MCA(S5)18

KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Housefed Complex, Dispur, Guwahati - 781 006

Master of Computer Applications

FORMAL LANGUAGES AND AUTOMATA

CONTENTS

Subject Expert

Prof. Anjana Kakati Mahanta, Deptt. of Computer Science, Gauhati University

Prof. Jatindra Kr. Deka, Deptt. of Computer Science and Engineering,
Indian Institute of Technology, Guwahati

Prof. Diganta Goswami, Deptt. of Computer Science and Engineering,
Indian Institute of Technology, Guwahati

Course Coordinator

Tapashi Kashyap Das, Assistant Professor, Computer Science, KKHSOU

Arabinda Saikia, Assistant Professor, Computer Science, KKHSOU

SLM Preparation Team

Units	Contributor
1 , 2, 3, 4	**Naba Jyoti Sarmah,** Guest Faculty, Deptt. of Computer Science, Gauhati University
5, 6, 7, 8	**Pranab Das**, Asst. Professor, Deptt. of Computer Science and Information Technology, Don Bosco College of Engineering and Technology, Azara, Guwahati

July 2013

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

The university acknowledges with thanks the financial support provided by the **Distance Education Council, New Delhi**, for the preparation of this study material.

Housefed Complex, Dispur, Guwahati- 781006; Web: www.kkhsou.net

COURSE INTRODUCTION

This is a course on ***Formal Languages and Automata***. Automata theory is the study of abstract computing devices or machines. In this course, we look at models that represent features at the core of all computers and their applications. To model the hardware of a computer, we introduce the notion of an automaton. An automaton is a contruct that processes all the indispensable features of a digital computer. A formal language is an abstraction of the general characteristics of programming languages.

This course contains eight essential units. The first unit is an introductory unit on finite automaton. The second unit is on Regular expressions. The third unit is on Regular languages and their properties. The fourth unit focuses on context-free grammar and languages. The fifth unit concentrates on push-down automata. The sixth unit discusses the properties of Context-free languages. The seventh unit gives us an introduction to turing machines. The eight unit is the last unit and it discusses the undecidability.

While going through a unit, you will notice some boxes along-side, which have been included to help you know some of the difficult, unseen terms. Some “ACTIVITY’ (s) have been included to help you apply your own thoughts. Again, we have included some relevant concepts in “LET US KNOW” along with the text. And, at the end of each section, you will get “CHECK YOUR PROGRESS” questions. These have been designed to self-check your progress of study. It will be better if you solve the given problems in these boxes immediately, after you finish reading the section in which these questions occur and then match your answers with “ANSWERS TO CHECK YOUR PROGRESS” given at the end of each unit.

MASTER OF COMPUTER APPLICATIONS

Formal Languages and Automata

DETAILED SYLLABUS

Unit 1: Introduction to Finite Automata (Marks: 15)
Introduction to Finite Automata; The central concepts of Automata theory; Deterministic finite automata; Nondeterministic finite automata.

Unit 2: Finite Automata and Regular Expressions (Marks:15)
An application of finite automata; Finite automata with Epsilon-transitions; Regular expressions; Finite Automata and Regular Expressions; Applications of Regular Expressions.

Unit 3: Regular Languages and Properties of Regular Languages (Marks:15)
Regular languages; Proving languages not to be regular languages; Closure properties of regular languages; Decision properties of regular languages; Equivalence and minimization of automata.

Unit 4: Contex-Free Grammars and Languages (Marks:15)
Context –free grammars; Parse trees; Applications; Ambiguity in grammars and Languages.

Unit 5: Pushdown Automata (Marks: 12)
Definition of the Pushdown automata; The languages of a PDA; Equivalence of PDA's and CFG's; Deterministic Pushdown Automata.

Unit 6: Properties of Context-Free Languages (Marks:12)
Normal forms for CFGs; The pumping lemma for CFGs; Closure properties of CFL

Unit 7: Introduction to Turing Machine (Marks: 8)
Problems that Computers cannot solve; The turning machine; Programming techniques for Turning Machines; Extensions to the basic Turning Machines; Turing Machine and Computers.

Unit 8: Undecidability (Marks:8)
A Language that is not recursively enumerable; An Undecidable problem that is RE; Post's Correspondence problem; other undecidable problems.

UNIT - 1: INTRODUCTION TO FINITE AUTOMATA

UNIT STRUCTURE

1.1 LEARNING OBJECTIVES

After going through this unit, you will be able to

- understand the basic concept of automata theory
- requirement of automata theory
- basic terms related to automata theory
- define DFA
- define NFA

1.2 INTRODUCTION

The main objective of this course is to study limitations of computers and computation. We are going to investigate limitations of computers and computations by studying the essence of computers and computations rather than all the variations of computer and computation. This essence is a device called Turing machine. It was first conceived of by Alan Turing in early 20-th century. It is a very simple device but remarkably, every task modern computers perform can also be accomplished by Turing machines. Though it has not been proven, it is generally believed that any "computation" humans do can be done by Turing machines and that "computation" is the computation performed by Turing machines. Thus by studying Turing machines we can learn capabilities hence limitations of computers.

Before proceeding to the study of Turing machines and their computations in this course, we study a simpler type of computing device called finite automata. Finite automata are very similar to Turing machines but a few restrictions are imposed on them. Consequently they are less capable than Turing machines but then their operations are simpler. So they provide a good introduction to our study of Turing machines. In addition finite automata can model a large number of systems used in practice. Thus they are a powerful tool to design and study those systems with.

Our first and one of the main topic for this course is language. A language is, in this course, a set of strings of symbols. Programming languages we use are a language in that sense. Others such as languages of logics, languages of mathematics, natural languages etc. are all languages in that sense.

What we are going to study on languages in this course are four classes of languages called (Chomsky) formal languages and their properties. The four classes are regular (or type 3) languages, context-free (or type 2) languages, context-sensitive (or type 1) languages and phrase structure (or type 0) languages.

These formal languages are characterized by grammars which are essentially a set of rewrite rules for generating strings belonging to a language as we see later. Also there are various kinds of computing devices called automata which process these types of languages

These formal languages and automata capture the essence of various computing devices and computation in a very simple way. Also for some important classes of problems, solving them can be seen as recognizing languages i.e. checking whether or not a string is in a language.

1.3 SOME BASIC DEFINITIONS

Symbol or letters: Symbols are indivisible objects or entity 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 $\sum$. The elements of $\sum$ are called letters.
Example:

$\sum$ = {0,1}
$\sum$ = {a, b, c}
$\sum$ = {a, b, c,…,z}
$\sum$ = {%, ^, &, *, $, #}

Word or String: A word or string over an alphabet $\sum$ is a finite sequence of concatenated symbols of $\sum$.
Example: if $\sum = \{0, 1\}$ is the given alphabet then the sequence 01, 001, 101, 1001, 11110001 are words on $\sum$ but 012 is not a word since 2 is not an element of $\sum$.

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 $\sum = \{0,1\}$ 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, the concatenation of *x* and *y* denoted by *xy* is the string that has a 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 $\sum$ and if we can write that ω = 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 a suffix of ω and any string u is a substring of ω if ω=xuy. The empty string ε is always a substring of any string. For example if ω=010011 is a string over $\sum = \{0,1\}$ 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 string formed by sequentially concatenating *n* copies of *x*. in other words $x^n = \varepsilon$, if $n = 0$; otherwise $x^n = xx^{n-1}$. For example if $x = 01$ then $x^3 = 010101$.

Power of Alphabets: We write $\sum^k$ (for some integer k) to denote the set of strings of length k with symbols from $\sum$. In other words, $\sum^k = \{ x \mid x \text{ is a string over } \sum \text{ and } |x| = k\}$. Hence, for any alphabet, $\sum^0$ denotes the set of all strings of length zero. That is, $\sum^0 = \{ \varepsilon \}$. For the binary alphabet { 0, 1 } we have the following-

$\sum^0 = \{ \varepsilon \}$
$\sum^1 = \{ 0, 1\}$
$\sum^2 = \{ 00, 01, 10, 11\}$
$\sum^3 = \{ 000, 001, 010, 011, 100, 101, 110, 111\}$

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

$\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup \ldots \cup \sum^n \cup \ldots$
$= \cup \sum^k$

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

Example : If $\sum = \{ a, b \}$, then
$\sum^* = \{ \varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, \ldots\}$.
The set of all nonempty strings over an alphabet $\sum$ is denoted by $\sum^+$. That is,
$\sum^+ = \sum^1 \cup \sum^2 \cup \ldots \cup \sum^n \cup \ldots$

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 $\sum$ is a collection of words over $\sum$. Since $\sum^*$ is the set of all words on $\sum$. Thus, a language L is simply a subset of $\sum^*$.
For example if $\sum = \{0,1\}$ is a alphabet then
$L_1 = \{0, 01, 01^2, 01^3, \ldots\}$
L_1 consist of all the strings starting with a 0 followed by any number of 1
$L_2 = \{0^m1^m : m>0\}$
L_2 consist of words beginning with one or more 0's followed by same number of 1's.
$L_3 = \{0^m1^n : m>0, n>0\}$
L_3 consist of words beginning with one or more 0's followed by one or more 1's.
$L_4 = \{\varepsilon\}$
L_4 is an empty language.

Operations on Languages

Since languages are set of strings we can apply set operations to languages.

Union : If L_1 and L_2 are two languages then the union of L_1 and L_2 denoted by $L_1 \cup L_2$, any word $x \in L_1 \cup L_2$ iff $x \in L_1$ or $x \in L_2$.
Example: $\{0, 11, 01, 011\} \cup \{1, 01, 110\} = \{0, 11, 01, 011, 111\}$

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, $\sum^*$ is referred as the universe of all the languages over the alphabet over $\sum$. So complement of any language is taken with respect to $\sum^*$. Thus for a language L, the complement is $\bar{L} = \{x \in \sum^* \text{ and } x \notin L\}$.
Example: Let $L = \{ x : |x| \text{ is even } \}$. Then its complement is the language $\bar{L} = \{x \in \sum^* : |x| \text{ is odd } \}$. 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 -

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

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 $\sum$ 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. $L_1L_2 \neq L_2 L_1$ in general.
2. $L\emptyset = \emptyset$
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**:**

$L^* = \cup L^k$

$= L^0 \cup L^1 \cup L^2 \cup \ldots \cup L^n \cup \ldots$

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

Also

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

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

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

$= \{e\} \cup \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots$

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

$= \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots$

1.4 GRAMMAR

A grammar is a mechanism used for describing languages. This is one of the most simple but yet powerful mechanism.

In everyday language, like English, we have a set of symbols (alphabet), a set of words constructed from these symbols, and a set of rules using which we can group the words to construct meaningful sentences. The grammar for English tells us what are the words in it and the rules to construct sentences. It also tells us whether a particular sentence is well-formed (as per the grammar) or not.

These concepts are generalized in formal language leading to formal grammars. The word 'formal' here refers to the fact that the specified rules for the language are explicitly stated in terms of what strings or symbols can occur.

Formal definitions of a Grammar

A grammar *G* is defined as a quadruple.

$G = (N, \Sigma, P, S)$

N is a non-empty finite set of non-terminals or variables,

Σ is a non-empty finite set of terminal symbols such that $N \cap \Sigma = \Phi$

$S \in N$, is a special non-terminal (or variable) called the start symbol, and $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ is a finite set of production rules.

The binary relation defined by the set of production rules is denoted by $\rightarrow$, i.e. $\alpha \rightarrow \beta$ iif $(\alpha, \beta) \in P$.

In other words, P is a finite set of production rules of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq (N \cup \Sigma)^+$ and $\beta \subseteq (N \cup \Sigma)^*$

Automata and Grammars

The production rules specify how the grammar transforms one string to another. Given a string $\delta\alpha y$, we say that the production rule $\alpha \rightarrow \beta$ is applicable to this string, since it is possible to use the rule $\alpha \rightarrow \beta$ to rewrite the α (in $\delta\alpha y$) to β obtaining a new string $\delta\beta y$. We say that $\delta\alpha y$ derives $\delta\beta y$ and is denoted as

$\delta\alpha y \Rightarrow \delta\beta y$

Successive strings are derived by applying the productions rules of the grammar in any arbitrary order. A particular rule can be used if it is applicable, and it can be applied as many times as described.

We write $\alpha \overset{*}{\Rightarrow} \beta$ if the string β can be derived from the string α in zero or more steps; $\alpha \overset{+}{\Rightarrow} \beta$ if β can be derived from α in one or more steps.

By applying the production rules in arbitrary order, any given grammar can generate many strings of terminal symbols starting with the special start symbol, *S*, of the grammar. The set of all such terminal strings is called the language generated (or defined) by the grammar.

Formally, for a given grammar G = (N, Σ, P, S) the language generated by *G* is

$L(G) = \{ \omega \in \Sigma^* \mid S \overset{*}{\Rightarrow} \omega \}$

That is $\omega \in L(G)$ iff $S \overset{*}{\Rightarrow} \omega$.

If $\omega \in L(G)$, we must have for some $n \geq 0$, $S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \ldots \Rightarrow \alpha_n = \omega$ denoted as a derivation sequence of ω, The strings $S = \alpha 1$, $\alpha_2, \alpha_3, \ldots \alpha_n = \omega$ are denoted as sentential forms of the derivation.

Example : Consider the grammar G = (N, Σ, P, S), where *N* = {*S*}, Σ={*a*, *b*} and P is the set of the following production rules

{ *S*→*ab*, *S*→*aSb*}

Some terminal strings generated by this grammar together with their derivation is given below.

$S \Rightarrow ab$

$S \Rightarrow aSb \Rightarrow aabb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$

It is easy to prove that the language generated by this grammar is

$L(G) = \{a^i b^i \mid i \geq 1\}$

By using the first production, it generates the string ab (for *i* =1).

To generate any other string, it needs to start with the production S→aSb and then the non-terminal S in the RHS can be replaced either by ab (in which we get the string aabb) or the same production S→aSb can be used one or more times. Every time it adds an '*a*' to the left and a '*b*' to the right of S, thus giving the

sentential form a^iSbi, $i \geq 1$. When the non-terminal is replaced by ab (which is then only possibility for generating a terminal string) we get a terminal string of the form a^ib^i, $i \geq 1$

There is no general rule for finding a grammar for a given language. For many languages we can devise grammars and there are many languages for which we cannot find any grammar.

Example: Find a grammar for the language L= $\{a^nb^{n+1} \mid n \geq 1\}$

It is possible to find a grammar for L by modifying the previous grammar since we need to generate an extra *b* at the end of the string a^nb^n, $n \geq 1$. We can do this by adding a production $S \rightarrow Bb$ where the non-terminal *B* generates a^ib^i, $i \geq 1$ as given in the previous example.

Using the above concept we devise the following grammar for *L*.

G = (N, Σ, P, S), where

$N = \{ S, B \}$

$P = \{ S \rightarrow Bb, B \rightarrow ab, B \rightarrow aBb \}$

CHECK YOUR PROGRESS 1

State True or False

1. $L^* = L^+L$
2. |aa|=2 if Σ ={aa, b}
3. L^0 = empty set for any Language L
4. $\{a,b\}^k$ is set of all strings of length k
5. $(L^*)^* = L^*$
6. L^+ is a subset of L^*
7. If $L^* = L^+$ then L is a empty set
8. $L_1(L_2 \cap L_3) = L_1L_2 \cap L_1L_3$

1.5 DETERMINISTIC FINITE AUTOMATA

Finite Automata: Automata (singular**:** automation) are a particularly simple, but useful, model of computation. They were initially proposed as a simple model for the behavior of neurons. The concept of a finite automaton appears to have arisen in the 1943 paper "A logical calculus of the ideas immanent in nervous activity", 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.

We said that automata are a model of computation. That means that they are a simplified abstraction of the real thing. 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.

Deterministic Finite Automata

Informally, 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. As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far.

Thus, 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 $\sum$.

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),
    - a current state, initially set to a start state,
    - a state transition function for changing the current state.

An automaton processes a 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 tape cell and sends the symbol s found there to the control. Then the tape head moves to the next cell.
2. The control takes s and the current state and consults the state transition function to get the next state, which becomes the new current state.

Once the entire string has been processed, the state in which the automation enters is examined. If it is an accept state, the input string is accepted; otherwise, the string is rejected. Summarizing all the above we can formulate the following formal definition:

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

- 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 "next state" transition function. Intuitively, δ is a function that tells which state to move to in response to an input, i.e., if M is in state q and sees input a, it moves to state $\delta(q, a)$.
- $q_0 \in Q$ 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").

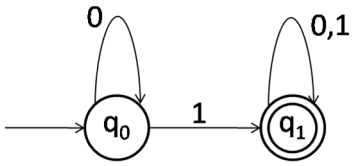
- Rows correspond to states,
- Columns correspond to input symbols,
- Entries correspond to next states
- The start state is marked with an arrow
- The accept states are marked with a star (*).

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



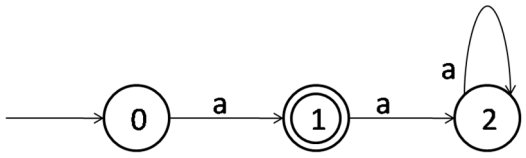
Here is an informal description how a DFA operates. An input to a DFA can be any string $\omega \in \Sigma^*$. Put a pointer to the start state q. Read the input string ω from left to right, one symbol at a time, moving the pointer according to the transition function, δ. If the next symbol of ω is *a* and the pointer is on state *p*, move the pointer to 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 $r \in F$ and rejected if $r \notin F$. Note that there is no formal mechanism for moving the pointer.

A language $L \in \Sigma^*$ 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.

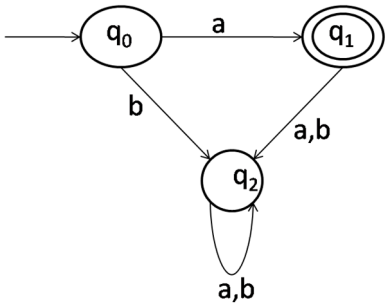


Example 2: Q = { 0, 1, 2 }, Σ = { a, b }, F = { 1 }, the initial state is 0 and δ is as shown in the following table.

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

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.

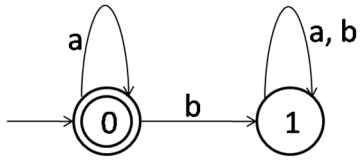
A state transition diagram for this DFA is given below.



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

It is convenient to introduce the extended transition function $\delta^* : Q \times \Sigma^* \rightarrow Q$. 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 we can define δ^* recursively by

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

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

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

String accepted by DFA

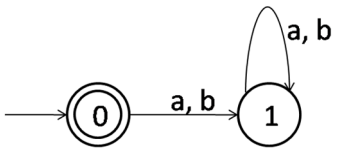
A string ω is accepted by a DFA $< Q , \Sigma , q_0 , \delta , F >$, if and only if $\delta^*(q_0 , \omega) \in F$. That is a string is accepted by a DFA if and only if the DFA starting at the initial state ends in an accepting state after reading the string.

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

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

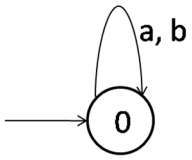
Example 1:



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

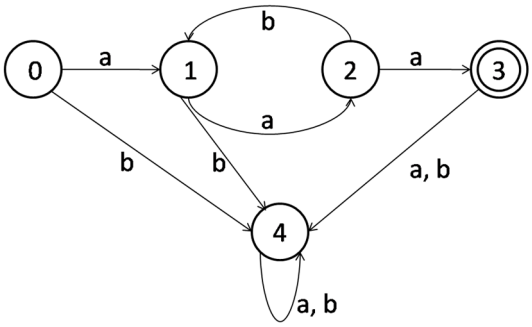
nothing else because any non-empty symbol would take it to state 1, which is not an accepting state, and it stays there.

Example2:



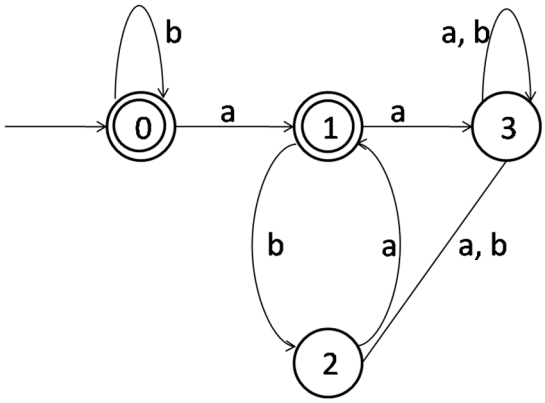
This DFA does not accept any string because it has no accepting state. Thus the language it accepts is the empty set Φ.

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 go to state 1 by reading one a. Then from state 1 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:

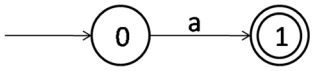


This DFA has two accepting states: 0 and 1. Thus the language that is accepted by this DFA is the union of the language accepted at

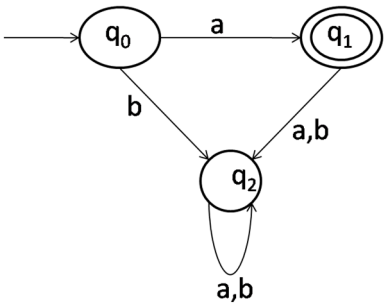
state 0 and the one accepted at state 1. The language accepted at state 0 is b^*. To find the language accepted at state 1, first at state 0 read any number of b's. Then go to state 1 by reading one a. At this point (b^*a) will have been read. At state 1 go through the cycle 1 - 2 - 1 any number of times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

1.6 NONDETERMINISTIC FINITE AUTOMATA

In the previous section we have seen DFAs that accept some simple languages such as ∅ , {ε} , and { a }. As you might have noticed, those DFAs have states and transitions which do not contribute to accepting strings and languages. For example all we need about an FA that accepts {a } is the following regardless of the alphabet (whether be it { a } , { a , b } or any other).



If Σ= {a, b}, it is not a DFA. A DFA that accepts { a } from Σ= {a, b} would need more states and transitions as shown in the example below



To avoid those redundant states and transitions and to make modeling easier we use finite automata called nondeterministic finite automata (NFA). Below we are going to formally define nondeterministic finite automata and see 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 with efficient nondeterministic solutions but no known efficient deterministic solutions. (Travelling salesman, Hamiltonian cycle, clique, etc). 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 than 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 this subject.

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

- multiple next state.
- 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. (Recall that, in a DFA there is exactly one start state and exactly one transition out of every state for each symbol in Σ).

This means that, in a state *q* and with input symbol 0, there could be one, more than one or zero next state to go, i.e. the value of $\delta(q,a)$ is a subset of *Q*. Thus $\delta(q,a) = (q_1, q_2,\ldots ,q_k)$ which means that any one of $q_1, q_2,\ldots ,q_k$ could be the next state.

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

ε- transitions :

In an -transition, the tape head doesn't do anything- it doesn't read and it doesn't move. However, the state of the automata can be changed - that is can go to zero, one or more states. This is written formally as $\delta(q, \varepsilon) = (q_1, q_2, \ldots, q_k)$ implying that the next state could by any one of $q_1, q_2,\ldots ,q_k$ without consuming the next input symbol.

Acceptance :

Informally, an NFA is said to accept its input **ω** if it is possible to start in some start state and process **ω**, moving according to the transition rules and making choices along the way whenever the next state is not uniquely defined, such that when **ω** is completely processed (i.e. end of **ω** is reached), the automata is in an accept state. There may be several possible paths through the automation in response to an input **ω** since the start state is not determined and there are choices along the way because of multiple next states. Some of these paths may lead to accept 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 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 (including the **ω** -transitions) in response to an input symbol we can think that the automation is having some "guessing" power to chose the correct one in case the input is accepted.

Formal definition of NFA :

Formally, an NFA is a quadruple $N = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0, and F bear the same meaning as for a DFA, but δ, the transition function is redefined as follows:

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

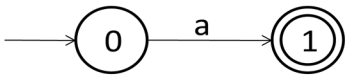
where $P(Q)$ is the power set of Q i.e. 2^Q.

- As in the case of DFA the set Q in the above definition is simply a set with a finite number of elements. Its elements can be interpreted as a state that the system (automaton) is in.
- The transition function is also called a **next state function**. Unlike DFAs an NFA moves into one of the states given by $\delta(q, a)$ if it receives the input symbol 'a' while in state q. Which one of the states in $\delta(q, a)$ to select is determined nondeterministically.
- Note that δ is a function. Thus for **each state** q of Q and for **each symbol** 'a' of Σ, $\delta(q, a)$ must be specified. But it can be the empty set, in which case the NFA aborts its operation.
- As in the case of DFA the accepting states are used to distinguish sequences of inputs given to the finite automaton. If the finite automaton is in an accepting state when the input ends i.e. ceases to come, the sequence of input symbols given to the finite automaton is "accepted". Otherwise it is not accepted.
- Note that any DFA is also a NFA.

Example 1: $Q = \{ 0, 1 \}$, $\Sigma = \{ a \}$, $F = \{ 1 \}$, 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.



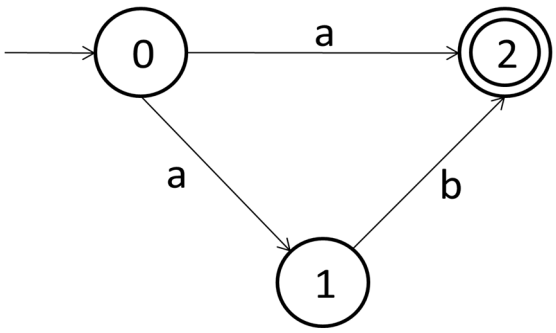
If the alphabet Σ is changed to {a, b} instead of { a }, this is still an NFA that accepts { a } .

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 extend the transition function, denoted as $\boldsymbol{\delta}^*$, takes a state q ∈ Q and a string ω ∈ Σ^*, and returns the set of states, $S \subseteq Q$, that the NFA is in after processing the string ω if it starts in state *q*.

Formally, $\boldsymbol{\delta}^*$ is defined as follows:

1. $\boldsymbol{\delta}^*(q, \varepsilon) = \{q\}$ that is, without rending any input symbol, an NFA doesn't change state.
2. Let ω = *xa* for some ω, x ∈ Σ^* and a ∈ Σ. Also assume that

   $\delta^*(q, x) = \{p_1, p_2, \ldots, p_k\}$

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

That is, $\delta(q, \omega)$ can be computed by first computing $\delta^*(q_0, x)$, and by then following any transitive from any of these stats that is labeled *a*.

The Language accepted by an NFA :

From the discussion of the acceptance by an NFA, we can give the formal definition of a language accepted by an NFA as follows:

If $N = (Q, \Sigma, \delta, q_0, F)$ is an NFA, then the language accepted by *N* is written as *L*(*N*) is given by

$L(N) = \{\omega \mid \delta^*(q_0, \omega) \cap F = \Phi \}$

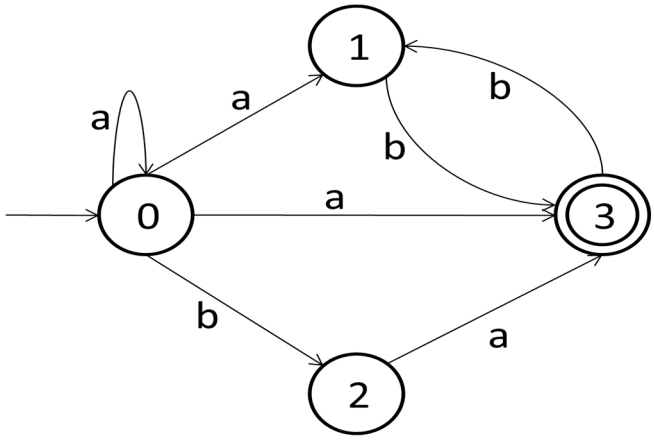
That is, *L*(*N*) is the set of all strings ω in Σ^* such that $\delta^*(q_0, \omega)$ contains at least one accepting state.

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 2

State True or False

1. In a DFA all states have the same number of transition
2. A DFA cannot have more than one accepting states
3. A DFA has finite number of states
4. A DFA accepts a Language
5. In a DFA $\delta^*(0,abb) = 0$ is possible
6. In a NFA all states have the same number of transition
7. An NFA cannot have more than one accepting states
8. An NFA has finite number of states
9. In an NFA $\delta^*(0,a) = \delta(0,a) = \{1,2\}$ is possible
10. In an NFA $\delta^*(0,abaa)) = \{1,2\}$ is possible

1.7 LET US SUM UP

- A language L over an alphabet $\sum$ is a collection of words over $\sum$. Since $\sum^*$ is the set of all words on $\sum$. Thus, a language L is simply a subset of $\sum^*$.
- A grammar is a mechanism used for describing languages. This is one of the most simple but yet powerful mechanism.
- 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 $M =(Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M, in formal notation

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

- 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.

1.8 FURTHER READINGS

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.

1.9 ANSWERS TO CHECK YOUR PROGRESS

CHECK YOUR PROGRESS – 1

1. False
2. False
3. False
4. True
5. True
6. True
7. False
8. False

CHECK YOUR PROGRESS – 2

1. True
2. False
3. True
4. True
5. False
6. False
7. False
8. True
9. True
10. True

1.10 PROBABLE QUESTIONS

1. Prove that $(xy)^R = y^R x^R$, for all x, y $\in \Sigma^*$
2. Consider the language L={ 01, 11, 011}. Which of the following strings are in L^*

   010101, 0001, 110, 010111101, 0111111110, 11010111111101, 110111110011, 11101101?
3. 4. Let L_1={ 00,11} and L_2={ ε, 0, 01 }

   a) List the strings in the set L_1L_2.

   b) List the strings of the set L_2^* of length three or less.

   c) How many strings of length 5 are there in L_1^*?
4. Design DFA and NFA to recognize the following set of strings

   abb, abaa, ab^*, a^*b assuming that Σ = {a,b}

*****

UNIT - 2: FINITE AUTOMATA AND REGULAR EXPRESSIONS

UNIT STRUCTURE

2.1 LEARNING OBJECTIVES

After going through this unit, you will be able to

- define NFA with ε transition
- define regular expression
- application of regular expression
- application of finite automata

2.2 INTRODUCTION

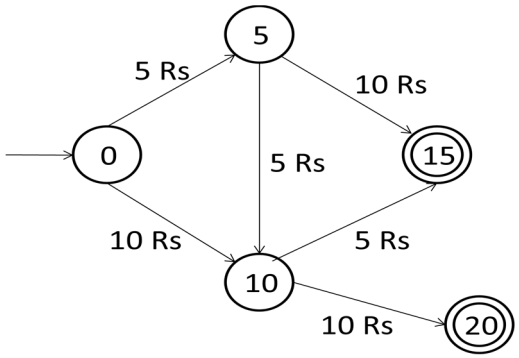
In this chapter we are going to learn about regular expression which is one of the ways to describe regular languages and different operations on regular expression. Here we also going to look at some of the application of finite automata and regular expression. One of the objectives of this chapter is to show that there is a one-to-one correspondence between regular languages and finite automata. We are going to do that by showing that a finite automaton can be constructed from a given regular expression by combining simpler FAs using union, concatenation and Kleene star operations. These operations on FAs can be described conveniently if ε-Transitions are used. Basically an NFA with ε-Transitions is an

NFA but can respond to an empty string ε and move to the next state. Here we are going to formally define NFA with ε-Transitions (abbreviated as NFA-ε) and see some examples. As we are going to see later, for any NFA-ε there is a NFA (hence DFA) which accepts the same language and vice versa.

2.3 APPLICATION OF FINITE AUTOMATA

Soft drink vending machine

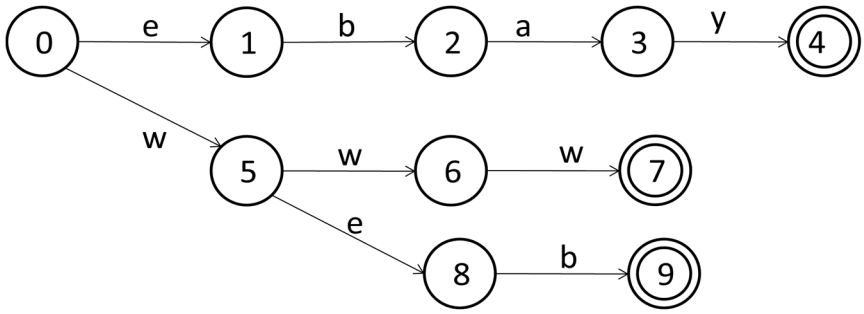
Let us consider the operation of a soft drink vending machine which charges 15 Rs for a can. The machine initially waiting for a customer to come and put some coins, that is, waiting-for-customer state. For simplicity let us assume that only 5 Rs and 10 Rs coins are used. When a customer comes and puts in the first coin, say 5 Rs, machine no longer in the waiting-for-customer state. Now it has received 5 Rs and waiting 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 in another state, say 15-Rs state. When the customer selects a soft drink, machine delivers the soft drink. After that it back to its initial state that 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 read 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, denote it by Q, after reading a digit and stays there as long as digits are read. This Q is an accepting state.

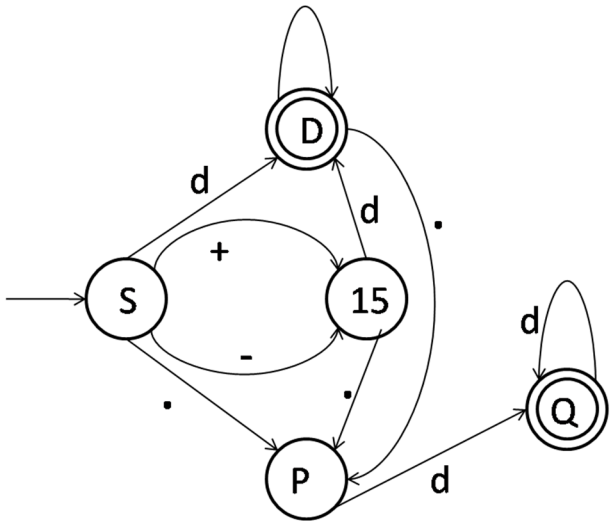
On the other hand if a digit has been read before a decimal point, i.e. it is in state D, then it can continue receiving digits and stay in D. D is another accepting state. If a decimal point is read while in D, then it goes to state P indicating that a decimal point has been read.

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 ∈ { 0 , 1 , 2 , . . . , 9 } . This system can be modeled by the following finite automaton:



2.4 NFA WITH ε TRANSITION

Definition of nondeterministic finite automaton with ε-Transitions

Let Q be a finite set and let Σ be a finite set of symbols. Also let δ be a function from Q x Σ ∪ {ε} to 2^Q, let q_0 be a state in Q and let F be a subset of Q. We call the elements of Q a state, δ the transition function, q_0 the initial state and F the set of accepting states.

Then a nondeterministic finite automaton with ε-Transitions is a 5-tuple < Q , Σ , q_0 , δ , 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 a NFA-ε.

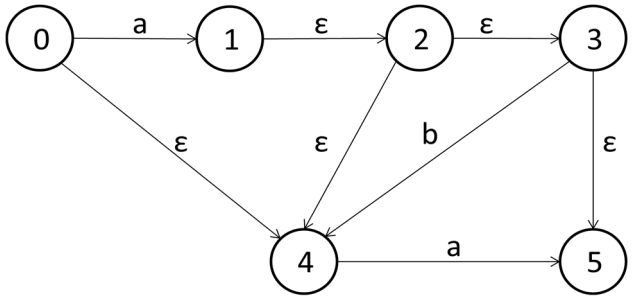
Example of NFA-ε

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	E	{ 4 }
1	E	{ 2 }
2	E	{ 3, 4 }

3	E	{ 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

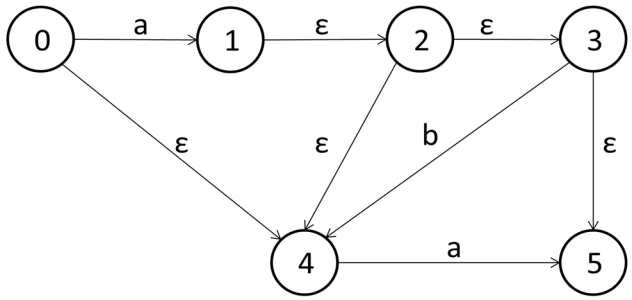
Let $< Q , \Sigma , q_0 , \delta , A >$ be an NFA-ε . Let us denote the ε-closure of a set S of states of Q by ε(S). Then ε(S) is defined recursively as follows:

Basis Clause: $S \subseteq \varepsilon(S)$

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

$$\delta(q , \varepsilon) \subseteq \varepsilon(S) .$$

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 } ⊆ ε ({2}), that is, 2 ∈ ε ({2}). Then since 2 ∈ ε ({2}), by the Inductive Clause,

$\delta(2, \varepsilon) \subseteq \varepsilon(\{2\})$.

Since $\delta(2, \varepsilon) = \{3, 4\}$, we now have $\{2, 3, 4\} \subseteq \varepsilon(\{2\})$.

Since 3 and 4 have been added to ε ({2}) , $\delta(3, \varepsilon) = \{5\}$ and $\delta(4, \varepsilon) = \Phi$ must be included in ε ({2}).

Thus now $\{2, 3, 4, 5\} \subseteq \varepsilon(\{2\})$.

Though 5 has become a member of the closure, since $\delta(5, \varepsilon)$ is empty, no new members are added to ε ({2}). Since $\delta(q, \varepsilon)$ 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 δ repeatedly 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,

$\delta^*(q, \varepsilon) = \varepsilon(\{q\})$.

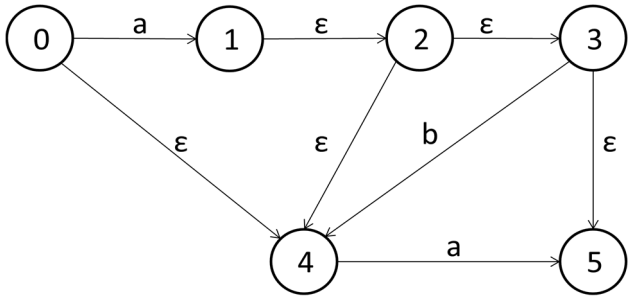
Inductive Clause: For any state q, a string y in Σ^* and a symbol ‘a’ in Σ,

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

What the Inductive Clause means is that $\delta^*(q, ya)$ is obtained by first finding the states that can be reached from q by reading y ($\delta^*(q, y)$), 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 δ^*(0, a). Since ε({1}) = {1, 2, 3} and ε({5}) = {5},

δ^*(0, a) = {1, 2, 3, 5}.

Then to find δ^*(0, ab) read b from each of the states in δ^*(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}, δ^*(0, ab) = {3, 4}.

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

The language accepted by an NFA- ε < Q , Σ, q_0 , δ , F > is the set of strings accepted by the NFA-ε.

CHECK YOUR PROGRESS 1

State True or False

1. In an NFA-ε all states have the same number of transition
2. An NFA-ε cannot have more than one accepting states
3. An NFA-ε has finite number of states
4. An NFA-ε can modify its input
5. When a is read at q, NFA-ε goes to a state in δ(q,a)

2.5 REGULAR LANGUAGE

The set of regular languages over an alphabet Σ is defined recursively as below. Any language belonging to this set is a **regular language** over Σ.

Definition of Set of Regular Languages:

1. ϕ, {ε} and {a} for any symbol a ∈ Σ are regular languages.
2. If L_r and L_s are regular languages, then $L_r \cup L_s$, $L_r L_s$ and L_r^* are regular languages.
3. Nothing is a regular language unless it is obtained from the above two.

For example, let Σ = {a, b}. Then since {a} and {b} are regular languages, {a, b} ({a}∪{b}) and {ab} ({a}{b}) are regular languages. Also since {a} is regular, $\{a\}^*$ is a regular language which is the set of strings consisting of a's such as ε, a, aa, aaa, aaaa etc. Note also that Σ^*, which is the set of strings consisting of a's and b's, is a regular language because {a, b} is regular.

Regular expression

Regular expressions are used to denote regular languages. They can represent regular languages and operations on them succinctly.

The set of regular expressions over an alphabet Σ is defined recursively as below. Any element of that set is a **regular expression**.

1. ϕ , ε and **a** are regular expressions corresponding to languages ϕ , $\{\varepsilon\}$ and {a}, respectively, where a is an element of Σ.
2. If **r** and **s** are regular expressions corresponding to languages L_r and L_s, then **(r+s)**, **(rs)** and $(\mathbf{r}^*)$ are regular expressions corresponding to languages $L_r \cup L_s$, $L_r L_s$ and L_r^* respectively.
3. Nothing is a regular expression unless it is obtained from the above two.

Conventions on regular expressions

1. The operation ***** has precedence over concatenation, which has precedence over union. Thus the regular expression $\mathbf{(a+(b(c^*)))}$ is written as $\mathbf{a + bc^*}$
2. The concatenation of k **r**'s , where **r** is a regular expression, is written as $\mathbf{r^k}$. Thus for example $\mathbf{rr} = \mathbf{r^2}$. The language corresponding to $\mathbf{r^k}$ is L_r^k, where L_r is the language corresponding to the regular expression **r**.
3. We use $\mathbf{(r^+)}$ as a regular expression to represent L_r^+ .

Examples of regular expression and regular languages corresponding to them

- $\mathbf{(a + b)^2}$ corresponds to the language {aa, ab, ba, bb}, that is the set of strings of length 2 over the alphabet {a, b}. In general $\mathbf{(a + b)^k}$ corresponds to the set of strings of length k over the alphabet {a, b}. $\mathbf{(a + b)^*}$ corresponds to the set of all strings over the alphabet {a, b}.
- $\mathbf{a^*b^*}$ corresponds to the set of strings consisting of zero or more a's followed by zero or more b's.
- $\mathbf{a^*b^+a^*}$ corresponds to the set of strings consisting of zero or more a's followed by one or more b's followed by zero or more a's.
- $\mathbf{(ab)^+}$ corresponds to the language {ab, abab, ababab, ... }, that is, the set of strings of repeated ab's.

Note: A regular expression is not unique for a language. That is, a regular language, in general, corresponds to more than one regular expressions. For example $\mathbf{(a + b)^*}$ and $\mathbf{(a^*b^*)^*}$ correspond to the set of all strings over the alphabet {a, b}.

Definition of Equality of Regular Expressions

Regular expressions are **equal** if and only if they correspond to the same language.

Thus for example $\mathbf{(a + b)^*} = \mathbf{(a^*b^*)^*}$, because they both represent the language of all strings over the alphabet {a, b}.

In general, it is not easy to see by inspection whether or not two regular expressions are equal.

Ex. 1: Find the shortest string that is not in the language represented by the regular expression $\mathbf{a^*(ab)^*b^*}$.

Solution: It can easily be seen that ε, a, b, which are strings in the language with length 1 or less. of the strings with length 2 aa, bb and ab are in the language. However, ba is not in it. Thus the answer is ba.

Ex. 2: For the two regular expressions given below,

(a) find a string corresponding to $\mathbf{r_2}$ but not to $\mathbf{r_1}$ and
(b) find a string corresponding to both $\mathbf{r_1}$ and $\mathbf{r_2}$.

$\mathbf{r_1 = a^* + b^*}$
$\mathbf{r_2 = ab^* + ba^* + b^*a + (a^*b)^*}$

Solution: (a) Any string consisting of only a's or only b's and the empty string are in $\mathbf{r_1}$. So we need to find strings of $\mathbf{r_2}$ which contain at least one a and at least one b. For example ab and ba are such strings.
(b) A string corresponding to $\mathbf{r_1}$ consists of only a's or only b's or the empty string. The only strings corresponding to $\mathbf{r_2}$ which consist of only a's or b's are a, b and the strings consisting of only b's (from $\mathbf{(a^*b)^*}$).

Ex. 3: Let $\mathbf{r_1}$ and $\mathbf{r_2}$ be arbitrary regular expressions over some alphabet. Find a simple (the shortest and with the smallest nesting of * and +) regular expression which is equal to each of the following regular expressions.

(a) $\mathbf{(r_1 + r_2 + r_1r_2 + r_2r_1)^*}$
(b) $\mathbf{(r_1(r_1 + r_2)^*)^+}$

Solution: One general strategy to approach this type of question is to try to see whether or not they are equal to simple regular expressions that are familiar to us such as $\mathbf{a}$, $\mathbf{a^*}$, $\mathbf{a^+}$, $\mathbf{(a + b)^*}$, $\mathbf{(a + b)^+}$ etc.

(a) Since $\mathbf{(r_1 + r_2)^*}$ represents all strings consisting of strings of $\mathbf{r_1}$ and/or $\mathbf{r_2}$, $\mathbf{r_1r_2 + r_2r_1}$ in the given regular expression is redundant, that is, they do not produce any strings that are not represented by $\mathbf{(r_1 + r_2)^*}$. Thus $\mathbf{(r_1 + r_2 + r_1r_2 + r_2r_1)^*}$ is reduced to $\mathbf{(r_1 + r_2)^*}$.

(b) $(\mathbf{r_1(r_1 + r_2)^*})^+$ means that all the strings represented by it must consist of one or more strings of $(\mathbf{r_1(r_1 + r_2)^*})$. However, the strings of $(\mathbf{r_1(r_1 + r_2)^*})$ start with a string of $\mathbf{r_1}$ followed by any number of strings taken arbitrarily from $\mathbf{r_1}$ and/or $\mathbf{r_2}$. Thus anything that comes after the first $\mathbf{r_1}$ in $(\mathbf{r_1(r_1 + r_2)^*})^+$ is represented by $(\mathbf{r_1 + r_2})^*$. Hence $(\mathbf{r_1(r_1 + r_2)^*})$ also represents the strings of $(\mathbf{r_1(r_1 + r_2)^*})^+$, and conversely $(\mathbf{r_1(r_1 + r_2)^*})^+$ represents the strings represented by $(\mathbf{r_1(r_1 + r_2)^*})$. Hence $(\mathbf{r_1(r_1 + r_2)^*})^+$ is reduced to $(\mathbf{r_1(r_1 + r_2)^*})$.

Ex. 4: Find a regular expression corresponding to the language *L* over the alphabet { a , b } defined recursively as follows:

1. $\varepsilon \in L$
2. If $x \in L$, then $aabx \in L$ and $xbb \in L$.
3. Nothing is in *L* unless it can be obtained from the above two clauses.

Solution: Let us see what kind of strings are in *L*. First of all $\varepsilon \in L$. Then starting with ε, strings of *L* are generated one by one by preceding aab or appending bb to any of the already generated strings. Hence a string of *L* consists of zero or more aab's in front and zero or more bb's following them. Thus $\mathbf{(aab)^*(bb)^*}$ is a regular expression for *L*.

Ex. 5: Find a regular expression corresponding to the language *L* defined recursively as follows:

1. $\varepsilon \in L$ and $a \in L$.
2. If $x \in L$, then $aabx \in L$ and $bbx \in L$.
3. Nothing is in *L* unless it can be obtained from the above.

Solution: Let us see what kind of strings are in *L*. First of all ε and a are in *L* . Then starting with ε or a, strings of *L* are generated one by one by preceding aab or bb to any of the already generated strings. Hence a string of *L* has zero or more of aab's and bb's in front possibly followed by 'a' at the end. Thus $\mathbf{(aab + bb)^*(a + \varepsilon)}$ is a regular expression for *L*.

Ex. 6: Find a regular expression corresponding to the language of all strings over the alphabet {a, b} that contain exactly two a's.

Solution: A string in this language must have at least two a's. Since any string of b's can be placed in front of the first 'a', behind the second 'a' and between the two a's, and since an arbitrary string of b's can be represented by the regular expression $\mathbf{b^*}$, $\mathbf{b^*a\,b^*a\,b^*}$ is a regular expression for this language.

Ex. 7: Find a regular expression corresponding to the language of all strings over the alphabet {a, b} that do not end with ab.

Solution: Any string in a language over { a , b } must end in a or b. Hence if a string does not end with ab then it ends with a or if it

ends with b the last b must be preceded by a symbol b. Since it can have any string in front of the last a or bb, **(a + b)*(a + bb)** is a regular expression for the language.

Ex. 8: Find a regular expression corresponding to the language of all strings over the alphabet {a, b} that contain no more than one occurrence of the string aa.

Solution: If there is one substring aa in a string of the language, then that aa can be followed by any number of b. If an a comes after that aa, then that a must be preceded by b because otherwise there are two occurences of aa. Hence any string that follows aa is represented by **(b + ba)***. On the other hand if an a precedes the aa, then it must be followed by b. Hence a string preceding the aa can be represented by **(b + ab)***. Hence if a string of the language contains aa then it corresponds to the regular expression
(b + ab)*aa(b + ba)* .
If there is no aa but at least one a exists in a string of the language, then applying the same argument as for aa to a, **(b + ab)*a(b + ba)*** is obtained as a regular expression corresponding to such strings.

If there may not be any a in a string of the language, then applying the same argument as for aa to ε, **(b + ab)*(b + ba)*** is obtained as a regular expression corresponding to such strings.

Altogether **(b + ab)*(ε + a + aa)(b + ba)*** is a regular expression for the language.

Ex. 9: Find a regular expression corresponding to the language of strings of even lengths over the alphabet of { a, b }.

Solution: Since any string of even length can be expressed as the concatenation of strings of length 2 and since the strings of length 2 are aa, ab, ba, bb, a regular expression corresponding to the language is **(aa + ab + ba + bb)***. Note that 0 is an even number. Hence the string **ε** is in this language.

Ex. 10: Describe as simply as possible in English the language corresponding to the regular expression **a*b(a*ba*b)*a*** .

Solution: A string in the language can start and end with a or b, it has at least one b, and after the first b all the b's in the string appear in pairs. Any numbe of a's can appear any place in the string. Thus simply put, it is the set of strings over the alphabet { a, b } that contain an odd number of b's.

Ex. 11: Describe as simply as possible in English the language corresponding to the regular expression **((a + b)^3)*(ε + a + b)**.

Solution: ((a + b)^3) represents the strings of length 3. Hence **((a + b)^3)*** represents the strings of length a multiple of 3. Since **((a + b)^3)*(a + b)** represents the strings of length 3n + 1, where n is

a natural number, the given regular expression represents the strings of length 3n and 3n + 1, where n is a natural number.
Ex. 12: Describe as simply as possible in English the language corresponding to the regular expression $\mathbf{(b + ab)^*(a + ab)^*}$.

Solution: $\mathbf{(b + ab)^*}$ represents strings which do not contain any substring aa and which end in b, and $\mathbf{(a + ab)^*}$ represents strings which do not contain any substring bb. Hence altogether it represents any string consisting of a substring with no aa followed by one b followed by a substring with no bb.

Theorems Related to Regular Languages

We say a set of languages is **closed** under an operation if the result of applying the operation to any arbitrary language(s) of the set is a language in the set.

For example a set of languages is closed under union if the union of any two languages of the set also belongs to the set.

The following theorem is immediate from the Inductive Clause of the definition of the set of regular languages.

Theorem 1: The set of regular languages over an alphabet Σ is closed under operations union, concatenation and Kleene star.

Proof: Let $\boldsymbol{L_r}$ and $\boldsymbol{L_s}$ be regular languages over an alphabet . Then by the definition of the set of regular languages, $\boldsymbol{L_r \cup L_s}$, $\boldsymbol{L_rL_s}$ and $\boldsymbol{L_r^*}$ are regular languages and they are obviously over the alphabet Σ. Thus the set of regular languages is closed under those operations.
Note 1: Later we shall see that the complement of a regular language and the intersection of regular laguages are also regular.
Note 2: The union of infinitely many regular languages is not necessarily regular. For example while $\{ a^kb^k \}$ is regular for any natural number k , $\mathbf{\{ a^nb^n \mid n \text{ is a natural number } \}}$ **which is the union of all the languages** $\mathbf{\{ a^kb^k \}}$ **, is not regular** as we shall see later.

The following theorem shows that any finite language is regular. We say a language is finite if it consists of a finite number of strings, that is, a **finite language** is a set of n strings for some natural number n.

Theorem 2: A finite language is regular.

Proof: Let us first assume that a language consisting of a single string is regular and prove the theorem by induction. We then prove that a language consisting of a single string is regular.

Claim 1: A language consisting of n strings is regular for any natural number n (that is, a finite language is regular) if { w } is regular for any string w.
Proof of the Claim 1: Proof by induction on the number of strings.
Basis Step: ϕ (corresponding to n = 0) is a regular language by the Basis Clause of the definition of regular language.
Inductive Step: Assume that a language *L* consisting of n strings is a regular language (induction hypothesis). Then since { ω } is a regular language as proven below, $L \cup \{ \omega \}$ is a regular language by the definition of regular language.
End of proof of Claim 1

Thus if we can show that { ω } is a regular language for any string w, then we have proven the theorem.
Claim 2: Let ω be a string over an alphabet Σ. Then { ω } is a regular language.
Proof of Claim 2: Proof by induction on strings.
Basis Step: By the Basis Clause of the definition of regular language, {ε} and { a } are regular languages for any arbitrary symbol a of Σ .
Inductive Step: Assume that { ω } is a regular language for an arbitrary string w over Σ. Then for any symbol a of Σ, { a } is a regular language from the Basis Step. Hence by the Inductive Clause of the definition of regular language { a }{ ω} is regular. Hence { aω } is regular.
End of proof for Claim 2

Note that Claim 2 can also be proven by induction on the length of string.
End of proof of Theorem 2.

2.7 REGULAR GRAMMAR

We have learned three ways of characterizing regular languages: regular expressions, finite automata and construction from simple languages using simple operations. There is yet another way of characterizing them, that is by something called grammar. A grammar is a set of rewrite rules which are used to generate strings by successively rewriting symbols. For example consider the language represented by a^+, which is {a, aa, aaa, . . . } . One can generate the strings of this language by the following procedure: Let S be a symbol to start the process with. Rewrite S using one of the following two rules: $S \rightarrow a$, and $S \rightarrow aS$. These rules mean that S is rewritten as a or as aS. To generate the string aa for example, start with S and apply the second rule to replace S with the right hand side of the rule, i.e. aS, to obtain aS. Then apply the first rule to aS to rewrite S as a. That gives us aa. We write $S \rightarrow aS$ to express that aS is obtained from S by applying a single

production. Thus the process of obtaining aa from S is written as S → aS → aa. If we are not interested in the intermediate steps, the fact that aa is obtained from S is written as S =>* aa , In general if a string β is obtained from a string α by applying productions of a grammar G, we write $\boldsymbol{\alpha \overset{*}{\rightarrow}_G \beta}$ and say that **β is derived from α** . If there is no ambiguity about the grammar G that is referred to, then we simply write $\boldsymbol{\alpha \overset{*}{\rightarrow} \beta}$

Formally a **grammar** consists of a set of non-terminals (or variables) V, a set of terminals Σ(the alphabet of the language), a start symbol S, which is a non-terminal, and a set of rewrite rules (productions) P. A production has in general the form γ → α, where γ is a string of terminals and non-terminals with at least one non-terminal in it and α is a string of terminals and non-terminals.

A **grammar is regular** if and only if γ is a single non-terminal and α is a single terminal or a single terminal followed by a single non-terminal, that is a production is of the form X → a or X → aY, where X and Y are non-terminals and 'a' is a terminal.

For example, Σ = {a, b}, V = { S } and P = { S → aS, S → bS, S → ε } is a regular grammar and it generates all the strings consisting of a's and b's including the empty string.

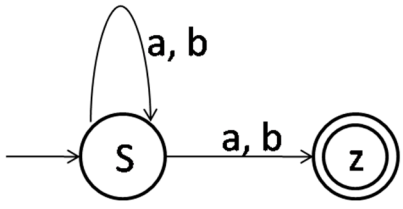
The following theorem holds for regular grammars

Theorem: A language L is accepted by an FA i.e. regular, if L - {ε} can be generated by a regular grammar.

This can be proven by constructing an FA for the given grammar as follows: For each non-terminal create a state. S corresponds to the initial state. Add another state as the accepting state Z. Then for every production X → aY, add the transition δ(X, a) = Y and for every production X → a add the transition δ (X, a) = Z.

For example Σ = {a, b}, V = {S} and P = { S → aS, S → bS, S → a, S → b } form a regular grammar which generates the language (a + b)$^+$. An NFA that recognizes this language can be obtained by creating two states S and Z, and adding transitions δ (S, a) = { S, Z } and δ (S, b) = { S, Z } , where S is the initial state and Z is the accepting state of the NFA.

The NFA thus obtained is shown below.



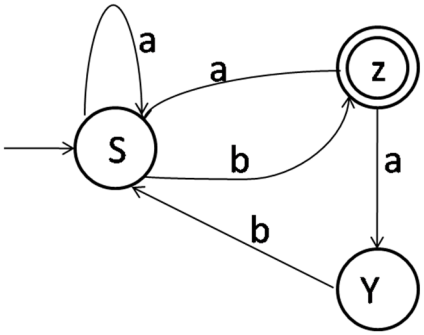
Thus L - {ε} is regular. If L contains ε as its member, then since { ε } is regular , L = (L -{ε}) ∪ {ε} is also regular.

Conversely from any NFA < Q, Σ, δ, q_0, F> a regular grammar < Q, Σ, P, q_0 > is obtained as follows:

for any 'a' in Σ, and non-terminals X and Y, X → aY is in P if and only if δ (X, a) = Y , and for any 'a' in Σ and any non-terminal X, X → a is in P if and only if δ (X, a) = Y for some accepting state Y.
Thus the following converse of Theorem 3 is obtained.

Theorem : If L is regular i.e. accepted by an NFA, then L - {ε} is generated by a regular grammar.

For example, a regular grammar corresponding to the NFA given below is < Q, { a, b }, P, S > , where Q = { S, X, Y } , P = { S → aS, S → aX, X → bS, X → aY, Y → bS, S → a }.



In addition to regular languages there are three other types of languages in Chomsky hierarchy: context-free languages, context-sensitive languages and phrase structure languages. They are characterized by context-free grammars, context-sensitive grammars and phrase structure grammars, respectively.

These grammars are distinguished by the kind of productions they have but they also form a hierarchy, that is the set of regular languages is a subset of the set of context-free languages which is in turn a subset of the set of context-sensitive languages and the set of context-sensitive languages is a subset of the set of phrase structure languages.

2.7 APPLICATION OF REGULAR EXPRESSION

Regular expressions are used in many programming languages and tools. They can be used in finding and extracting patterns in texts and programs. For example, using regular expressions, we can also specify and validate forms of data such as passwords, e-mail addresses, user IDs, etc. Here we will study the regular expression and their relationship with finite automata. In particular, we will describe methods that convert regular expressions to finite automata, and finite automata to regular expressions.

Regular expressions are useful in the production of syntax highlighting systems, data validation, and many other tasks.

While regular expressions would be useful on Internet search engines, processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regular expression. Although in many cases system administrators can run regular expression-based queries internally, most search engines do not offer regular expression support to the public.

A regular expression is a string that is used to describe or match a set of strings according to certain syntax rules. The specific syntax rules vary depending on the specific implementation, programming language, or library in use. Additionally, the functionality of regular expression implementations can vary between versions.

Lexical Analyser:

This is one of the oldest applications of regular expressions for specifying the components of a compiler called "lexical analyser". This component scans the source program and recognizes all tokens (substrings of consecutive characters that belong together logically). Keywords and identifiers are common examples of tokens but there are many others.

CHECK YOUR PROGRESS 2

State True or False

1. The empty set and ε are regular expression
2. If {a}, {b} are regular then {ab} and {a}{b} are regular
3. If L_1, L_2 are regular then union of L_1 and L_2 are regular
4. aba + ba + aaa = a(ba + b + aa)
5. $(a+b)^* = a+b)^* + a+b)^*$
6. 'b' is in the language $(a^*b)^+ a^*$
7. (ab)*a = a(ba)*
8. If L is a regular language then L* is also regular
9. aab is in the language (a+b)*(a+bb)
10. abaa is in the language (a+b)*(a+bb)
11. (a*+b)*= (a+b)*
12. (b+ab*a)* is set of strings containing even no of 'a'
13. (aa)*(ε+a)=a*
14. If L over Σ_1 is regular then L over Σ containing Σ_1 is regular
15. If L_1 over Σ_1 and L_2 over Σ_2 are regular then $L_1 \cup L_2$ is regular

2.8 LET US SUM UP

- Regular expression: this algebraic notation describes exactly the same language as finite automata. The regular expression operator are union, concatenation and closure(*).
- NFA-ε is a NFA with ε moves. In NFA-ε the automata can move to the next state without reading the next symbol.
- The set of regular languages over an alphabet Σ is closed under operations union, concatenation and Kleene star.
- Regular expressions are equal if and only if they correspond to the same language
- A **grammar is regular** if and only if γ is a single non-terminal and α is a single terminal or a single terminal followed by a single non-terminal, that is a production is of the form X → a or X → aY, where X and Y are non-terminals and 'a' is a terminal.

2.9 FURTHER READINGS

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.

2.10 ANSWERS TO CHECK YOUR PROGRESS

Check your progress 1

1. False
2. True
3. True
4. False
5. True

Check your progress 2

1. True
2. True
3. True
4. False
5. True
6. True
7. True
8. True
9. False
10. True
11. True
12. True
13. True
14. True
15. True

2.11 PROBABLE QUESTIONS

1. Explain the use of finite automata with the help of an example.
2. Explain NFA with ε transition.
3. Explain the use of regular expression.
4. Prove or disprove the following

   a) (R + S)* = R* + S*

   b) (RS + R)*R = R(SR+R)

   c) (R + S)*S = (R* S)*

*****

UNIT - 3: REGULAR LANGUAGES AND PROPERTIES OF REGULAR LANGUAGES

UNIT STRUCTURE

3.1 LEARNING OBJECTIVES

After going through this unit, you will be able to

- learn the property of regular language
- convert NFA to DFA
- minimize DFA

3.2 INTRODUCTION

In this chapter we will go through the different properties of regular language. We also going to look at the equivalence of different automata, how we can convert from one form to the other. We also going to look how we can minimize a DFA. We will look on the limitation of finite automata, language that ca not be defined by automata and prove that with a help of pumping lemma.

3.3 LIMITATIONS OF FINITE AUTOMATA AND NON REGULAR LANGUAGES

The class of languages recognized by FA s is strictly the regular set. There are certain languages which are non regular i.e. cannot be recognized by any FA

Consider the language $L = \{ a^n b^n \mid n \geq 0 \}$

In order to accept is language, we find that, an automaton seems to need to remember when passing the center point between *a*'s and *b*'s how many *a*'s it has seen so far. Because it would have to compare that with the number of *b*'s to either accept (when the two numbers are same) or reject (when they are not same) the input string.

But the number of *a*'s is not limited and may be much larger than the number of states since the string may be arbitrarily long. So, the amount of information the automaton need to remember is unbounded.

A finite automaton cannot remember this with only finite memory (i.e. finite number of states). The fact that *FA* shave finite memory imposes some limitations on the structure of the languages recognized. Inductively, we can say that a language is regular only if in processing any string in this language, the information that has to be remembered at any point is strictly limited. The argument given above to show that $a^n b^n$ is non regular is informal. We now present a formal method for showing that certain languages such as $a^n b^n$ are non regular

We can prove that a certain language is non regular by using a theorem called "Pumping Lemma". According to this theorem every regular language must have a special property. If a language does not have this property, than it is guaranteed to be not regular. The idea behind this theorem is that whenever a FA process a long string (longer than the number of states) and accepts, there must be at least one state that is repeated, and the copy of the sub string of the input string between the two occurrences of that repeated state can be repeated any number of times with the resulting string remaining in the language.

Pumping Lemma:

Let *L* be a regular language. Then the following property holds for *L*.

There exists a number $k \geq 0$ (called, the pumping length), where, if ω is any string in L of length at least k i.e. $|\omega| = k$, then ω may be divided into three sub strings $\omega = xyz$, satisfying the following conditions:

1. $y \neq \varepsilon$ i.e. $|y| > 0$
2. $|xy| \leq k$
3. For all $i \geq 0$ $xy^iz \in L$

Proof : Since L is regular, there exists a *DFA* $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes it, i.e. $L = L(M)$. Let the number of states in M is n.

Say, $Q = \{q_0, q_1, q_2, \ldots q_n\}$

Consider a string $\omega \in L$ such that $|\omega| \geq k$ (we consider the language L to be infinite and hence such a string can always be found). If no string of such length is found to be in L , then the lemma becomes vacuously true.

Since $\omega \in L$, $\delta^*(q_0, \omega) \in F$. Say $\delta^*(q_0, \omega) = q_m$ while processing the string ω, the *DFA M* goes through a sequence of states of states. Assume the sequence to be

$q_0, q_3, q_4, q_2, \ldots q_i, \ldots q_l, \ldots q_m$

start state to final state

Since $|\omega| \geq n$, the number of states in the above sequence must be greater than $n + 1$. But number of states in M is only n. hence, by pigeonhole principle at least one state must be repeated.

Let q_i and q_l be the q_l same state and is the first state to repeat in the sequence (there may be some more, that come later in the sequence). The sequence, now, looks like

$q_0, q_3, q_4, q_2, \ldots q_i, \ldots q_l, \ldots q_m$

which indicates that there must be sub strings x, y, z of w such that

$\delta^*(q_0, x) = q_i$

$\delta^*(q_i, y) = q_i$

$\delta^*(q_i, z) = q_m$

This situation is depicted in the figure

Since q_l $(=q_i)$ is the first repeated state, we have, $|xy| \leq n$ and at the same time y cannot be empty i.e $|y| > 0$. From the above, it

immediately follows that $\delta^*(q_0, xz) = q_m$. Hence $xz = xy^0z \in L$. Similarly,

$\delta^*(q_0, xy^2z) = q_m$ implying $xy^2z \in L$

$\delta^*(q_0, xy^3z) = q_m$ implying $xy^3z \in L$

and so on.

That is, starting at the loop on state can be omitted, taken once, twice, or many more times, (by the *DFA M*) eventually arriving at the final state

Thus, accepting the string *xz, xyz, xy^2z,...* i.e. xy^iz for all $i \geq 0$

Hence For all $i \geq 0$ $xy^iz \in L$.

We can use the pumping lemma to show that some languages are non regular.

3.4 PROPERTIES OF REGULAR LANGUAGE

Closure properties

Closure properties are theorems, which show that the class of regular language is closed under the operation mentioned. The theorems are of the form "if certain languages are regular, and a language L is formed from them by certain operation such as union, intersection etc. then L is also regular". In general closure properties convey the fact that when one (or several) languages are regular, then certain related languages are also regular.

The principal closure properties of regular languages are:

1. The union of two regular languages is regular.
   If L and M are regular languages, then so is $L \cup M$.
2. The intersection of two regular languages is regular.
   If L and M are regular languages, then so is $L \cap M$.
3. The compliment of two regular languages is regular.
   If L is a regular language over alphabet Σ, then Σ^*- L is also regular language.
4. The difference of two regular languages is regular.
   If L and M are regular languages, then so is L - M.
5. The reversal of a regular language is regular.
   The reversal of a string means that the string is written backward, i.e. reversal of abcde is edcba.
   The reversal of a language is the language consisting of reversal of all its strings, i.e. if L={001,110} then L^R = {100,011}.
6. The closure of a regular language is regular.
   If L is a regular language, then so is L*.

7. The concatenation of regular languages is regular.
   If L and M are regular languages, then so is L M.
8. The homomorphism of a regular language is regular.
   A homomorphism is a substitution of strings for symbol. Let the function h be defined by h(0) =a and h(1) = b then h applied to 0011 is simply aabb.
   If h is a homomorphism on alphabet S and a string of symbols w = abcd…z then
   h (w) = h (a) h (b) h(c) h (d)…h (z)
   The mathematical definition for homomorphism is
   h: $\Sigma^* \rightarrow \Gamma^*$ such that for all x, y $\in \Sigma^*$
   A homomorphism can also be applied to a language by applying it to each of strings in the language. Let L be a language over alphabet Σ, and h is a homomorphism on Σ, then
   h (L) = { h(ω) | ω is in L }
   The theorem can be stated as " If L is a regular language over alphabet Σ, and h is a homomorphism on Σ, then h(L) is also regular " .
9. The inverse homomorphism of two regular languages is regular.
   Suppose h be a homomorphism from some alphabet Σ to strings in another alphabet T and L be a language over T then h inverse of L, h'(L) is set of strings ω in Σ^* such that h(ω) is in L.
   The theorem states that "If h is a homomorphism from alphabet Σ to alphabet T, and L is a regular language on T, then h'(L) is also a regular language.

3.5 EQUIVALENCE OF AUTOMATA

ε -closures:

The concept used in the above construction can be made more formal by defining the ε-closure for a state (or 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) an input a $\in \Sigma$, we need to 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}

So, in the construction of equivalent *NFA N'* without ε-transition from any *NFA* with ε moves. the first rule can now be written as 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-ε 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:

$Q^2 = Q^1$,

$q_0^2 = q_0^1$

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

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

$= F^1$ 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 find $\delta^2(q , a)$ as follows:

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 $\delta^2(q, a)$.

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, that is 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

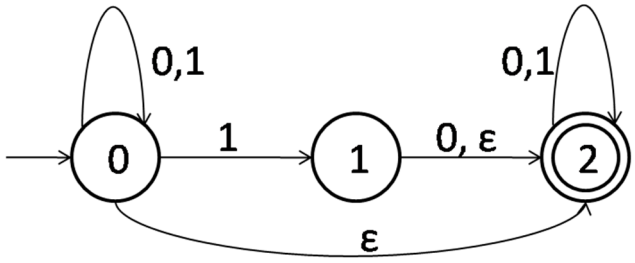
ε- transitions do not increase the power of an *NFA* . That is, any *NFA-ε* (*NFA* with ε transition), we can always construct an equivalent *NFA* without *ε*-transitions. The equivalent *NFA* must keep track where the *NFA-ε* goes at every step during computation. This can be done by adding extra transitions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

If we removed the ε - transition δ(p, ε) = q from the *NFA-ε*, then we need to moves from state *p* to all the state γ on input symbol $q \in \Sigma$ which are reachable from state q (in the *NFA-ε*) on same input symbol *q*. This will allow the modified *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 in the following theories.

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*

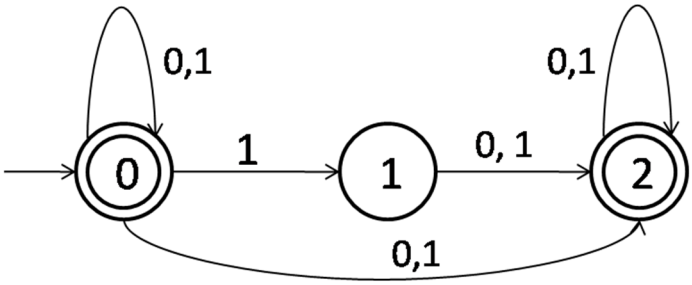
Ex: Consider the following *NFA* with ε transition.

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



The equivalent NFA is-

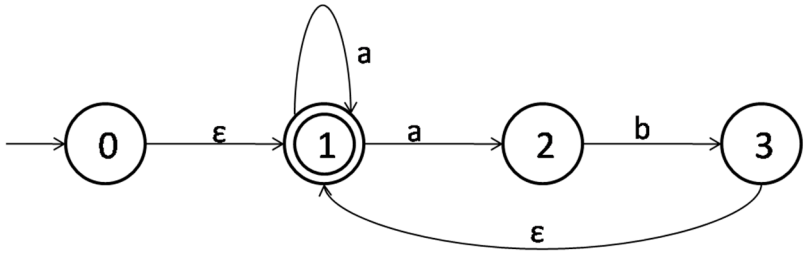
	0	1
→q_0 F	{q_0, q_2}	{q_0, q_1, q_2}
q_1	{q_2}	{q_2}
F q_2	{q_2}	{q_2}



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

Since δ(q_0, ε) = q_2 and δ(q_2, 0) = q_2 and δ(q_2, 1) = q_2 we add moves δ(q_0, 0) = q_2 and δ(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:

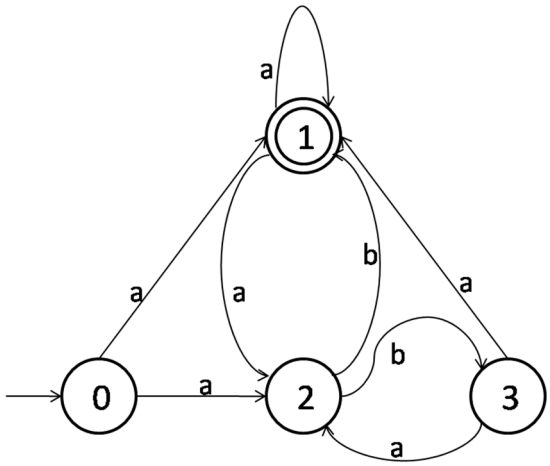
δ_2(0, a): First ε({0}) = {0, 1}. Then from the transition function of the NFA-ε

δ_1(0 , a) = Φ , and δ_1(1, a) = {1, 2}.

Hence δ_2(0, a) = ε({1, 2}) = {1, 2}.
For δ_2(0, b), since ε({0}) = {0, 1} and δ_1(0, b) = δ_1(1, b) = Φ, δ_2(0, b) = Φ.
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})	$\bigcup_{p\in\varepsilon(q)} \delta_1(p,$.	δ_2(q,a) (= ε($\bigcup_{p\in\varepsilon(q)} \delta_1(p,a)$))
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*

It is worth noting that a *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 generality enjoyed in terms 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 functions is defined from QxΣ to Q whereas 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 $q_i \in Q$
$\delta'(\{p\}, a) = \{(\delta(p,a)\}$ i.e.

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

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

If $\omega = a_1, a_2, \ldots, a_n \in L(D)$ 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 $q_n \in F$

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^n possible 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-transition and then use the above construction process to 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-transition by integrating the concept of ε-closure in the above construction.

Recall that, for any $R \subseteq Q$

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

In the equivalent *DFA* , at every step, we need to modify the transition functions 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 -transitions. This can be done by changing the initial state q_0^D to ε-closure (q_0^D).

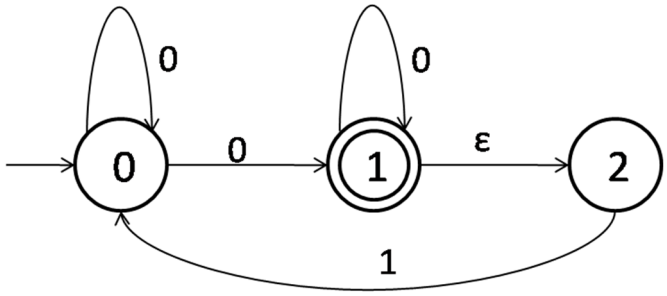
It is clear that, at every step in the processing of an input string by the *DFA D* , it enters a state that corresponds to the subset of states that the *NFA N* could be in at that particular point. This has been proved in the constructions of an equivalent NFA for any ε-*NFA*

If the number of states in the *NFA* is *n* , then there are 2^n states 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

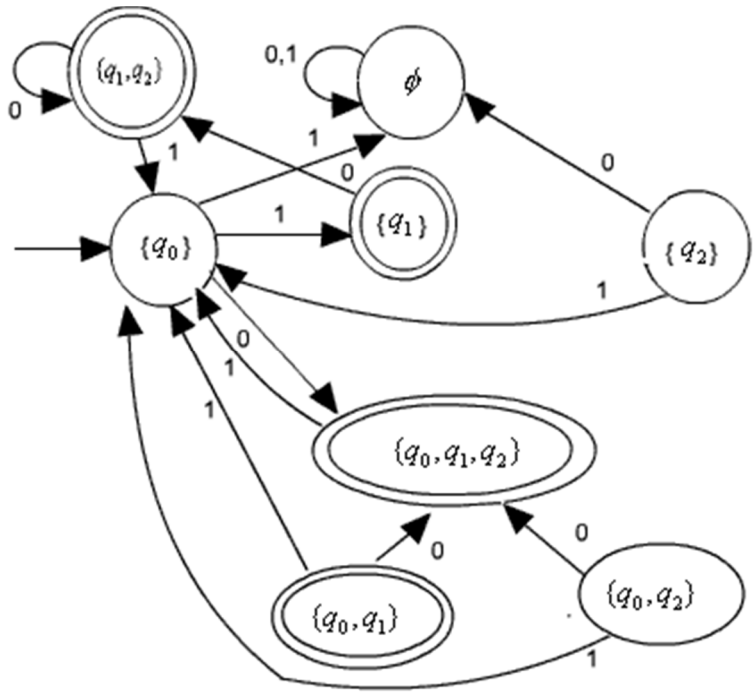
$$\delta^D(q_o, 0) = \varepsilon\text{-closure}(\delta(q_o, 0))$$

$$= \varepsilon\text{-closure}(\delta(q_o, q_1))$$

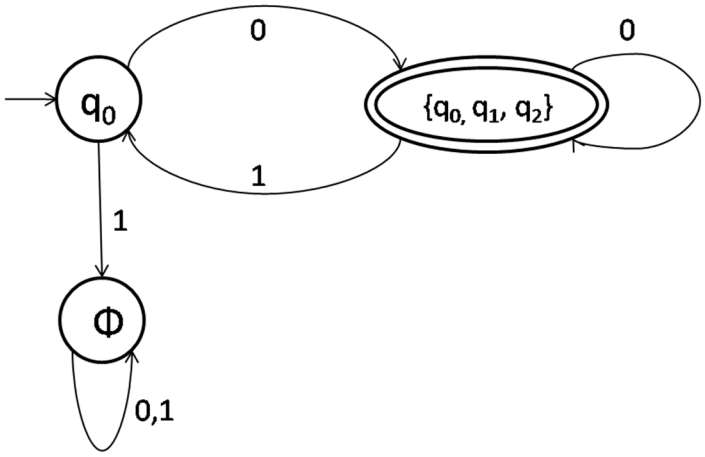
$$= \{q_0, q_1, q_2\}$$

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 δ of states which are accessible. Then for all a ∈ Σ. compute $\delta^D(S, a)$ because these set of states will also be accessible.

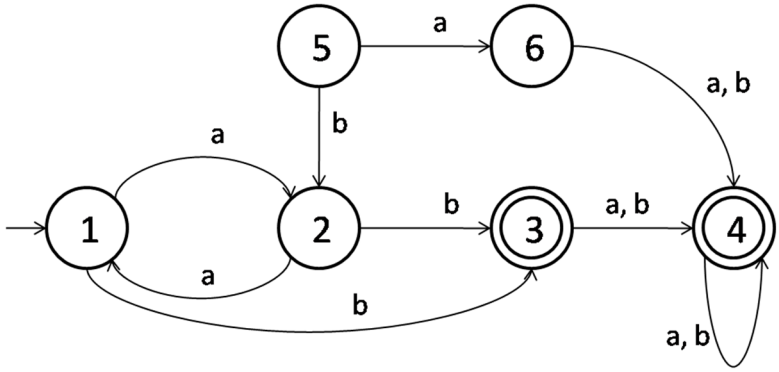
Following these steps in the above example, we get the transition table given below

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

3.6 Minimization of DFA

For any regular language *L* it may be possible to design different DFAs to accept *L*. Given two DFAs accepting the same language *L*, it is now natural to ask, which one is more simple? In this case, obviously, the one with less number of states would be simpler than the other. So, given a DFA accepting a language, we might wonder whether the DFA could further be simplified i.e. can we reduce the number of states accepting the same language.

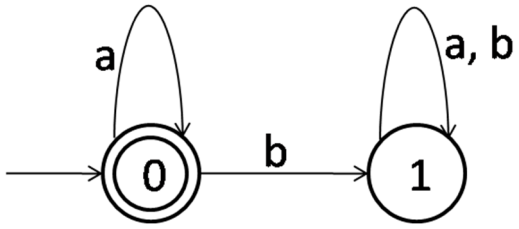
Consider the following DFA M_1,



A minute observation will reveal that it accepts the language of the regular expression

$a^*b(a+b)^*$

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



It is a fact that, for any regular language *L* there is a unique minimal state DFA, the uniqueness is up to isomorphism to be defined next.
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. sates *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 of it.
- Finally, merge all states which are "equivalent" or "indistinguishable". We need to define formally what is meant by equivalent or indistinguishable states; but at this point we assume that merging these states would not change the accepted language.

Inaccessible states can easily be found out by using a simple research e.g. depth first search. Removing trap states are also simple. In the 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.

we start with a definition and then proceed to find method to construct minimal state DFAs.

DFA Isomorphism:

Two DFAs are said to be isomorphism if they are identical up to renaming of the states. 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 $q \in Q_1$, $q \in F_1$, 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 as states 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 DFA 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 $\equiv_L$.
- 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 $x_i \in \Sigma^*$, $\delta(q_0, x_i) \in Q$. 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 $\equiv_L$, this proves that *f* is one-to-one).

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

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 $q_{0L} \equiv_L [\varepsilon]$. 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 stat 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 $\equiv_M$)

But, how do we decide in general when two states can be merged without changing the language accepted?
we now going to devise an algorithm for doing this until no more merging is possible. we start with the following observations.

- 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 $\delta(p, a)$ and $\delta(q, a)$, 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 $\delta^*(q, x) \notin F$ for some $x \in \Sigma^*$. Using the concept in the previous page, 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 indistingushability is an equivalence relation.

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

we say that, states *p* and *q* of a DFA *M* accepting a language *L* can be merged safely (i.e. 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 $\forall\ x,y \in \Sigma^*$, $\delta^*(q_0, x) = p$ and $\delta^*(q_0, x) = q \Rightarrow x \equiv_L y$.

A Minimization Algorithm :

We now produce an algorithm to construct the minimal state DFA from any given DFA accepting *L* by merging states inductively.

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. $p \not\equiv q$. 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.

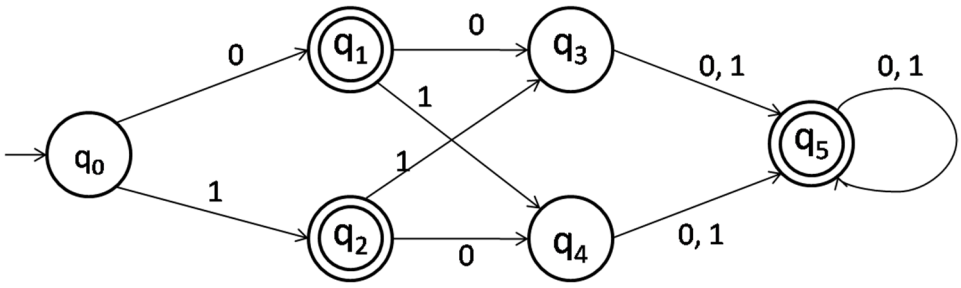
1. For every $p, q \in Q$, initially unmark all pairs (*p, q*).
2. If $p \in F$ and $q \notin F$ (or vice versa) then mark (*p, q*).

3. Repeat the following step until no more changes occur**:** If there exists an unmarked pair (*p, q*) such that (δ(p,a), δ(q,a)) is marked for some $a \in \Sigma$, then mark (*p, q*).
4. p ≡ q iff (*p, q*) is unmarked.

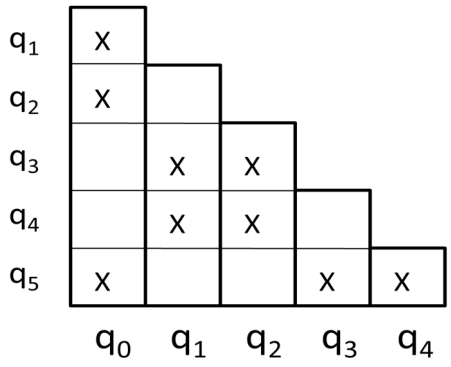
The algorithm correctly computes all pairs of states that can be distingusihed i.e. unmarked.

It is easy to show (by induction) that the pair (*p, q*) is mraked by the above algorithm iff ∃ x ∈ Σ^* such that δ^*(p, x) ∈F and δ^*(q, x) ∉ F (or vice versa) i.e. if p ≢ 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.)



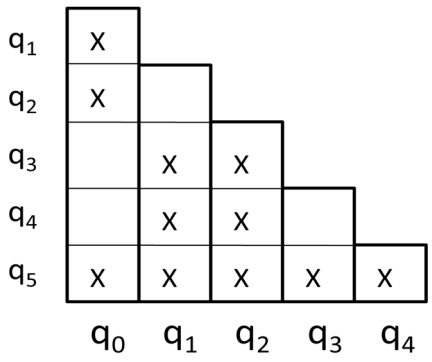
Initially, all cells are unmarked. (i.e. at step 1 of the algorithm) . After step 2, all cells representing pairs of states of which one is accepting and the other is non-accepting are marked by putting an X. The table above shows the status after this step.

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. we use the notation $(q_0,q_3) \xrightarrow{0} (q_1,q_5)$ 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.

- $(q_0,q_4) \xrightarrow{0} (q_1,q_5)$
- $(q_0,q_4) \xrightarrow{1} (q_2,q_5)$ cannot mark (q_0,q_4) since (q_1,q_5) & (q_2,q_5) are unmarked.
- $(q_1,q_2) \xrightarrow{0} (q_3,q_4)$
- $(q_1,q_2) \xrightarrow{1} (q_3,q_4)$ cannot mark (q_1,q_2) since (q_3,q_4) is unmarked.
- $(q_1,q_5) \xrightarrow{0} (q_3,q_5)$, (q_1,q_5) is marked since (q_3,q_5) is already marked.
- $(q_2,q_5) \xrightarrow{0} (q_4,q_5)$, (q_2,q_5) is marked since (q_3,q_5) is already marked.
- $(q_3,q_4) \xrightarrow{0} (q_5,q_5)$, (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.



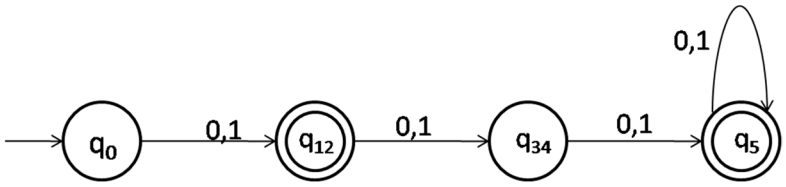
In the next pass we find that $(q_0,q_3) \xrightarrow{0} (q_1,q_5)$ and (q_1,q_5) is marked in the previous pass. Hence, (q_0,q_3) can be marked now.

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 $q_3 \equiv q_4$.

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.

CHECK YOUR PROGRESS 1

State True or False

1. δ(q, ε) always contains q
2. δ*(q, ε) does not always contain q
3. In the recursive definition of ε(S) the basis is a empty set
4. In the minimization of algorithm initially the states are grouped into two accepting and non-accepting states
5. For each regular language there is a unique DFA with smallest number of states

3.9 LET US SUM UP

- ε-closure of a state is the set of states which are reachable from the states by ε-moves without reading the input.
- The union of two regular languages is regular. The intersection of two regular languages is regular. The compliment of two regular languages is regular. The difference of two regular languages is regular. The reversal of a regular language is regular. The closure of a regular

language is regular. The concatenation of regular languages is regular. The homomorphism of a regular language is regular. The inverse homomorphism of two regular languages is regular.

- 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.

3.10 FURTHER READINGS

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.

3.11 ANSWERS TO CHECK YOUR PROGRESS

Check your progress 1

Check your Progress 2

1. False
2. False
3. False
4. True
5. True

3.12 PROBABLE QUESTIONS

1. Explain how we can convert a NFA to DFA.
2. Explain how we can convert a NFA-ε to DFA
3. 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 - 4: CONTEXT FREE GRAMMAR AND LANGUAGE

UNIT STRUCTURE

4.1 LEARNING OBJECTIVES

After going through this unit, you will be able to

- define context free grammar
- work on push down automata
- design parse tree
- know the problems related to context free grammar

4.2 INTRODUCTION

In addition to regular languages there are three other types of languages in Chomsky hierarchy: context-free languages, context-sensitive languages and phrase structure languages. They are characterized by context-free grammars, context-sensitive grammars and phrase structure grammars, respectively.

These grammars are distinguished by the kind of productions they have but they also form a hierarchy, that is the set of regular languages is a subset of the set of context-free languages which is in turn a subset of the set of context-sensitive languages and the set of context-sensitive languages is a subset of the set of phrase structure languages.

A grammar is a **context-free grammar** if and only if its production is of the form $X \rightarrow \alpha$, where α is a string of terminals and non-terminals, possibly the empty string. For example $P = \{ S \rightarrow aSb, S \rightarrow ab \}$ with $\Sigma = \{ a, b \}$ and $V = \{ S \}$ is a context-free grammar and it generates the language $\{ a^n b^n \mid n \text{ is a positive integer} \}$. As we shall see later this is an example of context-free language which is not regular.

A grammar is a **context-sensitive grammar** if and only if its production is of the form $\alpha_1 x \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, where X is a non-terminal and α_1, α_2 and β are strings of terminals and non-terminals, possibly empty except β .Thus the non-terminal X can be rewritten as β only in the context of $\alpha_1 x \alpha_2$. For example $P = \{ S \rightarrow XYZS_1, S \rightarrow XYZ, S_1 \rightarrow XYZS_1, S_1 \rightarrow XYZ, YX \rightarrow XY, ZX \rightarrow XZ, ZY \rightarrow YZ, X \rightarrow a, aX \rightarrow aa, aY \rightarrow ab, BY \rightarrow bb, bZ \rightarrow bc, cZ \rightarrow cc \}$ with $\Sigma = \{ a, b, c \}$ and $V = \{ X, Y, Z, S, S_1 \}$ is a context-sensitive grammar and it generates the language $\{ a^n b^n c^n \mid n \text{ is a positive integer} \}$. It is an example of context-sensitive language which is not context-free. Context-sensitive grammars are also characterized by productions whose left hand side is not longer than the right hand side, that is, for every production $\alpha \rightarrow \beta$, $|\alpha| \leq |\beta|$.

For a phrase structure grammar, there is no restriction on the form of production, that is a production of a phrase structure grammar can take the form α-> β, where α and β can be any string, but α must contain at least one non-terminal.

Here we are going to discuss about context-free grammars. Context free grammars are those whose productions have the form $X \rightarrow \alpha$, where X is a nonterminal and α is a nonempty string of terminals and nonterminals. The set of strings generated by a context-free grammar is called a context-free language and context-free languages can describe many practically important systems. Most programming languages can be approximated by context-free grammar and compilers for them have been developed based on properties of context-free languages. Let us define context-free grammars and context-free languages here.

4.3 CONTEXT-FREE GRAMMAR

Definition (Context-Free Grammar) :
A context-free grammar G is a 4-tuple $G = < V , \Sigma , S , P >$ is a context-free grammar (CFG) if V and Σ are finite sets sharing no elements between them, $S \in V$ is the start symbol, and P is a finite set of productions of the form $X \rightarrow \alpha$, where $X \in V$, and $\alpha \in (V \cup \Sigma)^*$.

A language is a context-free language (CFL) if all of its strings are generated by a context-free grammar.

Example 1: $L_1 = \{ a^n b^n \mid n \text{ is a positive integer} \}$ is a context-free language. For the following context-free grammar $G_1 = < V_1 , \Sigma, S, P_1 >$ generates L_1 :
$V_1 = \{ S \} , \Sigma = \{ a , b \}$ and $P_1 = \{ S \rightarrow aSb , S \rightarrow ab \}$.

Example 2: $L_2 = \{ ww^r \mid w \in \{a, b\}^+ \}$ is a context-free language , where w is a non-empty string and w^r denotes the reversal of string w, that is, w is spelled backward to obtain w^r . For the following context-free grammar $G_2 = < V_2 , \Sigma, S, P_2 >$ generates L_2 :
$V_2 = \{ S \} , \Sigma = \{ a, b \}$ and
$P_2 = \{ S \rightarrow aSa , S \rightarrow bSb , S \rightarrow aa , S \rightarrow bb \}$.

Example 3: Let L_3 be the set of algebraic expressions involving identifiers x and y, operations + and * and left and right parentheses. Then L_3 is a context-free language. For the following context-free grammar $G_3 = < V_3 , \Sigma_3, S , P_3 >$ generates L_3 :
$V_3 = \{ S \} , \Sigma_3 = \{ x, y, (,), +, * \}$ and
$P_3 = \{ S \rightarrow (S + S) , S \rightarrow S*S , S \rightarrow x , S \rightarrow y \}$.

Example 4: Portions of the syntaxes of programming languages can be described by context-free grammars. For example
{ < statement > → < if-statement > ,
< statement > → < for-statement > ,
< statement > → < assignment > , . . . ,
< if-statement >→ if (< expression >) < statement > ,
< for-statement > → for (< expression > ; < expression > ;
< expression >) < statement > , . . . ,
< expression > → < algebraic-expression > ,
< expression > → < logical-expression > , . . . }.

Properties of Context-Free Language

Theorem 1: Let L_1 and L_2 be context-free languages. Then $L_1 \cup L_2$, L_1L_2 , and L_1^* are context-free languages.

Proof
This theorem can be verified by constructing context-free grammars for union, concatenation and Kleene star of context-free grammars as follows:

Let $G_1 = < V_1 , \Sigma , S_1 , P_1 >$ and $G_2 = < V_2 , \Sigma , S_2 , P_2 >$ be context-free grammars generating L_1 and L_2 , respectively.

Then for $L_1 \cup L_2$, first relabeled symbols of V_2, if necessary, so that V_1 and V_2 don't share any symbols. Then let S_u be a symbol which is not in $V_1 \cup V_2$. Next define $V_u = V_1 \cup V_2 \cup \{ S_u \}$ and $P_u = P_1 \cup P_2 \cup \{S_u \rightarrow S_1 , S_u \rightarrow S_2 \}$.

Then it can be easily seen that $G_u = < V_u , \Sigma , S_u , P_u >$ is a context-free grammar that generates the language $L_1 \cup L_2$.

Similarly for L_1L_2, first relabeled symbols of V_2 , if necessary, so that V_1 and V_2 don't share any symbols. Then let S_c be a symbol which is not in $V_1 \cup V_2$. Next define $V_c = V_1 \cup V_2 \cup \{ S_c \}$ and $P_c = P_1 \cup P_2 \cup \{S_c \rightarrow S_1S_2 \}$.

Then it can be easily seen that $G_c = < V_c , \Sigma , S_c , P_c >$ is a context-free grammar that generates the language L_1L_2 .

For L_1^*, let S_s be a symbol which is not in V_1. Then let $P_s = P_1 \cup \{ S_s \rightarrow S_sS_1 , S_s \rightarrow \varepsilon \}$. It can be seen that the grammar $G_s = < V_s , \Sigma , S_s , P_s >$ is a context-free grammar that generates the language L_1^* .

4.4 PUSHDOWN AUTOMATA

Like regular languages which are accepted by finite automata, context-free languages are also accepted by automata but not finite automata. They need a little more complex automata called pushdown automata.

Let us consider a context-free language a^nb^n . Any string of this language can be tested for the membership for the language by a finite automaton if there is a memory such as a pushdown stack that can store a's of a given input string. For example, as a's are read by the finite automaton, push them into the stack. As soon as the symbol b appears stop storing a's and start popping a's one by one every time a b is read. If another a (or anything other than b) is read after the first b, reject the string. When all the symbols of the input string are read, check the stack. If it is empty, accept the string. Otherwise reject it.

This automaton behaves like a finite automaton except the following two points: First, its next state is determined not only by the input symbol being read, but also by the symbol at the top of the stack. Second, the contents of the stack can also be changed every time an input symbol is read. Thus its transition function specifies the new top of the stack contents as well as the next state.

Let us define this new type of automaton formally.

A **pushdown automaton** (or **PDA** for short) is a 7-tuple $M = < Q, \Sigma, \Gamma, q_0, Z , F, \delta >$, where

Q is a finite set of states,
Σ and Γ are finite sets (the input and stack alphabet, respectively).
q_0 is the initial state,
Z_0 is the initial stack symbol and it is a member of Γ,
F is the set of accepting states
δ is the transition function and
$\delta : Q \times (\Sigma \cup \varepsilon) \times \Gamma \rightarrow 2^Q \times \Gamma^*$.

Thus $\delta(p , a , X) = (q , \alpha)$ means the following:

The automaton moves from the current state of p to the next state q when it sees an input symbol 'a' at the input and X at the top of the stack, and it replaces X with the string α at the top of the stack.

Example 1 :

Let us consider the pushdown automaton $< Q, \Sigma, \Gamma, q_0 , Z_0, F, \delta >$, where $Q = \{ q_0 , q_1 , q_2 \}$, $\Sigma = \{ a, b \}$, $\Gamma = \{ A, Z_0 \}$, $F = \{ q_2 \}$ and let δ be as given in the following table:

State	**Input**	**Top of Stack**	**Move**
q_0	A	Z_0	(q_0 , AZ_0)
q_0	A	A	(q_0 , AA)
q_0	B	A	(q_1 , ε)
q_1	B	A	(q_1 , ε)
q_1	E	Z_0	(q_2 , Z_0)

This pushdown automaton accepts the language $a^n b^n$. To describe the operation of a PDA we are going to use a configuration of PDA. A **configuration** of a PDA $M = < Q, \Sigma, \Gamma, q_0, Z_0, A, \delta >$ is a triple (q , x , α) , where q is the state the PDA is currently in, x is the unread portion of the input string and α is the current stack contents, where the input is read from left to right and the top of the stack corresponds to the leftmost symbol of α. To express that the PDA moves from configuration (p , x , α) to configuration (q , y , β) in a single move (a single application of the transition function) we write

$$(p, x, \alpha) \vdash (q, y, \beta).$$

If (q , y , β) is reached from (p , x , α) by a sequence of zero or more moves, we write

$$(p, x, \alpha) \vdash^* (q, y, \beta).$$

Let us now see how the PDA of Example 1 operates when it is given the string aabb, for example.

Initially its configuration is (q_0 , aabb , Z_0). After reading the first 'a', its configuration is (q_0 , abb , AZ_0). After reading the second 'a', it is (q_0 , bb , AAZ_0). Then when the first 'b' is read, it moves to state q_1 and pops 'A' from the top of the stack. Thus the configuration is (q_1 , b , AZ_0). When the second 'b' is read, another 'A' is popped from the top of the stack and the PDA stays in state q_1. Thus the configuration is (q_1, ε, Z_0). Next it moves to the state q_2 which is the accepting state. Thus aabb is accepted by this PDA. This entire process can be expressed using the configurations as

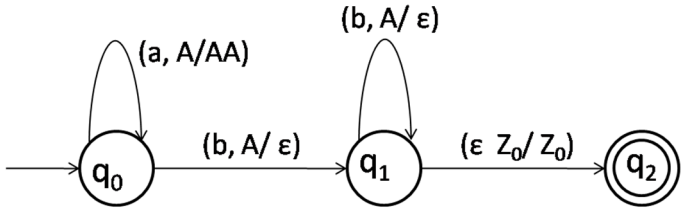
$$(q_0, aabb, Z_0) \vdash (q_0, abb, AZ_0) \vdash (q_0, bb, AAZ_0) \vdash (q_1, b, AZ_0) \vdash (q_1, \varepsilon , Z_0) \vdash (q_2, \varepsilon, Z_0).$$

If we are not interested in the intermediate steps, we can also write

$$(q_0 , aabb, Z_0) \vdash^* (q_2 , \varepsilon , Z_0)$$

A string x is accepted by a PDA if $(q_0, x, Z_0) \vdash^* (q, \varepsilon, \alpha)$, for some α in Γ^*, and an accepting state q.

Like FAs, PDAs can also be represented by transition diagrams. For PDAs, however, arcs are labeled differently than FAs. If $\delta(q, a, X) = (p, \alpha)$, then an arc from state p to state q is added to the diagram and it is labeled with (a , X / α) indicating that X at the top of the stack is replaced by α upon reading 'a' from the input. For example the transition diagram of the PDA of Example 1 is as shown below.



Example 2 :

Let us consider the pushdown automaton < Q, Σ, Γ, q_0 , Z_0, F, δ > , where Q = { q_0, q_1, q_2 }, Σ = { a, b, c }, Γ = { A, B, Z_0 }, F = { q_2 } and let δ be as given in the following table:

State	Input	Top of Stack	Move
q_0	A	Z_0	(q_0 , AZ_0)
q_0	B	Z_0	(q_0 , BZ_0)
q_0	A	σ	(q_0 , A σ)
q_0	B	σ	(q_0 , B σ)
q_0	C	σ	(q_1 , σ)
q_1	A	A	(q_1 , ε)
q_1	B	B	(q_1 , ε)
q_1	E	Z_0	(q_2 , Z_0)

In this table σ represents either a or b.

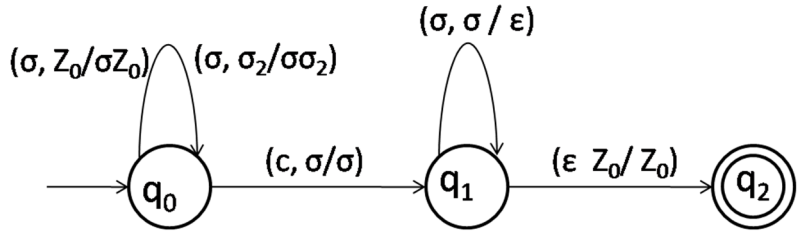
This pushdown automaton accepts the language $\{ wcw^r \mid w \in \{ a, b\}^* \}$, which is the set of palindromes with c in the middle.

For example for the input abbcbba, it goes through the following configurations and accepts it.

$(q_0 , abbcbba , Z_0) \vdash (q_0 , bbcbba , AZ_0) \vdash$
$(q_0 , bcbba , BAZ_0) \vdash (q_0 , cbba , BBAZ_0) \vdash$
$(q_1 , bba , BBAZ_0) \vdash (q_1 , ba , BAZ_0) \vdash$
$(q_1 , a , AZ_0) \vdash (q_1 , \varepsilon , Z_0) \vdash (q_2 , \varepsilon , Z_0)$.

This PDA pushes all the a's and b's in the input into stack until c is encountered. When c is detected, it ignores c and from that point on if the top of the stack matches the input symbol, it pops the stack. When there are no more unread input symbols and Z_0 is at the top of the stack, it accepts the input string. Otherwise it rejects the input string.

The transition diagram of the PDA of Example 2 is as shown below. In the figure σ, σ_1 and σ_2 represent a or b.



4.5 PARSING AND PARSE TREE

Consider the algebraic expression x + yz. Though we are accustomed to interpreting this as x + (yz) i.e. compute yz first, then add the result to x, it could also be interpreted as (x + y) z meaning that first compute x + y, then multiply the result by z. Thus if a computer is given the string x + yz, it does not know which interpretation to use unless it is explicitly instructed to follow one or the other. Similar things happen when English sentences are processed by computers. For example in the sentence "A man bites a dog", native English speakers know that it is the dog that bites and not the other way round. "A dog" is the subject, "bites" is the verb and "a man" is the object of the verb. However, a computer like non-English speaking people must be told how to interpret sentences such as the first noun phrase ("A dog") is usually the subject of a sentence, a verb phrase usually follow the noun phrase and the first word in the verb phrase is the verb and it is followed by noun phrases representing object(s) of the verb.

Parsing is the process of interpreting given input strings according to predetermined rules i.e. productions of grammars. By parsing sentences we identify the parts of the sentences and determine the structures of the sentences so that their meanings can be understood correctly.

Context-free grammars are powerful grammars. They can describe much of programming languages and basic structures of natural languages. Thus they are widely used for compilers for high level programming languages and natural language processing systems. The parsing for context-free languages and regular languages have been extensively studied.

Parsing is a process to determine how a string might be derived using productions of a given grammar. It can be used to check whether or not a string belongs to a given language. When a statement written in a programming language is input, it is parsed by a compiler to check whether or not it is syntactically correct and to extract components if it is correct. Finding an efficient parser is a nontrivial problem and a great deal of research has been conducted on parser design.

Here basic parsing techniques are introduced with examples and some of the problems involved in parsing are discussed together with brief explanations of some of the solutions to those problems
Two basic approaches to parsing are top-down parsing and bottom-up parsing. In the **top-down** approach, a parser tries to derive the given string from the start symbol by rewriting nonterminals one by one using productions. The nonterminal on the left hand side of a

production is replaced by it right hand side in the string being parsed. In the **bottom-up** approach, a parser tries to reduce the given string to the start symbol step by step using productions. The right hand side of a production found in the string being parsed is replaced by its left hand side.

Let us see how string *aababaa* might be parsed by these two approaches for the following grammar as an example:

$S \rightarrow aSa \mid bSb \mid a \mid b$

This grammar generates the palindromes of odd lengths.

Top-down approach proceeds as follows:

- The start symbol **S** is pushed into the stack without reading any input symbol.
- **S** is popped and **aSa** is pushed without reading any input symbol.
- As the first ***a*** in the input is read, ***a*** at the top of the stack is popped.
- **S** is popped and **aSa** is pushed without reading any input symbol.
- As the second ***a*** in the input is read, ***a*** at the top of the stack is popped.
- **S** is popped and **bSb** is pushed without reading any input symbol.
- As the first ***b*** in the input is read, ***b*** at the top of the stack is popped.
- **S** is popped and **a** is pushed without reading any input symbol.
- As the unread input symbols are read, ***abaa*** in the stack is popped one by one.

Since the stack is empty when the entire input string is read, the string is found to be in the language.

If we use configuration without state, that is, (unread portion of input, stack contents), this top-down parsing can be expressed as follows:

$(aababaa, Z_0) \vdash (aababaa, SZ_0) \vdash (aababaa, aSaZ_0) \vdash$
$(ababaa, SaZ_0) \vdash (ababaa, aSaaZ_0) \vdash (babaa, SaaZ_0) \vdash$
$(babaa, bSbaaZ_0) \vdash (abaa, SbaaZ_0) \vdash (abaa, abaaZ_0) \vdash$
$(baa, baaZ_0) \vdash (aa, aaZ_0) \vdash (a, aZ_0) \vdash (\varepsilon, Z_0)$

In general a PDA for top-down parsing has the following four types of transitions:

- For each production, pop the nonterminal on the left hand side of the production at the top of the stack and push its right hand side string;
- Pop the stack if the top of the stack matches the input symbol being read;
- Initially push the start symbol into the stack;
- Go to the final state if the entire input has been read and the stack is empty.

Bottom-up approach proceeds as follows:

- The string ***aababaa*** is read into the stack one by one from left until the middle ***a*** is reached.
- The middle ***a*** is replaced by **S** at the top of the stack without reading any input symbol.
- The second ***b*** is read and pushed into the stack.
- **bSb** at the top of the stack is replaced by **S**.
- The fourth ***a*** is read and pushed into the stack.
- **aSa** at the top of the stack is replaced by **S**.
- The last ***a*** is read and pushed into the stack.
- **aSa** at the top of the stack is replaced by **S**.

Since the stack has **S** when the entire input string is read, the string is found to be in the language.

If we use configuration without state, this bottom-up parsing can be expressed as follows:

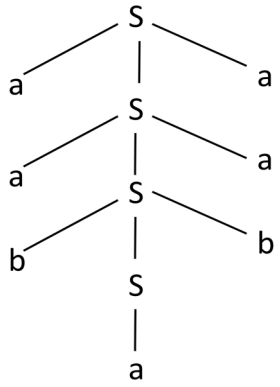
$$(aababaa, Z_0) \vdash (ababaa, aZ_0) \vdash (babaa, aaZ_0) \vdash$$
$$(abaa, baaZ_0) \vdash (baa, abaaZ_0) \vdash (baa, SbaaZ_0) \vdash$$
$$(aa, bSbaaZ_0) \vdash (aa, SaaZ_0) \vdash (a, aSaaZ_0) \vdash$$
$$(a, SaZ_0) \vdash (\varepsilon, aSaZ_0) \vdash (\varepsilon, SZ_0)$$

Note that the rightmost symbol on the right hand side of a production appears at the top of the stack.

In general a PDA for bottom-up parsing has the following four types of transitions:

- Push input symbol being read into the stack -- this is called *shift*;
- Replace the right hand side of a production at the top of the stack with its left hand side -- this is called *reduce*;
- Pop the stack if the top of the stack matches the input symbol being read;
- If the entire input has been read and only the start symbols is in the stack, then pop the stack and go to the final state.

The structure of a derivation of a string can be represented by a tree called **parse tree** or a **derivation tree**. A parse tree has the start symbol at its root. Its internal nodes correspond to the nonterminals that appear in the derivation. The children of a node are the symbols appearing on the right hand side of the production used to rewrite the nonterminal corresponding to the node in the derivation. For example the following figure shows the parse tree of the string *aababaa* of the above example.



The top-down parsing traverses this tree from the root down to the leaves, while the bottom-up parsing goes from the leaves up to the root

Example 2 for top-down and bottom-up parsing:

Given the grammar

$S \rightarrow S + X \mid X;$
$X \rightarrow X * Y \mid Y;$
$Y \rightarrow (S) \mid id$

let us parse the expression *a + b*c*.

Top-down parsing:

$(a + b * c, Z_0) \vdash (a + b * c, S\ Z_0) \vdash (a + b * c, S+X\ Z_0) \vdash$
$(a + b * c, X + X\ Z_0) \vdash (a + b * c, Y + X\ Z_0) \vdash$
$(a + b * c, a + X\ Z_0) \vdash (+ b * c, + X\ Z_0) \vdash (b * c,\ X\ Z_0) \vdash$
$(b * c,\ X * Y\ Z_0) \vdash (b * c,\ b * Y\ Z_0) \vdash (* c,\ * Y\ Z_0) \vdash$
$(c,\ Y\ Z_0) \vdash (c,\ c\ Z_0) \vdash (\varepsilon,\ Z_0)$

Bottom-up parsing:

$(a + b * c, Z_0) \vdash (+ b * c, a\ Z_0) \vdash (+ b * c, Y\ Z_0) \vdash$
$(+ b * c,\ X\ Z_0) \vdash (+ b * c, S\ Z_0) \vdash (b * c, + S\ Z_0) \vdash$
$(* c, b + X\ Z_0) \vdash (* c,\ Y + S\ Z_0) \vdash (* c,\ X + S\ Z_0) \vdash$
$(c,\ * X + S\ Z_0) \vdash (\varepsilon,\ c * X + S\ Z_0) \vdash (\varepsilon,\ Y * X + S\ Z_0) \vdash$
$(\varepsilon, X + S\ Z_0) \vdash (\varepsilon, S\ Z_0)$

Note again that the rightmost symbol on the right hand side of a production appears at the top of the stack.

Difficulties in parsing

The main difficulty in parsing is nondeterminism. That is, at some point in the derivation of a string more than one productions are applicable, though not all of them lead to the desired string, and one cannot tell which one to use until after the entire string is generated. For example in the parsing of *aababaa* discussed above, when **S** is at the top of the stack and ***a*** is read in the top-down parsing, there are two applicable productions, namely **S → aSa** and **S → a**. However, it is not possible to decide which one to choose with the information of the input symbol being read and the top of the stack. Similarly for the bottom-up parsing, it is impossible to tell when to apply the production **S → a** with the same information as for the top-down parsing. Some of these nondeterminisms are due to the particular grammar being used and they can be removed by transforming grammars to other equivalent grammars while others are the nature of the language the string belongs to. Below several of the difficulties are briefly discussed.

Factoring:

Consider the following grammar:

$$S \rightarrow T;\ T \rightarrow aTb \mid abT \mid ab.$$

With this grammar when string *aababaa* is parsed top-down, after **S** is replaced by **T** in the first step, there is no easy way of telling which production to use to rewrite **T** next. However, if we change this to the following grammar which is equivalent to this grammar, this nondeterminism disappears:

$$S \rightarrow aU;\ U \rightarrow Sb \mid bT;\ T \rightarrow S \mid \varepsilon.$$

This transformation operation is called **factoring** as *a* on the right hand side of productions for **T** in the original grammar are factored out as see n in the new grammar.

Left-recursion:

Consider the following grammar:

$$S \rightarrow Sa \mid Sb \mid a$$

When a string, say *aaba*, is parsed top-down for this grammar, after **S** is pushed into the stack, it needs to be replaced by the right hand side of some production. However, there is no simple way of telling which production to use and a parser may go into infinite loop especially if it is given an illegal string (a string which is not in the language). This kind of grammar is called **left-recursive**.

Left-recursions can be removed by replacing left-recursive pairs of productions with new pairs of productions as follows:
If $X \rightarrow X\alpha_1 \mid X\alpha_2 \mid \beta_1 \mid \beta_2$ are left-recursive productions, where β's don't start with **X**, then replace them with $X \rightarrow \beta_1 X' \mid \beta_2 X'$ and $X' \rightarrow \alpha_1 X' \mid \alpha_2 X' \mid \varepsilon$.
For example the left-recursive grammar given above can be transformed to the following non-recursive grammar:

$S \rightarrow aS';\ S' \rightarrow aS' \mid bS' \mid \varepsilon$

Ambiguous grammar :
A context-free grammar is called **ambiguous** if there is at least one string that has more than one distinct derivations (or, equivalently, parse trees). For example, the grammar

$S \rightarrow S + S \mid S * S \mid (S) \mid id$

where ***id*** represents an identifier, produces the following two derivations for the expression *x + y * z*

S => S + S => id + S => id + S * S
=> id + id * S => id + id * id,

which corresponds to *x + (y * z)* and

S => S * S => S + S * S => id + S * S
=> id + id * S => id + id * id,

which corresponds to *(x + y) * z* .

Though some context-free languages are inherently ambiguous and no unambiguous grammars can be constructed for them, it is often possible to construct unambiguous context-free grammars for unambiguous context-free languages. For example, for the language of algebraic expressions given above, the following grammar is unambiguous:

$S \rightarrow S + X \mid X;$
$X \rightarrow X * Y \mid Y;$
$Y \rightarrow (S) \mid id$

Nondeterministic language :
Lastly there are context-free languages that cannot be parsed by a deterministic PDA. This kind of languages need nondeterministic PDAs. Hence guess work is necessary in selecting the right production at certain steps of their derivation. For example take the language of palindromes. When parsing strings for this language, the middle of a given string must be identified. But it can be shown that no deterministic PDA can do that.

CHECK YOUR PROGRESS 1

State True or False

1. {S → aSb; S→ab} generates a context free language that is not regular
2. A PDA is a finite automata with a pushdown stack
3. A language is context free if and only if it is accepted by a PDA
4. {S → aS; S→bS; aSb→bX; X→a; X→b} is context free
5. Any number of nonterminal can appear on the right side of a production of a CFG

4.6 LET US SUM UP

- A CFG is a way to describing languages by recursive rules called productions. A CFG consist of a set of variables, a set of terminal symbols and a start symbol, as well as the productions. Each production consist of a head variable and a body consisting of a string 0f zero or more variable and/or terminals.
- Two basic approaches to parsing are top-down parsing and bottom-up parsing.
- In the top-down approach, a parser tries to derive the given string from the start symbol by rewriting nonterminals one by one using productions.
- In the bottom-up approach, a parser tries to reduce the given string to the start symbol step by step using productions.
- The structure of a derivation of a string in CFG can be represented by a tree called parse tree or a derivation tree
- A context-free grammar is called ambiguous if there is at least one string that has more than one distinct derivation. From which more than one parse tree for same set of strings can be generated.

4.7 FURTHER READINGS

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.

4.8 ANSWERS TO CHECK YOUR PROGRESS

1. True
2. True
3. True
4. False
5. True

4.11 PROBABLE QUESTIONS

1. Design CFG for the following
   a) $\{0^n1^n \; n>0\}$
   b) $\{a^n b^{2n} \; n>0\}$
2. The following grammar generates the language 0*1(0+1)*

   $S \rightarrow A1B$

   $A \rightarrow 0A \mid \varepsilon$

   $B \rightarrow 0B \mid 1B \mid \varepsilon$

Give the leftmost and rightmost derivation for the following

a) 00100

b) 1001

c) 00011

3. For each of the string draw the parse tree for the grammar given in the question no 2.

*****

UNIT 5 : PUSHDOWN AUTOMATA

UNIT STRUCTURE

5.1 LEARNING OBJECTIVES

After going through this unit, you will able to

- Understand Pushdown automata
- Know the languages accepted by Pushdown automata
- Build PDA using Context Free Grammar
- Understand the relationship between CFG and PDA
- Define Deterministic Pushdown Automata

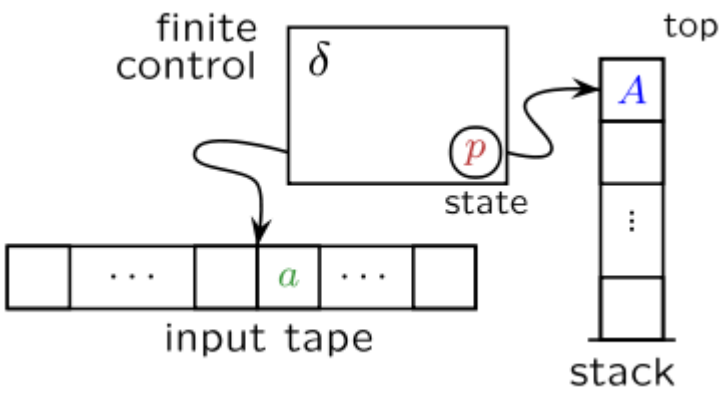
5.2 INTRODUCTION

In the previous units we discussed about FA and CFG , but there are certain limitations of FA . Finite Automata (FA) accept regular languages such as ab* . However, FA do not accept Context-Free Languages such as L= { $a^n cb^n$: where n >= 0 }. It is to be noted that L has strings with a matching number of a's and b's separated by a c . What is interesting about L is that it has a string pattern that is simi-

lar but not exactly the same to that of programming languages such as Java and C++. In fact, syntactic structures of a programming language are defined by Context-Free Grammars in a way that is similar to that of the Context-Free Grammar (CFG) of L given below:

S→aSb|c

This CFG generates or derives a balanced number of a's and b's. Pushdown Automata are designed to accept languages with strings that have similar patterns. That is, a Pushdown Automaton will accept strings like acb, aacbb, aaacbbb, ..., (that is, the strings of L). Pushdown Automata use a stack data structure for matching equal number of a's and b's without counting them directly. A stack is an interesting data-structure which allows operations such as push and pop and increase or decrease its stored contents in a Last-In-First-Out (LIFO) manner. Stacks are used for processing Context-Free Languages.



A diagram of the pushdown automaton

Pushdown automata differ from finite state machines in two ways:
They can use the top of the stack to decide which transition to take.
They can manipulate the stack as part of performing a transition.
Pushdown automata choose a transition by indexing a table by input signal, current state, and the symbol at the top of the stack. This means that those three parameters completely determine the transi-

tion path that is chosen. Finite state machines just look at the input signal and the current state: they have no stack to work with. Push-down automata add the stack as a parameter for choice.

The PDA is used in theories about what can be computed by machines. It is more capable than a finite-state machine but less capable than a Turing machine. Because its input can be described with a formal grammar, it can be used in parser design. The deterministic pushdown automaton can handle all deterministic context-free languages while the nondeterministic version can handle all context-free languages.

5.3 DEFINITION OF THE PUSHDOWN AUTOMATA

A PDA is formally defined as a 7-tuple:
$P=(Q, \sum, \Gamma, \delta, q_0, Z, F)$ where

Q is a finite nonempty set of states
$\sum$ is a finite set which is called the input alphabet
Γ is a finite set which is called the stack alphabet
δ is the transition function from Q X $(\sum \cup \{\wedge\})$ X Γ to the set of finite subsets Q X Γ^* .
q_0 is the start state
$Z \in \Gamma$ is the initial stack symbol
$F \subseteq Q$ is the set of accepting states.

Example 1: Let $M=(Q, \sum, \Gamma, \delta, q_0, Z, F)$ where

$Q=\{q_0,q_1,q_f\}$, $\sum=\{a,b\}$, $\Gamma=\{a, Z\}$, $F=\{q_f\}$

δ is given by $\delta(q_0,a,Z)=\{(q_0,aZ)\}$, $\delta(q_1,b,a)=\{(q_1,\wedge)\}$
$\delta(q_0,a,a)=\{(q_0,aa)\}$, $\delta(q_1,\wedge,Z)=\{(q_1,\wedge)\}$
$\delta(q_0,b,a)=\{(q_1,\wedge)\}$

In the above example to push a symbol on the stack i.e to push ‘a’ on to the stack $\delta(q_0,a,Z)=\{(q_0,aZ)\}$ is used .Similarly to pop a symbol ‘a’

from stack $\delta(q_1,b,a) = \{(q_1,\wedge)\}$ is used. PDA can also behave as do nothing machine, just read the input from the tape and don't make any change to the state and symbol at the stack like $\delta(q_0,a,Z)=\{(q_0,Z)\}$.

Instantaneous Description (ID) :

Let A=(Q, $\sum$, Γ , δ , q_0, Z, F) be a pda. An ID is (q,x,α), where $q \in Q$, $x \in \sum^*$,$\alpha \in \Gamma^*$. For example $(q,abcde....k,Z_1Z_2Z_3Z_4.....Z_m)$ is an ID.This describes the pda when the current state is q, the input string to be processed is abcde.....k. The pda will process abcde....k in that order.The pushdown store/stack (PDS) has $Z_1Z_2Z_3Z_4.....Z_m$ with Z_1 at the top. Z_2 is the second element from the top etc. and Z_m is the lowest element in PDS.

The relation i_1 |-- i_2 means:
PDA P can move in one step from ID i_1 to ID i_2

The relation i_1 |--* i_2 means:
PDA P can move in zero or more steps from ID i_1 to ID i_2

Example 2: Design PDA for the language $L=\{wcw^r \mid w \in \{a,b\}^*\}$.
Let P=(Q, $\sum$, Γ , δ , q_0, Z, F) be the pda
Q={s,f}
$\sum$={a,b,c}
Γ={a,b}
F={f}

$\delta(s,a,\varepsilon)=\{(s,a)\}$, $\delta(s,b,\varepsilon)=\{(s,b)\}$,
$\delta(s,c,\varepsilon)=\{(f,\varepsilon)\}$, $\delta(s,c,a)=\{(f,a)\}$, $\delta(s,c,b)=\{(f,b)\}$
$\delta(s,a,a)=\{(s,aa)\}$, $\delta(s,a,b)=\{(s,ab)\}$
$\delta(s,b,a)=\{(s,ba)\}$, $\delta(s,b,b)=\{(s,bb)\}$
$\delta(f,a,a)=\{(f,\varepsilon)\}$, $\delta(f,b,b)=\{(f,\varepsilon)\}$,$\delta(f,\varepsilon,\varepsilon)=\{(f,\varepsilon)\}$

This automata works in the following way. As it reads the first half of its input, it remains in its initial state and keeps on pushing the symbol on the stack until it reaches the middle symbol 'c'. At this stage it moves to state 'f' and then keeps on poping the symbol it reads from

the tape.

State	Input	Stack
s	abcba	$\in$
s	bcba	a
s	cba	ba
f	ba	ba
f	a	a
f	$\in$	$\in$

5.4 THE LANGUAGES OF A PDA

We have assumed that a PDA accepts its input by consuming it and entering an accepting state. We call this approach acceptance by final state. We may also define for any PDA the language accepted by empty stack, that is, the set of strings that cause the PDA to empty its stack, starting from the initial ID. These two methods are equivalent, in the sense that a language L has a PDA that accepts it by final state if and only if L has a PDA that accepts it by empty stack. However for a given PDA P, the languages that P accepts by final state and by empty stack are usually different. We will show conversion of a PDA accepting L by final state into another PDA that accepts L by empty stack, and vice-versa.

Acceptance by Final State

Let $P = (Q, \sum, \Gamma, \delta, q_0, Z, F)$ be a PDA. Then L(P), the language accepted by P. By final state, is

$L(P) = \{w|(q_0, w, Z) \vdash^* (q,\varepsilon,\alpha)\}$

for some state $q \in F$ and any stack string α. That is, starting in the initial ID with w waiting on the input, P consumes w from the input and enters an accepting state. The content of the stack at that time is irrelevant.

Acceptance by Empty Stack

Let $P = (Q, \sum, \Gamma, \delta, q_0, Z, F)$ be a PDA. Then L(P), the language accepted by P. By empty stack, is

$N(P) = \{w|(q_0, w, Z) \text{ |--* } (q,\varepsilon,\varepsilon) \}$
for any state q. That is, N(P) is the set of inputs w that P can consume and at the same time empty its stack. The N in N(P) stands for null stack, a synonym for empty stack.

From Empty Stack to Final State

Objective of this section is show the conversion from a PDA P_n that accepts a language L by empty stack to a PDA P_f that accepts L by final state.

Theorem: If $L = N(P_n)$ for some PDA $P_n = (Q_n, \sum, \Gamma_n, \delta_n, q_0, Z_0, F_n)$, then there is a PDA $P_f = (Q_f, \sum, \Gamma_f, \delta_f, p_0, X_0, F_f)$ such that $L = L(P_f)$.
Proof: The idea behind the proof is in Figure 1. We use a new symbol X_0, which must not be a symbol of Γ_n; X_0 is both the start symbol of P_f and a marker on the bottom of the stack that lets us know when P_n has reached an empty stack. That is, if P_f sees X_0 on top of the stack, then it knows that P_n would empty its stack on the same input. We also need a new start state, p_0, whose sole function is to push Z_0, the start state of P_n, onto the top of the stack and enter state q_0, the start state of P_n. Then, P_f simulates P_n, until the stack of P_n is empty, which P_f detects because it sees X_0 on the top of the stack. Finally, we need another new state, P_f, which is the accepting state of P_f; this PDA transfers to state P_f whenever it discover that P_n would have emptied its stack.

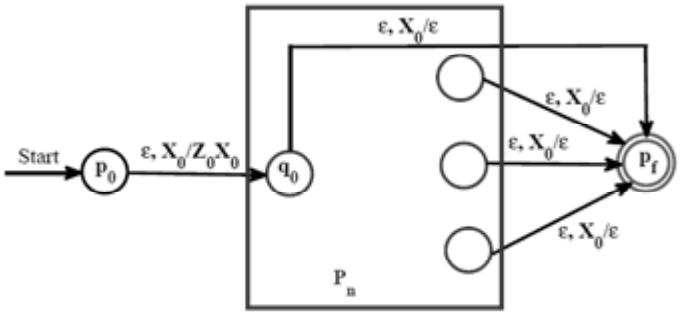


Figure 1: P_f simulates P_n and accepts if P_n empties its stack

The specification of P_f is as follows:

$Q_f = Q_n [\{p_0, p_f\}$.

$\Gamma_f = \Gamma_n U \{X_0\}$.

$F_f = \{p_f\}$.

δ_f is defined by

1. $\delta_f (p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$. In its start state, P_f makes a spontaneous transition to the start state of P_n, pushing its start symbol Z_0 onto the stack.
2. For all state $q \in Q_n$, inputs $a \in \sum_n$ or $a = \varepsilon$, and stak symbol $Y \in \Gamma_n$, $\delta_f (q, a, Y)$ contains all the pairs in $\delta_n (q, a, Y)$.
3. In addition to rule (2), $\delta_f(q, \varepsilon, X0)$ contains (p_f, ε) for every state $q \in Q_n$.

We must show that w is in $L(P_f)$ if and only if w is in $N(P_n)$.

(If) We are given that $(q_0,w, Z_0) \vdash^*_{pn} (q , \varepsilon, \varepsilon)$ for some state q. Insert X_0 at the bottom of the stack and conclude $(q_0,w, Z_0X_0) \vdash^*_{Pn} (q, \varepsilon, X_0)$. Since by rule (2) above, P_f has all the moves of P_n, we may also conclude that $(q_0,w, Z_0X_0) \vdash^*_{Pf} (q, \varepsilon, X_0)$. If we put this sequence of moves with the initial and final moves from rules (1) and (3) above, we get:

$(p_0,w,X_0) \vdash_{pf} (q_0,w, Z_0X_0) \vdash^*_{Pf} (q, \varepsilon, X_0) \vdash_{Pf} (q, \varepsilon, \varepsilon)$

Thus, P_f accepts w by final state.

5.5 EQUIVALENCE OF PDA'S AND CFG'S

From Grammar to Pushdown Automata: Given a CFG G, we construct a PDA that simulates the leftmost derivations of G. Any left-sentential form that is not a terminal string can be written as $xA\alpha$, where A is the leftmost variable, x is whatever terminals appear to its left, and α is the string of terminals and variables that appear to the right of A. We call A the tail of this left-sentential form. If a left-sentential form consists of terminals only, then its tail is ε.

The idea behind the construction of a PDA from a grammar is to have the PDA simulate the sequence of left-sentential forms that the

grammar uses to generate a given terminal string w. The tail of each sentential form $xA\alpha$ appears on the stack, with A at the top. At that time, x will be represented by having consumed x from the input, leaving whatever of w follows its prefix x. That is, if w = xy, then y will remain on the input.

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

Now, $(q, y, \beta\alpha)$ may not be a representation of the next left-sentential form, because β may have a prefix of terminals. In fact, β may have no variables at all, and α may have a prefix of terminals. Whatever terminals appear at the beginning of $\beta\alpha$ need to be removed, to expose the next variable at the top of the stack. These terminals are compared against the next input symbols, to make sure our guesses at the leftmost derivation of input string w are correct; if not, this branch of the PDA dies. If we succeed in this way to guess a leftmost derivation of w, then we shall eventually reach the left-sentential form w. At that point, all the symbols on the stack have either been expanded (if they are variables) or matched against the input (if they are terminals). The stack is empty, and we accept by empty stack.

The above informal construction can be made precise as follows.

Let G = (V, T,R, S)

be a CFG. Construct the PDA P that accepts L(G) by empty stack as follows:

$P = (\{q\}, T, V \cup T, \delta, q, S)$

where transition function δ is defined by:

1. For each variable A, $\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \rightarrow \beta$ is a production of P$\}$.
2. For each terminal a, $\delta(q, a, a) = \{(q, \varepsilon)\}$.

Example: Consider the grammar G = (V, T,R, S) with V = {S}, T = {a, b, c}, and R = {$S \rightarrow aSa$, $S \rightarrow bSb$, $S \rightarrow c$}, which generates

the language $\{wcw^R | w \in \{a, b\}^*\}$. The corresponding pushdown automaton acceptance by empty stack is

$P = (\{q\}, T, V \cup T, \delta, q, S)$, where the transition function δ is given by:

a) $\delta(q, \varepsilon, S) = \{(q, aSa), (q, bSb), (q, c)\}$

b) $\delta(q, a, a) = (q, \varepsilon), (q, b, b) = (q, \varepsilon), (q, c, c) = (q, \varepsilon)$

From PDA's to Grammar

The construction of an equivalent grammar uses variables each of which represent an event consisting of:

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

If $P=(Q, \sum, \Gamma, \delta, q_0, Z_0, F)$ is a PDA, then there is a context-free grammar $G = (V, \sum, R, S)$ such that $L(G) = N(P)$, where the set of variables V consists of :

1. The special symbol S, which is the start symbol of G and
2. All symbols of the form [pXy], where $p, q \in Q$ and $x \in \Gamma$.

The rules R of G are as follows:

a) For all states p, G has the rules $S \rightarrow [q_0 z_0 p]$

(since $(q_0, w, z_0) \vdash^* (p, \varepsilon, \varepsilon)$).

b) Let $\delta(q, a, X)$ contains the pair $(r, Y_1 Y_2 \ldots Y_k)$, where

1. a is either a symbol in $\sum$ or $a = \varepsilon$.
2. k be any number, including 0, in which case the pair is (r, ε).

Then for all lists of states $r_1, r_2, \ldots, r_k$, G has the rules

$[qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2] \ldots [r_{k-1}Y_kr_k]$

This rules says that one way to pop X and go from state q to state r_k is to read a (which may be), then use some input to pop Y_1 off the stack which going from state r to state r_1, then read some more input that pops Y_2 off the stack and goes from state r_1 to state r_2, and so on.

Example: Consider the PDA $PN = (\{q\}, \{0, 1\}, \{Z, A, B\}, \delta_N, q, Z)$ in

Figure 2.The corresponding context-free grammar G = (V, {0, 1},R, S) is given by:

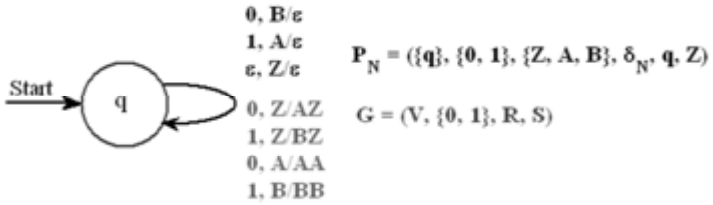


Figure 2: Example of PDA

V = {S, [qZq], [qAq], [qBq]}.

R =

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

5.6 DETERMINISTIC PUSHDOWN AUTOMATA

A deterministic pushdown automaton (DPDA or DPA) is a variation of the pushdown automaton . The DPDA accepts the deterministic context-free languages, a proper subset of context-free languages . A deterministic pushdown automaton: (DPDA) is a 7-tuple P=(Q, $\sum$, Γ , δ , q_0, Z_0, F) where Q, $\sum$, q_0, and F are defined as they are for a deterministic finite automaton, Γ is a finite state (the stack alphabet), and δ maps Q X ($\sum \cup \{\wedge\}$) X Γ to the set of finite subsets Q X Γ^* .We can use any symbols we want in the stack alphabet, Γ. As with state labels, in designing a DPDA, it is important to give symbols names that have meaning. Typically, we use as a special symbol, Z_0 often meaning the bottom of the stack.

We use label arrows in a DPDA as ; a $\in \sum$; b, c $\in \Gamma$

a, b $\rightarrow$ c means if the current input is a and the top-of-stack is b, follow this transition and pop the b off the stack, and push the c.

a,$\varepsilon \rightarrow$ c means if the current input is a, follow this transition and push c on the stack. (It doesn't matter what is on the stack.)

a, b $\rightarrow \varepsilon$ means if the current input is a and the top-of-stack is b, follow this transition and pop the b off the stack.

a, $\varepsilon \rightarrow \varepsilon$ means if the current input is a, follow this transition and don't modify the stack.

CHECK YOUR PROGRESS -1

1. PDA is the machine format of

(a) Type o language (b) Type 1 language

(c) Type 2 language (d) Type 3 language.

2. Which is not true for mechanical diagram of PDA?

(a) PDA contains a stack

(b) The head reads as well as writes

(c) The head moves from left to right

(d) Input string is surrounded by infinite number of blank in both side.

3. The difference between finite automata and PDA is in .

(a) Reading Head (b) Input tape (c) Finite Control (d) Stack

4. Which of the following is not true?

(a) Power of deterministic automata is equivalent to power of non-deterministic automata.

(b) Power of deterministic pushdown automata is equivalent to power of non-deterministic pushdown automata.

(c) Power of deterministic turing machine is equivalent to power of non-deterministic turing machine.

(d) All the above

5.he PDA is called non-deterministic PDA when there are more than one out going edges from…… state

(a)START or READ

(b)POP or REJECT

(c)READ or POP

(d)PUSH or POP

6. Identify the TRUE statement:

(a)A PDA is non-deterministic, if there are more than one READ states in PDA

(b)A PDA is never non-deterministic

(c)Like TG, A PDA can also be non-deterministic

(d)A PDA is non-deterministic, if there are more than one REJECT states in PDA

7. _____ states are called the halt states.
(a)ACCEPT and REJECT
(b)ACCEPT and READ
(c)ACCEPT AND START
(d)ACCEPT AND WRITE

8.Select correct option:
(a)All representations of a regular language are equivalent.
(b)All representations of a context free language are equivalent.
(c)All representations of a recursive language are equivalent
(d)Finite Automata are less powerful than Pushdown Automata.

5.7 LET US SUM UP

1.Pushdown Automata uses a stack data structure.

2. Pushdown automata differ from finite state machines in two ways:They can use the top of the stack to decide which transition to take.They can manipulate the stack as part of performing a transition.

3.A PDA is formally defined as a 7-tuple:
P=(Q, $\sum$, Γ , δ , q_0, Z, F) where

Q is a finite nonempty set of states
$\sum$ is a finite set which is called the input alphabet
Γ is a finite set which is called the stack alphabet
δ is the transition function from Q X ($\sum \cup \{\wedge\}$) X Γ to the set of finite subsets Q X Γ^* .
q_0 is the start state
Z $\in$ Γ is the initial stack symbol
F $\subseteq$ Q is the set of accepting states.

4.There are two methods , in the sense that a language L has a PDA that accepts it by final state if and only if L has a PDA that accepts it by empty stack.

5..Given a CFG G, we can construct a PDA that simulates the leftmost derivations of G.

6.A deterministic pushdown automaton (DPDA or DPA) is a variation of the pushdown automaton . The DPDA accepts the deterministic context-free languages, a proper subset of context-free languages .

5.8 Answers to Check Your Progress-1

1. c, 2. b, 3. d, 4. b, 5.c , 6.c , 7.a , 8.d

5.9 FURTHER READINGS

1. K.L.P. Mishra, N. Chandrasekaran, Theory of Computer Science, BPB Publication, Prentice-Hall of India, Second Edition.
2. H.R. Lewis and C.H.Papadimitriou, Elements of the Theory of Computation, Second Edition, Prentice Hall of India.
3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation,Narosa Publications.
4. J.C. Martin, Introduction to Languages and the Theory of Automata, Tata McGraw-Hill.
5. C.H. Papadimitriou, Computation Complexity, Addison-Wesley.

5.10 POSSIBLE QUESTIONS

Q1. Construct a PDA accepting by empty stack/store each of the languages.

a) $\{a^n b^m a^n \mid m,n \geq 1\}$

b) $\{a^n b^{2n} \mid n \geq 1\}$

c) $\{a^m b^m c^n \mid m,n \geq 1\}$

d) $\{a^m b^n \mid m > n \geq 1\}$

Q2. Construct a PDA accepting by final state each of the languages given in question 1.

Q3. Construct a PDA accepting the set of all even length palindromes over {a,b} by empty stack.

Q4. Show that the set of all strings over {a,b} consisting of equal number of a's and b's is accepted by a deterministic PDA.

Q5. Show that every regular set accepted by a finite automataon with n states is accepted by a deterministic PDA with one one state and n pushdown symbols.

Q6. Construct the equivalent PDA for the following CFGs.

a) $S \rightarrow Saa \mid aSa \mid aaS$

b) $S \rightarrow (S)\ (S) \mid a$

c) $S \rightarrow XaY \mid YbX$

$X \rightarrow YY \mid aY \mid b$

$Y \rightarrow b \mid bb$

Q7. Find the nondeteministic PDA that accepts the following language:

$L = \{ab(ab)^n b\ (ba)^n : n \geq 0\}$

Q8.Design a PDA which converts infix to prefix.

UNIT 6 : PROPERTIES OF CONTEXT-FREE LANGUAGES

UNIT STRUCTURE

6.1 LEARNING OBJECTIVES

After going through this unit, you will able to

- understand the types of normal forms for Context free Grammar
- Convert a Context free grammaras to Chomsky normal form.
- Convert a Context free grammaras to Greibach Normal Form
- Understand how pumping lemma can be used to prove whether a language is context free or not.
- Understand various Closure properties of CFL

6.2 INTRODUCTION

We have seen in previous unit, the class of languages defined by context free grammar and the machine for acceepting those languagesi.e. pushdown automata. Also we have seen how Pushdown Automata can be constructed from a given CFG . Equivalence of CFG and PDA. In this section, we see different normal forms of CFG i.e., one can express the rules of the CFG in a particular form. These normal form grammars are easy to handle and are useful in proving results. The most popular normal forms are Chomsky Nor-

mal Form (CNF), and Greibach Normal Form (GNF). Also we will discuss the properties of context free languages.

6.3 NORMAL FORMS FOR CFGS

It is often convenient to simplify a CFG .One of the simplest and most useful simplified forms of CFG is called the Chomsky normal form. Another normal form usually used in algebraic specifications is Greibach normal form.

Definition

A context-free grammar G is in Chomsky normal form if every rule is of the form:

$A \rightarrow BC$

$A \rightarrow a$

where a is a terminal, A,B,C are nonterminals, and B,C may not be the start variable .

Theorem : Any context-free language is generated by a context-free grammar in Chomsky normal form.

Proof :

• Show that any CFG G can be converted into a CFG G' in Chomsky normal form;

• Conversion procedure has several stages where the rules that violate Chomsky normal form conditions are replaced with equivalent rules that satisfy these conditions.

• Order of transformations:(1) add a new start variable, (2) eliminate all ε-rules, (3) eliminate unit-rules, (4) convert other rules.

• Check that the obtained CFG G' define the same language as the initial CFG G.

Let $G = (N,\Sigma,R, S)$ be the original CFG.

Step 1: add a new start symbol S_0 to N, and the rule

$S_0 \rightarrow S$ to R

This change guarantees that the start symbol of G' does not occur on the rhs of any rule.

Step 2: eliminate ε-rules

Repeat

1. Eliminate the ε- rule $A \to \varepsilon$ from R where A is not the start symbol;

2. For each occurrence of A on the rhs of a rule, add a new rule to R with that occurrence of A deleted.

Examples: (1) replace $B \to uAv$ by $B \to uAv|uv$;

(2) replace $B \to uAvAw$ by $B \to uAvAw|uvAw|aAvw|uvw$.

3. Replace the rule $B \to A$, (if it is present) by $B \to A|\varepsilon$ unless the rule $B \to \varepsilon$ has been previously eliminated;

until all ε rules are eliminated.

Step 3: remove unit rules

Repeat:

1. Remove a unit rule $A \to B \in R$;

2. For each rule $B \to u \in R$, add the rule $A \to u$ to R, unless $B \to u$ was a unit rule previously removed.

until all unit rules are eliminated, u is a string of variables and terminals.

Convert all remaining rules

Repeat:

1. Replace a rule $A \to u_1u_2 \ldots u_k$, $k \geq 3$, where each u_i, $1 \leq i \leq k$, is a variable or a terminal, by:

$A \to u_1A_1$; $A_1 \to u_2A_2, \ldots$; $A_{k-2} \to u_{k-1}u_k$

where $A_1, A_2, \ldots, A_{k-2}$ are new variables;

2. If $k \geq 2$ replace any terminal u_i with a new variable U_i and add the rule $U_i \to u_i$; until no rules of the form $A \to u_1u_2 \ldots u_k$ with $k \geq 3$, remain.

Consider the grammar G6 whose rules are:

$S \to ASA|aB$

$A \to B|S$

$B \to b|\varepsilon$

After first step of transformation we get:

$S_0 \to S$

$S \rightarrow ASA|aB$

$A \rightarrow B|S$

$B \rightarrow b|\varepsilon$

Removing $B \rightarrow \varepsilon$

$S_0 \rightarrow S$

$S \rightarrow ASA \mid aB \mid a$

$A \rightarrow B|S|\varepsilon$

$B \rightarrow b|\varepsilon$

Removing $A \rightarrow \varepsilon$

$S_0 \rightarrow S$

$S \rightarrow ASA \mid aB \mid a \mid AS \mid SA \mid S$

$A \rightarrow B|S$

$B \rightarrow b$

Removing $S \rightarrow S$

$S_0 \rightarrow S$

$S \rightarrow ASA|aB|a|SA|AS$

$A \rightarrow B|S$

$B \rightarrow b$

Removing $S_0 \rightarrow S$

$S \rightarrow ASA|aB|a|SA|AS$

$S_0 \rightarrow ASA|aB|a|SA|AS$

$A \rightarrow B|S$

$B \rightarrow b$

Removing $A \rightarrow B$: and Removing $A \rightarrow S$:

$S \rightarrow ASA|aB|a|SA|AS$

$S_0 \rightarrow ASA|aB|a|SA|AS$

$A \rightarrow ASA|aB|a|SA|AS|b$

$B \rightarrow b$

Converting the remaining rules

$S_0 \rightarrow AA_1|UB|a|SA|AS$

$S \rightarrow AA_1|UB|a|SA|AS$

$A \rightarrow b|AA_1|UB|a|SA|AS$

$A_1 \rightarrow SA$

$U \rightarrow a$

$B \rightarrow b$

Greibach Normal Form (GNF)

A CFG G = (V, T,R, S) is said to be in GNF if every production is 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.

Definition: A production $A \rightarrow A\alpha$ is said to be in the form left recursion, if for some $A \in V$.

• If $A \rightarrow A\alpha_1|A\alpha_2| \ldots |A\alpha_r|\beta_1|\beta_2| \ldots |\beta_s$, then replace the above rules by (i) $Z \rightarrow \alpha_i | \alpha_i Z$, $1 \leq i \leq r$ and (ii) $A \rightarrow \beta_i | \beta_i Z$, $1 \leq i \leq s$

• If G = (V, T,R, S) is a CFG, then we can construct another CFG $G_1 = (V_1, T, R_1, S)$ in Greibach Normal Form (GNF) such that $L(G_1) = L(G) - \{\varepsilon\}$.

The stepwise algorithm is as follows:

1. Eliminate null productions, unit productions and useless symbols from the grammar G and then construct a G′ = (V′, T , R′, S) in Chomsky Normal Form (CNF) generating the language $L(G') = L(G) - \{ \varepsilon \}$.

2. Rename the variables like $A_1, A_2, \ldots A_n$ starting with $S = A_1$.

3. Modify the rules in R′ so that if $A_i \rightarrow A_j\gamma \in R'$ then $j > i$

4. Starting with A_1 and proceeding to A_n this is done as follows:

(a) Assume that productions have been modified so that for $1 \leq i \leq k$, $A_i \rightarrow A_j\gamma \in R'$ only if $j > i$

(b) If $A_k \rightarrow A_j\gamma$ is a production with $j < k$, generate a new set of productions substituting for the A_j the body of each A_j production.

(c) Repeating (b) at most k - 1 times we obtain rules of the form $A_k \rightarrow A_p\gamma$, $p \geq k$

(d) Replace rules $A_k \rightarrow A_k\gamma$ by removing left-recursion as stated above.
5. Modify the $A_i \rightarrow A_j\gamma$ for i = n-1, n-2, ..., 1 in desired form at the same time change the Z production rules.

Example: Convert the following grammar G into Greibach Normal Form (GNF).
$S \rightarrow XA|BB$
$B \rightarrow b|SB$
$X \rightarrow b$
$A \rightarrow a$
To write the above grammar G into GNF, we shall follow the following steps:
1. Rewrite G in Chomsky Normal Form (CNF)
It is already in CNF.
2. Re-label the variables
S with A_1
X with A_2
A with A_3
B with A_4
After re-labeling the grammar looks like:
$A_1 \rightarrow A_2A_3|A_4A_4$
$A_4 \rightarrow b|A_1A_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$
3. Identify all productions which do not conform to any of the types listed below:
$A_i \rightarrow A_jx_k$ such that j > i

4. $A_4 \rightarrow A_1A_4$ ..... identified
5. $A_4 \rightarrow A_1A_4|b$.
To eliminate A_1 we will use the substitution rule $A_1 \rightarrow A_2A_3|A_4A_4$.
Therefore, we have $A_4 \rightarrow A_2A_3A_4|A_4A_4A_4|b$
The above two productions still do not conform to any of the types in step 3. Substituting for $A_2 \rightarrow b$
$A_4 \rightarrow bA_3A_4|A_4A_4A_4|b$

Now we have to remove left recursive production $A_4 \rightarrow A_4A_4A_4$

$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$

$Z \rightarrow A_4A_4|A_4A_4Z$

6. At this stage our grammar now looks like

$A_1 \rightarrow A_2A_3|A_4A_4$

$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$

$Z \rightarrow A_4A_4|A_4A_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

All rules now conform to one of the types in step 3.

But the grammar is still not in Greibach Normal Form!

7. All productions for A_2,A_3 and A_4 are in GNF

for $A_1 \rightarrow A_2A_3|A_4A_4$

Substitute for A_2 and A_4 to convert it to GNF

$A_1 \rightarrow bA_3|bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4$

for $Z \rightarrow A_4A_4|A_4A_4Z$

Substitute for A_4 to convert it to GNF

$Z \rightarrow bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4|bA_3A_4A_4Z|bA_4Z|bA_3A_4ZA_4Z|bZA_4Z$

8. Finally the grammar in GNF is

$A_1 \rightarrow bA_3|bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4$

$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$

$Z \rightarrow bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4|bA_3A_4A_4Z|bA_4Z|bA_3A_4ZA_4Z|bZA_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

6.4 THE PUMPING LEMMA FOR CFGS

The pumping lemma gives us a technique to show that certain languages are not context free .But the pumping lemma for CFL's is a bit more complicated than the pumping lemma for regular languages. Informally- The pumping lemma for CFL's states that for sufficiently long strings in a CFL, we can find two, short, nearby substrings that we can "pump" in tandem and the resulting string must also be in the language.

The Pumping Lemma for CFL's

Let L be a CFL. Then there exists a constant *p* such that if z is any string in L where $|z| \geq p$, then we can write z = uvwxy subject to the following conditions:

1. $|vwx| \leq p$. This says the middle portion is not larger than p.
2. $vx \neq \varepsilon$. We'll pump v and x. One may be empty, but both may not be empty.
3. For all $i \geq 0$, uv^iwx^iy is also in L. That is, we pump both v and x.

Example 1

Let L be the language $\{ 0^n1^n2^n \mid n \geq 1 \}$. Show that this language is not a CFL.

Suppose that L is a CFL. Then some integer p exists and we pick z = $0^p1^p2^p$.

Since z=uvwxy and $|vwx| \leq p$, we know that the string vwx must consist of either:

– all zeros
– all ones
– all twos
– a combination of 0's and 1's
– a combination of 1's and 2's

• The string vwx cannot contain 0's, 1's, and 2's because the string is not large enough to span all three symbols.

• Now "pump down" where i=0. This results in the string uwy and can no longer contain an equal number of 0's, 1's, and 2's because the strings v and x contains at most two of these three symbols. Therefore the result is not in L and therefore L is not a CFL.

Example 2

Let L be the language $\{ a^ib^jc^k \mid 0 \leq i \leq j \leq k \}$. Show that this language is not a CFL. This language is similar to the previous one, except proving that it is not context free requires the examination of more cases.

Suppose that L is a CFL.

Pick z = $a^p b^p c^p$ as we did with the previous language.

As before, the string vwx cannot contain a's, b's, and c's. We then pump the string depending on the string vwx as follows:

– There are no a's. Then we try pumping down to obtain the string uv^0wx^0y to get uwy. This contains the same number of a's, but fewer b'c or c's. Therefore it is not in L.

– There are no b's but there are a's. Then we pump up to obtain the string uv^2wx^2y to give us more a's than b's and this is not in L.

– There are no b's but there are c's. Then we pump down to obtain the string uwy. This string contains the same number of b's but fewer c's, therefore this is not in C.

– There are no c's. Then we pump up to obtain the string uv^2wx^2y to give us more b's or more a's than there are c's, so this is not in C.

Since we can come up with a contradiction for any case, this language is not a CFL language.

6.5 CLOSURE PROPERTIES OF CFL

The class of CFLs is closed under the union ($\cup$) operation.

Proof: Let L1, L2 be any two CFL, we will show that L = L1 $\cup$ L2 is a CFL. Since L1; L2 are CFLs, there must exist CFGs which generate these two languages. Let G1 and G2 generate the languages L1 and L2 respectively, where:

G1 = (V_1; $\sum_1$; R_1; S_1), and

G2 = (V_2; $\sum_2$; R_2; S_2)

We assume that the sets V_1 and V_2 are disjoint, or $V_1 \cap V_2 = \phi$

(we can always assume this because if the sets are not disjoint we can make them so, by renaming variables in one of the grammars).

Consider the following grammar:

G = ($V_1 \cup V_2 \cup \{S\}$; $\sum_1 \cup \sum_2$; $R_1 \cup R_2 \cup \{S \rightarrow S_1|S_2\}$; S)

The above grammar is basically a combination of the grammars G1 and G2 in which we have added the new start state S and a new production rule $S \rightarrow S_1|S_2$. Now we need to show that G generates L. For this we need to show the following two things:

1. For any string $s \in L$, G generates s: We know that either $s \in$ L1 or $s \in$ L2 which implies that either $S_1 \Rightarrow^* s$ or $S_2 \Rightarrow^* s$. Since G has the production $S \rightarrow S_1|S_2$ we can conclude that $S \Rightarrow^* s$ .So G generates s.

2. Let s be any string generated by G, then $s \in L$,: We have $S \Rightarrow^* s$, this means that either $S_1 \Rightarrow^* s$ or $S_2 \Rightarrow^* s$ Now since we have made sure that $V_1 \cap V_2 = \phi$, s is either derived from S_1 using the rules R_1 only or it is derived from S_2 using rules R_2 only. This means that $s \in L_1 \cup L_2$

Example $L = \{ 0^m1^n \mid m \neq n, m, n \geq 0\}$
$L = \{ 0^m1^n \mid m \neq n, m > n \geq 0\} \cup \{ 0^m1^n \mid m \neq n, n > m \geq 0\}$
Hence, L = L(G) for $G = (\{S, S_A, S_B\}, \{0,1\}, R, S)$
where
$R = \{ S \rightarrow S_A \mid S_B,$
$S_A \rightarrow 0 \mid 0S_A \mid 0S_A1,$
$S_B \rightarrow 1 \mid S_B1 \mid 0S_B1\}$

The class of CFLs is closed under concatenation.
Proof: Suppose $A = L(G_A)$ and $B = L(G_B)$ where
$G_A = (V_A, \Sigma_A, R_A, S_A)$
$G_B = (V_B, \Sigma_B, R_B, S_B)$
Without loss of generality, assume $V_A \cap V_B = \phi$.
(Otherwise, we may change some nonterminal symbols.)
Then AB = L(G) for $G=(V, \Sigma, R, S)$ where
$V = V_A \cup V_B \cup \{S\}$
$\Sigma = \Sigma_A \cup \Sigma_B$
$R = R_A \cup R_B \cup \{S \rightarrow S_AS_B \}$

Example : $L = \{xx^R w \mid x \in (0+1)^+, w \in (0+1)^*\}$
$L = \{xx^R \mid x \in (0+1)^* \}\{0,1\}^*$
$L=L(G)$ for $G = (\{S, S_A, S_B\}, \{0, 1\}, R, S)$
where

$R = \{ S \rightarrow S_A S_B,$

$S_A \rightarrow 00 \mid 11 \mid 0S_A0 \mid 1S_A1,$

$S_B \rightarrow \varepsilon \mid 0S_B \mid 1S_B \}$

The class of CFLs is closed under the kleene operation (*).

Proof : Suppose L = (G) for G=(V, $\sum$, R, S)

Then L* = L(G*) for G* =(V, $\sum$, R*, S)

where R* = R $\cup$ { $S \rightarrow \varepsilon \mid SS$}.

S represent L and S* represents L*.

Then $S^* \rightarrow \varepsilon \mid S^*S$. So $S^* \Rightarrow S$.

Example L= (0+1)*00

L=L(G) for G=({S, A}, {0,1}, R, S)

where

R={$S \rightarrow A00$

$A \rightarrow \varepsilon \mid AA \mid 0 \mid 1 \}$

R* = R U { $S \rightarrow \varepsilon \mid SS$ }

= { $S \rightarrow A00 \mid \varepsilon \mid SS$

$A \rightarrow \varepsilon \mid AA \mid 0 \mid 1 \}$

CHECK YOUR PROGRESS -1

1. The intersection of CFL and regular language is

(a) is always regular

(b) is always context free

(c) both (a) and (b)

(d) need not be regular

2. Context free grammer is not closed under

(a) product

(b) union

(c) complementation

(d) kleene star

3.Context free languages are closed under

(a) union,intersection

(b) union,kleene closure

(c) intersection,complement

(d) complement, kleene closure

4. If L_1 = {x | x is a palindrome in (0 + 1)*}

L_2 = {letter (letter + digit)* }; $L_3 = (0^n\ 1^n\ 2^n \mid n > 1\}$

$L_4 = \{a^m b^n a^{m+n} \mid m, n > 1\}$ then which of the following statement is incorrect ?

(a)L_1 is context free language and L_3 is context sensitive language

(b)L_2 is a regular set and L_4 is not a context free language

(c)Both L_1 and L_2 are regular sets

(d)Both L_3 and L_4 are context-sensitive languages.

5. Given A = (0,1) and L = A*. If $R = (0^n\ 1^n, n > 0)$, then language L U R and R are respectively.

(a)regular, regular

(b)not regular, regular

(c)regular, not regular

(d)context free, not regular

6. Define for a context free language L $\leq$ {0 ; 1} init (L) = {u/uv $\in$ L for some v in {0,1}}(in other words, init (L) is the set of prefixes of L)Let L {w/w is noempty and has an equal number of 0's and 1's)Then init (L) is

(a)set of all binary strings with unequal number of 0's and 1's

(b)set of all binary strings including the null string

(c)set of all binary strings with exactly one more 0's than the number of 1's or 1 more than the number of 0's

(d)none of these

7.L = (a^n b^n a^n | n = 1,2,3) is an example of a language that is

(a)context free

(b)not context free

(c)not context free but whose complement is CF

(d)both (b) and (c)

8.Pumping lemma is used for proving that

(a) given grammar is regular

(b) given grammar is not regular

(c) whether two given regular expressions are equivalent or not.

(d) None of these

6.6 LET US SUM UP

1.A context-free grammar G is in Chomsky normal form if every rule is of the form:

$A \rightarrow BC$

$A \rightarrow a$

where a is a terminal, A,B,C are nonterminals, and B,C may not be the start variable .

2.A CFG G = (V, T,R, S) is said to be in GNF if every production is of the form $A \rightarrow a\alpha$, where $a \in T$ and $a \in V^*$, i.e., α is a string of zero or more variables.

3.The pumping lemma gives us a technique to show that certain languages are not context free .

4.The Pumping Lemma for CFL's

Let L be a CFL. Then there exists a constant *p* such that if z is any string in L where $|z| \geq p$, then we can write z = uvwxy subject to the following conditions:

i). $|vwx| \leq p$. This says the middle portion is not larger than p.

ii). $vx \neq \varepsilon$. We'll pump v and x. One may be empty, but both may not be empty.

iii). For all $i \geq 0$, uv^iwx^iy is also in L. That is, we pump both v and x.

5. The class of CFLs is closed under the union (È) operation.

6. The class of CFLs is closed under concatenation.

7. The class of CFLs is closed under the kleene operation (*).

6.7 ANSWERS TO CHECK YOUR PROGRESS

1. b, 2. c, 3. b, 4. a, 5. d, 6. b, 7.d , 8.b

6.8 FURTHER READINGS

1. K.L.P. Mishra, N. Chandrasekaran, Theory of Computer Science, BPB Publication, Prentice-Hall of India, Second Edition.
2. H.R. Lewis and C.H.Papadimitriou, Elements of the Theory of Computation, Second Edition, Prentice Hall of India.
3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation,Narosa Publications.
4. J.C. Martin, Introduction to Languages and the Theory of Automata, Tata McGraw-Hill.
5. C.H. Papadimitriou, Computation Complexity, Addison-Wesley.

6.9 Possible Questions

Q1. Find a reduced grammar equivalent to the grammar

$S \rightarrow aAa$

$A \rightarrow bBB$

$B \rightarrow ab$

$C \rightarrow aB$

Q2. Given the grammar

$S \rightarrow AB$, $A \rightarrow a$, $B \rightarrow C|b$, $C \rightarrow D$, $D \rightarrow E$, $E \rightarrow a$

find an equivalent grammar which is reduced and has no unit productions.

Q3. Reduce the following grammars to chomsky normal form

a) $S \rightarrow 1A \mid 0B$, $A \rightarrow 1AA \mid 0S \mid 0$, $B \rightarrow 0BB \mid 1S \mid 1$

b) $S \rightarrow a \mid b \mid cSS$

c) $S \rightarrow abSb \mid a \mid aAb$, $A \rightarrow bS \mid aAAb$

Q4. Reduce the following grammars to Greibach normal form:

a) $S \rightarrow SS$, $S \rightarrow 0S1 \mid 01$

b) $S \rightarrow AB$, $A \rightarrow BSB$, $A \rightarrow BB$, $B \rightarrow aAb$

$B \rightarrow a$, $A \rightarrow b$

c) $S \rightarrow A0$, $A \rightarrow 0B$, $B \rightarrow A0$, $B \rightarrow 1$

Q4. Show that the following are not context free languages:

a)The set of all strings over {a,b,c} in which the number of occurrences of a,b,c is the same.

b) $\{ a^m b^m c^n \mid m \leq n \leq 2m \}$

c) $\{ a^m b^n \mid n=m^2 \}$

UNIT 7 : INTRODUCTION TO TURING MACHINE

UNIT STRUCTURE

7.1 LEARNING OBJECTIVES

After going through this unit, you will able to

- understand the most powerful abstract model of a computing device,the Turing machine.
- understand undecidable problems , the problems that computer cannot solve
- Understand Turing machine
- Understand the programming techinques to recognize any language by computer program.
- describe multi tape Turing machine
- understand the concept of Universal Turing machine

7.2 INTRODUCTION

We have seen several abstract models of computing devices: Deterministic Finite Automata, Nondeterministic Finite Automata, Nondeterministic Finite Automata with ε-Transitions, Pushdown Automata,and Deterministic Pushdown Automata.However, none of the above “seem to be” as powerful as a real computer.We now turn

our attention to a much more powerful abstract model of a computing device: a Turing machine. This model is believed to do everything that a real computer can do.

Turing machines are extremely simple calculating devices. A Turning machine remembers only one number, called its state. It moves back and forth along an infinite tape, scanning and writing symbols and changing its state. Its action at a given step in the calculation is based on only two factors: its current state number and the symbol that it is currently scanning on the tape. It continues in this way until it enters a special state called the halt state. In spite of their simplicity, Turing machines can perform any calculation that can be performed by any computer. In fact, certain individual Turing machines, called universal Turing machines, can actually execute arbitrary programs, just as a computer can.

7.3 PROBLEMS THAT COMPUTERS CANNOT SOLVE

It is important to know whether a program is correct, namely that it does what we expect. It is easy to see that the following C program

```
main()
{
printf(‘‘hello, world\n’’);
}
```

prints hello, world and terminates.

Femat’s theorem expressed the hello-world program as

```
main()
{
int n, total, x, y, z;
scanf(“%”, &n);
total = 3;
while (1) {
for (x = 1; x <= total -2; x++)
for (y = 1; y <= total -1 ; y++) {
```

```
z = total - x -y;
if (exp(x, n) + exp (y, n) == exp(z, n))
printf (“hello, word”);
}
total ++
}
}
```

The program (Fermat) takes an input n and looks for positive integer solutions to equation

If the program finds a solution, it prints hello, world . If it never finds integer x, y, z to satisfy the equation, then it continues searching forever, and never prints hello, world .If the value of n is 2, then it will find combinations of integers and thus:
For input n = 2 the program prints hello, world
For any integer n > 2, the program will never find a triple of positive integers to satisfy $x^n + y^n = z^n$.

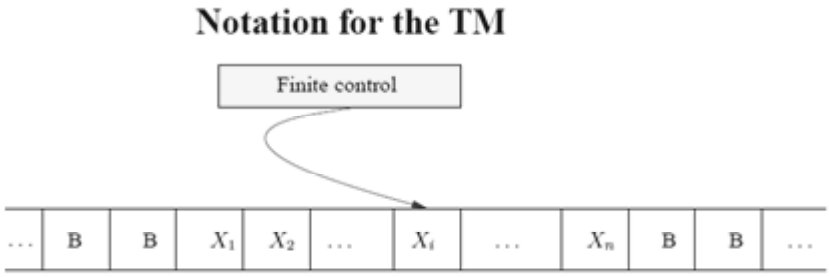
The Hypothetical “Hello World” Tester

Is it possible to have a program that could examine any program P and input I for P, and tell whether P, run with I as its input, would print hello,world?

Assume there is a program (H) that takes as input , a program P, input I and tells whether P within input I prints hello, world (Output is either Yes or No) .If a problem has an algorithm like H, that always tells correctly whether an instance of the problem has answer Yes or No, then the problem is said to be decidable. Otherwise, the problem

is undecidable. We need to prove that H does not exist.

7.4 THE TURNING MACHINE



Finite control: can be in any of a finite set of states
Tape: divided into cells; each cell can hold one of a finite number of symbols.Initially the **input** (a finite-length string) is placed on the tape
All other tape cells initially hold a special symbol: **blank (B)**
Blank is tape symbol (not an input symbol)
Tape head: always positioned at one of the tape cell. Initially, the tape head is at the leftmost cell that holds the input.

A move of the TM is a function of the state of the finite control and the tape symbol scanned. In one move the TM will
1. Change state
2. Write a tape symbol in the cell scanned.
3. Move the tape head left or right.

Definition : A Turing Machine is a 7 tuple

$M = (Q, \sum, \Gamma, \delta, q_0, B, F)$ where

Q : The finite set of states of the finite control
$\sum$: The finite set of input symbols
Γ : The complete set of tape symbols .$\sum$ is always a subset of Γ.
δ : The transition function. The arguments of $\delta(q,X)$ are: a state q and a tape symbol X. The value of $\delta(q,X)$, if it is defined, is (p, Y, D)

where: p is the next state, in Q .Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.D is a direction (either Left or Right), telling us the direction in which the head moves.

q_0 : The start state (q0 $\in$ Q) in which the finite control is found initially.

B : blank symbol (B $\in \Gamma$ but B $\notin \sum$).

F : the set of final or accepting states (F $\subseteq$ Q).

Instantaneous Descriptions for TM

We use the instantaneous description to describe the configuration.

An ID is represented by the string:

$X_1X_2 \ldots X_{i-1}qX_iX_{i+1} \ldots X_n$

where:

1. q is the state of the TM.

2. The tape head is scanning the ith symbol from the left.

3. $X_1X_2 \ldots X_n$ is the portion of the tape between the leftmost and the rightmost nonblank.

Moves in TM

Let M = (Q, $\sum$,Γ , δ, q_0,B, F)

We use the notation $|--_M$ (or |--) to represent moves of a TM M from one configuration to another.$|--^*_M$ is used as usual.

The next move is leftward:

If $\delta(q,X_i) = (p, Y, L)$ then:

$X_1X_2 \ldots X_{i-1}\, q\, X_i X_{i+1} \ldots X_n \;|--_M\; X_1X_2 \ldots X_{i-2}\, p\, X_{i-1}\, Y\, X_{i+1} \ldots X_n$

Exceptions:

1. If i = 1, then M moves to the blank to the left of X1

$q\, X_1X_2 \ldots \ldots X_n \;|--_M\; p\, B\, Y\, X_2 \ldots X_n$

2. If i = n and Y = B, then the symbol B written over X_n joins the infinite sequence of trailing blanks and does not appear in the next ID.

$X_1X_2 . . . . . X_{n-1}qX_n \ |--_M \ X_1X_2 . . . . . X_{n-2}pX_{n-1}$

The next move is rightward:

If $\delta(q,X_i) = (p, Y, R)$ then:

$X_1X_2 . . . X_{i-1} \ q \ X_i X_{i+1} . . . X_n \ |--_M \ X_1 X_2 . . . X_{i-2} \ X_{i-1} \ Y \ p \ X_{i+1} . . . X_n$

Exceptions:

1. If i = n, then the i + 1st cell holds a blank, and that cell was not part of the previous ID.

$X_1X_2 . . . X_{n-1} \ q \ X_n \ |--_M \ X_1 X_2 . . . X_{n-2} \ X_{n-1} \ Y \ p \ B$

2. If i = 1 and Y = B, then the symbol B written over X_1 joins the infinite sequence of trailing blanks and does not appear in the next ID.

$q \ X_1X_2 . . . . X_n \ |--_M \ p \ X_2 . . . . . X_n$

Example :

A TM for the language $\{0^n1^n \mid n \geq 1\}$

$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, , q_0, B, \{q_4\})$

State	Symbol				
	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

$q_0$0011 |-- X$q_1$011 |-- X0$q_1$11 |-- X$q_2$0Y 1 |-- q_2X0Y 1 |-- X$q_0$0Y 1
|-- XXq_1Y 1 |-- XXY $q_1$1 |-- XXq_2Y Y |-- Xq_2XY Y |-- XXq_0Y Y
|-- XXY q_3Y |-- XXY Y q_3B |-- XXY Y Bq_4B

7.5 PROGRAMMING TECHNIQUES FOR TURNING MACHINES

Writing down Turing machines for complicated languages can be dificult and boring. But one can use some programming techniques. The goal of this section is to convince the reader that Turing machines are indeed powerful enough to recognize any language that a computer program can recognize.

1. Storing a tape symbol in the finite control: We can build a TM whose states are pairs [q , X] where q is a state, and X is a tape symbol. The second component can be used in remembering a particular tape symbol. Consider the following TM that recognizes the language

L = ab* + ba*

The machine reads the first symbol, remembers it in the finite control, and checks that the same symbol does not appear anywhere else in the input word:

$\delta(q_0, a) = ([q, a], a, R)$

$\delta(q_0, b) = ([q, b], b, R)$

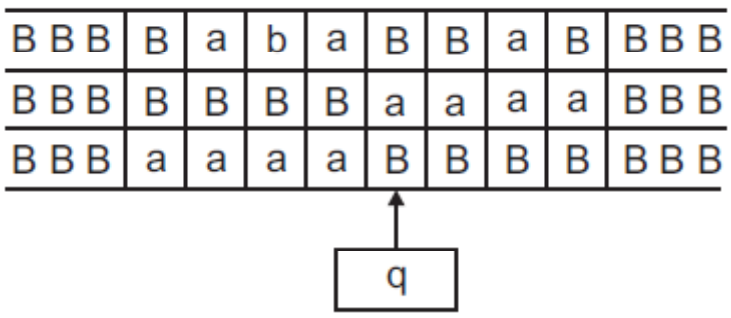
$\delta([q, a], b) = ([q, a], b, R)$

$\delta([q, b], a) = ([q, b], a, R)$

$\delta([q, a] B) = (q_F, B, R)$

$\delta([q, b], B) = (q_F, B, R)$

2. Multiple tracks: Sometimes it is useful to imagine that the tape consists of multiple tracks. We can store different intermediate information on different tracks:



For example, we can construct a TM with 3 track tape that recog-

nizes the language L = { a^p | p is a prime number } as follows. Initially the input is written on the first track and the other two tracks contain B's. (This means we identify a with [a, B, B] and B with [B, B, B].) The machine operates as follows.

It first checks the small cases: If the input is empty or a then the machine halts in a non-final state; if the input is aa it halts in the final state. Otherwise, the machine starts by placing two a's on the second track. Then it repeats the following instructions:

1. Copy the content of the first track to the third track.
2. Subtract the number on the second track from the third track as many times as possible. If the third track becomes empty, halt in a non-final state. (The number on the first track was divisible by the number on the second track.)
3. Increment the number on the second track by one. If the number becomes the same as the number on the first track halt in the final state . Else go back to step 1.

3. Checking of symbols. This simply means that we introduce a second track where we can place blank B or symbol √. The tick mark can be conveniently used in remembering which letters of the input have been already processed. It is useful when we have to count or compare letters.

For example, consider the language

L = {ww | w ∈ (a + b)* }

We first use the tick mark to find the center of the input word: Mark alternatively the first and last unmarked letters, one-by-one. The last letter to be marked is in the center. So we know where the second w should start. Using the "Storing a tape symbol in the finite control" - technique,one can check one-by-one the letters to verify that the letters in the first half and the second half are identical.

4. Shifting over: This means adding an new cell at the current location of the tape. This can be established by shifting all symbols one position to the right by scanning the tape from the current position to the right, remembering the content of the previous cell in the finite control, and writing it to the next cell on the right. Once the rightmost non-blank symbol is reached the machine can return to the new va-

cant cell that was introduced. (In order to recognize the rightmost non-blank symbol, it is convenient to introduce an end-of-tape symbol that is written in the first cell after the last non-blank symbol.)

5. Subroutines: We can use subroutines in TM in an analogous way as they are used in normal programming languages. A subroutine uses its own set of states, including its own "initial state" q'_0 and a return state q_r. To call a subroutine, the calling TM simply changes the state to q'_0 and makes sure the read-write head is positioned on the leftmost symbol of the "parameter list" to the subroutine.

Constructing TM to perform specific tasks can be quite complicated. Even to recognize some simple languages may require many states and complicated constructions. However, TM are powerful enough to be able to simulate any computer program. The claim that Turing machines can compute everything that is computable using any model of computation is known as Church-Turing thesis. Since the thesis talks about any model of computation, it can never be proved. But so far TM have been able to simulate all other models of computation that have been proposed. As an example, let us see how a Turing machine would simulate a register machine, a realistic model of a conventional computer. The tape contains all data the computer has in its memory. The data can be organized for example in such a way that word v_i in memory location i is stored on the tape as the word

$$\# 0_i * v_i \#$$

where # and * are special marker symbols. The contents of the registers of the CPU are stored on their own tracks on the tape.

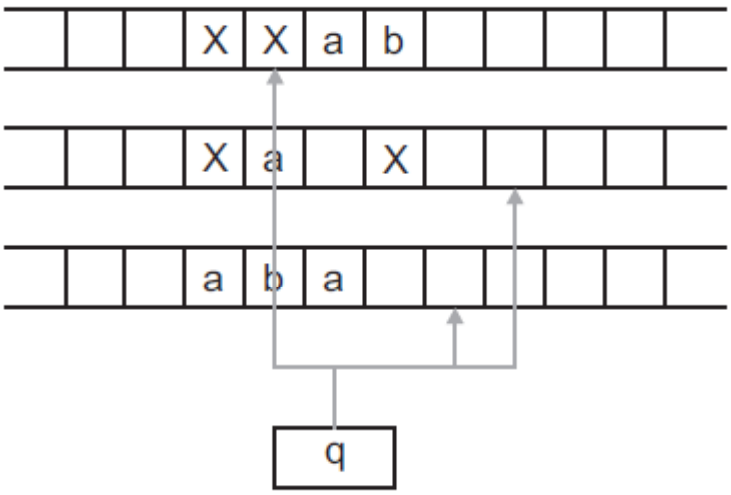
To execute the next instruction, the TM finds the memory location addressed by the specific Program Counter register. In order to do that the TM goes through all memory locations one by one and - using the tick marks - counts if the address i is the same as the content of the Program Counter register. When it finds the correct memory location i, it reads the instruction v_i and memorizes it in the finite control. There are only finitely many different instructions. To each instruction corresponds its own subroutine. To simulate the

instruction, the TM can use the same tick marking to find any required memory locations, and then execute the particular task. The task may be adding the content of a register to another register, for example. Adding two numbers can be easily implemented (especially if we decide to represent all number in the unary format so that number n is represented as the word a^n). Loading a word from the memory to a register is simple as well. To write a word to the memory may require shifting all cells on the right hand side of the memory location, but we know how to do that.

7.6 EXTENSIONS TO THE BASIC TURNING MACHINES

In this section one modifications to our TM model are briefly described. The variations are equivalent: They recognize exactly the same family of r.e. languages as the basic model.

Multiple tape TM : We can allow the TM to have more than one tape. Each tape has its own independent R/W head. This is different from the one tape TM with multiple tracks since the R/W heads of different tapes can now be at different positions.



Depending on the state of the finite control and the current tape symbols on all tapes the machine can change the state,overwrite

the currently scanned symbols on all tapes, and move each R/W head to left or right independently of each other.

Formally, the transition function δ is now a (partial) function from

$(Q \setminus \{ f \}) \times \Gamma^n$

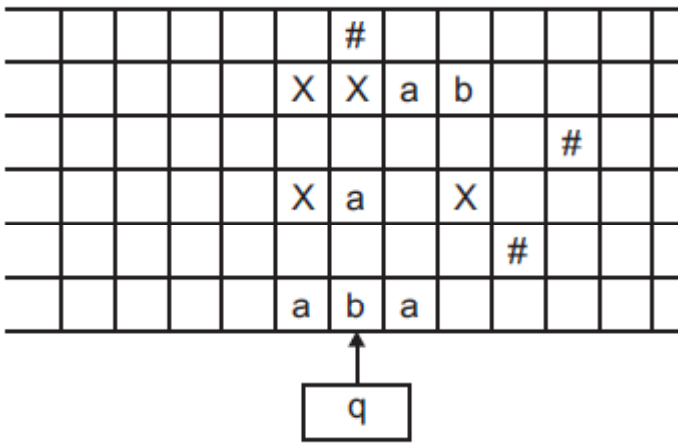
to $Q \times \Gamma^n \times \{ L , R \}^n$

where n is the number of tapes. The transition

$\delta(q , X_1 , X_2 ....., X_n) = (p, Y_1 , Y_2 , ... Y_n , d_1 , d_2 ....., d_n)$

(where $d_1 , d_2 ....., d_n \in \{ L , R \}$) means that the machine, in state q, reading symbols $X_1 , X_2 ....., X_n$ on the n tapes, changes its state to p, writes symbols $Y_1 , Y_2 , ... Y_n$ on the tapes, and moves the first, second, third, etc. R/W head to the directions indicated by $d_1 , d_2 ....., d_n$, respectively. Initially,the input is written on tape number one, and all other tapes are blank. A word is accepted if the machine eventually enters the final state f.

Let us see how a one tape TM can simulate an n-tape TM M. The single tape will have 2n tracks -- two tracks for every tape of M: One of the tracks contains the data of the corresponding tape in M; The other one contains a single symbol # indicating the position of the R/W head on that tape. The single R/W-head of the one-tape machine is located on the leftmost indicator #. For example, the 3-tape configuration illustrated above would be represented by the following ID with six tracks:



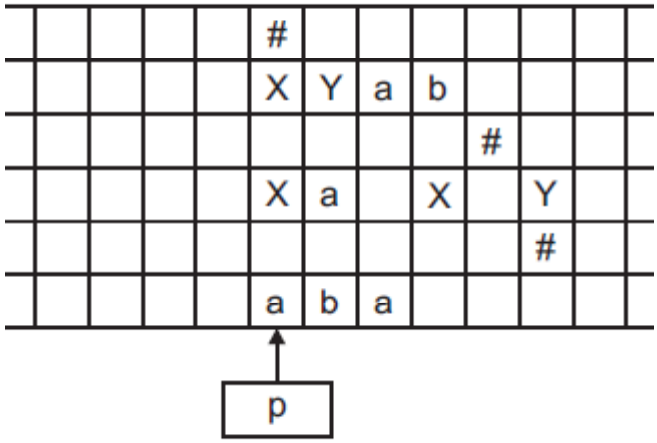
To simulate one move of the multitape machine M, the one-tape machine scans the tape from left to right, remembering in the finite

control the tape symbols indicated by the symbols #. Once all #'s have been encountered, the machine can figure out the new state p and the action taken on each tape. During another sweep over the tape, the machine can execute the instruction by writing the required symbols and moving the #'s on the tape left or right.

If, for example, we have

δ(q ,X ,B ,B) = (p , Y, Y, B, L, L ,R)

after one simulation round the one tape machine will be in the ID



Note that simulating one step of the multitape machine requires scanning through the input twice,so the one-tape machine will be much slower. But all that matters is that the machines accept exactly the same words. It is clear that multitape TM recognize exactly the family of r.e. languages,and multitape TM that halt with all inputs recognize exactly the family of recursive languages.

7.7 TURING MACHINE AND COMPUTERS

A Turing Machine is the mathematical tool equivalent to a digital computer. It was suggested by the mathematician Turing in the 30s, and has been since then the most widely used model of computation in computability and complexity theory. The problem with Turing Machines is that a different one must be constructed for every new computation to be performed, for every input output relation.This is

why the notion of a universal turing machine (UTM), was introduce which along with the input on the tape, takes in the description of a machine M. The UTM can go on then to simulate M on the rest of the contents of the input tape. A universal turing machine can thus simulate any other specific Turing machine, by defining states and symbols. The UTM is defined with certain capabilities. The UTM can define the symbols that the specific Turing machine will use. It can define the symbols that encode the states and transition rules for the specific Turing machine. It can encode the rules for that specific Turing machine onto the input tape. A single-tape UTM needs to define a marker to mark the end of the "specific" program and the start of the specific machine's initial tape. It must also shuffle the read/write head between the specific TM's program and its data. As noted, it is simpler to describe a UTM with multiple tapes.

The Universal Turing Machine is remarkably similar to the Von Neumann model of a computer, where both programs and data can be stored on the same medium. Any modern computer capable of copying a program file from one medium to another, and later running that program, follows this architecture.

The Universal Turing Machine Emulates Other Turing Machines

As noted, it is easier to describe any UTM as having three tapes, although it does not require them. The first tape encodes the set of states for the specific Turing machine to be emulated. The second tape is an input for that specific TM. The third tape is a working memory for the current state of the emulated machine.

The UTM's program must begin by reading the "program tape" to learn the initial state, and note this on the "status" tape. The UTM's states follow the following processes.

1. Read the current cell in the "data tape".
2. Read the "program tape" to find the instruction for the current status and the current data cell, and note this on the "status tape". This instruction includes the new state.
3. If the new state is "halt", then set the UTM itself into the "halt"

state; otherwise proceed to step 4.

4. Apply the instruction from the state to the “data tape”. This might rewrite a cell, and move the “data tape” to the right or left.
5. Update the “status tape”.
6. Continue at step #1 above.

Eventually the Universal Turing machine’s “data tape” will be identical to the tape produced by the standard Turing machine it is emulating, if the UTM is programmed correctly and given the same initial “data tape” as that regular Turing machine.

As well, both should either halt or continue processing forever. If they halt, they would do so in the same “accept” or “reject” state.

The statement that “the UTM emulates the specific Turning machine” means that the final state, and the data tape at completion, will be identical between the UTM and the specific Turing machine it is emulating. Clearly, the UTM must perform more steps than the machine it emulates. In the list above, steps 2 and 5 are extra. A single-tape UTM takes many extra steps to shuffle between the emulated program and the data, both of which are stored on the one tape. Of course, if a specific machine should fail to halt (for a particular input), then the UTM also would continue processing forever.

CHECK YOUR PROGRESS -1

1.Please choose the statement which is true?
(a)The tape of turing machine is infinite.
(b)The tape of turing machine is finite.
(c)The tape of turing machine is infinite when the language is regular
(d)The tape of turing machine is finite when the language is nonregular.

2.The language { ww| $w \in (0+1)^*$) is
(a) not accepted by an Turing machine
(b) accepted by some Turing machine, but by no push down automation
(c) accepted by some push down automation, but not context free
(d) context-free, but not regular.

3.Which of the following questions is ambiguous, according to Turing?
(a) Can a machine play the imitation game?
(b) Can a machine think?
(c) Can a machine be self-aware?
(d) Can a machine express emotions?

4.The statement, "A TM can't solve halting problem" is
(a) true (b) false (c) still an open question
(d) all of these

5. If there exists a TM which when applied to any problem in the class, terminates, if correct answer is yes and may or may not terminate otherwise is called
(a)stable (b)unsolvable
(c)partially solvable (d)unstable

6.Given a Turing machine T and a step-counting function f, is the language accepted by T in Time(f) ?This decision problem is

(a) solvable (b)unsolvable
(c)uncertain (d)none of these

7. A total recursive function is a
(a) partial recursive function (b)premitive recursive function
(c) both (a) and (b) (d)none of these

8. Bounded minimalization is a technique for
(a)proving whether a promotive recursive function is turning computable or not
(b)proving whether a primitive recursive function is a total function or not
(c)generating primitive recursive functions
(d)generating partial recursive functions

9.Universal TM influenced the concept of
(a) stored program computers
(b)interpretative implementation of program¬ming language
(c)computability
(d)all of these

10.A FSM can be considered, having finite tape length without rewinding capability and unidirectional tape movement
(a.) Turing machine
(b.) Pushdown automata
(c.) Context free languages
(d.) Regular languages

7.8 LET US SUM UP

1.If a problem has an algorithm like H, that always tells correctly whether an instance of the problem has answer Yes or No, then the problem is said to be decidable.

2.A Turing Machine is a 7 tuple: $M = (Q, å, G, d, q_0, B, F)$

3.An ID is represented by the string: $X_1X_2 \ldots X_{i-1}qX_iX_{i+1} \ldots X_n$ where q is the state of the TM. The tape head is scanning the ith symbol from the left.. $X_1X_2 \ldots X_n$ is the portion of the tape between the leftmost and the rightmost nonblank.

4. The notation $|--_M$ (or |--) to represent moves of a TM M from one configuration to another.$|--^*_M$ is used as usual.

5.We can build a TM whose states are pairs [q , X] where q is a state, and X is a tape symbol. The second component can be used in remembering a particular tape symbol.

6.Sometimes it is useful to imagine that the tape consists of multiple tracks. We can store different intermediate information on different tracks:

7.We can use subroutines in TM in an analogous way as they are used in normal programming languages.

8.We can allow the TM to have more than one tape. Each tape has its own independent R/W head. This is different from the one tape TM with multiple tracks since the R/W heads of different tapes can now be at different positions.

9. A universal turing machine can simulate any other specific Turing machine, by defining states and symbols. The UTM is defined with certain capabilities.

7.9 ANSWERS TO CHECK YOUR PROGRESS

1. a, 2.b , 3.b , 4. a, 5.c , 6.b , 7.d , 8.c, 9.d, 10.a

7.10 FURTHER READINGS

1. K.L.P. Mishra, N. Chandrasekaran, Theory of Computer Science, BPB Publication, Prentice-Hall of India, Second Edition.
2. H.R. Lewis and C.H.Papadimitriou, Elements of the Theory of Computation, Second Edition, Prentice Hall of India.
3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation,Narosa Publications.
4. J.C. Martin, Introduction to Languages and the Theory of Automata, Tata McGraw-Hill.
5. C.H. Papadimitriou, Computation Complexity, Addison-Wesley.

7.11 POSSIBLE QUESTIONS

Q1. Build a Turing Machine that accepts the language

$L=\{ a^n b^{n+1} \}$

Q2. Build a Turing Machine that accepts the language

$L=\{ b^n c^{2n} \}$

Q3.Build a Turing machine that accepts the language of all words that contain the substring bbb.

Q4. Build a Turing machine that accepts the language ODD PALINDROME.

Q5. Build a Turing machine that accepts all strings with more a's than b's, the language MORE.

Q6.Construct the Turing machine for the following languages:

a) aba^*b

b) $L = \{ w : |w| \text{ is even} \}$

c) $L = \{ w : |w| \text{ is a multiple of 3} \}$

d)$L= \{ a^n b^m a^{n+m} : n \geq 0 , m \geq 1 \}$

Q7. Prove that the following functions are computable functions :

a) $f(x)=3x$

b) $f(a,b)=2a+3b$

c) $f(a)=a \bmod 5$

d) $f(a,b) = a-b$ if $a>b$

$f(a,b) = 0$ if $a \leq b$

UNIT 8 : UNDECIDABILITY

UNIT STRUCTURE

8.1 LEARNING OBJECTIVES

After going through this unit, you will able to

- understand Recursively enumerable languages.
- understand Undecidable problem those are Recursively enumerable.
- understand and solve Post's Correspondence problem.
- understand Other Undecidable problems.

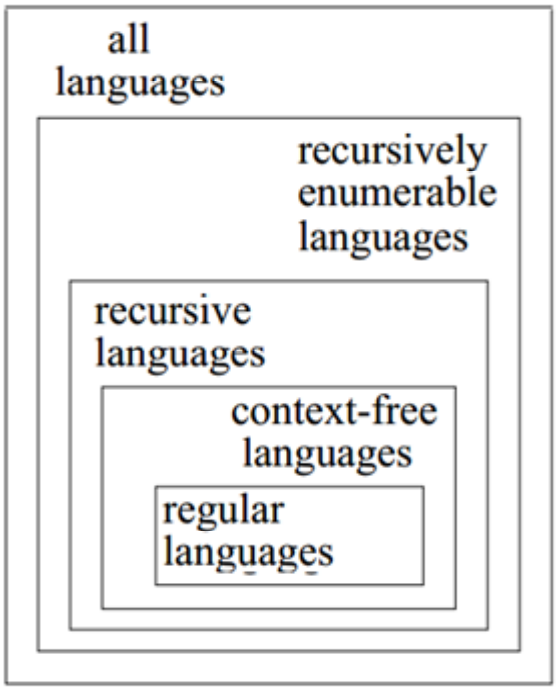
8.2 INTRODUCTION

In the previous unit we discussed about undecidable problems , the problems that computer cannot solve also the programming techinques to recognize any language by computer program and the multi tape Turing machine. In this unit we will discuss recursively enumerable languages and see if there exists a Turing machine that accepts every string of the language. We will also see how to find a non-r.e. language, using diagonalization.

8.3 A LANGUAGE THAT IS NOT RECURSIVELY ENUMERABLE

There are three possible outcomes of executing a Turing machine over a given input. The Turing machine may
1.Halt and accept the input.
2.Halt and reject the input or 3. Never halt.
A language is recursive if there exists a Turing machine that accepts every string of the language and rejects every string (over the same alphabet) that is not in the language.Note that, if a language L is recursive, then its complement must also be recursive.

A language is recursively enumerable if there exists a Turing machine that accepts every string of the language, and does not accept strings that are not in the language. Strings that are not in the language may be rejected or may cause the Turing machine to go into an infinite loop.Clearly, every recursive language is also recursively enumerable. It is not obvious whether every recursively enumerable language is also recursive.



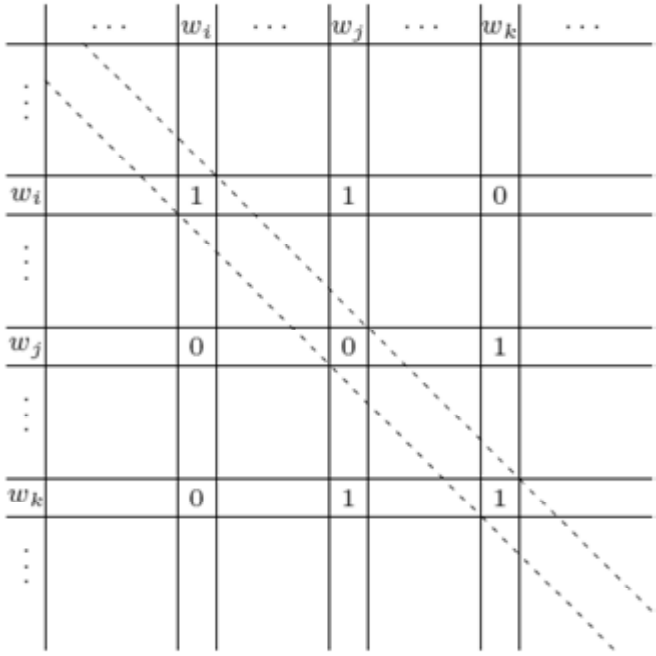
Theorem: Some languages are not recursively enumerable.
Proof: The set of strings is an infinite countable set. The set of languages is not countable because it is the powerset of the set of strings. Recursively enumerable languages are countable because TMs are countable. Therefore, recursively enumerable languages is a subset of all languages.

In this section, we will use a technique called diagonalization to find a natural language that isn't recursively enumerable. This will lead us to a language that is recursively enumerable but is not recursive. It will also enable us to prove the undecidability of the halting problem.

Diagonalization
To find a non-r.e. language, we can use diagonalization. Let Σ be the alphabet used to describe programs: the letters and digits, plus the elements of { comma, perc, tilde, openPar, closPar, less, great}. Every element of Σ^* either describes a unique closed program, or describes no closed programs.
Given $w \in \Sigma^*$, we write L(w) for:
• ϕ, if w doesn't describe a closed program; and
• L(pr), where pr is the unique closed program described by w, if w does describe a closed program. Thus L(w) will always be a set of strings, even though it won't always be a language.

Consider the infinite table of 0's and 1's in which both the rows and the columns are indexed by the elements of ", listed in ascending order according to our standard total ordering, and where a cell (w_n,w_m) contains 1 iff $w_n \in L(w_m)$, and contains 0 iff $w_n \notin L(w_m)$. Each recursively enumerable language is $L(w_n)$ for some (non-unique) n, but not all the $L(w_n)$ are languages.

Here is how part of this table might look, where w_i, w_j and w_k are sample elements of Σ^*.
Because of the table's data, we have that $w_i \in L(w_j)$ and $w_j \notin L(w_i)$



To define a non r.e. Σ language, we work our way down the diagonal of the table, putting w_n into our language just when cell (w_n,w_n) of the table is 0, i.e., when $w_n \notin L(w_n)$.

With our example table:

• $L(w_i)$ is not our language, since $w_i \in L(w_i)$, but w_i is not in our language;

• $L(w_j)$ is not our language, since $w_j \notin L(w_j)$, but w_j is in our language; and

• $L(w_k)$ is not our language, since $w_k \in L(w_k)$, but w_k is not in our language.

In general, there is no $n \in N$ such that $L(w_n)$ is our language.Consequently our language is not recursively enumerable.

We formalize the above ideas as follows. Define languages L_d ("d" for"diagonal") and L_a ("a" for "accepted") by:

$L_d = \{w \in \Sigma^* \mid w \notin L(w)\}$, and $L_a = \{w \in \Sigma^* \mid w \in L(w)\}$.

Thus $L_d = \Sigma^* - L_a$.

We have that, for all $w \in \Sigma^*$, $w \in L_a$ iff $w \in L(pr)$, where pr is the

unique closed program described by w.

Theorem : L_d is not recursively enumerable.
Proof. Suppose, toward a contradiction, that L_d is recursively enumerable. Thus, there is a closed program pr such that $L_d = L(pr)$.
Let $w \in \Sigma^*$ be the string describing pr. Thus $L(w) = L(pr) = L_d$.
There are two cases to consider.
• Suppose $w \in L_d$. Then $w \notin L(w) = L_d$—contradiction.
• Suppose $w \notin L_d$. Since $w \in \Sigma^*$, we have that
$w \in L(w) = L_d$—contradiction.
Since we obtained a contradiction in both cases, we have an overall contradiction. Thus L_d is not recursively enumerable.

8.4 AN UNDECIDABLE PROBLEM THAT IS RE

An Undecidable problem that is RE : Halting Problem

Decidability : The problem of decidability may be stated roughly as follows: is it possible for an algorithm to correctly answer a yes/no question for all possible input?

For example:
Is there an algorithm that will tell us whether or not two arbitrary DFAs recognize the same language?

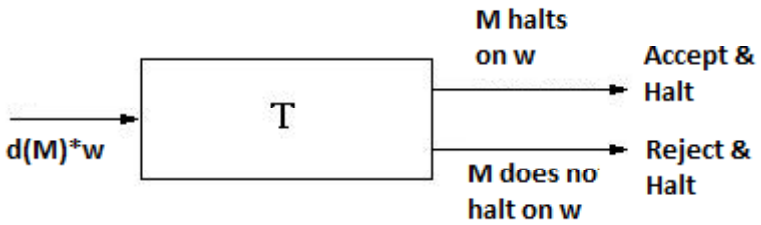
Is there an algorithm that will tell us whether or not two arbitrary context-free grammars generate the same language?

Given an arbitrary Turing machine and initial tape, will the Turing machine reach the Halt state?

A problem is decidable if such an algorithm exsits. The first problem (deciding whether or not two DFAs are equivalent) is decidable. The second two problems are undecidable: there is no algorithm that can correctly answer these questions for all possible input. The last problem (whether or not a Turing machine will reach the Halt state

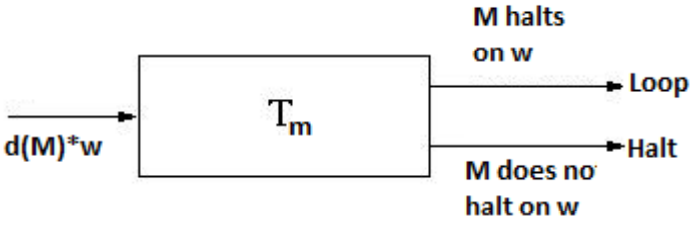
for some initial tape) is known as the Halting Problem, and is a very famous problem in the theory of computation.
Theorem : The halting problem is undecidable.

Proof : This is going to be proven by "proof by contradiction". Suppose that the halting problem is decidable. Then there is a Turing machine T that solves the halting problem. That is, given a description of a Turing machine M (over the alphabet Σ) and a string w, T writes "yes" if M halts on w and "no" if M does not halt on w, and then T halts.



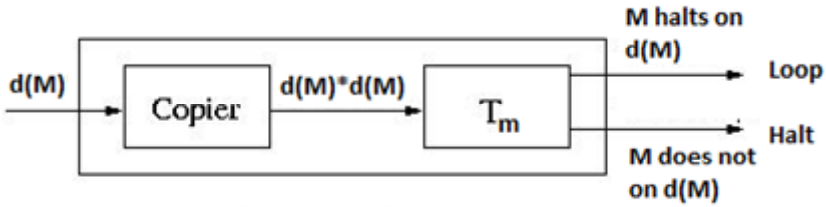
Turing machine T

We are now going to construct the following new Turing machine T_c. First we construct a Turing machine T_m by modifying T so that if T accepts a string and halts, then T_m goes into an infinite loop (T_m halts if the original T rejects a string and halts).



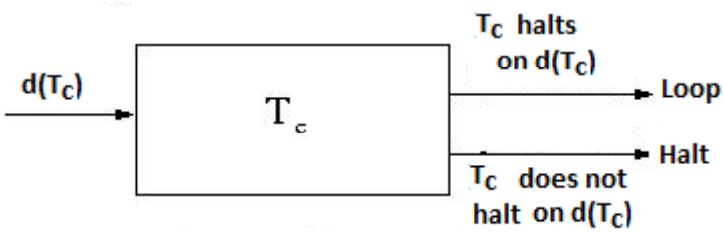
Turing machine T_m

Next using T_m we are going to construct another Turing machine Tc as follows: T_c takes as input a description of a Turing machine M, denoted by d(M), copies it to obtain the string d(M)*d(M), where * is a symbol that separates the two copies of d(M) and then supplies d(M)*d(M) to the Turing machine T_m .



Turing machine T_c

Let us now see what T_c does when a string describing T_c itself is given to it. When T_c gets the input $d(T_c)$, it makes a copy, constructs the string $d(T_c)*d(T_c)$ and gives it to the modified T. Thus the modified T is given a description of Turing machine Tc and the string $d(T_c)$.



Turing machine T_c on input $d(T_c)$

The way T was modified the modified T is going to go into an infinite loop if Tc halts on $d(T_c)$ and halts if T_c does not halt on $d(T_c)$.
Thus T_c goes into an infinite loop if T_c halts on $d(T_c)$ and it halts if T_c does not halt on $d(T_c)$. This is a contradiction. This contradiction has been deduced from our assumption that there is a Turing machine that solves the halting problem. Hence that assumption must be wrong. Hence there is no Turing machine that solves the halting problem.

8.5 POST CORRESPONDENCE PROBLEM

An instance to Post correspondence problem (PCP) consists of two lists of words over some alphabet §:

$L_1 : w_1 , w_2 , \ldots\ldots\ldots w_k$

$L_2 : x_1 , x_2 , \ldots\ldots\ldots x_k$

Both lists contain equally many words. We say that each pair (w_i , x_i)

forms a pair of corresponding words. A solution to the instance is any non-empty string $i_1 i_2 \ldots\ldots i_m$ of indices from {1, 2 ,....., k } such that

$w_{i1}w_{i2} \ldots\ldots\ldots\ldots w_{im} = x_{i1}x_{i2} \ldots\ldots\ldots\ldots x_{im}$:

In other words, we concatenate corresponding words w_i and x_i to form two words. We have a solution if the concatenated w_i's form the same word as the corresponding concatenated x_i's. The PCP asks whether a given instance has a solution or not. It turns out that PCP is undecidable.

Example Consider the following two lists:

$L_1 : a^2 , b^2 , ab^2$

$L_2 : a^2b , ba , b$

This instance has solution 1213 because

$w_1w_2w_1w_3$ = aa bb aa abb

$x_1x_2x_1x_3$ = aab ba aab b

are identical.

Example : The PCP instance

$L_1 : a^2b , a$

$L_2 : a^2 , ba^2$

does not have a solution: If it would have a solution, the solution would need to start with index 1. Since $w_1 = a^2b$ and $x_1 = a^2$, the second list has to catch up the missing b: The second index has to be 2. Because $w_1w_2 = a^2ba$ and $x_1x_2 = a^2ba^2$ the first list has to catch up. The next index cannot be 1 because $w_1w_2w_1 = a^2baa^2b$ and $x_1x_2x_1 = a^2ba^2a^2$ differ in the 7'th letter. So the third index is 2, yielding $w_1w_2w_2 = a^2baa$ and $x_1x_2x_2 = a^2ba^2ba^2$. Now the first list has to catch up ba^2 which is not possible since neither w_1 nor w_2 starts with letter b.

8.6 UNDECIDABLE PROBLEMS

1. The problem of determining if a word w is in the language generated by a grammar G is undecidable.

2. The problem of deciding if two grammars G1 and G2 gener -ate the same language is undecidable.

3. The problem of determining validity in the predicate calculus is undecidable language is undecidable.

4. The problem of determining the universality of a context-free language, i.e., the problem of determining if for a context-free grammar G one has $L(G) = \Sigma^*$ is undecidable.

5. The problem of determining the emptiness of the intersec -tion of context-free languages is undecidable.

6. The problem is to determine if, for two context-free gram mars G1 and G2, one has $L(G1) \cap L(G2) = \phi$.

7. Hilbert's tenth problem is undecidable. This problem is to de -termine if an equation

$$p(x_1 , x_2 , \ldots\ldots\ldots\ldots , x_n) = 0.$$

CHECK YOUR PROGRESS-1

1. The following problem(s) ---------- is/are called decidable problem(s).

(a)The two regular expressions define the same language

(b)The two FAs are equivalent

(c) Both a and b

(d)None of given

2.If there exists a language L, for which there exists a TM, T, that accepts every word in L and either rejects or loops for every word that is not in L, is called

(a)recursive

(b)recursively enumerable

(c)NP-HARD

(d)none of these

3.Which of the following statement(s) is/are correct?

(a)L = $\{a^n b^n a^n \mid n = 1, 2, 3...\}$ is recursively enumerable

(b)Recursive languages are closed under union

(c)Every recursive is closed under union

(d) All of these

4.Recursively enumerable languages are not closed under

(a) Complementation

(b) Union

(c) Intersection

(d) None of the above

5.Which of the following statement is wrong ?

(a) Recursive languages are closed under union.

(b) Recursive languages are closed under complementation.

(c) If a language and its complement are both regular then the language must be recursive.

(d) A language is accepted by FA if and only if it is recursive

8.7 LET US SUM UP

1.There are three possible outcomes of executing a Turing machine over a given input. The Turing machine may Halt and accept the input, Halt and reject the input or Never halt.

2.A language is recursively enumerable if there exists a Turing machine that accepts every string of the language, and does not accept strings that are not in the language.

3.Some languages are not recursively enumerable.

4.To find a non-r.e. language, we can use diagonalization.

5.The problem of decidability may be stated roughly as follows: is it possible for an algorithm to correctly answer a yes/no question for all possible input?

6.The halting problem is undecidable.

7.The problem of determining if a word w is in the language generated by a grammar G is undecidable.

8.The problem of deciding if two grammars G1 and G2 generate the same language is undecidable.

9.The problem of determining validity in the predicate calculus is undecidable language is undecidable.

10.The problem of determining the emptiness of the intersection of context-free languages is undecidable.

11.The problem is to determine if, for two context-free grammars G1 and G2, one has $L(G1) \cap L(G2) = \phi$.

8.8 Answers to Check Your Progress-1

1. c, 2. b, 3. d, 4. a, 5. d, 6.

8.9 Further Readings

1. K.L.P. Mishra, N. Chandrasekaran, Theory of Computer Science, BPB Publication, Prentice-Hall of India, Second Edition.
2. H.R. Lewis and C.H.Papadimitriou, Elements of the Theory of Computation, Second Edition, Prentice Hall of India.
3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation,Narosa Publications.
4. J.C. Martin, Introduction to Languages and the Theory of Automata, Tata McGraw-Hill.
5. C.H. Papadimitriou, Computation Complexity, Addison-Wesley.

8.10 Possible Questions

Q1. Prove that PCP with { (01,011), (1,10), (1,11)} has no solution.

Q2. Does the PCP with x=(b^3,ab^2) and y=(b^3,bab^3) have a solution.

Q3.Prove that there is no algorithm that can determine whether or not a given TM evantually halts with complete blank tape when it starts with a given tape configuration.

Q4.Prove that the problem of determining whether or not aa TM over {0,1} will ever print the symbol 1, with a given tape configuration is unsolvable.

Q5.Comment on the following : "We have developed an algorithm so complicated that no Turing machine can be constructed to execute the algorithm no matter how much (tape) space and time is allowed".

Q6.Prove that PCP is solvable if $|\Sigma|=1$.

Q7.Let x=(x_1.....x_n) and y=(y_1.....y_n) be two list of non empty strings over Σ and $|\Sigma|>2$.

i) Is PCP solvable for n=1?

ii) Is PCP solvable for n=2?