Contents

Chapter 1

Basic Mathematics and Notations

1.1 Sets and set notations 1
1.2 Other definitions 2
1.3 Operations on Sets 3
1.4 Strings and Languages 5
1.5 Structural representation 9
1.6 Grammar 10
1.7 Derivation 11
1.8 Sentence 13
1.9 Language 14
1.10 Applications 40
1.11 Programming Languages 40
1.12 Finite Automata 41
1.13 Digital Design 41

Chapter 2

Finite Automata

2.1 Deterministic Finite Automaton (DFA) 49
2.2 Language accepted by a DFA 51
2.3 Properties of Transition Functions 53
2.4 Representation of DFA using Transition graph/diagram 55
2.5 Representation of DFA using Transition table 56
2.6 General procedure to design a FA 69
2.7 Regular language 80
2.8 Applications of Finite Automata 81
2.9 Non deterministic finite automata(NFA) 86
2.10 Properties of transition function 88
2.11 Acceptance of language 88
2.12 Need for Non-deterministic finite automaton 89
2.13 Conversion from NFA to DFA (Subset construction method) 90
2.14 Finite Automata with Epsilon transitions (ε-NFA) 94
2.15 Properties of transition function 96
2.16 Acceptance of language 96
2.17 Conversion from ε-NFA to DFA 97
2.18 Difference between DFA, NFA and ε-NFA 107

Chapter 3

Regular Expressions and Languages

3.1 Regular expression 112
3.2 Relation between FA and Regular Expression 120
3.3 To obtain ε-NFA from the regular expression 120
3.4 To obtain RE from FA (Kleene's theorem) 126
3.5 To obtain RE from FA (by eliminating states) 135
3.6 Applications of Regular Expressions 138
3.7 Chomsky Hierarchy 142
3.8 Type 0 grammar or unrestricted grammar 142
3.9 Type 1 grammar or context sensitive grammar 142
3.10 Type 2 grammar or context free grammar 143
3.11 Type 3 grammar or Regular Grammar 143
3.12 Finite Automaton from regular grammar 144
3.13 Regular grammar from FA 148
3.14 Left linear grammar from FA 152
3.15 Regular Expressions in UNIX 157

Chapter 4

Properties of Regular Languages

4.1 Closure under Union, concatenation and star 164
4.2 Closure under complementation 164
4.3 Closure under intersection 164
4.4 Closure under difference 165
4.5 Closure under homomorphism 165
4.6 Limitations of Regular Languages 167
4.7 Non Regular Languages 167
4.8 Applications of Pumping Lemma 170
4.9 Limitations of finite automaton 177
4.10 Equivalence of two states 178
4.11 Minimization of DFA 182
4.12 Alternate procedure to minimize the states of DFA 190

Chapter 5

Context Free Grammars

5.1 Advantages of regular and non-regular languages 200
5.2 Context free grammars 200
5.3 Leftmost derivation 208
5.4 Rightmost derivation 210
5.5 Derivation Tree (Parse tree) 210
5.6 Ambiguous grammar 212
5.7 Parsing 222
5.8 Application of context free grammars 224

Chapter 6

CFG Simplification & Normal Forms

6.1 Substitution 232
6.2 Left Recursion 233
6.3 Simplification of CFG 235
6.4 Eliminating ε - productions 240
6.5 Eliminating unit productions 244
6.6 Chomsky Normal Form 250
6.7 Greibach normal form (GNF) 254

Chapter 7

Pushdown Automata

7.1 Transitions 266
7.2 Graphical representation of a PDA 269
7.3 Instantaneous description 271
7.4 Acceptance of a language by PDA 272
7.5 Construction of PDA 273
7.6 Deterministic and Non-deterministic PDA 291
7.7 CFG to PDA 296
7.8 Application of GNF 300
7.9 PDA to CFG 302

Chapter 8

Properties of Context Free Languages

8.1 Pumping Lemma 310
8.2 Applications of Pumping Lemma for CFLs 313
8.3 CFLs are closed under Union, concatenation and star 319
8.4 CFLs are not closed under intersection 321
8.5 CFLs are not closed under complementation 322

Chapter 9

Turing Machines

9.1 Turing machine Model 325
9.2 Transition Table 326
9.3 Instantaneous description (ID) 328
9.4 Acceptance of a language by TM 330
9.5 Construction of Turing Machine (TM) 331
9.6 Transition diagram for Turing Machine (TM) 335
9.7 Transducers 354
9.8 Church Turing Hypothesis (Church's/Church-Turing thesis) 364

Chapter 10

Extensions of Turing Machines

10.1 Multi-tape Turing Machines 367
10.2 Equivalence of single tape and multi-tape TM's 368
10.3 Nondeterministic Turing Machines 369
10.4 Turing machine with stay-option 371

Chapter 11

Programming Techniques of Turing Machine

11.1 Multiple tracks/ Multi track) 373
11.2 Subroutines 373

Chapter 12

Restricted Turing Machines

12.1 Turing machine with semi-infinite tape 383
12.2 Multi-stack machines 384
12.3 Counter machines 385
12.4 Off-line Turing machine 385
12.5 Linear bounded Automata (LBA) 386
12.6 A language that is not recursively enumerable 387
12.7 Halting problem 388
12.8 Post's correspondence problem 388

1

Notations and Definitions

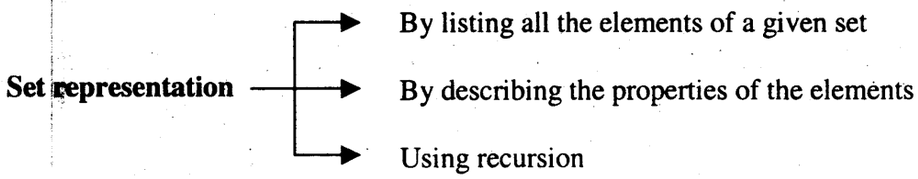
What are we studying in this chapter?

- Concept of sets and set notations
- Definitions and the notations of alphabets, strings, sub string, prefix suffix, language
- Various operations on strings such as concatenation, reversing a string, length of a string and various problems.
- The positive and star-closure over alphabets
- Definitions of
  - Grammar
  - Derivation
  - Sentence
  - Sentential form
- Various applications of automata and formal languages

1.1 Sets and set notations

In this section, let us answer the question **"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. The sets can be represented using various methods as shown below:



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,...etc and the elements within the set are denoted by lower case letters such as a, b, c etc.

By listing the elements of a given set 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 set shown in example 1.1 can also be represented by describing the properties of the elements as shown below:

By describing the properties of elements of a set The set of integers greater than 0, less than or equal to 25 and which are divisible by 5 can be represented as

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

Using recursion The set of integers greater than 0, less than or equal to 25 and which are divisible by 5 can be represented as

$S = \{a_0 \mid a_i = 5, a_{i+1} = a_i + 5\}$ where $a_i <= 20$. Here, the symbol '|' means "such that"

1.2 Other definitions

In this section, 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:

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

Now, let us see **"What is a subset?"**

Definition: A *set* A is a subset of B if every element of A is an element of 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

$A = \{a, b, c\}$ and $B = \{a, b, c\}$ 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 $\mathbf{2^A}$.

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?"**

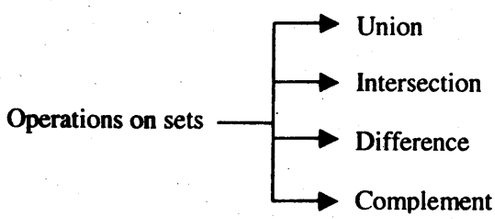
Definition: The Cartesian product of A and B is given by $A \times B = \{ (a, b) \mid a \in A$ and $b \in B\}$. Here, (a, b) is an ordered pair such that 'a' is an element of set A and 'b' is an element of set B.

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

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

1.3 Operations on Sets

The various operations that are carried on sets are shown below:



Now, let us see **"What is union of two sets and intersection of two sets?" Give example**

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

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

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

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

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 $\overline{A}$ and is defined as a set containing everything that is not in A. Formally, complement of A is defined as follows:

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

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

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

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

1.4 Strings and Languages

A language consists of alphabets from which the words, statements etc., can be obtained. The set of alphabets of a particular language is denoted by the symbol Σ.

Example: 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

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

Note: The symbol Σ is used to denote the alphabets of a language.

Note: The machine Language is made up of only 0's and 1's and so, the alphabets of machine language can be represented by

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

Now, let us see **"What is an alphabet? What is a string?" Give examples**

Definition: An *alphabet* is a finite, non empty set of symbols. The symbol Σ is used to denote an alphabet. For example, Σ = {0, 1} may denote the alphabets of machine language.

Definition: A *string* is a finite sequence of symbols from the alphabet Σ. The number of symbols in string is finite. An empty string is denoted by the symbol ε pronounced as epsilon (or λ) and note that $\varepsilon \notin \Sigma$.

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

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

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,....., along with the symbols such as +,-,(,),{,... are used to denote the symbols in Σ
- The lowercase letters such as u, v, w, x, ... Are 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 appending the symbols of *v* to the right of *u* i.e., if

$$u = a_1a_2a_3.....a_n$$

and

$$v = b_1b_2b_3.....b_m$$

then the concatenation of *u* and *v* is denoted by

$$uv = a_1a_2a_3.....a_nb_1b_2b_3.....b_m$$

and *u* is called the *prefix* and *v* is called the *suffix*.

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

$$u = Computer$$

and

$$v = Science$$

The concatenation of *u* and *v* denoted by *uv* will be

$$uv = ComputerScience$$

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 *xyz* = *w*, then *x*, *y* and *z* are all sub strings of string *w*. The sub string *x* is called the *prefix* of *w* i.e., A *prefix* is string of any number of leading symbols.

For example, the string *xyz* has prefix ε (empty string), *x*, *xy*, and *xyz*. The *suffix* is a string of any number of trailing symbols. For example, the string *xyz* has suffix ε

(empty string), *z*, *yz* and *xyz*. In the string "Rama", ε, "R", "Ra", "Ram" and "Rama" are the prefixes and the strings ε, "a", "ma", "ama" and "Rama" are all the suffixes.

Note: If $\Sigma = \{0, 1\}$, then

$\Sigma^0 = \{\varepsilon\}$ denote the set of words of length 0.

$\Sigma^1 = \{0, 1\}$ 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 is Kleene closure(star closure or Σ^*) ?"** Give example.

Definition: The Kleene closure is defined as follows:

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup$---- 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: Let $\Sigma = \{0, 1\}$. Then Σ^* is obtained as shown below:

$\Sigma^* = \{\varepsilon , 0 ,1 ,00 ,01 ,10 ,11 ,000 ,001,-----\}$ is the set of strings of 0's and 1's of any length which may include ε (epsilon) i.e., the null string.

Now, let us see **"What is Kleene positive (plus) closure?"**

Definition: The positive closure denoted by Σ^+ is defined as follows.

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup$ ..... which is the set of words of any length except the null string i.e., ε (epsilon).

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

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

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

Now, let us see **"What is a language?"**

Definition: A *language* can be defined as a set of strings obtained from Σ^* where Σ is set of alphabets of a particular language. So, if L is the language, then $L \subseteq \Sigma^*$. So, a language is a subset of Σ^*. For example,

1. A language of strings consisting of equal number of 0's and 1's can be represented as
   $\{\varepsilon, 01, 10, 0011, 1010, 0101, 0011, \ldots.\}$
2. 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:
   $\{\varepsilon, 01, 0011, 000111, \ldots\ldots\}$
3. An empty language is denoted by ε.

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

$$u = a_1a_2a_3\ldots.a_n$$

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

$$u^R = a_na_{n-1}a_{n-2}\ldots a_3a_2a_1$$

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

$$\varepsilon^R = \varepsilon$$
$$a^R = a$$

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

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

for each $a \in \Sigma$ and each $w \in \Sigma^*$

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

$$u = a_1a_2a_3\ldots.a_n$$

then the length u is given by

$$|u| = n$$

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

$$|\varepsilon| = 0$$

Note: $\varepsilon w = w\varepsilon = w$

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

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

for each $a \in \Sigma$ and each u, $w \in \Sigma^*$

1.5 Structural representation

A language can be normally represented using the following notations:

- Grammar
- Regular expressions

These two notations play a very important role in the study of *finite automata* and their applications. The finite automaton and their applications are discussed in detail in the next chapter.

Grammar: It is a mathematical model which is used in designing a software. The grammars are mainly used

- To identify the syntax of a language (for example, C/ C++ etc)
- To identify the syntax of a statement (for example, for statement, while-statement etc)
- To identify whether the expression is syntactically correct or not and so on.

In general, grammars are mainly used to identify whether a program written is syntactically correct or not. That is, they are used in the **designing of a compiler** to check whether the program written is syntactically correct or not.

Note: The style and the notations used in grammar are entirely different than the corresponding finite automata. A programming statement such as for-statement, if-statement etc., can be easily modeled using a grammar.

Regular expressions: It is also a mathematical model which is used to represent the text strings. The text strings can be easily represented using regular expressions. The patterns of strings described by regular expressions can be described using finite automata. But, the style and the notations used are entirely different when compared with finite automaton and the corresponding grammar. The regular expressions are dealt in detail in chapter 3 and 4.

1.6 Grammar

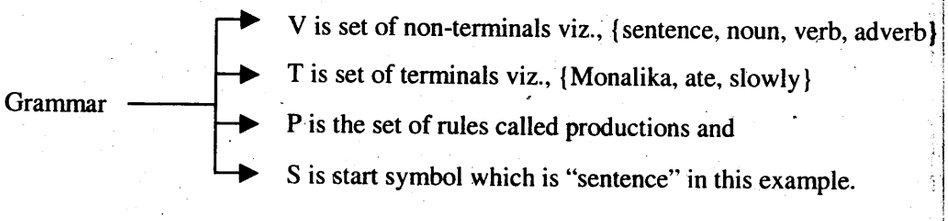
In this section, let us see the definition of a grammar and how a language can be constructed using the grammar. Consider the sentence "Monalika ate slowly". In this sentence the word "Monalika" is noun and is followed by the word "ate" which is a verb and is followed by the word "slowly" which is an 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

sentence → <noun> <verb> <adverb>
noun → Monalika
verb → ate
adverb → slowly

The above four rules which are used to obtain the sentence "Monalika ate slowly" are called productions. The words *sentence*, *noun*, *verb*, and *adverb* are called *non-terminals* or *variables*. Each production starts with *non-terminal* followed by an *arrow* followed by combinations of *non-terminals* and/or *terminals*. The left hand side of '→' in the first production is called the *start symbol* (in this case it is the word 'sentence'). The words "Monalika", "ate" and "slowly" can be considered as *terminals*.

Note: The *non-terminals* can be replaced by string of *terminals* and/or *non-terminals* whereas the *terminals* cannot be replaced or substituted.

The set of rules to form a sentence which in turn used to generate a language is called a grammar. Here, a grammar G is 4-tuple or quadruple which consists of the following:



Now, let us see **"What is a grammar? Give an example.** The formal definition of a grammar is shown below:

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
- S is the start symbol

Each production is of the form $\alpha \rightarrow \beta$ where α is a non empty string of terminals and/or non-terminals and β is string of terminals and/or non-terminals including the null string i.e., α is a string in $(V \cup T)^+$ and β is a string in $(V \cup T)^*$. This grammar is also called as *phase-structure* grammar.

Note that ε(epsilon) i.e., the null string cannot be there on the left hand side of any production and so positive closure $(V \cup T)^+$ is defined for α, where as * closure $(V \cup T)^*$ is defined for β. Note that in the examples shown below α is a single non-terminal. When α is a single non-terminal, the grammar is called *context free grammar* (CFG).

Example 1.1: Consider the productions shown below:

$S \rightarrow aCa$
$C \rightarrow aCa \mid b$

Here, grammar G = (V, T, P, S) where

V = { S, C }
T = { a, b}
P = {
S → aCa
C → aCa | b
}
S is the start symbol

Note: In productions, all the lowercase letters, digits or any other special characters such as (,), +, -, $, etc., are considered to be *terminals* and all uppercase letters are considered as *non-terminals* (or *variables*). If the symbol ε is used, it is neither a terminal nor a non-terminal. But, it is a special symbol indicating the NULL string or empty string.

1.7 Derivation

Consider the production of the form

$A \rightarrow aA \mid a$

and a string w of the form w = xAx. Here, the non-terminal A can be replaced by its right hand side i.e., either by the string **aA** or **a**. If A is replaced by **aA**, we get the string w of the form

$$w = xaAx$$

This can be written as

$$xAx \Rightarrow xaAx$$

and read as the string xAx *derives* xaAx in one step (i.e., by applying one production). A production can be applied any number of times in an arbitrary order. A production can be applied whenever it is applicable and successive strings can be derived. In general, consider the derivation of the form

$$w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \ldots\ldots w_n$$

The string w_n is derived from w_1 and can be written as

$$w_1 \overset{*}{\Rightarrow} w_n$$

Note:

1. The symbol '*' in '$\overset{*}{\Rightarrow}$' indicates that unspecified number of productions (including applying no production) are applied to get the string w_n from w_1.
2. The string w_n derived from w_1 by applying at least one production can be represented as

$$w_1 \Rightarrow^{+} w_n$$

The symbol + indicates that at least one or more productions are applied to get the string w_n from w_1. Now, let us see **"What is derivation?"** The formal definition of **derivation** is shown below.

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 derives 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

$$A \Rightarrow \alpha\beta\gamma$$

Thus, the 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

$$A \overset{*}{\Rightarrow} \alpha\beta\gamma$$

Note: If α is any string of terminals and variables then $\alpha \overset{*}{\Rightarrow} \alpha$ i.e., α derives itself.

Example 1.2: Consider the grammar shown below from which any arithmetic expression can be obtained.

$$\begin{aligned} E &\rightarrow E+E \\ E &\rightarrow E-E \\ E &\rightarrow E*E \\ E &\rightarrow E/E \\ E &\rightarrow \textbf{id} \end{aligned}$$

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 id + id * id and show the derivation for the same.

Solution: The derivation to get the string id + id * id is shown below.

$$\begin{aligned} E &\Rightarrow E+E \\ &\Rightarrow \textbf{id}+E \\ &\Rightarrow \textbf{id}+E*E \\ &\Rightarrow \textbf{id}+\textbf{id}*E \\ &\Rightarrow \textbf{id}+\textbf{id}*\textbf{id} \end{aligned}$$

Since the string **id + id * id** is obtained from the start symbol E by applying more than one production, this can be written as

$$E \Rightarrow^{+} \textbf{id} + \textbf{id} * \textbf{id}$$

1.8 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: In the derivation shown in example 1.2, **id + id * id** is the sentence of the grammar. If there is a derivation $S \overset{*}{\Rightarrow} \alpha$, where α contains string of terminals and/or non-terminals, then α is called *sentential form* of G. In the derivation shown in example 1.2,

E+E, **id** + E, **id** + E * E, **id** + **id** * E, **id** + **id** * **id**

are all *sentential forms* of the grammar.

1.9 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 see **"What is the formal definition of a language accepted 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

$$L(G) = \{ w \mid S \overset{*}{\Rightarrow} w \text{ and } w \in T^* \}$$

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 *sentences*.

Example 1.3: Let G = (V, T, P, S) where

V = { S, C }
T = { a, b}
P = {
S → aCa
C → aCa | b
}
S is the start symbol

What is the language generated by this grammar?

Solution: Consider the derivation

$S \Rightarrow aCa \Rightarrow aba$ (By applying the 1st and 3rd production)

So, the string aba ∈ L(G)

Consider the derivation

$S \Rightarrow aCa$	By applying $S \rightarrow aCa$
$\Rightarrow aaCaa$	By applying $C \rightarrow aCa$
$\Rightarrow aaaCaaa$	By applying $C \rightarrow aCa$
.	
.	
$\Rightarrow a^nCa^n$	By applying $C \rightarrow aCa$ n-1 times
$\Rightarrow a^nba^n$	By applying $C \rightarrow b$

So, the language L accepted by the grammar G is

$$L(G) = \{ a^nba^n \mid n \geq 1 \}$$

i.e., the language L derived from the grammar G is "The string consisting of *n* number of *a*'s followed by a '*b*' followed by *n* number of *a*'s.

Example 1.4: 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

$$S \rightarrow + \mid - \mid \varepsilon$$

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

$$D \rightarrow 0 \mid 1 \mid 2 \mid \ldots\ldots \mid 9$$

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

$$N \rightarrow D$$
$$N \rightarrow ND \mid DN$$

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

$$I \rightarrow N \mid SN$$

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, ...9 }
P = {

I → N | SN		[Generate signed or unsigned number]
N → D | ND | DN		[Generates one or more digits]
S → + | - | ε		[Generate the sign]
D → 0 | 1 | 2 | ..... | 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	⇒ N	I	⇒ SN
	⇒ ND		⇒ +N
	⇒ N5		⇒ +ND
	⇒ ND5		⇒ +N5
	⇒ N65		⇒ +ND5
	⇒ ND65		⇒ +N65
	⇒ N965		⇒ +ND6 5
	⇒ D965		⇒ +N965
	⇒ 1965		⇒ +D965
			⇒ +1965

Example 1.5: Let Σ = {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 productions is S → ε
2. *a* and *b* are palindromes. The equivalent productions are S → a | b
3. If *w* is a palindrome then the string *awa* and the string *bwb* are palindromes. The equivalent productions are

   S → aSa | bSb

So, the grammar to generate strings of palindromes over Σ is given by G = (V, T, P, S) where

V = {S}
T = { a, b}
P = {

S	→ ε		[Definition 1]
S	→ a | b		[Definition 2]
S	→ aSa | bSb		[Definition 3]

}
S is the start symbol

Example 1.6: 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

$$B \rightarrow aB \mid bB \mid \varepsilon$$

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	→	aSa | bSb	[Generates palindromes both on left side and right side]
S	→	A	[Generates a non-palindrome]
A	→	aBb | bBa	[Generates a non palindrome with B generating any number of a's and b's]
B	→	aB | bB | ε	[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.

$S \Rightarrow aSa$
$\Rightarrow abSba$
$\Rightarrow abAba$
$\Rightarrow abaBba$
$\Rightarrow ababba$

Example 1.7: Obtain the grammar to generate the language

$$L = \{ 0^m 1^m 2^n \mid m \geq 1 \text{ and } n \geq 0 \}$$

Solution: In the language $L = \{0^m1^m2^n\}$, if n = 0, the language L contains m number of 0's and m number of 1's. The grammar for this can be of the form

$$A \rightarrow 01 \mid 0A1$$

If n is greater than zero, the language L should contain m number of 0's followed by m number of 1's followed by one or more 2's i.e., the language generated from the non-terminal A should be followed by n number of 2's. So, the resulting productions can be written as

$$S \rightarrow A \mid S2$$
$$A \rightarrow 01 \mid 0A1$$

Thus, the grammar G to generate the language

$$L = \{0^m1^m2^n \mid m \geq 1 \text{ and } n \geq 0\}$$

can be written as G = (V, T, P, S) where

$V = \{S, A\}$
$T = \{0, 1, 2\}$
$P = \{$
$\quad S \rightarrow A \mid S2$
$\quad A \rightarrow 01 \mid 0A1$
$\}$
S is the start symbol

Note: Let us concentrate on another technique. Observe that the language

$$L = \{0^m1^m2^n \mid m \geq 1 \text{ and } n \geq 0\}$$

can be split up into two languages L_1 and L_2 where L_1 is made up of *m* number of 0's followed by *m* number of 1's for $m \geq 1$. The language L_2 is made up of *n* number of 2's for $n \geq 0$. The given language can be obtained by concatenating L_1 and L_2. The language L_1 consisting of *m* number of 0's followed by *m* number of 1's for $m \geq 1$ can be obtained using the productions:

$$A \rightarrow 01 \mid 0A1$$

The language L_2 consisting of zero or more 2's can be obtained using the productions:

$$B \rightarrow 2B \mid \varepsilon$$

Now, the given language L can be obtained by concatenating the languages generated from the above productions. This is achieved using the production:

$$S \rightarrow AB.$$

Thus, the grammar G to generate the language

$$L = \{0^m 1^m 2^n \mid m \geq 1 \text{ and } n \geq 0\}$$

is given by

$$G = (V, T, P, S) \text{ where}$$

$V = \{S, A, B\}$
$T = \{0, 1, 2\}$
$P = \{$
$S \rightarrow AB$
$A \rightarrow 01 \mid 0A1$
$B \rightarrow 2B \mid \varepsilon$
$\}$
S is the start symbol

Example 1.8: Obtain a grammar to generate the language

$$L = \{0^n 1^{n+1} \mid n \geq 0\}$$

Note: It is clear from the language that total number of 1's will be one more than the total number of 0's and all 0's precede all 1's. So, first let us generate the string 0^n1^n and add the digit 1 at the end of this string.

The recursive definition to generate the string 0^n1^n can be written as

$$A \rightarrow 0A1 \mid \varepsilon$$

If the production $A \rightarrow 0A1$ is applied n times we get the sentential form as shown below.

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow \quad \ldots\ldots 0^nA1^n$$

Finally if we apply the production

$$A \rightarrow \varepsilon$$

the derivation starting from the start symbol A will be of the form

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow \ldots\ldots 0^nA1^n \Rightarrow 0^n1^n$$

Thus, using these productions we get the string 0^n1^n. An extra 1 can be obtained by appending 1 at the end of A. This can be achieved by using the production

$$S \rightarrow A1$$

Note that from A, we get string 0^n1^n and 1 is appended at the end, resulting in the string 0^n1^{n+1}. So, the final grammar G to generate the language $L = \{0^n1^{n+1} \mid n \geq 0\}$ will be G = (V, T, P, S) where

$$
\begin{aligned}
V &= \{S, A\} \\
T &= \{0, 1\} \\
P &= \{ \\
&\qquad S \rightarrow A1 \\
&\qquad A \rightarrow 0A1 \mid \varepsilon \\
&\} \\
S &\quad \text{is the start symbol}
\end{aligned}
$$

Note: The language $L = \{0^n1^{n+1} \mid n \geq 0\}$ can also be obtained using the grammar with only two productions as shown below:

$$S \rightarrow 0S1 \mid 1$$

Example 1.9: 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:

1. An empty string denoted by ε has equal number of a's and b's (i.e., zero a's and zero b's).
2. The symbol 'a' can be followed by the symbol 'b'
3. The symbol 'b' can be followed by the symbol 'a'

The corresponding productions for these three cases can be written as

$$S \rightarrow \varepsilon$$
$$S \rightarrow aSb$$
$$S \rightarrow bSa$$

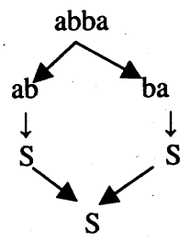
Using these productions the strings of the form ε, ab, ba, abab, baba etc., can be generated. But, the strings such as abba, baab, etc., where the string starts and ends with the same symbol, can not be generated from these productions (even though they are valid strings). So, to obtain the productions 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:

$$
\begin{aligned}
S &\Rightarrow aSb \qquad \text{(By applying } S \rightarrow aSb\text{)} \\
&\Rightarrow ab \qquad\qquad \text{(By applying } S \rightarrow \varepsilon\text{)}
\end{aligned}
$$

Similarly, the substring 'ba' can be generated from S and the derivation is shown below:

$S \Rightarrow bSa$ (By applying $S \rightarrow bSa$)
$\Rightarrow ba$ (By applying $S \rightarrow \varepsilon$)

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

V = {S}
T = {a, b}
P = {
S → ε
S → aSb
S → bSa
S → SS
}
S is the start symbol

Example 1.10: What is the language generated by the grammar

$S \rightarrow 0A \mid \varepsilon$
$A \rightarrow 1S$

Solution: The null string ε can be obtained by applying the production S→ ε and the derivation is shown below:

$S \Rightarrow \varepsilon$ (By applying $S \rightarrow \varepsilon$)

Consider the derivation

$S \Rightarrow 0A$ (By applying $S \rightarrow 0A$)
$\Rightarrow 01S$ (By applying $A \rightarrow 1S$)
$\Rightarrow 010A$ (By applying $S \rightarrow 0A$)
$\Rightarrow 0101S$ (By applying $A \rightarrow 1S$)
$\Rightarrow 0101$ (By applying $S \rightarrow \varepsilon$)

So, alternatively applying the productions $S \rightarrow 0A$ and $A \rightarrow 1S$ and finally applying the production $S \rightarrow \varepsilon$, 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

$$L = \{w \mid w \in \{01\}^*\} \text{ or } L = \{(01)^n \mid n \geq 0\}$$

Note: The above language can also be generated from the following two grammars:

$S \rightarrow 01S \mid \varepsilon$

and

$S \rightarrow S01 \mid \varepsilon$

Example 1.11: 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, S) where

$V = \{S\}$
$T = \{ (,),[,],\{,\} \}$
$P = \{$
$S \rightarrow (S)$
$S \rightarrow [S]$
$S \rightarrow \{S\}$
$S \rightarrow SS$
$S \rightarrow \varepsilon$
$\}$
S is the start symbol

Example 1.12:

Example 1.13: Obtain a grammar to generate the language $L = \{ww^R \mid w \in \{a,b\}^*\}$ where w^R is reverse of w.

Solution: The grammar to generate the language $L = \{ww^R \mid w \in \{a,b\}^*\}$ where w^R is reverse of w is given by

G = (V, T, P, S) where

V = { S }
T = {a, b}
P = {
S → aSa | bSb | ε
}
S is the start symbol.

Note: In the above grammar if the production

$$S \rightarrow \varepsilon$$

is replaced by

$$S \rightarrow c$$

the resulting grammar will generate the language of the form $L = \{wcw^R \mid w \in \{a,b\}^*\}$

Example 1.14: Obtain a grammar to generate the language
$L = \{0^n 1^{2n} \mid n \geq 0\}$

Solution: The grammar G to generate the language $L = \{0^n 1^{2n} \mid n \geq 0\}$ is G = (V, T, P, S) where

V = {S}
T = {0, 1}
P = {
S → 0S11 | ε
}
S is the start symbol

Example 1.15: Obtain a grammar to generate the language
$L = \{0^{n+2} 1^n \mid n \geq 1\}$

Solution: It is similar to the previous problem, with little bit of modifications. The grammar G to generate the language $L = \{0^{n+2} 1^n \mid n \geq 1\}$ is G = (V, T, P, S) where

V = {S, A}
T = {0, 1}
P = {
S → 00A
A → 0A1 | 01
}

Example 1.16: Obtain a grammar to generate the language
$L = \{0^i 1^j \mid i \neq j, i \geq 0 \text{ and } j \geq 0\}$

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) where

V = {S, A, B, C}
T = {0, 1}
P = {

S	→	BA|	[At least one 0 is preceded by 0^n1^n]
S	→	AC	[At least one 1 is followed by 0^n1^n]
A	→	0A1 | ε	[Generates $0^n1^n \mid n \geq 0$]
B	→	0B | 0	[At least one 0 is generated]
C	→	1C | 1	[At least one 1 is generated]

}
S is the start symbol

Note: The following grammar also generates the language $L = \{0^i1^j \mid i \neq j, i \geq 0 \text{ and } j \geq 0\}$

V = {S, A, B, C}
T = {0, 1}
P = {

S	→	0S1	[Generates 0^n1^n recursively]
S	→	A	[To generate more 0's than 1's]
S	→	B	[To generate more 1's than 0's]
A	→	0A | 0	[At least one 0 is generated]
B	→	1B | 1	[At least one 1 is generated]

}
S is the start symbol

Example 1.17: Obtain a grammar to generate the language
$L = \{a^{n+2}b^m \mid n \geq 0 \text{ and } m > n\}$

Solution:

Note: It is clear from the above statement that the set of strings that can be generated by this language can be represented as

L = {**aab**b*, **aaabb**b*, **aaaabbbb**b*, ...}

Observe that the language consists of a string $a^nb^n \mid n \geq 1$ preceded by one *a* and followed by zero or more *b*'s. This prompts us to have a production of the form

S → aAB

where the language $a^nb^n \mid n \geq 1$ is generated from the variable A and zero or more b's are generated from the variable B. The language $a^nb^n \mid n \geq 1$ 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

$$B \rightarrow bB \mid \varepsilon$$

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 → 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 → aAb | ab		[generates a^nb^n]
B → bB| ε		[Generates zero or more b's]

}
S - is the start symbol

Note: The above language can also be generated from the grammar shown below:

V = { S, A, B}
T = {a,b}
P = {

S → AB		[Generates $a^{n+1}b^n$ followed by by zero or more b's]
A → aAb | a		[generates $a^{n+1}b^n$]
B → bB| ε		[Generates zero or more b's]

}
S - is the start symbol

Example 1.18: Obtain a grammar to generate the language
$L = \{a^nb^m \mid n \geq 0, m > n\}$

Note: It is similar to the previous problem with some changes. It is clear from the above statement that the set of strings that can be generated by this language can be represented as

$$L = \{\varepsilon bb^*, \mathbf{ab}bb^*, \mathbf{aabb}bb^*, \ldots\ldots\ldots\}$$

where b* represents zero or more b's. Observe that the language consists of a string $a^nb^n \mid n \geq 0$ followed by one or more *b*'s. This prompts us to have a production of the form

$$S \rightarrow AB$$

where the language $a^nb^n \mid n \geq 0$ is generated from the variable A and one or more b's are generated from the variable B. The language $a^nb^n \mid n \geq 0$ is generated from the variable A using the productions shown below

$$A \rightarrow aAb \mid \varepsilon$$

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 → AB
A → aAb | ε [Generates $a^nb^n \mid n \geq 0$]
B → bB| b [Generates one or more b's]
}
S - is the start symbol

Note: The same language can also be generated from the following grammar:

S → aSb | B [Generates $a^nb^n \mid n \geq 0$ followed by at least one b]
B → bB| b [Generates one or more b's]

Example 1.19: Obtain a grammar to generate the language

$$L = \{a^nb^{n-3} \mid n \geq 3\}$$

Note: It is clear from the above statement that the set of strings that can be generated by this language can be represented as

$$L = \{aaa, aaaab, aaaaabb, aaaaaabbb, \ldots\ldots\}$$

It is observed from the above set that the string 'aaa' is followed by the string $a^nb^n \mid n \geq 0$. So, the first production that we can think of is

$$S \rightarrow aaaA$$

where the string $a^nb^n \mid n \geq 0$ can be obtained using A as shown below:

$$A \rightarrow aAb \mid \varepsilon$$

So, the final grammar that can generate the given language is

$V = \{ S, A \}$
$T = \{a,b\}$
$P = \{$
$S \rightarrow aaaA$
$A \rightarrow aAb \mid \varepsilon$
$\}$
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 1.20: Obtain a grammar to generate the language

$L = L_1L_2$

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 1.42. For convenience it is provided again

$V = \{ S_1, B\}$
$T = \{a,b\}$
$P = \{$
$S_1 \rightarrow aS_1b \mid bB$
$B \rightarrow bB \mid \varepsilon$
$\}$
S_1 - is the start symbol

The grammar corresponding to the language L_2 is already obtained in example 1.38. For convenience it is provided again

$V = \{S_2\}$
$T = \{0, 1\}$
$P = \{$
$S_2 \rightarrow 0S_211 \mid \varepsilon$
$\}$
S_2 is the start symbol

The resulting language

$$L = L_1L_2$$

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

$$S \rightarrow S_1S_2$$

where S is the start symbol of the resultant grammar. So, the final grammar which accepts

$$L = L_1L_2$$

is shown below:

$V = \{ S, S_1, S_2, B\}$
$T = \{a,b,0,1\}$
$P = \{$
$S \rightarrow S_1S_2$
$S_1 \rightarrow aS_1b \mid bB$
$S_2 \rightarrow 0S_211 \mid \varepsilon$
$B \rightarrow bB \mid \varepsilon$
$\}$
S_1 - is the start symbol

Example 1.21: Obtain a grammar to generate the language

$$L = L_1 \cup L_2$$

where

$$L_1 = \{a^nb^m \mid n \geq 0, m > n\}$$
$$L_2 = \{0^n1^{2n} \mid n \geq 0\}$$

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.

$$S \rightarrow S_1 \mid S_2$$

The rest of the productions remain same. The final grammar is shown below(Both the grammars below generate the same language)

$V = \{ S, S_1, S_2, A, B\}$
$T = \{a,b,0,1\}$
$P = \{$
$S \rightarrow S_1 \mid S_2$
$S_1 \rightarrow AB$
$S_2 \rightarrow 0S_211 \mid \varepsilon$
$A \rightarrow aAb \mid \varepsilon$
$B \rightarrow bB \mid b$
$\}$
S_1 - is the start symbol

$V = \{ S, S_1, S_2, B\}$
$T = \{a,b,0,1\}$
$P = \{$
$S \rightarrow S_1|S_2$
$S_1 \rightarrow aS_1b \mid bB$
$S_2 \rightarrow 0S_211 \mid \varepsilon$
$B \rightarrow bB \mid \varepsilon$
$\}$
S_1 - is the start symbol

Example 1.22: Obtain a grammar to generate the language $L = \{w : |w| \bmod 3 = 0\}$ on $\Sigma = \{a\}$

Solution: The language accepted by the grammar can also be written as

$L = \{\varepsilon, aaa, aaaaaa, aaaaaaaaa, aaaaaaaaaaaa, \ldots\ldots\ldots\}$

It is clear from this definition that any string generated should have the length multiples of 3 which can be easily done by the production

$$S \rightarrow aaaS \mid \varepsilon$$

So, the final grammar is

V = {S}
T = {a}
P = {
S → aaaS | ε Accepts w of |w| = 3i, i ≥ 0
}
S - is the start symbol

Example 1.23: Obtain a grammar to generate the language

$L = \{w : |w| \bmod 3 = 0\}$ on $\Sigma = \{a, b\}$

Solution:

Note: It is similar to the previous problem, except that the string is made up of a's and b's.

So, a string of length with multiples of 3 can be generated using the production

$$S \rightarrow AAAS \mid \varepsilon$$

where the non-terminal (or variable) A can be replaced by either *a* or *b* as shown below using the production

$$A \rightarrow a \mid b$$

So, the final grammar is

V = {S,A}
T = {a, b}
P = {
S → AAAS | ε Accepts w of |w| = 3i, i ≥ 0
A → a | b
}
S - is the start symbol

Example 1.24: Obtain a grammar to generate the language
$L = \{w : |w| \bmod 3 > 0\}$ on $\Sigma = \{a\}$

Solution: The language generated from this can also be represented as

L = {a, aa, aaaa, aaaaa, aaaaaa, aaaaaaa, ...}

Note that in the above set of strings, a string of length having multiples of 3 is not there.

Step1: To generate the above language, let us generate a single *a* followed by non-terminal A as shown

S → aA

Step 2: From A, we can generate ε thus producing single *a* using the production

A → ε

in turn producing a string with |w| = 1

Step 3: To generate a string w with |w| = 2, from A, we have to produce one more *a* followed by a non-terminal B as shown below:

A → aB

Step 4: From B, we can generate ε thus producing two *a*'s using the production

B → ε

in turn producing a string with |w| = 2

Step 5: To generate a string w with |w| = 3, from B, we have to produce one more *a* followed by a non-terminal C as shown below:

$$A \rightarrow aC$$

Since we should not accept a string of |w| = 3, we should not derive ε from C. So, if we stop at step 2 or step 4 we get a string of |w| = 1 or 2. If we are in step 5, we have three a's followed by C from which we should be in a position get strings of |w| = 4 or 5. Now, to get a string of |w| = 4 or 5, we have to get one more *a* or two more *a*'s which we have already obtained using step 2 and step 4.

It means that after accepting three *a*'s in step 5, to get one more *a* or two *a*'s, we can replace C by S and repeat the steps again. So, instead of using the production

$$A \rightarrow aC$$

we could have used the production

$$A \rightarrow aS$$

to get a string of |w| mod 3 > 0. So, the final grammar is

V = {S, A, B}
T = {a}
P = {
S → aA
A → aB | ε (Accepts w of |w| = 1,4,7,...)
B → aS | ε (Accepts w of |w| = 2,5,8,...)
}
S - is the start symbol

Note: The same language can also be obtained using the following grammar:

V = {S}
T = {a}
P = {
S → a | aa | aaaS
}
S - is the start symbol

Example 1.25: Obtain a grammar to generate the language
L = {w : |w| mod 3 ≠ |w| mod 2}on Σ = {a}

Solution: The above language can also be represented as

$$L = \{aa, aaa, aaaa, aaaaa, aaaaaaa, aaaaaaaa, aaaaaaaaa, \ldots\ldots\}$$

or

$$L = \{(w : w \in a^* \text{ and } |w| = 2i \text{ or } 3i \text{ or } 4i \text{ or } 5i \text{ for } i = 1, 4, 7, 10, 13, \ldots\}$$

This problem is similar to the previous problem. The reader is supposed to think and understand how these productions are evolved step by step. The complete grammar is shown below.

$V = \{S, A, B, C, D\}$
$T = \{a\}$
$P = \{$
$S \rightarrow aaA$
$A \rightarrow aB \mid \varepsilon$ (Accepts w of $|w| = 2,8,\ldots.$)
$B \rightarrow aC \mid \varepsilon$ (Accepts w of $|w| = 3,9,\ldots.$)
$C \rightarrow aD \mid \varepsilon$ (Accepts w of $|w| = 4,10,\ldots$)
$D \rightarrow aS \mid \varepsilon$ (Accepts w of $|w| = 5,11,\ldots$)
$\}$
S - is the start symbol

Note: The following grammar also accepts the same language

$V = \{S\}$
$T = \{a\}$
$P = \{$
$S \rightarrow aa \mid aaa \mid aaaa \mid aaaaa \mid aaaaaS$
$\}$
S - is the start symbol

Example 1.26: Obtain a grammar to generate the language

$$L = \{w : |w| \bmod 3 \geq |w| \bmod 2\} \text{ on } \Sigma = \{a\}$$

Solution: The above language can also be represented as

$$L = \{\varepsilon, a, aa, aaaa, aaaaa, aaaaaa, aaaaaaa, aaaaaaaa, aaaaaaaaa, \ldots\}$$

or

$$L = \{(w : w \in a^* \text{ and } |w| = i \text{ for } i=0,1,2,4,5,6,7,8,10,11,12,13,14,16,\ldots.\}$$

This problem is similar to the previous problem. The reader is supposed to think and understand how these productions are evolved step by step. The complete grammar is shown below.

V = {S, A, B, C, D}
T = {a}
P = {
S → aA | ε (Accept w of |w|=0,6,12,.....)
A → aB | ε (Accept w of |w|=1,7,13,.....)
B → aaC | ε (Accept w of |w|=2,8,14,.....)
C → aD | ε (Accept w of |w|=4,10,16,....)
D → aS | ε (Accept w of |w|=5,11,17,.....)
}
S - is the start symbol

Note: The above grammar can also be written as

V = {S}
T = {a}
P = {
S → ε | a | aa | aaaa | aaaaa | aaaaaS
}
S - is the start symbol

Example 1.27: Obtain a grammar to generate set of all strings with exactly one *a* when Σ = {a, b}

Since w ∈ 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 → 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 → bB | ε

So, the final grammar to generate at least one *a* is

V = {S, B}
T = {a, b}
P = {
S → bS | aB
B → bB | ε
}
S - is the start symbol

Example 1.28: 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

$$A \rightarrow aA \mid bA \mid \varepsilon$$

So, the final grammar to generate at least one *a* is

$$\begin{aligned} V &= \{S, A\} \\ T &= \{a, b\} \\ P &= \{ \\ &\quad S \rightarrow bS \mid aA \\ &\quad A \rightarrow aA \mid bA \mid \varepsilon \\ &\} \\ S &- \text{is the start symbol} \end{aligned}$$

Example 1.29: Obtain a grammar to generate the set of all strings with no more than three *a*'s when $\Sigma = \{a, b\}$

Soluion: 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

$$S \rightarrow bS \mid \varepsilon$$

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

$$A \rightarrow bA \mid \varepsilon$$

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:

$$B \rightarrow bB \mid \varepsilon$$

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

$$C \rightarrow bC \mid \varepsilon$$

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

$V = \{S, A, B, C\}$
$T = \{a,b\}$
$P = \{$
$S \rightarrow bS \mid aA \mid \varepsilon$
$A \rightarrow bA \mid aB \mid \varepsilon$
$B \rightarrow bB \mid aC \mid \varepsilon$
$C \rightarrow bC \mid \varepsilon$
$\}$
S - is the start symbol

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 $B \rightarrow bB \mid \varepsilon$. So, the final grammar to generate the given language is given by G = (V, T, P, S) where

$V = \{S, B\}$
$T = \{a, b\}$
$P = \{$
$S \rightarrow B \mid BaB \mid BaBaB \mid BaBaBaB$
$B \rightarrow bB \mid \varepsilon$
$\}$
S - is the start symbol

Example 1.30: Obtain a grammar to generate the language $L = \{w \mid n_a(w) = n_b(w) + 1\}$

Solution: The problem is similar to example 1.9, 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

$A \rightarrow aAb$
$A \rightarrow bAa$
$A \rightarrow AA$
$A \rightarrow \varepsilon$

So, the final grammar to generate the language

$$L = \{w \mid n_a(w) = n_b(w) + 1\}$$

is G = (V, T, P, S) where

V = {S, A}
T = {a, b}
P = {
S → AaA
A → aAb | bAa | AA | ε
}
S is the start symbol

Example 1.31: Obtain the grammar to generate the language

$$L = \{w \mid n_a(w) > n_b(w)\}$$

Solution: The problem is similar to example 1.9, except that w ∈ L should have more number of *a*'s than *b*'s. We have already seen that the following grammar produces equal number of *a*'s and *b*'s:

A → aAb
A → bAa
A → AA
A → ε

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 → aB | 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 → 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

V = {S, A, B}
T = {a, b}
P = {
S → AB|BA|ABA
A → aAb
A → bAa
A → AA
A → ε
B → aB | a
}
S is the start symbol

Example 1.32: Obtain the grammar to generate the language $L = \{w \mid n_a(w) = 2n_b(w)\}$

The explanation is similar to the problem 1.9, except that for every two *a*'s we are supposed to produce one b. So, the final grammar to generate the language

$$L = \{w \mid n_a(w) = 2n_b(w)\}$$

is G = (V, T, P, S) where

V = {S}
T = {a, b}
P = {
S → ε
S → aaSb
S → bSaa
S → SS
}
S is the start symbol

Example 1.33: 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

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

So, the final grammar to generate the given language is G = (V, T, P, S) where

$V = \{S\}$
$T = \{0,1\}$
$P = \{$
$\quad S \rightarrow A000A$
$\quad A \rightarrow 0A \mid 1A \mid \varepsilon$
$\}$
S is the start symbol

Example 1.34: Obtain a grammar to generate the following language
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:

$$A \rightarrow bAcc \mid \varepsilon$$

The substring thus generated from A-production is enclosed between a^n and c^n and the corresponding productions can be written as

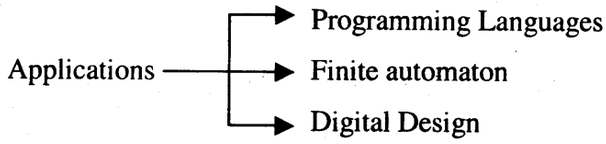
$$S \rightarrow aSc \mid A$$

So, the final grammar to generate the given language is G = (V, T, P, S) where

V = {S}
T = {a,b,c}
P = {
S → aSc | A
A → bAcc | ε
}
S is the start symbol

1.10 Applications

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



1.11 Programming Languages

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.

Example 1.35: 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

<identifier> → <letter><letter_digit>
<letter_digit> → <letter><letter_digit>|<digit><letter_digit>|∈
<letter> → a|b|....z|A|B|....|Z
<digit> → 0|1|2|....|9

where <identifier>, <letter_digit>, <letter>, <digit> are the variables or non-terminals and the symbols such as a, b ..z, A, B,....Z, 0, 1, 2,....9 are terminals.

1.12 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 Chapter 2. The figure 1.3 shows a finite automaton to accept only a valid identifier.

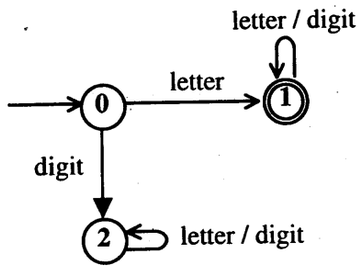


Fig. 1.3 FA to accept an identifier

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.

1.13 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. The implementation of a binary adder which is an integral part of any general purpose computer is discussed in this section. Let the two binary numbers to be added are:

$x_0x_1x_2.....x_n$ and $y_1y_2y_3....y_n$

where each x_i or y_i is a binary digit 0 or 1. When these two digits are added with or without the carry, the result of addition may be 0 or 1 with or without carry. The block diagram of binary adder is shown in figure 1.4.a. The possible inputs and the possible outputs and whether the carry is generated or not is shown in figure 1.4.b.

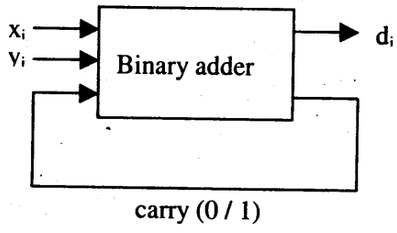
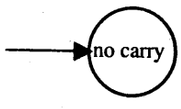


Fig. 1.4 (a) Binary adder

input		output	
x_i	y_i	d_i	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Fig. 1.4.b Truth table

Note that the output d_i of the binary adder can be 0 or 1 with or without carry. To start with there is no carry while adding x_i with y_i and so the state "no carry" is the start state which is designated with a straight arrow as shown below.

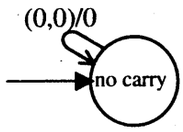


After adding x_i with y_i the result obtained is d_i and the state can either be "carry" or "no carry". This can be indicated generally as an edge with the label:

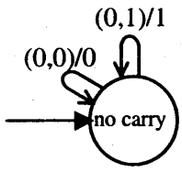
$(x_i, y_i) / d_i$

Let us take situations with and without carry and construct the binary adder.

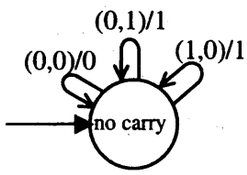
With no carry: If x_i is 0 and y_i is 0 then d_i is 0 and carry will not be generated. This can be represented as (0,0)/0 and the machine stays in "no carry" state as shown below:



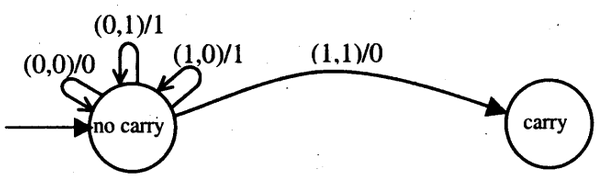
If x_i is 0 and y_i is 1 then d_i is 1and carry will not be generated. This can be represented as (0,1)/1 and the machine stays in "no carry" state and the automaton can be modified as shown below:



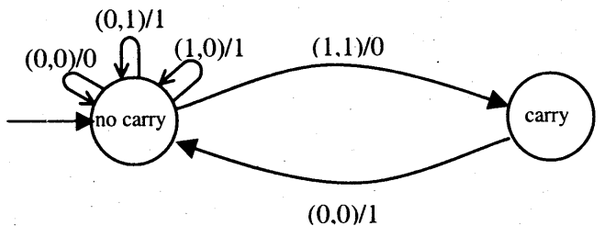
If x_i is 1 and y_i is 0 then d_i is 1and carry will not be generated. This can be represented as (1,0)/1 and the machine stays in "no carry" state and the automaton can be modified as shown below:



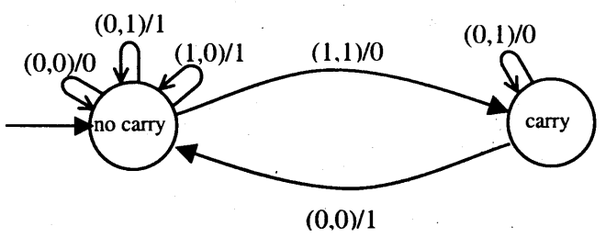
If x_i is 1 and y_i is 1 then d_i is 0 and carry is generated. This can be represented as (1,1)/0 and the machine goes to "carry" state as shown below:



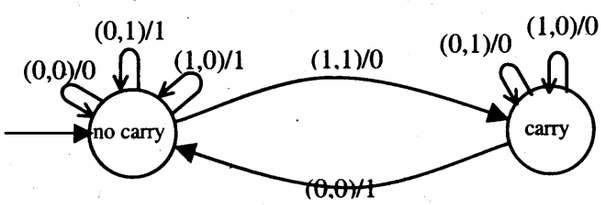
With carry: If x_i is 0 and y_i is 0 then d_i is 1 and the carry is not generated. This can be represented as (0,0)/1 and the machine goes to "no carry" state as shown below:



If x_i is 0 and y_i is 1 then d_i is 0 and a carry is generated. This can be represented as (0,1)/0 and the machine stays in "carry" state as shown below:



If x_i is 1 and y_i is 0 then d_i is 0 and a carry is generated. This can be represented as (1,0)/0 and the machine stays in "carry" state as shown below:



If x_i is 1 and y_i is 1 then d_i is 1 and a carry is generated. This can be represented as (1,1)/1 and the machine stays in "carry" state as shown below:

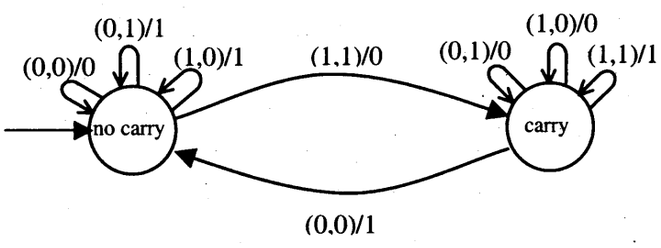


Fig. 1.5 Automaton for serial binary adder

The figure 1.5 shows the complete serial binary adder using the automaton. Note that the transition consists of input as well as the output. Such machines where the outputs are associated with transitions are called Mealy Machines and if the output is

associated with the state, the machines are called Moore Machines. The study of these machines is not the scope of this book and the reader is supposed to refer on these topics.

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
   $L = \{ 0^m1^m2^n \mid m \geq 1 \text{ and } n \geq 0 \}$
9. Obtain a grammar to generate the language $L = \{0^n1^{n+1} \mid n \geq 0\}$
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
    $S \rightarrow 0A \mid \varepsilon$
    $A \rightarrow 1S$
12. Obtain a CFG to generate a string of balanced parentheses
13. Obtain a grammar to generate the language
    $L = \{ww^R \mid w \in \{a,b\}^*\}$ where w^R is reverse of w
14. Obtain a grammar to generate the language $L = \{0^n1^{2n} \mid n \geq 0\}$
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 $L = \{0^i1^j \mid i \neq j, i \geq 0 \text{ and } j \geq 0\}$
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
    $L = \{a^nb^m \mid n \geq 0, m > n\}$
19. Obtain a grammar to generate the language $L = \{a^nb^{n-3} \mid n \geq 3\}$
20. Obtain a grammar to generate the language $L = L_1L_2$
    where
    $L_1 = \{a^nb^m \mid n \geq 0, m > n\}$
    $L_2 = \{0^n1^{2n} \mid n \geq 0\}$
21. Obtain a grammar to generate the language $L = L_1 \cup L_2$
    where
    $L_1 = \{a^nb^m \mid n \geq 0, m > n\}$
    $L_2 = \{0^n1^{2n} \mid n \geq 0\}$
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 grammar to identify an identifier
32. How an automaton can be used to add two binary numbers?

2

Finite Automata

What we will know after reading this chapter?

- Deterministic finite automaton (DFA)
- Languages accepted by DFA
- Transition diagram
- Representation of DFA using Transition diagram
- Solution to various types of problems while constructing DFA.
- Regular language
- Applications of Finite Automata
- Non-Deterministic Finite Automaton (NFA)
- Languages accepted by NFA
- Need for an NFA
- Differences between DFA and NFA
- Conversion from NFA to DFA
- Solutions for varieties of problems while converting from NFA to DFA
- Solution to more than 25 problems of various nature
- Finally, we will be having confidence to solve any given problem

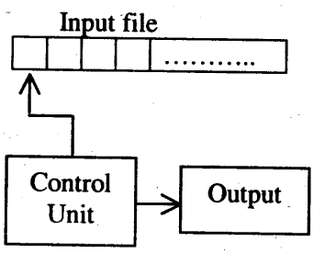
The *finite automata* theory is the study of *abstract machines* or *computing devices*. The automata theory even though was originally proposed to model brain function, it was extremely useful for a variety of other purposes. Automata are a useful model in both hardware and software. Now, let us see **"What are applications of finite automata?"** Some of the applications where automata plays an important are:

- In the design of digital circuits and checking the behavior of the digital circuits using software.
- In compiler design: During compilation, the *lexical analyzer* (which is a component of a compiler) breaks the input text into various logical units such as identifiers, keywords and punctuation.
- In designing a software for identifying the words, phrases and other patterns in large bodies of text (such as collection of web pages)
- In building the software to verify the systems having finite number of states (for example, communication protocols)

Note: In our life, we still remember some past events and some which are not important we forgot. Those events might have played very important role and based on those events our course of actions changes and hence the outcome of that action changes. On similar lines, in *finite automaton* some points might change the output of the system and some points will not affect the output. These points which affect the output of the system are called states. Each finite automaton has some states using which it can remember useful information.

Now, let us see **"What is finite automaton?"**

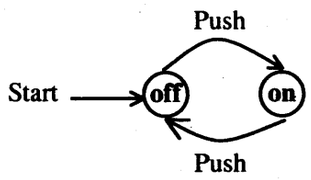
Definition: A finite automaton is a mathematical model for study of *abstract machines* or *abstract computing devices* with discrete inputs and discrete outputs. In short, it is an abstract model of a digital computer which with following components:



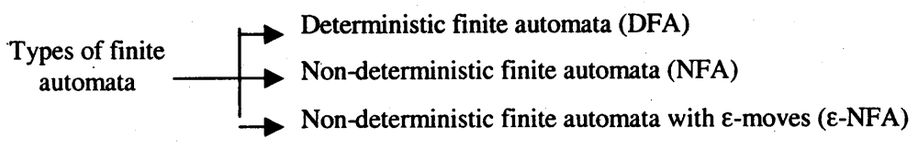
Input file: Input file is divided into cells. Each cell can hold one symbol.

Control Unit: Initially control unit will be in start state. Based on the input symbols, the state of the control unit may change.
Output: Output may be *accept* or *reject*. When end of input is encountered, the control unit may be in *accept* or *reject* state. If it is in accept state, we say that the input string is accepted by the machine. Otherwise, we say that the input string is rejected by the machine.

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:



Now, let us see **"What are the different types of finite automata?"** The different types of finite automata are shown below:



2.1 Deterministic Finite Automaton (DFA)

Let us take the pictorial representation of DFA shown in figure 2.1 and understand the various components of DFA.

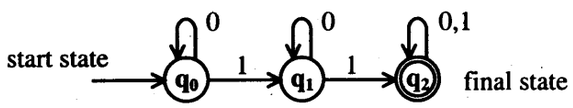


Fig 2.1 Sample DFA

It is clear from this diagram that, the DFA is represented using circles, arrows and arcs labeled with some digits, concentric circles etc. The circles are nothing but the states of DFA. In the DFA shown in figure 2.1, there are three states viz., q_0, q_1 and q_2. An arrow enters into state q_0 and is not originating from any state and so it is quite different from other states and is called the *start state* or *initial state*. The state q_2 has two concentric circles and is also a special state called the *final state* or *accepting state*.

In this DFA, there is only one final state. Based on the language accepted by DFA, there can be more than one final state also. The states other than start state and final states are called *intermediate states*. Always the machine initially will be in the *start state*. It is clear from the figure 2.1 that, the machine in state q_0, after accepting the symbol 0, stays in state q_0 and after accepting the symbol 1, the machine enters into state q_1. Whenever the machine enters from one state to another state, we say that there is a transition from one state to another state. Here we can say that there is a transition from state q_0 to q_1 on input symbol 1.

In state q_1, on input symbol is 0, the machine will stay in q_1 and on symbol 1, there is a transition to state q_2. In state q_2, on input symbol 0 or 1, the machine stays in state q_2 only. This DFA has three states q_0, q_1 and q_2 and can be represented as

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

The possible input symbols to this machine are 0 and 1 and can be represented as

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

which is set of input symbols (alphabets) for the machine. There will be a transition from one state to another based on the input alphabets. If there is a transition from V_i to V_j on an input symbol a, it can be represented as

$$\delta(V_i, a) = V_j$$

The transitions from each state of the machine shown in figure 2.1 based on the input alphabets {0, 1} are shown in table 2.1.

Current State	Input	Next state	Transition representation
q_0	0	q_0	$\delta(q_0,0) = q_0$
q_0	1	q_1	$\delta(q_0,1) = q_1$
q_1	0	q_1	$\delta(q_1,0) = q_1$
q_1	1	q_2	$\delta(q_1,1) = q_2$
q_2	0	q_2	$\delta(q_2,0) = q_2$
q_2	1	q_2	$\delta(q_2,1) = q_2$

Table 2.1 showing the transitions for the machine shown in fig. 2.1

Consider only the transitions (last column in table 2.1) defined for the automaton:

$\delta(q_0,0) = q_0$
$\delta(q_0,1) = q_1$
$\delta(q_1,0) = q_1$
$\delta(q_1,1) = q_2$
$\delta(q_2,0) = q_2$
$\delta(q_2,1) = q_2$

δ: $(Q \times \Sigma)$ to Q

where $Q = \{q_0, q_1, q_2\}$
$\Sigma = \{0,1\}$

Note that in each $\delta(q, a)$ defined above, $q \in Q$ and $a \in \Sigma$ and the ordered pair $(q, a) \in Q \times \Sigma$. The state q_0 is the initial state and the state with two concentric circles (i.e., q_2) is the final state. With this concept, now let us see **"What is a deterministic finite automaton (finite state machine)?"**. A deterministic finite automaton in short DFA can be defined as follows.

Definition: A DFA is 5-tuple or quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where

- Q - is non-empty, finite set of states.
- Σ - is non-empty, finite set of input alphabets.
- δ - is transition function, which is a mapping from $Q \times \Sigma$ to Q. Based on the current state and input symbol, the machine may enter into another state.
- $q_0 \in Q$ - is the start state.
- $F \subseteq Q$ - is set of accepting or final states.

Note: For each input symbol *a*, from a given state there is exactly one transition (there can be no transitions from a state also) and we are sure (or can determine) to which state the machine enters. So, the machine is called **Deterministic** machine. Since it has finite number of states the machine is called *Deterministic finite machine(automaton).*

2.2 Language accepted by a DFA

Consider the transition diagram or DFA shown in figure 2.2. The start state is q_0 and the final state is q_2. To start with the machine will be in start state q_0.

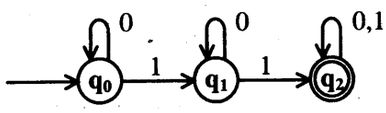


Fig 2.2 Sample DFA

Let us assume that the string 1011 is the input. On first input symbol 1, the machine enters into state q_1. In state q_1, on input symbol 0, the machine stays in state q_1 only. In state q_1, on input symbol 1, the machine enters into state q_2. In state q_2, on the input symbol 1, the machine stays in state q_2. Now we encounter end of the input and note that we are in the accepting state q_2. The moves made by the DFA for the string 1011 is shown in figure 2.3.

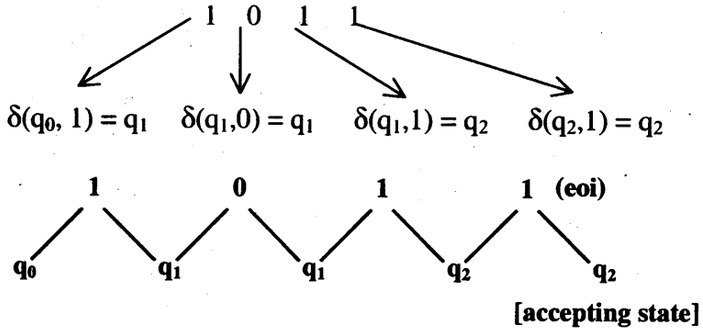


Fig.2.3 Acceptance of the string 1011

So, after scanning the input string 1011, the machine finally stays in state q_2 which is an accepting state. Hence we say that the string 1011 is accepted by the machine.

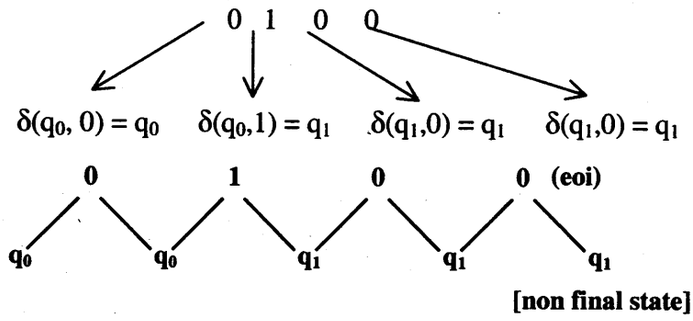


Fig.2.4 Rejection of the string 0100

Let us take the string 0100. The moves made by the machine for the string 0100 is evident from figure 2.4. Note that after scanning the string 0100 the machine stays in state q_1 which is non-final state. So, the string 0100 is rejected by the machine. Now, let

us see "When we say that a language is accepted by DFA?" the acceptance of the string or rejection of the string by a machine can formally be defined as follows:

Definition: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA where

Q - is set of finite states
Σ - is set of input alphabets (from which a string can be formed)
δ: $Q \times \Sigma \rightarrow Q$ - is transition function
q_0 - is the start state
F - is the final or accepting state.

The string (also called language) *w* accepted by an FA or DFA can be defined in formal notation as:

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

Non-acceptance means that after the string is processed, the DFA will not be in the final state and so the string is rejected. The non-acceptance of the string *w* by an FA or DFA can be defined in formal notation as:

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

In this definition, instead of using δ we used the extended transition function $\hat{\delta}$. The transition function δ is used whenever the second argument to δ is a symbol and we use $\hat{\delta}$ whenever the second argument to δ is a string. For example, in the figure 2.2,

$\delta(q_0, 1) = q_1$ and $\delta(q_1, 1) = q_2$. So, $\hat{\delta}(q_0, 11) = q_2$.

2.3 Properties of Transition Functions

The different properties of the transition function are:

Properties of transition functions

$$\delta(q, \varepsilon) = \hat{\delta}(q, \varepsilon) = q \qquad 2.1$$

$$\delta(q, wa) = \delta(\hat{\delta}(q, w), a) \qquad 2.2$$

$$\delta(q, aw) = \hat{\delta}(\delta(q, a), w) \qquad 2.3$$

where $q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$. Note that in the property 2.1, $\delta(q, \varepsilon) = q$. It means that when the current state of the machine is q and when there is no input (ε means no input or the empty string), the machine will not move to any new state, instead, it stays in the same state q.

Example 2.1: For the DFA shown in figure 2.2 what is $\hat{\delta}(q_0, 101)$?
We know from property (2.2) that

$$\hat{\delta}(q_0,101) = \delta(\hat{\delta}(q_0,10), 1) \qquad (2.4)$$

where w = 10 and a = 1. Now, let us find out what is $\hat{\delta}(q_0,10)$? Again by applying property (2.2) $\hat{\delta}(q_0,10)$ can be written as

$$\hat{\delta}(q_0,10) = \delta(\hat{\delta}(q_0, 1), 0) \qquad (2.5)$$

where w = 1 and a = 0. The transition function $\hat{\delta}(q_0, 1)$ can be written using the property (2.2) as

$$\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \varepsilon), 1) \qquad (2.6)$$

Using the property (2.1) $\hat{\delta}(q_0, \varepsilon) = q_0$.

Substituting this in (2.6) we get

$$\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \varepsilon), 1)$$
$$= \delta(q_0, 1) = q_1$$

Substituting $\hat{\delta}(q_0, 1) = q_1$ in (2.5) we get

$$\hat{\delta}(q_0,10) = \delta(\hat{\delta}(q_0, 1), 0)$$
$$= \delta(q_1, 0) = q_1$$

Substituting $\hat{\delta}(q_0,10) = q_1$ in (2.4) we get

$$\hat{\delta}(q_0,101) = \delta(\hat{\delta}(q_0,10), 1)$$
$$= \delta(q_1, 1) = q_2$$

which is the final state. So, $\hat{\delta}(q_0,101) = q_2$.

Example 2.2: DFA which accepts any number of a',s followed by a string ba and followed by string of a's and b's.

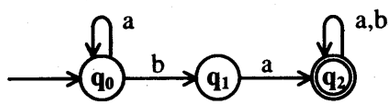


Fig 2.5 Finite automaton

Here, the machine M = (Q, Σ, δ, q_0, F) where

Q = {q_0, q_1, q_2}
Σ = {a, b}
q_0 is the start state
F = {q_2}
δ is shown using the table 2.2

	Σ	
States	a	b
→q_0	q_0	q_1
q_1	q_2	-
(q_2)	q_2	q_2

Table 2.2 Transition table for fig. 2.5

In the transition table 2.2, the arrow mark to the state q_0 indicates that q_0 is the start state and circle enclosing state q_2 indicates that q_2 is the final state. Since this machine has finite number of states, it is called finite automaton. After the input symbol we can determine to which state the machine enters and so this machine is called deterministic finite automata. The automaton such as this may have zero or one transition on an input symbol. Before developing the DFAs for specified problems, let us see how the DFAs are represented. The DFAs are represented usually using two different methods:

1. Pictorial representation (Transition graph / transition diagram)
2. Tabular form (Transition table)

2.4 Representation of DFA using Transition graph/diagram

The DFA can be expressed pictorially using circles, arrows and arcs with labels, double circles etc. as shown in figure 2.6.

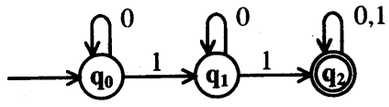


Fig 2.6 Sample DFA (Transition graph)

This pictorial representation of FA is called the *transition graph* or *transition diagram*. Now, let us see **"What is a transition graph?"** The formal definition of a *transition graph* or *transition diagram* is shown below:

Definition: The transition diagram or transition graph for the DFA M = (Q, Σ, δ, q_0, F) is defined as follows:

- Each state of Q corresponds to one node or vertex
- From each state $q \in Q$ and for each symbol $a \in \Sigma$, let the transition be $\delta(q,a) = p$. In such case, there is a directed edge from state q to *state* p with the edge labeled a.
- q_0 is the start state. The start state is a state which has an arrow entering into that state. Note that the arrow is not originated from some other state.
- F final state is represented by double circles where as a non-final state is represented by a single circle.

Note: The transition from one state to another state is indicated by the directed edge. If there is a directed edge from V_i to V_j and the edge is labeled a, then we say that there is a transition (denoted by the symbol δ) from state V_i to V_j on the input symbol a i.e., in the current state V_i, upon accepting an input symbol a, the machine enters into a new state Vj. This is denoted by $\delta(V_i,a) = V_j$.

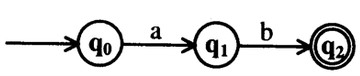
2.5 Representation of DFA using Transition table

In this method, the DFA is represented in the tabular form. This table is called transitional table. There is one row for each state, and one column for each input. Since, in the transition diagram shown in figure 2.6, there are three states, there are three rows for each state. The input symbols are only 0 and 1 and so, there are two columns for the input symbols. The transitional table for the diagram shown in figure 2.6 is shown in table 2.3.

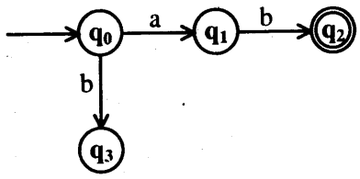
	Σ	
States	0	1
$\rightarrow q_0$	q_0	q_1
q_1	q_1	q_2
(q_2)	q_2	q_2

Table 2.3 Transition table for fig. 2.6

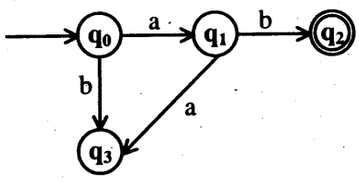
Example 2.3: Obtain a DFA to accept strings of a's and b's starting with the string ab
From the problem it is clear that the string should start with *ab* and so, the minimum string that can be accepted by the machine is *ab*. To accept the string *ab*, we need three states and the machine can be written as



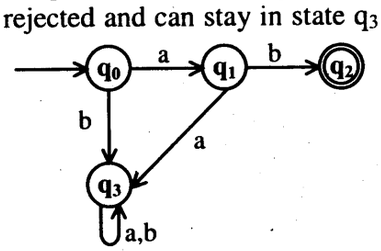
where q_2 is the final or accepting state. In state q_0, if the input symbol is b, the machine should reject b (note the string should start with a). So, in state q_0, on input b, we enter into the rejecting state q_3. The machine for this can be of the form



The machine will be in state q_1, if the first input symbol is *a*. If this *a* is followed by another *a*, the string *aa* should be rejected by the machine. So, in state q_1, if the input symbol is *a*, we reject it and enter into q_3 which is the rejecting state. The machine for this can be of the form



Whenever the string is not starting with *ab*, the machine will be in state q_3 which is the rejecting state. So, in state q_3, if the input string consists of of a's and b's of any length, the entire string can be rejected and can stay in state q_3 only. The resulting machine can be of the form



The machine will be in state q_2, if the input string starts with *ab*. After the string *ab*, the string containing any combination of a's and b's, can be accepted and so remain in state q_2 only. The complete machine to accept the strings of a's and b's starting with the string *ab* is shown in figure 2.7. The state q_3 is called dead state or trap state or rejecting state.

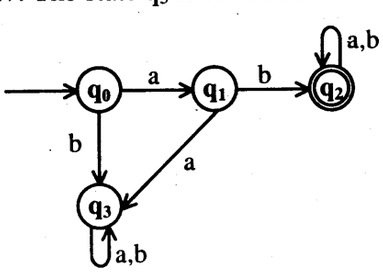


Fig.2.7 Transition diagram to accept string ab(a+b)*

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

$$L = \{ab(a+b)^n \mid n \geq 0\}$$

or

$$L = \{ab(a+b)^*\}$$

So, the DFA which accepts strings of a's and b's starting with the string *ab* 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 is the start state

$F = \{q_2\}$.

δ is shown the transition table 2.4.

	δ	$\leftarrow\Sigma\rightarrow$ a	b
↓ States →	$\rightarrow q_0$	q_1	q_3
	q_1	q_3	q_2
	(q_2)	q_2	q_2
	q_3	q_3	q_3

Table 2.4 Transition table for DFA shown in fig.2.7

To accept the string *abab*: The string is accepted by the machine and is evident from the figure 2.8.

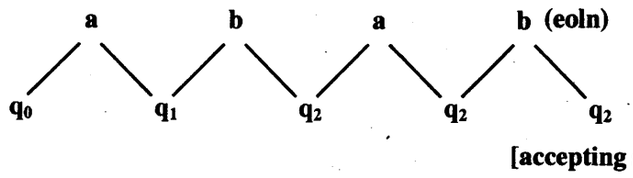


Fig.2.8 To accept the string abab

Here, $\hat{\delta}$ $(q_0, abab) = q_2$ which is the final state. So, the string **abab** is accepted by the machine.

To reject the string *aabb*: The string is rejected by the machine and is evident from the figure 2.9.

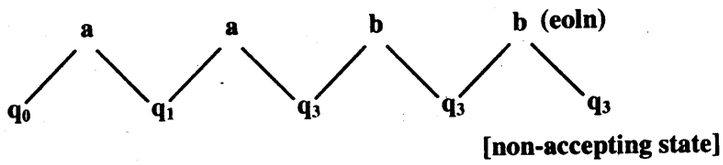
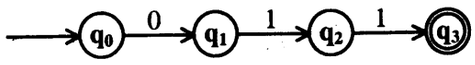


Fig.2.9 To reject the string aabb

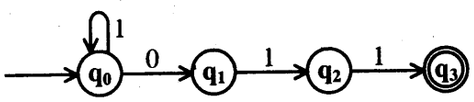
Here, $\hat{\delta}$ $(q_0,aabb) = q_3$ which is not an accepting state. So, the string *aabb* is rejected by the machine.

Example 2.4: Draw a DFA to accept string of 0's and 1's ending with the string 011.

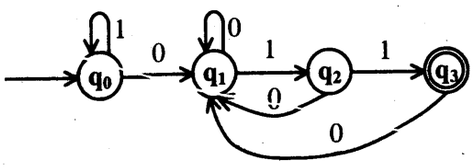
The minimum string that can be accepted by the machine is 011. It requires four states with q_0 as the start state and q_3 as the final state as shown below.



In state q_0, suppose we input the string 1111...011. Since the string ends with 011, the entire string has to be accepted by the machine. To accept the string 011 finally, the machine should be in state q_0. So, on any number of 1's the machine stays only in state q_0 and if the string ends with 011, the machine enters into the final state. The machine can be of the form



If the machine is in any of the states q_1, q_2 and q_3 and if the current input symbol is 0 and if the next input string is 11, the entire string should be accepted. This is because the string ends with 011. So, from all these states on the input symbol 0, there should be a transition to state q_1 so that if we enter the string 11 we can reach the final state. Now the machine can take the form as shown below.



In state q_3, if the input symbol is 1, enter into state q_0 so that if the next input string is 011, we can enter into the final state q_3. So, the final machine which accepts a string of 0's and 1's ending with the string 011 is shown in figure 2.10.

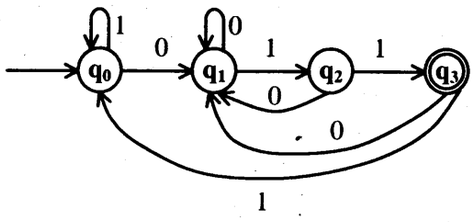


Fig.2.10 Transition diagram to accept (0+1)*011

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

$$L = \{ (0+1)^n 011 \mid n \geq 0 \}$$

or

$$L = \{ (0+1)^* 011 \}$$

So, the DFA which accepts strings of 0's and 1's ending with the string 011 is given by

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

$Q = \{ q_0, q_1, q_2, q_3 \}$

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

q_0 is the start state

$F = \{q_3\}$.

δ is shown using the transition table 2.5.

↓ States	δ	←Σ→ 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

Table 2.5 Transition table for the machine shown in fig. 2.10

To accept the string *0011*: This string is accepted by the machine and the sequence of moves made by the machine is shown in figure 2.11. Here, δ' $(q_0,0011) = q_3$ which is the final state. So, the string **0011** is accepted by the machine.

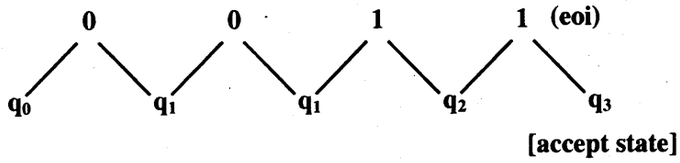


Fig.2.11 To accept the string 0011

To reject the string *0101*: The string is rejected by the machine and the sequence of moves made by the machine are shown in figure 2.12.

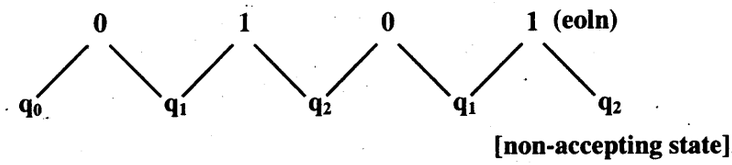
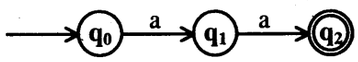


Fig.2.12 To reject the string 0101

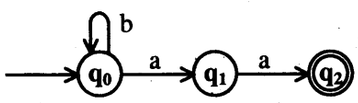
Here, $\hat{\delta}(q_0, 0101) = q_2$ which is not an accepting state. So, the string **0101** is rejected by the machine.

Example 2.5: Obtain a DFA to accept strings of a's and b's having a sub string aa

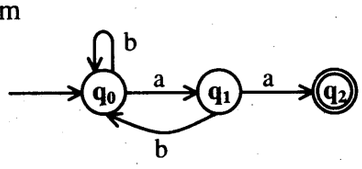
The minimum string that can be accepted by the machine is *aa*. To accept exactly two symbols, the DFA requires 3 states and the machine to accept the string *aa* can take the form



where q_0 is the start state and q_2 is the accepting state. In state q_0, if the input symbol is *b*, stay in q_0 so that when any number of b's ends with *aa*, the entire string is accepted. The machine for this can be of the form



There is a transition to state q_1 on input symbol *a*. In state q_1, if the input symbol is *b*, there will be a transition to state q_0 so that if this *b* is followed by *aa*, the machine enters into state q_2 so that the entire string is accepted by the machine. The transition diagram for this can be of the form



The machine enters into state q_2 when the string has a sub string *aa*. So, in this state even if we input any number of a's and b's the entire string has to be accepted. So, the machine should stay in q_2. The final machine which accepts strings of a's and b's having a sub string *aa* is shown in figure 2.13.

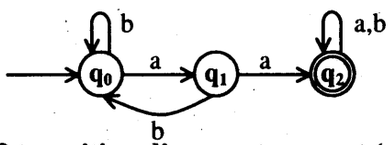


Fig.2.13 transition diagram to accept (a+b)*aa(a+b)*

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

$$L = \{ (a+b)^m aa(a+b)^n \mid m,n \geq 0 \}$$

or

$$L = \{(a+b)^* aa(a+b)^* \}$$

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

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

$\Sigma = \{a, b\}$

q_0 is the start state

$F = \{q2\}$

δ is shown using the transition table 2.6.

States ↓	δ	a	b
	→q_0	q_1	q_0
	q_1	q_2	q_0
	(q_2)	q_2	q_2

Table 2.6 The transition table

To accept the string *baab*: This string is accepted by the machine and the sequence of moves made by the machine is shown in figure 2.14.

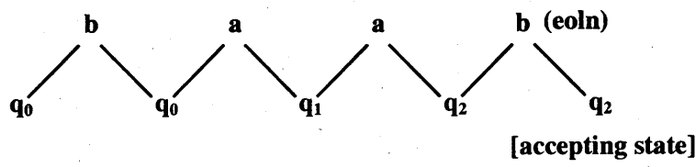


Fig.2.14 To accept the string baab

Here, $\hat{\delta}$ (q_0,baab) = q_2 which is the final state. So, the string *baab* is accepted by the machine. The string *baba* is rejected by the machine and the sequence of moves made by the machine is shown in figure 2.15.

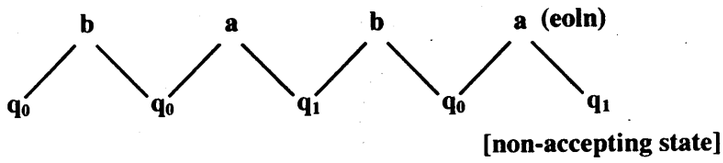


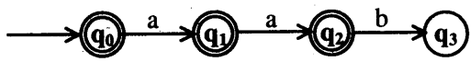
Fig.2.15 To reject the string baba

Here, $\hat{\delta}$ (q_0, baba) = q_1 which is not an accepting state. So, the string *baba* is rejected by the machine.

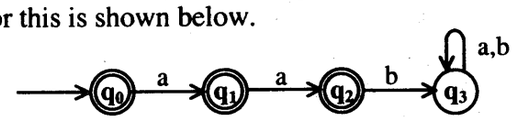
Example 2.6: Obtain a DFA to accept strings of a's and b's except those containing the substring aab.

Note: This can be solved in two ways. The first method is similar to the previous problem i.e., draw a DFA to accept strings of a's and b's having a substring *aab*. Then change the final states to non-final states and non-final states to final states. The resulting machine will accept the strings of a's and b's except those containing the sub-string *aab*.

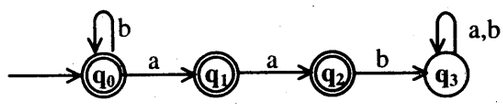
Here, the second method is explained. The minimum string that can be rejected by the machine is *aab*. To reject this string we need four states q_0, q_1, q_2 and q_3. Since the string *aab* has to be rejected, q_3 can not be the final state and the rest of the states will be the final states as shown below.



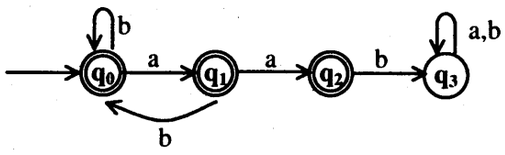
The machine enters into q_3 if the string has a sub string *aab*. In this state if we input any number of a's or/and b's, the entire string has to be rejected. So, stay in the state q_3 only. The machine for this is shown below.



In state q_0, if the input symbol is b, stay in q_0 so that if this b is followed by *aab*, the machine enters into state q_3 so that the string is rejected. The machine for this is shown below.



In state q_1, if the input symbol is *b*, enter into state q_0, so that if this *b* ends with the string *aab*, the entire string is rejected. The machine for this is shown below.



The machine will be in state q_2 if the string ends with *aa*. At this stage, if the input symbol is *a*, again the string ends with *aa* and so stay in state q_2 only. The complete machine to accept strings of a's and b's except those containing the sub string *aab* is shown in figure 2.16.

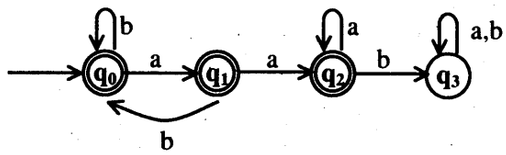


Fig.2.16 DFA to accept the string except the sub string aab.

So, the DFA M = (Q, Σ, δ, q_0, F) where

Q = {q_0, q_1, q_2, q_3}

Σ = {a, b}

q_0 is the start state

F = {q_0, q_1, q_2}

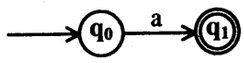
δ is shown using the transition table 2.7

↓ States →	δ	←Σ→ 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

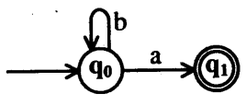
Table 2.7 Transition table

Example 2.7: Obtain DFAs to accept strings of a's and b's having exactly one a, atleast one a, not more than three a's.

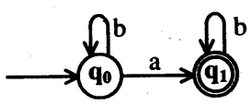
To accept exactly one a: To accept exactly one *a*, we need two states q_0 and q_1 and make q_1 as the final state. The machine to accept one *a* is shown below.



In q_0, on input symbol *b*, remain q_0 only so that any number of b's can end with one *a*. The machine for this can be of the form



In state q_1, on input symbol *b* remain in q_1 and the machine can take the form



But, in state q_1, if the input symbol is *a*, the string has to be rejected as the machine can have any number of b's but exactly one *a*. So, the string has to be rejected and we enter into a trap state q_2. Once the machine enters into trap state, there is no way to come out of the state and the string is rejected by the machine. The complete machine is shown in figure 2.17.

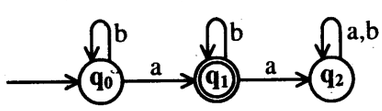


Fig.2.17 DFA to accept exactly one a.

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

$$L = \{ b^m a b^n \mid m, n \geq 0 \}$$

or

$$L = \{ b^* a b^* \}$$

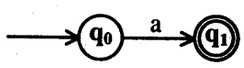
The machine $M = (Q, \Sigma, \delta, q_0, F)$ where

$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{ a, b \}$
q_0 is the start state
$F = \{q_1\}$
δ is shown below using the transition table 2.8.

↓ States →	δ	←Σ→ a	b
	→q_0	q_1	q_0
	(q_1)	q_2	q_1
	q_2	q_2	q_2

Table 2.8 Transition table

The machine to accept at least one *a*: The minimum string that can be accepted by the machine is *a*. For this, we need two states q_0 and q_1 where q_1 is the final state. The machine for this is shown below.



In state q_0, if the input symbol is *b*, remain in q_0. Once the final state q_1 is reached, whether the input symbol is *a* or *b*, the entire string has to be accepted. The machine to accept at least one *a* is shown in figure 2.18.

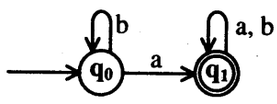


Fig.2.18 DFA to at least one a

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

$$L = \{ b^m a(a+b)^n \mid m, n \geq 0 \}$$

or

$$L = \{b^* a(a+b)^* \}$$

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

$Q = \{q_0, q_1\}$

$\Sigma = \{a, b\}$

q_0 is the start state

$F = \{q_1\}$

δ is shown using the transition table 2.9

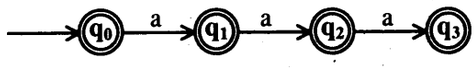
↓ States →	δ	←Σ→ a	b
	→q_0	q_1	q_0
	(q_1)	q_1	q_1

Table 2.9 Transition table

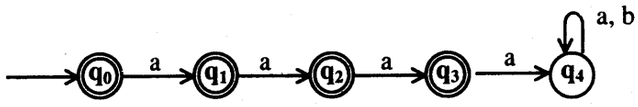
The machine to accept not more than three a's: The machine should accept not more than three a's means

- It can accept zero a's i.e., no a's
- It can accept one a
- It can accept two a's
- It can accept 3 a's
- But, it can not accept more than three a's.

In this machine maximum of three a's can be accepted i.e., the machine can accept zero a's, one a, two a's or three a's. So, we need maximum four states q_0, q_1, q_2 and q_3 where all these states are final states and q_0 is the start state. The machine can take the form



In state q_3, if the input symbol is *a*, the string has to be rejected and we enter into a trap state q_4. Once this trap state is reached, whether the input symbol is *a* or *b*, the entire string has to be rejected and remain in state q_4. Now, the machine can take the form as shown below.



In state q_0, q_1, q_2 and q_3, if the input symbol is *b*, stay in their respective states and the final transition diagram is shown in fig.2.19.

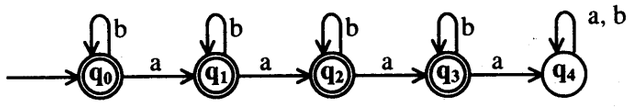


Fig.2.19 DFA to accept not more than 3 a's

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

$$L = \{ b^m a b^n a b^p a b^q \mid m, n, p, q \geq 0 \}$$

or

$$L = \{b^* a b^* a b^* a b^*\}$$

The DFA M = (Q, Σ, δ, q_0, F) where

Q = {q_0, q_1, q_2, q_3, q_4}

Σ = {a, b}

q_0 is the start state

$F = \{q_0, q_1, q_2, q_3\}$

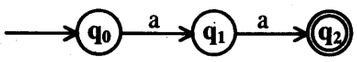
δ is shown using the transition table 2.10.

↓ States ↑	δ	←Σ→ a	b
	→q_0	q_1	q_0
	q_1	q_2	q_1
	q_2	q_3	q_2
	q_3	q_4	q_3
	q_4	q_4	q_4

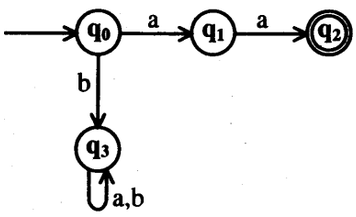
Table 2.10 Transition table for DFA shown in fig.2.19

Example 2.8: Obtain a DFA to accept the language L = {awa | w ∈ (a+b)* }

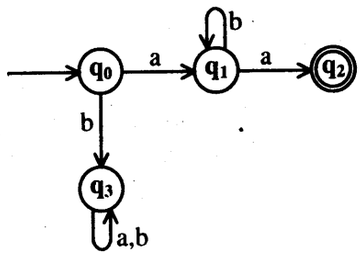
Here, w ∈ (a+b)* indicates the string consisting of a's and b's of any length including the null string. So, the language accepted by DFA is a string which starts with *a*, followed by a string of a's and b's (possibly including ε) of any length and followed by one *a*. If w is ε (epsilon i.e., the null string), the minimum string that can be accepted by the machine is *aa* and so, we need three states q_0, q_1 and q_2 to accept the string. The machine can be of the form



where q_0 is the start state and q_2 is the final state. In state q_0, if the input symbol is *b*, the string has to be rejected and so, we enter into a trap state q_3. Once the machine enters into trap state, whether the input is either *a* or *b*, the string has to be rejected and the machine for this is shown below.



In state q_1, if the input symbol is *b*, remain in q_1 and the machine takes the form



In state q_2, if the input symbol is *a*, the string ends with *a* and so remain in q_2. In state q_2, if the input symbol is *b*, enter into state q_1 so that after inputting the symbol *a*, the machine enters into q_2. The complete machine is shown in fig.2.20.

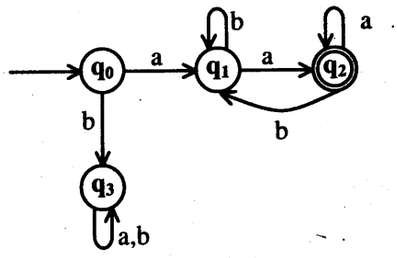


Fig.2.20 DFA to accept awa.

So, the machine M = (Q, Σ, δ, q_0, F) where

Q = {q_0, q_1, q_2, q_3}
Σ = {a, b}
q_0 is the start state
F = {q_2}
δ is shown using the transition table 2.11

	δ	←Σ→ a	b
	→q_0	q_1	q_3
↑ States	q_1	q_2	q_1
		q_2	q_1
↓	q_2	q_3	q_3	q_3

Table 2.11 Transition table for the DFA shown in fig. 2.20

2.6 General procedure to design a FA

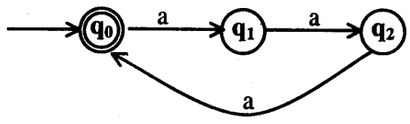
On the basis of construction of various FA's we can arrive at the following general procedure. Using this procedure, the number of errors in designing of FA will be minimized.

Step 1: Identify the minimum string that can be accepted by the machine and draw the FA
Step 2: Make sure that there is one start state and at least one final state.
Step 3: For the basic FA defined in step 1, consider one state at a time and obtain the various transitions on each input symbol so that the string stated in the problem is accepted
Step 4: Make sure that no two transitions are defined for the same symbol

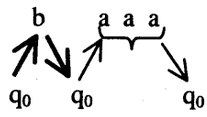
Example 2.9: Obtain a DFA to accept set of strings such that number of a's in every string is a multiple of 3 where $\Sigma = \{a, b\}$

Solution: The string can be any combination of a's and b's. But, the number of a's in those strings should be multiples of 3 i.e., {0, 3, 6, 9, ...}.

Step 1 and 2 : The basic FA can be written to accept multiples of three a's with one start state q_0 and one final state q_0 as shown below

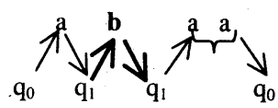


Step 2: Consider one state at a time and define the transitions for the input symbol b as shown below:
Consider q_0: q_0 on baaa can be obtained as shown below:

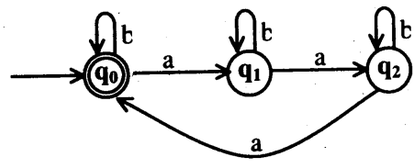


Observe that q_0 on **aaa** the machine will go to final state q_0. But, just before **aaa**, if we input **b**, there should be a transition from q_0 to q_0. So, $\delta(q_0, b) = q_0$.

Consider q_1: q_1 input symbol **b** can be obtained as shown below:

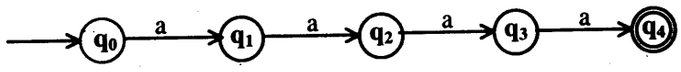


Basic string to be accepted is **a a a**. But, from the state q_1 on input symbol **b**, observe that the machine has to be in q_1. So, $\delta(q_1, b) = q_1$. Similarly $\delta(q_2, b) = q_2$. So, the final DFA is shown below:

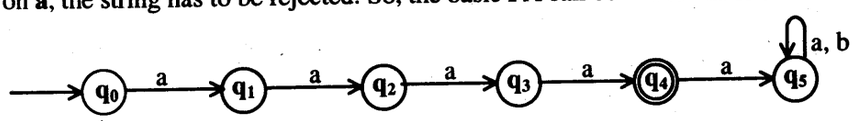


Example 2.10: Obtain a DFA to accept strings of a's and b's having four a's where $\Sigma = \{a, b\}$

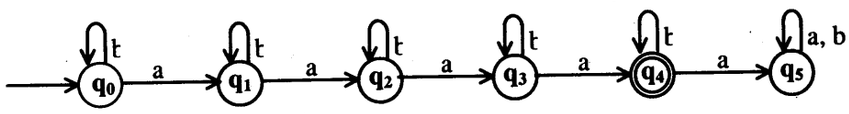
Solution:
The minimum string that is accepted by FA is aaaa. So, the machine should have at least 5 states as shown below:



But, q_4 on **a**, the string has to be rejected. So, the basic FA can be written as shown below:



From each state on **b**, the machine remains in the same state. So, the final DFA can be written as 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 2.11: Obtain a DFA to accept even number of a's and odd number of a's.

Solution: The machine to accept even number of a's is shown in fig.2.21.a and odd number of a's is shown in fig.2.21.b.

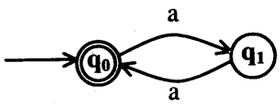


Fig. 2.21(a)

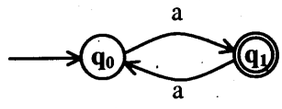


Fig. 2.21(b)

Example 2.12: Draw a DFA to accept strings of a's and b's with even number of a's and b's

Solution: The machine to accept even number of a's and b's is shown in fig.2.22.

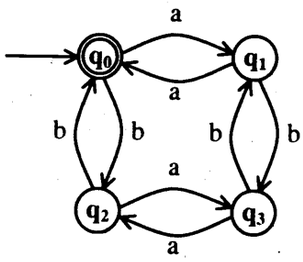


Fig.2.22 DFA to accept even no. of a's and b's

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

$$L = \{ w \mid w \in (a+b)^* \text{ and } N_a(w) \text{ as well as } N_b(w) \text{ is even } \}$$

or

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

Here, N_a is the total number of a's in the string *w* and N_b is the total number of b's in the string *w*.

Note: In the DFA shown in figure 2.22, instead of making q_0 as the final state, make q_2 as the final state. The DFA to accept even number of a's and odd number of b's is obtained and is shown in figure 2.23.

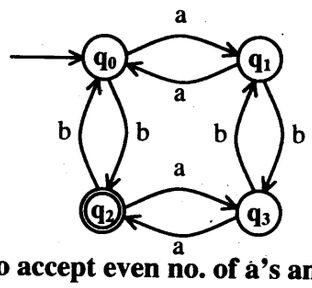


Fig.2.23 DFA to accept even no. of a's and odd number of b's

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

$$L = \{ w \mid w \in (a+b)^* \text{ and } N_a(w) \text{ is even and } N_b(w) \text{ is odd}\}$$

or

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

Note: In the DFA shown in figure 2.22, instead of making q_0 as the final state, make q_1 as the final state. The DFA to accept odd number of a's and even number of b's is obtained and is shown in figure 2.24.

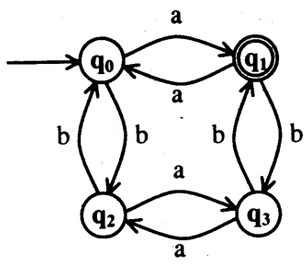


Fig.2.24 DFA to accept odd no. of a's and even number of b's

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

$$L = \{ w \mid w \in (a+b)^* \text{ and } N_a(w) \text{ is odd and } N_b(w) \text{ is even}\}$$

or

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

Note: In the DFA shown in figure 2.22, instead of making q_0 as the final state, make q_3 as the final state. The DFA to accept odd number of a's and odd number of b's is obtained and is shown in figure 2.25.

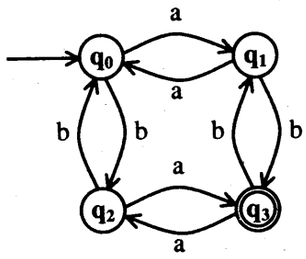


Fig.2.25 DFA to accept odd no. of a's and odd number of b's

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

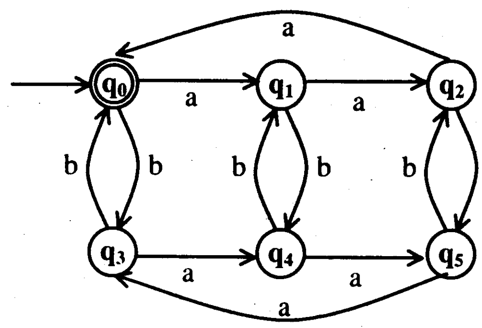
$$L = \{ w \mid w \in (a+b)^* \text{ and } N_a(w) \text{ and } N_b(w) \text{ is odd}\}$$

or

$$L = \{w \mid w \in (a+b)^* N_a(w) \bmod 2 = 1 \text{ and } N_b(w) \bmod 2 = 1\}$$

Example 2.13: Obtain a DFA to accept strings of a's and b's such that $L = \{w \mid w \in (a+b)^*$ such that $N_a(w) \bmod 3 = 10$ and $N_b(w) \bmod 2 = 0\}$

Solution: It is similar to the previous problem (figure 2.22) with the modifications shown below:



Note: If q_1 is the final state, the language accepted by above machine is

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

If q_2 is the final state, the language accepted by above machine is

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

If q_3 is the final state, the language accepted by above machine is

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

If q_4 is the final state, the language accepted by above machine is

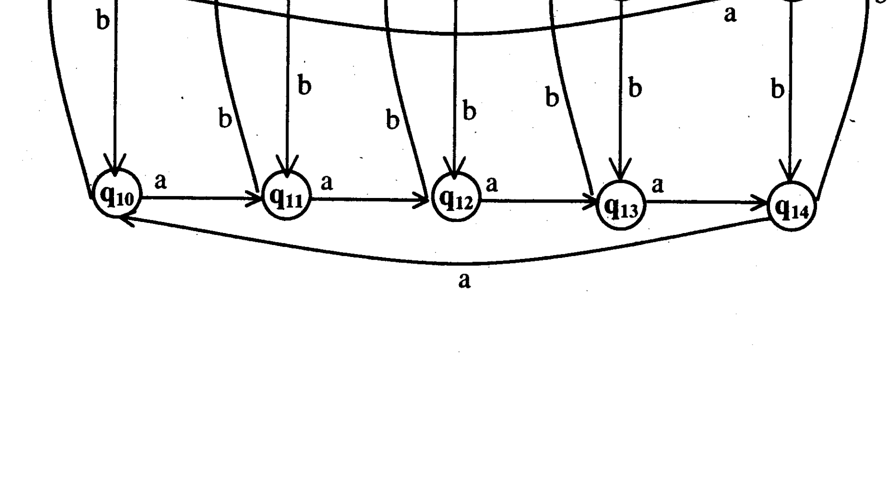
$L = \{w \mid w \in (a+b)^* \; N_a(w) \bmod 3 = 1 \text{ and } N_b(w) \bmod 2 = 1\}$

If q_5 is the final state, the language accepted by above machine is

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

Example 2.14: 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.

Solution: It is similar to the previous problem (figure 2.22) with the modifications shown below:



Based on the final state, the language accepted by the above machine changes as shown below:

- If q_0 is the final state, the language accepted by above machine is

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

or

The language accepted by the machine is the strings of a's and b's such that number of a's is divisible by 5 and number of b's is divisible by 3.

- If q_7 is the final state, the language accepted by above machine is

$$L = \{w \mid w \in (a+b)^* N_a(w) \bmod 5 = 2 \text{ and } N_b(w) \bmod 3 = 1\}$$

Thus, based on the final state, the language accepted by the machine changes.

Example 2.15: 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*'.

The machine to accept the corresponding string is shown in figure 2.26.

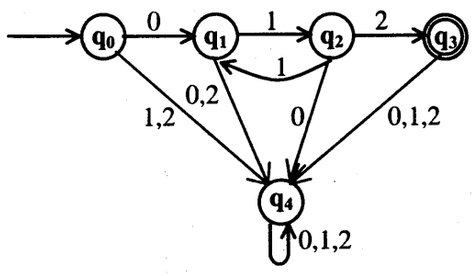


Fig.2.26 DFA to accept the language { w | w ∈ 0(11)*12 }

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

$$L = \{ w \mid w \in 0(11)^*12 \}$$

which is the language formed by words that begin with a '0' followed by an odd number of 1's and ending with a '2'.

Example 2.16: Obtain a DFA to accept odd and even numbers represented using binary notation

Before designing, let us think of how a problem can be solved. We know that any number ending with 1 is odd and ending with 0 is even. So, whenever an input is 0, let us enter into state q_1 and whenever the input is 1, we enter into state q_2. When the machine is in state q_1, it accepts even number and when the machine is in state q_2, it accepts odd umber. The machine to accept binary odd and even numbers is shown in fig.2.27.

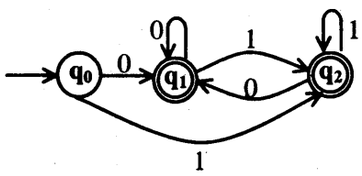


Fig.2.27 DFA to accept binary odd and even numbers

Note: Whenever the machine is in state q_1, even numbers are accepted and whenever the machine is in q_2, odd numbers are accepted.

Example 2.17: 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.

The machine to accept the corresponding language is shown in figure 2.28.

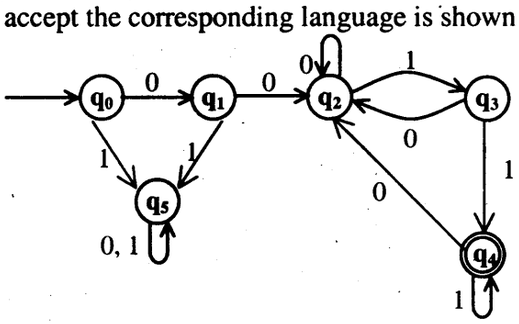


Fig.2.28 DFA to accept strings starting with at least two 0's and ending with at least two 1's

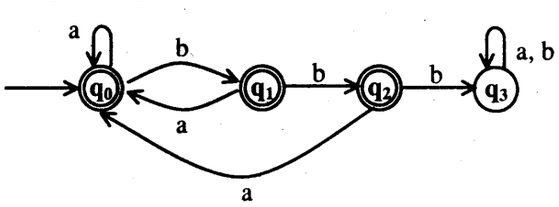
In set notation, the language accepted DFA can be represented as

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

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

Example 2.18: Obtain a DFA to accept strings of a's and b's with at most two consecutive b's.

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



Example 2.19: Obtain a DFA to accept the language L = { w : |w| mod 3 = 0} on Σ = {a, b}.

Note that the string having a length of mod 3 should be accepted. Using the set notation, the language can be denoted as

L = {ε, aaa, aaaaaa, aaaaaaaaa, ...and so on.}

Each string in the set L can be any combination of a's and b's with a length of multiples of 3 (The example shows only strings of a's). The machine to accept the corresponding language is shown below:

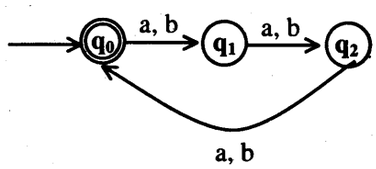


Fig.2.30 DFA to accept words of |w| mod 3 = 0

Example 2.20: Obtain a DFA to accept the language L = { w : |w| mod 5 ≠ 0 } on Σ = {a, b}.

The number of symbols in a string consisting of a's and b's should not have multiples of 5. The machine to accept the corresponding language is shown below:

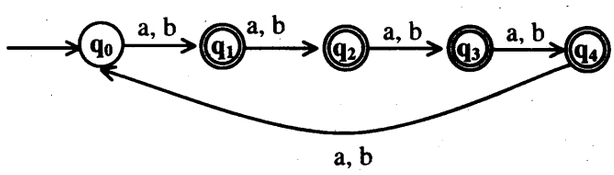


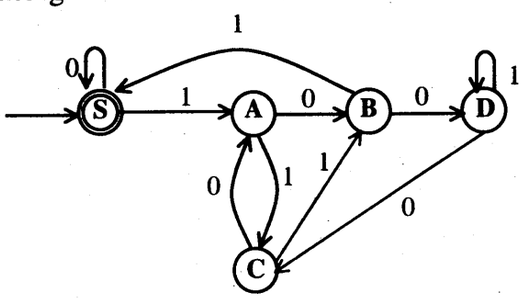
Fig.2.31 DFA to accept words of |w| mod 5 ≠ 0

Example 2.21: 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. After constructing the DFA, obtain the minimum DFA.

Note: If each string represents a modulo 5, the modulo 5 integer may represent either 0, 1, 2, 3 or 4 (equivalent decimal value). For each number we shall represent states S, A, B, C and D respectively. The states for the corresponding binary numbers can be obtained as shown:

Decimal	Binary	State
0	0000	S
1	0001	A
2	0010	B
3	0011	C
4	0100	D
5	0101	S
6	0110	A
7	0111	B
8	1000	C
9	1001	D
10	1010	S
11	1011	A
12	1100	B
13	1101	C
14	1110	D
15	1111	S

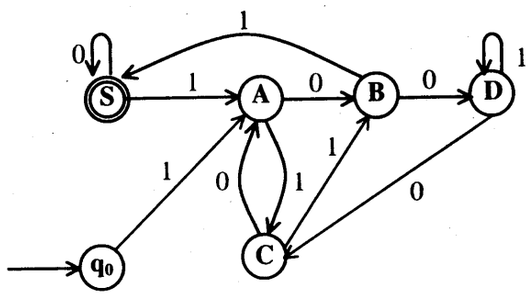
and so on. By looking at the above table, we can easily construct the DFA and the corresponding DFA is shown below:



After minimization, we get the same DFA. The DFA obtained can not be minimized.

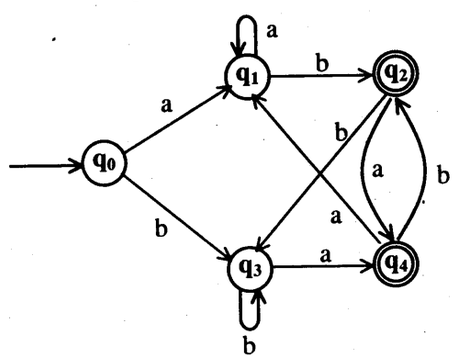
Example 2.22: Obtain a DFA which accepts the set of all strings beginning with a 1 that when interpreted as a binary integer, is a multiple of 5. For example, 101, 1010, 1111 etc are multiples of 5. Note that 0101 is not beginning with 1 and it should not be accepted.

Note: The solution to this problem is almost similar to the previous problem. But, the number always should start with a 1. If the strings starts with a 0, the number should never be accepted. So, if the first symbols is 1, let us have a new start state q_0 and on input symbol 1 enter into state A as shown below.



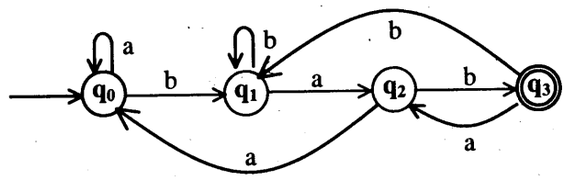
Example 2.23: Obtain a DFA to accept the language L = {w(ab + ba) | w ∈ {a, b}*}

Solution: The DFA to accept the language L = {w(ab+ba) | w ∈ {a, b}*} is shown below:



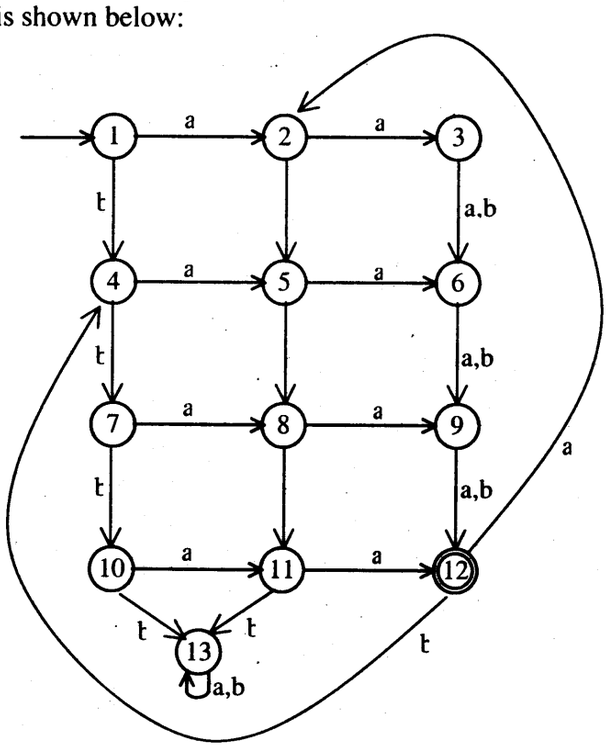
Example 2.24: Obtain a DFA to accept the language L = {wbab | w ∈ {a, b}*}

Solution: The DFA to accept the language L = {wbab | w ∈ {a, b}*} is shown below:



Example 2.25: Obtain a DFA to accepts strings of a's and b's such that each block of 5 consecutive symbols have at least two a's

In any of block of 5 consecutive symbols, if there are four or five b's the entire string should be rejected and so the trap state is reached. Otherwise, the string has at least two a's and the DFA is shown below:



2.7 Regular language

In this section, let us see **"What is a regular language?"**

Definition: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The language L is regular if there exists a machine M such that $L = L(M)$. That is, if a given language L is regular, definitely there exists a DFA M which can accept the language L(M) such that $L = L(M)$. In other words, if a language is regular we can definitely obtain a DFA M which accepts the same language. All the languages which have been accepted by the DFA's we have discussed so far are all regular.

Note 1: For a given language, if it is not possible to construct a DFA, the language is not regular language. So, whenever we want to check whether a language is regular or not, try to construct DFA. If we have a DFA for that language, the language is regular and if it is not possible to construct a DFA, the given language is not regular.

Note 2: We have obtained number of DFA's accepting the various languages. Since DFAs exists, all the corresponding languages are regular languages.

2.8 Applications of Finite Automata

We should know that there are infinite number of applications that uses the concept of finite automata in some form or the other. Any application that runs on the computer uses this concept. This section covers very few applications where the concept of finite automata is used. Some of the applications of finite automata are:

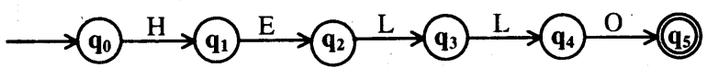
1. String matching/processing
2. Compiler Construction
3. Other applications

String matching/processing

While editing the text using text editor usually we search for a specified string in a file and then edit. In UNIX environment we will be using the UNIX program *grep* to search for a specified pattern in a file. In general, searching for a specified string is a very important activity in the field of computer science encouraging efficient algorithms to be written for string matching. The finite automaton will help us devising the efficient algorithms and test whether the program perfectly works or not even before its implementation. The following section explains how a pattern matching or string matching algorithm can be derived using finite automaton.

We know that DFA's are one way of representing the language. In our example, a language means a set of strings matching some pattern. Once the machine is ready to accept the specified string, we can easily write an algorithm or a program to recognize that pattern. We can test whether the pattern matches just by running the machine. Once the pattern string matches using the machine, definitely our algorithm or program works perfectly. There is no need to test it again. So, instead of explaining how to design an algorithm, let us see how to construct a machine or how to draw a DFA using circles and arrows. The pictorial representation of DFA using circles, arrows etc is nothing but the state diagram or the transition diagram.

Let us write a DFA to check whether the pattern string "HELLO" is present in a text file. The explanation is similar to the problem given in example 2.5. We are interested in the characters 'H', 'E', 'L', 'L', 'O' in that sequence only for pattern matching. Since the string length is 5, the maximum number of states will be 6. Let the 6 states be 'S', 'H', 'HE', 'HEL', 'HELL', 'HELLO" where 'S' is the start state. For simplicity, let us rename those states as q_0, q_1, q_2, q_3, q_4 and q_5 and the initial diagram can take the form as shown below:



It is clear from the figure that whenever

1. the machine is in state q_1 , it has so far accepted the character 'H'
2. when in state q_2 , it has so far accepted 'HE'
3. when in state q_3 , it has so far accepted 'HEL'
4. when in state q_4 , it has so far accepted 'HELL'
5. when in state q_5 , it has so far accepted 'HELLO'

So, when the machine is in state q_5, the pattern string is found and at this point, we can return *true*. If the machine is in any of the states such as q_0, q_1, q_2, q_3 and q_4, and if the input symbol is 'H', there should be a transition to state q_1. Other than those symbols specified at the respective states, if any other symbol is encountered, there should be a transition to the start state q_0. It means that the longest prefix "HELLO" is not found in the string and to recognize this string we have to start from the start state q_0. The complete diagram is shown in figure 2.32.

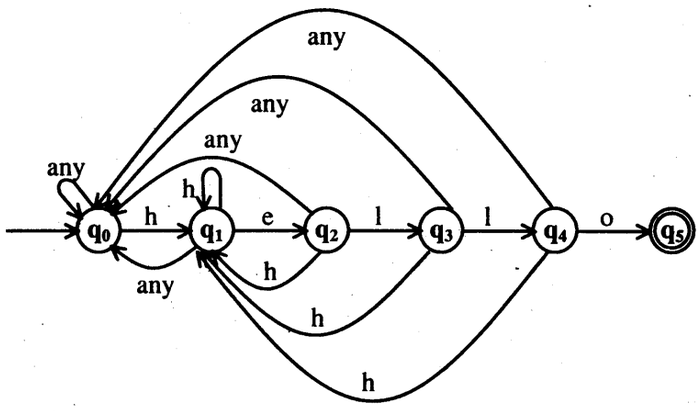


Fig.2.32 Transition diagram to match the string "HELLO"

By looking at the above DFA, we can write the C program as shown below:

```
#define Q0     0
#defineQ1      1
#define Q2     2
#define Q3     3
#define Q4     4
#define Q5     5
#define TRUE 1
#define FALSE 0
```

```
int search_pattern(char txt[])
{
        int     state;
        int     i;
        char    symbol;

        state = Q0;

        for ( i = 0; i < strlen(txt); i++ )
        {
                symbol = txt[i];

                switch (state)
                {
                        case Q0:
                                if (symbol == 'H') state = Q1;
                                break;
                        case Q1: /* Recognizes the string H */
                                if (symbol == 'E')
                                        state = Q2;
                                else if (symbol != 'H')
                                        state = Q0;
                                break;

                        case Q2: /* Recognizes the string HE */
                                if (symbol == 'H')
                                        state = Q1;
                                else if (symbol == 'L')
                                        state = Q3;
                                else
                                        state =  Q0;
                                break;

                        case Q3: /* Recognizes the string HEL */
                                if (symbol == 'H')
                                        state = Q1;
                                else if (symbol == 'L')
                                        state = Q4;
                                else
                                        state =  Q0;
                                break;
```

```
                    case Q4: /* Recognizes the string HELL */
                            if (symbol == 'H')
                                    state = Q1;
                            else if (symbol == 'O')
                                    state = Q5;
                            else
                                    state =  Q0;
                            break;

                    case Q5: /* Recognizes the string HELLO */
                            return TRUE;
            }
        }
        return FALSE;
}
```

The above function returns TRUE whenever the sub string "HELLO" is present in the text; otherwise, the function returns FALSE. Note that the string "HELLO" may appear number of times. The above program can be modified with little bit of effort to return the position of each occurrence of the string "HELLO" in the text.. This program can also be modified to implement the *grep* command in UNIX.

Compiler Construction

The various compilers such as C/C++, Pascal, Fortran or any other compiler is designed using the finite automata. The DFAs are extensively used in the building the various phases of compiler such as

- Lexical analysis (To identify the tokens, identifiers, to strip of the comments etc.)
- Syntax analysis (To check the syntax of each statement or control statement used in the program)
- Code optimization (To remove the un wanted code)
- Code generation (To generate the machine code)

The details of the compiler construction is not the scope of this subject. Even then let us see how the comments are eliminated from C programs using DFAs in lexical phase of the compiler construction. We know that before actual compilation starts, all the comments will be deleted and only the un-commented statements are used during the compilation process. The C processor (part of C compiler) is used to strip the comments. The DFA for this can take the form as shown in figure 2.33.

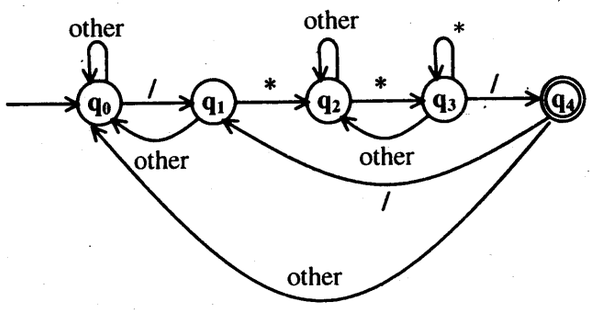


Fig.2.33 Transition diagram to remove the C-comments

By looking at the DFA, we can easily write the C program (similar to the pattern matching DFA and program in previous section). The transition table to strip out the comments is shown in table 2.12.

	δ	←Σ→ /	*	←Σ→ \n	other
	→q_0	q_1	q_0	q_0	q_0
↑	q_1	q_0	q_2	q_0	q_0
States	q_2	q_2	q_3	q_2	q_2
↓	q_3	q_4	q_3	q_2	q_2
	q_4	q_1	q_0	q_0	q_0

Table 2.12 Transition table to strip out the comments in C program

Other applications

The concept of finite automata is used in wide applications. It is not possible to list all the applications as there are infinite number of applications. This section lists some applications:

1. Large natural vocabularies can be described using finite automaton which includes the applications such as spelling checkers and advisers, multi-language dictionaries, to indent the documents, in calculators to evaluate complex expressions based on the priority of an operator etc. to name a few. Any editor that we use uses finite automaton for implementation.
2. Finite automaton is very useful in recognizing difficult problems i.e., sometimes it is very essential to solve an un-decidable problem. Even though there is no general solution exists for the specified problem, using theory of computation, we can find the approximate solutions.

3. Finite automaton is very useful in hardware design such as circuit verification, in design of the hardware board (mother board or any other hardware unit), automatic traffic signals, radio controlled toys, elevators, automatic sensors, remote sensing or controller etc.
4. In game theory and games, economics, computer graphics, linguistics etc., finite automaton plays a very important role.

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 | w ∈ (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| mod 3 = 0 on Σ = {a, b}.
22. Obtain a DFA to accept the language L = { w : |w| mod 5 ≠ 0 on Σ = {a, b}
23. What is a regular language?
24. What are the applications of finite automaton?
25. Draw a DFA to remove the comments from C program.

2.9 Non deterministic finite automata(NFA)

Consider the transition diagram shown in figure. 2.34. To start with the machine will be in state q_0. If the first input symbol is *a*, the machine can enter into either state q_1 or q_2

(since there are two transitions on input symbol *a* from state q_0). In state q_0, if the input symbol is *b* the machine enters into state q_2. Similarly from state q_1, there are multiple transitions on an input symbol *a* to the states q_1 and q_2 i.e., the machine can either enter into state q_2 or state q_1. At this point of time, we can not determine exactly in which state the machine will be. So, this is called Non Deterministic Finite Automaton (NFA).

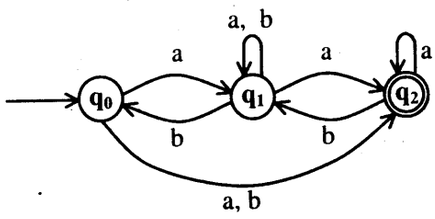


Fig. 2.34 Transition diagram of an NFA

The transitions from each state of the machine shown in figure 2.34, based on the input alphabets {a, b} are shown in table 2.13.

Current State	Input	Next state	Representation
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_1	$\delta(q_1,b) = \{q_0,q_1\}$
q_2	a	q_2	$\delta(q_2,a) = q_2$
q_2	b	q_1	$\delta(q_2,b) = q_1$

Table 2.13 Transitions for the machine shown in figure 2.34.

Consider only the transitions (last column in table 2.13) defined for the automaton. For our convenience, the transitions can be written as shown below:

$\delta(q_0,a) = \{q_1,q_2\}$
$\delta(q_0,b) = q_2$
$\delta(q_1,a) = \{q_1,q_2\}$
$\delta(q_1,b) = \{q_0,q_1\}$
$\delta(q_2,a) = q_2$
$\delta(q_2,b) = q_1$

set of subsets of Q i.e., 2^Q

Note: The set of subsets of set Q is called power set and is denoted by 2^Q.

δ: Q X Σ to 2^Q

where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0,1\}$. Note that in each $\delta(q, a)$ defined above, $q \in Q$ and $a \in \Sigma$. The state q_0 is the initial state and the state with two concentric circles (i.e., q_2) is the final state. With this concept, now, let us see **"What is a non-deterministic finite automata?"** A non-deterministic finite automaton (machine) can formally defined as follows.

Definition: An NFA is a 5-tuple or quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where

Q is non empty, finite set of states.

Σ is non empty, finite set of input alphabets.

δ is transition function which is a mapping from $Q \times \Sigma$ to subsets of 2^Q.

This function accepts two arguments as the input with first argument being $q \in Q$ and the second argument as the symbol $a \in \Sigma$ and returns a set of states which are reachable from q on input *a*. This function shows change of state from one state to a set of states based on the input symbol.

$q_0 \in Q$ is the start state.

$F \subseteq Q$ is set of final states.

Suppose $\delta(q, a) = P$ where $q \in Q$, $P \in 2^Q$ and $a \in \Sigma$. Here, there will be a transition form state q to set of states P on an input symbol *a*. The transition function δ can be extended to $\hat{\delta}$ whenever string operations are involved. The transition from state q to set of states P, on the input string w can be written as $\hat{\delta}(q, w) = P$.

2.10 Properties of transition function

The different properties of the transition function are:

$$\delta(q, \varepsilon) = \hat{\delta}(q, \varepsilon) = q \qquad 2.4$$

$$\delta(q, wa) = \delta(\hat{\delta}(q, w), a) = P_j \qquad 2.5$$

$$\delta(q, aw) = \hat{\delta}(\delta(q, a), w) = P_q \qquad 2.6$$

where $q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$ and P_j and P_q are the set of states which are reachable from q.

2.11 Acceptance of language

A string is a sequence of symbols obtained from Σ. The set of all strings recognized by an automaton is called language. The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as follows.

Definition: Let M = (Q, Σ, δ, q_0, F) be an NFA where Q is set of finite states, Σ is set of input alphabets (from which a string can be formed), δ is transition from Q x Σ to 2^Q, q_0 is the start state and F is the final or accepting state. The string (also called language) *w* accepted by an NFA can be defined in formal notation as:

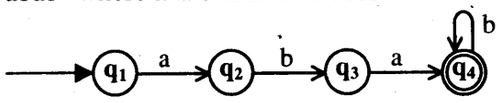
L(M) = { w | w $\in$ Σ*and $\hat{\delta}$ (q_0, w) = P with at least one component of P in F }. Here, P is set of states.

2.12 Need for Non-deterministic finite automaton

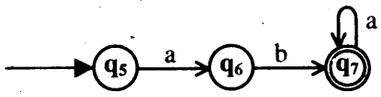
Digital computers are deterministic machines. Given the input, the state of the machine is predictable. Sometimes, constructing deterministic machine is difficult compared to non-deterministic machine. In such cases, there is a need to construct a machine very easily which can be achieved by constructing an NFA. After constructing an NFA, DFA can be easily constructed. This is an efficient mechanism to describe some complicated languages concisely. So, practically non-deterministic machines will not exist. But, one can construct an NFA easily and later that can be converted into DFA.

Example 2.26: Obtain an NFA to accept the following language L = {w | w $\in$ $abab^n$ or aba^n where n $\geq$ 0}

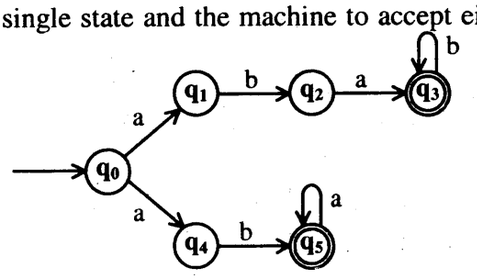
The machine to accept $abab^n$ where n $\geq$ 0 is shown below:



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 q1 and q5 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:



2.13 Conversion from NFA to DFA (Subset construction method)

Let $M_N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$ be an NFA and accepts the language $L(M_N)$. There should be an equivalent DFA $M_D = (Q_D, \Sigma_D, \delta_D, q_0, F_D)$ such that $L(M_D) = L(M_N)$. The procedure to convert an NFA to its equivalent DFA is shown below:

Step1:

The start state of NFA M_N is the start state of DFA M_D. So, add q_0(which is the start state of NFA) to Q_D and find the transitions from this state. The way to obtain different transitions is shown in step2.

Step2:

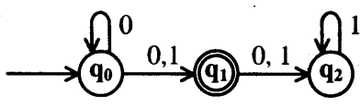
For each state $[q_i, q_j,....q_k]$ in Q_D, the transitions for each input symbol in Σ can be obtained as shown below:

1. $\delta_D([q_i, q_j,....q_k], a) = \delta_N(q_i, a) \cup \delta_N(q_j, a) \cup .....\delta_N(q_k, a)$
   $= [q_l, q_m,....q_n]$ say.
2. Add the state $[q_l, q_m,....q_n]$ to Q_D, if it is not already in Q_D.
3. Add the transition from $[q_i, q_j,....q_k]$ to $[q_l, q_m,....q_n]$ on the input symbol *a* iff the state $[q_l, q_m,....q_n]$ is not added to Q_D in the previous step.

Step3:

The state $[q_a, q_b,....q_c] \in Q_D$ is the final state, if at least one of the state in $q_a, q_b, ... q_c \in F_N$ i.e., at least one component in $[q_a, q_b,....q_c]$ should be the final state of NFA.

Example 2.27: Convert the following NFA to its equivalent DFA.



Step1: q_0 is the start of DFA (see step1 in the conversion procedure).

So, $Q_D = \{[q_0]\}$ (2.7)

Step2: Find the new states from each state in Q_D and obtain the corresponding transitions.

Consider the state $[q_0]$:

When $a = 0$

$$\delta_D([q_0], 0) = \delta_N([q_0], 0)$$
$$= [q_0, q_1] \quad (2.8)$$

When $a = 1$

$$\delta_D([q_0], 1) = \delta_N([q_0], 1) = [q_1] \quad (2.9)$$

Since the states obtained in (2.8) and (2.9) are not in Q_D(2.7), add these two states to Q_D so that

$$Q_D = \{[q_0], [q_0, q_1], [q_1]\} \quad (2.10)$$

Consider the state $[q_0, q_1]$:

When $a = 0$

$$\delta_D([q_0, q_1], 0) = \delta_N([q_0, q_1], 0) = \delta_N(q_0, 0) \cup \delta_N(q_1, 0) = \{q_0, q_1\} \cup \{q_2\} = [q_0, q_1, q_2] \quad (2.11)$$

When $a = 1$

$$\delta_D([q_0, q_1], 1) = \delta_N([q_0, q_1], 1) = \delta_N(q_0, 1) \cup \delta_N(q_1, 1) = \{q_1\} \cup \{q_2\} = [q_1, q_2] \quad (2.12)$$

Since the states obtained in (2.11) and (2.12) are the not defined in Q_D(see 2.10), add these two states to Q_D so that

$$Q_D = \{[q_0], [q_0, q_1], [q_1], [q_0, q_1, q_2], [q_1, q_2]\} \quad (2.13)$$

Consider the state $[q_1]$:

When $a = 0$

$$\delta_D([q_1], 0) = \delta_N([q_1], 0) = [q_2] \quad (2.14)$$

When $a = 1$

$$\delta_D([q_1], 1) = \delta_N([q_1], 1) = [q_2] \quad (2.15)$$

Since the states obtained in (2.14) and (2.15) are same and the state q_2 is not in Q_D(see 2.13), add the state q_2 to Q_D so that

$$Q_D = \{[q_0], [q_0, q_1], [q_1], [q_0, q_1, q_2], [q_1, q_2], [q_2]\} \quad (2.16)$$

Consider the state $[q_0,q_1,q_2]$:

When $a = 0$

$$\begin{aligned} \delta_D([q_0,q_1,q_2], 0) &= \delta_N([q_0,q_1,q_2], 0) \\ &= \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \cup \delta_N(q_2, 0) \\ &= \{q_0,q_1\} \cup \{q_2\} \cup \{\phi\} \\ &= [q_0,q_1,q_2] \end{aligned} \quad (2.17)$$

When $a = 1$

$$\begin{aligned} \delta_D([q_0,q_1,q_2], 1) &= \delta_N([q_0,q_1,q_2], 1) \\ &= \delta_N(q_0, 1) \cup \delta_N(q_1, 1) \cup \delta_N(q_2, 1) \\ &= \{q_1\} \cup \{q_2\} \cup \{q_2\} \\ &= [q_1, q_2] \end{aligned} \quad (2.18)$$

Since the states obtained in (2.17) and (2.18) are not new states (are already in Q_D, see 2.16), do not add these two states to Q_D.

Consider the state $[q_1,q_2]$:

When $a = 0$

$$\begin{aligned} \delta_D([q_1,q_2], 0) &= \delta_N([q_1,q_2], 0) \\ &= \delta_N(q_1, 0) \cup \delta_N(q_2, 0) \\ &= \{q_2\} \cup \{\phi\} \\ &= [q_2] \end{aligned} \quad (2.19)$$

When $a = 1$

$$\begin{aligned} \delta_D([q_1,q_2], 1) &= \delta_N([q_1,q_2], 1) \\ &= \delta_N(q_1, 1) \cup \delta_N(q_2, 1) \\ &= \{q_2\} \cup \{q_2\} \\ &= [q_2] \end{aligned} \quad 2.20)$$

Since the states obtained in (2.19) and (2.20) are not new states (are already in Q_D see 2.16), do not add these two states to Q_D.

Consider the state $[q_2]$:

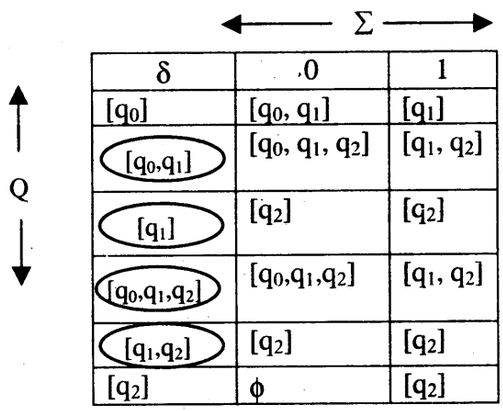
When $a = 0$

$$\delta_D([q_2], 0) = \delta_N([q_2], 0) = \{\phi\} \quad (2.21)$$

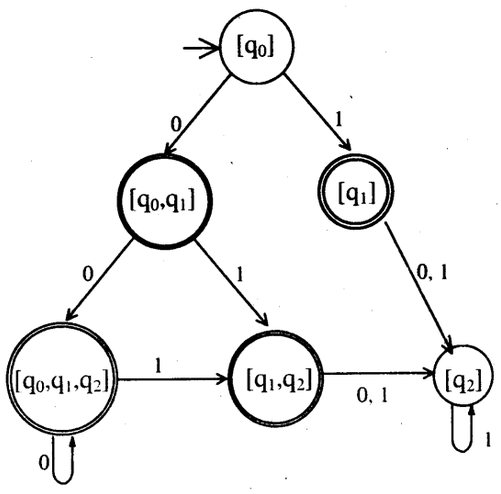
When $a = 1$

$$\delta_D([q_2], 1) = \delta_N([q_2], 1) = [q_2] \quad (2.22)$$

Since the states obtained in (2.21) and (2.22) are not new states (are already in Q_D, see 2.16), do not add these two states to Q_D. The final transitional table along with transition diagram is shown below:



δ	0	1
$[q_0]$	$[q_0, q_1]$	$[q_1]$
$[q_0,q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
$[q_1]$	$[q_2]$	$[q_2]$
$[q_0,q_1,q_2]$	$[q_0,q_1,q_2]$	$[q_1, q_2]$
$[q_1,q_2]$	$[q_2]$	$[q_2]$
$[q_2]$	ϕ	$[q_2]$



2.14 Finite Automata with Epsilon transitions (ε-NFA)

Consider the transition diagram shown in figure 2.36. To start with the machine will be in state q_0. If the first input symbol is *a*, the machine can enter into either state q_0 or q_1 (since there are two transitions on input symbol *a* from state q_0). In state q_0, if the input symbol is *b* the machine enters into state q_2. Also, if the machine is in state q_0 and if there is no input, the machine may stay in q_0 or it may enter into q_1/q_2 (using ε-transitions). Similarly from state q_1, there are multiple transitions on an input symbol *b* to the states q_1 and q_2 i.e., the machine can either enter into state q_2 or state q_1. In this FA, we can not determine exactly in which state the machine will be. So, this is Non Deterministic Finite Automaton (NFA). Because of ε-transitions, the NFA is called ε-NFA.

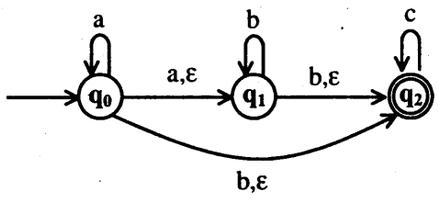


Fig. 2.36 Transition diagram of an ε-NFA

Before proceeding further, let us see **"What is ε-CLOSURE(q)?"** The ε-CLOSURE of q is formally defined as follows:

Definition: ε-CLOSURE(q) is read as ε-CLOSURE of q and is the set of all states which are reachable from q on ε-transitions only. The recursive definition of ε-CLOSURE(q) is:

1. ε-CLOSURE(q) = q for each $q \in Q$.
2. If ε-CLOSURE(q) = p and if $\delta(p, \varepsilon) = r$ then add r to ε-CLOSURE(q) i.e., ε-CLOSURE(q) = {p, r}

For the ε-NFA shown in figure 2.36, the ε-CLOSURE(q) for each $q \in Q$ is shown below:

$$\varepsilon\text{-CLOSURE}(q_0) = \{q_0, q_1, q_2\}$$
$$\varepsilon\text{-CLOSURE}(q_1) = \{q_1, q_2\}$$
$$\varepsilon\text{-CLOSURE}(q_2) = \{q_2\}$$

The transitions from each state of the machine shown in figure 2.36, based on the input alphabets {a, b, c} and ε (null character i.e., no input) are shown in table 2.15.

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	ϕ	
q_0	ε	q_1,q_2	$\delta(q_0, \varepsilon) = \{q_0,q_1\}$
q_1	a	ϕ	
q_1	b	q_1,q_2	$\delta(q_1,b) = \{q_1,q_2\}$
q_1	c	ϕ	
q_1	ε	q_2	$\delta(q_1, \varepsilon) = q_2$
q_2	a	ϕ	
q_2	b	ϕ	
q_2	c	q_2	$\delta(q_2,c) = q_2$
q_2	ε	ϕ	

Table 2.15 Transitions for the machine shown in figure 2.36.

Consider only the transitions (last column in table 2.15) defined for the automaton: $\delta(q_0,a) = \{q_0,q_1\}$, $\delta(q_0,b) = q_2$, $\delta(q_0, \varepsilon) = \{q_0,q_1\}$, $\delta(q_1,b) = \{q_1,q_2\}$, $\delta(q_1, \varepsilon) = q_2$, $\delta(q_2,c) = q_2$

For our convenience, the transitions can be written as shown below:

$\delta(q_0,a) = \{q_0,q_1\}$
$\delta(q_0,b) = q_2$
$\delta(q_0,\varepsilon) = \{q_0,q_1\}$
$\delta(q_1,b) = \{q_1,q_2\}$
$\delta(q_1,\varepsilon) = \{q_2\}$
$\delta(q_2,c) = q_2$

set of subsets of Q i.e., 2^Q

Note: The set of subsets of set Q is called power set and is denoted by 2^Q.

δ: Q x $(\Sigma \cup \varepsilon)$ to 2^Q

where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b, c\}$. Note that in each $\delta(q, a)$ defined above $q \in Q$, $a \in (\Sigma \cup \varepsilon)$. The state q_0 is the initial state and the state with two concentric circles (i.e., q_2) is the final state. With this concept, let us see **"What is ε-NFA?"** A non-deterministic finite automaton with ε-moves (ε-NFA) can formally be defined as follows.

Definition: An ε-NFA is a 5-tuple or quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where

Q is non empty, finite set of states.

Σ is non empty, finite set of input alphabets.

δ is transition function which is a mapping from Q x ($\Sigma \cup \varepsilon$) to subsets of 2^Q.

This function accepts two arguments as the input with first argument being $q \in Q$ and the second argument as the symbol $a \in (\Sigma \cup \varepsilon)$ and returns a set of states after taking the ε-CLOSURE of the states which are reachable from q on input *a*. This function shows change of state from one state to a set of states with or without the input symbol.

$q_0 \in Q$ is the start state.

$F \subseteq Q$ is set of final states.

Suppose $\delta(q, a)$ = ε-CLOSURE(P) where $q \in Q$, $P \in 2^Q$ and $a \in \Sigma$. Here, there will be a transition form state q to set of states P on an input symbol *a* then take the ε-CLOSURE(P). The transition function δ can be extended to $\hat{\delta}$ whenever string operations are involved. The transition from state q to set of states P, on the input string w can be written as

$$\hat{\delta}\ (q, w) = \varepsilon\text{-CLOSURE(P)}.$$

2.15 Properties of transition function

The different properties of the transition function are:

$$\delta(q, \varepsilon) = \hat{\delta}\ (q, \varepsilon) = \varepsilon\text{-CLOSURE}(q). \quad 2.4$$

$$\hat{\delta}\ (q, wa) = \delta(\hat{\delta}\ (q, w), a) = \varepsilon\text{-CLOSURE}(\delta(\hat{\delta}\ (q, w), a)). \quad 2.5$$

$$\hat{\delta}\ (q, aw) = \hat{\delta}\ (\delta(q, a), w) = \varepsilon\text{-CLOSURE}(\hat{\delta}\ (\delta(q, a), w)) \quad 2.6$$

2.16 Acceptance of language

A string is a sequence of symbols obtained from Σ. The set of all strings recognized by an automaton is called *language*. The language L accepted by an ε-NFA M = (Q, Σ, δ, q_0, F) is defined as follows.

Definition: Let M = (Q, Σ, δ, q_0, F) be an ε-NFA where Q is set of finite states, Σ is set of input alphabets (from which a string can be formed), δ is transition from Q x {$\Sigma \cup \varepsilon$} to 2^Q, q_0 is the start state and F is the final or accepting state. The string (also called language) *w* accepted by an NFA can be defined in formal notation as:

$$L(M) = \{ w \mid w \in \Sigma^* \text{and } \hat{\delta}(q_0, w) = P \text{ with at least one component of P in F } \}$$

Example 2.28: 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.