PGCA/Sc IT (S3) 09
Exam. Code: M3IDM

Introduction to Discrete Mathematics and Formal Languages and Automata

SEMESTER-III

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

BLOCK- 2

KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Experts

Prof. Anjana Kakati Mahanta, Gauhati University

Prof. (Retd.) Pranhari Talukdar, Gauhati University

Prof. Shikhar Kumar Sarma, Gauhati University

Dr. Jyotiprokash Goswami, Assam Engineering College

Programme Co-ordinator

Dr. Ridip Dev Choudhury, KKHSOU

Course Co-ordinators :

Dr. Tapashi Kashyap Das, KKHSOU

Dr. Sruti Sruba Bharali, KKHSOU

SLM Preparation Team

UNITS	CONTRIBUTORS
9	**Ms. Daisy Kalita,** USTM
10, 11	**Dr. Naba Jyoti Sarmah,** Nalbari Commerce College
12, 15	**Dr. Sufal Das,** NEHU
13	**Ms. Jonalee Barman Kakati,** NERIM
14	**Dr. Tapashi Kashyap Das,** KKHSOU

Editor

Content : **Dr. Gautam Chakraborty,** NERIM

Structure, Format & Graphics : Dr. Nabankur Pathak, KKHSOU

Jan, 2022 **ISBN: .....**

This Self Learning Material (SLM) of the Krishna Kanta Handiqui State Open University is made available under a Creative Commons Attribution-Non Commercial-Share Alike 4.0 License (international): http://creativecommons.org/licenses/by-nc-sa/4.0/

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University

Headquarters: Patgaon, Rani Gate, Guwahati-781017

City Centre: NH 37, Khanapara, Guwahati-781022; Web:www.kkhsou.in

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

Introduction to Discrete Mathematics and Formal Languages and Automata

Block-2

DETAILEDSYLLABUS

		Pages
UNIT 9 :	**Mathematical Preliminaries [6 marks]** Sets; Relations And Functions: Relations And Its Properties, Functions; Closure Of Relations; Symbols And Alphabets; Strings And Their Properties; Languages; Principles Of Induction	**7-26**
UNIT 10:	**The Theory of Automata [8 Marks]** Definition of Automaton; Finite Automata: Finite Automata Related Terminologies; Transition System- Transition Table, Transition function, String Acceptance by Finite Automata	**27-43**
UNIT 11:	**Types of Finite Automata [8 Marks]** Deterministic Finite Automata (DFA); Non-Deterministic Finite Automata (NFA); Equivalence of DFA and NFA ; Minimization of Finite Automata; Application of Finite Automata; Finite Automata with Å - Transition	**44-82**
UNIT 12:	**Formal Languages [7 Marks]** Basic Definition; Definition of a Grammar; Derivation and language generated by a grammar; Chomsky Classification of Languages	**83-90**
UNIT 13:	**Regular Expressions and Regular Grammar [8 Marks]** Regular expressions; Regular set and Regular grammar; Closure properties of Regular languages; Relationship between Finite	**91-110**

automata and Regular Expressions; Pigeon hole principle; Applications of Regular expressions

UNIT 14: Context-Free Grammars and Languages [8 Marks] **111-124**

Context-Free Grammars- Context-Free Languages; Derivation Tree-Leftmost Derivation and Rightmost Derivation, Ambiguity in Grammars and Languages; Applications of Context-Free Grammar

UNIT 15: Properties of Context-Free Languages [5 Marks] **125-135**

Normal forms for CFGs; The pumping lemma for CFGs; Closure properties of CFL

BLOCK INTRODUCTION

This is the second block of the course ***Introduction to Discrete Mathematics and Formal Languages and Automata.*** This block is on ***Formal languages and Automata***. Formal languages are used to design languages to write instructions for machines and automata theory is to design computing machines. The units of this blocks are provided with various examples that illustrate the basic concepts.

Unit 9 : deals with the mathematical prelimineries which includes set theory, strings, properties and operations on strings and principles of induction, which will be used for proving many theorems throughout the block.

Unit 10 : will begin with the study of automaton. Transition system and properties of transition functions are discussed in this unit. We also define the acceptability of strings by finite automata epsilon-transition at the end.

Unit 11 : mainly discusses the types of finite automata and its applications. DFA and NFA are discussed in this unit.

Unit 12 : introduces the concept of grammars and formal languages. This units gives the definitions of grammar and language generated by a grammar. Chomsky classification of languages are also discussed in this unit.

Unit 13 : is about regular expressions and regular grammar. Closure properties of regular languages, relationship between the finite automata and regular expressions are discussed in this unit.

Unit 14 : is on context-free grammars and languages. We define derivation trees, leftmost and rigntmost derivation with various examples and diagrams. We also learn about ambiguity of grammar and languages.

Unit 15 : is the last unit of this block. This unit discusses various properties of context-free languages including normal forms, pumping lemma for context CFGs and closure properties of context-free languages.

UNIT 9: MATHEMATICAL PRELIMINARIES

UNIT STRUCTURE

9.1 Learning Objects

9.2 Introduction

9.3 Sets

9.4 Relations and Functions

9.4.1 Relations and Its Properties

9.4.2 Functions

9.5 Closure of Relations

9.6 Symbols and Alphabets

9.7 Strings and Their Properties

9.8 Languages

9.9 Principles of Induction

9.10 Let Us Sum Up

9.11 Further Reading

9.12 Answer To The Check Your Progress

9.13 Model Questions

9.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- Learn how to define the set, relation and function and symbols to represent a real world problem and solve it.
- Analyze issues and apply more complex mathematical problem solving skills to plan courses of actions in highlevel decision-making situations.
- Explain preliminary concept of discrete math concepts such as formal languages, finite-state machines.

9.2 INTRODUCTION

A set is a collection of unordered and well-defined elements, which is not affected if we change the order of data. Set can be represented in

two different ways: set builder form and tabular form. We can performdifferent operations such as union, intersection, difference etc. on sets.Relation defines the relationship between sets of values of ordered pairs. Relation is helpful to find the relationship between input and output of a function. A function or mapping is a relationship from elements of one set X to elements of another set Y.To represent the set of elements, we can use some symbols. It may be either alphabet or digit. Finite sequence of alphabet represents a string and a set of string generate a language.

9.3 SETS

A set is a collection of unordered, well-defined elements. Using curly braces, a set can be written explicitly by listing its elements. It does not make any changes in the set if we change the order of the elements or any element of a set is repeated.

Some Example of Sets

- A set of all positive integers
- A set of all the planets in the solar system
- A set of all the states in India
- A set of all the lowercase letters of the alphabet

Representation of a Set

Sets can be represented in two ways –

- Roster or Tabular Form
- Set Builder Notation

Roster or Tabular Form

The set is represented by listing all its elements. The elements are enclosed within braces and separated by commas.

Example 1 – Set of vowels in English alphabet, A={a,e,i,o,u}

Example 2 – Set of multiples of 3 less than 10, B={3,6,9}

Set Builder Notation

The set is defined by specifying a property that elements of the set have in common. The set is described in set builder form is as A={x:p(x)}

Example 1 – The set {a,e,i,o,u} is written as –

A={x:x is a vowel in English alphabet}

Example 2 – The set {3,6,9} is written as –

B={x:1d$\leq$x<10 and (x%3)=0}

Some Important Sets

N – the set of all natural numbers = {1,2,3,4,.....}

Z – the set of all integers = {.....–3,–2,–1,0,1,2,3,.....}

Z^+ – the set of all positive integers

Q – the set of all rational numbers

R – the set of all real numbers

W – the set of all whole numbers

Cardinality of a Set

Cardinality of a set S, denoted by $|S|$, is the number of elements of the set.

It is also referred to as cardinal number. If a set has an infinite number of elements, its cardinality is ∞.

Example – $|\{1,4,3,5\}|=4$,

$|\{1,2,3,4,5,...\}|=\infty$

Consider there are two sets X and Y,

- If these two sets X and Y have same cardinality it is denoted by |X|=|Y|.
- If X's cardinality is less than or equal to set Y's cardinality then it is denoted as |X|d$\leq$|Y|
- If X's cardinality is less than or equal to set Y's cardinality then it is denoted as |X|<|Y|
- If |X|$\leq$|Y| and |X|$\geq$|Y| then |X|=|Y|, The sets X and Y are commonly referred as equivalent sets.

Types of Sets

Sets can be classified into many types. Some of which are finite, infinite, subset, universal, proper, singleton set, etc.

Finite Set

A set which contains a definite number of elements is called a finite set.

Example – S={x|x∈N and 70>x>50}

Infinite Set

A set which contains infinite number of elements is called an infinite set.

Example – S={x|x∈N and x>10}

Subset

If every element of X is an element of set Y, then set X is a subset of set Y (X⊆Y).

Example – Let, X={1,2,3,4,5,6} and Y={1,2}. As all the elements of Y present as element of X, so, Y is subset of set X. Hence, we can write Y⊆X.

Proper Subset

If every element of X is an element of set Y and |X|<|Y|, Then Set X is a proper subset of set Y (X⊂Y)

Example – Let, X={1,2,3,4,5,6} and Y={1,2}. HereY⊂X since all elements in Y are contained in X too and X has at least one element is more than set Y.

Universal Set

Any set which is a superset of all the sets under consideration and usually it is denoted as S or U.

Example – We may define U as the set of all animals on earth. In this case, set of all mammals is a subset of U, set of all fishes is a subset of U, set of all insects is a subset of U, and so on.

Empty Set or Null Set

An empty set contains no elements. It is denoted by ϕ.

Example – S={x|x∈N and 6<x<7}=ϕ

Singleton Set or Unit Set

Singleton set or unit set contains only one element. A singleton set is denoted by {s}.

Example – S={x|x∈N, 48<x<50} = {49}

Equal Set

If two sets contain the same elements, they are said to be equal.

Example – If A={1,2,3}and B={3,1,2}, they are equal as every element of set A is an element of set B and every element of set B is an element of set A.

Equivalent Set

If the cardinalities of two sets are same, they are called equivalent sets.

Example – If A={1,2,6}and B={11,12,24}, they are equivalent as cardinality of A is equal to the cardinality of B. i.e. |A|=|B|=3

Disjoint Set

If two sets A and B do not have any common element, they are called disjoint sets Therefore, disjoint sets have the following properties "

- $n(A) \cup B)=\phi$
- $n(A \cap B)=n(A)+n(B)$

Example – Let, A={1,2,6} and B={7,9,14}, there is not any single common element in these two sets, hence these sets are disjoint set.

Power Set

Power set is defined as collection of all subsets of a set. If A is the set then P(A) is denoted as its power set.

The number of elements contained by any power set can be calculated by $n[P(A)] = 2^n$ where n is the number of elements in set A.

For **example**, If A = {0, 2} then, P(A) = {ϕ, {0}, {2}, {0, 2}}

Number of elements in $P(A) = 2^2 = 4$

Set Operations

Basic Set Operations such as Union, Intersection, Difference, Cartesian Product can be applied on sets.

Union

The union of sets A and B ($A \in \cup B$) is the set of elements which are in A or in B or in both A and B. Hence, $A \cup B=\{x|x \in A \text{ OR } x \in B\}$

Example – If A={10,11,13} and B = {13,14,15,19} then

$A \cup B=\{10,11,12,13,14,15,19\}$

Intersection

The intersection of sets A and B ($A \cap B$) contains set of elements which are in both A and B. Hence, $A \cap B=\{x|x \in A \text{ AND } x \in B\}$

Example:

If $A=\{11,12,13,15\}$ and $B=\{13,14,16,17\}$ then $A \cap B=\{13\}$

Difference

The set difference of sets A and B (A–B) contains the elements which are only in A but not in B. Hence, $A–B=\{x|x \in A \text{ AND } x \notin B\}$.

Example -

If $A=\{10,11,12,13\}$ and $B=\{13,14,15\}$

then $(A–B)=\{10,11,12\}$

and $(B–A)=\{14,15\}$

In every cases of difference $(A–B) \neq (B–A)$

Cartesian product

The Cartesian product of n number of sets A_1, A_2, …….A_n denoted as $A_1xA_2x…….A_n$ can be defined as all possible ordered pairs (x_1, x_2, ……, x_n) where $x_1 \in A_1$, $x_2 \in A_2$……$x_n \in A_n$.

Example-

If we define two sets $A=\{1,3\}$ and $B=\{c,d\}$

Then the Cartesian product of A and B represent as

$AxB=\{(1,c),(1,d),(3,c),(3,d)\}$

9.4 RELATIONS AND FUNCTIONS

9.4.1 Relations

Relation defines the relationship between sets of values of ordered pairs. The set of elements in the first set are called domain which is related to the Range, another set of elements.

Relation is useful to find the relationship between input and output of a function.

A relation R, from a non-empty set P to another non-empty set Q, is a subset of P X Q. it is defined as xRy or R(x,y)

For example, Let P = {a, b} and Q = {3, 4} and

Let R = {(a, 3), (a, 4), (b, 3), (b, 4)}

Here R is a subset of A x B. Therefore, R is a relation from P to Q.

Types of Relations

Empty Relation

A relation is called empty when there is no relation between any elements of a set. It is also called void relation. For example, if set A = {1, 2, 3} then, one of the empty relations can be R = {x, y} where, |x – y| = 8. For empty relation,

$$R = \varphi \subset A \times A$$

Universal Relation

In a relation if every element of a set is related to each other that relation is called as universal (or full relation) relation. Consider set A = {a, b, c}. Now one of the universal relations will be R = {x, y} where, |x – y| e” 0. For universal relation,

$$R = A \times A$$

Identity Relation

In an identity relation, every element of a set is related to itself only. For example, in a set A = {a, b, c}, the identity relation will be I = {a, a}, {b, b}, {c, c}. For identity relation,

$$I = \{(a, a), a \in A\}$$

Inverse Relation

In a relation where every pair of a set is an inverse pair of another set, then this relation is called as Inverse relation. For example, if set A = {(a, b), (c, d)}, then inverse relation will be R^{-1} = {(b, a), (d, c)}. So, for an inverse relation,

$$R^{-1} = \{(b, a): (a, b) \in R\}$$

Reflexive Relation

In a reflexive relation, every element maps to itself. For example, consider a set A = {1, 2,}. Now an example of reflexive relation will be R = {(1, 1), (2, 2), (1, 2), (2, 1)}. The reflexive relation is given by

$$(a, a) \in R$$

Symmetric Relation

In a symmetric relation, if a=b is true then b=a is also true.

An example of symmetric relation will be R = {(1, 2), (2, 1)} for a set A = {1, 2}. So, for a symmetric relation,

$$aRb \Rightarrow bRa, \ \forall\ a, b \in A$$

Transitive Relation

For transitive relation, if $(x, y) \in R$, $(y, z) \in R$, then $(x, z) \in R$. For a transitive relation,

$$aRb \text{ and } bRc \Rightarrow aRc \ \forall\ a, b, c \in A$$

Equivalence Relation

If a relation is reflexive, symmetric and transitive at the same time, it is known as an equivalence relation.

Domain, Co-domain and Range of a Relation

Let A and B are two sets. If R be a relation from A to B then

- Domain of relation R (Dom(R)) is the set of all those elements $a \in A$ such that $(a, b) \in R$ for some $b \in B$.
- If R be a relation from A to B, then B is the co-domain of R.
- Range of relation R is the set of all those elements $b \in B$ such that $(a, b) \in R$ for some $a \in A$.

In short: Domain = Dom(R) = {a : (a, b) ∈ R} and Range (R) = {b : (a, b) ∈ R}

Range is always a subset of co-domain.

9.4.2 Functions

A function is used to represent the dependence of one quantity on the other. It defines the concept of mapping. In simple words, a function is a relation which derives one **output** for each **input.**

A function from set P to set Q is a rule that assigns to each element of set P, one and only one element of set Q.

Mathematically: If f: A -> B where y = f(x), $x \in P$ and $y \in Q$. Here y is the image of x under f.

Domain, Co-domain and Range of a Function

A function f from a set P to a set Q, represented as f: P-> Q,

It is a mapping of elements of P to elements of Yin a way that each element of P is related to some chosen element of Q. Here P is domain and Q is co-domain of the function. That is every element of *P must be mapped to some element of Q and only one element of Q*.

Domain:

For any function, y = f(x), the set of all the values of x is called the domain of the function, where we get set of all possible input values.

Co-domain:

The Co-domain is the set of values that could possibly come out. The Co-domain is actually part of the definition of the function.

Range:

Range of y = f(x) is a collection of all outputs f(x) corresponding to each real number in the domain. Range is the set of all the values of y. It describes the set of possible output values.

For example, consider the following relation.

{(2, 3), (4, 5), (6, 7)}

Here Domain = {2, 4, 6}

Range = { 3, 5, 7}

Types of Functions

A function can be classified as one to one or many to one.

Injective / One-to-one function

A function $f:A \rightarrow B$ is injective or one-to-one function if for every $b \in B$, there exists at most one $a \in A$ such that f(s)=t

This means a function **f** is injective if a1≠a2 implies f(a1)≠f(a2).

Example

- f:N→N,f(x)=5x is injective.
- f:N→N,f(x)=x^2 is injective.
- f:R→R,f(x)=x^2 is not injective as $(–x)^2=x^2$

Surjective / Onto function

A function f:A→B is surjective (onto) if the image of f equals its range. For every b”B, there exists some a”A such that f(a)=b.

Example

- f:N→N,f(x)=x+2 is surjective.
- f:R→R,f(x)=x^2 is not surjective since we cannot find a real number whose square is negative.

Bijective / One-to-one Correspondent

A function f:A→B is bijective or one-to-one correspondent if and only if **f** is both injective and surjective.

Problem

Prove that a function f:R’!R defined by f(x)=2x–3 is a bijective function.

Solution:

We have to prove this function is both injective and surjective.

If f(x1)=f(x2), then 2x1–3=2x2–3 and it implies that x1=x2

Hence, f is **injective**.

Here, 2x–3=y

So, x=(y+5)/3 which belongs to R and f(x)=y.

Hence, f is **surjective**.

Since **f** is both **surjective** and **injective**, we can say **f** is **bijective**.

Inverse of a Function

The inverse of a one-to-one corresponding function f:A→B, is the function g:B→A, holding the following property

$$f(x)=y \Leftrightarrow g(y)=x$$

The function f is called **invertible**, if its inverse function g exists.

Example

- A function f:Z→Z, f(x)=x+5, is invertible since it has the inverse function g:Z→Z, g(x)=x–5.
- A function f:Z→Z,f(x)=x^2 is not invertible since this is not one-to-one as $(–x)^2=x^2$.

Composition of Functions

Two functions f:A→B and g:B→C can be composed to give a composition gof. This is a function from A to C defined by (gof)(x)=g(f(x)).

Some notes about Composition of functions

- If f and g are one-to-one then the function (gof) is also one-to-one.
- If f and g are onto then the function (gof)is also onto.
- Composition always holds associative property but does not hold commutative property.

9.5 CLOSURE OF RELATIONS

Consider a relation R on set A. R may or may not have a property P, such as reflexivity, symmetry, or transitivity.

If there is a relation S with property P containing R such that S is the subset

of every relation with property P containing R, then S is called the closure of R with respect to P.

We can obtain closures of relations with respect to property P in the following ways:

1. **Reflexive Closure –** $\Delta = \{(\square, \square) | \square \in \square\}$ is the diagonal relation on set A. The reflexive closure of relation R on set A is $\square \cup \Delta$.
2. **Symmetric Closure –** Let R be a relation on set A, and let R^{-1} be the inverse of R. The symmetric closure of relation R on set A is $\square \cup \square^{-1}$.

3. **Transitive Closure –** Let R be a relation on set A. The **Connectivity Relation** is defined as $R^* = \bigcup_{n=1}^{\infty} R^n$. The transitive closure of R is R*.



CHECK YOUR PROGRESS

Q 1: What is the definition of set?

Q 2: Define finite and infinite set.

Q 3: What are the basic operations used on set?

Q 4: Define relation and its three types.

Q 5: Define one-to-one and invertible function.

Q 6: Define domain and co-domain of function.

9.6 SYMBOLS AND ALPHABETS

To represent the set of elements we can use some symbols e.g alphabet or digits. Commonly we use lower case letter at the beginning of the alphabet (or digits) to denote symbols.

An alphabet is a set of finite numbers of symbols and it is non empty, which is used to represent the input of a machine. Alphabets are typically represented by letters, characters, digits, signs, punctuation, etc. Conventionally, 'Σ' symbol is used for an alphabet. Common alphabets include:

- Σ = {0, 1}: The binary alphabets.
- Σ = {a, b, c, ……, z}: The set of all lower-case letters.
- Σ = {The set of all ASCII characters or the set of all printable ASCII characters}.

9.7 STRINGS AND THEIR PROPERTIES

A string is a finite sequence of symbols which are chosen from alphabet Σ. For example,'cabcad' is a string from the alphabet set Σ = {a, b, c, d} and the binary set {01011} is a string from the binary alphabet Σ = {0,1}.

One sequence (a_1, a_2, . . . , a_n), in the present context, which we prefer to write as $a_1a_2 \ldots a_n$, i.e. by justaposing the symbols in that order. Thisstring is also known as a word or a sentence. Normally, we use lower case letters towards the end of alphabet, namely w, x, y, z etc., to denote strings.

Empty String

The string with zero occurrences of symbols is called as Empty String. It is denoted by 'å' and may be chosen from any alphabet.

Length of a string

It is defined by the number of symbols in a string. It is useful to classify thestrings by their length. For example, 01101 have length 5.

The standard notation for length of string x is denoted by |x|. The length of the string w = abcdef is |w| = 6.

Powers of an Alphabet

If Σ is an alphabet, we can represent the set of all strings of a certain length from that alphabet by using exponential notation. The power of an alphabet is denoted by Σ^k and is the set of strings of length k, each of whose symbols is in Σ.

For example,

- Σ={0,1}
- Σ^1= {0,1} (2^1=2)
- Σ^2= {00,01,10,11} (2^2=4)
- Σ^3= {000,001,010,011,100,101,110,111} (2^3= 8)

Note: The $\Sigma$is defined asan alphabet with symbols 0 and 1. And the $\Sigma^1$is the set of strings with numbers as strings 0 and 1 of length 1.

The set of all strings over an alphabet Σ is denoted by Σ*.

For instance, {0,1}*={ε, 0, 1,00, 01, 10,11, 000, ……..,111}.

It can be put another way,

$$\Sigma^* = \Sigma^0 U \Sigma^1 U \Sigma^2 U \ldots\ldots\ldots$$

The power of alphabet is classified as two types "

- Kleene closure (Σ^*)
- Kleene plus (Σ^+)

Kleene Star Closure: Σ^*

Let Σ ={a,b}

$\Sigma^*= \Sigma^0 U\ \Sigma^1 U\ \Sigma^2 U\ \Sigma^3$……

={ε} U {a,b} U {aa,ab,ba,bb}.....

Set of all strings including epsilon is called Kleene Star closure.

Kleene Plus:Σ^+

Let Σ ={a,b}

$\Sigma^+= \Sigma^1 U\ \Sigma^2 U\ \Sigma^3$……

={a,b} U {aa,ab,ba,bb}.....

Set of all strings excluding epsilon is called kleene plus

$\Sigma^+= \Sigma^* - \{ \varepsilon\}$

Or

$\Sigma^+= \Sigma^1 U\ \Sigma^2 U\ \Sigma^3$

Concatenation of String

Let x and y be two strings. Then xy denotes the concatenation of x and y; the string formed by making a copy of content of x and followed by the copy of content of y.

For example,

let x=1010 and y=0011 then xy=10100011.

For any string w, the equations εw=wε=w where ε is the identity of concatenation.

9.8 LANGUAGES

A set of strings chosen from special language Σ^* where Σ is a particular alphabet, is called language. For example,

if Σ = {a, b, c} then Σ^* = {ε, a, b, c, aa, ab, ac, ba, . . . , aaaaaabbbaababa, . . .} .

Namely, Σ^* is the “full” language made of characters of Σ.

If Σ is an alphabet, and $L \subseteq \Sigma^*$ then L is a (formal) language over Σ. A language over Σ need not to include strings with all the symbols of Σ. After establishing L over Σ, we come to know that it is a language over any alphabet that is superset of Σ. For example, Programming language C where

the legal programs are a subset of the possible strings that can be formed from the alphabet of the language.

In the context of Automata theory, many languages can be derived such as:

1. The language of all strings consisting of n 0's followed by n 1's, for some $n \geq 0$:
   { ε, 01, 0011, 000111,……}
2. The set of binary numbers whose value is a prime:
   {10, 11, 101, 111, 1011,…….}
3. The set of strings of 0's and 1's with an equal number of each:
   { ε, 01, 10, 0011, 0101, 1001,…….}
4. Φ, the empty language, is a language over any alphabet.
5. {ε}, is a language consisting only empty string, is also a language over any alphabet.
   Where {ε}, $\neq \Phi$.
6. Σ^* is a language for any alphabet Σ.

9.9 PRINCIPLES OF INDUCTION

The Principle of Induction: Let a be an integer, and let P(n) be a statement about n for each integer $n \geq a$. The principle of induction is a way of proving that P(n) is true for all integers n e" a. It works in two steps:

(a) [Base case:] Prove that P(a) is true.

(b) [Inductive step:] Assume that P(k) is true for some integer $k \geq a$, and use this to prove that P(k + 1) is true.

Then we may conclude that P(n) is true for all integers n e" a. This principle is very useful in problem solving, especially when we observe a pattern and want to prove it.

Induction of regular languages

Induction of regular languages describes the task related to learninga formal description of a regular language from a given set of strings. After investigation on different subclasses, it is found that every regular language is not possible to learn in this way.

A regular language is represented as a set of strings that can be formed by one of the mathematical formalisms called Finite automation, regular expression or regular grammar, all of these with the same expressive power. Given a set Σ of symbols (a.k.a. alphabet), a regular expression can be any of

- Φ(denoting the empty set of strings),
- ε (denoting the singleton set containing just the empty string),
- *a* (where *a* is any character in Σ; denoting the singleton set),
- *r*+*s* (where *r* and *s* are, in turn, simpler regular expressions; denoting their set's union)
- ***r.s*** (denoting the set of all possible concatenations of strings from *r* 's and *s* 's set),
- r^+ (denoting the set of *n*-fold repetitions of strings from *r* 's set, for any $n \geq 1$), or
- r^* (similarly denoting the set of *n*-fold repetitions, but also including the empty string).

The task of regular language induction is to come up with a regular expression that denotes a set containing all of them forms a lattice, with the trivial over-generalized and the trivial under-generalized automaton as top and bottom element, respectively. From the undergeneralized automaton by an appropriate equivalence relation we can obtain each member of this lattice.

Kudo and Shimbo use the representation by automaton factorizations to give a unique framework for the following approaches

- k-reversible languages and the "tail clustering" follow-up approach,
- Successor automata and the predecessor-successor method,
- Pumping-based approaches

k-reversible languages

It is a deterministic automata in which each state can be realized from at most one state by following a transition chain of length *k*. If Σ, *Q*, and *δ* denote the input alphabet, the state set, and the transition function of an automaton *A*, respectively, then *A* is *k*-reversible if : $\forall\, a_0,\ldots,a_k \in \Sigma$

$\forall s_1, s_2 \in Q: \delta^*(s_1,a_0...a_k) = \delta^*(s_2,a_0...a_k) \Rightarrow s_1 = s_2$, where δ^* means the homomorphic extension of δ to arbitrary words.

Successor automata

It consists of one state for each distinct character and a transition between each two adjacent characters' states. For example, the singleton input set { *aaccaacc* } leads to an automaton corresponding to the regular expression $(a^+c^+)^*$.

Pumping-based approaches

For any regular language L, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists u, v, w $\in \Sigma^*$, such that x = uvw, and

(1) $|uv| \leq n$

(2) $|v| \geq 1$

(3) for all $i \geq 0$: $uv^iw \in L$

As a proof for irregularity of a language, Pumping Lemma is used. A regular language always satisfies pumping lemma. The language L is not regular if there exists at least one string made from pumping which is not in L.



CHECK YOUR PROGRESS

Q 7: What is symbol and alphabet?

Q 8: Define string.

Q 9: What is the meaning of length of a string?

Q 10: What is the difference between Kleene star closure and Kleene plus .

Q 11: Write about the two abstract examples of language



9.10 LET US SUM UP

- A set is a collection of unordered, well-defined elements.
- We can change the order of the elements.

- Set can be represented in either roster form or set builder notation.
- Cardinality of set defines numbers of elements in the set.
- Sets are of many types such as finite, infinite, unit, null set etc.
- Relation defines the relationship between input and output of a function.
- Domain, co-domain and range are related to every relation as well as function.
- Range is a subset of co-domain.
- *Function describes the mapping or relation from an element of a set to some element of another set.*
- Symbols are used to represent set elements.
- Alphabet is any finite set of symbols.
- A string is a finite-length sequence of symbols.
- A language is a set of strings, which are chosen from symbols of alphabet.
- Kleene closure and Kleene plus are two approaches of powers of a n alphabet.
- K-reversible languages, Successor automata and Pumping-based approaches are used to generate unique framework.



9.11 FURTHER READING

1. Kenneth H Rosen; Discrete Mathematics and Its Applications; McGraw Hill
2. Harry Lewis & Rachel Zax; Essential Discrete Mathematics for Computer Science; Princeton University Press
3. John E Hopcroft, Rajeev Motwani, Jeffrey D Ullman; Introduction to Automata Theory, Languages and Computation; Pearson Education



9.12 ANSWERS TO THE CHECK YOUR PROGRESS

Ans. to Q No 1: A set is a collection of unordered, well defined elements.

Ans. to Q No 2: Finite Set: A set which contains a definite number of elements is called a finite set.

Infinite Set: A set which contains infinite number of elements is called an infinite set.

Ans. to Q No 3: Union, Intersection, Difference and Cartesian Product

Ans. to Q No 4: Relation: Relation defines the relationship between sets of values of ordered pairs. It defines relationship between input and output of a function.

Three types of relations are:

i) Empty relation: An empty relation is defined as there is no relation between any elements of a set.

ii) Universal relation: every element of a set is related to each other.

iii) Identity relation: every element of a set is related to itself only.

Ans. to Q No 5: One-to-one function: A function $f:A \rightarrow B$ is injective or one-to-one function if for every $b \in B$, there exists at most one $a \in A$ such that f(s)=t.

Invertible Function: The function f is called invertible, if its inverse function g exists.

Ans. to Q No 6: Domain: For any function, y = f(x), the set of all the values of x is called the domain of the function, where we get set of all possible input values.

Co-domain: The Co-domain is the set of values that could possibly come out. The Co-domain is actually part of the definition of the function.

Ans. to Q No 7: Symbols are used to represent set elements and Alphabet is any finite set of symbols.

Ans. to Q No 8: A string is a finite sequence of symbols taken from alphabet Σ.

Ans. to Q No 9: Number of symbols present in string define the length of string.

Ans. to Q No 10: Kleene Plus is same as Kleene Star Closure except that it does not generate Λ (null string), automatically.

Ans. to Q No 11: i) Φ, the empty language, is a language over any alphabet.
ii) Σ^* is a language for any alphabet ".



9.13 MODEL QUESTIONS

Q 1: Let R be a relation on set {1,2,3,4} with R={ (1,1), (1,4),(2,3),(3,1), (3,4)} . Find the reflexive, symmetric, and transitive closure of R.

Q 2: Define set and its classification.

Q 3: What are the operations we can use on sets?

Q 4: Write about the different types of relation with example.

Q 5: Define functions and its categories.

Q 6: Define domain, co-domain and range of a function with example.

Q 7: How can we check a relation is an equivalence relation?

Q 8: Define alphabet and strings.

Q 9: How can we define languages? Give some examples of languages.

Q 10: What Is the Difference Between the Strings And The Words Of A Language?

Q 11: What Is Null String (Λ)?

Q 12: Define Regular Expression? What are the approaches we can use in regular expression?

Q 13: Define Induction principle.

*** ***** ***

UNIT 10 : THE THEORY OF AUTOMATA

UNIT STRUCTURE

10.1 Learning Objectives

10.2 Introduction

10.3 Definition of Automaton

10.4 Finite Automata

10.4.1 Finite Automata Related Terminologies

10.5 Transition System

10.5.1 Transition Table

10.5.2 Transition function

10.5.3 String Acceptance by Finite Automata

10.6 Let us Sum Up

10.7 Further Reading

10.8 Answers to Check Your Progress

10.9 Probable Questions

10.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- get in depth knowledge of automata
- understand finite automation system
- know about transition graph and transition function
- know the use of ε-Transition in finite automata

10.2 INTRODUCTION

In this chapter, we look at the basic ideas of automata theory. We have discussed the main ideas from mathematics that are required in automata theory.We will also take a look at the central concepts of languages, grammar, and automata. We will look at the concepts like transition graph, transition table and transition function so that we can have a clear idea about how finite automata work.

10.3 DEFINITION OF AUTOMATON

Automata theory is the study of abstract machines and automaton, as well as the problems that can be solved using these machines. Automata is the plural form of automaton**;** it is a self-operating machine that follows a predetermined sequence of operations or response of predetermined instructions that is used to check a computational problem whether it can be solved by that machine or not.

The word "automaton" is derived from the Greek word αΡτόματον. Automaton means "a moving mechanical device made in imitation of a human being". It is more often used to describe non-electronic moving machines, especially that have been made to resemble human or animal behaviors.

The theory of automation was initially proposed as a simple model for the behavior of neurons. The concept of a finite automaton appears to have arisen in the paper "A logical calculus of the ideas immanent in nervous activity", in 1943 by Warren McCullock and Walter Pitts. In 1951 Kleene introduced regular expressions to describe the behavior of finite automata. He also proved the important theorem saying that regular expressions exactly capture the behaviors of finite automata. In 1959, Dana Scott and Michael Rabin introduced non-deterministic automata and showed the surprising theorem that they are equivalent to deterministic automata.

States, Transitions and Finite-State Transition System:

Informally, a state of a system is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on. Transitions are changes of states that can occur spontaneously or in response to inputs to the states. Though transitions usually take time, we assume that state transitions are instantaneous. A system containing only a finite number of states and transitions among them is called a finite-state transition system. Finite-state transition systems can be modeled abstractly by a mathematical model called finite automation.

Automata are models of computation. That means they are simplified abstraction of the real things. We merely deal with states and transitions between states. One could say that an automaton is the machine and the program. This makes automata relatively easy to implement in either hardware or software. From the point of view of resource consumption, the essence of a finite automaton is that it is a strictly finite model of computation. Everything in it is of a fixed, finite size and cannot be modified in the course of the computation.

10.4 FINITE AUTOMATA

An automaton with a finite number of states is called a **Finite Automaton (Automata)** (FA) or **Finite State Machine** (FSM).

Finite Automata

- Finite automata are used to recognize string of symbols or patterns.
- FA take a string of symbols as input and change the states of the machine when desired symbol is matched with some functions.
- During a transaction, the automata either move to a next state or stay in the same state.
- Finite automata have two states, **Accept** or **Reject state**. When the input string is processed successfully and it reached a final state, then the statement has been accepted otherwise it is rejected.

Formal definition of a Finite Automaton

An automaton can be represented by a 5-tuple combination $(Q, \Sigma, \delta, q_0, F)$, where –

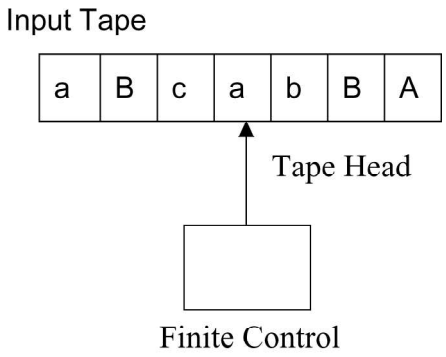
- **Q** is a finite set of states.
- Σ is a finite set of symbols, called the **alphabet** of the automaton.
- **δ** is the transition function.
- $\mathbf{q_0}$ is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

Finite Automata Model:

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

Input tape: A linear tape having the finite number of cells. Input symbols are placed one by one in the input tape.

Finite control: A control mechanism that decides the next state for a particular input symbol. The tape head reads one symbol at a time from the input tape and provides the symbol to finite control for processing.



10.4.1 Finite Automata Related Terminologies

Symbol or letters: Symbols are indivisible objects or entities of a language. A symbol is any single object such as a, 0, 1, #, etc. Usually, characters from a typical keyboard are only used as symbols.

Alphabet: An alphabet is a finite, nonempty set of symbols. The alphabet of a language is normally denoted by ". The elements of " are called letters.

Example:

Σ = {0,1}

Σ = {a, b, c}

Σ = {a, b, c,…,z}

Σ = {%, ^, &, *, $, #}

Word or String: A word or string over an alphabet Σ is a finite sequence of concatenated symbols of Σ.

Example: if Σ = {0, 1} is the given alphabet then the sequence 01, 001, 101, 1001, 11110001 are words on Σ but 012 is not a word since 2 is not an element of Σ.

Length of a String: Length of a string ω, denoted by |ω| or l (ω), is the number of symbols in the string. If 1001 is a word over Σ ={0,1} then |1001|=4

Empty Word or Empty String: The string of length zero is known as empty word or empty string, denoted by ε. | ε |=0.

Concatenation of strings: Let $x = a_1 a_2 a_3 \ldots a_n$ and $y = b_1 b_2 b_3 \ldots b_n$ be two strings. The concatenation of *x* and *y* denoted by *xy*, is the string $a_1 a_2 a_3 \ldots a_n b_1 b_2 b_3 \ldots b_n$. That is, copy of *x* followed by a copy of *y* without any intervening space between them. For example, the concatenation of 1011 and 001 is 1011001 and if ω is a string then ω = εω = ωε where ε is the empty string. If |x|=m and |Y|= n then |xy|= m+n.

Prefix, Suffix and substring: If ω is a string over some alphabet " and if we can write ω as ux, where u and x are two different strings then we can say that u is a **prefix** of ω. Similarly, we can say that x is the **suffix** of ω. Any string u is a **substring** of ω if ω=xuy, that is, concatenation of x, u and y forms the string ω. The empty string ε is always a substring of any string. For example, if ω=010011 is a string over Σ ={0,1} then 0, 01, 010 are the prefix of ω and 1, 11, 011 are suffix of ω. 0, 00, 1001 are substring of ω.

Power of a string: For any string *x* and integer $n \geq 0$, we use x^n to denote the power string. It is formed by sequentially concatenating *n* copies of *x*. In other words

$x^n = \varepsilon$, if $n = 0$;

or $x^n = xx^{n-1}$.

For example if $x = 01$ over " ={0,1} then $x^3 = 010101$.

Power of Alphabets: Power of alphabet denote the set of strings of length k with symbols from Σ denoted by Σ^k (for some integer k). In other words, Σ^k ={ x | x is a string over Σ and |x|= *k*}. Hence, for any alphabet, Σ^0 denotes the set of all strings of length zero. That is, $\Sigma^0 = \{ \varepsilon \}$.

For the binary alphabet { 0, 1 } we have the following-

Σ^0 = { ε }

Σ^1 = { 0, 1}

Σ^2 = { 00, 01, 10, 11}

Σ^3 = { 000, 001, 010, 011, 100, 101, 110, 111}

The set of all strings over an alphabet Σ is denoted by Σ^*. That is,

$\Sigma^* = \Sigma^0 \bigcup \Sigma^1 \bigcup \Sigma^2 \bigcup \ldots \bigcup \Sigma^n \bigcup \ldots$

$= \bigcup \Sigma^k$ where $0 \leq k \leq \infty$

The set Σ^* contains all the strings that can be generated by iteratively concatenating symbols from Σ any number of times.

Example: If Σ = { *a*, *b* }, then

Σ^* = { *ε*, *a*, *b*, *aa*, *ab*, *ba*, *bb*, *aaa*, *aab*, *aba*, *abb*, *baa*, *…*}.

The set of all nonempty strings over an alphabet Σ is denoted by Σ^+. That is, $\Sigma^+ = \Sigma^1 \bigcup \Sigma^2 \bigcup \ldots \bigcup \Sigma^n \bigcup \ldots$

Reversal of strings: For any string $\omega = x_1 x_2 x_3 \ldots x_{n-1} x_n$ the reversal of the string is $\omega^R = x_n x_{n-1} \ldots x_3 x_2 x_1$. For example, reverse of 1101 is 1011.

Language: A language **L** over an alphabet Σ is a collection of words over Σ. Since Σ^* is the set of all words on Σ. Thus, a language L is simply a subset of Σ^*.

For example, if Σ = {0,1} is an alphabet then

L_1 = {0, 01, 01^2, 01^3, …}

L_1 consists of all the strings starting with a 0 followed by any number of 1.

L_2 = {$0^m 1^m$: m>0}

L_2 consists of words beginning with one or more 0's followed by same number of 1's.

L_3 = {$0^m 1^n$: m>0, n>0}

L_3 consists of words beginning with one or more 0's followed by one or more 1's.

L_4 = {ε}

L_4 is an empty language.

Operations on Languages

Since languages are set of strings, we can apply set operations on the languages.

Union: If L_1 and L_2 are two languages then the union of L_1 and L_2 denoted by $L_1 \cup L_2$, a word $x \in L_1 \cup L_2$ iff x L_1 or $x \in L_2$.
Example: {0, 11, 01, 011} {1, 01, 110} = {0, 1, 11, 01, 011, 110}

Intersection: If L_1 and L_2 are two languages then the intersection of L_1 and L_2 denoted by $L_1 \cap L_2$, any word $x \in L_1 \cap L_2$ iff $x \in L_1$ and $x \in L_2$.
Example: {0, 11, 01, 011 } $\cap$ {1, 01, 110 } = { 01 }

Complement: Usually, Σ^* is referred as the universe of all the languages over the alphabet over Σ. So, complement of any language is taken with respect to Σ^*. Thus, for a language L, the complement is $\overline{L} = \{x \in \Sigma^* \text{ and } x \notin L\}$.
Example: Let L = { x : |x| is even }. Then its complement is the language $\overline{L} = \{x\Sigma^* : |x| \text{ is odd}\}$. Similarly, we can define other usual set operations on languages like relative complement, symmetric difference, etc.

Reversal of a language: The reversal of a language L, denoted as L^R, is defined as -

$$L^R = \{\omega^R : \omega \ L \}$$

Example:

1. If L = {0, 01, 011}. Then L^R = {0, 10, 110 }.
2. If L = { $a^n b^n$: n is an integer }. Then L^R= { $b^n a^n$: n is an integer }.

Concatenation: The concatenation of languages $L_1$and L_2 over an alphabet Σ is defined as-

$$L_1L_2 = \{ xy : x \in L_1 \text{ and } y \in L_2 \}$$

Example: if L_1 = {a, ab} and L_2 = {b, ba } then
L_1L_2 = {ab, aba, abb, abba}.
Some Properties of language concatenation

1. $L_1L_2 \neq L_2L_1$ in general.

2. $L \phi = \phi$
3. $L\{\varepsilon\} = L = \{\varepsilon\}L$

The operation L^n denotes the concatenation of L with itself n times. This is defined formally as follows:

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

$L^n = LL^{n-1}$

Example: if L = {0, 01}. Then according to the definition, we have

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

$L^1 = \{0, 01\}$

$L^2 = \{0, 01\}\{0, 01\} = \{00, 001, 010, 0101\}$

$L^3 = \{0, 01\}\{00, 001, 010, 0101\}$

$= \{000, 0001, 0010, 00101, 0001, 00101, 01001, 010101\}$

Kleene's Star operation:The Kleene star operation on a language L, denoted as L^* is defined as follows**:**

$L^* = \bigcup L^k$

$= L^0 \bigcup L^1 \bigcup L^2 \bigcup \ldots \bigcup L \bigcup \ldots$

= {x: x is the concatenation of zero or more strings from L}

Also

$L^+ = L^1 \bigcup L^2 \bigcup \ldots \bigcup L^n \bigcup \ldots$

Example: If L = { *a*, *ab* }. Then we have,

$L^* = L^0 \bigcup L^1 \bigcup L^2 \bigcup \ldots$

= { ε } {*a*, *ab*} {*aa*, *aab*, *aba*, *abab*} …

$L^+ = L^1 \bigcup L^2 \bigcup \ldots \bigcup L^n \bigcup \ldots$

= {*a*, *ab*} {*aa*, *aab*, *aba*, *abab*} …

10.5 TRANSITION SYSTEM

In formal automata, a **transition system** is the heart of computation. The transition system is used to describe the behavior of the systems. It consists of input symbols, states and transitions between states. The difference between transition function and finite-state automata are:

- In transition function, the set of states is not necessarily finite.
- In transition function, the set of transitions is not necessarily finite.

- No “start” state or “final” state are given in the transition function.

Formally, a **transition system** is a pair $(S,\rightarrow)$ where S is a set of states and $\rightarrow$is a relation of state transitions. A transition from state p to state q is written as $p\rightarrow q$.

Sometimes labels are given in the transition called as **labelled transition system.** Labelled transition system is represented as a tuple (S,,→) where S is a set of states, is a set of labels, and $\rightarrow$ is relation of the labelled transitions and it is written as

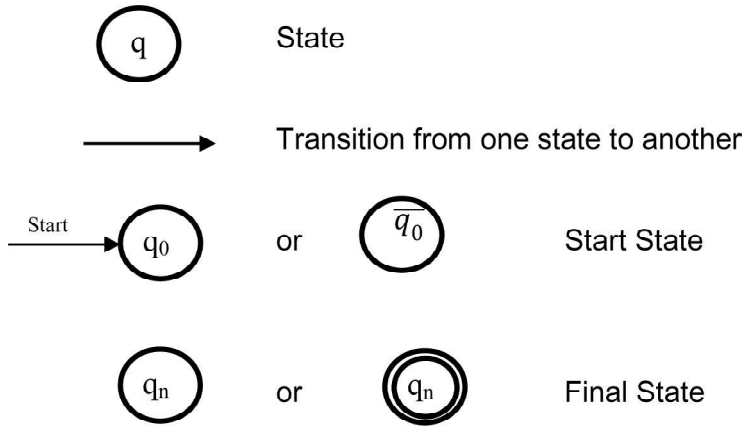
$$p \xrightarrow{\alpha} q$$

The above transition represents a transition from state p to state q with label α. Labels can represent different things depending on the language. The label can represent the input symbol, conditions for the transition, or actions performed during the transition.

The transition systems can be represented as directed graphs also. The directed graph can be constructed as follows:

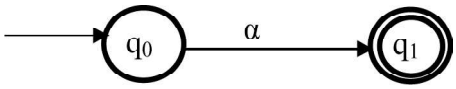
- There is a node for each state in Q, which is represented by the circle.
- There is a directed edge from node q to node p labelled a if δ(q, a) = p.
- In the start state, there is an arrow with no source.
- Accepting states or final states are indicated by a double circle.

Some Notations that are used in the transition diagram are depicted below:



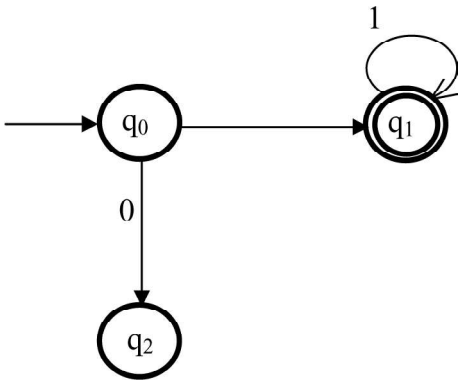
To visualize and represent finite automata, we use **transition graphs**, in which the vertices represent the states and the edges represent the transitions. The labels on the vertices are the names of the states,

while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some finite automata *M*, then the graph associated with *M* will have one vertex labelled q_0 and another vertex labelled q_1. An edge (q_0, q_1) labelled α represents the transition $\delta(q_0, \alpha) = q_1$.For finite automata, initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.



Example 1: Finite Automata with " = {0, 1} accepts all strings starting with 1.

Solution:



The finite automata can be represented using a transition graph. In the above diagram, the machine initially is in start state q_0 then on receiving input 1 the machine changes its state to q_1. From q_0 on receiving 0, the machine changes its state to q_2, which is the dead state. From q_1 on receiving input 0, 1, the machine changes its state to q1, which is the final state. The possible input strings that can be generated are 10, 11, 110, 101, 111....., that means all string starts with 1.

10.5.1 Transition Table

The transition table is basically a tabular representation of the transition function. It takes two arguments (a state and a symbol) and returns a state (the "next state").

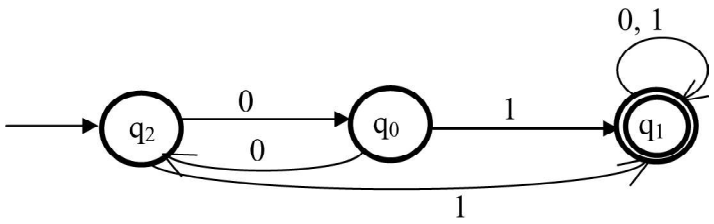
A transition table is represented by the following things:

- Columns correspond to input symbols.

- Rows correspond to states.
- Entries in the table correspond to the next state.
- The start state is denoted by an arrow with no source.

The accept state is denoted by a star

Example 1:



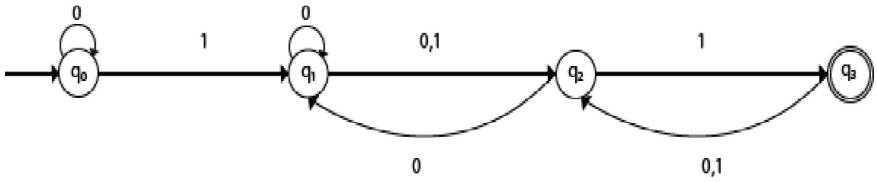
Solution:

Transition table of given DFA is as follows:

Present State	Next state on input '0'	Next state on input '1'
$\rightarrow q_0$	q_1	q_2
q_1	q_0	q_2
$*q_2$	q_2	q_2

In the above table, the first column indicates the states. In column 0 and 1, the next states are shown.The first row of the transition table can be read as: when the current state is q_0, on input 0 the next state will be q_1 and on input 1 the next state will be q_2.In the second row, when the current state is q_1, on input 0, the next state will be q_0, and on 1 input the next state will be q_2 and so on. The arrow in q_0 indicates that it is a start state and circle marked to q_2 indicates that it is a final state.

Example 2:



Solution:

Transition table of given NFA is as follows:

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_0	q_1
q_1	q_1, q_2	q_2
q_2	q_1	q_3
$^*q_3$	q_2	q_2

Explanation:

The first row of the transition table can be read as: when the current state is q_0, on input 0 the next state will be q_0 and on input 1 the next state will be q_1.In the second row, when the current state is q_1, on input 0 the next state will be either q_1 or q_2, and on 1 input the next state will be q_2 and so on.

10.5.2 Transition Function

In a finite automaton at any stage of time, the finite control is in some internal state. It processes only one input at a time and the head scans only one input symbol at a time. From the current internal state, the finite control unit determined the next state with the help of the transition function. This transition function provides the next state in terms of the current state, the input symbol.The information provided by the transition function is implemented when the finite automata move from the current state to the next state. In the formal definition of finite automata, we write the transition function as follows:

$$\delta: Q \times \Sigma \rightarrow Q$$

where δ is the transition function, Q is the set of states and Σ is the set of alphabets.

Extended Transition Function

The extended transition function is an extension of the transition function. Transition function reads only one input at a time whereas the extended transition function processes a string of input symbols at a time and moves from one step to another step denoted as δ^*.

$\delta^* : Q \times \Sigma^* \rightarrow Q$. extended transition function

Here, the second argument of δ^* is a string rather than a single symbol and the state return by $\delta^*$is the state of the automaton after reading that string.

For example,

if $\delta(q_0,a) = q_1$ and

$\delta(q_1,b) = q_2$, then

$\delta^*(q_0,ab) = q_2$.

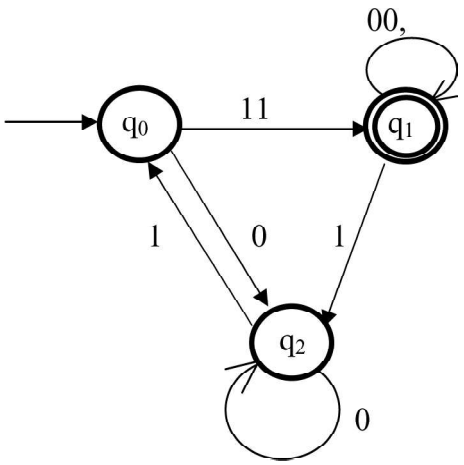
10.5.3 String Acceptance by Finite Automata

A finite automaton starts its processing at the initial state q_0 and reads the leftmost symbol in the input string. It checks for the transition function related to the state and the input symbol and moves to the next state. During each move of the automaton, the input mechanism advances one position to the next state, so after eachmove,we can say that it consumes one input symbol. When the end of the input string is reached, the string is accepted if the automaton is in one of its final states. Otherwise, the string is rejected. The input head can move only one step from left to right and reads exactly one symbol in each step. The transitions from one internal state to another are managed by the transition function δ.

For example, if δ (q_0, a) = q_1, then if the finite automata in state q_0 and the input symbol is a, thenit will move into state q_1. In discussing automata, it is essential to have a clear and intuitive picture to work with. Transitiongraphs are used to visualize this, in transition graph the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states and the labels on the edges are the input symbol. If in the transition graph, we reach the final state after moving through the graph starting from q0 by moving the edges as per input sequence if we reach a final state then the string will be accepted and if we reach into a non-final state after completing the processing of the string, then the string is rejected.

Example

Let us consider the finite automata shown in the figure. The acceptable strings are



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

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



CHECK YOUR PROGRESS

Q.1: State True or False

a. $L^* = L^+L$

b. |aa|=2 if Σ ={aa, b}

c. L^0 = empty set for any Language L

d. $\{a,b\}^k$ is set of all strings of length k

e. $(L^*)^* = L^*$

f. L^+ is a subset of L^*

g. If $L^* = L^+$ then L is a empty set

h. $L_1(L_2 \cap L_3) = L_1L_2 \cap L_1L_3$



10.6 LET US SUM UP

- Automata theory is the study of abstract machines and automaton, as well as the problems that can be solved using these machines.
- It is a self-operating machine that follow a predetermined sequence of operations or response of predetermined instructions that used to check a computational problem whether it can be solved by that machine or not.
- A finite automaton can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$
- **Transition system** is the heart of computation. The transition system is used to describe the behavior of the systems.
- To visualize and represent finite automata, **transition graphs are used**, where the vertices represent states and the edges represent transitions and the labels on the edges are the input symbol.
- The transition table is basically a tabular representation of the transition function.
- The finite automaton processes only one input at a time. The transition function provides the next state in respect to the current state and the input symbol.
- The extended transition function is an extension of the transition function. Transition function reads only one input at a time whereas the extended transition function processes a string of input symbols at a time
- When the finite automaton reaches the end of the input string, the string is accepted if the automaton if is in one of its final states. Otherwise, the string is rejected.
- An epsilon transition also called as epsilon move allow automaton to change its state without consuming an input symbol. It is present in almost all kinds of nondeterministic automaton.



10.7 FURTHER READING

1. Peter Linz, “An Introduction to Formal Language and Automata”, 4th Edition, Narosa Publishing house , 2006.
2. M.Sipser; Introduction to the Theory of Computation; Singapore: Brooks/Cole, Thomson Learning, 1997.
3. John.C.martin, “Introduction to the Languages and the Theory of Computation”,Third edition, Tata McGrawHill, 2003.
4. K.Krithivasan and R.Rama; Introduction to Formal Languages, Automata Theory and Computation; Pearson Education, 2009.
5. J.E.Hopcroft, R.Motwani and J.D.Ullman , “Introduction to Automata Theory Languages and computation”, Pearson Education Asia , 2001.



10.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1:
a. False
b. False
c. False
d. True
e. True
f. True
g. False
h. False



10.9 MODEL QUESTIONS

Q 1: What is transition function explain briefly?

Q 2: What do you mean by finite automata? Explain the formal definition of finite automata.

Q 3: What is extended transition function? Explain the use of extended transition function.

Q 4: What is the use of transition diagram in finite automata? Explain.

Q 5: What is language? What are the operations permitted on finite automata explain in brief?

Q 6: Explain the concept of alphabet, string, prefix, suffix and substring.

Q 7: What is ε-Transitions? What is the use of ε-Transitions in finite automata?

Q 8: Explain the concept of converting transition graph to transition table with suitable example.

*** ***** ***

UNIT 11: TYPES OF FINITE AUTOMATA

UNIT STRUCTURE

11.1 Learning Objectives

11.2 Introduction

11.3 Deterministic Finite Automata (DFA)

11.4 Non-Deterministic Finite Automata (NFA)

11.5 Equivalence of DFA and NFA

11.6 Minimization of Finite Automata

11.7 Application of Finite Automata

11.8 Finite Automata with ε-Transition

11.9 Let Us Sum Up

11.10 Further Reading

11.11 Answers to Check Your Progress

11.12 Probable Questions

11.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- define Deterministic Finite Automata (DFA) and design DFA
- define Non-Deterministic Finite Automata(NFA) and Design NFA
- know the applications of Finite Automata
- convert Non-Deterministic Finite Automata to Deterministic Finite Automata
- minimize an Automata

11.2 INTRODUCTION

In this chapter we are going to explore about Deterministic Finite Automata (DFA) and Non-Deterministic Finite Automata (NFA). A Finite automation is called DFA if for one input symbol at a time the machine produces only one specific path from current state to next state. DFA does not accept the null move, i.e., a DFA cannot change its state without any input character. On the other hand, a finite automation is called NFA if there exist many paths for one input symbol from current state to the next state.

NFA can accept null move, i.e., a NFA can change its state without any input character. Every DFA is a NFA but every NFA is not a DFA. Here we are also going to look at some of the application of finite automata. We also discuss the equivalence of NFA and DFA, how we can convert from one form to the other. We are also going to look how we can minimize a finite automaton.

11.3 DETERMINISTIC FINITE AUTOMATA (DFA)

A DFA (Deterministic Finite State Automaton) is a simple machine that reads an input string (one symbol at a time). After the input string has been completely processed, it decides whether to accept or reject the input string. As the symbols are read from the tape, the automaton changes its states to reflect status of what it has processed so far.

A DFA conceptually consists of 3 parts:

1. A tape to hold the input string. The tape is divided into a finite number of cells. Each cell holds a symbol from alphabets (Σ).
2. A tape head for reading symbols from the tape
3. A control, which itself consists of 3 things:
   - finite number of states that the machine is allowed to be in (zero or more states are designated as accept or final states),
   - the current state, initially the start state,
   - a state transition function for changing the current state to next state.

An automaton processes an input string on the tape by repeating the following actions until the tape head has traversed the entire string:

1. The tape head reads the current cell and sends the symbol (s) found there to the control. Then, the tape head moves to the next cell.
2. The control takes symbol (s) and the current state as input and check for the state transition function to get the next state, the state returns by the state transition function becomes the new current state.

Once the entire string has been processed, the state in which the automation enters is examined. If it is a final state, the input string is accepted; otherwise, the string is rejected.

We can formulate DFA with the following formal definition:

Deterministic Finite State Automaton:

A Deterministic Finite State Automaton (DFA) is a collection of 5-tuples**:** **$D=(Q, \Sigma, \delta, q_0, F)$**

- Q is a finite set of states.
- Σ is a finite set of input symbols or alphabet.
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function. Intuitively, δ is the function that produces which state to move from current state in response of the input, i.e., if Machine (M) is in state q and sees input a, it moves to state generated by $\delta(q, a)$.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accept or final states.

Transition table:

It is basically a tabular representation of the transition function that takes two arguments (a state and a symbol) and returns a value (the "next state").

- Rows correspond to states,
- Columns correspond to input symbols,
- Entries correspond to next states,
- The start state is marked with an arrow,
- The accept states are marked with a star (*).

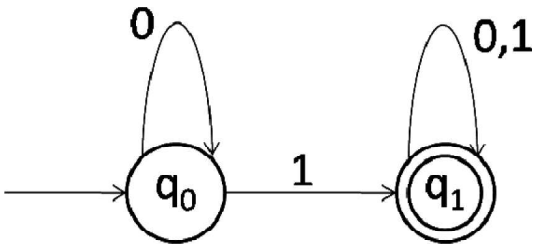
	0	1
q_0	q_0	q_1
* q_0	q_1	q_1

Transition diagram:

A state transition diagram or simply a transition diagram is a directed graph which can be constructed as follows:

1. For each state in *Q*, there is a node (vertex).

2. There is a directed edge from node *q* to node *p* labeled as *a* iff *δ(q, a) = p*. (If there are several input symbols that cause a transition, the edge is labeled as list)
3. There is an arrow with no source into the start state.
4. Accepting states are indicated by double circle.



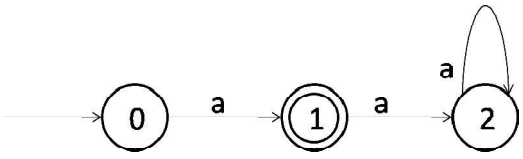
Here is an informal description how a DFA operates. An input to a DFA can be any string ω Σ*. Put a pointer to the start state q. Read the input string ω from left to right, one symbol at a time, moving the pointer according to the transition function, δ. If the next symbol of *ω* is *a* and the pointer is on state *p*, move the pointer to the state provided by δ (p, a). When the end of the input string *ω* is encountered, the pointer is on some state, *r*. The string is said to be accepted by the DFA if r F and rejected if r F. Note that there is no formal mechanism for moving the pointer.

A language L Σ* is said to be **regular** if $L = L(M)$ for some DFA *M*.

Example 1: Q = {0, 1, 2 }, Σ = { a }, F = {1 }, the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State (δ (q, a))
0	a	1
1	a	2
2	a	2

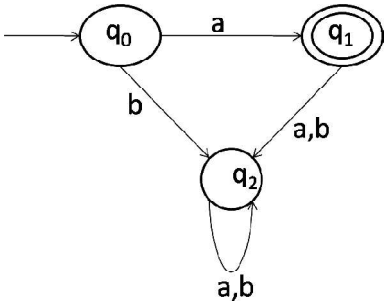
A state transition diagram for this DFA is given below.



Example 2: Q = { 0, 1, 2 }, Σ = { a, b }, F = { 1 }, the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State (δ (q, a))
0	a	1
0	b	2
1	a	2
1	b	2
2	a	2
2	b	2

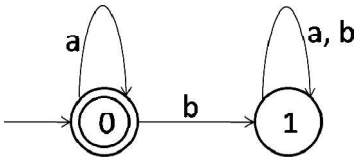
A state transition diagram for this DFA is given below



Example 3: Q = { 0, 1 }, Σ = { a, b }, F = { 0 }, the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State (δ (q, a))
0	a	0
0	b	1
1	a	1
1	b	1

A state transition diagram for this DFA is given below.



δ^* Definition

Instead of providing one symbol at a time to the transition function δ we can provide a string as input to the transition function, called extended transition function (δ^*)

$\delta^* : Q \times \Sigma^* \rightarrow Q$.

Here the second argument of δ^* is a string rather than a single symbol, and its value will be in after reading that string. For example, if

$\delta (q_0, a) = q_1$

$\delta (q_1, b) = q_2$

then $\delta^* (q_0, ab) = q_2$

Formally δ^* is defined recursively as

$\delta^* (q, \varepsilon) = q$

$\delta^* (q, \omega a) = \delta(\delta^* (q, \omega), a)$,

for all $q \in Q$, $\omega \in \Sigma^*$, $a \in \Sigma$.

String accepted by DFA

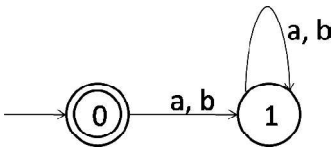
A string ω is accepted by a DFA $<Q, \Sigma, q_0, \delta, F>$ if and only if $\delta^*(q_0, \omega) \in F$. In other words, a string is accepted by a DFA if and only if the DFA starting at the initial state ends in an accepting state after reading the string.

Language accepted by DFA

The language accepted by a DFA $M =(Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M, in formal notation

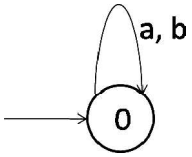
$L(M) = \{\omega \in \Sigma^* \mid \delta^* (q_0, \omega) \in F\}$

Example 1:



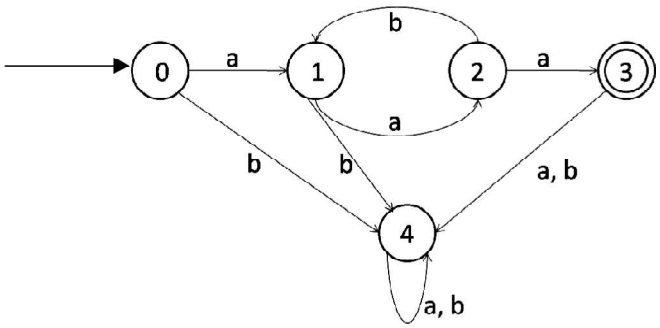
This DFA accepts { ε } because it can go from the initial state to the accepting state (also the initial state) without reading any symbol of the alphabet i.e. by reading an empty string ε . It accepts nothing else because any non-empty symbol would take it to state 1, which is not an accepting state, and it stays there.

Example2:



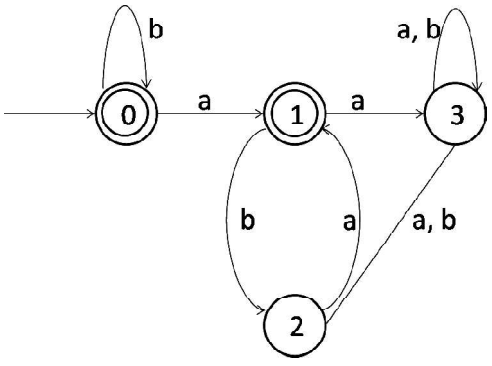
This DFA does not accept any string because it has no final state. Thus, the language it accepts is the empty set Φ.

Example3:



This DFA has a cycle: 1 - 2 - 1 and it can go through this cycle any number of times by reading substring ab repeatedly. To find the language it accepts, first from the initial state it can go to state 1 by reading one a. Then from state 1, it can go through the cycle 1 - 2 - 1 any number of times by reading substring ab any number of times to come back to state 1. This is represented by $(ab)^*$. Then from state 1 go to state 2 and then to state 3 by reading aa. Thus, a string that is accepted by this DFA can be represented by $a(ab)^*aa$.

Example 4:

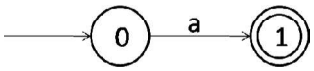


This DFA has two accepting states: 0 and 1. Thus the language that is accepted by this DFA is the union of the language accepted at state 0 and the one accepted at state 1. The language accepted at state 0 is $b^*$and { ε }.

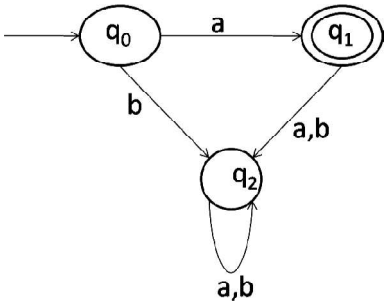
To find the language accepted at state 1, first at state 0 read any number of b's. Then go to state 1 by reading one a. At this point, (b^*a) will be read. At state 1 go through the cycle 1 - 2 - 1 any number of times by reading substring ba repeatedly. Thus, the language accepted at state 1 is $b^*a(ba)^*$.

11.4 NON-DETERMINISTIC FINITE AUTOMATA (NFA)

In the previous section we have studied DFAs that accept some simple languages such as ϕ, {ε}, and { a } etc. From this discussion, it has been noticed that these DFAs have some states and transitions which does not lead to final states. For example, if we need an FA that accepts language { a } regardless of the alphabet (whether Σ is { a } , { a , b } or any other) it could look like-



It is not a DFA if Σ= {a, b}. A DFA that accepts { a } from Σ= {a, b} would need more states and transitions as shown in below-



To avoid these redundant states and transitions and to make modeling easier, we can use Nondeterministic Finite Automata (NFA). Below we are going to formally define nondeterministic finite automata and discuss some examples.

Nondeterminism is an important abstraction in computer science. Importance of nondeterminism is found in the design of algorithms. For examples, there are many problems (Travelling salesman, Hamiltonian cycle, clique, etc.) with efficient nondeterministic solutionsbut there are no known efficient deterministic solutions present. Because the behavior of a process might depend on some messages from other processes that might arrive at arbitrary times with arbitrary contents.

It is easy to construct and comprehend an NFA rather than a DFA for a given regular language. The concept of NFA can also be used in proving many theorems and results. Hence, it plays an important role in Computer Science.

In this context, nondeterminism can be incorporated naturally on a FA. That is, an NFA is defined in the same way as the DFA but with the following two exceptions:

- multiple next state for same input symbol.
- ε - transitions.

Multiple Next State: In contrast to a DFA, the next state is not necessarily uniquely determined by the current state and input symbol in case of an NFA. (InDFA, there is exactly one start state and exactly one transition for every state and each symbol in Σ).

This means that, in a state *q* and with input symbol 0, there could be zero, one or more than one next state, i.e. the value of δ(q,a) is a subset of *Q*.

$\delta(q,a) = (q_1, q_2, \ldots, q_k)$

which means that any one of $q_1, q_2, \ldots, q_k$ could be the next state for δ(q,a).

The zero next state case is a special one giving δ(q,a)=Φ, which means that there is no next state on input symbol a when the automata is in state *q.* In such cases, we may think that the automata "hangs" and the input will be rejected.

ε- transitions: In an ε-transition, the tape head doesn't do anything, it simply stays in the same position without reading any symbol. However, the state of the automata can be changed,i.e.it can move to zero, one or more states. This is written as

$\delta(q, \varepsilon) = (q_1, q_2, \ldots, q_k)$

implying that the next state could be any one of $q_1, q_2, \ldots, q_k$ without consuming the any input symbol.

Acceptance of String:

An NFA is said to accept an input string **ω**if it is possible to start at a starting state and process **ω**, moving according to the transition rules and jump to

the next state (not uniquely), the automata reach in a final state. There may be several possible paths through the automation and in response to an input **ω it**can reach from the start state to the final state. Out of all the possible paths in the NFA, some of these paths may lead to final states while others may not. The automation is said to accept **ω** if at least one computation path on input **ω** starting from at least one start state leads to an accept state. Otherwise, the automation rejects the input **ω**. Alternatively, we can say that, **ω** is accepted iff there exists a path with label**ω** from some start state to some accept state. Since there is no mechanism for determining which state to start in or which of the possible next moves to take in response to an input symbol leads to a final state, we can only guess and check for an alternative path until we reach the final state or we explore all the possible paths.

Formal definition of NFA:

An NFA is a quadruple N = (Q, Σ, δ, q_0, F) where *Q*, Σ, q_0, and *F* bear the same meaning as for a DFA, but δ, the transition function is redefined as follows:

$$\delta: Q \times \{\Sigma \cup \{\varepsilon\}\} \rightarrow p(Q)$$

where *P*(*Q*) is the power set of *Q*.

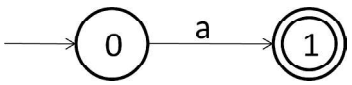
- As in the DFA, the set Q in the above definition is simply the set of finite number of states.
- The transition function is also called a **next state function**. Unlike DFAs, an NFA moves into one of the states given by the function δ(q, a)non-deterministically, if it receives the input symbol ‘a’ while in state q.
- In the transition function δ, for each state q in Q and for each symbol ‘a’ in Σ, δ(q, a) must be specified. But it can be the empty set, in which case the NFA aborts its operation.
- As in the case of DFA just by looking at the final state we can distinguish whether sequences of inputs given to the finite automaton is accepted or not. In case of NFA, it is not possible just by looking at one transition as there may be some other transition which may lead to an accepted state.

- Note that any DFA is also an NFA but vice-versa is not true.

Example 1: Q = { 0, 1 }, Σ = { a }, F = { 1 }, the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State (δ (q, a))
0	a	{ 1 }
1	a	Φ

A state transition diagram for this finite automaton is given below.



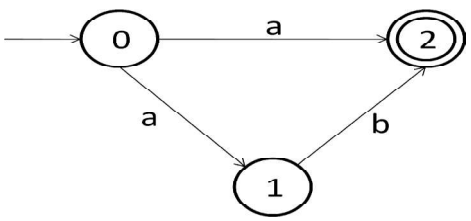
If the alphabet Σ is changed to {a, b} instead of { a }, this is still an NFA that accepts { a } .

Example 2: Q = { 0, 1, 2 }, Σ = { a, b }, F = { 2 }, the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State (δ (q, a))
0	a	{ 1 , 2 }
0	b	Φ
1	a	Φ
1	b	{ 2 }
2	a	Φ
2	b	Φ

Note that for each state there are two rows in the table for δ corresponding to the symbols ‘a’ and b, while in the Example 1 there is only one row for each state.

A state transition diagram for this finite automaton is given below.



The Extended Transition function δ*:

To describe acceptance by an NFA formally, it is necessary to define the extend transition function, denoted as **δ***. The extend transition function **δ*** takes a state q ∈ Q and a string ω Σ* as input and returns the set of states, *S* Q.

δ* is defined as:

1. **δ***(q, ε) = {q} that is, without rending any input symbol, the NFA remains in same state.
2. Let ω = *xa* for some ω, where x ∈ Σ* and a ∈ Σ. Also assume that δ*(q, *x*) = {p_1, p_2, …, p_k}

   Then δ*(q, ω) = $\bigcup_{i=1}^{k} \delta(p_i, a)$

That is, δ(q, ω) can be computed by first computing δ*(q_0, *x*), and by then following any transitive from any of these states that is labeled *a*.

The Language accepted by an NFA:

The formal definition of language accepted by an NFA as follows:

If N = (Q, Σ, δ, q_0, F) is an NFA, then the language accepted by *N* is written as *L*(*N*) is given by

L(N) = {ω | δ*(q_0, ω) F = Φ }

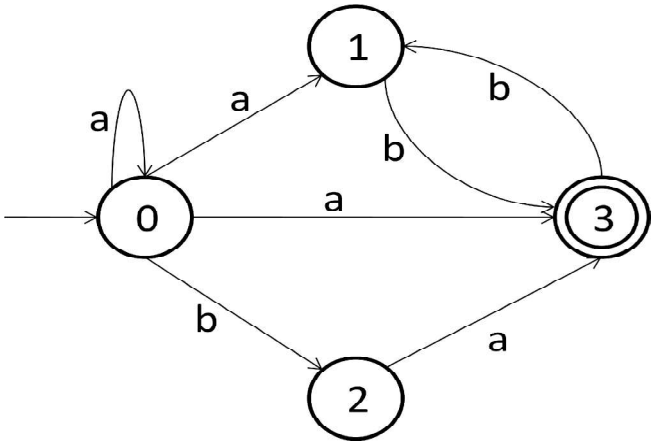
That is, *L*(*N*) is the set of all strings *ω* in Σ* such that δ*(q_0, ω) contains at least one accepting state.

Example

For example consider the NFA with the following transition table:

State (q)	Input (a)	Next State (δ (q, a))
0	a	{ 0 , 1 , 3 }
0	b	{ 2 }
1	a	Φ
1	b	{ 3 }
2	a	{ 3 }
2	b	Φ
3	a	Φ
3	b	{ 1 }

The transition diagram for this NFA is as given below.



the language it accepts is $a^*(ab + a + ba)(bb)^*$.



CHECK YOUR PROGRESS

Q 1: State True or False

a. In a DFA all states have the same number of transitions.

b. A DFA cannot have more than one accepting states.

c. A DFA has finite number of states.

d. A DFA accepts a Language.

e. In a DFA $\delta^*(0,abb) = 0$ is possible.

f. In an NFA, all states have the same number of transitions.

g. An NFA cannot have more than one accepting states.

h. An NFA has finite number of states.

i. In an NFA, $\delta^*(0,a)= \delta(0,a) = \{1,2\}$ is possible.

j. In an NFA, $\delta^*(0,abaa)) = \{1,2\}$ is possible.

11.5 EQUIVALENCE OF DFA AND NFA

ε -closures:The design of NFA and DFA can be made more formal by defining theε-closure for a state or for a set of states. The idea of ε-closure is that, when moving from a state *p* to a state *q* or from a set of states S_i to a set of states S_j for an input a $\in \Sigma$, we take account of all ε-moves that could be made after the transition. Formally, for a given state *q*,

ε-closures(q) = {p| p can be reached from q by zero or more ε-moves}

Similarly, for a given set $R \subseteq Q$

ε-closures(R)= {$p \in Q$ | p can be reached from any $q \in R$ by following zero or more ε-moves}

To remove the ε-transition from an NFA and to design anequivalent *NFAN'* without ε-transition. the first rule is

δ'(q, a)= ε-closure(δ(q,a))

Conversion of NFA-ε to NFA

Let $M_1 = < Q^1, \Sigma, q_0^1, \delta^1, F^1 >$ be an NFA with ε moves that recognizes a language L. Then the NFA $M_2 = < Q^2, \Sigma, q_0^2, \delta^2, F^2 >$ that satisfies the following conditions recognizes L:

$Q^2 = Q^1$,

$q_0^2 = q_0^1$

$\delta^2(q, a) = \delta^{1*}(q, a) = \varepsilon \left(\bigcup_{p \in \varepsilon(q)} \delta^1(p, a)\right)$

$F^2 = F^1 \{ q_0^1 \}$ if $\varepsilon(\{ q_0^1 \}) \cap F^1 \Phi$

$= F^1$ otherwise.

Thus, to obtain an NFA $M_2 = <Q^2, \Sigma, q_0^2, \delta^2, F^2 >$ which accepts the same language as the given NFA-ε $M_1 = < Q^1, \Sigma , q_0^1, \delta^1 , F^1 >$ does, first copy the states of Q_1 into Q_2.

Then, for each state q of Q_2 and each symbol a of Σ, finding $\delta^2(q, a)$ will be as follows:

1. Find ε({q}), that is all the states that can be reached from q by traversing ε arcs. Then, collect all the states that can be reached from each state of ε ({q}) by traversing one arc labeled with the symbol a. The ε closure of the set of those states is $\delta^2(q, a)$.

2. The set of accepting states F_2 is the same as F_1 if no accepting states can be reached from the initial state q_0^1 through ε arcs in M_1. Otherwise, if an accepting state can be reached from the initial state q_0^1 through ε arcs in M_1, then all the accepting states of M_1 plus state q_0^1 are the accepting states of M_2.

Removing ε transition

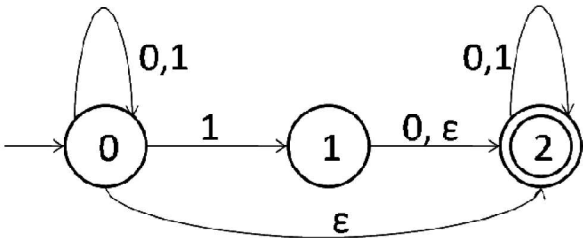
ε-transitions does not increase the power of an *NFA*. That is, for any *NFA-ε*(*NFA* with ε transition), we can always construct an equivalent *NFA* without *ε*-transitions. The equivalent *NFA* must keep track where the *NFA-ε* goes at every step during computation. This can be done by adding extra transitions for removal of every *ε*- transitions from the *NFA-ε* as follows:

If we remove the ε - transition δ(p, ε) = q from the *NFA-ε*, then we need to move from state *p* to all the state γ on input symbol q Σ which are reachable from state q (in the *NFA-ε*) on same input symbol *q*. This allowsus to modify the *NFA* to move from state *p* to all states on some input symbols which were possible in case of *NFA-ε* on the same input symbol. This process is stated formally by the following theory.

Theorem: If *L* is accepted by an *NFA-ε N*, then there is some equivalent *NFA N'* without *ε* transitions accepting the same language *L.*

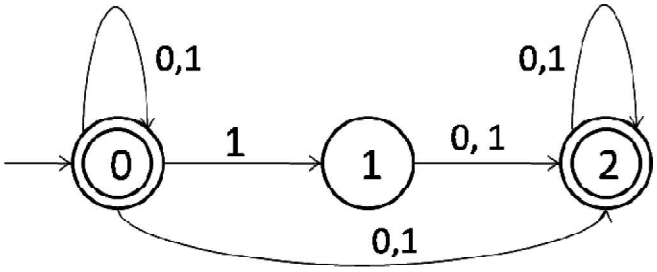
Example: Consider the following *NFA* with ε transition.

	0	1	E
→q_0	{q_0}	{q_0, q_1}	{q_2}
q_1	{q_2}	Φ	{q_2}
q_2(F)	{q_2}	Φ	{q_2}



The equivalent NFA is-

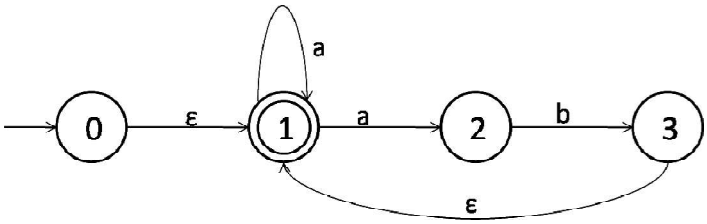
	0	1
→q_0(F)	{q_0, q_2}	{q_0, q_1, q_2}
q_1	{q_2}	{q_2}
q_2(F)	{q_2}	{q_2}



Since $\delta(q_0, \varepsilon) = q_2$ in NFA-ε the start state q_0 must be final state in the equivalent *NFA* .

Since $\delta(q_0, \varepsilon) = q_2$ and $\delta(q_2, 0) = q_2$ and $\delta(q_2, 1) = q_2$ we add moves $\delta(q_0, 0) = q_2$ and $\delta(q_0, 1) = q_2$ in the equivalent *NFA* . Other moves are also constructed accordingly.

Example: Let us convert the following NFA-ε to NFA.



The set of states Q_2 of NFA is {0, 1, 2, 3), the initial state is 0 and the accepting states are 1 and 0, since 1 is in ε({0}). The transition function δ_2 is obtained as follows:

$\delta_2(0, a)$: First ε({0}) = {0, 1}. Then from the transition function of the NFA-ε $\delta_1(0 , a) = \Phi$, and $\delta_1(1, a) = \{1, 2\}$.

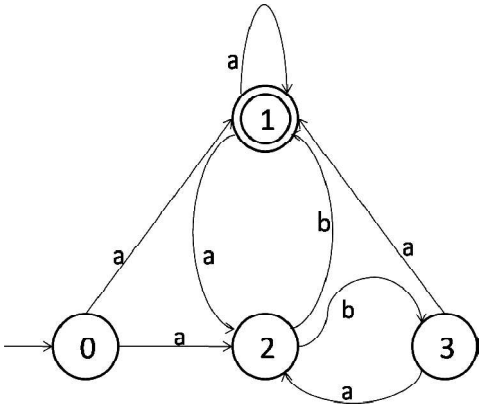
Hence, $\delta_2(0, a) = \varepsilon(\{1, 2\}) = \{1, 2\}$.

For $\delta_2(0, b)$, since ε({0}) = {0, 1} and $\delta_1(0, b) = \delta_1(1, b) = \Phi$, $\delta_2(0, b) = \Phi$.

Similarly, δ_2 can be obtained for other states and symbols. They are given in the table below together with ε({q}) and $\bigcup_{p \in \varepsilon(q)} \delta_1(p, a)$.

State q	Input 'a'	ε({q})	.	δ_2(q,a) (= ε( ))
0	a	{0, 1}	{ 1 , 2 }	{ 1 , 2 }
0	b	{0, 1}	Φ	Φ
1	a	{ 1 }	{ 1 , 2 }	{ 1 , 2 }
1	b	{ 1 }	Φ	Φ
2	a	{ 2 }	Φ	Φ
2	b	{ 2 }	{ 3 }	{ 1 , 3 }
3	a	{1, 3}	{ 1 , 2 }	{ 1 , 2 }
3	b	{1, 3}	Φ	Φ

The NFA thus obtained is shown below.



Equivalence of *NFA* and *DFA*

DFA is a special type of *NFA* and hence the class of languages accepted by *DFA* is a subset of the class of languages accepted by *NFAs*. Surprisingly, these two classes are in fact equal. *NFA*s appeared to have more power than *DFA*s because of ε-transition and multiple next states. But they are no more powerful than *DFA*s in terms of the languages they accept.

Converting *DFA* to *NFA*

Theorem: Every *DFA* has as equivalent *NFA*

Proof: A *DFA* is just a special type of an *NFA*. In a *DFA*, the transition function is defined from QxΣ to Qwhereas in case of an *NFA* it is defined from QxΣ to 2^Q and D=(Q, Σ, q_0, δ, F) be *a DFA*. We construct an equivalent *NFA* N=(Q', Σ, q_0, δ', F) as follows.

$\{q_i\} \in Q'$ for all q_i Q

$\delta'(\{p\}, a) = \{(\delta(p,a)\}$ i.e.

If $\delta(p,a) = q$ and $\delta'(\{p\}, a) = \{q\}$

All other elements of *N* are as in *D*.

If $\omega = a_1, a_2, \ldots, a_n$ L(D)then there is a sequence of states $q_0, q_1, q_2, \ldots, q_n$ such that

$\delta(q_{i-1}, a_i) = q_i$ and q_n F

Then it is clear from the above construction of *N* that there is a sequence of states (in *N*) $\{q_0\}, \{q_1\}, \{q_2\}, \ldots, \{q_n\}$ such that $\delta'(q_{i-1}, a_i) = \{q_i\}$ and $\{q_n\} \in F$ and hence $\omega \in L(N)$

Similarly, we can show the converse.

Hence, L(N) = L(D)

Converting NFA to DFA

Given any *NFA* we need to construct as equivalent *DFA* i.e. the *DFA* need to simulate the behavior of the *NFA*. For this, the *DFA* have to keep track of all the states where the NFA could be in at every step during processing a given input string.

There are 2^npossible subsets of states for any *NFA* with *n* states. Every subset corresponds to one of the possibilities that the equivalent *DFA* must keep track of. Thus, the equivalent *DFA* will have 2^n states.

Now, given any *NFA* with ε-transition, we can first construct an equivalent *NFA* without ε-transitionand then use the above construction processto construct an equivalent *DFA*, thus, proving the equivalence of *NFA*s and *DFA*s.

It is also possible to construct an equivalent*DFA*directly from any given *NFA* with ε-transitionby integrating the concept of ε-closure in the above construction.

Recall that, for any $R \subseteq Q$

ε-closures(R)= {p $\in$ Q | p can be reached from any q $\in$ R by following zero or more ε-moves}

In the equivalent *DFA*, at every step, we need to modify the transition function δ^D to keep track of all the states where the *NFA* can go on ε-

transitions. This is done by replacing δ(q,a) by ε-closure(δ(q,a)) ,i.e.we now compute $\delta^D(q^D, a)$ at every step as follows:

$\delta^D(q^D, a) = \{q \in Q \mid q \in \varepsilon\text{-closure}(\delta(q^D, a))\}$

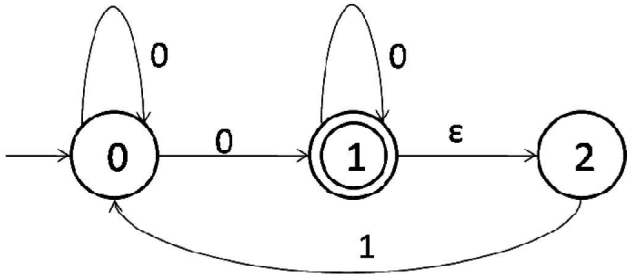
Besides this, the initial state of the *DFA D* has to be modified to keep track of all the states that can be reached from the initial state of *NFA* on zero or more ε-transitions. This can be done by changing the initial state q_0^D to ε-closure (q_0^D). It is clear that, at every step in the processing of an input string by the *DFA D*, it enters a state that corresponds to the subset of states that the *NFA N* could be in at that particular point. This has been proved in the constructions of an equivalent NFA for any ε-*NFA*

If the number of states in the *NFA* is *n*, then there are 2^nstates in the *DFA*. That is, each state in the *DFA* is a subset of state of the *NFA*.

But,it is important to note that most of these 2^n states are inaccessible from the start state and hence can be removed from the *DFA* without changing the accepted language. Thus, in fact, the number of states in the equivalent *DFA* would be much less than 2^n.

Example: Consider the NFA given below.

	0	1	E
→q_0	{q_0, q_1}	Φ	Φ
F q_1	{q_1}	Φ	{q_2}
q_2	Φ	Φ	{q_0}



Since there are 3 states in the NFA, there will be $2^3 = 8$ states (representing all possible subset of states) in the equivalent *DFA*. The transition table of the *DFA* constructed by using the subset constructions process is produced here.

	0	1
Φ	Φ	Φ
→q_0	{q_0, q_1, q_2}	Φ
F {q_1}	{q_1, q_2}	{q_0}
{q_2}	Φ	{q_0}
F {q_0, q_1}	{q_0, q_1, q_2}	{q_0}
{q_1, q_2}	{q_0, q_1, q_2}	{q_0}
F {q_1, q_2}	{q_1, q_2}	{q_0}
F {q_0, q_1, q_2}	{q_0, q_1, q_2}	{q_0}

The start state of the *DFA* is ε- closures(q_0) = {q_0}

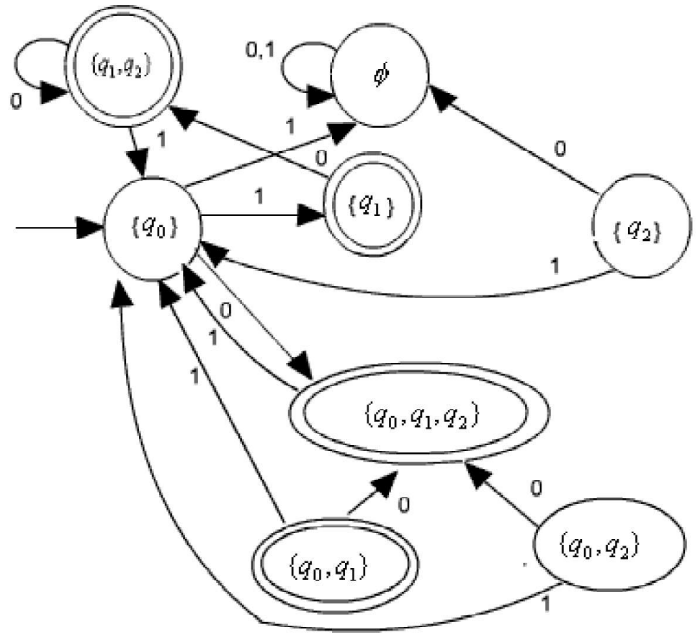
The final states are all those subsets that contains q_1 (since $q_1 \in F$ in the *NFA*).

Let us compute one entry

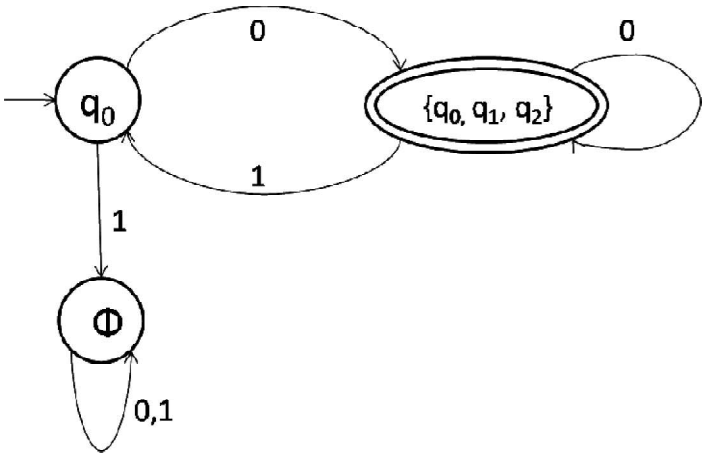
$$\delta^D(q_o, 0) = \varepsilon\text{-closure}(\delta(q_o, 0))$$
$$= \varepsilon\text{-closure}(\delta(q_o, q_1))$$
$$= \{q_0, q_1, q_2\}$$

Similarly, all other transitions can be computed.

Corresponding transition fig. for the DFA is shown as



Note that states $\{q_1\}$, $\{q_2\}$, $\{q_1, q_2\}$, $\{q_0, q_2\}$, and $\{q_0, q_1\}$are not accessible and hence can be removed. This gives us the following simplified *DFA* with only 3 states.



It is interesting to note that we can avoid encountering all those inaccessible or unnecessary states in the equivalent *DFA* by performing the following two steps inductively.

1. If q_0 is the start state of the NFA, then make ε-closure (q_0) the start state of the equivalent *DFA*. This is definitely the only accessible state.
2. If we have already computed a set of δ states which are accessible. Then for all a ∈ Σcompute $\delta^D(S, a)$ because these set of states will also be accessible.

Following these steps in the above example, we get the transition table given below

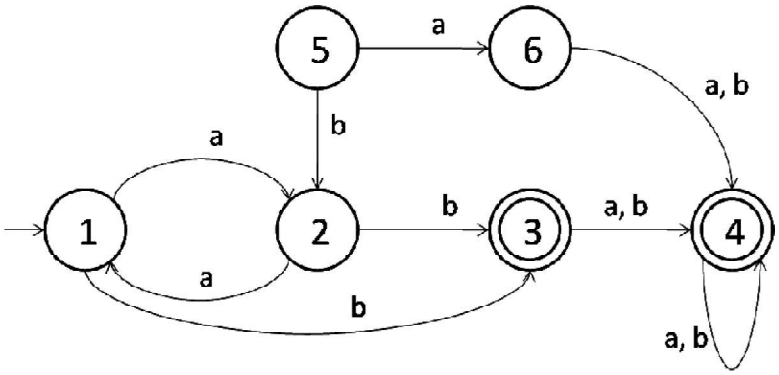
	0	1
$\rightarrow q_0$	$\{q_0, q_1, q_2\}$	Φ
F $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

11.6 MINIMIZATION OF FINITE AUTOMATA

For any regular language *L,* it may be possible to design an alternative DFAs to accept *L*. Given two DFAs accepting the same language *L*, one may ask, which one is simpler? Here the obvious answer

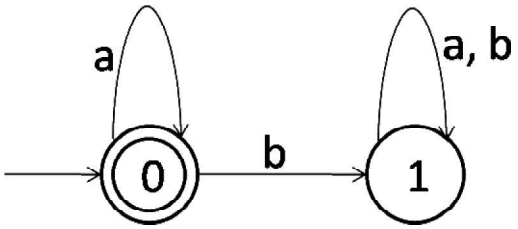
is that DFA with less number of states would be simpler than the other. So, now the question is whether can a given DFA further be simplified without affecting the language it accepts i.e. can we reduce the number of states accepting the same language.

Consider the following DFA M_1,



The above DFA accepts the language generated by the regular expression $a^*b(a+b)^*$.

The same language is accepted by the following simpler DFA M_2 as well.



So, for any regular language *L,* there may be a unique minimal state DFA, the uniqueness is depending on isomorphism.Isomorphism will be defined in next section.

For any given DFA *M* accepting *L* we can construct the minimal state DFA accepting L by using an algorithm which uses following steps.

- First, remove all the states (of the given DFA *M*) which are not accessible from the start state i.e. state *P* for which there is no string $x \in \Sigma^*$ such that $\delta^*(q_0, x) = p$. Removing these states, clearly, will not change the language accepted by the DFA.
- Second, remove all the trap states, i.e. all states *P* from which there is no transition out from the state.

- Finally, merge all states which are "equivalent" or "indistinguishable". We will formally define what is meant by equivalent or indistinguishable states;at this point we assume that merging these states would not change the accepted language.

Inaccessible states can easily be found out by using simple depth first search. Removing trap states are also simple.

In the above example, states 5 and 6 are inaccessible and hence can be removed; states 1 and 2 are equivalent and can be merged. Similarly, states 3 & 4 are also equivalent and can be merged together to have the minimal DFA M_2 as produced above.

To construct the minimal DFA, we need to see how to find out indistinguishable or equivalent states for merging.

DFA Isomorphism:

Two DFAs are said to be isomorphism if they are identical. Formally, DFA isomorphisms are defined as follows.

Definition: Two DFAs $M_{1-} = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_{2-} = (Q_2, \Sigma, \delta_2, q_2, F_2)$ are isomorphic if there is a bijection $f: Q_1 \rightarrow Q_2$ such that the following hold.

1. $f(q_1) = q_2$
2. for all $q \in Q_1$, $q \ F_1$, iff $f(q) = F_2$
3. for all $q \in Q_1$, for all $a \in \Sigma$,

   $f(\delta_1(q, a)) = \delta_2(f(q), a)$

Theorem: For any regular language *L,* there is a unique DFA that has a minimum number of states. In fact, the minimum DFA is the same as the one that has the equivalence classes of $\equiv_L$ (as defined in the context of Myhill-Nerode Theorem).

Proof: Let $M_{L-} = (Q_L, \Sigma, \delta_L, q_L, F_L)$ be the DFA which states are equivalence classes of $\equiv_L$. Let $M = (Q, \Sigma, \delta, q_0, F)$ be any other DFA recognizing *L*. We have already shown that

- $\equiv_M$ is a right invariant equivalence relation of finite index such that L is the union of some of its equivalence classes.
- $\equiv_M$ is a refinement of $\equiv_L$.

- This implies, the number of equivalence classes of $\equiv_M$ (which is equal to the number of states in *M*) must be greater than or equal to the number of equivalence classes of $\equiv_L$ (which is equal to the number of states in M_L, by construction).
- That is $|Q| \geq |Q_L|$
- If $|Q| > |Q_L|$, then we are done, i.e. M_L is the minimum state DFA for *L*.
- If $|Q| = |Q_L|$, then to prove the theorem we need to show that DFAs M_L and *M* are isomorphism.

Showing that M_L and M are isomorphic

To show that M_L and *M* are isomorphic, we have to define a bijection $f: Q_L \rightarrow Q$ that satisfies all the three conditions given in the definition of DF**A** isomorphism.

- Recall that the states of M_L are $[x_1]$, $[x_2]$, …, $[x_k]$ where x_1, x_2, …, x_k are the representatives of each *k*-equivalence classes of $_L$.
- Let us define $f: Q_L \rightarrow Q$ as follows

  $f([x_i]) = \delta(q_0, x_i)$

  That is, *f* maps state $[x_i]$ of M_L to the state in M which can be arrived at processing the string x_i from the start state of *M*. we know that for all $x_i \in \Sigma^*$, $\delta(q_0, x_i)$. Hence *f* is well-defined.
- *f* is onto since $|Q| = |Q_L|$
- To show that f is one-to-one, we need to show that for all $p,q \in Q_L$ if $f(p)=f(q)$ then $p = q$. That means, we need to show that all $x, y \in \Sigma^*$ if $f([x]) = f([y])$, then $x \equiv_L y$. (since x_1, x_2, …, x_k are the representative of different equivalence classes of $_L$, this proves that *f* is one-to-one).

Let $f([x]) = f([y]) = p$ Q .

Then$\delta(q_0, x) = \delta(q_0, y) = p$

Therefore, $\delta(q_0, xz) = \delta(q_0, yz) = \delta(p, z)$ for any $z \in \Sigma^*$.

Hence, by definition of $\equiv_L$,

$xz \in L$ *iff* $yz \in L$ *or* $x \equiv_L y$.

This shows that *f* is a bijection.

We now show in the following that it satisfies all the three conditions.

1. Note that, since *f* is a bijection, $x \equiv_L y \Rightarrow f([x]) = f([y])$. Also note that $q_{0LL} \equiv_L [\varepsilon]$. Hence, $f(q_{0L}) = f([\varepsilon]) = \delta(q_0, \varepsilon) = q_0$. Therefore, the initial state $[\varepsilon]$ of M_2 is mapped to the initial state q_0 of *M* thus satisfying the first condition.
2. We know that for any $x_i \in \Sigma^*$

   $x_i \in F$

   $\Leftrightarrow x_i \in L$ (by definition)

   $\Leftrightarrow \delta(q_0, x_i) \in F$ (Since *M* accepts *L*)

   $\Leftrightarrow f([x_i]) \in F$ (by definition of *f*)

   Thus, final state of M_L are mapped to final state of *M* , satisfying the second condition.
3. Observe that, for any $x_i \in \Sigma^*$, $a \in \Sigma$

   $f([x_i], a) = \delta(\delta(q_0, x_i), a)$ (by definition of *f*)

   $= \delta(q_0, x_i a)$

   $= f([x_i a])$ (by definition of *f*)

   $= f(\delta_L(x_i a))$ (since $[x_i a] \equiv_L \delta(x_i a)$)

This satisfies the third condition of the definition, thus proving that M_L and *M* are isomorphic. This also completes the prove that M_L is the minimal state DFA for *L* since, now, $|Q| \geq |Q_L|$, (i.e. the number of state *Q* in any arbitrary DFA *M* accepting the language *L* must be greater than or equal to the number of states Q_L of the DFA M_L that has as states the equivalence classes of $\equiv_L$.)

The minimal DFA

Given DFA *M* accepting a regular language *L*, we observe that

- M_L is the minimal state DFA accepting *L*.
- $\equiv_M$ refines $\equiv_L$, implying each equivalence classes of $\equiv_L$ is the union of some equivalence classes of $\equiv_M$.
- Hence, each state of M_L (which correspond to the equivalence class of $\equiv_L$) can be obtained by merging states of *M*. (which correspond to equivalence classes of $\equiv_M$)

Now, We are going to devise an algorithm for doing this until no more merging is possible. Following observations are seen-

- It is not possible to merge an accept state *p* and a non-accepting state *q*. Because if $p=\delta^*(q_0, x) \in F$ and $q=\delta^*(q_0, y) \notin F$ for some $x,y \in \Sigma^*$, then *x* must be accepted and *y* must be rejected after merging *p* and *q*. But, now, the resulting merged state can neither be considered as an accept state nor as a non-accepting one.
- If p and q are merged, then we need to merge δ(p, a) and δ(q, a), for every $a \in \Sigma$, as well, to maintain determinism.

From the above two observations we conclude that states p and q cannot be merged if $\delta^*(p, x) \in F$ and $\delta^*(q, x)$ F for some $x \in \Sigma^*$.

Using this concept, we now define an indistinguishability relation as follows:

Definition: States *p* and *q* are indistinguishable if for all

$x \in \Sigma^*$, $\delta^*(p, x) \in F$ iff $\delta^*(q, x) \in F$

and is denoted as $p \equiv q$. It is easy to see that indistinguishability is an equivalence relation.

In other words, we say that states *p* and *q* are "distinguishable" if $x \in \Sigma^*$ such that $\delta^*(p, x) \in F$ and $\delta^*(q, x) \notin F$ and is denoted as $p \not\equiv q$.

States *p* and *q* of a DFA *M* accepting a language *L* can be merged safely (without changing the accepted language *L*) if $p \equiv q$ i.e.if *p* and *q* are indistinguishable. we can prove this by showing that when *p* and *q* are merged,then they correspond to the same state in M_L.

Formally, $p \equiv q$ iff $x,y \in \Sigma^*$, $\delta^*(q_0, x) = p$ and $\delta^*(q_0, x) = q \Rightarrow x \equiv_L y$.

A Minimization Algorithm:

The algorithm to construct the minimal state DFA from any given DFA accepting *L* by merging states inductively defined as follows-

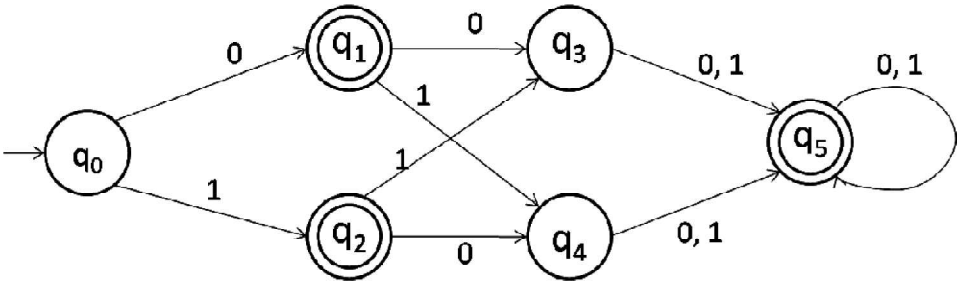
The algorithm assume that all states are reachable from the start state i.e. there is no inaccessible states. The algorithm keeps on marking pairs of states(*p*, *q*)as soon as it determines that *p* and *q* are distinguishable i.e. p q . The pairs are, of course, unordered i.e. pairs (*p*, *q*) and (*q* , *p*) are considered to be identical. The steps of the algorithm are given below.

1. For every $p, q \in Q$, initially unmark all pairs (p, q).
2. If $p \in F$ and $q \notin F$ (or vice versa) then mark (p, q).
3. Repeat the following step until no more changes occur**:** If there exists an unmarked pair (p, q) such that $(\delta(p,a), \delta(q,a))$ is marked for some $a \in \Sigma$, then mark (p, q).
4. $p \equiv q$ iff (p, q) is unmarked.

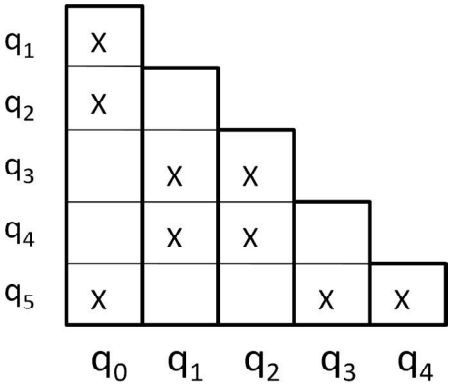
The algorithm correctly computes all pairs of states that can be distinguished i.e. unmarked.

It is easy to show (by induction) that the pair (p, q) is marked by the above algorithm iff $x \in \Sigma^*$ such that $\delta^*(p, x) \in F$ and $\delta^*(q, x) \notin F$(or vice versa) i.e. if $p \not\equiv q$.

Example : Let us minimize the DFA given below



We execute the algorithm and mark a pair by putting an X on the table as shown in following figure. (Note that the table is a diagonal one having $\binom{n}{2}$ entries for a DFA having n states.)



Initially, all cells are unmarked. (i.e. at step 1 of the algorithm). After step 2, all cells representing pairs of states of which one is accepting and the other is non-accepting are marked by putting an X. The table above shows the status after this step.

In step 3, we consider all unmarked pairs one by one. Considering the unmarked pair (q_0, q_3), we find that q_0 & q_3 go to q_1 and q_5, respectively, on input 0. The notation $(q_0,q_3) \xrightarrow{0} (q_1,q_5)$ is used to indicate this. Since the pair (q_1,q_5) is not marked, (q_0,q_3) cannot be marked at this point. Again, we see that, $(q_0,q_3) \xrightarrow{1} (q_2,q_5)$ and(q_2,q_5) is unmarked. Hence, we cannot mark $(q_0,q_3)$and since we have considered all input symbols (0 & 1) we need to examine other unmarked pairs. The observations and actions are shown below.

- (q_0,q_4) (q_1,q_5)
- (q_0,q_4) (q_2,q_5) cannot mark (q_0,q_4) since (q_1,q_5) & (q_2,q_5) are unmarked.
- (q_1,q_2) (q_3,q_4)
- (q_1,q_2) (q_3,q_4) cannot mark (q_1,q_2) since (q_3,q_4) is unmarked.
- (q_1,q_5) (q_3,q_5), (q_1,q_5) is marked since (q_3,q_5) is already marked.
- (q_2,q_5) (q_4,q_5), (q_2,q_5) is marked since (q_3,q_5) is already marked.
- (q_3,q_4) (q_5,q_5), (q_5,q_5) is never marked since it is not in the table hence (q_3,q_4) is not marked.
- (q_3,q_4) (q_5,q_5)
- $(q_3,q_4) \xrightarrow{0} (q_5,q_5)$, (q_5,q_5) is never marked since it is not in the table hence (q_3,q_4) is not marked.
- $(q_3,q_4) \xrightarrow{1} (q_5,q_5)$

The resulting table after this pass is given below.

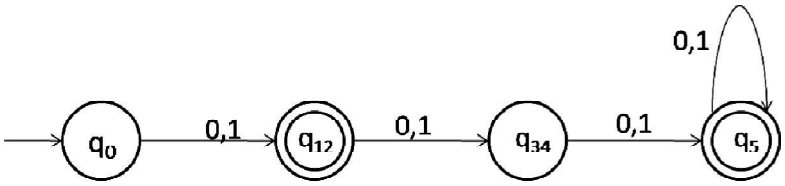
q_1	X				
q_2	X				
q_3		X	X		
q_4		X	X		
q_5	X	X	X	X	X
	q_0	q_1	q_2	q_3	q_4

In the next pass, we find that $(q_0,q_3) \xrightarrow{0} (q_1,q_5)$ and (q_1,q_5) is marked in the previous pass. Hence, (q_0,q_3) can be marked now.

Similarly, $(q_0,q_4) \xrightarrow{1} (q_2,q_5)$ and hence (q_0,q_4) can be marked since (q_2,q_5) has been marked in the previous pass. Other pairs cannot be marked and the resulting table is shown below. By executing step 3 again we observe that no more pairs can be marked and hence the algorithm stops with this table as the final result.

The unmarked pairs left in the table after execution of the algorithm are (q_1,q_2) and (q_3,q_4) implying $q_1 \equiv q_2$ and $q_3\ q_4$. Now, we merge q_1& $q_2$and q_3& q_4 to have new states q_{12}& q_{34}, respectively.

Transitions are adjusted appropriately to obtain the following minimal DFA.



q_{12} is a final state, since both q_1& q_2 were final states. Similarly, $q_{34}$is a non-final state.

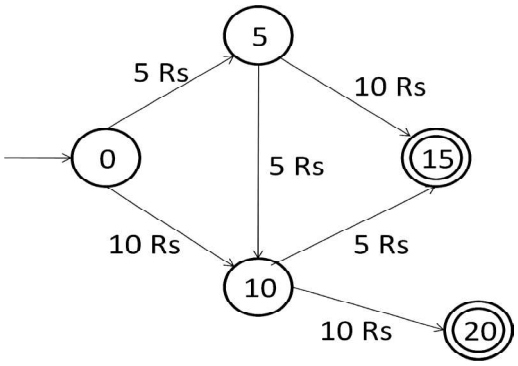
q_0 goes to q_{12} on input 0 and 1, since q_0 go to q_1 and q_2 respectively on 0 and 1.Similar, justifications suffice for other adjusted transitions.

11.7 APPLICATION OF FINITE AUTOMATA

Soft drink vending machine

Let us consider the operation of a soft drink vending machine which charges 15 rupees for a can. The machine initially waits for a customer to come and put some coins, that is, waiting-for-customer state. For simplicity let us assume that only 5 Rs and 10 Rs coins are used. When a customer comes and puts in the first coin, say 5 Rs, machine no longer in the waiting-for-customer state. Now it has received 5 Rs and waited for more coins to come. So, we might say it in the 5 Rs state. If the customer puts 10 Rs, then it received 15 Rs and wait for the customer to select a soft drink. So, it is in another state, say 15-Rs state. When the customer selects a soft drink, machine delivers the soft drink. After that it returns back to its initial state until another coin is put in to start the process. The states and the transitions between them of this vending machine can be represented with

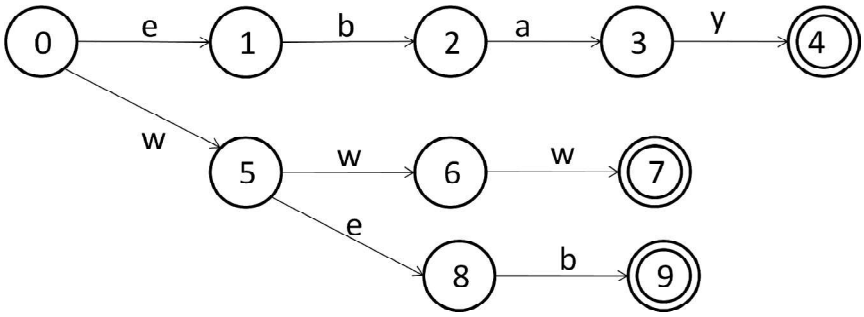
the diagram below. In the figure, circles represent states and arrows state transitions.



Nondeterministic finite automata for text search

Suppose we are given a set of words, which we shall call keywords, and we need to find out whether the input word is a keyword or not. For that we can define a NFA which have an initial state q_0 and it reads the keywords symbol by symbol. On q_0 if it receives a keyword, first match the first symbol of the keyword with the available outgoing transitions from q_0. If it finds so, it proceeds and reads the next symbol of the keyword and so on. After reading all the symbols from the keywords if it reaches a final state, then the NFA will accept the keyword otherwise reject it.

For example, if we want to design a NFA to accept the following keywords web, www, ebay then the NFA will look like



Number Recognizer

Our third example is a system that recognizes numbers with or without a sign such as 5.378, -15, +213.8 etc. One such system initially waits for the first symbol to come in. If the first symbol is a sign, then it goes into a state, denote it by G, that indicates that a sign has been received.

If the first digit is received before a decimal point, regardless of whether a sign has been read or not, it goes into a state, denote it by D, that indicates a digit has been read before a decimal point.

If a decimal point is received before a digit, then it goes into a state, denote it by P, that indicates that a decimal point has been read.

If a decimal point has been read (i.e. in state P), then it must receive at least one digit after that. After one digit it can continue receiving digits. Therefore, from state P it goes to another state, denoted by Q, after reading a digit and stays there as long as digits are read. This Q is an accepting state.

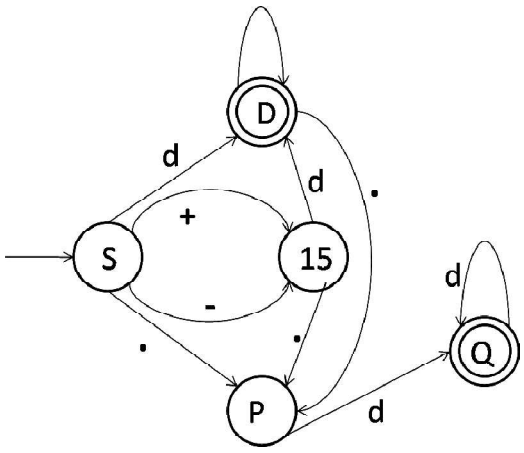
On the other hand, if a digit has been read before a decimal point, i.e. it is in state D, then it can continue receiving digits and stay in D. D is another accepting state. If a decimal point is read while in D, then it goes to state P indicating that a decimal point has been read.

This system can also be described by a regular expression.

Since these numbers are represented by strings consisting of a possible sign, followed by zero or more digits, followed by a possible decimal point, followed by one or more digits, they can be represented by the following regular expression:

$(s_+ + s_- + \varepsilon)\,(d^+.d^+ + d^+ + .d^+)$,

where s_+ and s_- represent the positive and negative signs, respectively and $d \in \{ 0 , 1 , 2 , \ldots , 9 \}$. This system can be modeled by the following finite automaton:





CHECK YOUR PROGRESS

Q 2: State True or False

a. δ(q, ε) always contains q

b. δ*(q, ε) does not always contain q

c. In the recursive definition of ε(S) the basis is a empty set

d. In the minimization of algorithm initially the states are grouped into two accepting and non-accepting states

e. For each regular language there is a unique DFA with smallest number of states

11.8 FINITE AUTOMATA WITH ε-TRANSITION

An epsilon transition also called as epsilon move allow automaton to change its state without consuming an input symbol. It is present in almost all kinds of nondeterministic automaton.

Definition of nondeterministic finite automaton with ε-Transitions

Let Q be a finite set of states and let Σ be a finite set of symbols. Also let δ be the transaction function from Q x Σ $\bigcup$ {ε} to 2^Q, let q_0 is the initial state belongs to Q and let F is a set of final states and a subset of Q. Then a nondeterministic finite automaton with ε-Transitions is a 5-tuple combination

$$< Q , \Sigma , q_0 , \delta , A >$$

Notes on the definition

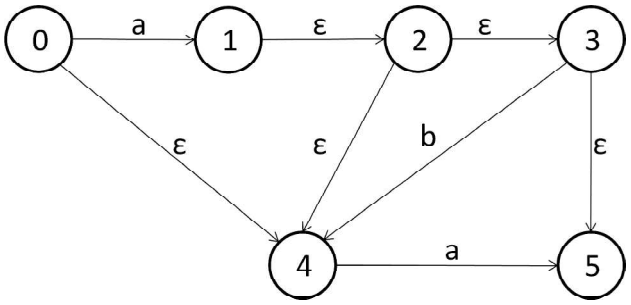
1. A transition on reading ε means that the NFA-ε makes the transition without reading any symbol in the input. Thus, the tape head does not move when ε is read.
2. Note that any NFA is also an NFA-ε.

Example of NFA-ε

Q = { 0, 1, 2, 3, 4, 5 }, Σ = { a, b }, F = Φ , the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State (δ (q, a))
0	a	{ 1 }
0	ε	{ 4 }
1	ε	{ 2 }
2	ε	{ 3, 4 }
3	ε	{ 5 }
3	b	{ 4 }
4	a	{ 5 }

Here the transitions to Φ are omitted from the table. A state transition diagram for this finite automaton is given below.



When a symbol 'a' is read at the initial state 0, for example, it can move to any of the states other than 0. For once you are in state 1, for example, you can go to state 2, 3, 4 and 5 without reading any symbol on the tape. If you read string ab, then you come to state 4. For though you go to states 1, 2, 3, 4 and 5 by reading a, there are no transitions on reading b except from state 3. Thus 4 is the only state you can go to from the initial state by reading ab.

δ^* for NFA - ε

To formally define δ^* for NFA-ε, we start with the concept of ε-closure for a state which is the set of states reachable from the state without reading any symbol. Using that concept, we define δ^* and then strings and languages accepted by NFA-ε.

Definition of ε-closure

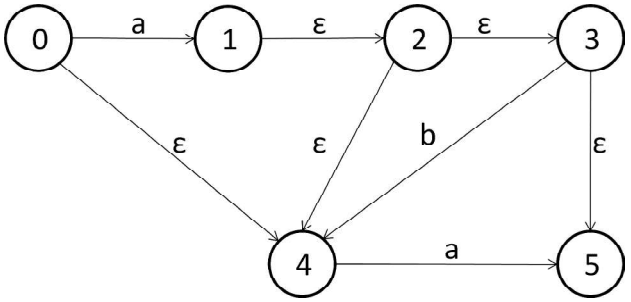
Let < Q , Σ , q_0 , δ , A > be an NFA-ε . Let us denote the ε-closure of a set S of states of Q by ε(S). Then ε(S) is defined recursively as follows:

Basis Clause: S $\subseteq$ ε(S)

Inductive Clause: For any state q of Q, if q $\in$ ε(S), then

δ(q , ε) ε(S) .

External Clause: Nothing is in ε(S) unless it is obtained by the above two clauses.



For the NFA-ε of the above figure, ε ({2}) is obtained as follows:

First { 2 } $\subseteq$ ε ({2}), that is, 2 $\in$ ε ({2}). Then since 2 ε ({2}), by the Inductive Clause,

δ (2, ε) ε ({2}).

Since δ(2, ε) = {3, 4}, we now have {2, 3, 4} ε ({2}).

Since 3 and 4 are added to ε({2}) , δ(3, ε) = {5} and δ(4, ε) = Φ must be included in ε ({2}).

Thus, now {2, 3, 4, 5} ε ({2}).

Though 5 has become a member of the closure, since δ(5, ε) is empty, no new members are added to ε({2}). Since δ(q, ε) has been examined for all the states currently in ε({2}) and no more elements are added to it, this process of generating the ε-closure terminates and ε({2}) = {2, 3, 4, 5} is obtained.

As we can see from the example, ε (S) is the set of states that can be reached from the states of S by traversing any number of ε arcs. That is, it is the set of states that can be reached from the states of S without reading any symbols in Σ.

Now with this ε-closure, we can define δ^* recursively as follows:

As in the cases of DFA and NFA, δ^* gives the result of applying the transition function δrepeatedly as dictated by the given string.

Definition of δ^*

δ^* is going to be defined recursively.

Let < Q , Σ, q_0 , δ , F > be an NFA-ε.

Basis Clause: For any state q of Q,

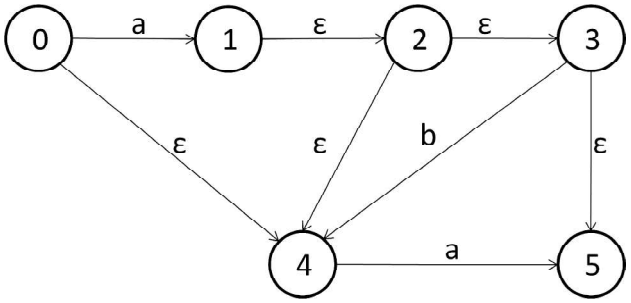
δ^* (q , ε) = ε ({q}) .

Inductive Clause: For any state q, a string y in Σ^* and a symbol 'a' in Σ,

δ^* (q , ya) = $\varepsilon\left(\bigcup_{p\in\delta^*(q,y)} \delta(p,a)\right)$

What the Inductive Clause means that δ^* (q , ya) is obtained by first finding the states that can be reached from q by reading y (δ^* (q , y)), then from each of those states p by reading 'a' (i.e. by finding δ (p , a)), and then by reading ε's (i.e. by taking the ε closure of the δ(p , a)'s).

Example: For the NFA-ε of the following figure, δ^* (0, ab) can be obtained as below:



First let us compute δ^* (0 , a)

For that we need ε({0}).

Since it is the set of states reached by traversing the ε arcs from state 0, ε({0}) = {0, 3, 4}.

Next from each of the states in ε({0}) we read symbol a and move to another state (i.e. apply δ). They are

δ (0, a) = {1},

δ(3, a) = δ(4, a) = {5}.

Hence $\bigcup_{p\in\delta^*(q,y)} \delta(p,a)$ = {1, 5} for q = 0.

We then traverse the ε arcs from {1, 5} to get to the states in $\delta^*(0, a)$. Since ε({1}) = {1, 2, 3} and ε({5}) = {5},

$\delta^*(0, a) = \{1, 2, 3, 5\}$.

Then to find $\delta^*(0, ab)$ read b from each of the states in $\delta^*(0, a)$ and then take the ε arcs from there.

Now, δ(1, b), δ(3, b) and δ(5, b) are empty sets, and δ(2, b) = {4}. Thus Since ε({4}) = {3, 4}, $\delta^*(0, ab) = \{3, 4\}$.

A string x is accepted by an NFA-ε $< Q, \Sigma, q_0, \delta, F >$ if and only if $\delta^*(q_0 , x)$ contains at least one accepting state.

The language accepted by an NFA- ε $< Q, \Sigma, q_0, \delta, F >$ is the set of strings accepted by the NFA-ε.



CHECK YOUR PROGRESS

Q 3: State True or False:

a. In an NFA-ε, all states have the same number of transitions.

b. An NFA-ε cannot have more than one accepting states.

c. An NFA-ε has finite number of states.

d. An NFA-ε can modify its input.

e. When a is read at q, NFA-ε goes to a state in δ(q,a).



11.9 LET US SUM UP

- A DFA (Deterministic Finite State Automaton) is a simple machine that reads an input string, one symbol at a time and then, after the input has been completely read, decides whether to accept or reject the input.
- The language accepted by a DFA M =(Q, Σ, δ, q_0, F) is the set of all strings on Σ accepted by M, in formal notation

  $L(M) = \{\omega \in \Sigma^* \mid \delta^* (q_0, \omega) \in F\}$

- An NFA is defined in the same way as the DFA but it can have multiple next states and it can also move to next state without reading any symbol from input.
- NFA-ε is an NFA with ε moves. In NFA-ε, the automata can move to the next state without reading the next symbol.
- ε-closure of a state is the set of states which are reachable from the states by ε-moves without reading the input.
- We can convert the different automata from one from to other by following certain number of steps.
- For every DFA, we can design a DFA with minimized states.
- Every DFA is an NFA but Every NFA is Not a DFA.



11.10 FURTHER READING

1. Peter Linz, “An Introduction to Formal Language and Automata”, 4th Edition, Narosa Publishing house , 2006.
2. M.Sipser; Introduction to the Theory of Computation; Singapore: Brooks/Cole, Thomson Learning, 1997.
3. John.C.martin, “Introduction to the Languages and the Theory of Computation”,Third edition, Tata McGrawHill, 2003.
4. K.Krithivasan and R.Rama; Introduction to Formal Languages, Automata Theory and Computation; Pearson Education, 2009.
5. J.E.Hopcroft, R.Motwani and J.D.Ullman , “Introduction to Automata Theory Languages and computation”, Pearson Education Asia , 2001.



11.11 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1:

a. True

b. False

c. True

d. True

e. False

f. False

g. False

h. True

i. True

j. True

Ans to Q No 2:

a. False

b. False

c. False

d. True

e. True

Ans to Q No 3:

a. False

b. True

c. True

d. False

e. True



11.12 MODEL QUESTIONS

Q 1: Design DFA to recognize the following set of strings
abb, abaa, ab^*, a^*b assuming that Σ = {a,b}

Q 2: Design NFA to recognize the following set of strings
abb, abaa, ab^*, a^*b assuming that Σ = {a,b}

Q 3: Explain NFA with ε transition.

Q 4: Explain the applications of Finite Automata.

Q 5: Explain how we can convert an NFA to DFA.

Q 6: Explain how we can convert an NFA-ε to DFA.

Q 7: For the following transition table, construct the minimum state equivalent DFA

	0	1
→A	B	A
B	A	C
C	D	B
*D	D	A
E	D	F
F	G	E
G	F	G
H	G	D

*** ***** ***

UNIT 12: FORMAL LANGUAGES

UNIT STRUCTURE

12.1 Learning Objectives

12.2 Introduction

12.3 Basic Definition

12.4 Definition of a Grammar

12.5 Derivation and language generated by a grammar

12.6 Chomsky Classification of Languages

12.7 Let us sum up

12.8 Further reading

12.9 Model Questions

12.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- know about formal language and its characteristics.
- learn how to define a formal language
- know about different types of formal language
- learn about grammar for a language
- learn about language generated by a grammar
- know about Chomsky classification of language.

12.2 INTRODUCTION

In the previous unit we have learned about finite automata which is a mathematical model for string acceptor. Based on different patterns for a set of strings, this model can be designed. When we define about a formal language, there should be a set of entities which can be considered. These entities are like alphabet, strings, grammar etc. In this unit we will learn about formal language including definition, characteristics, different types of formal language, grammar for a language. We will also discuss about the similarity between programming language and natural language like English.

12.3 BASIC DEFINITION

Language is used to communicate between two or more ends. There are several languages in the world through which living beings are interacting. Some languages are well documented, some are not. In this course, we will discuss the basic of well documented language which is called as formal language.

A formal language can be described with different terminologies like Symbol, Alphabet, Strings, Grammar.When a mother use to make her kid to learn English language, she starts with A for “Apple”, B for “Ball” etc. which is basically some Symbols.

Symbol represents a basic entity of a language.

A finite set of symbols is called as Alphabet. Greek letters “ and Γ are used to designate alphabet like “={a,b,c,....,z}.

After learning Alphabet, kid starts learning with some words like “Boy”, “School” etc. Basically, these are called Words of the language. Every word has some meaning as per the language. In respect to formal language this is called String.

String is considered as a finite sequence of symbols over an alphabet of a language.

Then mother tries to make her kid to learn sentences like “ I am a girl”, “I drink milk”, etc.

Sentence is considered as a finite sequence of words (strings) of a language.

The sequence of strings for an instruction is validated based on grammar of the language. Basically, a grammar provides the syntax of instruction for a language. In the other way, a grammar is used to produce a set of strings (instructions) of a language.

A grammar G is considered as collection of four tuples i.e. <N, T, P, S>

where,

N refers to a finite set of non-terminal symbols

T is a finite set of terminal symbols

P means a finite subset of $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$

and an element $(\alpha, \beta) \in P$ is written as $\alpha \rightarrow \beta$

S is a distinguished symbol in N and is called the start symbol

12.4 DEFINITION OF A GRAMMAR

In the previous section, we have learned that a grammar provides the syntax of sentence for a language. A grammar is applied to check the correct sequence for a string which is valid for a language. A grammar provides a set of rules which are considered to make a correct instruction for the language. Thus, to define a grammar, production rules are framed which are involved with some grammar symbols. There are two types of grammar symbols like terminals and non-terminals. Among terminals all symbols in an alphabet of the language should be present. Non-terminals are considered as variables so that these can be applied for different syntax to produce different sentences/ instructions.

So, formally a grammar G is considered as collection of four tuples i.e. <N, T, P, S>

where,

N is a finite set of non-terminal symbols

T is a finite set of terminal symbols

P is a finite subset of $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$

and an element $(\alpha, \beta) \in P$ is written as $\alpha \rightarrow \beta$

S is a distinguished symbol in N and is called the start symbol

Now, we can consider the following grammar production rules (P).

E -> E + T | T

T -> T * F | F

F -> (E) | **id**

Traditionally, all characters in upper case are considered as non-terminals and rest are considered as terminals. So, for this example

N = { E,T, F }

T = { +, *, (,), id }

and E is considered as starting symbol.

Now grammar can be classified based on different characteristics of production rules (P).

There are four types of grammar which are considered as Chomsky Hierarchy (classification of grammars). They are

1. Regular Grammar
2. Context-free Grammar
3. Context-sensitive Grammar
4. Unrestricted Grammar

A grammar is called regular if it is right-linear, where each production in P has the form, $A \to wB$ or $A \to w$. Here, A and B are non-terminals and w is a terminal. Similarly, it is also left-linear.

Context-free grammar (CFG) is a grammar where each production in P is of the form $A \to \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$

Context-sensitivegrammar (CSG) is a grammar where each production in P is of the form $\alpha \to \beta$, where there is a restriction regarding the length such that $|\alpha| \leq |\beta|$

Unrestricted grammar is a grammar where each production in P is of the form $\alpha \to \beta$, without any restriction regarding the length except $\alpha \neq \varepsilon$

12.5 DERIVATION AND LANGUAGE GENERATED BY A GRAMMAR

So far, we have understood that using a language, a set of strings can be generated. When we consider a string as a sequence of other strings then it will represent a sentence/ instruction. In this section, we will see how these strings can be derived or generated by a grammar of a language.

To derive any string for a language, we need to apply the grammar of that language. A grammar has four tuples such as N, T, P, S where

N is a finite set of non-terminals

T is a finite set of terminals

S is the starting non-terminal and belongs to N

P is a finite set of production rules

Generally, all symbols in the alphabet of a grammar are elements in T.

Now, based on the characteristic of the production rule, grammar will be classified. Here, we will consider Context-free Grammar (CFG) to discuss derivation of strings. A grammar is considered as CFG if all the production rules are in the following form.

A-> x, where

A is considered as head of the production and a non-terminal which belongs to N

X is considered as body of the production and any combination using terminals and non-
terminals of the grammar.

A grammar is used to generate a string by applying the production rules of the grammar in a sequence if the string is valid for the language. The task to find out the correct sequence of the production rules of the grammar is called derivation and the corresponding technique is considered as parsing. To derive any valid string of a language, generally a production rule is considered first where head of the production must be starting non-terminal of the grammar. So, to derive a string we start with S non-terminal and find out possible sequence of production rules so that finally the desired string can be produced. In the process all/some production rules can be used multiple times. Now, try to understand derivation by looking the following example.

The grammar is like

E -> E + E

E -> E * E

E -> -E

E -> (E)

E ->**id**

Now, the given string is–(id+id)**.** We have to check that the string has valid syntax as per the above grammar or not. This can be performed using derivation. If the string can be derived from the grammar, then the string is valid otherwise not.

The given string can be derived as

E => -E (by applying E => -E)

=> -(E) (by applying E -> (E))

=> -(E+E) (by applying E -> E + E)

=> -(**id**+E) (by applying E ->**id**)

=>-(**id**+**id**) (by applying E ->**id**)

12.6 CHOMSKY CLASSIFICATION OF LANGUAGES

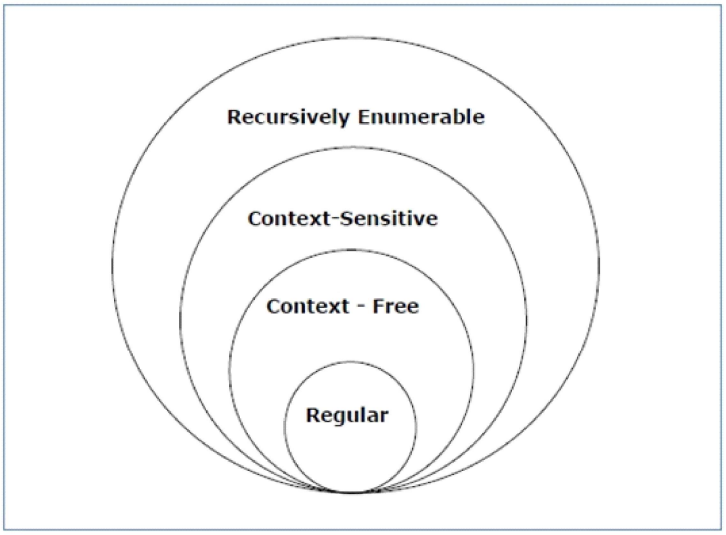
Already we have learned that there are four types of grammar as per following.

1. Regular Grammar
2. Context-free Grammar
3. Context-sensitive Grammar
4. Unrestricted Grammar

When we talk about classification of formal language, this classification will be based on the different types of grammar.

Thus, there are also four types of formal languages as per Chomsky classification of languages. They are

1. Regular language
2. Context-free language
3. Context-sensitive language
4. Recursively enumerable language



There is a relationship between classification of languages and classification of grammars which is represented as following.

Grammar Type	Grammar Accepted	Language Accepted	Automata
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automata
Type 2	Context-free grammar	Context-free language	Pushdown automata
Type 3	Regular grammar	Regular language	Finite state automata



EXERCISE

1. Define a formal language.
2. How do you define a grammar?
3. What is the role of a grammar for a language?
4. Discuss the Chomsky classification of grammar.
5. What is derivation of a string?



CHECK YOUR PROGRESS

- ✓ After reading the unit, you can make summary about formal language and grammar.
- ✓ You can try to find out the grammar for the language which is considered as a set of all strings which start with ‘a’ and end with ‘b’ over an alphabet = {‘a’, ‘b’, ‘c’ }.
- ✓ Also, you can check the efficiency of context-free language regarding derivation of string.



12.7 LET US SUM UP

- Languages are used to communicate between two or more points. So,there should be a proper representation of a formal language.
- Symbols, alphabet, strings, grammar are associated with formal language to provide suitable documentation for the language.
- Alphabet is used to provide all valid strings and grammar is considered to represent the correct syntax of the formal language.



12.8 FURTHER READING

1. J. E. Hopcroft, R. Motwani and J. D. Ullman, "Introduction to Automata Theory, Languages and Computation", 2nd Ed., Pearson Education, 2000.
2. H. R. Lewis and C. H. Papadimitriou, "Elements of the Theory of Computation", Pearson Education Asia, 2001.
3. Peter Linz, "An Introduction to Formal Languages and Automata", Narosa Publisher.



12.9 MODEL QUESTIONS

Q.1: How do you define a language? Explain with an example.

Q.2: What is the role of grammar in programming language?

Q.3: Find out context-free grammar for a language which has all strings having same number of 'a' and 'b' over an alphabet = { 'a', 'b' }

Q.4: Find out context-sensitive grammar for a language which has all strings in the form "wcw" such that w can be any combination of 'a' and 'b' where the alphabet = { 'a', 'b', 'c' }

Q.5: Discuss the Chomsky classification of languages.

*** ***** ***

UNIT 13: REGULAR EXPRESSIONS AND REGULAR GRAMMAR

UNIT STRUCTURE

13.1 Learning Objectives

13.2 Introduction

13.3 Regular expressions

13.4 Regular set and Regular grammar

13.5 Closure properties of Regular languages

13.6 Relationship between Finite automata and Regular Expressions

13.7 Pigeon hole principle

13.8 Applications of Regular expressions

13.9 Let us Sum up

13.10 Further Reading

13.11 Answers to check your progress

13.12 Model questions

13.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- The concept of regular expression
- Definition of regular expression
- The concept of regular sets and grammars
- Regular grammars and Regular languages
- The closure properties of regular languages
- The relation between finite automata and regular expressions
- What is pigeon hole principle and its applicability

13.2 INTRODUCTION

Language can be defined by expressing its grammatical rules through regular expression. The languages which are accepted by a finite automata (FA) machine are represented by regular expression. As the finite automata accepts the regular languages so the regular expressions are

also used to denote regular languages. A regular expression generates a set of strings, known as regular set.

13.3 REGULAR EXPRESSIONS

Regular expressions denote the regular languages in the form of short notations. Regular expression over ..... is defined recursively as follows:

1. Every character that belongs to ...., that is, a$\in$... is a regular expression over .....
2. $\prod$ (empty set) and $\sum$ (empty string of length zero) are also regular expressions.
3. If R_1 and R_2 are regular expressions over ...., then $R_1 + R_2$, $R_1.R_2$, $(R_1)^*$ are regular expressions.

A string is a regular expression if it is a primitive regular expression or it can be derived from the primitive regular expressions by applying a finite number of the operations +, * and concatenation.

Let ..... ={a,b,c} be a set of characters of a language L. Then, the following statements hold.

1. The alphabets a,b and c are regular expressions.
2. As a,b and c are regular expressions, so ab, bc, ac, aa, bb, cc,abc, bac, etc. are also regular expressions.
3. As a,b and c are regular expressions, so are a+b, b+a, b+c,a+c, aa+bb, ac+bb, cc+ab etc. regular expressions.
4. As a,b and c are regular expressions, so a*, b*, c*, (a+b)*, (aa+bc)*, (ab)*, (cc)*, etc. are also regular expressions.

Some identities related to Regular Expressions are given as follows:

Given R, P, L, Q as regular expressions, the following identities hold “

- $\varepsilon^* = \varepsilon$
- $\varepsilon^* = \varepsilon$
- $RR^* = R^*R$
- $R^*R^* = R^*$
- $(R^*)^* = R^*$
- $RR^* = R^*R$
- $(PQ)^*P = P(QP)^*$

- $(a+b)^* = (a^*b^*)^* = (a^*+b^*)^* = (a+b^*)^* = a^*(ba^*)^*$
- $R + \varepsilon = \varepsilon + R = R$ (Union)
- $R\,\varepsilon = \varepsilon\,R = R$ (Concatenation)
- $R + R = R$ (Idempotent law)
- $L(M + N) = LM + LN$ (Left distributive law)
- $(M + N)L = ML + NL$ (Right distributive law)
- $\varepsilon + RR^* = \varepsilon + R^*R = R^*$

Examples:

1. Write a regular expression for the language L={aa,aaaa,aaaaaa,……}

Solution: Here, aa repeats a number of times. $(aa)^*$ means repeatation of aa from zero to infinity. However, aa is also there in L. So, aa needs to be appended to $(aa)^*$. Thus, $aa(aa)^*$ is the regular expression for language L.

2. Write a regular expression for language L over {a,b} such that every string in L begins with aa and ends with bb.

Solution: $aa(a+b)^*bb$

3. Write a regular expression for language L over {0,1} such that every string in L ends with 01.

Solution: $(0+1)^*01$

4. Write a regular expression for language L over {a,b,c} such that every string in L contains a substring cc.

Solution: $(a+b+c)^*cc(a+b+c)^*$

5. Write regular expression for the following strings:
   i) $0^i1^j2^k$ where i, j, k are integers and i, j and k>=1
   ii) $0^i1^j2^k$ where i, j, k are integers and i, j >=1 and k>=0

Solution: i) $00^*11^*22^*$
ii) $00^*11^*2^*$

6. Write a regular expression for language L over {0,1} to generate all possible strings of length 4.

Solution: (0+1)(0+1)(0+1)(0+1)



CHECK YOUR PROGRESS

Q 1: Write a regular expression for language L over {0,1} such that every string in L contains alternate 0's and 1's.

Q 2: Write a regular expression for the following regular sets:

a. Set of all strings over {a,b,c} beginning with a and ending with cc

b. {00,0011,001111, 00111111, …….}

c. Set of all strings over {a,b} containing exactly two a's.

d. Set of all strings over {a,b} containing exactly two a's and two b's

e. Set of all strings over {0,1} containing exactly one 0

f. Set of all strings over {0,1} containing at most one 0

g. Set of all strings over {a,b} starting and ending with a's and with any number of b's in between

h. Set of all strings over {0,1} containing at least two consecutive 0's.

i. Set of all strings over {0,1,2} such that every string contains any number of 0's followed by any number of 1's followed by any number of 2's.

13.4 REGULAR SET AND REGULAR GRAMMAR

A regular expression generates a set of strings known as Regular set. The language corresponding to a regular expression is called a regular language. A regular set or a regular language is a language accepted by some FA (Finite Automata) that can be represented by a regular expression. In simple words, the class of regular sets over ..... is defined as the smallest class containing all finite sets of words over .....

The regular expression a* can generate any number of a's. It can be any number of a, from zero to infinity. The regular set corresponding to this is

{ε, a, aa,aaa,aaaa,aaaa,…..}, where ε is a null string.

The regular expression (a+b) means that the string can be either a or b. thus the corresponding set is {a,b}.

The expression (a+b)c means that the string would start with either a or b and would be always followed by c. Thus, the regular set for this is {ac,bc}

For the regular expression $\prod$, the regular set is a null or empty set. For the regular expression ε, the regular set is {ε}.

Example,

(c+da*bb)* = {c,dbb,dabb,daabb,...} "

= {ε,c,cc...,dbb,dbbdbb,..,dabb,dabbdabb,...,cdbb, cdabb,..}

Some more examples of regular expressions and their corresponding regular sets are as follows:

1. a = {a}
2. (ab)* = {ε,ab,abab,ababab,abababab,...}
3. a +b +c= {a,b,c}
4. ab+ba = {ab, ba}
5. (a+b)(c+d)p = {acp, adp, bcp, bdp}
6. abc*de = {abde, abcde, abccde, abcccde, …….}

Grammar are the rules for generating meaningful sentences. Two basic elements of a grammar are the terminal symbols and the non-terminal symbols. The terminal symbols are the components of the sentences which are generated by the grammar and are represented by lower case letters like a,b, c etc. The non-terminals are the symbols which take part in the sentence generation. They are represented by upper case letters like A, B, C etc.

Regular grammars are sometimes also known as right-linear grammar. The definition of regular grammar G is a quadruple

(N, T, P, S),

where:

N or V is the set of variables or non-terminals,

T or Σ is the set of terminals symbols,

P is the production rules for the terminals and the non-terminals. A production rule is of the form α→β, whereα and β are strings on V U T and at least one symbol of α belongs to V.

S is the start symbol and S □ N,

Example 1:

Grammar G1= (N,T,P,S)

Where, N = {A} # Set of non-terminal symbols

T = {a,b} # Set of terminal symbols

S = {A} #Start symbol, S " N

P = {A -> Aa, A -> Ab, A -> a, A -> b, A - > ε} # Set of Production rules

ε is a special symbol called NULL. The start symbol is A. From A, the strings Aa, Ab, a, b can be produced. From these strings, more other strings could be produced by replacing A by the strings mentioned in the production rules. As such, strings such as ε(null string), a, b, aa,ab,bb,ba, etc. could be produced. Therefore, in this example, the grammar can be used to produce strings of the form $(a+b)^*$.

Example 2:

Grammar G2 = (N,T,P,S)

Where, N = {S, A, B}

T = {a,b}

S = {S}

P = {S → AB, A → a, B →b}

A language is a set of strings that is made up of characters from alphabets or a set of symbols. Regular language is the language of regular expressions accepted by Non- deterministic finite automata (NFA) and Deterministic finite automata (DFA) are characterized by special grammars called regular grammars.A grammar is said to be a regular grammar if the rules in P:

- Have a Left Hand Side that is a single non-terminal
- Have a Right Hand Side that is: ε, or a single terminal, or a single terminal followed by a single non-terminal.

The Regular grammars as already defined are often called right-regular because they require productions to have the form $A \rightarrow aB$ or $A \rightarrow a$.

Example 1:

$S \rightarrow a$,

$S \rightarrow \varepsilon$,

and $T \rightarrow aS$

This is a regular grammar.

Example 2:

A regular grammar for the language of a*b* is

$S \rightarrow \Lambda \mid aS \mid T$

$T \rightarrow b \mid bT$

Where, Λ, ε are NULL strings.

Example 3:

For the regular expression a*bc* we have the following grammars.

$S \rightarrow aS \mid bT$

$T \rightarrow \Lambda \mid cT$

Question: Write a regular grammar for the language {ab, acb, accb. …, acnb, …}

Solution:

A regular expression for the language is ac*b.

A regular grammar is $S \rightarrow aT$

$T \rightarrow b \mid cT$

Thus, the grammar: $S \rightarrow aSa$ and $aSa \rightarrow T$ is not a Regular grammar as it does not follow the rules of regular grammar.

13.5 CLOSURE PROPERTIES OF REGULAR LANGUAGES

A language is called as regular if it is accepted by a finite state automaton. The term *Closure Propert*y means that the elements of the set

is closed under some mathematical operations. That is, a set is closed with respect to an operation if that operation can always be completed with elements in the set. For example, let I^+ be the set of positive integers and addition (+) and subtraction (-) be the operations performed on the elements of the set I^+. After the addition operation performed on the set the result obtained is some positive integer and belongs to I^+. The result obtained after performing the subtraction operation may or may not belong to the set I^+; thus the set I^+ is closed under the addition operation but is not closed under subtraction operation.

In case of regular sets, if an operation performed on the regular sets leads to another regular set, which is also a regular set, then it can be said that the regular set(s) are closed under that operation. Closure properties of languages express that when one language is regular, then certain languages are also regular.

Regular languages are closed under union, intersection, concatenation, complement, transpose, difference and Kleene's star.

1. The union of two regular languages is regular.

Let L_1 and L_2 be the languages over the alphabet set Σ. Then $L_1 \cup L_2$ is the language that contains all strings that are in either or both of L_1 and L_2. If L_1 and L_2 are regular languages, so is $L_1 \cup L_2$.

Example

$L_1 = \{a^n \mid n > 0\}$

$L_2 = \{b^n \mid n > 0\}$

Then, $L_3 = L_1 \cup L_2 = \{a^n \cup b^n \mid n > 0\}$ is also regular.

2. The intersection of two regular languages is regular.

Let L_1 and L_2 be the languages over the alphabet set Σ. Then $L_1 \cap L_2$ is the language that contains all strings that are both in L_1 and L_2. If L_1 and L_2 are regular languages, so is $L_1 \cap L_2$.

Example

$L_1 = \{a^m b^n \mid n > 0 \text{ and } m > 0\}$

$L_2 = \{a^m b^n \cup b^n a^m \mid n > 0 \text{ and } m > 0\}$

Then, $L_3 = L_1 \cap L_2 = \{a^m b^n \mid n > 0 \text{ and } m > 0\}$ are also regular.

3. The complement of a regular language is regular.

Let *L be a language over the alphabet set* Σ. Then, $\overline{L}$, complement of L is the set of strings in **.....***that are not in L. Complement of a language can be found by subtracting strings which are in L from all possible strings. Complement can be represented by $\overline{L}$ or L'.

Thus, if L is regular language, then $\overline{L} = \Sigma^* - L$ is also regular.

Example

L = {a^n | n > 3}

Then, L' = {a^n | n <= 3}

4. The difference of two regular languages is regular.

If L_1 and L_2 are two languages over the alphabet set Σ. Then L_1-L_2 is the language that contains all strings that are in L_1 but not in L_2. If L_1 and L_2 are regular languages, so is L_1-L_2.

5. The reversal of a regular language is regular.

Reversal of a string $a_1a_2a_3$……..a_n is the string written in the reverse order, a_na_{n-1}…..a_1. The reverse of a string is written as w^R. Thus, the reverse of $(0010)^R$ is 0100.

The reverse of a language L which is written as L^R is the language consisting of reversals of all its strings.

Reversal operation is another operation that preserves regular languages.

If L is a regular language, then L^R is also a regular language.

For example, if L = {01,001,1111} is a regular language then

L^R ={10,100,1111} is also a regular language.

6. The Kleene closure (star) of a regular language is regular.

Let*L be a language over the alphabet set* Σ*, then its Kleene closure L* would also be regular.*

Example

L = (a U b)

L = (a U b)**

7. The concatenation of two regular languages is regular.

If L_1 and $L_2$are two regular languagesover the alphabet set Σ, then *their concatenation L_1.L_2 would also be regular.*

Example

$L_1 = \{a^n \mid n > 0\}$

$L_2 = \{b^n \mid n > 0\}$

$L_3 = L_1.L_2 = \{a^m . b^n \mid m > 0 \text{ and } n > 0\}$ *is also regular.*

8. A homomorphism of a regular language is regular.

A string homomorphism is a function on strings that substitutes a particular string for each symbol. For example, the function *h* given by h(0)=abb and h(1)=ba is homomorphism which replaces each 0 by abb and each 1 by ba.

Hence, h(101) is baabbba.

Homomorphism can be applied to a language by applying it to each of the strings in the language. If L is a language over *the alphabet set* Σ, and h is a homomorphism on Σ, then h(L) = {h(w) | *w is in L}.*

Thus, if L is a regular language and h is a homomorphism on Σ, then h(L) is also regular.

If R = $(0 \cup 1)^*001(0 \cup 1)^*$ and h(0) = ab and h(1) = bc then

h(R) = $(ab \cup bc)$ "ababbc$(ab \cup bc)$

The effect of homomorphism on a language L is shown in the figure 1.

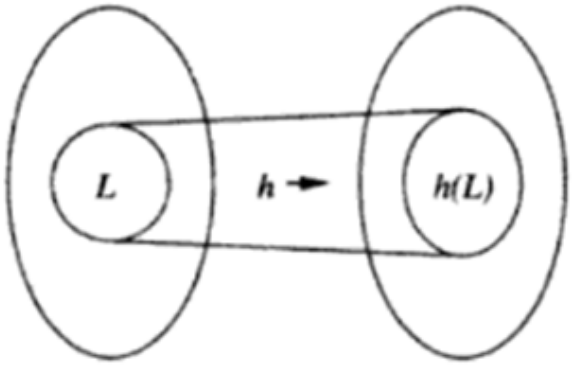


Figure 1. Homomorphism applied in forward direction

9. The inverse of homomorphism of a regular language is regular.

Inverse homomorphism is homomorphism that is applied backwards. Inverse homomorphism also preserves regular languages. Let L be a language. Then $h^{-1}(L)$ is the set of strings w in Σ^* such that h(w) is in L.

The effect of homomorphism on a language L in reverse way can be presented as shown in the following figure 2.

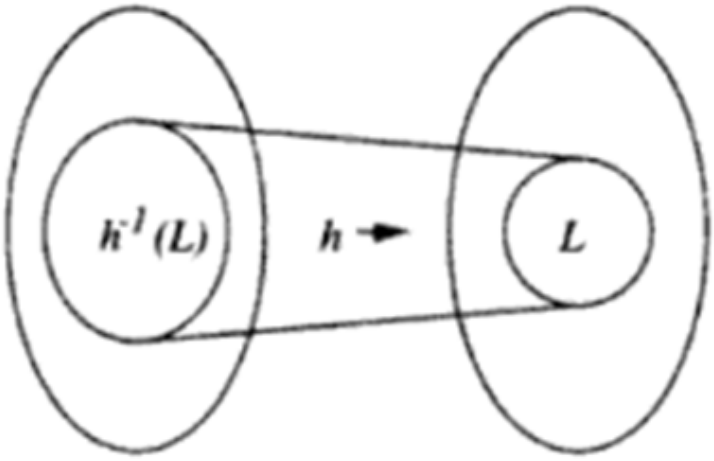


Figure 2. Homomorphism applied in inverse direction

Let Σ = {a, b}, and Δ = {0, 1}.

Let L = $(00 \cup 1)^*$ and h(a) = 01 and h(b) = 10.

Thus, h^{-1} (1001) = {ba},

h^{-1} (010110) = {aab}

h^{-1} (L) = $(ba)^*$

13.6 RELATIONSHIP BETWEEN FINITE AUTOMATA AND REGULAR EXPRESSIONS

Regular expression is the language which is used to describe the language and is accepted by finite automata. Regular expressions are the most effective way to represent any language.

Finite automata are an abstract computing device. It is a mathematical model of a system with discrete inputs, outputs, states and a set of transitions from state to state that occurs on input symbols from the alphabet Σ.

Finite automata is defined as 5-tuples

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

Where,

- Q: Finite set of states.

- Σ: Finite set of alphabets.
- δ: Q × Σ '! Q is the transition function.
- q_0 " Q is the start or initial state.
- F: Final or accept state.

The relationship between Finite Automata and Regular Expression is shown in Figure 3.

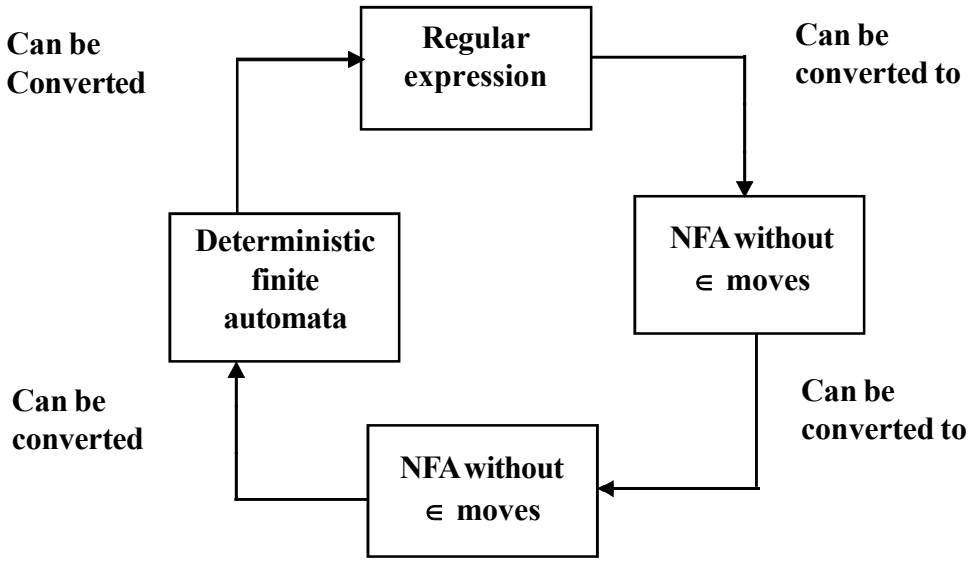


Figure 3. Relationship between Finite Automata and Regular Expression

The above figure explains that it is easy to convert

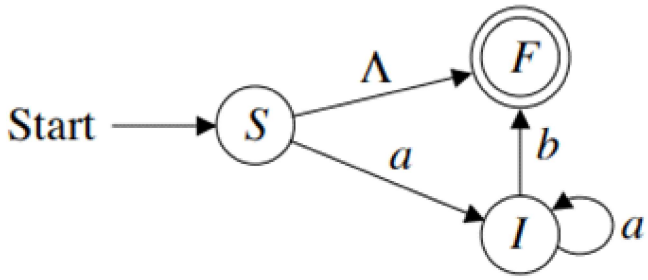
- Regular Expression to Non-deterministic finite automata (NFA) with epsilon moves.
- NFA with epsilon moves to without epsilon moves.
- NFA without epsilon moves to Deterministic Finite Automata (DFA).
- DFA can be converted easily to Regular Expression.

Transforming an NFA to a Regular Grammar

- The names of the states are the non-terminals. They can be named in uppercase letters.
- The start symbol of the grammar is the start state.
- For each state transition from P to Q labeled with x, we need to construct a production P → xQ.

- For each final state F, we need to construct a production F → Λ

Example 1: Transform the following NFA into a regular grammar.



Solution:

S → aI | F

I → aI | bF

F → Λ

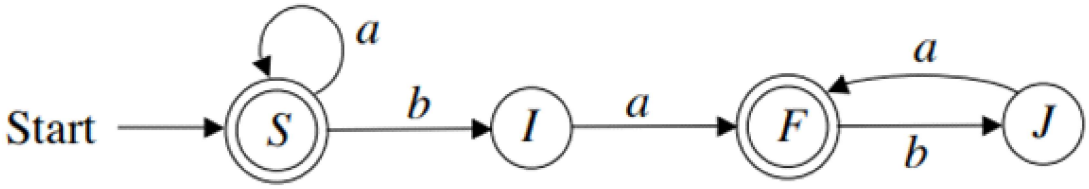
This grammar can be simplified to

S → aI | Λ

I → aI | b

Example 2: Transform the regular expression a*(ba)* into an NFA

Solution:



The resulting grammar from this NFA is:

S → aS | bI | Λ

I → aF

F → bJ | Λ

J → aF

Transforming a Regular Grammar to an NFA

- Replace any production with multiple terminals by productions with single terminals.
- The start state is the grammar start symbol.

- Transform I → aJ into a transition from I to J labeled with a.
- Transform I → J into a transition from I to J labeled with Λ.
- Transform each I → a into a transition from I to new single final state F labeled with a.
- The final states are F together with each state I with a production I → Λ.

Example:

Transform the following regular grammar into an NFA.

S → abS | T | Λ

T → cT | d

Solution:

Transform S → abS into S → aI

and I → bS,

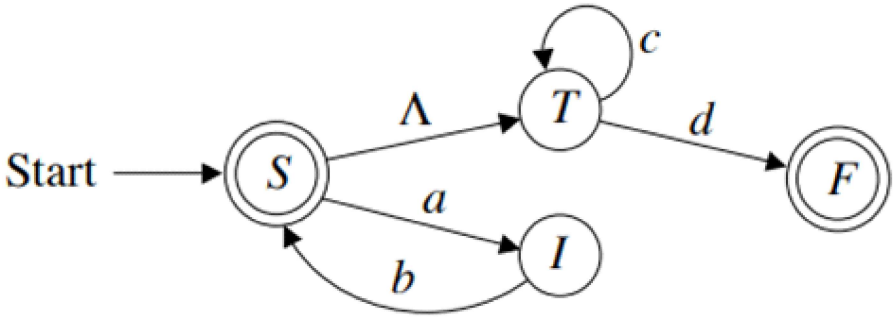
so the grammar becomes

S → aI | T | Λ

I → bS

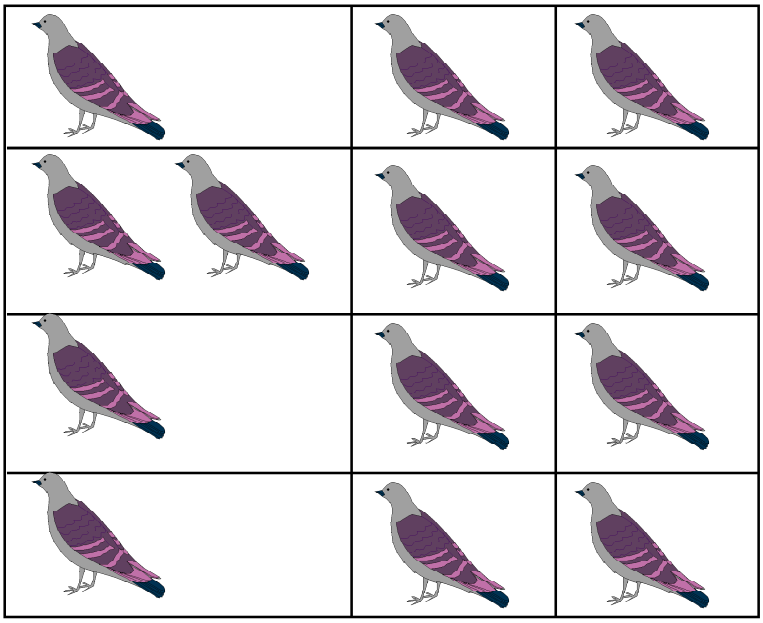
T → cT | d

The NFA can be drawn as follows:



13.7 PIGEON HOLE PRINCIPLE

The pigeonhole principle is a simple concept. Suppose there is a set of pigeons and a set of pigeonholes. If all the pigeons fly into a pigeonhole and there are more pigeons than pigeonholes, then one of the pigeonholes has to contain more than one pigeon. Say there is a flock of 13 pigeons that flies into a set of 12 pigeonholes to roost. Since, there are 13 pigeons with only 12 pigeonholes, a least one of these 12 pigeonholes must have at least two pigeons in it as shown in the following figure.



This illustrates a general principle called the pigeonhole principle, which states that if there are more pigeons than pigeonholes, then there must be at least one pigeonhole with at least two pigeons in it.

In other words, pigeonhole principle states that if n items are put into m containers, with n > m, then at least one container must contain more than one item. For example, in a group of 367 people, there must be at least 2 people having the same birthday as there are possible 366 birthdays in a year.

Theorem:

Let q1, q2, . . . , qn be positive integers.

If q1+ q2+ . . . + qn – n + 1 objects are put into n boxes, then either the 1st box contains at least q1 objects, or the 2nd box contains at least q2 objects, . . ., the nth box contains at least qn objects.

For Example:

In Psychology department, a student club can be formed with either 10 members from first semester or 8 members from second semester or 6 from third semester or 4 from final semester. What is the minimum no. of students that can be chosen randomly from the department so as to ensure that a student club is formed?

Solution:

Applying the above formula where,

q1 =10, q2 =8, q3 =6, q4 =4

and n=4

Thus, the minimum number of students required is

10 + 8 + 6 + 4 – 4 + 1 = 25

13.8 APPLICATIONS OF REGULAR EXPRESSIONS

Regular expressions are used in numerous tasks ranging from compilers to daily tasks like web search. They, thus, find application in pattern matching, web search engines, software engineering, lexical analysis, in Unix/Linux commands etc.

One of the common uses of regular expression is in the web search engines. In World Wide Web, the early search engines also used regular expressions to search through their indexes. Regular expressions were chosen for these early search engines because of both their power and easy implementation. Archie, one of the first search engines, used regular expressions exclusively to search through a database of filenames. In the case of a search engine, the input strings to the regular expression would be either whole web pages or an index of a web page. A query such as regular expression could be translated into the following regular expression.

$(\Sigma^*\text{regular}\Sigma^*\text{expression}\Sigma^*)^*U\ (\Sigma^*\text{expression}\Sigma^*\text{regular}\Sigma^*)^*$

Σ is the set of all characters in the character encoding used with this search engine.

Regular expressions are used in several Linux commands like grep, ed, vi, sed, awk etc. In software engineering, the regular expressions are generally used in the testing phase. In lexical analysis, parsing is done. There are two components, scanning and tokenization. A token is defined by regular expression. In tokenization, a finite state machine is generally used to define the lexical grammar of the language that is analyzed. To generate this finite state machine, regular expressions are used.

Regular expressions can be used in web scrapping (data collection). That is, the websites can be scrapped effectively with the help of regular expressions.

They also find applications in NLP (Natural Language Processing) in data pre-processing.



CHECK YOUR PROGRESS

Q 3: Find a regular grammar for each of the following:

a. 1 + 01

b. {00, 10, 01}

c. All strings which have an odd number of 1's

Q 4: Let R1 and R2 be two regular sets defined over the alphabet, then which of the following is true:

a. $R_1 \cap R_2$ is not regular

b. $R_1 \cup R_2$ is not regular

c. Σ^* is regular

d. R_1^* is not regular

Q 5: A string _____ is a function on strings that substitutes a particular string for each symbol. (Fill up the blank)

Q 6: The inverse of homomorphism of a regular language is _____. (Fill up the blank)



13.9 LET US SUM UP

- A string is a regular expression if it is a primitive regular expression or it can be derived from the primitive regular expressions by applying a finite number of the operations +, * and concatenation.
- Every character that belongs to Σ, that is, a $\in \Sigma$ is a regular expression over Σ.
- ϕ (empty set) and ε (empty string of length zero) are also regular expressions.

- If R_1 and R_2 are regular expressions over Σ, then $R_1 + R_2$, $R_1.R_2$, $(R_1)^*$ are regular expressions.
- A regular expression generates a set of strings known as Regular set.
- The language corresponding to a regular expression is called a regular language.
- A regular set or a regular language is a language accepted by some FA (Finite Automata) that can be represented by a regular expression.
- Grammar are the rules for generating meaningful sentences. The two basic elements of a grammar are the terminal symbols and the non-terminal symbol.
- Regular grammar G is a quadruple (N, T, P, S),
- A grammar is said to be a regular grammar if the rules in P:
  - ✧ have a Left Hand Side that is a single non-terminal
  - ✧ have a Right Hand Side that is: ε, or a single terminal, or a single terminal followed by a single non-terminal.
- A language is called as regular if it is accepted by a finite state automaton.
- *Closure Propert*y means that the elements of the set is closed under some mathematical operations.
- Regular languages are closed under union, intersection, concatenation, complement, transpose, difference and Kleene's star.
- Pigeonhole principle states that if there are more pigeons than pigeonholes, then there must be at least one pigeonhole with at least two pigeons in it.
- If q1+ q2+ . . . + qn – n + 1 objects are put into n boxes, then either the 1st box contains at least q1 objects, or the 2nd box contains at leastq2 objects, . . ., the nth box contains at least qn objects.
- Regular expressions are used in numerous tasks ranging from compilers to daily tasks like web search. They, thus, find application in pattern matching, web search engines, software engineering, lexical analysis, in Unix/Linux commands etc.



13.10 FURTHER READING

1. Theory of Computer Science, 3rd Edition, KLP Mishra, N Chandrasekaran, PHI
2. Theory of Computation, Vivek Kulkarni, Oxford
3. Theory of Computation, SN Sivanandam, M Janaki Meena, I K International



13.11 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No. 1: $0(10)^* + 0(10)^* 1 + 1(01)^* + 1(01)^*0$

Explanation: If a string w in L begins with a 0, then it should follow some finite number of 10. If a string w in L begins with a 1, then it should follow some finite number of 01.

Ans to Q No. 2:

a. $a(a+b+c)^*cc$

b. $00(11)^*$

c. $b^*ab^*ab^*$

d. aabb + abab + abba + baab + baba + bbaa

e. 1*01*

f. $1^* + 1^*01^*$

g. ab^*a

h. $(0+1)^* 00 (0+1)^*$

i. $00^* 11^* 22^*$

Ans to Q No.3:

a. The language contains only 2 strings, 1 and 01. The regular grammar for the language is :

S -> 1|0A

A-> 1

b. The language contains 3 strings, 1 and 01. The regular grammar for the language is :

S -> 1A | 0B

A-> 0 |1

B-> 0

c. The strings having odd number of 1's is :

L ={1, 111, 11111, ……}

The regular grammar for the language L is :

S -> 1 | 1A

A ->1B

B -> 1A |1

Ans to Q No.4: Option 3 is correct because

$\sum^* - R_1 -> \sum^* \bigcap R_1^*$ is regular since closed under complement.

Ans to Q No.5: Homomorphism

Ans to Q No.6: regular

Ans to Q No.7: two



13.12 MODEL QUESTIONS

Q.1. Define the following and give examples.

i Regular set

ii Regular expression

iii Regular language

Q.2. Regular set denoted by the regular expression (a+b)(a+b) is _____. (Fill up the blank)

Q.3. Write in brief the applications of finite automata.

Q.4. Explain he applications of regular expressions.

Q.5. What is the pigeon hole principle?

Q.6. If r=ab*a, describe the language of r in the form of a set.

*** ***** ***

UNIT 14: CONTEXT-FREE GRAMMARS AND LANGUAGES

UNIT STRUCTURE

14.1 Learning Objectives

14.2 Introduction

14.3 Context-Free Grammars.

14.3.1 Context-Free Languages

14.4 Derivation Tree

14.4.1 Leftmost Derivation and Rightmost Derivation

14.4.2 Ambiguity in Grammars and Languages

14.5 Applications of Context-Free Grammar

14.6 Let Us Sum Up

14.7 Further Reading

14.8 Answers to Check Your Progress

14.9 Model Questions

14.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- define Context-Free Grammar and Language
- learn the basic structure of Context-Free Grammar
- draw derivation tree
- identify the ambiguity in Context-Free Grammar
- derive strings using leftmost and rightmost derivation
- learn about the application of Context-Free Grammar, languages and derivation tree.

14.2 INTRODUCTION

We have already been acquainted with the regular grammar and various properties of regular languagein the previous unit. In this unit we will discuss context-free grammars andlanguages. We will learn the definition of derivation trees, leftmost and rightmost derivation, and methods

of identifying ambiguity in context-freegrammars. At the end we will learn the application of this type of grammars and languages.

14.3 CONTEXT-FREE GRAMMARS

Generally, there are four types of grammar in finite automata that have been defined by *Noem Chomsky* in the mid-1950s.

- Irregular Grammar, or Type 0
- Context-Sensitive Grammar (CSG), or Type 1
- Context-Free Grammar (CFG), or Type 2
- Regular Grammar, or Type 3

In this section we are going todiscuss the Context-Free Grammar (CFG) which is also known as Type 2 grammar.*Chomsky* proposed the notion of CFG as a model for describing natural languages. When it is applied to English language, it was found successful in describing only a small subset of simple sentences. CFG has been widely used for defining programming languages rather than natural languages.The tool used for implementing the CFG is the pushdown automaton (PDA). A pushdown automaton is much more powerful than a finite automaton because of its memory storage u=in the form of a stack.

Formally, a **Context-Free Grammar** is a four-tuple G = (V,T, P, S) where V is a finite set of variables (non-terminals)

T is a finite set of terminals

P is a set of production rules

S is the start symbol $(S \in V)$

Variables (V): V is the finite set of **variables** which are also known as **nonterminals** or **syntactic categories**. Variables are used to define the structure of sentences in a language. They do not participate in the formation of sentences but instances of variables form sentences. For example, in case of English language, noun, pronoun, adjective, verb, adverb etc. are the syntactic categories of the language. Instances of noun, pronoun, adjective, verb etc. are

actually used to form the sentences of the language. Let us consider a sentence "*Sumi sings well*". Here, "*Sumi*" is an instance of noun, "*sings*" is an instance of verb and "*well*" is an instance of adjective. Like this, instances participate to form sentences. The phase structure of the sentence mentioned here can be described grammatically in noun, verb and adverb as S = NVA where S is a sentence which is formed by combining noun phrase (N), verb phrase (V) and an adjective(A). In automata theory, the nonterminal are represented by capital letters.

Terminals (T): Terminals are a finite set of characters that are used to form all the valid words (or strings) in the language. In automata theory, terminals (sometimes denoted by the symbol Σ) are generally represented by small letters. In case of English language, the alphabets = {A, B, C, ...Z, a,b, ..., z} is the character set that form all the valid words of the English language. For a set {a,b} where there are only two elements a and b, there may be strings like "aabaaabbaaaabbb", abab, bababa, ab, ba. Here the elements a and b are termed as terminals.

Production (P): Productions are set of rules with the use of which valid sentences of a language can be made. It defines how to rewrite a variable into something else. A production has two sides, the left-hand side (LHS) and the right-hand side (RHS) which are separated by the forward arrow'! symbol.

In CFG, each production in the production set P is of the form

$$A \rightarrow (V \bigcup T)^*$$

The LHS contains a single variable, or non-terminal. But the string in the RHS can be any combination of variables (V) and terminals (T). And there can be more than one production rules with the same LHS. In the RHS, $\{V \cup T\}^*$ means it can only be a variable, or a terminal, or any combination of variables and terminals.

Some examples of productions of CFG are given below for learning the structure of productions.

$$S \to aSa$$
$$S \to bSb$$
$$S \to \in$$

More than one production can also be written in a single line. The three productions mentioned above can also be written as a whole as follows:

$$S \to aSa \mid bSb \mid \in$$

Some more examples of productions of CFG are as follows:

$$A \to aBBc$$
$$S \to BB$$
$$S \to a$$
$$S \to b$$
$$A \to aAb$$

where V={A,S, B} and T= {a, b, c}.

Starting Symbol (S): S is an initial production from where the process of string formation begins. Therefore, S is also known as the *starting sentential form.*

Example 14.1: Obtain the CFG which generates a string containing only 1's. For example, string like 1111111 etc.

Solution: If we consider the first production with start symbol S is

$$S \to 1S$$

We can obtain strings like 11, 111,1111, etc. by substituting and .

Thus,

$S \to 1S \to 11S \to 111S \to 1111S \to 11111S \to 111111S \to 1111111S \to$

$1111111 \in \to 1111111$

The CFG will be G = (V,T, P, S) where V = {S}, T = {1}, $P: S \to 1S, S \to \in$ and S={S}.

Example 14.2: Obtain a context-free grammar G generating all integers including sign.

Solution: Let G=(V,T,P.S)

where V={ S , (sign), (digit), (integer)}

T= {0,1, 2, 3, ….,9, +, - }

The productions P: $S \rightarrow < sign >< integer >$, $< sign > \rightarrow + | -$,

$< integer > \rightarrow < digit >< integer > | < difit >$

$< digit > \rightarrow 0|1|2| \ldots .|9$

Thus L(G)= the set of all integers. For example, let us derive -25. This can be done as follows:

$S \rightarrow < sign >< integer > \rightarrow -< integer > \rightarrow -< digit >< integer > \rightarrow$

$-2 < integer > \rightarrow -2 < digit > \rightarrow -25$

14.3.1 Context-Free Languages

The language generated by context-free grammar are known as ***context-free language***.A CFL is accepted by a pushdown automaton. Thus, a language L is context-free if there is a context-free grammar

G = (V, T, P, S), such that L is generated from G.

Example 14.3: Give the context-free language generated by the grammar

G = (V, T, P, S), where V = {S}, T= {0,1}, .

Solution: Here, 0 and 1 are the terminal, S is the start symbol as well as the onlyvariable. If we apply the first production n-1 times, followed by the second production, we get

$S \rightarrow 0S1$

$S \rightarrow 00S11$ using $S \rightarrow 0S1$

$S \rightarrow 000S111$ using $S \rightarrow 0S1$

$S \rightarrow 0000S1111$ using $S \rightarrow 0S1$

$S \rightarrow 00000S11111$ using $S \rightarrow 0S1$.

$S \rightarrow 0^n 1^n$

Hence, the language accepted by the given grammar is

$L(G) = \{0^n 1^n | n \geq 1\}$

14.4 DERIVATION TREE

A string w belongs to a context-free language, L if it can be derived from the Context-Free Grammar, G associated with L. We can use the notion L(G) to represent the language L corresponding to grammar G. One approach to check if a given string w belongs a CFL or not is known as ***derivation***. A context-free grammar specifies how to generate syntactically valid strings by the process of derivation.

Derivation is a process which starts with S, the starting sentential form. S is then replaced by its corresponding productions, which may still contain some variables. These variables are again replaced by their corresponding productions, which may still contain some variables. The process continues till all the variables are replaced by the terminals. This process of derivation can be diagrammatically represented by a tree like structure, which is known as ***derivation tree***. *Derivation tree* is also termed as ***parse tree***. The resultant string of terminals is the output of the tree. If the output string is denoted by w which is derived from some CFG corresponding to CFL L, then w is belong to L. Symbolically it can be written as

$$w \in L$$

It is created by the parser module of the compiler for checking the validity of an input string for being a valid statement in a given language.

Formally, a derivation tree for a CFG can be defined as $G = (V, T, P, S)$ is a tree satisfying the following conditions:

i) Every vertex has a label which is a variable or terminal or $\in$.

ii) The label of the root is S.

iii) The label of an internal vertex is a variable.

iv) If the vertices n_1, n_2, n_3, …, n_k written with labels X_1, X_2, X_3, …. X_k are the sons of vertex n with label A, then $A \rightarrow X_1 X_2 \ldots\ldots X_k$ is a production in P.

v) A vertex n is a leaf if its label is $a \in T$ or $\in$, T is the terminal. N is the only son of its father if its label is $\in$.

A ***derivation tree*** is a pictorial representation of derivations. A tree consists of some vertices and branches. The top most vertex is termed as root. Branches are simply the connections between the pairs of vertices.

Example 14.4: Let there be a CFG, $G = (V, T, P, S)$ where $T = \{a,b\}$, $V = \{S\}$,

$P : S \rightarrow aSa$

$S \rightarrow bSb$

$S \rightarrow \in$

Generate a parse three for any word/string of the grammar.

Solution:

$S \rightarrow aSa$

$\rightarrow aaSaa$ using $S \rightarrow aSa$

$\rightarrow aabSbaa$ using $S \rightarrow bSb$

$\rightarrow aab \in baa$ using $S \rightarrow S \in$

$\rightarrow aabbaa$

We can draw the parse three for the string “aabbaa”using the derivation as follows:

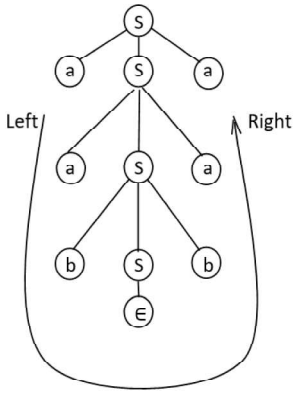


Fig.14.1: Derivation or parse tree for the string “aabbaa”

The string “aabbaa” can be obtained from the tree drawn in figure 14.1 by writing the terminals from LHS to RHS.For different string w of the language L generated by the grammar G, the structure of the parse tree will be different.We can replace S of the first production by “bSb” and the subsequent productions as shown in the following steps:

$S \rightarrow aSa$

$\rightarrow abSba$ using $S \rightarrow bSb$

$\rightarrow abaSaba$ using $S \rightarrow aSa$

$\rightarrow abaaSaaba$ using $S \rightarrow aSa$

$\rightarrow abaa \in aaba$ using $S \rightarrow \in$

$\rightarrow abaaaaba$

We obtained a different string “abaaaaba” and for this string the structure of the parse three will be different. This is shown in figure 4.2.

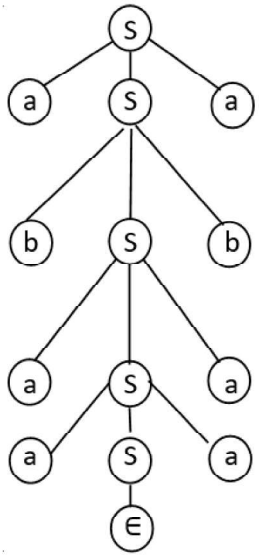


Fig.14.2: Derivation or parse tree for the string “abaaaaba”

The language generated by the CFG will be

$$L(G) = \{\, Word\ followed\ by\ same\ word\ but\ in\ reverse\ order \,\}$$

Or,

$$L(G) = \{ww^R | w \in \{a, b\}^*\}$$

Example 14.5: Given a grammar $G = (V, T, P, S)$ where

$$T = \{a, b\}, \qquad V = \{S, A, B\},$$

$$P:\ S \rightarrow aB|bA|\ \epsilon$$

$$A \rightarrow aS|a|bAA$$

$$B \rightarrow b|bS|aBB$$

Draw a parse three for any word/string of the grammar.

Solution:

$S \rightarrow aB$

$\rightarrow abS$ using $B \rightarrow bS$

$\rightarrow abbA$ using $S \rightarrow bA$

$\rightarrow abbbAA$ using $A \rightarrow bAA$

$\rightarrow abbbaSA$ *using* $A \rightarrow aS$

$\rightarrow abbba \in$ *using* $S \rightarrow \in$

$\rightarrow abbbaa$ *using* $A \rightarrow a$

The result of the derivation can be written with the symbol $\overset{*}{\Rightarrow}$ as follows:

$$S \overset{*}{\Rightarrow} abbbaa$$

The symbol indicates that the RHS was obtained by using some production rules to replace some non-terminals in LHS. The final sentential form is called the "yield".

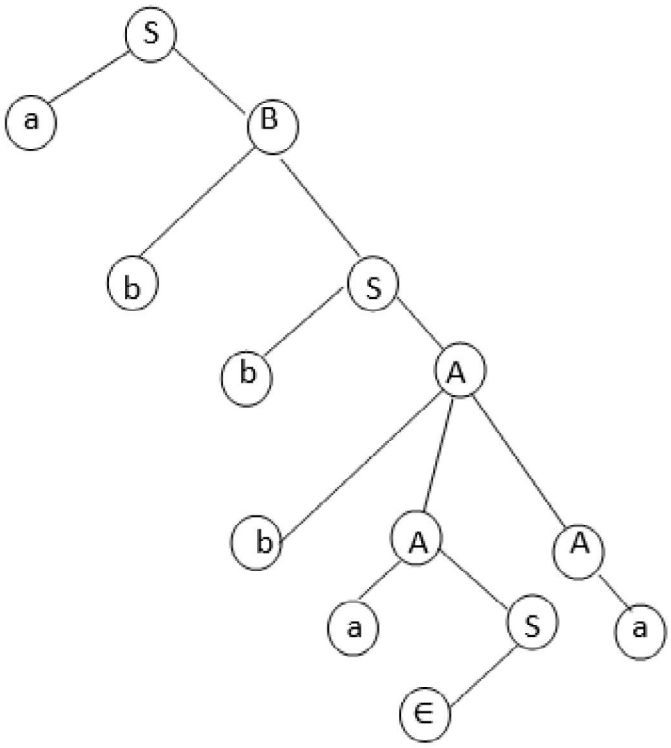


Fig.14.3: Parse tree for the string "abbbaa"



CHECK YOUR PROGRESS

Q 1: Give an example of CFG. Also give the language generated by the CFG.

Q 2: Fill in the blanks:

i) A _____ is a pictorial representation of derivations.

ii) A tree consists of some _____ and _____.

iii) The top most vertex of a parse tree is termed as_____.

iv) The derivations in a _____ can be represented using trees.

v) Context-Free Grammar is also known as _____ grammar.

14.4.1 Leftmost Derivation and Rightmost Derivation

Derivation is series of replacement operations that show how to derive a string of terminals from the start variable. A string or word w in a context-free language (CFL) can be derived in two ways:

- Leftmost derivation
- Rightmost derivation

In case of leftmost derivation, at each step of derivation, the leftmost variable is replaced by its one of its productions. But in right most derivation, at each step the rightmost variable is replaced y one of its productions.

Example 14.6: Given a grammar with the following productions rules:

$$P: \quad S \rightarrow A|AB$$
$$A \rightarrow a|AA|Ab| \in$$
$$B \rightarrow b|bc|Bc|bB$$

Obtain a string “aabbc” using leftmost derivation and rightmost derivation separately.

Solution: As S is the start symbol, and we will first take a production which starts with S.Let us apply **leftmost** derivation:

$S \rightarrow \boldsymbol{A}B$

$S \rightarrow \boldsymbol{AA}B$ using $A \rightarrow AA$

$S \rightarrow \boldsymbol{a}AB$ using $A \rightarrow a$

$S \rightarrow a\boldsymbol{a}B$ using $A \rightarrow a$

$S \rightarrow aabB$ using $B \rightarrow bB$

$S \rightarrow aab\boldsymbol{bc}$ using $B \rightarrow bc$

Thus, we have obtained the string "aabbc" applying the left mostderivation.

Now, let us apply **rightmost** derivation:

$S \rightarrow A\boldsymbol{B}$

$S \rightarrow A\boldsymbol{B}c$ using $B \rightarrow Bc$

$S \rightarrow A\boldsymbol{bB}c$ using $B \rightarrow bB$

$S \rightarrow Ab\boldsymbol{b}c$ using $B \rightarrow b$

$S \rightarrow \boldsymbol{AA}bbc$ using $A \rightarrow AA$

$S \rightarrow A\boldsymbol{a}bbc$ using $A \rightarrow a$

$S \rightarrow \boldsymbol{a}abbc$ using $A \rightarrow a$

Thus, we have obtained the string "aabbc" applying the rightmost derivation.

Example 14.7: Let there be a context-free language L for a CFG represented by the following productions:

P: $S \rightarrow ASA|BSB|a|b$

$A \rightarrow a$

$B \rightarrow b$

Show the leftmost and rightmost derivation for the string w=aaabbbaaa$\in L$.

Solution:

Applying leftmost derivation

$S \rightarrow ASA \rightarrow aSA \rightarrow aASAA \rightarrow aaSAA \rightarrow aaASAAA \rightarrow aaaBSBAAA \rightarrow aaabSBAAA \rightarrow aaabbbAAA \rightarrow aaabbbaAA \rightarrow aaabbbaaA \rightarrow \boldsymbol{aaabbbaaa}$

Applying rightmost derivation

$S \rightarrow ASA \rightarrow ASa \rightarrow AASAa \rightarrow AASaa \rightarrow AAASAaa \rightarrow AAABSBaaa \rightarrow AAABSbaaa \rightarrow AAABbbaaa \rightarrow AAAbbbaaa \rightarrow Aaabbbaaa \rightarrow \boldsymbol{aaabbbaaa}$

14.4.2 Ambiguity in Context-Free Grammar

Ambiguity is a property of a grammar which leads to multiple interpretations of the same sentence. Sometimes we come across sentences which carry double meaning. For example, the sentence "*I saw someone on the top floor of the building with a camera*". This sentence has dual meaning. Did you use a camera to see someone on the top floor of the building or did you see someone on the top floor of the building holding a camera?Thus, ambiguity affects the meaning of expression. The same situation may arise in context-free languages.

A context-free grammar is said to be ambiguous if it possesses more than one parse tree for the same string or expression. Thus, a context-free grammar G is ambiguous if some string has more than one leftmost or rightmost derivation or parse tree.

Example 14.8: Given a grammar with the following productions rules:

P: $S \rightarrow aS|aSb|X$

$X \rightarrow Xa|a$

Prove that the CFG is ambiguous.

Solution:I we find the same string with two different parse tree, then the grammar is ambiguous. Again, if we find more than one leftmost and rightmost derivation for the same string, then also the grammar is ambiguous.

$S \rightarrow a\boldsymbol{Sb} \rightarrow a\boldsymbol{aS}b \rightarrow aa\boldsymbol{aS}b \rightarrow aaa\boldsymbol{X}b \rightarrow aaa\boldsymbol{a}b$ …….. derivation 1

$S \rightarrow aS \rightarrow a\boldsymbol{aS} \rightarrow\rightarrow aa\boldsymbol{aSb} \rightarrow aaa\boldsymbol{X}b \rightarrow aaa\boldsymbol{a}b$ …….. derivation 2

Fig.14.4. a) parse tree from derivation 1 b) parse tree from derivation 2

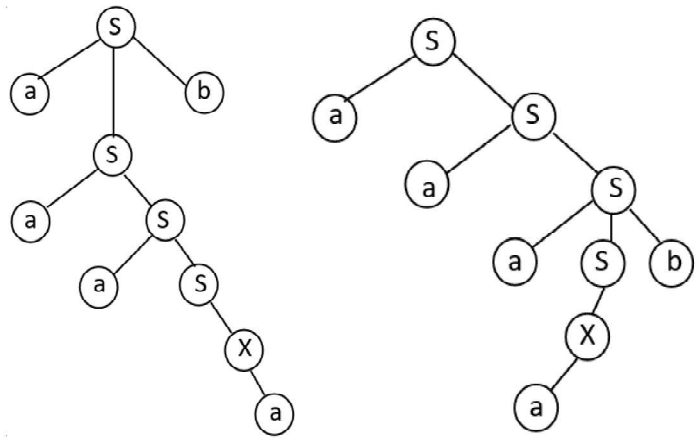


Fig.14.4. a) parse tree from derivation 1 b) parse tree from derivation 2

14.5 APPLICATIONS OF CONTEXT-FREE GRAMMAR

Context-free languages are very important in defining block structures in programming languages, simplifying translation of programming languages. They are applied in parser design.



CHECK YOUR PROGRESS

3 Q: Design a CFG for the language L over {0,1} to generate all possible strings of even length.

4 Q: Suppose a CFG is defined as G= (V, T, P, S) where V={S}, T= {(,)}, S={S}, and P: {P→ SS, $S \rightarrow (S)$, S→ ϵ}. Evaluate the expression

i. (()) ii. (() ())



14.6 LET US SUM UP

What we have learnt from this unit are summarized below:

- A context-free grammar is defined by a four tuple structure G=(V, T, P, S), where V is a set of nonterminals, T is a set of terminals, P is the production rules, and S is the starting sentential form.

- A grammar G is context-free if every production is of the form $A \rightarrow \alpha$, where $A \in V$ and $\alpha \epsilon (V \cup T)^*$.
- A derivation tree is a pictorial representation of derivations.A derivation tree also called the parse tree for a CFG.
- A string or a word can be derived by using either the leftmost or the rightmost derivation or the both.
- Sometime we find ambiguous sentences in the language we are using. This This may occur in context-free language also.
- A CFG is ambiguous if any one of the following happens:

  - There exists a string w in the CFG with multiple parse trees,

  - There exists a string w in the CFG with multiple leftmost derivations,

  - There exists a string w in the CFG with multiple rightmost derivations
- If same terminal string can be obtained from two different derivation trees of a grammar, then the grammar is said to be ambiguous. Thus, there may be two leftmost derivations of the string w and that leads to the definition of ambiguous sentences in a context-free language.



14.7 FURTHER READING

1. Mishra, K. L. P., & Chandrasekaran, N. (2006). *Theory of Computer Science: Automata, Languages and Computation*. PHI Learning Pvt. Ltd.
2. Nagpal, C. K. (2012). *Formal Languages and Automata Theory*. Oxford University Press.
3. Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007).*Introduction to automata theory, languages, and computation*. Pearson Education

14.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q.No.1:

Given a grammar G= ({S}, {a, b}, P, S). The set of rules P is

$S \rightarrow aSa$

$S \rightarrow bSb$

$S \rightarrow \in$

Using the production rules, this grammar generates strings like "aabbaa" and "bbbbbb". These can be derived as follows:

$S \rightarrow aSa \rightarrow a\boldsymbol{aSa}a \rightarrow aa\boldsymbol{bSb}aa \rightarrow \boldsymbol{aabbaa}$

$S \rightarrow bSb \rightarrow baSab \rightarrow babSbab \rightarrow \boldsymbol{babbab}$

$S \rightarrow bSb \rightarrow bbSbb \rightarrow bbbSbbb \rightarrow \boldsymbol{bbbbbb}$

$S \rightarrow bSb \rightarrow bbSbb \rightarrow bbbSbbb \rightarrow bbbaSabbb \rightarrow \boldsymbol{bbbaabbb}$

Thus the language generated by the grammar is

$L(G) = \{ww^R | w \in \{a, b\}^*\}$

Ans.Q.No.2: i) derivation tree ii) vertices, branches iii) root iv) CFG v) Type 2

Ans.Q.No.3:

The string of length zero (0) is denoted by $\in$ and the corresponding production with the start symbol S is S$\rightarrow \in$.

We know that the number 0 is always considered to be even. A string of length 2 over the alphabet {0,1} consists of one of the strings 00, 01,10, or 11.

The corresponding productions are S$\rightarrow$00|01|10|11.

An even length string 2n where n>1 contains one or more repetitions of the strings 00, 01,10, or 11. The corresponding productions will be S$\rightarrow$00S|01S|10S|11S.

For example, if we want to derive a string 010100110011 of {0,1} of even length, then it can be derived as follows:

$S \rightarrow 01S\ S \rightarrow 0101S \rightarrow 0101\mathbf{00}S \rightarrow 01010011S \rightarrow 0101001100S \rightarrow$

$010100110011S \rightarrow 010100110011$

Therefore, the production set for the CFG corresponding to the CFL is

S→∈ |00|01|10|11|00S|01S|10S|11S

Ans.Q. No. 4: i) $S \rightarrow (S) \rightarrow ((S)) \rightarrow ((\in)) \rightarrow (( ))$

ii) $S \rightarrow (S) \rightarrow (SS) \rightarrow ((S)S) \rightarrow ((\in)S) \rightarrow (( )(S)) \rightarrow (( ) ( ) \rightarrow ((\quad)(\quad))$



14.9 MODEL QUESTIONS

Q 1: Define a Context-Free Grammar (CFG). Give some examples of CFG.

Q 2: Define "derivation tree".

Q 3: What do you mean by sentential form?

Q 4: Explain rightmost and leftmost derivation.

Q 5: Define the term "Ambiguous Grammar".

Q 6: Design a CFG for the language $L = \{0^n 1^n | n \geq 0\}$

Q 7: Given a grammar with the following productions rules:

$$P: \quad S \rightarrow A|AB$$

$$A \rightarrow a|AA|Ab| \in$$

$$B \rightarrow b|bc|Bc|bB$$

Draw the parse threes for obtaining the string "aabbc" using leftmost derivation and rightmost derivation separately.

Q 8: Given grammar G with productions

$$P: \quad S \rightarrow aB|bA,$$

$$A \rightarrow a|aS|bAA, B \rightarrow b|bs|aBB.$$

For the string aaabbabbba, find a rightmost derivation, leftmost derivation, leftmost derivation and parse tree.

Q 9: For a grammar G with productions

$$P: \quad S \rightarrow aAS|a$$

$$A \rightarrow SbA|SS|ba$$

Derive the string "aabbaa" and construct a derivation tree for the same string.

Q 10: If G is the grammar $S \rightarrow SbS|a$, show that G is ambiguous.

*** ***** ***

UNIT 15: PROPERTIES OF CONTEXT-FREE LANGUAGES

UNIT STRUCTURE

15.1 Learning Objectives

15.2 Introduction

15.3 Normal forms for CFGs

15.4 The pumping lemma for CFGs

15.5 Closure properties of CFL

15.6 Let Us Sum Up

15.7 Further Reading

15.8 Model Questions

15.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- know about context-free language and its characteristics.
- learn how to define a context-free language
- know about different normal forms of formal language
- learn about closure properties of context-free language
- learn about pumping lemma for context-free language

15.2 INTRODUCTION

In the previous unit, we have learned about context-free grammar which has a specific characteristic due to that any string derivation becomes easy and simple. When the syntax of a language is defined by a context-free grammar then the language is called as context-free language. We have seen that there is a limitation of regular language as the corresponding acceptor like NFA/DFA has no auxiliary memory. Context-free grammar is more powerful since it can be considered for regular language as well as other non-regular language.

In this unit we will learn about different normal forms of context-free grammar with their applications and advantages. We will also learn about context-free language with its closure properties. We will also discuss about

the pumping lemma for context-free language which can be applied to check whether the given language is context-free or not.

15.3 NORMAL FORMS FOR CFGs

We can recall that a grammar G is considered as collection of four tuples i.e. <N, T, P, S>

where,

N is a finite set of non-terminal symbols

T is a finite set of terminal symbols

P is a finite subset of $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$

and an element $(\alpha, \beta) \in P$ is written as $\alpha \rightarrow \beta$

S is a distinguished symbol in N and is called the start symbol

A grammar is considered as context-free grammar (CFG) if all the production rules are in the following form.

A-> X, where

A is considered as head of the production and a non-terminal which belongs to N,

X is considered as body of the production and any combination using terminals and non-terminals of the grammar.

Normal form for grammar production rules is very much required as RHS of production rules have no specific format. Along with normalization, production rulesare performed using normal forms to standardize the grammar.

Since X can be any combination using terminals and non-terminals of a grammar, ε which is null symbol (terminal), can be as a body of a production rule of context-free grammar. This type of production rules can mislead derivation of input strings of the language since any input string does not have any **ε** symbol.

Thus, we need to learn about different normal forms of context-free grammar where all production rules are of a special form and no ε production rule is considered. In this unit, we will discuss two different normal forms of context-free grammar.

i) Chomsky Normal Form (CNF)

ii) Greibach Normal Form (GNF)

Chomsky Normal Form (CNF): A context-free grammar (CFG) is in Chomsky normal form if all production rules are in the form of any of the following:

i) A non-terminal generating two non-terminals. For example, $A \rightarrow BC$.

ii) A non-terminal generating a terminal. For example, $A \rightarrow a$.

The following steps are to be followed to standardize a context-free grammarinto CNF.

i) Eliminating $\in$ productions and unit productions (if any)

ii) Replace all production rules which are of the form $A \rightarrow B_1B_2B_3....B_n$ where $n > 2$ with $A \rightarrow B_1C$ where $C \rightarrow B_2B_3....B_n$. Repeat this step for all production rules which have more than two variables on RHS of the production rules.

iii) Replace all production rules which are of the form $A \rightarrow aB$ with $A \rightarrow XB$ and $X \rightarrow a$.

Consider the following grammar

$S \rightarrow a \mid aA \mid B$

$A \rightarrow aBB \mid \varepsilon$

$B \rightarrow Aa \mid b$

As the given grammar has $A \rightarrow \varepsilon$ null production, after removal this production rule from the grammar will be like

$S \rightarrow a \mid aA \mid B$

$A \rightarrow aBB$

$B \rightarrow Aa \mid b \mid a$

Now, the above grammar has Unit production, $S \rightarrow B$. After removal of the unit production, the grammar is like

$S \rightarrow a \mid aA \mid Aa \mid b$

$A \rightarrow aBB$

$B \rightarrow Aa \mid b \mid a$

Now applying step no (ii) and (iii) the grammar can be in the CNF as

$S \rightarrow a \mid XA \mid AX \mid b$

$A \rightarrow RB$

$B \rightarrow AX \mid b \mid a$

$X \rightarrow a$

$R \rightarrow XB$

Greibach Normal Form (GNF): A Context-free grammar G = (V, T, R, S) is considered to be in Greibach normal form if all production rules of the grammar are of the form $A \rightarrow a\alpha$, where $a \in T$ and $\alpha \in V^*$, i.e., α is a string of zero or more variables.

This normal form has a big advantage in derivation of a string or in parsing method. Since all production rules have body (RHS) which starts with an input symbol, derivation of a given string is very straight forward.

The following steps are to be followed to standardize a context-free grammarinto GNF.

i) Convert the givengrammar into Chomsky Normal Form (CNF).

ii) Eliminate left recursion production rules from the grammar if any.

iii) convert the given production rules of the grammar into GNF form (if not).

Example

$S \rightarrow aXB \mid bAA$

$A \rightarrow aB \mid a$

$B \rightarrow bXA \mid a$

$X \rightarrow a$

Consider the following grammar

$S \rightarrow AB$

$A \rightarrow aA \mid bB \mid b$

$B \rightarrow b$

Since the given grammar has all production rules of the form of CNF and there is no left recursion production rule, conversion will be done using step (iii) that uses substitution method.

$S \rightarrow aAB|bBB|bB$

$A \rightarrow aA|bB|b$

$B \rightarrow b$

15.4 THE PUMPING LEMMA FOR CFGs

In the section, we will discuss about pumping lemma for context-free language. A language is said to be a context-free language (CFL) if all valid strings for the language are derived by a context-free grammar. Definitely this language is more powerful than any regular language. Now to check a given language (set of strings) is CFL or not, we have to try to find out a corresponding context-free grammar to say the given language is context-free language. This task is very trivial. Fortunately, there is a lemma for context-free language which is called pumping lemma.

Pumping Lemma for CFL: If **L** is a context-free language and there is an integer (pumping length) **p** such that any string $\mathbf{w} \in \mathbf{L}$ such that $|\mathbf{w}| \geq \mathbf{p}$, can be written as $\mathbf{w = uvxyz}$, where $\mathbf{vy} \neq \boldsymbol{\varepsilon}$, $\mathbf{|vxy| \ d \leq p}$, then for all $\mathbf{i \geq 0}$, $\mathbf{w^i \in L}$such that $\mathbf{w^i = uv^ixy^iz}$.

Let us to find out whether the language $\mathbf{L = \{0^n1^n2^n \mid n \geq 1\}}$ is context free or not.

Assume**L** is context-free language and nis the pumping length.

Consider a string $z = 0^n1^n2^n$.

We have to break**z** into **uvwxy,** where$\mathbf{|vwx| \ d \leq n}$ **and** $\mathbf{vx \neq \varepsilon}$**.** So, that **vwx** cannot involve both 0s and 2s, since the last 0 and the first 2 are at least (n+1) positions apart.

There are two possible cases.

Case 1 – **vwx** has no 2s. Then **vx** has only 0s and 1s. Then **uwy**, which would have to be in **L**, has **n** 2s, but fewer than **n** 0s or 1s.

Case 2 – **vwx** has no 0s.

Here contradiction occurs which means our assumption was wrong. Hence, **L** is not a context-free language.

15.5 CLOSURE PROPERTIES OF CFL

So far, we have understood that context-free language is more efficient than regular language as it can cover a large set of language. Context-free grammar with its different normal forms helps to provide efficient derivation of strings of a language. We have also learnt about pumping lemma for context-free language. Now we will see different closure properties of context-free language.

Pushdown Automata is a mathematical model which is applied to accept Context Free Language (CFL). Again, context free languages can be generated by context free grammars.

The following properties of Context-free languages are closedover

i) Union

ii) Concatenation

iii) Kleene Star operation

Union Closure: If L_1 and L_2 are two context-free languages then $L_1 \bigcup L_2$ is also a context-free language.

Consider a language $L_1 = \{ a^n b^n, n > 0\}$ and the corresponding grammar G_1has

P: S1 $\rightarrow$ aAb|ab

and another language $L_2 = \{ c^m d^m, m \geq 0\}$ and the corresponding grammar G_2 has

P: S2 $\rightarrow$ cBb| ε

and L is a language which is union of L_1 and L_2 such that

$L = L_1 \bigcup L_2 = \{ a^n b^n \} \bigcup \{ c^m d^m \}$ will be also a context-free language.

The corresponding grammar G for language L will have the additional production ruleas

S $\rightarrow$ S1 | S2

Concatenation Closure: If L_1 and L_2 are two context-free languages, then L_1L_2 is also a context-free language.

Let L is a language which is concatenation of the languages L_1 and L_2, i.e. $L = L_1L_2 = \{ a^nb^nc^md^m \}$ where L_1 and L_2 are context-fee language. Then L will be also a context-free language.

The corresponding grammar G for language L will have the additional production rule as

S '! S1 S2

Kleene Star Closure: If L is a context-free language, then L* is also context-free language.

Consider a context-free language $L = \{ a^nb^n, n \text{ e" } 0\}$ and the correspondingcontext-free grammar G has production rules i.e.

$S \rightarrow aAb| \varepsilon$

and L_1 is language which is Kleene Star of L i.e. $L_1 = \{ a^nb^n\}^*$. Then L_1 is also a context-free language with the corresponding context-free grammar G_1 having additional production rules i.e.

$S1 \rightarrow SS_1 | \varepsilon$

On the other hand, context-free languages are not closed over following operation.

i) Intersection: If L_1 and L_2 are two context-free languages, then $L_1 \bigcap L_2$may not be a context-free language.

ii) Complement: If L_1 is a context-free language, then L_1' (complement of L) may not becontext-free language.



EXERCISE

Q1. Define a context-free language.

Q2. Discuss different normal forms of context-free grammar.

Q3. What is the advantage of normal form of CFG?

Q4. Discuss the closure properties of CFL.

Q5. Discuss pumping lemma for CFL with anapplication.



CHECK YOUR PROGRESS

✓ After reading the unit, you can make summary about context-free grammar and context-free language.

✓ You can try to understand the proof of pumping lemmacontext-free language with suitable example.

✓ Also, you can check the efficiency of context-free language with two normal forms.



15.6 LET US SUM UP

- Context-free grammar is simple and easy to apply in derivation of strings. There are two normal forms of context-free grammar, CNF and GNF which are more standardized with advantages.
- Context-free language is more powerful than regular language. A context-free language can be accepted by push-down automata which has a stack as auxiliary memory.
- Context-free languages have closure properties under union, concatenation and star closure.
- There is a pumping lemma which is a subsidiary or intermediate theorem to proof whether a given language is context-free or not.



15.7 FURTHER READING

1. H. R. Lewis and C. H. Papadimitriou, “Elements of the Theory of Computation”, Pearson Education Asia, 2001.
2. Peter Linz, “An Introduction to Formal Languages and Automata”, NarosaPublisher.
3. J. E. Hopcroft, R. Motwani and J. D. Ullman, “Introduction to Automata Theory, Languages and Computation”, 2nd Ed., Pearson Education, 2000.



7.13 MODEL QUESTIONS

Q 1: How do you define a context-free language? Explain with an example.

Q 2: What is the role of normal form of context-free grammar in derivation of string?

Q 3: Convert the following grammar into Chomsky normal form

$S \rightarrow bAB|BaA$

$A \rightarrow aA|bB|\ b$

$B \rightarrow b$

Q 4: Check the language which has all strings are in the form "$a^n b^n c^n$" such that n>=1 where the alphabet = { 'a', 'b', 'c' } is context-free or not.

*** ***** ***