

CHAPTER 5

THEORY OF COMPUTATION

Syllabus: Regular languages and finite automata, context-free languages and pushdown automata, recursively enumerable sets and Turing machines, undecidability.

5.1 INTRODUCTION

Theory of computation is a branch of computer science and mathematics that studies whether a problem can be solved using an algorithm on a model of computation. This branch also studies that how efficiently that problem can be solved. This deals with two types of theories. First is computability theory, deals with up to which extent problem can be solved. Second is complexity theory, which deals with efficiency of solution. In this finite automata, push down automata, Turing machines, regular expressions and various grammars will be discussed.

Automata theory is the study of abstract machines (or more appropriately, abstract “mathematical” machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata. Computability theory deals with

the question of the extent to which a problem is solvable on a computer, whereas complexity theory considers not only whether a problem can be solved at all on a computer, but also how efficiently it can be solved. Two major aspects are considered in this—time complexity and space complexity.

Basic introduction and various technical terms for understanding of concept are described in the following section.

5.2 FINITE AUTOMATA

There are several things to be done in the designing and implementation of an automatic machine, but the most important question is that, what will be the behaviour of the machine? “Theory of Computation” is the subject which solves this purpose.

“Automata theory is the subject which describes the behaviour of automatic machines mathematically”. In other words, we can say that “Automata Theory is the study of self-operating virtual machines to help in logical understanding of input and output process without or with intermediate stages of computation”.

The model of automation is shown in Fig. 5.1.

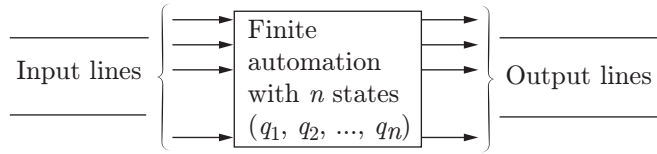


Figure 5.1 | Model of automation.

5.2.1 Finite Automaton Model Characteristics

The characteristics of finite automaton are described below:

- 1. Input:** Input values from input alphabet Σ contains $I_1, I_2, I_3, \dots, I_n$ at discrete time. are the input values from the input alphabet Σ at discrete time.
- 2. Output:** Output $O_1, O_2, O_3, \dots, O_n$ is the output set for this computation model are different outputs of this model.
- 3. States:** At any instance of time, the finite automaton will be in one of the states $q_1, q_2, q_3, \dots, q_n$, are the set of states on which the finite automaton will be at some instance of time.
- 4. States relation:** State relation describes that how to reach on next state using present input and output. At any instance of time, the next state of automaton is determined by the present input and present state.
- 5. Output relation:** Either one state or number of states can be obtained as output. The next state is achieved using state relation. Either only state or both the input and state are obtained as output. When an automaton reads an input symbol, it moves to the next state, which is given by the state relation.

5.2.2 Technical Terms

Technical terms that are required in finite automaton are given as following:

- 1. Alphabets:** A finite non-empty set of symbols are the alphabets of a language.

Example 5.1

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

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

Σ and Π are the sets which hold two alphabets. They are called binary alphabets for language.

$$(a + b)^* = \Sigma^* = \{\epsilon, a, b, aa, bb, ab, ba, \dots\}$$

where ϵ is called the identity element. Here, Σ^* contains every string possible by a, b .

$\Sigma^+ = \Sigma^* - \{\text{Null character } (\lambda)\}$, so Σ^+ will contain $\{a, b, aa, bb, ab, \dots\}$.

- 2. Strings:** Sequences of zero or more symbols are called string of a language. Let there be languages such as

$L_1 = a^*$, then it will consist $\{\epsilon, a, aa, aaa, aaaa, \dots\}$ string set.

$L_2 = ab^*$, then the string will be $\{a, ab, abb, abbb, \dots\}$.

$L_3 = 0^*1^*$, then it will consist $\{\epsilon, 0, 1, 01, 001, 0001, 011, 0111, \dots\}$ string set.

- 3. Concatenation:** Let there be two strings $U = ab$ and $V = aab$. Then concatenation of these two languages is $UV = abaab$ and $VU = aabab$. Concatenations of two strings satisfy associative property but not commutative. So, $UV \neq VU$.

- 4. Length of a string:** Let U, V and W be three strings. The length of a string tells the number of alphabets in that string.

If $U = 011$, length of $|U| = 3$.

If $V = ababa$, length of $|V| = 5$.

If $W = abababb$, length of $|W| = 7$.

- 5. Reversal of a string:** Let there be a string $W = aab$, then the reverse of W is

$$W^r = baa$$

If $W = W^r$, then the string is a palindrome. WW^r is called an even palindrome, and $W \times W^r$ is called an odd palindrome, where x is an alphabet. Let there be two strings U and V , then the reverse of UV is

$$|UV|^r = V^r U^r$$

- 6. Power of a string:** Let there be a string W , then

$$W^0 = \epsilon$$

$$W^1 = W$$

$$W^2 = W \cdot W$$

$$W^3 = W \cdot W^2, \text{ so } W^n + 1 = W \cdot W^n = W^n \cdot W$$

Let there be a string $U = \{010\}$, then the power of U :

$$U^0 = \epsilon$$

$$U^1 = \{010\}$$

$$U^2 = \{010010\}$$

- 7. Substring:** Any set of conjugative symbols is called a substring.

Example 5.2

Let $W = \{abc\}$ is a string. Substrings of W are $(\epsilon, a, b, c, ab, bc, abc)$, where ϵ is a substring of every string. If the length of a string is n , then the number of substrings possible is

$$\frac{n(n+1)}{2} + 1$$

- 8. Prefix of a string:** Prefix of a string $W = \{x | xy = W\}$

Consider a string $W = \{aabbcd\}$.

Prefix of $W = \{\epsilon, a, aa, aab, aabb, aabbc, aabbcd\}$

If the length of a string is n , then the number of prefix of that string will be $(n + 1)$.

- 9. Suffix of a string:** Suffix of a string $W = \{y | xy = W\}$

Consider a string $W = \{aabbcd\}$.

Suffix of $W = \{aabbcd, abbcd, bbcd, bcd, cd, d, \epsilon\}$

If the length of string is n , then the number of suffix of that string will be $(n + 1)$.

- 10. Language:** A language is a collection of strings of finite length constructed from alphabets of symbol. Or, we can say that a language is a subset of every string made by selected alphabets.

Let an alphabet set be $\{0, 1\}$, then Σ^* will contain all the strings made by 0 and 1. So language made by 0 and 1 will be the subset of Σ^* . Language $L \subseteq \Sigma^*$.

- 11. Language union:** Let $L_1 = \{ab, ba, aab\}$ and $L_2 = \{ab, abb, bb\}$ be two languages, then $L_1 \cup L_2 = \{ab, ba, aab, abb, bb\}$.
- 12. Language intersections:** Let $L_1 = \{ab, ba, aab\}$ and $L_2 = \{ab, abb, bb\}$ be two languages, then $L_1 \cap L_2 = \{ab\}$.
- 13. Language complement:** Let L be a language and L' the complement of language L . So, $L' = \{\Sigma^* - L\}$

Example 5.3

Given language L as $\{aa, ab, baa\}$, then complement of language L (L') = $\{a, b\}^* - \{aa, ab, baa\}$.

- 14. Language subtractions:** Let $L_1 = \{ab, ba, baa\}$ and $L_2 = \{aab, ba\}$ be two languages.

Then $L_1 - L_2 = \{ab, baa\}$ and $L_2 - L_1 = \{aab\}$.

- 15. Language XOR:** Let $L_1 = -ab, ba, baa''$ and $L_2 = -aab, ba''$ be two languages.

$$L_1 \oplus L_2 = (L_1 - L_2) \cup (L_2 - L_1) = (L_1 \cup L_2) - (L_1 \cap L_2)$$

$$L_1 \oplus L_2 = \{ab, baa, aab\}$$

- 16. $L_1 \cdot L_2$:** Let $L_1 = \{ab, ba, aab\}$ and $L_2 = \{ba, ab\}$, then

$$L_1 \cdot L_2 = \{abba, abab, baba, baab, aabba, aabab\}$$

- 17. L^* and L^+ :**

$$L^* = \{L^0 \cup L^1 \cup L^2 \cup L^3 \dots\}$$

$$L^+ = \{L^* - \lambda\}$$

5.2.3 Grammar

Grammar is a set of rules which generates every string in a language. A grammar is described by four terms $G = (V, T, S, P)$, where

$V \rightarrow$ finite non-empty set of variables (represented by capital letters)

$S \rightarrow$ starting symbol

$T \rightarrow$ finite non-empty set of terminals (represented by small letters)

$P \rightarrow$ finite non-empty set of production rule

Example 5.4

$$S \rightarrow AB \text{ (production 1)}$$

$$A \rightarrow a \text{ (production 2)}$$

$$B \rightarrow b \text{ (production 3)}$$

where S is the starting point, A and B are variables and a, b are terminals. We have three productions here. In grammar, V, T, P should be non-empty and finite.

5.2.4 Noam Chomsky Grammar Classification

Grammar is classified into the following four types:

- 1. Type 0:** This type of grammar is unrestricted, or phase structured, which generates all types of languages that can be recognized by a Turing machine. These languages are called recursive enumerable language. Unrestricted means these grammar have very less restrictions. Unrestricted grammar (G) = $u \rightarrow v$, where $u, v \in (V \cup T)^*$ and " u " must have at least one variable. V stands for variable and T for terminals.

Example 5.5

$$S \rightarrow aSb/\lambda$$

$$A \rightarrow aA/aB/Cb$$

$$Ba \rightarrow a/Da$$

- 2. Type 1:** This is context-sensitive grammar (CSG) which is used to generate context-sensitive languages. CSG (G) = $u \rightarrow v$, where $u, v \in (V \cup T)^*$ and " u " must have at least one variable. There is one additional rule also, $|u| \leq |v|$, which means length of " u " is less than or equal to " v ".

Example 5.6

$$S \rightarrow aSb/\lambda$$

$$Aa \rightarrow aA/aB/Cb$$

$$B \rightarrow a/Da$$

- 3. Type 2:** This is context-free grammar (CFG) which is used to generate context-free languages. CFG (G) = $u \rightarrow v$, where $v \in (V \cup T)^*$ and $|u| \leq |v|$. In addition to the CSG grammar, one extra rule, that is, only single variable is allowed in “ u ”.

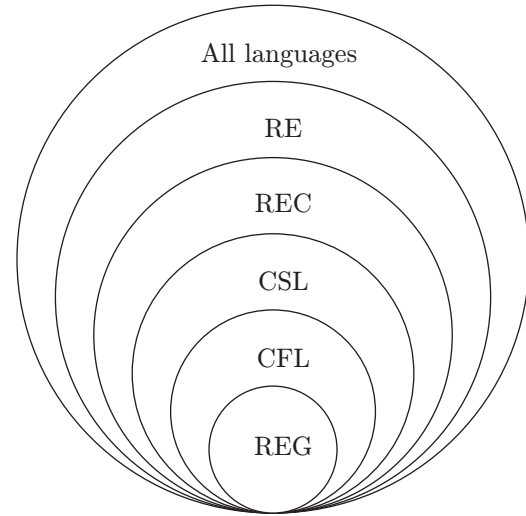
Example 5.7

$$\begin{aligned} S &\rightarrow aSb/\lambda \\ A &\rightarrow aA/aB/Cb \\ B &\rightarrow a/Da \end{aligned}$$

- 4. Type 3:** This is regular grammar, therefore language generated by these grammars is called regular languages. Regular grammar (G) = $u \rightarrow v$, where $|u| \leq |v|$, $\{v \in (T^*, T^*V, VT^*)$ and $u \in V\}$ and “ u ” has only one variable.

Example 5.8

$$\begin{aligned} S &\rightarrow aS/\lambda \\ A &\rightarrow aA/aB/Cb \\ B &\rightarrow a/Da \end{aligned}$$



REG—regular language; CFL—context-free language; CSL—context-sensitive language; REC—recursive language; RE—recursive enumerable.

Figure 5.2 | Chomsky hierarchy.

5.2.5 Chomsky Hierarchy

The hierarchy of grammar was described by Noam Chomsky in 1956, shown in Fig. 5.2.

5.2.6 Different Grammar Types

There have been numerous grammar types proposed for different languages. Table 5.1 provides a summary of the same.

Table 5.1 | Comparison of different types of grammar

Type	Grammar	Accepted by	Restriction on Production ($x \rightarrow y$)	Example
Type 0	Unrestricted Grammar	Turing Machine	1. x must have at least one variable (V) 2. y can be any string which consist any combination of variable (V) and terminal (T)	$AB \rightarrow a \mid Ba$
Type 1	Context Sensitive Grammar	Linear Bounded Automata	1. x must have at least one variable (V) 2. y can be any string which consist combination of variable (V) and terminal (T) 3. Length of y should be less than or equal to x .	$Aa \rightarrow aAb \mid aB \mid Cb$
Type 2	Context Free Grammar	Push Down Automata	1. x can have exactly one Variable (V) 2. y can be any string which consist combination of variable (V) and terminal (T) 3. Length of y should be less than or equal to x .	$A \rightarrow aBa \mid aB \mid Ba$
Type 3	Regular Grammar	Finite Automata	1. x can have exactly one Variable (V) 2. y can be any string which consist any combination from (VT^*, T^*V, T^*) , where V = one variable and T^* = any number of terminals 3. Length of y should be less than or equal to x .	$A \rightarrow a \mid Bb \mid bB$

5.3 FINITE AUTOMATA AND REGULAR LANGUAGE

Finite automata is classified into two categories—deterministic finite automata and non-deterministic finite automata.

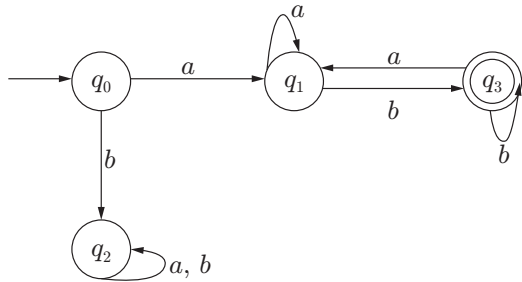
5.3.1 Deterministic Finite Automata

A deterministic finite automata (DFA) is defined by a five-tuple set $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite non-empty set of states.
2. Σ is a finite non-empty set of input called alphabets.
3. δ is a transition function which maps $Q \times \Sigma \rightarrow Q$.
4. $q_0 \in Q$, known as the initial state or starting state.
5. $F \subseteq Q$, known as a set of final states.

Example 5.9

Consider a DFA $M = (\{q_0, q_1, q_2, q_3\} \{a, b\} \{\delta\} \{q_0\} \{q_3\})$ as shown in the figure below.



The DFA can be represented in a transition table shown below, which can be determined by the transition function (δ).

δ	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_1	q_3
q_2	q_2	q_2
q_3	q_1	q_3

q_0 – Starting state

q_3 – Final state (accepting state)

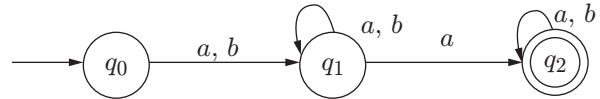
5.3.2 Non-deterministic Finite Automata

A non-deterministic finite automata (NFA) is defined by a five-tuple set $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite non-empty set of states.
2. Σ is a finite non-empty set of input called alphabets.
3. δ is a transition function which maps $Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$.
4. $q_0 \in Q$, known as the initial state or starting state.
5. $F \subseteq Q$, known as a set of final states.

Example 5.10

Consider an NFA $M = (\{q_0, q_1, q_2\} \{a, b\} \{\delta\} \{q_0\} \{q_2\})$ as shown in figure below.



The NFA can be represented in a transition table shown below, which can be determined by the transition function (δ).

δ	a	b
$\rightarrow q_0$	q_1	q_1
q_1	$\{q_1, q_2\}$	q_1
q_2	q_2	q_2

q_0 – Starting state

q_2 – Final state (accepting state)

5.3.3 Comparison of DFA and NFA

The comparison between DFA and NFA is given in Table 5.2.

5.3.4 DFA and NFA Design

We have taken a few examples and discussed how to design a DFA and an NFA for these cases. The following are four types of states which are used to design a DFA and an NFA:

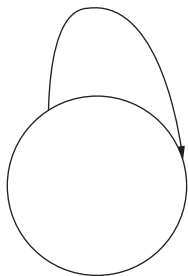
1. **Permanent accept state:** In this case, the initial and final states are the same. Due to self-loop this machine will always be on the same state, which is also final state. Hence, this is called permanent accept state.



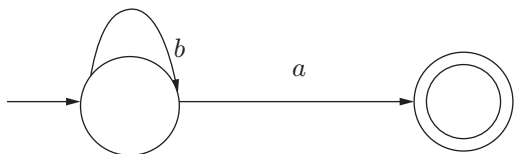
Table 5.2 | DFA vs. NFA

Title	DFA	NFA
Power	NFA and DFA powers are same	NFA and DFA powers are same
Supremacy	Some NFA are DFA, but not all	All DFA are NFA
Time complexity	Time needed to execute an input string is less than that required by NFA	Time needed to execute an input string is more than that required by DFA
Transition function	Maps $Q \times \Sigma \rightarrow Q$, number of next states at an input is exactly one	Maps $Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$, the number of next states at an input is zero, one or more.
Null moves	Does not allow null moves	Allows null moves
Moves	Only one move for single input alphabet	There can be choice (more than one move can be there) for single input alphabet

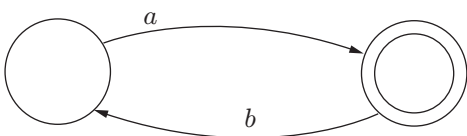
2. **Permanent reject state (trap state):** The given state is permanent reject state because there is self-loop to some non-final state.



3. **Temporary reject state:** The given state is temporary reject state because for input ‘b’, it has a loop on the non-final state. However, if after input ‘b’ there is another input ‘a’, then it will halt on final state.

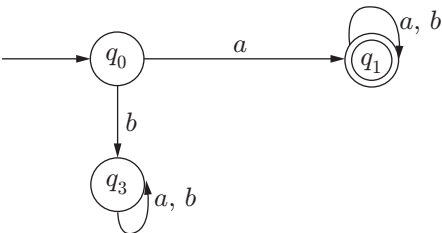


4. **Temporary accepting state:** The given state is temporary accepting state, as it reaches the final state if it has input symbol ‘a’, but if it gets another input symbol ‘b’ on that final state it will halt on non-final state.



Example 5.11

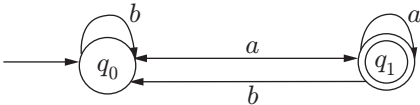
DFA which accepts string starting with “a”



Regular expression: $a(a + b)^*$

Example 5.12

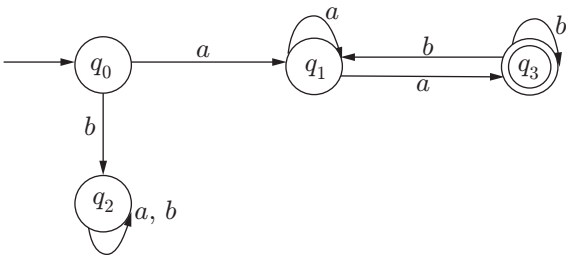
DFA which accepts string ending with “a”



Regular expression: $(a + b)^*a$

Example 5.13

DFA which accepts string starting with “a” and ending with “b”

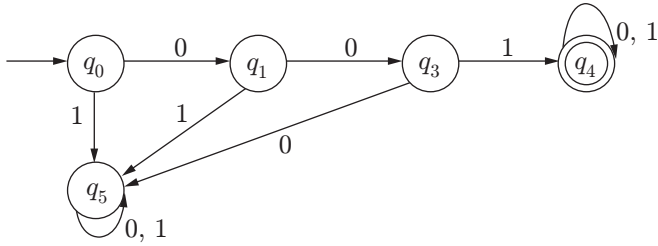


Regular expression:

- 1. $a(a + b)^* \cap (a + b)^*b$ (from statement)
- 2. $aa^*b(b + aa^*b)^*$

Example 5.14

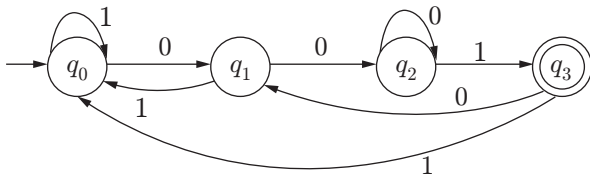
DFA which accepts string starting with “001”



Regular expression: $001(0 + 1)^*$

Example 5.15

DFA which accepts string ending with “001”



Regular expression: $(0 + 1)^*001$

5.3.5 Applications of DFA/NFA

NFA and DFA have various applications as given below:

1. Lexical analyser
2. Text editor
3. Spell checker
4. Sequential circuit design
5. Unix graph (pattern search)

5.3.6 Regular Expression and Grammar

Finite automata accepts regular grammar. First of all, regular expressions are discussed and then how to construct regular grammar from finite automaton is described (Table 5.3).

1. Regular expression: A language is regular iff \exists a regular expression r , that is, $L = L(r)$.

- There can be more than one regular expression for a language L .
- There can be more than one machine for a language L .
- A language L can be produced by more than one grammar.

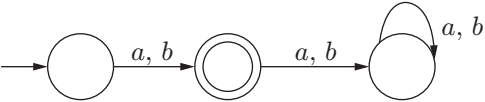
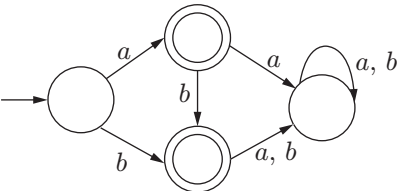
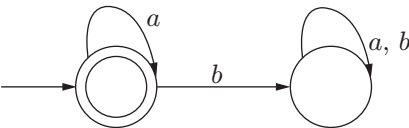
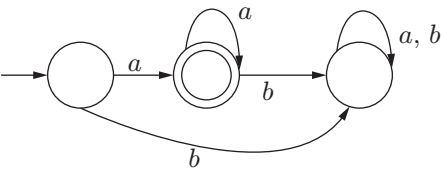
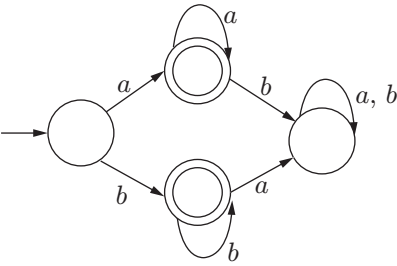
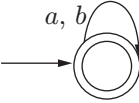
Symbols to represent regular expression are $\{a, b, \varepsilon, \Sigma, \emptyset, \lambda, +, ., *, ()\}$.

Table 5.3 | Some standard regular languages and their regular expression, grammar and machine

Regular Expression (r)	Language (L)	Grammar (G)	Machine (M)
\emptyset	$\{\}$	$S \rightarrow A$	
Λ	$\{\lambda\}$	$S \rightarrow \varepsilon$	
A	$\{a\}$	$S \rightarrow a$	
B	$\{b\}$	$S \rightarrow b$	

(Continued)

Table 5.3 | Continued

Regular Expression (r)	Language (L)	Grammar (G)	Machine (M)
$a + b$	$\{a, b\}$	$S \rightarrow a$ $S \rightarrow b$	
$a + b + ab$	$\{a, b, ab\}$	$S \rightarrow aab$ $S \rightarrow ab$ $S \rightarrow ba$	
a^*	$\{\lambda, a, aa, aaa, aaaa, \dots\}$	$S \rightarrow aS$ $S \rightarrow \epsilon$	
a^+	$\{a, aa, aaa, aaaa, \dots\}$	$S \rightarrow aS$ $S \rightarrow a$	
$a^* + b^*$	$\{\lambda, a, aa, aaa, \dots, b, bb, bbb, \dots\}$	$S \rightarrow S_1/S_2$ $S_1 \rightarrow aS_1/\epsilon$ $S_2 \rightarrow bS_2/\epsilon$	
$(a + b)^*$	$\{\lambda, a, b, ab, ba, aab, aaab, baa, bab, bbb, baba, \dots\}$	$S \rightarrow aS/bS$ $S \rightarrow a/b$	

2. Regular grammar: Type 3 grammar is also called regular grammar and it generates regular language. Regular grammar is divided into two types:

Left linear grammar: $V \rightarrow VT^* + T^*$

Right linear grammar: $V \rightarrow T^*V + T^*$

A grammar is a regular grammar iff \exists a left linear or right linear grammar for it.

5.3.7 Pumping Lemma

Pumping lemma is very useful to prove that a certain language is not a regular language. We know that the loop in finite automata (FA) makes it able to accept those strings which have length equal to and greater than its total number of states.

Let there be a string w which has a bigger length (more than its states), then we can break this string into three parts x, y (y should not be null) and z . Let FA has loop for y and $w = xyz \in L$ accepted by FA, so $w = xy^iz$ for $i = 0, 1, \dots$ is also accepted by FA.

5.3.8 Myhill–Nerode Theorem

Myhill–Nerode theorem tells that the given language might be regular or not regular. A given language L is a regular language if and only if the set of equivalence classes of L is finite, or it satisfies following three conditions:

1. The language L divides the set of all possible strings into mutually separate classes.
2. If L is a regular language, then the number of classes created is finite.
3. If the number of classes that L creates is finite, then L is a regular language.

Problem 5.1: Consider a language L consists all strings over $\{a, b\}$ ending in b . Prove that L is a regular language.

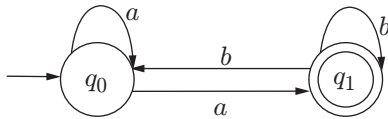
Solution: Let DFA $M = \{Q, S, \delta, q_0, F\}$ recognizes $L = \{b, bb, ab, aab, abb, \dots\}$.

We have two classes C_0 and C_1 defined as follows:

$C_0 = \{\text{All strings that end in } a\}$

$C_1 = \{\text{All strings that end in } b, \text{ i.e., accepted strings } (b, ab, abb, bbb, \dots)\}$

Transition diagram for the DFA M is shown below.



Classes C_0 and C_1 correspond to states q_0 and q_1 , respectively. Applying Myhill–Nerode theorem:

1. Two classes C_0 and C_1 are mutually exclusive.
2. L is regular and creates finite classes (here two classes).
3. The number of classes created by DFA is finite.

As all the three statements of Myhill–Nerode theorem are satisfied, hence L is regular.

5.3.9 Transducer or Finite State Machine

A finite state machine (FSM) is similar to FA except that it has the additional capability of producing an output.

FSM = FA + Output capability

There are two types of FSMs:

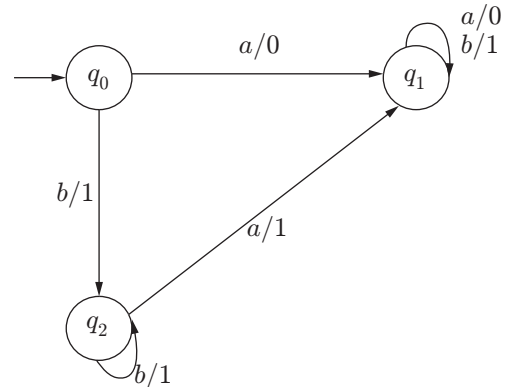
1. Mealy machines
2. Moore machines

5.3.9.1 Mealy Machines

In a Mealy machine, the output of an FSM is dependent on the present state and present input. A Mealy machine can be described by a six-tuple set $(Q, \Sigma, \Delta, \delta, \lambda, S)$, where

1. Q is a finite and non-empty set of states.
2. Σ is an input alphabet.
3. Δ is an output alphabet.
4. δ is a transition function which maps the present state and input symbol on to the next state $Q \times \Sigma \rightarrow Q$.
5. λ is the output function which maps $Q \times \Sigma \rightarrow \Delta$ ((present state + present symbol) \rightarrow output).
6. $S \in Q$ is the starting state or initial state.

Problem 5.2: Consider the Mealy machine shown in the figure. Construct the transition table and find the output for input $aabba$.



Solution: Transition table for the above Mealy machine is:

Present State	Next State		Output	
	$x = a$	$x = b$	$x = a$	$x = b$
q_0	q_1	q_2	0	1
q_1	q_1	q_1	0	1
q_2	q_1	q_2	1	1

Input: $aabba$

Output: 00110

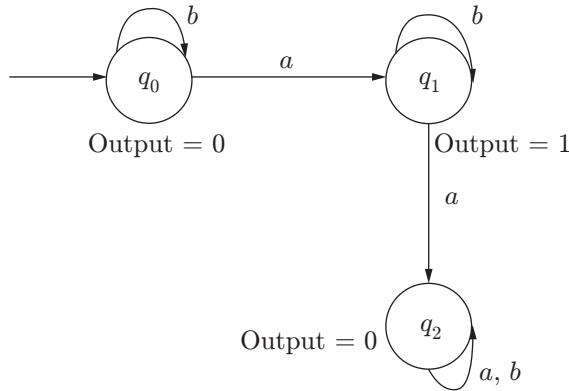
(**Note:** Output length of the Mealy machine is equal to the input length.)

5.3.9.2 Moore Machines

In Moore machine, the output of an FSM is dependent on the present state only. A Moore machine can be described by a six-tuple set $(Q, \Sigma, \Delta, \delta, \lambda, S)$, where

1. Q is a finite and non-empty set of states.
2. Σ is an input alphabet.
3. Δ is an output alphabet.
4. δ is a transition function which maps the present state and input symbol on to the next state $Q \times \Sigma \rightarrow Q$.
5. λ is the output function which maps $Q \rightarrow \Delta$ (present state \rightarrow output)
6. $S \in Q$ is the starting state or initial state.

Problem 5.3: Consider the Moore machine shown in the figure. Construct the transition table and find the output for input $aabba$.



Solution: Transition table for the above Mealy machine:

Present State	Next State		Output
	$x = a$	$x = b$	
q_0	q_1	q_0	0
q_1	q_2	q_1	1
q_2	q_2	q_2	0

Input: $aabba$

Output: 010000

(**Note:** Output length of Moore machine is one greater than the input length because the first output symbol is additional without reading any symbol from the input.)

5.3.10 Conversion in Finite Automata

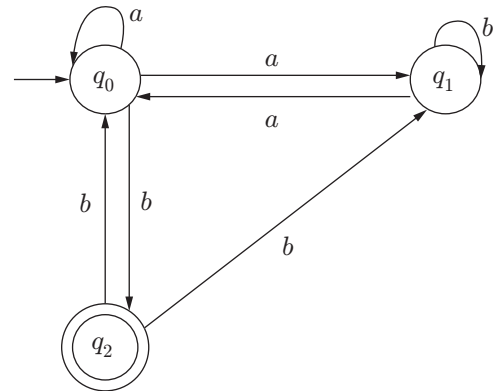
Power of NFA and DFA is the same, which means both can accept the same set of strings. Here we discuss interconversion of NFA and DFA.

5.3.10.1 NFA to DFA

To convert NFA to DFA, following points should be considered:

1. In NFA and DFA there is strictly one initial or starting state.
2. If there are n states in NFA, then equivalent DFA contains $\leq 2^n$ states.
3. DFA can simulate the behaviour of NFA by increasing the number of states.

Problem 5.4: Construct a DFA equivalent of given NFA.



Solution:

Step 1: Construct a transition table of the given NFA

δ	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	q_2
q_1	q_0	q_1
q_2	–	$\{q_0, q_1\}$

q_0 – Starting state

q_2 – Final state (accepting state)

Step 2: Construct a DFA transition table with the help of an NFA transition table. Put the first row same from the NFA transition table, then put one by one states from the right side. After the first row, put $\{q_0, q_1\}$ in the transition column and find the next state on a from q_0 and q_1 separately and write them under column a , repeat the same process for b . Now pick another state q_2 from the right side and follow the same process. This process has to be repeated until all the unique value set comes as a state under the transition table.

Now make the starting state of DFA the same state as in NFA. And make all those states final which

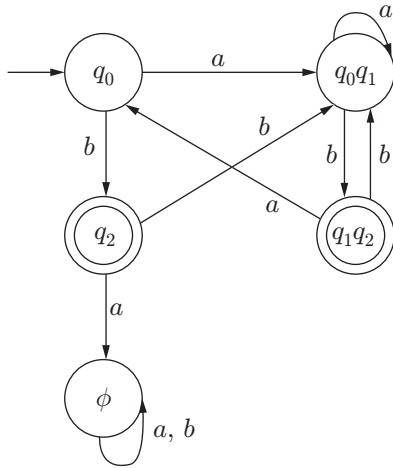
include the final state of NFA; in our example it was q_2 , so q_1q_2 and q_2 states will be final states in DFA.

δ	a	b
$\rightarrow q_0$	$\{q_0q_1\}$	q_2
$\{q_0q_1\}$	$\{q_0q_1\}$	$\{q_1q_2\}$
(q_2)	ϕ	$\{q_0q_1\}$
$(\{q_1q_2\})$	q_0	$\{q_0q_1\}$
ϕ	ϕ	ϕ

\rightarrow Represent starting state

\bigcirc Represent final or accepting state

Step 3: Draw DFA from the transition table.

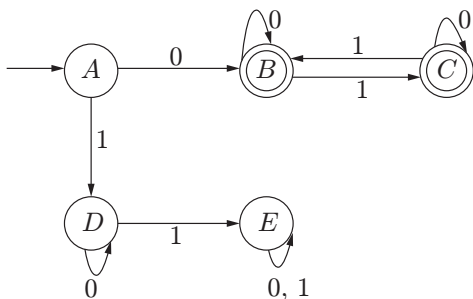


5.3.10.2 DFA to Minimum DFA

Minimization of DFA means reducing the unnecessary states. During minimization, the following steps are considered.

1. DFA with n states when converted into minimal DFA will contain N states, where $1 \leq N \leq n$.
2. We have to eliminate the same states from the given DFA to convert it into minimal DFA.
3. Two states $q_1 \equiv q_2$ iff $\forall (w \in \Sigma^*)$, $\{\delta(q_1, w), \delta(q_2, w)\}$ both are from the same set.
4. If a final state (accepting state) is equivalent to a non-final state, then it cannot be reduced.

Problem 5.5: Convert the given DFA into minimal DFA.



Solution:

Step 1: Make two sets of final and non-final states.

Iteration 1 = ($\{A, D, E\}$ $\{B, C\}$)

Step 2: Check within the set if two of them are equivalent states or not.

Let us first check in $\{A, D, E\}$, first we pick A and D .

Present State	Next State on Input	
	$X = 0$	$X = 1$
A	B	D
D	D	E
E	E	E

We can clearly see that state A is going outside on state B at input 0, which is in another set, whereas the other two states are indicating within the non-final state set. So, we will put state A in a new set.

Iteration 2 = ($\{A\}$ $\{D, E\}$ $\{B, C\}$)

Repeat step 2 until last two iterations become same.

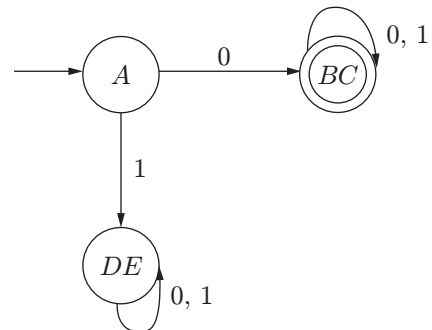
Now check for B and C .

Present State	Next State on Input	
	$X = 0$	$X = 1$
B	B	C
C	C	B

States B and C are not going outside from their own set at inputs 0 and 1, so there will be no change.

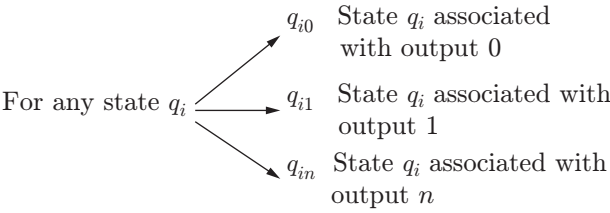
Iteration 3 = ($\{A\}$ $\{D, E\}$ $\{B, C\}$)

We get two same iterations so we will stop this process. We will merge those states which lie in a set. There are three states in minimal DFA (A), (DE) and (BC). Now we can draw the minimal DFA.



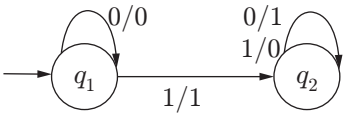
5.3.10.3 Mealy to Moore Machine

In a Mealy machine, the output depends upon both the present state and the input, and the Moore machine output depends only on the present state. While converting Mealy to Moore we develop a procedure so that all the states of Mealy machines, which are associated with different outputs, find them. After that, split those states into n parts if the states have n outputs.



Through this procedure all the states are associated with a single output, so all the states are considered as unique state. We can understand this concept clearly with an example.

Problem 5.6: Consider the 2's complement Mealy machine given below and convert it into Moore machine. At input "1101", the output is "0011". Input string will be read from the right side.



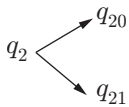
Solution: First we will make a transition table for the given Mealy machine.

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
q_1	q_1	q_2	0	1
q_2	q_2	q_1	1	0

In the transition table, we have to find those states which are associated with more than one output in the next state column.

For q_1 at $(q_1, 0)$, we have output 0.

For state q_2 at $(q_2, 0)$ output is 1 and at $(q_2, 1)$ output is 0. So there is a need to split q_2 .

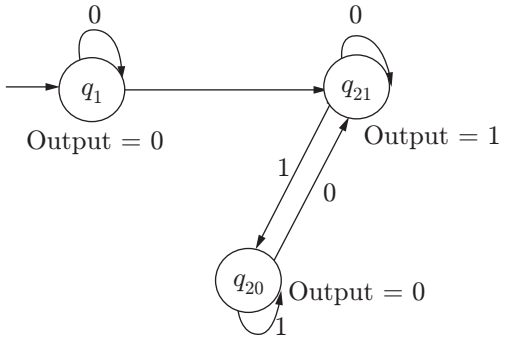


Now the Moore machine has three states $\{q_1, q_{20}, q_{21}\}$ with output $\{0, 0, 1\}$, respectively.

Now we will make a transition table of the Moore machine.

Present State	Next State		Output
	$x = 0$	$x = 1$	
$\rightarrow q_1$	q_1	q_{21}	0
q_{20}	q_{21}	q_{20}	0
q_{21}	q_{21}	q_{20}	1

Moore machine of 2's complement:



Note: m states, n outputs Mealy machine \equiv Moore machine contains $\leq mn + 1$ state.

5.3.10.4 Moore to Mealy Machine

Consider the Moore machine given in Fig. 5.3 and convert it into Mealy machine.

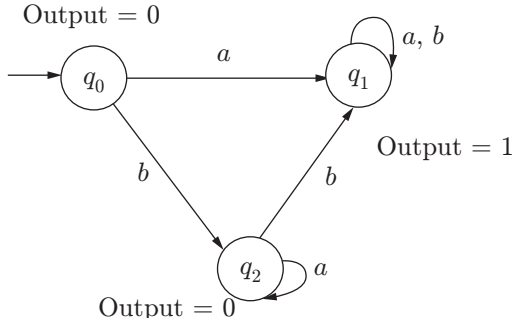


Figure 5.3 | Moore machine.

In the transition table of Moore machine (Table 5.4), the states (q_0, q_1, q_2) have outputs $(0, 1, 0)$, respectively.

Table 5.4 | Transition table of Moore machine

Present State	Next State		Output
	$x = a$	$x = b$	
$\rightarrow q_0$	q_1	q_2	0
q_1	q_1	q_1	1
q_2	q_2	q_1	0

So, now we can make a transition table for the Mealy machine (Table 5.5).

Table 5.5 | Transition table of Mealy machine

Present State	Next State		Output	
	$x = a$	$x = b$	$x = a$	$x = b$
$\rightarrow q_0$	q_1	q_2	1	0
q_1	q_1	q_1	1	1
q_2	q_2	q_1	0	1

Mealy machine equivalent to given Moore machine (Fig. 5.4):

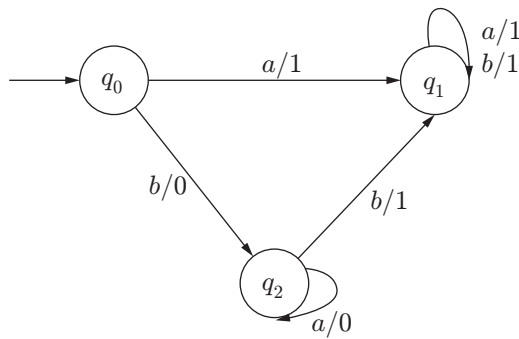


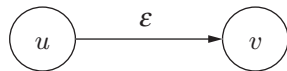
Figure 5.4 | Mealy machine.

Note: m states, n outputs Moore machine \equiv Mealy machine contains $\leq m$ states.

5.3.10.5 NFA with ϵ -Moves to NFA Without ϵ -Moves

There are three steps to convert NFA with ϵ moves to NFA without ϵ moves:

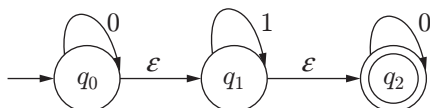
1. If there is an ϵ move from u to v , then every arrow starting from v is also starting from u .



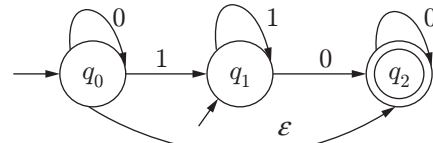
2. When ϵ move removes from u to v , and if u is a starting state, then make v also a starting state.
3. When ϵ move removes from u to v , and if v is a final state, then make u also a final state.

Problem 5.7: Convert given NFA with ϵ moves to NFA without ϵ moves.

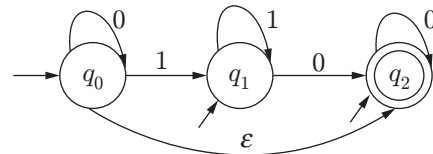
Solution:



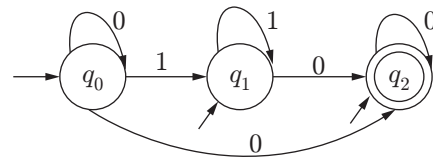
Step 1: Follow above steps to remove first ϵ move between q_0 and q_1 .



Step 2: Follow above steps to remove ϵ move between q_1 and q_2 .



Step 3: Follow above steps to remove ϵ move between q_0 and q_2 .



After removing all ϵ moves, this is the final NFA.

Note: If NFA with ϵ -moves has n states then NFA without ϵ -moves will have exactly n states.

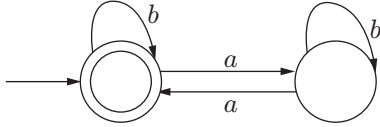
5.3.11 Properties of Regular Sets/Languages

1. Classes of two regular sets S_1 and S_2 are closed under union operation, means $S_1 \cup S_2$ is also a regular set.
2. Classes of two regular sets S_1 and S_2 are closed under concatenation operation, means $S_1 S_2$ is also a regular set.
3. A class of regular set S_1 is closed under Kleene closure operation, means S_1^* is also a regular set.
4. If S is a regular set on some alphabet Σ , then complement of S is denoted by \bar{S} is also a regular set. Steps to make complement of an NFA are as follows:
 - Change all final states to non-final states of NFA for Σ .
 - Change all non-final states to final states of NFA for Σ .
5. Classes of two regular sets S_1 and S_2 are closed under intersection operation, means $S_1 \cap S_2$ is also a regular set.
6. If S is a regular set on some alphabet Σ , then transpose of S is denoted by S^R is also a regular set.

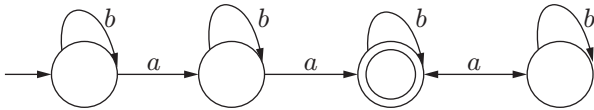
5.3.12 Some Important DFA

Some examples of important DFAs are given below:

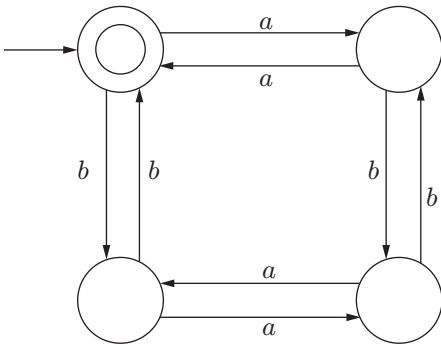
1. Containing even number of a 's. This can be written as $L = \{w \mid na(w) \bmod 2 = 0\}$:



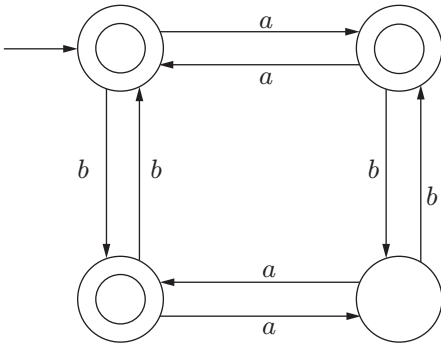
2. DFA having exactly two a 's:



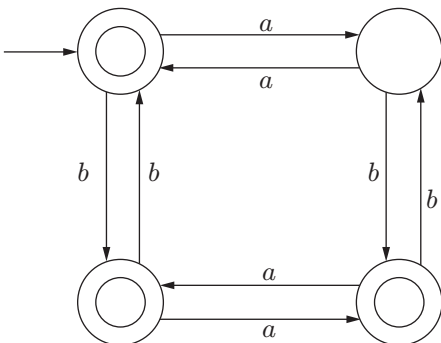
3. $L = \{w \mid na(w) \bmod 2 = 0 \text{ and } nb(w) \bmod 2 = 0\}$:



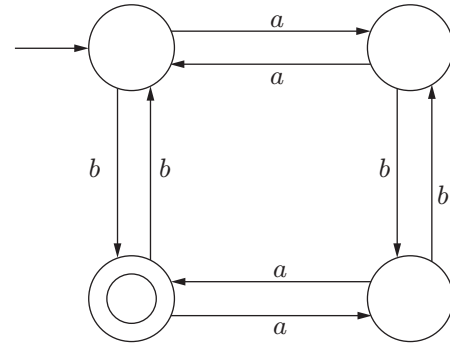
4. $L = \{w \mid na(w) \bmod 2 = 0 \text{ or } nb(w) \bmod 2 = 0\}$:



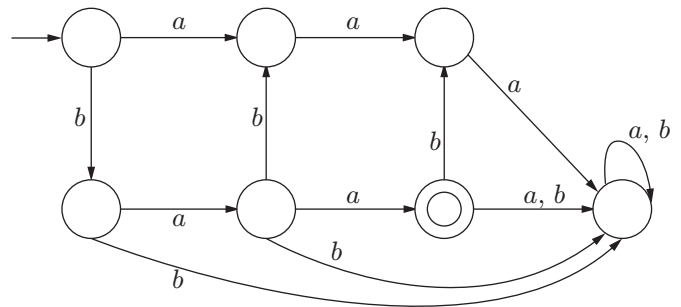
5. $L = \{w \mid na(w) \bmod 2 = 0 \text{ or } nb(w) \bmod 2 = 1\}$:



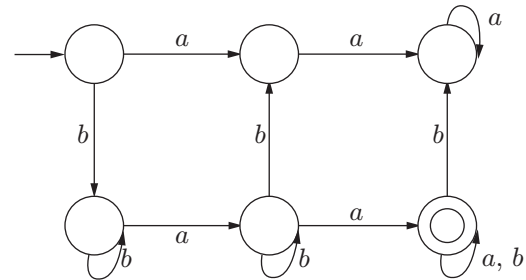
6. $L = \{w \mid na(w) \bmod 2 = 0 \text{ and } nb(w) \bmod 2 = 1\}$:



7. DFA which accepts exactly two a 's and one b :



8. DFA which accepts at least two a 's and at least one " b ":



5.4 PUSHDOWN AUTOMATA

Pushdown automata (PDA) are devices that recognize context-free languages. PDA has finite memory in terms of a stack. A pushdown automaton is a non-deterministic device. The deterministic version of automata accepts only a subset of CFL known as deterministic context-free language (DCFL).

5.4.1 Model of PDA

PDA can be defined as a seven-tuple set $\{Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\}$, where

1. Q is a finite and non-empty set of states.
2. Σ is a finite and non-empty set of input symbol.

3. Γ is a finite non-empty set of stack alphabets.
4. δ is the transition function which maps $Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow Q \times \Gamma^*$
5. q_0 is the initial state (start state).
6. $Z_0 \in \Gamma$ is the starting (top most) stack symbol.
7. F is the set of final states and $F \subseteq Q$.

Figure 5.5 shows a model of PDA.

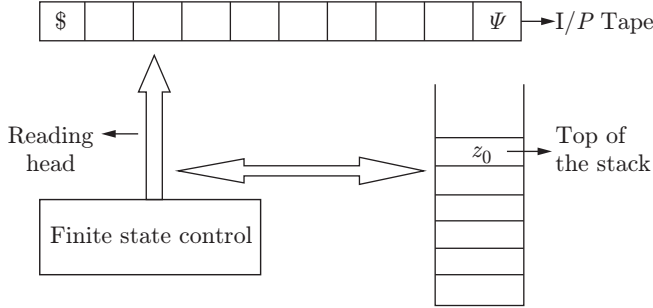


Figure 5.5 | Model of PDA.

5.4.2 Transition Function

Three types of operations are performed under the transition function (δ) of PDA.

1. **Push element into stack:** Let the current stack symbol be z_0 , current state be q_0 and current input symbol be “ a ”. So after reading “ a ”, z_2 is to be pushed into stack and next state may or may not be changed according to the transition function written as

$\delta(\text{current state, current input symbol, top of stack symbol}) \rightarrow (\text{next state, new top of stack})$

$$\delta(q_0, a, z_0) = (q_1, az_0)$$

2. **Pop element from stack:** If “ a ” is input symbol on state q_0 with stack top z_1 and z_1 is popped up after processing “ a ” and next state is q_1 . Then transition function δ is written as

$\delta(\text{current state, current input symbol, top of stack symbol}) \rightarrow (\text{next state, top of stack})$

$$\delta(q_0, a, z_1) = (q_1, \lambda)$$

3. **No change in state element:** If on an input symbol “ a ” at state q_0 on stack top z_1 nothing is pushed or popped, then transition function can be written as

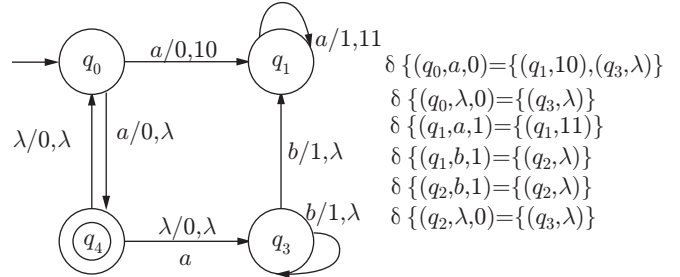
$$\delta(q_0, a, z_1) = (q_1, z_1)$$

5.4.3 Non-deterministic PDA

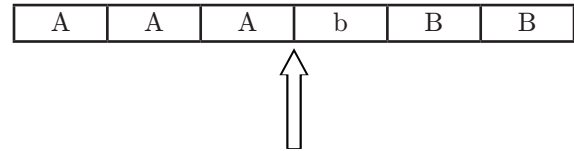
Non-deterministic pushdown automata (NPDA) have more than one choice for a single input such as NFA. Transition function of NPDA is $Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow Q \times \Gamma^*$. An NPDA accepts an input string if a sequence

leads to some final state of PDA or cause PDA to empty its stack. A non-deterministic automaton is equivalent to a CFG and more powerful than a deterministic PDA.

Example 5.16



This NPDA will recognize the string “ $aaabbb$ ” by the following moves:



Input head start from left and starting stack symbol is 0.

1. $(q_0, a, 0) \vdash (q_1, 10)$
2. $(q_1, a, 10) \vdash (q_1, 110)$
3. $(q_1, a, 110) \vdash (q_1, 1110)$
4. $(q_1, b, 1110) \vdash (q_2, 110)$
5. $(q_2, b, 110) \vdash (q_2, 10)$
6. $(q_2, b, 10) \vdash (q_2, 0)$
7. $(q_2, \lambda, 0) \vdash (q_3, \lambda)$

As $q_3 \in F$, the string is accepted.

5.4.4 Deterministic PDA

A deterministic pushdown automata (DPDA) is one for which every input string has a unique way through the machine. A PDA $M = \{Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\}$ is deterministic if it satisfies the following two conditions:

1. $\delta(q, a, z)$ is either empty or only single move.
2. $\delta(q, \lambda, z) \neq \emptyset$ means $\delta(q, a, z) = \emptyset$ for each $a \in \Sigma$.

The language accepted by DPDA is known as deterministic context-free language (DCFL), and it is noticeable that not all CFLs are DCFLs.

Example 5.17

Language $L = \{a^n \times b^n : n \geq 0\}$, it can be accepted by the DPDA because all ‘ a ’ characters are inserted into stack till character ‘ x ’ come and then symbol ‘ a ’ is removed as many times as many symbol ‘ b ’ occur. In the last stack will be empty.

5.4.5 Parsing

Parsing is a technique to construct a parse or a derivation tree. It is to check whether there is some leftmost derivation for a given string using a certain grammar. Parse tree generated through parsing is also used by a syntax analyzer in a compiler for checking a string is according to a given grammar or not.

5.4.5.1 Top-Down Parsing

In leftmost derivation, we start with initial and replace the leftmost variable in each step and finally get the string. This approach is known as top-down parsing. In the parse, start symbol is root, terminals are leaves (input symbols) and other nodes are variables. We start from the root and replacing the intermediate nodes one by one from left to right reach the leaves. This approach is also known as recursive descent.

Here, we have constructed the leftmost derivation looking ahead one symbol in the given string. A grammar having this property is known as LL(1). In general, if a leftmost derivation is constructed by looking at the k symbol ahead in the given string, then the grammar is known as LL(k) grammar, where $k \geq 1$.

Steps of construction of top-down parser:

- Step 1:** Eliminate left recursion (if any) from the given grammar.
- Step 2:** Eliminate left factoring (if any) from the given grammar.
- Step 3:** If resulting grammar is LL(k) for some $k \geq 1$, then construct a parsing table.

5.4.5.2 Bottom-Up Parsing

In bottom-up parsing, we start with the input string and replace the terminals by respective variables such that these replacements lead to the starting symbol of the grammar. So every step takes input string close to the starting symbol. This approach is the reverse of top-down approach. It reads the input from left and uses the rightmost derivation in reverse order. Bottom-up parser is also known as shift-reduce (SR) parser. There are three actions in bottom-up parsing:

- Step 1:** Shift the current input (token) on the stack and read the next token.
- Step 2:** Reduce by some suitable LHS of production rule.
- Step 3:** Accept, final reduction which leads to starting symbol.

5.5 CONTEXT-FREE GRAMMAR AND LANGUAGES

A grammar $G = (V_n, T, S, P)$ is said to be a CFG if the production of G is of the form $A \rightarrow \alpha$, where $\alpha \in (V_n \cup S)^*$ and $A \in V_n$.

As we know that a CFG has no context either on left or on right, this is the reason why it is also known as context-free.

Problem 5.8: Consider a grammar $G = (V_n, T, S, P)$ having production $\{S \rightarrow aSa|bSb|x\}$. Check the production and find the language generated.

Solution: Let $P_1: S \rightarrow aSa$

$P_2: S \rightarrow bSb$

$P_3: S \rightarrow x$

(a, b, x) are terminals and S is a variable. As all the productions are of the form $A \rightarrow \alpha$, where $\alpha \in (V_n \cup S)^*$ and $A \in V_n$, hence G is a CFG, and it will produce context-free language.

Language generated: $L(G) = \{w x w^R : w \in (a + b)^*\}$

5.5.1 Context-Free Grammar

A grammar $G = (V_n, T, S, P)$ is said to be a CFG if production $P \{u \rightarrow v\}$ follows these conditions:

1. $u \rightarrow v$, where $v \in (V \cup T)^*$
2. $|u| \leq |v|$ (length of u is less than v)
3. Only single variable is allowed in the left side (means u has single variable only)

Example 5.18

$S \rightarrow aSb/\lambda$

$A \rightarrow aA/aB/Cb$

$B \rightarrow a/Da$

5.5.2 Standard Context-Free Language

There are some standard languages under CFL such as:

1. $a^n b^n, a^n b^{2n}, a^{2n} b^n, a^n b^{2n+3}, a^{2n+2} b^{3n+5}$, etc.
2. $n_a(w) = n_b(w)$ (means number of "a" equals to number of "b" in a language), $a^m b^n$, where ($m = n, m > n, m < n$, etc.)
3. Palindrome such as $[w w^R]$ (even palindrome), $w a w^R$ (odd palindrome), etc.]

4. Generate language from grammar:
 - Grammar is $S \rightarrow aSbb| \epsilon$
Language generated by the above grammar is $a^n b^{2n}$.
 - Grammar is $S \rightarrow aSb|bSa|a|b$
Language generated by the above grammar is $w x w^R$, ($x \in a, b$).
5. Make grammar for given language:
 - Language $L = \{a^{2n}b^n\}$, where $n \geq 1$
Grammar for given language: $S \rightarrow aaSb| \epsilon$
 - Language $L = \{w\#w^R : w \in (a, b)^+, \# \neq \Sigma\}$
Grammar for a given language: $S \rightarrow aSb|bSa| \#$

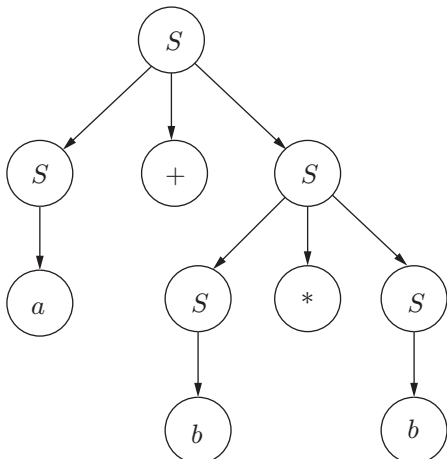
5.5.3 Derivation Tree or Parse Tree

The string generated by CFG $G = (V_n, T, S, P)$ is represented by a hierarchical structure called tree. A derivation tree or parse tree for a CFG is a tree that satisfies the following conditions:

1. If $A \rightarrow \alpha_1, \alpha_2, \alpha_3, \dots, \alpha_n$ is a production in G , then A becomes the father of nodes labelled as $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_n$.
2. The root has label S (starting symbol).
3. Every vertex (or node) has a label.
4. Internal nodes should be labelled with variables only.
5. The leaf nodes are labelled with ϵ or terminal symbols.
6. The collection of leaves from left to right yields the string w .

Problem 5.9: Consider the grammar $S \rightarrow S + S | S^* S | a | b$. Construct derivation (or parse) tree for the string $w = a + b^*b$.

Solution:



5.5.3.1 Leftmost Derivation Tree

A derivation tree is called as leftmost derivation (LMD) tree if the ordering of decomposed variable is from left to right. Thus, for generating a string $w = aab$ from grammar:

- $$\begin{aligned}
 S &\rightarrow AB \text{ (production 1)} \\
 A &\rightarrow aaA \text{ (production 2)} \\
 A &\rightarrow \epsilon \text{ (production 3)} \\
 B &\rightarrow bB \text{ (production 4)} \\
 B &\rightarrow \epsilon \text{ (production 5)}
 \end{aligned}$$

LMD:

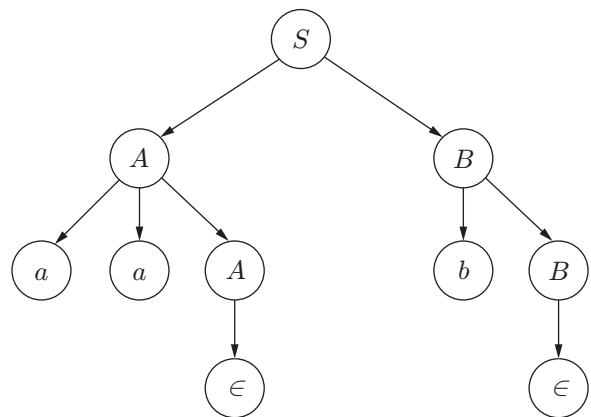
- $$\begin{aligned}
 S &\rightarrow \underline{A}B \text{ by production 1)} \\
 S &\rightarrow aa\underline{A}B \text{ (by production 2)} \\
 S &\rightarrow aa\underline{B} \text{ (by production 3)} \\
 S &\rightarrow aab\underline{B} \text{ (by production 4)} \\
 S &\rightarrow aab \text{ (by production 5)}
 \end{aligned}$$

5.5.3.2 Rightmost Derivation Tree

A derivation tree is called rightmost derivation (RMD) tree if the ordering of decomposed variable is from right to left. Thus, for generating a string $w = aab$ from the above grammar:

RMD:

- $$\begin{aligned}
 S &\rightarrow A\underline{B} \text{ (by production 1)} \\
 S &\rightarrow A\underline{b}B \text{ (by production 4)} \\
 S &\rightarrow \underline{A}b \text{ (by production 5)} \\
 S &\rightarrow aa\underline{A}b \text{ (by production 2)} \\
 S &\rightarrow aab \text{ (by production 3)}
 \end{aligned}$$

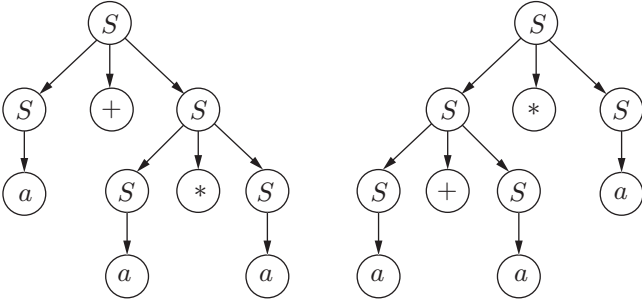


Left to right

5.5.4 Ambiguous Grammar

A grammar G is called ambiguous if for some string $w \in L(G)$, there exists two or more derivation tree (two or more LMD or two or more rightmost derivation tree). Let us consider a CFG grammar having production:

$S \rightarrow S + S | S^* S | a | b$, for string $w = a + a^* a$ have more than one LMD tree.



Note: A language (L) is called ambiguous language if and only if every grammar which generates it is ambiguous. The only known ambiguous language is $\{a^n b^m c^n\} \cup \{a^n b^m c^m\}$.

Left recursion and left factoring are the major cause for a grammar to be ambiguous. But presence of these in a grammar does not mean that grammar is ambiguous, and similarly absence of these does not mean that grammar is unambiguous.

5.5.5 Removal of Ambiguity

Ambiguities are left recursion and left factoring. In this subsection, removal of various ambiguities is discussed step by step.

5.5.5.1 Removal of Left Recursion

A production of grammar $G = (V_n, T, S, P)$ is said to be left recursive grammar if it has one of the production in a given form.

$$A \rightarrow A\alpha, \text{ where } A \text{ is a variable and } \alpha \in (V_n \cup S)^*.$$

Elimination of left recursion: Let the variable A have left recursive problem as follows:

$A \rightarrow A\alpha_1 | A\alpha_2 | A\alpha_3 | \dots | A\alpha_n | \beta_1 | \beta_2 | \beta_3 | \dots | \beta_m$, where $\beta_1, \beta_2, \beta_3, \dots, \beta_m$ do not begin with A , then we replace A production by:

$$\{A \rightarrow \beta_1 A^1 | \beta_2 A^1 | \beta_3 A^1 | \dots | \beta_n A^1\},$$

where $A^1 \rightarrow \alpha_1 A^1 | \alpha_2 A^1 | \alpha_3 A^1 | \dots | \alpha_n A^1 | \epsilon$

Problem 5.10: Let grammar $S \rightarrow S + S | S^* S | a | b | c$

Solution: After removing left factoring, the productions are replaced by

$$S^1 \rightarrow + SS^1 | * SS^1 | \epsilon$$

$$S \rightarrow aS^1 | bS^1 | cS^1$$

5.5.5.2 Removal of Left Factoring

In grammar G , two or more production of variable A are said to have left factoring if the production are in the following form:

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \dots | \alpha\beta_m$$

where $\{\beta_1 | \beta_2 | \beta_3 | \dots | \beta_m\} \in (V_n \cup S)^*$ and does not start with α . All these production have common left factor α .

Elimination of left factoring: Let variable A have (left factoring) production as follows:

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \dots | \alpha\beta_m | \gamma_1 | \gamma_2 | \gamma_3 | \dots | \gamma_m$, where $\gamma_1, \gamma_2, \gamma_3, \dots, \gamma_m$ and $\beta_1, \beta_2, \beta_3, \dots, \beta_m$ do not contain α as a prefix, then we replace this production by

$$A \rightarrow \alpha A^1 | \gamma_1 | \gamma_2 | \gamma_3 | \dots | \gamma_m$$

$$A \rightarrow \beta_1 | \beta_2 | \beta_3 | \dots | \beta_m$$

Problem 5.11: Let grammar $A \rightarrow abc | abd | abe$ remove left factoring.

Solution: After removing left factoring, the productions are replaced by

$$A \rightarrow abA^1$$

$$A \rightarrow c | d | e$$

5.5.6 Context-Free Grammar Simplification

For simplification of context-free language, it is necessary to eliminate variable, terminal and production having the following properties:

1. Unit production ($A \rightarrow B$)
2. Null production ($A \rightarrow \lambda$)
3. Useless production, a variable which does not occur even in a single derivation

5.5.6.1 Removal of Unit Production

A production (P) in CFG is said to be a unit production if it is in the following form:

$A \rightarrow B$ where $A, B \in V$ (variable). There are three steps to remove a unit production.

Step 1: Find $\hat{P} = P - (\text{unit production})$

Step 2: Find unit production and draw a unit production dependency graph.

Step 3: Add new rules to the grammar.

Problem 5.12: Remove the unit production from the given CFG grammar.

$$S \rightarrow Aa|B$$

$$B \rightarrow A|bb$$

$$A \rightarrow a|bc|B$$

Solution:

Step 1: Find a non-unit production (\hat{P}) = $P - (\text{unit production})$

$$S \rightarrow Aa$$

$$B \rightarrow bb$$

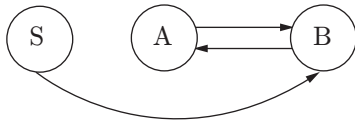
$$A \rightarrow a|bc$$

Step 2: Find a unit production and draw a unit production dependency graph.

$$S \rightarrow B$$

$$B \rightarrow A$$

$$A \rightarrow B$$



Step 3: Add new rules to the grammar. Check that from each variable through other variables on which terminals we can reach.

$$S \rightarrow a|bc|bb$$

$$A \rightarrow bb$$

$$B \rightarrow a|bc$$

Final grammar after removing a unit production is as follows:

$$S \rightarrow Aa|a|bc|bb$$

$$A \rightarrow a|bc|bb$$

$$B \rightarrow bb|a|bc$$

Note: Removal of a unit production has made B and the associated production useless.

5.5.6.2 Removal of Null Production

$A \rightarrow \lambda$ is a null production and A is a nullable variable; λ should be removed because it creates difficulties for the compilers. There are two steps to remove a null production.

Step 1: Identify the nullable variable.

Step 2: Find \hat{P} production without null.

Problem 5.13: Remove null (λ) production from the given CFG grammar.

$$S \rightarrow ABaC$$

$$A \rightarrow BC$$

$$B \rightarrow b|\lambda$$

$$C \rightarrow D|\lambda$$

$$D \rightarrow d$$

Solution:

Step 1: Identify nullable variables.

Variables B and C directly produce null $N = \{B, C\}$. But variable A is related to B and C , so A will also produce a null production.

$$N = \{A, B, C\}$$

Step 2: Find \hat{P} production without null.

$$S \rightarrow ABaC$$

$$A \rightarrow BC$$

$$B \rightarrow b$$

$$C \rightarrow D$$

$$D \rightarrow d$$

Generate grammar without null production (G^1) by putting null value to nullable variables one by one in production without null. For example, put $A = \lambda$ in $S \rightarrow ABaC$, we will get $S \rightarrow BaC$. Do this process for all nullable variables.

$$S \rightarrow ABaC|BaC|AaC|ABa|aC|aB|Aa|a$$

$$A \rightarrow BC|B|C$$

$$B \rightarrow b$$

$$C \rightarrow D$$

$$D \rightarrow d$$

5.5.6.3 Removal of Useless Production

Let $G = (V_n, T, S, P)$ be a CFG. A variable $A \in V$ is said to be useless if there is not even a single derivation which uses that production.

Example 5.19

$$S \rightarrow A$$

$$A \rightarrow aA|a$$

$$B \rightarrow bA$$

The variable B is useless and so is the production $B \rightarrow bA$. Although b can drive a terminal string, there is no way to reach on B from the starting variable S .

Problem 5.14: Consider a given grammar and eliminate useless production.

$$\begin{aligned} S &\rightarrow aS|A|C \\ A &\rightarrow a \\ B &\rightarrow aa \\ C &\rightarrow aCb \end{aligned}$$

Solution:

Step 1: First find out those variables which lead to terminal strings. Because $A \rightarrow a$ and $B \rightarrow aa$, variables A and B directly come under this set. But due to the dependency on A , variable S also generates terminal string $S \rightarrow A \rightarrow a$. So,

$$\begin{aligned} S &\rightarrow aS|A \\ A &\rightarrow a \\ B &\rightarrow aa \end{aligned}$$

Step 2: Now we have to find those variables which cannot be reached from the starting variable. B is the only variable which cannot be reached from S . So B is a useless variable. Grammar after eliminating useless production and variables is as follows:

$$\begin{aligned} S &\rightarrow aS|A \\ A &\rightarrow a \end{aligned}$$

5.5.7 Chomsky Normal Form

Chomsky normal form (CNF) has restriction on the length of the right-hand side of a production, either two variables or single terminal is allowed in the right side of a production such as $S \rightarrow AB$, $S \rightarrow a$.

Problem 5.15: Convert the given grammar into CNF form:

$$\begin{aligned} S &\rightarrow aAD \\ A &\rightarrow aB|bAB \\ B &\rightarrow b \\ D &\rightarrow d \end{aligned}$$

Solution:

$$\begin{aligned} S &\rightarrow aAD \text{ or } \{S \rightarrow XD, X \rightarrow YA, Y \rightarrow a\} \\ A &\rightarrow aB \text{ or } \{A \rightarrow ZB, Z \rightarrow a\} \end{aligned}$$

$$A \rightarrow bAB \text{ or } \{A \rightarrow KB, K \rightarrow LA, L \rightarrow b\}$$

This grammar can be written as in CNF G^1 as

$$\begin{aligned} V_n &= \{S, A, B, D, K, L, X, Y, Z\} \\ \Sigma &= \{a, b, d\} \\ P^1 &= \{S \rightarrow XD, X \rightarrow YA, Y \rightarrow a, A \rightarrow ZB, Z \rightarrow a, \\ &\quad A \rightarrow KB, K \rightarrow LA, L \rightarrow b\} \end{aligned}$$

5.5.8 Greibach Normal Form

A CFG is called in Greibach normal form (GNF) if all productions have the form $\{A \rightarrow ax \text{ or } A \rightarrow a\}$, where $a \in T$ and $x \in V^*$. Means RHS of production are either a terminal or a terminal followed by variables.

Problem 5.16: Convert the given grammar into GNF

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA|bB|b \\ B &\rightarrow b \end{aligned}$$

Solutions: The GNF of the given grammar is

$$\begin{aligned} S &\rightarrow aAB|bBB|bB \\ A &\rightarrow aA|bB|b \\ B &\rightarrow b \end{aligned}$$

5.5.9 Identification of Language

1. If language is finite then it will surely be a regular language. Example: Let $L = \{a^m b^n, m + n = 10\}$ as m and n are finite, so it is a regular language. But it does not mean that if a language is infinite then it cannot be regular such as $(a^* b^*)$ is an infinite and regular language.
2. If there is a comparison between m and n then it will be CFL. Example: Let $L = \{a^m b^n, m \geq n\}$.
3. Let language $L = \{a^m b^n\}$, and if m or n is non-linear then it will not be accepted by PDA, so it will fall in the category of CSL.
4. If there are more than one comparison then it will be CSL. Example: Language $L = \{a^m b^n c^o, m \geq n \text{ and } o \geq n\}$.
5. All modular machines are regular language.
6. All palindrome languages are CFL language.

5.6 TURING MACHINE

In 1936, an English mathematician Alan Turing suggested the concept of Turing machine (TM). This machine can simulate the behaviour of a general purpose

computer. A TM is a generalization of a PDA, which uses a tape instead of tape and stack. The length of the tape is assumed to be infinite. A tape is divided into cells, and one cell can hold one input symbol. The head of the TM is capable to read and write on the tape and can move left or right or remain static.

TMs accept the languages defined by type 1 grammar, and these languages are called recursively enumerable or type 1 languages.

5.6.1 Model of a Turing Machine

A TM consists of a tape which is finite at the left end and infinite at the right end and has a finite control and read/write head (Fig. 5.6).

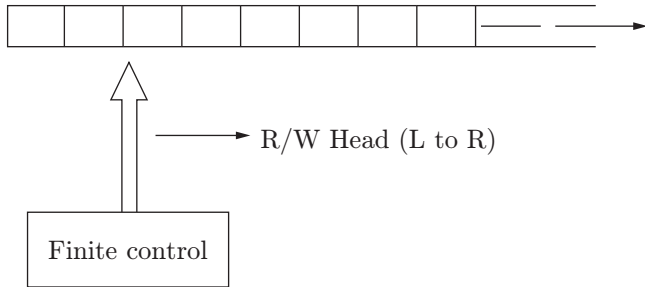


Figure 5.6 | Model of a turing machine.

5.6.1.1 Mathematical Description of a Turing Machine

A TM can be described by a seven-tuple set $(Q, \Sigma, \Gamma, \delta, q_0, \#, F)$, where

1. Q is the finite set of states (not including the halt state (h)).
2. Σ is the input alphabet which is a subset of the tape alphabet (not including the blank symbol $\#$).
3. Γ is the finite set of symbols called the tape alphabet.
4. δ is the transition function which maps from $Q \times \Gamma \rightarrow Q \times \Gamma \times [L, R]$.
5. $\# \in \Gamma$ is a special symbol called blank.
6. $q_0 \in Q$ is the initial state.
7. $F \subseteq Q$ is the set of final states.

A TM can be deterministic or non-deterministic depending on the number of moves in a transition. If a TM has at the most one move in a transition, then it is a deterministic Turing machine (DTM). If there is more than one move, then it is a non-deterministic Turing machine (NTM). A standard TM reads one cell of the tape and changes its state (optional) and moves left or right by one cell.

The transition function (δ) is defined as $\delta(p, a) = (q, b, D)$, where p is the present state, q is the next state, $a, b \in \Gamma$ and D is the movement $\{(left (L) \text{ or } right (R))\}$.

After reading an input $a \in \Sigma$, TM does one of the following:

1. Replaces a by b and moves right (R) as $\delta(p, a) = (q, b, R)$.
2. Without write, anything moves right (R) as $\delta(p, a) = (q, a, R)$.
3. Replace a by b and move left (L) as $\delta(p, a) = (q, b, L)$.
4. Without write, anything moves left (L) as $\delta(p, a) = (q, a, L)$.

The change of state in the above transition is optional (means may or may not change).

5.6.1.2 Language Reorganization by a Turing Machine

A TM can be used as a language recognizer. It recognizes all languages (regular, CFL, CSL, type 0).

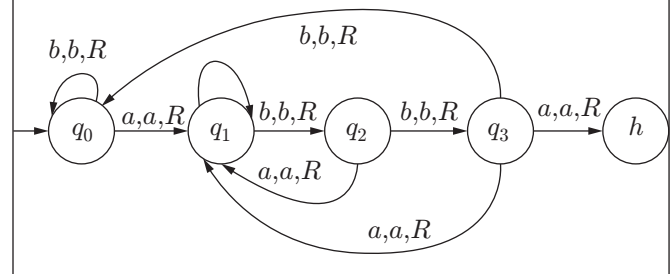
Let w be a string to be written on tap. The Turing machine is started from initial state q_0 , read write head is placed on the left most symbol of w . with the sequence of moves, if TM enters to a final state and halts then string w is said to be accepted by Turing machine.

Problem 5.17: Design a TM for $L = \{w: w \in (a + b)^* \text{ ending in substring } abb\}$

Solution: Let TM $M = \{(q_0, q_1, q_2, q_3, h), (a, b), (a, b, \#), \delta, q_0, \#, h\}$ accepts language L , where δ is defined as follows:

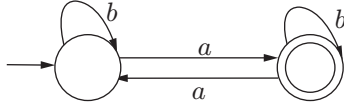
$$\begin{aligned} \delta(q_0, b) &= (q_0, b, R) \\ \delta(q_0, a) &= (q_1, a, R) \\ \delta(q_1, a) &= (q_1, a, R) \\ \delta(q_1, b) &= (q_2, b, R) \\ \delta(q_2, a) &= (q_1, a, R) \\ \delta(q_2, b) &= (q_3, b, R) \\ \delta(q_3, a) &= (q_1, a, R) \\ \delta(q_3, b) &= (q_0, b, R) \\ \delta(q_3, \#) &= (h, \#, S) \end{aligned}$$

The transition diagram is shown in the following figure.

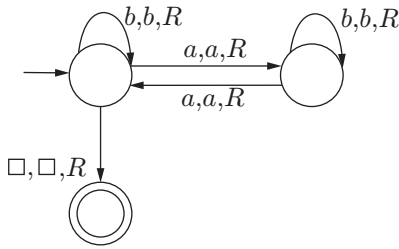


5.6.2 Designing a Turing Machine

Let us consider a problem for which we will design a TM. Suppose we have to design a TM which will halt on a final state if a string has an even number of “a”s’. So, first we will design a DFA which accepts a string having an even number of “a”s’, and after that we will design a TM for the same problem.



This regular machine will accept only those strings which have an even number of a’s. Now we will make a final state as non-final and add one final state on blank input.

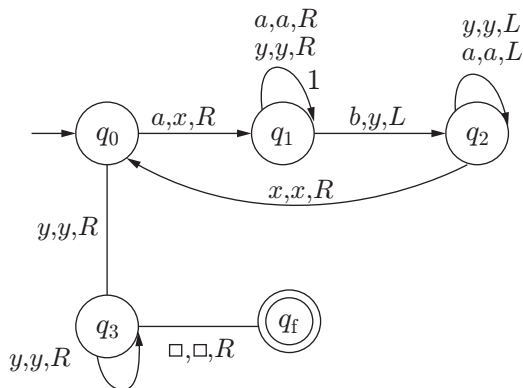


Problem 5.18: Design a TM which accepts a language $L = \{a^n b^n, n \geq 1\}$.

Solution: First make a transition function move based on an input in the tape which is shown in the figure.

- $$\begin{aligned}\delta(q_0, a) &= (q_1, x, R) \\ \delta(q_1, a) &= (q_1, a, R) \\ \delta(q_1, y) &= (q_1, y, R) \\ \delta(q_1, b) &= (q_2, y, L) \\ \delta(q_2, y) &= (q_2, y, L) \\ \delta(q_2, a) &= (q_2, a, L) \\ \delta(q_2, x) &= (q_0, x, R) \\ \delta(q_0, y) &= (q_3, y, R) \\ \delta(q_3, y) &= (q_3, y, R) \\ \delta(q_3, \#) &= (q_f, \#, R)\end{aligned}$$

A TM for a given language $L = \{a^n b^n, n \geq 1\}$.



5.6.3 Recursive and Recursive Enumerable Languages

The properties of recursive and recursive enumerable languages are as follows:

1. A language L is recursive enumerable (RE) if and only if there exists a TM that accepts it, machine must halt and accept $\forall w \in L$.
2. A language L is said to be recursive (REC) if and only if there exists a TM that accepts it and which halts on $\forall w \in \Sigma^*$.
3. REC is a decidable language.
4. RE is a semi-decidable language.
5. Both RE and REC have enumerable procedure (EP). A procedure is called an EP iff it lists all members of L in a finite amount of time.
6. Only REC has membership algorithm.
7. A set S is countable if it has an EP.
8. If A and B are countable then $(A \times B)$ is also countable.
9. Real numbers are uncountable infinite.
10. If a language L is RE then L has an EP but \bar{L} does not have an EP.
11. If a language L is REC then both L and \bar{L} have an EP. Every subset of countable set is countable, so every language L is countable because $L \subseteq \Sigma^*$ which is countable. It means every language has an EP but not those languages which are “not RE” or outside RE.
12. Set of all TM is countable.
13. If L is REC then \bar{L} also will be REC.
14. If L is RE then \bar{L} may or may not be RE.
15. If \bar{L} is RE then L may or may not be RE.
16. Languages (regular, CFL, CSL, REC, RE) are countable infinite.
17. If both L and \bar{L} is Turing enumerable then they are REC.

5.6.4 Variation of Turing Machine

Different variants of Turing machines are available. These variations are given as follows:

1. Multi-track Turing machine
2. Multi-tape Turing machine
3. Non-deterministic Turing machine
4. Universal Turing machine
5. Offline Turing machine
6. Turing machine with semi-infinite machine

5.7 CLOSURE AND DECIDABILITY

There are different closure and decidability properties for various languages. Tables 5.6 and 5.7 describe these properties for different languages.

Table 5.6 | Closure property of languages

Operation	Languages					
	Regular	DCFL	CFL	CSL	REC	RE
\cup	✓	✗	✓	✓	✓	✓
\cap	✓	✗	✗	✓	✓	✓
L^c	✓	✓	✗	✗	✓	✗
$-$	✓	✗	✓	✓	✓	✓
$*$	✓	✗	✓	✓	✓	✓
$L_1 \cup R$	✓	✗	✓	✓	✓	✓
$L_1 \cap R$	✓	✓	✓	✓	✓	✓
L^R	✓	✗	✓	—	—	—
$h(L)$	✓	✗	✓	—	—	—
$h^{-1}(L)$	✓	✓	✓	—	—	—
L_1/L_2	✓	✓	—	—	—	—

✓ {Surely exists}

✗ {May or may not exist}

— {Not known}

Table 5.6 is interpreted as follows:

1. $\{\cup\}$ means that if there are two languages L_1 and L_2 which are regular then their union will be regular, whereas if languages are DCFL then their union will not be DCFL.
2. $\{L_1 \cup R\}$ means that from two languages if one is regular (R) and the other is different, then their union will be result in a language similar to the

second language. For example, if one language is regular and the other is DCFL, then their union will be a DCFL.

3. L^R represents reverse of a language; if there are two CFLs L_1 and L_2 then their union will be a CFL and their intersection will not be a CFL.
4. $(.)$ shows concatenation operation.
5. $(*)$ denotes multiplication operation.

Table 5.7 | Decidability table for languages

Algorithm Exists	Languages				
	Regular	CFL	CSL	REC	RE
Membership: $w \in \Sigma^*, L$ then $w \in L$?	✓	✓	✓	✓	✓
Finiteness: Given language L is finite or infinite?	✓	✓	✗	✗	✗
Emptiness: Language L is empty or not?	✓	✓	✗	✗	✗
Equivalence: Two languages L_1 and L_2 are equal or not?	✓	✗	✗	✗	✗
Ambiguity: Language L is ambiguous or not?	✓	✗	✗	✗	✗
Regularity: Language is regular or not?	✓	✗	✗	✗	✗
Disjointness: Two language intersection is empty or not?	✓	✗	✗	✗	✗
Everything: Language L , then $L = \Sigma^*$	✓	✗	✗	✗	✗

✓ {Denotes that algorithm exists to decide about the question (decidable)}

✗ {Denotes that algorithm does not exist about that question (means undecidable)}

IMPORTANT FORMULAS

1. Number of states in a machine which accept m a 's and n b 's:

$$[(m + 1)(n + 1) + 1]$$
2. Mod n machines have n states.
3. Mod machine have no trap states.
4. If a machine accepts string length exactly n , then it has $(n + 2)$ states.
5. If a machine accepts string length $\leq n$, then it has $(n + 2)$ states.
6. If a machine accepts string length $\geq n$, then it has $(n + 1)$ states.
7. m States, n outputs Mealy machine \equiv Moore machine containing $\leq mn + 1$ states.
8. m States, n outputs Moore machine \equiv Mealy machine containing $\leq m$ states.
9. Output length of Mealy machine is equal to input length.
10. Output length of Moore machine is one greater than the input length because first output symbol is additional without reading any symbol from the input.
11. If language is finite then it will surely be regular. Example: Let $L = \{a^m b^n, m + n = 10\}$ as m and n are finite so it is a regular language. But it does not mean that if a language is infinite then it cannot be regular such as $(a^* b^*)$ is an infinite and regular language.
12. If there is a comparison between m and n then it will be a CFL. Example: Let $L = \{a^m b^n, (m \geq n)\}$.
13. Let language $L = \{a^m b^n\}$ and if m or n is non-linear then it will not be accepted by PDA, so it will fall in the category of CSL.
14. If there are more than one comparison then it will be a CSL. Example: Language $L = \{a^m b^n c^o, m \geq n \text{ and } o \geq n\}$.
15. All modular machines are regular language.
16. All palindrome languages are CFL language.

Variation of NFA/DFA

17. DFA/NFA + (left \leftrightarrow right) move \equiv (2 - DFA)/(2 - NFA) means whatever a 2 - DFA can do, that can be done by simple DFA with left - right move.
18. DFA/NFA + (left \leftrightarrow right) move + (read/write) head \equiv (2 - DFA)/(2 - NFA).
19. DFA + 1 stack \equiv DPDA).
20. NFA + 1 stack \equiv NPDA).
21. DFA + 2 stack \equiv TM.
22. DFA/NFA + 2 counter \equiv TM.
23. DFA/NFA + 2 stack \equiv DFA/NFA + 2 counter.
24. Power of $\{(DFA/NFA) < (DFA/NFA + 1 \text{ counter}) < (DFA/NFA + 1 \text{ stack})\}$
25. $\{(DFA/NFA + 1 \text{ counter}) < (DFA/NFA + 2 \text{ counter})\}$.

SOLVED EXAMPLES

1. Which of the following is a regular expression?

- (a) $L = \{0^m 1^n \mid m, n \geq 0\}$
- (b) $L = \{0^n 1^n \mid n \geq 0\}$
- (c) $L = \{xx \mid x \in \{0, 1\}^*\}$
- (d) $L = \{1^{n^2} 0^n \mid n \geq 0\}$

Solution: In option (a), the given expression will generate any number of 0's followed by any number of 1's, which can be accepted by FA. So, it is a regular expression.

In option (b), the given expression will generate number of 0's followed by number of 1's (both should be same in number), which will be accepted by PDA. So, it is not a regular expression.

In option (c), the given expression will generate any string "x" twice, which will be accepted by PDA. So it is not a regular expression.

In option (d), the given expression will generate twice the n number of 1's followed by n number of 0's, which cannot be accepted by FA. So, it is not a regular expression.

Ans. (a)

2. The class of context-free language is not closed under

- (a) union
- (b) star
- (c) repeated concatenation
- (d) intersection

Solution: Context-free language is not closed under intersection. Please refer table 5.6 (closure property of languages) in text.

Ans. (d)

3. Which of the following is true for regular sets $A = (1101 + 1)^*$ and $((1101)^*1^*)^*$?

- (a) $A \subset B$ (b) $B \subset A$
(c) $A = B$ (d) A and B are incomparable

Solution: Both the regular expressions will generate the same set of strings, so both the regular sets are equal.

Ans. (c)

4. The language generated by the following grammar is $S \rightarrow aSa|bSb|\epsilon$

- (a) $a^m b^n$ $n \geq 0, m \geq 0$
(b) $a^n b^m$ $n \geq 1, m \geq 1$
(c) Odd length palindrome
(d) Even length palindrome

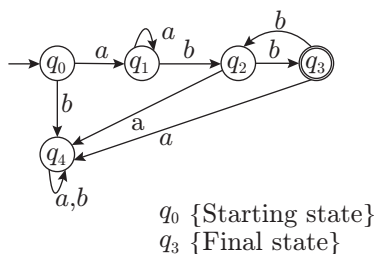
Solution: In the given grammar, variable S will generate a pair of symbol 'a' or symbol 'b' with variable S in the middle of them, and this is in loop so it can be generated any number of times. In the last variable, S will be replaced by ϵ . Some of the strings which can be generated by the given grammar are $\{aa, bb, abba, aabbaa, baab, \text{etc.}\}$. So, the given grammar will generate even length palindrome.

Ans. (d)

5. Number of states in DFA accepting the following language is $L = \{a^n b^{2m} | n, m \geq 1\}$

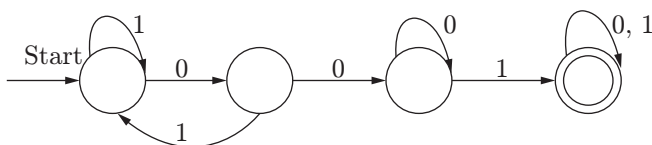
- (a) 2 (b) n
(c) m (d) 5

Solution: DFA for given language $L = a^n b^{2m} | n, m \geq 1$ is



Ans. (d)

6. Consider the following DFS automaton M .



Let S denote the seven bits binary strings in which the first, fourth and last bits are 1. The number of strings in S that are accepted by M is

- (a) 6 (b) 5
(c) 4 (d) 7

Solution: Strings which will be accepted by machine M can be found by fixing 1 at places first, fourth and last. We will have following four strings:

1001001 1001011 1001111 1111001

Ans. (c)

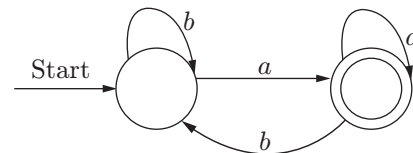
7. Which of the following CFG will generate the language $L = \{a^m b^n c^p d^q | m + n = p + q\}$

- (a) $S \rightarrow aSd|A|B$ (b) $S \rightarrow aSd|A|B$
 $A \rightarrow aAc|C$ $A \rightarrow aAc|D$
 $B \rightarrow bBd|D$ $B \rightarrow bBd|C$
 $C \rightarrow bCc|\epsilon$ $C \rightarrow CBc|\epsilon$
(c) $S \rightarrow aSd|A|B$ (d) $S \rightarrow aSd|A|B$
 $A \rightarrow aAB|C$ $A \rightarrow aAc|C$
 $B \rightarrow bBd|C$ $B \rightarrow bBd|C$
 $C \rightarrow bCc|\epsilon$ $C \rightarrow bCc$

Solution: Strings generated by L should have sum of number of 'a' and 'b' equal to total number of 'c' and 'd'. Only in option (c), the grammar can generate such string which has number of $(a + b)$ equal to number of $(c + d)$.

Ans. (c)

8. Consider the FA shown in the figure below, which language is accepted by the FA



- (a) $b + (a + b)^* b$ (b) $(b + a + b)^* a$
(c) $b + a^* b$ (d) $b + a^* b^*$

Solution: Given that FA will accept any string which has the last symbol 'a'. So, only option (b) expression $(b + a + b)^* a$ can generate all combinations of strings which end with symbol 'a'.

Ans. (b)

9. Let $L = L_1 \cap L_2$, where L_1 and L_2 are languages defined below

$L_1 = \{a^m b^m c a^n b^m | m, n \geq 0\}$ and $L_2 = \{a^i b^j c^k | i, j, k \geq 0\}$. Then L is

- (a) regular but not recursive
(b) context-free
(c) context-free but not regular
(d) recursively enumerable but not context-free

Solution: Language L_1 is CFL and language L_2 is regular. So, the result of $L_1 \cap L_2$ will be context free.

Ans. (c)

10. Which of the following is/are correct?

- I. A language is context-free if and only if it is accepted by the PDA.
- II. PDA is a finite automata with push-down stack.

- (a) Only I is true
- (b) Only II is true
- (c) Both I and II are true
- (d) Both I and II are false

Solution: Both I and II are true.

Ans. (c)

11. How many strings of length less than 4 contains the language described by the regular expression $(x + y)^*y(a + ab)^*$?

- (a) 3
- (b) 9
- (c) 10
- (d) 11

Solution: From the regular expression $(x + y)^*y(a + ab)^*$, it can be seen that it contains 11 strings as follows:

$\{xy, yy, y, ya, yab, xya, yya, yaa, xxy, yyy, xyy\}$

Ans. (d)

12. Consider the following laws:

- $L_1: (L^*)^* = L^*$
- $L_2: \emptyset^* = \emptyset$
- $L_3: \varepsilon^* = \varepsilon$
- $L_4: L^+ = L^* + \varepsilon$
- $L_5: L^* = \varepsilon + L^+$

Which of the above laws hold for regular expression?

- (a) L_1, L_2 and L_5
- (b) L_1, L_3 and L_5
- (c) L_2, L_3 and L_4
- (d) L_1, L_4 and L_5

Solution:

L_1 : both the expression generates any string made up of input symbol of L . it also include ε .

L_2 : \emptyset^* can generate $\{\emptyset \text{ and } \varepsilon\}$

L_3 : ε^* can only generate ε .

L_4 : L^+ contain all the string made up of input symbol of L excluding ε .

L_5 : L^* contain all the string made up of input symbol of L including ε .

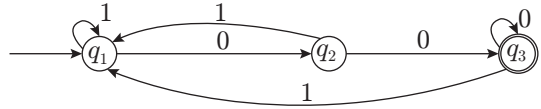
So only L_1, L_3 and L_5 are applicable.

Ans. (b)

13. How many states does the DFA constructed for the set of all strings ending with "00" have?

- (a) 1
- (b) 3
- (c) 2
- (d) 2^2

Solution: DFA will construct as:



Ans. (b)

14. Which of the following CFG cannot be simulated by an FSM?

- (a) $S \rightarrow Sa|b$
- (b) $S \rightarrow aSb|ab$
- (c) $S \rightarrow abX, X \rightarrow cY, Y \rightarrow d|aX$
- (d) None of these

Solution: Given Grammar $S \rightarrow aSb|ab$ generates language $\{a^n b^n\}$ means language contains equal number of a 's and b 's. So, it cannot be simulated by finite state machine.

Ans. (b)

15. Let $r = 1(1 + 0)^*$, $s = 11^*0$ and $t = 1^*0$ be three regular expressions. Which one of the following is true?

- (a) $L(s) \subseteq L(r)$ and $L(s) \subseteq L(t)$
- (b) $L(r) \subseteq L(s)$ and $L(s) \subseteq L(t)$
- (c) $L(s) \subseteq L(t)$ and $L(s) \subseteq L(r)$
- (d) $L(t) \subseteq L(s)$ and $L(s) \subseteq L(r)$

Solution:

$r = 1(1 + 0)^*$, the language corresponds to r having all strings starting with 1.

$s = 11^*0$, the language corresponds to s having all strings starting with 1 followed by any number of 1 and ending with 0.

$t = 1^*0$, the language corresponds to t having all strings ending with 0.

So, $L(s) \subseteq L(r)$ and $L(s) \subseteq L(t)$. Therefore, option (a) is correct.

Ans. (a)

16. Which two of the following four regular expressions are equivalent?

- (i) $(00)^*(\varepsilon + 0)$
- (ii) $(00)^*$
- (iii) 0^*
- (iv) $0(00)^*$

- (a) (i) and (ii)
- (b) (ii) and (iii)
- (c) (i) and (iii)
- (d) (iii) and (iv)

Solution:

(i) $(00)^*(\varepsilon + 0)$ represents any number of 0's

(ii) $(00)^*$ represents even no. of 0's

(iii) 0^* represents any number of 0's

(iv) $0(00)^*$ represents odd number of 0's

So, (i) and (iii) are the same expressions.

Ans. (c)