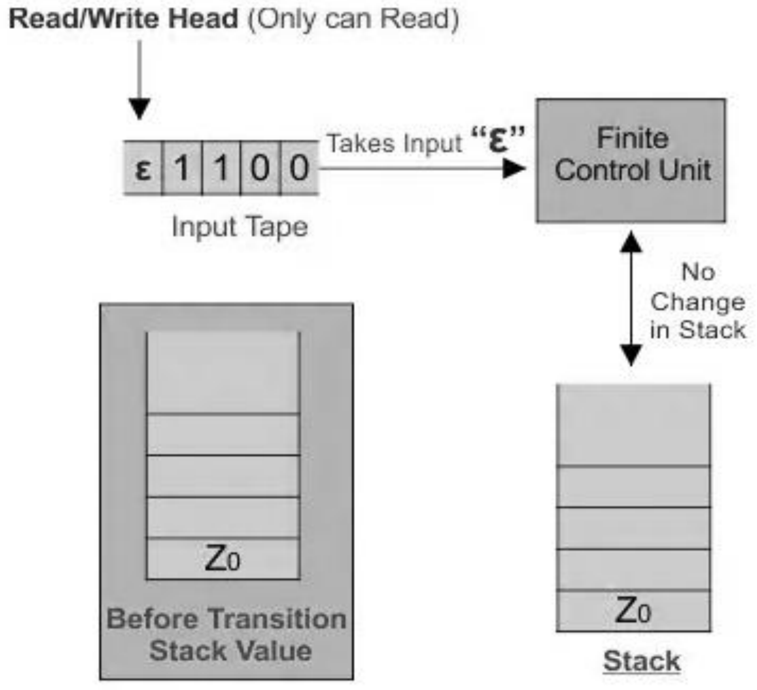


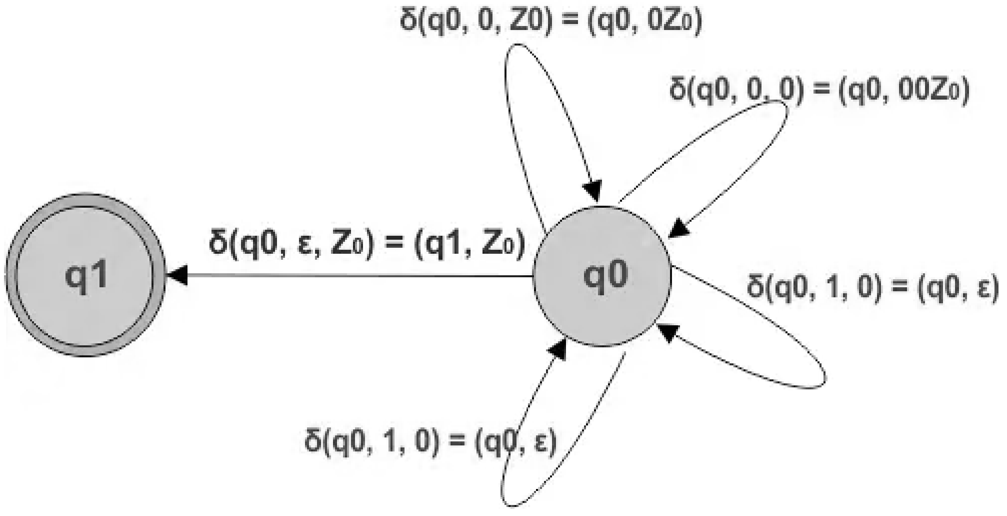
Note: It means both values are POP out from the stack through the above transition functions.

Case 03: If the value of Γ and Γ* are both the same (i.e. both empty) in value then there will be **no change** in stack. Let see the following transition function example

At this stage, the stack is empty (Z0), and the last Input string having the last symbol is epsilon (∈). So, there will be no change in the stack. The transition function with a detailed diagram is given below

δ (q0, ∈, Z_0) = (q1, ∈)   // Z_0 represents epsilon





Pushdown Automata

The above transition function has no effect on the stack but can change the current state (i.e., q0 to q1).

In Pushdown Automata (PDA),

- Only One Input symbol is read at a time.
- Only one input symbol is either PUSH or POP in the stack at a time.

Equivalence of PDA's and CFG's The goal is to prove that the following three classes of the languages are all the same class. 1. The context-free languages (The language defined by CFG's). 2. The languages that are accepted by empty stack by some PDA. 3. The languages that are accepted by final state by some PDA

Figure 1: Organization of constructions showing equivalence of three ways of defining the CFL's We have already shown that (2) and (3) are the same. Now, we prove that (1) and (2) are same. Recall the following theorem from the chapter context-free grammar. Theorem: Let G = (V,T,R,S) be a CFG, and suppose there is a parse tree with root labeled by variable A and with yield w(∈ T∗). Then there is a leftmost derivation A ∗ ⇒lm w in grammar G. 1.1 From Grammar to Pushdown Automata Given a CFG G, we construct a PDA that simulates the leftmost derivations of G. Any left-sentential form that is not a terminal string can be written as xAα, where A is the leftmost variable, x is whatever terminals appear to its left, and α is the string of terminals and variables that appear to the right of A. We call Aα the tail of this left-sentential form. If a left-sentential form consists of terminals only, then its tail is ϵ. The idea behind the construction of a PDA from a grammar is to have the PDA simulate the sequence of left-sentential forms that the grammar uses to generate a given terminal string w. The tail of each sentential form xAα appears on the stack

with A at the top. At that time, x will be represented by our having consumed x from the input, leaving whatever of w follows its prefix x. That is, if w = xy, then y will remain on the input

Suppose the PDA is in an ID (q,y,Aα), representing left-sentential form xAα. It guesses the production to use to expand A, say A → β. The move of the PDA is to replace A on the top of the stack by β, entering ID (q,y,βα). Note that there is only one state, q, for this PDA.

Now, (q,y,βα) may not be a representation of the next left-sentential form, because β may have a prefix of terminals. In fact, β may have no variables at all, and α may have a prefix of terminals. Whatever terminals appear at the beginning of βα need to be removed, to expose the next variable at the top of the stack. These terminals are compared against the next input symbols, to make sure our guesses at the leftmost derivation of input string w are correct; if not, this branch of the PDA dies

If we succeed in this way to guess a leftmost derivation of w, then we shall eventually reach the left-sentential form w. At that point, all the symbols on the stack have either been expanded (if they are variables) or matched against the input (if they are terminals). The stack is empty, and we accept by empty stack

The above informal construction can be maid precise as follows. Let G = (V,T,R,S) be a CFG. Construct the PDA P that accepts L(G) by empty stack as follows:

P =({q},T,V ∪T,δ,q,S)

where transition function δ is defined by

1. For each variable A, δ(q,ϵ,A) = {(q,β)|A → β is a production of P}.

2. For each terminal a, δ(q,a,a) = {(q,ϵ)}

Theorem: If PDA P is constructed from CFG G by the construction above, then N(P) =L(G).

See Hopcroft, Motwani and Ullman book for proof of this theorem

From PDA's to Grammar The construction of an equivalent grammar uses variables each of which represent an event consisting of:

1. The net popping of some symbol X from the stack.

2. A change in state from some p at the beginning to q when X has finally been replaced by ϵ on the stack

(V,Σ,R,S) such that L(G) = N(P), where the set of variables V consists of :

1. The special symbol S, which is the start symbol of G and

2. All symbols of the form [pXy], where p,q ∈ Q and x ∈ Γ

The rules R of G are as follows

a) For all states p, G has the rules S → [q0z0p] (since (q0,w,z0)

b) Let δ(q,a,X) contains the pair (r,Y1Y2 ...Yk), where ∗ ⊢ (p,ϵ,ϵ))

1. a is either a symbol in Σ or a = ϵ. 2.

k be any number, including 0, in which case the pair is (r,ϵ)

Then for all lists of states r1,r2,...,rk, G has the rules

[qXrk] → a[rY1r1][r1Y2r2]...[rk−1Ykrk]

This rules says that one way to pop X and go from state q to state rk is to read a (which may be ϵ), then use some input to pop Y1 off the stack which going from state r to state r1, then read some more input that pops Y2 off the stack and goes from state r1 to state r2, and so on.

Example: Consider the PDA PN = ({q},{0,1},{Z,A,B},δN,q,Z) in Figure 2. The corresponding context-free grammar G = (V,{0,1},R,S) is given by:

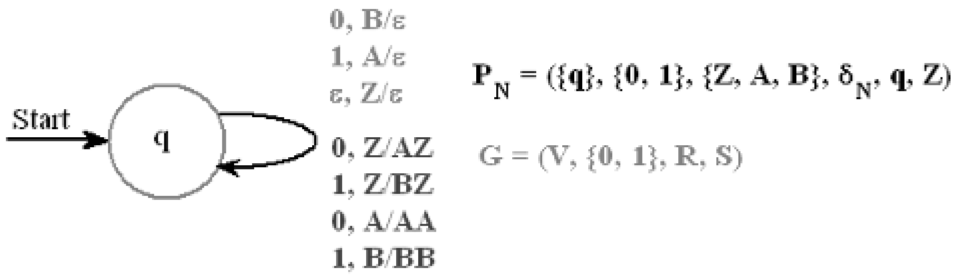


Figure 2: Example of PDA

1. S →[qZq]
2. [qZq] → 0[[qAq][qZq] (since δN(q,0,Z) contains (q,AZ))
3. [qZq] → 1[[qBq][qZq] (since δN(q,1,Z) contains (q,BZ))
4. [qAq] → 0[[qAq][qAq] (since δN(q,0,A) contains (q,AA))
5. [qBq] → 1[[qBq][qBq] (since δN(q,1,B) contains (q,BB))
6. [qAq] → 1 (since δN(q,1,A) contains (q,ϵ))
7. [qBq] → 0 (since δN(q,0,B) contains (q,ϵ))
8. [qZq] → ϵ (since δN(q,ϵ,Z) contains (q,ϵ))

Vyasapuri, Bandlaguda, Post:Keshavgiri
Hyderabad-500005,Telangana, India
Tel:040-29880079, 86,8978380692, 9642703342
9652216001, 9550544411, Website:www.mist.ac.in
Email:principal@mist.ac.in
principal.mahaveer@gmail.com
Counseling code:MHVR, University Code:E3

MAHAVEER
INSTITUTE OF SCIENCE & TECHNOLOGY
(AN UGC AUTONOMOUS INSTITUTION)
Approved by AICTE, Affiliated to JNTU,Hyderabad
Accredited by NAAC with 'A' Grade
Recognized Under 2(f) of UGC Act 1956,ISO 9001:2015 Certified

ESTD : 2001