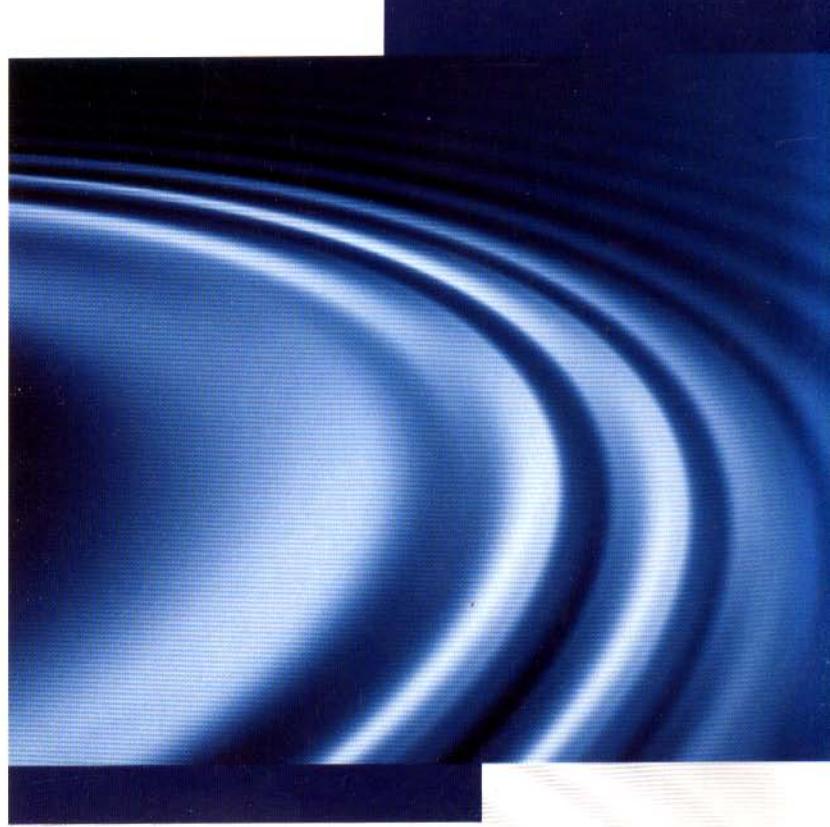
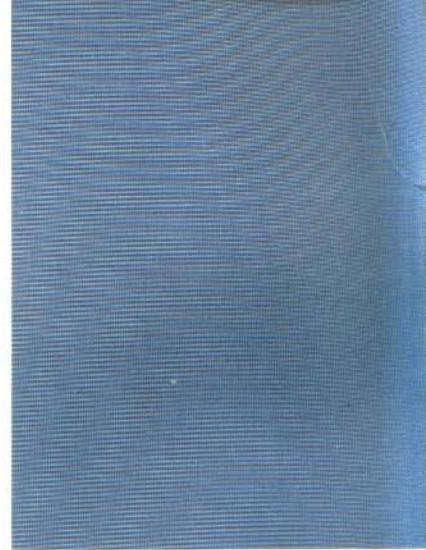


Finite Automata and Formal Languages

A Simple Approach



A. M. Padma Reddy

10125 + 23

Finite Automata and Formal Languages



REHBERG & BROOK

Finite Automata and Formal Languages

A Simple Approach

A.M. Padma Reddy

Professor and Head

Department of Computer Science & Information Science

Sai Vidya Institute of Technology

Rajanukunte, Bangalore – 560 064, India



PEARSON

Table of Contents

Chapter 1. Introduction to Finite Automata

1.1. Introduction	1
1.2. Basic Notations and Terminologies Used in FAFL	4
1.2.1. Alphabet	4
1.2.2. String	5
1.2.3. Language	9
1.3. Finite Automata	11
1.4. Deterministic Finite Automata (DFA)	14
1.5. Simple Notations for DFA's	17
1.5.1. Transition Diagram	17
1.5.2. Transition Table	18
1.5.3. How a DFA Processes Strings	19
1.5.4. Extended Transition Function of DFA to Strings	20
1.5.5. Language Accepted by a DFA	22
1.6. DFA Design Techniques	25
1.6.1. Pattern Recognition Problems	25

1.6.2. Divisible by k Problems	52
1.6.3. Modulo k Counter Problems	57
1.7. Applications of Finite Automata	79
Exercises	80
1.8. Disadvantages of DFA	82
1.9. Why NFA?	82
1.10. Non-Deterministic Finite Automaton	83
1.10.1. Moves made by NFA	86
1.10.2. Extended Transition Function of NFA to Strings	87
1.10.3. Language Accepted by a NFA	90
1.11. Conversion from NFA to DFA	93
1.11.1. Conversion from NFA to DFA (Subset Construct Method)	93
1.11.2. Disadvantage of Subset Construction Method	97
1.11.3. Conversion from NFA to DFA (Lazy Evaluation Method)	98
Exercises	109

Chapter 2. Finite Automata and Regular Expressions

2.1. ϵ -NFA (Finite Automata with Epsilon Transitions)	111
2.2. Extended Transition Function of ϵ -NFA to Strings	115
2.3. Conversion from ϵ -NFA to DFA (Algorithm to Convert ϵ -NFA to DFA)	116
2.4. Difference Between DFA, NFA and ϵ -NFA	130
2.5. Regular expressions	130
2.6. Finite Automata and Regular Expressions	143
2.6.1. To Obtain ϵ -NFA from Regular Expression	144
2.6.2. To Obtain RE from FA (Kleene's Theorem)	149
2.6.3. To Obtain RE from FA (By Eliminating States)	157
2.7. Applications of Regular Expressions	161
Exercises	164

Chapter 3. Regular Languages and Properties of Regular Languages

3.1. Proving Languages Not to be Regular	167
3.2. Pumping Lemma for Regular Languages	168
3.3. Applications of Pumping Lemma	170
Exercises	177
3.4. Properties of Regular Languages	178
3.4.1. Regular Languages are Closed Under Union, Concatenation and Star	179
3.4.2. Closure Under Complementation	179
3.4.3. Closure Under Intersection	180
3.4.4. Closure Under Difference	183
3.4.5. Closure Under Reversal	183
3.4.6. Closure Under Homomorphism	185
3.5. Limitations of Finite Automaton	187
3.6. Equivalence and Minimization of Finite Automata	188
3.6.1. Equivalence of Two States	189
3.6.2. Table Filling Algorithm	189
3.6.3. Minimization of DFA (Algorithm or Procedure)	190
Exercises	206

Chapter 4. Context Free Grammars and Languages

4.1. Grammar	209
4.2. Chomsky Hierarchy	211
4.2.1. Type 0 Grammar	211
4.2.2. Type 1 Grammar or Context Sensitive Grammar	212
4.2.3. Type 2 Grammar or Context Free Grammar	213
4.2.4. Type 3 Grammar or Regular Grammar	213
4.3. Grammar from Finite Automata	215
4.4. Grammar from Regular Expressions	222

4.5. Derivation	223
4.5.1. Sentence	224
4.5.2. Language	225
4.6. Grammars for Other Languages	226
4.7. Leftmost Derivation	248
4.8. Rightmost Derivation	250
4.9. Derivation Tree (Parse Tree)	251
4.10. Ambiguous Grammar	253
4.11. Applications of Context Free Grammars	264
Exercises	268

Chapter 5. Pushdown Automata

5.1. Transitions	276
5.2. Graphical Representation of a PDA	280
5.3. Instantaneous Description	281
5.4. Acceptance of a Language by PDA	282
5.5. Construction of PDA	283
5.6. Deterministic and Non-deterministic PDA	305
5.7. CFG to PDA	310
5.8. Application of GNF	315
5.9. PDA to CFG	317
Exercises	321

Chapter 6. Properties of Context Free Languages

6.1. Substitution	326
6.2. Left Recursion	327
6.3. Simplification of CFG	329
6.4. Eliminating ϵ -productions	336

6.5. Eliminating Unit Productions	340
6.6. Chomsky Normal Form	347
6.7. Greibach Normal Form (GNF)	351
Exercise	359
Properties of Context Free Languages	363
6.8. Pumping Lemma	364
Applications of Pumping Lemma for CFLs	367
6.9. CFLs are Closed Under Union, Concatenation and Star	374
6.10. CFLs are Not Closed Under Intersection	376
6.11. CFLs are Not Closed Under Complementation	377
Exercises	378

Chapter 7. Turing Machines

7.1. Turing Machine Model	379
7.2. Transition Table	380
7.3. Instantaneous Description (ID)	382
7.4. Acceptance of a Language by TM	385
7.5. Construction of Turing Machine (TM)	386
7.6. Transition Diagram for Turing Machine (TM)	390
7.7. Transducers	410
7.8. Multi-tape Turing Machines	422
7.9. Equivalence of Single Tape and Multi-tape TM's	423
7.10. Nondeterministic Turing Machines	424
7.11. Turing Machine with Stay-option	427
Exercises	427
Programming Techniques for Turing Machines	427
Achieving Complicated Tasks Using TM	427
7.12. Multiple Tracks (Multi Track)	428

7.13. Subroutines	428
7.14. Turing Machine with Semi-infinite Tape	437
7.15. Multi-stack Machines	438
7.16. Counter Machines	439
7.17. Off-line Turing Machine	439
7.18. Linear Bounded Automata (LBA)	440
Exercises	441
 Chapter 8. Undecidability	
8.1. Introduction	443
8.2. A Language that is Not Recursively Enumerable	444
8.3. Undecidable Problems that are RE	444
8.4. Halting Problem	445
8.5. Post's Correspondence Problem	445
8.6. Church Turing Hypothesis (Church's/Church-Turing Thesis)	446
Church-Turing Thesis	447
Bibliography	449
Index	451

Chapter 1

Introduction to Finite Automata

What are we studying in this chapter . . .

- ▶ *Introduction*
- ▶ *Central concepts of Automata Theory*
- ▶ *Deterministic Finite Automata*
- ▶ *Non-deterministic Finite Automata*

1.1. Introduction

Before understanding the various terminologies and notations, let us first know some simple but very important mathematical notations used in set theory. First, let us see “What is a set?” Give the example.

❖ **Definition:** A *set* is a collection of distinct elements. All the elements of the set should be enclosed between ‘{’ and ‘}’ separated by commas.

■ Example 1: The set of positive integers greater than 0 and less than or equal to 25 which are divisible by 5 can be represented as

$$S = \{5, 10, 15, 20, 25\}$$

The elements of a set can be in any order. The above set can also be written as

$$S = \{10, 20, 5, 25, 15\}$$

Note: Observe that no elements are repeated in this set. This representation is useful if the number of elements is less. The above set can also be represented by describing the properties of the elements as shown below:

■ Example 2: The set of integers greater than 0, less than or equal to 25 and which are divisible by 5 can be represented as

$$S = \{5x \mid x \text{ is a positive integer where } 0 < x \leq 5\}$$

Note: If x is an element in the set S , we write $x \in S$. If x is not an element in the set S , we write $x \notin S$. The sets are denoted by capital letters such as A, B, C, \dots etc. and the elements within the set are denoted by lower case letters such as a, b, c, \dots etc.

Now, let us see “What is an empty set?”.

❖ **Definition:** A set which has no elements is called an *empty set* or *null set* and is denoted by $\{\}$ or ϕ . For example, the set S that does not contain any element can be represented as shown below:

$$S = \{\} \text{ or } S = \phi$$

Now, let us see “What is a subset?”.

❖ **Definition:** A set A is a subset of set B if every element of set A is an element of set B and is denoted by

$$A \subseteq B$$

If $A \subseteq B$ and B contain an element which is not in A , then A is a proper subset of B and is denoted by $A \subset B$.

Now, let us see “When we say that two sets are equal?”.

❖ **Definition:** The two sets A and B are same (i.e., $A = B$) iff $A \subseteq B$ and $B \subseteq A$ i.e., every element of set A is an element of set B and every element of set B is also an element of set A . For example, if

$$A = \{a, b, c\} \text{ and } B = \{a, b, c\} \text{ then set } A = B.$$

Now, let us see “What is a power set?”.

❖ **Definition:** Let A be the set. The set of all subsets of set A is called *power set* of A and is denoted by 2^A .

Example: Let $A = \{1, 2, 3\}$. The subsets of the set A are shown below:

$$\{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}, \{\}$$

The set of these subsets is called ***power set*** and is denoted by 2^A .

$$\text{i.e., } 2^A = \{ \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}, \{\} \}$$

Note: $|A|$ denotes the number of elements in set A and $2^{|A|}$ denote the number of subsets of set A. In the above example, $|A| = 3$ i.e., the number of items in A and $2^{|A|} = 8$ i.e., the number of items in 2^A .

Now, let us see “What is Cartesian product (cross product) of two sets?”.

❖ **Definition:** The Cartesian product of A and B is given by $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$. Here, (a, b) is an ordered pair such that ‘a’ is an element of set A and ‘b’ is an element of set B.

Example: Let $A = \{a, b, c\}$, $B = \{0, 1\}$. The cross product of A and B is given by

$$A \times B = \{(a,0), (a,1), (b,0), (b,1), (c,0), (c,1)\}$$

Now, let us see “What is union of two sets and intersection of two sets?”.

❖ **Definition:** The union of two sets A and B is given by $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.

Example: Let $A = \{a, b, c\}$ $B = \{0, 1\}$. Union of A and B is given by

$$A \cup B = \{a, b, c, 0, 1\}$$

❖ **Definition:** The intersection of two sets A and B is given by

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$

which is the collection of common elements in both the sets A and B.

■ **Example 1:** Let $A = \{a, b, c\}$ $B = \{c, d, e\}$. The intersection of A and B is given by

$$\begin{aligned} A \cap B &= \{a, b, c\} \cap \{c, d, e\} \\ &= \{c\} \end{aligned}$$

Note: If two sets A and B have no common elements then the two sets are called Disjoint Sets.

■ Example 2: Let $A = \{a, b, c\}$ $B = \{0, 1\}$. The intersection of A and B is given by

$$\begin{aligned}A \cap B &= \{ \{a, b, c\} \cap \{0, 1\}\} \\A \cap B &= \emptyset\end{aligned}$$

Now, let us see “What is the difference of two sets? What is complement of a set?”.

❖ **Definition:** The difference of two sets A and B is given by

$$A - B = \{x \mid x \in A \text{ and } x \notin B\}$$

Example: Let $A = \{a, b, c, d\}$ $B = \{a, d\}$. Obtain $A - B$.

$$A - B = \{ \{a, b, c, d\} - \{a, d\} \} = \{b, c\}$$

❖ **Definition:** The complement of A is denoted by \bar{A} and is defined as a set containing everything that is not in A. Formally, complement of A is defined as follows:

$$\bar{A} = U - A \text{ where } U \text{ is the universal set. i.e.,}$$

$$\bar{A} = \{x \mid x \in U \text{ and } x \notin A\}$$

Example: Let $U = \{1, 2, 3, 4, 5, 6\}$ $A = \{1, 2\}$. Obtain \bar{A}

$$\begin{aligned}\bar{A} &= U - A \\&= \{1, 2, 3, 4, 5, 6\} - \{1, 2\} \\&= \{3, 4, 5, 6\}\end{aligned}$$

1.2. Basic Notations and Terminologies Used in FAFL

Before we see the definition of Finite Automata and Formal Languages (FAFL), let us have familiarity with basic notations and terminologies used in this book.

1.2.1. Alphabet

First, let us “Define an alphabet with example”.

❖ **Definition:** A language consists of various symbols from which the words, statements etc., can be obtained. These symbols are called alphabets. Formally, an alphabet is defined as a finite non-empty set of symbols. The symbol Σ denotes the set of alphabets of a language.

Example 1: The alphabets of C language has the letters from A to Z, a to z, digits from 0 to 9, symbols such as +, -, *, /, (,), {}, etc. and is denoted by

$$\Sigma = \{a, b, \dots, z, A, B, \dots, Z, 0, 1, \dots, 9, \#, (,), \{, \}, <, >, !, [], \dots\}$$

Example 2: The machine Language is made up of only 0's and 1's and so, the alphabets of machine language is denoted by

$$\Sigma = \{0, 1\}$$

1.2.2. String

Now, let us see “What is a string? Explain with example”.

❖ **Definition:** The sequence of symbols obtained from the alphabets of a language is called a string. Formally, a *string* is defined as a finite sequence of symbols from the alphabet Σ . **Note:** An empty string is denoted by the symbol ϵ (pronounced as epsilon) or λ (pronounced as lambda) and note that $\epsilon \notin \Sigma$ i.e., ϵ is not part of Σ .

Note: Let us use the symbol ϵ indicating an empty string instead of the symbol λ .

Example 1: Let $\Sigma = \{0, 1\}$ is set of alphabets. The various strings that can be obtained from Σ are

$$\{0, 1, 00, 01, 10, 11, 010101, 1010, \dots\}$$

Note: Note that an infinite number of strings can be generated from Σ and once the string is generated, it has finite number of symbols in it and has a definite sequence.

Notations used: Normal notations used in this subject are shown below:

- The symbol ϵ is used to denote an empty string.
- The lowercase letters a, b, c, etc along with the symbols such as +, -, (,), {}, and so on are used to denote the symbols in Σ .
- The lowercase letters such as u, v, w, x, y z are normally used to indicate the strings. For example, we can write

$$w = 010101$$

where the symbols 0 and 1 are in Σ and the letter w denotes the string with a specific value.

Now, let us see “What is concatenation of two strings?”.

❖ **Definition:** The concatenation of two strings u and v is the string obtained by writing the letters of string u followed by the letters of string v (i.e., appending the symbols of v to the right of u) i.e., if

$$u = a_1 a_2 a_3 \dots a_n$$

and

$$v = b_1 b_2 b_3 \dots b_m$$

then the concatenation of u and v is denoted by

$$uv = a_1 a_2 a_3 \dots a_n b_1 b_2 b_3 \dots b_m$$

■ **Example 1:** Let the two strings u and v be

$$u = \text{Computer}$$

and

$$v = \text{Science}$$

The concatenation of u and v denoted by uv will be

$$uv = \text{ComputerScience}$$

Note: Concatenation is not commutative. For example, let

$$u = \text{"RAMA"} \quad v = \text{"KRISHNA"}$$

Then $uv = \text{"RAMAKRISHNA"}$ and $vu = \text{"KRISHNARAMA"}$. So,

$$uv \neq vu$$

Now, let us see “What is sub string? What is suffix? What is prefix?” Give example.

❖ **Definition:** Let w is a string obtained from the symbols in Σ . The string w if it can be decomposed into three strings x , y and z such that

$$w = xyz$$

then x is a sub string, y is a substring and z is a substring of string w .

❖ **Definition:** A *prefix* is string of any number of leading symbols.

■ **Example 1:** Let w is the string and let $w = xyz$. The string w has prefix ϵ (empty string), x , xy , and xyz . In the string “Rama”, the various prefixes are ϵ , “R”, “Ra”, “Ram” and “Rama”.

❖ **Definition:** The *suffix* is a string of any number of trailing symbols.

■ **Example 1:** If $w = xyz$, then the string w has suffix ϵ (empty string), z , yz , and xyz . In the string “Rama”, the strings ϵ , “a”, “ma”, “ama” and “Rama” are all the suffixes.

Now, let us see “What is reversal of a string?”.

Definition: The *reversal* of a string is obtained by writing the symbols in reverse order i.e., if

$$u = a_1 a_2 a_3 \dots a_n$$

then the reverse of u is denoted by u^R and is given by

$$u^R = a_n a_{n-1} a_{n-2} \dots a_3 a_2 a_1$$

So, if u is an empty string denoted by ϵ and u has only one symbol then,

$$\epsilon^R = \epsilon$$

$$a^R = a$$

The reverse of a string can be defined recursively as follows:

❖ **Definition:** If a is the symbol and w is the string derived from the alphabet Σ , the reverse of a string can be defined as

$$w^R = \begin{cases} \epsilon & \text{if } w = \epsilon \\ a & \text{if } w = a \\ (xa)^R = ax^R & \text{Otherwise (i.e., if } w=xa) \end{cases}$$

Note: x^R in the definition indicates the reversed string of string x .

Now, let us see “What is the length of a string?”.

❖ **Definition:** The *length* of a string u is the number of symbols in u and is denoted by $|u|$ i.e., if

$$u = a_1 a_2 a_3 \dots a_n$$

then the length u is given by

$$|u| = n$$

The length of an empty string ϵ is 0 and is denoted by

$$|\epsilon| = 0$$

Note: $\epsilon w = w\epsilon = w$.

Now, let us see “What is the power of an alphabet?”.

❖ **Definition:** The power of an alphabet denoted by Σ^i is the set of words of length i . For example, if $\Sigma = \{0, 1\}$, then

$\Sigma^0 = \{\epsilon\}$ denote the set of words of length 0. Here, ϵ represents an empty string;

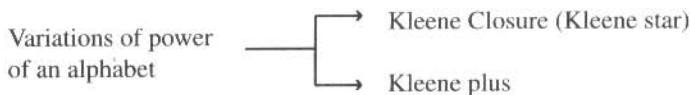
$\Sigma^1 = \{0, 1\}$ denote the set of words of length 1;

$\Sigma^2 = \{00, 01, 10, 11\}$ denote the set of words of length 2;

$\Sigma^3 = \{000, 001, 010, 100, 011, 101, 110, 111\}$ denote the set of words of length 3

and so on.

Now, let us see “What are the two variations of power of an alphabet?” The variations of power of an alphabet are shown below:



Now, let us see “What is Kleene closure(or Kleene star/star operator)? Give example”.

❖ **Definition:** The Kleene closure is defined as follows:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

which is the set of words of any length (possibly ϵ i.e., the null string). Each string is made up of symbols only from Σ .

■ **Example 1:** Let $\Sigma = \{0, 1\}$. Then Σ^* is obtained as shown below:

$$\Sigma^0 = \{\epsilon\} \quad \text{set of words of length 0}$$

$$\Sigma^1 = \{0, 1\} \quad \text{set of words of length 1}$$

$$\Sigma^2 = \{00, 01, 10, 11\} \quad \text{set of words of length 2}$$

$$\Sigma^3 = \{000, 001, 010, 100, 011, 101, 110, 111\} \quad \text{set of words of length 3}$$

.....

.....

$$\text{So, } \Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$$

$$\{0, 1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$$

which is the set of strings of 0's and 1's of any length.

■ **Example 2:** Kleene star can be applied to set of strings. $\{\text{“a”}, \text{“bc”}\}^*$ is set of strings. The set of strings consisting of a's and bc's of any length can be obtained as shown below:

$$\Sigma^0 = \{\epsilon\}$$

$$\Sigma^1 = \{\text{“a”}, \text{“bc”}\}$$

$\Sigma^2 = \{\text{"aa"}, \text{"abc"}, \text{"bcbc"}, \text{"bca"}\}$ i.e., string made up of a's and bc's.

.....

.....

So, $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$

$\Sigma^* = \{\epsilon, \text{"a"}, \text{"bc"}, \text{"aa"}, \text{"abc"}, \text{"bcbc"}, \text{"bca"}, \dots\}$

which is the set of strings of a's and bc's of any length.

Now, let us see "What is Kleene plus? Give example".

❖ **Definition:** The Kleene plus is a variation of Kleene star operator. The Kleene plus denoted by Σ^+ is defined as follows:

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$$

which is the set of words of any length except the null string i.e., ϵ (Σ^0) is not part of Σ^+ and hence $\epsilon \notin \Sigma^+$.

■ **Example 1:** Let $\Sigma = \{0, 1\}$. Then Σ^+ is shown below:

$\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, 001, \dots\}$ is the set of strings of 0's and 1's of any length except the null string.

Note: $\Sigma^* = \Sigma^+ + \epsilon$. This can be written as $\Sigma^+ = \Sigma^* - \epsilon$ (See the above two examples for clarity).

Now, let us see "What is the length of the string? Give the recursive definition". The length of a string can be defined recursively as follows:

❖ **Definition:** If a is the symbol and w is the string derived from the alphabet Σ , the length of a string can be defined as

$$|w| = \begin{cases} 0 & \text{if } w = \epsilon \\ 1 & \text{if } w = a \\ |ua| = |u| + 1 & \text{if } w = ua \end{cases}$$

for each $a \in \Sigma$ (i.e., a is a symbol in Σ) and each $u, w \in \Sigma^*$ (i.e., u and w are strings).

1.2.3. Language

Now, let us see "What is a language? Give example".

❖ **Definition:** A *language* can be defined as a set of strings obtained from Σ^* where Σ is set of alphabets of a particular language. In other words, a language is subset of Σ^* which is denoted by $L \subseteq \Sigma^*$. For example,

- A language of strings consisting of equal number of 0's and 1's can be represented as $\{\epsilon, 01, 10, 0011, 1010, 0101, 0011, \dots\}$
- The language of strings consisting of n number of 0's followed by n number of 1's can be represented using the set as shown below:
 $\{\epsilon, 01, 0011, 000111, \dots\}$
- A language containing empty string ϵ is denoted by $\{\epsilon\}$
- An empty language is denoted by \emptyset .

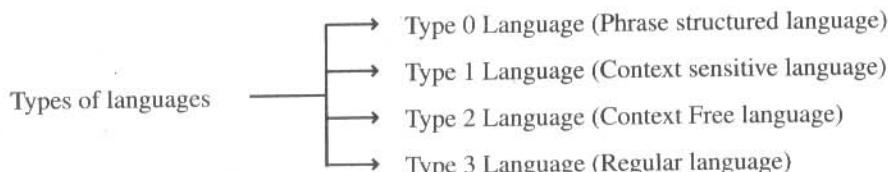
Now, let us see “What is a sentence? Give example”.

❖ **Definition:** A string that belongs to a language is called word or sentence of that language. For example, a language of strings consisting of equal number of 0's and 1's can be represented as shown below:

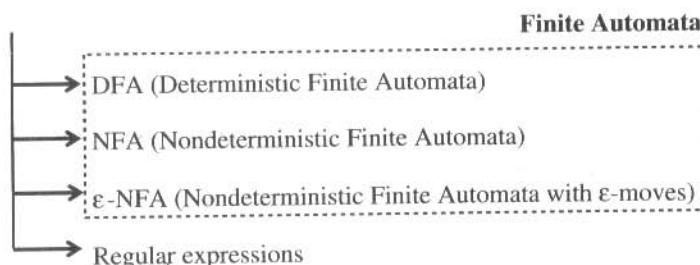
$$\{\epsilon, 01, 10, 0011, 1010, 0101, 0011, \dots\}$$

Each word separated by comma is a sentence (also called word). So, 01 is a sentence, 10 is a sentence, 0011 is a sentence and so on.

Now, let us see “What are the different types of languages?” The languages are classified as shown below:



In this chapter let us discuss about Regular languages and how they are accepted. Now, let us see “What are the different ways of describing the regular languages?” The regular languages can be described using four different methods:



Now, the question is “What are regular languages?” The regular languages are defined as the languages accepted by DFA’s, NFA’s and ϵ -NFA’s. They are also the languages defined by regular expressions.

In the subsequent sections, let us discuss how the regular languages are accepted by various types of finite automata and how to design various types of finite automata.

1.3. Finite Automata

The concept of an algorithm is one of the basic concepts in mathematics. We know that an algorithm is defined as unambiguous, step by step procedure (instructions) to solve a given problem in a finite number of steps by accepting a set of inputs and producing the desired output. The execution of an algorithm is carried out automatically by a computer (**Note:** Since a computer is a machine, we use the word machine in place of computer). In this subject, we use only abstract machines to execute our programs. So, first let us see “What is an abstract machine?”.

Note: In English, an abstract is a brief summary of a research article, thesis or a document. In mathematics, abstract is nothing but a theoretical concept without thinking of a specific example.

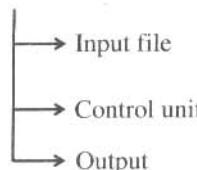
❖ **Definition:** An abstract machine (also called abstract computer) is a conceptual or theoretical model of a computer hardware or software system which really does not exist. These machines are not actual machines and hence, they are also called hypothetical computers. These machines have commonly encountered hardware features and concepts and avoids most of the details that are often found in real machines. The various types of abstract machines (or abstract computers) are:

- Finite automata
- Linear bounded automata
- Push down automata
- Turing machine

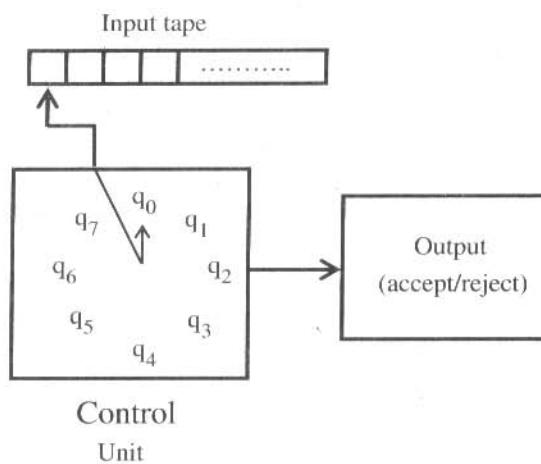
Now, let us “Define finite automata”.

❖ **Definition:** A finite automaton is a mathematical model which is used to study the *abstract machines* or *abstract computing devices* with the inputs chosen from Σ . Here, Σ stands for set alphabets using which any string can be obtained. On reading the string, the machine may accept the string or reject the string. Using this abstract model, the behavior of the actual system can

be understood and build to perform various activities. Finite automaton is an abstract model of a digital computer which has three components as shown below:



The pictorial representation (block diagram) of FA is shown below:

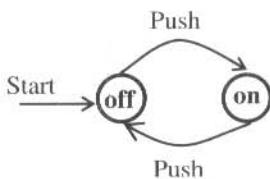


- **Input tape:** The input tape is divided into cells each of which can hold one symbol. The string to be processed is stored in these cells.
- **Control Unit:** The machine has some states one of which is the start state designated as q_0 and at least one final state. Apart from these, it has some finite states designated by q_1, q_2 and so on. Based on the current input symbol, the state of the machine can change.
- **Output:** Output may be *accept* or *reject*. When end of input is encountered, the control unit may be in *accept* or *reject* state.

Working: The finite automaton works as shown below:

- The machine is assumed to be in start state q_0 .
- The input pointer points to the first cell of the tape pointing to the string to be processed.
- After scanning the current input symbol, the machine can enter into any of the states q_0, q_1, q_2 and so on and the input pointer automatically points to the next character by moving one cell towards right.
- When the end of the string is encountered, the string is accepted if and only if the automaton will be in one of the final states. Otherwise, the string is rejected.

For example, consider an electric switch which has only two states “OFF” and “ON”. To start with, the switch will be in OFF state. When we push the button, it goes to ON state. If we push once again it goes to OFF state. This can be represented as shown below:



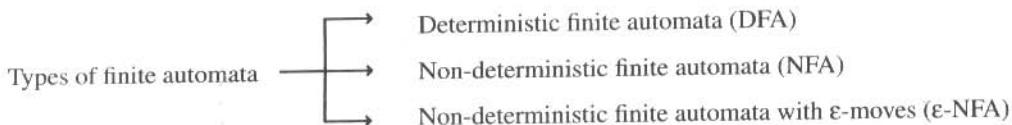
Note: The circles represent the states and the labels associated with arcs represent the input given to go to another state. The arrow with the label Start (not originating from any state) is considered as the start state.

Symbols Used in FA

Now, let us see “What are the various notations used during designing of FA?”.

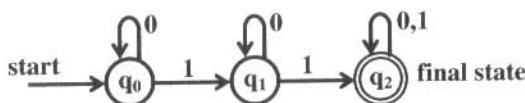
Symbol	Meaning
	A circle is used to represent a state. Here, q_0 is a state of the machine.
	A circle with an arrow which is not originating from any node represents the start state of machine.
	Two circles are used to represent a final state. Here, q_0 is the final state.
	An arrow with label 1 goes from state q_0 to state q_1 . This indicates there is a transition (change of state) from state q_0 on input symbol 1 to state q_1 . This is represented as: $\delta(q_0, 1) = q_1$
	An arrow with label 0 starts and ends in q_0 . This indicates the machine in state q_0 on reading a 0, remains in same state q_0 . This is represented as: $\delta(q_0, 0) = q_0$
	An arrow with label 0,1 goes from state q_0 to state q_1 . This indicates that the machine in state q_0 on reading a 0 or a 1 enters into state q_1 . This is represented as: $\delta(q_0, 0) = q_1$ $\delta(q_0, 1) = q_1$

Now, we can easily answer the question “How a FA processes the string?” The string w is processed by the various types of FA as shown below:



1.4. Deterministic Finite Automata (DFA)

Before worrying about the definition, let us consider the following pictorial representation of DFA.



From the above figure, observe following components of DFA:

- States:** The circles are called vertices or nodes or states. Each state is identified by a name i.e., q_0 , q_1 and q_2 . The states are classified and identified as shown below:

Start state: q_0 with an arrow labeled start is considered as start state.

Final state: q_2 with two concentric circles represent a final state.

Intermediate state: q_1 is neither a start state nor a final state. It is called an intermediate state.

This DFA has three states q_0 , q_1 and q_2 and can be represented as

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

Note: A DFA can have one or more final states. If there is no final state in DFA, the DFA accepts empty language.

- Input alphabets:** Each edge is labeled with 0 or 1 and represent the input alphabets which can be denoted as:

$$\Sigma = \{0, 1\}$$

- Transitions:** Transition is nothing but change of state after consuming an input symbol. If there is a change of state from q_i to q_j on an input symbol a , then we write

$$\delta(q_i, a) = q_j$$

For example, in the above diagram, the various transitions shown are represented as shown below:

$\delta(q_0, 0) = q_0$	There is a transition from state q_0 on 0 to state q_0
$\delta(q_0, 1) = q_1$	There is a transition from state q_0 on 1 to state q_1
$\delta(q_1, 0) = q_1$	There is a transition from state q_1 on 0 to state q_1
$\delta(q_1, 1) = q_2$	There is a transition from state q_1 on 1 to state q_2
$\delta(q_2, 0) = q_2$	There is a transition from state q_2 on 0 to state q_2
$\delta(q_2, 1) = q_2$	There is a transition from state q_2 on 2 to state q_2
\downarrow	
$\delta: (Q \times \Sigma) \rightarrow Q$	
\downarrow	
which is the cross product of $Q = \{q_0, q_1, q_2\}$ and $\Sigma = \{0, 1\}$	

Now, let us see “What is a transition function?” The transition function δ is defined as:

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

which is read as “ δ is a transition function which maps $Q \times \Sigma$ to Q ”. For example, the change of state from state q on input symbol a to state p is denoted by

$$\delta(q, a) = p$$

where

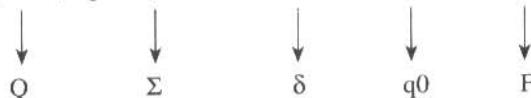
- δ : is a function called transition function.
- q : is the first parameter representing the current state of the machine.
- a : is the second parameter representing the current input symbol read.
- p : is the next state of machine which is returned by the transition function.

Note: In other words, the transition function δ accepts two parameters namely state q and input symbol a and returns a next state p :

- Start state (q_0): q_0 with the label start is treated as the start state.
- Final state (q_2): q_2 with two concentric circles is treated as the final state.

Note: From this discussion, it is observed that the DFA has five components:

(states, input alphabets, transitions, start state, final states)



With this concept, now let us see “What is a deterministic finite automaton (DFA)?”.

❖ **Definition:** The Deterministic Finite Automaton in short DFA is 5-tuple or quintuple (indicating five components):

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

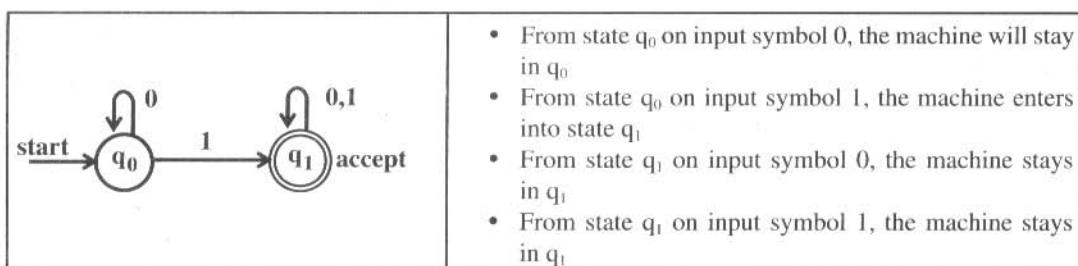
where

- M is the name of the machine. It can also be called by any name.
- Q is non-empty, finite set of states.
- Σ is non-empty, finite set of input alphabets.
- $\delta : Q \times \Sigma \rightarrow Q$ i.e., δ is transition function which is a mapping from $Q \times \Sigma$ to Q . Based on the current state and input symbol, the machine enters into another state.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is set of accepting or final states.

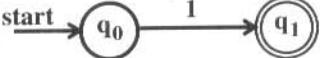
Thus, name DFA emerges from the following facts:

- D (Deterministic) – There is exactly one transition for every input symbol from the state. So, it is possible to determine exactly to which state the machine enters into after consuming the input symbol. So, the machine is deterministic.
- F (Finite) – Has finite number of states and arcs. So, it is deterministic and finite.
- A (Automaton) – Automaton is a machine which may accept the string or reject the string. So, it is deterministic finite automaton.

Note: The DFA can have only one transition from a state on an input symbol. Consider the following DFA and observe the various transitions:



Observe from the above diagram that, there is exactly one transition defined from a state on an input symbol. Sometimes, there can be no transitions from a state as shown below:

	<ul style="list-style-type: none"> From state q_0 on input symbol 1, the machine enters into state q_1 But, from state q_1, the transition is not defined on any of the input symbol. We say there is a zero transition from state q_1
--	--

Note: In short, in a DFA there can be zero or one transition from a state on an input symbol and at any point of time, the DFA will be in one state.

1.5. Simple Notations for DFA's

When we design a DFA, specifying a DFA as a 5-tuple such as $M = (Q, \Sigma, \delta, q_0, F)$ along with the detailed description of each transition is very tedious and difficult to understand. So, a DFA can be specified using simpler and more effective notations. Now, let us see "What are the various simpler notations used to specify a DFA?" The two notations using which the DFA's can be easily represented are:

- Transition diagram (Transition graph)
- Transition table

1.5.1. Transition Diagram

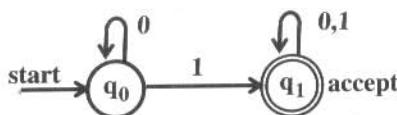
Definition: Now, let us see "What is a transition diagram?". The transition diagram for DFA

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

is defined as a graph with circles, arrows and arcs with labels, two circles etc. It is formally defined as shown below:

- Each state of Q corresponds to one node or vertex represented using a circle or two circles.
- Alphabets in Σ are represented as labels along with the edges.
- The transition from one state to another state is indicated by the directed edge. Let $\delta(q_i, a) = q_j$. This indicates that there is a directed edge from q_i to q_j and the edge is labeled a .
- The start state is a state which has an arrow not originating from any node and entering into the state. This is labeled with start.
- The final states or accepting states which are in F are represented by double circles. The states which are not in F are represented by a single circle.

For example, the following diagram represents a transition diagram of a DFA



- It has two nodes corresponding to the two states of machine q_0 and q_1 .
- q_0 has a start arrow (incoming arrow not originating from any node) and hence it is the start state.
- q_1 is the accepting state and hence it is denoted by two circles.
- There are 3 arcs: the first one with label 0 from q_0 to q_0 , the second with label 1 from q_0 to q_1 and third arc with labels 0, 1 both from q_1 to q_1 .

1.5.2. Transition Table

Definition: Now, let us see “What is a transition table?”. The transition table for DFA

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

is defined as a conventional, tabular representation of a transition function such as δ which takes two arguments and returns a value with:

- The rows of the table correspond to the states of DFA obtained from Q .
- The columns correspond to the input symbols obtained from Σ . Note: There is one row for each state, and one column for each input.
- If q is the current state of DFA and a is the current input symbol, the value returned from $\delta(q, a)$ represent the next state of DFA and is entered in row q and column a .
- The state marked with an arrow is the start state.
- The final state is marked with a star.

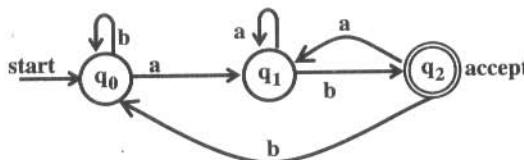
For example, the transition diagram and its equivalent transition table are shown below:

Transition diagram		Transition table									
Rows	Columns										
	δ	<table border="1" style="display: inline-table;"> <tr> <td></td><td style="text-align: center;">0</td><td style="text-align: center;">1</td></tr> <tr> <td style="text-align: center;">q_0</td><td style="text-align: center;">q_0</td><td style="text-align: center;">q_1</td></tr> <tr> <td style="text-align: center;">$*q_1$</td><td style="text-align: center;">q_1</td><td style="text-align: center;">q_1</td></tr> </table>		0	1	q_0	q_0	q_1	$*q_1$	q_1	q_1
	0	1									
q_0	q_0	q_1									
$*q_1$	q_1	q_1									
Transition diagram	Rows	Columns									
<ul style="list-style-type: none"> The transition diagram of DFA has two states q_0 and q_1 		<ul style="list-style-type: none"> Represented using two rows q_0 and q_1 									

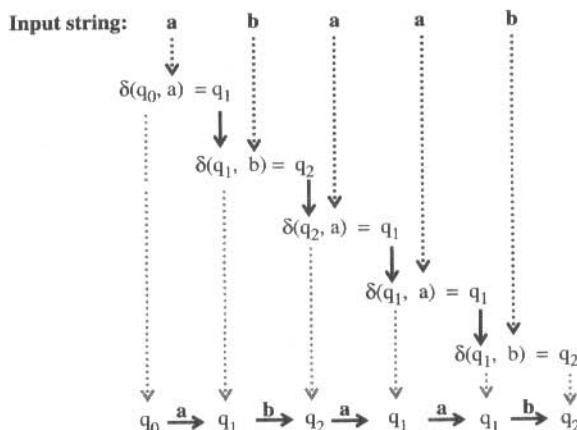
• There are two input symbols 0 and 1		• The two input symbols 0 and 1 correspond to two columns
• Start state is represented using an arrow mark not originating from any state and labeled start		• The start state is identified by putting an arrow with direction towards right
• The final states are represented by two circles.		• The final states are represented by putting stars (*)'s by the side of states
• The transition from state q on input symbol a to state p is represented by a directed edge originating from state q and ending at state p with label a.		• The equivalent transition is represented by writing p in row q and column a

1.5.3. How a DFA Processes Strings

Now, let us see “What are the moves made by the following DFA while processing the string abaab and abb?”.



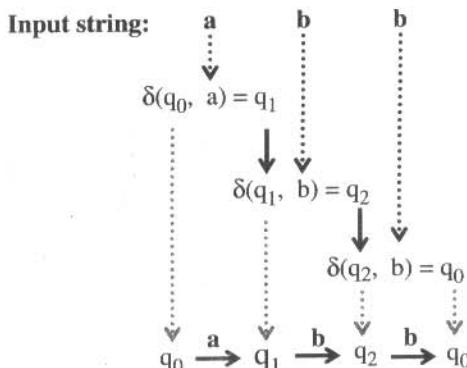
Solution: The moves made by the DFA for the input string abaab is shown below:



(States a DFA is in during the processing of abaab)

Note: At the end of the string abaab, the DFA will be in state q_2 which is the final state. So, the string abaab is accepted by the machine.

The moves made by the DFA for the input string abb is shown below:



(States a DFA is in during the processing of abaab)

Note: At the end of the string abb, the DFA will be in state q_0 which is not the final state. So, the string abb is rejected by the machine.

Now, we can easily answer the question “How a DFA processes the string?” The string w is processed by the DFA as shown below:

- **Initialization:** Let q_0 is the start of DFA and let w is the string to be processed. Initially, the input pointer points to the left most symbol in string w .
- **Processing:** The DFA reads one symbol from the input string at a time. The machine in state q_0 on reading the input symbol, consult the transition function δ . If there is a transition:

$$\delta(q_0, a_1) = q_1$$

then the machine changes its state to q_1 and the input pointer points to the next symbol. Now, the machine in state q_1 , after reading the next input symbol may change its state to q_2 and so on.

- **Accept or reject:** When the input points to the end of the string, if the DFA is in final state, the string w is accepted by DFA. After the end of the input, if the DFA is not in a final state, then the string w is rejected by the DFA.

1.5.4. Extended Transition Function of DFA to Strings

Note: The transition $\delta(q, a) = p$ accepts two parameters namely state q and input symbol a as the parameters and returns a state p which is the next state of the machine. But, if there is a change of state from state q to state p on input string w , then we use extended transition function denoted by δ^* .

Note: We can also use $\hat{\delta}$ in place of δ^* . But, in our book let us use δ^* to denote extended transition. Now, let us see “What is extended transition function δ^* ?”.

❖ **Definition:** The extended transition function δ^* describes what happens to a state of machine when the input is a string (sequence of symbols). Let $M = (Q, \Sigma, \delta, q_0, F)$ be a FA. The extended transition function $\delta^*: Q \times \Sigma^* \rightarrow Q$ is defined recursively as shown below:

- Basis: $\delta^*(q, \epsilon) = q$. This indicates that if the machine is in state q and read no input, then the machine is still in state q .
- Induction: Let $w = xa$ where a is the last symbol of w and x is the remaining string of w . Let q is the current state and w is the string to be processed and after consuming the string w , let the state of the machine is p . Then

$$\delta^*(q, w) = \delta^*(q, xa) = \delta(\delta^*(q, x), a) = p$$

Note: Thus, various properties of extended transition functions are:

- $\delta^*(q, \epsilon) = q$
- $\delta^*(q, w) = \delta^*(q, xa) = \delta(\delta^*(q, x), a)$ where $w = xa$
- $\delta^*(q, w) = \delta^*(q, ax) = \delta^*(\delta(q, a), x)$ where $w = ax$ (modification of above)

For example, change of state from state q on input string w to state p is denoted by:

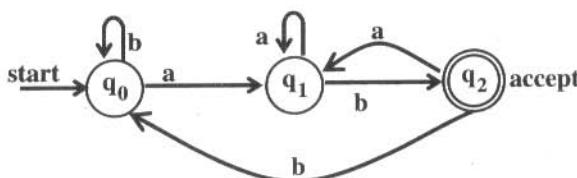
$$\delta^*(q, w) = p$$

where

- δ^* : is a function called extended transition function
- q : is the first parameter representing the current state of the machine
- w : is the second parameter representing the current input string being read
- p : is the next state of machine which is returned by the transition function

Note: The transition function δ^* accepts two parameters namely state q and input string w as the parameters and returns a state p which is the next state of the machine.

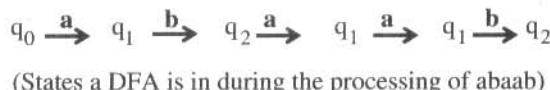
Now, let us see “What are the moves made by the following DFA while processing the string abaab using the extended transition function?”.



Solution: The moves made by the DFA for the input string abaab using δ^* is obtained starting from ϵ and taking prefix of abaab in increasing size as shown below:

- For prefix ϵ : $\delta^*(q_0, \epsilon) = q_0$ (1)
- For prefix a:
$$\begin{aligned} \delta^*(q_0, a) &= \delta(\delta^*(q_0, \epsilon), a) \\ &= \delta(q_0, a) \\ &= q_1 \end{aligned}$$
 Substituting (1) (2)
- For prefix ab:
$$\begin{aligned} \delta^*(q_0, ab) &= \delta(\delta^*(q_0, a), b) \\ &= \delta(q_1, b) \\ &= q_1 \end{aligned}$$
 Substituting (2) (3)
- For prefix aba:
$$\begin{aligned} \delta^*(q_0, aba) &= \delta(\delta^*(q_0, ab), a) \\ &= \delta(q_1, a) \\ &= q_1 \end{aligned}$$
 Substituting (3) (4)
- For prefix abaa:
$$\begin{aligned} \delta^*(q_0, abaa) &= \delta(\delta^*(q_0, aba), a) \\ &= \delta(q_1, a) \\ &= q_1 \end{aligned}$$
 Substituting (4) (5)
- For prefix abaab:
$$\begin{aligned} \delta^*(q_0, abaab) &= \delta(\delta^*(q_0, abaa), b) \\ &= \delta(q_1, b) \\ &= q_2 \end{aligned}$$
 Substituting (5)

It is observed from (1) to (5) that the sequence of states the DFA is in during the processing of string abaab is shown below:



Note: After the string abaab, the machine is in state q_2 which is the final state. So, the string abaab is accepted by DFA. Now, let us see “When a language is accepted by DFA?”.

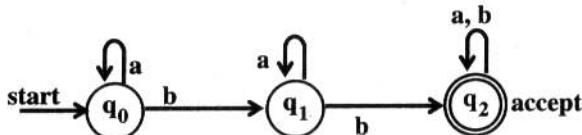
1.5.5. Language Accepted by a DFA

Now, let us “Define the language accepted by DFA” The language accepted by DFA is formally be defined as follows:

❖ **Definition:** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. A string w is accepted by the machine M , if it takes the machine from initial state q_0 to final state i.e., $\delta^*(q_0, w)$ is in F . Thus, the language accepted by DFA represented as $L(M)$ can be formally written as:

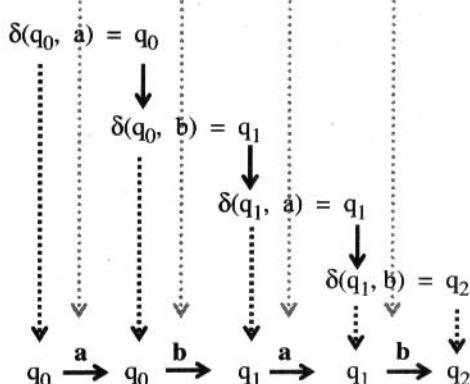
$$L(M) = \{w \mid w \in \Sigma^* \text{ and } \delta^*(q_0, w) \text{ is in } F\}$$

For example, consider the following DFA with q_0 as the start state and q_2 as the final state.



Input string: Assume **abab** is the input string. The moves made by the DFA is shown below:

Input string: a b a b



Note: At the end of the string abab, the DFA will be in state q_2 which is the final state. So, the string abab is accepted by the machine.

(States a DFA is in during the processing of abaab)

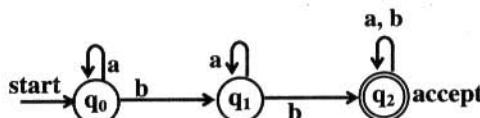
Now, let us see “What language is rejected by DFA?”.

- ❖ **Definition:** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The non-acceptance of a language indicates that after the string w is processed, the DFA will not be in the final state.

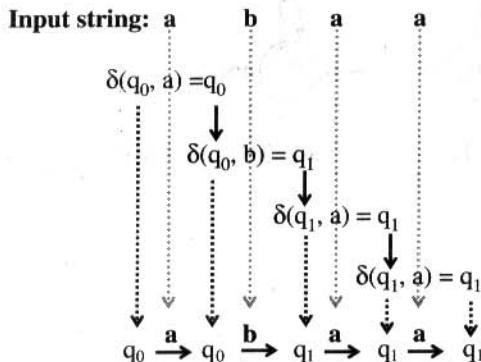
Thus, the non-acceptance of the string w by a FA or DFA can be defined as:

$$L(M) = \{w \mid w \in \Sigma^* \text{ and } \delta^*(q_0, w) \text{ is not in } F\}$$

For example, consider the following DFA with q_0 as the start state and q_2 as the final state:

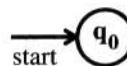


Input string: Assume **abaa** is the input string. The moves made by the DFA is shown below:



(States a DFA is in during the processing of abaa)

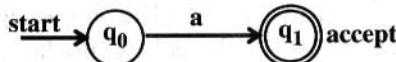
■ **Example 1:** DFA to accept empty language $L = \{ \phi \}$ is shown below:



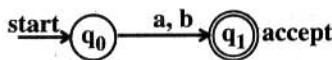
■ **Example 2:** DFA to accept an empty string $L = \{ \epsilon \}$ is shown below:



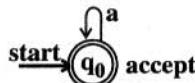
■ **Example 3:** DFA to accept exactly one a is shown below:



■ **Example 4:** DFA to accept one a or one b is shown below:



■ **Example 5:** DFA to accept zero or more a 's is shown below:



Now, let us see “How to construct the DFA for complex languages?” in the coming sections.

1.6. DFA Design Techniques

Now, let us see “What are the various problem types for which we can construct DFA?”. There are three types of problems for which we can construct a DFA:

- Pattern recognition problems
- Divisible by k problems
- Modulo-k-counter problems

1.6.1. Pattern Recognition Problems

For the type of problems that involve pattern recognition, the DFA can be constructed very easily. Now, let us see “What are the general steps to be followed while designing the DFA for pattern recognition problems?”. The various steps to be followed are:

Step 1: Identify the minimum string.

Step 2: Identify the alphabets.

Step 3: construct a skeleton DFA.

Step 4: Identify other transitions not defined in step 3 **Note:** This is difficult step. So, give more attention to this step while constructing DFA.

Step 5: Construct the DFA using transitions in step 3 and step 4.

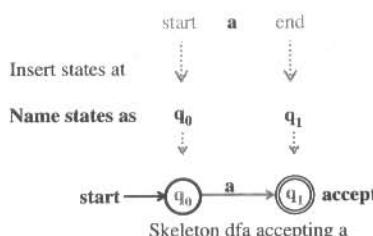
■ **Example 1:** Now, let us “Draw a DFA to accept string of a’s having at least one a ”.

Solution: The DFA can be constructed as shown below:

Step 1: Identifying the minimum string: In this case, it is a .

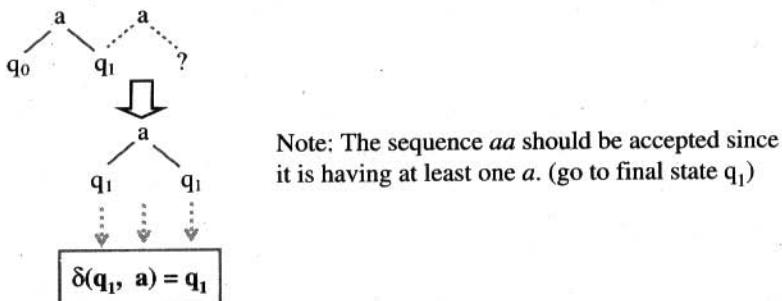
Step 2: Identify the alphabets: In this case $\Sigma = \{a\}$.

Step 3: Construct a skeleton DFA: The DFA for the string a identified in step 1 can be constructed as shown below:



Step 4: Identify the transitions not defined in step 3:

step (i) : $\delta(q_1, a) = ?$ Move from q_0 to q_1 on a (see skeleton DFA) and then think of transition from q_1 on a.

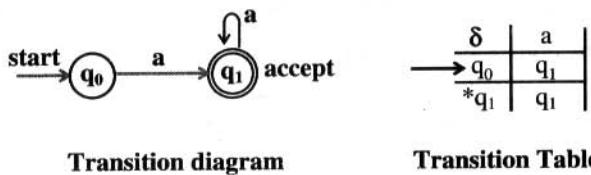


Step 5: Construct the DFA. The DFA can be obtained by skeleton DFA and transitions obtained from previous step. The DFA is defined as:

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

where

- $Q = \{q_0, q_1\}$
- $\Sigma = \{a\}$
- q_0 is the start state
- $F = \{q_1\}$
- δ is shown below using the transition diagram and table as shown below:



The language to be accepted by DFA can be written as shown below:

$$L = \{ a, aa, aaa, aaaa, \dots \}$$

or

$$L = \{ a^n \mid n \geq 1 \}$$

or

$$L = \{ w : n_a \geq 1, w \in \{a\}^* \}''$$

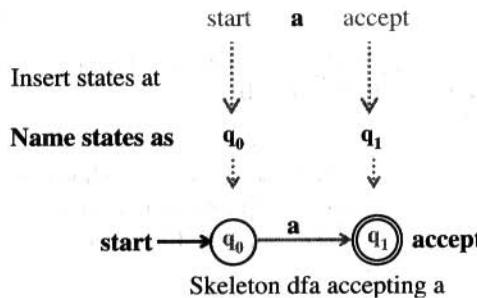
Example 2: Now, let us “Draw a DFA to accept strings of a’s and b’s having at least one a”.

Solution: The DFA can be constructed as shown below:

Step 1: Identifying the minimum string: In this case it is *a*.

Step 2: Identify the alphabets: In this case $\Sigma = \{a, b\}$.

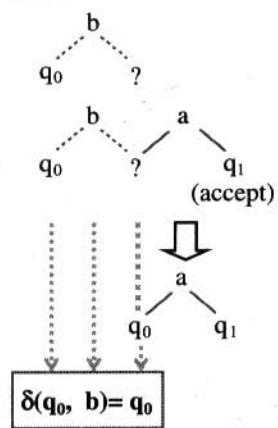
Step 3: Construct the skeleton DFA: The DFA for the string *a* identified in step 1 can be constructed as shown below:



Step 4: Identify the transitions not defined in step 3: We need to compute the following:

- | | |
|----------------------|-----------|
| $\delta(q_0, b) = ?$ | Step (i) |
| $\delta(q_1, a) = ?$ | Step(ii) |
| $\delta(q_1, b) = ?$ | Step(iii) |

Step (i) : $\delta(q_0, b) = ?$

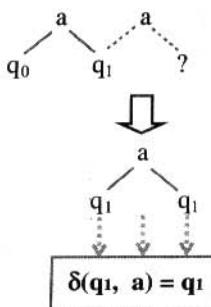


Note: The sequence *b* should not be accepted since it is not having at least one *a*.

Note: But, *b* followed by *a* resulting in string *ba* has to be accepted since it has at least one *a*. So, go to final state *q*1

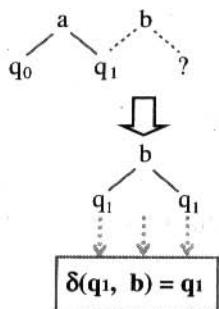
But, before the suffix *a*, the machine is in state *q*0 (see skeleton DFA)

Step(ii): $\delta(q_1, a) = ?$ Move from *q*0 to *q*1 after reading symbol *a* (see skeleton DFA) and then think of transition from *q*1 after reading symbol *a*.



Note: The sequence aa should be accepted since it is having at least one a . So, go to final state q_1 .

Step(iii): $\delta(q_1, b) = ?$ Move from q_0 to q_1 after reading symbol a (see skeleton DFA) and then think of transition from q_1 after reading symbol b .



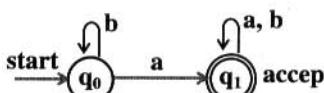
Note: The sequence ab should be accepted since it is having at least one a . So, go to final state q_1 .

Step 5: Construction of DFA. The DFA can be obtained by skeleton DFA and transitions obtained from previous step. The DFA is defined as:

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

where

- $Q = \{q_0, q_1\}$
- $\Sigma = \{a, b\}$
- q_0 is the start state
- $F = \{q_1\}$
- δ is shown below using the transition diagram and table as shown below:



δ	a	b
q_0	q_1	q_0
$*q_1$	q_1	q_1

$$L = \{w : n_a(w) \geq 1, w \in \{a, b\}^*\}$$

Note: Once the machine enters into state q_1 , irrespective of any number of inputs of a's and b's, the machine remains in same state q_1 and can not come out of the state. So, the machine is trapped in state q_1 and hence it is called trap state. Since, q_1 is also a final state, the state q_1 is called trapped final state.

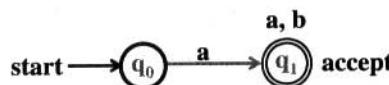
Example 3: Now, let us “Draw a DFA to accept strings of a's and b's having exactly one ‘a’”.

Solution: The DFA can be constructed as shown below:

Step 1: Identifying the minimum string: In this case it is a .

Step 2: Identify the alphabets: In this case $\Sigma = \{a, b\}$.

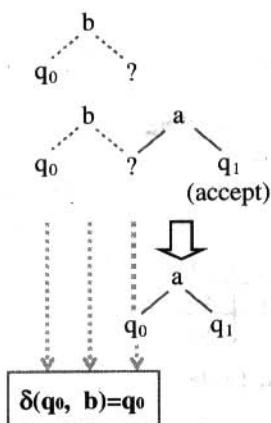
Step 3: Construct the skeleton DFA: The DFA for the string a identified in step 1 can be constructed as shown below:



Step 4: Identify the transitions not defined in step 3: We need to compute the following:

- | | |
|----------------------|---------------|
| $\delta(q_0, b) = ?$ | See step (i) |
| $\delta(q_1, a) = ?$ | See step(ii) |
| $\delta(q_1, b) = ?$ | See step(iii) |

Step (i) : $\delta(q_0, b) = ?$



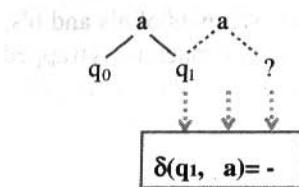
Note: The sequence b should not be accepted since it is not having one a .

Note: But, b followed by a resulting in string ba has to be accepted since it has one a . So, go to final state q_1 .

But, before the suffix a , the machine is in state q_0 (see skeleton DFA)

$$\delta(q_0, b) = q_0$$

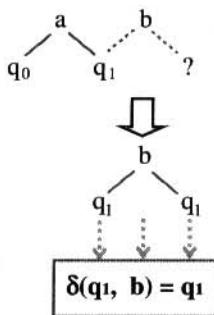
Step(ii): $\delta(q_1, a) = ?$ Move from q_0 to q_1 after reading symbol a (see skeleton DFA) and then think of transition from q_1 after reading symbol a .



Note: The sequence aa should not be accepted since it is not having exactly one a . So, the string aa should be rejected. For the reject state, the transition should not be defined.

Note: The symbol ‘-’ indicates that the transition is not defined.

Step(iii): $\delta(q_1, b) = ?$ Move from q_0 to q_1 after reading symbol a (see skeleton DFA) and then think of transition from q_1 after reading symbol b .

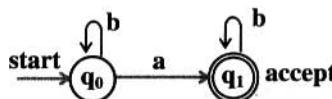


Note: The sequence ab should be accepted since it is having one a . So, go to final state q_1 .

Step 5: Construction of DFA. The DFA can be obtained by skeleton DFA and transitions obtained from previous step. The DFA is defined as:

$$M = (Q, \Sigma, \delta, q_0, F) \text{ where}$$

- $Q = \{q_0, q_1\}$
- $\Sigma = \{a, b\}$
- q_0 is the start state
- $F = \{q_1\}$
- δ is shown below using the transition diagram and table as shown below:

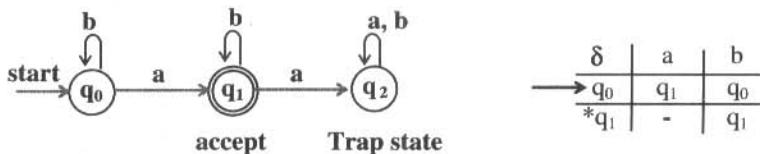


δ	a	b
q_0	q_1	q_0
$*q_1$	-	q_1

$$L = \{w : n_a(w) = 1, w \in \{a, b\}^*\}$$

Transition Table

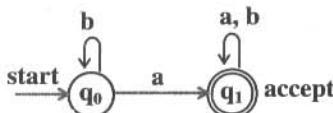
Note: In the above DFA, there is no transition from state q_1 for the input symbol a . But, we can include a transition from state q_1 for the input symbol a to a non final reject state as shown below:



Here, state q_2 is called a trap state. Now, the question is “What is a trap state?”.

❖ **Definition:** A state for which there exists transitions to itself for all the input symbols chosen from Σ is called a trap state.

For example, in the above DFA, once the machine changes its state to q_2 , for any of the input symbols a or b , there is no escape from this state. The machine is trapped in this state and hence the name trap state. A trap state can be non final state (state q_2 in above figure). The non final trap state is also called dead state. A final state can also be a trap state (state q_1 shown in transition diagram below). But, it is not a dead state since the string is accepted at this state.

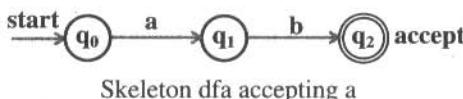


■ **Example 4:** Obtain a DFA to accept strings of a's and b's starting with the string ab .

Step 1: Identifying the minimum string: In this case it is ab .

Step 2: Identify the alphabets: In this case $\Sigma = \{a, b\}$.

Step 3: Construct the skeleton DFA: The DFA for the string ab identified in step 1 can be constructed as shown below:

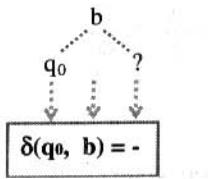


Step 4: Identify the transitions not defined in step 3: We need to compute the following:

$$\delta(q_0, b) = ? \quad \text{See step (i)} \quad \delta(q_2, a) = ? \quad \text{See step (iii)}$$

$$\delta(q_1, a) = ? \quad \text{See step (ii)} \quad \delta(q_2, b) = ? \quad \text{See step (iv)}$$

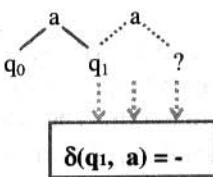
Step (i) : $\delta(q_0, b) = ?$



Note: The sequence b should be rejected since it is not starting with substring ab . For the reject state, the transition should not be defined.

Note: The symbol ‘-’ indicates that the transition is not defined.

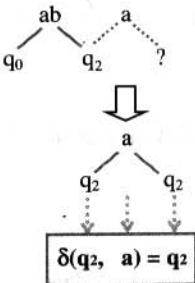
Step (ii): $\delta(q_1, a) = ?$ Move from q_0 to q_1 after reading symbol a (see skeleton DFA) and then think of transition from q_1 after reading symbol a .



Note: The sequence aa should not be accepted since it is not starting with ab . So, the string aa should be rejected. For the reject state, the transition should not be defined.

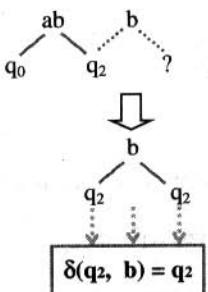
Note: The symbol ‘-’ indicates that the transition is not defined.

Step (iii): $\delta(q_2, a) = ?$ Move from q_0 to q_2 after reading the sequence ab (see skeleton DFA) and then think of transition from q_2 after reading symbol a .



Note: The sequence aba should be accepted since it is having substring ab . So, go to final state q_2 .

Step (iv): $\delta(q_2, b) = ?$ Move from q_0 to q_2 after reading the sequence ab (see skeleton DFA) and then think of transition from q_2 after reading symbol b .



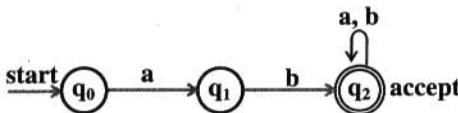
Note: The sequence abb should be accepted since it is having substring ab . So, go to final state q_2 .

Step 5: Construction of DFA. The DFA can be obtained by skeleton DFA and transitions obtained from previous step. The DFA is defined as:

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

where

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{a, b\}$
- q_0 is the start state
- $F = \{q_2\}$
- δ is shown below using the transition diagram and transition table as shown below:

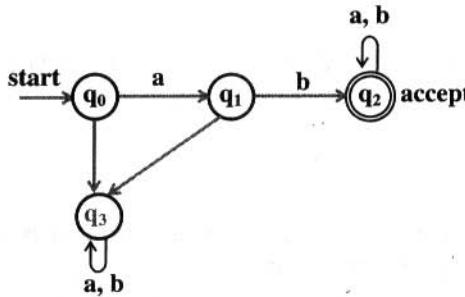


Transition diagram

δ	a	b
q_0	q_1	-
q_1	-	q_2
* q_2	q_2	q_2

Transition Table

Note: Since the transitions are not defined on q_0 and q_1 for the input symbol b and a respectively, we can have transitions to the trap state. So, the above DFA can also be written as shown below.

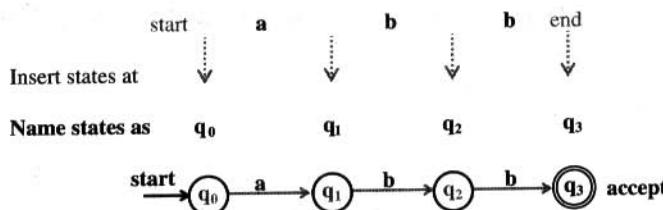


Example 5: Now, let us “Draw a DFA to accept string of a’s and b’s ending with the string abb”.

Step 1: Identify the minimum string: In this case abb.

Step 2: Identify the alphabets. In this case $\Sigma = \{a, b\}$.

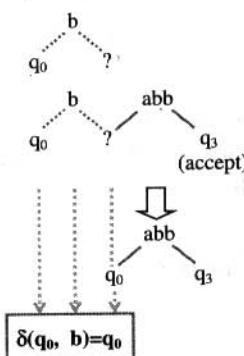
Step 3: construct a skeleton DFA: The DFA for the string abb identified in step 1 can constructed as shown below:



Step 4: Identify other transitions not defined in step 3: In this case, we need to compute the following:

- $\delta(q_0, b) = ?$ See step (i)
- $\delta(q_1, a) = ?$ See step (ii)
- $\delta(q_2, a) = ?$ See step (iii)
- $\delta(q_3, a) = ?$ See step (iv)
- $\delta(q_3, b) = ?$ See step (v)

Step (i) : $\delta(q_0, b)=?$



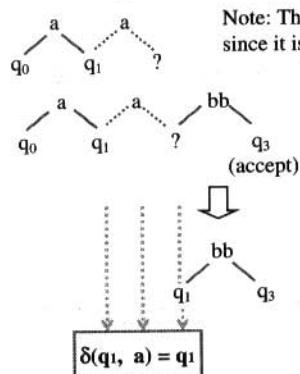
Note: The sequence *b* should not be accepted since it is not ending with *abb*.

Note: But, *b* followed by *abb* resulting in string *babb* has to be accepted since it ends with *abb*. So, go to final state q_3

But, before the suffix *abb*, the machine is in state q_0 (see skeleton DFA)

$$\boxed{\delta(q_0, b) = q_0}$$

Step (ii): $\delta(q_1, a) = ?$ Move from q_0 to q_1 after reading symbol *a* (see skeleton DFA) and then think of transition from q_1 after reading symbol *a*.



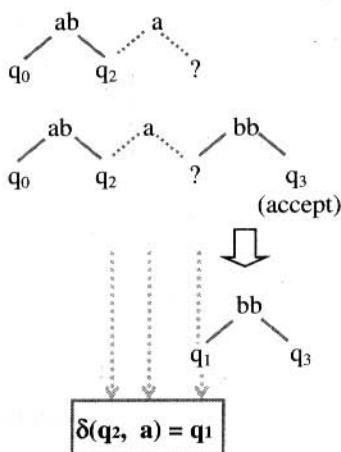
Note: The sequence *aa* should not be accepted since it is not ending with *abb*.

Note: But, *aa* followed by *bb* resulting in string *aabb* has to be accepted since it ends with *abb*. So, go to final state q_3

But, before the suffix *bb*, the machine is in state q_1 (see skeleton DFA)

$$\boxed{\delta(q_1, a) = q_1}$$

Step (iii): $\delta(q_2, a) = ?$ Move from q_0 to q_2 after reading sequence ab (see skeleton DFA) and then think of transition from q_2 after reading symbol a .

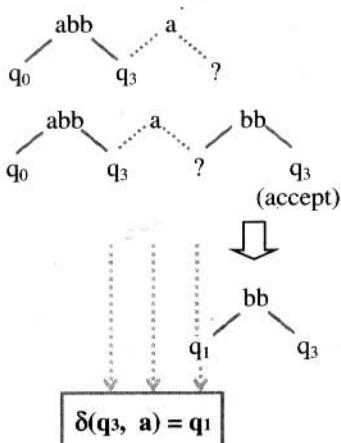


Note: The sequence aba should not be accepted since it is not ending with abb .

Note: But, aba followed by bb resulting in string $ababb$ has to be accepted since it ends with abb . So, go to final state q_3

But, before the suffix bb , the machine is in state q_1 (see skeleton DFA)

Step (iv): $\delta(q_3, a) = ?$ Move from q_0 to q_3 after reading sequence abb (see skeleton DFA) and then think of transition from q_3 after reading symbol a .

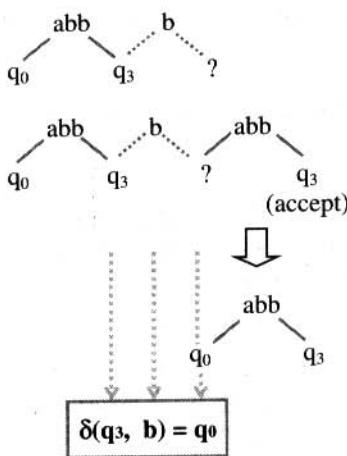


Note: The sequence $abba$ should not be accepted since it is not ending with abb .

Note: But, $abba$ followed by bb resulting in string $abbabb$ has to be accepted since it ends with abb . So, go to final state q_3

But, before the suffix bb , the machine is in state q_1 (see skeleton DFA)

Step (v): $\delta(q_3, b) = ?$ Move from q_0 to q_3 after reading sequence abb (see skeleton DFA) and then think of transition from q_3 after reading symbol b .



Note: The sequence *abbb* should not be accepted since it is not ending with *abb*.

Note: But, *abbb* followed by *abb* resulting in string *abbbabb* has to be accepted since it ends with *abb*. So, go to final state *q₃*

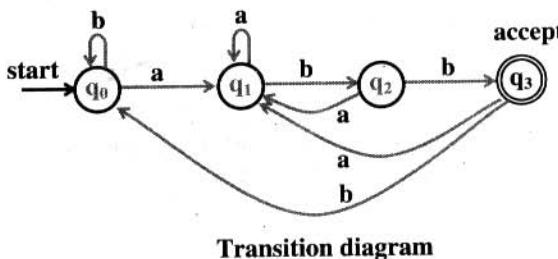
But, before the suffix *abb*, the machine is in state *q₀* (see skeleton DFA)

Step 5: Construct the DFA. The DFA to accept strings of a's and b's ending with *abb* is given by:

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

where

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b\}$
- q_0 = start state
- $F = \{q_3\}$
- δ is shown using the transition diagram/table as shown below:



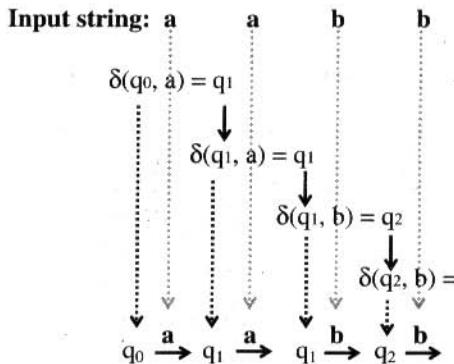
δ	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_3
$*q_3$	q_1	q_0

Transition table

Note: The language accepted by above DFA can be formally defined as:

$$L = \{ (a+b)^n abb : n \geq 0 \}$$

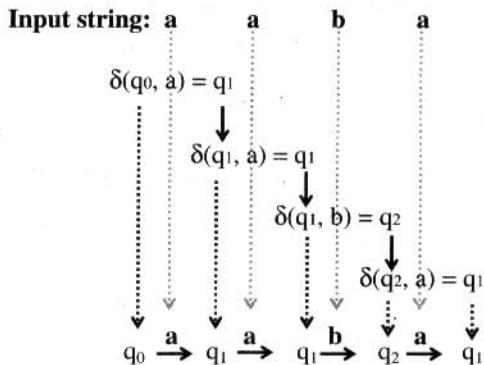
Now, let us “Show the sequence of moves made by the DFA for the string **aabb**”. The moves made by the DFA for the string **aabb** is shown below:



Note: At the end of the string aabb, the DFA will be in state q_3 which is the final state. So, the string aabb is accepted by the machine.

(States a DFA is in during the processing of aabb)

Now, let us “Show the sequence of moves made by the DFA for the string aaba”. The moves made by the DFA for the string **aaba** is shown below:



Note: At the end of the string aaba, the DFA will be in state q_1 which is not the final state. So, the string aaba is rejected by the machine.

(States a DFA is in during the processing of aaba)

■ **Example 6:** Now, let us “Draw a DFA to accept string of a’s and b’s which do not end with the string abb”.

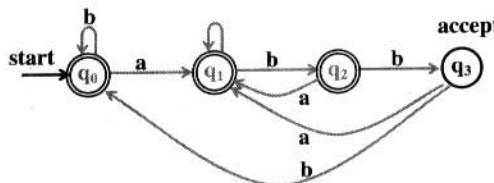
Solution: The design is exactly same as the previous problem. But, make final states as non final states and non final states as final states in the previous DFA. The resulting DFA is given by:

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

where

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b\}$

- q_0 = start state
- $F = \{q_0, q_1, q_2\}$
- δ is shown using the transition diagram/table as shown below:



Transition diagram

δ	a	b
* q_0	q_1	q_0
* q_1	q_1	q_2
* q_2	q_1	q_3
q_3	q_1	q_0

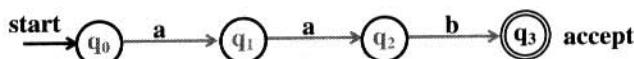
Transition table

Example 7: Now, let us “Draw a DFA to accept string of a’s and b’s having a substring aab”.

Step 1: Identify the minimum string: In this case aab.

Step 2: Identify the alphabets. In this case $\Sigma = \{a, b\}$.

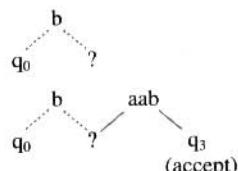
Step 3: Construct a skeleton DFA: The DFA for the string aab identified in step 1 can be constructed as shown below:



Step 4: Identify other transitions not defined in step 3. In this case, we need to compute the following:

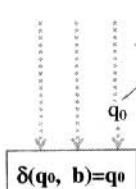
- $\delta(q_0, b) = ?$ See step (i)
 $\delta(q_1, b) = ?$ See step (ii)
 $\delta(q_2, a) = ?$ See step (iii)
 $\delta(q_3, a) = ?$ See step (iv)
 $\delta(q_3, b) = ?$ See step (v)

Step (i) : $\delta(q_0, b)=?$



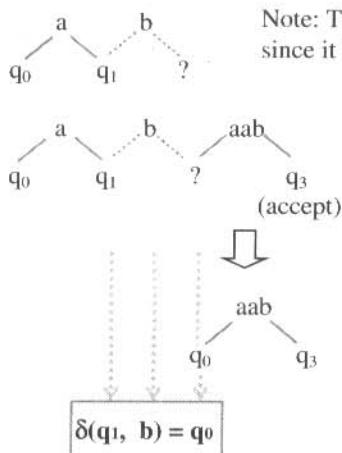
Note: The sequence b should not be accepted since it is not having the substring aab.

Note: But, b followed by aab resulting in string baab has to be accepted since it has the substring aab. So, go to final state q_3



But, before the suffix aab, the machine is in state q_0 (see skeleton DFA)

Step (ii): $\delta(q_1, b) = ?$ Move from q_0 to q_1 after reading symbol a (see skeleton DFA) and then think of transition from q_1 after reading symbol b .

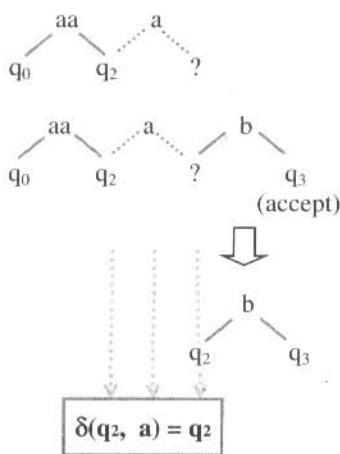


Note: The sequence ab should not be accepted since it is not having the substring aab .

Note: But, ab followed by aab resulting in string $abaab$ has to be accepted since it has the substring aab . So, go to final state q_3

But, before the suffix *aab*, the machine is in state q_0 (see skeleton DFA)

Step (iii): $\delta(q_2, a) = ?$ Move from q_0 to q_2 after reading sequence aa (see skeleton DFA) and then think of transition from q_2 after reading symbol a .

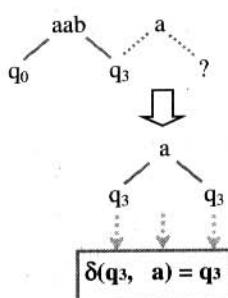


Note: The sequence *aaa* should not be accepted since it is not having the substring *aab*.

Note: But, aaa followed by b resulting in string $aaab$ has to be accepted since it has the substring aab . So, go to final state q_3

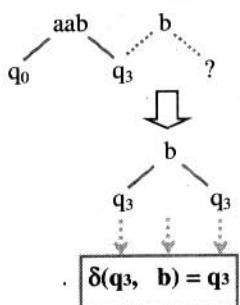
But, before the suffix b , the machine is in state q_2 (see skeleton DFA)

Step (iv): $\delta(q_3, a) = ?$ Move from q_0 to q_3 after reading sequence aab (see skeleton DFA) and then think of transition from q_3 after reading symbol a .



Note: The sequence $aaba$ should be accepted since it is having the substring aab . So, go to final state q_3

Step (v): $\delta(q_3, b) = ?$ Move from q_0 to q_3 after reading sequence aab (see skeleton DFA) and then think of transition from q_3 after reading symbol b .



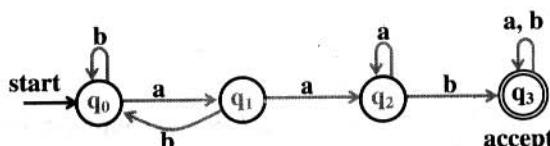
Note: The sequence $aabb$ should be accepted since it is having the substring aab . So, go to final state q_3

Step 5: Construct the DFA. The DFA to accept strings of a's and b's having a substring aab is given by:

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

where

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b\}$
- q_0 = start state
- $F = \{q_3\}$
- δ is shown using the transition diagram/table as shown below:



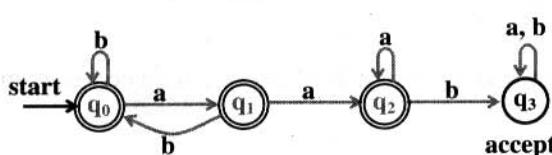
Transition diagram

δ	a	b
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_3
$*q_3$	q_3	q_3

Transition table

Example 8: Now, let us “Draw a DFA to accept string of a’s and b’s except those having the substring aab”.

Solution: The procedure remains same as the previous problem. But, the DFA can be obtained by making non final states as final states and final state as non final state in the previous problem. The DFA obtained after the necessary changes is shown below:



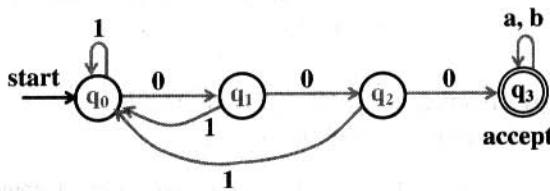
Transition diagram

δ	a	b
* q_0	q_1	q_0
* q_1	q_2	q_0
* q_2	q_2	q_3
q_3	q_3	q_3

Transition table

Example 9: Now, let us “Draw a DFA to accept string of 0’s and 1’s having three consecutive 0’s”.

Solution: The language can also be interpreted as strings of 0’s and 1’s having a substring 000. The procedure remains same as Example 7. But, start with the minimum string 000. The complete DFA is shown below:



Example 10: Now, let us “Draw a DFA to accept string of a’s and b’s such that

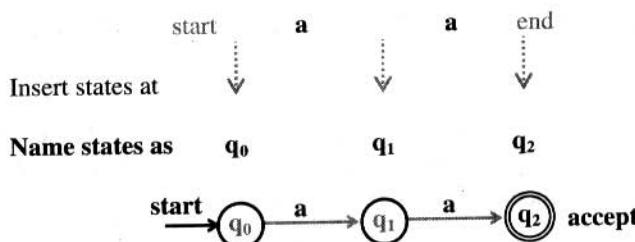
$$L = \{ awa \mid w \in (a+b)^n \text{ where } n \geq 0 \}$$

Solution: The language can also be interpreted as “Strings of a’s and b’s starting with one a and ending with one a with minimum string aa”.

Step 1: Identify the minimum string: In this case aa.

Step 2: Identify the alphabets. In this case $\Sigma = \{a, b\}$.

Step 3: Construct a skeleton DFA: The DFA for the string aa identified in step 1 can be constructed as shown below:



Step 4: Identify other transitions not defined in step 3. In this case, we need to compute the following:

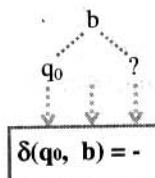
$$\delta(q_0, b) = ? \quad \text{See step (i)}$$

$$\delta(q_1, b) = ? \quad \text{See step (ii)}$$

$$\delta(q_2, a) = ? \quad \text{See step (iii)}$$

$$\delta(q_2, b) = ? \quad \text{See step (iv)}$$

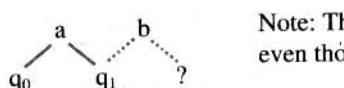
Step (i) : $\delta(q_0, b) = ?$



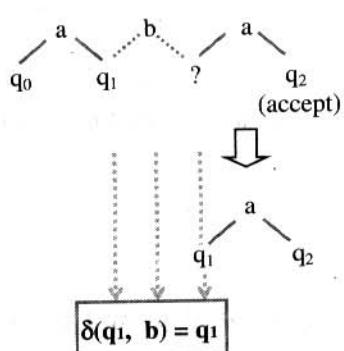
Note: The sequence b should be rejected since it is not starting with a and not ending with a . For the reject state, the transition should not be defined

Note: The symbol '-' indicates that the transition is not defined.

Step (ii): $\delta(q_1, b) = ?$ Move from q_0 to q_1 after reading symbol a (see skeleton DFA) and then think of transition from q_1 after reading symbol b .



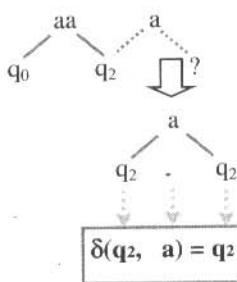
Note: The sequence ab should not be accepted. Because, even though it starts with a , it is not ending with a .



Note: But, ab followed by a resulting in string aba has to be accepted since it starts and ends with a . So, go to final state q_2

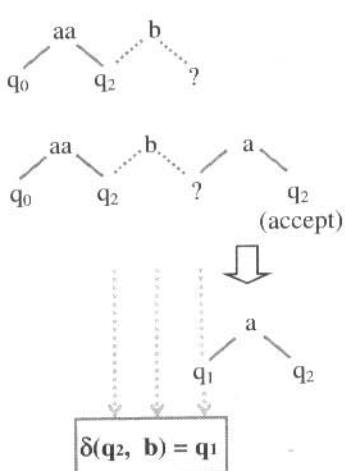
But, before the suffix a , the machine is in state q_1 (see skeleton DFA)

Step (iii): $\delta(q_2, a) = ?$ Move from q_0 to q_2 after reading sequence aa (see skeleton DFA) and then think of transition from q_2 after reading symbol a .



Note: The sequence aaa should be accepted since it starts and ends with a . So, go to final state q_1 .

Step (iv): $\delta(q_2, b) = ?$ Move from q_0 to q_2 after reading sequence aa (see skeleton DFA) and then think of transition from q_2 after reading symbol b .



Note: The sequence aab should not be accepted since it is not ending with a (even though it starts with a)

Note: But, aab followed by a resulting in string $aaba$ has to be accepted since it starts and ends with a . So, go to final state q_2

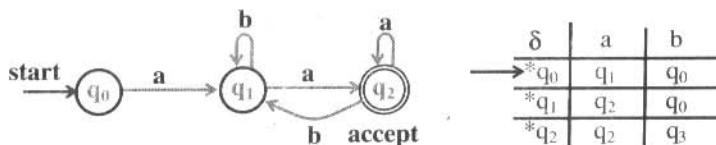
But, before the suffix a , the machine is in state q_1 (see skeleton DFA)

Step 5: Construct the DFA. The DFA to accept strings of a 's and b 's ending with abb is given by:

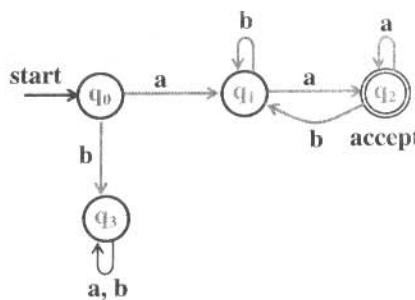
$$M = (Q, \Sigma, \delta, q_0, F)$$

where

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b\}$
- q_0 = start state
- $F = \{q_3\}$
- δ is shown using the transition diagram/table as shown below:



Note: Since the transition is not defined on q_0 for the input symbol b , we can have transition to the trap state thus rejecting the string totally. So, the above DFA can also be written as shown below.



Example 11: Now, let us “Draw a DFA to accept string of a’s and b’s ending with ab or ba .

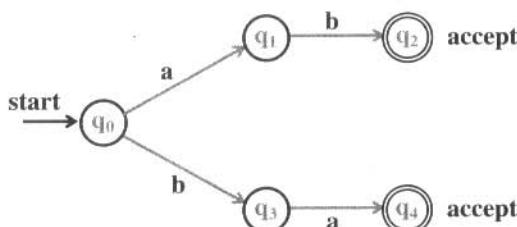
Solution: The given language can also be written as shown below:

$$L = \{w(ab + ba) \mid w \in \{a, b\}^*\}$$

Step 1: Identify the minimum string: In this case ab or ba .

Step 2: Identify the alphabets. In this case $\Sigma = \{a, b\}$.

Step 3: Construct a skeleton DFA: The DFA to accept the string ab or ba identified in step 1 can be constructed as shown below:

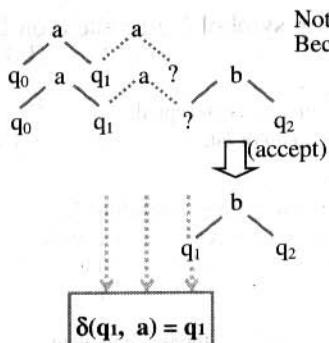


Step 4: Identify other transitions not defined in step 3: In this case, we need to compute the following:

$\delta(q_1, a) = ?$	See step (i)	$\delta(q_3, b) = ?$	See step (iv)
$\delta(q_2, a) = ?$	See step (ii)	$\delta(q_4, a) = ?$	See step (v)
$\delta(q_2, b) = ?$	See step (iii)	$\delta(q_4, b) = ?$	See step (vi)

Note: Whenever the string ends with *ab* let us go to state q_3 and whenever the string ends with *ba* let us go to state q_4 .

Step (i): $\delta(q_1, a) = ?$ Move from q_0 to q_1 after reading symbol *a* (see skeleton DFA) and then think of transition from q_1 after reading symbol *a*.

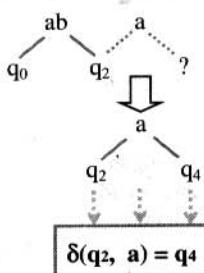


Note: The sequence *aa* should not be accepted. Because, it is not ending with *ab* or *ba*.

Note: But, *aa* followed by *b* resulting in string *aab* has to be accepted since it ends with *ab*. Because, it ends with *ab* let us go to final state q_2

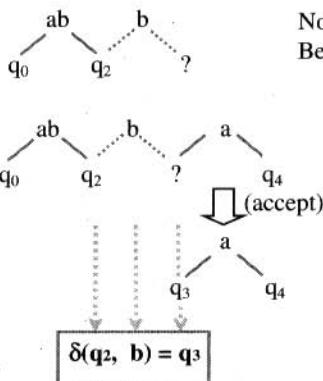
But, before the suffix *b*, the machine is in state q_1 (see skeleton DFA)

Step (ii): $\delta(q_2, a) = ?$ Move from q_0 to q_2 after reading string *ab* (see skeleton DFA) and then think of transition from q_2 after reading symbol *a*.



Note: The sequence *aba* should be accepted. Because, it is ending with *ba*. Since, the string ends with *ba*, the machine should go to q_4 .

Step (iii): $\delta(q_2, b) = ?$ Move from q_0 to q_2 after reading string *ab* (see skeleton DFA) and then think of transition from q_2 after reading symbol *b*.

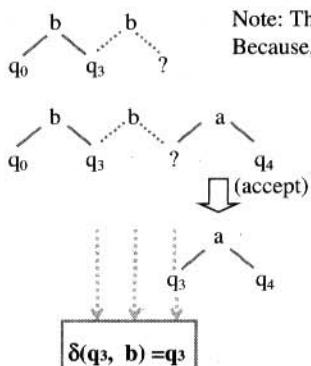


Note: The sequence *abb* should not be accepted. Because, it is not ending with *ab* or *ba*.

Note: But, *abb* followed by *a* resulting in string *abba* has to be accepted since it ends with *ba*. Because, it ends with *ba* let us go to final state q_4

But, before the suffix *a*, the machine is in state q_3 (see skeleton DFA)

Step (iv): $\delta(q_3, b) = ?$ Move from q_0 to q_3 after reading symbol *b* (see skeleton DFA) and then think of transition from q_3 after reading symbol *b*.

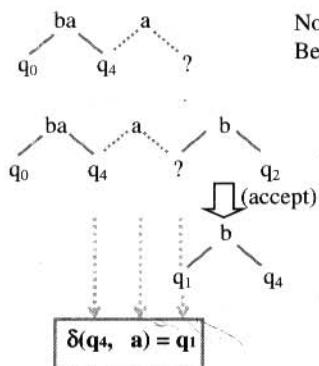


Note: The sequence *bb* should not be accepted. Because, it is not ending with *ab* or *ba*.

Note: But, *bb* followed by *a* resulting in string *bba* has to be accepted since it ends with *ba*. Because, it ends with *ba* let us go to final state q_4

But, before the suffix *a*, the machine is in state q_3 (see skeleton DFA)

Step (v): $\delta(q_4, a) = ?$ Move from q_0 to q_4 after reading string *ba* (see skeleton DFA) and then think of transition from q_4 after reading symbol *a*.

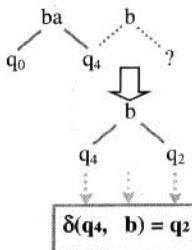


Note: The sequence *baa* should not be accepted. Because, it is not ending with *ab* or *ba*.

Note: But, *baa* followed by *b* resulting in string *baab* has to be accepted since it ends with *ab*. Because, it ends with *ab* let us go to final state q_2

But, before the suffix *b*, the machine is in state q_1 (see skeleton DFA)

Step (vi): $\delta(q_4, b) = ?$ Move from q_0 to q_4 after reading string ba (see skeleton DFA) and then think of transition from q_4 after reading symbol b .



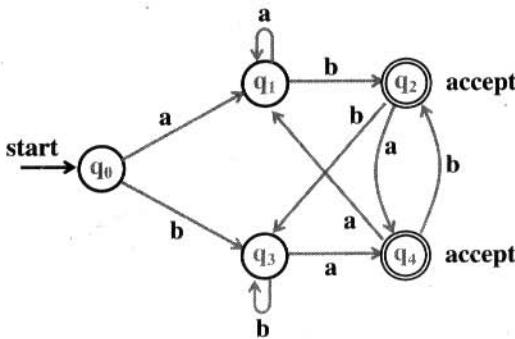
Note: The sequence bab should be accepted.
Because, it is ending with ab . Since, the string ends with ab , the machine should go to q_2 .

Step 5: Construct the DFA. The DFA to accept strings of a's and b's ending with ab or ba is given by:

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

where

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $\Sigma = \{a, b\}$
- q_0 = start state
- $F = \{q_2, q_4\}$
- δ is shown using the transition diagram/table as shown below:



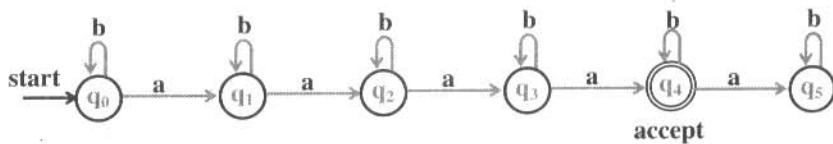
Transition diagram

δ	a	b
q_0	q_1	q_3
q_1	q_1	q_2
$*q_2$	q_4	q_3
q_3	q_4	q_3
$*q_4$	q_1	q_2

Transition table

Example 12: Now, let us “Obtain a DFA to accept strings of a's and b's having four a's where $\Sigma = \{a, b\}$.

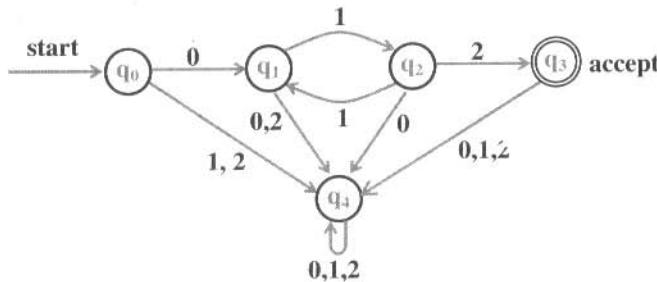
Solution: The DFA can be obtained using similar methods as explained earlier. The final DFA is shown below:



Note: A dead state is state where the FA remains in the same state for any input symbol. Here, q_5 is a dead state.

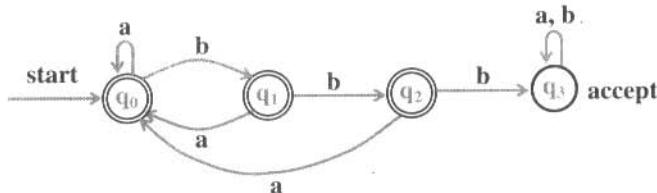
Example 13: Now, let us “Obtain a DFA to accept strings of 0’s, 1’s and 2’s beginning with a ‘0’ followed by odd number of 1’s and ending with a ‘2’”.

Solution: The machine to accept the corresponding string is shown below:



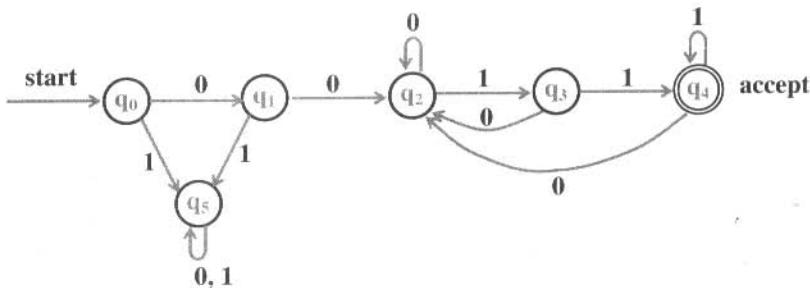
Example 14: Now, let us “Obtain a DFA to accept strings of a’s and b’s with at most two consecutive b’s”.

Solution: The machine to accept strings of a’s and b’s with at most two consecutive b’s is shown below:



Example 15: Now, let us “Obtain a DFA to accept strings of 0’s and 1’s starting with at least two 0’s and ending with at least two 1’s”.

Solution: The machine to accept the corresponding string is shown below:



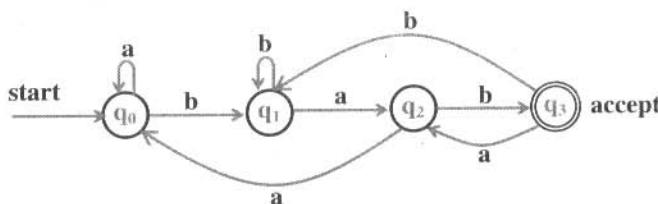
Note: In set notation, the language accepted DFA can be represented as

$$L = \{ w \mid w \in 00(0+1)^*11 \}$$

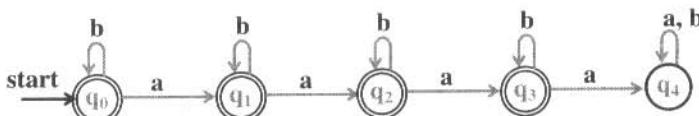
which is the language formed by words that begin with at least two 0's and ending with at least two 1's.

Example 16: Now, let us “Obtain a DFA to accept the language $L = \{ wbab \mid w \in \{a, b\}^*\}$ ”.

Solution: The DFA to accept the language $L = \{ wbab \mid w \in \{a, b\}^*\}$ is shown below:



Example 17: Now, let us “Draw a DFA to accept string of a's and b's having not more than three a's”.

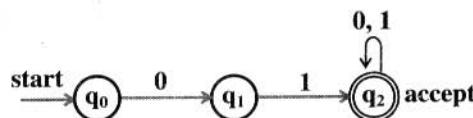


The language can also be represented as:

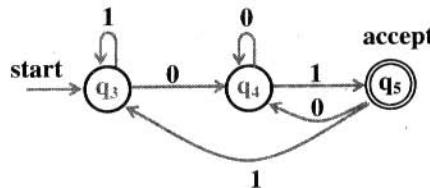
$$L = \{ w : n_a(w) \leq 3, w \in \{a, b\}^* \}$$

Example 18: Now, let us “Draw a DFA to accept set of all strings on the alphabet $\Sigma = \{0, 1\}$ that either begins or ends or both with the substring 01”.

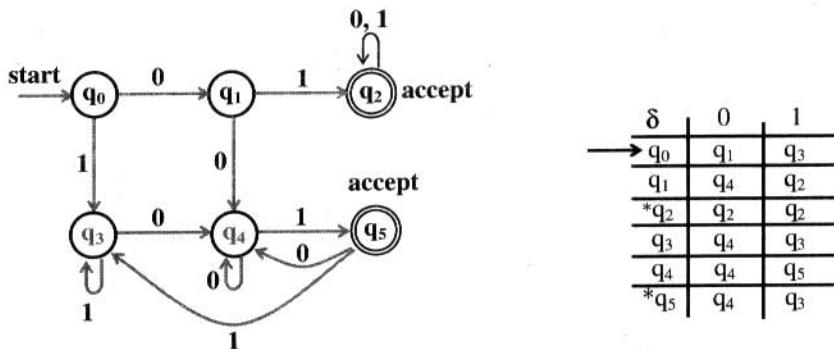
Solution: The DFA to accept strings of 0's and 1's starting with string 01 can be written as shown below (procedure remains same as Example 4):



The DFA to accept strings of 0's and 1's ending with string 01 can be written as shown below (procedure remains same as Example 5):



The two DFA can be joined to accept strings of 0's and 1's beginning with 01 and ending with 01 or both can be written as shown below:



So, formally a DFA can be defined as $M = (Q, \Sigma, \delta, q_0, F)$ where

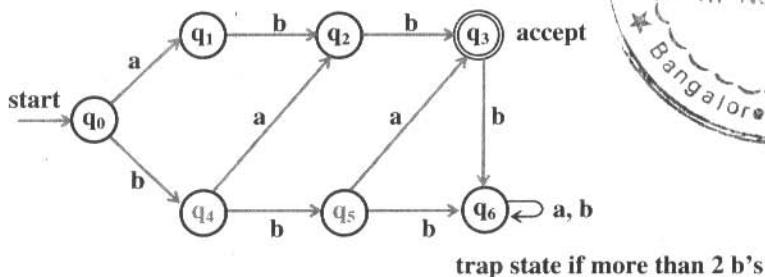
- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
- $\Sigma = \{0, 1\}$
- δ is shown above using the transition diagram and transition table
- q_0 is the start state
- $F = \{q_2, q_5\}$

Example 19: Now, let us “Draw a DFA to accept the language $L = \{w : n_a(w) \geq 1, n_b(w) = 2\}$.

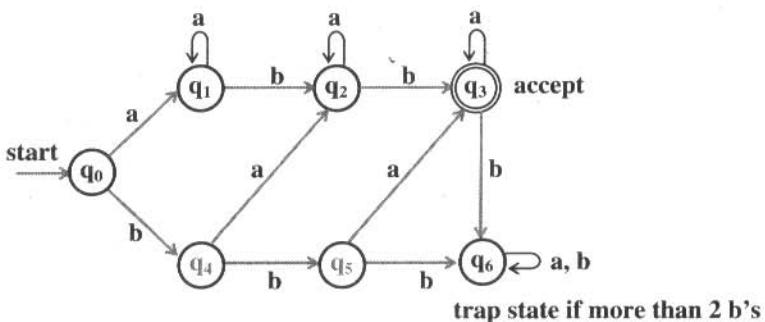
Solution: Note that the string should have exactly two b 's and at least one a . So, the minimum string that can be accepted by the DFA may be:

- abb
- bab
- bba

The skeleton DFA is shown below:



But, one or more a 's can be present in the string. So, the above DFA can be written (using the same steps of designing technique) as shown below:

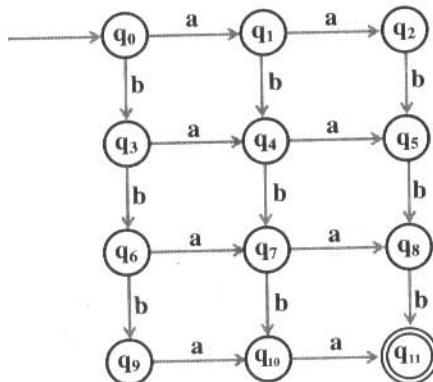


Example 20: Now, let us “Draw a DFA to accept the language $L = \{w : n_a(w) = 2, n_b(w) \geq 3\}$.

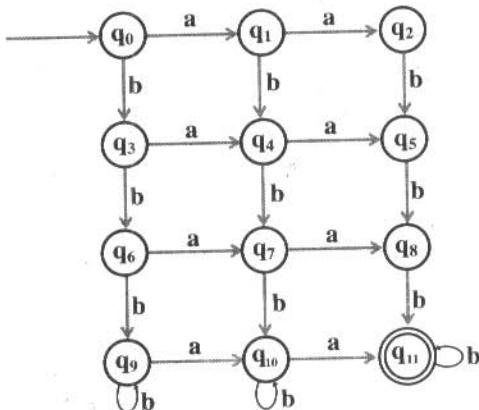
Solution: It is observed from the given language that the string that is accepted by DFA should have exactly two a 's and at least three b 's. So, the minimum strings that can be accepted are:

- aabbba
- baabb
- bbaab
- bbbaa
- ababb
- babab
- bbaba
- abbab
- babba
- abbbba

The skeleton DFA that accepts all the above strings is shown below:



Since, minimum three b's should be there, after three b's we can consume any number of b's.
So, the above DFA can be written as shown below:



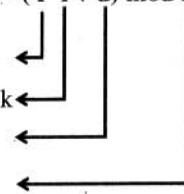
Note: From states q2, q5 and q8 do not define the transitions

1.6.2. Divisible by k Problems

In this section, let us discuss a method of constructing DFA that divide a number by k . For these types of problems, the transitions can be obtained using the following relation:

$$\delta(q_i, a) = q_j \text{ where } j = (r * i + d) \bmod k$$

- r is the radix of input. For binary $r = 2$
- i is the remainder obtained after dividing by k
- d represent digits. For binary $d = \{0, 1\}$
- k is divisor



The steps to be followed to find FA using these types of problems is shown below:

Step 1: Identify the radix, input alphabets and the divisor k .

Step 2: Compute the possible remainders: These remainders represent the states of DFA.

Step 3: Find the transitions using $\delta(q_i, a) = q_j$ where $j = (r * i + d) \bmod k$.

Step 4: Construct the DFA using the transitions obtained in step 3.

I Example 1: Now, let us “Construct a DFA which accepts strings of 0’s and 1’s where the value of each string is represented as a binary number. Only the strings representing zero modulo five should be accepted. For example, 0000, 0101, 1010, 1111, etc. should be accepted”.

Solution: The DFA can be obtained as shown below:

Step 1: Identify the radix, input alphabets and the divisor k : In this case, $r = 2$:

$$d = \{0, 1\}, k = 5$$

Step 2: Compute the possible remainders: After dividing by k , the possible remainders are:

$$i = 0, 1, 2, 3, 4$$

Step 3: Compute transitions: The transitions can be computed using the following relation:

$$\delta(q_i, a) = q_j \text{ where } j = (r * i + d) \bmod k$$

$$\text{with } r = 2 \quad \text{and} \quad k = 5$$

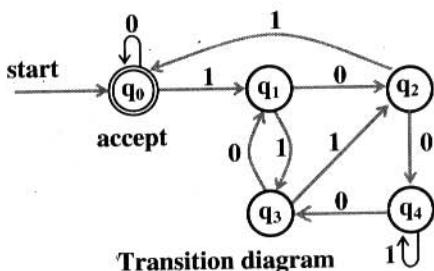
$$\text{So, } j = (2i + d) \bmod 5$$

remainder	d	$(2^* i + d) \text{ mod } 5 = j$	$\delta(q_i, d) = q_j$
$i = 0$	0	$(2^* 0 + 0) \text{ mod } 5 = 0$	$\delta(q_0, 0) = q_0$
	1	$(2^* 0 + 1) \text{ mod } 5 = 1$	$\delta(q_0, 1) = q_1$
$i = 1$	0	$(2^* 1 + 0) \text{ mod } 5 = 2$	$\delta(q_1, 0) = q_2$
	1	$(2^* 1 + 1) \text{ mod } 5 = 3$	$\delta(q_1, 1) = q_3$
$i = 2$	0	$(2^* 2 + 0) \text{ mod } 5 = 4$	$\delta(q_2, 0) = q_4$
	1	$(2^* 2 + 1) \text{ mod } 5 = 0$	$\delta(q_2, 1) = q_0$
$i = 3$	0	$(2^* 3 + 0) \text{ mod } 5 = 1$	$\delta(q_3, 0) = q_1$
	1	$(2^* 3 + 1) \text{ mod } 5 = 2$	$\delta(q_3, 1) = q_2$
$i = 4$	0	$(2^* 4 + 0) \text{ mod } 5 = 3$	$\delta(q_4, 0) = q_3$
	1	$(2^* 4 + 1) \text{ mod } 5 = 4$	$\delta(q_4, 1) = q_4$

Transitions of resulting DFA

Step 4: The DFA can be defined as $M = (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- q_0 is the start state
- $F = \{q_0\}$
- δ is shown below using the transition diagram and transition table as shown below:



δ	0	1
q_0	q_0	q_1
q_1	q_2	q_3
q_2	q_4	q_0
q_3	q_1	q_2
q_4	q_3	q_4

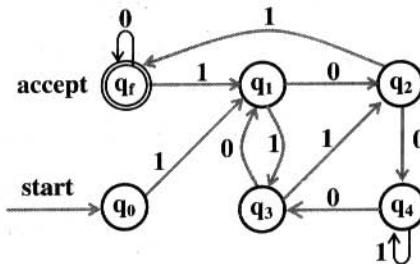
Transition Table

Note: Observe that for modulo 5, number of states of DFA will be 5.

In general, for modulo k, number of states of DFA will be k.

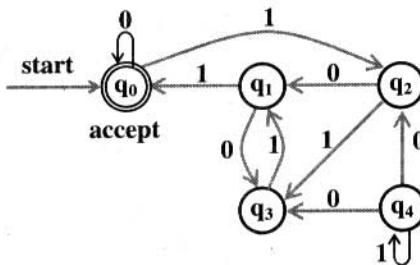
Example 2: Now, let us “obtain a DFA which accepts the set of all strings beginning with a 1 that when interpreted as a binary integer, is a multiple of 5. For example, 101, 1010, 1111 etc are multiples of 5. Note that 0101 is not beginning with 1 and it should not be accepted”.

Solution: The solution to this problem is same as above problem. But, the number always should start with a 1. If a binary number starts with a 0, the number should never be accepted. So, let us rename the final state as q_f and have the new start state q_0 and from this symbol on 1, enter into state q_1 . The resulting DFA is shown using the transition diagram as shown below:



Example 3: Now, let us “obtain a DFA that accepts set of all strings that, when interpreted in reverse as a binary integer, is divisible by 5. Examples of strings in the language are 0, 10011, 1001100 and 0101”.

Solution: The solution remains same as Example 1. But, reverse the direction of all arrow marks (except the arrow labeled with start). The resulting DFA is shown below:



Example 4: Now, let us “Draw a DFA to accept decimal strings divisible by 3”.

Solution: The DFA can be obtained as shown below:

Step 1: Identify the radix, input alphabets and the divisor k: In this case, $r = 10$:

$$d = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, k = 3$$

Step 2: Compute the possible remainders: After dividing any decimal number by 3, results in following remainders:

$i = 0, 1, 2$ which implies q_0, q_1 and q_2 are the states of DFA

Step 3: Compute transitions: The transitions can be computed using the following relation:
 $\delta(q_i, a) = q_j$ where $j = (r * i + d) \bmod k$

with $r = 10$ and $k = 3$

$$\text{So, } j = (2i + d) \bmod 3$$

Note: For the sake of convenience, let us group the digits from 0 to 9 based on the remainders we get after dividing by 3 as shown below:

- $\{0, 3, 6, 9\}$ with 0 as the remainder. So, δ from $\{0, 3, 6, 9\} \Rightarrow \delta$ from $\{0\}$
- $\{1, 4, 7\}$ with 1 as the remainder. So, δ from $\{1, 4, 7\} \Rightarrow \delta$ from $\{1\}$
- $\{2, 5, 8\}$ with 2 as the remainder. So, δ from $\{2, 5, 8\} \Rightarrow \delta$ from $\{2\}$

remainder	d	$(2 * i + d) \bmod 3 = j$	$\delta(q_i, d) = q_j$
-----------	---	---------------------------	------------------------

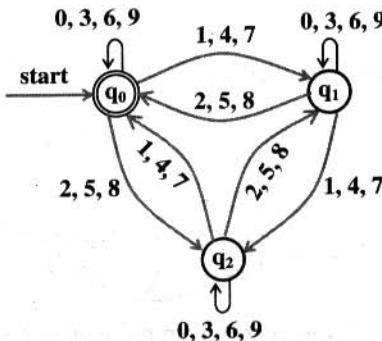
$i = 0$	0	$(10 * 0 + 0) \bmod 3 = 0$	$\delta(q_0, 0) = q_0$	$\Rightarrow \delta(q_0, \{0, 3, 6, 9\}) = q_0$
	1	$(10 * 0 + 1) \bmod 3 = 1$	$\delta(q_0, 1) = q_1$	$\Rightarrow \delta(q_0, \{1, 4, 7\}) = q_1$
	2	$(10 * 0 + 2) \bmod 3 = 2$	$\delta(q_0, 2) = q_2$	$\Rightarrow \delta(q_0, \{2, 5, 8\}) = q_2$
$i = 1$	0	$(10 * 1 + 0) \bmod 3 = 1$	$\delta(q_1, 0) = q_1$	$\Rightarrow \delta(q_1, \{0, 3, 6, 9\}) = q_1$
	1	$(10 * 1 + 1) \bmod 3 = 2$	$\delta(q_1, 1) = q_2$	$\Rightarrow \delta(q_1, \{1, 4, 7\}) = q_2$
	2	$(10 * 1 + 2) \bmod 3 = 0$	$\delta(q_1, 2) = q_0$	$\Rightarrow \delta(q_1, \{2, 5, 8\}) = q_0$
$i = 2$	0	$(10 * 2 + 0) \bmod 3 = 2$	$\delta(q_2, 0) = q_2$	$\Rightarrow \delta(q_2, \{0, 3, 6, 9\}) = q_2$
	1	$(10 * 2 + 1) \bmod 3 = 0$	$\delta(q_2, 1) = q_0$	$\Rightarrow \delta(q_2, \{1, 4, 7\}) = q_0$
	2	$(10 * 2 + 2) \bmod 3 = 1$	$\delta(q_2, 2) = q_1$	$\Rightarrow \delta(q_2, \{2, 5, 8\}) = q_1$

$\underbrace{\hspace{10em}}$ Transitions of DFA

Step 4: The DFA can be defined as $M = (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- q_0 is the start state

- $F = \{q_0\}$
- δ is shown below using the transition diagram and transition table as shown below:



1.6.3. Modulo k Counter Problems

Example 1: Let us “Obtain a DFA to accept strings of even number of a’s”. The DFA can be constructed by following the steps one by one as shown below:

Solution: It is given that $L = \{w : w \text{ has even number of a's}\}$.

Identify the number of states: The string w may have even number of a’s or odd number of a’s and results in following two cases (two states):

- **Case 0:** Strings that accepts Even number of a’s is denoted by:

E

- **Case 1:** Strings that accepts Odd number of a’s is denoted by:

O

Identify the start state and final state: Before reading any of the input symbols, number of a’s will be zero which represent Even a’s. So, the state E with even number of a’s is the start state.

Since, it is required to accept Even number of a’s the state E which accepts even number of a’s is the final state.

Design: Once the start state and final states are identified the transitions can be easily obtained as shown below:

- From a state E, on reading a results in odd number of a’s and hence change to odd state O. The transition is:

$$\delta(E, a) = O$$

- From a state O, on reading a results in even number of a 's and hence change to even state E.
The transition is:

$$\delta(O, a) = E$$

So, the DFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as:

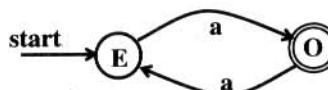
- $Q = \{E, O\}$
- $\Sigma = \{a\}$
- δ = shown in transition diagram



DFA to accept even number of a's

- $q_0 = E$ is the start state
- $F = \{E\}$

Note: The DFA to obtain odd number of a's can be obtained by making O state (odd state) as the final state and E state as the start state as shown below:



DFA to accept odd number of a's

Example 2: Let us “Obtain a DFA to accept the language $L = \{ w : |w| \bmod 3 = 0 \}$ where $\Sigma = \{a\}$ ”.

Solution: It is given that $L = \{ w : |w| \bmod 3 = 0 \}$ where $\Sigma = \{a\}$ } which indicates that the language consists of strings of multiples of 3 a's.

Identify the number of states: The language can be interpreted as strings of a's such that the number of a's in string is divisible by 3. Note that $|w| \bmod 3$ results in three cases:

- Case 0:** Results in remainder 0: The state is identified as q_0 .



- Case 1:** Results in remainder 1. The state is identified as q_1 .



- Case 2:** Results in remainder 2. The state is identified as q_2 .



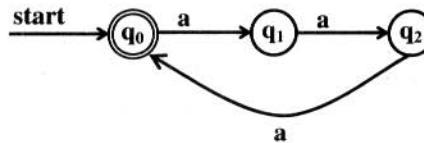
Identify the start state and final state: Before reading any inputs, number of a's will be zero. So, $0 \bmod 3 = 0$ which results in case 0. Hence, state q_0 is start state. Since, it is required to accept string of a's such that $|w| \bmod 3 = 0$, which results in case 0, q_0 is considered as final state.

Design: Once the start state and final states are identified the transitions can be easily obtained as shown below:

- From a state with remainder 0 (case 0) denoted by q_0 , on reading a results in remainder 1 (case 1) denoted by q_1 . So, $\delta(q_0, a) = q_1$
- From a state with remainder 1 (case 1) denoted by q_1 , on reading a results in remainder 2 (case 2) denoted by q_2 . So, $\delta(q_1, a) = q_2$
- From a state with remainder 2 (case 2) denoted by q_2 , on reading a results in remainder 0 (case 0) denoted by q_0 . So, $\delta(q_2, a) = q_0$

So, the DFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as:

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{a\}$
- δ : the transitions are shown using transition diagram as shown below:



- q_0 = start state
- $F = \{q_0\}$

The given language accepted by above DFA can also be represented as:

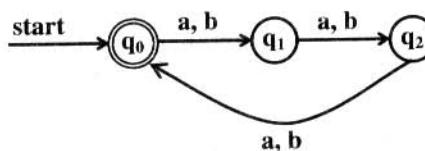
- $L = \{w : |w| \bmod 3 = 0\} \text{ where } \Sigma = \{a\}\}$
or
- $L = \{w : n_a(w) \text{ are divisible by 3 where } \Sigma = \{a\}\}$
or
- $L = \{a^{3n} : n \geq 0\}$

Example 3: Let us “Obtain a DFA to accept the language $L = \{ w : |w| \bmod 3 = 0\}$ on $\Sigma = \{a, b\}$ ”.

Solution: The language consists of strings of a's and b's whose length is a multiple of 3 and can be represented as:

$$L = \{\epsilon, aaa, aab, aba, abb, baa, bab, bba, bbb, \dots\}$$

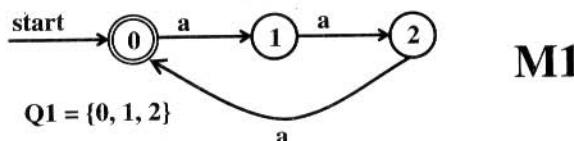
The design is similar to that of the previous problem. But, add one extra label b for each edge as shown below:



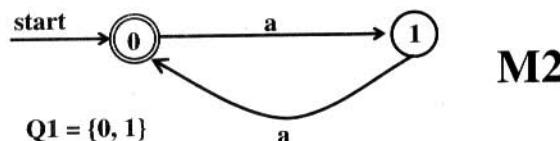
Example 4: Now, let us “Obtain a DFA to accept the following language $L = \{w \text{ such that}$

- a) $|w| \bmod 3 \geq |w| \bmod 2$ where $w \in \Sigma^*$ and $\Sigma = \{a\}$
- b) $|w| \bmod 3 \neq |w| \bmod 2$ where $w \in \Sigma^*$ and $\Sigma = \{a\}$

Solution: The DFA to accept a string w such that $|w| \bmod 3 = 0$ can be written as shown below (see Example 2):



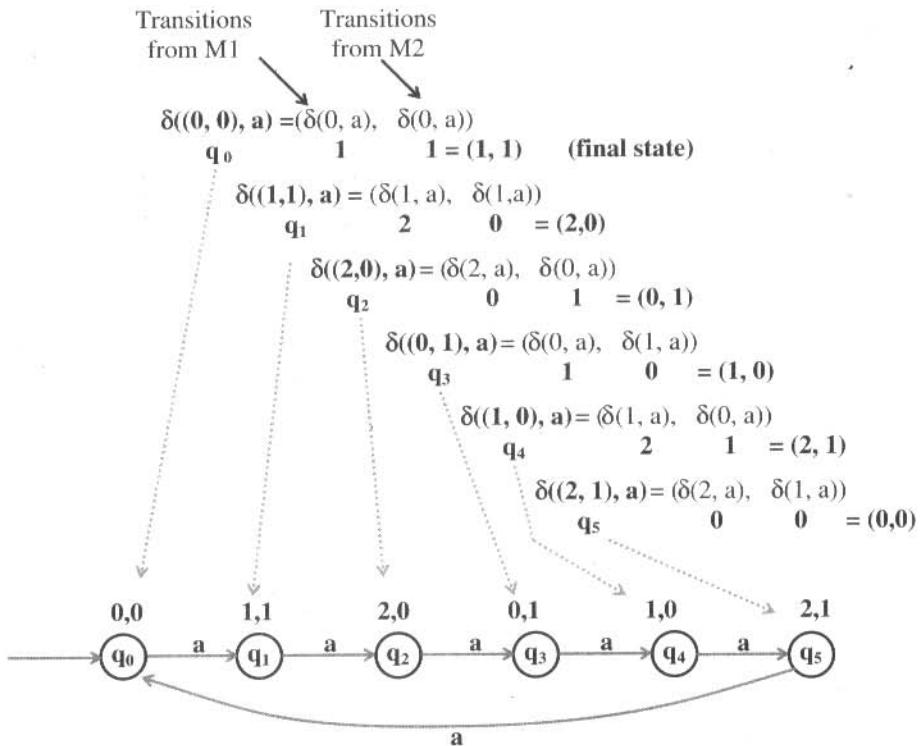
On similar lines, the DFA to accept a string w such that $|w| \bmod 2 = 0$ can be written as shown below (see Example 1):



Transitions: The transitions of DFA which has strings of w with $|w| \bmod 3$ and $|w| \bmod 2$ can be obtained by taking the cross product of $Q1$ and $Q2$ as shown below:

$$Q1 \times Q2 = \{(0,0), (0,1), (1,0), (1,1), (2,0), (2,1)\}$$

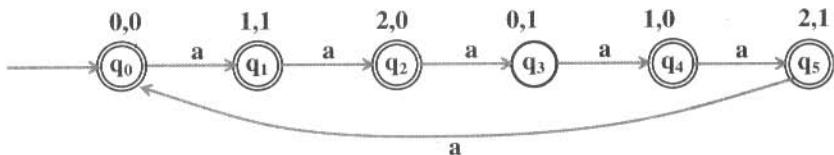
The transitions on each of the pair (x, y) can be obtained as shown below:



Case 1: To accept strings of w such that $|w| \bmod 3 \geq |w| \bmod 2$, the pairs (x, y) such that $x \geq y$ are final states. So, in the above DFA, the final states are:

$$F = \{ (0, 0), (1, 1), (2, 0), (1, 0), (2, 1) \}$$

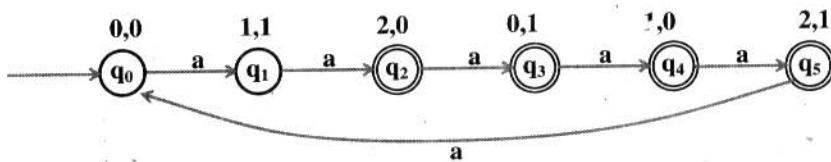
So, the DFA to accept the given language is shown below:



Case 2: To accept strings of w such that $|w| \bmod 3 \neq |w| \bmod 2$, $w \in \Sigma^*$ and $\Sigma = \{a\}$ the pairs (x, y) such that $x \neq y$ are final states. So, the final states in the DFA are:

$$F = \{ (2, 0), (0, 1), (1, 0), (2, 1) \}$$

So, the DFA to accept the given language is shown below:



Example 5: Now, let us “Obtain a DFA to accept the following language $L = \{ w \text{ such that}$

- a) $|w| \bmod 3 \geq |w| \bmod 2$ where $w \in \Sigma^*$ and $\Sigma = \{a, b\}$
- b) $|w| \bmod 3 \neq |w| \bmod 2$ where $w \in \Sigma^*$ and $\Sigma = \{a, b\}$

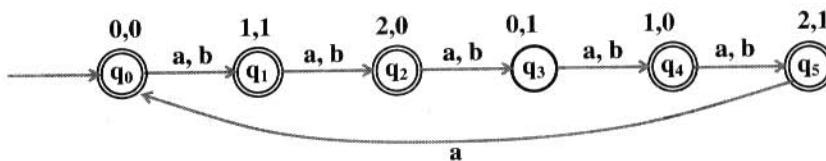
Solution: The solution is exactly similar to the above, but, the extra label b should be added along with a as shown below:

Case 1: To accept strings of w such that $|w| \bmod 3 \geq |w| \bmod 2$, $w \in \Sigma^*$ and $\Sigma = \{a, b\}$.

The pairs (x, y) such that $x \geq y$ are final states. So, in the above DFA, the final states are:

$$F = \{ (0, 0), (1, 1), (2, 0), (1, 0), (2, 1) \}$$

So, the DFA to accept the given language is shown below:

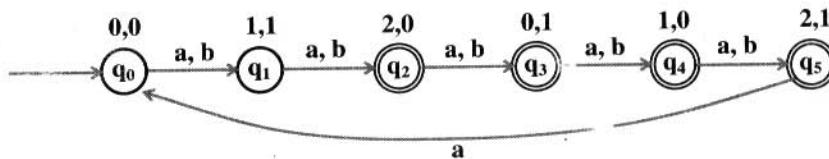


Case 2: To accept strings of w such that $|w| \bmod 3 \neq |w| \bmod 2$, $w \in \Sigma^*$ and $\Sigma = \{a, b\}$.

The pairs (x, y) such that $x \neq y$ are final states. So, the final states in the DFA are:

$$F = \{ (2, 0), (0, 1), (1, 0), (2, 1) \}$$

So, the DFA to accept the given language is shown below:



Example 6: Let us “Obtain a DFA to accept the language $L = \{ w : |w| \bmod 5 \neq 0 \}$ on $\Sigma = \{a\}$ ”.

Solution: It is given that $L = \{ w : |w| \bmod 5 \neq 0 \}$ where $\Sigma = \{a\}$ which indicates that the language consists of strings of a's which are not multiples of 5 a's.

Identify the number of states: The given language can be interpreted as strings of only a's such that the number of a's in the string is not divisible by 5. Note that

$$|w| \bmod 5$$

results in remainder 0, 1, 2, 3 and 4 which results in five cases as shown below:

- **Case 0:** Results in remainder 0: The state is identified as q_0 
- **Case 1:** Results in remainder 1. The state is identified as q_1 
- **Case 2:** Results in remainder 2. The state is identified as q_2 
- **Case 3:** Results in remainder 3. The state is identified as q_3 
- **Case 4:** Results in remainder 4. The state is identified as q_4 

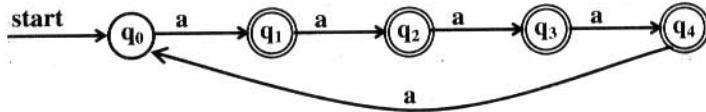
Identify the start state and final state: Before reading any of the input symbols, number of a's will be zero. So, $0 \bmod 5 = 0$ which results in case 0. Hence, state q_0 is start state. Since, it is required to accept string of a's such that $|w| \bmod 5 \neq 0$ (i.e., remainder is not zero), other than q_0 all other states can be the final states. So, the states q_1, q_2, q_3 and q_4 are final states.

Design: Once the start state and final states are identified, the transitions can be easily obtained as shown below:

- From a state with remainder 0 (case 0) denoted by q_0 , on reading a results in remainder 1 (case 1) denoted by q_1 . So, $\delta(q_0, a) = q_1$
- From a state with remainder 1 (case 1) denoted by q_1 , on reading a results in remainder 2 (case 2) denoted by q_2 . So, $\delta(q_1, a) = q_2$
- From a state with remainder 2 (case 2) denoted by q_2 , on reading a results in remainder 3 (case 3) denoted by q_3 . So, $\delta(q_2, a) = q_3$
- From a state with remainder 3 (case 3) denoted by q_3 , on reading a results in remainder 4 (case 4) denoted by q_4 . So, $\delta(q_3, a) = q_4$
- From a state with remainder 4 (case 4) denoted by q_4 , on reading a results in remainder 0 (case 0) denoted by q_0 . So, $\delta(q_4, a) = q_0$

So, the DFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as:

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $\Sigma = \{a\}$
- δ = shown in transition diagram as shown below:

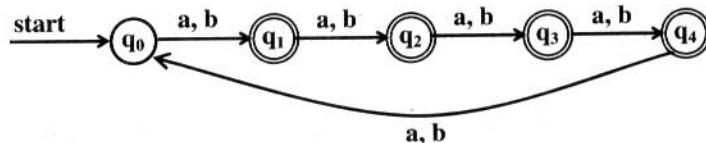


- q_0 = start state
- $F = \{q_1, q_2, q_3, q_4\}$

Example 7: Let us “Obtain a DFA to accept the language $L = \{ w : |w| \bmod 5 \neq 0 \}$ on $\Sigma = \{a,b\}$ ”.

Solution: The design is exactly similar to the previous problem but each arrow is labeled with a , b . So, the DFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as:

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $\Sigma = \{a, b\}$
- δ = shown in transition diagram as shown below:



- q_0 = start state
- $F = \{q_1, q_2, q_3, q_4\}$

Example 8: Let us “Obtain a DFA to accept strings of a’s and b’s having even number of a’s and even number of b’s”.

Solution: It is given that $L = \{ w : w \text{ has even number of a's and even number of b's.}\}$

Identify the number of states: The string w made up of a’s and b’s may have:

- Even number of a’s denoted by E_a
- Even number of b’s denoted by E_b
- Odd number of a’s denoted by O_a
- Odd number of b’s denoted by O_b

So, even/odd a's along with even/odd b's results in following four cases (four states):

- **Case 0:** Strings having Even a's and Even b's is denoted by: q_0
- **Case 1:** Strings having Even a's, Odd b's is denoted by: q_1
- **Case 2:** Strings having Odd a's, Even b's is denoted by: q_2
- **Case 3:** Strings having Odd a's, Odd b's is denoted by: q_3



Identify the start state and final state: Before reading any of the input symbols, number of a's and number of b's will be zero which represent Even a's and Even b's (case 0). So, the state q_0 is the start state.

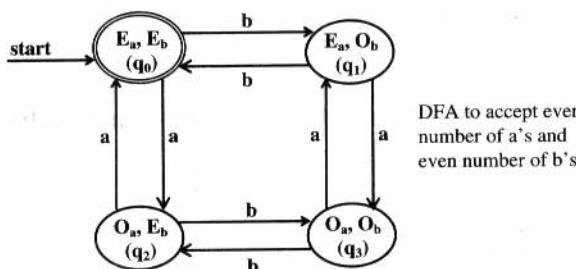
Since, it is required to accept Even a's and Even b's, the state q_0 with even a's and even b's is the final state.

Design: Once the start state and final states are identified the transitions can be easily obtained as shown below:

- From a state E_a with even number of a's, on reading input symbol a , results in odd number of a's denoted by O_a . So, $\delta(E_a, a) = O_a$
- From a state O_a with odd number of a's, on reading input symbol a , results in even number of a's denoted by E_a . So, $\delta(O_a, a) = E_a$
- From a state E_b with even number of b's, on reading input symbol b , results in odd number of b's denoted by O_b . So, $\delta(E_b, b) = O_b$
- From a state O_b with odd number of b's, on reading input symbol b , results in even number of b's denoted by E_b . So, $\delta(O_b, b) = E_b$

So, the DFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as:

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b\}$
- δ = shown in transition diagram



- q_0 = start state
- $F = \{q_0\}$

Note: The language accepted by above DFA can be written as:

$$L = \{w : w \text{ has even number of } a's \text{ and even number of } b's.\}$$

or

$$L = \{w : \text{Both } N_a(w) \text{ and } N_b(w) \text{ are divisible by 2}\}$$

or

$$L = \{w : \text{Both } N_a(w) \text{ and } N_b(w) \text{ are multiples of 2}\}$$

or

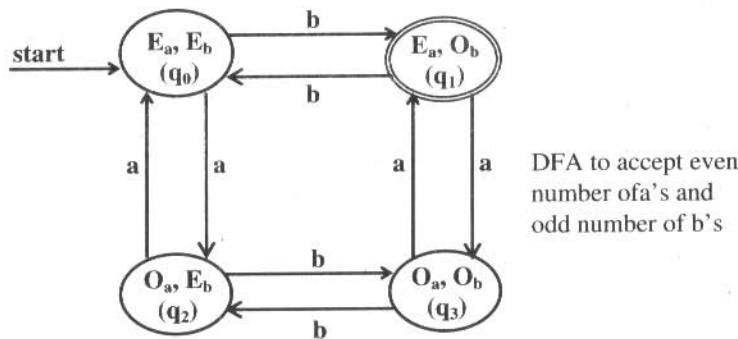
$$L = \{w \mid N_a(w) \bmod 2 = 0 \text{ and } N_b(w) \bmod 2 = 0\}$$

Note: $N_a(w)$ is the total number of a 's in the string w and $N_b(w)$ is the total number of b 's in the string w .

Note: The DFA to accept even number of a 's and odd number of b 's can be obtained by making:

$$\begin{matrix} E_a, O_b \\ (q_1) \end{matrix}$$

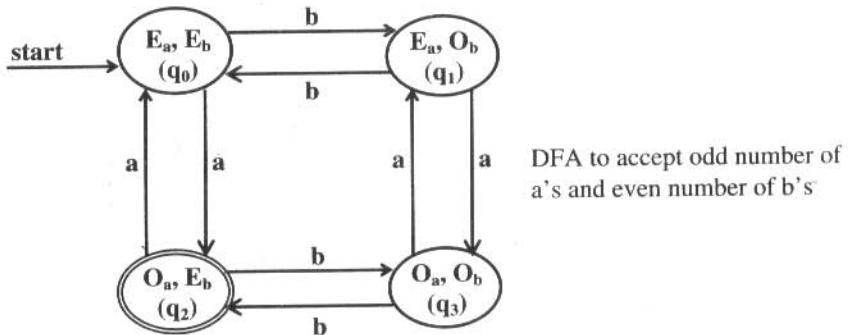
as the only final state as shown below:



Note: The DFA to accept odd number of a 's and even number of b 's can be obtained by making:

$$\begin{matrix} O_a, E_b \\ (q_2) \end{matrix}$$

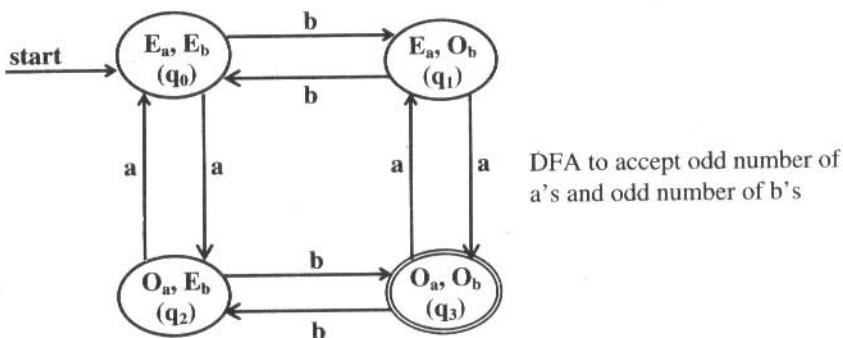
as the only final state as shown below:



Note: The DFA to accept odd number of a's and odd number of b's can be obtained by making:

O_a, O_b
(q_3)

as the only final state as shown below:



Example 9: Obtain a DFA to accept strings of a's and b's such that

$$L = \{ w \mid w \in (a+b)^* \text{ such that } N_a(w) \bmod 3 = 0 \text{ and } N_b(w) \bmod 2 = 0 \}$$

Solution: The $N_a(w) \bmod 3$ gives the remainder after dividing number of a's by 3.

The possible remainders are { 0, 1, 2 } and can be represented as:

$$\begin{array}{c} \downarrow \quad \downarrow \quad \downarrow \\ Q_1 = \{ A_0, A_1, A_2 \} \end{array} \quad (1)$$

The $N_b(w) \bmod 2$ gives the remainder after dividing number of b's by 2.

The possible remainders are {0, 1} and can be represented as:

$$\begin{array}{c} \downarrow \\ Q_2 = \{B_0, B_1\} \\ \downarrow \end{array} \quad (2)$$

Identify the states of DFA: Since each state of DFA should keep track of $N_a(w) \bmod 3$ and $N_b(w) \bmod 2$, the possible states of the DFA can be obtained by $Q_1 \times Q_2$ (cross product) and can be represented as shown below:

$$Q_1 \times Q_2 = \{(A_0, B_0), (A_0, B_1), (A_1, B_0), (A_1, B_1), (A_2, B_0), (A_2, B_1)\}$$

where

- A_0 indicates that $N_a(w) \bmod 3 = 0$
- A_1 indicates that $N_a(w) \bmod 3 = 1$
- A_2 indicates that $N_a(w) \bmod 3 = 2$
- B_0 indicates that $N_b(w) \bmod 2 = 0$
- B_1 indicates that $N_b(w) \bmod 2 = 1$

Identify the start state: Before reading any of the input symbols, number of a's and number of b's will be zero. So, $N_a(w) \bmod 3 = 0$ and $N_b(w) \bmod 2 = 0$ which can be denoted by the state (A_0, B_0) . So, (A_0, B_0) is the start state.

Identify the final state: Since, it is required to accept the language

$$L = \{w : N_a(w) \bmod 3 = 0 \text{ and } N_b(w) \bmod 2 = 0\}$$

$$\begin{array}{cc} \downarrow & \downarrow \\ A_0 & B_0 \end{array}$$

the state (A_0, B_0) is the final state.

Design: Once the start state and final states are identified, the transitions for the input symbol a can be obtained as shown below:

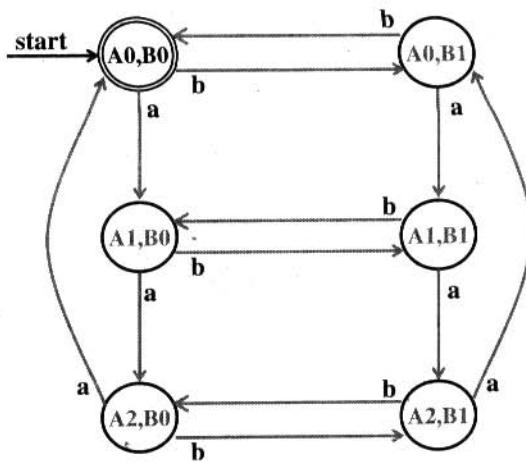
- From state A_0 , on reading input symbol a , the machine should change the state to A_1 . So, $\delta(A_0, a) = A_1$
- From state A_1 , on reading input symbol a , the machine should change the state to A_2 . So, $\delta(A_1, a) = A_2$
- From state A_2 , on reading input symbol a , the machine should change the state to A_0 . So, $\delta(A_2, a) = A_0$

Similarly, the transitions for the input symbol b can be obtained as shown below:

- From state B_0 , on reading input symbol b , the machine should change the state to B_1 . So, $\delta(B_0, b) = B_1$
- From state B_1 , on reading input symbol b , the machine should change the state to B_0 . So, $\delta(B_1, b) = B_0$

So, the DFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as:

- $Q = \{(A0, B0), (A0, B1), (A1, B0), (A1, B1), (A2, B0), (A2, B1)\}$
- $\Sigma = \{a, b\}$
- $q_0 = \{(A0, B0)\}$ is the start state
- $F = \{(A0, B0)\}$
- δ = shown in transition diagram



Note: By changing the final states of DFA various languages can be accepted by DFAs as shown below:

- To accept the language $L = \{w \mid w \in (a+b)^* N_a(w) \bmod 3 = 1 \text{ and } N_b(w) \bmod 2 = 0\}$

$$\begin{array}{cc} \downarrow & \downarrow \\ \textbf{A1} & \textbf{B0} \end{array}$$

make the state $(A1, B0)$ as the final state.

- To accept the language $L = \{w \mid w \in (a+b)^* N_a(w) \bmod 3 = 2 \text{ and } N_b(w) \bmod 2 = 0\}$

$$\begin{array}{cc} \downarrow & \downarrow \\ \textbf{A2} & \textbf{B0} \end{array}$$

make the state $(A2, B0)$ as the final state.

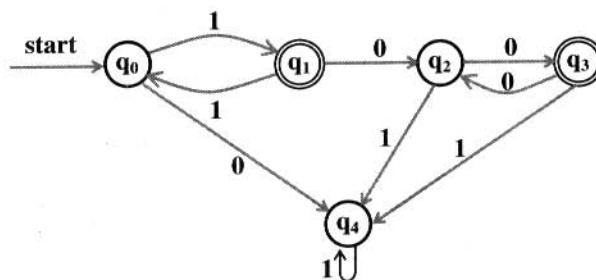
- To accept the language $L = \{w \mid w \in (a+b)^* N_a(w) \bmod 3 = 2 \text{ and } N_b(w) \bmod 2 = 1\}$

$$\begin{array}{cc} \downarrow & \downarrow \\ \textbf{A2} & \textbf{B1} \end{array}$$

make the state $(A2, B1)$ as the final state and so on.

Example 10: Now, let us “Draw a DFA to accept the language: $L = \{w : w \text{ has odd number of 1's and followed by even number of 0's}\}$ Completely define DFA and transition function:

Solution: The DFA to accept strings of 0's and 1's such that the string has odd number of 1's and followed by even number of 0's is shown below:



Example 11: Now, let us “Obtain a DFA to accept strings of a's and b's such that the number of a's is divisible by 5 and number of b's is divisible by 3”.

Note: Observe that the problem is similar to that the previous problem with more number of states.

Solution: The given language can be interpreted as shown below:

$$L = \{w \mid w \in (a+b)^* \text{ } N_a(w) \bmod 5 = 0 \text{ and } N_b(w) \bmod 3 = 0\}$$

The $N_a(w) \bmod 5$ gives the remainder after dividing number of a's by 5.

The possible remainders are { 0, 1, 2, 3, 4 } and can be represented as:

↓ ↓ ↓ ↓ ↓

$$Q_1 = \{A0, A1, A2, A3, A4\} \quad (1)$$

The $N_b(w) \bmod 3$ gives the remainder after dividing number of b's by 3.

The possible remainders are { 0, 1, 2 } and can be represented as:

↓ ↓ ↓

$$Q_2 = \{B0, B1, B2\} \quad (2)$$

Identify the states of DFA: Since each state of DFA should keep track of $N_a(w) \bmod 3$ and $N_b(w) \bmod 2$, the possible states of the DFA can be obtained by $Q_1 \times Q_2$ (cross product) and can be represented as shown below:

$$\begin{aligned} Q_1 \times Q_2 = & \{(A_0, B_0), (A_0, B_1), (A_0, B_2), \\ & (A_1, B_0), (A_1, B_1), (A_1, B_2), \\ & (A_2, B_0), (A_2, B_1), (A_2, B_2), \\ & (A_3, B_0), (A_3, B_1), (A_3, B_2), \\ & (A_4, B_0), (A_4, B_1), (A_4, B_2)\} \end{aligned}$$

where

- A0 indicates that $N_a(w) \bmod 5 = 0$
- A1 indicates that $N_a(w) \bmod 5 = 1$
- A2 indicates that $N_a(w) \bmod 5 = 2$
- A3 indicates that $N_a(w) \bmod 5 = 3$
- A4 indicates that $N_a(w) \bmod 5 = 4$
- B0 indicates that $N_b(w) \bmod 3 = 0$
- B1 indicates that $N_b(w) \bmod 3 = 1$
- B2 indicates that $N_b(w) \bmod 3 = 2$

Identify the start state: Before reading any of the input symbols, number of a's and number of b's will be zero. So, $N_a(w) \bmod 5 = 0$ and $N_b(w) \bmod 3 = 0$ which can be denoted by the state (A0, B0). So, (A0, B0) is the start state.

Identify the final state: Since, it is required to accept the language

$$\begin{array}{c} L = \{w : N_a(w) \bmod 5 = 0 \text{ and } N_b(w) \bmod 3 = 0\} \\ \downarrow \qquad \downarrow \\ \mathbf{A0} \qquad \qquad \mathbf{B0} \end{array}$$

the state (A0, B0) is the final state.

Design: Once the start state and final states are identified, the transitions for the input symbol a can be obtained as shown below:

- From state A0, on reading input symbol a , the machine should change the state to A1. So, $\delta(A_0, a) = A_1$
- From state A1, on reading input symbol a , the machine should change the state to A2. So, $\delta(A_1, a) = A_2$
- From state A2, on reading input symbol a , the machine should change the state to A3. So, $\delta(A_2, a) = A_3$
- From state A3, on reading input symbol a , the machine should change the state to A4. So, $\delta(A_3, a) = A_4$
- From state A4, on reading input symbol a , the machine should change the state to A0. So, $\delta(A_4, a) = A_0$

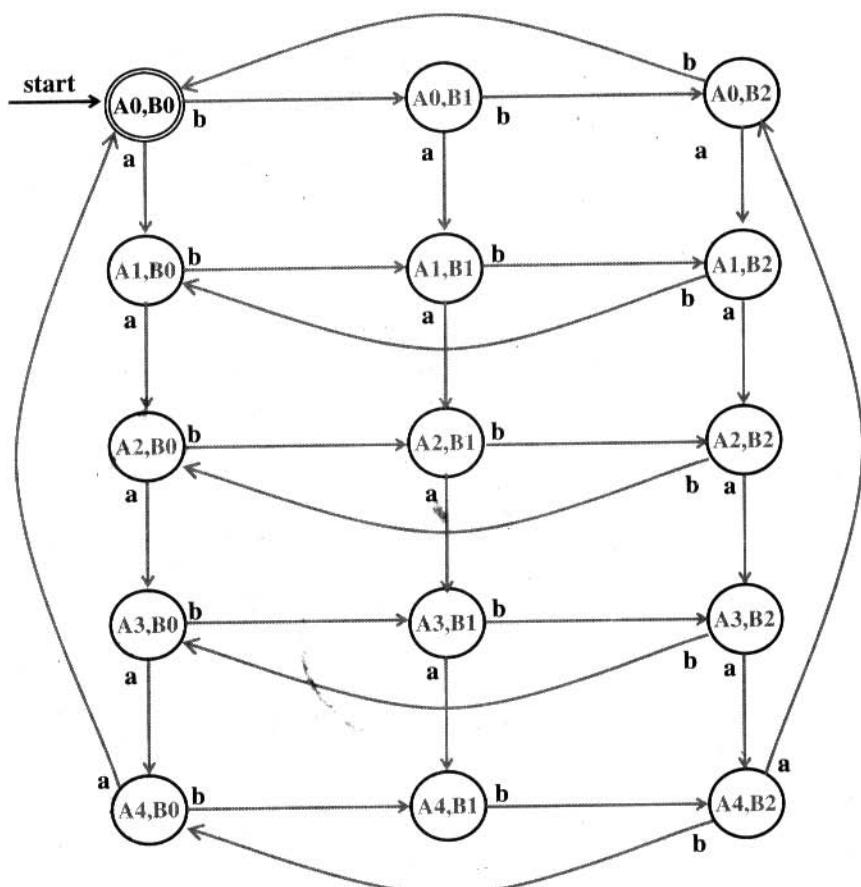
Similarly, the transitions for the input symbol b can be obtained as shown below:

- From state B0, on reading input symbol b , the machine should change the state to B1. So, $\delta(B_0, b) = B_1$
- From state B1, on reading input symbol b , the machine should change the state to B2. So, $\delta(B_1, b) = B_2$

- From state B_2 , on reading input symbol b , the machine should change the state to B_0 . So, $\delta(B_2, b) = B_0$

So, the DFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as:

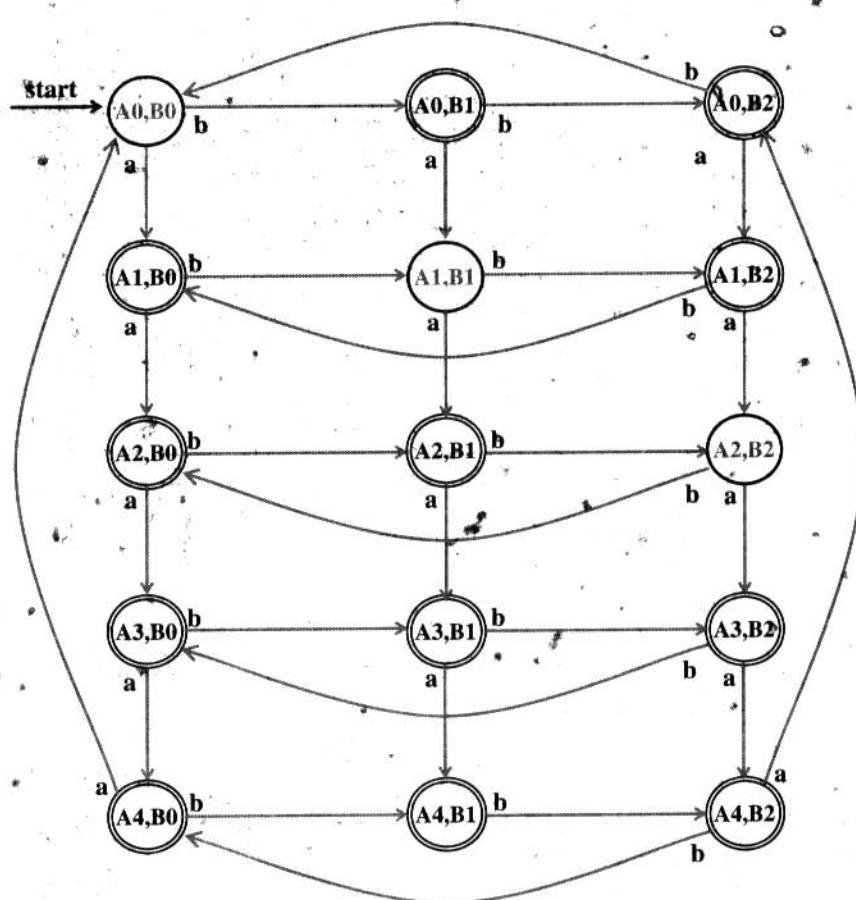
- $Q = \{(A_0, B_0), (A_0, B_1), (A_0, B_2), (A_1, B_0), (A_1, B_1), (A_1, B_2), (A_2, B_0), (A_2, B_1), (A_2, B_2), (A_3, B_0), (A_3, B_1), (A_3, B_2), (A_4, B_0), (A_4, B_1), (A_4, B_2)\}$
- $\Sigma = \{a, b\}$
- $q_0 = (A_0, B_0)$ is the start state
- $F = \{(A_0, B_0)\}$
- δ = shown below using the transition diagram



Now, let us "Construct a DFA to accept the following language":

$$L = \{w : n_a(w) \bmod 5 \neq n_b(w) \bmod 3\}$$

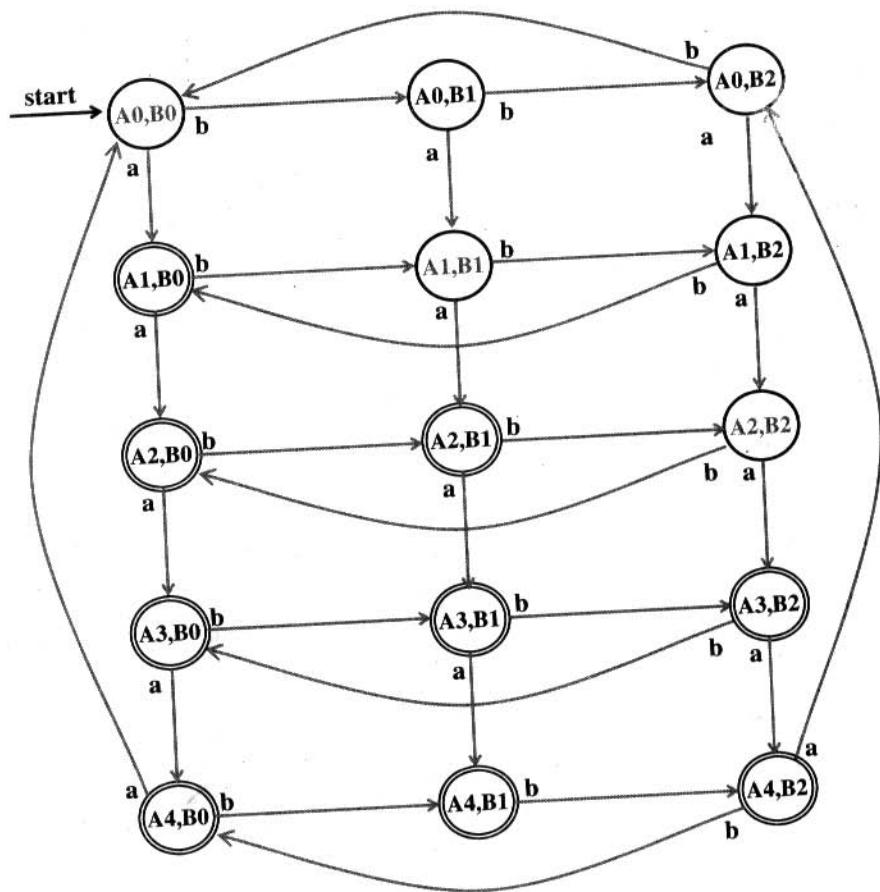
Note: The design is exactly same as the above problem. Except the states (A_0, B_0) , (A_1, B_1) and (A_2, B_2) , the rest of the states are final states. The DFA to accept the given language is shown below:



Now, let us "Construct a DFA to accept the following language":

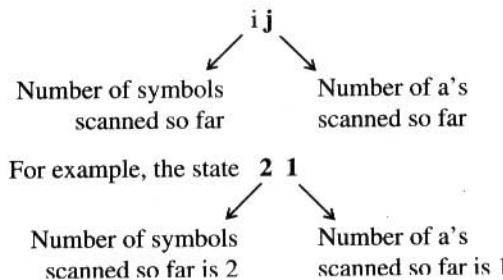
$$L = \{w : n_a(w) \bmod 5 > n_b(w) \bmod 3\}$$

Note: The design is exactly similar to the above problem. But, in all possible states (A_i, B_j) , the index i should be greater than index j . So, the final DFA to accept the above language is shown below:



Example 12: Now, let us “Obtain a DFA to accept strings of a’s and b’s such that each block of 5 consecutive symbols have at least two a’s”.

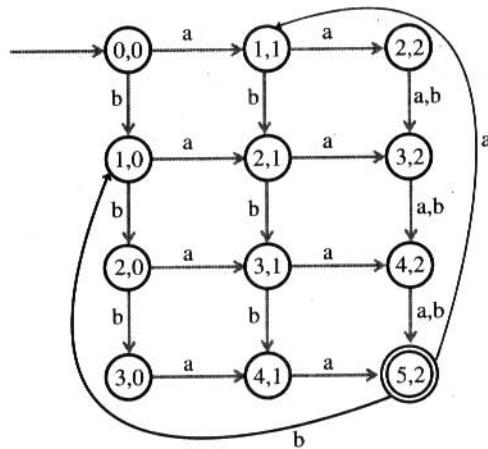
Solution: This DFA should keep track of number of symbols scanned for and number of a’s. Hence, each state of DFA is represented as shown below:



The transition can be obtained using the following relationships:

- The state 00 is the start state of DFA.
- If $i \leq 4$ and $j \leq 1$ then $\delta(ij, a) = \delta(i+1, j+1)$.
- If $i \leq 4$ and $j = 2$ then $\delta(ij, x) = \delta(i+1, j)$ where x can be either a or b .
- 52 indicates that current block has 5 symbols and has at least 2 a's. So, the string has to be accepted and 52 is the final state.
- 50 and 51 states indicate that the block has 5 symbols and at least 2 a's are not present. So, the string has to be rejected and represent the trap state.
- $\delta(52, a) = 11$ indicate that the beginning of the next block has scanned one symbol and has one a .
- $\delta(52, b) = 10$ indicate that the beginning of the next block has scanned one symbol and has no a .

The complete DFA is shown below:



Example 13: Now, let us “Prove that $\delta^*(q, xa) = \delta(\delta^*(q, x), a)$ ” where x is a string and a is the current input symbol.

Proof: It is required to prove that $\delta^*(q, xa) = \delta(\delta^*(q, x), a)$. From this statement it is observed that:

$$w = xa$$

where

- a is the last symbol of w
- x is the remaining string of w
- q is the current state of the machine

From state q on input string w let us assume that the machine enters into state p . This is given by the transition:

$$\delta^*(q, w) = p \quad (1)$$

Since $w = xa$, the above transition can be written as shown below:

$$\delta^*(q, xa) = p \quad (2)$$

Since $w = xa$, first, we should find the transition on string x and then on symbol a .

On string x: From state q on input string x , let the machine enters into state r . This is denoted by:

$$\delta^*(q, x) = r \quad (3)$$

On symbol a: From state r on input symbol a , the machine should enter into state p . This is denoted by:

$$\delta(r, a) = p$$

Replacing the state r in above transition using equation (3), we have:

$$\delta(\delta^*(q, x), a) = p \quad (4)$$

Comparing equation (2) and (4) observe that R.H.S's are equal. Therefore, L.H.S.'s are also equal. So,

$$\delta^*(q, xa) = \delta(\delta^*(q, x), a)$$

i.e., $\delta^*(q, w) = \delta^*(q, xa) = \delta(\delta^*(q, x), a)$

Example 14: Show that $\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$ for any state w and strings x and y .

Note: The symbol δ^* can also be denoted by the symbol $\hat{\delta}$.

Solution: The statement to be proved is:

$$\delta^*(q, xy) = \delta^*(\delta^*(q, x), y) \quad (1)$$

Let us proceed by induction on the length of y .

Basis: Let $\delta(q, x) = p$

(2)

If $y = \epsilon$, L.H.S. of Eq. (1) = $\delta^*(q, xy)$

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

$$= p$$

[By replacing y by ϵ]

[from Eq. (2)]

Now, consider R.H.S. of Eq. (1) = $\delta^*(\delta^*(q, x), y)$

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

$$= \delta^*(p, \epsilon)$$

$$= p$$

[By relation (2)]

[Replacing y by ϵ]

[According to the property: $\delta(q, \epsilon) = q$]

Since, L.H.S. is same as R.H.S the relation (1) holds good.

Induction: Let us split the string y into za i.e., $y = za$

a is last symbol of y

(3)

Consider R.H.S. of relation (1) = $\delta^*(\delta^*(q, x), y)$

$$= \delta^*(\delta^*(q, x), za)$$

Substituting relation (3)

$$= \delta^*(\delta^*(q, x), z, a)$$

Treat $\delta^*(q, x)$ as a state

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

Inductive hypothesis

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

Inductive hypothesis

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

By definition

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

Using relation (3)

$$= \text{L.H.S. of relation (1)}$$

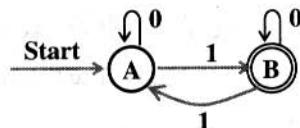
Hence, the proof.

■ **Example 15:** Consider the DFA with following transition table:

	δ	0	1
\rightarrow	A	A	B
$*B$	B	B	A

Informally describe the language accepted by this DFA and prove by induction on the length of an input string, that your description is correct.

Solution: The transition diagram corresponding to the transition table is shown below:



State A: The machine in state A can consume any number of 0's. But, the number of 1's consumed at state A is 0 which is EVEN. So, from EVEN state on consuming a 1, the machine enters into ODD state called B.

State B: In ODD state B, the machine can consume any number of 0's. But, if the input symbol is 1, the machine goes to state A which consumes odd number of 1's.

Informal description of language: So, the language accepted by the machine is “Strings of 0's and 1's with odd number of 1's”.

Proof: It is required to show by induction that $\delta^*(A, w) = A$ if and only if w has even number of 1's.

Basis: $|w| = 0$. Since w is an empty string, surely has an even number of 1's, namely zero 1's, and so $\delta^*(A, w) = A$.

Induction: Let us consider a string shorter than w . So, let $w = za$ where a is either 0 or 1. (1)

Case 1: Let $a = 0$. So, if w has an odd number of 1's, z also has odd number of 1's.

By the inductive hypothesis, $\delta^*(A, z) = B$. The transitions of the DFA tell us

$$\delta^*(A, w) = B$$

If w has an even number of 1's, then z also has even number of 1's. By the inductive hypothesis, $\delta^*(A, z) = A$. The transitions of the DFA tell us

$$\delta^*(A, w) = A$$

Thus, in this case, $\delta^*(A, w) = A$ if and only if w has an even number of 1's.

$\delta^*(A, w) = B$ if and only if w has an odd number of 1's.

Case 2: Let $a = 1$. So, if w has an even number of 1's, then z has an odd number of 1's. By the inductive hypothesis, $\delta^*(A, z) = B$. The transitions of the DFA tell us

$$\delta^*(A, w) = A$$

If w has an odd number of 1's, then z has an even number of 1's. By the inductive hypothesis, $\delta^*(A, z) = A$. The transitions of the DFA tell us

$$\delta^*(A, w) = B$$

Thus, in this case as well, $\delta^*(A, w) = A$ if and only if w has an even number of 1's

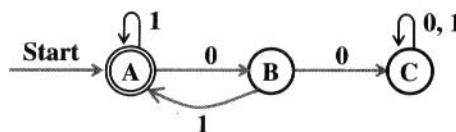
$\delta^*(A, w) = B$ if and only if w has odd number of 1's.

Example 16: Consider the DFA with following transition table:

δ	0	1
*A	B	A
*B	C	A
C	C	C

Informally describe the language accepted by this DFA and prove by induction on the length of an input string, that your description is correct.

Solution: The transition diagram corresponding to the transition table is shown below:



Observe the following facts from the above transition diagram:

- A string w having either ϵ or ends in 1 but does not have the substring 00 is consumed at state A
- A string w ends in 0 and does not have the substring 00 is consumed at state B
- A string w that contains the substring 00 is consumed at state C

Informal description of the language: Since A is the final state, the language accepted by DFA is “Strings of 0’s and 1’s having ϵ or ending with 1 but does not have a substring 00”.

Proof: To start with machine will be in start state A.

Basis: If $w = \epsilon$, the machine will be in state A which is the final state. Thus, empty string is accepted by DFA.

Induction: If w has only 1’s the machine will be in state A. So, $w1$ does not contain the substring 00 and hence the string is accepted by DFA.

If w ends with 0, the machine goes to state B and hence string ends with one 0. But, on 1 the machine enters into state A and thus the substring 00 is not accepted.

1.7. Applications of Finite Automata

Now, let us see “Why to study finite automata? or What are applications of finite automata?”

Some of the applications where automata play an important role are shown below:

- Design of digital circuits: The FA is used during designing and checking the behavior of the digital circuits using software. The FA is very useful in hardware design such as circuit verification, in design of automatic traffic signals etc.
- Compiler construction: Used in the design of *lexical analyzer* (the first phase of compiler design) which breaks the input text into various units such as identifiers, keywords, punctuation etc.
- String matching: In designing a software for identifying the words, phrases and other patterns in large bodies of text (such as collection of web pages).
- String processing: To write software for processing the natural language (Ex: Speech processing). Large natural vocabularies can be described which includes the applications such as spelling checkers and advisers, multi-language dictionaries, indenting the documents etc.
- Software design: In building the software to verify the systems having finite number of states (for example, communication protocols in computer networks).
- Other applications: The FA are used in variety of applications in Artificial intelligence and knowledge engineering, in game theory and games, computer graphics, linguistics, etc.

Exercises

1. What is DFA? Explain with an example.
2. When we say that a language is accepted by the machine? Explain with example.
3. When a given language is not accepted by DFA? Explain with example.
4. How DFA's can be represented? Explain with example.
5. What is a transition diagram/graph?
6. Obtain a DFA to accept strings of a's and b's starting with the string ab.
7. Draw a DFA to accept string of 0's and 1's ending with the string 011.
8. Obtain a DFA to accept strings of a's and b's having a sub string aa.
9. Obtain a DFA to accept strings of a's and b's except those containing the substring aab.
10. Obtain DFAs to accept strings of a's and b's having exactly one a, atleast one a, not more than three a's.
11. Obtain a DFA to accept the language $L = \{ awa \mid w \in (a+b)^* \}$.
12. Obtain a DFA to accept even number of a's, odd number of a's.
13. Obtain a DFA to accept strings of a's and b's having even number of a's and b's.
14. Obtain a DFA to accept odd number of a's and even number of b's.
15. Obtain a DFA to accept even number of a's and odd number of b's.

16. Obtain a DFA to accept strings of a's and b's having odd number of a's and b's.
17. Obtain a DFA to accept strings of 0's, 1's and 2's beginning with a '0' followed by odd number of 1's and ending with a '2'.
18. Obtain a DFA to accept binary odd numbers.
19. Obtain a DFA to accept strings of a's and b's starting with at least two a's and ending with at least two b's.
20. Obtain a DFA to accept strings of a's and b's with at most two consecutive b's.
21. Obtain a DFA to accept the language $L = \{ w : |w| \bmod 3 = 0 \text{ on } \Sigma = \{a, b\} \}$.
22. Obtain a DFA to accept the language $L = \{ w : |w| \bmod 5 \neq 0 \text{ on } \Sigma = \{a, b\} \}$.
23. What is a regular language?
24. What are the applications of finite automaton?

1.8. Disadvantages of DFA

Now, let us see "What are the various disadvantages of DFA?" The various disadvantages of DFA are listed below:

- Constructing a DFA is difficult
- The DFA cannot guess about its input
- The DFA is not very powerful
- At any point of time, the DFA is in only state. So, a DFA does not have the power to be in several states at once

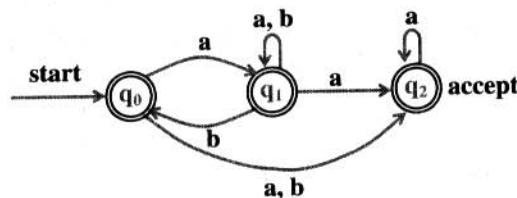
1.9. Why NFA?

Computers are completely deterministic machines (DFA). The state of the computer can be predicted from the input and initial state. We cannot find a computer which is non-deterministic. In such case, the question is "Why non-deterministic Finite Automata?" All the disadvantages of DFA mentioned earlier can be overcome using Non-Deterministic Finite Automata (in short NFA or NDFA). The various advantages of NFA are:

- Very easy to construct

- A “non-deterministic” finite automaton has the ability to guess something about its input
- A “non-deterministic” finite automaton is more powerful than DFA
- It has the power to be in several states at once
- An NFA is an efficient mechanism to describe some complicated languages concisely

Before defining NFA, let us consider the following transition diagram:



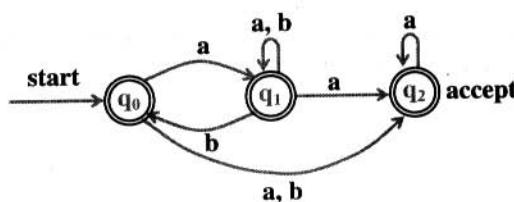
Let us see “What is the specialty of the above finite automaton?” Observe the following facts:

- From a given state there is possibility of zero, one or more edges leaving that state after consuming the input symbol.
- Example 1:** From state q_2 , there are zero edges (indicates no edges) for the input symbol b .
- Example 2:** From state q_2 , there is one edge (q_2, q_2) labeled a .
- Example 3:** From state q_0 , there are two edges (q_0, q_1) and (q_0, q_2) labeled a .

Note: In the FA shown above, there can be zero, one or more transitions on an input symbol. Such machines are called non-deterministic finite state machines or non-deterministic finite automata.

1.10. Non-Deterministic Finite Automaton

Before worrying about the definition, let us consider the following pictorial representation of NFA.



From the above figure, observe following components of NFA:

- **States:** The NFA has three states q_0 , q_1 and q_2 and can be represented as

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

Note: The power set of Q is denoted by 2^Q which is set of subsets of set Q. This is denoted as shown below:

$$2^Q = \{ \emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\} \}$$

- **Input alphabets:** Each edge is labeled with a or b and represent the input alphabets which can be denoted as:

$$\Sigma = \{a, b\}$$

- **Transitions:** Transition is nothing but change of state after consuming an input symbol. If there is a change of state from q_i to q_j on an input symbol a , then we write

$$\delta(q_i, a) = q_j$$

If there is a change of state from q_i to q_j and q_j to q_k on an input symbol a , then we write

$$\delta(q_i, a) = \{q_j, q_k\}$$

The transition diagram of above NFA is shown below:

Current state	Input	Next state	Representations
q_0	a	q_1, q_2	$\delta(q_0, a) = \{q_1, q_2\}$
q_0	b	q_2	$\delta(q_0, b) = q_2$
q_1	a	q_1, q_2	$\delta(q_1, a) = \{q_1, q_2\}$
q_1	b	q_0, q_2	$\delta(q_1, b) = \{q_0, q_2\}$
q_2	a	q_2	$\delta(q_2, a) = q_2$
q_2	b	q_1	$\delta(q_2, b) = q_1$

$\delta: Q \times \Sigma \rightarrow 2^Q$ [power set]

Now, let us see "What is a transition function?" The transition function δ is defined as:

$$\delta: Q \times \Sigma \rightarrow 2^Q$$

which is read as “ δ is a transition function which maps $Q \times \Sigma$ to 2^Q ”. For example, the change of state from state q on input symbol a to states p_1, p_2, \dots, p_n is denoted by

$$\delta(q, a) = \{ p_1, p_2, \dots, p_n \}$$

where

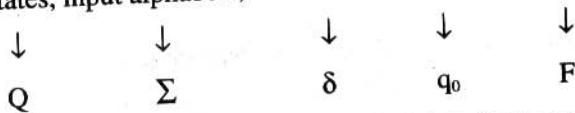
- δ : is a function called transition function
- q : is the first parameter representing the current state of the machine
- a : is the second parameter representing the current input symbol read
- p_1, p_2, \dots, p_n represent possible states of machine

Note: In other words, the transition function δ accepts two parameters namely state q and input symbol a as the parameters and returns set of states the machine enters into.

- **Start state (q_0):** q_0 with the label start is treated as the start state.
- **Final state (q_2):** q_2 with two circles is treated as the final or accept state.

Note: From this discussion, it is observed that the NFA has five components:

(states, input alphabets, transitions, start state, final states)



With this concept, now let us see “What is a non-deterministic finite automaton (NFA)?”

❖ **Definition:** The non-deterministic finite automaton in short NFA is 5-tuple or quintuple indicating five components:

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

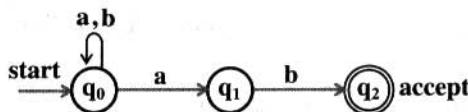
where

- M is the name of the machine. It can also be called by any name.
- Q is non-empty, finite set of states.
- Σ is non-empty, finite set of input alphabets.
- $\delta: Q \times \Sigma \rightarrow 2^Q$ i.e. δ is transition function which is a mapping from $Q \times \Sigma$ to 2^Q . Based on the current state and input symbol, the machine enters into one or more states.

- $q_0 \in Q$ – is the start state.
- $F \subseteq Q$ – is set of accepting or final states.

Suppose $\delta(q, a)$ is in P where P is in 2^Q and a is in Σ . This indicates that there is a transition from state q on an input symbol a to set of states P . Note: In an NFA there can be zero, one or more transitions on an input symbol.

For example, an NFA to accept strings of a's and b's ending with ab is shown below:



The above NFA can be specified formally as $M = (Q, \Sigma, \delta, q_0, F)$ where

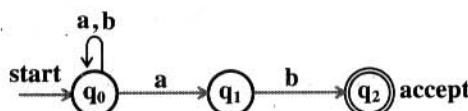
- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{a, b\}$
- q_0 is the start state
- $F = \{q_2\}$
- δ is shown below using the transition table:

δ	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset

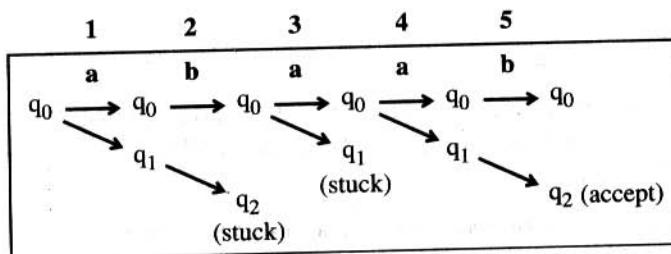
Note: In the transition table of NFA, each entry in the table should be denoted by a set. Also, when there is no transition at all from a given state on an input symbol, make an entry \emptyset indicating an empty set.

1.10.1. Moves made by NFA

Now, let us see “What are the states an NFA is in during the processing of input sequence abaab and abb?”.



Solution: The states an NFA is in during the processing of input sequence *abaab* is shown below:



The machine always starts from initial state q_0 . The various actions performed by NFA for the string **abaab** is shown below:

- 1) From q_0 on reading a , the NFA may go to q_0 or q_1

- 2) q_0 on b NFA goes to state q_0

q_1 on b NFA goes to state q_2

- 3) q_0 on a NFA goes to q_0 or q_1

Note: q_2 on a there is no transition and hence NFA is stuck.

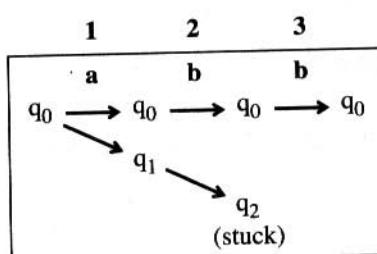
- 4) q_0 on a NFA goes to q_0 or q_1

Note: q_1 on a there is no transition and hence NFA is stuck.

- 5) q_0 on b NFA goes to q_0 . At the end, the NFA is in q_0 (non-final state). So, this path is not chosen.

But, q_1 on b NFA goes to q_2 . So, at the end, NFA is in q_2 which is a final state and hence the input string **abaab** is accepted by the machine.

The states an NFA is in during the processing of input sequence **abb** is shown below:



The machine always starts from initial state q_0 . The various actions performed by NFA for the string **abaab** is shown below:

- 1) From q_0 on reading a , the NFA may go to q_0 or q_1

- 2) q_0 on b NFA goes to state q_0

q_1 on b NFA goes to state q_2

3) q_0 on b NFA goes to state q_0

Note: q_2 on b there is no transition and hence NFA is stuck.

After the string abb , the machine is in state q_0 which is non-final state. So the string abb is rejected by the machine.

1.10.2. Extended Transition Function of NFA to Strings

Note: The transition $\delta(q, a) = P$ accepts two parameters namely state q and input symbol a as parameters and returns a set of states P where $P = \{p_1, p_2, p_3, \dots, p_n\}$ which represent the possible states of the NFA at a given instance. But, if there is a change of state from q to set of states P where $P \neq \{p_1, p_2, p_3, \dots, p_n\}$ on input string w , then we use extended transition function denoted by δ^* .

Note: We can also use $\hat{\delta}$ in place of δ^* . But, in our book let us use δ^* to denote extended transition. Now, let see "What is extended transition function δ^* for NFA?".

♦ **Definition:** The extended transition function δ^* describes what happens to a state of machine when the input is a string (sequence of symbols). Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA. The extended transition function $\delta^*: Q \times \Sigma^* \rightarrow 2^Q$ is defined recursively as shown below:

- **Basis:** $\delta^*(q, \epsilon) = \{q\}$. This indicates that if the machine is in state q and read no input, then the machine is still in state q .
- **Induction:** Let $w = xa$ where a is the last symbol of w and x is the remaining string of w . Let q is the current state and x is the string to be processed and after consuming the string x , let the state of the machine is $\{p_1, p_2, p_3, \dots, p_m\}$.

$$\text{i.e. } \delta^*(q, x) = \{p_1, p_2, p_3, \dots, p_m\}$$

Let the transition from $\{p_1, p_2, p_3, \dots, p_m\}$ on input symbol a is:

$$\delta(\{p_1, p_2, p_3, \dots, p_m\}, a) = \{r_1, r_2, r_3, \dots, r_k\}$$

$$\text{i.e. } \bigcup_{i=1}^m \delta(p_i, a) = \{r_1, r_2, r_3, \dots, r_k\}$$

Then $\delta^*(q, w) = \delta^*(q, xa) = \{r_1, r_2, r_3, \dots, r_k\}$.

Note: Thus, various properties of extended transition functions for an NFA can be:

- $\delta^*(q, \epsilon) = \{q\}$
- $\delta^*(q, w) = \delta^*(q, xa) = \{\delta(\delta^*(q, x)), a\}$ where $w = xa$
- $\delta^*(q, w) = \delta^*(q, ax) = \{\delta^*(\{\delta(q, x)\}, x)\}$ where $w = ax$

For example, change of state from state q on input string w to set of states $\{r_1, r_2, r_3, \dots, r_k\}$ is denoted by

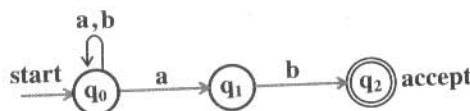
$$\delta^*(q, w) = \{r_1, r_2, r_3, \dots, r_k\}$$

where

- δ^* : is a function called as extended transition function
- q : is the first parameter representing the current state of the machine
- w : is the second parameter representing the current input string being read
- $\{r_1, r_2, r_3, \dots, r_k\}$: represent the possible states of the machine which is returned by the transition function

Note: The transition function δ^* accepts two parameters namely state q and input string w as the parameters and returns set of states of the machine.

Now, let us see “What are the moves made by the following NFA while processing the string $abaab$ using the extended transition function?”.



Solution: The moves made by the DFA for the input string $abaab$ using δ^* is obtained starting from ϵ and taking prefix of $abaab$ in increasing size as shown below:

$$\begin{aligned}
 \bullet \text{ For prefix } \epsilon: & \quad \delta^*(q_0, \epsilon) = \{q_0\} & \dots \dots \dots (1) \\
 \bullet \text{ For prefix } a: & \quad \delta^*(q_0, a) = \delta(\delta^*(q_0, \epsilon), a) & \text{Substituting (3)} \\
 & = \delta(q_0, a) \\
 & = \{q_0, q_1\} & \dots \dots \dots (2)
 \end{aligned}$$

- For prefix ab: $\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b)$ Substituting (2)
 $= \delta([q_0, q_1], b)$
 $= \delta(q_0, b) \cup \delta(q_1, b)$
 $= \{q_0\} \cup \{q_1\} \emptyset$
 $= \{q_0, q_1\}$ (3)
- For prefix aba: $\delta^*(q_0, aba) = \delta(\delta^*(q_0, ab), a)$ Substituting (3)
 $= \delta([q_0, q_2], a)$
 $= \delta(q_0, a) \cup \delta(q_2, a)$
 $= \{q_0, q_1\} \cup \emptyset$
 $= \{q_0, q_1\}$ (4)
- For prefix abaa: $\delta^*(q_0, abaa) = \delta(\delta^*(q_0, aba), a)$ Substituting (3)
 $= \delta([q_0, q_1], a)$
 $= \delta(q_0, a) \cup \delta(q_1, a)$
 $= \{q_0, q_1\} \cup \emptyset$
 $= \{q_0, q_1\}$ (5)
- For prefix abaab: $\delta^*(q_0, abaab) = \delta(\delta^*(q_0, abaa), b)$ Substituting (4)
 $= \delta([q_0, q_1], b)$
 $= \delta(q_0, b) \cup \delta(q_1, b)$
 $= \{q_0\} \cup \{q_2\}$
 $= \{q_0, q_2\}$

Note: After the string abaab, the states of NFA = {q₀, q₂}. Since the possible states of NFA after consuming abaab has one final state, the string abaab is accepting by the machine.

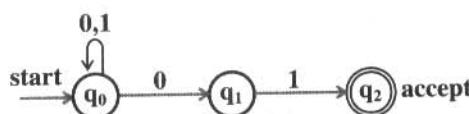
1.10.3. Language Accepted by a NFA

Now, let us “Define the language accepted by NFA”. The language accepted by NFA is formally be defined as follows:

❖ **Definition:** Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA. A string w is accepted by the machine M , if it takes the initial state q_0 to final state, i.e., $\delta^*(q_0, w)$ is in F . Thus, the languages accepted by NFA represented as $L(M)$ can be formally written as:

$$L(M) = \{w \mid w \in \Sigma^* \text{ and } \delta^*(q_0, w) \text{ is in } F\}$$

Now, let us “Prove formally that the following NFA accepts the language $L = \{w: w \text{ ends in } 01\}$ ”.



Proof: It is given that the above NFA accepts the language:

$$L = \{w: w \text{ ends in } 01\}$$

The statement can be proved by mutual induction on the length of string w accepted by the three states shown in three cases as shown below:

- Case 1: $\delta^*(q_0, w) = q_0$ for every w
- Case 2: $\delta^*(q_0, w) = q_1$ for every w ending in 0
- Case 3: $\delta^*(q_0, w) = q_2$ for every w ending in 01

Basis: Consider a string $w = \epsilon$ where $|w| = 0$.

Case 1: $\delta(q_0, \epsilon) = q_0$ by definition. Hence case 1 is proved.

Case 2: $\delta(q_0, \epsilon) = q_0$ by definition. Here, ϵ do not end with 0 and hence $\delta(q_0, \epsilon)$ does not contain q_1 . Hence case 2 is proved.

Case 3: $\delta(q_0, \epsilon) = q_0$ by definition. Here, ϵ do not end with 01 and hence $\delta(q_0, \epsilon)$ does not contain q_2 . Hence case 3 is proved.

Induction hypotheses: Let $w = xa$ where a is either 0 or 1 and assume the three statements case 1 through case 2 holds good for x .

Let $|x| = n$. So, $|w| = n+1$

We have to prove that the three cases mentioned above are true for $n + 1$.

Proof: The proof for all the three cases can be obtained as shown below:

Case 1: $\delta^*(q_0, w) = q_0$. This statement is true. Because, on any of the input symbols 0 and 1, the NFA may stay in q_0 . Hence, case 1 is proved.

Case 2: Assume w ends in 0, i.e., $a = 0$. We proved that $\delta^*(q_0, w) = q_0$. So, for the string x we can write $\delta^*(q_0, x) = q_0$. Since there is a transition from q_0 to q_1 on 0, we conclude that $\delta^*(q_0, x) = q_1$. Hence, case 2 is proved.

Case 3: Assume w ends in 01. Let $w = xa$ where $a = 1$ and x is a string such that $\delta^*(q_0, x) = q_1$ where the string ends with 0. Since there is a transition from q_1 to q_2 on 1, we conclude that $\delta^*(q_0, w) = q_2$. Hence, case 3 is proved.

Note: Thus, the different properties of the transition function with respect to NFA are:

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

$$\delta(q, wa) = \delta(\delta^*(q, w), a) = P_j$$

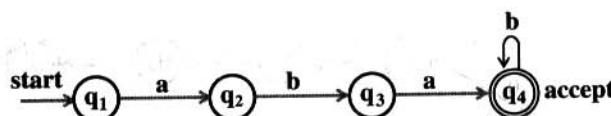
$$\delta(q, aw) = \delta^*(\delta(q, a), w) = P_q$$

where q is in Q , a is in Σ , w is in Σ^* and P_j and P_q are the set of states which are reachable from q .

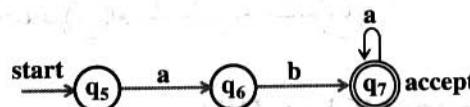
Example 1: Obtain an NFA to accept the following language:

$$L = \{w \mid w \in abab^n \text{ or } aba^n \text{ where } n \geq 0\}$$

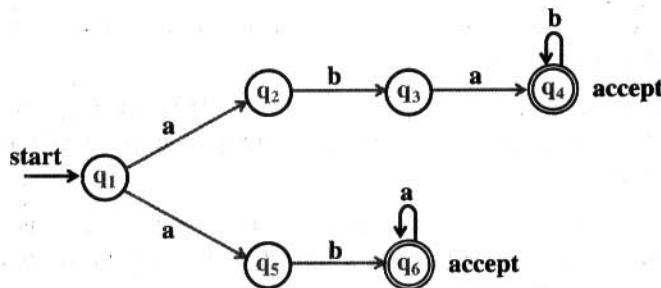
Solution: The machine to accept $abab^n$ where $n \geq 0$ is shown below:



The machine to accept aba^n where $n \geq 0$ is shown below:

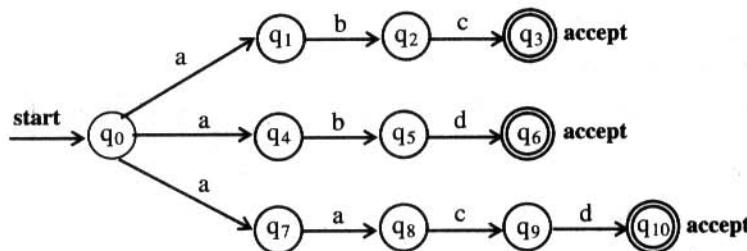


Since both the machines accept a as the first input symbol, the states q_1 and q_5 can be merged into a single state and the machine to accept either $abab^n$ or aba^n where $n \geq 0$ is shown below:



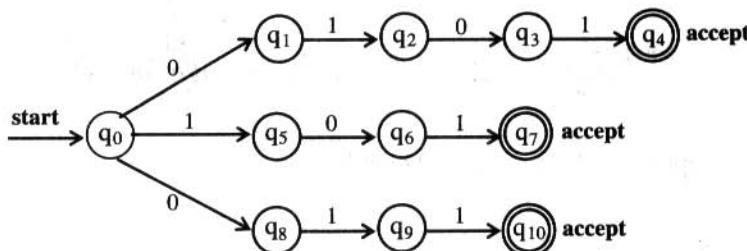
Example 1: Design an NFA to recognize the following set of strings abc, abd and aacd.

Solution: An NFA to recognize the following set of strings abc, abd and aacd is shown below:



Example 2: Obtain an NFA to recognize the following set of strings 0101, 101 and 011.

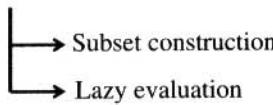
Solution: An NFA to recognize the following set of strings 0101, 101 and 011 is shown below:



1.11. Conversion from NFA to DFA

Digital computers are deterministic machines. Given the input, the state of the machine is predictable. Sometimes, constructing DFA is difficult compared to NFA. So, given any problem we construct

a NFA. This is an efficient mechanism to describe some complicated languages concisely. Practically, non-deterministic machines will not exist. So, we convert an NFA in to a DFA. Now, let us see “What are the methods using which an NFA can be converted into a DFA?”. The NFA can be converted into DFA using two methods:



1.11.1. Conversion from NFA to DFA (Subset Construct Method)

Now, let us “Describe the subset construction procedure to convert an NFA into a DFA”

Procedure NFA_DFA: Given an NFA $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ which accepts the language $L(M_N)$, we can find an equivalent DFA $M_D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ such that $L(M_D) = L(M_N)$.

Step 1: Identify the start state of DFA: Since q_0 is the start state of NFA, $\{q_0\}$ is the start state of DFA.

Step 2: Identify the alphabets of DFA: The input alphabets of DFA are the input alphabets of NFA. So, $\Sigma = \{a, b\}$.

Step 3: Identify Q_D which are the states of DFA: The set of subsets of Q_N will be the states of DFA Q_D . So, if Q_N has n states then Q_D will have 2^n states. For example,

Let $Q_N = \{q_0, q_1, q_2\}$ then $|Q_N| = 3$

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

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

So, $|Q_D| = 8$

Note: In this example, the number of states of NFA = 3 and hence number of states of DFA = 8. If n is number of states of NFA the number of states of DFA will be 2^n .

In general, $\{q_i, q_j, \dots, q_k\}$ is considered as a state in Q_D .

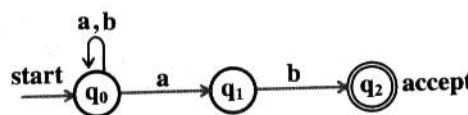
Step 4: Identify the final states of DFA: If $\{q_i, q_j, \dots, q_k\}$ is a state in Q_D , then $\{q_i, q_j, \dots, q_k\}$ will be final state of DFA provided one of q_i, q_j, \dots, q_k is the final state of NFA.

Step 5: Identify the transitions (i.e., δ_D) of DFA: For each state $\{q_i, q_j, \dots, q_k\}$ in Q_D and for each input symbol a in Σ , the transition can be obtained as shown below:

$$\delta_D(\{q_i, q_j, \dots, q_k\}, a) = \delta_N(q_i, a) \cup \delta_N(q_j, a) \cup \dots \cup \delta_N(q_k, a)$$

Thus, DFA can be obtained using subset construction method.

Example: Now, let us “Obtain the DFA for the following NFA using subset construction method”.



Solution: The transition table for the above DFA can be written as shown below:

δ	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset

Step 1: Identify the start state of DFA: Since q_0 is the start state of NFA, $\{q_0\}$ is the start state of DFA.

Step 2: Identify the alphabets of DFA: The input alphabets of NFA are the input alphabets of DFA. So, $\Sigma = \{a, b\}$.

Step 3: Identify Q_D which are the states of DFA: Here, $QN = \{q_0, q_1, q_2\}$. Its subsets are the states of DFA. So, states of DFA are:

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

Step 4: Identify the final states of DFA: Since q_2 is the final state of NFA in the above set, wherever q_2 is present as an element, the corresponding set is the final state of DFA. So

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

Step 5: Identify the transitions (i.e., δ_D) of DFA: Obtain the transitions for each of the states of Q_D obtained in step 3 as shown below:

For state \emptyset :

Input symbol = a

$$\delta_D(\emptyset, a) = \emptyset$$

Input symbol = b

$$\delta_D(\emptyset, b) = \emptyset$$

For state $\{q_0\}$:

Input symbol = a

$$\delta_D(\{q_0\}, a) = \{q_0, q_1\}$$

Input symbol = b

$$\delta_D(\{q_0\}, b) = \{q_0\}$$

For state $\{q_1\}$:

Input symbol = a

$$\delta_D(\{q_1\}, a) = \emptyset$$

Input symbol = b

$$\delta_D(\{q_1\}, b) = q_2$$

For state $\{q_2\}$:

Input symbol = a

$$\delta_D(\{q_2\}, a) = \emptyset$$

Input symbol = b

$$\delta_D(\{q_2\}, b) = \emptyset$$

For state $\{q_0, q_1\}$:

Input symbol = a

$$\delta_D(\{q_0, q_1\}, a) = \delta_N(\{q_0, q_1\}, a)$$

$$= \delta_N(q_0, a) \cup \delta_N(q_1, a)$$

$$= \{q_0, q_1\} \cup \emptyset$$

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

Input symbol = b

$$\delta_D(\{q_0, q_1\}, b) = \delta_N(\{q_0, q_1\}, b)$$

$$= \delta_N(q_0, b) \cup \delta_N(q_1, b)$$

$$= \{q_0\} \cup \{q_2\}$$

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

For state $\{q_0, q_2\}$:

Input symbol = a

$$\delta_D(\{q_0, q_2\}, a) = \delta_N(\{q_0, q_2\}, a)$$

$$= \delta_N(q_0, a) \cup \delta_N(q_2, a)$$

$$= \{q_0, q_1\} \cup \emptyset$$

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

Input symbol = b

$$\begin{aligned}\delta_D(\{q_0, q_2\}, b) &= \delta_N(\{q_0, q_2\}, b) \\ &= \delta_N(q_0, b) \cup \delta_N(q_2, b) \\ &= \{q_0\} \cup \emptyset \\ &= \{q_0\}\end{aligned}$$

For state {q₁, q₂}:

Input symbol = a

$$\begin{aligned}\delta_D(\{q_1, q_2\}, a) &= \delta_N(\{q_1, q_2\}, a) \\ &= \delta_N(q_1, a) \cup \delta_N(q_2, a) \\ &= \emptyset \cup \emptyset \\ &= \emptyset\end{aligned}$$

Input symbol = b

$$\begin{aligned}\delta_D(\{q_1, q_2\}, b) &= \delta_N(\{q_1, q_2\}, b) \\ &= \delta_N(q_1, b) \cup \delta_N(q_2, b) \\ &= \{q_2\} \cup \emptyset \\ &= \{q_2\}\end{aligned}$$

For state {q₀, q₁, q₂}:

Input symbol = a

$$\begin{aligned}\delta_D(\{q_0, q_1, q_2\}, a) &= \delta_N(\{q_0, q_1, q_2\}, a) \\ &= \delta_N(q_0, a) \cup \delta_N(q_1, a) \cup \delta_N(q_2, a) \\ &= \{q_0, q_1\} \cup \emptyset \cup \emptyset \\ &= \{q_0, q_1\}\end{aligned}$$

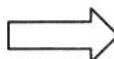
Input symbol = b

$$\begin{aligned}\delta_D(\{q_0, q_1, q_2\}, b) &= \delta_N(\{q_0, q_1, q_2\}, b) \\ &= \delta_N(q_0, b) \cup \delta_N(q_1, b) \cup \delta_N(q_2, b) \\ &= \{q_0\} \cup \{q_2\} \cup \emptyset \\ &= \{q_0, q_2\}\end{aligned}$$

Now, all the above transitions can be represented using transition table as shown below:

δ	a	b
ϕ	ϕ	ϕ
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	ϕ	$\{q_2\}$
$*\{q_2\}$	ϕ	ϕ
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	ϕ	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

By renaming the states of DFA as
A, B, C, D, E, F, G, H



δ	a	b
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

Note: Even though we have 8 states, observe from the table that B is the start state. The states reachable from B are B, E, F. The rest of the states are not reachable and hence can be eliminated.

1.11.2. Disadvantage of Subset Construction Method

Now, let us see “What are the disadvantages of subset construction method?”. Observe that from the above table that there are 2^n states and from each state we have the transition from input symbol in Σ and hence the time complexity to convert an NFA to DFA is $\Sigma * 2^n$. Since the time complexity is exponential, the procedure takes very long time to construct the table. This exponential time complexity can be avoided using another technique called “Lazy evaluation” on the subsets.

1.11.3. Conversion from NFA to DFA (Lazy Evaluation Method)

Now, let us “Describe the lazy evaluation method to convert NFA into a DFA”. The lazy evaluation method of obtaining a DFA from NFA is shown below:

Procedure NFA_DFA: Given an NFA $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ which accepts the language $L(M_N)$, we can find an equivalent DFA $M_D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(M_D) = L(M_N)$.

Step 1: Identify the start state of DFA: Since q_0 is the start state of NFA, $\{q_0\}$ is the start state of DFA.

Step 2: Identify the alphabets of DFA: The input alphabets of DFA are the input alphabets of NFA. So, $\Sigma = \{a, b\}$.

Step 3: Identify the transitions (i.e., δ_D) of DFA: For each state $\{q_i, q_j, \dots, q_k\}$ in Q_D and for each input symbol a in Σ , the transition can be obtained as shown below:

$$\delta_D(\{q_i, q_j, \dots, q_k\}, a) = \delta_N(q_i, a) \cup \delta_N(q_j, a) \cup \dots \cup \delta_N(q_k, a)$$

$$= [q_l, q_m, \dots, q_n] \text{ say}$$

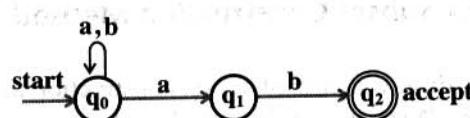
- Add the state $[q_l, q_m, \dots, q_n]$ to Q_D , if it is not already in Q_D .
- Add the transitions from $[q_i, q_j, \dots, q_k]$ to $[q_l, q_m, \dots, q_n]$ on the input symbol a

Note: The step 3 has to be repeated for each state that is added to Q_D .

Step 4: Identify the final states of DFA: If $\{q_i, q_j, \dots, q_k\}$ is a state in Q_D and if one of q_i, q_j, \dots, q_k is the final state of NFA, then $\{q_i, q_j, \dots, q_k\}$ will be the final state of DFA.

Thus, DFA can be obtained using lazy evaluation method.

Example: Now, let us “Obtain the DFA for the following NFA using lazy evaluation method”.



Solution: The transition table for the above DFA can be written as shown below:

δ	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset

Step 1: Identify the start state of DFA: Since q_0 is the start state of NFA, $\{q_0\}$ is the start state of DFA.

Step 2: Identify the alphabets of DFA: The input alphabets of NFA are the input alphabets of DFA. So, $\Sigma = \{a, b\}$.

Step 3: Identify the transitions (i.e., δ_D) of DFA: Start from the start state q_0 and find the transitions as shown below:

For state $\{q_0\}$:

Input symbol = a

$$\delta_D(\{q_0\}, a) = \{q_0, q_1\}$$

Input symbol = b

$$\delta_D(\{q_0\}, b) = \{q_0\}$$

For state $\{q_0, q_1\}$:

Input symbol = a

$$\delta_D(\{q_0, q_1\}, a) = \delta_N(\{q_0, q_1\}, a)$$

$$= \delta_N(q_0, a) \cup \delta_N(q_1, a)$$

$$= \{q_0, q_1\} \cup \emptyset$$

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

Input symbol = b

$$\delta_D(\{q_0, q_1\}, b) = \delta_N(\{q_0, q_1\}, b)$$

$$= \delta_N(q_0, b) \cup \delta_N(q_1, b)$$

$$= \{q_0\} \cup \{q_2\}$$

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

For state $\{q_0, q_2\}$:

Input symbol = a

$$\delta_D(\{q_0, q_2\}, a) = \delta_N(\{q_0, q_2\}, a)$$

$$= \delta_N(q_0, a) \cup \delta_N(q_2, a)$$

$$= \{q_0, q_1\} \cup \emptyset$$

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

Input symbol = b

$$\delta_D(\{q_0, q_2\}, b) = \delta_N(\{q_0, q_2\}, b)$$

$$= \delta_N(q_0, b) \cup \delta_N(q_2, b)$$

$$= \{q_0\} \cup \emptyset$$

$$= \{q_0\}$$

Since, no new state is generated this step is terminated.

Step 4: Identify the final states of DFA: Since q_2 is the final state of NFA in the above set, wherever q_2 is present as an element, the corresponding set is the final state of DFA. So, the final state is $\{q_0, q_2\}$.

Now, all the above transitions can be represented using transition table as shown below:

δ	a	b
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

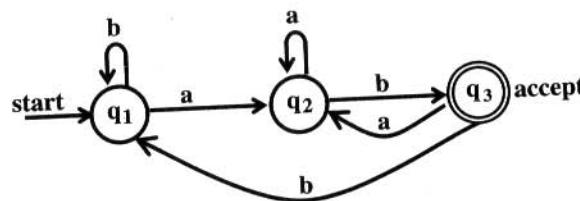
By renaming the states of DFA as
A, B, C



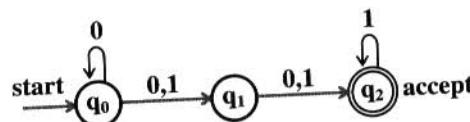
δ	a	b
A	B	A
B	B	C
*C	B	A

So, the final DFA is given by $M = (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{A, B, C\}$
- $\Sigma = \{a, b\}$
- $q_0 = A$
- $F = \{C\}$
- δ is shown below using the transition table:



Example: Now, let us “convert the following NFA to its equivalent DFA”.



Solution: The transition table for the above DFA can be written as shown below:

δ	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_1\}$
$*q_1$	$\{q_2\}$	$\{q_2\}$
q_2	\emptyset	$\{q_2\}$

Step 1: Identify the start state of DFA: Since q_0 is the start state of NFA, $\{q_0\}$ is the start state of DFA.

Step 2: Identify the alphabets of DFA: The input alphabets of NFA are the input alphabets of DFA. So, $\Sigma = \{0, 1\}$.

Step 3: Identify the transitions (i.e., δ_D) of DFA: Start from the start state q_0 and find the transitions as shown below:

For state $\{q_0\}$:

Input symbol = 0

$$\begin{aligned}\delta_D(\{q_0\}, 0) &= \delta_N(\{q_0\}, 0) \\ &= \{q_0, q_1\}\end{aligned}$$

Input symbol = 1

$$\begin{aligned}\delta_D(\{q_0\}, 1) &= \delta_N(\{q_0\}, 1) \\ &= \{q_1\}\end{aligned}$$

For state $\{q_0, q_1\}$:

Input symbol = 0

$$\begin{aligned}\delta_D(\{q_0, q_1\}, 0) &= \delta_N(\{q_0, q_1\}, 0) \\ &= \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \\ &= \{q_0, q_1\} \cup \{q_2\} \\ &= \{q_0, q_1, q_2\}\end{aligned}$$

Input symbol = 1

$$\begin{aligned}\delta_D(\{q_0, q_1\}, 1) &= \delta_N(\{q_0, q_1\}, 1) \\ &= \delta_N(q_0, 1) \cup \delta_N(q_1, 1) \\ &= \{q_1\} \cup \{q_2\} \\ &= \{q_1, q_2\}\end{aligned}$$

The above two states are added to Q_D shown in previous step. The resulting states are shown below:

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

For state $\{q_1\}$:

Input symbol = 0

$$\begin{aligned}\delta_D(\{q_1\}, 0) &= \delta_N(\{q_1\}, 0) \\ &= \{q_2\}\end{aligned}$$

Input symbol = 1

$$\begin{aligned}\delta_D(\{q_1\}, 1) &= \delta_N(\{q_1\}, 1) \\ &= \{q_2\}\end{aligned}$$

The above two states are added to Q_D obtained in previous step so that

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

For state $\{q_0, q_1, q_2\}$:

Input symbol = 0

$$\begin{aligned}\delta_D(\{q_0, q_1, q_2\}, 0) &= \delta_N(\{q_0, q_1, q_2\}, 0) \\ &= \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \cup \delta_N(q_2, 0) \\ &= \{q_0, q_1\} \cup \{q_2\} \cup \{\phi\} \\ &= \{q_0, q_1, q_2\}\end{aligned}$$

Input symbol = 1

$$\begin{aligned}\delta_D(\{q_0, q_1, q_2\}, 1) &= \delta_N(\{q_0, q_1, q_2\}, 1) \\ &= \delta_N(q_0, 1) \cup \delta_N(q_1, 1) \cup \delta_N(q_2, 1) \\ &= \{q_1\} \cup \{q_2\} \cup \{q_2\} \\ &= \{q_1, q_2\}\end{aligned}$$

The above two states are added to Q_D obtained in previous step so that

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

For state $\{q_1, q_2\}$:

Input symbol = 0

$$\begin{aligned}\delta_D(\{q_1, q_2\}, 0) &= \delta_N(\{q_1, q_2\}, 0) \\ &= \delta_N(q_1, 0) \cup \delta_N(q_2, 0) \\ &= \{q_2\} \cup \phi \\ &= \{q_2\}\end{aligned}$$

Input symbol = 1

$$\begin{aligned}\delta_D(\{q_1, q_2\}, 1) &= \delta_N(\{q_1, q_2\}, 1) \\ &= \delta_N(q_1, 1) \cup \delta_N(q_2, 1)\end{aligned}$$

$$\begin{aligned}
 &= \{q_2\} \cup \{q_2\} \\
 &= \{q_2\}
 \end{aligned}$$

The above two states are added to Q_D obtained in previous step so that

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

For state $\{q_2\}$:

Input symbol = 0

$$\begin{aligned}
 \delta_D(\{q_2\}, 0) &= \delta_N(\{q_2\}, 0) \\
 &= \emptyset
 \end{aligned}$$

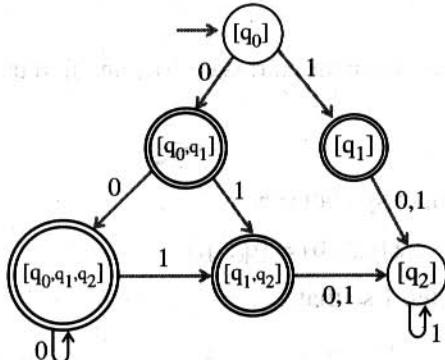
Input symbol = 1

$$\begin{aligned}
 \delta_D(\{q_2\}, 1) &= \delta_N(\{q_2\}, 1) \\
 &= \{q_2\}
 \end{aligned}$$

The above two states are added to Q_D obtained in previous step so that

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

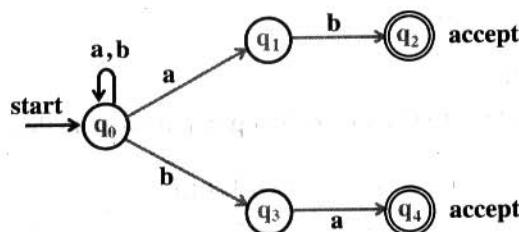
The final transition table along with transition diagram is shown below:



δ	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_1\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
$\{q_1\}$	$\{q_2\}$	$\{q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
$*\{q_1, q_2\}$	$\{q_2\}$	$\{q_2\}$
$\{q_2\}$	\emptyset	$\{q_0\}$

Example: now, let us “Obtain an NFA to accept strings of a’s and b’s ending with ab or ba. From this obtain an equivalent DFA”.

Solution: The NFA to accept strings of a’s and b’s ending ab or ba is shown below:



The transition table for the above transition diagram is shown below:

δ	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0, q_3\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset
q_3	$\{q_4\}$	\emptyset
$*q_4$	\emptyset	\emptyset

Step 1: Identify the start state of DFA: Since q_0 is the start state of NFA, $\{q_0\}$ is the start state of DFA. So, $Q_D = \{ \{q_0\} \}$.

Step 2: Identify the alphabets of DFA: The input alphabets of NFA are the input alphabets of DFA. So, $\Sigma = \{a, b\}$.

Step 3: Identify the transitions (i.e., δ_D) of DFA: Start from the start state $\{q_0\}$ and find the transitions as shown below:

For state $\{q_0\}$:

Input symbol = a

$$\delta_D(\{q_0\}, a) = \{q_0, q_1\}$$

Input symbol = b

$$\delta_D(\{q_0\}, b) = \{q_0, q_3\}$$

The above two states are added to Q_D obtained in step 1 so that

$$Q_D = \{ \{q_0\}, \{q_0, q_1\}, \{q_0, q_3\} \}$$

For state $\{q_0, q_1\}$:

Input symbol = a

$$\delta_D(\{q_0, q_1\}, a) = \delta_N(\{q_0, q_1\}, a)$$

$$= \delta_N(q_0, a) \cup \delta_N(q_1, a)$$

$$\begin{aligned}
 &= \{q_0, q_1\} \cup \emptyset \\
 &= \{q_0, q_1\}
 \end{aligned}$$

Input symbol = b

$$\begin{aligned}
 \delta_D(\{q_0, q_1\}, b) &= \delta_N(\{q_0, q_1\}, b) \\
 &= \delta_N(q_0, b) \cup \delta_N(q_1, b) \\
 &= \{q_0, q_3\} \cup \{q_2\} \\
 &= \{q_0, q_2, q_3\}
 \end{aligned}$$

The above two states are added to Q_D so that

$$Q_D = \{ \{q_0\}, \{q_0, q_1\}, \{q_0, q_3\}, \{q_0, q_2, q_3\} \}$$

For state $\{q_0, q_3\}$:

$$\begin{aligned}
 \text{Input symbol} &= a \\
 \delta_D(\{q_0, q_3\}, a) &= \delta_N(\{q_0, q_3\}, a) \\
 &= \delta_N(q_0, a) \cup \delta_N(q_3, a) \\
 &= \{q_0, q_3\} \cup \{q_2\} \\
 &= \{q_0, q_2, q_3\}
 \end{aligned}$$

Input symbol = b

$$\begin{aligned}
 \delta_D(\{q_0, q_3\}, b) &= \delta_N(\{q_0, q_3\}, b) \\
 &= \delta_N(q_0, b) \cup \delta_N(q_3, b) \\
 &= \{q_0, q_3\} \cup \emptyset \\
 &= \{q_0, q_3\}
 \end{aligned}$$

The above two states are added to Q_D so that

$$Q_D = \{ \{q_0\}, \{q_0, q_1\}, \{q_0, q_3\}, \{q_0, q_2, q_3\}, \{q_0, q_1, q_4\} \}$$

On similar lines, the reader is supposed to find the transitions for other states specified in Q_D . The reader is advised to verify the following answers:

$$\begin{aligned}
 \delta_D(\{q_0, q_2, q_3\}, a) &= \{q_0, q_1, q_4\} \\
 \delta_D(\{q_0, q_2, q_3\}, b) &= \{q_0, q_3\} \\
 \delta_D(\{q_0, q_1, q_3\}, a) &= \{q_0, q_1\} \\
 \delta_D(\{q_0, q_1, q_4\}, a) &= \{q_0, q_2, q_3\}
 \end{aligned}$$

The final DFA obtained along with transition diagram and transition table is shown below:

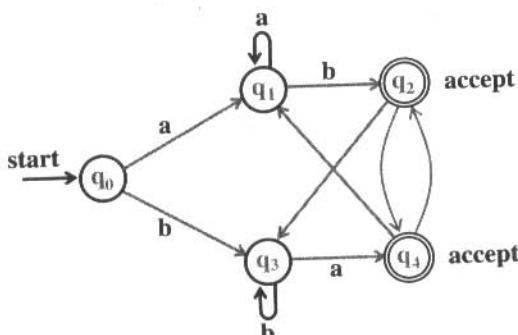
δ	a	b
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0, q_3\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2, q_3\}$
$\{q_0, q_3\}$	$\{q_0, q_1, q_4\}$	$\{q_0, q_3\}$
$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_4\}$	$\{q_0, q_3\}$
$\{q_0, q_2, q_4\}$	$\{q_0, q_1\}$	$\{q_0, q_2, q_3\}$

The states of the above DFA are:

$$\{q_0\}, \{q_0, q_1\}, \{q_0, q_2, q_3\}, \{q_0, q_3\}, \{q_0, q_1, q_4\}$$

By renaming q_0, q_1, q_2, q_3, q_4

The final transition diagram and transition table are shown below:



Transition diagram

δ	a	b
$\rightarrow q_0$	q_1	q_3
q_1	q_1	q_2
$*q_2$	q_4	q_3
q_3	q_4	q_3
$*q_4$	q_1	q_2

Transition table

Now, it is observed that for every NFA there exists some DFA that accepts the same language as accepted by NFA. Now, let us "Formally prove that every NFA N can be converted into a DFA M such that $L(D) = L(M)$ ".

Theorem: If there exists NFA $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ which accepts the language $L(M_N)$, there exists an equivalent DFA $M_D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(M_D) = L(M_N)$.

Proof: It is required to prove that

$$\delta^*_D(q_0, w) = \delta^*_N(q_0, w)$$

We know that if Q_N represents states of NFA then power set of Q_N which contains the set of subsets of set Q_N are the states of DFA denoted by Q_D . The DFA interprets each set as a single state in DFA.

Basis: Consider a string $w = \epsilon$ where $|w| = 0$

$$\delta_D^*(\{q_0\}, \epsilon) = \{q_0\} \text{ by definition of extended transition function of DFA}$$

$$\delta_N^*(\{q_0\}, \epsilon) = \{q_0\} \text{ by definition of extended transition function of NFA}$$

Hence $\delta_D^*(q_0, w) = \delta_N^*(q_0, w)$ is proved when $w = \epsilon$.

Induction hypotheses: Now, let us assume that

$$\delta_D^*(q_0, w) = \delta_N^*(q_0, w) \text{ for some } w \text{ where } |w| = n$$

Now, it is required to prove that the statement:

$$\delta_D^*(q_0, w) = \delta_N^*(q_0, w) \text{ is true for some } w \text{ where } |w| = n + 1$$

Inductive proof: Let $w = xa$ where a is the last symbol of w and x is the remaining string of w :

$$\text{So, } |x| = n \text{ and } |xa| = |w| = n + 1$$

By extended definition δ^* of NFA, we know that:

$$\begin{aligned} \delta_N^*(q_0, w) &= \delta_N^*(q_0, xa) \\ &= \delta_N(\underbrace{\delta_N^*(q_0, x)}, a) \end{aligned} \tag{1}$$

Now, x is the string to be processed and after consuming the string x , let the states of the machine be $\{p_i, p_j, \dots, p_k\}$

$$\text{i.e., } \delta_N^*(q_0, x) = \{p_i, p_j, \dots, p_k\}$$

Substituting this in Eq. (1), we have

$$\begin{aligned} \delta_N^*(q_0, w) &= \delta_N(\{p_i, p_j, \dots, p_k\}, a) \\ &= \delta_N(p_i, a) \cup \delta_N(p_j, a) \cup \dots \cup \delta_N(p_k, a) \end{aligned} \tag{2}$$

By extended definition δ^* of DFA, we know that

$$\begin{aligned} \delta_D^*(q_0, w) &= \delta_D^*(q_0, xa) \\ &= \delta_D(\underbrace{\delta_N^*(q_0, x)}, a) \end{aligned} \tag{3}$$

Now, x is the string to be processed and after consuming the string x , let the states of the machine be $\{p_i, p_j, \dots, p_k\}$.

$$\text{i.e., } \delta^*_D(q_0, x) = \{p_i, p_j, \dots, p_k\}$$

Substituting this in Eq. (3), we have

$$\begin{aligned} \delta^*_D(q_0, w) &= \delta_D(\{p_i, p_j, \dots, p_k\}, a) \\ &= \delta_N(p_i, a) \cup \delta_N(p_j, a) \cup \dots \cup \delta_N(p_k, a) \end{aligned} \quad (4)$$

By comparing Eqs. (2) and (4), we have

$$\delta^*_D(\{q_0\}, w) = \delta^*_N(\{q_0\}, w)$$

So, if $\delta^*_D(\{q_0\}, w)$ is in F_N and $\delta^*_N(\{q_0\}, w)$ is in F_N , then both enters into final state accepting the same language. Thus, $L(M_N) = L(M_D)$. Hence, the proof.

Theorem: Now, let us “Prove that a language L is accepted by some DFA if and only if L is accepted by some NFA”.

Proof: The above statement can be proved as shown below:

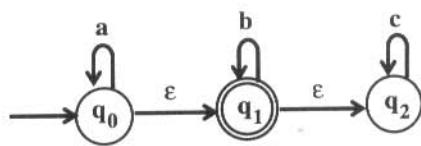
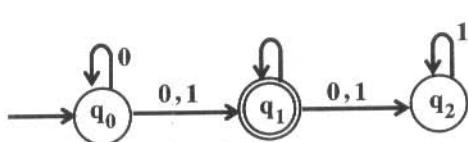
Note: Write the subset construction method of converting an NFA to DFA + proof of previous theorem.

Since in the subset construction, all the transitions defined for NFA are also defined for DFA, the language accepted by NFA is same as the language accepted by DFA. So, w is accepted by DFA M_D if and only if w is accepted by M_N , i.e., $L(M_N) = L(M_D)$. Hence the proof.

Exercises

1. What is an NFA? Explain with example.
2. What is the need for an NFA?
3. What is the difference between DFA and NFA?
4. Give a general procedure to convert an NFA to DFA.

5. Convert the following NFA into an equivalent DFA.



6. Draw an NFA to accept the string of a's and b's such that it can accept either the string consisting of one a followed by any number of a's or one b followed by any number of b's (i.e. $aa^* \mid bb^*$) and obtain the corresponding DFA.

Chapter 2

Finite Automata and Regular Expressions

What are we studying in this chapter . . .

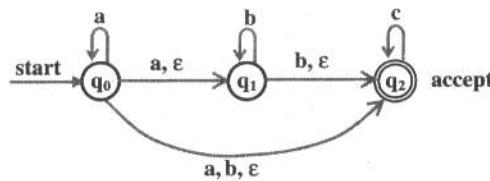
- ▶ *An application of finite automata*
- ▶ *Finite automata with ϵ -transitions*
- ▶ *Regular expressions*
- ▶ *Finite automata and regular expressions*
- ▶ *Applications of regular expressions*

2.1. ϵ -NFA (Finite Automata with Epsilon Transitions)

In this section, let us see the extended model of NFA called ϵ -NFA. An NFA with zero or more ϵ -transitions is called an ϵ -NFA. Now, let us see “What is an ϵ -transition?”

❖ **Definition:** A transition with an empty input string is called an ϵ -transition (read as epsilon transition). That is, if there is a transition from one state to another state without any input (i.e.,

no input implies empty string denoted by ϵ) is called ϵ -transition. For example, consider the finite automaton shown below:



From above FA, observe the following points:

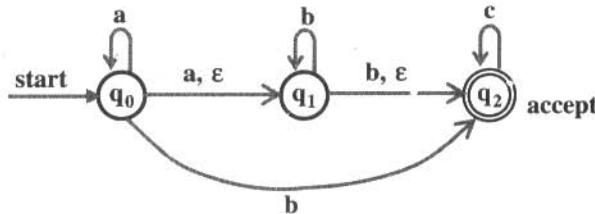
- In the above FA, we have more than one transition from state q_0 with input symbol a . So, it is an NFA.
- There is a transition from state q_0 to q_1 with ϵ as the input. There is another transition from state q_1 to q_2 with ϵ as the input. So, the above FA is an NFA with ϵ -transitions and hence it is an ϵ -NFA.

Note: If zero or more ϵ -transitions are present in a FA then it is an ϵ -NFA.

Note: If there is an ϵ -transition, then NFA makes transition without receiving any input symbol.

ϵ -NFA

Before worrying about the definition, let us consider the following pictorial representation of ϵ -NFA.



From the above figure, observe following components of ϵ -NFA:

- States:** The ϵ -NFA has three states q_0 , q_1 and q_2 and can be represented as

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

Note: The power set of Q is denoted by 2^Q which is set of subsets of set Q . This is denoted as shown below:

$$2^Q = \{ \{ \}, \{q_0\}, \{q_1\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\} \}$$

- Input alphabets:** Each edge is labeled with a or b and represent the input alphabets which can be denoted as:

$$\Sigma = \{a, b\}$$

- Transitions:** Transition is nothing but change of state after consuming an input symbol. If there is a change of state from q_i to q_j on an input symbol a , then we write

$$\delta(q_i, a) = q_j$$

If there is a change of state from q_i to q_j and q_i to q_k on an input symbol a , then we write

$$\delta(q_i, a) = \{q_j, q_k\}$$

If there is a change of state from q_i to q_j on ϵ (no input is read), then we write

$$\delta(q_i, \epsilon) = \{q_j\}$$

The transitions for above ϵ -NFA is shown below:

Current State	Input	Next state	Representation
q_0	a	$\{q_0, q_1\}$	$\delta(q_0, a) = \{q_0, q_1\}$
q_0	b	$\{q_2\}$	$\delta(q_0, b) = q_2$
q_0	c	\emptyset	$\delta(q_0, c) = \emptyset$
q_0	ϵ	$\{q_1\}$	$\delta(q_0, \epsilon) = \{q_1\}$
q_1	a	\emptyset	$\delta(q_1, a) = \emptyset$
q_1	b	$\{q_1, q_2\}$	$\delta(q_1, b) = \{q_1, q_2\}$
q_1	c	\emptyset	$\delta(q_1, c) = \emptyset$
q_1	ϵ	$\{q_2\}$	$\delta(q_1, \epsilon) = \{q_2\}$
q_2	a	\emptyset	$\delta(q_2, a) = \emptyset$
q_2	b	\emptyset	$\delta(q_2, b) = \emptyset$
q_2	c	$\{q_2\}$	$\delta(q_2, c) = \{q_2\}$
q_2	ϵ	\emptyset	$\delta(q_2, \epsilon) = \emptyset$

$$\downarrow \downarrow \downarrow \downarrow \downarrow \downarrow$$

$\delta: Q \times (\Sigma \cup \epsilon) \rightarrow 2^Q$ [power set]

Now, let us see “What is a transition function for ϵ -NFA?”. The transition function δ is defined as:

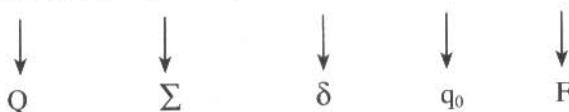
$$\delta: Q \times (\Sigma \cup \epsilon) \rightarrow 2^Q$$

which is read as “ δ is a transition function which maps $Q \times (\Sigma \cup \epsilon)$ to 2^Q ”, i.e., the function accepts:

- a state q as the first parameter
- input symbol in Σ or ϵ as the second parameter and
- returns set of states the machine enters into.
- **Start state (q_0):** q_0 with the label start is treated as the start state.
- **Final state (q_2):** q_2 with two circles is treated as the final or accept state.

Note: From this discussion, it is observed that the ϵ -NFA has five components:

(states, input alphabets, transitions, start state, final states)



With this concept, now let us see “What is an ϵ -NFA?”.

❖ **Definition:** The ϵ -NFA is 5-tuple or quintuple indicating five components:

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

where

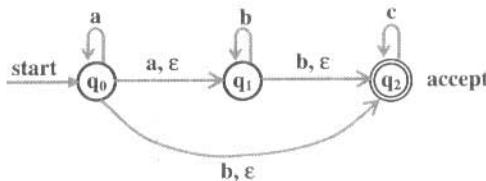
- Q is non-empty, finite set of states.
- Σ is non-empty, finite set of input alphabets.
- $\delta : Q \times (\Sigma \cup \epsilon) \rightarrow 2^Q$ i.e., δ is transition function which is a mapping from $Q \times (\Sigma \cup \epsilon)$ to 2^Q . Based on the current state there can be a transition to other states with or without any input symbols.
- $q_0 \in Q$ - is the start state.
- $F \subseteq Q$ - is set of accepting or final states.

Note: In an ϵ -NFA there can be zero, one or more transitions with or without any input symbol. Before proceeding further, let us see “What is ϵ -CLOSURE?”.

❖ **Definition:** The ϵ -CLOSURE of q denoted by $\text{ECLOSE}(q)$ is the set of all states which are reachable from q on ϵ -transitions only. It is recursively defined as shown below:

- State q is in $\text{ECLOSE}(q)$, i.e., $\text{ECLOSE}(q) = q$
- If $\text{ECLOSE}(q)$ contains p and if there is a transition from state p to state r labeled ϵ , then state r is also in $\text{ECLOSE}(q)$.

For example, the ϵ -CLOSURE(q) for each $q \in Q$ is shown below:



The states q_0 , q_1 and q_2 are reachable from q_0 without giving any input. So,
 $\text{ECLOSE}(q_0) = \{q_0, q_1, q_2\}$

- The states q_1 and q_2 are reachable from q_1 without giving any input. So,
 $\text{ECLOSE}(q_1) = \{q_1, q_2\}$
- The state q_2 is reachable from q_2 without giving any input. So,
 $\text{ECLOSE}(q_2) = \{q_2\}$

2.2. Extended Transition Function of ϵ -NFA to Strings

Now, let us see “What is extended transition function δ^* for ϵ -NFA?”.

❖ **Definition:** The extended transition function δ^* describes what happens to a state of machine when the input is a string (sequence of symbols). Let $M = (Q, \Sigma, \delta, q_0, F)$ be an ϵ -NFA. The extended transition function $\delta^*: Q \times (\Sigma \cup \epsilon)^* \rightarrow 2^Q$ is defined recursively as shown below:

- Basis: $\delta^*(q, \epsilon) = \text{ECLOSE}(q)$. This indicates that if the machine is in state q and read no input, then the machine is still in state q .
- Induction: Let $w = xa$ where a is the last symbol of w and x is the remaining string of w . Let q is the current state and x is the string to be processed and after consuming the string x , let the states of the machine is $\{p_1, p_2, p_3, \dots, p_m\}$:

$$\text{i.e., } \delta^*(q, x) = \{p_1, p_2, p_3, \dots, p_m\}$$

Let the transition from $\{p_1, p_2, p_3, \dots, p_m\}$ on input symbol a is

$$\begin{aligned} \delta(\{p_1, p_2, p_3, \dots, p_m\}, a) &= \delta(p_1, a) \cup \delta(p_2, a) \cup \delta(p_3, a) \dots \cup \delta(p_m, a) \\ &= \{r_1, r_2, r_3, \dots, r_m\} \end{aligned}$$

Now, $\delta^*(q, w) = \text{ECLOSE}(r_1, r_2, r_3, \dots, r_m)$

$$= \text{ELCOSE}(r_1) \cup \text{ECLOSE}(r_2) \cup \dots \cup \text{ECLOSE}(r_m)$$

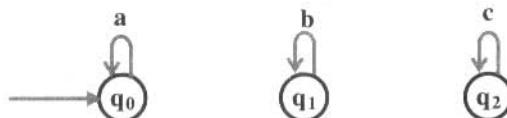
Note: Thus, various properties of extended transition functions for an ϵ -NFA can be:

- $\delta^*(q, \epsilon) = \text{ECLOSE}(q)$

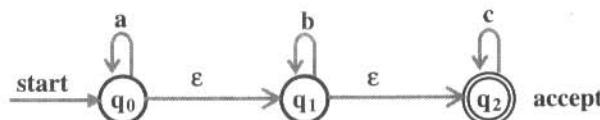
- $\delta^*(q, w) = \delta^*(q, xa) = \text{ECLOSE}(\delta(\delta^*(q, x), a))$ where $w = xa$
- $\delta^*(q, w) = \delta^*(q, ax) = \text{ECLOSE}(\delta^*(\delta(q, a), x))$ where $w = ax$

Example 1: Now, let us “Obtain an ϵ -NFA which accepts strings consisting of zero or more a’s followed by zero or more b’s followed by zero or more c’s”.

Solution: The ϵ -NFA that accepts strings of zero or more a’s, zero or more b’s and zero or more c’s can be represented as shown below:



But, it is given that zero or more a’s should be followed by zero or more b’s followed by zero or more c’s. So, the corresponding ϵ -NFA is shown below:



Thus, an ϵ -NFA is given by $M = (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{a, b, c\}$
- q_0 is start state
- $F = \{q_2\}$
- δ is shown below using the transition table:

δ	a	b	c	ϵ
\rightarrow	q_0	\emptyset	\emptyset	q_1
q_1	\emptyset	q_1	\emptyset	q_2
$*q_2$	\emptyset	\emptyset	q_2	\emptyset

2.3. Conversion from ϵ -NFA to DFA (Algorithm to Convert ϵ -NFA to DFA)

We have seen in the previous section that an NFA can be converted into DFA using subset construction. On similar lines, it is possible to convert an ϵ -NFA to DFA. Now, let us see “What is the procedure (or algorithm) to obtain a DFA from an ϵ -NFA?”. The procedure (or algorithm) is shown below:

Procedure (Algorithm): Let $M_E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ be an ϵ -NFA where Q_E is set of finite states, Σ is set of input alphabets, δ_E is transition from $Q_E \times \{\Sigma \cup \epsilon\}$ to 2^{Q_E} , q_{0E} is the start state and F_E is the set of final states. The equivalent DFA

$$M_D = (Q_D, \Sigma, \delta_D, q_{0D}, F_D)$$

can be obtained as shown below:

Step 1: If q_0 is the start state of NFA, then $\text{ECLOSE}(q_0)$ is the start state of DFA

$$\text{i.e., } q_{0D} = \text{ECLOSE}(q_0)$$

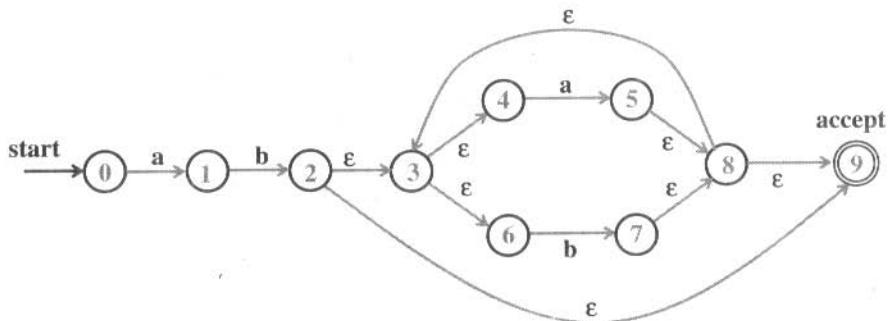
Step 2: Compute the transitions for DFA. Let $\{q_i, q_j, \dots, q_k\}$ is a state in DFA. Then, the transitions from this state on a i.e., $\delta_D(\{q_i, q_j, \dots, q_k\}, a)$ is computed as shown below:

- Let $\delta_E(\{q_i, q_j, \dots, q_k\}, a) = \{p_1, p_2, \dots, p_m\}$
- Then take $\text{ECLOSE}(\{p_1, p_2, \dots, p_m\})$

Thus, $\delta_D(\{q_i, q_j, \dots, q_k\}, a) = \text{ECLOSE}(\{p_1, p_2, \dots, p_m\})$.

Step 3: If $\{q_i, q_j, \dots, q_k\}$ is a state in DFA and if this set contains at least one final state of ϵ -NFA, then $\{q_i, q_j, \dots, q_k\}$ is a final state of DFA.

Example: Convert the following NFA to its equivalent DFA.



Note: δ_E represent the transition for ϵ -NFA.

Step 1: Identify the start state of DFA: Since 0 is the start state of ϵ -NFA, $\text{ECLOSE}(0)$ is the start state of DFA i.e., $\text{ECLOSE}(0) = \{0\}$ (A)

Consider the state A:

When input is a :

$$\begin{aligned}
 \delta(A, a) &= \text{ECLOSE } (\delta_E(A, a)) \\
 &= \text{ECLOSE } (\delta_E(0, a)) \\
 &= \{1\}
 \end{aligned} \tag{B}$$

When input is b :

$$\begin{aligned}
 \delta(A, b) &= \text{ECLOSE } (\delta_E(A, b)) \\
 &= \text{ECLOSE } (\delta_E(0, b)) \\
 &= \{\phi\}
 \end{aligned}$$

Consider the state B:

When input is a :

$$\begin{aligned}
 \delta(B, a) &= \text{ECLOSE } (\delta_E(B, a)) \\
 &= \text{ECLOSE } (\delta_E(1, a)) \\
 &= \emptyset
 \end{aligned}$$

When input is b :

$$\begin{aligned}
 \delta(B, b) &= \text{ECLOSE } (\delta_E(B, b)) \\
 &= \text{ECLOSE } (\delta_E(1, b)) \\
 &= \text{ECLOSE } (\{2\}) \\
 &= \{2, 3, 4, 6, 9\}
 \end{aligned} \tag{C}$$

Consider the state C:

When input is a :

$$\begin{aligned}
 \delta(C, a) &= \text{ECLOSE } (\delta_E(C, a)) \\
 &= \text{ECLOSE } (\delta_E(\{2, 3, 4, 6, 9\}, a)) \\
 &= \text{ECLOSE } \{5\} \\
 &= \{5, 8, 9, 3, 4, 6\} \\
 &= \{3, 4, 5, 6, 8, 9\} \text{(ascending order)}
 \end{aligned} \tag{D}$$

When input is b :

$$\begin{aligned}
 \delta(C, b) &= \text{ECLOSE } (\delta_E(C, b)) \\
 &= \text{ECLOSE } (\delta_E(\{2, 3, 4, 6, 9\}, b)) \\
 &= \text{ECLOSE } \{7\} \\
 &= \{7, 8, 9, 3, 4, 6\} \\
 &= \{3, 4, 6, 7, 8, 9\} \text{(ascending order)}
 \end{aligned} \tag{E}$$

Consider the state D:

When input is a :

$$\begin{aligned}
 \delta(D, a) &= \text{ECLOSE}(\delta(D, a)) \\
 &= \text{ECLOSE}(\delta_E(\{3,4,5,6,8,9\}, a)) \\
 &= \text{ECLOSE}(\{5\}) \\
 &= \{5, 8, 9, 3, 4, 6\} \\
 &= \{3, 4, 5, 6, 8, 9\} \text{(ascending order)}
 \end{aligned} \tag{D}$$

When input is b :

$$\begin{aligned}
 \delta(D, b) &= \text{ECLOSE}(\delta(D, b)) \\
 &= \text{ECLOSE}(\delta_E(\{3,4,5,6,8,9\}, b)) \\
 &= \text{ECLOSE}(\{7\}) \\
 &= \{7, 8, 9, 3, 4, 6\} \\
 &= \{3, 4, 6, 7, 8, 9\} \text{(ascending order)}
 \end{aligned} \tag{E}$$

Consider the state E:

When input is a :

$$\begin{aligned}
 \delta(E, a) &= \text{ECLOSE}(\delta(E, a)) \\
 &= \text{ECLOSE}(\delta_E(\{3,4,6,7,8,9\}, a)) \\
 &= \text{ECLOSE}(\{5\}) \\
 &= \{5, 8, 9, 3, 4, 6\} \\
 &= \{3, 4, 5, 6, 8, 9\} \text{(ascending order)}
 \end{aligned} \tag{D}$$

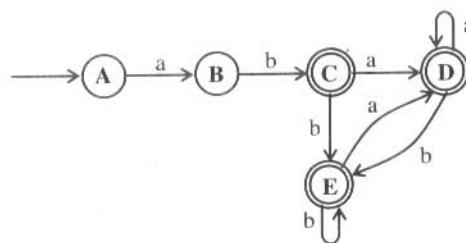
When input is b :

$$\begin{aligned}
 \delta(E, b) &= \text{ECLOSE}(\delta(E, b)) \\
 &= \text{ECLOSE}(\delta_E(\{3,4,6,7,8,9\}, b)) \\
 &= \text{ECLOSE}(\{7\}) \\
 &= \{7, 8, 9, 3, 4, 6\} \\
 &= \{3, 4, 6, 7, 8, 9\} \text{(ascending order)}
 \end{aligned} \tag{E}$$

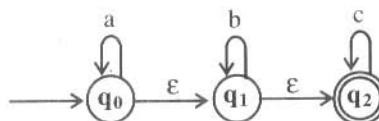
Since there are no new states, we can draw the transition table for the DFA as shown below:

δ	a	b
A	B	ϕ
B	ϕ	C
*C	D	E
*D	D	E
*E	D	E

Note: The states C,D and E are final states, since 9 which is final state of ϵ -NFA is present in C, D and E. The final transition diagram of DFA is shown below:



Example: Convert the following NFA to its equivalent DFA.



Note: Identify the start state of DFA: Since q_0 is the start state of ϵ -NFA, $\text{ECLOSE}(q_0)$ is the start state of DFA i.e., $\text{ECLOSE}(q_0) = \{q_0, q_1, q_2\}$ (A)

Consider the state $\{q_0, q_1, q_2\}$:

On input a :

$$\begin{aligned}\delta(\{q_0, q_1, q_2\}, a) &= \text{ECLOSE}(\{q_0\}) \\ &= \{q_0, q_1, q_2\}\end{aligned}\quad (\text{A})$$

On input b :

$$\begin{aligned}\delta(\{q_0, q_1, q_2\}, b) &= \text{ECLOSE}(\{q_1\}) \\ &= \{q_1, q_2\}\end{aligned}\quad (\text{B})$$

On input c :

$$\begin{aligned}\delta(\{q_0, q_1, q_2\}, c) &= \text{ECLOSE}(\{q_2\}) \\ &= q_2\end{aligned}\quad (\text{C})$$

Consider the state $\{q_1, q_2\}$:

On input a :

$$\delta(\{q_1, q_2\}, a) = \emptyset$$

On input b :

$$\begin{aligned}\delta(\{q_1, q_2\}, b) &= \text{ECLOSE}(\{q_1\}) \\ &= \{q_1, q_2\}\end{aligned}\quad (\text{D})$$

On input c :

$$\begin{aligned}\delta(\{q_1, q_2\}, c) &= \text{ECLOSE}(\{q_2\}) \\ &= q_2\end{aligned}\tag{C}$$

Consider the state $\{q_2\}$:

On input a :

$$\delta(q_0, a) = \emptyset$$

On input b :

$$\delta(q_1, b) = \emptyset$$

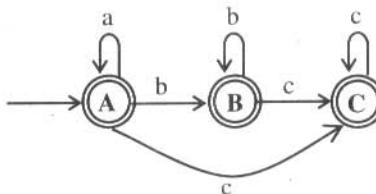
On input c :

$$\begin{aligned}\delta(q_2, c) &= \text{ECLOSE}(q_2) \\ &= q_2\end{aligned}\tag{C}$$

The transition table along with transition diagram is shown below:

δ	a	b	c
*A	A	B	C
*B	\emptyset	B	C
*C	\emptyset	\emptyset	C

The DFA can be written as shown below:



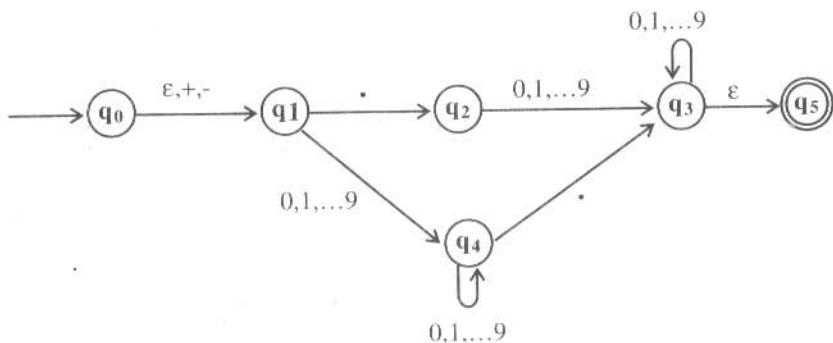
Example: Now, let us “Obtain an NFA with ϵ -transitions (ϵ -NFA) to accept decimal numbers and obtain $\delta^*(q_0, 4.7)$ ”.

Question to be changed for integers and floating point numbers.

Note: A decimal number has

1. an optional + or - sign followed by ‘.’ followed by string of digits
or
2. an optional + or - sign followed by a string of digits followed by a ‘.’ and in turn followed by string of digits

The ϵ -NFA is shown in figure below:



Here, the machine $M = (Q, \Sigma, \delta, q_0, F)$ where

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

$$\Sigma = \{+, -, ., 0, 1, 2, \dots, 9\}$$

q_0 is the start state

$$F = \{q_5\}$$
 is the final state

δ is shown below using the transition table

δ	ϵ	$+$	$-$	$.$	0 to 9
\rightarrow	q_0	q_1	q_1	\emptyset	\emptyset
q_1	\emptyset	\emptyset	\emptyset	q_2	q_4
q_2	\emptyset	\emptyset	\emptyset	\emptyset	q_3
q_3	q_5	\emptyset	\emptyset	\emptyset	q_3
q_4	\emptyset	\emptyset	\emptyset	q_3	q_4
q_5	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset

To obtain $\delta^*(q_0, 4.7)$: Since q_0 is the start state of ϵ -NFA, ECLOSE(q_0) is the start state of DFA i.e., $ECLOSE(q_0) = \{q_0, q_1\}$

$$\begin{aligned}
 \delta^*(q_0, 4) &= ECLOSE(\delta_E(\{q_0, q_1\}, 4)) \\
 &= ECLOSE(\delta_E(q_0, 4) \cup \delta_E(q_1, 4)) \\
 &= ECLOSE(\emptyset \cup q_4) \\
 &= \{q_4\}
 \end{aligned}$$

$$\begin{aligned}
 \delta^*(\{q_4\}, .) &= ECLOSE(\delta_E(\{q_4\}, .)) \\
 &= ECLOSE(q_3) \\
 &= \{q_3, q_5\}
 \end{aligned}$$

$$\begin{aligned}
 \delta^*(\{q_3, q_5\}, 7) &= ECLOSE(\delta_E(q_3, 7) \cup \delta_E(q_5, 7)) \\
 &= ECLOSE(q_3 \cup \emptyset)
 \end{aligned}$$

$$\begin{aligned}
 &= \text{ECLOSE } (q_3) \\
 &= \{q_3, q_5\}
 \end{aligned}$$

Thus, $\delta^*(q_0, 4.7) = \{q_3, q_5\}$

Example: Now, let us “Obtain the equivalent DFA for the ϵ -NFA shown above”.

Note: Identify the start state of DFA: Since q_0 is the start state of ϵ -NFA, $\text{ECLOSE}(q_0)$ is the start state of DFA i.e., $\text{ECLOSE}(q_0) = \{q_0, q_1\} \dots \dots \dots \text{(A)}$

Consider the state [A]:

When input is \pm :

$$\begin{aligned}
 \delta(A, \pm) &= \text{ECLOSE } (\delta_E(A, \pm)) \\
 &= \text{ECLOSE } (\delta_E(\{q_0, q_1\}, \pm)) \\
 &= \text{ECLOSE } (q_1) = \{q_1\}
 \end{aligned} \tag{B}$$

When input is .

$$\begin{aligned}
 \delta(A, .) &= \text{ECLOSE } (\delta_E(A, .)) \\
 &= \text{ECLOSE } (\delta_E(\{q_0, q_1\}, .)) \\
 &= \text{ECLOSE } (q_2) = \{q_2\}
 \end{aligned} \tag{C}$$

When input is 0,1,2,...9

$$\begin{aligned}
 \delta(A, \{0, 1, \dots, 9\}) &= \text{ECLOSE } (\delta_E(A, \{0, 1, \dots, 9\})) \\
 &= \text{ECLOSE } (\delta_E(\{q_0, q_1\}, \{0, 1, \dots, 9\})) \\
 &= \text{ECLOSE } (q_4) = \{q_4\}
 \end{aligned} \tag{D}$$

Consider the state [B]:

When input is \pm :

$$\begin{aligned}
 \delta(B, \pm) &= \text{ECLOSE } (\delta_E(B, \pm)) \\
 &= \text{ECLOSE } (\delta_E(q_1, \pm)) \\
 &= \text{ECLOSE } (\emptyset) = \emptyset
 \end{aligned}$$

When input is .

$$\begin{aligned}
 \delta(B, .) &= \text{ECLOSE } (\delta_E(B, .)) \\
 &= \text{ECLOSE } (\delta_E(q_1, .)) \\
 &= \text{ECLOSE } (q_2) = \{q_2\}
 \end{aligned} \tag{C}$$

When input is 0,1,2,...9

$$\begin{aligned}
 \delta(B, \{0,1,\dots,9\}) &= ECLOSE(\delta_E(B, \{0,1,\dots,9\})) \\
 &= ECLOSE(\delta_E(q_1, \{0,1,\dots,9\})) \\
 &= ECLOSE(q_4) = \{q_4\}
 \end{aligned} \tag{D}$$

Consider the state [C]:

When input is \pm :

$$\begin{aligned}
 \delta(C, \pm) &= ECLOSE(\delta_E(C, \pm)) \\
 &= ECLOSE(\delta_E(q_2, \pm)) \\
 &= ECLOSE(\emptyset) = \emptyset
 \end{aligned}$$

When input is .

$$\begin{aligned}
 \delta(C, .) &= ECLOSE(\delta_E(C, .)) \\
 &= ECLOSE(\delta_E(q_2, .)) \\
 &= ECLOSE(\emptyset) = \emptyset
 \end{aligned}$$

When input is 0,1,2,...9

$$\begin{aligned}
 \delta(C, \{0,1,\dots,9\}) &= ECLOSE(\delta_E(C, \{0,1,\dots,9\})) \\
 &= ECLOSE(\delta_E(q_2, \{0,1,\dots,9\})) \\
 &= ECLOSE(q_3) \\
 &= \{q_3, q_5\}
 \end{aligned} \tag{E}$$

Consider the state [D]:

When input is \pm :

$$\begin{aligned}
 \delta(D, \pm) &= ECLOSE(\delta_E(D, \pm)) \\
 &= ECLOSE(\delta_E(q_4, \pm)) \\
 &= ECLOSE(\emptyset) = \emptyset
 \end{aligned}$$

When input is .

$$\begin{aligned}
 \delta(D, .) &= ECLOSE(\delta_E(D, .)) \\
 &= ECLOSE(\delta_E(q_4, .)) \\
 &= ECLOSE(q_3) \\
 &= \{q_3, q_5\}
 \end{aligned} \tag{E}$$

When input is 0,1,2,...9

$$\begin{aligned}
 \delta(D, \{0,1,\dots,9\}) &= ECLOSE(\delta_E(D, \{0,1,\dots,9\})) \\
 &= ECLOSE(\delta_E(q_4, \{0,1,\dots,9\})) \\
 &= ECLOSE(q_4) \\
 &= \{q_4\}(D)
 \end{aligned}$$

Consider the state [E]:

When input is \pm :

$$\begin{aligned}\delta(E, \pm) &= ECLOSE(\delta_E(E, \pm)) \\ &= ECLOSE(\delta_E(\{q_3, q_5\}, \pm)) \\ &= ECLOSE(\emptyset) = \emptyset\end{aligned}$$

When input is $.$

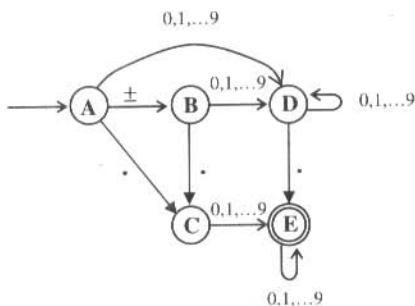
$$\begin{aligned}\delta(E, .) &= ECLOSE(\delta_E(E, .)) \\ &= ECLOSE(\delta_E(\{q_3, q_5\}, .)) \\ &= ECLOSE(\emptyset) = \emptyset\end{aligned}$$

When input is $0, 1, 2, \dots, 9$

$$\begin{aligned}\delta(E, \{0, 1, \dots, 9\}) &= ECLOSE(\delta_E(E, \{0, 1, \dots, 9\})) \\ &= ECLOSE(\delta_E(\{q_3, q_5\}, \{0, 1, \dots, 9\})) \\ &= ECLOSE(q_3) \\ &= \{q_3, q_5\}\end{aligned} \tag{E}$$

The transition table along with transition diagram is shown below:

δ	$+$	$-$	$.$	$0 \text{ to } 9$
A	B	B	C	D
B	\emptyset	\emptyset	C	D
C	\emptyset	\emptyset	\emptyset	E
D	\emptyset	\emptyset	E	D
*E	\emptyset	\emptyset	\emptyset	E



Note: Since $E = \{q_3, q_5\}$ has a final state of NFA, E is the final state in DFA.

Now, let us "Prove that the language L is accepted by DFA M iff L is accepted by an (ϵ -NFA) N".

Theorem: A language L is accepted by DFA M iff L is accepted by an (ϵ -NFA) N.

Proof: Let $N = (Q_E, \Sigma, \delta_E, q_0, F_E)$ be an ϵ -NFA. Applying the modified subset construction (by incorporating ϵ -CLOSURE), let the DFA obtained is

$$M = (Q_D, \Sigma, \delta_D, q_{0D}, F_D)$$

Since all the transitions of DFA are obtained from the transitions of ϵ -NFA, the language accepted by ϵ -NFA is accepted by DFA. This can be proved using mathematical induction as shown below:

Basis: if $w = \epsilon$, we know that

$$\delta_E^*(q_0, \epsilon) = ECLOSE(q_0) \quad (1)$$

We also know that

$$q_{0D} = ECLOSE(q_0)$$

is correct since that is how the start state of DFA is obtained. So, if $w = \epsilon$, then

$$\delta_D^*(q_{0D}, \epsilon) = ECLOSE(q_0) \quad (2)$$

By comparing the two relations (1) and (2) we have

$$\delta_E^*(q_0, \epsilon) = \delta_D^*(q_{0D}, \epsilon)$$

Thus, we have proved that language accepted by ϵ -NFA is same as language accepted by DFA when $w = \epsilon$.

Induction hypotheses: Let $w = x$ and assume that the result is true for the string x i.e.,

$$\delta_E^*(q_0, x) = \delta_D^*(q_{0D}, x) = \{q_i, q_j, \dots, q_k\}$$

Inductive proof: Let $w = xa$ where a is the final symbol of w and we know that:

$$\delta_E^*(q_0, x) = \delta_D^*(q_{0D}, x) = \{q_i, q_j, \dots, q_k\}$$

is true for the string x (induction hypotheses). Now, we have to prove that the result is true for $w = xa$. By definition of δ^* for ϵ -NFA, we compute $\delta_E^*(q_0, w)$ as shown below:

- Let $\delta_E(\{q_i, q_j, \dots, q_k\}, a) = \{p_1, p_2, \dots, p_m\}$

- Then $\delta_E^*(q_0, w) = ECLOSE(\{p_1, p_2, \dots, p_m\})$

Thus,

$$\delta_E^*(\{q_0, w\}) = ECLOSE(\{p_1, p_2, \dots, p_m\}) \quad (5)$$

If we examine the construction of DFA from ϵ -NFA, the above two steps [shown in relations (3) and (4)] are used. Thus, $\delta_D^*(q_0, w) = \delta_E^*(q_0, w)$.

Example: Now, let us “Obtain a NFA’s to recognize the strings abc , abd and $aacd$ assuming $\Sigma = \{a, b, c, d\}$ ”.

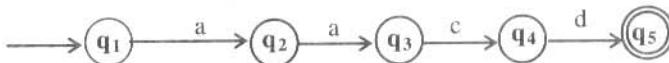
Solution: The machine to accept the string abc is shown below:



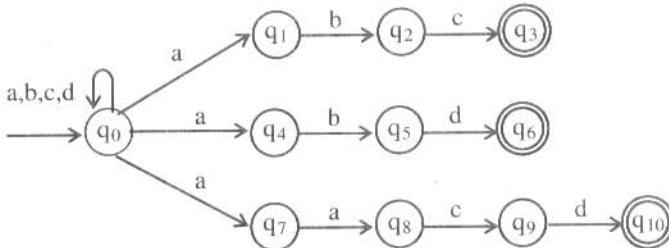
The machine to accept the string abd is shown below:



The machine to accept the string aacd is shown below:



But, all these strings can be preceded by strings of a's, b's, c's and d's and observe that the first symbol in each of these machines is *a*. So, the complete NFA to accept the strings abc, abd and aacd can be written as shown below:

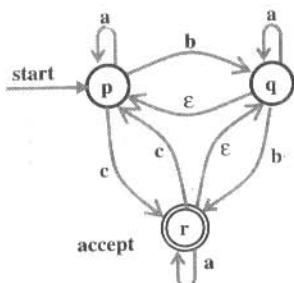


Example: Now, let us “Consider the following ϵ -NFA”:

δ	ϵ	a	b	c
p	\emptyset	{p}	{q}	{r}
q	{p}	{q}	{r}	\emptyset
*r	{q}	{r}	\emptyset	{p}

- (i) Compute ϵ -CLOSURE of each state.
- (ii) Give all the strings of length three or less accepted by the automata.
- (iii) Convert the automata to DFA.

Solution: The transition diagram for the given ϵ -NFA is shown below:



1. **Solution:** The ϵ -CLOSURE of each state can be computed as shown below:

$$\text{ECLOSE}(p) = \{p\}$$

$$\text{ECLOSE}(q) = \{p, q\}$$

$$\text{ECLOSE}(r) = \{p, q, r\}$$

2. Solution: All the strings of length three or less accepted by the automata can be obtained as shown below:

- We can move from state to p to state r using the strings: c , bb and bc .
 - After reaching r , on either b or c we can return to r using ϵ -transitions. So, observe that strings of b 's and c 's of length 3 or less except b are accepted i.e., the following strings are accepted:
 - c , bb , bc , cb and cc (length 2 or less)
 - bbb , bbc , bcb , bcc , cbb , cbc , ccb , ccc (of length 3)
 - When we insert a 's into above strings at any of the positions keeping length 3 or less, we find that strings of a 's, b 's and c 's with at most one a is accepted.
 - The strings consisting of one c and at most two a 's are accepted.

3. Solution: The DFA can be obtained using ϵ -NFA as shown below:

Note: Identify the start state of DFA: Since p is the start state of ϵ -NFA, ECLOSE(p) is the start state of DFA i.e., $\text{ECLOSE}(p) = \{p\}$(A)

Consider the state $\{p\}$ i.e., A:

On input a :

$$\delta(\{p\}, a) = \text{ECLOSE}(\{p\}) \\ = \{p\}$$

On input b :

$$\delta(\{p\}, b) = \text{ECLOSE}(\{q\}) \\ = \{p, q\}(B)$$

On input c :

$$\delta(\{p\}, c) = \text{ECLOSE } (\{r\}) \\ = \{p, q, r\}$$

Consider the state $\{p,q\}$ i.e., B:

On input a :

$$\begin{aligned}\delta(\{p,q\}, a) &= \text{ECLOSE } (\delta(p,a) \cup \delta(q, a)) \\ &= \text{ECLOSE } (p, q) \\ &= \text{ECLOSE } (p, q) \\ &= \{p, q\}(B)\end{aligned}$$

On input b :

$$\begin{aligned}\delta(\{p,q\}, b) &= \text{ECLOSE } (\delta(p,b) \cup \delta(q, b)) \\ &= \text{ECLOSE } (q, r) \\ &= \text{ECLOSE } (p) \cup \text{ECLOSE } (r) \\ &= \{p, q, r\}(C)\end{aligned}$$

On input c :

$$\begin{aligned}\delta(\{p,q\}, c) &= \text{ECLOSE } (\delta(p,c) \cup \delta(q, c)) \\ &= \text{ECLOSE } (r, \emptyset) \\ &= \text{ECLOSE } (r) \\ &= \{p, q, r\}(C)\end{aligned}$$

Consider the state $\{p, q, r\}$ i.e., C:

On input a :

$$\begin{aligned}\delta(\{p,q,r\}, a) &= \text{ECLOSE } (\delta(p,a) \cup \delta(q, a) \cup \delta(r,a)) \\ &= \text{ECLOSE } (p, q, r) \\ &= \text{ECLOSE } (p, q, r) \\ &= \{p, q, r\}(C)\end{aligned}$$

On input b :

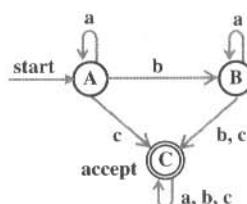
$$\begin{aligned}\delta(\{p,q,r\}, b) &= \text{ECLOSE } (\delta(p,b) \cup \delta(q, b) \cup \delta(r,b)) \\ &= \text{ECLOSE } (q, r, \emptyset) \\ &= \text{ECLOSE } (p, q, r) \\ &= \{p, q, r\}(C)\end{aligned}$$

On input c :

$$\begin{aligned}\delta(\{p,q,r\}, c) &= \text{ECLOSE } (\delta(p,c) \cup \delta(q, c) \cup \delta(r,c)) \\ &= \text{ECLOSE } (r, \emptyset, \emptyset) \\ &= \text{ECLOSE } (r) \\ &= \{p, q, r\}(C)\end{aligned}$$

The transition table along with transition diagram is shown below:

δ	a	b	c
A	A	B	C
B	B	C	C
*C	C	C	C



2.4. Difference Between DFA, NFA and ϵ -NFA

Now, let us see “What are the difference between DFA, NFA and ϵ -NFA?” Strictly speaking the difference between DFA and NFA lies only in the definition of δ . Using this difference some more points can be derived and can be written as shown:

DFA	NFA	ϵ -NFA
<p>The DFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where</p> <ul style="list-style-type: none"> • Q is set of finite states • Σ is set of input alphabets • $\delta : Q \times \Sigma \rightarrow Q$ • q_0 is the start state • $F \subseteq Q$ is set of final states 	<p>An NFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where</p> <ul style="list-style-type: none"> • Q is set of finite states • Σ is set of input alphabets • $\delta : Q \times \Sigma \rightarrow 2^Q$ • q_0 is the start state • $F \subseteq Q$ is set of final states 	<p>An ϵ-NFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where</p> <ul style="list-style-type: none"> • Q is set of finite states • Σ is set of input alphabets • $\delta : Q \times (\Sigma \cup \epsilon) \rightarrow 2^Q$ • q_0 is the start state • $F \subseteq Q$ is set of final states
<ul style="list-style-type: none"> • There can be zero or one transition from a state on an input symbol 	<ul style="list-style-type: none"> • There can be zero, one or more transitions from a state on an input symbol 	<ul style="list-style-type: none"> • There can be zero, one or more transitions from a state with or without giving any input
<ul style="list-style-type: none"> • More number of transitions 	<ul style="list-style-type: none"> • Less number of transitions 	<ul style="list-style-type: none"> • Relatively more transitions when compared with NFA
<ul style="list-style-type: none"> • Difficult to construct 	<ul style="list-style-type: none"> • Easy to construct 	<ul style="list-style-type: none"> • Easy to construct using regular expressions
<ul style="list-style-type: none"> • Less powerful since at any point of time it will be in only one state 	<ul style="list-style-type: none"> • More powerful than DFA since at any point of time it will be in more than one state 	<ul style="list-style-type: none"> • More powerful than NFA since at any point of time it will be in more than one state with or without giving any input

2.5. Regular Expressions

The language accepted by DFA or NFA and ϵ -NFA is called regular language. A regular language can be described using regular expressions consisting of the symbols such as alphabets in Σ , the operators such as ‘.’, ‘+’ and ‘*’. The three operators used to obtain a regular expression are:

- (+ operator) union operator(least precedence)
- (. operator) concatenation operator(next least precedence)
- (*) operator closure operator(highest precedence)

Note: The symbols ‘(’ and ‘)’ can be used in regular expressions. The order of evaluation of the regular expression is determined by the parenthesis and the priority associated with other operators.

Now, let us see “What is a regular expression?”. A regular expression can be formally defined as follows.

❖ **Definition:** A regular expression is recursively defined as follows.

1. \emptyset is a regular expression denoting an empty language.
2. ϵ -(epsilon) is a regular expression indicates the language containing an empty string.
3. a is a regular expression which indicates the language containing only {a}.
4. If R is a regular expression denoting the language L_R and S is a regular expression denoting the language L_S , then
 - a) $R+S$ is a regular expression corresponding to the language $L_R \cup L_S$.
 - b) $R.S$ is a regular expression corresponding to the language $L_R.L_S$.
 - c) R^* is a regular expression corresponding to the language L_R^* .
5. The expressions obtained by applying any of the rules from 1 to 4 are regular expressions.

The following table shows some examples of regular expressions and the language corresponding to these regular expressions:

Regular expressions	Meaning
a^*	String consisting of any number of a's (or string consisting of zero or more a's)
a^+	String consisting of at least one a (or string consisting of one or more a's)
$(a+b)$	String consisting of either one a or one b
$(a+b)^*$	Set of strings of a's and b's of any length including the NULL string
$(a+b)^*abb$	Set of strings of a's and b's ending with the string abb

$ab(a+b)^*$	Set of strings of a's and b's starting with the string ab
$(a+b)^*aa(a+b)^*$	Set of strings of a's and b's having a sub string aa
$a^*b^*c^*$	Set of string consisting of any number of a's (may be empty string also) followed by any number of b's (may include empty string) followed by any number of c's (may include empty string)
$a^*b^*c^+$	Set of string consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'
$aa^*bb^*cc^*$	Set of strings consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'
$(a+b)^* (a + bb)$	Set of strings of a's and b's ending with either a or bb
$(aa)^*(bb)^*b$	Set of strings consisting of even number of a's followed by odd number of b's
$(0+1)^*000$	Set of strings of 0's and 1's ending with three consecutive zeros (or ending with 000)
$(11)^*$	Set of strings consisting of even number of 1's
$01^* + 1$	The language represented is the string 1 or the string consisting of a zero followed by any number of 1's possibly including none
$(01)^* + 1$	The language consists of a string 1 or strings of (01)'s that repeat zero or more times
$0(1^* + 1)$	Set of strings consisting of a zero followed by any number of 1's
$(1+\varepsilon)(00^*1)^*0^*$	Strings of 0's and 1's without any consecutive 1's
$(0+10)^*1^*$	Strings of 0's and 1's ending with any number of 1's (possibly none)
$(a+b)(a+b)$	Strings of a's and b's whose length is 2

(1) Obtain a regular expression representing strings of a's and b's having length 2.

Solution: Strings of a's and b's with length two are: aa, ab, ba and bb.

So, RE is: $(aa + ab + ba + bb)$

Note: The above regular expression can also be written as:

So, R.E is $(a+b)(a+b)$

(2) Obtain a regular expression to accept strings of a's and b's of length ≤ 2 .

The strings of a's and b's whose length is ≤ 2 can be written as shown below:

$$\epsilon + a + b + aa + ab + ba + bb$$

The above strings can be obtained using the regular expression:

$$(\epsilon + a + b)(\epsilon + a + b)$$

Using short hand notation, the above R.E can be written as:

$$(\epsilon + a + b)^2$$

So, R.E is $(\epsilon + a + b)^2$

(3) Obtain a regular expression to accept strings of a's and b's of length ≤ 10 .

By using the previous problem, we can write the expression as:

$$(\epsilon + a + b)^{10}$$

So, R.E is $(\epsilon + a + b)^{10}$

(4) Obtain a regular expression representing strings of a's and b's having even length.

Solution: The regular expression is obtained by having zero or more combinations of strings aa, ab, ba, and bb.

So, RE is: $(aa + ab + ba + bb)^*$

Note: The above regular expression can also be written as:

So, R.E is $((a+b)(a+b))^*$

(5) Obtain a regular expression representing strings of a's and b's having odd length.

Solution: The regular expression is obtained by prefixing or suffixing a or b denoted by $(a+b)$ to the regular expression shown in previous problem as shown below:

So, RE is: $(aa + ab + ba + bb)^* (a+b)$

or

$(a+b) (aa + ab + ba + bb)$

Note: The above regular expression can also be written as:

So, R.E is $((a+b)(a+b))^* (a + b)$

or

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

(6) Obtain a regular expression such that $L(R) = \{w \mid w \in \{0, 1\}^* \text{ with at least three consecutive } 0's\}$.

The regular expression representing 3 consecutive zero's is shown below:

000

The string 000 can be preceded by or followed by an arbitrary string consisting of 0's and 1's can be represented by the regular expression:

$(0+1)^*$

So, the regular expression can be written as

$(0+1)^* 000 (0+1)^*$

So, R.E is $(0+1)^* 000 (0+1)^*$

Note: The language corresponding to the regular expression can be written as

$L(R) = \{ (0+1)^m 000 (0+1)^n \mid m \geq 0 \text{ and } n \geq 0 \}$

(7) Obtain a regular expression to accept strings of 0's and 1's having no two consecutive zeros.

The first observation from the statement is that whenever a 0 occurs it should be followed by 1. But, there is no restriction on the number of 1's. So, it is a string consisting of any combinations

of 1's and 01's. So, the partial regular expression for this can be of the form

$$(1 + 01)^*$$

No doubt that the above expression is correct. But, suppose the string ends with a 0. What to do? For this, the string obtained from above regular expression may end with 0 or may end with ϵ which can be represented by the regular expression:

$$(0 + \epsilon)$$

So, R.E is $(1 + 01)^* (0 + \epsilon)$

(8) Obtain a regular expression to accept string of a's and b's starting with 'a' and ending with 'b'.

Strings of a's and b's of arbitrary length can be written as

$$(a + b)^*$$

But, to get the string starting with 'a' and end with 'b' we have to precede the above RE with a and following the above RE with b .

So, R.E is $a (a + b)^* b$

(9) Obtain a regular expression to accept string of a's and b's whose second symbol from the right end is a.

The first symbol from the right end can be a or b which can be represented by the RE

$$(a + b)$$

The second symbol from the right end should be a which can be represented by the RE

$$a$$

But, the third symbol from the right can start with any combinations of a's and b's denoted by the RE:

$$(a+b)^*$$

So, R.E is $(a + b)^* a (a+b)^*$

Note: On similar lines, strings of a's and b's whose third symbol from the right end is *a* can be written as

$$\text{So, R.E is } (a + b)^* a(a+b) (a+b)$$

(10) Obtain a regular expression representing string of a's and b's whose tenth symbol from the right end is a.

On similar lines to the previous problem, the RE representing strings of a's and b's whose tenth symbol from the right end is *a* can be written as shown below:

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

Since there are ten (a+b) expressions to the right of *a*, the above expression using short hand notation can be written as shown below:

$$\text{So, R.E} = (a + b)^* a(a+b)^9$$

Note: It is clear from this expression that all strings must be of length 10 or more. Here, we need to track the last 10 characters.

(11) Obtain a regular expression to accept strings of a's and b's such that third symbol from the right is *a* and fourth symbol from the right is *b*.

We are interested only in the third and fourth symbol so that third symbol from the right end is *a* and fourth symbol from the right end is *b* and the regular expression for this is given by

$$ba(a + b) (a + b)$$

But, this substring can be preceded by any combinations of a's and b's which is given by the regular expression

$$(a + b)^*$$

By concatenating the above two regular expressions, we can write the regular expression.

$$\text{So, R.E} = (a + b)^* ba(a + b) (a + b)$$

(12) Obtain a regular expression to accept the words with two or more letters but beginning and ending with the same letter where $\Sigma = \{a, b\}$.

The string consisting of a's and b's can be denoted by the regular expression

$$(a + b)^*$$

Since, the string should start and end with the same letter, the above regular expression can be preceded and followed by a as shown below:

$$\text{RE 1} = a(a+b)^*a$$

or preceded by and followed by b as shown below:

$$\text{RE 2} = b(a+b)^*b$$

So, the final regular expression with two or more letters but beginning and ending with the same letter where $\Sigma = \{a, b\}$ can be written as:

$$\text{So, R.E} = a(a+b)^*a + b(a+b)^*b$$

(13) Obtain a regular expression to accept strings of a's and b's whose length is either even or multiples of 3 or both.

The regular expression whose length is even can be obtained using

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

and the regular expression whose length is multiples of 3 can be obtained using

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

Thus, the regular expression whose length is either even or multiples of 3 or both is obtained by taking various combinations of strings obtained using the above regular expressions:

$$\text{So, R.E} = ((a + b)(a + b))^* + ((a + b)(a + b)(a + b))^*$$

(14) Obtain a regular expression to accept strings of a's and b's such that every block of four consecutive symbols contains at least two a's.

It is required to have a block of 4 symbols with at least two a's which can be represented by the RE:

$$a(a+b)(a+b)$$

or

$$a(a+b)a(a+b)$$

or

$$a(a+b)(a+b)a$$

or

$$(a+b)aa(a+b)$$

or

$$(a+b)a(a+b)a$$

or

$$(a+b)(a+b)aa$$

We can have any combinations of strings obtained from above REs. But, it is not possible to have star operator * (which indicates zero or more symbols) since it encompasses even the empty string. So, we have to use the + operator (which indicates one or more symbols) and can be expressed as shown below:

$$\text{So, R.E So, RE} = (aa(a+b)(a+b) + a(a+b)a(a+b) + a(a+b)(a+b)a + (a+b)aa(a+b) + (a+b)a(a+b)a + (a+b)(a+b)aa)^+$$

(15) Obtain a regular expression for the language $L = \{a^n b^m \mid m + n \text{ is even}\}$.

Solution: It is given that

- n represent number of a's
- m represent number of b's

It is also given that $m + n$ is even. This results in following two cases:

Case 1: ($m + n$ is even) Even number of a's followed by even number of b's results in even number of symbols which is given by the regular expression:

$$\text{RE} = (aa)^* (bb)^*$$

Case 2: ($m + n$ is even) Odd number of a's followed by odd number of b's results in even number of symbols which is given by the regular expression:

$$\text{RE} = a(aa)^* b(bb)^*$$

The final regular expression is obtained by adding the above two regular expressions.

$$\text{So, R.E} = (aa)^* (bb)^* + a(aa)^* b(bb)^*$$

(16) Obtain a regular expression for the language $L = \{a^n b^m \mid m \geq 1, n \geq 1, nm \geq 3\}$.

Solution: The possible cases to satisfy the given relations are shown below:

Case 1: Since $nm \geq 3$, if $m = 1$ then $n \geq 3$. The equivalent RE is given by:

$$RE = aaaa^* b$$

Case 2: Since $nm \geq 3$, if $n = 1$ then $m \geq 3$. The equivalent RE is given by:

$$RE = a bbbb^*$$

Case 3: Since $nm \geq 3$, if $m \geq 2$ then $n \geq 2$. The equivalent RE is given by:

$$RE = aaa^* bbb^*$$

So, the final regular expression is obtained by adding all the above regular expressions.

$$\text{So, R.E} = aaaa^* b + a bbbb^* + aaa^* bbb^*$$

(17) Obtain a regular expression for the language $L = \{a^{2n} b^{2m} \mid n \geq 0, m \geq 0\}$.

Solution: For every $n \geq 0$, a^{2n} results in even number of a's and for every $m \geq 0$, b^{2m} results in even number of b's. The regular expression representing even number of a's is given by

$$RE = (aa)^*$$

The regular expression representing even number of b's is given by

$$RE = (bb)^*$$

So, regular expression corresponding to $L = \{a^{2n} b^{2m} \mid n \geq 0 \text{ and } m \geq 0\}$ is obtained by concatenating the above two regular expressions.

$$\text{So, R.E} = (aa)^* (bb)^*$$

(18) Obtain a regular expression for strings of a's and b's containing not more than three a's.

Solution: Strings of a's and b's containing not more than three a's results in following cases:

- ϵ indicating zero a's

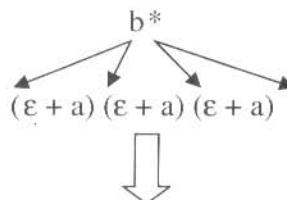
or

- one a
or
- two a's
or
- three a's

Note: At any point of time, number of a's should never exceed three a's. The regular expression for all the above strings can be written as:

$$\text{RE: } (\epsilon + a)(\epsilon + a)(\epsilon + a)$$

Now, we can insert any number of b's represented by b^* at various positions in above expression as shown below:



$$\boxed{\text{So, R.E} = b^*(\epsilon + a)b^*(\epsilon + a)b^*(\epsilon + a)b^*}$$

(19) Obtain a regular expression for the language $L = \{a^n b^m : n \geq 4, m \leq 3\}$.

Solution: The given language consists of n number of a's followed by m number of b's where $n \geq 4$ and $m \leq 3$. This language can be interpreted as concatenation of two languages:

- The first language consists of strings of four 4's followed by any number of a's (since $n \geq 4$). This is represented by the regular expression:

$$\text{RE} = \text{aaaa}(a)^* \quad (1)$$

- The second language consist of strings of at most three b's i.e., it may have zero b's or one b, or two b's or three b's but not more than three b's. This can be represented by the regular expression:

$$\text{RE} = (\epsilon + b)(\epsilon + b)(\epsilon + b) \quad (2)$$

So, the final regular expression is obtained by concatenating RE shown in (1) and RE shown in (2).

$$\boxed{\text{So, R.E} = \text{aaaa}(a)^* (\epsilon + b)(\epsilon + b)(\epsilon + b)}$$

(20) Obtain a regular expression for strings of a's and b's whose lengths are multiples of 3.

Solution: The strings of a's and b's whose length is 3 is given by the following regular expression:

$$\text{RE} = (a + b) (a + b) (a + b)$$

The regular expression for strings of a's and b's whose lengths are multiples of 3 can be obtained by using the * operator:

$$\text{So, R.E} = ((a + b) (a + b) (a + b))^*$$

(21) Obtain a regular expression for the language $L = \{w : |w| \bmod 3 = 0 \text{ where } w \in (a,b)^*\}$.

It is given that the language consists of strings whose length is multiple of 3 which can be written as shown below:

$$L = \{ \epsilon, aaa, aab, aba, abb, baa, bab, bba, bbb, \dots \}$$

The strings of a's and b's whose length is 3 is given by the following regular expression:

$$\text{RE: } (a + b) (a + b) (a + b)$$

Note that minimum string has to be ϵ whose length is 0 which is divisible by 3. The string whose length is multiples of 3 can be obtained using * operator for the above regular expression.

$$\text{So, R.E} = ((a + b) (a + b) (a + b))^*$$

Note: The above regular expression and the regular expression obtained in previous problem are same.

(22) Obtain a regular expression for the language $L = \{w : n_a(w) \bmod 3 = 0 \text{ where } w \in (a,b)^*\}$.

Observe that even though the string is made up of a's and b's we are interested in only number of a's. The number of a's in the string should be divisible by 3. So, number of a's can be 0, 3, 6 and so on.

Note: There is no restriction on the number of b's. So, any number of b's denoted by b^* can be inserted.

The minimum string that correspond to the language is

aaa

By incorporating any number of b's denoted by b^* in all possible places in the above expression, we have the following regular expression:

RE: $b^*ab^*ab^*ab^*$

To incorporate ϵ in the above regular expression, we can use * operator:

$$\text{So, R.E} = (b^*ab^*ab^*ab^*)^*$$

(23) Obtain a regular expression for the set of all strings that do not end with 01, over }0, 1}*.

It is given that the string should not end with 01. So, other strings whose length is 2 and that do not end with 01 are 00, 10 and 11. These strings can be preceded by any combinations of 0's and 1's denoted by $(0 + 1)^*$. These strings can end with 00 or 10 and 11:

$$\text{So, R.E} = (0 + 1)^* (00 + 10 + 11)$$

(24) Obtain a regular expression for L = {vuv : u, v ∈ {a, b}* and |v| = 2}.

Since $|v| = 2$, the string having length 2 can be represented by the regular expression:

$$v = aa + ab + ba + bb$$

Since there is no restriction on string u , the string u can be any combinations of a's and b's and is represented by the following regular expression:

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

So, the final regular expression to accept the language:

$$L = \{vuv : u, v \in \{a, b\}^* \text{ and } |v| = 2\}$$

is obtained by concatenating the regular expressions V, U and V.

$$\text{So, R.E} = aa(a + b)^*aa + ab(a + b)^*ab + ba(a + b)^*ba + bb + bb(a + b)^*bb$$

(25) Obtain a regular expression for $L = \{w : \text{string ends with ab or ba where } w \in \{a, b\}^*\}$.

The regular expression corresponding to the string ending with ab or ba can be written as:

$$RE = ab + ba$$

But, the strings of a's and b's denoted by the regular expression $(a+b)^*$ ending with ab or ba can be written as:

$$RE = (a+b)^*(ab+ba)$$

Note: Now, let us see “What are precedence of regular expression operators?”. In all above regular expressions that we have discussed so far have the operators $+$, $*$ and concatenation operator with the following precedence of operators.

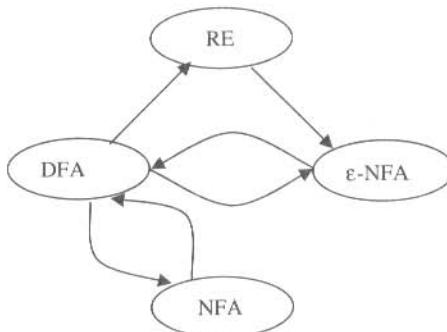
- The star operator (denoted by $*$) has the highest precedence
- The concatenation operator (also called dot operator) has the next highest precedence
- The union operator (denoted by $+$ symbol) has the least precedence

Note: The precedence of all the above operators can be changed using parentheses. For example, $(a+b)a^*$. In this expression

- $(a+b)$ has the highest precedence.
- Next preference is given for $*$ operator.
- Next preference is given for concatenation operator.

2.6. Finite Automata and Regular Expressions

This section shows the relation between DFA, NFA, ϵ -NFA and regular expression:



Now, let us see how to obtain an ϵ -NFA from the regular expression and later to obtain the regular expression from a given FA.

2.6.1. To Obtain ϵ -NFA from Regular Expression

Theorem: Let R be a regular expression. Then there exists a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ which accepts $L(R)$.

or

Prove that there exists a finite automataon to accept the language $L(R)$ corresponding to the regular expression R .

Proof: By definition, ϕ , ϵ and a are regular expressions. So, the corresponding machines to recognize the language for the respective expressions are shown in Figure 2.2.

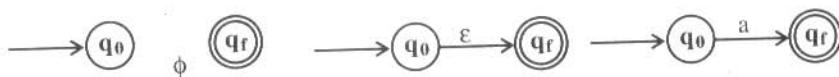


Figure 2.2. ϵ -NFAs to accept ϕ , ϵ and a

The schematic representation of a regular expression R to accept the language $L(R)$ is shown in Figure 2.3, where q is the start state and f is the final state of machine M .

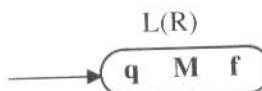


Figure 2.3. Schematic representation of FA accepting $L(R)$

In the definition of a regular expression it is clear that if R and S are regular expression, then $R+S$ and $R.S$ and R^* are regular expressions which clearly uses three operators '+', '*' and '.'. Let us take each case separately and construct equivalent machine.

Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, f_1)$ be a machine which accepts the language $L(R_1)$ corresponding to the regular expression R_1 . Let $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, f_2)$ be a machine which accepts the language $L(R_2)$ corresponding to the regular expression R_2 . Then the various machines corresponding to the regular expressions $R_1 + R_2$, $R_1 . R_2$ and R^* are shown below:

Case 1: $R = R_1 + R_2$. We can construct an NFA which accepts either $L(R_1)$ or $L(R_2)$ which can be represented as $L(R_1 + R_2)$ as shown in Figure 2.4.

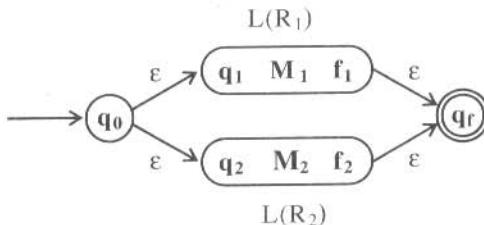


Figure 2.4. To accept the language $L(R_1 + R_2)$

It is clear from Figure 2.4 that the machine can either accept $L(R_1)$ or $L(R_2)$. Here, q_0 is the start state of the combined machine and q_f is the final state of combined machine M .

Case 2: $R = R_1 \cdot R_2$. We can construct an NFA which accepts $L(R_1)$ followed by $L(R_2)$ which can be represented as $L(R_1 \cdot R_2)$ as shown in Figure 2.5.

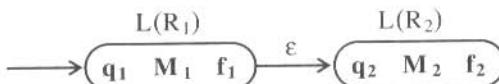


Figure 2.5. To accept the language $L(R_1 \cdot R_2)$

It is clear from Figure 2.5 that the machine after accepting $L(R_1)$ moves from state q_1 to f_1 . Since there is a ϵ -transition, without any input there will be a transition from state f_1 to state q_2 . In state q_2 , upon accepting $L(R_2)$, the machine moves to f_2 which is the final state. Thus, q_1 which is the start state of machine M_1 becomes the start state of the combined machine M and f_2 which is the final state of machine M_2 , becomes the final state of machine M and accepts the language $L(R_1 \cdot R_2)$.

Case 3: $R = (R_1)^*$. We can construct an NFA which accepts either $L(R_1)^*$ as shown in Figure 2.6.

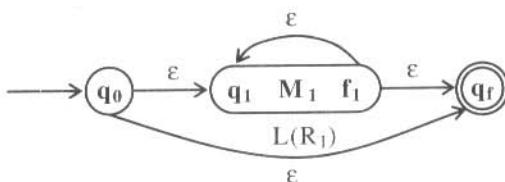


Figure 2.6. To accept the language $L(R_1)^*$

It is clear from Figure 2.6 that the machine can either accept ϵ or any number of $L(R_1)$'s thus accepting the language $L(R_1)^*$. Here, q_0 is the start state q_f is the final state.

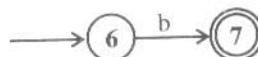
(1) Obtain an NFA which accepts strings of a's and b's starting with the string ab.

The regular expression corresponding to this language is $ab(a+b)^*$.

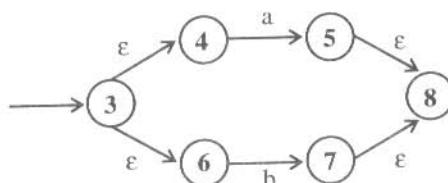
Step 1: The machine to accept 'a' is shown below.



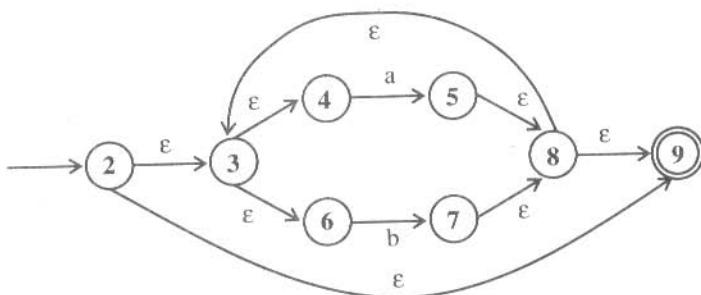
Step 2: The machine to accept 'b' is shown below.



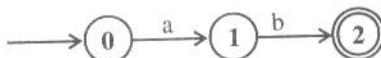
Step 3: The machine to accept $(a + b)$ is shown below.



Step 4: The machine to accept $(a+b)^*$ is shown below.



Step 5: The machine to accept ab is shown below.



Step 6: The machine to accept $ab(a+b)^*$ is shown below.

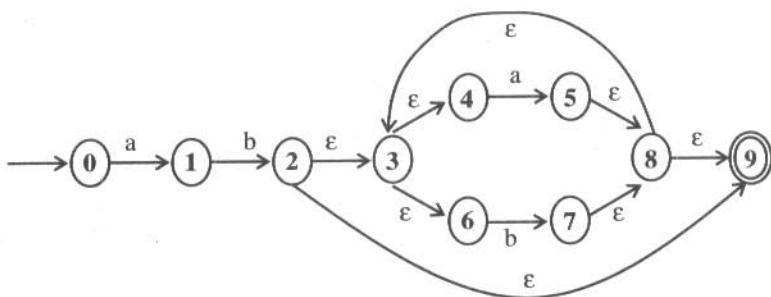
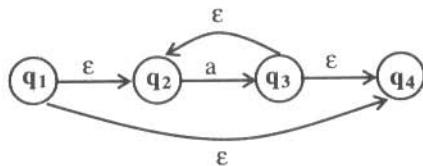


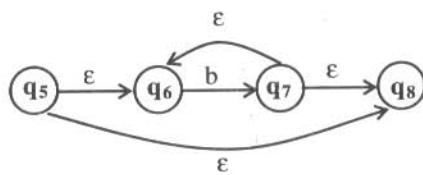
Figure 2.7. To accept the language $L(ab(a+b)^*)$

(2) Obtain an NFA for the regular expression $a^* + b^* + c^*$.

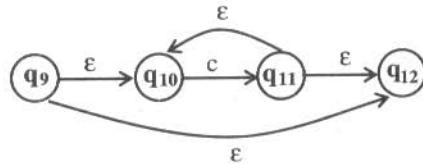
The machine corresponding the regular expression a^* can be written as



The machine corresponding the regular expression b^* can be written as



The machine corresponding the regular expression c^* can be written as



The machine corresponding the regular expression $a^* + b^* + c^*$ is shown in Figure 2.8.

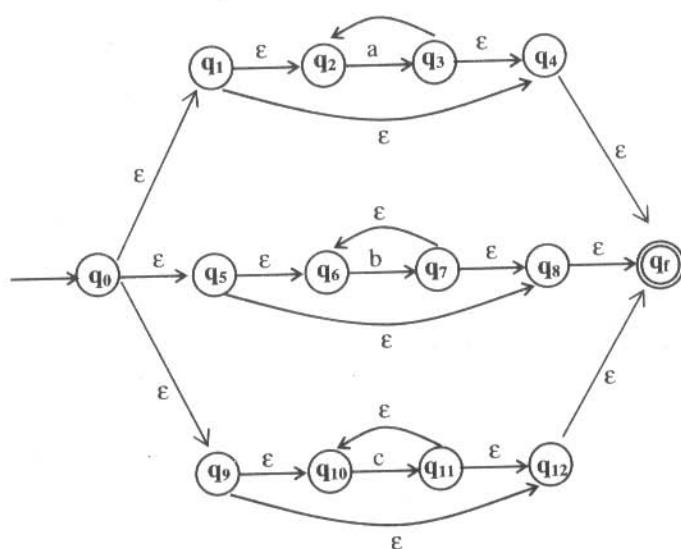
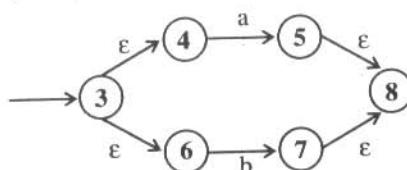


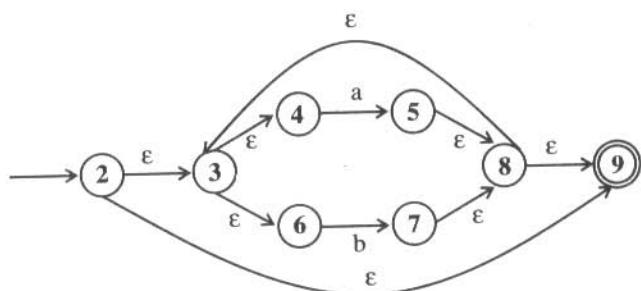
Figure 2.8. To accept the language $L(a^* + b^* + c^*)$

(3) Obtain an NFA for the regular expression $(a+b)^*aa(a+b)^*$.

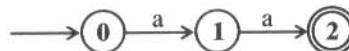
Step 1: The machine to accept $(a + b)$ and $(a+b)^*$ are shown below:



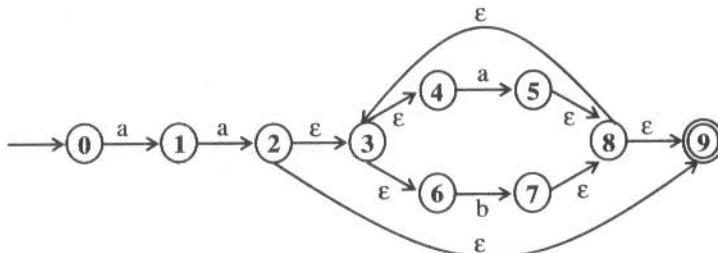
Step 2: The machine to accept $(a+b)^*$ is shown below:



Step 3: The machine to accept aa is shown below:



Step 4: The machine to accept aa(a+b)* is shown below:



Step 5: The machine to accept (a+b)*aa(a+b)* is shown in Figure 2.9.

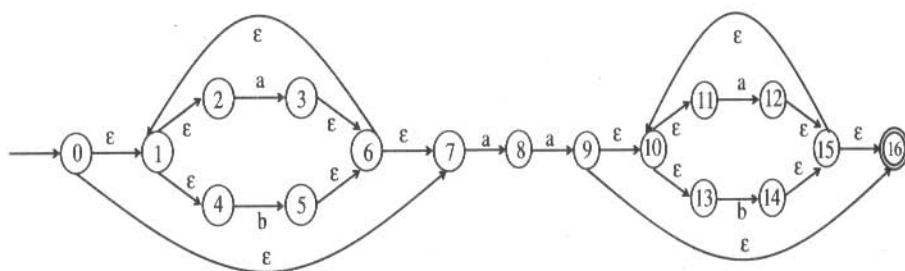


Figure 2.9. NFA to accept $(a+b)^*aa(a+b)^*$

2.6.2. To Obtain RE from FA (Kleene's Theorem)

In this section let us see how to obtain a regular expression from FA using Kleene's theorem.

Theorem: Let $M = (Q, \Sigma, \delta, q_1, F)$ be an FA recognizing the language L . Then there exists an equivalent regular expression R for the regular language L such that $L = L(R)$.

or

If the language L is accepted by a DFA then prove that there is a regular expression R for the language L so that $L = L(R)$

or

Prove that if $L = L(A)$ for some DFA A, then there is a regular expression R such that $L = L(R)$.

Note: The proof along with the procedure to obtain a regular expression from finite automaton is shown below:

Proof: Let $Q = \{q_1, q_2, \dots, q_n\}$ are the states of machine M where n is the number of states. The path from state to i to state j through an intermediate state whose number is not greater than k is given by the regular expression R_{ij}^k as shown below:

$$R_{ij}^k = \{w \in \Sigma^* \mid w \text{ is label (path) from } i \text{ to state } j \\ \text{that goes through an intermediate state} \\ \text{whose number is not greater than } k\}$$

where $i > k$ and $j > k$. The string w can be written as

$$w = xy$$

where $|x| > 0$ and $|y| > 0$ and $\hat{\delta}(i, x) = k$ and $\hat{\delta}(k, y) = j$.

Basis: $k = 0$. This indicates that there is no intermediate state and the path from state i to state j is given by the following two conditions:

1. There is a direct edge from state i to state j . This is possible when $i \neq j$. Here, a DFA M with all input symbols a such that there is a transition from state i to state j is considered with following cases:

- a) No input symbol and the corresponding regular expression is given by

$$R_{ij}^{(0)} = \emptyset$$

- b) Exactly one input symbol a on which there is a transition from state i to state j and the corresponding regular expression is given by

$$R_{ij}^{(0)} = a$$

- c) There are multiple inputs $a_1, a_2, a_3 \dots a_k$ where there is a transition from each symbol from state i to state j and the corresponding regular expression is given by

$$R_{ij}^{(0)} = a_1 + a_2 + a_3 + \dots + a_k$$

2. There is only one state such that $i = j$ and there exists a path from i to itself on input symbol a forming a self loop or a path of length 0 (i.e., if no loop path from state i to state i then there is a path of length 0) and is denoted by ϵ . Thus the regular expressions corresponding to various cases for 1.a, 1.b and 1.c is given by

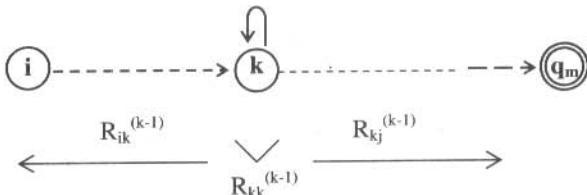
$$R_{ij}^{(0)} = \emptyset + \epsilon$$

$$R_{ij}^{(0)} = a + \epsilon$$

$$R_{ij}^{(0)} = a_1 + a_2 + a_3 + \dots + a_k + \epsilon$$

Induction: Suppose, there exists a path from i to j through a state which is not higher than k . This situation leads to two cases:

1. There exists a path from state i to state j which does not go through k and so the language accepted is $R_{ij}^{(k-1)}$.
2. There exists a path from state i to state j through k as shown below:



The path from state i to state j can be broken into several pieces as shown below:

1. The path from state i to state k and not passing through a state higher than k is given by

$$R_{ik}^{(k-1)}$$

2. The path from state k to state k and not passing through a state higher than k (may exist zero or more times) is given by

$$4. \quad (R_{kk}^{(k-1)})^*$$

3. The path from state k to state j and not passing through a state higher than k is given by

$$R_{kj}^{(k-1)}$$

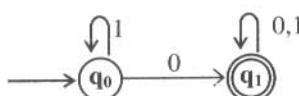
So, the regular expression for the path from state i to state j through no state higher than k is given by the concatenation of above 3 regular expressions as shown below:

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} (R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

By constructing the regular expressions in increasing order of subscripts, each $R_{ij}^{(k)}$ depends only on expressions with a smaller superscript and all the regular expressions are available whenever there is a need. Finally, we will have $R_{ij}^{(n)}$ for all i and j .

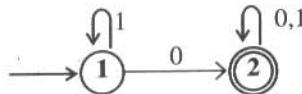
Note: Assume that state 1 is the initial state and the regular expression for the language is the sum of all regular expressions $R_{ij}^{(n)}$ where state j is an accepting state.

(1) Obtain a regular expression for the FA shown below:



What is the language corresponding to the regular expression?

Solution: Let $q_0 = 1$ and $q_1 = 2$. By renaming the states, the above FA can be written as shown below:



By following the procedure shown in Kleene's theorem (section 2.4) we have:

Basis: when $k = 0$

$$R_{11}^{(0)} = \epsilon + 1$$

$$R_{12}^{(0)} = 0$$

$$R_{21}^{(0)} = \phi$$

$$R_{22}^{(0)} = \epsilon + 0 + 1$$

Note: If the beginning and ending states are same add ϵ which denotes the length 0

Induction: The regular expression corresponding to the path from state i to state j through a state which is not higher than k is given by

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} [R_{kk}^{(k-1)}]^* R_{kj}^{(k-1)}$$

When $k = 1$ (i.e., path from state i to state j through a state not higher than 1): The various regular expressions obtained are shown below:

$$\begin{aligned} R_{11}^{(1)} &= R_{11}^{(0)} + R_{11}^{(0)} [R_{11}^{(0)}]^* R_{11}^{(0)} \\ &= (\epsilon + 1) + (\epsilon + 1)(\epsilon + 1)^*(\epsilon + 1) \\ &= (\epsilon + 1) + (\epsilon + 1)1^*(\epsilon + 1) \\ &= (\epsilon + 1) + 1^* \\ &= 1^* \end{aligned}$$

$$\begin{aligned} R_{12}^{(1)} &= R_{12}^{(0)} + R_{11}^{(0)} [R_{11}^{(0)}]^* R_{12}^{(0)} \\ &= 0 + (\epsilon + 1)(\epsilon + 1)^* 0 \\ &= 0 + 1^* 0 \\ &= 1^* 0 \end{aligned}$$

$$\begin{aligned} R_{21}^{(1)} &= R_{21}^{(0)} + R_{21}^{(0)} [R_{11}^{(0)}]^* R_{11}^{(0)} \\ &= \phi + \phi(\epsilon + 1)^*(\epsilon + 1) \\ &= \phi \end{aligned}$$

$$\begin{aligned} R_{22}^{(1)} &= R_{22}^{(0)} + R_{21}^{(0)} [R_{11}^{(0)}]^* R_{12}^{(0)} \\ &= (\epsilon + 0 + 1) + \phi(\epsilon + 1)^* 0 \\ &= (\epsilon + 0 + 1) \end{aligned}$$

When $k = 2$ (i.e., path from state i to state j through a state not higher than 2): The various regular expressions are given by

$$R_{11}^{(2)} = R_{11}^{(1)} + R_{12}^{(1)} [R_{22}^{(1)}]^* R_{21}^{(1)}$$

$$= 1^* + 1^* 0 (\epsilon + 0 + 1)^* \phi$$

$$= 1^*$$

$$R_{12}^{(2)} = R_{12}^{(1)} + R_{12}^{(1)} [R_{22}^{(1)}]^* R_{22}^{(1)}$$

$$= 1^* 0 + 1^* 0 (\epsilon + 0 + 1)^* (\epsilon + 0 + 1)$$

$$= 1^* 0 + 1^* 0 (0 + 1)^* (\epsilon + 0 + 1)$$

$$= 1^* 0 + 1^* 0 (0 + 1)^*$$

$$= 1^* 0 (0 + 1)^*$$

$$R_{21}^{(2)} = R_{21}^{(1)} + R_{22}^{(1)} [R_{22}^{(1)}]^* R_{21}^{(1)}$$

$$= \phi + (\epsilon + 0 + 1) (\epsilon + 0 + 1)^* \phi$$

$$= \phi$$

$$R_{22}^{(2)} = R_{22}^{(1)} + R_{22}^{(1)} [R_{22}^{(1)}]^* R_{22}^{(1)}$$

$$= (\epsilon + 0 + 1) + (\epsilon + 0 + 1) (\epsilon + 0 + 1)^* (\epsilon + 0 + 1)$$

$$= (\epsilon + 0 + 1) + (0 + 1)^*$$

$$= (0 + 1)^*$$

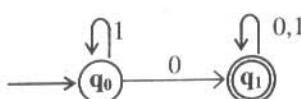
Note: Since the total number of states in the given machine is 2, maximum value of k should be 2.

Since the start state is 1 and final state is 2, the regular expression is given by

$$R_{12}^{(2)} = 1^* 0 (0 + 1)^*$$

So, the regular expression for the given DFA is $1^* 0 (0 + 1)^*$ which is the language consisting of any number of 1's followed by a zero and then followed by strings of 0's and 1's. This can also be expressed as strings of 0's and 1's with at least one zero.

(2) Obtain a regular expression for the FA shown below:



What is the language corresponding to the regular expression?

Note: The solution for this problem is already given in previous example. Another approach to solve this problem is that, instead of calculating the regular expressions for R_{ij} from $k = 0$ to n , start from $k = n$, and then work back to the case when $k = 0$.

In the current example, number of states $n = 2$ and hence to start with assume $k = 2$. The regular expression is given by

$$R_{12}^{(2)} = R_{12}^{(1)} + R_{12}^{(1)} [R_{22}^{(1)}]^* R_{22}^{(1)}$$

(1)

It is clear from the above expression that $R_{12}^{(1)}$ and $R_{22}^{(1)}$ are required and are obtained using the following regular expressions:

$$R_{12}^{(1)} = R_{12}^{(0)} + R_{11}^{(0)} [R_{11}^{(0)}]^* R_{12}^{(0)} \quad (2)$$

$$R_{22}^{(1)} = R_{22}^{(0)} + R_{21}^{(0)} [R_{11}^{(0)}]^* R_{12}^{(0)} \quad (3)$$

The regular expressions for these two equations can be obtained if we know $R_{11}^{(0)}$, $R_{12}^{(0)}$, $R_{21}^{(0)}$ and $R_{22}^{(0)}$ which are obtained when $k = 0$ as shown below:

When $k = 0$:

$$R_{11}^{(0)} = \epsilon + 1$$

$$R_{12}^{(0)} = 0$$

$$R_{21}^{(0)} = \phi$$

$$R_{22}^{(0)} = \epsilon + 0 + 1$$

Substituting these values in Eq. (2) and Eq. (3) we have,

$$\begin{aligned} R_{12}^{(1)} &= R_{12}^{(0)} + R_{11}^{(0)} [R_{11}^{(0)}]^* R_{12}^{(0)} \\ &= 0 + (\epsilon + 1)(\epsilon + 1)^* 0 \\ &= 0 + 1^* 0 \\ &= 1^* 0 \end{aligned}$$

$$\begin{aligned} R_{22}^{(1)} &= R_{22}^{(0)} + R_{21}^{(0)} [R_{11}^{(0)}]^* R_{12}^{(0)} \\ &= (\epsilon + 0 + 1) + \phi(\epsilon + 1)^* 0 \\ &= (\epsilon + 0 + 1) \end{aligned}$$

Substituting for $R_{12}^{(1)}$ and $R_{22}^{(1)}$ in Eq. (1) we have

$$\begin{aligned} R_{12}^{(2)} &= R_{12}^{(1)} + R_{12}^{(1)} [R_{22}^{(1)}]^* R_{22}^{(1)} \\ &= 1^* 0 + 1^* 0 (\epsilon + 0 + 1)^* (\epsilon + 0 + 1) \\ &= 1^* 0 + 1^* 0 (0 + 1)^* (\epsilon + 0 + 1) \\ &= 1^* 0 + 1^* 0 (0 + 1)^* \\ &= 1^* 0 (0 + 1)^* \end{aligned}$$

So, the regular expression for the given DFA is $1^* 0 (0 + 1)^*$ which is the language consisting of any number of 1's followed by a zero and then followed by strings of 0's and 1's. This can also be expressed as strings of 0's and 1's with at least one zero.

(3) Consider the DFA shown below:

States	Σ	
	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

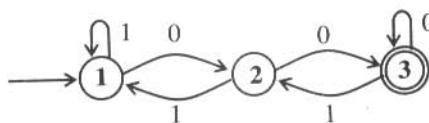
Obtain the regular expressions $R_{ij}^{(0)}$, $R_{ij}^{(1)}$ and simplify the regular expressions as much as possible.

Note: The symbol * preceding state q_3 indicates that q_3 is the final state.

Solution: Rename the states 1, 2 and 3 in order and the resulting DFA is shown below:

States	Σ	
	0	1
$\rightarrow 1$	2	1
2	3	1
*3	3	2

where state 3 is the final state and state 1 is the start state. The corresponding transition diagram is shown below:



By following the procedure shown in Kleene's theorem (Section 2.4) we have:

Basis: when $k = 0$

$$R_{11}^{(0)} = \epsilon + 1$$

$$R_{12}^{(0)} = 0$$

$$R_{13}^{(0)} = \phi$$

$$R_{21}^{(0)} = 1$$

$$R_{22}^{(0)} = \phi + \epsilon = \epsilon$$

$$R_{23}^{(0)} = 0$$

$$R_{31}^{(0)} = \phi$$

$$R_{32}^{(0)} = 1$$

$$R_{33}^{(0)} = \epsilon + 0$$

Note: If the beginning and ending states are same add ϵ which denotes the length 0 (i.e., whenever $i = j$)

Induction: The regular expression corresponding to the path from state i to state j through a state which is not higher than k is given by

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} [R_{kk}^{(k-1)}]^* R_{kj}^{(k-1)}$$

When k = 1 (i.e., path from state i to state j through a state not higher than 1): The various regular expressions obtained are shown below:

$$\begin{aligned} R_{11}^{(1)} &= R_{11}^{(0)} + R_{11}^{(0)} [R_{11}^{(0)}]^* R_{11}^{(0)} \\ &= (\epsilon + 1) + (\epsilon + 1)(\epsilon + 1)^*(\epsilon + 1) \\ &= (\epsilon + 1) + (\epsilon + 1)1^*(\epsilon + 1) \\ &= (\epsilon + 1) + 1^* \\ &= 1^* \end{aligned}$$

$$\begin{aligned} R_{12}^{(1)} &= R_{12}^{(0)} + R_{11}^{(0)} [R_{11}^{(0)}]^* R_{12}^{(0)} \\ &= 0 + (\epsilon + 1)(\epsilon + 1)^* 0 \\ &= 0 + 1^* 0 \\ &= 1^* 0 \end{aligned}$$

$$\begin{aligned} R_{13}^{(1)} &= R_{13}^{(0)} + R_{11}^{(0)} [R_{11}^{(0)}]^* R_{13}^{(0)} \\ &= \phi + (\epsilon + 1)^*(\epsilon + 1)\phi \\ &= \phi \end{aligned}$$

$$\begin{aligned} R_{21}^{(1)} &= R_{21}^{(0)} + R_{21}^{(0)} [R_{11}^{(0)}]^* R_{11}^{(0)} \\ &= 1 + 1(\epsilon + 1)^*(\epsilon + 1) \\ &= 1 + 11^* = 11^* \end{aligned}$$

$$\begin{aligned} R_{22}^{(1)} &= R_{22}^{(0)} + R_{21}^{(0)} [R_{11}^{(0)}]^* R_{12}^{(0)} \\ &= \epsilon + 1(\epsilon + 1)^* 0 \\ &= \epsilon + 11^* 0 \end{aligned}$$

$$\begin{aligned} R_{23}^{(1)} &= R_{23}^{(0)} + R_{21}^{(0)} [R_{11}^{(0)}]^* R_{13}^{(0)} \\ &= 0 + 1(\epsilon + 1)^*\phi \\ &= 0 \end{aligned}$$

$$\begin{aligned} R_{31}^{(1)} &= R_{31}^{(0)} + R_{31}^{(0)} [R_{11}^{(0)}]^* R_{11}^{(0)} \\ &= \phi + \phi(\epsilon + 1)^*(\epsilon + 1) \\ &= \phi \end{aligned}$$

$$\begin{aligned} R_{32}^{(1)} &= R_{32}^{(0)} + R_{31}^{(0)} [R_{11}^{(0)}]^* R_{12}^{(0)} \\ &= 1 + \phi(\epsilon + 1)^* 0 \\ &= 1 \end{aligned}$$

$$\begin{aligned} R_{33}^{(1)} &= R_{33}^{(0)} + R_{31}^{(0)} [R_{11}^{(0)}]^* R_{13}^{(0)} \\ &= (\epsilon + 0) + \phi(\epsilon + 0)^*\phi \\ &= (\epsilon + 0) \end{aligned}$$

When k = 2 (i.e., path from state i to state j through a state not higher than 2): The various regular expressions obtained are given by

$$\begin{aligned} R_{11}^{(2)} &= R_{11}^{(1)} + R_{12}^{(1)} [R_{22}^{(1)}]^* R_{21}^{(1)} \\ &= 1^* + 1^* 0(\epsilon + 11^* 0)^* 11^* \\ &= 1^* + 1^* 0(11^* 0)^* 11^* \end{aligned}$$

$$\begin{aligned} R_{12}^{(2)} &= R_{12}^{(1)} + R_{12}^{(1)} [R_{22}^{(1)}]^* R_{22}^{(1)} \\ &= 1^* 0 + 1^* 0(\epsilon + 11^* 0)^*(\epsilon + 11^* 0) \\ &= 1^* 0 + 1^* 0(11^* 0)^*(\epsilon + 11^* 0) \end{aligned}$$

$$\begin{aligned}
 R_{13}^{(2)} &= R_{13}^{(1)} + R_{12}^{(1)} [R_{22}^{(1)}]^* R_{23}^{(1)} \\
 &= \phi + 1^* 0 (\epsilon + 11^* 0)^* 0 \\
 &= 1^* 0 (11^* 0)^* 0 \\
 R_{21}^{(2)} &= R_{21}^{(1)} + R_{22}^{(1)} [R_{22}^{(1)}]^* R_{21}^{(1)} \\
 &= 11^* + (\epsilon + 11^* 0) (\epsilon + 11^* 0)^* 11^* \\
 &= 11^* + (\epsilon + 11^* 0) (11^* 0) 11^* \\
 R_{22}^{(2)} &= R_{22}^{(1)} + R_{22}^{(1)} [R_{22}^{(1)}]^* R_{22}^{(1)} \\
 &= (\epsilon + 11^* 0) + (\epsilon + 11^* 0) (\epsilon + 11^* 0)^* (\epsilon + 11^* 0) \\
 &= (\epsilon + 11^* 0) + (\epsilon + 11^* 0) (11^* 0)^* (\epsilon + 11^* 0) \\
 R_{23}^{(2)} &= R_{23}^{(1)} + R_{22}^{(1)} [R_{22}^{(1)}]^* R_{23}^{(1)} \\
 &= 0 + (\epsilon + 11^* 0) (\epsilon + 11^* 0)^* 0 \\
 &= 0 + (\epsilon + 11^* 0) (11^* 0)^* 0 \\
 R_{31}^{(2)} &= R_{31}^{(1)} + R_{32}^{(1)} [R_{22}^{(1)}]^* R_{21}^{(1)} \\
 &= \phi + 1(\epsilon + 11^* 0)^* 11^* \\
 &= 1(11^* 0)^* 11^* \\
 R_{32}^{(2)} &= R_{32}^{(1)} + R_{32}^{(1)} [R_{22}^{(1)}]^* R_{22}^{(1)} \\
 &= 1 + 1(\epsilon + 11^* 0)^* (\epsilon + 11^* 0) \\
 &= 1 + 1(11^* 0)^* (\epsilon + 11^* 0) \\
 R_{33}^{(2)} &= R_{33}^{(1)} + R_{32}^{(1)} [R_{22}^{(1)}]^* R_{23}^{(1)} \\
 &= (0 + \epsilon) + 1(11^* 0)^* 0
 \end{aligned}$$

The regular expression is given by R_{13} can be calculated as shown below:

$$\begin{aligned}
 R_{13}^{(3)} &= R_{13}^{(2)} + R_{13}^{(2)} [R_{33}^{(2)}]^* R_{33}^{(2)} \\
 &= 1^* 0 (11^* 0)^* 0 + 1^* 0 (11^* 0)^* 0 [(0 + \epsilon) + 1(11^* 0)^* 0]^* (0 + \epsilon) + 1(11^* 0)^* 0
 \end{aligned}$$

2.6.3. To Obtain RE from FA (By Eliminating States)

In this section let us see how to obtain a regular expression from FA.

Theorem: Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FA recognizing the language L . Then there exists an equivalent regular expression R for the regular language L such that $L = L(R)$.

The general procedure to obtain a regular expression from FA is shown below. Consider the generalized graph:

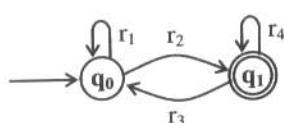


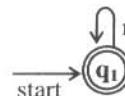
Figure 2.10. Generalized transition graph

where r_1 , r_2 , r_3 and r_4 are the regular expressions and correspond to the labels for the edges. The regular expression for this can take the form:

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^*(2.1)$$

Note:

- For each final state (accepting state) q , apply the reduction procedure and bring the graph to the form shown in Figure 2.10.
- If q is not the start state, the reduced graph obtained will be of the form shown in figure 2.10. and use the equation 2.1 to obtain the regular expression.
- If the start state is also an accepting state, the state elimination process has to be performed so that we should get rid of every state except the start state. The final automaton will be of the form



- The final regular expression is the sum of all the regular expressions obtained from the reduced automata for each accepting state.

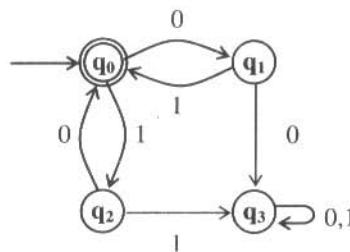
For example, If r_3 is not there in Figure 2.10, the regular expression can be of the form

$$r = r_1^* r_2 r_4^*(2.2)$$

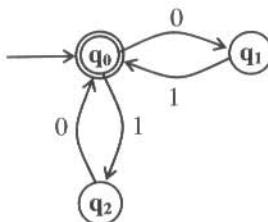
If q_0 and q_1 are the final states, then the regular expression can be of the form

$$r = r_1^* + r_1^* r_2 r_4^* (2.3)$$

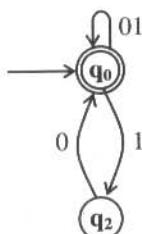
(1) Obtain a regular expression for the FA shown below:



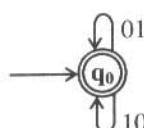
It is clear from the FA that q_3 is the dead state (i.e., once the state q_3 is reached irrespective of the input, the machine stays in q_3 only and there is no way to reach the final state) and so all the edges connected to q_3 can be removed and the resulting figure is shown below:



It is clear from the figure that if we input the string 01, the machine goes to state q_1 and comes back to q_0 and the process can be repeated. So, instead of q_1 , we can loop back on the string 01 as shown below:



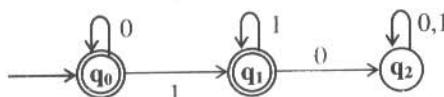
Similarly on 10, the machine comes back to q_0 and so we can replace it by another loop with the edge labeled 10 as shown:



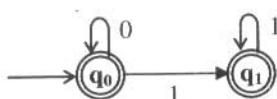
It is clear from this figure that the machine accepts strings of 01's and 10's of any length and the regular expression can be of the form

$$(01 + 10)^*$$

(2) What is the language accepted by the following FA.



Since, state q_2 is the dead state, it can be removed and the following FA is obtained.



The state q_0 is the final state and at this point it can accept any number of 0's which can be represented using notation as

$$0^*$$

q_1 is also the final state. So, to reach q_1 one can input any number of 0's followed by 1 and followed by any number of 1's and can be represented as

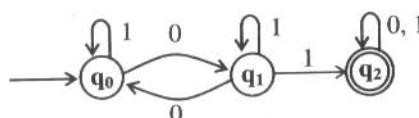
$$0^*11^*$$

So, the final regular expression is obtained by adding 0^* and 0^*11^* . So, the regular expression is

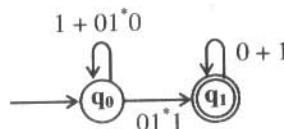
$$\begin{aligned} R.E &= 0^* + 0^*11^* \\ &= 0^*(\epsilon + 11^*) \\ &= 0^*(\epsilon + 1^*) \\ &= 0^*(1^*) = 0^*1^* \end{aligned}$$

It is clear from the regular expression that language consists of any number of 0's (possibly ϵ) followed by any number of 1's (possibly ϵ).

(3) Obtain a regular expression for the FA shown below:



The graph should be converted into generalized graph (shown in Figure 2.10) by eliminating state q_1 as shown below.



By comparing this figure with Figure 2.10, we have

$$\begin{aligned} r_1 &= 1 + 01*0 \\ r_2 &= 01*1 \\ r_3 &= \phi \\ r_4 &= 0 + 1 \end{aligned}$$

By substituting these in Eq. (2.1) we have

$$\begin{aligned} r &= (1 + 01*0)^* 01*1 [(0+1) + \phi(1 + 01*0)^*01*1]^* \\ &= (1 + 01*0)^* 01*1 [(0+1) + \phi]^* \\ &= (1 + 01*0)^* 01*1 (0+1)^* \end{aligned}$$

So, the regular expression for the given FA is $(1 + 01*0)^* 01*1 (0+1)^*$

2.7. Applications of Regular Expressions

Regular expressions in UNIX: Regular expressions are extensively used in UNIX operating system. But, certain short hand notations are used in UNIX platform using which complex regular expressions are avoided. For example, the symbol ‘.’ stands for any character, the sequence [abcde...] stands for the regular expression “a + b + c + d + e...”, the operator | is used in place of +, the operator ? means “zero or one of” etc. Most of the commands are invoked invariably uses regular expressions. For example, grep (Global search for Regular Expression and Print) used to search for a pattern of string.

Pattern Matching refers to a set of objects with some common properties. We can match an identifier or a decimal number or we can search for a string in the text.

Lexical analysis: Regular expressions are extensively used in the design of lexical analyzer phase (This is the first phase of the compiler design). This phase scans the source program and recognizes all the tokens which are logically together. The UNIX commands such as *lex* accepts regular expressions as the input and produces the lexical analyzer generator. This generator takes a high-level description of a lexical analyzer as the input and produces lexical analyzer.

Obtain a regular expression to identify an identifier.

An identifier starts with a letter. This letter can be followed by combination of zero or more letters and digits i.e., an identifier can be a single letter followed by strings of letters and digits of any length and can be represented as

letter (letter + digit)*

Obtain a regular expression to identify an integer.

An integer can start with any of the signs +, - or ε (null string means no sign) followed by one or more digits ranging from 0 to 9. This can be represented using a regular expression as

s d⁺ or sdd*

where *s* stands for sign and *d* stands for the digits from 0 to 9.

An application of regular expression in UNIX editor ed.

In UNIX operating system, we can use the editor *ed* to search for a specific pattern in the text. For example, if the command specified is

/acb*c/

then the editor searches for a string which starts with *ac* followed by zero or more *b*'s and followed by the symbol *c*. Note that the editor *ed* accepts the regular expression and searches for that particular pattern in the text. As the input can vary dynamically, it is challenging to write programs for string patterns of these kinds.

Regular Expressions in UNIX

A regular expression is a set of characters that specify a pattern. They are used to search for specific lines of text containing a particular pattern. The term “regular” is used to describe formal languages. Majority of the UNIX utilities operate on ASCII files one line at a time. The regular expressions will be used in majority of the editors to search for a specific pattern.

Now, let us see “What are the notations that are used in UNIX operation system?”. The various notations are shown below:

- .. A dot will match any single character except a newline character
- *,+ Star and plus are used to match zero/one or more of the preceding expressions
- ? Matches zero or one copy of the preceding expression
- | A logical ‘or’ statement - matches either the pattern before it, or the pattern after
- ^ Matches the very beginning of a line
- \$ Matched the end of a line
- / Matches the preceding regular expression, but only if followed by the subsequent expression
- [] Brackets are used to denote a character class, which matches any single character within the brackets. If the first character is a ‘^’, this negates the brackets causing them to match any character except those listed. The ‘-’ can be used in a set of brackets to denote a range. C escape sequences must use a ‘\’
- “ ” Match everything within the quotes literally
- () Group everything in the parentheses as a single unit for the rest of the expression

The various regular expressions are described below:

- | | |
|--------------|---|
| [0-9] | A single digit |
| [0-9]+ | An unsigned positive integer |
| [t\n]+ | Match white spaces such as tab, newline and space |
| [aeiouAEIOU] | Match the vowels |
| [a-zA-Z] | Match the consonants |
| “+”?[0-9]+ | Match any positive integer. The operator “+” preceding the symbol |

“?”	indicates that “+” is optional
-[0-9]+	Match any negative integer

Using the shorthand notations, the regular expressions can be built. For example, consider the definition of regular expression as shown below:

`exp ([eE]([-+])?[0-9]+)?`

Here, “exp” is the shorthand name for the regular expression

`([eE]([-+])?[0-9]+)?`

describing the exponent part of the fraction. The symbol “?” indicates that it is the optional part. Using the above shorthand notation, we can write the regular expression to identify the positive fraction as shown below:

`“+”?[0-9]*“.”[0-9]+{exp}`

Similarly, the regular expression to identify a negative fraction can be written as

`-[0-9]*“.”[0-9]+{exp}`

Without using the shorthand notation, the regular expressions to identify a positive fraction and a negative fraction can be written as shown below:

```
“+”?[0-9]*“.”[0-9]+{([eE]([-+])?[0-9]+)?} /* Identify a positive fraction */
-[0-9]*“.”[0-9]+{([eE]([-+])?[0-9]+)?} /* Identify a negative fraction */
```

We know that an identifier starts with a letter from “a” through “z” or “A” through “Z” and followed by any combination of letters or digits and can be represented using regular expression as shown below:

`[a-zA-Z]{1}[a-zA-Z]{0,}{0-9}*{1}`

The character sets can be combined by placing them next to each other. For example, if it is required to search for a word that

- Started with a capital letter "S" as the first word on a line
- The second letter was a lower case letter
- was exactly three letters long, and
- The third letter was a vowel

the regular expression would be “^S[a-z][aeiou]”. The following table gives the regular expressions and equivalent UNIX notation for the regular expressions.

Regular expressions	UNIX Notation
$(a+b)^*$	[ab]*
$(a+b)^*abb$	[ab]*"abb"
$ab(a+b)^*$	"ab"[ab]*
$(a+b)^*aa(a+b)^*$	[ab]*"aa"[ab]*
$a^*b^*c^*$	[a]*[b]*[c]*
$a^*b^+c^+$	[a]+[b]+[c]+
$aa^*bb^*cc^*$	[a][a]*[b][b]*[c][c]*
$(a+b)^* (a + bb)$	[ab]*[a(bb)]
$(aa)^*(bb)^*b$	[(aa)]*[(bb)*]"b"
$(0+1)^*000$	[01]*"000"
$(11)^*$	[(11)*]
$01^* + 1$	"0"[1]* 1

Exercises

1. Obtain the regular expressions for the following assuming $\Sigma = \{0, 1\}$:

- a. String containing even number of 0's
- b. String not ending with 001
- c. string containing at least one 0
- d. String containing exactly one 0
- e. String containing not more than 3 0's

2. Obtain the regular expressions for the following assuming $\Sigma = \{0, 1\}$:

a. $L = \{a^n b^m c^p \mid n \leq 4, m \geq 2, p \leq 2\}$

Ans: $(\epsilon + a + aa + aaa + aaaa) (bbb^*) (\epsilon + c + cc^*)$

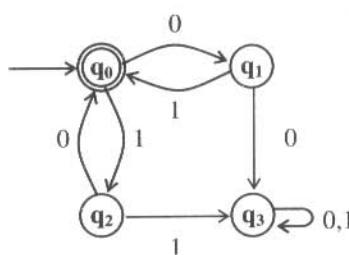
b. $L = \{a^{2n} b^{2m+1} \mid n \geq 0, m \geq 0\}$

Ans: $(aa)^*b(bb)^*$

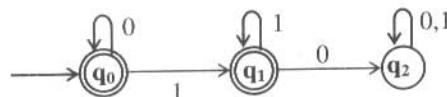
c. $L = \{w : |w| \bmod 3 = 0\}$

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

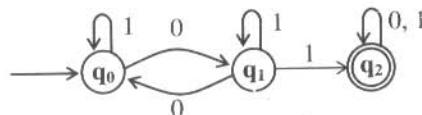
3. Find DFAs to accept the following languages:
- $L(00^* + 010^* 01)$
 - $L(0(0+1)^*11)$
 - $L(ab(a+ab)^* (a+aa))$
4. Construct an NFA to accept to accept the regular expression $(0+1)^*(00+11)(0+1)^*$.
5. Construct an NFA for the regular expression $10 + (0 + 11) 0^*1$ and obtain the corresponding DFA.
Also reduce the states of DFA.
6. What does the following regular expression imply:
- $a(a+b)^*ab$
 - $(0^*1 + 1^*0)^*0$
 - $0^* + (01+0)^*$
7. Explain what each of the regular expressions represent in English:
- $(a+b)^*aa(a+b)^*$
 - $a^*b^*c^*$
 - $a^*b^*c^*$
 - $aa^*bb^*cc^*$
 - $a+b)^* (a + bb)$
 - $(aa)^*(bb)^*b$
 - $(0+1)^*000$
8. Obtain a regular expression to accept a language consisting of strings of a's and b's of even length.
9. Obtain a regular expression to accept a language consisting of strings of a's and b's of odd length.
10. Obtain a regular expression such that $L(R) = \{w \mid w \in \{0, 1\}^*\}$ with at least three consecutive 0's.
11. Obtain a regular expression to accept strings of a's and b's ending with 'b' and has no sub string aa.
12. Obtain a regular expression for the FA shown below:



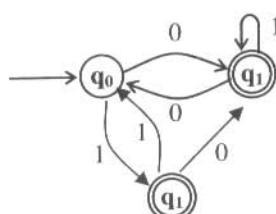
13. Obtain a regular expression to accept strings of 0's and 1's having no two consecutive zeros.
14. Obtain a regular expression to accept strings of a's and b's of length ≤ 10 .
15. How an NFA can be obtained from R.E? Give the procedure.
16. Obtain an NFA which accepts strings of a's and b's starting with the string ab.
17. Obtain an NFA for the regular expression $a^* + b^* + c^*$.
18. Obtain an NFA for the regular expression $(a+b)^*aa(a+b)^*$.
19. What is the language accepted by the following FA:



20. Obtain a regular expression for the FA shown below:



21. What are the applications of Regular expression.
22. Obtain a regular expression to identify an identifier.
23. Obtain a regular expression to identify an integer.
24. Discuss the application of regular expression for finding patterns in text.
25. Convert the following DFA to a regular expression using state-elimination technique:



Chapter 3

Regular Languages and Properties of Regular Languages

What are we studying in this chapter . . .

- ▶ *Regular languages*
- ▶ *Proving languages not to be regular languages*
- ▶ *Closure properties of regular languages*
- ▶ *Decision properties of regular languages*
- ▶ *Equivalence and minimization of automata*

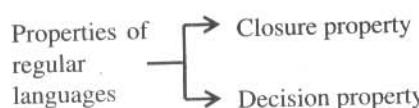
3.1. Proving Languages Not to be Regular

We have already described regular languages in previous chapters. For easy understanding let us see once again “What is a regular language?”

❖ **Definition:** The class of languages known as regular languages have four different descriptions. The regular languages are the languages accepted by DFA's, NFA's and ϵ -NFA's and defined by regular expressions. We know that each FA can be expressed in terms of other and all of them

accept the same languages. Thus, we can define regular languages as the languages accepted by DFA's or NFA's or ϵ -NFA's.

Once we know the definition of regular language, let us see “What are the various properties of regular languages?” The two important properties of regular languages are shown below:



- Closure property of regular languages: This property is a useful tool for building complex automata. Using the closure property we can build language recognizers. The closure property helps us to construct more complex finite automata.

For example, the intersection of two regular languages is also regular. So, if there are two automata recognizing two different languages, we can construct an automaton that recognizes the intersection of these two languages.

- Decision property of regular languages: Using this property, we can decide whether two automata define the same language. If so, we can minimize the states of automata with as few states as possible. The minimization of automaton is very important in the design of switching circuits. This is because, as the number of states of automaton decreases the size of the circuit and hence the cost decreases.

Any finite language can be expressed using regular expressions and we can construct finite automata (DFA or NFA or ϵ -NFA). Though the regular languages are important, there are non-regular languages which are very interesting and important. For example, following are some of the non-regular languages:

1. $L = \{w : w \in \{0, 1\}^*\text{ and has equal number of }0\text{'s and }1\text{'s}\}$
2. $L = \{0^n 1^n \text{ for } n \geq 0\}$
3. $L = \{a^p \text{ where } p \geq 2 \text{ is a prime number}\}$
4. Language consisting of matching parentheses

Apart from above four languages, there are so many languages which are not regular. Now, the question is “How to prove that certain languages are not regular?” We can prove that certain languages are not regular using one powerful tool called pumping lemma.

3.2. Pumping Lemma for Regular Languages

Now, let us “State and prove pumping lemma for regular languages”.

Theorem: Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FA and has n number of states. Let L be the regular language accepted by M . Let for every string x in L , there exists a constant n such that $|x| \geq n$. Now, if the string x can be broken into three substrings u, v and w such that

$$x = uvw$$

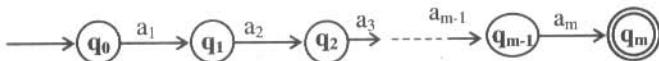
satisfying the following constraints:

- $v \neq \epsilon$ i.e., $|v| \geq 1$
- $|uv| \leq n$

then uv^iw is in L for $i \geq 0$

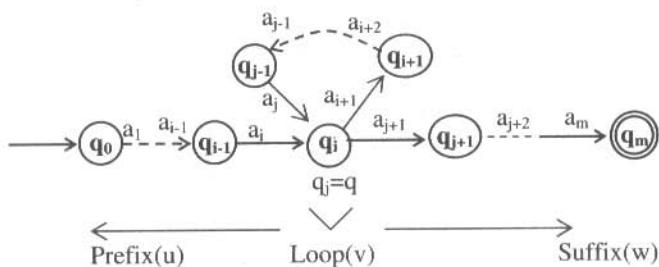
Proof: Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FA and Let L is the language accepted by DFA and is regular. Let $x = a_1a_2a_3\dots a_m$ where $m \geq n$ and each a_i is in Σ . Here, n represent the states of DFA.

Since we have m input symbols, naturally we should have $m+1$ states in the sequence $q_0, q_1, q_2, \dots, q_m$, where q_0 will be the start state and q_m will be the final state as shown below:



Since $|x| \geq n$, by the pigeon hole principle it is not possible to have distinct transitions. Once of the state can have a loop. Let the string x is divided into three substrings as shown below:

- The first group is the string prefix from $a_1a_2\dots a_i$ i.e., $u = a_1a_2\dots a_i$
- The second group is the loop string from $a_{i+1}a_{i+2}\dots a_{j-1}a_j$ i.e., $v = a_{i+1}a_{i+2}\dots a_{j-1}a_j$
- The third group is the string suffix from $a_{j+1}a_{j+2}\dots a_m$ i.e., $w = a_{j+1}a_{j+2}\dots a_m$



Observe from above figure that, the prefix string u takes the machine from q_0 to q_i , the loop string v takes the machine from q_i to q_j (Note: $q_i = q_j$) and suffix string w takes the machine from q_j to q_m . The minimum string that can be accepted by the above FA is uw with $i = 0$.

But, when $i=1$, the string uvw is accepted by DFA When $i = 2$, the string $uvvw$ is accepted by DFA. So, if $i > 0$, the machine goes from q_0 to q_i on input string u , circles from q_i to q_i based on

the value of i and then goes to accepting state on input string w . In general, if the string x is split into sub strings uvw , then for all $i \geq 0$,

$$uv^i w \in L$$

This can be expressed using the transitions as shown below:

$$\begin{aligned} \delta(q_0, a_1 a_2 \dots a_{i-1} a_i a_{j+1} a_{j+2} \dots a_m) \\ &= \delta(\delta(q_0, a_1 a_2 \dots a_{i-1} a_i), a_{j+1} a_{j+2} \dots a_m) \\ &= \delta(q, a_{j+1} a_{j+2} \dots a_m) \\ &= \delta(q_k, a_{k+1} a_{k+2} \dots a_m) \\ &= q_m \end{aligned}$$

Also, after the string

$$a_1 a_2 \dots a_i$$

the machine will be in state q_i . Since q_i and q_j are same, we can input the string

$$a_{i+1} a_{i+2} \dots a_j$$

any number of times and the machine will stay in q_i only. Finally, if the input string is

$$a_{j+1} a_{j+2} \dots a_m$$

the machine enters into final state q_m .

3.3. Applications of Pumping Lemma

Now, let us see “What are the applications of pumping lemma?” The pumping lemma is a very powerful tool and has the following applications:

- Pumping lemma is used to prove that certain languages are non-regular. But, it cannot be used to prove that a given language is regular.
- Using pumping lemma, it is possible to check whether a language accepted by FA is finite or infinite.

The general strategy used to prove that certain language is not regular is shown below.

Step 1: Assume that the language L is regular and n be the number of states of FA.

Step 2: Select the string x such that $|x| \geq n$ and break it into substrings u , v and w so that $x = uvw$ with the constraints:

- $v \neq \epsilon$ i.e., $|v| \geq 1$
- $|uv| \leq n$

Step 3: Find any i such that uv^iw is not in L i.e., $uv^iw \notin L$. According to pumping lemma, uv^iw is in L for $i \geq 0$. So, the result is contradiction to the assumption that the language is regular. Therefore, the given language L is not regular.

■ **Example 1:** Show that $L = \{ww^R \mid w \in (0+1)^*\}$ is not regular.

Step 1: Let L is regular and n be the number of states in FA. Consider the string:

$$x = \underbrace{1 \dots 1}_n \underbrace{0 \dots 0}_n \underbrace{0 \dots 0}_n \underbrace{1 \dots 1}_n$$

w w^R

where n is the number of states of FA, $w = 1\dots10\dots0$ and reverse of w is given by

$$w^R = 0\dots01\dots1.$$

Step 2: Since $|x| \geq n$, we can split the string x into uvw such that $|uv| \leq n$ and $|v| \geq 1$ as shown below:

$$x = \underbrace{1 \dots}_{u} \underbrace{1}_{v} \underbrace{0 \dots 0}_{w} \underbrace{0 \dots 0}_{w} \underbrace{1 \dots 1}_{w}$$

where $|u| = n-1$ and $|v| = 1$ so that $|uv| = |u| + |v| = n-1 + 1 = n$ which is true. According to pumping lemma, $uv^iw \in L$ for $i = 0, 1, 2, \dots$

Step 3: If $i = 0$, i.e., v does not appear and so the number of 1's on the left of x will be less than the number of 1's on the right of x and so the string is not of the form ww^R . So, $uv^iw \notin L$ when $i = 0$. This is a contradiction to the assumption that the language is regular.

So, the language $L = \{ww^R \mid w \in (0+1)^*\}$ is not regular.

■ **Example 2:** Show that $L = \{a^n b^n \mid n \geq 0\}$ is not regular.

Step 1: Let L is regular and n be the number of states in FA.

Consider the string $x = a^n b^n$.

Step 2: Since $|x| = 2n \geq n$, we can split x into uvw such that $|uv| \leq n$ and $|v| \geq 1$ as shown below:

$$x = \underbrace{aaaaaa}_n \underbrace{a}_v \underbrace{bbbbbb}_n$$

u v w

where $|u| = n-1$ and $|v| = 1$ so that $|uv| = |u| + |v| = n-1 + 1 = n$ and $|w| = n$. According to pumping lemma, $uv^i w \in L$ for $i = 0, 1, 2, \dots$

Step 3: If $i = 0$, the string v does not appear and so the number of a 's will be less than the number of b 's and so the string x does not have n number of a 's followed by n number of b 's. But, according to pumping lemma, n number of a 's should be followed by n number of b 's which is a contradiction to the assumption that the language is regular.

So, the language $L = \{a^n b^n \mid n \geq 0\}$ is not regular.

■ **Example 3:** Show that $L = \{a^i b^j \mid i > j\}$ is not regular.

Step 1: Let L is regular and n be the number of states in FA.

Consider the string $x = a^{n+1} b^n$.

Step 2: Since $|x| = 2n+1 \geq n$, we can split x into uvw such that $|uv| \leq n$ and $|v| \geq 1$ as shown below:

$$\begin{array}{ccccccc} x &= &a^{n+1}b^n &=& a^j & a^k & ab^n \\ &&\text{u}&&\text{v}&&\text{w} \end{array}$$

where $|u| = j$ and $|v| = k \geq 1$ and so that $|uv| = |u| + |v| = j+k \leq n$.

Step 3: According to pumping lemma, $uv^i w \in L$ for $i \geq 0$

i.e., $a^j (a^k)^i ab^n \in L$ for $i \geq 0$

Now, if we choose $i = 0$, number of a 's in string u will not be more than the number of b 's in w which is contradiction to the assumption that number of a 's are more than the number of b 's.

So, the language $L = \{a^i b^j \mid i > j\}$ is not regular.

■ **Example 4:** Show that $L = \{a^n b^l \mid n \neq l\}$ is not regular.

Solution: Similar to the previous solution.

■ **Example 5:** Show that $L = \{a^n b^l c^{n+l} \mid n, l \geq 0\}$ is not regular.

Step 1: Let L is regular and n be the number of states in FA.

Step 2: Since L is regular, it is closed under homomorphism. So, we can take

$$h(a) = a, \quad h(b) = a \quad \text{and} \quad h(c) = c$$

Now, the language L is reduced to

$$L = \{ a^n a^l c^{n+l} \mid n+l \geq 0 \}$$

which can be written as

$$L = \{ a^{n+l} c^{n+l} \mid n+l \geq 0 \}$$

which is of the form

$$L = \{ a^i b^i \mid i \geq 0 \}$$

We know that the above language is not regular (proved in Example 2) which is contradiction to the assumption that the language is regular. So, the given language

$$L = \{ a^n b^l c^{n+l} \mid n, l \geq 0 \}$$

is not regular.

Example 6: Show that $L = \{ a^{n!} \mid n \geq 0 \}$ is not regular.

Note: $n! = 1*2*3*...*n$.

Step 1: Let L is regular and n be the number of states in FA.

Step 2: Let $x = a^{n!}$. It is clear that $|x| \geq n$. So, we can split x into uvw such that $|uv| \leq n$ and $|v| \geq 1$ as shown below:

$$\begin{array}{ccccccc} x &= &a^{n!} &=& a^j & a^k & a^{n!-j-k} \\ &&\text{u}&&\text{v}&&\text{w} \end{array}$$

where $|u| = j$ and $|v| = k \geq 1$ and so that $|uv| = |u| + |v| = j+k \leq n$.

Step 3: According to pumping lemma, $uv^i w \in L$ for $i = 0, 1, 2, \dots$

i.e.,

$$a^j (a^k)^i a^{n!-j-k} \in L$$

So, if we choose $i = 0$, it means that

$$a^j a^{n!-j-k} \in L \text{ (i.e., } uw \in L)$$

\Downarrow (implies)

$$a^{n!-k} \in L$$

It is very clear that $n! > n! - k$. Now, when $k = 1$,

$$n! > n! - 1$$

But, according to Pumping Lemma $n! = n! - 1$ which is not and is a contradiction. So, L cannot be regular.

So, the language $L = \{a^{n!} \mid n \geq 0\}$ is not regular.

Example 7: Show that $L = \{0^n \mid n \text{ is prime}\}$ is not regular.

Note: The language generated from this can take the following form

$$L = \{00, 000, 00000, \dots\}$$

Step 1: Let L is regular and n be the number of states in FA. Let us choose the value of x which depends on n .

Let $x = 0^n \in L$ where n is prime.

Step 2: Note that $|x| = n$ and so, we can split x into uvw such that $|uv| \leq n$ and $|v| \geq 1$ as shown below.

$$\begin{matrix} x = 0^n = & 0^j & 0^k & 0^{n-j-k} \\ & u & v & w \end{matrix}$$

where $|u| = j$ and $|v| = k \geq 1$ and so that $|uv| = |u| + |v| = j+k \leq n$.

Step 3: According to pumping lemma, $uv^iw \in L$ for $i = 0, 1, 2, \dots$

i.e.,

$$0^j (0^k)^i 0^{n-j-k} \in L$$

i.e.,

$$j + ki + n - j - k = n + k(i-1) \text{ is prime for all } i \geq 0$$

Now, if we choose $i = n+1$, then

$$n + k(n+1) = n + kn = n(k+1)$$

is also a prime for each

$$k \geq 1$$

which is a contradiction (because if $k = 1$, it will not be a prime) to the assumption that the language is regular. So, the language $L = \{0^n \mid n \text{ is prime}\}$ is not regular.

Example 8: Show that $L = \{w \mid n_a(w) = n_b(w)\}$ is not regular. Is L^* regular?

Note: The language generated from this can take the following form

$$L = \{\epsilon, ab, ba, abab, aabb, bbaa, \dots\}$$

Step 1: Let L is regular and n be the number of states in FA. Let us choose the value of x which depends on n .

Let $x = a^n b^n \in L$

Step 2: Note that $|x| = 2n \geq n$ and so, we can split x into uvw such that $|uv| \leq n$ and $|v| \geq 1$ as shown below.

$$\begin{array}{cccc} x = a^n b^n & = & a^j & a^k b^n \\ & & u & v w \end{array}$$

where $|u| = j$ and $|v| = k \geq 1$ and so that $|uv| = |u| + |v| = j+k \leq n$.

Step 3: According to pumping lemma, $uv^i w \in L$ for $i = 0, 1, 2, \dots$

i.e.,

$$a^j (a^k)^i b^n \in L \text{ for } i = 0, 1, 2, \dots$$

Now, if we choose $i = 0$, number of a 's will be less than the number of b 's and if we choose $i = 2$, uv^2w will have more a 's than b 's which is contradiction to the assumption that it has equal number of a 's and b 's.

So, the language $L = \{ w \mid n_a(w) = n_b(w) \}$ is not regular. Since L is not regular L^* is also not regular.

■ **Example 9:** Show that $L = \{ w \mid n_a(w) < n_b(w) \}$ is not regular.

Step 1: Let L is regular and n be the number of states in FA. Let us choose the value of x which depends on n .

Let $x = a^{n-1} b^n \in L$

Step 2: Note that $|x| = 2n - 1 \geq n$ and so, we can split x into uvw such that $|uv| \leq n$ and $|v| \geq 1$ as shown below.

$$\begin{array}{cccc} x = a^{n-1} b^n & = & a^{n-1} & b^k b^{n-k} \\ & & u & v w \end{array}$$

where $|u| = n-1$ and $|v| = k$ and so that $|uv| = |u| + |v| = n-1 + k \leq n$. To satisfy this condition the value of k should be less than or equal to 1 (in this case).

Step 3: According to pumping lemma, $uv^i w \in L$ for $i = 0, 1, 2, \dots$

i.e.,

$$a^{n-1} (b^k)^i b^{n-k} \in L \text{ for } i = 0, 1, 2, \dots$$

Now, if we choose $i = 0$,

$$a^{n-1} b^{n-k} \in L$$

 (implies)

$$a^{n-1} b^{n-1} \in L \text{ (when } k = 1\text{)}$$

So, when $i = 0$, the number of a's and b's are same, which is contradiction to the assumption that the number of a's will be less than the number of b's. So, the language $L = \{ w \mid n_a(w) < n_b(w) \}$ is not regular.

■ **Example 10:** Show that $L = \{ ww \mid w \in \{a,b\}^* \}$ is not regular.

Step 1: Let L is regular and n be the number of states in FA. Let us choose the value of x which depends on n . Let $x = a^n b a^n b \in L$

Step 2: Note that $|x| = 2n + 2 \geq n$ and so, we can split x into uvw such that $|uv| \leq n$ and $|v| \geq 1$ as shown below:

$$\begin{array}{ccccccc} x & = & a^n & b & a^n & b \\ & & u & v & w \end{array}$$

where $|u| = j$ and $|v| = k \geq 1$ and so that $|uv| = |u| + |v| = j+k \leq n$.

Step 3: According to pumping lemma, $uv^i w \in L$ for $i = 0, 1, 2, \dots$

i.e.,

$$a^j (a^k)^i b a^n b \in L \text{ for } i = 0, 1, 2, \dots$$

Now, if we choose $i = 0$, number of a's on the left of first b will be less than the number of a's after the first b which is contradiction to the assumption that they are not equal. So, the language $L = \{ ww \mid w \in \{a,b\}^* \}$ is not regular.

■ **Example 11:** Show that $L = \{ a^n \mid n = k^2 \text{ for } k \geq 0 \}$ is not regular.

Note: The above language can also be defined as: $L = \{ a^n \mid n \text{ is a perfect square} \}$.

Step 1: Let L is regular and n be the number of states in FA. Let us choose the value of x which depends on n . Let $x = a^m \in L$ (where $m = n^2$).

Step 2: Note that $|x| \geq n$ and so, we can split x into uvw such that $|uv| \leq n$ and $|v| \geq 1$ as shown below.

$$\begin{array}{ccccccc} x & = & a^m & = & a^j & a^k & a^{m-j-k} \\ & & & & u & v & w \end{array}$$

where $|u| = j$ and $|v| = k \geq 1$ and so that $|uv| = |u| + |v| = j+k \leq n$.

Step 3: According to pumping lemma, $uv^iw \in L$ for $i = 0, 1, 2, \dots$

i.e.,

$$a^j a^{ki} a^{m-j-k} \in L \text{ for } i = 0, 1, 2, \dots$$

Now, if we choose $i = 2$, we have

$$uv^2w \in L$$

i.e.,

$$a^j a^{2k} a^{m-j-k} \in L$$

i.e.,

$$a^{m+k} \in L$$

Since $k \geq 1$, we have,

$$|a^{m+k}| = m + k = n^2 + k \text{ (since } m = n^2)$$

Note that

$$n^2 < n^2 + k < n^2 + 1 \text{ (when } k = 1) < (n^2 + 2n + 1) = (n+1)^2$$

and so

$$n^2 < (n+1)^2$$

Since n^2+k (nothing but $m+k$) lies between n^2 and $(n+1)^2$, it is not a perfect square which is contradiction to the assumption that it should be a perfect square (According to Pumping lemma). So, the language $L = \{ a^n \mid n = k^2 \text{ for } k \geq 0 \}$ is not regular.

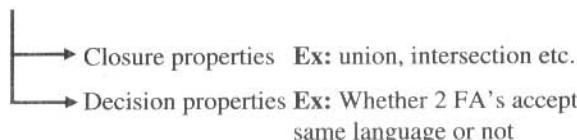
Exercises

1. Prove that the following languages are not regular:
 - i) $\{0^n 1^m 2^n \mid n, m \geq 1\}$
 - ii) $0^n 1^{2n} \mid n \geq 1\}$
2. Use pumping lemma and prove that the following language is not regular $L = \{a^n b^{n+1} \mid n > 1\}$.
3. Show that $L = \{0^n 10^n \mid n \geq 1\}$ is not regular.
4. Show that $L = \{0^n 1^m 2^n \mid n \text{ and } m \text{ are arbitrary integers}\}$ is not regular.
5. Show that $L = \{0^n 1^m \mid n \leq m\}$ is not regular.
6. Show that $L = \{0^n 1^{2n} \mid n \geq 1\}$ is not regular.
7. Show that $L = \{0^n \mid n \text{ is a perfect square}\}$ is not regular.

8. Show that $L = \{0^n \mid n \text{ is a perfect cube}\}$ is not regular page 96 of Pandey.
9. Show that $L = \{0^n \mid n \text{ is a power of 2}\}$ is not regular.

3.4. Properties of Regular Languages

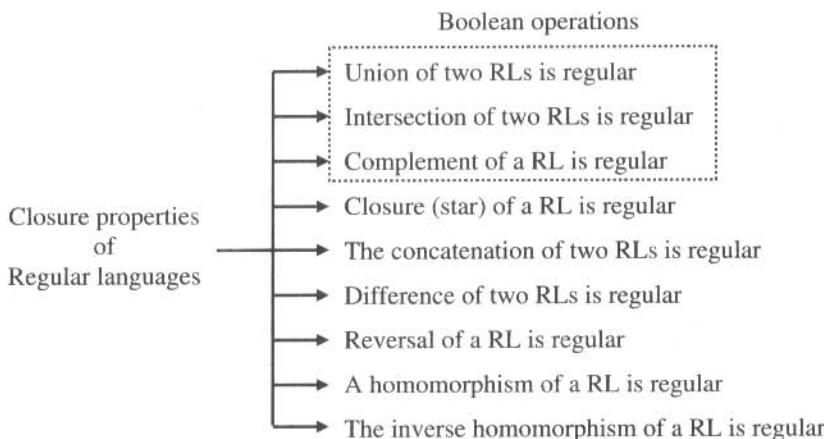
Now, let us see “What are various types of properties of regular languages?”. The regular languages exhibit two types of properties:



Now, let us see “What are closure properties?” Some new languages can be constructed from already existing languages using certain operations such as union, intersection, concatenation etc. We can build recognizers for these new languages using some of the properties of regular languages. These properties are called closure properties.

For example, the intersection of two regular languages is regular. Thus, given two finite automata that recognized two different languages, we can construct an automaton that recognizes the intersection of these two languages. The automaton thus obtained may have more states than either of the two automata. Thus using the closure properties we can build more complex automata.

Now, let us see “What are the various closure properties of regular languages?”. The closure properties of regular languages (RLs) are shown below:



Note: The first three closure properties: union, intersection and complement are called Boolean operations. The regular languages are closed under these three Boolean operations.

Note: RL stands for Regular Language

RLs stands for Regular Languages

3.4.1. Regular Languages are Closed Under Union, Concatenation and Star

In this section, let us “Show that if L_1 and L_2 are regular, then $L_1 \cup L_2$, $L_1.L_2$ and L_1^* are also regular”.

Theorem: If L_1 and L_2 are regular, then $L_1 \cup L_2$, $L_1.L_2$ and L_1^* also denote the regular languages.

Proof: It is given that L_1 and L_2 are regular languages. So, there exists regular expressions R1 and R2 such that:

$$L_1 = L(R1)$$

$$L_2 = L(R2)$$

By the definition of regular expressions, we have

- $R1 + R2$ is a regular expression denoting the language $L_1 \cup L_2$
- $R1 . R2$ is a regular expression denoting the language $L_1.L_2$
- $R1^*$ is a regular expression denoting the language L_1^*

So, the regular languages are closed under union, concatenation and star operations.

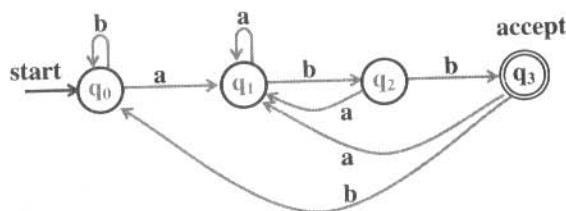
3.4.2. Closure Under Complementation

In this section, let us “Show that if L is a regular language, then complement of L denoted by \bar{L} is also regular”.

Theorem: If L is regular language, then the complement of L denoted by \bar{L} is also regular. In other words, the set of regular languages is closed under complementation.

Proof: Let $M_1 = (Q, \Sigma, \delta, q_0, F)$ be a DFA which accepts the language L . Since, the language is accepted by a DFA, the language is regular. Now, let us define the machine $M_2 = (Q, \Sigma, \delta, q_0, Q-F)$ which accepts \bar{L} . Note that there is no difference between M_1 and M_2 except the final states. The non-final states of M_1 are the final states of M_2 and final states of M_1 are the non-final states of M_2 . So, the language which is rejected by M_1 is accepted by M_2 and vice versa. Thus, we have a machine M_2 which accepts all those strings denoted by \bar{L} that are rejected by machine M_1 . So, a regular language is closed under complementation.

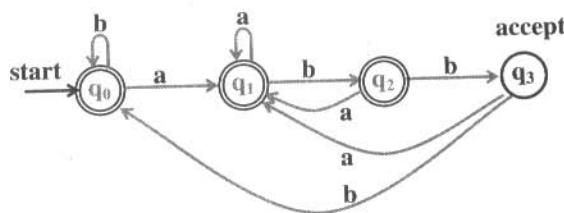
Example 1: The DFA to accept strings of a's and b's ending with abb is shown below (for details see Example 5):



According to the theorem, the regular languages are closed under complement. So, the complement of the given language is “Strings of a's and b's that do not end with *abb*”. Using the closure property of complement, the DFA is obtained by:

- Making the final states in above DFA as non-final states in new DFA
- Making non-final states in above DFA as final states in new DFA.

The resulting DFA accepting strings of a's and b's that do not end with *abb* is shown below:



3.4.3. Closure Under Intersection

Now, let us “Show that if L and M are regular languages, then so, is $L \cap M$ ”.

Theorem: If L_1 and L_2 are regular, then the regular language is closed under intersection.

Proof: Let us consider $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ which accepts L_1 and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ which accepts L_2 . It is clear from these two machines that the alphabets of both machines are same. Let us assume both the machines are DFAs. To accept the language $L_1 \cap L_2$, let us construct the machine M that simulates both M_1 and M_2 where the states of the machine M are the pairs (p, q) where $p \in Q_1$ and $q \in Q_2$. The transition for the machine M from the state (p, q) on input symbol $a \in \Sigma$ is the $(\delta_1(p, a), \delta_2(q, a))$ i.e., if $\delta_1(p, a) = r$ and $\delta_2(q, a) = s$, then the machine moves from the

state (p, q) to the state (r, s) on input symbol a . In this manner, the machine M can simulate the effect of M_1 and M_2 . Now, the machine $M = (Q, \Sigma, \delta, q, F)$ recognizes $L_1 \cap L_2$ where

$$Q = Q_1 \times Q_2$$

$q = (q_1, q_2)$ where q_1 and q_2 are the start states of machine M_1 and M_2 respectively

$$F = \{ (p, q) \mid p \in F_1 \text{ and } q \in F_2 \}$$

$$\delta: Q \times \Sigma \rightarrow Q \text{ is defined by } \delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$$

The string w is accepted if and only if

$$\hat{\delta}((q_1, q_2), w) \text{ is in } F$$

$$\text{i.e., } (\hat{\delta}_1(q_1, w), \hat{\delta}_2(q_2, w)) \text{ is in } F$$

This will happen if and only if

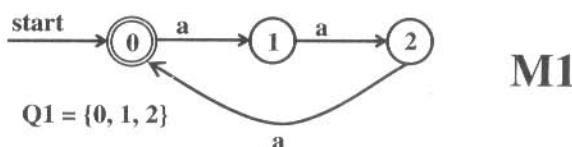
$$\hat{\delta}_1(q_1, w) \in F_1 \text{ and } \hat{\delta}_2(q_2, w) \in F_2$$

i.e., if and only if $w \in L_1 \cap L_2$. So, the regular language is closed under intersection.

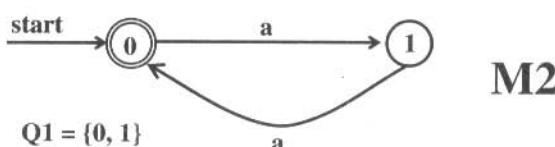
Example: Now, let us “Obtain a DFA to accept the following language”:

$$L = \{ w : |w| \bmod 3 = |w| \bmod 2 \text{ where } w \in \Sigma^* \text{ and } \Sigma = \{a\} \}$$

Solution: The DFA to accept a string w that results in $|w| \bmod 3$ can be written as shown below:



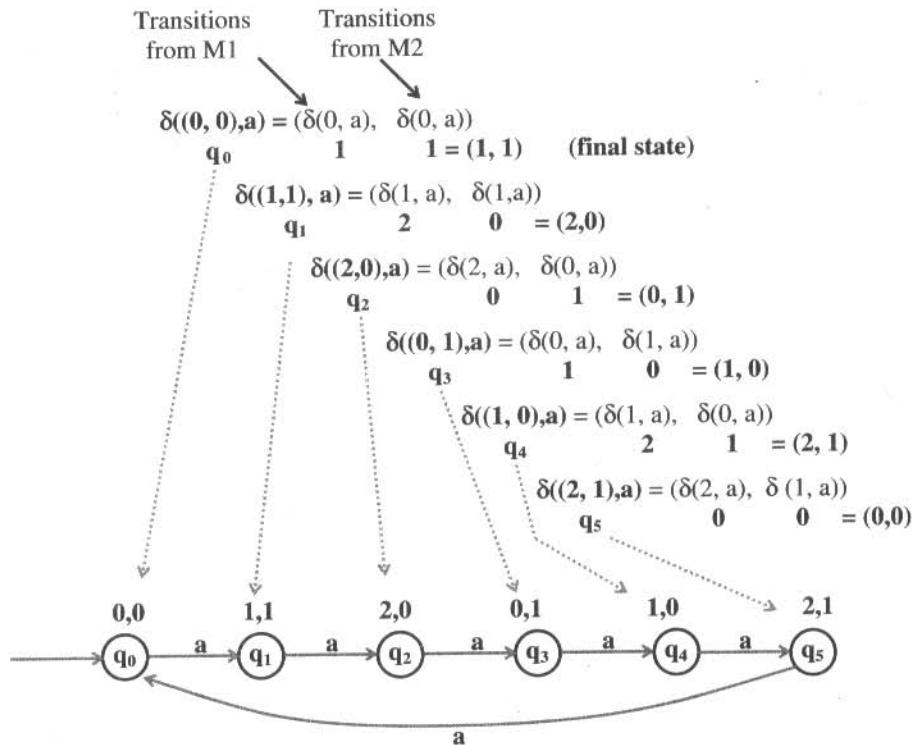
On similar lines, the DFA to accept a string w that results in $|w| \bmod 2$ can be written as shown below:



Transitions: Transitions of DFA which has strings of w with $|w| \bmod 3 = |w| \bmod 2$ can be obtained by taking the cross product of Q_1 and Q_2 as shown below:

$$Q_1 \times Q_2 = \{ (0,0), (0,1), (1,0), (1,1), (2,0), (2,1) \}$$

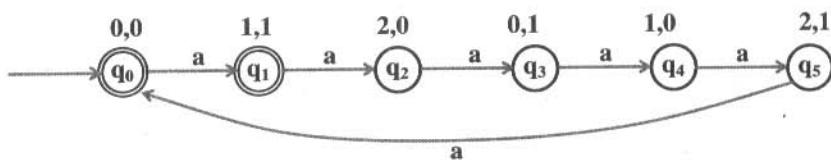
The transitions on each of the pair (x, y) can be obtained as shown below:



To accept strings of w such that $|w| \bmod 3 = |w| \bmod 2$, the pairs (x, y) such that $x = y$ are final states. So, in the above DFA, the final states are:

$$F = \{ (0, 0), (1, 1) \}$$

So, the DFA to accept the given language is shown below:



3.4.4. Closure Under Difference

In this section, let us “Show that if L_1 and L_2 are regular languages, the $L_1 - L_2$ is also regular”.

Theorem: If L_1 and L_2 are regular languages, then the regular language is closed under difference. In other words, if L_1 and L_2 are regular languages, the $L_1 - L_2$ is also regular.

Proof: Let us consider $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ which accepts L_1 and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ which accepts L_2 . We define $M = (Q, \Sigma, \delta, q, F)$ recognizing $L_1 - L_2$ as follows:

- Q is $Q_1 \times Q_2$ where (p, q) is in Q
- Σ is same for both the machines
- $\delta: Q \times \Sigma \rightarrow Q$ is defined by $\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$
- $q = (q_1, q_2)$ is the start state where q_1 is start state of M_1 and q_2 is start state of M_2
- $\delta: Q \times \Sigma \rightarrow Q$ is defined by $\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$
- $F = \{ (p, q) \mid p \in F_1 \text{ and } q \notin F_2 \}$

The string w is accepted if and only if

- $\hat{\delta}((q_1, q_2), w)$ is in F
- i.e., $(\hat{\delta}_1(q_1, w), \hat{\delta}_2(q_2, w))$ is in F .

This will happen if and only if

$$\hat{\delta}_1(q_1, w) \in F_1 \text{ and } \hat{\delta}_2(q_2, w) \notin F_2$$

i.e., if and only if w is in $L_1 - L_2$. So, the regular language is closed under difference.

3.4.5. Closure Under Reversal

Now, let us see “What is reversal of a string?”

❖ **Definition:** Let $w = a_1a_2a_3\dots a_n$. The reversal of a string w denoted by w^R is defined as a string which is written backwards i.e., $w^R = a_n a_{n-1} \dots a_3 a_2 a_1$. For example,

- Reversal of 0111 denoted by 0111^R is 1110
- Reversal of ϵ denoted by ϵ^R is ϵ

Now, let us see “What is reversal of a language?”

❖ **Definition:** The reversal of a language L is denoted by L^R . The reversal of a language L^R is defined as the set of all strings of L that are reversed. For example,

if

$$L = \{ \epsilon, 0, 10, 110, 11100 \}$$

then

$$L^R = \{\epsilon, 0, 01, 011, 00111\}$$

Note: Given a finite automata, the reversal of finite automata can be obtained by reversing all the arcs in the transition diagram.

Now, let us “Show that if the language L is regular, then L^R is also regular”.

Theorem: If L is regular, then L^R is also regular

Proof: Let L is the language corresponding to regular expression E . It is required to prove that there is another regular expression E^R such that

$$L(E^R) = (L(E))^R$$

which is read as “Language of E^R is the reversal of language of E ”.

Basis: By definition of regular expression E we have:

- ϕ is a regular expression
- ϵ is a regular expression
- a is a regular expression

So, the reversal of regular expression E^R is given by:

- $\{\epsilon\}^R = \{\epsilon\}$
- $\{\phi\}^R = \phi$
- $\{a\}^R = \{a\}$

Induction: Again, by definition of regular expressions, if E_1 and E_2 are regular expressions, then:

- $E_1 + E_2$ is a regular expression
- $E_1 \cdot E_2$ is a regular expression
- E_1^* is a regular expression

which results in three different cases. Now, let us prove each of these cases:

Case 1: $E = E_1 + E_2$. If $E = E_1 + E_2$ is a regular expression, then

$$E^R = E_1^R + E_2^R$$

is a regular expression denoting the languages:

$$L(E^R) = L(E_1^R) \cup L(E_2^R)$$

Case 2: $E = E_1 \cdot E_2$. If $E = E_1 \cdot E_2$ is a regular expression, then

$$E^R = E_2^R \cdot E_1^R$$

is a regular expression denoting the languages:

$$L(E^R) = L(E2^R) \cdot L(E1^R)$$

Case 3: $E = E1^*$. If $E = E1^*$ is a regular expression, then

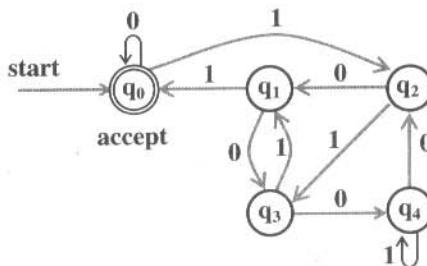
$$E^R = E1^R$$

is a regular expression denoting the languages:

$$L(E^R) = L(E1^R)$$

Example 1: Obtain a DFA to accept the set of all strings that when interpreted in reverse as a binary integer, is divisible by 5. For example, the string 1001100 should be accepted. This is because, even though 1001100 is not divisible by 5, its reversal 0011001 is divisible by 5.

Solution: The DFA shown below is obtained by reversing the arcs of the DFA shown in (Example 1) except the start state:



Note that the string 1001100 has to be accepted since its reversal is divisible by 5.

3.4.6. Closure Under Homomorphism

Now, let us see “What is homomorphism?”.

Definition: Let Σ and Γ are set of alphabets. The homomorphic function

$$h: \Sigma \rightarrow \Gamma^*$$

is called homomorphism i.e., a substitution where a single letter is replaced by a string. If

$$w = a_1 a_2 a_3 \dots a_n,$$

then

$$h(w) = h(a_1)h(a_2)\dots h(a_n)$$

If L is made of alphabets from Σ , then $h(L) = \{h(w) \mid w \in L\}$ is called homomorphic image.

Example: Let $\Sigma = \{0, 1\}$ $\Gamma = \{0, 1, 2\}$ and $h(0) = 01$, $h(1) = 112$. What is $h(010)$? If $L = \{00, 010\}$, what is homomorphic image of L ?

By definition we have

$$h(w) = h(a_1)h(a_2)\dots h(a_n)$$

So,

$$\begin{aligned} h(010) &= h(0)h(1)h(0) \\ &= 0111201 \\ L(00, 010) &= L(h(00), h(010)) \\ &= L(h(0)h(0), h(0)h(1)h(0)) \\ &= L(0101, 0111201) \end{aligned}$$

Therefore,

$$h(010) = 0111201$$

$$L(00, 010) = L(0101, 0111201)$$

Example: If $\Sigma = \{0, 1\}$ $\Gamma = \{1, 2, 3\}$, $h(0) = 3122$, $h(1) = 132$, what is $(0+1)^*(00)^*$?

By definition we have

$$h(w) = h(a_1)h(a_2)\dots h(a_n)$$

So,

$$\begin{aligned} (0+1)^*(00)^* &= (h(0) + h(1))^* (h(0)h(0))^* \\ &= (3122 + 132)^* (31223122)^* \end{aligned}$$

Prove that if the language L is regular over an alphabet Σ , and h is a homomorphism on Σ then $h(L)$ is regular.

Theorem: If L is regular and h is homomorphism, then homomorphic image $h(L)$ is regular.

Proof: Let R be the regular expression and $L(R)$ be the corresponding regular language. We can easily find $h(R)$ by substituting $h(a)$ for each a in Σ . By definition of regular expression, $h(R)$ is a regular expression and so $h(L)$ is regular language. So, the regular language is closed under homomorphism.

3.5. Limitations of Finite Automaton

Note: There are so many problems for which we can not construct a DFA and still we want some solution to solve those problems. For example,

1. Check for the matching parentheses (not possible using DFA).
2. Count number of a's and then the number of b's (not possible using DFA) and so it can not be used as a counter.

Now, let us see “What are the various limitations of finite automaton?”. The limitations of finite automaton are:

- An FA has finite number of states and so it does not have the capacity to remember arbitrary long amount of information.
- Since it does not have memory, FA can not remember a long string. For example, to check for matching parentheses, check whether the string is a palindrome or not etc. are not possible using FA.
- Finite automata or finite state machines have trouble recognizing various types of languages involving counting, calculating, storing the string.

The solution for all these problems is to have a machine which is more general than DFA which can recognize these set of languages. In the coming chapters we discuss how these problems can be solved using Grammars, push down automaton, Turing machines, etc.

Example: Suppose $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and we are interested in the language L of all strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ is in F and also for every state q in Q there is some prefix x_q of w such that $\hat{\delta}(q_0, x_q) = q$. Is L regular? Prove your answer

Proof: It is given that $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA. It is also given that

$$\hat{\delta}(q_0, w) \text{ is in } F.$$

This imply that from state q_0 on input string w , the machine enters into final state accepting the string w . Let the final state be q_f . This can be represented as shown below:

$$\hat{\delta}(q_0, w) = q_f \quad (1)$$

It is required to prove that $\hat{\delta}(q, w) = \hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$. From this statement it is observed that:

$$w = xa$$

where

- a is the last symbol of w
- x is the remaining string of w .
- q is the current state of the machine.

From state q on input string w let us assume that the machine enters into state p . This is given by the transition:

$$\hat{\delta}(q, w) = p(1)$$

Since $w = xa$, the above transition can be written as shown below:

$$\hat{\delta}(q, xa) = p(2)$$

Since $w = xa$, first, we should find the transition on string x and then on symbol a .

On string x : From state q on input string x , let the machine enters into state r . This is denoted by:

$$\hat{\delta}(q, x) = r(3)$$

On symbol a : From state r on input symbol a , the machine should enter into state p . This is denoted by:

$$\delta(r, a) = p$$

Replacing the state r in above transition using Eq. (3), we have:

$$\delta(\hat{\delta}(q, x), a) = p(4)$$

Comparing Equations (2) and (4), observe that R.H.S.s are equal. Therefore, L.H.S.s are also equal. So,

$$\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$$

$$\text{i.e., } \hat{\delta}(q, w) = \hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$$

3.6. Equivalence and Minimization of Finite Automata

The decision properties of regular languages give us algorithms for deciding whether two automata define the same language. If the two automata accept the same language, we can minimize the automata that have as few states as possible. The minimization of finite automata is

very important in the design of switching circuits. This is because, as the number of states of the automaton implemented by the circuit decreases, the cost of the circuit tends to decrease.

3.6.1. Equivalence of Two States

The language generated by a DFA is unique. But, there can exist many DFA's that accept the same language. In such cases, the DFA's are said to be equivalent. During computations, it is desirable to represent the DFA with fewer states since the space is proportional to the number of states of DFA. For storage efficiency, it is required to reduce the number of states and hence it is required to minimize the DFA. This can be achieved first by finding the distinguishable and indistinguishable states. First, let us see "What are distinguishable and indistinguishable states?".

❖ **Definition:** Two states p and q of a DFA are *equivalent (indistinguishable)* if and only if $\delta(p, w)$ and $\delta(q, w)$ are final states or both $\delta(p, w)$ and $\delta(q, w)$ are non-final states for all $w \in \Sigma^*$ i.e. if

$$\delta(p, w) \in F \text{ and } \delta(q, w) \in F$$

then the states p and q are *indistinguishable*. If

$$\delta(p, w) \notin F \text{ and } \delta(q, w) \notin F$$

then also the states p and q are *indistinguishable*. If there is at least one string w such that one of

$$\delta(p, w) \text{ and } \delta(q, w)$$

is final state and the other is non-final state, then the states p and q are not equivalent and are called *distinguishable* states.

Note: The *distinguishable* and *indistinguishable* states can be obtained using *table-filling algorithm* (also called *mark* procedure).

3.6.2. Table Filling Algorithm

Now, let us see "What is table filling algorithm?" The table filling algorithm is used to find the set of states that are distinguishable and indistinguishable states. The algorithm is recursively defined as shown below:

Step 1: Identify the initial markings: For each pair (p, q) where $p \in Q$ and $q \in Q$, if $p \in F$ and $q \notin F$ or vice versa then, the pair (p, q) is distinguishable and mark the pair (p, q) [by putting say 'x' for the pair (p,q)].

Step 2: Identify the subsequent markings: For each pair (p, q) and for each $a \in \Sigma$, find $\delta(p, a) = r$ and $\delta(q, a) = s$. If the pair (r, s) is already marked as distinguishable then the pair (p, q) is also distinguishable and mark it as say ‘x’. Repeat step 2 until no previously unmarked pairs are marked.

Note: If the pair (p, q) obtained using the above *table filling algorithm* (mark procedure) are indistinguishable then the two states p and q are equivalent and they can be merged into one state thus minimizing the states of DFA.

3.6.3. Minimization of DFA (Algorithm or Procedure)

Once we have found the distinguishable and indistinguishable pairs we can easily minimize the number of states of a DFA and accepting the same language as accepted by original DFA. Now, let us “Write the algorithm or the procedure to minimize a DFA using table filling algorithm”.

The algorithm to minimize the DFA is shown below:

Step 1: Find the distinguishable and indistinguishable pairs: using the *table-filling* algorithm (discussed in previous section).

Step 2: Obtain the states of minimized DFA: These groups consist of indistinguishable pairs obtained in previous step and individual distinguishable states. The groups obtained are the states of minimized DFA.

Step 3: Compute the transition table: If $[p_1, p_2, \dots, p_k]$ is a group and if $\delta([p_1, p_2, \dots, p_k], a) = [r_1, r_2, \dots, r_m]$ then place an edge from the group $[p_1, p_2, \dots, p_k]$ to the group $[r_1, r_2, \dots, r_m]$ and label the edge with the symbol a . Follow this procedure for each group obtained in step 2 and for each $a \in \Sigma$.

Step 4: Identify the start state: If one of the component in the group $[p_1, p_2, \dots, p_k]$ consists of a start state of given DFA then $[p_1, p_2, \dots, p_k]$ is the start state of minimized DFA.

Step 5: Identify the final state: If the group $[p_1, p_2, \dots, p_k]$ contains a final state of given DFA then the group $[p_1, p_2, \dots, p_k]$ is final state of minimized DFA.

■ **Example 1:** Obtain the distinguishable table for the automaton and then minimize the states of following DFA.

δ	a	b
A	B	F
B	G	C
*C	A	C
D	C	G
E	H	F
F	C	G
G	G	E
H	G	C

Solution: The DFA can be minimized using table filling algorithm as shown below:

The various states of given DFA are: A, B, C, D, E, F, G, H.

Step 1: Obtain the various pairs of states: The various pairs from the above states can be obtained by drawing the table shown below:

- Vertically, we write all the states from second state to the last state. In this case, we write the states B, C, D, E, F, G and H vertically (see table below)
- Horizontally, we write all the states from first state to last but one state. In this case, we write the states A, B, C, D, E, F and G (see table below)
- Then draw the vertical and horizontal lines as shown in the table below:

*	B						
	C						
	D						
	E						
	F						
	G						
	H						
	A	B	C	D	E	F	G

Initial Horizontal markings: Since state C is the final state, the pairs (A,C) and (B,C) has one final state and other non-final state. So, we mark the pairs (A,C) and (B,C) horizontally (see the marking x horizontally).

Initial Vertical markings: Since state C is the final state, the pairs (H,C), (G,C), (F,C), (E,C), (D,C) has one final state and non-final state. So, they are marked vertically (see the marking x vertically) as shown in table below:

Horizontal markings $\rightarrow *$

B						
C	x	x				
D			x			
E			x			
F			x			
G			x			
H			x			
	A	B	C	D	E	F

*

Step 2: For each pair (p, q) and for each $a \in \Sigma$ find $\delta(p, a) = r$ and $\delta(q, a) = s$. If the pair (r, s) is already marked as distinguishable then the pair (p, q) is also distinguishable and mark it as say ‘x’. The various markings can be obtained as shown below:

δ	a	b
(A,B)	(B,G)	(F,C)
(A,D)	(B,C)	(F,G)
(A,E)	(B,H)	(F,F)
(A,F)	(B,C)	(F,G)
(A,G)	(B,G)	(F,E)
(A,H)	(B,G)	(F,C)
(B,D)	(G,C)	(C,G)
(B,E)	(G,H)	(C,F)
(B,F)	(G,C)	(C,G)
(B,G)	(G,G)	(C,E)
(B,H)	(G,G)	(C,C)
(D,E)	(C,H)	(G,F)
(D,F)	(C,C)	(G,G)
(D,G)	(C,G)	(G,E)
(D,H)	(C,G)	(G,C)
(E,F)	(H,C)	(F,G)
(E,G)	(H,G)	(F,E)
(E,H)	(H,G)	(F,C)
(F,G)	(C,G)	(G,E)
(F,H)	(C,G)	(G,C)
(G,H)	(G,G)	(E,C)

(F,C) is marked. So, mark (A, B)
 (B,C) is marked. So, mark (A, D)

 (B,C) is marked. So, mark (A, F)

 (F, C) is marked. So, mark (A,H)
 (G, C) is marked. So, mark (B,D)
 (C, F) is marked. So, mark (B,E)
 (G, C) is marked. So, mark (B, F)
 (C, E) is marked. So, mark (B, G)

 (C, H) is marked. So, mark (D, E)

 (C, G) is marked. So, mark (D, G)
 (C, G) is marked. So, mark (D,H)
 (H, C) is marked. So, mark (E, F)

 (F,C) is marked. So, mark (E,H)
 (C, G) is marked. So, mark (F, G)
 (C, G) is marked. So, mark (F, H)
 (E, C) is marked. So, mark (G, H)



The pairs shown in black background are to be marked

B							
C	x	x					
D			x				
E	x						
F	x						
G	x						
H	x						
*	A	B	C	D	E	F	G

The resulting marking table



B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	
*	A	B	C	D	E	F	G

Again consider the pairs (p,q) which are not marked in the above table and see whether the corresponding pairs (r, s) on 0 and 1 are marked as shown below:

(p,q)	a	b
(p,q)	(r,s)	(r,s)
(A,E)	(B,H)	(F,F)
(A,G)	(B,G)	(F,E)
(B,H)	(G,G)	(C,C)
(D,F)	(C,C)	(G,G)
(E,G)	(H,G)	(F,E)

Mark the pairs (A,G) and (E,G) with 'x' as shown below:

B	x					
C	x	x				
D	x	x	x			
E	x	x	x	x		
F	x	x	x	x	x	
G	x	x	x	x	x	x
H	x	x	x	x	x	x
A	B	C	D	E	F	G

Indistinguishable pairs:
(A,E), (B,H) and (D,F)

Distinguishable pairs:
C and G

Minimizing DFA: The DFA can be minimized as shown below:

Step 1: Find the distinguishable and indistinguishable pairs:

(A,E), (B,H) and (D,F) are in-distinguishable

C, G are distinguishable (see previous page)

Step 2: Obtain the states of minimized DFA: The states of the minimized DFA

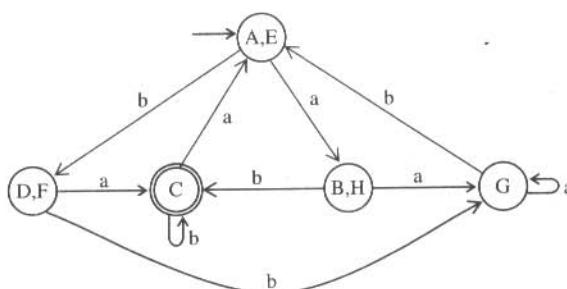
(A,E), (B,H), C, (D,F), G

Step 3: Compute the transition table: Use the transition table of given DFA and obtain the transitions for the minimized DFA as shown below:

δ	a	b
(A,E)	(B,H)	(D,F)
(B,H)	G	C
*C	(A,E)	C
(D,F)	C	G
G	G	(A,E)

Step 4: Identify the start state: Since the group (A,E) has the start state of DFA, the group (A, E) is the start state of minimized DFA.

Step 5: Identify the final state: The group C is the final state of given DFA. So, C will be the final state of minimized DFA. The transition diagram of the minimized DFA is shown below:



Example 2: Find the minimized DFA for the following:

δ	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

Solution: The DFA can be minimized using table filling algorithm as shown below:

The various states of given DFA are: A, B, C, D, E, F, G, H.

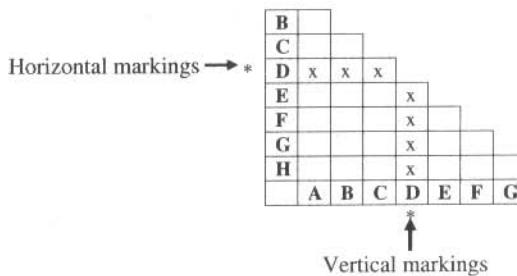
Step 1: Obtain the various pairs of states: The various pairs from the above states can be obtained by drawing the table shown below:

- Vertically, we write all the states from second state to the last state. In this case, we write the states B, C, D, E, F, G and H vertically (see table below)
- Horizontally, we write all the states from first state to last but one state. In this case, we write the states A, B, C, D, E, F and G (see table below)
- Then draw the vertical and horizontal lines as shown in the table below:

B							
C							
*							
D							
E							
F							
G							
H							
	A	B	C	D	E	F	G
*							

Initial Horizontal markings: Since state D is the final state, the pairs (A,D), (B,D) and (C,D) has one final state and other non-final state. So, we mark the (A,D), (B,D) and (C,D) horizontally (see the marking x horizontally).

Initial Vertical markings: Since state D is the final state, the pairs (D,E), (D,F), (D,G) and (D,H) has one final state and non-final state. So, they are marked vertically (see the marking x vertically as shown in table below):



Step 2: For each unmarked pair (p, q) and for each $a \in \Sigma$ find

$$\delta(p, a) = r \text{ and } \delta(q, a) = s.$$

If the pair (r, s) is already marked as distinguishable then the pair (p, q) is also distinguishable and mark it as say 'x'. The various markings can be obtained as shown below:

δ	a	b
(A,B)	(A,B)	(A,C)
(A,C)	(B,D)	(A,B)
(A,E)	(B,D)	(A,F)
(A,F)	(B,G)	(A,E)
(A,G)	(B,F)	(A,G)
(A,H)	(B,G)	(A,D)
(B,C)	(A,D)	(C,B)
(B,E)	(A,D)	(C,F)
(B,F)	(A,G)	(C,E)
(B,G)	(A,F)	(C,G)
(B,H)	(A,G)	(C,D)
(C,E)	(D,D)	(B,F)
(C,F)	(D,G)	(B,E)
(C,G)	(D,F)	(B,G)
(C,H)	(D,G)	(B,D)
(E,F)	(D,G)	(F,E)
(E,G)	(D,F)	(F,G)
(E,H)	(D,G)	(F,D)
(F,G)	(G,F)	(E,G)
(F,H)	(G,G)	(E,D)
(G,H)	(F,G)	(D,G)

(B,D) is marked. So, mark (A, C)
(B,D) is marked. So, mark (A, E)

(A,D) is marked. So, mark (A, H)
(A,D) is marked. So, mark (B,C)
(A,D) is marked. So, mark (B,E)

(C,D) is marked. So, mark (B,H)

(D,G) is marked. So, mark (C,F)
(D,F) is marked. So, mark (C,G)
(D,G) is marked. So, mark (C,H)
(D,G) is marked. So, mark (E,F)
(D,F) is marked. So, mark (E,G)
(D,G) is marked. So, mark (E,H)

(E,D) is marked. So, mark (F,H)
(D,G) is marked. So, mark (G,H)



The pairs shown in
black background
are to be marked

* 

B							
C							
D	x	x	x				
E			x				
F			x				
G			x				
H			x				
	A	B	C	D	E	F	G

*

The resulting
marking table



* 

B							
C	x	x					
D	x	x	x				
E	x	x		x			
F			x	x	x		
G			x	x	x		
H	x	x	x	x	x	x	x
	A	B	C	D	E	F	G

*

Again consider the pairs (p,q) which are not marked in the above table and see whether the corresponding pairs (r,s) on 0 and 1 are marked as shown below:

	a	b	
(p,q)	(r,s)	(r,s)	
(A,B)	(B,A)	(A,C)	(A,C) is marked. So, mark (A,B)
(A,F)	(B,G)	(A,E)	(A,E) is marked. So, mark (A,F)
(A,G)	(B,F)	(A,G)	
(B,F)	(A,G)	(C,E)	
(B,G)	(A,F)	(C,G)	(C,G) is marked. So, mark (B,G)
(C,E)	(D,D)	(B,F)	
(F,G)	(G,F)	(E,G)	(E,G) is marked. So, mark (F,G)

Mark the pairs (A,B), (A,F), (B,G) and (F,G) with 'x' as shown below:

B	x						
C	x	x					
D	x	x	x				
E	x	x		x			
F	x		x	x	x		
G		x	x	x	x	x	
H	x	x	x	x	x	x	x
A	B	C	D	E	F	G	

Indistinguishable pairs:
(A,G), (B,F) and (C,E)

Distinguishable states:
D, H

Minimizing DFA: The DFA can be minimized as shown below:

Step 1: Find the distinguishable and indistinguishable pairs:

(A,G), (B,F) and (C,E) are indistinguishable
D,H are distinguishable (see previous page)

Step 2: Obtain the states of minimized DFA: The states of the minimized DFA

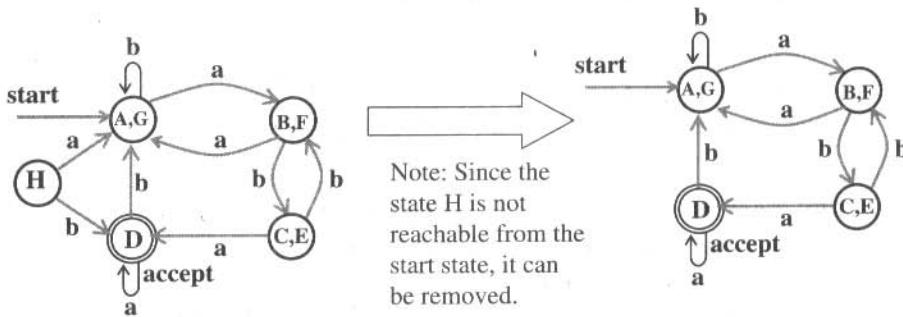
(A,G), (B,F), (C,E), D, H

Step 3: Compute the transition table: Use the transition table of given DFA and obtain the transitions for the minimized DFA as shown below:

δ	a	b
(A,G)	(B,F)	(A,G)
(B,F)	(A,G)	(C,E)
(C,E)	D	(B,F)
*D	D	(A,G)
H	(A,G)	D

Step 4: Identify the start state: Since the group (A,G) has the start state of DFA, the group (A, G) is the start state of minimized DFA.

Step 5: Identify the final state: The group D is the final state of given DFA. So, D will be the final state of minimized DFA. The transition diagram of the minimized DFA is shown below:



Example 3: Minimize the following DFA using table-filling algorithm where A is the start state. The states C F and I are final states.

δ	0	1
A	B	E
B	C	F
*C	D	H
D	E	H
E	F	I
*F	G	B
G	H	B
H	I	C
*I	A	E

Solution: The DFA can be minimized using table filling algorithm as shown below:

The various states of given DFA are: A, B, C, D, E, F, G, H, I.

Step 1: Obtain the various pairs of states: The various pairs from the above states can be obtained by drawing the table shown below:

- Vertically, we write all the states from second state to the last state. In this case, we write the states B, C, D, E, F, G, H and I vertically (see table below).
 - Horizontally, we write all the states from first state to last but one state. In this case, we write the states A, B, C, D, E, F, G and H (see table below).
 - Then draw the vertical and horizontal lines as shown in the table below:

*	B							
*	C							
*	D							
*	E							
*	F							
*	G							
*	H							
*	I							
	A	B	C	D	E	F	G	I
	*			*		*		*

Initial Horizontal markings: The final states are C, F and I.

- For the final state C, the pairs (A,C) and (B,C) has one final state and other non-final state. So, the pairs (A,C) and (B,C) are marked horizontally.
 - For the final state F, the pairs (A, F), (B,F), (D,F), (E,F) has one final state and other non-final state. So, the pairs (A, F), (B,F), (D,F), (E,F) are marked horizontally.
 - For the final state I, the pairs (A,I), (B,I), (D,I), (E,I),(G,I), (H,I) has one final state and other non-final state. So, the pairs (A,I), (B,I), (D,I), (E,I),(G,I), (H,I) are marked horizontally.

Initial Vertical markings: The vertical markings can be obtained as shown below:

- For state C, the pairs (D,C), (E,C),(G,C) and (H,C) has one final state and other non-final state. So, the pairs (D,C), (E,C),(G,C) and (H,C) are marked vertically.
 - For the state F, the pairs (G,F) and (H,F) has one final state and other non-final state. So, the pairs (G,F) and (H,F) are marked vertically.

The initial horizontal and vertical markings are shown below:

*	B								
*	C	x	x						
*	D			x					
*	E			x					
*	F	x	x		x	x			
*	G			x			x		
*	H			x			x		
*	I	x	x		x	x		x	x
	A	B	C	D	E	F	G	H	
	*			*			*		

Step 2: For each pair (p, q) and for each $a \in \Sigma$ find $\delta(p, a) = r$ and $\delta(q, a) = s$. If the pair (r, s) is already marked as distinguishable then the pair (p, q) is also distinguishable and mark it as say 'x'. The various markings can be obtained as shown below:

δ	a	b
(A,B)	(B,C)	(B,F)
(A,D)	(B,E)	(B,H)
(A,E)	(B,F)	(B,I)
(A,G)	(B,H)	(B,B)
(A,H)	(B,I)	(B,C)
(B,D)	(C, E)	(F,H)
(B,E)	(C, F)	(F,I)
(B,G)	(C, H)	(F,B)
(B,H)	(C, I)	(F,C)
(D,E)	(E,F)	(H,I)
(D,G)	(E,H)	(H,B)
(D,H)	(E,I)	(H,C)
(E,G)	(F,H)	(I,B)
(E,H)	(F,I)	(I,C)
(H,G)	(I,H)	(C,B)

(B,C) is marked. So, mark (A,B)

(B,F) is marked. So, mark (A,E)

(B,C) is marked. So, mark (A,H)

(C,E) is marked. So, mark (B,D)

(C,H) is marked. So, mark (B,G)

(H,I) is marked. So, mark (D,E)

(E,I) is marked. So, mark (D,H)

(F,H) is marked. So, mark (E,G)

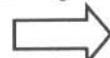
(C,B) is marked. So, mark (H,G)



The pairs shown in black background are to be marked

*	B							
*	C	x	x					
*	D			x				
*	E			x				
*	F	x	x		x	x		
*	G			x			x	
*	H			x		x		
*	I	x	x		x	x		x
*	A	B	C	D	E	F	G	H

The resulting marking table



*	B	x						
*	C	x	x					
*	D		x	x				
*	E	x		x	x			
*	F	x	x		x	x		
*	G		x	x		x	x	
*	H	x		x	x		x	x
*	I	x	x		x	x		x
*	A	B	C	D	E	F	G	H

So, the final markings in the table are shown below:

*	B	x						
*	C	x	x					
*	D		x	x				
*	E	x		x	x			
*	F	x	x		x	x		
*	G		x	x		x	x	
*	H	x		x	x		x	x
*	I	x	x		x	x		x
*	A	B	C	D	E	F	G	H

Indistinguishable pairs:

$$(A,D) (A,G) = (A,D,G)$$

$$(D,G)$$

$$(B,E) (B,H) = (B, E, H)$$

$$(E,H)$$

$$(C,F) (C,I) = (C,F,I)$$

$$(F,I)$$

Distinguishable states: does not exist

Minimizing DFA: The DFA can be minimized as shown below:

Step 1: Find the distinguishable and indistinguishable pairs:

$$(A,D,G), (B,E,H) \text{ and } (C,F,I) \text{ are in-distinguishable}$$

Step 2: Obtain the states of minimized DFA: The states of the minimized DFA:

$$(A,D,G), (B,E,H) \text{ and } (C,F,I)$$

Step 3: Compute the transition table: Use the transition table of given DFA and obtain the transitions for the minimized DFA as shown below:

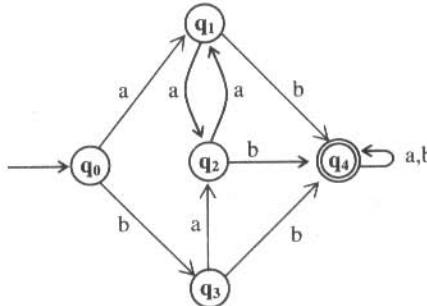
δ	0	1
$\rightarrow (A,D,G)$	(B,E,H)	(B,E,H)
(B,E,H)	(C,F,I)	(C,F,I)
$*(C,F,I)$	(A,D,G)	(B,E,H)

Step 4: Identify the start state: Since the group (A,D,H) has the start state of DFA, the group (A,D,G) is the start state of minimized DFA.

Step 5: Identify the final state: The states C,F and I are the final states of given DFA. So, the group (C,F,I) will be the final state of minimized DFA. The transition table of the minimized DFA is shown below:

δ	0	1
$\rightarrow (A,D,G)$	(B,E,H)	(B,E,H)
(B,E,H)	(C,F,I)	(C,F,I)
$*(C,F,I)$	(A,D,G)	(B,E,H)

Example 4: Minimize the following DFA.



Solution: First let us obtain the table for inequalities for the states (distinguishable or indistinguishable states). This we can do by knowing the various states of given DFA. The states of the DFA are:

$q_0 \quad q_1 \quad q_2 \quad q_3 \quad q_4$

The pairs can be obtained as shown below: (This can be obtain by writing all the pairs in the matrix form and take only the upper triangular matrix to avoid duplicate elements and cycles)

- $(q_0, q_1)(q_0, q_2)(q_0, q_3)(q_0, q_4)$
- $(q_1, q_2)(q_1, q_3)(q_1, q_4)$
- $(q_2, q_3)(q_2, q_4)$
- (q_3, q_4)

The above pairs can be represented in the tabular form as shown below and note that no pairs are marked.

q_1				
q_2				
q_3				
q_4				
	q_0	q_1	q_2	q_3

Step 1: For each pair (p, q) where $p \in Q$ and $q \in Q$, if $p \in F$ and $q \notin F$ or vice versa then, the pair (p, q) is distinguishable and mark the pair (p, q) [by putting say 'x' for the pair (p,q)]. This clearly indicates that one of the pair should be a final state and the other should be a non final state. So, the various pairs which satisfy the above criteria are all the pairs involving the final state q_4 and are marked as shown below:

q_1				
q_2				
q_3				
q_4	x	x	x	
	q_0	q_1	q_2	q_3

Step 2: For each pair (p, q) and for each $a \in \Sigma$ find $\delta(p, a) = r$ and $\delta(q, a) = s$. If the pair (r, s) is already marked as distinguishable (Note: The distinguished states are already marked as 'x' in the first step) then the pair (p, q) is also distinguishable and mark it as say 'x'. Repeat step 2 until no previously unmarked pairs are marked.

Now consider the pairs (p, q) which are not marked in step 1 and obtain the corresponding pairs (r, s) on input symbols 0 and 1 as shown below:

δ	a	b
(p,q)	(r,s)	(r,s)
(q_0, q_1)	(q_1, q_2)	(q_3, q_4)
(q_0, q_2)	(q_1, q_1)	(q_3, q_4)
(q_0, q_3)	(q_1, q_2)	(q_3, q_4)
(q_1, q_2)	(q_2, q_1)	(q_4, q_4)
(q_1, q_3)	(q_2, q_2)	(q_4, q_4)
(q_2, q_3)	(q_1, q_2)	(q_4, q_4)

Note: If the pairs (r, s) on a and b are marked in step 1, the corresponding pair (p, q) should be marked. For example, for the pair (q_0, q_1) on b is (q_3, q_4) and is marked as 'x' in step 1. So, mark the pair (q_0, q_1) (shown using shading). On similar lines identify the pairs (r, s) which are marked in step 1 (shown in bold phase) and take the corresponding pairs (p, q) (shown in shading) and update the marking table as shown below.

q_1	x			
q_2	x			
q_3	x			
q_4	x	x	x	x
q_0	q_1	q_2	q_3	

Again consider the pairs (p, q) which are not marked in the above table and see whether the corresponding pairs (r, s) on a and b are marked as shown below:

δ	a	b
(p,q)	(r,s)	(r,s)
(q_1, q_2)	(q_2, q_1)	(q_4, q_4)
(q_1, q_3)	(q_2, q_2)	(q_4, q_4)
(q_2, q_3)	(q_1, q_2)	(q_4, q_4)

None of the unmarked pairs (r, s) are marked and final table is shown below:

q_1	x			
q_2	x			
q_3	x			
q_4	x	x	x	x
q_0	q_1	q_2	q_3	

Observe that the pairs (q_1, q_2) , (q_1, q_3) and (q_2, q_3) are not at all marked and are considered to be indistinguishable states. Note that the state q_1 is present in the pairs (q_1, q_2) , (q_1, q_3) and the state q_2 is present in the pairs (q_1, q_2) , (q_2, q_3) . So, the states q_1 , q_2 , q_3 together forms a group consisting of indistinguishable states and can be represented as (q_1, q_2, q_3) . So, the indistinguishable groups are

(q_1, q_2, q_3)

and distinguishable states are

(q_0) and (q_4)

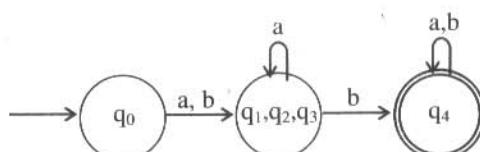
Step 3: So, the various groups that represent the states of minimized DFA are:

(q_0) (q_1, q_2, q_3) (q_4)

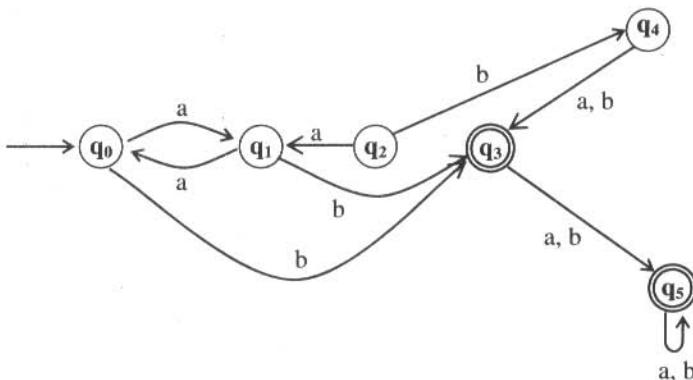
Step 4: The minimized transition table is shown below. Note that the start state of original DFA is the start state of minimized DFA and the group which consists of a final state of original DFA is the final state of minimized DFA.

States	Σ	
	a	b
$\rightarrow (q_0)$	(q_1, q_2, q_3)	(q_1, q_2, q_3)
(q_1, q_2, q_3)	(q_1, q_2, q_3)	(q_4)
(q_4)	(q_4)	(q_4)

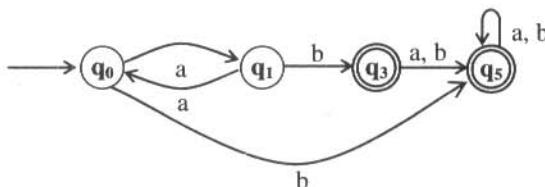
The transition diagram obtained after minimization is shown below:



Example 5: Minimize the following DFA.



Note: It is clear from the above that the states q_2 and q_4 are not reachable from the start state and they can be removed. The resulting DFA after removing those states is shown below:



First let us obtain the table for inequalities for the states (distinguishable or indistinguishable states). This we can do by knowing the various states of given DFA. The states of the DFA are:

$q_0 \quad q_1 \quad q_3 \quad q_5$

The pairs can be obtained as shown below: (This can be obtained by writing all the pairs in the matrix form and take only the upper triangular matrix to avoid duplicate elements and cycles)

(q_0, q_1) (q_0, q_3) (q_0, q_5)
 (q_1, q_3) (q_1, q_5)
 (q_3, q_5)

The above pairs can be represented in the tabular form as shown below and note that no pairs are marked.

q_1			
q_3			
q_5			
	q_0	q_1	q_3

Step 1: For each pair (p, q) where $p \in Q$ and $q \in Q$, if $p \in F$ and $q \notin F$ or vice versa then, the pair (p, q) is distinguishable and mark the pair (p, q) [by putting say 'x' for the pair (p, q)]. This clearly indicates that one of the pair should be a final state and the other should be a non final state. So, the various pairs which satisfy the above criteria are all the pairs involving the one of the final states q_3 or q_5 i.e., (q_0, q_3) , (q_0, q_5) , (q_1, q_3) and (q_1, q_5) and are marked as shown below:

q_1			
q_3	x	x	
q_5	x	x	
	q_0	q_1	q_3

Step 2: For each pair (p, q) and for each $a \in \Sigma$ find $\delta(p, a) = r$ and $\delta(q, a) = s$. If the pair (r, s) is already marked as distinguishable (Note: The distinguished states are already marked as 'x' in the first step) then the pair (p, q) is also distinguishable and mark it as say 'x'. Repeat step 2 until no previously unmarked pairs are marked.

Now consider the pairs (p, q) which are not marked in step 1 and obtain the corresponding pairs (r, s) on input symbols a and b as shown below:

δ	a	b
(p, q)	(r, s)	(r, s)
(q_0, q_1)	(q_1, q_0)	(q_3, q_3)
(q_3, q_5)	(q_5, q_5)	(q_5, q_5)

Note: If the pairs (r, s) on a and b are marked in step 1, the corresponding pair (p, q) should be marked. But, the pairs (r, s) on a and b are not marked in the previous step and the final table that provides the inequalities between the various states are shown below:

q_1			
q_3	x	x	
q_5	x	x	
	q_0	q_1	q_3

Observe that the pairs (q_0, q_1) and (q_3, q_5) are not at all marked and are considered to be indistinguishable states.

Step 3: Finding the states of minimized DFA. It is clear from the table obtained in step 2 that the pairs (q_0, q_1) and (q_3, q_5) are indistinguishable and the various groups that represent the states of minimized DFA are:

$$(q_0, q_1) (q_3, q_5)$$

Step 4: The minimized transition table is shown below. Note that the start state of minimized DFA is the group which has the start state of DFA to be minimized and the group which consists of a final state of original DFA is the final state of minimized DFA.

States	Σ	
	a	b
$\rightarrow(q_0, q_1)$	(q_0, q_1)	(q_3, q_5)
$^*(q_3, q_5)$	(q_3, q_5)	(q_3, q_5)

Exercises

1. Show that the regular languages are closed under:
a. Union b. Concatenation c. Star closure
2. Show that the regular languages are closed under complementation.
3. Show that the regular languages are closed under intersection.
4. Show that the regular languages are closed under difference.
5. What is homomorphism? Explain with example.
6. Show that the regular languages are closed under homomorphism.
7. What are the languages accepted by FA and what are not accepted by FA? Specify some languages which are not accepted by FA.
8. What is Pigeonhole principle?
9. State and prove Pumping Lemma.
10. What is Pumping Lemma? Why it is used?
11. What are the applications of Pumping Lemma?
12. What is the general strategy used in Pumping Lemma for proving certain languages are not regular?
13. Show that $L = \{ww^R \mid w \in (0+1)^*\}$ is not regular.
14. Show that $L = \{a^n b^n \mid n \geq 0\}$ is not regular.
15. Show that $L = \{a^n b^l \mid n \neq l\}$ is not regular.
16. Show that $L = \{a^i b^j \mid i > j\}$ is not regular.
17. Show that $L = \{a^n b^l c^{n+l} \mid n, l \geq 0\}$ is not regular.

18. Show that $L = \{a^n \mid n \geq 0\}$ is not regular.
19. Show that $L = \{a^n \mid n \text{ is prime}\}$ is not regular.
20. Show that $L = \{w \mid n_a(w) = n_b(w)\}$ is not regular. Is L^* regular?
21. Show that $L = \{w \mid n_a(w) < n_b(w)\}$ is not regular.
22. Show that $L = \{ww \mid w \in \{a,b\}^*\}$ is not regular.
23. Show that $L = \{(ab)^n a^k \mid n > k, k \geq 0\}$ is not regular.
24. Show that $L = \{a^n \mid n = k^2 \text{ for } k \geq 0\}$ is not regular.
25. Show that $L = \{0^n \text{ such that } n \text{ is not a prime number}\}$ is not regular.
26. Show that $L = \{0^n 1^n 2^n \text{ for } n \geq 0\}$ is not regular.

Chapter 4

Context Free Grammars and Languages

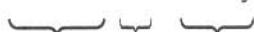
What are we studying in this chapter . . .

- ▶ *Context free grammars*
- ▶ *Parse trees*
- ▶ *Applications*
- ▶ *Ambiguity in grammars and Languages*

4.1. Grammar

In this section, let us see the definition of a grammar and how a language can be constructed using the grammar and various types of grammars. Consider the sentence:

Monalika ate slowly

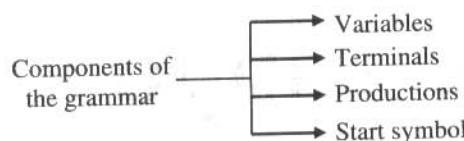


Noun verb adverb

Thus, a sentence in this example starts with *noun* followed by *verb* followed by *adverb*. After replacing *noun*, *verb* and *adverb* with appropriate words, a grammatically correct sentence can be obtained. The rules to form this sentence can be written as

1. sentence \rightarrow <noun> <verb> <adverb>
2. noun \rightarrow Monalika
3. verb \rightarrow ate
4. adverb \rightarrow slowly

Now, let us see “What are the various components of the above grammar?”. The various components of the grammar are shown below:



Variables: In the above four rules *sentence*, *noun*, *verb*, and *adverb* are called *variables*. The variables are also called *non-terminals*.

Terminals: The words “Monalika”, “ate” and “slowly” are called terminals.

Productions: The above four rules which are used to obtain the sentence “Monalika ate slowly” are called productions.

- Each production starts with *non-terminal*
- Followed by an *arrow*
- Followed by combinations of *non-terminals* and/or *terminals*

Start symbol: The left hand side of ‘ \rightarrow ’ in the first production is called the *start symbol*. The start symbol is usually denoted by the letter S.

Note: The *non-terminals* can be replaced by terminals or non-terminals whereas the *terminals* cannot be replaced.

Now, let us see “What is a grammar? Give an example”.

Definition: A grammar G is 4-tuple or quadruple $G = (V, T, P, S)$ where

- V is set of variables or non-terminals.
- T is set of terminals.
- P is set of productions. Each production is of the form $\alpha \rightarrow \beta$ where α is a string in $(V \cup T)^*$ and hence α can not be ϵ and ϵ can not occur on the right hand side of any production. But, β is a string in $(V \cup T)^*$ and hence it includes ϵ also. So, ϵ can occur on the right hand side of the production.
- S is the start symbol.

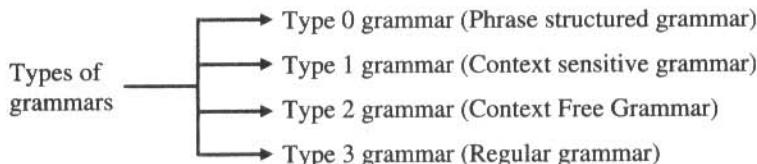
Note: The null string is denoted by symbol ϵ (epsilon). The null string cannot occur on the left hand side of any production.

Before constructing grammar, let us see “What are the notations used while constructing the grammar?”. The various notations used are shown below:

- The following are the terminals:
 - The keywords such as if, for, while, do-while, etc.;
 - Digits from 0 to 9;
 - Symbols such as +, -, *, /, etc.;
 - The lower case letters near the beginning of the alphabets such as *a*, *b*, *c*, *d*, etc.;
 - The bold faced letters such as **id**.
- The following are non-terminals:
 - The lower case names such as expression, operator, operand, statement, etc.;
 - The capital letters near the beginning of the alphabets such as A, B, C, D, etc.;
 - The letter S is the start symbol.
- The lower case letters near the end of the alphabets such as u, v, w, x, y, z represent string of terminals.
- The capital letters near the end of the alphabets such as X, Y, Z, etc. represent grammar symbols. The grammar symbol can be terminal or non-terminal.
- The Greek letters such as α represent string of grammar symbols.

4.2. Chomsky Hierarchy

Noam Chomsky who is the founder of formal language theory has classified the grammar into various categories. Now, let us see “What are various types of grammars?” or “Explain Chomsky Hierarchy?” or “Explain the classification of grammars”. The grammar can be classified as shown below:



4.2.1. Type 0 Grammar

Now, let us see “What is type 0 grammar?”. Type 0 grammar is defined as follows:

Definition: A grammar $G = (V, T, P, S)$ is said to be type 0 grammar or unrestricted grammar or Phrase structured grammar if all the productions are of the form $\alpha \rightarrow \beta$ where

$$\alpha \in (V \cup T)^+ \text{ and } \beta \in (V \cup T)^*$$

In this type of grammar there are no restrictions on length of α and β . The only restriction is that α can not be ϵ i.e., ϵ can not appear on the left hand side of any production. But, ϵ can appear on the right hand side of the function. This is the largest family of grammars more powerful than all other types of grammars. Any language can be obtained from this grammar.

The language generated from this grammar is called type 0 language or recursively enumerable language. Only Turing machine can recognize this language. In the subsequent chapters, we shall see how to obtain the Turing machines. Consider the example shown below:

$$\begin{array}{lcl} S & \rightarrow & aAb \mid \epsilon \\ aA & \rightarrow & bAA \\ bA & \rightarrow & a \end{array}$$

4.2.2. Type 1 Grammar or Context Sensitive Grammar

Now, let us see “What is type 1 grammar?”. The type 1 grammar or context sensitive grammar is defined as follows.

❖ **Definition:** A grammar $G = (V, T, P, S)$ is said to be type 1 grammar or context sensitive grammar if all the productions are of the form $\alpha \rightarrow \beta$ as in type 0 grammar. But, there is a restriction on length of β . The length of β must be at least as much as the length of α i.e., $|\beta| \geq |\alpha|$ and α and $\beta \in (V \cup T)^*$ i.e., ϵ can not appear on the left hand or right hand side of any production. It is an ϵ -free grammar.

The language generated from this grammar is called type 1 language or context sensitive language. Linear bounded automata (LBA) can be constructed to recognize the language generated from this grammar and we see the construction of LBA in later chapters. The following example shows type 1 grammar or context sensitive grammar:

$$\begin{array}{lcl} S & \rightarrow & aAb \\ aA & \rightarrow & bAA \\ bA & \rightarrow & aa \end{array}$$

4.2.3. Type 2 Grammar or Context Free Grammar

Now, let us see “What is Type 2 grammar or context free grammar?”. The type 2 grammar also called context free grammar (CFG) is defined as follows.

♦ **Definition:** A grammar $G = (V, T, P, S)$ is said to be type 2 grammar or context free grammar if all the productions are of the form $A \rightarrow \alpha$ where $\alpha \in (VUT)^*$ and A is non-terminal. The symbol ϵ can appear on the right hand side of any production. The context free grammar was discussed in section 1.14. The language generated from this grammar is called type 2 language or context free language. Pushdown automaton (PDA) can be constructed to recognize the language generated from this grammar. The grammar

$$\begin{array}{lcl} S & \rightarrow & aB \mid bA \mid \epsilon \\ A & \rightarrow & aA \mid b \\ B & \rightarrow & bB \mid a \mid \epsilon \end{array}$$

is an example for context free grammar.

4.2.4. Type 3 Grammar or Regular Grammar

Now, let us see “What is type 3 grammar?”. The type 3 grammar or context free grammar is defined as follows.

♦ **Definition:** The grammar $G = (V, T, P, S)$ is said to be type 3 grammar or regular grammar iff the grammar is right linear or left linear. A grammar G is said to be right linear if all the productions are of the form

$$A \rightarrow wB$$

or

$$A \rightarrow w$$

where A, B are variables and $w \in T^*$, i.e., w is string of terminals. A grammar G is said to be left linear if all the productions are of the form

$$A \rightarrow Bw$$

or

$$A \rightarrow w$$

where $A, B \in V$ and $w \in T^*$. The following grammar

$$\begin{array}{lcl} S & \rightarrow & aaB \mid bbA \mid \epsilon \\ A & \rightarrow & aA \mid b \\ B & \rightarrow & bB \mid a \mid \epsilon \end{array}$$

is a right linear grammar. Note that ϵ and string of terminals can appear on RHS of any production and if non-terminal is present on R.H.S of any production, only one non-terminal should be present and it has to be the right most symbol on R.H.S. The grammar

$$\begin{array}{lcl} S & \rightarrow & Baa \mid Abb \mid \epsilon \\ A & \rightarrow & Aa \mid b \\ B & \rightarrow & Bb \mid a \mid \epsilon \end{array}$$

is a left linear grammar. Note that ϵ and string of terminals can appear on RHS of any production and if non-terminal is present on R.H.S of any production, only one non-terminal should be present and it has to be the left most symbol on R.H.S.

Consider the following grammar

$$\begin{array}{lcl} S & \rightarrow & aA \\ A & \rightarrow & aB \mid b \\ B & \rightarrow & Ab \mid a \end{array}$$

In this grammar, each production is either left linear or right linear. But, the grammar is not either left linear or right linear. Such type of grammar is called linear grammar. So, a grammar which has at most one non-terminal on the right side of any production without restriction on the position of this non-terminal (note the non-terminal can be leftmost or right most) is called linear grammar.

Note that the language generated from the regular grammar is called regular language. So, there should be some relation between the regular grammar and the FA, since, the language accepted by FA is also regular language. So, we can construct a finite automaton given a regular grammar and vice versa. Let us discuss how to obtain finite automaton from the regular expression.

In this chapter, let us concentrate only on context free grammar.

4.3. Grammar from Finite Automata

Now, let us see “What is the easiest method of constructing a grammar?”. A grammar can be obtained very easily using finite automaton. The general procedure is shown below:

Procedure: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automata accepting L where

$$\begin{aligned} Q &= \{q_0, q_1, \dots, q_n\} \\ \Sigma &= \{a_1, a_2, \dots, a_m\} \end{aligned}$$

A grammar $G = (V, T, P, S)$ can be constructed where

- $V = \{q_0, q_1, \dots, q_n\}$ i.e., the states of DFA will be the variables in the grammar.
- $T = \Sigma$ i.e., the input alphabets of DFA are the terminals in grammar.
- $S = q_0$ i.e., the start state of DFA is the start symbol in the grammar.
- The productions P from the transitions can be obtained as shown below:

Step 1: If $\delta(q_i, a) = q_j$ then introduce the transition:

$$q_i \rightarrow aq_j$$

Step 2: If $q \in F$, i.e., if q is the final state in FA, then introduce the production

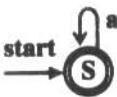
$$q \rightarrow \epsilon$$

Using the above two steps we can convert any finite automaton (DFA or NFA or ϵ -NFA) into grammar.

Now, let us see “How to obtain context free grammar for simple languages using finite automata?”. The table shows the DFA accepting the language, the transitions defined and the equivalent grammar:

I Example 1: Obtain grammar to generate string consisting of any number of a's.

Solution: The DFA to accept string consisting of any number of a's is shown below:

DFA	Transitions	Grammar
	<ul style="list-style-type: none"> • S is a final state • $\delta(S, a) = S$ 	$S \rightarrow \epsilon$
		$S \rightarrow a$
		$S \xrightarrow{} S \rightarrow aS$

So, the grammar to generate string consisting of any number of a 's is given by:

$$\begin{array}{l} S \rightarrow \epsilon \\ S \rightarrow aS \end{array}$$

The language generated by grammar can be formally written as shown below:

$$L = \{a^n : n \geq 0\}$$

Example 2: Obtain grammar to generate string consisting of at least one a .

Solution: The DFA to accept string consisting of at least one a is shown below:

Method 1: By replacing ϵ by a in the above grammar, the minimum string generated is a . So, the grammar to generate string consisting of at least one a is shown below:

$$\left. \begin{array}{l} S \rightarrow a \\ S \rightarrow aS \end{array} \right\} \text{Grammar to generate at least one } a$$

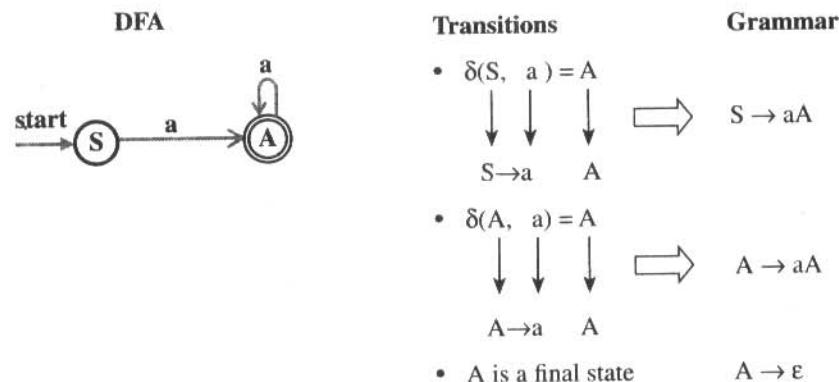
Note: The above productions start from S and hence can also be written as:

$$S \rightarrow a \mid aS$$

The language generated by grammar can be formally written as shown below:

$$L = \{a^n : n \geq 1\}$$

Method 2: The above grammar can be generated by constructing a DFA and then converting DFA into equivalent grammar as shown below:



So, the final grammar to generate to at least one a is shown below:

$$\begin{array}{l} S \rightarrow aA \\ A \rightarrow aA \\ A \rightarrow \epsilon \end{array} \quad \left. \right\} \text{Grammar to generate at least one } a$$

The language generated by grammar can be formally written as shown below:

$$L = \{a^n : n \geq 1\}$$

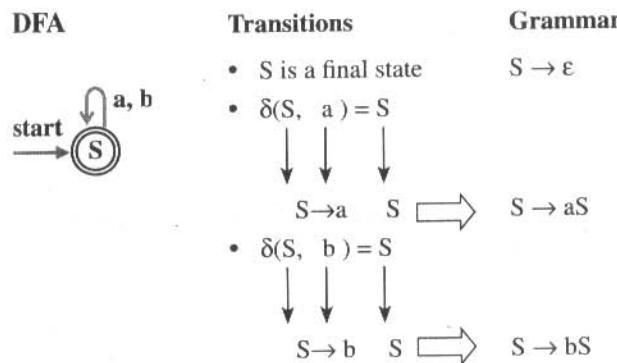
Note: Observe from method 1 and method 2 that one or more different grammars may exist that generate same language.

Note: For each final state introduce the ϵ transition. For example, in a DFA, if the states A and B are final states, then in the grammar we should introduce two ϵ -productions:

$$\begin{array}{l} A \rightarrow \epsilon \\ B \rightarrow \epsilon \end{array}$$

■ **Example 3:** Obtain grammar to generate string consisting of any number of a's and b's.

Solution: The DFA to accept string consisting of any number of a's and b's is shown below:



So, the grammar to generate string consisting of any number of a's and b's is given by:

$$\begin{array}{l} S \rightarrow \epsilon \\ S \rightarrow aS \\ S \rightarrow bS \end{array} \quad \left. \right\} \text{Grammar to generate string consisting of any number of } a\text{'s and } b\text{'s}$$

Note: Since all productions start from S, the above productions can also be written as:

$$S \rightarrow \epsilon \mid aS \mid bS$$

■ **Example 4:** Obtain grammar to generate string consisting of at least two a's.

Solution: In Example 2, we have a production

$$S \rightarrow a \mid aS$$

using which one a is generated. To generate at least two a's, in the above production replace one a by two a 's. So, the grammar to generate string consisting of at least two a's is given by:

$$S \rightarrow aa$$

$$S \rightarrow aS$$

Grammar

■ **Example 5:** Obtain grammar to generate string consisting of even number of a's.

Solution: The given language can be written as:

$$L = \{\epsilon, aa, aaaa, aaaaaa, \dots\}$$

or

$$L = \{w : n_a(w) \bmod 2 = 0 \text{ where } w \in a^*\}$$

In Example 1 we have productions

$$S \rightarrow \epsilon \mid aS$$

To get the required language we should have ϵ , but subsequently we should generate multiples of two a's. This is done by replacing a by aa in the production:

$$S \rightarrow aS$$

So, the grammar is given by:

$$S \rightarrow \epsilon \mid aaS$$

I Example 6: Obtain grammar to generate string consisting of multiples of three a's.

Solution: The given language can be written as:

$$L = \{\epsilon, aaa, aaaaa, aaaaaaaaa, \dots\}$$

or

$$L = \{w : n_a(w) \bmod 3 = 0 \text{ where } w \in a^*\}$$

In the previous problem instead of aa we replace it by aaa. So, the final grammar is:

$$S \rightarrow \epsilon \mid aaaS$$

I Example 7: Obtain grammar to generate strings of a's and b's such that string length is multiple of 3.

Solution: The grammar shown in Example 6 can also be written as:

$$S \rightarrow \epsilon \mid AAAS$$

$$A \rightarrow a$$

By replacing a by a or b we get strings of a's and b's whose length is a multiple of three. So, the grammar is

$$S \rightarrow \epsilon \mid AAAS$$

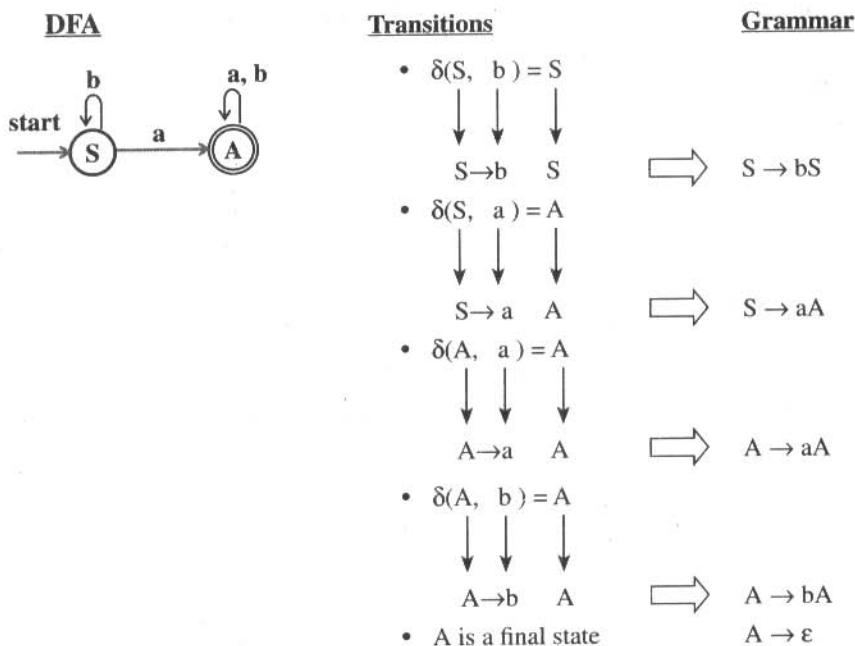
$$A \rightarrow a \mid b$$

I Example 8: Obtain grammar to generate string consisting of any number of a's and b's with at least one a .

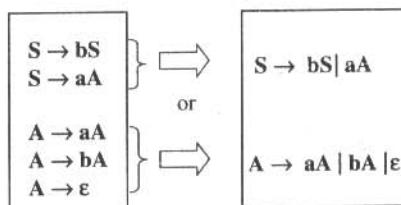
Solution (Method 1): Same as grammar shown in Example 3 but replacing ϵ by a . So, the resultant grammar is:

$$\begin{array}{l} S \rightarrow a \\ S \rightarrow aS \\ S \rightarrow bS \end{array} \quad \left. \right\} \text{Grammar to generate strings of } a\text{'s and } b\text{'s with at least one } a$$

Method 2: The DFA to accept strings of a's and b's consisting of at least one a is shown below:



So, the grammar to generate string consisting of any number of a's and b's is given by:



Note: All productions starting from S can be grouped and all productions starting from A can be grouped as shown above. The language can be formally written as:

$$L = \{w : n_a(w) \geq 1, w \in \{a, b\}^*\}$$

Example 9: Obtain grammar to generate string consisting of any number of a's and b's with at least one b.

Solution: Same as grammar shown in Example 3 but replacing ϵ by b. So, the resultant grammar is

$$\left. \begin{array}{l} S \rightarrow b \\ S \rightarrow aS \\ S \rightarrow bS \end{array} \right\} \text{Grammar to generate strings of a's and b's with at least one b}$$

Example 10: Obtain grammar to generate string consisting of any number of a's and b's with at least one a or at least one b .

Solution: Same as grammar shown in Example 3 but replacing ϵ by a or b . So, the resultant grammar is

$$\begin{array}{l} S \rightarrow a \mid b \\ S \rightarrow aS \\ S \rightarrow bS \end{array} \quad \left. \begin{array}{l} S \rightarrow a \mid b \\ S \rightarrow aS \\ S \rightarrow bS \end{array} \right\}$$

Note: $a \mid b$ is read either a or b .

Grammar to generate strings of a's and b's with at least one b

Example 11: Obtain grammar to accept the following language:

$$L = \{w : |w| \bmod 3 > 0 \text{ where } w \in \{a\}^*\}$$

Solution: The minimum string that can be accepted is either a or aa . So, the productions to produce a or aa is given by:

$$S \rightarrow a \mid aa$$

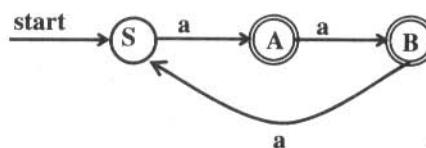
The string aaa should not be accepted. But, aaa followed by a or aa can be accepted. This is obtained using the production:

$$S \rightarrow aaaS$$

So, the final grammar is given by:

$$S \rightarrow a \mid aa \mid aaaS$$

Note: Alternate method is write a DFA and then obtain the equivalent productions. The DFA is shown below:



$$\delta(S, a) = A$$

$$\delta(A, a) = B$$

$$\delta(B, a) = S$$

A is final state

B is final state

$$\Rightarrow S \rightarrow aA$$

$$\Rightarrow A \rightarrow aB$$

$$\Rightarrow B \rightarrow aS$$

$$\Rightarrow A \rightarrow \epsilon$$

$$\Rightarrow B \rightarrow \epsilon$$

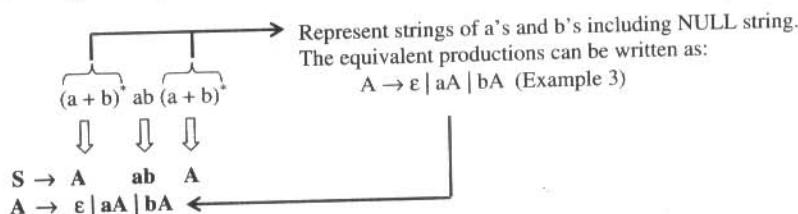
Grammar

4.4. Grammar from Regular Expressions

We have learnt in previous chapter that regular language can also be represented using regular expressions. Now, let us see “How to get the grammar from regular expressions?”.

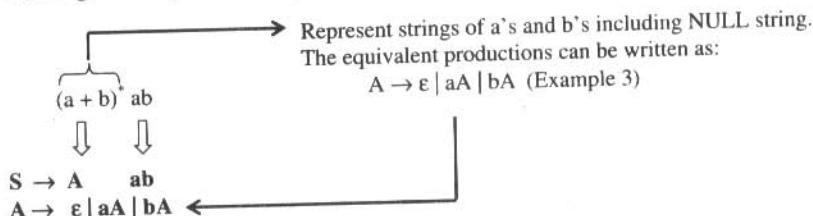
Example 12: Obtain grammar to generate strings of a's and b's having a substring ab.

Solution: The regular expression representing strings of a's and b's having a substring is given by:



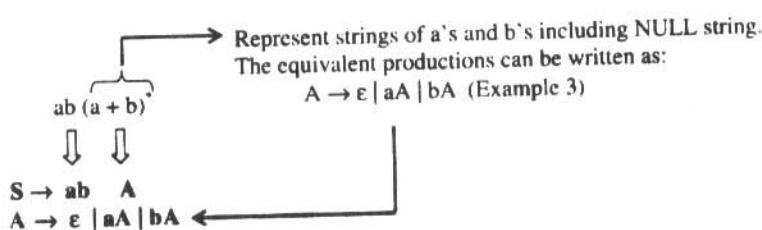
Example 13: Obtain grammar to generate strings of a's and b's ending with string ab.

Solution: The regular expression representing strings of a's and b's ending with ab is given by:



Example 14: Obtain grammar to generate strings of a's and b's starting with ab.

Solution: The regular expression representing strings of a's and b's having a substring is given by:



Example 15: Obtain grammar to generate the following language:

$$L = \{w : n_a(w) \bmod 2 = 0 \text{ where } w \in \{a, b\}^*\}$$

Solution: String consisting of any number of b's given by the grammar:

$$S \rightarrow \epsilon \mid bS \quad (1)$$

(See Example 1)

The regular expression for the given language can be written as shown below:

$$\begin{array}{c} (b^* \ a \ b^* \ a \ b^*)^* \\ \downarrow \qquad \downarrow \qquad \downarrow \\ S \rightarrow S \ a \ S \ a \ S \end{array} \quad (2)$$

(from S we can get b^* , for details see (1) above)

So, the final grammar is obtained by combining (1) and (2) as shown below:

$$S \rightarrow \epsilon \mid bS \mid SaSaS$$

4.5. Derivation

Now, let us see “What is derivation?”.

❖ Definition: Let $A \rightarrow \alpha B \gamma$ and $B \rightarrow \beta$ are productions in grammar G, where α , β and γ are strings of terminals and/or non-terminals, A and B are non-terminals. The non-terminal A produces the string $\alpha \beta \gamma$ by replacing the non-terminal B in $\alpha B \gamma$ by the string β by applying the production $B \rightarrow \beta$ and can be written as

$$A \Rightarrow \alpha \beta \gamma$$

This process of obtaining string of terminals and/or non-terminals from the start symbol by applying some or all productions is called *derivation*. If a string is obtained by applying only one production, then it is called one-step derivation and is denoted by the symbol ‘ \Rightarrow ’. If one or more productions are applied to get the string $\alpha \beta \gamma$ from A, then we write

$$A \Rightarrow^* \alpha \beta \gamma$$

If zero or more productions are applied to get the string $\alpha \beta \gamma$ from A, then we write

$$A \xrightarrow{*} \alpha \beta \gamma$$

Note: If α is any string of terminals and variables then $\alpha \Rightarrow \alpha$ i.e., α derives itself.

■ **Example 16:** Consider the grammar shown below from which any arithmetic expression can be obtained.

$$\begin{array}{lcl} E & \rightarrow & E + E \\ E & \rightarrow & E - E \\ E & \rightarrow & E * E \\ E & \rightarrow & E / E \\ E & \rightarrow & \text{id} \end{array}$$

The non-terminal E is used instead of using the word *expression*. The left-hand side of the first production i.e., E is considered to be the start symbol. Obtain the string $\text{id} + \text{id} * \text{id}$ and show the derivation for the same.

Solution: The derivation to get the string $\text{id} + \text{id} * \text{id}$ is shown below:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \text{id} + E \\ &\Rightarrow \text{id} + E * E \\ &\Rightarrow \text{id} + \text{id} * E \\ &\Rightarrow \text{id} + \text{id} * \text{id} \end{aligned}$$

Since the string $\text{id} + \text{id} * \text{id}$ is obtained from the start symbol E by applying more than one production, this can be written as

$$E \Rightarrow^+ \text{id} + \text{id} * \text{id}$$

4.5.1. Sentence

Using the grammar, let us see “How to get a sentence?” Before proceeding further let us see “What is a sentence or sentential form?”.

❖ **Definition:** Let $G = (V, T, P, S)$ be a grammar. The string w obtained from the grammar G such that $S \Rightarrow^+ w$ is called sentence of grammar G . Here, w is the string of terminals.

Example 17: In the derivation shown in Example 16, $\text{id} + \text{id} * \text{id}$ is the sentence of the grammar. If there is a derivation $S \Rightarrow \alpha$, where α contains string of terminals and/or non-terminals, then α is called *sentential form* of G. In the derivation shown in Example 16:

$E+E, \text{id} + E, \text{id} + E * E, \text{id} + \text{id} * E, \text{id} + \text{id} * \text{id}$

are all *sentential forms* of the grammar.

4.5.2. Language

A given grammar can generate a set of strings consisting of only of terminals by applying productions in different order. The set of such strings is called the language. Now, let us formally see “What is the language generated by grammar?”. The formal definition of the language accepted by a grammar is defined as shown below.

❖ **Definition:** Let $G = (V, T, P, S)$ be a grammar. The language $L(G)$ generated by the grammar G is

$$L(G) = \{w \mid S \xrightarrow{*} w \text{ and } w \in T^*\}$$

i.e., w is a string of terminals (may be ϵ) obtained from the start symbol S by applying an arbitrary number of productions. The intermediate string of terminals and/or non-terminals obtained during the derivation process is called sentential form of G. The various strings which are the elements of $L(G)$ are called *sentence*s.

Example 18: Consider the following grammar:

$$\begin{array}{lcl} S & \rightarrow & ACa \\ C & \rightarrow & aCa \mid b \end{array}$$

What is the language generated by this grammar?

Solution: Consider the derivation

$$\begin{aligned} S &\Rightarrow aCa \Rightarrow aba \text{ (By applying the 1st and 3rd production)} \\ &\text{So, the string } aba \in L(G) \end{aligned}$$

Consider the derivation

$$\begin{aligned} S &\Rightarrow aCaB \text{ By applying } S \rightarrow aCa \\ &\Rightarrow aaCaaB \text{ By applying } C \rightarrow aCa \\ &\Rightarrow aaaCaaaB \text{ By applying } C \rightarrow aCa \\ &\vdots \end{aligned}$$

$\Rightarrow a^n Ca^n$ By applying $C \rightarrow aCa$ $n-1$ times
 $\Rightarrow a^n ba^n$ By applying $C \rightarrow b$

So, the language L accepted by the grammar G is

$$L(G) = \{ a^n ba^n \mid n \geq 1 \}$$

4.6. Grammars for Other Languages

■ **Example 19:** Obtain a grammar to generate the following language:

$$L = \{ a^n b^n : n \geq 0 \}$$

Solution: In Example 1, we have the following grammar:

$$S \rightarrow \epsilon \mid aS$$

that generates string consisting of any number of a's. Now, for every a one b has to be generated. This is obtained by suffixing aS with a b . So, the grammar to generate the given language is:

$$S \rightarrow \epsilon \mid aSb$$

■ **Example 20:** Obtain a grammar to generate the following language:

$$L = \{ a^n b^n : n \geq 1 \}$$

Solution: It is similar to the previous example. But, instead of generating at least ϵ , we should generate at least ab . This is achieved by replacing ϵ by ab . So, the final grammar is given by:

$$S \rightarrow ab \mid aSb$$

■ **Example 21:** Obtain a grammar to generate the following language:

$$L = \{ a^{n+1} b^n : n \geq 0 \}$$

Solution: It is similar to Example 19. But, one extra a should be generated. This is achieved by replacing ϵ with a . So, the final grammar to generate the given language is:

$$S \rightarrow a \mid aSb$$

Example 22: Obtain a grammar to generate the following language:

$$L = \{a^n b^{n+1} : n \geq 0\}$$

Solution: It is similar to Example 19. But, one extra b should be generated. So, the final grammar to generate the given language is

$$S \rightarrow b \mid aSb$$

Example 23: Obtain a grammar to generate the following language:

$$L = \{a^n b^{n+2} : n \geq 0\}$$

Solution: It is similar to Example 19. But, two extra b 's should be generated. So, the final grammar to generate the given language is

$$S \rightarrow bb \mid aSb$$

Example 24: Obtain a grammar to generate the following language:

$$L = \{a^n b^{2n} : n \geq 0\}$$

Solution: Consider the grammar shown in Example 19:

$$S \rightarrow \epsilon \mid aSb$$

The above grammar generates the language $a^n b^n$. But, we want $a^n b^{2n}$ i.e., for every a we want two b 's. This is achieved by suffix one more b to the above grammar. So, the final grammar is

$$S \rightarrow \epsilon \mid aSbb$$

Example 25: Let $\Sigma = \{a, b\}$. Obtain a grammar G generating set of all palindromes over Σ .

Solution: The recursive definition of a palindrome along with corresponding productions is shown below:

1. ϵ is a palindrome. The equivalent production is $S \rightarrow \epsilon$
2. a and b are palindromes. The equivalent productions are $S \rightarrow a \mid b$
3. If w is a palindrome then the string awa and the string bwb are palindromes. The equivalent productions are

$$S \rightarrow aSa \mid bSb$$

So, the grammar to generate strings of palindromes over Σ is given by $G = (V, T, P, S)$ where

$$\begin{aligned}
 V &= \{S\} \\
 T &= \{a, b\} \\
 P &= \{ \\
 &\quad S \rightarrow \epsilon && [\text{Definition 1}] \\
 &\quad S \rightarrow a \mid b && [\text{Definition 2}] \\
 &\quad S \rightarrow aSa \mid bSb && [\text{Definition 3}] \\
 &\} \\
 S &\text{ is the start symbol}
 \end{aligned}$$

Example 26: Obtain a grammar to generate the following language:

$$L = \{ww^R \text{ where } w \in \{a, b\}^*\}$$

Solution: The language can be written as:

$$L = \{aa, bb, abba, baab, aaaa, bbbb, \dots\}$$

Observe that the given string is a palindrome of even length. This is achieved by deleting the production $S \rightarrow a \mid b$. So, the final grammar is given by:

$$\begin{aligned}
 S &\rightarrow \epsilon \\
 S &\rightarrow aSa \mid bSb
 \end{aligned}$$

Note: In the above grammar if the production

$$S \rightarrow \epsilon$$

is replaced by

$$S \rightarrow c$$

the resulting grammar will generate the language in $L = \{wcw^R \mid w \in \{a,b\}^*\}$.

Example 27: Obtain a grammar to generate a language consisting of all non-palindromes over $\{a, b\}$.

Solution: When we scan from left to right and right to left simultaneously, we may find same symbols for a while, but at some point while scanning we may find a symbol on the left which is

different from the symbol on the right. Then the given string is not a palindrome. The corresponding S-productions for this can be

$$S \rightarrow aSa \mid bSb \mid A$$

where A generates string of only non palindromes. To get a string of non-palindrome, the string derivable from A should have different symbols in the beginning and in the end, the length of which should be greater than or equal to 2 and so the A-productions can take the following form

$$A \rightarrow aBb \mid bBa$$

From production B, if we can generate any string of a's and b's including ϵ i.e., $(a+b)^*$, the string derivable from A is still a non-palindrome. So, the B-productions can be written as

$$B \rightarrow aB \mid bB \mid \epsilon$$

So, the complete grammar to generate strings of non-palindromes is given by $G = (V, T, P, S)$ where

$$V = \{ S, A, B \}$$

$$T = \{ a, b \}$$

$$P = \{$$

$S \rightarrow aSa \mid bSb$	[Generates palindromes both on left side and right side]
$S \rightarrow A$	[Generates a non-palindrome]
$A \rightarrow aBb \mid bBa$	[Generates a non palindrome with B generating any number of a's and b's]
$B \rightarrow aB \mid bB \mid \epsilon$	[Generates any combination of a's and b's]

}

S is the start symbol

The derivation for the string ababba which is not a palindrome is shown below.

$$\begin{aligned} S &\Rightarrow aSa \\ &\Rightarrow abSba \\ &\Rightarrow abAba \\ &\Rightarrow abaBba \\ &\Rightarrow ababba \end{aligned}$$

Example 28: Obtain the grammar to generate the language:

$$L = \{ 0^m 1^m 2^n \mid m \geq 1 \text{ and } n \geq 0 \}$$

Solution: Given the language the productions can be generated as shown below:

$$L = \{ 0^m 1^m 2^n \mid m \geq 1 \text{ and } n \geq 0 \}$$

$$\begin{array}{c} \swarrow \\ S \rightarrow A \quad B \end{array}$$

(I)

where

- The variable A should produce m number of 0's followed by m number of 1's with a minimum string 01 (Since $m \geq 1$). This is achieved using the following production:

$$A \rightarrow 01 \mid 0A1 \quad (\text{Similar to Example 20})$$

- B should produce any number of 2's. Any number of 2's can be generated using the production:

$$B \rightarrow \epsilon \mid 2B \quad (\text{Similar to Example 1})$$

So, final grammar to accept the given language is

$$\left. \begin{array}{l} S \rightarrow A \quad B \\ A \rightarrow 01 \mid 0A1 \\ B \rightarrow \epsilon \mid 2B \end{array} \right\} \text{Grammar to accept } L = \{ 0^m 1^m 2^n \mid m \geq 1 \text{ and } n \geq 0 \}$$

The following grammar also generates the same language. The reader is required to verify the answer.

$$\begin{array}{l} S \rightarrow A \mid S2 \\ A \rightarrow 01 \mid 0A1 \end{array}$$

Example 29: Obtain the grammar to generate the language:

$$L = \{ w \mid n_a(w) = n_b(w) \}$$

Note: $n_a(w) = n_b(w)$ means, number of a's in the string w should be equal to number of b's in the string w . To get equal number of a's and b's, we know that there are three cases:

- An empty string denoted by ϵ has equal number of a's and b's (i.e., zero a's and zero b's).
- The symbol 'a' can be followed by the symbol 'b'.
- The symbol 'b' can be followed by the symbol 'a'.

The corresponding productions for these three cases can be written as

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow aSb \\ S &\rightarrow bSa \end{aligned}$$

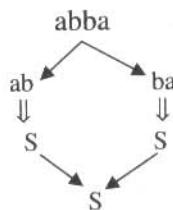
Using these productions the strings of the form ϵ , ab, ba, abab, baba etc., can be generated. But, the strings such as abba, baab, etc., where the string starts and ends with the same symbol, can not be generated from these productions (even though they are valid strings). So, to obtain the productions to generate such strings, let us divide the string into two substrings. For example, let us take the string 'abba'. This string can be split into two substrings 'ab' and 'ba'. The substring 'ab' can be generated from S and the derivation is shown below:

$$\begin{aligned} S &\Rightarrow aSb \text{(By applying } S \rightarrow aSb) \\ &\Rightarrow ab \text{(By applying } S \rightarrow \epsilon) \end{aligned}$$

Similarly, the substring 'ba' can be generated from S and the derivation is shown below:

$$\begin{aligned} S &\Rightarrow bSa \text{(By applying } S \rightarrow bSa) \\ &\Rightarrow ba \text{(By applying } S \rightarrow \epsilon) \end{aligned}$$

i.e., the first sub string 'ab' can be generated from S as shown in the first derivation and the second sub string 'ba' can also be generated from S as shown in second derivation. So, To get the string 'abba' from S, perform the derivation in reverse order as shown below:



So, to get a string such that it starts and ends with the same symbol, the production to be used is

$$S \rightarrow SS$$

The final grammar to generate the language $L = \{w \mid n_a(w) = n_b(w)\}$ is $G = (V, T, P, S)$ where

$$\begin{aligned} V &= \{S\} \\ T &= \{a, b\} \\ P &= \{ \\ &\quad S \rightarrow \epsilon \mid aSb \mid bSa \mid SS \\ &\quad \} \\ S & \text{ is the start symbol} \end{aligned}$$

Example 30: What is the language generated by the grammar:

$$\begin{aligned} S &\rightarrow 0A \mid \epsilon \\ A &\rightarrow 1S \end{aligned}$$

Solution: The null string ϵ can be obtained by applying the production $S \rightarrow \epsilon$ and the derivation is shown below:

$$S \Rightarrow \epsilon \text{(By applying } S \rightarrow \epsilon \text{)}$$

Consider the derivation:

$$\begin{aligned} S &\Rightarrow 0A \text{(By applying } S \rightarrow 0A \text{)} \\ &\Rightarrow 01S \text{(By applying } A \rightarrow 1S \text{)} \\ &\Rightarrow 010A \text{(By applying } S \rightarrow 0A \text{)} \\ &\Rightarrow 0101S \text{(By applying } A \rightarrow 1S \text{)} \\ &\Rightarrow 0101 \text{(By applying } S \rightarrow \epsilon \text{)} \end{aligned}$$

So, alternatively applying the productions $S \rightarrow 0A$ and $A \rightarrow 1S$ and finally applying the production $S \rightarrow \epsilon$, we get string consisting of only of 01's. So, both null string i.e., ϵ and string consisting of 01's can be generated from this grammar. So, the language generated by this grammar is

$$L = \{w \mid w \in \{01\}^*\} \quad \text{or} \quad L = \{(01)^n \mid n \geq 0\}$$

Note: The above language can also be generated from the following two grammars:

$$S \rightarrow 01S \mid \epsilon$$

or

$$S \rightarrow S01 \mid \epsilon$$

Example 31: Obtain a CFG to generate a string of balanced parentheses.

Solution: The grammar G to generate a string of balanced parentheses is given by $G = (V, T, P)$ where

$$\begin{aligned} V &= \{S\} \\ T &= \{(), \}\} \\ P &= \{ \\ &\quad S \rightarrow (S) \\ &\quad S \rightarrow SS \\ &\quad S \rightarrow \epsilon \\ &\quad \} \end{aligned}$$

S is the start symbol

Example 32: Obtain a grammar to generate the language:

$$L = \{0^i 1^j \mid i \neq j, i \geq 0 \text{ and } j \geq 0\}$$

Note: It is clear from the statement that if a string has n number of 0's as the prefix, this prefixed string should not be followed by n number of 1's i.e., we should not have equal number of 0's and 1's. At the same time 0's should precede 1's. The grammar for this can be written as

$$G = (V, T, P, S) \text{ where}$$

$$V = \{S, A, B, C\}$$

$$T = \{0, 1\}$$

$$P = \{$$

$$S \rightarrow BA \quad [\text{At least one 0 is preceded by } 0^n 1^n]$$

$$S \rightarrow AC \quad [\text{At least one 1 is followed by } 0^n 1^n]$$

$$A \rightarrow 0A1 \mid \epsilon \quad [\text{Generates } 0^n 1^n \mid n \geq 0]$$

$$B \rightarrow 0B \mid 0 \quad [\text{At least one 0 is generated}]$$

$$C \rightarrow 1C \mid 1 \quad [\text{At least one 1 is generated}]$$

$$\}$$

S is the start symbol

Note: The following grammar also generates the language $L = \{0^i 1^j \mid i \neq j, i \geq 0 \text{ and } j \geq 0\}$

$$V = \{S, A, B, C\}$$

$$T = \{0, 1\}$$

$$P = \{$$

$$S \rightarrow 0S1 \quad [\text{Generates } 0^n 1^n \text{ recursively}]$$

$$S \rightarrow A \quad [\text{To generate more 0's than 1's}]$$

$$S \rightarrow B \quad [\text{To generate more 1's than 0's}]$$

$$A \rightarrow 0A \mid 0 \quad [\text{At least one 0 is generated}]$$

$$B \rightarrow 1B \mid 1 \quad [\text{At least one 1 is generated}]$$

$$\}$$

S is the start symbol

Example 33: Obtain a grammar to generate the language:

$$L = \{a^{n+2}b^m \mid n \geq 0 \text{ and } m > n\}$$

Solution: It is clear from the above statement that the set of strings that can be generated by this language can be represented as

$$L = \{aabb^*, aaabbb^*, aaaabbbb^*, \dots\}$$

Observe that the language consists of a string $a^n b^n \mid n \geq 1$ preceded by one a and followed by zero or more b 's. This prompts us to have a production of the form

$$S \rightarrow aAB$$

where the language $a^n b^n \mid n \geq 1$ is generated from the variable A and zero or more b 's are generated from the variable B. The language $a^n b^n \mid n \geq 1$ is generated from the variable A using the productions shown below:

$$A \rightarrow aAb \mid ab$$

and zero or more b 's are generated from the production:

$$B \rightarrow bB \mid \epsilon$$

So, the final grammar $G = (V, T, P, S)$ which can generate the given language is

$S \rightarrow aAB$	[‘a’ followed by a string having n number of a 's followed by n number of b 's followed by zero or more b 's]
$A \rightarrow aAb \mid ab$	[generates $a^n b^n$]
$B \rightarrow bB \mid \epsilon$	[Generates zero or more b 's]

Example 34: Obtain a grammar to generate the language:

$$L = \{a^n b^m \mid n \geq 0, m > n\}$$

Solution: It is clear from the above statement that the set of strings that can be generated by this language can be represented as

$$\begin{array}{lll} n = 0 & n = 1 & n = 2 \\ m \geq 1 & m \geq 2 & m \geq 3 \end{array}$$

$$L = \{\epsilon b b^*, a b b b^*, a a b b b b^*, \dots\}$$

where b^* represents zero or more b's. Observe that the language consists of a string $a^n b^n \mid n \geq 0$ followed by one or more b's. This prompts us to have a production of the form

$$S \rightarrow AB$$

where the language $a^n b^n \mid n \geq 0$ is generated from the variable A and one or more b's are generated from the variable B. The language $a^n b^n \mid n \geq 0$ is generated from the variable A using the productions shown below:

$$A \rightarrow AAb \mid \epsilon$$

and one or more b's are generated from the production:

$$B \rightarrow bB \mid b$$

So, the final grammar $G = (V, T, P, S)$ which can generate the given language is

$$V = \{ S, A, B \}$$

$$T = \{ a, b \}$$

$$P = \{$$

$$S \rightarrow AB$$

$$A \rightarrow aAb \mid \epsilon \quad [\text{Generates } a^n b^n \mid n \geq 0]$$

$$B \rightarrow bB \mid b \quad [\text{Generates one or more b's}]$$

}

S - is the start symbol

Note: The same language can also be generated from the following grammar:

$$S \rightarrow aSb \mid B \quad [\text{Generates } a^n b^n \mid n \geq 0 \text{ followed by at least one b}]$$

$$B \rightarrow bB \mid b \quad [\text{Generates one or more b's}]$$

Example 35: Obtain a grammar to generate the language:

$$L = \{ a^n b^{n-3} \mid n \geq 3 \}$$

Note: It is clear from the above statement that the set of strings that can be generated by this language can be represented as

$$n=3 \quad n=4 \quad n=5 \quad n=6$$

$$L = \{ aaa, aaaab, aaaaabb, aaaaaabb, \dots \}$$

It is observed from the above set that the string ‘aaa’ is followed by the string $a^n b^n \mid n \geq 0$. So, the first production that we can think of is

$$S \rightarrow aaaA$$

where the string $a^n b^n \mid n \geq 0$ can be obtained using A as shown below:

$$A \rightarrow aAb \mid \epsilon$$

So, the final grammar that can generate the given language is

$$V = \{ S, A \}$$

$$T = \{ a, b \}$$

$$P = \{$$

$$S \rightarrow aaaA$$

$$A \rightarrow aAb \mid \epsilon$$

}

S - is the start symbol

Note: The same language can also be generated from the grammar with the following productions:

$$S \rightarrow aSb \mid aaa$$

■ **Example 36:** Obtain a grammar to generate the language:

$$L = L_1 L_2$$

where

$$L_1 = \{ a^n b^m \mid n \geq 0, m > n \}$$

$$L_2 = \{ 0^n 1^{2n} \mid n \geq 0 \}$$

Solution: The grammar corresponding to the language L_1 is already obtained in Example 3.

For convenience it is provided again:

$$S_1 \rightarrow aS_1b \mid bB$$

$$B \rightarrow bB \mid b$$

The grammar corresponding to the language L_2 is already obtained in Example 24. For convenience it is provided again:

$$\begin{aligned}V &= \{S_2\} \\T &= \{0, 1\} \\P &= \{ \\&\quad S_2 \rightarrow 0S_211 \mid \epsilon \\&\}\end{aligned}$$

S_2 is the start symbol

The resulting language

$$L = L_1 L_2$$

can be generated from the grammar obtained by concatenating the start symbol S_1 of first grammar with the start symbol S_2 of the second grammar as

$$S \rightarrow S_1 S_2$$

where S is the start symbol of the resultant grammar. So, the final grammar which accepts

$$L = L_1 L_2$$

is shown below:

$$\begin{aligned}V &= \{S, S_1, S_2, B\} \\T &= \{a, b, 0, 1\} \\P &= \{ \\&\quad S \rightarrow S_1 S_2 \\&\quad S_1 \rightarrow aS_1b \mid bB \\&\quad S_2 \rightarrow 0S_211 \mid \epsilon \\&\quad B \rightarrow bB \mid b \\&\}\end{aligned}$$

S_1 is the start symbol

Example 37: Obtain a grammar to generate the language:

$$L = L_1 \cup L_2$$

where

$$\begin{aligned}L_1 &= \{a^n b^m \mid n \geq 0, m > n\} \\L_2 &= \{0^n 1^{2n} \mid n \geq 0\}\end{aligned}$$

Note: This problem is similar to the previous problem, except that from the start symbol S either S_1 is derived or S_2 is derived as shown below.

$$S \rightarrow S_1 | S_2$$

The rest of the productions remain same. The final grammar is shown below (both the grammars below generate the same language):

$$V = \{ S, S_1, S_2, A, B \}$$

$$T = \{a, b, 0, 1\}$$

$$P = \{$$

$$S \rightarrow S_1 | S_2$$

$$S_1 \rightarrow AB$$

$$S_2 \rightarrow 0S_211 | \epsilon$$

$$A \rightarrow aAb | \epsilon$$

$$B \rightarrow bB | b$$

}

S_1 - is the start symbol

$$V = \{ S, S_1, S_2, B \}$$

$$T = \{a, b, 0, 1\}$$

$$P = \{$$

$$S \rightarrow S_1 | S_2$$

$$S_1 \rightarrow aS_1b | bB$$

$$S_2 \rightarrow 0S_211 | \epsilon$$

$$B \rightarrow bB | \epsilon$$

}

S_1 - is the start symbol

Example 38: Obtain a grammar to generate the language:

$$L = \{w : |w| \bmod 3 \neq |w| \bmod 2\} \text{ on } \Sigma = \{a\}$$

Solution: The following grammar also accepts the same language:

$$V = \{S\}$$

$$T = \{a\}$$

$$P = \{$$

$$S \rightarrow aa | aaa | aaaa | aaaaa | aaaaaaS$$

}

S - is the start symbol

Note: Another easy method and totally different grammar can be obtained by drawing the DFA and then converting it into grammar (see the DFA shown in Chapter 1):

Example 39: Obtain a grammar to generate the language:

$$L = \{w : |w| \bmod 3 \geq |w| \bmod 2\} \text{ on } \Sigma = \{a\}$$

Solution: The grammar to accept the above language is shown below:

$V = \{S\}$
 $T = \{a\}$
 $P = \{$
 $S \rightarrow \epsilon | a | aa | aaaa | aaaaa | aaaaaaS$
 $\}$
 $S - \text{is the start symbol}$

Note: Another easy method and totally different grammar can be obtained by drawing the DFA and then converting it into grammar (see the DFA shown in Chapter 1).

■ **Example 40:** Obtain a grammar to generate set of all strings with exactly one a when $\Sigma = \{a, b\}$.

Since $w \in L$ should have exactly one a , this single a can be preceded by any number of b 's which can be achieved using the production

$$S \rightarrow bS | aB$$

Note that each time bS is substituted in place of S , one extra b is generated. Finally, if S is replaced by aB , we will have exactly one a followed by a non-terminal B . From this B , we should generate any number of b 's and can be achieved using the production

$$B \rightarrow bB | \epsilon$$

So, the final grammar to generate at least one a is

$V = \{S, B\}$
 $T = \{a, b\}$
 $P = \{$
 $S \rightarrow bS | aB$
 $B \rightarrow bB | \epsilon$
 $\}$
 $S - \text{is the start symbol}$

■ **Example 41:** Obtain a grammar to generate set of all strings with at least one a when $\Sigma = \{a, b\}$.

Solution: String consisting of at least one a implies one or more a 's. These one or more a 's can be preceded by any number of b 's which can be achieved using the production

$$S \rightarrow bS$$

Note that each time bS is substituted in place of S , one extra b is generated. Once there are no b 's, we should be in a position to generate at least one a from S and can be generated by the production

$$S \rightarrow aA$$

Once one a is generated, this a can be followed by any number of a 's and/or b 's. The productions to generate such a 's and b 's are

$$A \rightarrow aA \mid bA \mid \epsilon$$

So, the final grammar to generate at least one a is

$$\begin{aligned} V &= \{S, A\} \\ T &= \{a, b\} \\ P &= \{ \\ &\quad S \rightarrow bS \mid aA \\ &\quad A \rightarrow aA \mid bA \mid \epsilon \\ &\quad \} \\ S &- \text{is the start symbol} \end{aligned}$$

Example 42: Obtain a grammar to generate the set of all strings with no more than three a 's when $\Sigma = \{a, b\}$.

Solution: String containing note more than three a 's implies that

1. there can be no a 's at all
2. there can be only one a
3. there can be only two a 's
4. there can be only three a 's

But, at any point of time number of a 's should not exceed 3 and the string can have any number of b 's. Let us take all these four cases one by one.

Case 1: there can be no a 's at all, but there is no restriction on the number of b 's. The production corresponding to this is

$$S \rightarrow bS \mid \epsilon$$

Case 2: There can be only one a . The single a can be obtained by the start symbol S using the production

$$S \rightarrow aA$$

After applying this production, we get one a followed by a non-terminal A. From this we can generate any number of b 's using the production

$$A \rightarrow bA \mid \epsilon$$

and the string can be accepted. After the input of some b 's, we can input one more a which is the next case.

Case 3: There can be two a 's. When we get the non-terminal A, we would have got one a . To get one more a , we can have the production

$$A \rightarrow aB$$

After applying this production, we would have got some number of b 's and exactly two a 's where the second a is followed by a non-terminal B. Any number of b 's can be generated from B and the corresponding productions are:

$$B \rightarrow bB \mid \epsilon$$

After applying this production some number of times, we have two a 's followed by zero or more b 's. After some b 's, we can input the third a which also should be accepted and the corresponding production is

$$B \rightarrow aC$$

After applying this production, we have accepted three a 's which can be followed by any number of b 's. The corresponding productions are

$$C \rightarrow bC \mid \epsilon$$

Now we have accepted maximum of three a 's and no more a 's can be accepted thus producing not more than three a 's. So, the final grammar is

$$\begin{aligned} S &\rightarrow bS \mid aA \mid \epsilon \\ A &\rightarrow bA \mid aB \mid \epsilon \\ B &\rightarrow bB \mid aC \mid \epsilon \\ C &\rightarrow bC \mid \epsilon \end{aligned}$$

Note: For the language another grammar can be generated as shown below: String containing note more than three a 's implies that

1. there can be no a 's at all. The corresponding production can be $S \rightarrow B$
2. there can be only one a . The corresponding production is $S \rightarrow BaB$
3. there can be only two a 's. The corresponding production is $S \rightarrow BaBaB$
4. there can be only three a 's. The corresponding production is $S \rightarrow BaBaBaB$

Now, the string containing any number of b's can be generated from the production $B \rightarrow bB | \epsilon$.
 So, the final grammar to generate the given language is given by $G = (V, T, P, S)$ where

$$\begin{aligned} V &= \{S, B\} \\ T &= \{a, b\} \\ P &= \{ \\ &\quad S \rightarrow B \mid BaB \mid BaBaB \mid BaBaRaB \\ &\quad B \rightarrow bB \mid \epsilon \\ &\quad \} \\ S &= \text{is the start symbol} \end{aligned}$$

Example 43: Obtain a grammar to generate the language $L = \{w \mid n_a(w) = n_b(w) + 1\}$.

Solution: The problem is similar to Example 19, except that $w \in L$ should have one more a either in the beginning or at the end or at the middle. This can be achieved using the production

$$S \rightarrow AaA$$

where A generates equal number of a 's and b 's using the productions

$$\begin{aligned} A &\rightarrow aAb \\ A &\rightarrow bAa \\ A &\rightarrow AA \\ A &\rightarrow \epsilon \end{aligned}$$

So, the final grammar to generate the language

$$L = \{w \mid n_a(w) = n_b(w) + 1\}$$

is shown below:

$$\begin{aligned} S &\rightarrow AaA \\ A &\rightarrow aAb \mid bAa \mid AA \mid \epsilon \end{aligned}$$

Example 44: Obtain the grammar to generate the language

$$L = \{w \mid n_a(w) > n_b(w)\}$$

Solution: We have already seen that the following grammar produces equal number of a 's and b 's:

$$\begin{aligned} A &\rightarrow aAb \\ A &\rightarrow bAa \\ A &\rightarrow AA \\ A &\rightarrow \epsilon \end{aligned}$$

Since number of a's should be greater than number of b's, w should be in a position to generate 1 or more a's. One or more a's can be generated using the production:

$$B \rightarrow aB \mid a$$

Since more number of a's can occur in the beginning, at the end or at the middle, we should have a production to generate as many a's as possible in the respective places. Equal number of a's and b's can be followed by one or more a's which can be achieved by introducing the production

$$S \rightarrow AB$$

Equal number of a's and b's can be preceded by one or more a's which can be achieved by introducing the production

$$S \rightarrow BA$$

Equal number of a's and b's can have one or more a's in the middle which can be achieved by introducing the production

$$S \rightarrow ABA$$

So, the final grammar to generate the language

$$L = \{w \mid n_a(w) > n_b(w)\}$$

is $G = (V, T, P, S)$ where

$$\begin{aligned} V &= \{S, A, B\} \\ T &= \{a, b\} \\ P &= \{ \\ &\quad S \rightarrow AB|BA|ABA \\ &\quad A \rightarrow aAb \\ &\quad A \rightarrow bAa \\ &\quad A \rightarrow AA \\ &\quad A \rightarrow \epsilon \\ &\quad B \rightarrow aB \mid a \\ &\quad \} \\ S &\text{ is the start symbol} \end{aligned}$$

Example 45: Obtain a grammar to generate a language of strings of 0's and 1's having a substring 000

$$\text{i.e., } L = \{w \mid w \in \{0,1\}^* \text{ with at least one occurrence of '000'}\}$$

Solution: It is clear from this definition that the substring 000 can be preceded and followed by strings of 0's and 1's of any length which can be generated using the production of the form

$$S \rightarrow A000A$$

where any strings of 0's and 1's are generated from the non-terminal A using the productions

$$A \rightarrow 0A \mid 1A \mid \epsilon$$

So, the final grammar to generate the given language is $G = (V, T, P, S)$ where

$$\begin{aligned} V &= \{S\} \\ T &= \{0,1\} \\ P &= \{ \\ &\quad S \rightarrow A000A \\ &\quad A \rightarrow 0A \mid 1A \mid \epsilon \\ &\quad \} \\ S &\text{ is the start symbol} \end{aligned}$$

■ **Example 46:** Obtain a grammar to generate the following language

$$\text{i.e., } L = \{a^n b^m c^k \mid n + 2m = k \text{ for } n \geq 0, m \geq 0\}$$

Solution: It is required to generate the grammar for the language

$$L = \{a^n b^m c^k \mid n + 2m = k \text{ for } n \geq 0, m \geq 0\}$$

Let us express k in terms of n and m in $a^n b^m c^k$. So, after substituting for k using $k = n + 2m$ we have:

$$L = \{a^n b^m c^{n+2m} \mid n \geq 0, m \geq 0\}$$

which is equivalent to

$$L = \{a^n b^m c^{2m} c^n \mid n \geq 0, m \geq 0\}$$

So, it is clear from these examples that the given language can be expressed as:

$$L = \{w \mid a^n b^m c^{2m} c^n \text{ where } m \geq 0, n \geq 0\}$$

The sub string $b^m c^{2m}$ can be generated from the following productions:

$$A \rightarrow bAcc \mid \epsilon$$

The substring thus generated from A-production is enclosed between a^n and c^n and the corresponding productions can be written as

$$S \rightarrow aSc | A$$

So, the final grammar to generate the given language is $G = (V, T, P, S)$ where

$$\begin{aligned} V &= \{S\} \\ T &= \{a,b,c\} \\ P &= \{ \\ &\quad S \rightarrow aSc | A \\ &\quad A \rightarrow bAcc | \epsilon \\ &\quad \} \\ S &\text{ is the start symbol} \end{aligned}$$

I Example 47: Obtain a grammar to generate integers.

Solution: The sign of a number can be '+' or '-' or ϵ . The production for this can be written as

$$S \rightarrow + | - | \epsilon$$

In the above production, if ϵ is derived from S, the sign for a number will not be generated. A number can be formed from any of the digits 0,1,2, ..., 9. The production to obtain these digits can be written as

$$D \rightarrow 0 | 1 | 2 | \dots | 9$$

A number N can be recursively defined as follows:

1. A digit is a number (i.e., $N \rightarrow D$)
2. The number followed by a digit is a number (i.e., $N \rightarrow ND$)

or

a digit followed by number is also a number (i.e., $N \rightarrow DN$)

The productions for this recursive definition can be written as

$$\begin{aligned} N &\rightarrow D \\ N &\rightarrow ND | DN \end{aligned}$$

An integer number I can be a number N or a sign (an optional plus and a minus) followed by number N. The production for this can be written as

$$I \rightarrow N | SN$$

So, the grammar G to obtain integer numbers can be written as

$$G = (V, T, P, S)$$
 where

$V = \{ D, S, N, I \}$	
$T = \{ +, -, 0, 1, 2, \dots, 9 \}$	
$P = \{$	
$I \rightarrow N \mid SN$	[Generate signed or unsigned number]
$N \rightarrow D \mid ND \mid DN$	[Generates one or more digits]
$S \rightarrow + \mid - \mid \epsilon$	[Generate the sign]
$D \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$	[Generate the digits]
}	
$S = I$ which is the start symbol	

The unsigned number 1965 and signed number +1965 can be derived as shown below:

$I \Rightarrow N$	$I \Rightarrow SN$
$\Rightarrow ND$	$\Rightarrow +N$
$\Rightarrow N5$	$\Rightarrow +ND$
$\Rightarrow ND5$	$\Rightarrow +N5$
$\Rightarrow N65$	$\Rightarrow +ND5$
$\Rightarrow ND65$	$\Rightarrow +N65$
$\Rightarrow N965$	$\Rightarrow +ND65$
$\Rightarrow D965$	$\Rightarrow +N965$
$\Rightarrow 1965$	$\Rightarrow +D965$
	$\Rightarrow +1965$

■ **Example 48:** Let $G = (V, T, P, S)$ be a CFG where

$$\begin{aligned} V &= \{ S \} \\ T &= \{ a, b \} \\ P &= \{ \\ &\quad S \rightarrow aSa \mid bSb \mid \epsilon \\ &\quad \} \end{aligned}$$

S is the start symbol.

What is the language generated by this grammar?

Solution: The null string ϵ can be obtained by applying the production $S \rightarrow \epsilon$ and the derivation is shown below:

$$S \Rightarrow \epsilon \quad (\text{By applying } S \rightarrow \epsilon)$$

Consider the derivation

$$\begin{aligned}
 S &\Rightarrow aSa && (\text{By applying } S \rightarrow aSa) \\
 &\Rightarrow abSba && (\text{By applying } S \rightarrow bSb) \\
 &\Rightarrow abbSbba && (\text{By applying } S \rightarrow bSb) \\
 &\Rightarrow abbbSbbbba && (\text{By applying } S \rightarrow bSb) \\
 &\Rightarrow abbbbbba && (\text{By applying } S \rightarrow \epsilon)
 \end{aligned}$$

So, by applying the productions

$$S \rightarrow aSa \quad \text{and} \quad S \rightarrow bSb$$

any number of times and in any order and finally applying the production

$$S \rightarrow \epsilon$$

we get a string w followed by reverse of w denoted by w^R . So, the language generated by this grammar is

$$L = \{w w^R \mid w \in \{a+b\}^*\}$$

As this language is generated from the context free grammar, this is context free language.

Example 49: Obtain a grammar to generate an arithmetic expression using the operators $+$, $-$, $*$, $/$ and $^\wedge$ (indicating power). An identifier can start with any of the letters from $\{a, b, c\}$ and can be followed by zero or more symbols from $\{a, b, c\}$.

An arithmetic expression can be recursively defined as follows:

1. An expression E can be an identifier.
 2. If E is any arithmetic expression then
 - i. $E + E$
 - ii. $E - E$
 - iii. $E * E$
 - iv. E / E
 - v. $E ^ E$
 - vi. (E)
- are all arithmetic expressions.

An identifier I of length at least one can be generated using any combinations of a's, b's and c's using the following productions

$$I \rightarrow Ia \mid Ib \mid Ic \mid a \mid b \mid c$$

By first definition of an arithmetic expression I is an arithmetic expression which can be obtained using the production

$$E \rightarrow I$$

By the second definition of an arithmetic expression the various productions that can be obtained are:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow E^{\wedge} E \\ E &\rightarrow (E) \end{aligned}$$

So, the complete grammar to generate an arithmetic expression is shown below:

$$\begin{aligned} V &= \{E, I\} \\ T &= \{+, -, *, /, ^, a, b, c\} \\ P &= \{ \\ &\quad E \rightarrow I \\ &\quad E \rightarrow E + E \\ &\quad E \rightarrow E - E \\ &\quad E \rightarrow E * E \\ &\quad E \rightarrow E / E \\ &\quad E \rightarrow E^{\wedge} E \\ &\quad E \rightarrow (E) \\ &\quad I \rightarrow Ia | Ib | Ic | a | b | c \\ &\} \end{aligned}$$

S is the start symbol

The productions in P can also be written as shown below:

$$\begin{aligned} E &\rightarrow I \\ E &\rightarrow E + E | E - E | E * E | E / E | E^{\wedge} E | (E) \\ I &\rightarrow Ia | Ib | Ic | a | b | c \end{aligned}$$

4.7. Leftmost Derivation

Example 50: Consider the grammar shown below from which any arithmetic expression can be obtained.

$$\begin{aligned}
 E &\rightarrow E + E \\
 E &\rightarrow E - E \\
 E &\rightarrow E * E \\
 E &\rightarrow E / E \\
 E &\rightarrow E^E \\
 E &\rightarrow \text{id}
 \end{aligned}$$

Note: The symbol \wedge denotes exponentiation.

The non-terminal E is used instead of using the word *expression*. The left-hand side of the first production i.e., E is considered to be the start symbol. Obtain the string $\text{id} + \text{id} * \text{id}$ and show the derivation for the same.

Solution: The derivation to get the string $\text{id} + \text{id} * \text{id}$ is shown below:

$$\begin{aligned}
 E &\xrightarrow{\text{lm}} E + E \\
 &\Rightarrow \text{id} + E \\
 &\Rightarrow \text{id} + E * E \\
 &\Rightarrow \text{id} + \text{id} * E \\
 &\Rightarrow \text{id} + \text{id} * \text{id}
 \end{aligned}$$

Note that at each step in the derivation process, only the left most variable E is replaced and so the derivation is said to be leftmost. The string $\text{id} + \text{id} * \text{id}$ is obtained from the start symbol E by applying leftmost derivation and can be written as

$$E \xrightarrow[\text{lm}]{+} \text{id} + \text{id} * \text{id}$$

Indicating the string $\text{id} + \text{id} * \text{id}$ is derived from E by applying more than one production. The symbol $\xrightarrow[\text{lm}]{+}$ denotes that one or more steps are used in the derivation whereas the symbol $\xrightarrow[\text{lm}]{*}$ denotes that zero or more steps are used in the derivation. Now, let us see “What is left most derivation?” The leftmost derivation can be defined as follows:

❖ **Definition:** In the derivation process if a left most variable is replaced at every step, then the derivation is said to be leftmost and is shown in Example 49. It is clear from leftmost derivation that each of the following:

$$\{E, E + E, \text{id} + E, \text{id} + E * E, \text{id} + \text{id} * E, \text{id} + \text{id} * \text{id}\}$$

can be obtained from the start symbol. Each string in the set is called the sentential form.

Now, let us see “What is sentential form?” The formal definition of *sentential form* is shown below:

❖ **Definition:** Let $G = (V, T, P, S)$ be a CFG. Any string $w \in (V \cup T)^*$ which is derivable from the start symbol S (denoted by $S \xrightarrow{*} \alpha$) is called a sentence or *sentential form* of G . If there is a

derivation of the form $S \xrightarrow{lm}^* \alpha$, where at each step in the derivation process only a left most variable is replaced, then α is called *left-sentential* form and if there is a derivation of the form $S \xrightarrow{rm}^* \alpha$, where at each step in the derivation process only a right most variable is replaced, then α is called *right-sentential* form.

Example 51: Obtain the leftmost derivation for the string aaabbabbba using the following grammar:

$$\begin{array}{lcl} S & \rightarrow & aB \mid bA \\ A & \rightarrow & aS \mid bAA \mid a \\ B & \rightarrow & bS \mid aBB \mid b \end{array}$$

The leftmost derivation for the string aaabbabbba is shown below:

$$\begin{array}{ll} S & \xrightarrow{lm} aB \quad (\text{Applying } S \rightarrow aB) \\ & \Rightarrow aaBB \quad (\text{Applying } B \rightarrow aBB) \\ & \Rightarrow aaaBBB \quad (\text{Applying } B \rightarrow aBB) \\ & \Rightarrow aaabBB \quad (\text{Applying } B \rightarrow b) \\ & \Rightarrow aaabbB \quad (\text{Applying } B \rightarrow b) \\ & \Rightarrow aaabbaBB \quad (\text{Applying } B \rightarrow aBB) \\ & \Rightarrow aaabbabB \quad (\text{Applying } B \rightarrow b) \\ & \Rightarrow aaabbabbS \quad (\text{Applying } B \rightarrow bS) \\ & \Rightarrow aaabbabbA \quad (\text{Applying } S \rightarrow bA) \\ & \Rightarrow aaabbabbba \quad (\text{Applying } A \rightarrow a) \end{array}$$

4.8. Rightmost Derivation

❖ **Definition:** In the derivation process if a right most variable is replaced at every step, then the derivation is said to be rightmost.

The rightmost derivation for the grammar shown in Example 49 is shown below.

$$\begin{array}{ll} E & \xrightarrow{rm} E + E \\ & \Rightarrow E + E * E \\ & \Rightarrow E + E * id \\ & \Rightarrow E + id * id \\ & \Rightarrow id + id * id \end{array}$$

Note that at each step in the derivation process, only the right most variable E is replaced and so the derivation is said to be right most. The string $\text{id} + \text{id} * \text{id}$ is obtained from the start symbol E by applying right most derivation and can be written as

$$E \xrightarrow{\text{Rm}} \text{id} + \text{id} * \text{id}$$

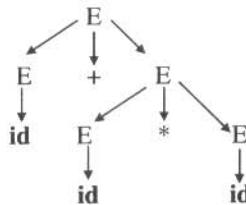
4.9. Derivation Tree (Parse Tree)

The derivation can be shown in the form of a tree. Such trees are called derivation or parse trees. The leftmost derivation as well as the right most derivation can be represented using derivation trees. Now, let us see “What is derivation tree or parse tree?” The derivation tree can be defined as shown below.

Definition (Parse tree or Derivation Tree): Let $G = (V, T, P, S)$ be a CFG. The tree is derivation tree (parse tree) with the following properties.

1. The root has the label S.
2. Every vertex has a label which is in $(V \cup T \cup \epsilon)$.
3. Every leaf node has label from T and an interior vertex has a label from V.
4. If a vertex is labeled A and if $X_1, X_2, X_3, \dots, X_n$ are all children of A from left, then $A \rightarrow X_1 X_2 X_3 \dots X_n$ must be a production in P.

The parse tree for the right most derivation shown in Section 4.8 is shown below:



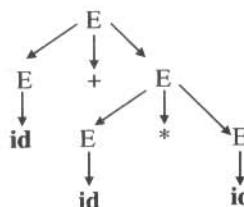
In the above derivation tree, let us read only the leaf nodes from left to right (leaving the ϵ -symbols if any). The string obtained is

$$\text{id} + \text{id} * \text{id}$$

is called the *yield* of the tree. Now, let us see “What is the yield of the tree?”. Then, the *yield* of a tree can be formally defined as follows:

❖ **Definition:** The *yield* of a tree is the string of symbols obtained by only reading the leaves of the tree from *left to right* without considering the ϵ -symbols. The yield of the tree is derived always from the root and the yield of the tree is always a terminal string.

For example, consider the derivation tree (or parse tree) shown below:



If we read only the terminal symbols in this tree from left to right we get **id + id * id** and is the yield of the given parse tree. Now, let us see “What is partial derivation tree?”. The partial parse tree(partial derivation tree) is defined as follows.

Definition (Partial Parse Tree or Partial Derivation Tree): Let $G = (V, T, P, S)$ be a CFG. The tree is partial derivation tree (parse tree) with the following properties:

1. The root has the label S.
2. Every vertex has a label which is in $(V \cup T \cup \epsilon)$.
3. Every leaf node has label from $(V \cup T \cup \epsilon)$.
4. If a vertex is labeled A and if $X_1, X_2, X_3, \dots, X_n$ are all children of A from left, then $A \rightarrow X_1 X_2 X_3 \dots X_n$ must be a production in P.

Note: The difference between parse tree and partial parse tree is same except in the property 3 mentioned.

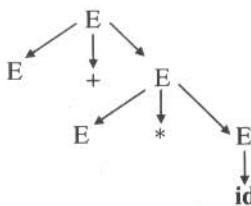
Consider the partial derivation (by applying right most derivation) for the grammar

$$E \rightarrow E + E \mid E * E \mid id$$

is shown below:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow E + E * E \\ &\Rightarrow E + E * id \end{aligned}$$

For this partial right most derivation, the partial derivation tree is shown below:



It is clear from the *parse tree* and *partial parse tree* that all the leaves in parse tree are the symbols from $(T \cup \epsilon)$ whereas in partial parse tree the leaves will be from $(V \cup T \cup \epsilon)$.

4.10. Ambiguous Grammar

In this section, let us see “What is ambiguous grammar”.

❖ **Definition:** Let $G = (V, T, P, S)$ be a context free grammar. A grammar G is *ambiguous* if and only if there exists at least one string $w \in T^*$ for which two or more different parse trees exist by applying either the left most derivation or right most derivation. Note that after applying leftmost derivation or right most derivation even though the derivations look different and if the structure of parse trees obtained are same, we can not jump to the conclusion that the grammar is ambiguous. It is not the multiplicity of the derivations that cause ambiguity. But, it is the existence of two or more parse trees for the same string w derived from the root labeled S .

Note:

1. Obtain the leftmost derivation and get a string w . Obtain the right most derivation and get a string w . For both the derivations construct the parse tree. If there are two different parse trees, then the grammar is ambiguous.
2. Obtain the string w by applying leftmost derivation twice and construct the parse tree. If the two parse trees are different, the grammar is ambiguous.
3. Obtain the string w by applying rightmost derivation twice and construct the parse tree. If the two parse trees are different, the grammar is ambiguous.
4. Apply the leftmost derivation and a get string. Apply the leftmost derivation again and a get a different string. The parse trees obtained will naturally be different and do not come to the conclusion that the grammar is ambiguous.

■ **Example 52:** Consider the grammar shown below from which any arithmetic expression can be obtained.

$$\begin{aligned}
 E &\rightarrow E + E \\
 E &\rightarrow E - E \\
 E &\rightarrow E * E \\
 E &\rightarrow E / E \\
 E &\rightarrow (E) | I \\
 I &\rightarrow id
 \end{aligned}$$

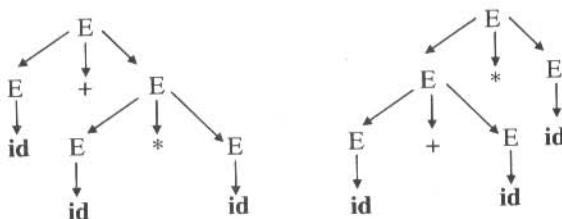
Show that the grammar is ambiguous.

Note: Leftmost derivation, rightmost derivation and parse trees are equivalent.

The sentence **id + id * id** can be obtained from leftmost derivation in two ways as shown below:

$$\begin{array}{ll}
 \begin{array}{l}
 E \Rightarrow E + E \\
 \Rightarrow id + E \\
 \Rightarrow id + E * E \\
 \Rightarrow id + id * E \\
 \Rightarrow id + id * id
 \end{array} &
 \begin{array}{l}
 E \Rightarrow E * E \\
 \Rightarrow E + E * E \\
 \Rightarrow id + E * E \\
 \Rightarrow id + id * E \\
 \Rightarrow id + id * id
 \end{array}
 \end{array}$$

The corresponding derivation trees for the two leftmost derivations are shown below:



Since the two parse trees are different for the same sentence **id + id * id** by applying leftmost derivation, the grammar is ambiguous.

■ **Example 53:** Is the following grammar ambiguous?

$$\begin{aligned}
 S &\rightarrow aS | X \\
 X &\rightarrow aX | a
 \end{aligned}$$

Consider the two leftmost derivations for the string **aaaa**.

$$\begin{array}{l} S \Rightarrow aS \\ \Rightarrow aaS \\ \Rightarrow aaaS \\ \Rightarrow aaaX \\ \Rightarrow aaaa \end{array}$$

$$\begin{array}{l} S \Rightarrow X \\ \Rightarrow aX \\ \Rightarrow aaX \\ \Rightarrow aaaX \\ \Rightarrow aaaa \end{array}$$

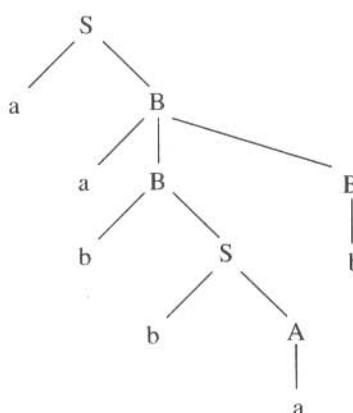
Since there are two leftmost derivations for the same sentence aaaa, the given grammar is ambiguous.

Example 54: Is the following grammar ambiguous?

$$\begin{array}{l} S \rightarrow aB | bA \\ A \rightarrow aS | bAA | a \\ B \rightarrow bS | aBB | b \end{array}$$

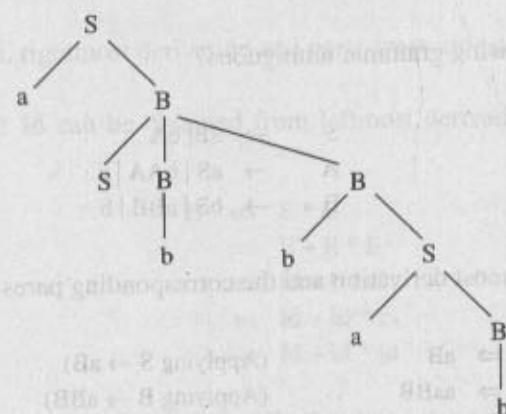
Solution: Consider the leftmost derivation and the corresponding parse tree shown below:

$$\begin{array}{ll} S \Rightarrow aB & (\text{Applying } S \rightarrow aB) \\ \Rightarrow aaBB & (\text{Applying } B \rightarrow aBB) \\ \Rightarrow aabSB & (\text{Applying } B \rightarrow bS) \\ \Rightarrow aabbAB & (\text{Applying } S \rightarrow bA) \\ \Rightarrow aabbaB & (\text{Applying } A \rightarrow a) \\ \Rightarrow aabbab & (\text{Applying } B \rightarrow b) \end{array}$$



The same string $aabbab$ can be obtained again by applying leftmost derivation as shown below:

$S \Rightarrow aB$	(Applying $S \rightarrow aB$)
$\Rightarrow aaBB$	(Applying $B \rightarrow aBB$)
$\Rightarrow aabB$	(Applying $B \rightarrow b$)
$\Rightarrow aabbS$	(Applying $B \rightarrow bS$)
$\Rightarrow aabbaB$	(Applying $S \rightarrow aB$)
$\Rightarrow aabbab$	(Applying $B \rightarrow b$)



Note that there are two parse trees for the string $aabbab$ by applying leftmost derivation and so the given grammar is ambiguous.

Example 55: Obtain the string $aaabbabbba$ by applying left most derivation and the parse tree for the grammar shown below. Is it possible to obtain the same string again by applying leftmost derivation but by selecting different productions?

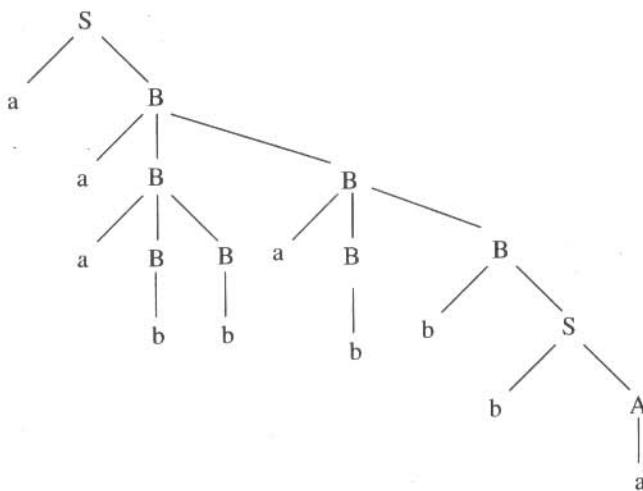
$$\begin{array}{lcl} S & \rightarrow & aB \mid bA \\ A & \rightarrow & aS \mid bAA \mid a \\ B & \rightarrow & bS \mid aBB \mid b \end{array}$$

The leftmost derivation for the string $aaabbabbba$ is shown below:

$S \Rightarrow aB$	(Applying $S \rightarrow aB$)
$\Rightarrow aaBB$	(Applying $B \rightarrow aBB$)
$\Rightarrow aaaBBB$	(Applying $B \rightarrow aBB$)
$\Rightarrow aaabBB$	(Applying $B \rightarrow b$)

$\Rightarrow aaabbB$	(Applying $B \rightarrow b$)
$\Rightarrow aaabbaBB$	(Applying $B \rightarrow aBB$)
$\Rightarrow aaabbabB$	(Applying $B \rightarrow b$)
$\Rightarrow aaabbabbS$	(Applying $B \rightarrow bS$)
$\Rightarrow aaabbabbbA$	(Applying $S \rightarrow bA$)
$\Rightarrow aaabbabbbba$	(Applying $A \rightarrow a$)

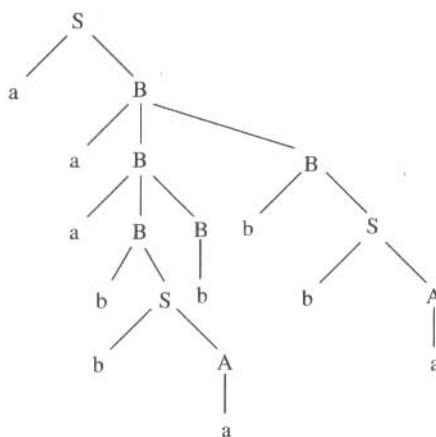
The parse tree for the above derivation is shown below:



The leftmost derivation for the same string aaabbabbbba but by applying different set of productions is shown below:

$S \Rightarrow aB$	(Applying $S \rightarrow aB$)
$\Rightarrow aaBB$	(Applying $B \rightarrow aBB$)
$\Rightarrow aaaBBB$	(Applying $B \rightarrow aBB$)
$\Rightarrow aaabSBB$	(Applying $B \rightarrow bS$)
$\Rightarrow aaabbABB$	(Applying $S \rightarrow bA$)
$\Rightarrow aaabbaBB$	(Applying $A \rightarrow a$)
$\Rightarrow aaabbabB$	(Applying $B \rightarrow b$)
$\Rightarrow aaabbabbS$	(Applying $B \rightarrow bS$)
$\Rightarrow aaabbabbbA$	(Applying $S \rightarrow bA$)
$\Rightarrow aaabbabbbba$	(Applying $A \rightarrow a$)

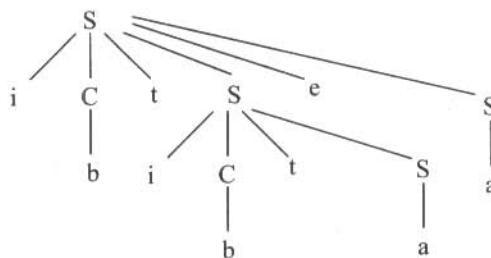
The parse tree for this derivation is shown in figure below:



The string ibtibtaea can be obtained again by applying the leftmost derivation but using different sets of productions as shown below:

$$\begin{aligned} S &\Rightarrow iCtSeS \\ &\Rightarrow ibtSeS \\ &\Rightarrow ibtiCtSeS \\ &\Rightarrow ibtibtSeS \\ &\Rightarrow ibtibtaeS \\ &\Rightarrow ibtibtaea \end{aligned}$$

The parse tree for this is shown below:



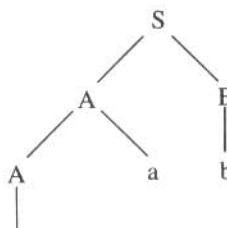
Since there are two different parse trees for the string 'ibtibtaea' by applying leftmost derivation the given grammar is ambiguous.

I Example 57: Is the grammar ambiguous?

$$\begin{aligned} S &\rightarrow AB \mid aaB \\ A &\rightarrow a \mid Aa \\ B &\rightarrow b \end{aligned}$$

Consider the left most derivation for the string *aab* and the corresponding pares tree:

$$\begin{aligned} S &\Rightarrow AB \\ &\Rightarrow \\ &\Rightarrow \\ &\Rightarrow aab \end{aligned}$$



Consider the left most derivation again for the string aab but using different set of production along with the parse tree:

$$\begin{array}{l} S \Rightarrow aaB \\ \Rightarrow aab \end{array} \quad \begin{array}{c} S \\ | \\ a \quad a \quad B \\ | \\ b \end{array}$$

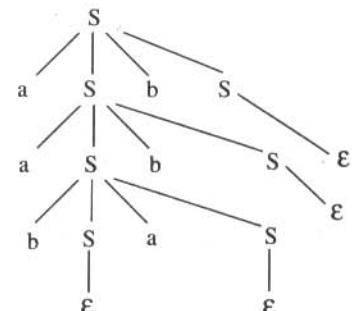
Since there are two parse trees for the string aab , the given grammar is ambiguous.

Example 58: Show that the following grammar is ambiguous

$$\begin{array}{l} S \rightarrow aSbS \\ S \rightarrow bSaS \\ S \rightarrow \epsilon \end{array}$$

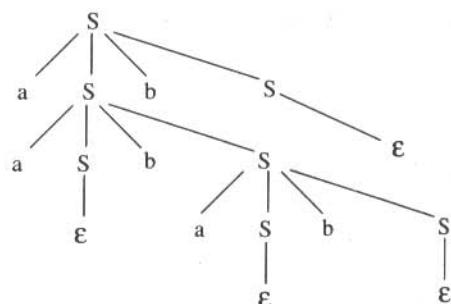
Consider the leftmost derivation for the string $aababb$ and the corresponding parse tree:

$$\begin{array}{ll} S \Rightarrow aSbS & \text{by using } S \rightarrow aSbS \\ \Rightarrow & \\ \Rightarrow aabSaSbSbS & \text{by using } S \rightarrow bSaS \\ \Rightarrow aabaSbSbS & \text{by using } S \rightarrow \epsilon \\ \Rightarrow & \\ \Rightarrow aababbS & \text{by using } S \rightarrow \epsilon \\ \Rightarrow & \end{array}$$



Consider the left most derivation again for the string $aababb$ but using different set of productions:

$$\begin{array}{ll} S \Rightarrow aSbS & \text{by using } S \rightarrow aSbS \\ \Rightarrow aaSbSbS & \text{by using } S \rightarrow aSbS \\ \Rightarrow aabSbS & \text{by using } S \rightarrow \epsilon \\ \Rightarrow AabaSbSbS & \text{by using } S \rightarrow aSbS \\ \Rightarrow aababSbS & \text{by using } S \rightarrow \epsilon \\ \Rightarrow aabbS & \text{by using } S \rightarrow \epsilon \\ \Rightarrow aababb & \text{by using } S \rightarrow \epsilon \end{array}$$



Since there are two parse trees for the string *aababb* by applying leftmost derivation, the grammar is ambiguous.

Note: Instead of deriving the string **aababb** in the above example, we can derive the string “**abab**” so that two different parse trees are obtained and hence we can show that the grammar is ambiguous.

I Example 59: Obtain the unambiguous grammar for the grammar shown below:

$$\begin{array}{l} E \rightarrow E + E \mid E - E \\ E \rightarrow E * E \mid E / E \\ E \rightarrow (E) \mid I \\ I \rightarrow a \mid b \mid c \end{array}$$

and obtain the derivation for the expression $(a+b)^* (a-b)$.

It has been proved that the grammar is ambiguous. This grammar can be converted into unambiguous grammar based on the precedence. We know that identifiers such as *a*, *b* and *c* have the highest precedence, then the expression within ‘(’ and ‘)’ and then * or / whichever occurs first from left to right and finally + or – whichever occurs first from left to right. So, the final grammar which is unambiguous is shown below by assuming * and / have the highest priority compared to that of + and – and also by assuming all these operators are left associative:

$$\begin{array}{l} I \rightarrow a \mid b \mid c \\ F \rightarrow (E) \mid I \\ T \rightarrow T * F \mid T / F \mid F \\ E \rightarrow E + T \mid E - T \mid T \end{array}$$

So, the final grammar which is unambiguous is shown below:

$$\begin{array}{l} E \rightarrow E + T \mid E - T \mid T \\ T \rightarrow T * F \mid T / F \mid F \\ F \rightarrow (E) \mid I \\ I \rightarrow a \mid b \mid c \end{array}$$

The derivation for the string $(a + b)^* (a - b)$ is shown below:

$$\begin{aligned} E &\Rightarrow T \\ &\Rightarrow T * F \\ &\Rightarrow F * F \\ &\Rightarrow (E) * F \\ &\Rightarrow (E+T) * F \end{aligned}$$

$$\begin{aligned}
 &\Rightarrow (T+T) * F \\
 &\Rightarrow (F+T) * F \\
 &\Rightarrow (I+T) * F \\
 &\Rightarrow (a+T) * F \\
 &\Rightarrow (a+F) * F \\
 &\Rightarrow (a+I) * F \\
 &\Rightarrow (a+b) * F \\
 &\Rightarrow (a+b) * (E) \\
 &\Rightarrow (a+b) * (E-T) \\
 &\Rightarrow (a+b) * (T-T) \\
 &\Rightarrow (a+b) * (F-T) \\
 &\Rightarrow (a+b) * (I-T) \\
 &\Rightarrow (a+b) * (a-T) \\
 &\Rightarrow (a+b) * (a-F) \\
 &\Rightarrow (a+b) * (a-I) \\
 &\Rightarrow (a+b) * (a-b)
 \end{aligned}$$

So, the string $(a+b)^* (a-b)$ is derived from the given grammar.

Note: If we assume + and – are having highest priority compared to that of * and / and all the operators left associate the grammar will be of the form:

$$\begin{aligned}
 E &\rightarrow E * T \mid E / T \mid T \\
 T &\rightarrow T + F \mid T - F \mid F \\
 F &\rightarrow (E) \mid I \\
 I &\rightarrow a \mid b \mid c
 \end{aligned}$$

Note: If we assume + and – are having highest priority compared to that of * and / and all the operators right associate the grammar will be of the form:

$$\begin{aligned}
 E &\rightarrow T * E \mid T / E \mid T \\
 T &\rightarrow F + T \mid F - T \mid F \\
 F &\rightarrow (E) \mid I \\
 I &\rightarrow a \mid b \mid c
 \end{aligned}$$

Note: If we assume + and – are having highest priority and left associate when compared to that of * and / and if * and / operators are right associate the grammar will be of the form:

$$\begin{array}{lcl} E & \rightarrow & T^* E \mid T/E \mid T \\ T & \rightarrow & T+F \mid T-F \mid F \\ F & \rightarrow & (E) \mid I \\ I & \rightarrow & a \mid b \mid c \end{array}$$

Note: Whenever a left associative operator is involved use left recursive production and if the operator involved is right-associative, use right recursive production.

Note: Let us see “What is inherently ambiguous grammar?”. A grammar G is *ambiguous* if there exists some string $w \in L(G)$ for which there are two or more distinct derivation trees. If there exists a language L for which there is no unambiguous grammar, then the language is called *inherently ambiguous language* and the grammar from which the language is derived is called *inherently ambiguous grammar*.

For example, the grammar shown in Example 58 is ambiguous. But, we have an equivalent unambiguous for the grammar (see Example 58) as shown above.

There are some inherently ambiguous grammars (for these grammars unambiguous grammars do not exist). For example, consider the language:

$$L = \{a^n b^n c^m d^m \mid m \geq 1, n \geq 1\} \cup \{a^n b^m c^m d^n \mid m \geq 1, n \geq 1\}$$

The grammar to generate the language is shown below:

$$\begin{array}{lcl} S & \rightarrow & AB \mid C \\ A & \rightarrow & aAb \mid ab \\ B & \rightarrow & cBd \mid cd \\ C & \rightarrow & aCd \mid aDd \\ D & \rightarrow & bDc \mid bc \end{array}$$

The string $abcd$ can be derived from the grammar as shown below:

$$\begin{array}{ll} S \Rightarrow AB & \text{by using } S \rightarrow AB \\ \Rightarrow abB & \text{by using } A \rightarrow ab \\ \Rightarrow abcd & \text{by using } B \rightarrow cd \end{array}$$

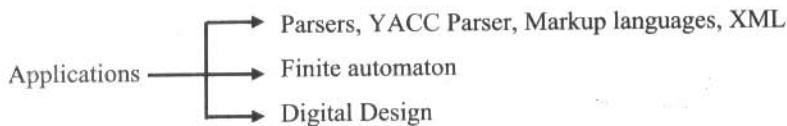
The same string $abcd$ can be derived from the by applying different set of productions as shown below:

$$\begin{array}{ll} S \Rightarrow C & \text{by using } S \rightarrow C \\ \Rightarrow aDd & \text{by using } C \rightarrow aDd \\ \Rightarrow abcd & \text{by using } D \rightarrow bc \end{array}$$

So, if we write the parse trees for both the derivations, the parse trees are different and naturally the grammar is ambiguous. It is not possible to obtain the unambiguous grammar for this and so it is inherently ambiguous grammar.

4.11. Applications of Context Free Grammars

There are numerous applications of the grammars and formal languages in the field of computer science. Let us concentrate on only few applications such as



Programming Languages (Parsers, YACC Parser, Markup languages, XML)

The grammar and formal languages are very suitable to express any programming language. Building a programming language like Pascal, C, C++, etc., is very expensive and time consuming and is not the scope of the book. Just to understand that grammars are very useful in designing a programming language, let us take an example of how to find the identifiers of a particular language.

I Example 60: Obtain the grammar to identify an identifier.

Note: An identifier can a variable name or a function name etc. If we define an identifier such that it should start with a letter and that letter can be followed by any combinations of letters or digits, the grammar to generate an identifier is

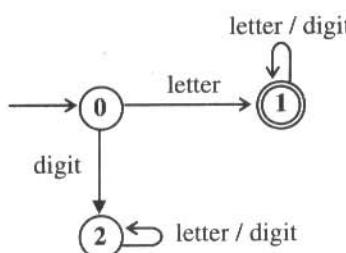
$$\begin{aligned}
 <\text{identifier}> &\rightarrow <\text{letter}> <\text{letter_digit}> \\
 <\text{letter_digit}> &\rightarrow <\text{letter}> <\text{letter_digit}> \mid <\text{digit}> <\text{letter_digit}> \mid \in \\
 <\text{letter}> &\rightarrow a \mid b \mid \dots \mid z \mid A \mid B \mid \dots \mid Z \\
 <\text{digit}> &\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9
 \end{aligned}$$

where $<\text{identifier}>$, $<\text{letter_digit}>$, $<\text{letter}>$, $<\text{digit}>$ are the variables or non-terminals and the symbols such as $a, b, \dots, z, A, B, \dots, Z, 0, 1, 2, \dots, 9$ are terminals.

Finite Automata

An automaton can be represented using a directed graph with vertices representing the states and the edges representing the transitions. The labels of the graph represent the input to be given from

one state to another state. The detailed discussion of how to construct a finite automaton is given in Chapters 1 and 2. The figure below shows a finite automaton to accept only a valid identifier:



The machine initially will be in the start state 0. If the input is a *letter* it is a valid identifier and enters into state 1. In this state if the input contains a string of any combinations of *letters* or *digits*, the string will be accepted by the machine and the machine stays in state 1 accepting all digits and letters. The two concentric circles in the finite automaton indicate the final state or an accepting state. In state 0, if the first input is a *digit*, the symbol is invalid and enters into the state 2 which is the rejecting state. Once the machine enters into the rejecting state, the input string should be rejected and so the machine stays in state 2 only.

Digital Design

An automaton will be very useful in digital design because many of the circuits are based on concepts from automata theory. Even though logical implementation of a circuit is through transistors, resistors, gates, flip-flops etc., an automaton serves a bridge between its implementation and functional description of a circuit.

YACC Parser-generator

The UNIX system provides a YACC command using which efficient parsers can be generated. The input to this command is a CFG but represented using different notations. In the notation used which is slightly different from that of notations used in CFG, each production is associated with an *action* which is the C code to be executed when the parse tree is created. For example, the grammar

$$\begin{aligned}
 E &\rightarrow E + E \mid E - E \\
 E &\rightarrow E * E \mid E / E \\
 E &\rightarrow (E) \mid I \\
 I &\rightarrow a \mid b \mid c
 \end{aligned}$$

using YACC notation can be written as shown below:

```

E : I           {...}
E '+' E        {...}
E '-' E        {...}
E '*' E        {...}
E '/' E        {...}
| E '^' E      {...}
| '(' E ')'   {...}
;
I: 'a'          {...}
| 'b'          {...}
| 'c'          {...}
;

```

Note the notations used in this representation when compared with notations used in CFG:

1. The symbol ‘ \rightarrow ’ is replaced by the symbol ‘:’
2. All the productions with a given head (i.e., the non-terminals towards left of \rightarrow) are grouped together and the body of the corresponding productions are separated by vertical bars and ends with terminator ‘;’
3. The terminals are enclosed within single quotes and the variables are not enclosed within single quotes

Markup Languages

The most familiar markup language is HTML ((Hyper Text Markup Language). HTML is used to create Hypertext documents for use on the World Wide Web. In HTML a block of text is surrounded with codes that indicate how it should appear on the computer screen. In HTML we can specify that a block of text, or a word, is linked to another file on the Internet. Hypertext Markup Language is the code used to write most documents on the World Wide Web. We can write the code using any text editor.

XML

XML stands for Extensible Markup Language. XML is a system for defining, validating, and sharing document formats. XML uses **tags** to distinguish document structures, and **attributes**. Instead of concentrating on formatting the text, XML is used to define the semantics.

YACC Parser-generator

The UNIX system provides a YACC command using which efficient parsers can be generated. The input to this command is a CFG but represented using different notations. In the notation used which is slightly different from that of notations used in CFG, each production is associated with an *action* which is the C code to be executed when the parse tree is created. For example, the grammar

$$\begin{array}{l} E \rightarrow E + E \mid E - E \\ E \rightarrow E * E \mid E / E \\ E \rightarrow (E) \mid I \\ I \rightarrow a \mid b \mid c \end{array}$$

using YACC notation can be written as shown below:

$$\begin{array}{ll} E : I & \{ \dots \} \\ | E '+' E & \{ \dots \} \\ | E '-' E & \{ \dots \} \\ | E '*' E & \{ \dots \} \\ | E '/' E & \{ \dots \} \\ | E '^' E & \{ \dots \} \\ | '(' E ')' & \{ \dots \} \\ ; & \\ I : 'a' & \{ \dots \} \\ | 'b' & \{ \dots \} \\ | 'c' & \{ \dots \} \end{array}$$

Note the notations used in this representation when compared with notations used in CFG:

1. The symbol ' \rightarrow ' is replaced by the symbol ':'
2. All the productions with a given head (i.e., the non-terminals towards left of \rightarrow is called a head) are grouped together and the body of the corresponding productions are separated by vertical bars and ends with terminator ':';
3. The terminals are enclosed within single quotes and the variables are not enclosed within single quotes

Markup Languages

The most familiar markup language is HTML ((Hyper Text Markup Language). HTML is used to create Hypertext documents for use on the World Wide Web. In HTML a block of text is surrounded with codes that indicate how it should appear on the computer screen. In HTML we can

specify that a block of text, or a word, is linked to another file on the Internet. Hypertext Markup Language is the code used to write most documents on the World Wide Web. We can write the code using any text editor.

XML

XML stands for Extensible Markup Language. XML is a system for defining, validating, and sharing document formats. XML uses **tags** to distinguish document structures, and **attributes**. Instead of concentrating on formatting the text, XML is used to define the semantics.

Exercises

1. Define the following terms with examples:
 - a. Alphabet
 - b. String
 - c. Concatenation of two strings
 - d. Star-closure
 - e. Positive-closure
 - f. Language
 - g. Reverse of a string
 - h. Length of a string
2. What is a grammar? Explain with example.
3. What is derivation? Explain with example.
4. Define the following terms with examples:
 - a. grammar
 - b. sentence of a grammar
 - c. sentential form
5. Obtain the grammar to generate integer.
6. Let $\Sigma = \{a, b\}$. Obtain a grammar G generating set of all palindromes over Σ .
7. Obtain a grammar to generate a language of all non-palindromes over $\{a, b\}$.
8. Obtain the grammar to generate the language:
$$L = \{ 0^m 1^n 2^n \mid m \geq 1 \text{ and } n \geq 0 \}$$

9. Obtain a grammar to generate the language $L = \{0^n 1^{n+1} \mid n \geq 0\}$.

10. Obtain the grammar to generate the languages:

- a. $L = \{w \mid n_a(w) = n_b(w)\}$
- b. $L = \{w \mid n_a(w) = n_b(w)+1\}$
- c. $L = \{w \mid n_a(w) > n_b(w)\}$
- d. $L = \{w \mid n_a(w) = 2n_b(w)\}$

11. What is the language generated by the grammar:

$$S \rightarrow 0A \mid \epsilon$$

$$A \rightarrow 1S$$

12. Obtain a CFG to generate a string of balanced parentheses.

13. Obtain a grammar to generate the language:

$$L = \{ww^R \mid w \in \{a,b\}^*\} \text{ where } w^R \text{ is reverse of } w$$

14. Obtain a grammar to generate the language $L = \{0^n 1^{2n} \mid n \geq 0\}$.

15. Obtain a grammar to generate the language $L = \{0^{n+2} 1^n \mid n \geq 1\}$.

16. Obtain a grammar to generate the language $L = \{0^i 1^j \mid i \neq j, i \geq 0 \text{ and } j \geq 0\}$.

17. Obtain a grammar to generate the language:

$$L = \{a^{n+2} b^m \mid n \geq 0 \text{ and } m > n\}$$

18. Obtain a grammar to generate the language:

$$L = \{a^n b^m \mid n \geq 0, m > n\}$$

19. Obtain a grammar to generate the language $L = \{a^n b^{n-3} \mid n \geq 3\}$.

20. Obtain a grammar to generate the language $L = L_1 L_2$

where

$$L_1 = \{a^n b^m \mid n \geq 0, m > n\}$$

$$L_2 = \{0^n 1^{2n} \mid n \geq 0\}$$

21. Obtain a grammar to generate the language $L = L_1 \cup L_2$

where

$$L_1 = \{a^n b^m \mid n \geq 0, m > n\}$$

$$L_2 = \{0^n 1^{2n} \mid n \geq 0\}$$

22. Obtain a grammar to generate the language $L = \{w : |w| \bmod 3 = 0\}$ on $\Sigma = \{a\}$.

23. Obtain a grammar to generate the language $L = \{w : |w| \bmod 3 = 0\}$ on $\Sigma = \{a, b\}$.
24. Obtain a grammar to generate the language $L = \{w : |w| \bmod 3 > 0\}$ on $\Sigma = \{a\}$.
25. Obtain a grammar to generate the language $L = \{w : |w| \bmod 3 \neq |w| \bmod 2\}$ on $\Sigma = \{a\}$.
26. Obtain a grammar to generate the language $L = \{w : |w| \bmod 3 \geq |w| \bmod 2\}$ on $\Sigma = \{a\}$.
27. Obtain a grammar to generate the set of all strings with exactly one a when $\Sigma = \{a, b\}$.
28. Obtain a grammar to generate the set of all strings with at least one a when $\Sigma = \{a, b\}$.
29. Obtain a grammar to generate the set of all strings with no more than three a 's when $\Sigma = \{a, b\}$.
30. Name some of the applications of formal languages and grammar.
31. Obtain a grammar to identify an identifier.
32. How an automaton can be used to add two binary numbers?
33. What is leftmost derivation? Explain with example.
34. What is rightmost derivation? Explain with example.
35. What is a derivation tree (or parse tree)? Explain with example.
36. What is the *yield* of a tree? Explain with example.
37. What is partial parse tree (or partial derivation tree)? Explain with example.
38. What is an ambiguous grammar?
39. Is the following grammar ambiguous?

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

40. Is the grammar ambiguous?

$$\begin{aligned} S &\rightarrow AB \mid aaB \\ A &\rightarrow A \mid Aa \\ B &\rightarrow b \end{aligned}$$

41. Show that the following grammar is ambiguous:

$$\begin{aligned} S &\rightarrow aSbS \\ S &\rightarrow bSaS \\ S &\rightarrow \epsilon \end{aligned}$$

42. Obtain the unambiguous grammar for the grammar shown below:

$$\begin{aligned} E &\rightarrow E + E \mid E - E \\ E &\rightarrow E * E \mid E / E \\ E &\rightarrow (E) \mid I \\ I &\rightarrow a \mid b \mid c \end{aligned}$$

and obtain the derivation for the expression $(a+b)^* (a-b)$.

43. What is inherently ambiguous grammar?

44. What is an S-Grammar (Simple Grammar)? Explain with example.

45. Find a Simple Grammar (S-Grammar) for the regular expression $aaa^*b + b$.

46. Find a Simple Grammar (S-Grammar) to generate the language $L = \{a^n b^n \mid n \geq 1\}$.

47. In what way BNF notations are different from the usual representation of the grammar? Give an example.

48. Use BNF notation and describe the while statement in C language. Assume that assignment statement and condition for while are defined already.

49. Give the BNF notation to write a C program. Provide 6 or 7 productions generally describing the main program. Assume the rest are defined.

50. What language is accepted by the following grammars?

- a. $S \rightarrow 0S0 \mid 1S1 \mid \epsilon$
 - b. $S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1$
 - c. $S \rightarrow 0S1 \mid 1S1 \mid \epsilon$
- A $\rightarrow 0B1 \mid 1B0$
- B $\rightarrow 0B0 \mid 1B1 \mid 0 \mid 1 \mid \epsilon$
- d. $S \rightarrow cS \mid cSdS \mid \epsilon$
 - e. $S \rightarrow cS \mid dS \mid c$
 - f. $S \rightarrow SS \mid dS \mid Sd \mid c$

51. Is the grammar $S \rightarrow SS \mid (S) \mid \epsilon$ ambiguous? (Ans : Yes) obtain two derivations by applying LMD or obtain two derivations by applying RMD.

52. What are the limitations of regular languages?

53. What is a context free grammar? Explain with example.

54. Let $G = (V, T, P, S)$ be a CFG where

$$V = \{ S \}$$

$$T = \{a, b\}$$

$$P = \{ \begin{array}{l} S \rightarrow aSa \mid bSb \mid \epsilon \\ \end{array} \}$$

S is the start symbol.

What is the language generated by this grammar?

55. Show that the language $L = \{ a^m b^n \mid m \neq n \}$ is context free.
56. Draw a CFG to generate a language consisting of equal number of a's and b's.
57. Draw a CFG on {a, b} to generate a language $L = \{ a^n w w^R b^n \mid w \in \Sigma^*, n \geq 1 \}$.
58. Obtain a context free grammar to generate properly nested parentheses structures involving three kinds of parentheses (), [] and { }.
59. Obtain a context free grammar to generate the following language:

$$L = \{w \mid w \in \{a, b\}^*, n_a(v) \geq n_b(v) \text{ where } v \text{ is any prefix of } w\}$$
60. Obtain a context free grammar to generate the following language:

$$L = \{01(1100)^n 110(10)^n \mid n \geq 0\}$$
61. Is the following language Context free?

$$L = \{a^n b^n \mid n \geq 0\}$$
62. Obtain a context free grammar to generate the following language:

$$L = \{a^n b^m \mid m > n \text{ and } n \geq 0\}$$
63. Obtain a CFG to generate unequal number of a's and b's.
64. For the regular expression $(011+1)^*(01)^*$ obtain the context free grammar.
65. What is leftmost derivation? Explain with example.
66. What is rightmost derivation? Explain with example.
67. What is a derivation tree (or parse tree)? Explain with example.
68. What is the yield of a tree? Explain with example.
69. What is partial parse tree (or partial derivation tree)? Explain with example.
70. What is an ambiguous grammar?

71. Consider the grammar shown below from which any arithmetic expression can be obtained:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow (E) \mid \text{id} \end{aligned}$$

Show that the grammar is ambiguous.

72. Is the following grammar ambiguous?

$$\begin{aligned} S &\rightarrow AS \mid X \\ X &\rightarrow aX \mid a \end{aligned}$$

73. Is the following grammar ambiguous?

$$\begin{aligned} S &\rightarrow ICtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

74. Is the grammar ambiguous?

$$\begin{aligned} S &\rightarrow AB \mid aaB \\ A &\rightarrow A \mid Aa \\ B &\rightarrow b \end{aligned}$$

75. Show that the following grammar is ambiguous:

$$\begin{aligned} S &\rightarrow aSbS \\ S &\rightarrow bSaS \\ S &\rightarrow \epsilon \end{aligned}$$

76. Obtain the unambiguous grammar for the grammar shown below:

$$\begin{aligned} E &\rightarrow E + E \mid E - E \\ E &\rightarrow E * E \mid E / E \\ E &\rightarrow (E) \mid I \\ I &\rightarrow a \mid b \mid c \end{aligned}$$

and obtain the derivation for the expression $(a + b)^* (a - b)$.

77. What is inherently ambiguous grammar?

78. What is an S-Grammar (Simple Grammar)? Explain with example.
79. Find a Simple Grammar (S-Grammar) for the regular expression $a^* b + b$.
80. Find a Simple Grammar (S-Grammar) to generate the language $L = \{a^n b^n \mid n \geq 1\}$.
81. In what way BNF notations are different from the usual representation of the grammar? Give an example.
82. Use BNF notation and describe the while statement in C language. Assume that assignment statement and condition for while are defined already.
83. Give the BNF notation to write a C program. Provide 6 or 7 productions generally describing the main program. Assume the rest are defined.
84. Obtain a derivation tree for the string $a + b * a + b$ from the grammar shown below:

$$\begin{array}{l} E \rightarrow E + E \\ E \rightarrow E - E \\ E \rightarrow E * E \\ E \rightarrow E / E \\ E \rightarrow a | b \end{array}$$

85. Consider the grammar:

$$\begin{array}{l} S \rightarrow 0 | 01S1 | 0A1 \\ A \rightarrow 1S | 0AA1 \end{array}$$

Is the grammar ambiguous? Ans: yes.

86. What language is accepted by the following grammars?

- a. $S \rightarrow 0S0 | 1S1 | \epsilon$
- b. $S \rightarrow 0S0 | 1S1 | 0 | 1$
- c. $S \rightarrow 0S1 | 1S1 | \epsilon$
 $A \rightarrow 0B1 | 1B0$
 $B \rightarrow 0B0 | 1B1 | 0 | 1 | \epsilon$
- d. $S \rightarrow cS | cSdS | \epsilon$
- e. $S \rightarrow cS | dS | c$
- f. $S \rightarrow SS | dS | Sd | c$

87. Is the grammar $S \rightarrow SS | (S) | \epsilon$ ambiguous? (Ans : Yes) obtain two derivations by applying LMD or obtain two derivations by applying RMD.

Chapter 5

Pushdown Automata

What are we studying in this chapter . . .

- ▶ *Definition of the Pushdown automata*
- ▶ *The languages of a PDA*
- ▶ *Equivalence of PDA's and CFG's*
- ▶ *Deterministic Pushdown Automata*

In the previous chapter we have seen how a context free language can be described using context free grammars. But, we have not encountered an automaton to recognize the context free languages as we had finite automata to recognize the regular languages. Now, the question is “Is there any need for pushdown automata when finite automata already exists?”

A DFA (or NFA) is not powerful enough to recognize many context-free language. Since the finite automaton have strict finite memories, it is not possible to construct those machines to accept the context free languages. The automaton to recognize the CFL may require additional amount of storage which will be used to store the data:

- For example, during parentheses matching, if we encounter ‘(’, it is required to store the left parentheses on the stack and as we encounter right parentheses i.e., ‘)’ we may have to pop the corresponding ‘(’ from the stack. Thus, stack is used to remember the scanned symbols. This is evident as we design the automaton to accept CFL.
- Sometimes, it is required to count the symbols also. For example, if we have the language $L = \{a^n b^n \mid n \geq 0\}$ after reading n number of a 's, we should see that n number of b 's exist. So, an automaton that

we construct to accept CFL's should be in a position to count at the same time should have sufficient memory to hold the string scanned.

Since the DFA's or NFA's can not count and can not store the input for future reference, we are forced to have a new machine called **Pushdown Automaton (PDA)**. Note that **PDA** is a Finite Automata with the exception that PDA has an extra stack. So, the definition of PDA is similar to the definition of NFA with slight changes. Let us take some typical examples and define the PDA.

Example 5.1: Consider the language $L = \{ wCw^R \mid w \in (a + b)^* \}$ where w^R is reverse of w and $C \in \Sigma$ indicates middle of string.

It is clear from the language $L = wCw^R$ that if w is abb then w^R is bba which is reverse of w . The language generated will be $abbCbba$ which is a palindrome. So, the language generates strings of palindromes.

Design: Let us device a method to accept a string of palindrome. In a given string, the letter C is the middle of the string. To start with, machine will be in the start state say q_0 . Now keep pushing all input symbols on to the stack and stay in state q_0 till the input symbol C is encountered.

Immediately after scanning the input symbol C , change the state to q_1 . Now, we have passed the middle of the string. If the given string is a palindrome, for each character encountered after the midpoint, there will be a corresponding character on the stack.

So, in state q_1 , after scanning the input symbol, compare it with most recently pushed symbol on the stack. If they are same, discard both the symbols and scan the next symbol and compare it with the symbol on the stack and repeat the process and remain in state q_1 . If there is a mismatch the machine halts and the string will be rejected. Once the last symbol in the input is matched with symbol on the stack, finally stack will be empty. Once the stack is empty, it is evident that the given string is a palindrome. So, change the state to q_2 which is an accepting state.

Note: It is clear from this example that, the PDA has set of states Q and set of input symbols Σ . Since it has stack also as a component, some symbols will be pushed on to the stack. So, the stack has stack alphabets denoted by Γ . The stack contains a special symbol Z_0 which is present in the stack initially (Note that this is the initial condition). It means that an empty stack contains Z_0 as the top of the stack.

5.1. Transitions

For the Example 5.1 shown above, the transitions can be written as shown in Table 5.1. Assume that q_0 is the start state and Z_0 is the initial symbol on the stack indicating stack is empty.

Since there is an ϵ -transition, the PDA is actually Non-deterministic. In a non-deterministic PDA, there will be ϵ -transitions or there may be multiple transitions from the same state on a given input when a specified symbol is there on the stack.

Table 5.1. Transitions

$\delta(q_0, a, Z_0)$	=	(q_0, aZ_0)
$\delta(q_0, b, Z_0)$	=	(q_0, bZ_0)
$\delta(q_0, a, a)$	=	(q_0, aa)
$\delta(q_0, b, a)$	=	(q_0, ba)
$\delta(q_0, a, b)$	=	(q_0, ab)
$\delta(q_0, b, b)$	=	(q_0, bb)
$\delta(q_0, c, Z_0)$	=	(q_1, Z_0)
$\delta(q_0, c, a)$	=	(q_1, a)
$\delta(q_0, c, b)$	=	(q_1, b)
$\delta(q_1, a, a)$	=	(q_1, ϵ)
$\delta(q_1, b, b)$	=	(q_1, ϵ)
$\delta(q_1, \epsilon, Z_0)$	=	(q_2, Z_0)

$$\delta: Q \times (\Sigma \cup \epsilon) \times \Gamma \text{ to } Q \times \Gamma^*$$

For example, consider the transitions

$$\begin{aligned}\delta(q_0, a, Z) &= (q_1, bZ) \\ \delta(q_0, a, Z) &= (q_2, cZ)\end{aligned}$$

These transitions can also be written as

$$\delta(q_0, a, Z) = \{ (q_1, bZ), (q_2, cZ) \}$$

In the above transition, when the PDA is currently in state q_0 , on scanning the input symbol a and when the top of the stack contains Z , the machine can perform one of the transitions defined, i.e., either

1. The PDA can go to state q_1 after pushing the symbol b on to the stack
or
2. The PDA can go to state q_2 after pushing the symbol c on to the stack

Unless otherwise specified, let us call **Push Down Automata (PDA)** as **Non Deterministic Push Down Automata (NPDA)**. Now, let us see “What is pushdown automata?”. The formal definition of PDA is shown below:

- ❖ **Definition:** A pushdown automata (PDA) is a seven tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

Q - is set of finite states.

Σ - Set of input alphabets.

Γ - Set of stack alphabets.

δ - transition from $Q \times (\Sigma \cup \epsilon) \times \Gamma$ to finite sub set of $Q \times \Gamma^*$

δ is called the transition function of M .

$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F \subseteq Q$ is set of final states.

The actions (i.e., transitions) performed by the PDA depends on

1. The current state.
2. The next input symbol.
3. The symbol on top of the stack.

The action performed by the machine consists of

1. Changing the states from one state to another
2. Replacing the symbol on the stack.

Note: In general, the transition function accepts three parameters namely a state, an input symbol and stack symbol and returns a new state after changing the top of the stack i.e. the transition function has the form:

$$\delta(state, input_symbol, stack_symbol) = (next_state, stack_symbol)$$

- **Example 5.2:** What does each of the following transitions represent?

- a. $\delta(p, a, Z) = (q, aZ)$
- b. $\delta(p, a, Z) = (q, \epsilon)$
- c. $\delta(p, a, Z) = (q, r)$
- d. $\delta(p, \epsilon, Z) = (q, r)$
- e. $\delta(p, \epsilon, \epsilon) = (q, Z)$
- f. $\delta(p, \epsilon, Z) = (q, \epsilon)$

The transition

$$\delta(p, a, Z) = (q, aZ)$$

means that the PDA in current state p after scanning the input symbol $a \in \Sigma$ and if $Z \in \Gamma$ is on top of the stack, then the PDA enters into new state q pushing $a \in \Sigma$ on to the stack. The transition

$$\delta(p, a, Z) = (q, \epsilon)$$

means that in state p , on scanning the input symbol a , and when top of this stack is Z , the machine enters into state q and the topmost symbol Z is deleted from the stack. The transition

$$\delta(p, a, Z) = (q, r)$$

means that in state p , on scanning the input symbol a and when top of the stack is Z , the machine enters into state q and topmost symbol Z on the stack is replaced by r . The transition

$$\delta(p, \epsilon, Z) = (q, r)$$

means that in state p , on scanning the empty string and when top of the stack is Z , the machine enters into q and topmost symbol Z on the stack is replaced by r . The transition

$$\delta(p, \epsilon, Z) = (q, \epsilon)$$

means that in state p , on scanning the empty string and when top of the stack empty, the machine enters into q pushing the symbol Z on the stack. The transition

$$\delta(p, \epsilon, Z) = (q, Z)$$

means that in state p , on scanning the empty string and when top of the stack is Z , the machine enters into q and topmost symbol Z on the stack is deleted.

Note: δ is a transition function with three arguments. The first two are same as NFA i.e., the state and either ϵ or a symbol from the input alphabet. The third argument is *the symbol on top of the stack*. As the input symbol is *consumed* when the transition(or function) is applied, the symbol on top of the stack may be deleted or it may be altered or some times one or more items may be pushed on to the stack (depends on the problem being solved).

The **move** of a machine can be defined as follows.

❖ **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. The move of machine M

$$\delta(p, a, Z) = (q, \alpha)$$

imply that, when the PDA is currently in state p , if the scanned input symbol is a and the top of the stack is Z , the PDA enters into new state q and pushes Z on the top of the stack. The symbol on top of the stack and the recently pushed symbol is denoted by α . Here $p, q \in Q, a \in \Sigma, Z \in \Gamma$ and $\alpha \in \Gamma^*$.

5.2. Graphical Representation of a PDA

The various actions performed by a PDA in state q on an input symbol a and when the top of the stack represented by Z can be represented using two methods:

1. Using δ notation (as discussed in previous section)
2. Using graphical representation

Now, let us discuss how a PDA is represented graphically using the notations that have been used to represent an FA in which:

- a) The states of the PDA correspond to the nodes in a graph and are represented using circles
- b) The start state of the PDA is denoted by an arrow labeled *start*
- c) The nodes of the graph represented by two concentric circles are the final states of the PDA
- d) If there is a transition of the form

$$\delta(p, a, Z) = (q, \alpha)$$

then, there will be an arc from state p to state q and the arc is labeled with

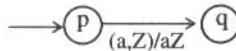
$$a, Z/\alpha$$

indicating a is the current symbol, Z is the symbol on top of the stack and α represent the top of the stack along with the recently pushed symbol.

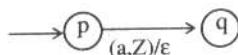
Example 5.3: For each of the transitions obtain the corresponding transition diagrams?

- a. $\delta(p, a, Z) = (q, aZ)$
- b. $\delta(p, a, Z) = (q, \epsilon)$
- c. $\delta(p, a, Z) = (q, r)$
- d. $\delta(p, \epsilon, Z) = (q, r)$
- e. $\delta(p, \epsilon, \epsilon) = (q, Z)$
- f. $\delta(p, \epsilon, Z) = (q, \epsilon)$

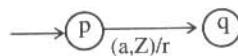
The transition $\delta(p, a, Z) = (q, aZ)$ means that the PDA in current state p after scanning the input symbol $a \in \Sigma$ and if $Z \in \Gamma$ is on top of the stack, then the PDA enters into new state q pushing $a \in \Gamma$ on to the stack. The graphical representation is shown below:



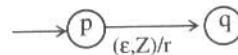
The transition $\delta(p, a, Z) = (q, \epsilon)$ means that in state p, on scanning the input symbol a , and when top of this stack is Z , the machine enters into state q and the topmost symbol Z is deleted from the stack. The corresponding graphical representation is



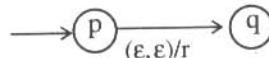
The transition $\delta(p, a, Z) = (q, r)$ means that in state p, on scanning the input symbol a and when top of the stack is Z , the machine enters into state q and topmost symbol Z on the stack is replaced by r . The corresponding graphical notation is shown below:



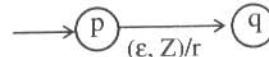
The transition $\delta(p, \epsilon, Z) = (q, r)$ means that in state p, on scanning the empty string and when top of the stack is Z , the machine enters into q and topmost symbol Z on the stack is replaced by r and the equivalent graphical representation is shown below:



The transition $\delta(p, \epsilon, \epsilon) = (q, Z)$ means that in state p, on scanning the empty string and when top of the stack empty, the machine enters into q pushing the symbol Z on the stack. The equivalent graphical representation is shown below:



The transition $\delta(p, \epsilon, Z) = (q, \epsilon)$ means that in state p, on scanning the empty string and when top of the stack is Z , the machine enters into q and top most symbol Z on the stack is deleted. The graphical representation for this is shown below:



5.3. Instantaneous Description

The current configuration of PDA at any given instant can be described by an *instantaneous description* (in short we can call ID). An ID gives the current state of the PDA, the remaining string to be processed and the entire contents of the stack. Thus, an instantaneous description ID can be defined as shown below.

❖ **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. An ID (instantaneous description) is defined as 3-tuple or a triple

$$(q, w, \alpha)$$

where q is the current state, w is the string to be processed and α is the current contents of stack. The leftmost symbol in the string α is on top of the stack and rightmost symbol in α is at the bottom of the stack.

Let the current configuration of PDA be

$$(q, aw, Z\alpha)$$

It means

- q - is the current state;
- aw - is the string to be processed;
- $Z\alpha$ - is the current contents of the stack with Z as the topmost symbol on the stack.

If the transition defined is $\delta(q, a, Z) = (p, \beta)$ then the new configuration obtained will be

$$(p, w, \beta\alpha)$$

The move from the current configuration to next configuration is given by

$$(q, aw, Z\alpha) \vdash (p, w, \beta\alpha)$$

This can be read as “the configuration $(q, aw, Z\alpha)$ derives $(p, w, \beta\alpha)$ in one move”. If an arbitrary number of moves are used to move from one configuration to another configuration then the moves are denoted by the symbol \vdash^* and \vdash^+ .

For example,

$$(q, aw, Z\alpha) \vdash^* (p, w, \beta\alpha)$$

means that the current configuration of the PDA will be $(q, aw, Z\alpha)$ and after applying zero or more number of transitions, the PDA enters into new configuration $(p, w, \beta\alpha)$.

Note: The instantaneous description $(q, aw, Z\alpha) \vdash^+ (p, w, \beta\alpha)$ indicates that the configuration $(p, w, \beta\alpha)$ is obtained from the configuration $(q, aw, Z\alpha)$ by applying one or more transitions.

5.4. Acceptance of a Language by PDA

There are two cases wherein a string w is accepted by a PDA:

- Get the final state from the start state.
- Get an empty stack from the start state.

In the first case, we say that the language is accepted by a **final state** and in the second case we say that the language is accepted by an **empty stack** or **null stack**. The formal definitions to accept the string by a final state and by an empty stack are defined as follows.

◆ **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. The language $L(M)$ accepted by a final state is defined as

$$L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \alpha)\}$$

for some $\alpha \in \Gamma^*$, $p \in F$ and $w \in \Sigma^*$. It means that the PDA, currently in state q_0 , after scanning the string w enters into a final state p . Once the machine is in state p , the input symbol should be ϵ and the contents of the stack are irrelevant. Any thing can be there on the stack. The only point to remember here is that when all the symbols in string w have been read and when the machine is in the final state the final contents of the stack are irrelevant.

We can also define a language $N(M)$ accepted by an empty stack (Null stack) as

$$N(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)\}$$

for $w \in \Sigma^*$, $q_0, p \in Q$. It means that when the string w is accepted by an empty stack, the final state is irrelevant, the input should be completely read and the stack should be empty. Here the state p is not the final state, only thing is that the string w should be completely read and stack should be empty.

5.5. Construction of PDA

Now, so far we have seen some concepts on PDA. In this section, we shall see how PDA's can be constructed.

■ **Example 5.4:** Obtain a PDA to accept the language $L(M) = \{wCw^R \mid w \in (a + b)^*\}$ where w^R is reverse of w by a final state.

It is clear from the language $L(M) = \{wCw^R\}$ that if

$$w = abb$$

then reverse of w denoted by w^R will be

$$w^R = bba$$

and the language L will be wCw^R i.e.,

$$abbCbba$$

which is a string of palindrome. So, we have to construct a PDA which accepts a palindrome consisting of a 's and b 's with the symbol C in the middle.

General Procedure: To check for the palindrome, let us push all scanned symbols onto the stack till we encounter the letter C. Once we pass the middle string, if the string is a palindrome, for each scanned input symbol, there should be a corresponding symbol (same as input symbol) on the stack. Finally, if there is no input and stack is empty, we say that the given string is a palindrome.

Step 1: Input symbols can be *a* or *b*.

Let q_0 be the initial state and Z_0 be the initial symbol on the stack. In state q_0 and when top of the stack is Z_0 , whether the input symbol is *a* or *b* push it on to the stack, and remain in q_0 . The transitions defined for this can be of the form:

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_0, aZ_0) \\ \delta(q_0, b, Z_0) &= (q_0, bZ_0)\end{aligned}$$

Once the first scanned input symbol is pushed on to the stack, the stack may contain either *a* or *b*. Now, in state q_0 , the input symbol can be either *a* or *b*. Note that irrespective of what is the input or what is there on the stack, we have to keep pushing all the symbols on to the stack, till we encounter C. So, the transitions defined for this can be of the form:

$$\begin{aligned}\delta(q_0, a, a) &= (q_0, aa) \\ \delta(q_0, b, a) &= (q_0, ba) \\ \delta(q_0, a, b) &= (q_0, ab) \\ \delta(q_0, b, b) &= (q_0, bb)\end{aligned}$$

Step 2: Input symbol is C.

Now, if the next input symbol is C, the top of the stack may be *a* or *b*. Another possibility is that, in state q_0 , the first symbol itself can be C. In this case *w* is null string and Z_0 will be on the stack. In all these cases, the input symbol is C i.e., the symbol which is present in the middle of the string. So, change the state to q_1 and do not alter the contents of the stack. The transitions defined for this can be of the form:

$$\begin{aligned}\delta(q_0, c, Z_0) &= (q_1, Z_0) \\ \delta(q_0, c, a) &= (q_1, a) \\ \delta(q_0, c, b) &= (q_1, b)\end{aligned}$$

Now, we have passed the middle of the string.

Step 3: Input symbols can be *a* or *b*.

To be a palindrome, for each input symbol there should be a corresponding symbol (same as input symbol) on the stack. So, whenever the input symbol is same as symbol on the stack, remain

in state q_1 and delete that symbol from the stack and repeat the process. The transitions defined for this can be of the form:

$$\delta(q_1, a, a) = (q_1, \epsilon)$$

$$\delta(q_1, b, b) = (q_1, \epsilon)$$

Step 4: Finally, in state q_1 , if the string is a palindrome, there is no input symbol to be scanned and the stack should be empty i.e., the stack should contain Z_0 . Now, change the state to q_2 and do not alter the contents of the stack. The transition for this can be of the form:

$$\delta(q_1, \epsilon, Z_0) = (q_2, Z_0)$$

So, the PDA M to accept the language:

$$L(M) = \{wCw^R \mid w \in (a,b)^*\}$$

along with transition graph is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

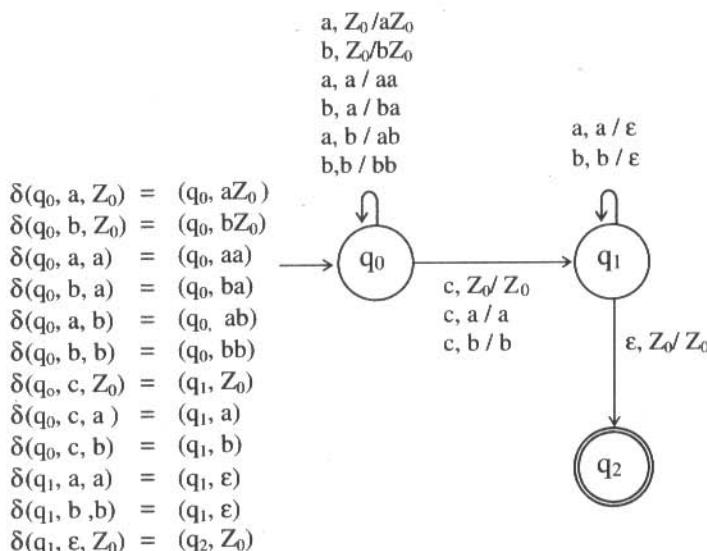
where

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

$$\Sigma = \{a, b, C\}$$

$$\Gamma = \{a, b, Z_0\}$$

δ : is shown below:



$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Sigma$ is the initial symbol on the stack.

$F = \{q_2\}$ is the final state.

To accept the string: The sequence of moves made by the PDA for the string aabCbba is shown below:

Initial ID

$(q_0, aabCbba, Z_0)$	\vdash	$(q_0, abCbba, aZ_0)$
	\vdash	$(q_0, bCbba, aaZ_0)$
	\vdash	$(q_0, Cbba, baaZ_0)$
	\vdash	$(q_1, baa, baaZ_0)$
	\vdash	(q_1, aa, aaZ_0)
	\vdash	(q_1, a, aZ_0)
	\vdash	(q_1, ϵ, Z_0)
	\vdash	(q_2, ϵ, Z_0)

(Final Configuration)

Since q_2 is the final state and input string is ϵ in the final configuration, the string

aabCbba

is accepted by the PDA.

To reject the string: The sequence of moves made by the PDA for the string aabCbab is shown below:

Initial ID

$(q_0, aabCbab, Z_0)$	\vdash	$(q_0, abCbab, aZ_0)$
	\vdash	$(q_0, bCbab, aaZ_0)$
	\vdash	$(q_0, Cbab, baaZ_0)$
	\vdash	$(q_1, bab, baaZ_0)$
	\vdash	(q_1, ab, aaZ_0)
	\vdash	(q_1, b, aZ_0)

(Final Configuration)

Since the transition $\delta(q_1, b, a)$ is not defined, the string

aabCbab

is not a palindrome and the machine halts and the string is rejected by the PDA.

Note: The same problem can be converted to accept the language by an empty stack. Only the change is, instead of the final transition namely

$$\delta(q_1, \epsilon, Z_0) = (q_2, Z_0)$$

replace it by the transition

$$\delta(q_1, \epsilon, Z_0) = (q_1, \epsilon)$$

When a language is accepted by an empty stack, finally the stack should not contain any thing including Z_0 . Note that q_1 is not a final state. There is no final state.

Example 5.5: Obtain a PDA to accept the language $L = \{a^n b^n \mid n \geq 1\}$ by a final state.

Note: The machine should accept n number of a 's followed by n number of b 's.

General Procedure: Since n number of a 's should be followed by n number of b 's, let us push all the symbols on to the stack as long as the scanned input symbol is a . Once we encounter b 's, we should see that for each b in the input, there should be corresponding a on the stack. When the input pointer reaches the end of the string, the stack should be empty. If stack is empty, it indicates that the string scanned has n number of a 's followed by n number of b 's.

Step 1: Let q_0 be the start state and Z_0 be the initial symbol on the stack. As long as the next input symbol to be scanned is a , irrespective of what is there on the stack, keep pushing all the symbols on to the stack and remain in q_0 . The transitions defined for this can be of the form :

$$\delta(q_0, a, Z_0) = (q_0, aZ_0)$$

$$\delta(q_0, a, a) = (q_0, aa)$$

Step 2: In state q_0 , if the next input symbol to be scanned is b and if the top of the stack is a , change the state to q_1 and delete one b from the stack. The transition for this can be of the form:

$$\delta(q_0, b, a) = (q_1, \epsilon)$$

Step 3: Once the machine is in state q_1 , the rest of the symbols to be scanned will be only b 's and for each b there should be corresponding symbol a on the stack. So, as the scanned input symbol

is b and if there is a matching a on the stack, remain in q_1 and delete the corresponding a from the stack. The transitions defined for this can be of the form:

$$\delta(q_1, b, a) = (q_1, \epsilon)$$

Step 4: In state q_1 , if the next input symbol to be scanned is ϵ and if the top of the stack is Z_0 (it means that for each b in the input there exists corresponding a on the stack) change the state to q_2 which is an accepting state. The transition defined for this can be of the form:

$$\delta(q_1, \epsilon, Z_0) = (q_2, Z_0)$$

So, the PDA to accept the language

$$L = \{a^n b^n \mid n \geq 1\}$$

along with transition diagram is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

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

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a, Z_0\}$$

δ : is shown below:

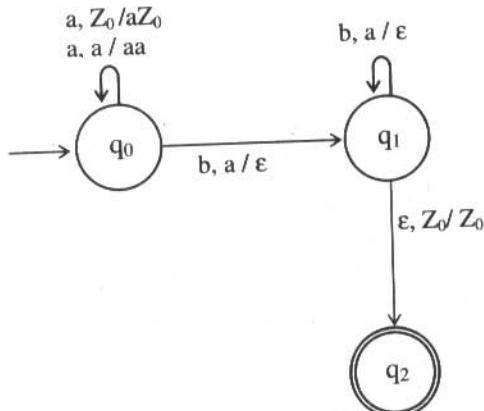
$$\delta(q_0, a, Z_0) = (q_0, aZ_0)$$

$$\delta(q_0, a, a) = (q_0, aa)$$

$$\delta(q_0, b, a) = (q_1, \epsilon)$$

$$\delta(q_1, b, a) = (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, Z_0) = (q_2, Z_0)$$



$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{q_2\}$ is the final state.

To accept the string: The sequence of moves made by the PDA for the string aaabbb is shown below:

Initial ID

$(q_0, aaabbb, Z_0)$	\vdash	$(q_0, aabbb, aZ_0)$
	\vdash	$(q_0, abbb, aaZ_0)$
	\vdash	$(q_0, bbb, aaaZ_0)$
	\vdash	(q_1, bb, aaZ_0)
	\vdash	(q_1, b, aZ_0)
	\vdash	(q_1, ϵ, Z_0)
	\vdash	(q_2, ϵ, Z_0)
		(Final Configuration)

Since q_2 is the final state and input string is ϵ in the final configuration, the string

aaabbb

is accepted by the PDA.

To reject the string: The sequence of moves made by the PDA for the string aabbb is shown below:

Initial ID

$(q_0, aabbb, Z_0)$	\vdash	$(q_0, abbb, aZ_0)$
	\vdash	(q_0, bbb, aaZ_0)
	\vdash	(q_1, bb, aZ_0)
	\vdash	(q_1, b, Z_0)
		(Final Configuration)

Since the transition $\delta(q_1, b, Z_0)$ is not defined, the string

aabbb

is rejected by the PDA.

Note: By changing the final transition from:



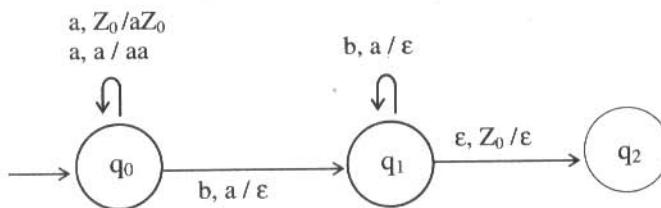
$$\delta(q_1, \epsilon, Z_0) = (q_2, Z_0)$$

to

$$\delta(q_1, \epsilon, Z_0) = (q_2, \epsilon)$$

the PDA accepted by an empty stack is obtained. Note that q_2 is not the final state.

The corresponding transition diagram accepting by an empty stack is shown below:



Example 5.6: Obtain a PDA to accept the language $L(M) = \{w \mid w \in (a+b)^* \text{ and } n_a(w) = n_b(w)\}$ by a final state i.e., number of a 's in string w should be equal to number of b 's in w .

The language accepted by the machine should consist strings of a 's and b 's of any length. Only restriction is that number of a 's in string w should be equal to number of b 's. The order of a 's and b 's is irrelevant. For example ϵ , ab , ba , $aaabbb$, $ababab$, $aabbab$ etc. are all the strings in the language $L(M)$.

General Procedure: The first scanned input symbol is pushed on to the stack. From this point onwards, if the scanned symbol is same as the symbol on to the stack, push the current input symbol on to the stack. If the input symbol is different from the symbol on the top of the stack, pop one symbol from the stack and repeat the process. Finally, when end of string is encountered, if the stack is empty, the string w has equal number of a 's and b 's, otherwise number of a 's and b 's are different.

Step 1: Let q_0 be the start state and Z_0 be the initial symbol on the stack. When the machine is in state q_0 and when top of the stack contains Z_0 , scan the input symbol (either a or b) and push it on to the stack. The transitions defined for this can be of the form:

$$\delta(q_0, a, Z_0) = (q_0, aZ_0)$$

$$\delta(q_0, b, Z_0) = (q_0, bZ_0)$$

Step 2: Once the first input symbol is pushed on to the stack, the top of stack may contain either a or b and the next input symbol to be scanned may be a or b . If the input symbol is same as the symbol on top of the stack, push the current input symbol on to the stack and remain in state q_0 only. Otherwise, pop an element from the stack. The transitions defined for this can be of the form:

$$\begin{aligned}
 \delta(q_0, a, a) &= (q_0, aa) \\
 \delta(q_0, b, b) &= (q_0, bb) \\
 \delta(q_0, a, b) &= (q_0, \epsilon) \\
 \delta(q_0, b, a) &= (q_0, \epsilon)
 \end{aligned}$$

Step 3: In state q_0 , if the next symbol to be scanned is ϵ (empty string) and top of the stack is Z_0 , it means that for each symbol a there exists a corresponding b and for each symbol b , there exists a symbol a . So, the string w consists of equal number of a 's and b 's and change the state to q_1 . The transition defined for this can be of the form:

$$\delta(q_0, \epsilon, Z_0) = (q_1, Z_0)$$

so, the PDA to accept the language

$$L = \{w \mid n_a(w) = n_b(w)\}$$

is given by

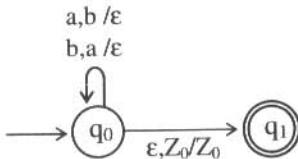
$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

$$\begin{aligned}
 Q &= \{q_0, q_1\} \\
 \Sigma &= \{a, b\} \\
 \Gamma &= \{a, b, Z_0\}
 \end{aligned}
 \quad
 \begin{aligned}
 a, Z_0/aZ_0 \\
 b, Z_0/bZ_0 \\
 a, a / aa \\
 b, b / bb \\
 a,b / \epsilon
 \end{aligned}$$

δ : is shown below:

$$\begin{aligned}
 \delta(q_0, a, Z_0) &= (q_0, aZ_0) \\
 \delta(q_0, b, Z_0) &= (q_0, bZ_0) \\
 \delta(q_0, a, a) &= (q_0, aa) \\
 \delta(q_0, b, b) &= (q_0, bb) \\
 \delta(q_0, a, b) &= (q_0, \epsilon) \\
 \delta(q_0, b, a) &= (q_0, \epsilon) \\
 \delta(q_0, \epsilon, Z_0) &= (q_1, Z_0)
 \end{aligned}$$



$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{q_1\}$ is the final state.

To accept the string: The sequence of moves made by the PDA for the string abbbbaa is shown below:

Initial ID

$$\begin{aligned}
 (q_0, abbbbaa, Z_0) &\xrightarrow{} (q_0, bbbbaa, aZ_0) \\
 &\xrightarrow{} (q_0, bbaa, Z_0) \\
 &\xrightarrow{} (q_0, baa, bZ_0) \\
 &\xrightarrow{} (q_0, aa, bbZ_0) \\
 &\xrightarrow{} (q_0, a, bZ_0) \\
 &\xrightarrow{} (q_0, \epsilon, Z_0) \\
 &\xrightarrow{} (q_1, \epsilon, Z_0)
 \end{aligned}$$

(Final Configuration)

Since q_1 is the final state and input string is ϵ in the final configuration, the string

abbbbaa

is accepted by the PDA.

To reject the string: The sequence of moves made by the PDA for the string aabbbb is shown below:

Initial ID

$$\begin{aligned}
 (q_0, aabbbb, Z_0) &\xrightarrow{} (q_0, abbb, aZ_0) \\
 &\xrightarrow{} (q_0, bbb, aaZ_0) \\
 &\xrightarrow{} (q_0, bb, aZ_0) \\
 &\xrightarrow{} (q_0, b, Z_0) \\
 &\xrightarrow{} (q_0, \epsilon, bZ_0)
 \end{aligned}$$

(Final Configuration)

Since the transition $\delta(q_0, \epsilon, b)$ is not defined, the string

aabbbb

is rejected by the PDA.

Note: To accept the language by an empty stack, the final state is irrelevant where as the next input symbol to be scanned should be ϵ and stack should be empty. Even Z_0 should not be there

on the stack. So, to obtain the PDA to accept equal number of *a*'s and *b*'s using an empty stack, change only the last transition in Example 5.5. The last transition

$$\delta(q_0, \epsilon, Z_0) = (q_1, Z_0)$$

can be changed as

$$\delta(q_0, \epsilon, Z_0) = (q_1, \epsilon)$$

So, the PDA to accept the language

$$L = \{w \mid n_a(w) = n_b(w)\}$$

by an empty stack is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \epsilon)$$

where

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

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a, b, Z_0\}$$

δ : is shown below:

$$\delta(q_0, a, Z_0) = (q_0, aZ_0)$$

$$\delta(q_0, b, Z_0) = (q_0, bZ_0)$$

$$\delta(q_0, a, a) = (q_0, aa)$$

$$\delta(q_0, b, b) = (q_0, bb)$$

$$\delta(q_0, a, b) = (q_0, \epsilon)$$

$$\delta(q_0, b, a) = (q_0, \epsilon)$$

$$\delta(q_0, \epsilon, Z_0) = (q_1, \epsilon)$$

$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{\phi\}$. Note that PDA is accepted by an empty stack.

The sequence of moves made by the PDA for the string aabbab by an empty stack is shown below:

Initial ID

$$\begin{aligned}
 (q_0, aabbab, Z_0) &\xrightarrow{} (q_0, abbab, aZ_0) \\
 &\xrightarrow{} (q_0, bbab, aaZ_0) \\
 &\xrightarrow{} (q_0, bab, aZ_0) \\
 &\xrightarrow{} (q_0, ab, Z_0) \\
 &\xrightarrow{} (q_0, b, aZ_0) \\
 &\xrightarrow{} (q_0, \epsilon, Z_0) \\
 &\xrightarrow{} (q_1, \epsilon, \epsilon)
 \end{aligned}$$

(Final Configuration)

Since the next input symbol to be scanned is ϵ and the stack is empty, the string aabbab is accepted by an empty stack.

Note: A PDA accepting by a final state and by an empty stack are equivalent. They can be obtained from each other i.e., if we know a PDA by a final state, we can obtain a PDA by an empty stack and vice versa.

Example 5.7: Obtain a PDA to accept a string of balanced parentheses. The parentheses to be considered are (,), [,].

Note 1: Some of the valid strings are:

[() () ([])], ϵ , [] [] []

and invalid strings are:

[) () [],) (, []

Note 2: ϵ (null string) is valid.

Note 3: Left parentheses can either be '(' or '[' and right parentheses can either be ')' or ']'.

Step 1: Let q_0 be the start state and Z_0 be the initial symbol on the stack. The state q_0 itself is the final state accepting ϵ (an empty string).

Step 2: In state q_0 , if the first scanned parentheses is '(' or '[', push the scanned symbol on to the stack and change the state to q_1 . The transition defined for this can be of the form:

$$\begin{aligned}
 \delta(q_0, (, Z_0) &= (q_1, (, Z_0) \\
 \delta(q_0, [, Z_0) &= (q_1, [, Z_0)
 \end{aligned}$$

Step 3: If at least one parentheses either ‘(’ or ‘[’ is present on the stack and if the scanned symbol is left parentheses, then push the left parentheses on to the stack. The transitions defined for this can be of the form:

$$\begin{aligned}\delta(q_1, (, ()) &= (q_1, (()) \\ \delta(q_1, (, [) &= (q_1, ([)) \\ \delta(q_1, [, ()) &= (q_1, [() \\ \delta(q_1, [, [) &= (q_1, [[))\end{aligned}$$

Step 4: If the scanned symbol is ‘)’ and if the top of the stack is ‘(’ pop an element from the stack. Similarly, if the scanned symbol is ‘]’ and if the top of the stack is ‘[s’ pop an element from the stack. The transitions defined for this can be of the form:

$$\begin{aligned}\delta(q_1,), () &= (q_1, \epsilon) \\ \delta(q_1,], [) &= (q_1, \epsilon)\end{aligned}$$

Step 5: When top of the stack is Z_0 , it indicates that so far all the parentheses have been matched. At this point, on ϵ -transition, the PDA enters into state q_0 and all the steps from step 1 are repeated. The transition for this can be of the form:

$$\delta(q_1, \epsilon, Z_0) = (q_0, Z_0)$$

So, the PDA to accept the language consisting of balanced parentheses is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

$$\begin{aligned}Q &= \{q_0, q_1\} \\ \Sigma &= \{(,), [,]\} \\ \Gamma &= \{(, [, Z_0\}\end{aligned}$$

δ : is shown below:

$$\begin{aligned}\delta(q_0, (, Z_0) &= (q_1, (Z_0)) \\ \delta(q_0, [, Z_0) &= (q_1, [Z_0)) \\ \delta(q_1, (, ()) &= (q_1, (()) \\ \delta(q_1, (, [) &= (q_1, ([))\end{aligned}$$

$$\begin{aligned}
 \delta(q_1, [, ()] &= (q_1, [()]) \\
 \delta(q_1, [, []] &= (q_1, [[]]) \\
 \delta(q_1,), () &= (q_1, \epsilon) \\
 \delta(q_1,], [] &= (q_1, \epsilon) \\
 \delta(q_1, \epsilon, Z_0) &= (q_0, Z_0)
 \end{aligned}$$

$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{q_0\}$. Note that even ϵ is accepted by PDA and is valid.

The sequence of moves made by the PDA for the string $[()()([])]$ is shown below:

Initial ID

$$\begin{aligned}
 (q_0, [([()()([])]], Z_0) &\vdash (q_1, ()()([])], [Z_0]) \\
 &\vdash (q_1,)()([])], ([Z_0]) \\
 &\vdash (q_1, ()([])], [Z_0]) \\
 &\vdash (q_1,)([])], ([Z_0]) \\
 &\vdash (q_1, ([])], [Z_0]) \\
 &\vdash (q_1, [])], ([Z_0]) \\
 &\vdash (q_1,])], [([Z_0]) \\
 &\vdash (q_1,)], ([Z_0]) \\
 &\vdash (q_1,), [Z_0]) \\
 &\vdash (q_1, \epsilon, Z_0) \\
 &\vdash (q_0, \epsilon, Z_0)
 \end{aligned}$$

(Final Configuration)

Since the next input symbol to be scanned is ϵ and the stack is empty, the string $[()()([])]$ is accepted by the PDA.

Example 5.8: Obtain a PDA to accept the language $L = \{w \mid w \in (a, b)^* \text{ and } n_a(w) > n_b(w)\}$.

Note: The solution is similar to that of the problem discussed in Example 5.5 in which we are accepting strings of a 's and b 's of equal numbers. When we encounter end of the input i.e., ϵ and top of the stack is Z_0 , it has equal number of a 's and b 's. But, what we want is a machine to

accept more number of a 's than b 's. For this, only change to be made is that when we encounter ϵ (i.e., end of the input), if top of the stack contains at least one a , then change the final state to q_1 and do not alter the contents of the stack. The transition defined remains same as problem shown in Example 5.5, except the last transition. The last transition is of the form:

$$\delta(q_0, \epsilon, a) = (q_1, a)$$

So, the PDA to accept the language

$$L = \{w \mid n_a(w) > n_b(w)\}$$

is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

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

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a, b, Z_0\}$$

δ : is shown below:

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_0, aZ_0) \\ \delta(q_0, b, Z_0) &= (q_0, bZ_0) \\ \delta(q_0, a, a) &= (q_0, aa) \\ \delta(q_0, b, b) &= (q_0, bb) \\ \delta(q_0, a, b) &= (q_0, \epsilon) \\ \delta(q_0, b, a) &= (q_0, \epsilon) \\ \delta(q_0, \epsilon, a) &= (q_1, a)\end{aligned}$$

$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{q_1\}$ is the final state.

Note: On similar lines we can find a PDA to accept the language

$$L = \{w \mid w \in (a,b)^* \text{ and } n_a(w) < n_b(w)\}$$

i.e., strings of a 's and b 's where number of b 's are more than number of a 's. To achieve this only change to be made in the above machine is change the final transition

$$\delta(q_0, \epsilon, a) = (q_1, a)$$

to

$$\delta(q_0, \epsilon, b) = (q_1, b)$$

So, the PDA to accept the language

$$L = \{w \mid w \in (a,b)^* \text{ and } n_a(w) < n_b(w)\}$$

is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

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

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a, b, Z_0\}$$

δ : is shown below:

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_0, aZ_0) \\ \delta(q_0, b, Z_0) &= (q_0, bZ_0) \\ \delta(q_0, a, a) &= (q_0, aa) \\ \delta(q_0, b, b) &= (q_0, bb) \\ \delta(q_0, a, b) &= (q_0, \epsilon) \\ \delta(q_0, b, a) &= (q_0, \epsilon) \\ \delta(q_0, \epsilon, b) &= (q_1, b)\end{aligned}$$

$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{q_1\}$ is the final state.

■ **Example 5.9:** Obtain a PDA to accept the language $L = \{a^n b^{2n} \mid n \geq 1\}$.

Note: The machine should accept n number of a 's followed by $2n$ number of b 's.

General Procedure Since n number of a 's should be followed by $2n$ number of b 's, for each a in the input, push two a 's on to the stack. Once we encounter b 's, we should see that for each b in the input, there should be corresponding a on the stack. When the input pointer reaches the end

of the string, the stack should be empty. If stack is empty, it indicates that the string scanned has n number of a 's followed by $2n$ number of b 's.

Step 1: Let q_0 be the start state and Z_0 be the initial symbol on the stack. For each scanned input symbol a , push two a 's on to the stack. The transitions defined for this can be of the form:

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_0, aaZ_0) \\ \delta(q_0, a, a) &= (q_0, aaa)\end{aligned}$$

Step 2: In state q_0 , if the next input symbol to be scanned is b and if the top of the stack is a , change the state to q_1 and delete one b from the stack. The transition for this can be of the form:

$$\delta(q_0, b, a) = (q_1, \epsilon)$$

Step 3: Once the machine is in state q_1 , the rest of the symbols to be scanned will be only b 's and for each b there should be corresponding symbol a on the stack. So, as the scanned input symbol is b and if there is a matching a on the stack, remain in q_1 and delete the corresponding a from the stack. The transitions defined for this can be of the form:

$$\delta(q_1, b, a) = (q_1, \epsilon)$$

Step 4: In state q_1 , if the next input symbol to be scanned is ϵ and if the top of the stack is Z_0 , (it means that for each b in the input there exists corresponding a on the stack) change the state to q_2 which is an accepting state. The transition defined for this can be of the form:

$$\delta(q_1, \epsilon, Z_0) = (q_2, \epsilon)$$

So, the PDA to accept the language

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

is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

$$\begin{aligned}Q &= \{q_0, q_1, q_2\} \\ \Sigma &= \{a, b\} \\ \Gamma &= \{a, Z_0\}\end{aligned}$$

δ : is shown below:

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_0, aaZ_0) \\ \delta(q_0, a, a) &= (q_0, aaa)\end{aligned}$$

$$\delta(q_0, b, a) = (q_1, \epsilon)$$

$$\delta(q_1, b, a) = (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, Z_0) = (q_2, \epsilon)$$

$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{q_2\}$ is the final state.

To accept the string: The sequence of moves made by the PDA for the string aabbba is shown below:

Initial ID

$$\begin{aligned}
 (q_0, aabbba, Z_0) &\vdash (q_0, abbb, aaZ_0) \\
 &\vdash (q_0, bbbb, aaaaZ_0) \\
 &\vdash (q_0, bbb, aaaZ_0) \\
 &\vdash (q_1, bb, aaZ_0) \\
 &\vdash (q_1, b, aZ_0) \\
 &\vdash (q_1, \epsilon, Z_0) \\
 &\vdash (q_2, \epsilon, Z_0) \\
 &\quad (\text{Final Configuration})
 \end{aligned}$$

Since q_2 is the final state and input string is ϵ in the final configuration, the string

aabbba

is accepted by the PDA.

To reject the string: The sequence of moves made by the PDA for the string aabba is shown below:

Initial ID

$$\begin{aligned}
 (q_0, aabba, Z_0) &\vdash (q_0, abbb, aaZ_0) \\
 &\vdash (q_0, bbb, aaaaZ_0) \\
 &\vdash (q_0, bb, aaaZ_0) \\
 &\vdash (q_0, b, aaZ_0) \\
 &\vdash (q_0, \epsilon, aZ_0) \\
 &\quad (\text{Final Configuration})
 \end{aligned}$$

Since the transition $\delta(q_0, \epsilon, a)$ is not defined, the string

aabbb

is rejected by the PDA.

Example 5.10: Obtain a PDA to accept the language $L = \{ww^R \mid w \in (a + b)^*\}$ by a final state.

It is clear from the language $L(M) = \{ww^R\}$ that if

$w = abb$

then reverse of w denoted by w^R will be

$w^R = bba$

and the language L will be ww^R , i.e.,

abbbbba

which is a string of palindrome.

So, we have to construct a PDA which accepts a palindrome consisting of a 's and b 's. This problem is similar to the problem discussed in Example 5.3. Only difference is that in Example 5.3, an extra symbol C acts as a pointer to the middle string. But, here there is no way to find the mid point for the string.

General Procedure: To check for the palindrome, let us push all scanned symbols onto the stack till we encounter the mid point (Remember that there is no way to find the midpoint). Once we pass the middle string, to be a palindrome, for each scanned input symbol, there should be a corresponding symbol (same as input symbol) on the stack. Finally, if there is no input and stack is empty, we say that the given string is a palindrome.

Step 1: Let q_0 be the initial state and Z_0 be the initial symbol on the stack. In state q_0 and when top of the stack is Z_0 , whether the input symbol is a or b push it on to the stack, and remain in q_0 . The transitions defined for this can be of the form:

$$\delta(q_0, a, Z_0) = (q_0, aZ_0)$$

$$\delta(q_0, b, Z_0) = (q_0, bZ_0)$$

Note: In state q_0 , if we encounter ϵ , the empty string has to be accepted and we should be in a position to reach the final state and so, we may have one more transition of the form:

$$\delta(q_0, \epsilon, Z_0) = (q_1, Z_0)$$

Once the first scanned input symbol is pushed on to the stack, the stack may contain either a or b . Now, in state q_0 , the input symbol can be either a or b . Note that irrespective of what is the input or what is there on the stack, we have to keep pushing all the symbols on to the stack, till we encounter midpoint (But, there is no way to find mid point. We continue this process till we encounter mid point through our common sense).

So, the transitions defined for this can be of the form:

$$\begin{aligned}\delta(q_0, a, a) &= (q_0, aa) \\ \delta(q_0, b, a) &= (q_0, ba) \\ \delta(q_0, a, b) &= (q_0, ab) \\ \delta(q_0, b, b) &= (q_0, bb)\end{aligned}$$

Step 2: Now, once we reach the midpoint, the top of the stack may be a or b . To be a palindrome, for each input symbol there should be a corresponding symbol (same as input symbol) on the stack. So, whenever the input symbol is same as symbol on the stack, change the state to q_1 and delete that symbol from the stack. The transitions defined for this can be of the form:

$$\begin{aligned}\delta(q_0, a, a) &= (q_1, \epsilon) \\ \delta(q_0, b, b) &= (q_1, \epsilon)\end{aligned}$$

Step 3: Now, once we are in state q_1 , it means that we have passed the mid point. Now, the top of the stack may be a or b . To be a palindrome, for each input symbol there should be a corresponding symbol (same as input symbol) on the stack. So, whenever the input symbol is same as symbol on the stack, remain in state q_1 and delete that symbol from the stack. The transitions defined for this can be of the form:

$$\begin{aligned}\delta(q_1, a, a) &= (q_1, \epsilon) \\ \delta(q_1, b, b) &= (q_1, \epsilon)\end{aligned}$$

Step 4: Finally, in state q_1 , if the string is a palindrome, there is no input symbol to be scanned and the stack should be empty i.e., the stack should contain Z_0 . Now, change the state to q_2 and do not alter the contents of the stack. The transition for this can be of the form:

$$\delta(q_1, \epsilon, Z_0) = (q_2, Z_0)$$

So, the PDA M to accept the language

$$L(M) = \{ww^R \mid w \in (a,b)^*\}$$

is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

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

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a, b, Z_0\}$$

δ : is shown below:

$$\begin{array}{ll}
 \delta(q_0, \epsilon, Z_0) &= (q_1, Z_0) \\
 \delta(q_0, a, Z_0) &= (q_0, aZ_0) \\
 3 \quad \delta(q_0, b, Z_0) &= (q_0, bZ_0) \\
 \delta(q_0, a, a) &= (q_0, aa) \\
 \delta(q_0, b, a) &= (q_0, ba) \\
 6 \quad \delta(q_0, a, b) &= (q_0, ab) \\
 7 \quad \delta(q_0, b, b) &= (q_0, bb) \\
 8 \quad \delta(q_0, a, a) &= (q_1, \epsilon) \\
 \delta(q_0, b, b) &= (q_1, \epsilon) \\
 \delta(q_1, a, a) &= (q_1, \epsilon) \\
 \delta(q_1, b, b) &= (q_1, \epsilon) \\
 \delta(q_1, \epsilon, Z_0) &= (q_2, Z_0)
 \end{array}$$

$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{q_2\}$ is the final state.

Note that the transitions numbered 3 and 7, 6 and 8 can be combined and the transitions can be written as shown below also.

$$\begin{array}{ll}
 \delta(q_0, \epsilon, Z_0) &= (q_1, Z_0) \\
 \delta(q_0, a, Z_0) &= (q_0, aZ_0) \\
 \delta(q_0, b, Z_0) &= (q_0, bZ_0) \\
 \delta(q_0, a, a) &= \{(q_0, aa), (q_1, \epsilon)\} \\
 \delta(q_0, b, a) &= (q_0, ba)
 \end{array}$$

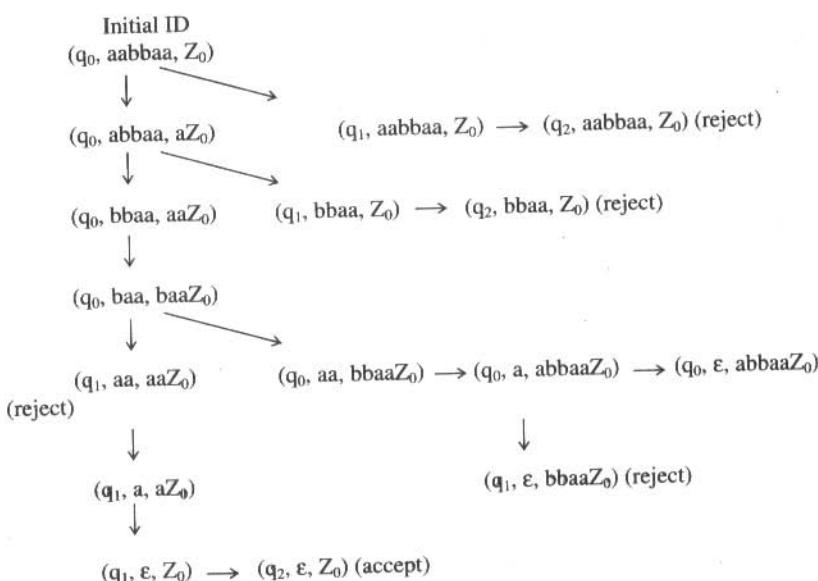
$$\begin{aligned}
 \delta(q_0, a, b) &= (q_0, ab) \\
 \delta(q_0, b, b) &= \{ (q_0, bb), (q_1, \epsilon) \} \\
 \delta(q_1, a, a) &= (q_1, \epsilon) \\
 \delta(q_1, b, b) &= (q_1, \epsilon) \\
 \delta(q_1, \epsilon, Z_0) &= (q_2, Z_0)
 \end{aligned}$$

Note that once the following transitions are applied

$$\begin{aligned}
 \delta(q_0, a, a) &= \{ (q_0, aa), (q_1, \epsilon) \} \\
 \delta(q_0, b, b) &= \{ (q_0, bb), (q_1, \epsilon) \}
 \end{aligned}$$

If the input symbol is same as the symbol on top of the stack, the machine may push the current symbol on to the stack, or it may pop an element from the stack. At this point, the machine makes appropriate decision so that if the string is a palindrome, it has to accept. This machine is clearly a non-deterministic PDA (in short we call NPDA).

To accept the string: The sequence of moves made by the PDA for the string *aabbbaa* is shown below:



Since q_2 is the final state and input string is ϵ in the final configuration, the string *aabbbaa* is accepted by the PDA.

5.6. Deterministic and Non-deterministic PDA

In Example 5.10, we have seen that there can be multiple transitions defined from a state. The PDA defined in Example 5.10, is clearly a non-deterministic PDA. Now, let us see the difference between *deterministic* PDA and *non-deterministic* PDA.

❖ **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. The PDA is deterministic if

1. $\delta(q, a, Z)$ has only one element;
2. If $\delta(q, \epsilon, Z)$ is not empty, then $\delta(q, a, Z)$ should be empty.

Both the conditions should be satisfied for the PDA to be deterministic. If one of the conditions fails, the PDA is non-deterministic.

In the PDAs discussed in Examples 5.3 to 5.10, let us see what are deterministic PDAs and what are non-deterministic PDAs.

■ **Example 5.11:** Is the PDA to accept the language $L(M) = \{ wCw^R \mid w \in (a + b)^*\}$ discussed in Example 5.4 is deterministic?

The transitions defined for this machine are obtained in Example 5.4 and are reproduced for the sake of convenience.

$$\begin{aligned}
 \delta(q_0, a, Z_0) &= (q_0, aZ_0) \\
 \delta(q_0, b, Z_0) &= (q_0, bZ_0) \\
 \delta(q_0, a, a) &= (q_0, aa) \\
 \delta(q_0, b, a) &= (q_0, ba) \\
 \delta(q_0, a, b) &= (q_0, ab) \\
 \delta(q_0, b, b) &= (q_0, bb) \\
 \delta(q_0, c, Z_0) &= (q_1, Z_0) \\
 \delta(q_0, c, a) &= (q_1, a) \\
 \delta(q_0, c, b) &= (q_1, b) \\
 \delta(q_1, a, a) &= (q_1, \epsilon) \\
 \delta(q_1, b, b) &= (q_1, \epsilon) \\
 \delta(q_1, \epsilon, Z_0) &= (q_2, Z_0)
 \end{aligned}$$

The PDA should satisfy the two conditions shown in the definition to be deterministic:

1. $\delta(q, a, Z)$ has only one element: Note that in the transitions, for each $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$, there is only one component defined and the first condition is satisfied.
2. The second condition states that if $\delta(q, \epsilon, Z)$ is not empty, then $\delta(q, a, Z)$ should be empty, i.e., If there is an ϵ -transition, (in this case it is $\delta(q_1, \epsilon, Z_0)$), then there should not be any transition from the state q_1 when top of the stack is Z_0 which is true.

Since, the PDA satisfies both the conditions, the PDA is deterministic.

Example 5.12: Is the PDA corresponding to the language $L = \{a^n b^n \mid n \geq 1\}$ by a final state is deterministic?

The transitions defined for this machine discussed in Example 5.5 are reproduced here for the sake of convenience.

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_0, aZ_0) \\ \delta(q_0, a, a) &= (q_0, aa) \\ \delta(q_0, b, a) &= (q_1, \epsilon) \\ \delta(q_1, b, a) &= (q_1, \epsilon) \\ \delta(q_1, \epsilon, Z_0) &= (q_2, \epsilon)\end{aligned}$$

The first condition to be deterministic is $\delta(q, a, Z)$ should have only one component. In this case, for each $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$, there exists only one definition. So, the first condition is satisfied. To satisfy the second condition, consider the transition

$$\delta(q_1, \epsilon, Z_0) = (q_2, \epsilon)$$

Since the transition is defined, the transition $\delta(q_1, a, Z_0)$ where $a \in \Sigma$ should not be defined which is true. Since both the conditions are satisfied, the given PDA is deterministic.

Example 5.13: Is the PDA to accept the language $L(M) = \{w \mid w \in (a+b)^*\text{ and }n_a(w) = n_b(w)\}$ is deterministic?

The transitions defined for this machine discussed in Example 5.6 are reproduced here for the sake of convenience.

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_0, aZ_0) \\ \delta(q_0, b, Z_0) &= (q_0, bZ_0)\end{aligned}$$

$$\begin{aligned}
 \delta(q_0, a, a) &= (q_0, aa) \\
 \delta(q_0, b, b) &= (q_0, bb) \\
 \delta(q_0, a, b) &= (q_0, \epsilon) \\
 \delta(q_0, b, a) &= (q_0, \epsilon) \\
 \delta(q_0, \epsilon, Z_0) &= (q_1, Z_0)
 \end{aligned}$$

The first condition to be deterministic is $\delta(q, a, Z)$ should have only one component. In this case, for each $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$, there exists only one component. So, the first condition is satisfied. To satisfy the second condition, consider the transition

$$\delta(q_0, \epsilon, Z_0) = (q_1, Z_0)$$

Since this transition is defined, the transition $\delta(q_0, a, Z_0)$ where $a \in \Sigma$ should not be defined. But, there are two transitions

$$\begin{aligned}
 \delta(q_0, a, Z_0) &= (q_0, aZ_0) \\
 \delta(q_0, b, Z_0) &= (q_0, bZ_0)
 \end{aligned}$$

defined from q_0 when top of the stack is Z_0 . Since the second condition is not satisfied, the given PDA is *non-deterministic* PDA.

Example 5.14: Is the PDA to accept the language consisting of balanced parentheses is deterministic?

The transitions defined for this machine discussed in Example 5.7 are reproduced here for the sake of convenience.

$$\begin{aligned}
 \delta(q_0, (, Z_0) &= (q_1, (Z_0) \\
 \delta(q_0, [, Z_0) &= (q_1, [Z_0) \\
 \delta(q_1, (, ()) &= (q_1, (()) \\
 \delta(q_1, (, [) &= (q_1, ([)) \\
 \delta(q_1, [, ()) &= (q_1, [()) \\
 \delta(q_1, [, []) &= (q_1, [[]) \\
 \delta(q_1,), () &= (q_1, \epsilon) \\
 \delta(q_1,], [) &= (q_1, \epsilon) \\
 \delta(q_1, \epsilon, Z_0) &= (q_0, Z_0)
 \end{aligned}$$

The first condition to be deterministic is $\delta(q, a, Z)$ should have only one component. In this case, for each $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$, there exists only one component. So, the first condition is satisfied. To satisfy the second condition, consider the transition

$$\delta(q_1, \epsilon, Z_0) = q_0, Z_0$$

Since this transition is defined, the transition $\delta(q_1, a, Z_0)$ where $a \in \Sigma$ should not be defined which is true. Since both the conditions are satisfied, the given PDA is deterministic.

Example 5.15: Is the PDA to accept the language $L = \{w \mid w \in (a, b)^* \text{ and } n_a(w) > n_b(w)\}$ is deterministic?

The transitions defined for this machine discussed in Example 5.8 are reproduced here for the sake of convenience.

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_0, aZ_0) \\ \delta(q_0, b, Z_0) &= (q_0, bZ_0) \\ \delta(q_0, a, a) &= (q_0, aa) \\ \delta(q_0, b, b) &= (q_0, bb) \\ \delta(q_0, a, b) &= (q_0, \epsilon) \\ \delta(q_0, b, a) &= (q_0, \epsilon) \\ \delta(q_0, \epsilon, a) &= (q_1, a)\end{aligned}$$

The first condition to be deterministic is $\delta(q, a, Z)$ should have only one component. In this case, for each $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$, there exists only one component. So, the first condition is satisfied. To satisfy the second condition, consider the transition

$$\delta(q_0, \epsilon, a) = (q_1, a)$$

Since this transition is defined, the transition $\delta(q_0, f, a)$ where $f \in \Sigma$ should not be defined. But, there are two transitions

$$\begin{aligned}\delta(q_0, a, a) &= (q_0, aa) \\ \delta(q_0, b, a) &= (q_0, \epsilon)\end{aligned}$$

defined from q_0 when top of the stack is a . Since the second condition is not satisfied, the given PDA is *non-deterministic* PDA.

Example 5.16: Is the PDA to accept the language $L = \{a^n b^{2n} \mid n \geq 1\}$ is deterministic?

The transitions defined for this machine discussed in Example 5.9 are reproduced here for the sake of convenience.

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_0, aaZ_0) \\ \delta(q_0, a, a) &= (q_0, aaa) \\ \delta(q_0, b, a) &= (q_1, \epsilon) \\ \delta(q_1, b, a) &= (q_1, \epsilon) \\ \delta(q_1, \epsilon, Z_0) &= (q_2, \epsilon)\end{aligned}$$

The first condition to be deterministic is $\delta(q, a, Z)$ should have only one component. In this case, for each $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$, there exists only one definition. So, the first condition is satisfied. To satisfy the second condition, consider the transition

$$\delta(q_1, \epsilon, Z_0) = (q_2, \epsilon)$$

Since the transition is defined, the transition $\delta(q_1, a, Z_0)$ where $a \in \Sigma$ should not be defined which is true. Since both the conditions are satisfied, the given PDA is deterministic.

Example 5.17: Is the PDA to accept the language $L = \{ww^R \mid w \in (a+b)^*\}$ is deterministic?

The transitions defined for this machine discussed in Example 5.10 are reproduced here for the sake of convenience.

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_0, aZ_0) \\ \delta(q_0, b, Z_0) &= (q_0, bZ_0) \\ \delta(q_0, a, a) &= (q_0, aa), (q_1, \epsilon) \\ \delta(q_0, b, a) &= (q_0, ba) \\ \delta(q_0, a, b) &= (q_0, ab) \\ \delta(q_0, b, b) &= (q_0, bb), (q_1, \epsilon) \\ \delta(q_1, a, a) &= (q_1, \epsilon) \\ \delta(q_1, b, b) &= (q_1, \epsilon) \\ \delta(q_1, \epsilon, Z_0) &= (q_2, Z_0)\end{aligned}$$

The first condition to be deterministic is $\delta(q, a, Z)$ should have only one component. But, there are two transitions each having two components viz.,

$$\begin{aligned}\delta(q_0, a, a) &= (q_0, aa), (q_1, \epsilon) \\ \delta(q_0, b, b) &= (q_0, bb), (q_1, \epsilon)\end{aligned}$$

So, the first condition fails. To be deterministic, both the conditions shown in the definition should be satisfied. Since one of the conditions is not met, the PDA is *non-deterministic*.

5.7. CFG to PDA

It is quite easy to get a PDA from the context free grammar. This is possible only from a CFG which is in GNF. So, given any grammar, first obtain the grammar in GNF and then obtain the PDA. The steps to be followed to convert a grammar to its equivalent PDA are shown below:

1. Convert the grammar into GNF
2. Let q_0 be the start state and Z_0 is the initial symbol on the stack. Without consuming any input, push the start symbol S onto the stack and change the state to q_1 . The transition for this can be

$$\delta(q_0, \epsilon, Z_0) = (q_1, SZ_0)$$

3. For each production of the form

$$A \rightarrow a\alpha$$

introduce the transition

$$\delta(q_1, a, A) = (q_1, \alpha)$$

4. Finally, in state q_1 , without consuming any input, change the state to q_f which is an accepting state. The transition for this can be of the form

$$\delta(q_1, \epsilon, Z_0) = (q_f, Z_0)$$

■ Example 5.18: For the below grammar

$$\begin{array}{lcl} S & \rightarrow & aABC \\ A & \rightarrow & aB|a \\ B & \rightarrow & bA|b \\ C & \rightarrow & a \end{array}$$

obtain the corresponding PDA.

Let q_0 be the start state and Z_0 the initial symbol on the stack.

Step 1: Push the start symbol S on to the stack and change the state to q_1 . The transition for this can be of the form

$$\delta(q_0, \epsilon, Z_0) = (q_1, SZ_0)$$

Step 2: For each production $A \rightarrow a\alpha$ introduce the transition

$$\delta(q_1, a, A) = (q_1, \alpha)$$

This can be done as shown below:

Production		Transition	
S	\rightarrow	$\delta(q_1, a, S)$	$= (q_1, ABC)$
A	\rightarrow	$\delta(q_1, a, A)$	$= (q_1, B)$
A	\rightarrow	$\delta(q_1, a, A)$	$= (q_1, \epsilon)$
B	\rightarrow	$\delta(q_1, b, B)$	$= (q_1, A)$
B	\rightarrow	$\delta(q_1, b, B)$	$= (q_1, \epsilon)$
C	\rightarrow	$\delta(q_1, a, C)$	$= (q_1, \epsilon)$

Step 3: Finally in state q_1 , without consuming any input change the state to q_f which is an accepting state, i.e.,

$$\delta(q_1, \epsilon, Z_0) = (q_f, Z_0)$$

So, the PDA M is given by $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where

$$Q = \{q_0, q_1, q_f\}$$

$$\Sigma = \{a, b\}$$

$$\Gamma = \{S, A, B, C, Z_0\}$$

δ : is shown below:

$$\begin{aligned}
 \delta(q_0, \epsilon, Z_0) &= (q_1, SZ_0) \\
 \delta(q_1, a, S) &= (q_1, ABC) \\
 \delta(q_1, a, A) &= (q_1, B) \\
 \delta(q_1, a, A) &= (q_1, \epsilon) \\
 \delta(q_1, b, B) &= (q_1, A) \\
 \delta(q_1, b, B) &= (q_1, \epsilon) \\
 \delta(q_1, a, C) &= (q_1, \epsilon) \\
 \delta(q_1, \epsilon, Z_0) &= (q_f, Z_0)
 \end{aligned}$$

$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{q_f\}$ is the final state.

Note that the terminals of grammar G will be the input symbols in PDA and the non-terminals will be the stack symbols in PDA. The derivation from the grammar is shown below:

$$\begin{aligned} S &\Rightarrow aABC \\ &\Rightarrow aaBBC \\ &\Rightarrow aabBC \\ &\Rightarrow aabbC \\ &\Rightarrow aabba \end{aligned}$$

The string aabba is derived from the start symbol S. The same string should be accepted by PDA also. The moves made by the PDA are shown below:

Initial ID

$$\begin{aligned} (q_0, aabba, Z_0) &\vdash (q_1, aabba, SZ_0) \quad \text{By Rule 1} \\ &\vdash (q_1, abba, ABCZ_0) \quad \text{By Rule 2} \\ &\vdash (q_1, bba, BBCZ_0) \quad \text{By Rule 3} \\ &\vdash (q_1, ba, BCZ_0) \quad \text{By Rule 6} \\ &\vdash (q_1, a, CZ_0) \quad \text{By Rule 6} \\ &\vdash (q_1, \epsilon, Z_0) \quad \text{By Rule 7} \\ & (q_f, \epsilon, Z_0) \quad \text{By Rule 8 (Final configuration)} \end{aligned}$$

Since q_f is the final state and input string is ϵ in the final configuration, the string

aabba

is accepted by the PDA.

■ **Example 5.19:** For the grammar

$$\begin{aligned} S &\rightarrow aABB \mid aAA \\ A &\rightarrow aBB \mid a \end{aligned}$$

$$\begin{array}{lcl} B & \rightarrow & bBB \mid A \\ C & \rightarrow & a \end{array}$$

obtain the corresponding PDA.

Note: To obtain a PDA from CFG, the grammar should be in GNF. All the productions except the production

$$B \rightarrow A$$

are in GNF. Since one of the production is not in GNF, the given grammar is not in GNF. This can be easily converted into GNF. Note that all A productions are in GNF. So, by substituting for A in the above production we get B-productions also in GNF as shown below:

$$B \rightarrow bBB \mid aBB \mid a$$

Now, the new grammar which is in GNF can take the form:

$$\begin{array}{lcl} S & \rightarrow & aABB \mid aAA \\ A & \rightarrow & aBB \mid a \\ B & \rightarrow & bBB \mid aBB \mid a \\ C & \rightarrow & a \end{array}$$

Now, the CFG can be converted into PDA. Let q_0 be the start state and Z_0 the initial symbol on the stack.

Step 1: Push the start symbol S on to the stack and change the state to q_1 . The transition for this can be of the form

$$\delta(q_0, \epsilon, Z_0) = (q_1, SZ_0)$$

Step 2: For each production $A \rightarrow a\alpha$ introduce the transition

$$\delta(q_1, a, A) = (q_1, \alpha)$$

The following table shows how the various transitions are obtained from the productions from the CFG.

Production		Transition	
S → aABB		$\delta(q_1, a, S)$	= (q ₁ , ABB)
S → aAA		$\delta(q_1, a, S)$	= (q ₁ , AA)
A → aBB		$\delta(q_1, a, A)$	= (q ₁ , BB)
A → a		$\delta(q_1, a, A)$	= (q ₁ , ε)
B → bBB		$\delta(q_1, b, B)$	= (q ₁ , BB)
B → aBB		$\delta(q_1, a, B)$	= (q ₁ , BB)
B → a		$\delta(q_1, a, B)$	= (q ₁ , ε)
C → a		$\delta(q_1, a, C)$	= (q ₁ , ε)

Step 3: Finally in state q₁, without consuming any input change the state to q_f which is an accepting state, i.e.,

$$\delta(q_1, \epsilon, Z_0) = (q_f, Z_0)$$

So, the PDA M is given by $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

where

$$Q = \{q_0, q_1, q_f\}$$

$$\Sigma = \{a, b\}$$

$$\Gamma = \{S, A, B, C, Z_0\}$$

δ : is shown below:

$\delta(q_0, \epsilon, Z_0)$	=	(q ₁ , SZ ₀)
$\delta(q_1, a, S)$	=	(q ₁ , ABB)
$\delta(q_1, a, S)$	=	(q ₁ , AA)
$\delta(q_1, a, A)$	=	(q ₁ , BB)
$\delta(q_1, a, A)$	=	(q ₁ , ε)
$\delta(q_1, b, B)$	=	(q ₁ , BB)
$\delta(q_1, a, B)$	=	(q ₁ , BB)
$\delta(q_1, a, B)$	=	(q ₁ , ε)
$\delta(q_1, a, C)$	=	(q ₁ , ε)
$\delta(q_1, \epsilon, Z_0)$	=	(q _f , Z ₀)

$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{q_f\}$ is the final state.

Note that the terminals of grammar G will be the input symbols in PDA and the non-terminals will be the stack symbols in PDA. It is left to the reader to derive the string from grammar G and to show that the same string can be generated by the PDA which can be done similar to the previous problem.

5.8. Application of GNF

Note: Instead of using the method shown in section 5.5, it is much easier to obtain CFG, convert it into GNF and then obtain the PDA. Obtain the GNF notation for those grammars and use the approach given in previous section, to obtain the PDAs (FG GNF, see Section 6.7).

This section provides some of the grammars and their equivalent PDA just to show how easy to obtain a PDA if the grammar is in GNF.

■ **Example 5.20:** Obtain a the PDA to accept the language $L = \{a^n b^n \mid n \geq 1\}$.

The equivalent CFG to generate the language is

$$\begin{array}{l} S \rightarrow aSb \\ S \rightarrow ab \end{array}$$

The corresponding grammar which is in GNF is

$$\begin{array}{l} S \rightarrow aSB \\ S \rightarrow aB \\ B \rightarrow b \end{array}$$

The equivalent transitions for the above productions are:

$$\begin{array}{lcl} \delta(q_1, a, S) & = & (q_1, SB) \\ \delta(q_1, a, S) & = & (q_1, B) \\ \delta(q_1, b, B) & = & (q_1, \epsilon) \end{array}$$

In Section 5.7, we have seen that the transition

$$\delta(q_0, \epsilon, Z_0) = (q_1, SZ_0)$$

is used to push S on to the stack initially and the last transition

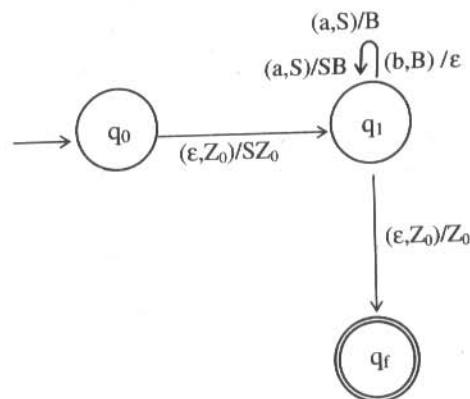
$$\delta(q_1, \epsilon, Z_0) = (q_f, Z_0)$$

is used to move to the final state. So, the PDA is given by

$$M = (\{q_0, q_1, q_f\}, \{a, b\}, \{S, A, B, Z_0\}, \delta, q_0, Z_0, q_f)$$

where δ is shown using the table. The corresponding transition diagram or the state diagram is also shown:

$$\begin{aligned}\delta(q_0, \epsilon, Z_0) &= \\ \delta(q_1, a, S) &= (q_1, SB) \\ \delta(q_1, a, S) &= \\ \delta(q_1, b, B) &= (q_f, \epsilon) \\ \delta(q_1, \epsilon, Z_0) &= (q_f, Z_0)\end{aligned}$$



Example 5.21: Obtain a the PDA to accept the language $L = \{ww^R : |w| \geq 1 \text{ for } w \in (a+b)^*\}$.

The equivalent CFG to generate the language is

$$\begin{aligned}S &\rightarrow aSa \mid aa \\ S &\rightarrow bSb \mid bb\end{aligned}$$

The corresponding grammar which is in GNF is

$$\begin{aligned}S &\rightarrow aSA \mid aA \\ S &\rightarrow bSB \mid bB \\ A &\rightarrow a \\ B &\rightarrow b\end{aligned}$$

The equivalent transitions for the above productions are:

$$\begin{aligned}
 \delta(q_1, a, S) &= \{(q_1, SA), (q_1, A)\} \\
 \delta(q_1, b, S) &= \{(q_1, SB), (q_1, B)\} \\
 \delta(q_1, a, A) &= (q_1, \epsilon) \\
 \delta(q_1, b, B) &= (q_1, \epsilon)
 \end{aligned}$$

In Section 5.7, we have seen that the transition

$$\delta(q_0, \epsilon, Z_0) = (q_1, SZ_0)$$

is used to push S on to the stack initially and the last transition

$$\delta(q_1, \epsilon, Z_0) = (q_f, Z_0)$$

is used to move to the final state. So, the PDA is given by

$$M = (\{q_0, q_1, q_f\}, \{a, b\}, \{S, A, B, Z_0\}, \delta, q_0, Z_0, q_f)$$

where δ is shown below:

$$\begin{aligned}
 \delta(q_0, \epsilon, Z_0) &= (q_1, SZ_0) \\
 \delta(q_1, a, S) &= \{(q_1, SA), (q_1, A)\} \\
 \delta(q_1, b, S) &= \{(q_1, SB), (q_1, B)\} \\
 \delta(q_1, a, A) &= (q_1, \epsilon) \\
 \delta(q_1, b, B) &= (q_1, \epsilon) \\
 \delta(q_1, \epsilon, Z_0) &= (q_f, Z_0)
 \end{aligned}$$

5.9. PDA to CFG

As we have converted CFG to PDA, we can convert a given PDA to CFG. The general procedure for this conversion is shown below:

1. The input symbols of PDA will be the terminals of CFG.

2. If the PDA moves from state to q_i to state q_j on consuming the input $a \in \Sigma$ when Z is the top of the stack, then the non-terminals of CFG are the triplets of the form $(q_i Z q_j)$.
3. If q_0 is the start state and q_f is the final state then $(q_0 Z q_f)$ is the start symbol of CFG.
4. The productions of CFG can be obtained from the transitions of PDA as shown below:

- a. For each transition of the form

$$\delta(q_i, a, Z) = (q_j, AB)$$

introduce the productions of the form

$$(q_i Z q_j) \rightarrow a (q_j A q_j) (q_j B q_j)$$

where q_k and q_l will take all possible values from Q .

- b. For each transition of the form

$$\delta(q_i, a, Z) = (q_j, \epsilon)$$

introduce the production

$$(q_i Z q_j) \rightarrow a$$

Note: Using this procedure, we may introduce lot of useless symbols, which in any way can be eliminated.

■ **Example 5.22:** Obtain a CFG for the PDA shown below:

$$\delta(q_0, a, Z) = (q_0, AZ)$$

$$\delta(q_0, a, A) = (q_0, A)$$

$$\delta(q_0, b, A) = (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, Z) = (q_2, \epsilon)$$

Note: To obtain a CFG from the PDA, all the transitions should be of the form

$$\delta(q_i, a, Z) = (q_j, AB)$$

or

$$\delta(q_i, a, Z) = (q_j, \epsilon)$$

In the given transitions except the second transition, all transitions are in the required form. So, let us take the second transition

$$\delta(q_0, a, A) = (q_0, A)$$

and convert it into the required form. This can be achieved if we have understood what the transition indicates. It is clear from the transition that when input symbol a is encountered and top of the stack is A , the PDA remains in state q_0 and contents of the stack are not altered. This can be interpreted as delete A from the stack and insert A onto the stack.

So, once A is deleted from the stack we enter into new state q_3 . But, in state q_3 without consuming any input we add A on to the stack. The corresponding transitions are:

$$\begin{aligned}\delta(q_0, a, A) &= (q_3, \epsilon) \\ \delta(q_3, \epsilon, Z) &= (q_0, AZ)\end{aligned}$$

So, the given PDA can be written using the following transitions:

$$\begin{aligned}\delta(q_0, a, Z) &= (q_0, AZ) \\ \delta(q_0, a, A) &= (q_3, \epsilon) \\ \delta(q_3, \epsilon, Z) &= (q_0, AZ) \\ \delta(q_0, b, A) &= (q_1, \epsilon) \\ \delta(q_1, \epsilon, Z) &= (q_2, \epsilon)\end{aligned}$$

Now, the transitions

$$\begin{aligned}\delta(q_0, a, A) &= (q_3, \epsilon) \\ \delta(q_0, b, A) &= (q_1, \epsilon) \\ \delta(q_1, \epsilon, Z) &= (q_2, \epsilon)\end{aligned}$$

can be converted into productions as shown below:

For δ of the form $\delta(q_i, a, Z) = (q_j, \epsilon)$	Resulting Productions $(q_i Z q_j) \rightarrow a$
$\delta(q_0, a, A) = (q_3, \epsilon)$	$(q_0 A q_3) \rightarrow a$
$\delta(q_0, b, A) = (q_1, \epsilon)$	$(q_0 A q_1) \rightarrow b$
$\delta(q_1, \epsilon, Z) = (q_2, \epsilon)$	$(q_1 Z q_2) \rightarrow \epsilon$

Now, the transitions

$$\delta(q_0, a, Z) = (q_0, AZ)$$

$$\delta(q_3, \epsilon, Z) = (q_0, AZ)$$

can be converted into productions using rule 4.a as shown below:

For δ of the form $\delta(q_i, a, Z) = (q_j, AB)$	Resulting Productions $(q_i Z q_k) \rightarrow a (q_j A q_j) (q_j B q_k)$
$\delta(q_0, a, Z) = (q_0, AZ)$	$(q_0 Z q_0) \rightarrow a (q_0 A q_0) (q_0 Z q_0) \mid a (q_0 A q_1) (q_1 Z q_0) \mid$ $a (q_0 A q_2) (q_2 Z q_0) \mid a (q_0 A q_3) (q_3 Z q_0)$ $(q_0 Z q_1) \rightarrow a (q_0 A q_0) (q_0 Z q_1) \mid a (q_0 A q_1) (q_1 Z q_1) \mid$ $a (q_0 A q_2) (q_2 Z q_1) \mid a (q_0 A q_3) (q_3 Z q_1)$ $(q_0 Z q_2) \rightarrow a (q_0 A q_0) (q_0 Z q_2) \mid a (q_0 A q_1) (q_1 Z q_2) \mid$ $a (q_0 A q_2) (q_2 Z q_2) \mid a (q_0 A q_3) (q_3 Z q_2)$ $(q_0 Z q_3) \rightarrow a (q_0 A q_0) (q_0 Z q_3) \mid a (q_0 A q_1) (q_1 Z q_3) \mid$ $a (q_0 A q_2) (q_2 Z q_3) \mid a (q_0 A q_3) (q_3 Z q_3)$
$\delta(q_3, \epsilon, Z) = (q_0, AZ)$	$(q_3 Z q_0) \rightarrow (q_0 A q_0) (q_0 Z q_0) \mid (q_0 A q_1) (q_1 Z q_0) \mid$ $(q_0 A q_2) (q_2 Z q_0) \mid (q_0 A q_3) (q_3 Z q_0)$ $(q_3 Z q_1) \rightarrow (q_0 A q_0) (q_0 Z q_1) \mid (q_0 A q_1) (q_1 Z q_1) \mid$ $(q_0 A q_2) (q_2 Z q_1) \mid (q_0 A q_3) (q_3 Z q_1)$ $(q_3 Z q_2) \rightarrow (q_0 A q_0) (q_0 Z q_2) \mid (q_0 A q_1) (q_1 Z q_2) \mid$ $(q_0 A q_2) (q_2 Z q_2) \mid (q_0 A q_3) (q_3 Z q_2)$ $(q_3 Z q_3) \rightarrow (q_0 A q_0) (q_0 Z q_3) \mid (q_0 A q_1) (q_1 Z q_3) \mid$ $(q_0 A q_2) (q_2 Z q_3) \mid (q_0 A q_3) (q_3 Z q_3)$

The start symbol of the grammar will be $q_0 Z q_2$.

Example 5.23: Obtain a CFG that generates the language accepted by PDA $M = (\{q_0, q_1\}, \{a, b\}, \{A, Z\}, \delta, q_0, Z, \{q_1\})$, with the transitions

$$\delta(q_0, a, Z) = (q_0, AZ)$$

$$\delta(q_0, b, A) = (q_0, AA)$$

$$\delta(q_0, a, A) = (q_1, \epsilon)$$

Now, the transition

$$\delta(q_0, a, A) = (q_1, \epsilon)$$

can be converted into production as shown below:

For δ of the form $\delta(q_i, a, Z) = (q_j, \epsilon)$	Resulting Productions $(q_i Z q_j) \rightarrow a$
$\delta(q_0, a, A) = (q_1, \epsilon)$	$(q_0 A q_1) \rightarrow a$

Now, the transitions

$$\delta(q_0, a, Z) = (q_0, AZ)$$

$$\delta(q_0, b, A) = (q_0, AA)$$

can be converted into productions using rule 4.a as shown below:

For δ of the form $\delta(q_i, a, Z) = (q_j, AB)$	Resulting Productions $(q_i Z q_k) \rightarrow a (q_j A q_l)(q_k B q_m)$
$\delta(q_0, a, Z) = (q_0, AZ)$	$(q_0 Z q_0) \rightarrow a (q_0 A q_0)(q_0 Z q_0) \mid a (q_0 A q_1)(q_1 Z q_0) (q_0 Z q_1) \rightarrow a (q_0 A q_0)(q_0 Z q_1) \mid a (q_0 A q_1)(q_1 Z q_1)$
$\delta(q_0, b, A) = (q_0, AA)$	$(q_0 A q_0) \rightarrow b (q_0 A q_0)(q_0 A q_0) \mid b (q_0 A q_1)(q_1 A q_0)$ $(q_0 A q_1) \rightarrow b (q_0 A q_0)(q_0 A q_1) \mid b (q_0 A q_1)(q_1 A q_1)$

The start symbol of the grammar will be $q_0 Z q_1$.

Exercises

- What are the demerits of regular languages when compared to context free languages?
- What are the demerits of DFA (or NFA) when compared with PDA?
- Why FAs are less powerful than the PDAs?
- What is the difference between NFA and PDA?
- What is a PDA? Explain with an example.
- What does each of the following transitions represent?
 - $\delta(p, a, Z) = (q, aZ)$
 - $\delta(p, a, Z) = (q, \epsilon)$
 - $\delta(p, a, Z) = (q, r)$
 - $\delta(p, \epsilon, Z) = (q, r)$

- e. $\delta(p, \epsilon, \epsilon) = (q, Z)$
- f. $(p, \epsilon, Z) = (q, \epsilon)$
7. How the transition/move of a PDA defined?
8. What is an instantaneous description? Explain with respect to PDA.
9. When a language is accepted by a final state and when a language is accepted by an empty stack?
10. Obtain a PDA to accept the language $L(M) = \{wCw^R \mid w \in (a+b)^*\}$ where W^R is reverse of W . Show the sequence of moves made by the PDA for the strings $aabCbba$, $aabCbab$.
11. Obtain a PDA to accept the language $L = \{a^n b^n \mid n \geq 1\}$ by a final state.
12. Obtain a PDA to accept the language $L(M) = \{w \mid w \in (a+b)^* \text{ and } n_a(w) = n_b(w) \text{ i.e., number of } a's \text{ in string } w \text{ should be equal to number of } b's \text{ in } w\}$.
13. Obtain a PDA to accept a string of balanced parentheses. The parentheses to be considered are $(,)$, $[,]$, ${$ and $}$.
14. Obtain a PDA to accept the language $L = \{w \mid w \in (a, b)^* \text{ and } n_a(w) > n_b(w)\}$.
15. Obtain a PDA to accept the language $L = \{a^n b^{2n} \mid n \geq 1\}$.
16. Obtain a PDA to accept the language $L = \{ww^R \mid w \in (a+b)^*\}$.
17. When the PDA is deterministic and when it is called non-deterministic?
18. Is the PDA to accept the language $L(M) = \{wCw^R \mid w \in (a+b)^*\}$ deterministic?
19. Is the PDA corresponding to the language $L = \{a^n b^n \mid n \geq 1\}$ by a final state deterministic?
20. Is the PDA to accept the language $L(M) = \{w \mid w \in (a+b)^* \text{ and } n_a(w) = n_b(w) \text{ is deterministic?}$
21. Is the PDA to accept the language consisting of balanced parentheses is deterministic?
22. Is the PDA to accept the language $L = \{w \mid w \in (a, b)^* \text{ and } n_a(w) > n_b(w)\}$ is deterministic?
23. Is the PDA to accept the language $L = \{a^n b^{2n} \mid n \geq 1\}$ is deterministic?
24. Is the PDA to accept the language $L = \{ww^R \mid w \in (a+b)^*\}$ is deterministic?
25. What is the procedure to convert a CFG to PDA?

26. For the grammar

$$\begin{array}{lcl} S & \rightarrow & aABC \\ A & \rightarrow & aB|a \\ B & \rightarrow & bA|b \\ C & \rightarrow & a \end{array}$$

obtain the corresponding PDA.

27. For the grammar

$$\begin{array}{lcl} S & \rightarrow & aABB|aAA \\ A & \rightarrow & aBB|a \\ B & \rightarrow & bBB|A \\ C & \rightarrow & a \end{array}$$

obtain the corresponding PDA.

28. What is the application of GNF notation of a CFG?

29. Obtain a the PDA to accept the language $L = \{a^n b^n \mid n \geq 1\}$.

30. Obtain a the PDA to accept the language $L = \{ww^R : |w| \geq 1 \text{ for } w \in (a+b)^*\}$.

31. What is the general procedure used to convert from PDA to CFG?

32. Obtain a CFG for the PDA shown below:

$$\begin{array}{lll} \delta(q_0, a, Z) & = & (q_0, AZ) \\ \delta(q_0, a, A) & = & (q_0, A) \\ \delta(q_0, b, A) & = & (q_1, \epsilon) \\ \delta(q_1, \epsilon, Z) & = & (q_2, \epsilon) \end{array}$$

33. Obtain a CFG that generates the language accepted by PDA $M = (\{q_0, q_1\}, \{a, b\}, \{A, Z\}, \delta, q_0, Z, \{q_1\})$, with the transitions

$$\begin{array}{lll} \delta(q_0, a, Z) & = & (q_0, AZ) \\ \delta(q_0, b, A) & = & (q_0, AA) \\ \delta(q_0, a, A) & = & (q_1, \epsilon) \end{array}$$

Chapter 6

Properties of Context Free Languages

What are we studying in this chapter . . .

- ▶ *Normal forms for CFG*
- ▶ *The pumping lemma for CFGs*
- ▶ *Closure properties of CFLs*

Even though there is no restriction on the right hand side of the production for any CFG, it is better in fact necessary to eliminate some of the useless symbols and productions. In the grammar G, some of the symbols or productions may not be used to derive a string. Some symbols and productions may never be used while deriving a string. So, these symbols and productions which will never be used are useless and the corresponding productions can be eliminated. For example, consider the grammar

$$S \rightarrow aA \mid B$$

$$A \rightarrow aA \mid a$$

In this grammar if we apply the production $S \rightarrow B$, a string can never be derived. So, the symbol B and the production $S \rightarrow B$ are useless and can be eliminated. In the following sections, we discuss how to eliminate

- symbols in V from which string of terminals can not be derived
- symbols in $(V \cup T)$ and not appearing in any sentential form
- ϵ -productions
- The productions of the form $A \rightarrow B$ i.e., unit productions

Thus, a CFG can be simplified by eliminating ϵ -productions, useless symbols, unit production etc. This chapter covers the simplification process and two normal forms: Chomsky normal form and Greibach normal form.

6.1. Substitution

Sections 6.1 and 6.2 provides different substitution methods. This section deals with a simple substitution wherein a non-terminal is replaced by the corresponding symbols on the right hand side.

Theorem 6.1: Let $G = (V, T, P, S)$ be a context free grammar. Consider the productions

$$A \rightarrow x_1 B x_2$$

and

$$B \rightarrow y_1 | y_2 | \dots | y_n$$

The production

$$A \rightarrow x_1 B x_2$$

can be replaced by

$$A \rightarrow x_1 y_1 x_2 | x_1 y_2 x_2 | \dots | x_1 y_n x_2$$

and the production

$$B \rightarrow y_1 | y_2 | \dots | y_n$$

in P can be deleted. The resulting productions are added to P_1 and the variables are added to V_1 . The language generated by the resulting grammar $G_1 = (V_1, T, P_1, S)$ is same as the language accepted by G , i.e., $L(G_1) = L(G)$.

■ **Example 6.1:** Consider the productions

$$A \rightarrow aBa$$

$$B \rightarrow ab | b$$

Simplify the grammar by substitution method.

Consider the production

$$A \rightarrow aBa$$

The right hand side of the production contains a non terminal B. The non-terminal B can be replaced by the production

$$B \rightarrow ab \mid b$$

as shown below:

$$A \rightarrow aaba \mid aba$$

So, the resulting grammar is $G = (V, T, P, S)$ where

$$V = \{A\}$$

$$T = \{a, b\}$$

$$P = \{A \rightarrow aaba \mid aba\}$$

A is the start symbol

6.2. Left Recursion

A grammar can be changed from one form to another accepting the same language. Another important substitution method which is often useful is left recursion. If a grammar has left recursive property, it is undesirable as the parser constructed from this left recursive grammar will enter into an infinite loop and the system may crash. For this reason left recursion should be eliminated from the grammar. The left recursion is defined as follows.

❖ **Definition:** A grammar G is said to be left recursive if there is some non-terminal A such that

$$A \xrightarrow{*} A\alpha$$

In other words, if the first symbol on the right hand side in a sentential form (either left sentential or right sentential) is a variable and if the derivation is obtained from the same non-terminal, then we say that the grammar is having left recursion.

The left recursion in a grammar G can be eliminated as shown below.

Consider the A-production of the form

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \dots A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \dots \beta_m$$

where β_i 's do not start with A. Then the A productions can be replaced by

$$A \rightarrow \beta_1 A^1 | \beta_2 A^1 | \beta_3 A^1 | \dots | \beta_m A^1$$

$$A^1 \rightarrow \alpha_1 A^1 | \alpha_2 A^1 | \alpha_3 A^1 | \dots | \alpha_n A^1 | \epsilon$$

Note that α_i 's do not start with A^1 .

Example 6.2: Eliminate left recursion from the following grammar:

$$E \rightarrow E + T | T$$

$$T \rightarrow T * F | F$$

$$F \rightarrow (E) | id$$

The left recursion can be eliminated as shown below:

Given $A \rightarrow A\alpha_i \beta_i$	Substitution	Without left recursion $A \rightarrow \beta_i A^1$ and $A^1 \rightarrow \alpha_i A^1 \epsilon$
$E \rightarrow E + T T$	$A = E$ $\alpha_1 = +T$ $\beta_1 = T$	$E \rightarrow TE^1$ $E^1 \rightarrow + TE^1 \epsilon$
$T \rightarrow T * F F$	$A = T$ $\alpha_1 = *F$ $\beta_1 = F$	$T \rightarrow FT^1$ $T^1 \rightarrow *FT^1 \epsilon$
$F \rightarrow (E) id$	Not applicable	$F \rightarrow (E) id$

The grammar obtained after eliminating left recursion is

$$E \rightarrow TE^1$$

$$E^1 \rightarrow + TE^1 | \epsilon$$

$$T \rightarrow FT^1$$

$$T^1 \rightarrow *FT^1 | \epsilon$$

$$F \rightarrow (E) | id$$

Example 6.3: Eliminate left recursion from the following grammar:

$$S \rightarrow Ab | a$$

$$A \rightarrow Ab | Sa$$

The non terminal S, even though is not having immediate left recursion, it has left recursion since

$$S \Rightarrow Ab \Rightarrow Sab$$

i.e.,

$$S \stackrel{+}{\Rightarrow} Sab.$$

Substituting for S in the A-production can eliminate the indirect left recursion from S. So, the given grammar can be written as

$$S \rightarrow Ab \mid a$$

$$A \rightarrow Ab \mid Aba \mid aa$$

Now, A-production has left recursion and can be eliminated as shown below:

Given $A \rightarrow A\alpha_i \mid \beta_i$	Substitution	Without left recursion $A \rightarrow \beta_i A^l \text{ and } A^l \rightarrow \alpha_i A^l \mid \epsilon$
$S \rightarrow Ab \mid a$	Not applicable	$S \rightarrow Ab \mid a$
$A \rightarrow Ab \mid Aba \mid aa$	$A = A$ $\alpha_1 = b$ $\alpha_2 = ba$ $\beta_1 = aa$	$A \rightarrow aaA^l$ $A^l \rightarrow bA^l \mid baA^l \mid \epsilon$

The grammar obtained after eliminating left recursion is

$$S \rightarrow Ab \mid a$$

$$A \rightarrow aaA^l$$

$$A^l \rightarrow bA^l \mid baA^l \mid \epsilon$$

6.3. Simplification of CFG

In a CFG, it may be necessary to eliminate some of the useless symbols and productions. Let $G = (V, T, P, S)$ be a CFG. In the grammar G, some of the symbols or productions may not be used to derive a string and some symbols and productions may not be reachable from the start symbol. So, these symbols and productions which are not used in any sentential from are useless and the corresponding productions can be eliminated. For example, consider the grammar

$$S \rightarrow aA \mid B$$

$$A \rightarrow aA \mid a$$

In this grammar if we apply the production $S \rightarrow B$, from B , a string can never be derived. So, the symbol B and the production $S \rightarrow B$ are useless and can be eliminated. So, in this section, let us concentrate on how a grammar can be simplified by eliminating useless symbols and variables.

Theorem 6.2: Let $G = (V, T, P, S)$ be a CFG. We can find an equivalent grammar $G^1 = (V^1, T^1, P^1, S)$ such that for each A in $(V^1 \cup T^1)^*$ there exists α and β in $(V^1 \cup T^1)^*$ and x in T^* for which

$$S \stackrel{+}{\Rightarrow} \alpha A \beta \stackrel{+}{\Rightarrow} x.$$

Note: It means that any variable or symbols which are not reachable from the start symbol and which are not used while deriving a string of terminals the symbols are useless and all productions which contains those symbols are also useless.

Proof: The grammar G^1 can be obtained from G in two steps.

Stage 1: Obtain the set of variables and productions which derive only string of terminals, i.e., Obtain a grammar $G_1 = (V_1, T_1, P_1, S)$ such that V_1 contains only the set of variables A for which

$$A \stackrel{+}{\Rightarrow} x$$

where $x \in T^*$. The algorithm to obtain a set of variables from which only string of terminals can be derived is shown below.

Step 1: [Initialize old_variables denoted by ov to \emptyset]

$$ov = \emptyset$$

Step 2: Take all productions of the form $A \rightarrow x$ where $x \in T^*$, i.e., if the R.H.S of the production contains only string of terminals consider those productions and corresponding non terminals on L.H.S. are added to new_variables denoted by nv . This can be expressed using the following statement:

$$nv = \{A \mid A \rightarrow x \text{ and } x \in T^*\}$$

Step 3: Compare ov and nv . As long as the elements in ov and nv are not equal, repeat the following statements. Otherwise go to step 4.

a) [Copy new_variables to old_variables]

$$ov = nv$$

a) Add all the elements in ov to nv . Also add the variables which derive a string consisting of terminals and non-terminals which are in ov , i.e.,

$$nv = ov \cup \{A \mid A \rightarrow y \text{ and } y \in (ov \cup T)^*\}$$

Step 3 can be written in algorithmic notation as

```

while(ov != nv)
{
    ov = nv;
    nv = ov U{A | A → y and y ∈ (ov U T)*}
}

```

Step 4: When the loop is terminated, nv(or ov) contains all those non-terminals from which only the string of terminals are derived and add those variables to V_1 .

i.e., $V_1 = ov$

Step 5: [Terminate the algorithm]

return V_1

The complete algorithm is shown below:

```

ov = φ
nv = ov U{A | A → y and y ∈ (ov U T)* }
while (ov != nv)
{
    ov = nv;
    nv = ov U{A | A → y and y ∈ (ov U T)*}
}
 $V_1 = ov$ 

```

Note that the variable V_1 contains only those variables from which string of terminals are obtained. The productions used to obtain V_1 are added to P_1 and the terminals in these productions are added to T_1 . The grammar $G_1 = (V_1, T_1, P_1, S)$ contains those variables A in V_1 such that

$$A \xrightarrow{*} x$$

for some x in T^* . Since each derivation in G_1 is a derivation of G ,

$$L(G_1) = L(G)$$

Stage 2: Obtain the set of variables and terminals which are reachable from the start symbol. The productions which are not used are useless. This can be obtained as shown below:

Given a CFG $G_1 = (V_1, T_1, P_1, S)$, we can find an equivalent grammar $G^1 = (V^1, T^1, P^1, S)$ such that for each X in $(V^1 \cup T^1)$ there exists some α such that

$$S \xrightarrow{*} \alpha$$

where X is a symbol in α i.e., if X is a variable, $X \in V^1$ and if X is a terminal $X \in T^1$. Each symbol X in $(V^1 \cup T^1)$ is reachable from the start symbol S . The algorithm for this is shown below:

$$V^1 = \{S\}$$

For each A in V^1

If $A \rightarrow \alpha$ then

Add the variables in α to V^1

Add the terminals in α to T^1

Endif

Endfor

Using this algorithm all those symbols (whether variables or terminals) that are not reachable from the start symbol are eliminated. The grammar G^1 does not contain any useless symbol or production. For each $x \in L(G^1)$ there is a derivation

$$S \xrightarrow{*} \alpha X \beta \xrightarrow{*} x$$

Using these two steps we can effectively find G^1 such that $L(G) = L(G^1)$ and the two grammars G and G^1 are equivalent.

❖ **Definition:** A symbol X is useful if there is a derivation of the form

$$S \xrightarrow{*} \alpha X \beta \xrightarrow{*} w$$

Otherwise, the symbol X is useless. Note that in a derivation, finally we should get string of terminals and all these symbols must be reachable from the start symbol S . Those symbols and productions which are not at all used in the derivation are useless.

■ **Example 6.4:** Eliminate the useless symbols in the grammar

$$S \rightarrow aA \mid bB$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow bB$$

$$D \rightarrow ab \mid Ea$$

$$E \rightarrow aC \mid d$$

Stage 1: Applying the algorithm shown in stage 1, we can obtain a set of variables from which we get only string of terminals and is shown below.

ov	nv	Productions
ϕ	A, D, E	$A \rightarrow a$ $D \rightarrow ab$ $E \rightarrow d$
A,D,E	A,D,E,S	$S \rightarrow aA$ $A \rightarrow aA$ $D \rightarrow Ea$
A,D,E,S	A,D,E,S	-

The resulting grammar $G_1 = (V_1, T_1, P_1, S)$ where

$$V_1 = \{A, D, E, S\}$$

$$T_1 = \{a, b, d\}$$

$$P_1 = \{$$

$$A \rightarrow a \mid aA$$

$$D \rightarrow ab \mid Ea$$

$$E \rightarrow d$$

$$S \rightarrow aA$$

}

S is the start symbol

contains all those variables in V_1 such that $A \xrightarrow{*} w$ where $w \in T^*$.

Stage 2: Applying the algorithm given in stage 2 of Theorem 6.2, we obtain the symbols such that each symbol X is reachable from the start symbol S as shown below:

P^1	T^1	V^1
-	-	S
$S \rightarrow aA$	a	S, A
$A \rightarrow a \mid aA$	a	S, A

The resulting grammar $G^1 = (V^1, T^1, P^1, S)$ where

$$V^1 = \{ S, A \}$$

$$T^1 = \{ a \}$$

$$P^1 = \{$$

$$S \rightarrow aA$$

$$A \rightarrow a \mid aA$$

}

S is the start symbol

such that each symbol X in $(V^1 \cup T^1)$ has a derivation of the form

$$S \xrightarrow{*} \alpha X \beta \xrightarrow{*} w.$$

■ Example 6.5: Simplify the following grammar

$$S \rightarrow aA \mid a \mid Bb \mid cC$$

$$A \rightarrow aB$$

$$B \rightarrow a \mid Aa$$

$$C \rightarrow cCD$$

$$D \rightarrow ddd$$

Stage 1: Applying the algorithm shown in stage 1 of Theorem 6.2, we can obtain a set of variables from which we get only string of terminals and is shown below:

ov	nv	Productions
ϕ	S, B, D	$S \rightarrow a$ $B \rightarrow a$ $D \rightarrow ddd$

S, B, D	S, B, D, A	S → Bb A → aB
S, B, D, A	S, B, D, A	S → aA B → Aa

The resulting grammar $G_1 = (V_1, T_1, P_1, S)$ where

$$V_1 = \{S, B, D, A\}$$

$$T_1 = \{a, b, d\}$$

$$\begin{aligned} P_1 = \{ & \\ & S \rightarrow a \mid Bb \mid aA \\ & B \rightarrow a \mid Aa \\ & D \rightarrow ddd \\ & A \rightarrow aB \\ \} \end{aligned}$$

S is the start symbol

contains all those variables in V_1 such that $A \stackrel{*}{\Rightarrow} w$.

Stage 2: Applying the algorithm given in stage 2 of Theorem 6.2, we obtain the symbols such that each symbol X is reachable from the start symbol S as shown below:

P ¹	T ¹	V ¹
-	-	S
$S \rightarrow a \mid Bb \mid Aa$	a, b	S, A, B
$A \rightarrow aB$	a, b	S, A, B
$B \rightarrow a \mid Aa$	a, b,	S, A, B

The resulting grammar $G^1 = (V^1, T^1, P^1, S)$ where

$$V^1 = \{S, A, B\}$$

$$T^1 = \{a, b\}$$

$$\begin{aligned} P^1 = \{ & \\ & S \rightarrow a \mid Bb \mid aA \\ & A \rightarrow aB \\ & B \rightarrow a \mid Aa \\ \} \end{aligned}$$

S is the start symbol

such that each symbol X in $(V^1 \cup T^1)$ has a derivation of the form

$$S \xrightarrow{*} \alpha X \beta \xrightarrow{*} w.$$

6.4. Eliminating ϵ -productions

A production of the form $A \rightarrow \epsilon$ is undesirable in a CFG, unless an empty string is derived from the start symbol. Suppose, the language generated from a grammar G does not derive any empty string and the grammar consists of ϵ -productions. Such ϵ -productions can be removed. An ϵ -production is defined as follows:

❖ **Definition:** Let $G = (V, T, P, S)$ be a CFG. A production in P of the form

$$A \rightarrow \epsilon$$

is called an ϵ -production or NULL production. After applying the production the variable A is erased. For each A in V , if there is a derivation of the form

$$A \xrightarrow{*} \epsilon$$

then A is a nullable variable.

■ **Example 6.6:** Consider the grammar

$$S \rightarrow ABCa \mid bD$$

$$A \rightarrow BC \mid b$$

$$B \rightarrow b \mid \epsilon$$

$$C \rightarrow c \mid \epsilon$$

$$D \rightarrow d$$

In this grammar, the productions

$$B \rightarrow \epsilon$$

$$C \rightarrow \epsilon$$

are ϵ -productions and the variables B, C are nullable variables. Because there is a production

$$A \rightarrow BC$$

and both B and C are nullable variables, then A is also a nullable variable.

❖ **Definition:** Let $G = (V, T, P, S)$ be a CFG where V set of variables, T is set of terminals, P is set of productions and S is the start symbol. A nullable variable is defined as follows.

1. If $A \rightarrow \epsilon$ is a production in P , then A is a nullable variable.
2. If $A \rightarrow B_1B_2 \dots B_n$ is a production in P , and if B_1, B_2, \dots, B_n are nullable variables, then A is also a nullable variable.
3. The variables for which there are productions of the form shown in steps 1 and 2 are nullable variables.

Even though a grammar G has some ϵ -productions, the language may not derive a language containing empty string. So, in such cases, the ϵ -productions or NULL productions are not needed and they can be eliminated.

❖ **Theorem 6.3:** Let $G = (V, T, P, S)$ where $L(G) \neq \epsilon$. We can effectively find an equivalent grammar G^1 with no ϵ -productions such that $L(G^1) = L(G) - \epsilon$.

Proof: The grammar G^1 can be obtained from G in two steps.

Step 1: Find the set of nullable variables in the grammar G using the following algorithm.

```

ov =  $\emptyset$ 
nv = {A | A  $\rightarrow \epsilon$ }
while (ov != nv)
{
    ov = nv
    nv = ov  $\cup$  {A | A  $\rightarrow \alpha$  and  $\alpha \in ov^*$ }
}
V = ov

```

Once the control comes out of the while loop, the set V contains only the nullable variables.

Step 2: Construction of productions P^1 . Consider a production of the form

$$A \rightarrow X_1X_2X_3 \dots X_n, \quad n \geq 1$$

where each X_i is in $(V \cup T)$. In a production, take all possible combinations of nullable variables and replace the nullable variables with ϵ one by one and add the resulting productions to P^1 . If the given production is not an ϵ -production, add it to P^1 .

Suppose, A and B are nullable variables in the production, then

1. First add the production to P^1 .
2. Replace A with ϵ in the given production and add the resulting production to P^1 .
3. Replace B with ϵ in the given production and add the resulting production to P^1 .
4. Replace A and B with ϵ and add the resulting production to P^1 .
5. If all symbols on right side of production are nullable variables, the resulting production is an ϵ -production and do not add this to P^1 .

Thus, the resulting grammar G^1 obtained, generates the same language as generated by G without ϵ and the proof is straight forward.

■ Example 6.7: Eliminate all ϵ -productions from the grammar

$$S \rightarrow ABCa \mid bD$$

$$A \rightarrow BC \mid b$$

$$B \rightarrow b \mid \epsilon$$

$$C \rightarrow c \mid \epsilon$$

$$D \rightarrow d$$

Step 1: Obtain the set of nullable variables from the grammar. This can be done using step 1 of Theorem 6.3 as shown below:

ov	nv	Productions
\emptyset	B,C	$B \rightarrow \epsilon$ $C \rightarrow \epsilon$
B,C	B,C,A	$A \rightarrow BC$
B,C,A	B,C,A	–

$V = \{B,C,A\}$ are all nullable variables.

Step 2: Construction of productions P^1 .

Productions	Resulting productions (P^1)
$S \rightarrow ABCa$	$S \rightarrow ABCa \mid BCa \mid ACa \mid ABa \mid Ca \mid Aa \mid Ba \mid a$
$S \rightarrow bD$	$S \rightarrow bD$

$A \rightarrow BC \mid b$	$A \rightarrow BC \mid B \mid C \mid b$
$B \rightarrow b \mid \epsilon$	$B \rightarrow b$
$C \rightarrow c \mid \epsilon$	$C \rightarrow c$
$D \rightarrow d$	$D \rightarrow d$

The grammar $G^1 = (V^1, T^1, P^1, S)$ where

$$V^1 = \{S, A, B, C, D\}$$

$$T^1 = \{a, b, c, d\}$$

$$P^1 = \{$$

$$S \rightarrow ABCa \mid BCa \mid ACa \mid ABa \mid Ca \mid Aa \mid Ba \mid a \mid bD$$

$$A \rightarrow BC \mid B \mid C \mid b$$

$$B \rightarrow b$$

$$C \rightarrow c$$

$$D \rightarrow d$$

}

S is the start symbol

I Example 6.8: Eliminate all ϵ -productions from the grammar

$$S \rightarrow BAAB$$

$$A \rightarrow 0A2 \mid 2A0 \mid \epsilon$$

$$B \rightarrow AB \mid 1B \mid \epsilon$$

Step 1: Obtain the set of nullable variables from the grammar. This can be achieved using step 1 of Theorem 6.3 as shown below:

ov	nv	Productions
\emptyset	A, B	$A \rightarrow \epsilon$ $B \rightarrow \epsilon$
A, B	A, B, S	$S \rightarrow BAAB$
A, B, S	A, B, S	-

$V = \{S, A, B\}$ are all nullable variables.

Step 2: Construction of productions P^1 . Add a non ϵ -production in P to P^1 . Take all the combinations of nullable variables in a production, delete subset of nullable variables one by one and add the resulting productions to P^1 .

Productions	Resulting productions (P^1)
$S \rightarrow BAAB$	$S \rightarrow BAAB AAB BAB BAA AB BB BA AA A B$
$A \rightarrow 0A2$	$A \rightarrow 0A2 02$
$A \rightarrow 2A0$	$A \rightarrow 2A0 20$
$B \rightarrow AB$	$B \rightarrow AB B A$
$B \rightarrow 1B$	$B \rightarrow 1B 1$

We can delete the productions of the form $A \rightarrow A$. In P^1 , the production $B \rightarrow B$ can be deleted and the final grammar obtained after eliminating ϵ -productions is shown below:

The grammar $G^1 = (V^1, T^1, P^1, S)$ where

$$V^1 = \{S, A, B\}$$

$$T^1 = \{0, 1, 2\}$$

$$P^1 = \{$$

$$S \rightarrow BAAB | AAB | BAB | BAA | AB | BB | BA | AA | A | B$$

$$A \rightarrow 0A2 | 02 | 2A0 | 20$$

$$B \rightarrow AB | A | 1B | 1$$

}

S is the start symbol

6.5. Eliminating Unit Productions

Consider the productions $A \rightarrow B$. The left hand side of the production and right hand side of the production contains only one variable. Such productions are called unit productions. Formally, a unit production is defined as follows.

❖ **Definition:** Let $G = (V, T, P, S)$ be a CFG. Any production in G of the form

$$A \rightarrow B$$

where $A, B \in V$ is a unit-production.

In any grammar, the unit productions are undesirable. This is because one variable is simply replaced by another variable. Consider the productions

$$A \rightarrow B$$

$$B \rightarrow aB \mid b$$

In this example,

$$B \rightarrow aB$$

$$B \rightarrow b$$

are non unit productions. Since B is generated from A, whatever is generated by B, the same things can be generated from A also. So, we can have

$$A \rightarrow aB$$

$$A \rightarrow b$$

and the production $A \rightarrow B$ can be deleted.

❖ **Theorem 6.4:** Let $G = (V, T, P, S)$ be a CFG and has unit productions and no ϵ -productions. An equivalent grammar G_1 without unit productions can be obtained such that $L(G) = L(G_1)$ i.e., any language generated by G is also generated by G_1 . But, the grammar G_1 has no unit productions.

A unit production in grammar G can be eliminated using the following steps:

1. Remove all the productions of the form $A \rightarrow A$.
2. Add all non unit productions to P_1 .
3. For each variable A find all variables B such that

$$A \Rightarrow B$$

i.e., in the derivation process from A, if we encounter only one variable in a sentential form say B (no terminals should be there), obtain all such variables.

4. Obtain a dependency graph. For example, if we have the productions

$$A \rightarrow B$$

$$B \rightarrow C$$

$$C \rightarrow B$$

the dependency graph will be of the form



5. Note from the dependency graph that

- a. $A \xrightarrow{*} B$ i.e., B can be obtained from A

So, all non-unit productions generated from B can also be generated from A

- b. $A \xrightarrow{*} C$ i.e., C can be obtained from A

So, all non-unit productions generated from C can also be generated from A

- c. $B \xrightarrow{*} C$ i.e., C can be obtained from B

So, all non-unit productions generated from C can also be generated from B

- d. $C \xrightarrow{*} B$ i.e., B can be obtained from C

So, all non-unit productions generated from B can also be generated from C

6. Finally, the unit productions can be deleted from the grammar G.

7. The resulting grammar G_1 , generates the same language as accepted by G.

■ Example 6.9: Eliminate all unit productions from the grammar

$$S \rightarrow AB$$

$$A \rightarrow a$$

$$B \rightarrow C \mid b$$

$$C \rightarrow D$$

$$D \rightarrow E \mid bC$$

$$E \rightarrow d \mid Ab$$

The non unit productions of the grammar G are shown below:

$$S \rightarrow AB$$

$$A \rightarrow a$$

$$B \rightarrow b$$

$$D \rightarrow bC$$

$$E \rightarrow d \mid Ab$$

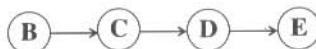
The unit productions of the grammar G are shown below:

$$B \rightarrow C$$

$$C \rightarrow D$$

$$D \rightarrow E$$

The dependency graph for the unit-productions is shown below:



It is clear from the dependency graph that $D \Rightarrow E$. So, all non unit productions generated from E can also be generated from D. The non unit productions from E are

$$E \rightarrow d \mid Ab \quad (6.2)$$

can also be obtained from D

$$D \rightarrow d \mid Ab$$

The resulting D productions are

$$D \rightarrow bC \quad (\text{from 6.1})$$

$$D \rightarrow d \mid Ab \quad (6.3)$$

From the dependency graph it is clear that, $C \Rightarrow E$. So, the non unit productions from E shown in (6.2) can be generated from C. Therefore,

$$C \rightarrow d \mid Ab$$

From the dependency graph it is clear that, $C \Rightarrow D$. So, the non unit productions from D shown in (6.3) can be generated from C. Therefore,

$$C \rightarrow bC$$

$$C \rightarrow d \mid Ab \quad (6.4)$$

From the dependency graph it is clear that $B \Rightarrow C$, $B \Rightarrow D$, $D \Rightarrow E$. So, all the productions obtained from B can be obtained using (6.1), (6.2), (6.3) and (6.4) and the resulting productions are:

$$B \rightarrow b$$

$$B \rightarrow d \mid Ab$$

$$B \rightarrow bC \quad (6.5)$$

The final grammar obtained after eliminating unit productions can be obtained by combining the productions (6.1), (6.2), (6.3), (6.4) and (6.5) and is shown below:

$$V^1 = \{S, A, B, C, D, E\}$$

$$T^1 = \{a, b, d\}$$

$$\begin{aligned}
 P^1 = \{ & \\
 S \rightarrow AB & \\
 A \rightarrow a & \\
 B \rightarrow b \mid d \mid Ab \mid bC & \\
 C \rightarrow bC \mid d \mid Ab & \\
 D \rightarrow bC \mid d \mid Ab & \\
 E \rightarrow d \mid Ab & \\
 \} &
 \end{aligned}$$

S is the start symbol

■ **Example 6.10:** Eliminate unit productions from the grammar

$$\begin{aligned}
 S \rightarrow A0 \mid B & \\
 B \rightarrow A \mid 11 & \\
 A \rightarrow 0 \mid 12 \mid B &
 \end{aligned}$$

The dependency graph for the unit productions

$$\begin{aligned}
 S \rightarrow B & \\
 B \rightarrow A & \\
 A \rightarrow B &
 \end{aligned}$$

is shown below.



The non unit productions are

$$\begin{aligned}
 S \rightarrow A0 & \\
 B \rightarrow 11 & \\
 A \rightarrow 0 \mid 12 & \\
 \end{aligned} \tag{6.6}$$

It is clear from the dependency graph that $S \Rightarrow B$, $S \Rightarrow A$, $B \Rightarrow A$ and $A \Rightarrow B$. So, the new productions from S , A and B are

$$\begin{aligned}
 S \rightarrow 11 \mid 0 \mid 12 & \\
 B \rightarrow 0 \mid 12 & \\
 A \rightarrow 11 & \\
 \end{aligned} \tag{6.7}$$

The resulting grammar without unit productions can be obtained by combining (6.6) and (6.7) and is shown below:

$$V^t = \{S, A, B\}$$

$$T^t = \{0, 1, 2\}$$

$$P^t = \{$$

$$S \rightarrow A0 \mid 11 \mid 0 \mid 12$$

$$A \rightarrow 0 \mid 12 \mid 11$$

$$B \rightarrow 11 \mid 0 \mid 12$$

}

S is the start symbol

■ Example 6.11: Eliminate unit productions from the grammar

$$S \rightarrow Aa \mid B \mid Ca$$

$$B \rightarrow aB \mid b$$

$$C \rightarrow Db \mid D$$

$$D \rightarrow E \mid d$$

$$E \rightarrow ab$$

The dependency graph for the unit productions

$$S \rightarrow B$$

$$C \rightarrow D$$

$$D \rightarrow E$$

is shown below:



The non unit productions are

$$S \rightarrow Aa \mid Ca$$

$$B \rightarrow aB \mid b$$

$$C \rightarrow Db$$

$$D \rightarrow d$$

$$E \rightarrow ab$$

(6.8)

It is clear from the first dependency graph that $S \Rightarrow^* B$ and so whatever is derivable from B it is also derivable from S and the resulting S-productions are:

$$S \rightarrow aB \mid b \quad (6.9)$$

It is clear from the second dependency graph $C \Rightarrow^* D$ and $D \Rightarrow^* E$. So, whatever is derivable from E is also derivable from D and the resulting D-productions are:

$$D \rightarrow ab \mid d \quad (6.10)$$

and the D productions are also derivable from C since $C \Rightarrow^* D$. So, the resulting C-productions are:

$$C \rightarrow Db \mid ab \mid d \quad (6.11)$$

The resulting grammar without unit productions can be obtained by combining (6.8), (6.9), (6.10) and (6.11) and is shown below:

$$V^I = \{S, A, B, C, D, E\}$$

$$T^I = \{a, b, d\}$$

$$P^I = \{$$

$$S \rightarrow Aa \mid Ca \mid aB \mid b$$

$$B \rightarrow aB \mid b$$

$$C \rightarrow Db \mid ab \mid d$$

$$D \rightarrow d \mid ab$$

$$E \rightarrow ab$$

}

S is the start symbol

Note: Given any grammar, all undesirable productions can be eliminated by removing

1. ϵ -productions using Theorem 6.3
2. unit productions using Theorem 6.4
3. useless symbols and productions using Theorem 6.2

in sequence. The final grammar obtained does not have any undesirable productions.

6.6. Chomsky Normal Form

In a CFG, there is no restriction on the right hand side of a production. The restrictions are imposed on the right hand side of productions in a CFG resulting in various normal forms. The different normal forms that we discuss are:

1. Chomsky Normal Form (CNF)
2. Greiback Normal Form (GNF)

Chomsky normal form can be defined as follows.

❖ **Definition:** Let $G = (V, T, P, S)$ be a CFG. The grammar G is said to be in CNF if all productions are of the form

$$A \rightarrow BC$$

or

$$A \rightarrow a$$

where A, B and $C \in V$ and $a \in T$.

Note that if a grammar is in CNF, the right hand side of the production should contain two symbols or one symbol. If there are two symbols on the right hand side those two symbols must be non-terminals and if there is only one symbol, that symbol must be a terminal.

❖ **Theorem 6.5:** Let $G = (V, T, P, S)$ be a CFG which generates context free language without ϵ . We can find an equivalent context free grammar $G^1 = (V^1, T, P^1, S)$ in CNF such that

$$L(G) = L(G^1)$$

i.e., all productions in G^1 are of the form

$$A \rightarrow BC$$

or

$$A \rightarrow a$$

Proof: Let the grammar G has no ϵ -productions and unit productions. The grammar G^1 can be obtained using the following steps.

Step 1: Consider the productions of the form

$$A \rightarrow X_1 X_2 X_3 \dots X_n$$

where $n \geq 2$ and each $X_i \in (V \cup T)$ i.e., consider the productions having more than two symbols on the right hand side of the production. If X is a terminal say a , then replace this terminal by a corresponding non-terminal B_a and introduce the production

$$B_a \rightarrow a$$

The non-terminals on the right hand side of the production are retained. The resulting productions are added to P_1 . The resulting context free grammar $G_1 = (V_1, T, P_1, S)$ where each production in P_1 is of the form

$$A \rightarrow A_1 A_2 \dots A_n$$

or

$$A \rightarrow a$$

generates the same language as accepted by grammar G . So, $L(G) = L(G_1)$.

Step 2: Restrict the number of variables on the right hand side of the production. Add all the productions of G_1 which are in CNF to P^1 . Consider a production of the form

$$A \rightarrow A_1 A_2 \dots A_n$$

where $n \geq 3$ (Note that if $n = 2$, the production is already in CNF and n can not be equal to 1. Because if $n = 1$, there is only one symbol and it is a terminal which again is in CNF). The A -production can be written as

$$A \rightarrow A_1 D_1$$

$$D_1 \rightarrow A_2 D_2$$

$$D_2 \rightarrow A_3 D_3$$

⋮

$$D_{n-2} \rightarrow A_{n-1} A_n$$

These productions are added to P^1 and new variables are added to V^1 . The grammar thus obtained is in CNF. The resulting grammar $G^1 = (V^1, T, P^1, S)$ generates the same language as accepted by G i.e. $L(G) = L(G^1)$.

■ **Example 6.12:** Consider the grammar

$$S \rightarrow 0A \mid 1B$$

$$A \rightarrow 0AA \mid 1S \mid 1$$

$$B \rightarrow 1BB \mid 0S \mid 0$$

Obtain the Grammar in CNF

All productions which are in CNF are added to P_1 . The productions which are in standard form and added to P_1 are:

$$\begin{aligned} A &\rightarrow 1 \\ B &\rightarrow 0 \end{aligned} \tag{6.8}$$

Consider the productions, which are not in CNF. Replace the terminal a on right hand side of the production by a non-terminal A and introduce the production $A \rightarrow a$. This step has to be carried out for each production which are not in CNF.

The table below shows the action taken indicating which terminal is replaced by the corresponding non-terminal and what is the new production introduced. The last column shows the resulting productions.

Given Productions	Action	Resulting productions
$S \rightarrow 0A \mid 1B$	Replace 0 by B_0 and introduce the production $B_0 \rightarrow 0$ Replace 1 by B_1 and introduce the production $B_1 \rightarrow 1$	$S \rightarrow B_0A \mid B_1B$ $B_0 \rightarrow 0$ $B_1 \rightarrow 1$
$A \rightarrow 0AA \mid 1S$	Replace 0 by B_0 and introduce the production $B_0 \rightarrow 0$ Replace 1 by B_1 and introduce the production $B_1 \rightarrow 1$	$A \rightarrow B_0AA \mid B_1S$ $B_0 \rightarrow 0$ $B_1 \rightarrow 1$
$B \rightarrow 1BB \mid 0S$	Replace 0 by B_0 and introduce the production $B_0 \rightarrow 0$ Replace 1 by B_1 and introduce the production $B_1 \rightarrow 1$	$B \rightarrow B_1BB \mid B_0S$ $B_1 \rightarrow 1$ $B_0 \rightarrow 0$

The grammar $G_1 = (V_1, T, P_1, S)$ can be obtained by combining the productions obtained from the last column in the table and the productions shown in (6.8).

$$V_1 = \{S, A, B, B_0, B_1\}$$

$$T_1 = \{0, 1\}$$

$$P_1 = \{$$

$$S \rightarrow B_0A \mid B_1B$$

$$A \rightarrow B_0AA \mid B_1S \mid 1$$

$$B \rightarrow B_1BB \mid B_0S \mid 0$$

$$B_0 \rightarrow 0$$

$$B_1 \rightarrow 1$$

}

S is the start symbol

Step 2: Restricting the number of variables on the right hand side of the production to 2. The productions obtained after step 1 are:

$$S \rightarrow B_0A \mid B_1B$$

$$A \rightarrow B_0AA \mid B_1S \mid 1$$

$$B \rightarrow B_1BB \mid B_0S \mid 0$$

$$B_0 \rightarrow 0$$

$$B_1 \rightarrow 1$$

In the above productions, the productions which are in CNF are

$$S \rightarrow B_0A \mid B_1B$$

$$A \rightarrow B_1S \mid 1$$

$$B \rightarrow B_0S \mid 0$$

$$B_0 \rightarrow 0$$

$$B_1 \rightarrow 1$$

(6.9)

and add these productions to P^1 . The productions which are not in CNF are

$$A \rightarrow B_0AA$$

$$B \rightarrow B_1BB$$

The following table shows how these productions are changed to CNF so that only two variables are present on the right hand side of the production.

Given productions	Action	Resulting productions	
$A \rightarrow B_0AA$	Replace AA on R.H.S with variable D_1 and introduce the production $D_1 \rightarrow AA$	$A \rightarrow B_0D_1$ $D_1 \rightarrow AA$	
$B \rightarrow B_1BB$	Replace BB on R.H.S with variable D_2 and introduce the production $D_2 \rightarrow BB$	$B \rightarrow B_1D_2$ $D_2 \rightarrow BB$	(6.10)

The final grammar which is in CNF can be obtained by combining the productions in (6.9) and (6.10). The grammar $G^1 = (V^1, T, P^1, S)$ is in CNF where

$$V^1 = \{S, A, B, B_0, B_1, D_1, D_2\}$$

$$T^1 = \{0, 1\}$$

$$P^1 = \{$$

$$S \rightarrow B_0A \mid B_1B$$

$$A \rightarrow B_1S \mid 1 \mid B_0D_1$$

$$B \rightarrow B_0S \mid 0 \mid B_1D_2$$

$$B_0 \rightarrow 0$$

$$B_1 \rightarrow 1$$

$$D_1 \rightarrow AA$$

$$D_2 \rightarrow BB$$

}

S is the start symbol

6.7. Greibach Normal Form (GNF)

In CNF there is restriction on the number of symbols on the right hand side of the production. Note that in CNF not more than two symbols on R.H.S of the production are permitted. If there is only symbol that symbol must be a terminal and if there are two symbols, those two symbols must be variables.

In GNF there is no restriction on the number of symbols on the right hand side, but there is restriction on the terminals and variables appear on the right hand side of the production.

❖ **Definition:** Let $G = (V, T, P, S)$ be a CFG. The CFG G is said to be in GNF if all the productions are of the form

$$A \rightarrow a\alpha$$

where $a \in T$ and $\alpha \in V^*$, i.e., the first symbol on the right hand side of the production must be a terminal and it can be followed by zero or more variables.

❖ **Theorem 6.6:** Let $G = (V, T, P, S)$ be a CFG generating the language L without ϵ . An equivalent grammar G_1 generating the same language exists for which every production is of the form

$$A \rightarrow a\alpha$$

where A is a variable, a is a terminal and α is string of zero or more variables.

It means that any CFG can be converted into GNF notation. GNF is a very useful notation. If a grammar is in GNF, it can be easily converted into Pushdown Automaton which can accept only the context free languages. The Pushdown Automaton will be discussed in the next chapter. Now, let us see how to convert a given CFG to its equivalent GNF notation.

Procedure to obtain the grammar in GNF:

Step 1: Obtain the grammar in CNF.

Step 2: Rename the non-terminals to A_1, A_2, A_3, \dots

Step 3: Using substitution method as discussed in Section 6.1, obtain the productions to the form

$$A_i \rightarrow A_j \alpha \quad \text{for } i < j$$

where $\alpha \in V^*$. Note if all the productions are in this manner, the number of steps will be reduced while converting.

Step 4: After substitution, if a grammar has left-recursion, we should eliminate left recursion as discussed in Section 6.2.

Step 5: It may be necessary to apply step 3 and/or step 4 more than once to get the grammar in GNF.

Now, let us concentrate on how a grammar can be converted into GNF.

|| **Example 6.13:** Convert the following grammar

$$S \rightarrow AB1 \mid 0$$

$$A \rightarrow 00A \mid B$$

$B \rightarrow 1A1$

into GNF.

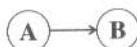
Note: A grammar should not have any unit productions and ϵ -productions. In case if the grammar has ϵ -productions and unit productions perform the following operations one after the other:

1. Eliminate all ϵ -productions.
2. Eliminate all unit productions.
3. Obtain the grammar in CNF.
4. Finally convert the grammar into GNF.

Since the grammar does not have any ϵ -productions, in the next let us eliminate the unit production

$A \rightarrow B$

The dependency graph for this can be



It is clear from the dependency graph that $A \Rightarrow B$. So, all the symbols derivable from B are also derivable from A. So, in the production

$A \rightarrow 00A \mid B$

the variable B can be replaced by the string 1A1 using the production

$B \rightarrow 1A1$

Now, the A-production can be written as

$A \rightarrow 00A \mid 1A1$

The resulting grammar obtained after eliminating unit productions is shown below:

$S \rightarrow AB1 \mid 0$

$A \rightarrow 00A \mid 1A1$

$B \rightarrow 1A1$

Now, the grammar has to be converted into CNF. Now replace the terminals by non-terminals if they are not in CNF and the resulting grammar is

$$S \rightarrow ABA_1 \mid 0$$

$$A \rightarrow A_0A_0A \mid A_1AA_1$$

$$B \rightarrow A_1AA_1$$

$$A_1 \rightarrow 1$$

$$A_0 \rightarrow 0$$

Now restrict the number of variables on the right hand side of the production to two and the resulting grammar in CNF notation is

$$S \rightarrow AD_1 \mid 0$$

$$A \rightarrow A_0D_2 \mid A_1D_3$$

$$B \rightarrow A_1D_3$$

$$A_1 \rightarrow 1$$

$$A_0 \rightarrow 0$$

$$D_1 \rightarrow BA_1$$

$$D_2 \rightarrow A_0A$$

$$D_3 \rightarrow AA_1$$

Now let us rename all the variables as shown below:

Let $S = A_1$, $A = A_2$, $B = A_3$, $A_0 = A_4$, $A_1 = A_5$, $D_1 = A_6$, $D_2 = A_7$, $D_3 = A_8$. Now, the grammar can be re-written as

$$A_1 \rightarrow A_2A_6 \mid 0$$

$$A_2 \rightarrow A_4A_7 \mid A_5A_8$$

$$A_3 \rightarrow A_5A_8$$

$$A_5 \rightarrow 1$$

$$A_4 \rightarrow 0$$

$$A_6 \rightarrow A_3A_5$$

$$A_7 \rightarrow A_4A_2$$

$$A_8 \rightarrow A_2A_5$$

In the above productions, note that A_4 and A_5 -productions are in GNF.

Consider A_3 production: Substituting for A_5 in A_3 -production we get

$$A_3 \rightarrow A_5A_8 = 1A_8$$

Now, A_3 -production is in GNF.

Consider A₂ production: Since A₄ and A₅ productions are in GNF, substituting these productions in A₂-production we get

$$A_2 \rightarrow A_4 A_7 | A_5 A_8 = 0A_7 | 1A_8$$

Now, A₂-production is also in GNF.

Consider A₁ production: Since A₂ production is in GNF, substituting for A₂ in A₁-production we get

$$A_1 \rightarrow A_2 A_6 | 0 = (0A_7 | 1A_8)A_6 | 0 = 0A_7A_6 | 1A_8A_6 | 0$$

Now, A₁ production is also in GNF.

Consider A₆-production: Now, in A₆-production, replacing the first A₃ by A₃-production

$$A_6 \rightarrow A_3 A_5 = (A_5 A_8) A_5$$

we get the A₆ production

$$A_6 \rightarrow A_5 A_8 A_5$$

which is not in desired form. In the above production, replacing the first A₅ by A₅-production, we get A₆-production in GNF as shown below:

$$A_6 \rightarrow A_5 A_8 A_5 = 1A_8 A_5$$

Consider A₇ production: Replacing the first A₄ in A₇-production we get

$$A_7 \rightarrow 0A_2$$

which is in GNF.

Consider A₈ production: Since A₂ production is in GNF, substituting for A₂ in A₈-production we get

$$A_8 \rightarrow A_2 A_5 = (0A_7 | 1A_8) A_5 = 0A_7 A_5 | 1A_8 A_5$$

which is in GNF. Since all productions are in GNF, the whole grammar is in GNF. The final grammar which is in GNF is G = (V, T, P, S) where

$$V = \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$$

$$T = \{0, 1\}$$

$$P = \{$$

$$A_1 \rightarrow 0A_7 A_6 | 1A_8 A_6 | 0$$

$$\begin{aligned}
 A_2 &\rightarrow 0A_7 \mid 1A_8 \\
 A_3 &\rightarrow 1A_8 \\
 A_4 &\rightarrow 0 \\
 A_5 &\rightarrow 1 \\
 A_6 &\rightarrow 1A_8A_5 \\
 A_7 &\rightarrow 0A_2 \\
 A_8 &\rightarrow 0A_7A_5 \mid 1A_8A_5 \\
 \}
 \end{aligned}$$

S is the start symbol

Note: In this problem there are only substitutions and after repeated substitutions, we get the grammar in GNF.

■ **Example 6.14:** Convert the following grammar

$$\begin{aligned}
 A &\rightarrow BC \\
 B &\rightarrow CA \mid b \\
 C &\rightarrow AB \mid a
 \end{aligned}$$

into GNF.

Let $A = A_1$, $B = A_2$, $C = A_3$ and the resulting grammar is

$$\begin{aligned}
 A_1 &\rightarrow A_2A_3 \\
 A_2 &\rightarrow A_3A_1 \mid b \\
 A_3 &\rightarrow A_1A_2 \mid a
 \end{aligned}$$

First two productions are of the form

$$A_i \rightarrow A_j\alpha \quad \text{for } i < j$$

So, we consider A_3 -production.

Consider A_3 -production: Substituting for A_1 in A_3 -production we get

$$A_3 \rightarrow A_1A_2 \mid a = (A_2A_3)A_2 \mid a = A_2A_3A_2 \mid a$$

Again replacing the first A_2 in A_3 -production we get,

$$\begin{aligned}
 A_3 &\rightarrow A_2A_3A_2 \mid a = (A_3A_1 \mid b)A_3A_2 \mid a \\
 &= A_3A_1A_3A_2 \mid bA_3A_2 \mid a
 \end{aligned}$$

we get the resulting A_3 -production as

$$A_3 \rightarrow A_3 A_1 A_3 A_2 | b A_3 A_2 | a$$

which is having left recursion. After eliminating left recursion we get

$$A_3 \rightarrow b A_3 A_2 | a | b A_3 A_2 Z | a Z$$

$$Z \rightarrow A_1 A_3 A_2 | A_1 A_3 A_2 Z$$

Now, all A_3 -productions are in GNF.

Consider A_2 -productions: Since all A_3 -productions are in GNF, substituting A_3 -productions in A_2 -production we get

$$\begin{aligned} A_2 &\rightarrow (b A_3 A_2 | a | b A_3 A_2 Z | a Z) A_1 | b \\ &= b A_3 A_2 A_1 | a A_1 | b A_3 A_2 Z A_1 | a Z A_1 | b \end{aligned}$$

which is in GNF. Now, all A_2 -productions are in GNF.

Consider A_1 -productions: Since all A_2 -productions are in GNF, substituting A_2 -productions in A_1 -production we get,

$$\begin{aligned} A_1 &\rightarrow A_2 A_3 = (b A_3 A_2 A_1 | a A_1 | b A_3 A_2 Z A_1 | a Z A_1 | b) A_3 \\ &= b A_3 A_2 A_1 A_3 | a A_1 A_3 | b A_3 A_2 Z A_1 A_3 | a Z A_1 A_3 | b A_3 \end{aligned}$$

Now, A_1 -productions are also in GNF.

Consider Z -productions: Since A_1 -productions are in GNF, substituting A_1 -production in Z -production we get Z -productions also in GNF as shown below:

$$\begin{aligned} Z &\rightarrow A_1 A_3 A_2 | A_1 A_3 A_2 Z \\ &= (b A_3 A_2 A_1 A_3 | a A_1 A_3 | b A_3 A_2 Z A_1 A_3 | a Z A_1 A_3 | b A_3) A_3 A_2 \\ &\quad (b A_3 A_2 A_1 A_3 | a A_1 A_3 | b A_3 A_2 Z A_1 A_3 | a Z A_1 A_3 | b A_3) A_3 A_2 Z \end{aligned}$$

which can be written as

$$Z \rightarrow b A_3 A_2 A_1 A_3 A_3 A_2 | a A_1 A_3 A_3 A_2 | b A_3 A_2 Z A_1 A_3 A_3 A_2 | a Z A_1 A_3 A_3 A_2 | b A_3 A_3 A_2$$

$$Z \rightarrow b A_3 A_2 A_1 A_3 A_3 A_2 Z | a A_1 A_3 A_3 A_2 Z | b A_3 A_2 Z A_1 A_3 A_3 A_2 Z | a Z A_1 A_3 A_3 A_2 Z | b A_3 A_3 A_2 Z$$

Since all productions are in GNF, the resulting grammar is also in GNF. So, the final grammar obtained in GNF notation is

$$G = (V, T, P, S)$$

where

$$V = \{A_1, A_2, A_3, Z\}$$

$$T = \{a, b\}$$

$$P = \{$$

$$A_1 \rightarrow bA_3A_2A_1A_3 \mid aA_1A_3 \mid bA_3A_2ZA_1A_3 \mid aZA_1A_3 \mid bA_3$$

$$A_2 \rightarrow bA_3A_2A_1 \mid aA_1 \mid bA_3A_2ZA_1 \mid aZA_1 \mid b$$

$$A_3 \rightarrow bA_3A_2 \mid a \mid bA_3A_2Z \mid aZ$$

$$Z \rightarrow bA_3A_2A_1A_3A_3A_2 \mid aA_1A_3A_3A_2 \mid bA_3A_2ZA_1A_3A_3A_2 \mid aZA_1A_3A_3A_2 \mid bA_3A_3A_2$$

$$Z \rightarrow bA_3A_2A_1A_3A_3A_2Z \mid aA_1A_3A_3A_2Z \mid bA_3A_2ZA_1A_3A_3A_2Z \mid aZA_1A_3A_3A_2Z \mid bA_3A_3A_2Z$$

}

A_1 is the start symbol

■ **Example 6.15:** Convert the following grammar

$$S \rightarrow AA \mid 0$$

$$A \rightarrow SS \mid 1$$

into GNF.

Let $S = A_1$ and $A = A_2$. After substitution, the resulting grammar obtained is shown below:

$$A_1 \rightarrow A_2 A_2 \mid 0$$

$$A_2 \rightarrow A_1 A_1 \mid 1$$

First production is of the form

$$A_i \rightarrow A_j \alpha \quad \text{for } i < j$$

So, we consider A_2 -production.

Consider A_2 -production: Substituting for A_1 in A_2 -production we get

$$A_2 \rightarrow A_2 A_2 A_1 \mid 0A_1 \mid 1$$

The above grammar is having left recursion. After eliminating left recursion we get

$$A_2 \rightarrow 0A_1 \mid 1 \mid 0A_1Z \mid 1Z$$

$$Z \rightarrow A_2A_1 \mid A_2A_1Z$$

Now, all A_2 -productions are in GNF.

Consider A₁-productions: Since all A₂-productions are in GNF, substituting A₂-productions in A₁-production we get

$$A_1 \rightarrow 0A_1A_2 \mid 1A_2 \mid 0A_1ZA_2 \mid 1ZA_2 \mid 0$$

which is in GNF. Now, all A₁-productions are in GNF.

Consider Z-productions: Since A₂-productions are in GNF, substituting A₂-production in Z-production we get Z-productions also in GNF as shown below:

$$Z \rightarrow 0A_1A_1 \mid 1A_1 \mid 0A_1ZA_1 \mid 1ZA_1$$

$$Z \rightarrow 0A_1A_1Z \mid 1A_1Z \mid 0A_1ZA_1Z \mid 1ZA_1Z$$

So, the final grammar obtained which is in GNF is shown below:

$$A_1 \rightarrow 0A_1A_2 \mid 1A_2 \mid 0A_1ZA_2 \mid 1ZA_2 \mid 0$$

$$A_2 \rightarrow 0A_1 \mid 1 \mid 0A_1Z \mid 1Z$$

$$Z \rightarrow 0A_1A_1 \mid 1A_1 \mid 0A_1ZA_1 \mid 1ZA_1$$

$$Z \rightarrow 0A_1A_1Z \mid 1A_1Z \mid 0A_1ZA_1Z \mid 1ZA_1Z$$

Exercises

1. Is the following grammar ambiguous?

$$S \rightarrow aSb \mid SS \mid \epsilon$$

2. Show that the following grammar is ambiguous

$$S \rightarrow SbS \mid a$$

3. Let G = (V, T, P, S) be a CFG, where

$$V = \{E, I\}$$

$$T = \{a, b, c, d, +, *, (,)\}$$

$$P = \{$$

$$E \rightarrow I \mid E+E \mid E^*E \mid (E)$$

$$I \rightarrow a \mid b \mid c \mid d$$

$$\}$$

Obtain the derivations for the strings (a+b)*c*d and a+b*c and the parse tree for each derivation.

4. Obtain a reduced grammar for the grammar shown below:

$S \rightarrow aAa$

$A \rightarrow Sb \mid bCC \mid aDA$

$C \rightarrow ab \mid aD$

$E \rightarrow aC$

$D \rightarrow aAD$

5. Find an equivalent grammar without ϵ -productions for the grammar shown below:

$S \rightarrow aSa \mid bSb \mid A$

$A \rightarrow aBb \mid bBa$

$B \rightarrow aB \mid bB \mid \epsilon$

6. Remove the unit productions from the grammar:

$S \rightarrow A \mid B \mid Cc$

$A \rightarrow aBb \mid B$

$B \rightarrow aB \mid bb$

$C \rightarrow Cc \mid B$

7. Eliminate useless productions from the grammar:

$S \rightarrow aA \mid a \mid B \mid C$

$A \rightarrow aB \mid \epsilon$

$B \rightarrow aA$

$C \rightarrow cCD$

$D \rightarrow abd$

Note: First remove ϵ and unit productions, then simplify CFG.

8. Obtain the following grammar in CNF:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid I$

$I \rightarrow a \mid b \mid c \mid Ia \mid Ib \mid Ic$

9. Obtain the following grammar in CNF and GNF:

$S \rightarrow aA \mid a \mid B \mid C$

$A \rightarrow aB \mid \epsilon$

$B \rightarrow aA$

$C \rightarrow cCD$

$D \rightarrow abd$

10. Simplify the following CFG and convert it into CNF:

$S \rightarrow AaB \mid aaB$

$A \rightarrow \epsilon$

$B \rightarrow bbA \mid \epsilon$

11. Convert the following grammar into GNF:

$S \rightarrow abAB$

$A \rightarrow bAB \mid \epsilon$

$B \rightarrow Aa \mid \epsilon$

12. Explain the method of substitution with example.

13. What is left recursion? How it can be eliminated?

14. Eliminate left recursion from the following grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

15. Eliminate the useless symbols in the grammar:

$S \rightarrow aA \mid bB$

$A \rightarrow aA \mid a$

$B \rightarrow bB$

$D \rightarrow ab \mid Ea$

$E \rightarrow aC \mid d$

16. Eliminate left recursion from the following grammar:

$S \rightarrow Ab \mid a$

$A \rightarrow Ab \mid Sa$

17. What is the need for simplifying a grammar?

18. What is a useless symbol? Explain with example.

19. Simplify the following grammar:

$S \rightarrow aA \mid a \mid Bb \mid cC$

$A \rightarrow aB$

$B \rightarrow a \mid Aa$

$C \rightarrow cCD$

$D \rightarrow ddd$

20. What is an ϵ -production? What is a Nullable variable? Explain with example.

21. Eliminate all ϵ -productions from the grammar:

$S \rightarrow ABCa \mid bD$

$A \rightarrow BC \mid b$

$B \rightarrow b \mid \epsilon$

$C \rightarrow c \mid \epsilon$

$D \rightarrow d$

22. Eliminate all ϵ -productions from the grammar:

$S \rightarrow BAAB$

$A \rightarrow 0A2 \mid 2A0 \mid \epsilon$

$B \rightarrow AB \mid 1B \mid \epsilon$

23. What is an unit production? Give general method of eliminating unit productions from the grammar.

24. Eliminate all unit productions from the grammar

S	\rightarrow	AB
A	\rightarrow	a
B	\rightarrow	$C \mid b$
C	\rightarrow	D
D	\rightarrow	$E \mid bC$
E	\rightarrow	$d \mid Ab$

25. What is Chomsky Normal Form (CNF) and Greiback Normal Form(GNF)?

26. Prove that for every CFG we can have an equivalent grammar using CNF notations where a language does not contain ϵ .

27. Eliminate unit productions from the grammar

$S \rightarrow A0 \mid B$

$$\begin{aligned}B &\rightarrow A \mid 11 \\A &\rightarrow 0 \mid 12 \mid B\end{aligned}$$

28. Eliminate unit productions from the grammar

$$\begin{aligned}S &\rightarrow Aa \mid B \mid Ca \\B &\rightarrow aB \mid b \\C &\rightarrow Db \mid D \\D &\rightarrow E \mid d \\E &\rightarrow ab\end{aligned}$$

29. Consider the grammar in CNF

$$\begin{aligned}S &\rightarrow 0A \mid 1B \\A &\rightarrow 0AA \mid 1S \mid 1 \\B &\rightarrow 1BB \mid 0S \mid 0\end{aligned}$$

30. What is the general procedure to convert a grammar into its equivalent GNF notation?

31. Convert the following grammar into GNF

$$\begin{aligned}S &\rightarrow AB1 \mid 0 \\A &\rightarrow 00A \mid B \\B &\rightarrow 1AI\end{aligned}$$

32. Convert the following grammar into GNF

$$\begin{aligned}A &\rightarrow BC \\B &\rightarrow CA \mid b \\C &\rightarrow AB \mid a\end{aligned}$$

Properties of Context Free Languages

As we have discussed in previous chapters that some of the non-regular languages can be represented using context free grammars from which we can obtain the context free languages. For example, the non-regular language such as $L = \{a^n b^n \mid n \geq 1\}$ can be easily generated using context free grammars. There are so many other instances such as matching parentheses, to match the nested if statements, whether a statement is syntactically correct or not and so on most of which can be easily represented using CFGs. So, it is very important for us to learn the properties of context free languages. The different closure properties covered in this chapter are union, concatenation, star-closure, intersection, complementation etc. We have to remember that just because the regular languages are closed under union, concatenation, * -closure, intersection,

complementation, etc., it is not true that they are also closed under all these operations. This also covers pumping lemma, which is a very useful concept in determining whether the given language is context free or not.

Note:

1. In the derivation process, if a non terminal A occurs in some sentential form starting from the start symbol and if a string of terminals can be derived from this sentential form, then the non terminal A is useful. Otherwise, it is useless.
2. The non-terminal A can be recursive if and only if it can generate a string containing itself. For example, Consider the derivation

$$S \xrightarrow{*} uAy$$

The non terminal A is recursive

- a. If there is a production of the form

$$A \rightarrow x_1Ax_2$$

for some strings $x_1, x_2 \in (V \cup T)^*$ (Direct recursion)

or

- b. If there is a production of the form

$$A \rightarrow \dots X_1 \dots$$

$$X_1 \rightarrow \dots X_2 \dots$$

$$X_2 \rightarrow \dots X_3 \dots$$

$$X_3 \rightarrow \dots X_4 \dots$$

⋮

$$X_n \rightarrow \dots A \dots$$
 (indirect recursion)

Now let us see how to prove that certain languages are not context free similar to the proof as we did for regular languages using Pumping Lemma. The Pumping Lemma for Context Free Languages (CFL) can be stated as follows:

6.8. Pumping Lemma

Pumping Lemma for Context Free Languages: Let L be the context free language and is infinite. Let z is sufficiently long string and $z \in L$ so that $|z| \geq n$ where n is some positive integer. If the string z can be decomposed into combinations of strings

$$z = uvwxy$$

such that $|vwx| \leq n$, $|vx| \geq 1$, then $uv^iwx^i y \in L$ for $i = 0, 1, 2, \dots$

Note: The following observations can be made from the Pumping Lemma.

1. n is the length of the longest string that can be generated by the parse tree where the same non-terminal never occurs twice on the same path through the tree.
2. The string z is sufficiently long so that it can be decomposed into various sub strings u, v, w, x and y in that sequence.
3. The two sub strings v and x are present somewhere in z .
4. The sub string u appears before v , the sub string w is in between v and x and the sub string y appears after x .
5. The string w in between v and x cannot be too long since $|vwx| \leq n$ for some positive integer n .
6. Both the sub strings v and x cannot be empty since $|vx| \geq 1$. One of them can be empty.
7. If all the points mentioned from 1 to 5 are satisfied, and if we duplicate sub string v and x same number of times, the resultant string will definitely be in L and the string $z \in L$ is context free. Otherwise, the string $z \in L$ is not context free.

Proof: According to Pumping Lemma, it is assumed that string $z \in L$ is finite and is context free language. We know that z is string of terminals which is derived by applying series of productions. Proof of this theorem leads to the following two cases.

Case 1: To generate a sufficiently long string z (it is assumed that the string is infinite), one or more variables (non-terminals) must be recursive (Note that infinite string can be generated if the grammar has some non-terminal A such that

$$A \xrightarrow{*} \alpha A \beta$$

for some α and β) and should be applied more than once.

Case 2: $z \in L$ implies that after applying some/all productions some number of times, we get finally string of terminals and the derivation stops. If we can prove these two cases, we have the proof for Pumping Lemma.

Proof of case 1: To generate a sufficiently long string z (it is assumed that the string is infinite), one or more variables (non-terminals) must be recursive. Let us assume that the language is finite and the grammar has a finite number of variables (assume that all are useful variables) and each production has finite length. The only way to derive sufficiently long string using such productions is that the grammar should have one or more recursive variables. Assume that no variable is recursive.

Since no non-terminal (variable) is recursive, each variable must be defined only in terms of terminal(s) and/or other variables. Since those variables are also non-recursive, they have to be defined in terms of terminals and other variables and so on. If we keep applying the productions like this, there are no variables at all in the final derivation and finally we get string of terminals and the generated string is finite. From this we conclude that there is a limit on the length of the

string that is generated from the start symbol S. This contradicts our assumption that the language is finite. Therefore, the assumption that one or more variables are non-recursive is incorrect. It means that one or more variables are recursive and hence the proof.

Proof of case 2: $z \in L$ implies that after applying some/all productions some number of times, we get finally string of terminals and the derivation stops. Let $z \in L$ is sufficiently long string and so the derivation must have involved recursive use of some non-terminal A and the derivations must have the form

$$S \xrightarrow{*} uAy$$

Note that any derivation should start from the start symbol S. Since A is used recursively, the derivation can take the following form:

$$S \xrightarrow{*} uAy \xrightarrow{*} uvAxy$$

and the final derivation should be of the form

$$S \xrightarrow{*} uAy \xrightarrow{*} uvAxy \xrightarrow{*} uvwxy = z$$

It implies that the following derivations

$$A \xrightarrow{*} vAx$$

and

$$A \xrightarrow{*} w$$

are also possible. From this we can easily conclude that the derivation

$$A \xrightarrow{*} vwx$$

must also be possible. Next we have to prove that the longest string vwx is generated without recursion since it is assumed that $|vwx| \leq n$. This can be easily proved since CFG that generates CFL does not contain ϵ -productions or unit productions. It shows that every derivation step either increases the length of the sentential form (using recursive variable) or introduces a terminal. The derivation

$$A \xrightarrow{*} vAx$$

used earlier clearly shows that

$$|vx| \geq 1$$

Note from the derivation

$$S \xrightarrow{*} uAy \xrightarrow{*} uvAxy$$

that $uvAxy$ occurs in the derivation, and

$$A \xrightarrow{*} vAx$$

and

$$A \xrightarrow{*} w$$

are also possible, it follows that

$$uv^iwx^i y \in L$$

and hence the proof.

Applications of Pumping Lemma for CFLs

The Pumping Lemma for CFLs is used to prove that certain languages are not context free languages. Note that Pumping Lemma can not be used to prove that certain languages are context free. In this section let us show that certain languages are not context free using Pumping lemma (similar to the problems by showing that certain languages are non-regular languages).

The general strategy used to prove that a given language is not context free is shown below:

1. Assume that the language L is infinite and it is context free.
2. Select the string say z and break it into sub strings u, v, w, x and y such that $z = uvwxy$ where $|vwx| \leq n$ and $|vx| \geq 1$.
3. Find any i such that $uv^iwx^i y \notin L$. According to pumping lemma, $uv^iwx^i y \in L$. So, the result is a contradiction to the assumption that the language is context free. Therefore, the given language L is not context free.

Example 6.16: Show that $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context free.

Step 1: Let L is context free and is infinite. Let $z = a^n b^n c^n \in L$.

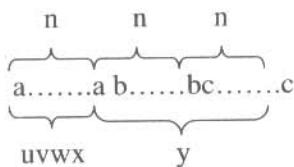
Step 2: Note that $|z| > n$ and so we can split z into $uvwxy$ such that

$$|vwx| \leq n \text{ and } |vx| \geq 1$$

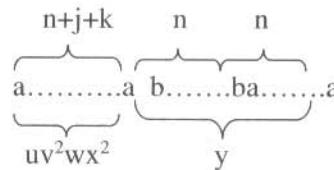
and so $uv^iwx^i y \in L$ for $i = 0, 1, 2, \dots$ (This is according to Pumping Lemma). Let us take the various cases.

Case 1: The string vwx is within a^n .

Let $v = a^j, x = a^k$ where $|vx| = j + k \geq 1$ and $|vwx| \leq n$ which can be shown pictorially as



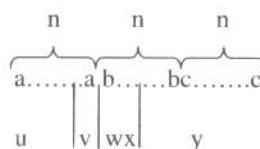
Now, according to Pumping Lemma, $uv^2wx^2y \in L$ for $i = 2$ and the language generated is as shown below:



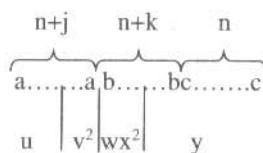
Note that $uv^2wx^2y = a^{n+j+k}b^n c^n \notin L$ when $j+k >= 1$ (Note that the string should have some number of a's followed by equal number of b's and c's). But according to pumping lemma, $uv^2wx^2y \in L$, which is the contradiction.

Case 2: The string vwx is $a^n b^n$.

Let $v = a^j$, $x = b^k$ where $|vx| = j + k \geq 1$ and $|vwx| \leq n$ which can be shown pictorially as



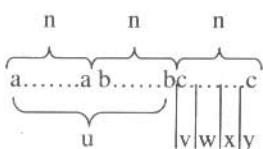
Now, according to Pumping Lemma, $uv^2wx^2y \in L$ for $i = 2$ and the language generated is as shown below:



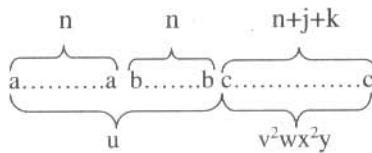
Note that $uv^2wx^2y = a^{n+j}b^{n+k}c^n \notin L$ when $j+k >= 1$ (Note that the string should have some number of a's followed by equal number of b's and c's). But according to pumping lemma, $uv^2wx^2y \in L$, which is the contradiction.

Case 3: The string vwx is within c^n .

Let $v = c^j$, $x = c^k$ where $|vx| = j + k \geq 1$ and $|vwx| \leq n$ which can be shown pictorially as



Now, according to Pumping Lemma, $uv^2wx^2y \in L$ for $i = 2$ and the language generated is as shown below:



Note that $uv^2wx^2y = a^n b^n c^{n+j+k} \notin L$ when $j + k >= 1$ (Note that the string should have some number of a's followed by equal number of b's and c's). But according to pumping lemma, $uv^2wx^2y \in L$, which is the contradiction.

But, according to pumping lemma, n number of a's should be followed by n number of b's which in turn should be followed by n number of c's. In all the three cases we get contradiction to the assumption that the language is context free.

So, the language $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context free.

Example 6.17: Show that $L = \{w \mid w \in \{a,b,c\}^* \text{ where } n_a(w) = n_b(w) = n_c(w)\}$ is not context free.

The language

$$L_1 = \{a^n b^n c^n \mid n \geq 0\}$$

is obtained by the intersection of L and the regular language represented by the regular expression $a^* b^* c^*$, i.e.,

$$\{a^n b^n c^n \mid n \geq 0\} = \{a^* b^* c^* \cap \{w \mid w \in \{a,b,c\}^* \text{ where } n_a(w) = n_b(w) = n_c(w)\}\}$$

We know that intersection of context free language and regular language is also a context free. But, we have already proved that the language

$$L_1 = \{a^n b^n c^n \mid n \geq 0\}$$

is not context free. Since L_1 is not context free, it implies that the given language

$$L = \{w \mid w \in \{a,b,c\}^* \text{ where } n_a(w) = n_b(w) = n_c(w)\}$$

is also not context free.

Example 6.18: Show that $L = \{ww \mid w \in \{a,b\}^*\}$ is not context free.

Step 1: Assume that L is context free and is infinite. Let $z = a^n b^n a^n b^n \in L$.

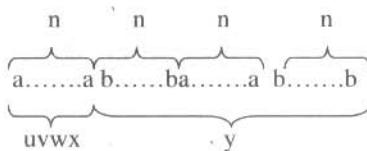
Step 2: Since $|z| > n$, according to Pumping Lemma we can split z into u, v, w, x and y such that

$$|vwx| \leq n \text{ and } |vx| \geq 1$$

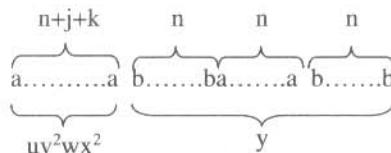
and $uv^iwx^i y \in L$ for $i = 0, 1, 2, \dots$. Let us take the various cases of splitting the string into u, v, w, x and y.

Case 1: The string vwx is within first a^n .

Let $v = a^j, x = a^k$ where $|vx| = j + k \geq 1$ and $|vwx| \leq n$ which can be shown pictorially as



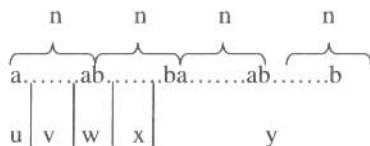
Now, according to Pumping Lemma, $uv^2wx^2y \in L$ for $i = 2$.



Note that $a^{n+j+k}b^n a^n b^n = uv^2wx^2y \notin L$ when $j+k \geq 1$ (Note that the string ww of the form is not generated). But by pumping lemma, $uv^2wx^2y \in L$, which is the contradiction.

Case 2: Let v is in first a^n and x is in first b^n .

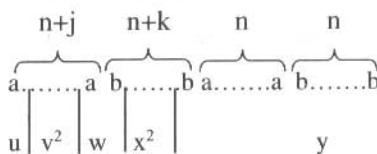
Let $v = a^j, x = b^k$ such that $|vx| = j + k \geq 1$ and $|vwx| \leq n$



Note: w has a's and b's

Note: w has a's and b's.

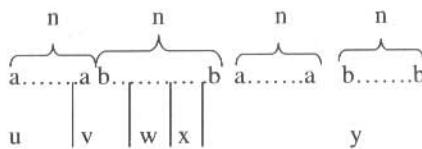
Now, according to Pumping Lemma, $uv^2wx^2y \in L$ for $i = 2$. The string uv^2wx^2y generated is as shown below:



Note that $a^{n+j}b^{n+k}a^n b^n = uv^2wx^2y \notin L$ when $j+k >= 1$ (Note that the string ww of the form is not generated). But by pumping lemma, $uv^2wx^2y \in L$, which is the contradiction.

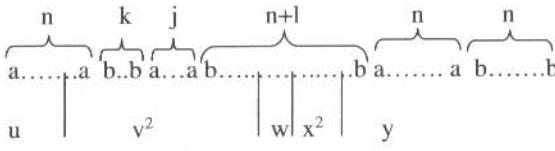
Case 3: Let v overlaps the first $a^n b^n$ and x is the first b^n .

Let $v = a^l b^k$ and $x = b^l$ such that $|vx| = j + k + l \geq 1$ and $|vwx| \leq n$ as shown below:



Note: v has a's and b's, w and x has b's, y has b's followed by $a^m b^m$.

According to Pumping Lemma, $uv^2wx^2y \in L$ for $i = 2$. The string uv^2wx^2y generated is as shown below:



It is very clear from the above figure that $uv^2wx^2y \notin L$ when $j+k+l \geq 1$. (Note that the string ww of the form is not generated). But by pumping lemma $uv^2wx^2y \in L$ which is contradiction.

In all the three cases we get the contradiction. Even in one of the cases we get the contradiction, we can say that the language generated is not context free. So, it has been shown that the language

$$L = \{ww \mid w \in \{a,b\}^*\}$$

is not context free.

Example 6.19: Show that $L = \{a^{n!} \mid n \geq 0\}$ is not context free.

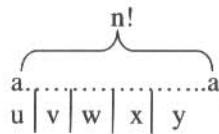
Step 1: Assume that L is context-free and is infinite. Let $z = a^{n!} \in L$ where $|a^{n!}| > n$.

Step 2: Since $|z| > n$, according to Pumping Lemma we can split $z = a^{n!}$ into u, v, w, x and y such that

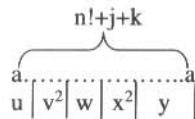
$$|vwx| \leq n \text{ and } |vx| \geq 1$$

so that $uv^iwx^iy \in L$ for $i = 0, 1, 2, \dots$

The splitting of the string z is shown in figure below:



Let $v = a^j$, $x = a^k$ such that $|vx| = j + k \geq 1$ and $|vwx| \leq n$. The string uv^2wx^2y can be generated and is shown in the figure below:



$$uv^2wx^2y = a^{n!+j+k} \text{ whenever } j+k \geq 1$$

When $n = 2$,

$$\begin{aligned} n! + j + k &= n! + j + k \leq n! + n \\ &< n! + n!n \\ &= n!(n+1) \\ &= (n+1)! \end{aligned}$$

$$\downarrow \\ n! < n! + j + k < (n+1)!$$

Since $n!+j+k$ it lies between $n!$ and $(n+1)!$, the string generated

$$uv^2wx^2y = a^{n!+j+k} \notin L$$

which is a contradiction. So, the language

$$L = \{a^{n!} \mid n \geq 0\}$$

is not context free.

Example 6.20: Show that $L = \{a^p b^q \mid p = q^2\}$ is not context free.

Step 1: Assume that L is context-free and is infinite. So, we can apply Pumping Lemma.

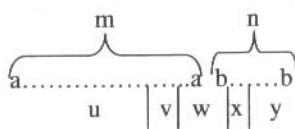
Let $z = a^m b^n \in L$ where $m = n^2$ and $|a^m b^n| > n$.

Step 2: Since $|z| > n$, according to Pumping Lemma we can split the string $z = uvwxy = a^m b^n$ such that

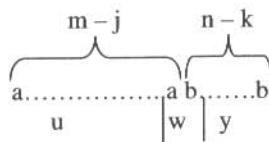
$$|vwx| \leq n \text{ and } |vx| \geq 1$$

so that $uv^i wx^i y \in L$ for $i = 0, 1, 2, \dots$

Step 3: Assume that the string vwx is within $a^m b^n$. Let $v = a^j$ $x = b^k$ such that $|vx| = j+k \geq 1$.



When $j+k \geq 1$, and $i = 0$, then we have $uv^i wx^i y$ as shown below:



i.e.,

$$a^{m-j} b^{n-k} = uw y \text{ and } j \neq 0 \text{ and } k \neq 0$$

\Downarrow (implies)

$$\begin{aligned} (n-k)^2 &\leq (n-1)^2 \\ &= n^2 - 2n + 1 \\ &= m - 2n + 1 \text{ (since } m = n^2) \\ &< m - j \end{aligned}$$

\Downarrow (implies)

$$m - j \neq (n - k)^2$$

It means that $m-j$ is not a perfect square. According to Pumping Lemma it should be a perfect square. So,

$$a^{m-j}b^{n-k} = uwv \notin L$$

which is contradiction.

Step 4: So, the language

$$L = \{a^p b^q \mid p = q^2\}$$

is not context free.

6.9. CFLs are Closed Under Union, Concatenation and Star

Theorem: If L_1 and L_2 are CFLs, then $L_1 \cup L_2$, $L_1 \cdot L_2$ and L_1^* also denote the CFLs and so the Context Free Languages are closed under union, concatenation, start-closure.

Proof: Let L_1 and L_2 are two CFLs generated by the CFGs

$$G_1 = (V_1, T_1, P_1, S_1)$$

and

$$G_2 = (V_2, T_2, P_2, S_2)$$

respectively and assume that V_1 and V_2 are disjoint.

Case 1: Union of two CFLs is CFL.

Now, let us consider the language L_3 generated by the grammar

$$G_3 = (V_1 \cup V_2 \cup S_3, T_1 \cup T_2, P_3, S_3)$$

where

- S_3 is a the start symbol for the grammar G_3 and $S_3 \notin (V_1 \cup V_2)$
- $P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 \mid S_2\}$

It is clear from this that the grammar G_3 is context free and the language generated by this grammar is context free. It is easy to prove that

$$L_3 = L_1 \cup L_2$$

If we assume $w \in L_1$, then the possible derivation from S_3 is

$$S_3 \Rightarrow S_1 \xrightarrow{*} w$$

On similar lines if we assume $w \in L_2$, then the possible derivation from S_3 is

$$S_3 \Rightarrow S_2 \xrightarrow{*} w$$

So, if $w \in L_3$, one of the derivations

$$S_3 \Rightarrow S_1$$

or

$$S_3 \Rightarrow S_2$$

is possible. In the first case, all the variables in V_1 and all the terminals in T_1 may be used to get the derivation

$$S_1 \xrightarrow{*} w$$

which uses only the productions in P_1 . Similarly all the variables in V_2 and all the terminals in T_2 may be used to get the derivation

$$S_2 \xrightarrow{*} w$$

which uses only the productions in P_2 and it follows that

$$L_3 = L_1 \cup L_2$$

Thus, it is proved that context free languages are closed under union.

Case 2: Concatenation of two CFLs is CFL.

Now, let us consider the language L_4 generated by the grammar

$$G_4 = (V_1 \cup V_2 \cup S_4, T_1 \cup T_2, P_4, S_4)$$

where

- S_4 is a the start symbol for the grammar G_4 and $S_4 \notin (V_1 \cup V_2)$.
- $P_4 = P_1 \cup P_2 \cup \{S_4 \rightarrow S_1 S_2\}$

It is clear from this that the grammar G_4 is context free and the language generated by this grammar is context free and so

$$L_3 = L_1 L_2$$

Thus, it is proved that context free languages are closed under concatenation.

Case 3: CFLs are closed under star-closure.

Now, let us consider the language L_5 generated by the grammar

$$G_5 = (V_1 \cup S_5, T_1, P_5, S_5)$$

where

- S_5 is a the start symbol for the grammar G_5
- $P_5 = P_1 \cup \{S_5 \rightarrow S_1 S_5 \mid \epsilon\}$

It is clear from this that the grammar G_5 is context free and the language generated by this grammar is context free and so

$$L_5 = L_5^*$$

Thus, it is proved that context free languages are closed under star-closure.

Thus, we have proved that the context free languages are closed under union, concatenation and star-closure.

6.10. CFLs are Not Closed Under Intersection

Theorem: The CFLs are not closed under intersection. If L_1 and L_2 are context free languages, it is not always true that $L_1 \cap L_2$ is context free language.

Proof: Let us prove this theorem by taking counter examples. Consider the two languages

$$L_1 = \{a^n b^n c^m \mid n \geq 0, m \geq 0\}$$

and

$$L_2 = \{a^n b^m c^m \mid n \geq 0, m \geq 0\}$$

The two languages are context free, as we can easily obtain the corresponding context free grammars

$$\begin{array}{ll} S & \rightarrow S_1 S_2 \\ S_1 & \rightarrow aS_1b \mid \epsilon \\ S_2 & \rightarrow CS_2 \mid \epsilon \end{array}$$

and

$$\begin{array}{ll} S & \rightarrow aS \mid S_1 \\ S_1 & \rightarrow bS_1c \mid \epsilon \end{array}$$

Now, let us take

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$$

We have already proved earlier that this language is not context free. Thus we can prove that the family of context free languages is not closed under intersection.

6.11. CFLs are Not Closed Under Complementation

Theorem: The CFLs are not closed under complementation. If L is context free language, it is not true that complement of L is context free language.

Proof: Let us prove this theorem by contradiction. Suppose that context-free languages are closed under complementation. So, if L_1 and L_2 are context-free languages, then \bar{L}_1 and \bar{L}_2 are also context free. We have already proved that CFLs are closed under union.

So,

$$\bar{L}_1 \cup \bar{L}_2$$

must be context free. Since we have assumed that the CFLs are closed under complementation,

$$\bar{L}_1 \cup \bar{L}_2$$

must be context free. But, according to de Morgan's law

$$\bar{L}_1 \cup \bar{L}_2 = L_1 \cap L_2$$

So,

$$L_1 \cap L_2$$

must be context free which is a contradiction (As we have already proved that the context free languages are not closed under intersection). Since the context free languages are not closed under intersection, our assumption that the CFLs are closed under complementation is not true. So, the family of context free languages are closed under complementation.

Note: We have seen that the following languages

1. $L = \{a^n b^n c^n \mid n \geq 0\}$
2. $L = \{w \mid w \in \{a,b,c\}^* \text{ where } n_a(w) = n_b(w) = n_c(w)\}$
3. $L = \{ww \mid w \in \{a,b\}^*\}$
4. $L = \{a^{n!} \mid n \geq 0\}$
5. $L = \{a^p b^q \mid p = q^2\}$

are not context free and it is not possible to represent using context free grammars and so we can not have the corresponding PDA for these types of languages. Even then some of the important points we can make at this stage are:

1. The regular languages are the subset of context free languages and so context free languages are more powerful than the regular languages.

2. The statement 1 automatically implies that PDAs are more powerful than the finite automaton.
3. But, the PDAs are not strong enough to accept some of the languages as pointed out earlier that are not context free and so we need much more powerful automaton than the PDA such as Linear bounded Automat or Turing Machine. Let us concentrate on Turing Machines in the next chapter.

Exercises

1. State and prove Pumping Lemma for context free languages?
2. What are the applications of Pumping Lemma?
3. Show that $L = \{a^nb^n \mid n \geq 0\}$ is not context free.
4. Show that $L = \{w \mid w \in \{a,b,c\}^* \text{ where } n_a(w) = n_b(w) = n_c(w)\}$ is not context free.
5. Show that $L = \{ww \mid w \in \{a,b\}^*\}$ is not context free.
6. Show that $L = \{a^{n!} \mid n \geq 0\}$ is not context free.
7. Show that $L = \{a^pb^q \mid p = q^2\}$ is not context free.
8. Prove that CFLs are closed under union, concatenation and star-closure.
9. Prove that CFLs are not closed under intersection.
10. Prove that CFLs are not closed under complementation.

Chapter 7

Turing Machines

What are we studying in this chapter . . .

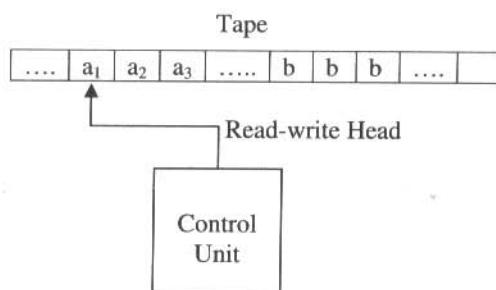
- ▶ *Turing machines*
- ▶ *Programming techniques for Turing machines*
- ▶ *Extensions to the basic Turing machines*
- ▶ *Turing machines and computers*

Turing Machine (TM) is modified version of the PDA and it is much more powerful than PDA. Instead of using stack as in PDA, the TM uses the tape to store the symbols. The Turing machine is a generalized machine which can recognize all types of languages viz, regular languages (generated from regular grammar also known as type 3 grammar), context free languages (generated from context free grammars also known as type 2 grammars) and context sensitive language (generated from context sensitive grammar also known as type 1 grammar). Apart from these languages, the turing machine also accepts the language generated from type 0 grammar (also known as unrestricted grammar). Thus, Turing machine can accept any language. This chapter mainly concentrates on building the turing machines for any language.

7.1. Turing Machine Model

The Turing machine model is shown in figure below. It is a finite automaton connected to read-write head with the following components:

- Tape
- Read-write head
- Control unit



Tape is used to store the information and is divided into cells. Each cell can store the information of only one symbol. The string to be scanned will be stored from the leftmost position on the tape. The string to be scanned should end with blanks. The tape is assumed to be infinite both on left side and right side of the string.

Read-write head: The read-write head can read a symbol from where it is pointing to and it can write into the tape to where it points to.

Control Unit: The reading from the tape or writing into the tape is determined by the control unit. The different moves performed by the machine depends on the current *scanned symbol* and the *current state*. The control unit consults *action table* i.e., *transition table* and carry out the tasks.

The read-write head can move either towards left or right i.e., movement can be on both the directions. The various actions performed by the machine are:

1. Change of state from one state to another state.
2. The symbol pointing to by the read-write head can be replaced by another symbol.
3. The read-write head may move either towards left or towards right.

If there is no entry in the table for the current combination of symbol and state, then the machine will halt. The Turing machines can be represented using various notations such as

- Transition tables
- Instantaneous descriptions
- Transition diagram

7.2. Transition Table

Consider the transition table shown in Table 7.1. Later sections describe how to obtain the transition table 7.1. Note that for each state q , there can be a corresponding entry for the symbol in Γ .

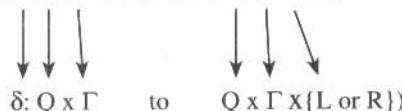
In this table the symbols a and b are input symbols and can be denoted by the symbol Σ . The symbols a, b, X, Y and B are denoted by Γ and $\Sigma \subseteq \Gamma$. The symbol B indicates a blank character and usually the string ends with infinite number of B 's, i.e., blank characters. The undefined entries in the table indicate that there are no-transitions defined or there can be a transition to dead state. When there is a transition to the dead state, the machine halts and the input string is rejected by the machine.

Table 7.1. Transition table

δ	Tape symbols (Γ)				
States	a	b	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	(q_1, a, R)	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	(q_2, a, L)	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

The transitions shown in the table can also be written as

$\delta(q_0, a)$	=	(q_1, X, R)
$\delta(q_0, Y)$	=	(q_3, Y, R)
$\delta(q_1, a)$	=	(q_1, a, R)
$\delta(q_1, b)$	=	(q_2, Y, L)
$\delta(q_1, Y)$	=	(q_1, Y, R)
$\delta(q_2, a)$	=	(q_2, a, L)
$\delta(q_2, X)$	=	(q_0, X, R)
$\delta(q_2, Y)$	=	(q_2, Y, L)
$\delta(q_3, Y)$	=	(q_3, Y, R)
$\delta(q_3, B)$	=	(q_4, B, R)



In general, δ can be defined as follows:

$\delta: Q \times \Gamma \rightarrow (Q \times \Gamma \times \{L, R\})$ i.e., cross product of Q , Γ and $\{L, R\}$

where

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a, b, X, Y, B\}$$

q_0 is the start state

B is a special symbol indicating blank character

$F = \{q_4\}$ which is the final state

Thus, formally a Turing Machine M can be defined as follows.

❖ **Definition:** The Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

Q is set of finite states

Σ is set of input alphabets

Γ is set of tape symbols

δ is transition function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$

q_0 is the start state

B is a special symbol indicating blank character

$F \subseteq Q$ is set of final states.

Since there can be several variations of TM (which we see in the coming chapters), the TM that we discuss now can be called **standard Turing Machine** with the following features:

1. The Turing machine has a tape that is divided into number of cells with each cell capable of storing only one symbol. The tape is unbounded (i.e., no boundary on the left as well as on the right) with any number of left or right moves.
2. The machine is deterministic. It can have either zero or one transition for each configuration.
3. Some of the symbols on the tape can be considered as the input. The symbols on the tape (some symbols or all the symbols) can be considered as the output whenever the TM halts.

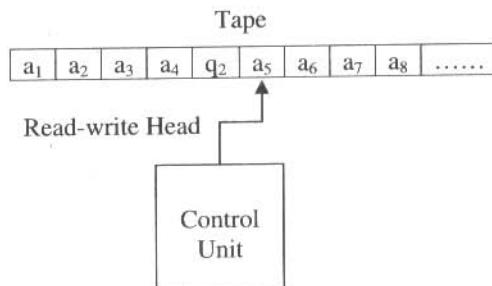
7.3. Instantaneous Description (ID)

Unlike the ID described in PDA, in Turing machine (TM), the ID is defined on the whole string (not on the string to be scanned) and the current state of the machine. The formal definition of *instantaneous description* (ID) in case of TM is defined as shown below:

❖ **Definition:** An ID of TM is a string in $\alpha q \beta$, where q is the current state, $\alpha \beta$ is the string made from tape symbols denoted by Γ i.e., α and $\beta \in \Gamma^*$. The read-write head points to the first character

of the substring β . The initial ID is denoted by $q\alpha\beta$ where q is the start state and the read-write head points to the first symbol of α from left. The final ID is denoted by $\alpha\beta qB$ where $q \in F$ is the final state and the read-write head points to the blank character denoted by B .

Example 7.1: Consider the snapshot of a Turing machine:



In this machine, each $a_i \in \Gamma$ (i.e., each a_i belongs to the tape symbol). In this snapshot, the symbol a_5 is under read-write head which is the next symbol to be scanned and the symbol towards left of a_5 i.e., q_2 is the current state. So, in this case an ID is denoted by

$$\underbrace{a_1 a_2 a_3 a_4}_{\alpha} \underbrace{q_2}_{q} \underbrace{a_5 a_6 a_7 a_8 \dots}_{\beta}$$

general form of ID

where the substring

$a_1 a_2 a_3 a_4$

towards left of the state q_2 is the left sequence, the substring

$a_5 a_6 a_7 a_8 \dots$

towards right of the state q_2 is the right sequence and

q_2

is the current state of the machine. The symbol a_5 is the next symbol to be scanned. Assume that the current ID of the Turing machine is

$a_1 a_2 a_3 a_4 q_2 a_5 a_6 a_7 a_8 \dots$

as shown in snapshot of Example 7.1. Suppose, there is a transition

$$\delta(q_2, a_5) = (q_3, b_1, R)$$

It means that if the machine is in state q_2 and the next symbol to be scanned is a_5 , then the machine enters into state q_3 replacing the symbol a_5 by b_1 and R indicates that the read-write head is moved one symbol towards right. The new configuration obtained is

$a_1a_2a_3a_4b_1q_3a_6a_7a_8\dots$

This can be represented by a *move* as shown below:

$a_1a_2a_3a_4q_2a_5a_6a_7a_8\dots \vdash a_1a_2a_3a_4b_1q_3a_6a_7a_8\dots$

Similarly if the current ID of the Turing machine is

$a_1a_2a_3a_4q_2a_5a_6a_7a_8\dots$

and there is a transition

$$\delta(q_2, a_5) = (q_1, c_1, L)$$

means that if the machine is in state q_2 and the next symbol to be scanned is a_5 , then the machine enters into state q_1 replacing the symbol a_5 by c_1 and L indicates that the read-write head is moved one symbol towards left. The new configuration obtained is

$a_1a_2a_3q_1a_4c_1a_6a_7a_8\dots$

This can be represented by a *move* as shown below:

$a_1a_2a_3a_4q_2a_5a_6a_7a_8\dots \vdash a_1a_2a_3q_1a_4c_1a_6a_7a_8\dots$

This configuration indicates that the new state is q_1 , the next input symbol to be scanned is a_4 . In general, the actions performed by TM depends on

1. The current state.
2. The whole string to be scanned.
3. The current position of the read-write head.

The action performed by the machine consists of

1. Changing the states from one state to another.
2. Replacing the symbol pointed to by the read-write head.
3. Movement of the read-write head towards left or right.

The formal definition of *move* for TM is shown below:

❖ **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM. Let the ID of M be

$a_1a_2a_3\dots a_{k-1}qa_ka_{k+1}\dots a_n$

where $a_j \in \Gamma$ for $1 \leq j \leq n$, $q \in Q$ is the current state and a_k as the next symbol to be scanned. If there is a transition

$$\delta(q, a_k) = (p, b, R)$$

then the *move* of machine M will be

$$a_1 a_2 a_3 \dots a_{k-1} q a_k a_{k+1} \dots a_n \vdash a_1 a_2 a_3 \dots a_{k-1} b p a_{k+1} \dots a_n$$

If there is a transition

$$\delta(q, a_k) = (p, b, L)$$

then the *move* of machine M will be

$$a_1 a_2 a_3 \dots a_{k-1} q a_k a_{k+1} \dots a_n \vdash a_1 a_2 a_3 \dots a_{k-2} p a_{k-1} b a_{k+1} \dots a_n$$

7.4. Acceptance of a Language by TM

Note: The Turing Machine can do one of the following things:

- a) Halt and accept by entering into final state.
- b) Halt and reject. This is possible if the transition is not defined, i.e., $\delta(q, a)$ is not defined.
- c) TM will never halt and enters into an infinite loop.

It is true that there is no algorithm to determine and tell whether a given machine always halts.

The language accepted by TM is defined as follows.

❖ **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM. The language $L(M)$ accepted by M is defined as

$$L(M) = \{w \mid q_0 w \xrightarrow{*} \alpha_1 p \alpha_2 \text{ where } w \in \Sigma^*, p \in F \text{ and } \alpha_1, \alpha_2 \in \Gamma^*\}$$

where $q_0 w$ is the initial ID and $\alpha_1 p \alpha_2$ is the final ID. The set of all those words w in Σ^* which causes M to move from start state q_0 to the final state p.

The string w which is the string to be scanned should end with infinite number of blanks. Initially, the machine will be in the start state q_0 with read-write head pointing to the first symbol of w from left. After some sequence of moves, if the Turing machine enters into final state and halts, then we say that the string w is accepted by Turing machine. The language accepted by TM is called **recursively enumerable language** or **RE language**. The formal definition is shown below:

❖ **Definition:** A language L is *recursively enumerable*, if it is accepted by a TM, i.e., given a string w which is input to TM, the machine halts and outputs Yes if it belongs to the language. If w does not belong to the language L , the TM halts and outputs NO.

The languages with Turing Machine which will always halt and output yes if it belongs to the language or output no if it does not belong to the language are called *decidable languages* or *recursive languages*. The Turing machines that always halt irrespective of whether they accept or not are a good model for an algorithm. If an algorithm exists to solve a given problem, then the problem is *decidable* otherwise it is un-decidable problem.

7.5. Construction of Turing Machine (TM)

In this section, we shall see how TMs can be constructed.

■ **Example 7.2:** Obtain a Turing machine to accept the language

$$L = \{0^n 1^n \mid n \geq 1\}$$

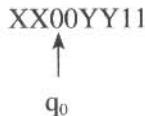
It is given that the language accepted by TM should have n number of 0's followed by n number of 1's. For this let us take an example of the string $w = 00001111$. The string w should be accepted as it has four zeroes followed by four 1's.

General Procedure

Let q_0 be the start state and let the read-write head points to the first symbol of the string to be scanned. The general procedure to design TM for this case is shown below:

1. Replace the left most 0 by X and change the state to q_1 and then move the read-write head towards right. This is because, after a zero is replaced, we have to replace the corresponding 1 so that number of zeroes matches with number of 1's.
2. Search for the leftmost 1 and replace it by the symbol Y and move towards left (so as to obtain the leftmost 0 again). Steps 1 and 2 can be repeated.

Design: Consider the situation

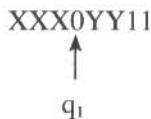


where first two 0's are replaced by Xs and first two 1's are replaced by Ys. In this situation, the read-write head points to the left most zero and the machine is in state q_0 . With this as the configuration, now let us design the TM.

Step 1: In state q_0 , replace 0 by X, change the state to q_1 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_0, 0) = (q_1, X, R)$$

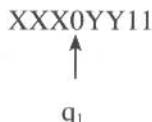
The resulting configuration is shown below:



Step 2: In state q_1 , we have to obtain the left-most 1 and replace it by Y. So, let us move the pointer to point to leftmost 1. When the pointer is moved towards 1, the symbols encountered may be 0 and Y. Irrespective what symbol is encountered, replace 0 by 0, Y by Y, remain in state q_1 and move the pointer towards right. The transitions for this can be of the form:

$$\begin{aligned}\delta(q_1, 0) &= (q_1, 0, R) \\ \delta(q_1, Y) &= (q_1, Y, R)\end{aligned}$$

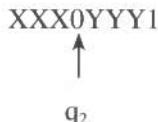
When these transitions are repeatedly applied, the following configuration is obtained:



Step 3: In state q_1 , if the input symbol to be scanned is a 1, then replace 1 by Y, change the state to q_2 and move the pointer towards left. The transition for this can be of the form:

$$\delta(q_1, 1) = (q_2, Y, L)$$

The resulting configuration is shown below:



Note that the pointer should be moved towards left. This is because, a zero is replaced by X and the corresponding 1 is replaced by Y. Now, we have to scan for the left most 0 and so, the pointer was move towards left.

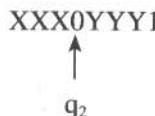
Step 4: Note that to obtain leftmost zero, we need to obtain right most X first. So, we scan for the right most X. During this process we may encounter Y's and 0's. Replace Y by Y, 0 by 0,

remain in state q_2 only and move the pointer towards left. The transitions for this can be of the form:

$$\delta(q_2, Y) = (q_2, Y, L)$$

$$\delta(q_2, 0) = (q_2, 0, L)$$

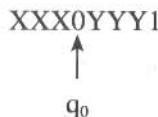
The following configuration is obtained:



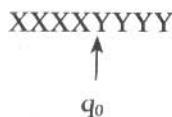
Step 5: Now, we have obtained the right most X. To get leftmost 0, replace X by X, change the state to q_0 and move the pointer towards right. The transition for this can be of the form:

$$\delta(q_2, X) = (q_0, X, R)$$

and the following configuration is obtained:



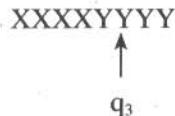
Now, repeating the steps 1 through 5, we get the configuration shown below:



Step 6: In state q_0 , if the scanned symbol is Y, it means that there are no more 0's. If there are no 0's we should see that there are no 1's. For this we change the state to q_3 , replace Y by Y and move the pointer towards right. The transition for this can be of the form:

$$\delta(q_0, Y) = (q_3, Y, R)$$

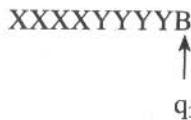
and the following configuration is obtained:



In state q_3 , we should see that there are only Ys and no more 1's. So, as we scan replace Y by Y and remain in q_3 only. The transition for this can be of the form:

$$\delta(q_3, Y) = (q_3, Y, R)$$

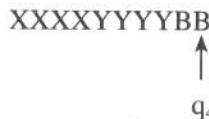
Repeatedly applying this transition, the following configuration is obtained:



Note that the string ends with infinite number of blanks and so, in state q_3 if we encounter the symbol B, means that end of string is encountered and there exists n number of 0's ending with n number of 1's. So, in state q_3 , on input symbol B, change the state to q_4 , replace B by B and move the pointer towards right and the string is accepted. The transition for this can be of the form:

$$\delta(q_3, B) = (q_4, B, R)$$

where q_4 is the final state and the following configuration is obtained:



So, the Turing machine to accept the language

$$L = \{a^n b^n \mid n \geq 1\}$$

is given by

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

where

$$Q = \{q_0, q_1, q_2, q_3\}$$

$$\Sigma = \{0, 1\}$$

$$\Gamma = \{0, 1, X, Y, B\}$$

$q_0 \in Q$ is the start state of machine

$B \in \Gamma$ is the blank symbol

$F = \{q_4\}$ is the final state

δ is shown below:

$$\delta(q_0, 0) = (q_1, X, R)$$

$$\delta(q_1, 0) = (q_1, 0, R)$$

$$\delta(q_1, Y) = (q_1, Y, R)$$

$$\delta(q_1, 1) = (q_2, Y, L)$$

$$\delta(q_2, Y) = (q_2, Y, L)$$

$$\delta(q_2, 0) = (q_2, 0, L)$$

$$\delta(q_2, X) = (q_0, X, R)$$

$$\delta(q_0, Y) = (q_3, Y, R)$$

$$\delta(q_3, Y) = (q_3, Y, R)$$

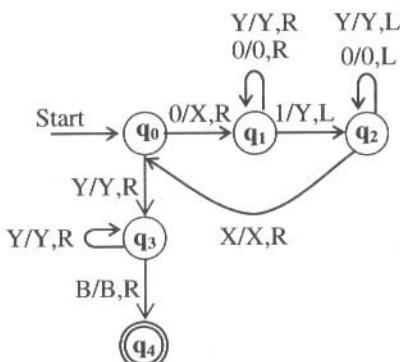
$$\delta(q_3, B) = (q_4, B, R)$$

The transitions can also be represented using tabular form as shown below:

δ	Tape symbols (Γ)				
States	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	-	-	-	-	-

7.6. Transition Diagram for Turing Machine (TM)

The Turing Machine can be represented using transition diagram. The transition diagram consists of nodes corresponding to the states of Turing Machine. An edge from state q to state p will have a label of the form $(X / Y, D)$ where X and Y are tape symbols and D is the direction either ‘L’ or ‘R’ where ‘L’ stands for *left* and ‘R’ stands for *right* i.e., the movement of the head can be either left or right. Here, X is the scanned symbol and Y is the symbol written on to the tape. The start state of the Turing Machine is indicated by an arrow entering the state with label ‘Start’. The final states are represented by two concentric circles. The transition diagram for the Example 7.2 is shown below:



To accept the string: The sequence of moves or computations (IDs) for the string 0011 made by the Turing machine is shown below:

(Initial ID) $q_00011 \vdash Xq_1011 \vdash X0q_111 \vdash Xq_20Y1 \vdash q_2X0Y1 \vdash Xq_00Y1 \vdash XXq_1Y1 \vdash XXYq_11 \vdash XXq_2YY \vdash Xq_2XYY \vdash XXq_0YY \vdash XXYq_3Y \vdash XXYYq_3 \vdash XXYYBq_4$ (Final ID)

Since the final state q_4 is reached, the string 0011 is accepted.

■ Example 7.3: Obtain a Turing machine to accept the language

$$L(M) = \{0^n1^n2^n \mid n \geq 1\}$$

It is given that the language should consist of n number of 0's followed by n number of 1's which in turn should be followed by n number of 2's. Let us consider the string 000011112222 and we shall see how to design the Turing Machine. To design the Turing Machine, consider the situation where first two 0's are replaced by X's, first two 1's are replaced by Y's and first two 2's are replaced by Z's as shown in Figure 7.2a.

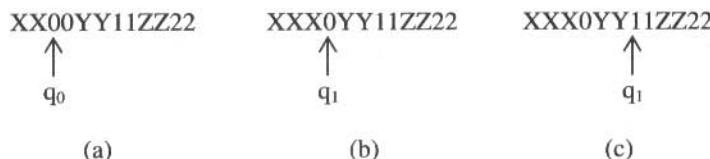


Figure 7.2. Various configurations

Now, with Figure 7.2a as the current configuration, let us design the Turing machine. In state q_0 , if the next scanned symbol is 0 replace it by X, change the state to q_1 and move the pointer towards right and the situation shown in Figure 7.2b is obtained. The transition for this can be of the form:

$$\delta(q_0, 0) = (q_1, X, R)$$

In state q_1 , we have to search for the leftmost 1. It is clear from Figure 7.2b that, when we are searching for the symbol 1, we may encounter the symbols 0 or Y. So, replace 0 by 0, Y by Y and move the pointer towards right and remain in state q_1 only. The transitions for this can be of the form:

$$\begin{aligned} \delta(q_1, 0) &= (q_1, 0, R) \\ \delta(q_1, Y) &= (q_1, Y, R) \end{aligned}$$

The configuration shown in Figure 7.2c is obtained. In state q_1 , on encountering 1 change the state to q_2 , replace 1 by Y and move the pointer towards right. The transition for this can be of the form:

$$\delta(q_1, 1) = (q_2, Y, R)$$

and the configuration shown in Figure 7.3a is obtained.

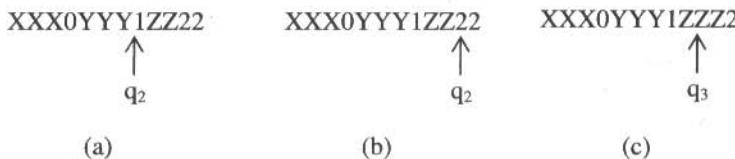


Figure 7.3. Various configurations

In state q_2 , we have to search for the leftmost 2. It is clear from Figure 7.3a that, when we are searching for the symbol 2, we may encounter the symbols 1 or Z. So, replace 1 by 1, Z by Z and move the pointer towards right and remain in state q_2 only and the configuration shown in Figure 7.3b is obtained. The transitions for this can be of the form:

$$\delta(q_2, 1) = (q_2, 1, R)$$

$$\delta(q_2, Z) = (q_2, Z, R)$$

In state q_2 , on encountering 2, change the state to q_3 , replace 2 by Z and move the pointer towards left. The transition for this can be of the form:

$$\delta(q_2, 2) = (q_3, Z, L)$$

and the configuration shown in Figure 7.3c is obtained. Once the TM is in state q_3 , it means that first 0 is replaced by X, first 1 is replaced by Y and first 2 is replaced by Z. At this point, we have to search for the rightmost X to get leftmost 0. During this process, it is clear from Figure 7.3c that the symbols such as Z's, 1's, Y's, 0's and X are scanned respectively one after the other. So, replace Z by Z, 1 by 1, Y by Y, 0 by 0, move the pointer towards left and stay in state q_3 only. The transitions for this can be of the form:

$$\delta(q_3, Z) = (q_3, Z, L)$$

$$\delta(q_3, 1) = (q_3, 1, L)$$

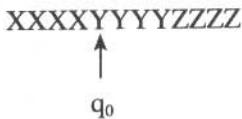
$$\delta(q_3, Y) = (q_3, Y, L)$$

$$\delta(q_3, 0) = (q_3, 0, L)$$

Only on encountering X, replace X by X, change the state to q_0 and move the pointer towards right to get leftmost 0. The transition for this can be of the form:

$$\delta(q_3, X) = (q_0, X, R)$$

All the steps shown above are repeated till the following configuration is obtained:



In state q_0 , if the input symbol is Y, it means that there are no 0's. If there are no 0's we should see that there are no 1's also. For this to happen change the state to q_4 , replace Y by Y and move the pointer towards right. The transition for this can be of the form:

$$\delta(q_0, Y) = (q_4, Y, R)$$

In state q_4 search for only Y's, replace Y by Y, remain in state q_4 only and move the pointer towards right. The transition for this can be of the form:

$$\delta(q_4, Y) = (q_4, Y, R)$$

In state q_4 , if we encounter Z, it means that there are no 1's and so we should see that there are no 2's and only Z's should be present. So, on scanning the first Z, change the state to q_5 , replace Z by Z and move the pointer towards right. The transition for this can be of the form:

$$\delta(q_4, Z) = (q_5, Z, R)$$

But, in state q_5 only Z's should be there and no more 2's. So, as long as the scanned symbol is Z, remain in state q_5 , replace Z by Z and move the pointer towards right. But, once blank symbol B is encountered change the state to q_6 , replace B by B and move the pointer towards right and say that the input string is accepted by the machine. The transitions for this can be of the form:

$$\delta(q_5, Z) = (q_5, Z, R)$$

$$\delta(q_5, B) = (q_6, B, R)$$

where q_6 is the final state.

So, the TM to recognize the language $L = \{0^n 1^n 2^n \mid n \geq 1\}$ is given by

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

where

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$$

$$\Sigma = \{0, 1, 2\}$$

$$\Gamma = \{0, 1, 2, X, Y, Z, B\}$$

q_0 is the start state

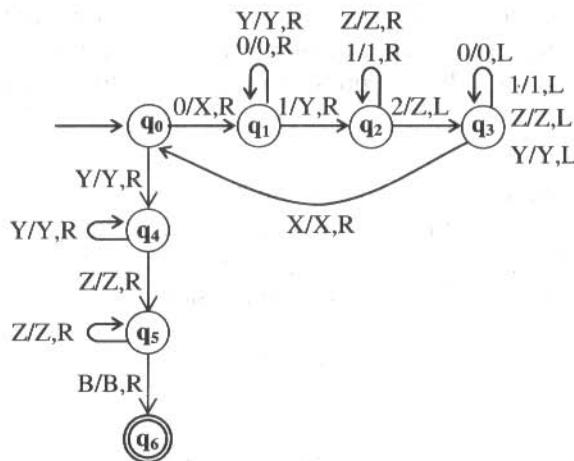
B is blank character

F = $\{q_6\}$ is the final state

δ is shown below using the transitional table:

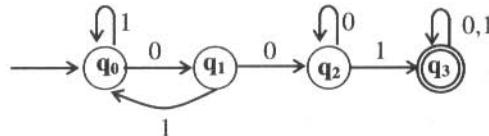
	Γ						
states	0	1	2	Z	Y	X	B
q_0	q_1, X, R				q_4, Y, R		
q_1	$q_1, 0, R$	q_2, Y, R			q_1, Y, R		
q_2		$q_2, 1, R$	q_3, Z, L	q_2, Z, R			
q_3	$q_3, 0, L$	$q_3, 1, L$		q_3, Z, L	q_3, Y, L	q_0, X, R	
q_4				q_5, Z, R	q_4, Y, R		
q_5				q_5, Z, R			
q_6							(q_6, B, R)

The transition diagram for this can be of the form:



Example 7.4: Obtain a TM to accept the language $L = \{w \mid w \in (0+1)^*\}$ containing the sub-string 001.

The DFA which accepts the language consisting of strings of 0's and 1's having a sub string 001 is shown below:



The transition table for the DFA is shown below:

	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_3
q_3	q_3	q_3

We have seen in chapter 3 that any language which is accepted by a DFA is regular. As the DFA processes the input string from left to right in only one direction, TM also processes the input string in only one direction(unlike the previous examples, where the read-write header was moving in both the directions). For each scanned input symbol (either 0 or 1), in whichever state the DFA was in, TM also enters into the same states on same input symbols, replacing 0 by 0 and 1 by 1 and the read-write head moves towards right. So, the transition table for DFA and TM remains same (the format may be different. It is evident in both the transition tables). So, the transition table for TM to recognize the language consisting of 0's and 1's with a substring 001 is shown below:

	0	1	B
q_0	$q_1, 0, R$	$q_0, 1, R$	-
q_1	$q_2, 0, R$	$q_0, 1, R$	-
q_2	$q_2, 0, R$	$q_3, 1, R$	-
q_3	$q_3, 0, R$	$q_3, 1, R$	q_4, B, R
q_4			

The TM is given by

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

where

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$

$$\Sigma = \{0, 1\}$$

$$\Gamma = \{0, 1\}$$

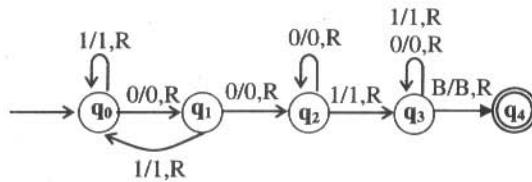
δ is shown in the form of transition table above

q_0 is the start state

B blank character

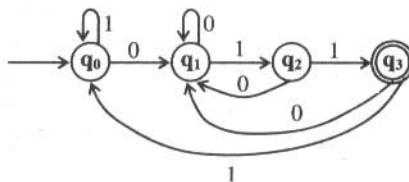
$F = \{q_4\}$ is the final state

The transition diagram for this is shown below:



Example 7.5: Obtain a Turing machine to accept the language containing strings of 0's and 1's ending with 011.

The DFA which accepts the language consisting of strings of 0's and 1's ending with the string 001 is shown below:



The transition table for the DFA is shown below:

δ	0	1
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_3
q_3^*	q_1	q_0

We have seen in Chapter 3 that any language which is accepted by a DFA is regular. As the DFA processes the input string from left to right in only one direction, TM also processes the input string in only one direction (similar to Example 7.5). For each scanned input symbol (either 0 or 1), in whichever state the DFA was in, TM also enters into the same states on same input symbols, replacing 0 by 0 and 1 by 1 and the read-write head moves towards right. So, the transition table for DFA and TM remains same (the format may be different. It is evident in both the transition tables). So, the transition table for TM to recognize the language consisting of 0's and 1's ending with a substring 001 is shown below:

δ	0	1	B
q_0	$q_1, 0, R$	$q_0, 1, R$	-

q_1	$q_1, 0, R$	$q_2, 1, R$	-
q_2	$q_1, 0, R$	$q_3, 1, R$	-
q_3	$q_1, 0, R$	$q_0, 1, R$	q_4, B, R
q_4	-	-	-

The TM is given by

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

where

$$Q = \{q_0, q_1, q_2, q_3\}$$

$$\Sigma = \{0, 1\}$$

$$\Gamma = \{0, 1, B\}$$

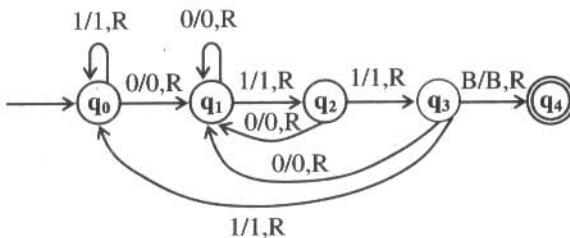
δ is defined already

q_0 is the start state

B blank character

$F = \{q_4\}$ is the final state

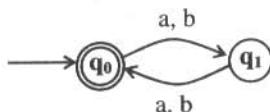
The transition diagram for this is shown below:



Example 7.6: Obtain a Turing machine to accept the language

$$L = \{w \mid w \text{ is even and } \Sigma = \{a, b\}\}$$

The DFA to accept the language consisting of even number of characters is shown below:



The transition table for the DFA is shown below:

	a	b
q_0	q_1	q_1
q_1	q_0	q_0

We have seen in chapter 3 that any language which is accepted by a DFA is regular. As the DFA processes the input string from left to right in only one direction, TM also processes the input string in only one direction (similar to Example 7.5). For each scanned input symbol (either a or b), in whichever state the DFA was in, TM also enters into the same states on same input symbols, replacing a by a and b by b and the read-write head moves towards right. So, the transition table for DFA and TM remains same (the format may be different.) So, the transition table for TM to recognize the language consisting of a's and b's having even number of symbols is shown below:

δ	A	b	B
q_0	q_1, a, R	q_1, b, R	q_2, B, R
q_1	q_0, a, R	q_0, b, R	-
q_2	-	-	-

The TM is given by

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

where

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

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a, b, B\}$$

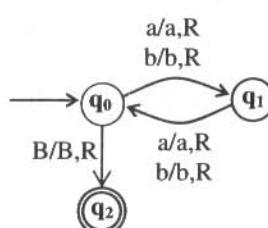
δ is defined in the form of table above

q_0 is the start state

B blank character

$F = \{q_2\}$ is the final state

The transition diagram of TM is given by



Example 7.7: Obtain a TM to compute the function $\dot{-}$ which is called *monus* or proper subtraction and is defined by $m \dot{-} n = \max(m-n, 0)$

Note: The monus operation is defined as

$$m \dot{-} n = m - n \text{ if } m \geq n$$

and

$$m \dot{-} n = 0 \text{ if } m < n.$$

It is clear from this definition that the Turing Machine is supposed to perform *monus* operation and will not accept anything and so the concept of final state will not come into picture. To start with the tape consists of $0^m 1 0^n$ which is surrounded by blanks and the machine halts with 0^{m-n} on its tape surrounded by blanks. Here, m number of 0's and n number of 0's are separated by the delimiter 1.

General Procedure: The sequence of 0's is partitioned into first group with m number of 0's followed by a 1 and followed by second group with n number of 0's. The machine finds the leftmost 0 and is replaced by blank B. Then move towards right to search for 1. After finding 1, it searches leftmost 0 in the second group and is replaced by 1 and move towards left to get leftmost 0 in the first group. This procedure is repeated till one of the following conditions are satisfied:

- When searching for a 0 in second group, if B is encountered it means that n number 0's in the second group are replaced by 1's and $n+1$ zeros in the first group are changed to B's. Now, the second group will have $n+1$ ones. The machine replaces $n+1$ 1's by one 0 and n B's and observe that only $m-n$ 0's exists on the tape
- In the first group if the machine M can not find a 0 (since first m 0's have already been changed to B's) it means that $m < n$ and so no 0's and 1's should be there on the tape.

The brief description of each state to achieve the above task is shown below:

In state q_0 : On encountering a 0, change the state to q_1 , replace 0 by B and move the head towards right using the transition

$$\delta(q_0, 0) = (q_1, B, R)$$

On encountering a 1 (means that all 0's in the first portion are replaced by B's) change the state to q_5 as shown below:

$$\delta(q_0, 1) = (q_5, B, R)$$

In state q_1 : we search for leftmost 1. Keep updating the head towards right till we encounter 1 replacing 0 by 0 and remaining in q_0 . On encountering 1 change the state to q_2 and move towards right using the transition

$$\delta(q_1, 0) = (q_1, 0, R)$$

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

In state q_2 : if we encounter 0 replace it by 1, change the state to q_3 (to get leftmost 0 in left portion) and move the header towards left as shown below:

$$\delta(q_2, 0) = (q_3, 1, L)$$

If we encounter 1 replace it by 1, remain in q_2 and move the head towards right to obtain leftmost 0 as shown below:

$$\delta(q_2, 1) = (q_2, 1, R)$$

If we encounter B, it means that no more 0's are found and change the state to q_4 which indicates n 0's out of m 0's are cancelled and subtraction is complete.

$$\delta(q_2, B) = (q_4, B, L)$$

Now, in state q_4 we have to convert all 1's to blanks.

In state q_3 : To get leftmost 0, replace 1 by 1, replace 0 by 0, remain in q_3 and move the head towards left using the transitions:

$$\delta(q_3, 0) = (q_3, 0, L)$$

$$\delta(q_3, 1) = (q_3, 1, L)$$

On encountering B, change the state to q_0 and move the head towards right using

$$\delta(q_3, B) = (q_0, B, R)$$

In state q_4 : Let us convert all 1's to blanks using the following transitions

$$\delta(q_4, 1) = (q_4, B, L)$$

$$\delta(q_4, 0) = (q_4, 0, L)$$

Note that out of m 0's, n + 1 0's are replaced by blanks. But, we are supposed to replace only n 0's. So, one blank should be replaced by 0 and halt the machine by entering into state q_6 .

$$\delta(q_4, B) = (q_6, 0, R)$$

In state q_5 : In state q_5 , the output should be 0. The tape should not have any symbols, except B's. So, replace all 1's and all remaining 0's with B's using the transitions

$$\delta(q_5, 0) = (q_5, B, R)$$

$$\delta(q_5, 1) = (q_5, B, R)$$

On encountering B, change the state to q_6 .

$$\delta(q_5, B) = (q_6, B, R)$$

So, the TM for monus function is shown below:

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

where

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$$

$$\Sigma = \{0, 1\}$$

$$\Gamma = \{0, 1, B\}$$

q_0 is the start state

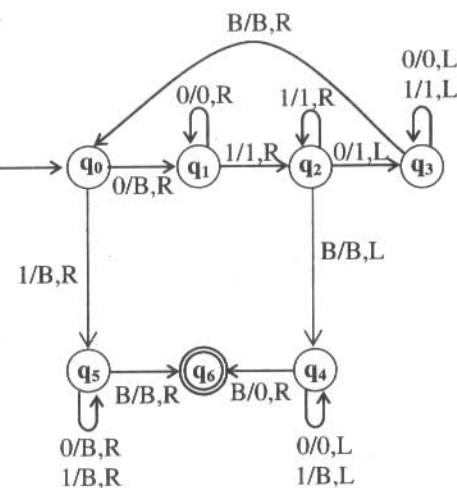
B is the blank character

$$F = \emptyset$$

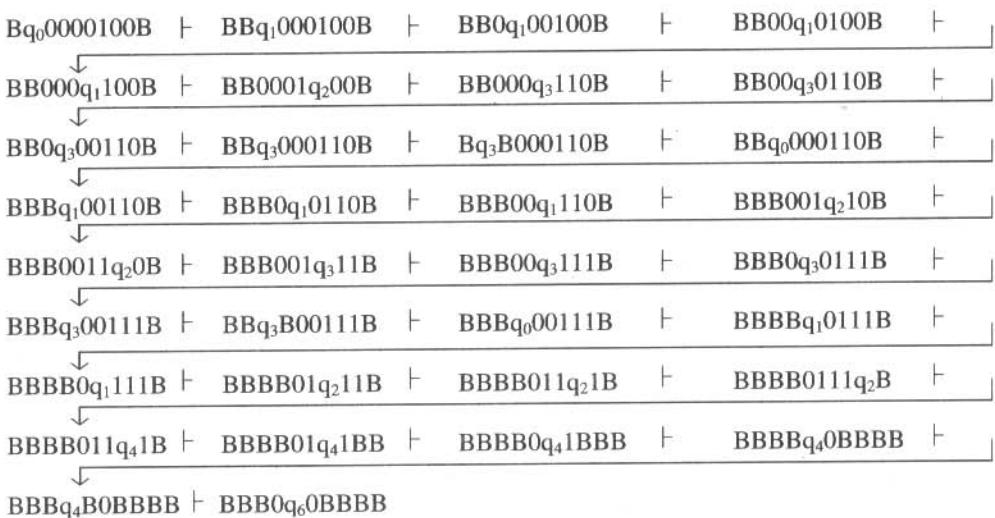
δ is shown below using the transition table:

δ	0	1	B
q_0	q_1, B, R	q_5, B, R	
q_1	$q_1, 0, R$	$q_2, 1, R$	
q_2	$q_3, 1, L$	$q_2, 1, R$	q_4, B, L
q_3	$q_3, 0, L$	$q_3, 1, L$	q_0, B, R
q_4	$q_4, 0, L$	q_4, B, L	$q_6, 0, R$
q_5	q_5, B, R	q_5, B, R	q_6, B, R
$*q_6$			

The transition diagram for this can be of the form:



The sequence of moves made by the TM for the string 0000100B outputting m – n is shown below:



Since q_6 on 0 is not defined, the Turing Machine halts. Observe that number of 0's on the tape is 2 which is 4–2. The sequence of moves made by the machine for the string B0100B is shown below:



Since the transition is not defined for the state q_6 , the Turing machine halts. Observe that no zeros are present on the tape since number of 0's in first portion is less than the number of 0's in the second portion.

Example 7.8: Obtain a TM to accept a string w of a's and b's such that $N_a(w)$ is equal to $N_b(w)$, i.e., the number of a's and b's in the string w should be equal.

General Procedure

Let q_0 be the start state and let the read-write head points to the first symbol of the string to be scanned which can either be a or b . The general procedure to design a TM will result in three cases depending on the next input symbol to be scanned namely:

1. On encountering B
2. On encountering a
3. On encountering b

Case 1: On encountering B

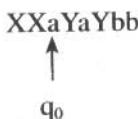
Change the state from q_0 to q_f , replace B by B and move the pointer towards right and the Turing machine halts. The transition for this is shown below:

$$\delta(q_0, B) = (q_f, B, R)$$

Case 2: On encountering a

General procedure: In state q_0 , if we encounter a , we skip all the subsequent symbols till we encounter b . Then come back to the next leftmost symbol and repeat any of the three cases based on the next symbol to be scanned.

Detail procedure: The first a is replaced by X and the first b is replaced by Y. For example, consider the string $aaababbb$ and consider the scenario where first two a 's replaced by X's and first two b 's are replaced by Y's and the read-write head points to the next symbol to be scanned as shown below:



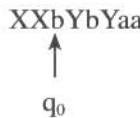
- In state q_0 , on encountering a , change the state to q_1 , replace a by X and move the pointer towards right to get leftmost b . The corresponding transition is
 $\delta(q_0, a) = (q_1, X, R)$
- It is clear from the figure that when we search for leftmost b , we may get a or Y. In such cases, the head should move towards right replacing a by a, Y by Y and remaining in state q_1 . The corresponding transitions are:
 $\delta(q_1, a) = (q_1, a, R)$
 $\delta(q_1, Y) = (q_1, Y, R)$
- In state q_1 on encountering b , replace b by Y, change the state to q_2 and move the pointer towards left to get the next rightmost X. The corresponding transition is
 $\delta(q_1, b) = (q_2, Y, L)$
- When searching for X, we may encounter Y's or a 's. In such cases remain in q_2 only and move the head towards left. The corresponding transitions are:
 $\delta(q_2, Y) = (q_2, Y, L)$
 $\delta(q_2, a) = (q_2, a, L)$

- In state q_2 on encountering X change the state to q_0 , replace X by X and move the pointer towards right using the transition
 $\delta(q_2, X) = (q_0, X, R)$
- In state q_0 , on encountering Y it indicates that so far the number of a's and b's are equal and so simply move the pointer towards right using the transition
 $\delta(q_0, Y) = (q_0, Y, R)$
- Repeat one of the three cases

Case 3: On encountering b

General procedure: In state q_0 , if we encounter b , we skip all the subsequent symbols till we encounter a . Then come back to the next leftmost symbol and repeat any of the three cases based on the next symbol to be scanned.

Detail procedure: The first b is replaced by X and the first a is replaced by Y. For example, consider the string $bbbabaaa$ and consider the scenario where first two b 's replaced by X's and first two a 's are replaced by Y's and the read-write head points to the next symbol to be scanned as shown below:



- In state q_0 , on encountering b , change the state to q_3 , replace b by X and move the pointer towards right to get leftmost a . The corresponding transition is
 $\delta(q_0, b) = (q_3, X, R)$
- It is clear from the figure that when we search for leftmost a , we may get b or Y . In such cases, the head should move towards right replacing b by b , Y by Y and remaining in state q_3 . The corresponding transitions are:
 $\delta(q_3, b) = (q_3, b, R)$
 $\delta(q_3, Y) = (q_3, Y, R)$
- On encountering a , replace a by Y , change the state to q_4 and move the pointer towards left to get the next rightmost X. The corresponding transition is
 $\delta(q_3, a) = (q_4, Y, L)$
- When searching for X, we may encounter Y 's or b 's. In such cases remain in q_4 only and move the head towards left. The corresponding transitions are:
 $\delta(q_4, Y) = (q_4, Y, L)$
 $\delta(q_4, b) = (q_4, b, L)$
- In state q_4 on encountering X change the state to q_0 , replace X by X and move the pointer towards right using the transition
 $\delta(q_4, X) = (q_0, X, R)$

- In state q_0 , on encountering Y it indicates that so far the number of a's and b's are equal and so simply move the pointer towards right using the transition

$$\delta(q_0, Y) = (q_0, Y, R)$$

- Repeat one of the three cases

So, the TM to accept strings of a's and b's such that number of a's is equal to number of b's is

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

where

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_f\}$$

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a, b, X, Y, B\}$$

q_0 is the start state

B is the blank character

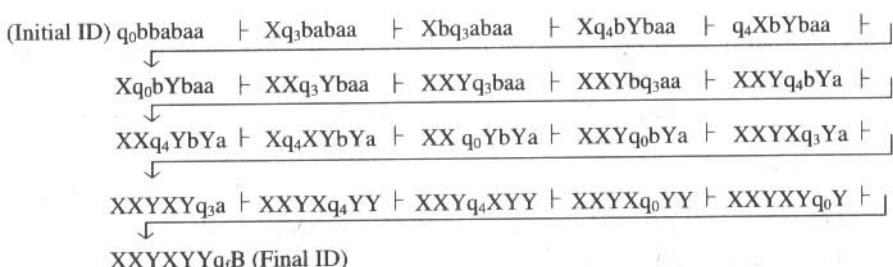
$$F = \{q_f\}$$

δ is shown below using the transition table:

	δ	Γ				
		a	b	X	Y	B
\rightarrow	q_0	q_1, X, R	q_3, X, R		q_0, Y, R	q_f, B, R
	q_1	q_1, a, R	q_2, Y, L		q_1, Y, R	
	q_2	q_2, a, L		q_0, X, R	q_2, Y, L	
	q_3	q_4, Y, L	q_3, b, R		q_3, Y, R	
	q_4		q_4, b, L	q_0, X, R	q_4, Y, L	
	$*q_f$	Final state				

The sequence of moves made by the Turing Machine for the string bbabaa is shown below:

To accept the string: The sequence of moves or computations (IDs) for the string *bbabaa* made by the Turing machine are shown below:



Similarly the sequence of moves for the string $bbaaab$ is shown below:

$$\begin{array}{ccccccccc}
 (\text{Initial ID}) & q_0 b b a a a b & \xrightarrow{\quad} & X q_3 b a a a b & \xrightarrow{\quad} & X b q_3 a a a b & \xrightarrow{\quad} & X q_4 b Y a a b & \xrightarrow{\quad} q_4 X b Y a a b \xrightarrow{\quad} \\
 & \downarrow & & & & & & & \\
 & X q_0 b Y a a b & \xrightarrow{\quad} & X X q_3 Y a a b & \xrightarrow{\quad} & X X Y q_3 a a b & \xrightarrow{\quad} & X X Y a q_3 a b & \xrightarrow{\quad} \dots \dots \text{ and so on.}
 \end{array}$$

Example 7.9: Obtain a Turing machine to accept a palindrome consisting of a's and b's of any length.

Let us assume that the first symbol on the tape is blank character B and is followed by the string which in turn ends with blank character B. Now, we have to design a Turing machine which accepts the string, provided the string is a palindrome. For the string to be a palindrome, the first and the last character should be same. The second character and last but one character should be same and so on. The procedure to accept only string of palindromes is shown below. Let q_0 be the start state of Turing machine.

Step 1: Move the read–write head to point to the first character of the string. The transition for this can be of the form

$$\delta(q_0, B) = (q_1, B, R)$$

Step 2: In state q_1 , if the first character is a , replace it by B and change the state to q_2 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_1, a) = (q_2, B, R)$$

Now, we move the read-write head to point to the last symbol of the string and the last symbol should be a . The symbols scanned during this process are a 's, b 's and B. Replace a by a , b by b and move the pointer towards right. The transitions defined for this can be of the form

$$\delta(q_2, a) = (q_2, a, R)$$

$$\delta(q_2, b) = (q_2, b, R)$$

But, once the symbol B is encountered, change the state to q_3 , replace B by B and move the pointer towards left. The transition defined for this can be of the form

$$\delta(q_2, B) = (q_3, B, L)$$

In state q_3 , the read-write head points to the last character of the string. If the last character is a , then change the state to q_4 , replace a by B and move the pointer towards left. The transitions defined for this can be of the form

$$\delta(q_3, a) = (q_4, B, L)$$

At this point, we know that the first character is a and last character is also a . Now, reset the read-head to point to the first non blank character as shown in step 5.

In state q_3 , if the last character is B (blank character), it means that the given string is an odd palindrome. So, replace B by B change the state to q_7 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_3, B) = (q_7, B, R)$$

Step 3: If the first character is the symbol b , replace it by B and change the state from q_1 to q_5 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_1, b) = (q_5, B, R)$$

Now, we move the read-write head to point to the last symbol of the string and the last symbol should be b . The symbols scanned during this process are a 's, b 's and B. Replace a by a , b by b and move the pointer towards right. The transitions defined for this can be of the form

$$\delta(q_5, a) = (q_5, a, R)$$

$$\delta(q_5, b) = (q_5, b, R)$$

But, once the symbol B is encountered, change the state to q_6 , replace B by B and move the pointer towards left. The transition defined for this can be of the form

$$\delta(q_5, B) = (q_6, B, L)$$

In state q_6 , the read-write head points to the last character of the string. If the last character is b , then change the state to q_6 , replace b by B and move the pointer towards left. The transitions defined for this can be of the form

$$\delta(q_6, b) = (q_4, B, L)$$

At this point, we know that the first character is b and last character is also b . Now, reset the read-write head to point to the first non blank character as shown in step 5.

In state q_6 , If the last character is B(blank character), it means that the given string is an odd palindrome. So, replace B by B, change the state to q_7 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_6, B) = (q_7, B, R)$$

Step 4: In state q_1 , if the first symbol is blank character (B), the given string is even palindrome and so change the state to q_7 , replace B by B and move the read-write head towards right. The transition for this can be of the form

$$\delta(q_1, B) = (q_7, B, R)$$

Step 5: Reset the read-write head to point to the first non blank character. This can be done as shown below. If the first symbol of the string is a , step 2 is performed and if the first symbol of the string is b , step 3 is performed. After completion of step 2 or step 3, it is clear that the first

symbol and the last symbol match and the machine is currently in state q_4 . Now, we have to reset the read-write head to point to the first nonblank character in the string by repeatedly moving the head towards left and remain in state q_4 . During this process, the symbols encountered may be a or b or B (blank character). Replace a by a , b by b and move the pointer towards left. The transitions defined for this can be of the form

$$\delta(q_4, a) = (q_4, a, L)$$

$$\delta(q_4, b) = (q_4, b, L)$$

But, if the symbol B is encountered, change the state to q_1 , replace B by B and move the pointer towards right. The transition defined for this can be of the form

$$\delta(q_4, B) = (q_1, B, R)$$

After resetting the read-write head to the first non-blank character, repeat through step 1.

So, the TM to accept strings of palindromes over $\{a, b\}$ is given by

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

where

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$$

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a, b, B\}$$

q_0 is the start state

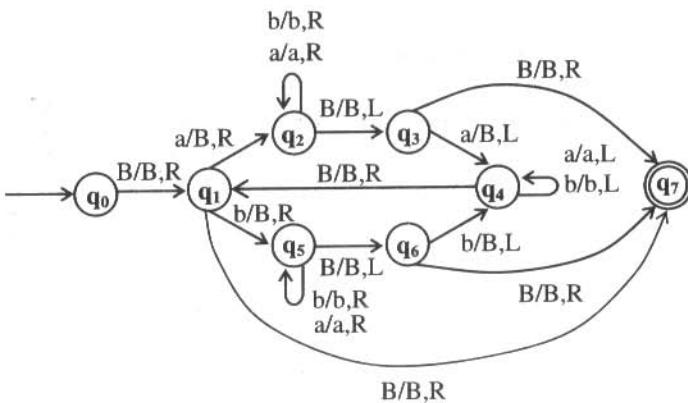
B is the blank character

$$F = \{q_7\}$$

δ is shown below using the transition table:

	δ	Γ		
		a	b	B
\rightarrow	q_0	-	-	q_1, B, R
	q_1	q_2, B, R	q_5, B, R	q_7, B, R
	q_2	q_2, a, R	q_2, b, R	q_3, B, L
	q_3	q_4, B, L	-	q_7, B, R
	q_4	q_4, a, L	q_4, b, L	q_1, B, R
	q_5	q_5, a, R	q_5, b, R	q_6, B, L
	q_6	-	q_4, B, L	q_7, B, R
	$*q_7$	-	-	-

The transition diagram to accept palindromes over $\{a, b\}$ is given by



The reader can trace the moves made by the machine for the strings abba, aba and aaba and is left as an exercise.

Example 7.10: Obtain a TM to accept the language $L = \{ww^R \mid w \in (a+b)^*\}$.

Note: ww^R is nothing but a palindrome but of even length. So, it is same as the previous problem except that from states q_3 and q_6 on B no transitions are defined as shown below:

δ	Γ		
	a	b	B
q_0	-	-	q_1, B, R
q_1	q_2, B, R	q_5, B, R	q_7, B, R
q_2	q_2, a, R	q_2, b, R	q_3, B, L
q_3	q_4, B, L	-	
q_4	q_4, a, L	q_4, b, L	q_1, B, R
q_5	q_5, a, R	q_5, b, R	q_6, B, L
q_6	-	q_4, B, L	
q_7	-	-	-

7.7. Transducers

A transducer accepts some input and transform that input into the desired output. In this sense, the TM can be called as a transducer. The primary purpose of any computer is to accept some input and transform into the desired output. Using Turing machines, an abstract model of a digital computer can be obtained. The input for Turing machine will be the non-blank symbols on the tape and after processing, the output will be the symbols on the tape. So, the transducer for a Turing machine is a function f defined by

$$f(w) = w'$$

where w is the input before computation and w' is the output after computation such that

$$q_0 w \stackrel{*}{\vdash} q_f w' \text{ for } q_f \in F$$

❖ **Definition:** Let $M = (Q, \Sigma, \delta, q_0, B, F)$ be a Turing machine. The function f is Turing computable (also called computable) if and only if

$$q_0 w \stackrel{*}{\vdash} q_f w' \text{ for } q_f \in F \text{ and } w \in \Gamma^*$$

The arithmetic operations such as addition, subtraction etc. including the common mathematical functions are Turing computable. Some of the operations covered are:

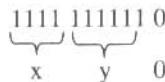
- Addition
- Concatenation of two strings
- Arithmetic comparison

■ **Example 7.11:** Let x and y are two positive integers. Obtain a Turing machine to perform $x + y$.

Let us see how to represent positive integers. We know that binary digits are 0 and 1. On similar lines, we can have a unary number which is made up of only one digit. Let us assume that 1 is the unary digit. So, a number is made up of only 1's. Let x and y are two unary numbers over $\{1\}^*$. Assume that both the unary integers x and y are stored on the tape one after the other separated by a 0. For example, if x is 1111 and y is 111111 then store x on the tape, end with a 0 and then store integer y as shown below:

1111 0 111111
 x 0 y

If this is the input to the transducer, the output should be of the form:



In general the moves made by the Turing machine should be of the form

$$q_0 x 0 y \xrightarrow{*} q_f x y 0$$

where q_f is the final state.

It is clear from the problem definition that to solve this problem the following steps are performed:

General Procedure

Keep updating the pointer till a 0 is encountered. Replace the symbol 0 by 1 and move till the last 1 is reached. Replace last 1 by 0 and reset the read-write head to point the first 1 on the tape.

Let q_0 be the start state and assume that the integers x and y are separated by 0 and enclosed between two B's as shown below:

$$Bx0yB$$

and the read-write head points to the first 1 in the integer x . The TM can be constructed as shown below:

Keep updating read-write head till a 0 is encountered. While scanning for a 0, we encounter 1's in x . So, replace 1 by 1 and move the read-write head towards right and stay in the state q_0 . The transition for this can be of the form

$$\delta(q_0, 1) = (q_0, 1, R)$$

On encountering a 0, change the state to q_1 , replace 0 by 1 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_0, 0) = (q_1, 1, R)$$

Now, the read-write head points to the first 1 of integer y . Now, move the read-write head to point to the last 1 of integer y . To achieve this, replace 1 by 1, move the read-write head towards right and stay in q_1 only. The transition for this can be of the form

$$\delta(q_1, 1) = (q_1, 1, R)$$

On encountering B, change the state to q_2 , replace B by B and move the pointer towards left. The transition for this can be of the form

$$\delta(q_1, B) = (q_2, B, L)$$

Now, the read-write head points to the last 1 of integer y. Now, change that 1 to 0, change the state to q_3 and move the head towards left. The transition for this can be of the form

$$\delta(q_2, 1) = (q_3, 0, L)$$

Now, we have the pattern

$xy0$

on the tape. But, we should move the pointer to the first 1 of the integer x. To achieve this, scan each symbol, replace 1 by 1, move the pointer towards left and remain in state q_3 . The transition for this can be of the form

$$\delta(q_3, 1) = (q_3, 1, L)$$

Once the symbol B is encountered, replace B by B, change the state to q_4 and move the read-write towards right. The transition for this is

$$\delta(q_3, B) = (q_4, B, R)$$

So, the TM to achieve $x + y$ is given by

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

where

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$

$$\Sigma = \{1\}$$

$$\Gamma = \{1, 0, B\}$$

q_0 is the start state

B is blank character

$$F = \{q_4\}$$

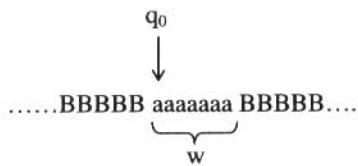
δ is shown below using the transition table:

	δ	Γ		
		1	0	B
\rightarrow	q_0	$q_0, 1, R$	$q_1, 1, R$	-
	q_1	$q_1, 1, R$	-	q_2, B, L
	q_2	$q_3, 0, L$	-	-
	q_3	$q_3, 1, L$	-	q_4, B, R
	$*q_4$	-	-	-

It is left to the reader to take an example and show the sequence of moves made by the TM. By looking at the transitional table we can easily write the transition diagram which is also left to the reader as an exercise.

■ Example 7.12: Given a string w , design a TM that generates the string ww where $w \in a^*$.

Let q_0 be the start state and assume that the string w is enclosed between infinite number of B's and the read/write head points to the first a of the string w as shown below:



The general procedure to concatenate the string w with itself is shown below:

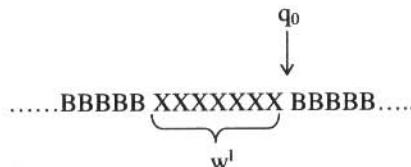
1. Replace each symbol in w with x .
2. Find the rightmost x .
3. Replace rightmost x by the symbol a .
4. Move to the right of rightmost a and replace it by a .
5. Find the rightmost x .
6. Repeat through step 3 till we find no more x 's.

Let us obtain the transitions for each of the steps shown above:

Step 1: Replace each symbol in w with x . This can be easily done by replacing each a by the symbol X and then move the read/write head towards right till we get the symbol 'B'. The transitions defined to achieve this are:

$$\delta(q_0, a) = (q_0, X, R)$$

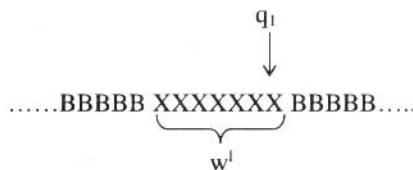
The contents of the tape and position of read/write head after applying these transitions will be



Step 2: Find the rightmost x. This is achieved only after all a's are replaced by X's as shown in figure above. In state q_0 , once we encounter the symbol 'B' as the input, change the state to q_1 , replace B by B and move the pointer towards left. The corresponding transition will be

$$\delta(q_0, B) = (q_1, B, L)$$

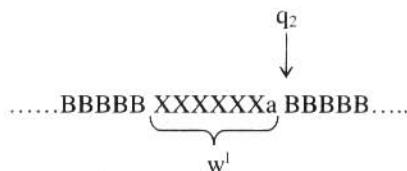
Now, the pointer points to the rightmost X as shown in the figure below:



Step 3: Replace rightmost x by the symbol a. Since the pointer points to the rightmost x, replace this X by a, change the state to q_2 and move the pointer towards right. The transition for this will be

$$\delta(q_1, X) = (q_2, a, R)$$

The contents of the tape and position of read/write head after applying these transitions will be



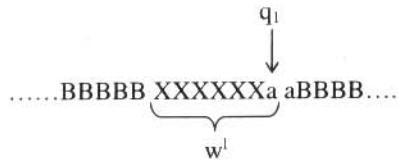
Step 4: Move to the right of rightmost a and replace it by a. This can be achieved by repeatedly replacing they symbol a by a, remain in the same state q_2 and move the pointer towards right using the transition

$$\delta(q_2, a) = (q_2, a, R)$$

and when the symbol B is encountered, change the state to q_1 , replace B by a and move the pointer towards left using the transition

$$\delta(q_2, B) = (q_1, a, L)$$

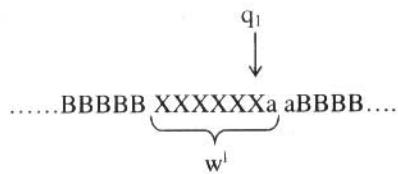
The contents of the tape and position of read/write head after applying these transitions will be



Step 5: Find the rightmost x. Now, to get the rightmost X, as we encounter a in the input, remain state q_1 , replace a by a and move the pointer towards left. The transition for this will be

$$\delta(q_1, a) = (q_1, a, L)$$

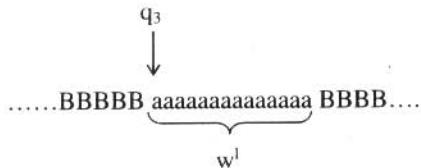
The contents of the tape and position of read/write head after applying these transitions will be



Step 6: Repeating the steps through step 3, there will not be any more X's and the left of the left-most 1 will be B. Once this B is encountered, change the state to q_3 which is the final state, replace B by B and move the pointer towards right. The transition will be

$$\delta(q_1, B) = (q_3, B, R)$$

The final contents of the tape and position of read/write head is shown below:



So, given the string w , the TM to obtain the string ww is given by

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

where

$$Q = \{q_0, q_1, q_2, q_3\}$$

$$\Sigma = \{a\}$$

$$\Gamma = \{a, X, B\}$$

q_0 is the start state

B is blank character

$$F = \{q_3\}$$

δ is shown below using the transition table:

	δ	Γ		
		a	X	B
\rightarrow	q_0	(q_0, X, R)	-	(q_1, B, L)
	q_1	(q_1, a, L)	(q_2, a, R)	(q_3, B, R)
	q_2	(q_2, a, R)	-	(q_1, a, L)
	$*q_3$	-	-	-

It is left to the reader to take an example and show the sequence of moves made by the TM. By looking at the transitional table we can easily write the transition diagram which is also left to the reader as an exercise.

Example 7.13: Construct a TM that stays in the final state q_f whenever $x \geq y$ and non-final state q_n whenever $x < y$ where x and y are positive integers represented in unary notation.

Let q_0 be the start state and assume the two unary integers x and y are separated by #. Also, assume the string $x\#y$ is enclosed between B's as shown below:

Bx#yB

The initial ID will be of the form

B $q_0x\#yB$

with the read/write head pointing to the first leftmost digit of x and the final configuration will be either

B $q_fx\#yB$ whenever $x \geq y$

or

B $q_nx\#yB$ whenever $x < y$

In other words,

B $q_0x\#yB \xrightarrow{*} Bq_fx\#yB$ whenever $x \geq y$

B $q_0x\#yB \xrightarrow{*} Bq_nx\#yB$ whenever $x < y$

Note: While designing the TM for the language $L = a^n b^n$ in Example 7.2, each leftmost symbol a was matched with leftmost symbol b . On similar lines we can solve this problem also.

General procedure: Let q_0 be the start state and let the read-write head points to the first digit of integer x . The general procedure to design TM for this case is shown below:

1. Replace the left most digit of integer x by X and then move the read/write head till we get the leftmost digit of integer y .
2. Replace it by X and move towards left till the leftmost 1 of integer x is obtained.
3. Repeat through step 1 till one of the condition is satisfied.
4. No more 1's in integer x and y .
5. More 1's in integer x which results in no 1's in integer y .
6. More 1's in integer y which results in no 1's in integer x .

So, final contents of the tape will have one of

1. XXXXX#XXXXXB whenever $x = y$ with output q_f .
2. XXXXX11#XXXXXB whenever $x > y$ with output q_f .
3. XXXXX#XXXXX11B whenever $x < y$ with output q_n .

If the condition shown in step 3.a or 3.b is encountered, change the state to q_f ; Otherwise change the state to q_n . Now, consider the situation

XX11#XX11B
↑
 q_0

where first two 1's of integer x are replaced by X 's and first two 1's of integer y are also replaced by X 's. In this situation, the read-write head points to the left most 1 of integer x and the machine is in state q_0 . With this as the configuration, now let us design the TM.

Step 1: In state q_0 , when the digit 1 is encountered, change the state to q_1 , replace 1 by X and move pointer towards right using the transition

$$\delta(q_0, 1) = (q_1, X, R)$$

But, in state q_0 , if the symbol # is encountered, it means that there are no 1's in integer x and change the state to q_5 using the transition

$$\delta(q_0, \#) = (q_5, \#, R)$$

Step 2: In state q_1 , the pointer should move towards right till # is encountered and then change the state to q_2 which can be done using the transitions

$$\delta(q_1, 1) = (q_1, 1, R)$$

$$\delta(q_1, \#) = (q_2, \#, R)$$

Step 3: In state q_2 , TM may encounter X's when searching for leftmost 1 in y. If so, the TM should replace X by X and move the pointer towards right using

$$\delta(q_2, X) = (q_2, X, R)$$

Once the machine encounters 1, it should change the state to q_3 , replace 1 by X and move the pointer towards left using

$$\delta(q_2, 1) = (q_3, X, L)$$

But, if the symbol B is encountered, it means that $x > y$ and the machine should enter into final state q_6 using the transition

$$\delta(q_2, B) = (q_6, B, L)$$

and finally from q_6 we can enter into state q_f .

Step 4: In state q_3 , every X should be replaced by X and the pointer should be moved towards left using the transition

$$\delta(q_3, X) = (q_3, X, L)$$

But, once # is encountered, move the pointer towards left, change the state to q_4 to search for leftmost 1 in x using the transition

$$\delta(q_3, \#) = (q_4, \#, L)$$

Step 5: In state q_4 , replace 1 by 1 and move towards left using

$$\delta(q_4, 1) = (q_4, 1, L)$$

But, once X is encountered, change the state to q_0 , replace X by X and move the pointer towards right using

$$\delta(q_4, X) = (q_0, X, R)$$

Step 6: Whenever the machine is in q_5 , it means that there are no 1's in x. If there are no 1's in y, then y will have only X's followed by B in that case the machine should enter into state q_6 . The transitions will be

$$\delta(q_5, X) = (q_5, X, R)$$

$$\delta(q_5, B) = (q_6, B, L)$$

and from state q_6 we can reach the final state q_f . But, in state q_5 , if 1's are encountered, it means that $x < y$ and the machine goes to state q_7 from which non-final state q_n is reached using the transition

$$\delta(q_5, 1) = (q_7, 1, L)$$

Step 7: From state q_6 , the machine should enter into final state q_f and the pointer should point first digit of x which can be done using

$$\delta(q_6, 1) = (q_6, 1, L)$$

$$\delta(q_6, X) = (q_6, X, L)$$

$$\delta(q_6, \#) = (q_6, \#, L)$$

$$\delta(q_6, B) = (q_f, B, R)$$

Step 8: From state q_7 , the machine should enter into final state q_n and the pointer should point first digit of x which can be done using

$$\delta(q_7, X) = (q_7, X, L)$$

$$\delta(q_7, \#) = (q_7, \#, L)$$

$$\delta(q_7, B) = (q_n, B, R)$$

So, the final TM is given by

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

where

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_n, q_f\}$$

$$\Sigma = \{1, \#\}$$

$$\Gamma = \{1, X, \#, B\}$$

q_0 is the start state

B is blank character

$$F = \{q_f, q_n\}$$

δ is shown below using the transition table:

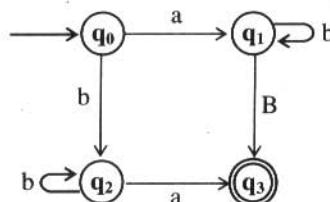
	δ	Γ			
		1	#	X	B
\rightarrow	q_0	(q_1, X, R)	($q_5, \#, R$)	-	-
	q_1	($q_1, 1, R$)	($q_2, \#, R$)		
	q_2	(q_3, X, L)	-	(q_2, X, R)	(q_6, B, L)
	q_3	-	($q_4, \#, L$)	(q_3, X, L)	-
	q_4	($q_4, 1, L$)	-	(q_0, X, R)	-

	q_5	$(q_7, 1, L)$	-	(q_5, X, R)	(q_6, B, L)
	q_6	$(q_6, 1, L)$	$(q_6, \#, L)$	(q_6, X, L)	(q_f, B, R)
	q_7	-	$(q_7, \#, L)$	(q_7, X, L)	(q_n, B, R)
	$*q_n$	-	-	-	-
	$*q_f$	-	-	-	-

Example 7.14: What language is accepted by the machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where $Q = \{ q_0, q_1, q_2, q_3 \}$, $\Sigma = \{ a, b \}$, $\Gamma = \{ a, b, B \}$, q_0 is start state, B is blank character, $q_f = \{ q_3 \}$ where δ is defined as follows:

$$\begin{aligned}\delta(q_0, a) &= (q_1, a, R) \\ \delta(q_0, b) &= (q_2, b, R) \\ \delta(q_1, b) &= (q_1, b, R) \\ \delta(q_1, B) &= (q_3, B, R) \\ \delta(q_2, b) &= (q_2, b, R) \\ \delta(q_2, a) &= (q_3, a, R)\end{aligned}$$

Note: It is clear from the TM that the movement of the read/write pointer is only towards right and it can be compared with the FA. So, the equivalent FA for these transitions can be written as



It is clear from the graph that to reach the final state the FA can take only two paths yielding the language containing either bb^*a or ab^* (leaving B as it is not the input symbol) which can be defined as

$$L = \{ bb^*a + ab^* \}$$

which is the string consisting of at least one b followed by one a or a single a followed by zero or more b 's.

Exercises

1. Explain the Turing machine model.
2. Define Turing machine.

3. What is an ID with respect to TM?
4. Define move of a TM.
5. What language is accepted by TM?
6. What is a recursively enumerable language?
7. Obtain a Turing machine to accept the language $L = \{0^n 1^n \mid n \geq 1\}$.
8. Obtain a Turing machine to accept the language $L(M) = \{0^n 1^n 2^n \mid n \geq 1\}$.
9. Obtain a TM to accept the language $L = \{w \mid w \in (0+1)^*\}$ containing the sub string 001.
10. Obtain a Turing machine to accept the language containing strings of 0's and 1's ending with 011.
11. Obtain a Turing machine to accept the language $L = \{w \mid w \text{ is even and } \Sigma = \{a,b\}\}$.
12. Obtain a Turing machine to accept a palindrome consisting of a's and b's of any length.
13. On what basis we say that TM is a transducer?
14. What is Turing computable?
15. Let x and y are two positive integers. Obtain a Turing machine to perform $x + y$.
16. Given a string w , design a TM that generates the string ww where $w \in a^*$.
17. Construct a TM that stays in the final state q_f whenever $x \geq y$ and non-final state q_n whenever $x < y$ where x and y are positive integers represented in unary notation.
18. What language is accepted by the machine $M = (Q, \Sigma, \delta, q_0, B, F)$ where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, B\}$, q_0 is start state, B is blank character, $q_f = \{q_3\}$ where δ is defined as follows:
 - $\delta(q_0, a) = (q_1, a, R)$
 - $\delta(q_0, b) = (q_2, b, R)$
 - $\delta(q_1, b) = (q_1, b, R)$
 - $\delta(q_1, B) = (q_3, B, R)$
 - $\delta(q_2, b) = (q_2, b, R)$
 - $\delta(q_2, a) = (q_3, a, R)$
19. How to achieve complex tasks using TM?
20. Let x and y are two positive integers represented using unary notation. Design a TM that computes the function:

$$f(x, y) = \begin{cases} x + y & \text{if } x \geq y \\ xx & \text{if } x < y \end{cases}$$
21. What are the various variations of TM?

22. Define the following:

- Turing machine with stay-option
- Turing machine with multiple tracks
- Turing machine with semi-infinite tape
- Off-line Turing machine
- Multi-tape Turing machine
- Linear bounded Automaton

23. What is a multi-tape Turing machine? Show how it can be simulated using single tape Turing machine.

Extensions of Turing Machines

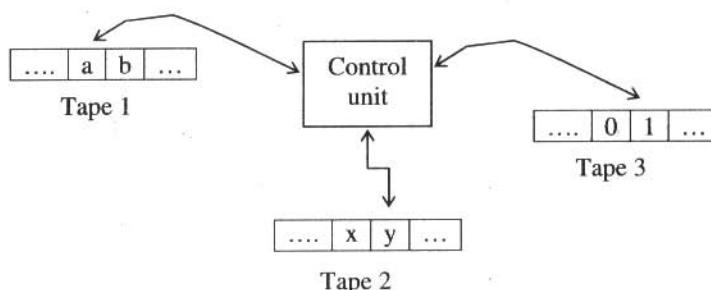
So far in the previous sections, we have discussed the concepts of Standard Turing Machines. Now, we shall concentrate on the variations or extensions of the Standard Turing machine and using simulators we show that the extensions of Turing machines in fact are equivalent to Standard Turing machines. Instead of providing the complete simulation, we shall provide only broad outline to show that the machines are equivalent. We can have so many variations of Standard Turing machines. With minor modification we can have the following Turing machines:

- Multi-tape Turing Machine
- Non-deterministic Turing Machine

This section discusses these two variations of TM. Other variations by imposing certain restrictions (restricted TMs) are discussed in coming sections.

7.8. Multi-tape Turing Machines

A multi-tape Turing Machine is nothing but a standard Turing Machine with more number of tapes. Each tape is controlled independently with independent read-write head. The pictorial representation of multi-tape Turing machine is shown in figure below:



The various components of multi-tape Turing Machine are:

- a) Finite control.
- b) Each tape having its own symbols and read/write head.

Each tape is divided into cells which can hold any symbol from the given alphabet. To start with the TM should be in start state q_0 . If the read/write head pointing to tape 1 moves towards right, the read/write head pointing to tape 2 and tape 3 may move towards right or left depending on the transition. The formal definition of Multi-tape Turing machine can be defined as follows.

❖ **Definition:** The Multi-tape Turing Machine is an n-tape machine

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

where

- Q is set of finite states
- Σ is set of input alphabets
- Γ is set of tape symbols
- δ is transition function from $Q \times \Gamma^n$ to $Q \times \Gamma^n \times \{L, R\}^n$
- q_0 is the start state
- B is a special symbol indicating blank character
- $F \subseteq Q$ is set of final states

The move of the multi-tape TM depends on the current state and the scanned symbol by each of the tape heads. For example, if number of tapes in TM is 3 as shown in the figure and if there is a transition

$$\delta(q, a, b, c) = (p, x, y, z, L, R, S)$$

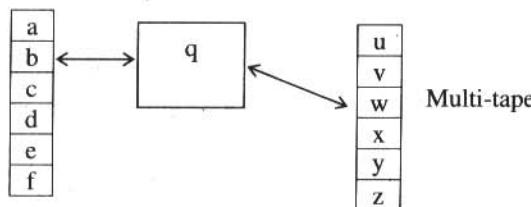
where q is the current state. The transition can be interpreted as follows. The TM in state q will be moved to state p only when the first read/write head points to a , the second read-write head points to b and third read/write head points to c and the read/write head will be moved to left in the first case and right in the second case. But, the read/write head with respect to third tape will not be altered. At the same time, the symbols a , b and c will be replaced by x , y and z . It can be easily shown that the n-tape TM in fact is equivalent to the single tape Standard Turing Machine as shown below.

7.9. Equivalence of Single Tape and Multi-tape TM's

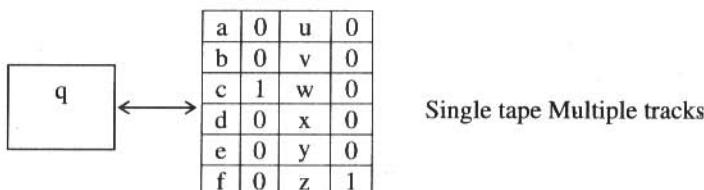
❖ **Theorem:** Every language accepted by a multi-tape TM Is recursively enumerable.

Note: The theorem clearly indicates that every language accepted by a multi-tape TM is also accepted by a standard TM.

Proof: This can be shown by simulation. For example, consider a TM with two tapes as shown below:



The above 2-tape TM can be simulated using single tape TM which has four tracks as shown in figure below:



The first and third tracks consist of symbols from first and second tape respectively. The second and fourth track consists of the positions of the read/write head with respect to first and second tape respectively. The value 1 indicates the position of the read/write head. It is clear from the above figure that, the machine in state q and when the first read/write head points to the symbol c , the second read/write head points to the symbol z , then the machine enters into state p , if and only if this transition is defined for TM with multi-tapes. So, whatever transitions have been applied for multi-tape TM, if we apply the same transitions for the new machine that we have constructed, then the two machines are equivalent.

7.10. Nondeterministic Turing Machines

The difference between nondeterministic TM and deterministic TM lies only in the definition of δ . The formal definition of *nondeterministic* TM is shown below:

❖ **Definition:** The *non-deterministic* Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

Q is set of finite states

Σ is set of input alphabets

Γ is set of tape symbols

δ is transition function from $Q \times \Gamma$ to $2^{Q \times \{L,R\}}$

q_0 is the start state

B is a special symbol indicating blank character

$F \subseteq Q$ is set of final states

It is clear from the definition of δ that for each state q and tape symbol X, $\delta(q,X)$ is a set of triples

$$\{(q_1, X_1, D), (q_2, X_2, D), (q_3, X_3, D), \dots, (q_i, X_i, D)\}$$

where i is a finite integer and D is the direction with 'L' indicating left or 'R' indicating right. The machine can choose any of the triples as the next move. The language accepted by TM is defined as follows.

❖ **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a nondeterministic TM. The language (M) accepted by M is defined as

$$L(M) = \{w \mid q_0 w \vdash^* \alpha_1 p \alpha_2 \text{ where } w \in \Sigma^*, p \in F \text{ and } \alpha_1, \alpha_2 \in \Gamma^*\}$$

The language is thus a set of all those words w in Σ^* which causes M to move from start state q_0 to the final state p .

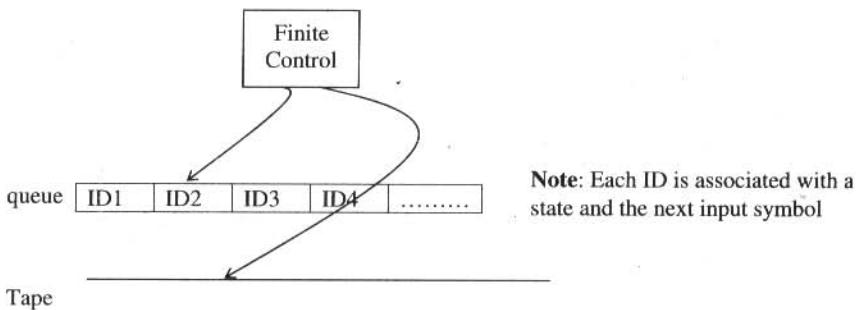
A nondeterministic TM may have many moves that does not lead to a final state. But, we are interested in only those moves that leads to the final states. The nondeterministic TM in fact is no more powerful than the deterministic TM. Any language accepted by nondeterministic TM can be accepted by deterministic and both are equivalent. We can simulate a deterministic TM from a nondeterministic TM as shown below:

Theorem: For every nondeterministic TM (NTM) there exists a deterministic TM (DTM) such that $L(NTM) = L(DTM)$.

Proof : Given a string w , NTM starts at the initial configuration(initial ID) and goes through a sequence of configurations(IDs) until it reaches one of the conditions:

- Final state is reached and the machine halts
- The transition is not defined and the machine halts
- Goes into an infinite loop

To go to the next configuration (ID), the NTM has to choose from a finite set of configurations (IDs). All these configurations (IDs) which can be obtained by NTM for a given string w can be represented by a tree. The way NTM is simulated by DTM is shown using the figure shown below:



It is clear from the figure that all the IDs of NTM (represented by the nodes of the tree) are placed in the queue one after the other. Each ID inserted into the queue is also associated with the state and the next input symbol to be scanned. The first ID to be explored and examined will be at the front end of the queue. Given any ID in the queue, all IDs to the left of designated ID are assumed to be deleted (marked) and the IDs towards right are assumed to be explored or to be examined. For example, if current ID is ID3 then ID1 and ID2 are already explored and we need not consider them whereas the ID4, ID5, etc., are the nodes to be explored later. The steps carried by DTM are shown below:

1. The current ID is examined (For the first time current ID is ID1 and is marked/deleted) based on the state and the scanned symbol. If the state in the current ID is final state then DTM accepts the string and the machine halts.
2. If the state is non final state, the current ID(configuration) is explored and various IDs(configurations) obtained are inserted at the end of the queue.
3. Steps 1 and 2 are repeated until no more ID is examined or queue is empty.

The above steps can be clearly explained using the tree representation. The root node corresponds to the initial configuration (initial ID) and it is the only vertex of level 0. All the configurations (IDs) that are obtained by applying the transition function of NTM only once will be the children of the initial configuration (ID). These new vertices which are derived from the root are at level 1. In general, from the configurations (IDs) at level i , the configurations at level $i+1$ can be obtained. Since the configurations are finite, the number of children at various levels is finite.

The easiest way to simulate NTM using DTM is to traverse this tree using BFS (Breadth-First-Search) from the root until the halt state (the string w is accepted or rejected) is reached. At each level of the tree, the DTM applies the transition function corresponding to the NTM to each configuration (ID) at that level and computes its children (new IDs). These new IDs are the configurations of the next level and they are stored on the tape (if necessary a second tape may be used). If there is a halting state among these children, then DTM accepts the string and halts. It can be easily seen that DTM accepts a string if and only if NTM accepts it. Thus any language accepted by a NTM is also accepted by a DTM.

7.11. Turing Machine with Stay-option

In the Standard Turing Machine, after scanning the symbol and after replacing the symbol on the tape, the read/write head used to move either left or right based on the control mechanism. Apart from having the read/write head either towards left or right, if there is one more option where in the read/write head stays in the same position after updating the symbol on the tape (no movement of read/write head either to the left or right), then the Turing machine is called TM with stay-option. Formally, the machine can be defined as follows:

The formal definition of TM with stay-option is provided below:

❖ **Definition:** The Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

Q is set of finite states

Σ is set of input alphabets

Γ is set of tape symbols

δ is transition function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R, S\}$ indicating the TM may move towards left or right or stay in the same position after updating the symbol on the tape

q_0 is the start state

B is a special symbol indicating blank character

$F \subseteq Q$ is set of final states

Exercises

1. Explain the general structure of Multi-tape machines. Show that they are equivalent to standard Turing machines.
2. Define non-deterministic Turing machine and show that the language accepted by NTM is also accepted by DTM and they are equivalent.
3. Define Turing machine with stay option.

Programming Techniques for Turing Machines

Achieving Complicated Tasks Using TM

In Chapter 6, we have shown how the simple operations such as addition, subtraction, concatenation and comparison can be achieved using TM. These are some of the basic operations found in all the computers. Using these basic operations, more complex operations can be achieved using the computers. On similar lines, more complex operations can be achieved using TM also using the

primitive operations. To achieve this, let us use block diagrams, which contains boxes and arrows representing the action performed by each box and arrows representing flow of control. The functionality of each component in the block diagram is described, but the implementations are hidden or rather assumed to be implemented (in fact we can show how the functionality of each component can be implemented). Let us discuss how more complicated functions are performed by TMs using the primitive operations (which can be considered as subroutines or functions).

7.12. Multiple Tracks (Multi Track)

In the Standard Turing Machine, the tape was divided into squares where each square holds only one symbol. It can be extended so that each tape consist of several tracks. This can be done by dividing the tape into number of tracks and each track is divided into a number of squares with each square holding only one symbol. If there are n -tracks and the read/write head points to m^{th} square, then all the symbols under different tracks beneath that read/write head are the symbols to be scanned. So, all the symbols under the read/write head can be considered as set of characters under one square. Such a TM is called Turing machine with multiple tracks.

7.13. Subroutines

We know that a program consists of zero or more functions (subroutines). On similar lines a Turing Machine can be a collection of zero or more Turing Machine subroutines. A Turing Machine subroutine is a set of states that performs some pre-defined task. The TM subroutine has a start state and a state without any moves. This state which has no moves serves as the return state and passes the control to the state which calls the subroutine.

Example 7.15: Let x and y are two positive integers represented using unary notation. Design a TM that computes the function

$$\begin{aligned} f(x, y) &= x + y && \text{if } x \geq y \\ f(x, y) &= xx && \text{if } x < y \end{aligned}$$

The block diagram to add x and y can be written as shown in Figure 7.1.

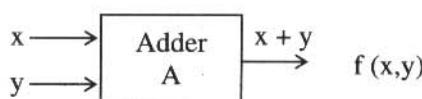


Figure 7.1. Block diagram (subroutine) to add x and y

Here x and y are the inputs to the adder, the output of which will be $x+y$ which can be easily implemented (Example 7.11). The block diagram to concatenate x with x can be written as shown in Figure 7.2.

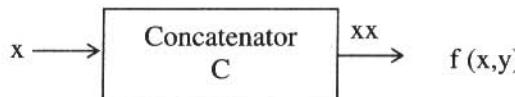


Figure 7.2. Block diagram (subroutine) to concatenate x with x

For this concatenator, x is the only input. The output will be the integer xx which can be implemented (Example 7.12). It is clear from the given function that the addition should be performed if $x \geq y$ and concatenation of x with x has to be performed whenever $x < y$. The comparator also can be implemented as shown in Example 7.13 the block diagram for which can be written as shown in Figure 7.3.

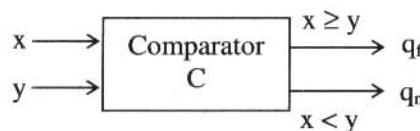


Figure 7.3. Block diagram (subroutine) to compute $x+y$ or xx

Whenever $x \geq y$, the comparator enters into the state q_f which acts as a trigger to invoke the Adder A to add x and y . When $x < y$, the comparator enters into the state q_n which can act as trigger to the concatenator which concatenates x with x . The complete high-level block diagram to compute the function $f(x,y)$ using Turing machine is shown in Figure 7.4.

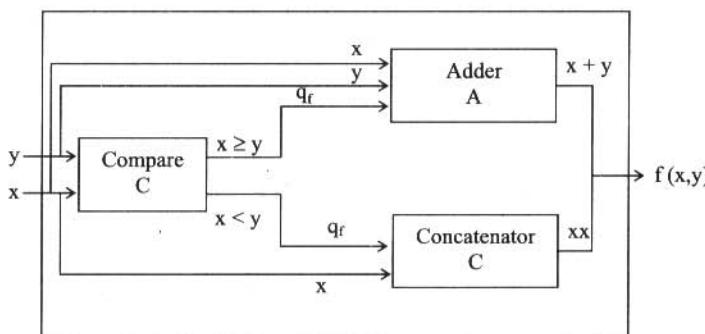


Figure 7.4. Block diagram (subroutine) to compare x and y

Note: The TM can also be represented as the combination of blocks which each block representing a TM which can do some primitive operation (like functions in C). All these TMs that can do some primitive operations can be invoked by another TM, which can be considered as the main program.

Example 7.16: Obtain a TM to multiply two unary numbers separated by the delimiter 1.

Note: Let us assume we have two unary numbers x and y such that x has m number of 0's and y has n number of 0's separated by the delimiter 1 as shown below:

$$0^m 1 0^n$$

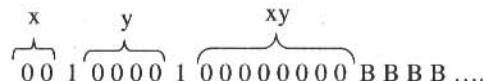
The product of x and y should be stored on the tape and the original numbers should not be destroyed. This can be visualized as shown below:

$$0^m 1 0^n 1 0^{mn}$$

To start with let us assume $x = 00$ and $y = 0000$ and are separated by delimiter 1. Assume y ends with 1 which can act as the delimiter for the input string which in turn is followed by infinite number of blanks (B's) as shown below:



The output should be of the form



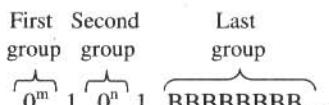
Note: It is clearly visible from the above figure that the unary number y is copied two times (equal to the number of 0's in the number x) in the result xy (which is 00000000). So, to start with we think of how to copy the unary number y once so that we can call this TM (which is represented as a subroutine) repeatedly m (the number of 0's in x) times to achieve the result.

General Procedure (Algorithm)

1. To start with let us assume a tape will have a string of the form

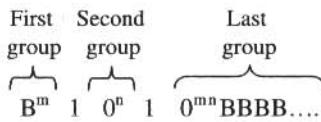
$$0^m 1 0^n 1 BBBBBBBB \dots$$

which is divided into three groups as shown below:



where $x = 0^m$ and $y = 0^n$ and are separated by the delimiter 1. So, if the input is $0^m 1 0^n$ then this string is followed by 1 (which acts as delimiter between the input and the result. Note that all B's will be replaced by the result 0^{mn} later). Now, change a 0 in the first group (x) to B.

2. Copy n number 0's from the second group to the last group by replacing n number of B's by n number of 0's.
3. It is clear from second step that "we copy a group of n 0's (replace n B's) to the last group whenever a 0 is the first group is changed to B". When all 0's in the first group are changed to B's there will definitely be mn number of 0's in the last group (which is the product of first and second group i.e., product of x and y).
4. Once first three steps are completed, the contents of the tape will be of the form



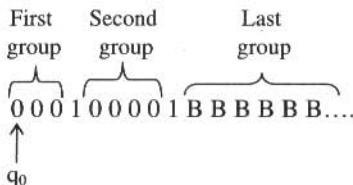
So, by replacing the string $10^n 1$ which is enclosed between the first group and the last group by B's we will have only 0^{mn} number of 0's on the tape with ID $q_0 0^{mn}$ which is the result.

Now, we shall implement all these steps one after the other in detail.

Step 1 (details): To implement step 1 in detail, let us take a string of the form:

000100001BBBBBB....

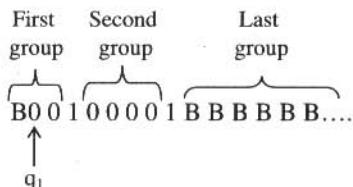
where the first group, second group and last group are identified as shown below:



Assume q_0 is the start state. As per the algorithm in step 1, replace 0 by B, change the state to q_1 and move the head towards right and the corresponding transition for this will be of the form:

$$\delta(q_0, 0) = (q_1, B, R)$$

After applying the above transition, the situation will be of the form:



Now, we should copy n 0's from the second group to the last group. This can be achieved by moving the pointer head towards right till we encounter 1 repeatedly using the transition

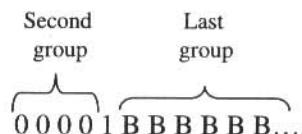
$$\delta(q_1, 0) = (q_1, 0, R)$$

On encountering 1, change the state to q_2 and the corresponding transition is

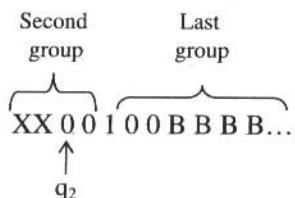
$$\delta(q_1, 1) = (q_2, 1, R)$$

Now, the pointer points to first zero in the second group and we shall copy n 0's from the second group to the last group. The details are shown in step 2.

Step 2 (details): (Copy function) This second step of the algorithm (procedure) can be implemented using a subroutine called *copy*. Now, let us think of how to replace n number of B's in the last group to n number of 0's from the second group. For this reason, let us take only the second and last group as shown below:



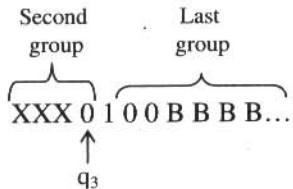
We have to copy n number of 0's from the second group to the last group. This can be achieved by replacing a B by 0 in the last group immediately after changing a 0 to X in the second group. To obtain the required transitions let us assume the situation shown below:



In this situation, two leftmost 0's in second group are replaced by X and the corresponding two leftmost B's are replaced by two 0's in the last group. Let us assume the machine is in state q_2 and the next symbol to be scanned is the symbol pointed to by q_2 . It is clear from the above figure that in state q_2 on input 0, change the state to q_3 , replace 0 by X and move the pointer towards right using the transition

$$\delta(q_2, 0) = (q_3, X, R)$$

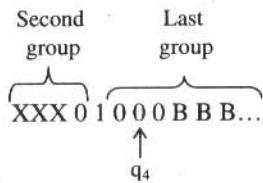
After applying the above transition, the situation is shown below:



Now, in state q_3 on any number of 0's or 1's we can stay in q_3 and simply move the head towards right. But, on encountering a B, change the state to q_4 , replace B by 0 and move the head towards left (to replace leftmost zero by X in the second group) using the following transitions:

$$\begin{aligned}\delta(q_3, 0) &= (q_3, 0, R) \\ \delta(q_3, 1) &= (q_3, 1, R) \\ \delta(q_3, B) &= (q_4, 0, L)\end{aligned}$$

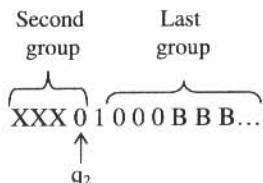
After these transitions, the situation will be of the form shown below:



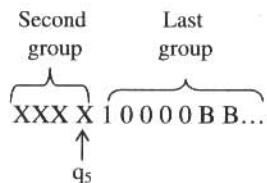
In state q_4 , we should search for rightmost X (to get leftmost 0). So, keep updating the head towards left on encountering 0's or 1's and on encountering an X, change the state to q_2 , replacing X by X and move the pointer towards right. The corresponding transitions are:

$$\begin{aligned}\delta(q_4, 0) &= (q_4, 0, L) \\ \delta(q_4, 1) &= (q_4, 1, L) \\ \delta(q_4, X) &= (q_2, X, R)\end{aligned}$$

After applying the above transitions, the situation is shown below:



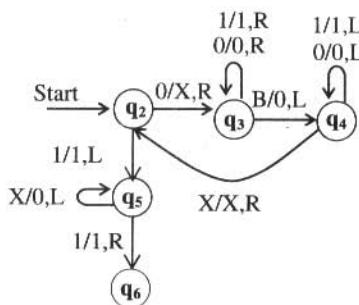
It is clear from this situation shown in above figure that, if there are only X's in second group (means no 0's left), there will be a transition from state q_2 on 1 which implies that n number of B's in the last group are replaced by n number of 0's. So, q_2 on 1 we move towards left and change the state to q_5 as shown in the situation below using the transition $\delta(q_2, 1) = (q_5, 1, L)$:



In state q_5 we should see that all X's are replaced by 0's. So, when we move towards left, we may encounter a delimiter 1. In that case, we simply move the head towards right by changing the state to q_6 . The corresponding transitions are:

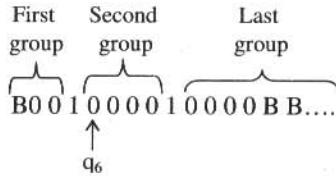
$$\begin{aligned}\delta(q_5, X) &= (q_5, 0, L) \\ \delta(q_5, 1) &= (q_6, 1, R)\end{aligned}$$

Now, step 2 of the general procedure is completed when a leftmost 0 in first group is replaced by B and the corresponding transition diagram is shown below:



Using the above transition diagram, we can have the actual C function to implement copy function and we call this function as the “Copy function”.

Step 3 (details): Once step 2 is completed the current contents of tape will be of the form shown below:



It is clear from this figure that, when leftmost 0 in first group is replaced by a B, n 0's from the second group have been copied into last group. Now, we should replace the next leftmost 0 in first group to B and repeat the process again. So, q_6 on 0, change the state to q_7 and move the head towards left using the transition

$$\delta(q_6, 0) = (q_7, 0, L)$$

In state q_7 , on 1 change the state to q_8 and move the head towards left using the transition

$$\delta(q_7, 1) = (q_8, 1, L)$$

In state q_8 on input zero move the head towards left using the transition

$$\delta(q_8, 0) = (q_9, 0, L)$$

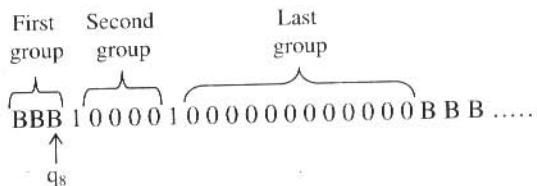
In state q_9 on any number of 0's move the head towards left using the transition

$$\delta(q_9, 0) = (q_9, 0, L)$$

But, if we encounter a B, change the state to q_0 and move the pointer towards right using the transition

$$\delta(q_9, B) = (q_0, B, R)$$

The purpose of the states q_7 , q_8 and q_9 is to take control after copying a block of n 0's from the second group to the last group so as to obtain the leftmost zero in first group. But, in state q_8 on encountering B, it means that n 0's have been copied from the second group to the last group in number of times and the situation is shown below:



Now, let us replace the delimiter 1 which precede and follow the second group including the second group by B's. This is possible using the transitions shown below:

$$\delta(q_8, B) = (q_{10}, B, R)$$

$$\delta(q_{10}, 1) = (q_{11}, B, R)$$

$$\delta(q_{11}, 0) = (q_{11}, B, R)$$

$$\delta(q_{11}, 1) = (q_{12}, B, R)$$

So, the Turing machine to accept the given language is given by

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

where

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}\}$$

$$\Sigma = \{0, 1\}, \Gamma = \{0, 1, X, B\}$$

$q_0 \in Q$ is the start state of machine

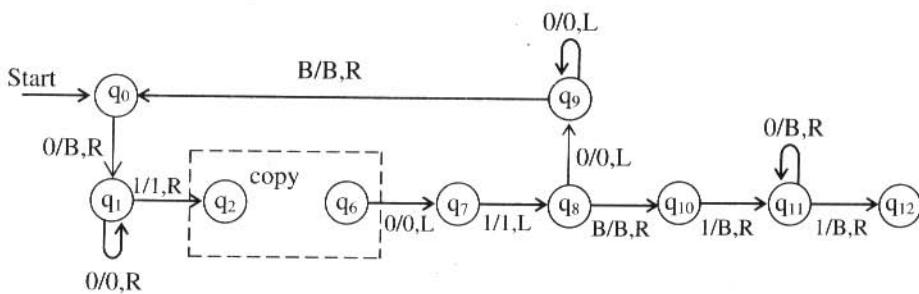
$B \in \Gamma$ is the blank symbol

$$F = \emptyset$$

δ is shown below:

$$\begin{aligned}\delta(q_0, 0) &= (q_1, B, R) \\ \delta(q_1, 0) &= (q_1, 0, R) \\ \delta(q_1, 1) &= (q_2, 1, R) \\ \delta(q_2, 0) &= (q_3, X, R) \\ \delta(q_3, 0) &= (q_3, 0, R) \\ \delta(q_3, 1) &= (q_3, 1, R) \\ \delta(q_3, B) &= (q_4, 0, L) \\ \delta(q_4, 0) &= (q_4, 0, L) \\ \delta(q_4, 1) &= (q_4, 1, L) \\ \delta(q_4, X) &= (q_2, X, R) \\ \delta(q_5, X) &= (q_5, 0, L) \\ \delta(q_5, 1) &= (q_6, 1, R) \\ \delta(q_6, 0) &= (q_7, 0, L) \\ \delta(q_7, 1) &= (q_8, 1, L) \\ \delta(q_8, 0) &= (q_9, 0, L) \\ \delta(q_9, 0) &= (q_9, 0, L) \\ \delta(q_9, B) &= (q_{10}, B, R) \\ \delta(q_{10}, B) &= (q_{10}, B, R) \\ \delta(q_{10}, 1) &= (q_{11}, B, R) \\ \delta(q_{11}, 0) &= (q_{11}, B, R) \\ \delta(q_{11}, 1) &= (q_{12}, B, R)\end{aligned}$$

The corresponding transition diagram is shown below:



Restricted Turing Machines

So far in the previous sections, we have discussed various concepts of Standard Turing Machines and other variations of TM. Now, we shall concentrate on the other variations of TM by imposing certain restriction on TM. Instead of providing the complete simulation, we shall provide only

broad outline to show that the machines are equivalent. We can have so many variations of Standard Turing machines. With minor modification we can have the following Turing machines:

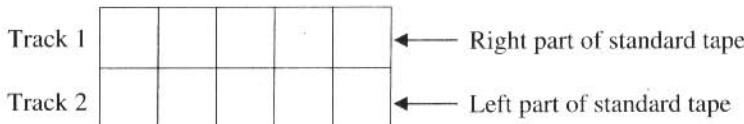
- Turing machine with semi-infinite tape
- Multi-stack Machines
- Counter Machines
- Off-line Turing machine
- Linear bounded automata

7.14. Turing Machine with Semi-infinite Tape

In the Standard Turing Machine, there was no boundary specified for the left side as well as right side of the tape. The read/write head can be moved infinitely towards left as well as towards right i.e., the tape was unbounded in both the directions. Now, if we restrict the read/write head to move only in one direction say towards right (i.e., bounded on the left and unbounded on the right), then the Turing machine is called TM with semi-infinite tape and it can be visualized as shown in figure below:



It is clear from the above figure that there are no cells to the left hand side of the initial head position and this restriction does not affect the power of the machine. The TM M can simulate the semi-infinite TM M_S using two tracks for a tape as shown below:

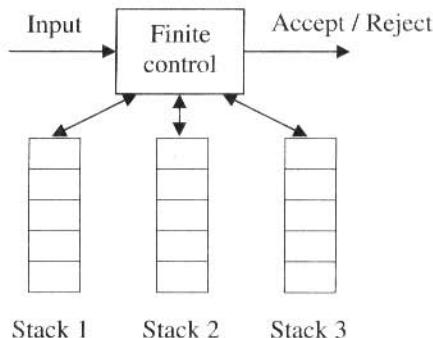


The upper part represents track 1 and in these cells let us store the information which lies to the right of some reference point. Initially, the reference point could be the position of the read-write head. The lower part, i.e., track 2 contains the left part of the reference point in reverse order. The machine M which is simulating M_S will use the information of track 1 as long as the read-write head of the M_S is towards right of the reference point. When M_S moves towards left of the reference point, the machine M uses cells of track 2 from right to left. We can partition the states

of M into Q_1 and Q_2 so that the states of Q_1 are used when working with respect to track 1 and the states of Q_2 are used when working with respect to track 2. Some special markers such as $\#$ s can be placed at the left hand side of track 1 and track 2 to enable the programmer to switch between the tracks. Following in this direction, we can show that M and M_S are equivalent.

7.15. Multi-stack Machines

All the languages which are accepted by PDA are accepted by TM. At the same time, any language which is not accepted by PDA is also accepted by TM. TM is a generalized machine which can accept all languages. The multi-stack machine is generalized model of PDA. A machine with three stacks is shown below:



The pictorial representation of 3-stack machine will have 3 stacks. In general, a k -stack machine is a deterministic PDA with k stacks. Similar to the PDA, it accepts input chosen from the alphabets Σ and has a finite control. All the k stacks will have the alphabets chosen from stack alphabet Γ . The move of the multi-stack depends on:

1. The current state of the finite control
2. The input symbol chosen from Σ
3. The symbol on top of each stack

Once the transition is applied for the current state, input symbol and the current top of each stack, the multistack machine may:

1. Change the state (or it may remain the same state)
2. It may replace the top of the stack

Thus, the general transition for a k-stack machine may take the form:

$$\delta(q, a, X_1, X_2, \dots, X_k) = (p, \alpha_1, \alpha_1, \alpha_1, \dots, \alpha_k)$$

where X_i is on top of the stack for $1 \leq i \leq k$. It is clear from this transition that each symbol on top of the stack can be replaced by different symbols. It can be shown that any language accepted by Turing machine, is also accepted by two-stack machine.

7.16. Counter Machines

The counter machine is a restricted multi-stack machine which can be interpreted in several ways:

The counter machine is similar to that of multi-stack machine with some modifications. Each stack in the machine is replaced by a counter. Each counter holds a non-negative number. The move of a machine depends on the current state, current input symbol and if one of the value of the counter is zero. The machine can do the following activities in one move:

- a) It can change the state.
- b) It can add or subtract 1 from the counters. But, if the counter is zero, subtracting 1 from that counter is not possible.

Some observations of the languages accepted by counter machines are shown below:

1. The languages accepted by counter machines are recursively enumerable i.e., we can obtain an equivalent TM to accept those languages which are accepted by counter machines.
2. A language accepted by only one counter machine is context free language.

7.17. Off-line Turing Machine

In the Standard Turing machine we have assumed that the tape has both the input and output. To start with contents of the tape are assumed to be input symbols and once the TM reaches the final state, the contents of the tape are considered as the output. Now, if we have a separate input buffer as we had in case of finite automaton, we get Off-line Turing machine. But, only difference is that the move of TM is now depends on the state the machine currently in, the current symbol read from the input buffer and the symbol which is currently pointed by read/write head. The formal definition of Off-line Turing machine is shown below:

- ❖ **Definition:** The Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

Q is set of finite states

Σ is set of input alphabets

Γ is set of tape symbols

δ is transition function from $Q \times \Sigma \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$

q_0 is the start state

B is a special symbol indicating blank character

$F \subseteq Q$ is set of final states

7.18. Linear Bounded Automata (LBA)

In the previous sections, we have seen that the power of Turing Machine can not be extended beyond the power of Standard Turing Machine by complicating the TM with multiple tapes or by using multi-dimensional tapes. Instead, the power of TM can be restricted by restricting the tape usage. An example of this is the Push Down Automaton which can be considered as a non-deterministic Turing machine. In PDA, it can be assumed that the tape is used like a stack. We can also assume finite portion of the tape as input that leads to finite automaton. With slight alteration in usage of the tape, let us restrict the workspace on the tape using two delimiters '[' and ']'. The given string has to be enclosed between these two limiters. So, longer the string, longer the workspace. This leads to another class of machine called Linear Bounded Automaton (LBA). So, linear bounded automaton is a TM which is bounded based on the length of the input string. Thus, using TM and by restricting the usage of tape, we can obtain LBA, we construct PDA and finite automaton. It is clear from all these types of machines that TM is superset of all these machines. The formal definition of LBA is provided below:

❖ **Definition:** The Linear Bounded Automaton is a TM

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

where

Q is set of finite states

Σ is set of input alphabets which also has two special symbols '[' and ']'

Γ is set of tape symbols

δ is transition function from $Q \times \Gamma$ to $Q \times \Gamma^{2QXTX(L,R)}$ with two more transitions of the form $\delta(q_i, [) = (q_j, [R)$ and $\delta(q_i,]) = (q_j,]L)$ forcing the read/write head to be within the boundaries '[' and ']'

q_0 is the start state

B is a special symbol indicating blank character

$F \subseteq Q$ is set of final states

It is clear from the definition that the read/write head cannot go out of the boundaries specified as '[' and ']'. Now, the string can be accepted by LBA only if there is a sequence of moves such that

$$q_0[w] \xrightarrow{*} [xq_fz]$$

for some $q_f \in F$ and $x, z \in \Gamma^*$.

Exercises

Write short notes on the following variations of TM

- Turing machine with stay-option
- Turing machine with multiple tracks
- Turing machine with semi-infinite tape
- Off-line Turing machine
- Multi-tape Turing machine
- Linear bounded Automaton

Chapter 8

Undecidability

What are we studying in this chapter . . .

- ▶ A Language that is not recursively enumerable
- ▶ An Un-decidable problem that is RE
- ▶ Post's Correspondence problem
- ▶ Other un-decidable problems.

8.1. Introduction

A machine may accept a language or it may not accept(reject) a language. So, the output of the machine may be *accept* or *reject*. This problem is called *membership problem*. Formally, the membership problem can be stated as “Given a machine M and a string x , does M accept x ?”. The output will be *yes/no*. Given a language, the machine may have to identify whether the language is *finite* or *infinite*. All such problems with two answers *yes/no*, *accept/reject*, *finite/infinite* are called **decision problems**. For majority of the decision problems, we can design decision algorithms.

According to Church-Turing thesis, an appropriate way to formulate the idea of a decision algorithm precisely is to use a Turing machine. For some decision problems we can write the algorithms for any given specific instance (which indicates that the problem can be solved using TM), but it is not possible to write a general decision algorithm that works for any given instance,

which indicates that a Turing machine does not exist to solve a decision problem. Thus, there are problems that are solvable by TM and that are not solvable by TM. With respect to this, we can divide problems into two groups:

1. The problems for which the solution exists in the form of algorithms i.e., If there is an algorithm to solve a problem, there exists a Turing machine that halts whether the input is accepted or rejected.
2. The problems that run forever. i.e., a Turing machine will not halt on inputs that they do not accept.

Now, we shall see “What are solvable/decidable problems and what are not solvable (unsolvable/undecidable) problems?”

8.2. A Language that is Not Recursively Enumerable

❖ **Definition:** The decision problems that have decision algorithms the output of which is *yes/no* are called solvable problems. According to Church-Turing thesis, an appropriate way to formulate precisely the idea of a decision algorithm is to use a Turing machine. The language L accepted by a Turing machine M is *recursively enumerable language* if and only if $L = L(M)$. Any instance of a problem for which the Turing machine halts whether the input is accepted or rejected is called ***solvable*** or ***decidable problem***. There are so many problems that are solvable. But, there are some problems that are *not solvable*. Now, we shall see what are called *unsolvable/undecidable problems*.

8.3. Undecidable Problems that are RE

❖ **Definition:** The problems that run forever on a Turing machine are not *solvable*. In other words, there are some problem input instances for which Turing machines will not halt on inputs that they do not accept. Those problems are called ***unsolvable*** or ***undecidable problems***. In general, if there is no general algorithm capable of solving every instance of the problem, then the decision problem is *unsolvable*. More precisely, if there is no Turing machine recognizing the language of all strings for various instances of the problems input for which the answer is *yes* or *no*, then the decision problem is *unsolvable*.

Let us assume M is a Turing machine with input alphabets $\{0, 1\}$, w is a string of 0's and 1's and M accepts w . If this problem with inputs restricted to binary alphabets $\{0, 1\}$ is undecidable, then the general problem is undecidable and can not be solved with a Turing machine with any alphabet.

Note: The term *unsolvable* and *undecidable* are used interchangeably.

Note: Even though we are able to answer the question in many specific instances, a problem may be undecidable. It means that there is no single algorithm guaranteed to provide an answer for every case.

Note: If a language L is not accepted by a Turing machine, then the language is not recursively enumerable. One important problem which is not recursively enumerable that is unsolvable/undecidable decision problem is “**Halting problem**”.

8.4. Halting Problem

The “**Halting Problem**” can informally be stated as “Given a Turing machine M and an input string w with the initial configuration q_0 , after some (or all) computations do the machine M halts?” In other words we have to identify whether (M, w) where M is the Turing machine, halts or does not halt when w is applied as the input. The domain of this problem is to be taken as the set of all Turing machines and all w i.e., Given the description of an arbitrary Turing machine M and the input string w , we are looking for a single Turing machine that will predict whether or not the computation of M applied to w will halt.

When we state decidability or undecidability results, we must always know what the domain is, because this may affect the conclusion. The problems may be decidable on some domain but not on another.

It is not possible to find the answer for Halting problem by simulating the action of M on w by a universal Turing machine, because there is no limit on the length of the computation. If M enters into an infinite loop, then no matter how long we wait, we can never be sure that M is in fact in a loop. The machine may be in a loop because of a very long computation. What is required is an algorithm that can determine the correct answer for any M and w by performing some analysis on the machine’s description and the input.

Formally, the Halting Problem is stated as “Given an arbitrary Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, A)$ and the input $w \in \Sigma^*$, does M halt on input w ?”.

8.5. Post's Correspondence Problem

The Post correspondence problem can be stated as follows. Given two sequences of n strings on some alphabet Σ say

$$A = w_1, w_2, \dots, w_n$$

and

$$B = v_1, v_2, \dots, v_n$$

we say that there exists a Post correspondence solution for pair (A,B) if there is a nonempty sequence of integers i, j, \dots, k , such that

$$w_i w_j \dots w_k = v_i v_j, \dots, v_k.$$

The Post correspondence problem is to devise an algorithm that will tell us, for any (A,B) whether or not there exists a PC-solution.

For example, Let $\Sigma = \{0, 1\}$. Let A is w_1, w_2, w_3 as shown below:

$$w_1 = 11, w_2 = 100, w_3 = 111$$

Let B is v_1, v_2, v_3 as shown below:

$$v_1 = 111, v_2 = 001, v_3 = 11$$

For this case, there exists a PC-solution as shown below:

w1	w2			w3		
1	1	1	0	0	1	1
v1			v2			v3
1	1	1	0	0	1	1

If we take

$$w_1 = 00, w_2 = 001, w_3 = 1000$$

$$v_1 = 0, v_2 = 11, v_3 = 011$$

there cannot be any PC-solution simply because any string composed of elements of A will be longer than the corresponding string from B.

8.6. Church Turing Hypothesis (Church's/Church-Turing Thesis)

Church's thesis: Various formal models of computations such as *Recursive functions* and *Post systems* were established by three prominent persons A. Church, S.C. Kleene and E. Post.

A function is called *primitive recursive* if and only if it can be constructed from the basic functions by successive composition and primitive recursion.

A *Post system* is similar to unrestricted grammar consisting of an alphabet and some production rules by which successive strings can be derived.

In addition to recursive functions and Post systems, many other formal computations models have been proposed. On examination it was found that though the computational models looked

quite different, they expressed the same thing. This observation was formalized in *Church's thesis* which is stated as follows:

Any “effective computation” or “any algorithmic” procedure that can be carried out by a human being or a team of human beings or a computer, can be carried out by some Turing machine. In other words, there is an effective procedure to solve a decision problem P if and only if there is a Turing machine that answers yes on inputs $\omega\psi \in \leftarrow P\psi$ and no for $\omega\psi \notin \leftarrow P$.

This theory maintains that all the models of computations those are proposed and yet to be proposed, are equivalent in their power to recognize languages or compute functions. This thesis predicts that it is unable to construct models of computation more powerful than the existing ones.

The above statement is known as “Church's thesis” named after the logician A.Church. Since the *Church's thesis* is closely related to Turing's thesis which states that we cannot go beyond Turing machines or their equivalent, it is also called Church-Turing thesis.

Church-Turing Thesis

Since there is no precise definition for “effective computation” or there is no precise definition for “algorithmic procedure”, Church's thesis is not a mathematically precise statement. So, this statement is not proved at the same time it has been not been disproved. Even though it is simply stated and not proved, now majority of scientists have accumulated enough evidence over the years that has caused Church's thesis to be generally accepted.

Bibliography

1. Introduction to Automata Theory, Languages and Computation, 2nd Edition. John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. Pearson Education, 2001.
2. Introduction to Languages and Theory of Computation. John C. Martin. Tata McGraw-Hill.
3. Introduction to Computer Theory, 2nd Edition. Daniel A. Cohen. John Wiley & Sons, 1991.
4. An Introduction to Formal Languages and Automata, 2nd Edition. Peter Lenz. Nervosa Publishing House, 1997.
5. Introduction to Theory of Computer Science, E.V. Krishnamurthy.
6. Principles of Compiler Design. Alfred V. Aho, Jeffrey D. Ullman. Narosa Publishing House.
7. Theory of Computer Science (Automata, Languages and Computation), K.L.P. Mishra, N. Chandrashekaran. PHI.
8. Switching and Finite Automata Theory. Kohavi. ZVI.
9. Discrete Mathematical Structures with Applications to Computer Science. Tremblay, Manohar. McGraw-Hill.

Index

A

- Abstract machine 11
- Alphabet 4
- Ambiguous grammar 253
- Application of GNF 315
- Applications of finite automata 79
- Applications of pumping lemma 170
- Applications of pumping lemma for CFLs 367
- Applications of the grammars 264

B

- Balanced parentheses 232

C

- ϵ -CLOSURE 114
- Cartesian product 3
- CFG to PDA 310
- CFLs are closed under union, concatenation and star 374
- CFLs are not closed under complementation 377
- CFLs are not closed under intersection 376
- Chomsky hierarchy 211

- Chomsky normal form 347
- Church-Turing thesis 447
- Church's/Church-Turing thesis 446
- Church's thesis 446
- Closure properties 178
- Closure property of regular languages 168
- Closure under complementation 179
- Closure under difference 183
- Closure under homomorphism 185
- Closure under intersection 180
- Closure under reversal 183
- Concatenation of two strings 5
- Construction of PDA 283
- Construction of Turing machine 386
- Context free grammar 213
- Counter machine 439
- Counter machines 437

D

- Decidable languages 386
- Decision problems 443
- Decision property of regular languages 168
- Derivation 223

Derivation tree 251
 Deterministic finite automaton 16
 Deterministic PDA 305
 DFA from an ϵ -NFA 116
 DFA processes the string 20
 Distinguishable states 189
 Divide a number by k 52

E

Eliminating useless symbols 330
 Empty set 2
 Empty stack 283
 Equivalence of single tape and multi-tape
 TM's 423
 Equivalence of two states 189
 Extended transition function 21, 115
 Extensions of Turing machines 422

F

Final state 14
 Finite automaton 11, 12
 Functions (subroutines) 428

G

GNF 352
 Grammar 210
 Grammar from finite automata 215
 Grammar from regular expressions 222
 Graphical representation of a PDA 280
 Greibach normal Form 347, 351

H

Halting problem 445
 Homomorphic image 186
 Homomorphism 185

I

ID 281
 Indistinguishable states 189
 Inherently ambiguous grammar 263
 Input alphabets 14
 Instantaneous description (ID) 281, 382

Intermediate state 14
 Intersection of two sets 3

K

Kleene's theorem 149
 Kleene plus 9

L

Language 9, 225
 Language accepted by DFA 22
 Leftmost derivation 249
 Left recursion 327, 328
 Length of a string 7, 9
 Linear Bounded Automaton 437, 440

M

Mark procedure 189
 Membership problem 443
 Minimization of DFA 190
 Minimization of Finite Automata 188
 Modulo k Counter Problems 57
 Monus 399
 Move of a machine 279
 Multi-stack machine 437, 438
 Multi-tape Turing machine 422, 423
 Multiple Tracks (Multi Track) 428

N

ϵ -NFA 114
 Non-acceptance of a language 23
 Non-deterministic PDA 305
 Non-deterministic Turing machine 422, 424
 Non-palindromes 228
 Nullable variable 336, 337, 338
 NULL production 336
 Null stack 283

O

Off-line Turing 439
 Off-line Turing machine 437
 One-step derivation 223

P

ϵ -productions 336
 Palindrome 227, 228, 406
 Parse tree 251
 Partial derivation tree 252
 Partial parse tree 252
 PDA to CFG 317
 Post correspondence problem 445
 Power of an alphabet 7
 Power set 3

Prefix 6
 Productions 210
 Programming techniques for
 Turing machines 427

Proper subtraction 399
 Properties of context free languages 363
 Pumping lemma 168
 Pumping Lemma for context free languages 364
 Pushdown automata 276, 278, 213

R

Recursive languages 386
 Recursively enumerable 386
 Recursively enumerable language 444
 RE from FA 149
 Regular expression 131
 Regular language 10, 11, 130, 167
 Restricted Turing machines 436
 Reversal of a language 183
 Reversal of a string 7, 183
 Rightmost derivation 250

S

Sentence 10, 224
 Sentential form 224, 249
 Set 1
 Simplification of CFG 329
 Standard Turing machine 382
 Start state 14
 String 5
 Subset 2
 Substitution methods 326

Sub string 6
 Suffix 6

T

ϵ -transition 111
 Table-filling algorithm 189
 Terminals 210
 Transducer 410
 Transition 14
 Transition diagram 17
 Transition diagram for Turing machine 390
 Transition function 15
 Transition table 18
 Turing machine model 379
 Turing machine with semi-infinite tape 437
 Turing machine with stay-option 427
 Type 0 grammar 211
 Type 0 language 212
 Type 1 Grammar 212
 Type 1 language 212
 Type 2 grammar 213
 Type 3 grammar 213
 Types of grammars 211
 Types of languages 10

U

Union of two sets 3
 Unit-production 340
 Unsolvable or undecidable problems 444
 Useless symbols 332

V

Variables 210

W

Word 10

Y

Yield 252
 Yield of the tree 251

Finite Automata and Formal Languages

A Simple Approach

A. M. Padma Reddy

This book has been written as an introductory subject for UG students of computer science and information science. The text provides a comprehensive study of the subject in easy-to-understand language with a self-learning style. It covers basic mathematics and notations, finite automata, regular languages, context-free grammar, CFG simplification, normal forms, PDA and turing machines. Numerous problems have been solved to understand the concepts of finite automata and formal languages.

A. M. Padma Reddy is Professor and Head of the Department of Computer Science and Information Science Engineering, Sai Vidya Institute of Technology, Bangalore. His industrial and teaching experience spans over 20 years with different roles such as teacher, course designer, project manager, and researcher. His areas of interests include compiler design, finite automata and formal languages, design and analogy of algorithms, data structures, networking, and UNIX. He has a number of books published in different areas of computer science and engineering to his credit and is a very popular author in the student community.

SANGUINETM

www.sanguineindia.com

ISBN 978-81-317-6047-5



9 788131 760475

www.pearsoned.co.in