S.Y.B.Sc.

(Computer Science)

SEMESTER - III (CBCS)

THEORY OF COMPUTATION

SUBJECT CODE : USCS301

Prof. Suhas Pednekar
Vice-Chancellor,
University of Mumbai,

Prof. Ravindra D. Kulkarni
Pro Vice-Chancellor,
University of Mumbai,

Prof. Prakash Mahanwar
Director,
IDOL, University of Mumbai,

Programme Co-ordinator : **Shri Mandar Bhanushe**
Head, Faculty of Science and Technology,
IDOL, University of Mumbai, Mumbai

Course Co-ordinator : **Mr. Sumedh Shejole**
Asst. Professor,
IDOL, University of Mumbai, Mumbai

Editor : **Ms. Prachita Sawant**
Assistant Professor,
Smt. Janakibai Rama Salvi College of Arts
Commerce and Science, Kalwa (W)

Writers **Mrs. Swapna Augustine Nikale**
B. K. Birla College of Arts,
Science and Commerce, kalyan

: **Prof.Smt.Rohini Daund**
K.R.T. Arts, B.H. Commerce and
A.M. Science (K.T.H.M.) College, Nashik

: **Smt. Jyotsna Daund**
M. V. P. Samaj's K. K. Wagh Arts,
Science & Commerce College, Pimpalgaon (B.)

May 2022, Print - I

Published by : Director
Institute of Distance and Open Learning ,
University of Mumbai,
Vidyanagari, Mumbai - 400 098.

DTP Composed and Printed by : Mumbai University Press
Vidyanagari, Santacruz (E), Mumbai - 400098

CONTENTS

SEMESTER III

THEORY

<table>
<tr><td>Course:
USCS301</td><td colspan="2">TOPICS (Credits : 02 Lectures/Week:03)
Theory of Computation</td></tr>
<tr><td colspan="3">Objectives:
To provide the comprehensive insight into theory of computation by understanding grammar, languages and other elements of modern language design. Also to develop capabilities to design and develop formulations for computing models and identify its applications in diverse areas.
Expected Learning Outcomes:
1. Understand Grammar and Languages
2. Learn about Automata theory and its application in Language Design
3. Learn about Turing Machines and Pushdown Automata
4. Understand Linear Bound Automata and its applications</td></tr>
<tr><td>Unit I</td><td>Automata Theory: Defining Automaton, Finite Automaton, Transitios and Its properties, Acceptability by Finite Automaton, Nondeterministic Finite State Machines, DFA and NDFA equivalence, Mealy and Moore Machines, Minimizing Automata.
Formal Languges: Defining Grammar, Derivations, Languges generated by Grammar, Comsky Classification of Grammar and Languages, Recursive Enumerable Sets, Operations on Languages, Languages and Automata</td><td>15L</td></tr>
<tr><td>Unit II</td><td>Regular Sets and Regular Grammar: Regular Grammar, Regular Expressions, Finite automata and Regular Expressions, Pumping Lemma and its Applications, Closure Properties, Regular Sets and Regular Grammar
Context Free Languages: Context-free Languages, Derivation Tree, Ambiguity of Grammar, CFG simplification, Normal Forms, Pumping Lemma for CFG
Pushdown Automata: Definitions, Acceptance by PDA, PDA and CFG</td><td>15L</td></tr>
</table>

Unit III	**Linear Bound Automata:** The Linear Bound Automata Model, Linear Bound Automata and Languages. **Turing Machines:** Turing Machine Definition, Representations, Acceptability by Turing Machines, Designing and Description of Turing Machines, Turing Machine Construction, Variants of Turing Machine, **Undecidability:** The Church-Turing thesis, Universal Turing Machine, Halting Problem, Introduction to Unsolvable Problems	**15L**

Tutorials :

1. Problems on generating languages for given simple grammar
2. Problems on DFA and NDFA equivalence
3. Problems on generating Regular Expressions
4. Problems on drawing transition state diagrams for Regular Expressions
5. Problems on Regular Sets and Regular Grammar
6. Problems on Ambiguity of Grammar
7. Problems on working with PDA
8. Problems on working with Turing Machines
9. Problems on generating derivation trees
10. Problems on Linear Bound Automata/Universal Turing Machine

Textbook(s):

1) Theory of Computer Science, K. L. P Mishra, Chandrasekharan, PHI,3rd Edition
2) Introduction to Computer Theory, Daniel Cohen, Wiley,2nd Edition
3) Introductory Theory of Computer Science, E.V. Krishnamurthy,Affiliated East-West Press.

Additional Reference(s):

1) Theory of Computation, Kavi Mahesh, Wiley India
2) Elements of The Theory of Computation, Lewis, Papadimitriou, PHI
3) Introduction to Languages and the Theory of Computation, John E Martin, McGraw-Hill Education
4) Introduction to Theory of Computation, Michel Sipser, Thomson

Unit I

1

AUTOMATA THEORY

Unit Structure

1.0 OBJECTIVES

After going through this chapter, the learner will be able to:

- define the acceptability of strings by finite automata
- convert a Non-deterministic Finite Automata (NDFA) to Deterministic Finite Automata (DFA)
- define Mealy and Moore machine
- find minimization of Deterministic Finite Automata (DFA)

1.1 INTRODUCTION

Automata theory is the study of abstract machines and computational problems. It is a theory in theoretical computer science and the

word automata originate from a Greek word that means "self-acting, self-willed, self-moving". In this chapter, we will discuss much in detail about what is an automaton, its types, and conversion among the types.

1.2 DEFINITION OF AN AUTOMATON

Automaton refers to a system that transforms and transmits the information which is used for some action without any human intervention. Some of the examples of such systems are automatic coffee maker, automatic sentence completion, automatic ticket generation, etc. In computer science, an "automaton" can be elaborated more accurately using the following three components.

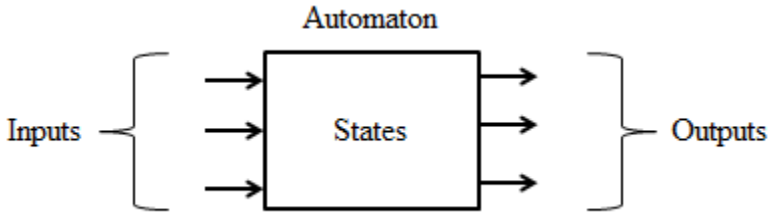


Fig. 1.2 Components of an Automaton

(i) **Input:** It indicates the value taken from the input alphabet Σ and applied to the automaton at the discrete instant of time. Represented as I_1, I_2, …., I_n.

(ii) **Output:** It refers to the response of the automaton based on the input taken. Represented as O_1, O_2, …., O_n.

(iii) **State:** At a specific instant of time, the automaton can take any state. Represented as q_1, q_2, …., q_n

(iv) **State Relation:** It refers to the next state of automaton based on the current state and current input.

(v) **Output Relation:** The output is related to either state only or boththe input and the state.

Note:

Automaton with memory: Output of automaton depends only on input

Automaton without memory: Output of automaton depends on input as well as states

1.3 FINITE AUTOMATON

A finite automaton is represented using 5-tuple such as (Q, Σ, δ, q_0, F), where,

(i) Q = finite nonempty set of states

(ii) Σ = finite non empty set of input symbols (input alphabet)

(iii) δ = direct transition function (a function that maps $Q \times \Sigma$ into Q which describes the change of states)

(iv) q_0= initial state

(v) F = set of final states ($F \subseteq Q$)

Note: A finite automaton can have more than one final state.

1.4 Transition Systems

An automaton can be represented diagrammatically througha transition system, also called a transition graph. The symbols used to draw a transition graph and its description is given below.

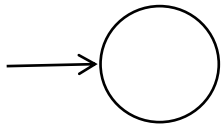
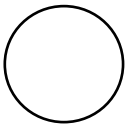
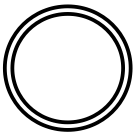
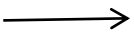
Symbol	**Description**
	Start state
	Intermediate state
	Final state
	State Relation (label on the edge represents input)

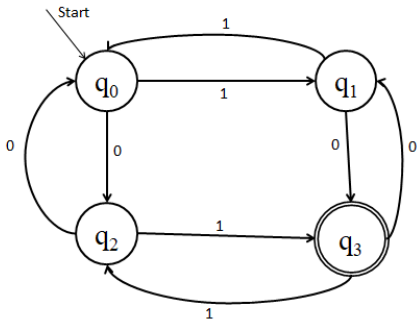
Table 1.4 Symbols used in the transition graph

Example 1.4. Consider an finite automaton, ($Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0,1\}$, δ, q_0, $F=\{q_3\}$) where its transition state table is given below,

State\Input	0	1
$q_0 \rightarrow$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
$*q_3$	q_1	q_2

Note: → near the state represents "start" state and * represents the final state

Solution: The automaton is represented using the transition diagram given below,



0010 is an accepted string whereas 101 is not an accepted string. For any string to be accepted, it should commence from the start state (vertex) and end atthe final state (vertex).

1.4.1 Properties of Transition Systems

Property 1: The state of thesystem can be changed only by an input symbol.

Property 2: For all strings xand input symbols b,

$\delta(q, bx) = \delta(\delta(q, b), x)$

$\delta(q, xb) = \delta(\delta(q, x), b)$

The above property states that an automaton reads thefirst symbol of a string bx and the state after the automaton consumes a prefix of the string xb. Consider example 1.4, where string "0010" is the acceptable string of the automaton. Here the state reads the first symbol of string "0010" say, "0" where the start state was q_0. After which it would have reached the next state say, q_2. Thus, the prefix of q_2 would be the string "0". The same is applicable for all the states.

1.5 ACCEPTABILITY BY FINITE AUTOMATON

A string (finite sequence of symbols) is said to be acceptable if and only if the last state is a final state. If the final state is not reached at the end of the string, then the string is said to be not acceptable.

Example 1.5.1. Consider an finite automaton, ($Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0,1\}$, δ, q_0, $F=\{q_3\}$) where its transition state table is given below,

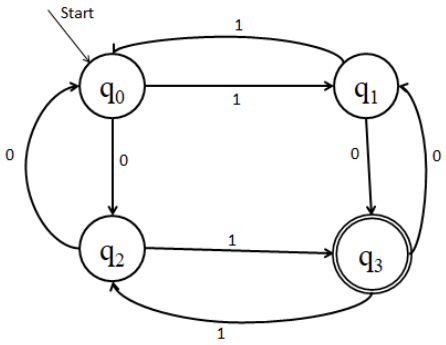
State\Input	0	1
$q_0 \longrightarrow$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
$*q_3$	q_1	q_2

Which among the following string is/are acceptable?

(i) 101010 (ii) 11100

Solution:

The automaton is represented using the transition diagram given below,



(i) 101010

$$\delta(q_0, \underset{\downarrow}{1}01010) = \delta(q_1, \underset{\downarrow}{0}1010)$$
$$= \delta(q_3, \underset{\downarrow}{1}010)$$
$$= \delta(q_2, \underset{\downarrow}{0}10)$$
$$= \delta(q_0, \underset{\downarrow}{1}0)$$
$$= \delta(q_1, \underset{\downarrow}{0})$$
$$= q_3 \text{ (Final State)}$$

Since the final state q_3 is reached at the end of the string, the string "101010" is acceptable by the finite automaton.

Note: The ↓ indicates the current input symbol which is in the process by the automaton.

(ii) 11100

$$\delta(q_0, \underset{\downarrow}{1}1100) = \delta(q_1, \underset{\downarrow}{1}100)$$
$$= \delta(q_0, \underset{\downarrow}{1}00)$$
$$= \delta(q1, \underset{\downarrow}{0}0)$$
$$= \delta(q_3, \underset{\downarrow}{0})$$
$$= q_1 \text{ (Non-final State)}$$

Since the state q_1 is not a final state which was reached at the end of the string, the string "11100" is not acceptable by the finite automaton.

1.6 NON-DETERMINISTIC FINITE STATE MACHINES

In non-deterministic finite state machines, the next state of automaton cannot be precisely defined which means that for the same input symbol there may exist a different next state. Consider the non-deterministic finite state machine given below,

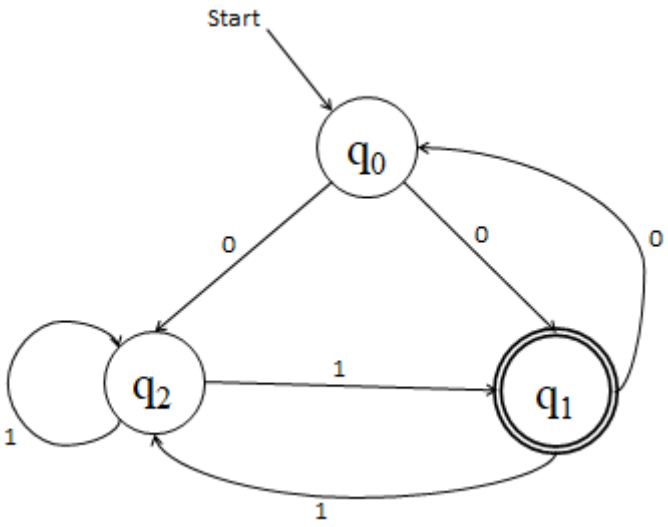


Fig. 1.6 Non-deterministic Finite State Machine

It can be noted that at state q_0 with the input symbol "0", the machine can either reach state $q_1$or q_2. Similarly, at state q_2 with the input symbol "1", the machine can either reach state q_2 itself or q_1. This property makes the automaton non-deterministic.

A non-deterministic finite automaton (NDFA) is represented using 5-tuple such as $(Q, \Sigma, \delta, q_0, F)$, where,

(i) Q = finite nonempty set of states

(ii) Σ = finite non empty set of input symbols (input alphabet)

(iii) δ = direct transition function (a function that maps $Q \times \Sigma$ into Q which describes the change of states)

(iv) q_0 = initial state

(v) F = set of final states ($F \subseteq Q$)

The only difference between a deterministic finite automaton (DFA) and a non-deterministic finite automaton (NDFA) is that in DFA, the transition function δ can contain only one state whereas in NDFA, the transition function δ can contain a subset of states.

The transition state table for Figure 1.6 is given below,

State\Input	0	1
$\rightarrow q_0$	$\{q_1, q_2\}$	-
$*q_1$	q_0	q_2
q_2	-	$\{q_1, q_2\}$

Note: The above transition state table for and NDFA contains a subset of states at $\{q_0,0\}$ and $\{q_2, 1\}$

1.7 DFA AND NDFA EQUIVALENCE

Listed below are the two properties that state the relation between DFA and NFA.

- The performance of NDFA can be simulated by DFA by adding new states.
- For one input symbol, NDFA can have zero, one, or more than one move for a specific state.

Consider a non-deterministic finite automaton (NDFA) is represented using 5-tuple such as $(Q, \Sigma, \delta, q_0, F)$ and deterministic finite automaton (DFA) is represented using 5-tuple such as $(Q', \Sigma, \delta, q_0, F')$. The steps for converting an NDFA to DFA are listed below.

Step 1: Initially let Q' be an empty set

Step 2: Add the start vertex q_0 to Q'

Step 3: For every state which is added to Q', find the possible set of states for each input symbol with the help of the transition state table. Two of the following actions are to be performed.

- If the state is a new entrant, then add it to Q'
- If the state is not a new entrant, then ignore

Step 4: Repeat Step 3 until no new states are met

Step 5: All the states that contain the final state of NDFA are marked as the final state.

Example 1.7.1 Construct a DFA from the NFA given below.

State\Input	0	1
$q_0 \longrightarrow$	$\{q_1, q_3\}$	$\{q_2, q_3\}$
q_1	q_1	q_3
q_2	q_3	q_2
$*q_3$	-	-

Solution:

Step 1: Initially let Q' be an empty set

State\Input	0	1
-	-	-

Step 2: Add the start vertex q_0 to Q'

State\Input	0	1
$q_0 \rightarrow$	$\{q_1, q_3\}$	$\{q_2, q_3\}$

Step 3: For every state which is added to Q', find the possible set of states for each input symbol with the help of the transition state table. Two of the following actions are to be performed.

- If the state is a new entrant, then add it to Q'
- If the state is not a new entrant, then ignore

New state 1: $\{q_1, q_3\}$

Check both $q_1$andq_3 entries from the NDFA transition table

State\Input	0	1
$q_0 \rightarrow$	$\{q_1, q_3\}$	$\{q_2, q_3\}$
$\{q_1, q_3\}$	q_1	q_3

Step 4: Repeat Step 3 until no new states are met

New state 2: $\{q_2, q_3\}$

State\Input	0	1
$q_0 \rightarrow$	$\{q_1, q_3\}$	$\{q_2, q_3\}$
$\{q_1, q_3\}$	q_1	q_3
$\{q_2, q_3\}$	q_3	q_2

New state 3: q_1

State\Input	0	1
~~$q_0 \rightarrow$~~	$\{q_1, q_3\}$	$\{q_2, q_3\}$
$\{q_1, q_3\}$	q_1	q_3
$\{q_2, q_3\}$	q_3	q_2
q_1	q_1	q_3

New state 4: q_2

State\Input	0	1
$q_0 \rightarrow$	$\{q_1, q_3\}$	$\{q_2, q_3\}$
$\{q_1, q_3\}$	q_1	q_3
$\{q_2, q_3\}$	q_3	q_2
q_1	q_1	q_3
q_2	q_3	q_2

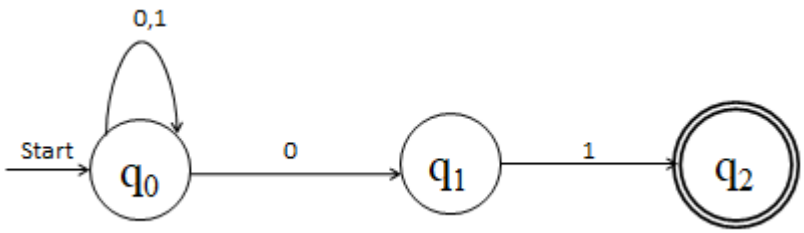
New state 5: q_3

State\Input	0	1
$q_0 \rightarrow$	$\{q_1, q_3\}$	$\{q_2, q_3\}$
$\{q_1, q_3\}$	q_1	q_3
$\{q_2, q_3\}$	q_3	q_2
q_1	q_1	q_3
q_2	q_3	q_2
$*q_3$	-	-

Step 5: All the states that contain the final state of NDFA are marked as the final state.

State\Input	0	1
$q_0 \rightarrow$	$\{q_1, q_3\}$	$\{q_2, q_3\}$
$*\{q_1, q_3\}$	q_1	q_3
$*\{q_2, q_3\}$	q_3	q_2
q_1	q_1	q_3
q_2	q_3	q_2
$*q_3$	-	-

Example 1.7.2 Construct a DFA from the NFA given below.



Solution: The transition state table is generated as,

State\Input	0	1
$q_0 \rightarrow$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	-	q_2
$*q_2$	-	-

Step 1: Initially let Q' be an empty set

State\Input	0	1
-	-	-

Step 2: Add the start vertex q_0 to Q'

State\Input	0	1
~~$q_0 \rightarrow$~~	$\{q_0, q_1\}$	$\{q_0\}$

Step 3: For every state which is added to Q', find the possible set of states for each input symbol with the help of the transition state table. Two of the following actions are to be performed.

- If the state is a new entrant, then add it to Q'
- If the state is not a new entrant, then ignore

New state 1: $\{q_0, q_1\}$

Check both $q_0$andq_1 entries from the NDFA transition table

State\Input	0	1
~~$q_0 \rightarrow$~~	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Step 4: Repeat Step 3 until no new states are met

New state 2: $\{q_0, q_2\}$

State\Input	0	1
$q_0 \rightarrow$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

Step 5: All the states that contain the final state of NDFA are marked as the final state.

State\Input	0	1
$q_0 \longrightarrow$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

1.8 MEALY AND MOORE MACHINES

A model in which the output function depends both on present state q_i and input value x_i is called a Mealy machine.

A model in which the output function depends only on present state q_i and is independent of input value x_i is called a Moore machine.

Consider the example given below where the Mealy machine is represented using the transition state table.

Present State	**Next State**			
	Input 0		**Input 1**	
	State	**Output**	**State**	**Output**
$q_1 \longrightarrow$	q_4	0	q_2	1
q_2	q_1	1	q_4	0
q_3	q_3	1	q_3	0
$*q_4$	q_2	0	q_1	1

For an input string "1011", the transition states are given by q_1 ->q_2 ->q_1 -> q_2 -> q_4. The output string is "1110"

Consider the example given below where the Moore machine is represented using the transition state table.

Present State	**Next State**		**Output**
	Input 0	**Input 1**	
$q_1 \longrightarrow$	q_4	q_2	1
q_2	q_1	q_4	0
q_3	q_3	q_3	0
$*q_4$	q_2	q_1	1

For an input string "1011", the transition states are given by q_1 ->q_2 ->q_1 -> q_2 -> q_4. The output string is "1010"

1.9 MINIMIZING AUTOMATA

Suppose there is a DFA (Q, Σ, δ, q_0, F) which recognizes a language LThen the minimized DFA (Q', Σ, δ, q_0, F') can be constructed for language L as:

Step 1: We will divide Q (set of states) into two sets.

One set will contain all final states and the other set will contain non-final states.

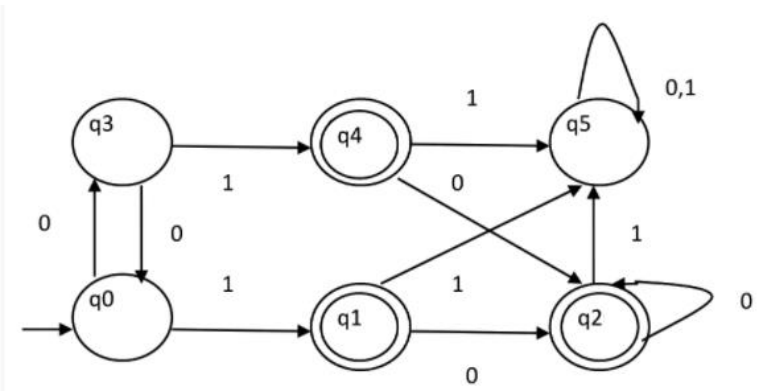
Step2: Initialize k = 1

Step 3: Find Q_k by partitioning the different sets of Q_{k-1}. In each set of Q_{k-1}, take all possible pairs of states. If two states of a set are distinguishable, we will split the sets into different sets in Q_k.

Step 4: Stop when $Q_k = Q_{k-1}$ (No change in the partition)

Step 5: All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in Q_k.

Consider the example of DFA given below:



Step1. Q_0 will have two sets of states. One set will contain the final states of DFA and another set will contain the remaining states. So,Q_0 = {{q_1,q_2,q_4}, {q_0, q_3, q_5}}.

Step 2. To calculate Q_1, check whether sets of partition Q_0 can be partitioned or not.

i) For set { q_1, q_2, q_4} :

Since q_1 and q_2 are not distinguishable and q_1 and q_4 are also not distinguishable, So q_2 and q_4 are not distinguishable. So, { q_1, q_2, q_4} set will not be partitioned in Q_1.

ii) For set { q_0, q_3, q_5 } :

Since q_0 and q_3 are not distinguishable and q_0 and q_5 are distinguishable, So q_3 and q_5 are not distinguishable. So, set {q_0, q_3, q_5} will be partitioned into {q_0, q_3} and {q_5}.

So, Q_1 = {{q_1,q_2,q_4}, {q_0, q_3}, {q_5}}.

iii) For set { q_1, q_2, q_4}:

Since q_1 and q_2 are not distinguishable and q_1 and q_4 are also not distinguishable, So q_2 and q_4 are not distinguishable. So, { q_1, q_2, q_4} set will not be partitioned in Q_2

iv) For set { q0, q3 } :

q_0 and q_3 are not distinguishable

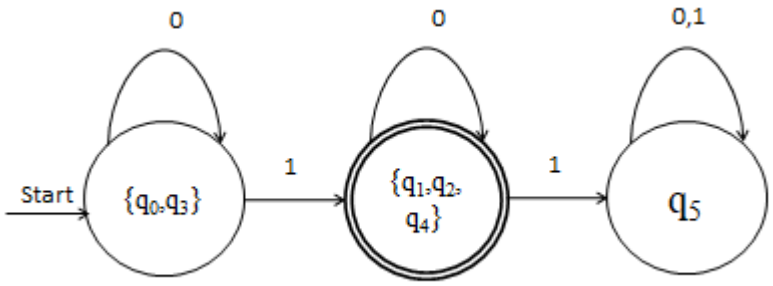
v) For set { q5 }: .

Since only one state ispresent in this set, it cannot be further partitioned.

So, Q_2 = {{q_1,q_2,q_4}, {q_0, q_3}, {q_5}}.

Since Q_1= Q_2. So, this is the final partition.

The minimized DFA is given below,



1.10 SUMMARY

- Automata theory is the study of abstract machines and computational problems.
- Automaton refers to a system that transforms and transmits the information which is used for some action without any human intervention.
- A finite automaton is represented using 5-tuple such as (Q, Σ, δ, q_0, F).
- An automaton can be represented diagrammatically through a transition system, also called a transition graph.
- A string (finite sequence of symbols) is said to be acceptable if and only if the last state is a final state.
- In non-deterministic finite state machines, the next state of automaton cannot be precisely defined which means that for the same input symbol there may exist a different next state.
- A model in which the output function depends both on present state q_i and input value x_i is called a Mealy machine.
- A model in which the output function depends only on present state q_i and is independent of input value x_i is called a Moore machine.

❖❖❖❖

2

FORMAL LANGUAGES

Unit Structure

2.0 OBJECTIVES

After going through this chapter, the learner will be able to:

- understand the concepts of grammars and formal languages
- discuss the Chomsky classification of languages
- study the relationship between the four classes of languages
- implement various operations on languages

2.1 INTRODUCTION

Linguists were trying in the early 1950s to define preciselyvalid sentences and give structural descriptions of sentences. They wanted todefine formal grammar (i.e. to describe the rules of grammar in a rigorousmathematical way) to describe English. Itwas Noam Chomsky who gave a mathematical model of grammar in 1956.Although it was not useful for describing natural languages such as English, itturned alit to be useful for computer languages.

2.2 DEFINING GRAMMAR

A grammar is a quadruple G = (N, Σ, P, S)

where,

1. N is a finite set of nonterminals,

2. Σ is a finite set of terminals,

3. S ∈ N is the start symbol, and

4. P is a finite subset of N × V* called the set of production rules. Here, V = N ∪ Σ. It is convenient to write A → α, for the production rule (A, α) ∈ P.

Consider a grammar G1 − ({S, A, B}, {a, b}, S, {S → AB, A → a, B → b})

Here,

- **S, A,** and **B** are Non-terminal symbols;
- **a** and **b** are Terminal symbols
- **S** is the Start symbol, S ∈ N
- Productions, **P : S → AB, A → a, B → b**

Let P = {S → ab, S → bb, S → aba, S → aab} with Σ = {a, b} and N = {S}. Then G = (N, Σ, P, S) is a context-free grammar. Since the left-hand side of each production rule is the start symbol S and their right-hand sides are terminal strings, every derivation in G is of length one. We precisely have the following derivation in G.

1. S ⇒ ab

2. S ⇒bb

3. S ⇒aba

4. S ⇒aab

Hence, the language generated by G, L(G) = {ab, bb, aba, aab}.

2.3 DERIVATIONS AND LANGUAGES GENERATED BY GRAMMAR

2.3.1 **Derivations generated by Grammar**

Strings may be derived from other strings using the productions in grammar. If a grammar G has a production α → β, we can say that x α y derives x β y in G. This derivation is written as,

$$x \alpha y \Rightarrow G\ x \beta y$$

Let us consider the grammar,

$G = (\{S, A\}, \{a, b\}, S, \{S \rightarrow aAb, aA \rightarrow aaAb, A \rightarrow \varepsilon\})$

Some of the strings that can be derived are,

$S \Rightarrow aAb$ using production $S \rightarrow aAb$

$\Rightarrow aaAbb$ using production $aA \rightarrow aAb$

$\Rightarrow aaaAbbb$ using production $aA \rightarrow aaAb$

$\Rightarrow aaabbb$ using production $A \rightarrow \varepsilon$

2.3.2 Languages generated by Grammar

The set of all strings that can be derived from a grammar is said to be the language generated from that grammar. A language generated by a grammar G is a subset formally defined by

$$L(G)=\{W|W \in \sum^*, S \Rightarrow_G W\}$$

If L(G1) = L(G2), the Grammar G1 is equivalent to the Grammar G2.

Suppose we have the following grammar,

$G = \{S, A, B\}$ $T = \{a, b\}$ $P = \{S \rightarrow AB, A \rightarrow aA|a, B \rightarrow bB|b\}$

The language generated by this grammar $L(G) = \{ab, a^2b, ab^2, a^2b^2, \ldots\ldots\}$

$= \{a^m b^n \mid m \geq 1 \text{ and } n \geq 1\}$

2.4 CHOMSKY CLASSIFICATION OF GRAMMAR AND LANGUAGES

According to Noam Chomsky, there are four types of grammar such as,

- Type 0 (Unrestricted grammar)
- Type 1 (Context-sensitive grammar)
- Type 2 (Context-free grammar)
- Type 3 (Regular grammar)

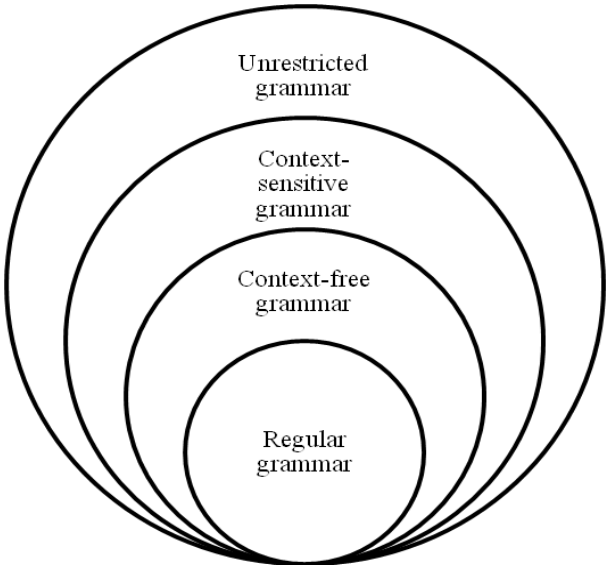


Fig. 2.4 Types of Grammar

Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form $X \rightarrow a$ or $X \rightarrow aY$

where $X, Y \in N$ (Non-terminal),and $a \in T$ (Terminal)

The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule.

Example 2.4.1

$A \rightarrow \varepsilon$

$A \rightarrow a \mid aB$

$B \rightarrow b$

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form $A \rightarrow \gamma$

where $A \in N$ (Non-terminal)

and $\gamma \in (T \cup N)^*$ (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example 2.4.2

$S \rightarrow A\ a$

$A \rightarrow a$

$A \rightarrow aA$

$A \rightarrow abc$

$A \rightarrow \varepsilon$

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages.

The productions must be in the form, $\alpha\ A\ \beta \rightarrow \alpha\ \gamma\ \beta$

where $A \in N$ (Non-terminal)

and $\alpha, \beta, \gamma \in (T \cup N)^*$ (Strings of terminals and non-terminals)

The strings α and β may be empty, but γ must be non-empty.

The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example 2.4.3

$AB \rightarrow AbBc$

$A \rightarrow bcA$

$B \rightarrow b$

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of $\alpha \rightarrow \beta$ where α is a string of terminals and nonterminals with at least one non-terminal and α cannot be null. β is a string of terminals and non-terminals.

Example 2.4.4

$S \rightarrow ACaB$

$Bc \rightarrow acB$

$CB \rightarrow DB$

$aD \rightarrow Db$

2.5 LANGUAGES AND THEIR RELATIONS

Regular Languages are the most restricted types of languages and are accepted by finite automata. Regular Expressions are used to denote regular languages. An expression is regular if,

- ϕ is a regular expression for regular language ϕ.
- ε is a regular expression for regular language $\{\varepsilon\}$.
- If $a \in \Sigma$ (Σ represents the input alphabet), a is regular expression with language {a}.
- If a and b are regular expressions, a + b is also a regular expression with language {a,b}.
- If a and b are regular expressions, ab (concatenation of a and b) is also regular.
- If a is a regular expression, a* (0 or more times a) is also regular.

Regular Grammar: A grammar is regular if it has rules of form A -> a or A -> aB or A -> ε where ε is a special symbol called NULL.

Regular Languages: A language is regular if it can be expressed in terms of a regular expression.

2.6 RECURSIVE ENUMERABLE SETS

Recursive Enumerable languages or type-0 languages are generated by type-0 grammars. A Recursive Enumerable language can be accepted or recognized by the Turing machine which means it will enter into the final state for the strings of language and may or may not enter into rejecting state for the strings which are not part of the language. It means the Turing machine can loop forever for the strings which are not a part of the language. RE languages are also called Turing recognizable languages.

2.7 OPERATIONS ON LANGUAGES

1. Union

If L1 and If L2 are two regular languages, their union L1 $\cup$ L2 will also be regular.
For example,
L1 = $\{a^n \mid n \geq 0\}$ and L2 = $\{b^n \mid n \geq 0\}$
L3 = L1 $\cup$ L2 = $\{a^n \cup b^n \mid n \geq 0\}$ is also regular.

2. Intersection

If L1 and If L2 are two regular languages, their intersection L1 $\cap$ L2 will also be regular.
For example,
L1= $\{a^m b^n \mid n \geq 0$ and $m \geq 0\}$ and L2= $\{a^m b^n \cup b^n a^m \mid n \geq 0$ and $m \geq 0\}$
L3 = L1 $\cap$ L2 = $\{a^m b^n \mid n \geq 0$ and $m \geq 0\}$ is also regular.

3. Concatenation

If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular.
For example,
L1 = {an | n ≥ 0} and L2 = {bn | n ≥ 0}
L3 = L1.L2 = {am . bn | m ≥ 0 and n ≥ 0} is also regular.

4. Kleene Closure

If L1 is a regular language, its Kleene closure L1* will also be regular.
For example,
L1 = (a ∪ b)
L1* = (a ∪ b)*

5. Complement

If L(G) is a regular language, its complement L'(G) will also be regular. The complement of a language can be found by subtracting strings that are in L(G) from all possible strings.
For example,
L(G) = {an | n > 3}
L'(G) = {an | n <= 3}

2.8 LANGUAGES AND AUTOMATA

The relationship among the languages and automata are represented through the following figure,

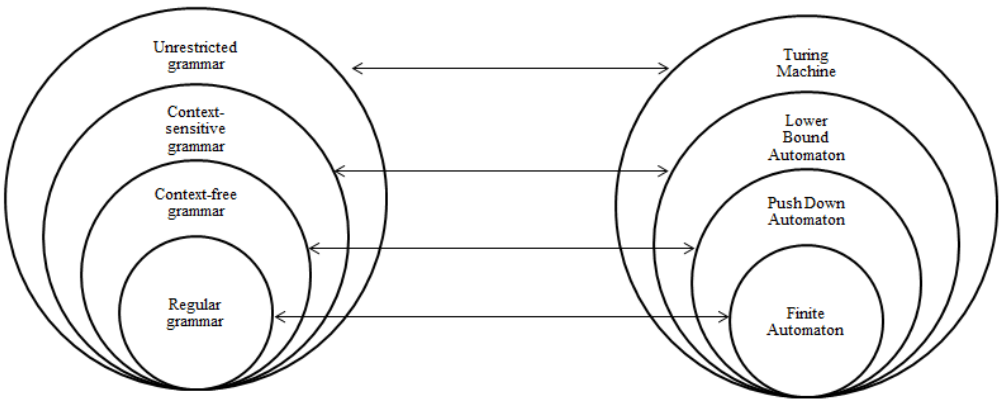


Fig. 2.8 Languages and Automata

2.9 SUMMARY

- A grammar is a quadruple G = (N, Σ, P, S)
- Strings may be derived from other strings using the productions in grammar.
- The set of all strings that can be derived from a grammar is said to be the language generated from that grammar.

- According to Noam Chomsky, there are four types of grammar such as Type 0 (Unrestricted grammar), Type 1 (Context-sensitive grammar), Type 2 (Context-free grammar), and Type 3 (Regular grammar)
- Regular Languages are the most restricted types of languages and are accepted by finite automata.
- Regular Expressions are used to denote regular languages.
- Recursive Enumerable languages or type-0 languages are generated by type-0 grammars.
- Union, Intersection, Concatenation, Kleene Closure, Complement are some of the operations on languages.

❖❖❖❖

Unit II

3

REGULAR SETS AND REGULAR GRAMMAR

Unit Structure

3.0 OBJECTIVES

After the end of this unit, Students will be able to:

- To Understand Concept of Regular Grammar and Regular Expressions
- To Understand Concept of Finite automata and Regular Expressions
- To Learn what is Pumping Lemma for regular languages and its Applications
- To Learn Closure Properties of Regular set
- To Learn about Regular Sets and Regular Grammar

3.1 INTRODUCTION

In this chapter we are learn the concept of the Regular Expression. We first describe regular expressions as a means of representing subsets of strings over ∑ and prove that regular sets are exactlythose accepted by finite automata (FA) or transition systems. We study about pumping lemma for regularLanguages to prove that certain Languages are not regular. Then we studyclosure properties of regular sets. At the and we discuss the relation between regular sets and regular grammars

3.2 REGULAR GRAMMAR:

If Grammar has rules of form

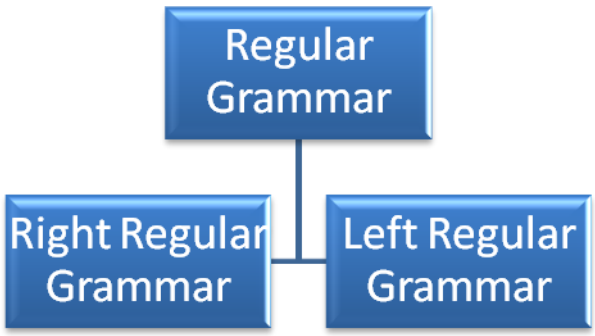
S -> a or

S ->aB or

S -> ε

Where ε is a special symbol called NULL then this grammar called as regular Grammar.

Regular Grammar having two types:



Right Regular Grammars:

Rules of the forms

S → ε

S → a

S → aP

S, P: variables and

a: terminal

Right Regular Grammars:

Rules of the forms

S → ε

S → a

S → Pa

S, P: variables and

a: terminal

3.3 REGULAR EXPRESSIONS

The regular expressions are useful for representing certain sets of strings in an Algebraic fashion. Here, these describe the languages accepted by finite state automata. We give a formal recursive definition of regular expressions over ∑ as follows:

1. Any terminal symbol (i.e. an element of ∑), ε and Ø are regular expressions. When we view *a* in ∑ as a regular expression, we denote it by a.
2. The union of two regular expressions R, and R, written as R1 + R2,is also a regular expression.
3. The concatenation of two regular expressions Rj and R2, written as Rj R2, is also a regular expression.
4. The iteration (or closure) of a regular expression R written as R*, is also a regular expression. 5. If R is a regular expression, then (R) is also a regular expression.
5. The regular expressions over ∑are precisely those obtained recursively by the application of the rules *1-5* once or several times.

Notes:

1. We use x for a regular expression just to distinguish it from the symbol (or string) *x.*
2. The parentheses used in Rule 5 influence the order of evaluation of a regular expression.
3. In the absence of parentheses, we have the hierarchy of operations as follows: iteration (closure), Concatenation, and union. That is, in evaluating a regular expression involving various operations.

3.3.1 Regular Expression Identities

Two regular expressions P and Q are equivalent, then it is written as P = Q if P and Q represent thesame set of strings.

Following are the identities for Regular Expression:

i. Associativity and Commutativity

1. P+Q=Q +P

2. (P + Q) + N=P+(Q+N)

3. (PQ)N = P(QN)

Here note that PM ≠ MP

ii. Distributivity

1. P(M+N)=PM+PN
2. (M + N)P = MP + NP

iii. Identities and Anhilators

1. Ø +P=P+ Ø =P
2. εP = P ε=P
3. Ø P = P Ø = Ø

iv. Idempotent Law

1. P+P=P

v. Closure Laws

1. (P*)* =P*
2. (Ø)* = ε
3. ε * = ε
4. P^+ = PP* = P*P
5. P* = P^++ ε
6. P* P* = P*
7. (PQ) * P= P(QP) *
8. (P+Q) *=(P*Q*)*=(L*+M*)*

3.3.2 Regular Language Definition and Examples

Definition:"The languages that are associated with regular expressions are called languages and are also said to be defined by finite representation."

Examples:

Q.1 Describe the language defined by the regular expression r=ab*a

Solution:

It is the set of all strings of a's and b's that have at least two letters, that beginand end with a's, and that have nothing but b'sinside (if anything at all) language.

∴ the strings in the Language L are

L = **{**aa**,** *aba, abba, abbba, abbbba, ...***}**

In above strings we can observe that there are minimum two a's it means that strings start and end with '**a'**and in-between any number of b's it may be ε

∴ It represents a Language Lcontain strings over {a, b} for regular expression r=ab*a

Q.2 Find out Regular Language of Regular Expression

Solution:

To find out language for Regular Expression = a + b + c + d

We can take all possible strings from the given expression

Consider, r = a + b + c + d

Then, the strings in language are

$\therefore$ L = { a, b, c, d}

So, from above strings Regular Expression represents Language L consisting of

Stringsover {a, b}

Q.3 Find out regular language of Regular Expression $a^*+ b^+ + c + d$

Solution:

To find out language for Regular Expression = $a^*+ b^+ + c + d$

We can take all possible strings from the given expression

Consider, $r = a^* + b^+ + c + d$

For our understanding break the given regular expression into parts

Then, the possible strings in language are

$a^* = \{\varepsilon , a , aa, aaa,\ldots\ldots \}$

$b^+ = \{b, bb, bbb,..\}$

Now concatenating all strings resultant language is,

$\therefore$ L = {ε, a, aa, aaa, aaaa ….b, bb, bbb, bbbb, c, d}

So, from above strings Regular Expression represents Language L consisting of stringsover {a,b,c,d}

Q.4 Find out regular language of Regular Expression $a^*b + c^+ d$

Solution:

To find out language for Regular Expression = $a^*b + c^+ d$

We can take all possible strings from the given expression

Consider, $r = a^*b + c^+ d$

For our understanding break the given regular expression into parts

Then, the possible strings in language are

$a^* = \{\varepsilon , a, aa, aaa,\ldots..\}$

a*b = {b, ab, aab, aaab,……}

c^+ = {c, cc, ccc, cccc,……}

c^+ d = {cd, ccd, cccd, ccccd,…….}

Now concatenating all strings resultant language is,

∴L = {b, ab, aab, aaab…cd, ccd, cccd…}

So, from above strings Regular Expression represents Language L consisting of

Stringsover {a,b,c,d}

Q.5 Find out Regular Expression for A language L consists of strings over {a, b} contains at least one 'a'

Solution:

The given language consists of strings where at least one 'a' must be present. It may have zero or more occurrences of leading a's and b's and trailing a's and b's.

So the required regular expression for given language will be

∴r= (a + b) * a (a + b) *

Q.6 Find out a regular expression for a language L over ∑ *, where ∑ = {0, 1} such thatevery string in the language begin and end with either 00 or 11.

Solution:

Here given that string must begin and end with either 00 or 11 so we will denote regularexpression as (00+ 11). Now in between zero or more occurrences of 0 or 1 is valid so we can denote regular expression (0 + 1)*. After concatenating this we will get regular expression which represents given language L∴r=(00 + 11) (0+1)* (00+ 11)

3.4 FINITE AUTOMATA AND REGULAR EXPRESSIONS:

Conversion of Regular Expression to Finite automata:

Regular expression represents regular set means language accepted by some automata; for every regular expression there exists an FA which is equivalent to it, accepting the same languages. If there is a simple regular expression, we can directly draw DFA or say NFA without much trouble. But if the regular expression is complicated then it is not possible to draw FA just by looking at it.

There are certain rules to convert regular expression to NFA with ϵ moves which can be followed to convert given regular expression to NFA with ϵ

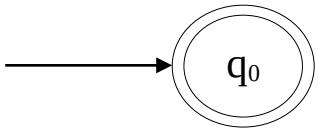
moves which then can be transformed into NFA or directly DFA, by our usual methods.

Method: 1 Basis Zero Operation:

The expression r must be ϵ, ϕ or 'a'. For some a in Σ

We can draw NFA for all these condition as shown in following diagrams.

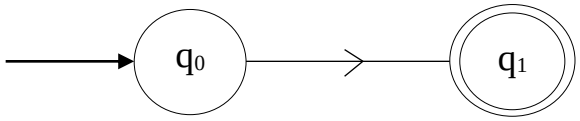
1. r = ϵ



2. r = ϕ

3. r = a

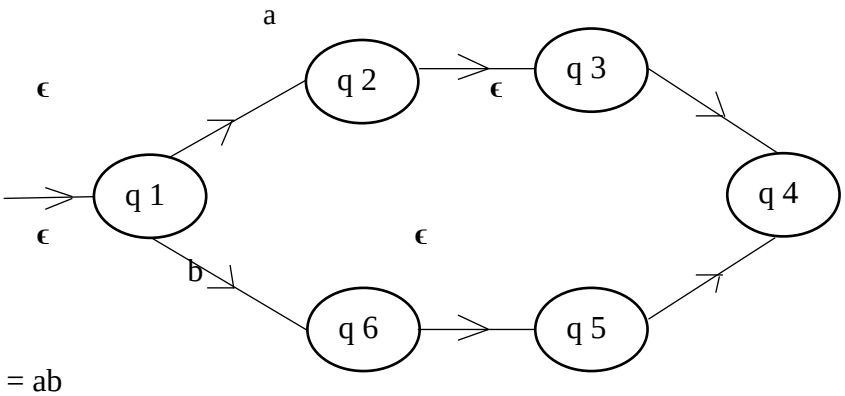
a



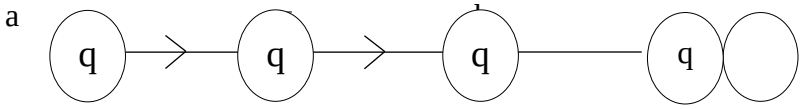
Method 2: Induction (One or More Operations)

Hereare regular expressionssuch that such that there exist an NFA with ε transition that accept Language L(r).There are three cases depending on the forms of r.

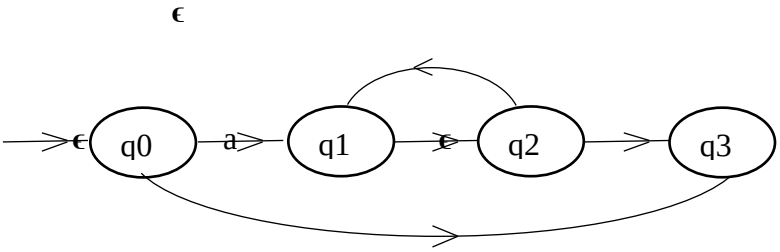
Case1: r = a + b



Case: 2 r = ab



Case 3 r = a*

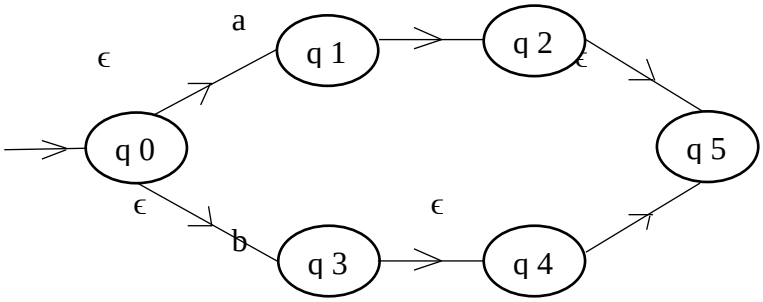


Examples 1

Draw FA with ϵ moves for the regular expression given as a (a+b)*.

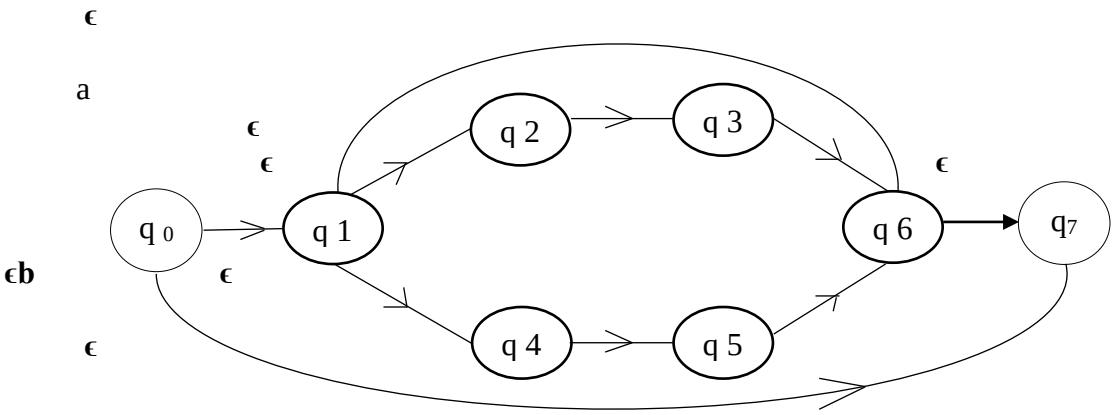
Solution:

Using the above methods, steps for converting given regular expression to FA

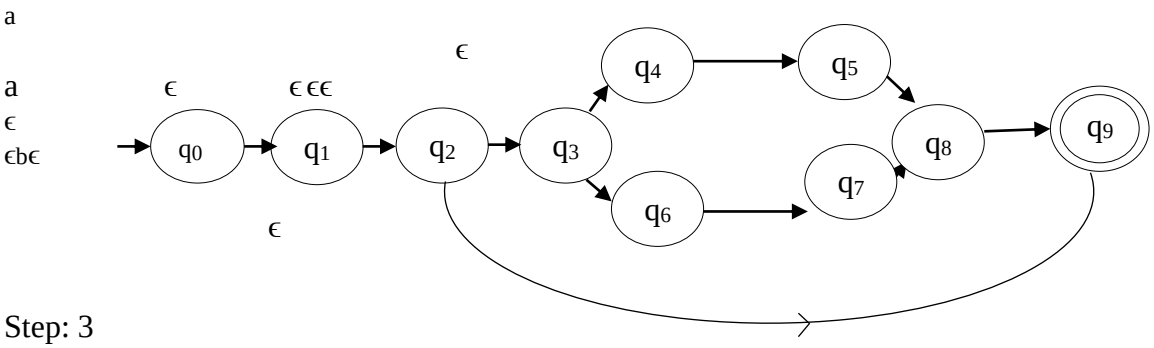


Step: 1

FA with ϵ moves for (a +b)



Step: 2FA with ϵ moves for (a +b)*



Step: 3

Final FA with ϵ moves for a (a+b)*

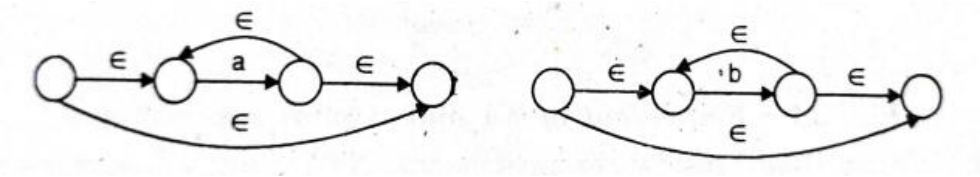
∴ M= ({q_0,q_1,q_2,q_3,q_4,q_5,q_6,q_7,q_8}, {a, b}, p,q_0,q_9)

Example: 2

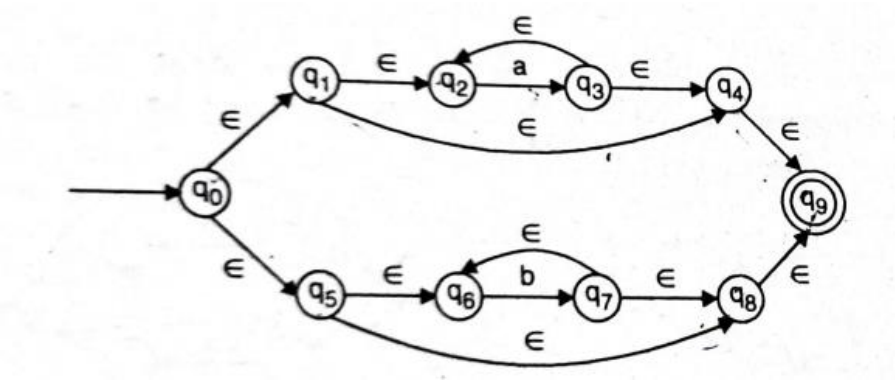
Draw FA with ϵ moves for the regular expression (a*+b*)

Solution:

Using the rules for converting regular expression to FA with ϵ moves we can get FA with ϵ moves for (a*+b*) as in following figure:



Step: 1 FA with ϵ moves for a* **Step: 2**FA with ϵ moves for b*



Step: 3 Final FA with ϵ moves for (a*+ b*)

∴ M= ({q_0,q_1,q_2,q_3,q_4,q_5,q_6,q_7,q_8}, {a, b}, p,q_0,q_9)

3.5 Pumping Lemma and its Applications

Statement of Pumping Lemma:

It states that given any sufficiently long string accepted by an FSM, we can find a substring near the beginning of the string that may be repeated (or Pumped) as many times as we like and the resulting string will still be accepted by the same FSM.

Proof:

Let L (M) be a regular language accepted be a given DFA, M = (Q_1 Σ, δ, q_0, F) with some particular number of notes 'n'

Consider an input of 'n' or more symbols a_1, a_2, a_3 ... am, m>nand for i = 1,2,3,……..m

Let δ (q_0, a_1, a_2, ….a_i,) = q_1

It is not possible for each of the (n+1) states q_0, q_1, q_2… $q_n$to be distinct; because there are only 'n' states and to recognize the string of length m ≥ n it requires at least (n+1) states if we want them to be distinct.

Thus, there exists two integers 'j' and 'k' where, 0 ≤ j < k ≤ Such that q_j = q_kConsider the transition diagram for the DFA M, as given in the following figure.

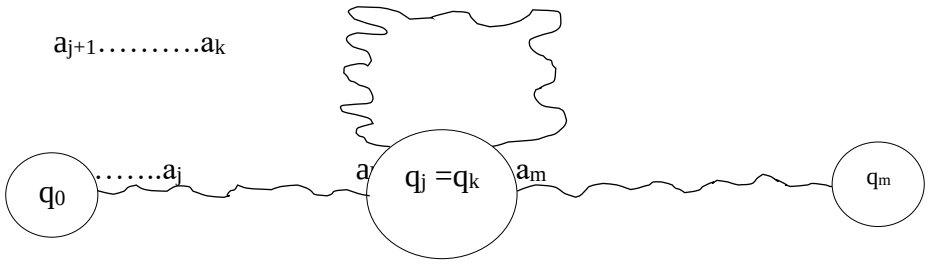


Figure: Pumping Lemma

Since j < k, the string "a_{j+1} ……a_k" is of the length at least one and since k ≤ n, its length is no more than 'n'.

i.e. $1 \leq \left| a_{j+1} \ldots\ldots a_k \right| \leq n$

It $q_m \subset F$ i.e. if $q_m$is final state that means, " a_1, a_2,…..a_m" is in L (M), then "a1,a2……aj,a_{k+1} a_{k+2}……a_m" is also in L(m); since there is a path from q_0 to q_m that goes through $q_j$but not around the loop labeled a_{j+1} …..a_k.

Similarly, we can go around the loop as many times as we like and the resultant string will still be in L (m).

i.e. a_1……..a_j $(a_{j+1}a_k)^I$ a_{k+1} …….$a_m \subset$ L(M)

For any i≥0 (i.e closure – zero or more occurrences) Hence the proof.

Formal statement of pumping Lemma:

Let 'L' be a regular set. Then there is a constant 'n' such that if'z' is any word in 'L' and $\left| Z \right| \geq n$, we may write z = uvw in such a waythat $\left| uv \right| \leq n$, $\left| v \right|$. i.e. $1 \leq \left| v \right| \leq n$ and for all $i \geq 0$, u v^l w is in L.Proof (of the formal statement):

Consider, z = a, a,

u = a, a2

V =a_{j+1}……..a_k

W= a_{k+1}

Using above consideration, the previous proof can be a proof for theformal statement.

Application of pumping Lemma:

It is a dominant tool for demonstrating certain languages non-regular. Given a language, with the assistance of pumping lemma, we can define whether it is a regular language or non-regular language.

Example: 1

Prove that, the following language is non-regular using Pumping lemma,

$\{a^n\ b^{n+1} \mid n > 0\}$

Solution:

a) given $n > 0$

For $n = 1$, $a^n b^{n+1} = a\ b^2$, length $= 1 + 2 = 3$

$n = 2$, $a^n b^{n+1} = a^2 b^3$, length $= 2 + 3 = 5$

$n = 3$, $a^n b^{n+1} = a^3 b^4$', length $= 3 + 4 = 7$

Now, from observation, we can find out the property of the language given and is that it consist of strings having odd length.

b) Assume that the given language

$L = (a^n b^{n+1} \mid n > 0)$ is regular.

c) Let '$\lfloor$' be the constant of pumping lemma.

d) Let $z = a^{\lfloor}. b^{\lfloor+1}$, where

Length of $z = \mid 2 \mid = \lfloor + \lfloor + 1 = 2 \lfloor + 1$

e) By Pumping lemma we can write 'z' as

$z = uvw$ where,

$1 \leq \mid v \mid \leq \lfloor$

anduv'w for $i \geq 0$ is in L.

f) Let $i = 2$

as we know, from Pumping lemma,

$i \leq \mid v \mid \leq \lfloor$

$(2 \lfloor +1) +1 \leq \mid uv^2 w \mid \leq \lfloor + (2 \lfloor + 1)$

Because, $\mid uvw \mid = 2 \lfloor + 1$

Therefore,

$2 \lfloor + 2 \leq \mid uv^2w \mid \leq 3 \lfloor + 1$

g) $2 \lfloor + 2 \leq \mid u\ v^2\ w \mid \leq 3 \lfloor + 1$

i.e. $2 \lfloor + 1 < \mid u\ v^2w \mid < 3 \lfloor + 2$

Consider, $\lfloor = 1$

$3 < \mid uv^2w \mid < 5$

i.e. length= 4 (not odd)

for, ⌊ = 2

$5 < 1\ uv^2w \mid < 8$

i.e. length = 6, 7 (not always odd)

Thus, the length of "uv^2w" is not always add. That means "uv^2w" isnot in L.

But that is the paradox with Pumping lemma. Therefore, as per our assumption that 'L' is regular, must be wrong,

Therefore, given language $L = \{a^n b^{n+1} \mid n > 0\}$ is non-regular.

Example: 2 Prove that $L = \{a^i b^j c^k \mid k > i + j\}$ is not regular.

Solution:

Step: 1 Assume that L is regular. Let L = T (M) for some DFA with n States.

Step: 2 Let $w = a^n b^n c^{3n}$ in L

By pumping lemma we write w = xyz with $\mid xy \mid \leq n$ and $\mid y \mid \geq 0$

Now consider

$w = a^n b^n c^{3n}$

w = xyz

$xy = a^i$ for some $i \leq n$

Step: 3 Then $xy^{k+1} z = a^{n+jk} b^n c^{3n}$

By choosing k large enough so that n+jk>2n

We can make n+jk+n>3n.

So $xy^{k+1} z \notin L$.

This is paradox to our assumption.

∴ L is not regular.

3.6 CLOSURE PROPERTIES

There are number of operations, when we applied to regular sets and it give result inRegular sets. Means, number of operations on language preserves regularsets.For example, the Union of two regular sets is also generating regular set. Similarly,the concatenation of regular sets is also generating regular set and the Kleeneclosure of regular set is also regular set.

If a class of language is closed under a specificoperation thatfact isentitled as closure property of the class of language.

Theorems

1. The regular sets are closed under union, concatenation and Kleene closure. If X and Y are regular sets.

Then X U Y, (X + Y), X Y and $X^*$are also regular

Proof:

X+Y that means the language of all words in either X orY. Regular expressions forX and Y are r_1 and r_2 respectively.

Then $r_1 + r_2$ is regular expression for X U Y

r_1r_2 regular expressionfor XY.

r_1*is regular expression for X*.

Therefore, all three types of these sets of words adefinable by regular expressions and so are themselves regular sets.

2. Regular set is closed under complementation. If X is regular set, then X'is also regular.

If X is a regular set and $X \subseteq \sum^*$and $\sum^*$ X is a regular set.

If X is a language over alphabet$\sum$, we define its complement X' to the language of all strings of letters for that are not words in X'

Proof:

Let X be X (M). Some of states of this FA, M are final states and some are not.Let's reverse the states of each state, i.e., if it was a final state make it non-final and if it was non-final, make it final.

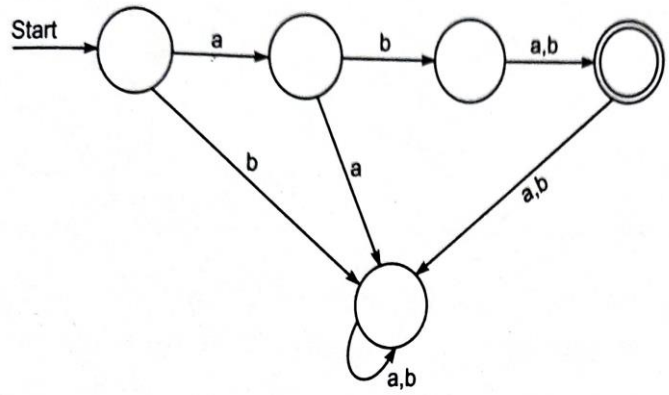
The new Finite Automata accepts all strings that were not accepted by the original FA(X).

∴ Machine accepts the language X'

∴ X' is a regular set.

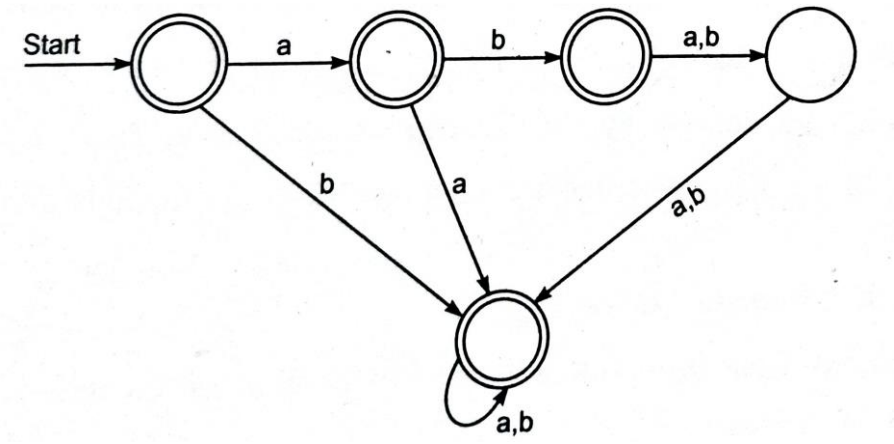
By using Finite Automata we can proof this,

Construct DFA for a language over {a,b} that accept only the strings aba and abb is shown below



We can complement each state

Here make all final states to non–final state and non–final to final states.



Above Finite Automata shows that it accept all strings other than aba and aTherefore, we can prove that complement of regular set is also regular.

3. The regular sets are closed under intersection. If X and Y are regular sets.

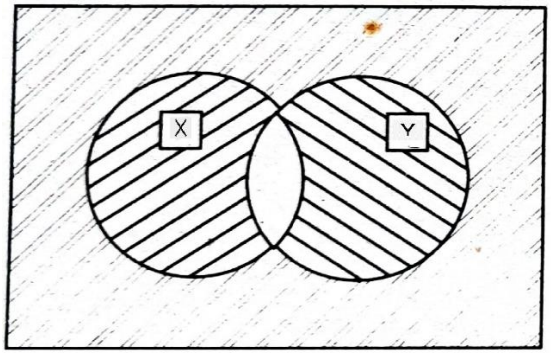
Then X ∩ Yis also regular

Proof:

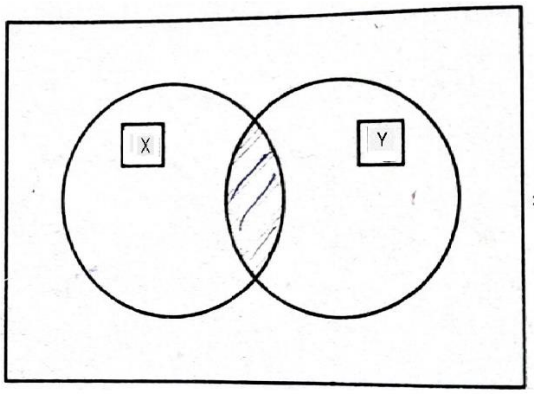
By Using De-Morgan's Law

X ∩ Y = (X' U Y') ' = (X'+Y')' this can be stated by Venn diagram

(X' U Y')



(X'+Y')'



We observed above Venn diagram the language X ∩ Y consists of all words that are not in either X' or Y'. Since X and Y are regular, then so are X' and Y'. Since X' and Y'are regular, so is X' + Y'. And since X' and Y' is regular, then so is $(X' + Y')'$, which meansX ∩ Y, is regular.

Therefore, we can prove that intersection of regular sets is also regular.

3.7 REGULAR SETS AND REGULAR GRAMMAR

Regular Sets

Regular set is theset that represents the value of the Regular Expression.

The class of Regular Set over ∑ is defined as

a) Every finite set of words over alphabet ∑ (including Ø, the empty set or nullset) is a regular set.
b) If X and Y are regular sets over then X U Y (union) and X Y(concatenation) are also regular sets.
c) If P is a regular set over alphabet ∑ then so its closure i.e. S is the smallest class

In other words, the class of regular sets over alphabet ∑containing all finite sets of words over alphabet ∑ and closed under union, concatenation and star operation.

Note: Any set which is predictable by an FSM is regular; and conversely, every regular set can be predictable by some FSM. Regular set is represented by value of regular expression.

Properties of Regular Sets

Property 1. The union of tworegular sets is regular

Proof:

Let us take two regular expression

r_1 = (aa)* and r_2 = a (aa)*

So X = {ε, aa, aaaa, aaaaaa…} (even length h strings including NULL)and

Y = {a, aaa,aaaaa,aaaaaaa,…….} (odd length strings excluding NULL)

X U Y ={ε, a, aa, aaa, aaaa,aaaaa,…….}

(allpossible length strings including NULL)

(X U Y) = a^* (this is also regular expression itself)

∴ **Union of Two sets is regular**

Property 2. The Intersection of Two regular sets is regular

Proof:

Let us take two regular expression

r_1 = a(a*) and r_2 = (aa)*

So X = {a, aa, aaa, aaaa,aaaaa,…….}

(all possible length strings excluding NULL)

and Y = {ε, aa, aaaa, aaaaaa,…….} (even length strings including NULL)

X ∩ Y = { aa, aaaa, aaaaaa,…….} (even length strings excluding NULL)

X ∩ Y = aa$(aa)^*$ (this is also regular expression itself)

∴**Intersection of Two sets is regular**

Property 3. The Complement of a regular set is regular

Proof:

Let us take a regular expression

r = (aa)*

So X = {ε, aa, aaaa, aaaaaa…} (even length h strings including NULL)

Complement of X which is all strings that is not in X

So X'= {a, aaa,aaaaa,aaaaaaa…} (odd length strings excluding NULL)

Regular Expression($X^{'}$) = $a(aa)^*$ (this is also regular expression itself)

∴Complement of a regular set is regular

Property 4. The difference of two regular sets is regular

Proof:

Let us take two regular expression

r1 = a(a*) and r2 = (aa)*

So X = {a, aa, aaa, aaaa,aaaaa,…….}

(all possible length strings excluding NULL)

and Y = {ε, aa, aaaa, aaaaaa,…….} (even length strings including NULL)

X - Y = {a, aaa,aaaaa,aaaaaaa,…….} (odd length strings excluding NULL)

X - Y = $a(aa)^*$ (this is also regular expression itself)

∴ Difference of Two sets is regular

Property 5. The Reversal of a regular set is regular

Proof:

Let us take a regular expression

r = {01+10+11+10}

So X = {01, 10, 11, 10}

Reversal of X is X^R which is all strings that is reverse of X

R^R= {10+01+00+01}

So X^R= {10,01,00,01} which is also regular

∴ Reversal of a regular set is regular

Property 6. The closure of a regular set is regular

Proof:

Let us take a regular expression

r = a (aa)*

So X = {a, aaa,aaaaa,aaaaaaa,…….}

(Odd length strings excluding NULL)

Closure of X is X *

So X'= {a, aa, aaa, aaaa,aaaaa,…….}

(all possible length strings excluding NULL)

Regular Expression(X^*) = a(a)* (this is also regular expression itself)

∴Closure of a regular set is regular

Property 7. The concatenation of two regular sets is regular

Proof:

Let us take two regular expression

r1 = (0+1)*0 and r2 = 01(0+1)*

So X = {0, 00, 10,000, 1100…} (set of Strings ending with 0)

and Y = {01, 010,011,…….} (set of string start with 01)

thenXY = {001,0010,0011,0001,00010,00011,1001, ...} (Set of strings containing 001 as a substring which can be represented by regular expression (0+1)001(0+1) *)

(X U Y) = a* (this is also regular expression itself)

∴Concatenation of two sets is regular

Regular Grammar:

In this Grammar there are following restrictions on type of productions:

1) Left-hand side of each product should contain only one nonterminal
2) Right hand side can contain at the most one non-terminal symbolwhich is allowed to appear as the right most symbol or leftmostsymbol.

The languages generated using this grammar means regular languages are primitive and can be generated and generated using FSM (finite state machine). These regular languages can also be expressed by expressions called as regular expression.

Depending on the position of a non-terminal whether it is leftmost or rightmost, regular grammar is further classified as

1) Left-linear grammar and
2) Right-linear grammar.

1) Left-linear grammar:

We know, regular grammar can contain at the most one non-terminal on the right-hand side of its production. If this variable looks as the leftmost symbol on the right-hand side, the regular grammar is called as be left-linear grammar,Following are forms of productions in left-linear grammar are

$A \rightarrow Bx$, $A \rightarrow \varepsilon$ or $A \rightarrow x$

Where, 'A' and 'B' are non-terminal and 'x' is a string of terminals.

e.g. Consider the following grammar

G = ({S, B, A), (a, b), P, S)

Where, 'P' contains following set of production rule,

S → Aa | Bb

A→ Bb

B → Balb

Above grammar is left-linear in each production has only one nonterminal on the right-hand side and that is the leftmost symbol on the right-hand side.

2) Right-linear grammar:

A regular grammar contains of productions with at the most onenon-terminal on the right-hand side and the right most symbol appears on the right-hand side of the production then the grammar is called right-linear grammar

Following are forms of productions in right-linear grammar are

$A \rightarrow x B$,

$A \rightarrow x$ or

$A \rightarrow \varepsilon$

Where, 'A' and 'B' are non-terminal and 'x' is a string of terminals.

E.g. consider the following grammar

G = ({S, B), (a, b, ε), P, S)

Where, 'P' contains following set of production rule,

S → aB

B → aBl ε

3.8 SUMMARY

- Regular set is the set that represents the value of the Regular Expression.
- Regular expression represents Regularset.
- Pumping Lemma is powerful tool to prove that certain language not regular

- The regular sets are closed under union, concatenation and Kleene closure.
- The regular sets are closed under complementation and also intersection
- By using certain rules we can convert regular expression to NFA with ϵ moves
- Regular grammar is further classified as
  1) Left-linear grammar and
  2) Right-linear grammar.

3.9 REFERENCES

1) Theory of Computer Science, K. L. P Mishra, Chandrasekharan, PHI,3rdEdition
2) Introduction to Computer Theory, Daniel Cohen, Wiley,2ndEdition
3) Introductory Theory of Computer Science, E.V. Krishnamurthy, Affiliated East-West Press.
4) Introduction to Languages and the Theory of Computation, John E Martin, McGraw-HillEducation.

3.10 REVIEW QUESTIONS

Q1. Define following

1) Regularexpression
2) Regular set
3) Regular Grammar

Q2. Construct FA for the following regular expression

1) r = a (a+b)* abba (a+b)*b
2) r = a*b + c^+ d^*
3) r = a (b+c)*a

Q3. Find out regular expression for the following

1) Find out regular (RE) of regular language such that all strings begin & end with ‘a’ i and in between any word usingb
2) Find out RE for not having consecutive Zero
3) Find out RE for at the most 1 Pair of zero’s & one’s

❖❖❖❖

4

CONTEXT FREE LANGUAGE

Unit Structure

4.0 OBJECTIVES:

At the end of this unit, Students will be able to:

- To Understand concept of Context-free Grammar and Context-free Languages
- To Understand concept of Derivation, Derivation Tree and Ambiguous Grammar
- To Learn Simplification of Grammar
- To Learn about Normal Forms
- To Learn about Pumping Lemma for CFG

4.1 INTRODUCTION

This Chapter deals with concepts of grammar and especially about context free Grammarand Context-free Languages.It gives details information about Derivation, Derivation Tree and Ambiguous Grammar.The need of Simplification of Grammar means we can remove Ambiguity of Grammar. Also discussion about Normal Forms and Properties of CFL.The property of CFG is that all productions are of form one Non-terminal→ finite string or terminals and/or nonterminal. The language created by a CFG is called a context-free language.

4.2 CONTEXT-FREE LANGUAGES

It is language generated by CFG means Context Free Grammar;

L (G) = {w/w ⊂ T* and it can be derived from start symbol's'}

Examples:

Q.1.the context free grammar is given as,

S → aSb | ab

Find the CFL generated by the above grammar.

Solution:

Let us start listing or generating the strings that we can generatewith the above CFG. Let us start with minimal length string. Let usnumber the productions as,

Rule (1) S → a S b

Rule (2) S → a b

i) From production (2), we can device string "ab" in one step as,

S => ab

ii) To start with new derivation, we can have,

S => aSb , rule (1)

=> aabb , rule (2)

iii) S => aSb , rule (1)

=> aaSbb , rule (1)

=> aaabbb , rule (2)

iv) S => aSb , rule (1)

=> aaSbb , rule (1)

=> aaaSbbb , rule (1)

=> aaaabbbb , rule (2)

Thus, the language can be listed in the form of set as,

L = {ab, aabb, aaabbb, aaaabbbb…….}

i.e. $L = (a^n b^n \mid n \geq 1)$

Thus, the CFG which is given to us defines the language containingstrings of the form $a^n b^n$ for $n \geq 1$.

Q.2. Write a grammar for generating strings over Σ= {a} containing anynumber

(zero or-more) of 'a's.

Solution:

Zero number of 'a's can be generated using production

SE $\rightarrow \epsilon$ (ϵ - production)

If we want one or more 'a's we can generate them with

S $\rightarrow$ a | as

Combining the two, the grammar that we get is

S $\rightarrow$ aS | a | ϵ

We can represent the grammar formally as

G = ({S}), {a, ϵ }, {S $\rightarrow$ As, S $\rightarrow \epsilon$}, S)

Let us try for the string "aaa"

Using leftmost derivation,

S => as , rule (1)

=> aas , rule (1)

=> aaa , rule (2)

Note:

i) As we know, the language is a regular language and we can denoteit by regular expression; a*.

ii) Above grammar can be simplified further to,

S $\rightarrow$ as | ϵ

This grammar and above grammar are equivalent. Let us do it againfor the string "aaa."

S => as , rule (1)

=> aas , rule (1)

=> aaas , rule (1)

=> aaaϵ , rule (2)

(Because, ' ϵ ' is zero length string or empty string)

Q.3. Write a grammar for the language represented by regular expression,

(a + b)*

Solution:

The regular expression (a + b) represents regular language containingany number of 'a's or 'b's.

The grammar is,

S → a S │bS│ ϵ

(1) (2) (4)

Let us try deriving string with 4 'a's and 1 'b'.

i.e "aaba".

S => aS , rule (1)

=> aaS , rule (1)

=> aabS , rule (2)

=> aabaS , rule (1)

=> aaba , rule (4)

Note: If we want it for language $(a + b)^+$ i.e. strings containing at leastone occurrance of 'a' or 'b', it can be written as,

S→ aS │bs│ a │b

This grammar now cannot generate an empty string i.e the stringcontaining zero number of 'a's and zero number of 'b's, because P does notconsist of the production of the form, S → ϵ.

Closure Properties of CFL:

The flexibility of the rule of context – free grammars is used to establish closure results for the set of context free languages. Operations that preserve context free languages provide another tool for proving that languages are context free. These operations along with pumping lemma, can also the used to show that certain languages are not context free.

Properties:

i. CFL'S are closed under union.
ii. CFL'S are closed under concatenation.

iii. CFL'S are closed under kleene closure and positive closure.

Formalization of the Grammar:

For building a formal model, we should consider two aspects of thegiven grammar:

1) The generative capacity of the grammar i.e, the grammar used' shouldgenerate all and only the sentences of the language for which it is written
2) Grammatical constituents, like terminals and non-terminals.

A grammar that is based on the constituent structure as describedabove, is called as constituent structure grammar or phase structuregrammar.

Formal Definition of Grammar:

A phrase structure grammar is denoted by a quadruple of the form,

G = (V, T, P, S)

Where,

V: Finite set of non-terminals (variables)

T: finite set of terminals.

S: S is a non-terminals N called as the starting symbol,corresponding to the sentence symbol.

P: finite set of productions of the form,

$\alpha \rightarrow B$

Where, $\alpha, \beta \subset (V \cup T)^*$ and 'α' involving at least one symbol from 'V'i.e. at least one non-terminal.

Here we know,

$V \cap T = \emptyset$ = null set.

'α' and 'β' consists of any number of terminals as well asnon-terminals and they are usually termed as sentential forms. Chomsky had classified the grammars in four major categories. Outof which there is one, with the productions of the form

$A \rightarrow \alpha$

Where, 'A' is any non-terminal

and 'α' is any sentential formis called as Context-free grammar.

As we can observe in this typeof grammar there is a restriction that, on the left hand side of eachproduction there should be only one Non-terminal, e.g. The grammar thatwe have considered generating statement 'Dog runs'

can also be consideredas an example of context free grammar, in short, termed as CFG.

4.3 DERIVATION TREE:

Derivations:

For any string, derivable from start symbol of the grammar, usingthe productions of the grammar, there are two different derivations possiblenamely,

1) Leftmost derivation
2) Rightmost derivation

Example: 1

Consider the grammar given as,

G = ({S, A}, (a, b), P, S)

Where P consists of

S → a A S | a

A → S b A / S S / ba

Derive "aabbaa" using leftmost derivation and rightmost derivation.

Solution:

From the given information, 'S' is a start symbol. Let us number theproductions as,

Rule (1) S → AS

Rule (2) S → a

Rule (4) A → SbA

Rule (4) A → SS

Rule (5) A → ba

(i) Leftmost derivation:

S => aAS by using rule (1)

=> aSAS by using rule (4)

=> aabAS by using rule (2)

=> aabbas by using rule (5)

=> aabbaa by using rule (2)

(ii) Rightmost derivation:

S => a A S by using rule (1)

=> a A a by using rule (2)

=> a S b A a by using rule (4)

=> a S b b a a by using rule (5)

=> a a b b a a by using rule (2)

When a string is to be generated from the given production rules, then it will be veryconvenient to shown the generation of string pictorially. This generation (also calledderivation) when drawn graphically takes a tree form and so it is called derivation tree oralso called parse tree. We observe the following regarding the derivation tree.

i. The root node in the derivation tree is always labelled with the start symbol as all strings are imitative from start symbol
ii. All the leaf nodes are labeled with some terminal symbols of the grammar. (i.e. the elements of Σ). So these nodes are called terminal nodes.
iii. All other nodes are labelled with some non-terminal symbols of the grammar (i.e. the elements of V_N). These are called non-terminal nodes.
iv. If the string *w* of the language generated by the grammar has a length *n*, then there are *n* terminal nodes, arranged from left to right.

Example 2. Consider the following Grammar

G = ({S, A, B}, {a, b}, P, S), where

P = {S → AB

A → a

B → b

}

NOW Consider a string *ab*. The derivationof this string is as shown in Fig.1.1. Notethat the root is considered as S, the start symbol.There are two leaf nodes considered *a* and *b*. Theother nodes corresponds to some non-terminalsymbols of G. Since $|ab|$ is 2, there are two terminal nodes arranged from left to right.

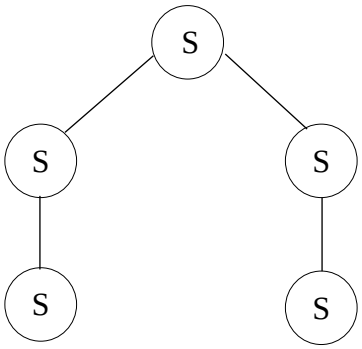


Figure 1.1

Derivation tree for string *ab*

Draw the derivation tree for a string *aabbaa* using following Grammar G.

G = (V_N, Σ, P, S), where

V_N = (S, A) Σ = {*a*, *b*} and

P = {S → aAS

S → a

A → SbA

A → SS

A → ba }

Solution:As *Sis* the start symbol, any string generated by the grammer will be derived from *S*

So we will use

S → aAS.

Obviously we will not use S → a to start with as then we cannot create anything other than *a*

Since we want to generate aabbaa. We should select a proper A-production such that it generates a string beginning with a followed by

So we select

A → SbA

So we get

S→ aSbAS

S→ aabAS

Now we want that A-production which results in a String that begins with *b* followedby a. So we must choose A→ba.

So

S→ aabbas

Which then gives S→ aabbaa the required string. This is shown as below.

S→ aAS

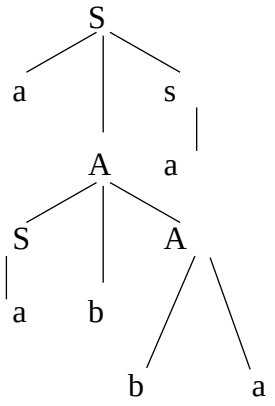
S→ aSbAS by A → SbA

S→ aabAS by S → a

S→ aabbaSby A→ ba

S→ aabbaa by S→ a

The derivation tree is as shown Fig. 2.



For the following grammar show the derivation tree for aaabbabbba.

The grammar G = (V_N, E, P, S)

V_N ={S, A, B}, Σ= {a,b}

P = {S → AB | bA

A → a | As | bAA

B→ b | bS | aBB

}

The derivation of the string is as follows

Start with S

→ aB

→ aaBB

→ aaaBBB

→ aaabSBB

→ aaabbABB

→ aaabbaBB

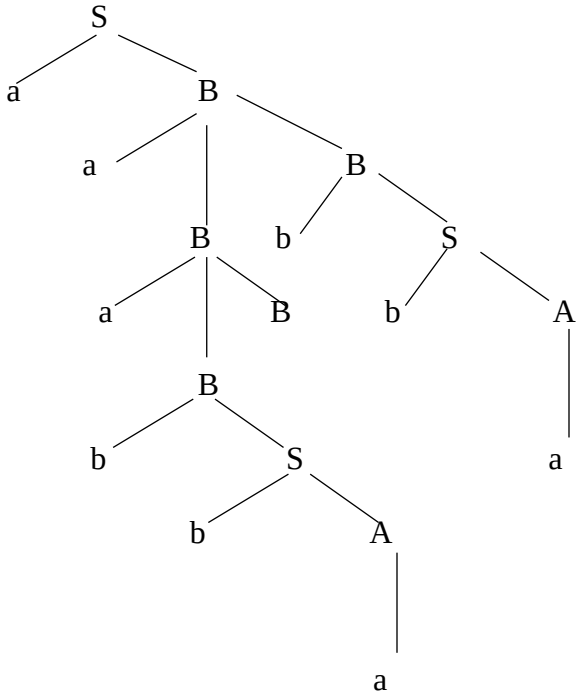
→ aaabbabB

→ aaabbabbS

→ aaabbabbA

→ aaabbabbba

The derivation tree will be as shown in Fig.4.



1. Left and Right Derivation:

Now we will discuss the two methods in which a string Can De derived from theSymbol. These are called left derivation and right derivation.

As Seen earlier, the derivation is either one-step derivation or multi-step derivation each step, Some non-terminal symbol on the right hand side or a production is replacedits definition. Therefore the question is, if there are two or more nonterminal symbols inthe RHS of a production, in what order these nonterminal Symbols can De changed? Doesthe order matters what a resultant string derived?Two orderings are possible. If at each step of derivation, if the leftmost symbol of sentential form is changed by its definition then the derivation is called leftmost derivation

If at each step of derivation, if the rightmost symbol of a sentential form is replaced byits definition then the derivation is called rightmost derivation.

It is Crucial however to note that the ordering used does not matter the generatedstring. e.g. a given string can be derived using either leftmost or rightmost derivation.Consider the string *ab* and the above grammar again. The two derivations are as shownbelow.

S → A B	S → AB
S → a B	S → Ab
S → a D	S → ab
(a) Left most	(b) Rightmost

As stated above, the ordering does not disturb the generated string. However in manyapplications, it is Convenient to use leftmost derivation.

For the grammar,

S → aB 1bA

A → a | aS | bAA

B → b | bS | aBB

Example:1 Write leftmost and rightmost derivation for the string aaabbabbba

Solution:

Leftmost derivation

S→ aB

→ aabB

→ aaaBBB

→ aaabSBB

→ aaabbABB

→ aaabbaBB

→ aaabbabB

→ aaabbabbs

→ aaabbabbbA

→ aaabbabbba

Right most derivation

a→ aB

→ aaB

→ aaBbs

→ aaBbbA

→ aaBbba

→ aaaBBbba

→ aaaBbbba

→ adabSbbba

→ aaabbAbbba

→ aaabbabbba

For the following grammar, give the leftmost and rightmost derivation for the string

abaabb.

G = (V_N, Σ, P, S), where

V_N ={S. X} = {a, b}

P = {S → X baa X

S → ax

X → X a

X → X b

Give leftmost and rightmost derivation.

Leftmost derivation is as follows.

S→ X b aa X

→ Xa baa X

→ ab aa X

→ abaa Xb

→ abaaa X bb

→ abaa bb

Rightmost derivation is as follows

S→ Xbaa X

→ Xbaa Xb

→ XbaaX bb

→ Xbaa blb

→ Xa baa bb

→ ab aa bb

4.4 AMBIGUITY OF GRAMMAR

A CFG is called ambiguous if for at least one word in the language that it createsthere are two probable derivations of the word that corresponds to different syntax trees For this purpose following example can be Consider

Consider the grammar (CFG) G for the language L = $\{a\}^+$

G = ({s}, {a}, P, S), Where

P = { S → aS | Sa | a}

Now, consider a string a^4 (i.e. aaa). This string can be derived in thefollowing different ways as shown in following figure.

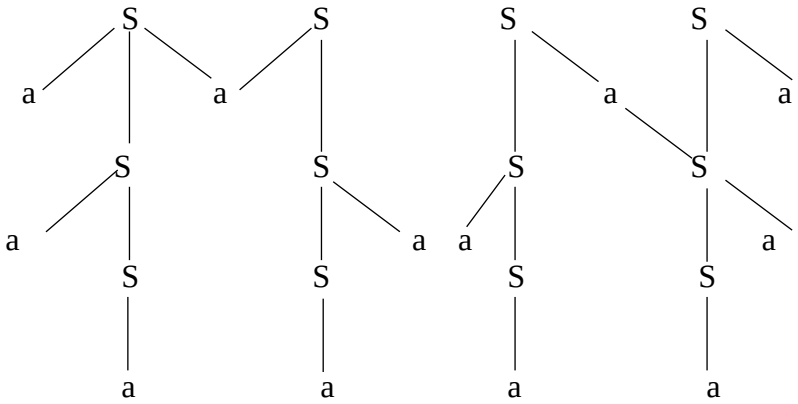


Fig. 4 Four different ways to generate a string aaa

So the grammar is ambiguous.

Example: 1

We will now discuss the best example of an ambiguous grammarthat rises in the context of compiler design. Consider the followgrammar.

G = ({s}, {+, *, d}, P, S), where

P = {S→ S + S | S + S | d }

This grammar is for generating arithmetic expressions made up of operators + and *.Undertake the terminal *d* stands for digit. Now consider a string 5 + 6 * 7. We knowthatexpression of a grammar. But the string can be derived in two differentshown in following Figure and so the grammar is ambiguous. Therefore, the grammar is ambiguous.

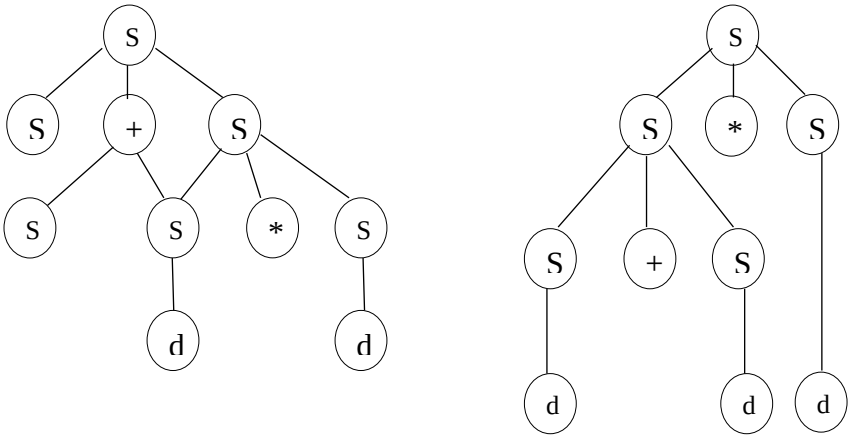


Fig. Two ways to derive a string d + d*d

4.5 CFG SIMPLIFICATION

Following are the rules for having the given context-free grammar inthe reduced form:

1) Each variable and each terminal of CFG should appear in thederivation of at least one word in L (G)
2) There should not be productions of the form A→ B, where 'A' and'B' are both non-terminals.

Simplification of Grammar

Method 1. Removal of Useless Symbol:

A Symbol ‘X’ is useful if

i) Any string must be derivable from ‘X’
ii) ‘X’ must appear in the derivation of at least one string derivable from S (Start Symbol)

- **Removal of Useless Symbols:**

1. A symbol ‘X’ is useful, if there exists a derivation, S => α x β => w
2. Where, 'α', 'β' are sentential forms and 'w' is any string in T*; (w⊂ T*).
3. Otherwise, if no such derivation exists, then symbol 'X' won't appearin any of the derivations, that means, X' is a useless symbol.

- **Three Aspects of Usefulness of a Non-terminalX Are as Follows:**

i) Some string must be derivable from 'X'.
ii) X must appear in the derivation of at least one string derivable from'S'

(Start symbol).

iii) It should not occur in any sentential form that contain a variablefrom which no terminal string can be derived.

Examples: Simplification of Grammar

i. $S \rightarrow AB \mid a \qquad S$

$A \rightarrow a \qquad A$

Simplify the given grammar by removing useless symbol

Step: 1 $S \rightarrow AB \mid a$

$A \rightarrow a$

B is Useless

$\therefore S \rightarrow a$

$A \rightarrow a$

Step: 2 A is Useless

$\{S \rightarrow a\}$

ii. $S \rightarrow AB \mid BC$

$A \rightarrow aAa \mid aAa$

$B \rightarrow Bb \mid b$

$D \rightarrow a\,D \mid d$

Step 1 – C Useless

$S \rightarrow AB$

$A \rightarrow aAa \mid aAb$

$B \rightarrow Bb \mid b$

$D \rightarrow d\,D \mid d$

Step 2 - A & d ARE Useless

A whole grammar is useless

A is useless because no sentence will be derived from D

D is useless because it is not been used in any derived process.

Method 2. Elimination of unit production

A production of the form 'A → B' where, 'A' and 'B' both arenon-terminals, are called ds. Unit productions. All other productions(including ϵ - productions) are Non unit productions.

Elimination Rule:

For every pair of non-terminals 'A' and 'B',

i) If the CFG has a unit production of the form 'A → B' or,

ii) If there is a chain of unit productions leading from 'A' to 'B' such as,

A => X_1 => X_2 =>…..... => B

Where, all X_is (i > 0) are non- terminals, then introduce newproduction (s) according to the rule stated as follows:

"If the non-unit productions for 'B' are,

$B \rightarrow \alpha_1 \mid \alpha_2 \mid ...$

Where, 'α_1, α_2' ... are all sentential forms (not containing only onenon-terminal)

then, create the productions for 'A' as,

$A \rightarrow \alpha_1, \mid \alpha_2 \mid \ldots\ldots$

Single capital letters replaced with its production

Examples: Simplification of Grammar

1) A → B
B → a | b
→ A → B

A → a | b

2) S → Sool | F
F → S | | O | F
T → OS l | 1 | lSlO

→ S → SOOl | F

F → SllO | OS_l | 1 | lSlO

S → SOOl | SllO | OS_l | 1 | lSlO

3) S → A | bb → S → A | bb
A → B | b A → S | a | b
B → S | a s → S | a | b | bb

S → a | b | bb

Method: 3 Removal of ϵ production:

Production of the form 'A → where, 'A' is any non-terminal, iscalled ϵ production

Elimination Procedure:

The procedure for eliminationof ϵ -productions can be stated as follows;the steps involved are,

i. Delete all ϵ-productions from the grammar.
ii. Identify nullable non-terminals.
iii. If there is a production of the form 'A → α', where 'α' is any sentential form containing at least one nullable nonterminal, then add new productions having right hand side formed by deleting all possible subsets of nullable nonterminal from 'α'.
iv. If using step (i) above, we get production of the form 'A → ϵ'then, do not add that to the final grammar.

Examples: Simplification of Grammar

i. S → a S a | b S b | ϵ

→ S → a S a | b S b | aa | bb

ii. S → AB
A → A | BB | Bb
B → b | a A | ϵ

→ S → AB | A

A → SA | BB | Bb | b | B | S

B → b | aA | a

iii. S → ABA
A → aA | ϵ
B → bB | ϵ

→ S → ABA | AA | BA | AB | B | A | ϵ

A → Aa | a

B → bB | b

S → ABA | AA | BA | AB | B | A

A → aA | a

B → bB | b

4.6 NORMAL FORMS

Now we have to discuss the concept of normal form of a Grammar. In a context-freegrammar, the RHS of a production can be any string of variables (nonterminal) andterminals. When the productions in G satisfy certain constraints, then G is said to be in a"normal form'. The two important normal forms which we will now discuss are: Chomsky NormalForm (CNF) and the Greibach Normal Form (GNF).

4.6.1 Chomsky Normal Form:

Definition

If a CFG has only productions of the form

Nonterminal → String of two Nonterminal

Or Nonterminal → one terminal

Then the grammar is in Chomsky Normal Form, CNF.

Note the difference between the CNF and the form of productions we came across in the previous section. The CNF the RHS of each of the production will either contain exactly two nonterminal or a single terminal, while as in the previous form, the RHS of each of the production will either contain string of nonterminal or a single terminal. Thus, CNF is more restricted than the previous one.

Also, that any context-free language that does not contain ϵ as a word has a CFO in CNF that generates exactly it. However, if the CFL contains ϵ, then when we convert the CFG into CNF, the ϵ word drops out of the language while all other words stay the same,

Example 1: Convert the following CFG into CNF.

S → aSa │bSb │a│b│aa│bb.

Solution: First we detached the terminals from the nonterminal.

S→ ASA

S → BSB

S → AA

S→ BB

S→ a

S → b

A → a

B → b

Now all the productions except S → ASA and S → BSB are in required form. Toconvert these productions into the required form we add additional non-terminals, say

R_1, R_2……etc

So we get

S→ AR_1

S→ AA

S→ BB

S → BR_2

S→ a

S→ b

A → a

B→ b

R_1 → SA

R_2 → SB

The grammar is now in CNF.

Example: 2convert the following grammar into CNF.

S → bA │aB

A → bAA │ as │a

B→ aBB │ aS │b

Solution: In the first step we get

S → bA │XB

A → bAA │aS │a

B → aBB │aS │ b

Note that we leave alone the productions A → a and B → b as it is because they arealready in required form.

In the next step, we just need to take care of productions.

A → YAA and B → XBB because they are not in required form.

So, A → YR_1 and B → XR_2

Where R_1 → AA andR_2 → BB

So the grammar in CNF will be,

$S \rightarrow YA \mid XB$

$A \rightarrow YR_1 \mid XS \mid a$

$B \rightarrow XR_2 \mid YS \mid b$

$X \rightarrow a$

$Y \rightarrow b$

$R_1 \rightarrow AA$

$R_2 \rightarrow BB$

Example 3: Convert the following CFG into CNF.

$S \rightarrow aaaaS \mid aaaa$

Which generates the language a^{4n} for n = 1, 2, 4……

Solution: In the first step we get

$S \rightarrow AAAAS$

$S \rightarrow AAAA$

$A \rightarrow a$

Now,

$S \rightarrow AR_1$

$R_1 \rightarrow AR_2$

$R_2 \rightarrow AR_4$

$R_4 \rightarrow AS$

$S \rightarrow AR_4$

$R_4 \rightarrow AR_5$

$R_5 \rightarrow AA$

$A \rightarrow a$

The grammar is now in CNF.

4.6.2 Greibach Normal Form:

We will now look at one more normal form of the grammar.

Definition:

If each production in a CFG is of the form

$A \rightarrow aB$, where

a is a terminal andB is a string of non-terminals (possibly empty), then the grammar is in GreibachNormal Form (GNF).

For example, the following grammar is in GNF.

$S \rightarrow bA \mid aB$

$A \rightarrow bAA \mid aS \mid a$

$B \rightarrow aBB \mid bS \mid b$

All the productions start with a terminal on RHS and followed by a string,non-terminals (sometimes ε).

Before looking at how to convert a given grammar into GNF, we have to discuss two important auxiliary results, which are helpful for converting a grammar to GNF.

Lemma: 1: If $A \rightarrow B\gamma$ is a production, where A and B are non-terminals and they are B- production of the form

$B \rightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_s$ then

We can replace the production $A \rightarrow B\gamma$ by

$A \rightarrow B_i \gamma \mid 1 \leq i \leq S$

For example, take into consideration following grammar

$A \rightarrow Bab$

$B \rightarrow aA \mid bB \mid aa \mid AB$

So using Lemma. 1 in above, we get

$A \rightarrow aA\ ab \mid bBab \mid aaab \mid A\ Bab$

Lemma. 2: If a CFG consists of production of the form

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid .... \mid A\alpha_r \mid \beta_1 \mid \beta_2 \mid ..... \mid \beta_s$ such that each of the B_i's do not start with A then an equivalent grammar can be formed as follows:

$A \rightarrow \beta_1 \mid B_2 \mid ..... \mid B_S$

$A \rightarrow \beta_1 Z \mid \beta_2 Z \mid ..... \mid \beta_S Z$

$Z \rightarrow \alpha_1 \mid \alpha_2 \mid ..... \mid \alpha_r$

$Z \rightarrow \alpha_1 Z \mid \alpha_2 Z \mid ..... \mid \alpha_r Z$

For example, consider following grammar. Here

$A \rightarrow aBD \mid b\ DB \mid c$

$A \rightarrow AB \mid AD$

$\alpha_1 \rightarrow B$, $\alpha_2 = D$

$\beta_1 \rightarrow$ a BD, β_2 = bDB and β_4 = C

So applying Lemma 2 we get

A → a BD | DB | C

A→ a BDZ | bDBZ | cZ

Z → B | D

Z → BZ | DZ

Lemma 2 is useful in dealing with left-recursive productions i.e. the productions of form A → Aα.

We will make use of these lemmas to convert CFG to GNF.

Example:1 Construct a grammar in GNF equivalent to the grammar S → AA | a and

A→ SS | b.

Solution:

Observe that the CFG is in CNF. If we rename S as A_1, and as A A_2 (for conversion purpose) respectively, the productions will be then

$A_1 \rightarrow A_2 A_2 \mid \alpha$

$A_2 \rightarrow A_1 A_1 \mid b$

and

We leave $A_2 \rightarrow b$ as it is in the required form.

Now consider $A_2 \rightarrow A_1 A_1$. To convert this we will use lemma 1 to get

$A_2 \rightarrow A_2A_2A_1$

$A_2 \rightarrow aA_1$

i.e. by replacing the first A_1 on RHS of $A_2 \rightarrow A_1A_1$ by definition of A_1. Now the Production $A_2 \rightarrow aA_1$ is in required form. But we need to use Lemma 2 for

$A_2 \rightarrow A_2A_2A_1$ as it is of form A → Aα.

Apply Lemma 2 to the productions of A_2. A_2 productions are

$A_2 \rightarrow A_2 A_2 A_1$

$A_2 \rightarrow aA_1$

$A_2 \rightarrow b$

Here, $\beta_1 = \alpha A_1$, $\beta_2 = b$, $\alpha = A_2 A_1$

So we have now by adding new non-terminal.

$A_2 \rightarrow aA_1 \mid b$

$A_2 \rightarrow aA_1 Z_2 \mid bZ_2$

$Z_2 \rightarrow A_2 A_1$

$Z_2 \rightarrow A_2 A_1Z_2$

Now all A_2 productions are in required form.

Now we will save to consider the A_1, production,

$A_1 \rightarrow A_2A_2 \mid a$

Out of these $A_1 \rightarrow a$ is in required form.

So consider, $A_1 \rightarrow A_2A_2$

Applying Lemma 1, we get

$A_1 \rightarrow aA_1\ A_2 \mid\ bA_2 \mid\ aA_1\ Z_2A_2 \mid\ bZ_2\ A_2$

So adding to this list the $A_1 \rightarrow a$ production, we have retained all A_1 productions and they are

$A_1 \rightarrow aA_1\ A_2 \mid bA_2 \mid\ aA_1\ Z_2,\ A_2, \mid bZ_2A_2 \mid\ a$

Now we amend Z_2 productions. Applying lemma to Z_2 productions we get

$Z_2 \rightarrow aA_1\ A_1 \mid\ bA_1 \mid aA_1\ Z_2\ A_1 \mid\ bZ_2\ A_1$

$Z_2 \rightarrow aA_1\ A_1\ Z_2 \mid bA_1\ Z_2 \mid aA_1\ Z_2\ A_1\ Z_2 \mid bA_1,\ Z_2$

So the grammar in GNF will be

$G' = (\{A_1, A_2, Z_2,\}, \{a, b\}, P', A_1)$

$P' = \{$

Where

$A_1 \rightarrow a \mid aA_1\ A_2 \mid bA_2 \mid aA_1\ Z_2\ A_2 \mid\ bZ_2\ A_2$

$A_2 \rightarrow aA_1 \mid b \mid\ aA_1\ Z_2\ \mid b\ Z_2$

$Z_2 \rightarrow aA_1\ A_1 \mid bA_1 \mid aA_1\ Z_2A_1 \mid bZ_2A_1$

$Z_2 \rightarrow aA_1\ A_1\ Z_2 \mid bA_1\ Z_2 \mid aA_1\ Z_2\ A_1\ Z_2 \mid bZ_2\ A_1\ Z_2$

$\}$

4.7 PUMPING LEMMA FOR CFG:

The pumping lemma for CFLs gives a method of generating infinite number of stringsfrom a given sufficiently long string in a context-free language L. It is used to prove thatcertain languages are not context-free. The construction we make use of in provingpumping lemma yields some decision algorithms regarding context-free languages.

The pumping lemma for regular sets states that every sufficientlylong string in a regular set contains a short substring that can be pumped. That is, insertingas many copies of the substring as we like, always yields a string in the regular set.The pumping lemma for CFL's states that there are always two short substrings closetogether that can be repeated, both the same number of times, as often as we like. Theformal statement is as follows:

Lemma: 1 (The pumping lemma for context-free languages):

Let *L* be any context-free language. Then there is a constant *n*, depending only on *L*,such that if *Z* is in *L* and $|z| \geq n$, then we may write z = u v w x y such that

(i) $|vx| \geq 1$

(ii) $|vwx| \leq n$ and

(ii) For all $i \geq 0$, $u\ v^i\ w\ x^i\ y$ is in *L*.

4.8 SUMMARY

- **Context-free Languages**

  It is language generated by CFG means Context Free Grammar;

  L(G) = {w/w ⊂ T* and it can be derived from start symbol 's'}

- **Formal Definition of Grammar**:

A phrase structure grammar is denoted by a quadruple of the form,
G = (V, T, P, S)

- **Closure Properties of CFL:**

i) CFL'S are closed under union.
ii) CFL'S are closed under concatenation.
iii) CFL'S are closed under kleene closure and positive closure.

- **Ambiguity of Grammar:**

A CFG is called ambiguous if for at least one word in the language that it createsthere are two probable derivations of the word that corresponds to different syntax trees

- **CFG simplification**

Method 1. Removal of Useless Symbol:

Method 2. Elimination of unit production

Method 3. Removal of ϵ production:

4.9 REFERENCES

1. Theory of Computer Science, K. L. P Mishra, Chandrasekharan, PHI,4rd Edition
2. Introduction to Computer Theory, Daniel Cohen, Wiley,2nd Edition
3. Introductory Theory of Computer Science, E.V. Krishnamurthy, Affiliated East-West Press.
4. Introduction to Languages and the Theory of Computation, John E Martin, McGraw-Hill Education

4.10 REVIEW QUESTIONS:

1. Define following terms.

a) Derivation Tree

b) CFG

2. What is Context Free Language (CFL)?

3. Find the Context Free Language (CFL) associated with the CFG.

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

4. Construct CFG for

$L = \{a\ m\ b\ n\ c\ m \mid n, m \geq 1\}$

5. Remove ambiguity from following grammars.

a) $S \rightarrow aS \mid Sa \mid \epsilon$

b) $S \rightarrow SS \mid AB$

$A \rightarrow Aa \mid a$

$B \rightarrow Bb \mid b$

6. Draw Derivation Tree for a substring “001100”.

7. Construct a grammar to generate stings with no consecutive a’s but may or may notwith consecutive b’s.

8. Convert following CFG to CNF

$S \rightarrow aAab \mid Aba$

$A \rightarrow aS \mid bB$

$B \rightarrow ASb \mid a$

5

PUSHDOWN AUTOMATA

Unit Structure

5.0 OBJECTIVES

In this chapter, Students will be able to:

- To Understand Concept of PDA
- To Understand use of PDA
- To learn acceptance byPDA using Final State and Empty Stack
- To know about how to construct PDA for CFG

5.1 INTRODUCTION

In this chapter we are learn the concept of the PDA. In case of FSM, we have seen that it does not have memory to remember arbitrarily long sequences of the input. So PDM (Pushdown Stack-Memory Machine) is more powerful than FSM (Finite Automata Machine) and more capabilities. FSM accept only the regularlanguages and PDM is consider as a CFL-acceptor or CFL-recognizer. While FA is a mathematical model

of FSM likewise PDA (Push-down Automata) is mathematical model of PDM.

5.2 DEFINITION OF PDA:

A Pushdown automaton M is a seven tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Where, q = finite nonempty set of states

Σ = an alphabet called input alphabet

Γ = an alphabet called the stack alphabet q_o in Q is the initial

Z_0 in Γ is a particular stack symbol called start stack symbol

$F \subseteq Q$ isa set of final states

δ: mapping from $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$.

5.2.1 Elements in PDM:

A PDM is a collection of eight elements described as follows:

1. An input alphabets. (Σ)
2. An input tape (bounded on one end and unbounded or infinite in theother direction). Initially, input string is written on the tape with restof the tape blank.
3. An alphabet of stack symbols. Γ
4. A pushdown stacks. Initially the stack is empty and assumed to becontaining (blank) at the bottom to represent stack empty.
5. Start state.
6. Halt states: Accept and Reject.
7. Non branching state: Push.
8. Branching states: Read, Pop.

PDM thus, can be visualized to have an input tape, a finite control(that we have seen for FSM) and a stack. Thus, we can see that, PDMonly can read from the tape and cannot write onto it; also the direction ofmovement of head is always in one direction from left to the right.Obviously, as it can use an external stock from which it can pop (read)and push (write) into it, it becomes powerful compared to FSM, but notpowerful than TM (head can move to left, to right and remain stationaryalso).

5.2.2 ModelofPushdownAutomaton:

In PDA model the read-only input tape is:

i. Infinite in size.

ii. Divided into equal sized blocks known as cells to store input alphabets.

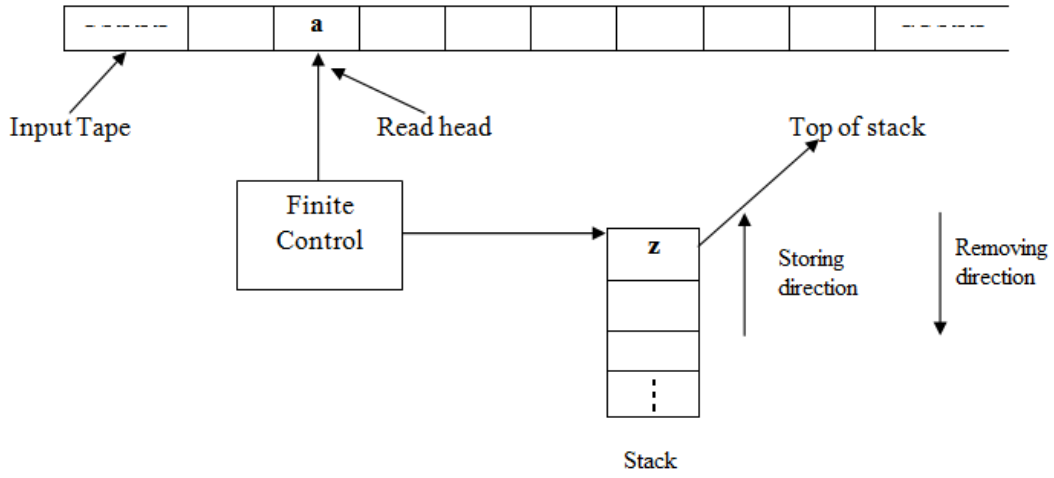
iii. Initially filled with blank symbols (ε).

iv. Using read head weread one letter at a time, when tape is being process on machine. Read head is one directional.

v. Finite control will process the input symbol read and advances Read Head one position ahead.

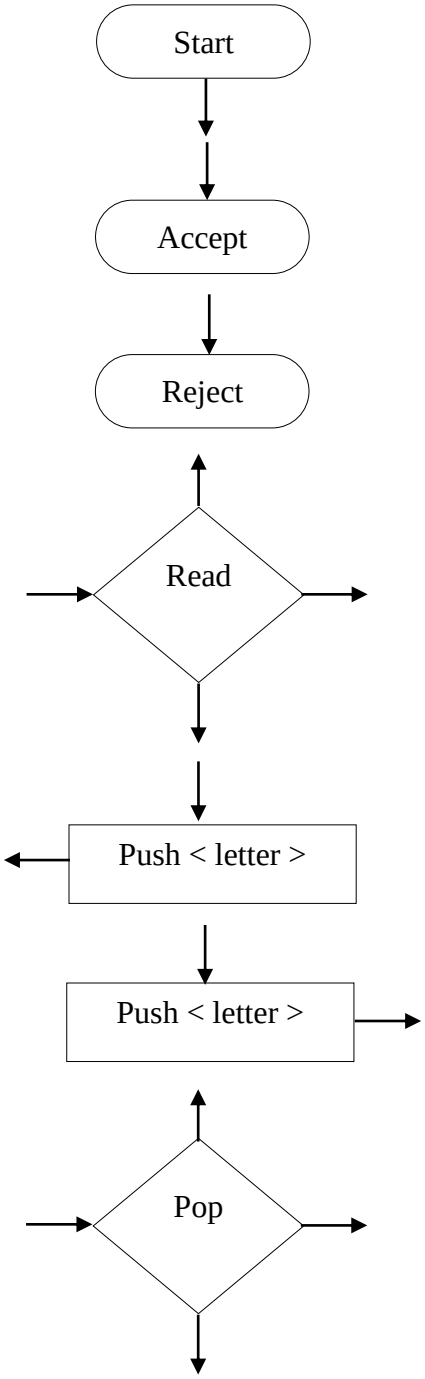
vi. The symbol is either pushed in a stack or popped out from the stack, when they are processing by finite control depending on the logic

vii. While moving towards right, reading the input symbols, when we reach the first blank cell we stop.



5.2.3 Pictorial Representation forPDA:

PDA can be represented by some flow-chart like notations. All thesepictorial representations for different types of states of PDA are given infigure on next page.We can clearly see that, 'start should be the initial state for everyPDA and either 'accept or reject would be the final halt state dependingon the input, whether machine has accepted it or rejected it "Read stateis a conditional block and represented like that because depending on whatsymbol read, the machine could go to different states. Similarly, the Pop'state is also represented by conditional block. "Push' state is anintermediate state and a symbol has to be provided to it which we wantto push.



Start the Process

Accept halt State

Reject Halt State

Read the input symbol from tape, (This is a conditional block and depending on the symbol read, it performs different operations.

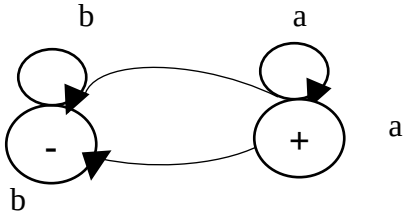
Push a < letter >i.e some symbol on to block and proceed.

Pop a symbol from the stack (It also conditional like read block

Pictorial representation for PDA

Example: 1

Construct PDA recognizing the language accepted by the DFA given in following figure.



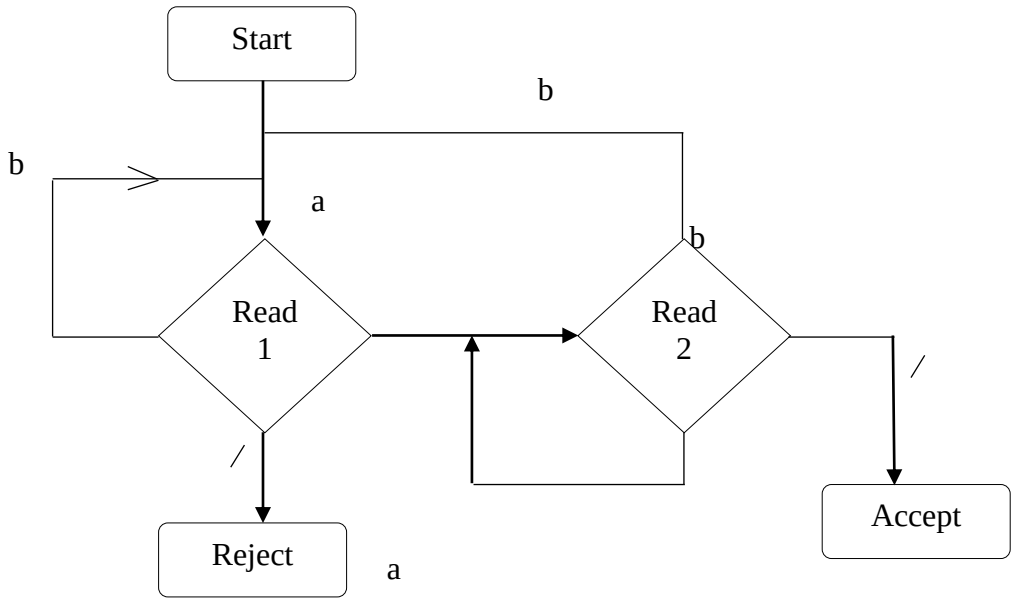
An Example DFA

Solution:

As we know the DFA given in above figure is the acceptor for the regular language represented by regular expression.

b * a a* (b b* a a*)

We can construct the PDA equivalent to given DFA as shown in following figure.

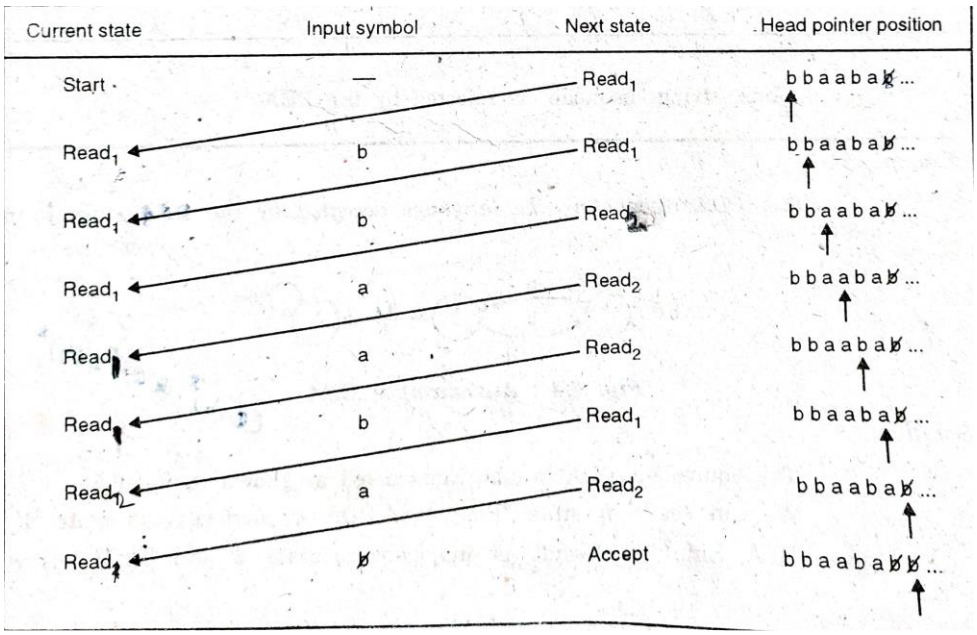


PDA equivalent to FA in above figure

We can see that '$Read_1$' state is analogous to initial state for given DFA and '$Read_2$' state is analogous to the final state of the given DFA. As '$Read_2$' is analogous to the final state, if input ends in '$Read_1$' i.e. if we get a blank 'b' on top in '$Read_1$' machine will move to 'reject' state, else it

moves to 'accept' state as shown. Let us simulate the working of the PDA for the strings 'bbaaba' and 'baaabab'.

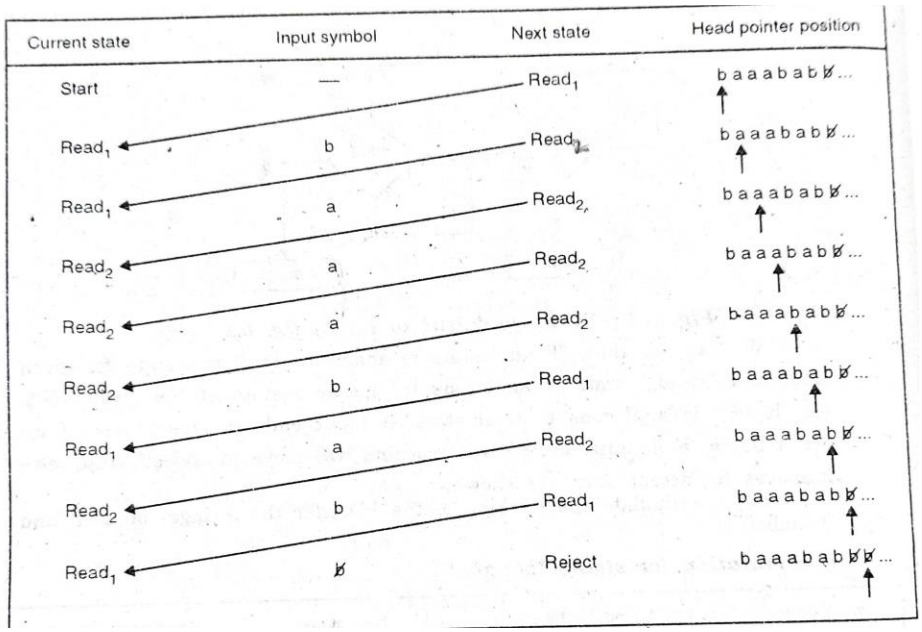
Simulation for string bbaaba.



Current state	Input symbol	Next state	Head pointer position
Start	—	$Read_1$	b b a a b a ⊬ ...
$Read_1$	b	$Read_1$	b b a a b a ⊬ ...
$Read_1$	b	$Read_2$	b b a a b a ⊬ ...
$Read_1$	a	$Read_2$	b b a a b a ⊬ ...
$Read_2$	a	$Read_2$	b b a a b a ⊬ ...
$Read_2$	b	$Read_1$	b b a a b a ⊬ ...
$Read_1$	a	$Read_2$	b b a a b a ⊬ ...
$Read_2$	⊬	Accept	b b a a b a ⊬ ⊬ ...

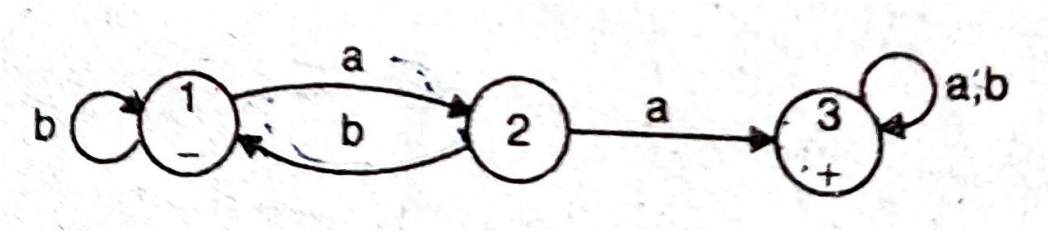
Thus the string 'bbaaba' is accepted by the designed PDA

Simulation of string baaabab



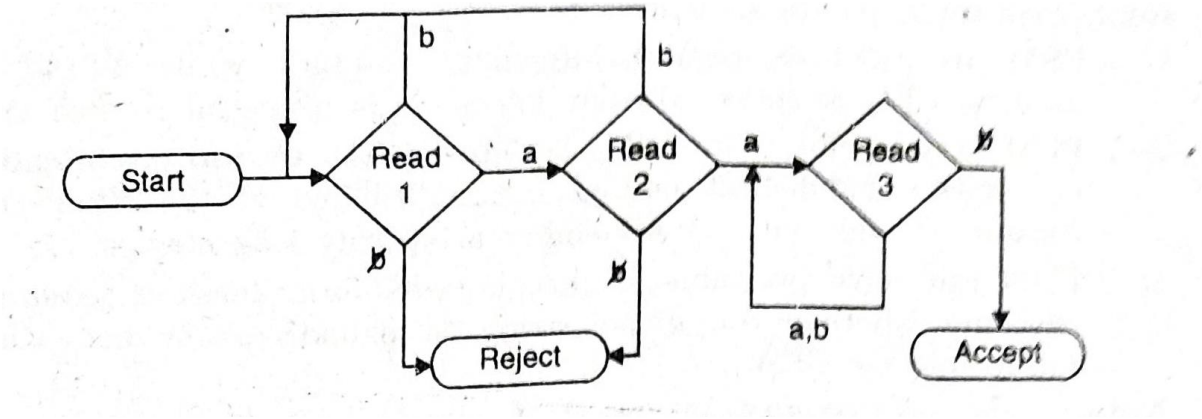
Current state	Input symbol	Next state	Head pointer position
Start	—	$Read_1$	b a a a b a b ⊬ ...
$Read_1$	b	$Read_1$	b a a a b a b ⊬ ...
$Read_1$	a	$Read_2$	b a a a b a b ⊬ ...
$Read_2$	a	$Read_2$	b a a a b a b ⊬ ...
$Read_2$	a	$Read_2$	b a a a b a b ⊬ ...
$Read_2$	b	$Read_1$	b a a a b a b ⊬ ...
$Read_1$	a	$Read_2$	b a a a b a b ⊬ ...
$Read_2$	b	$Read_1$	b a a a b a b ⊬ ...
$Read_1$	⊬	Reject	b a a a b a b ⊬ ⊬ ...

Q.2 Construct PDA recognizing the language accepted by the DFA given in the following figure.



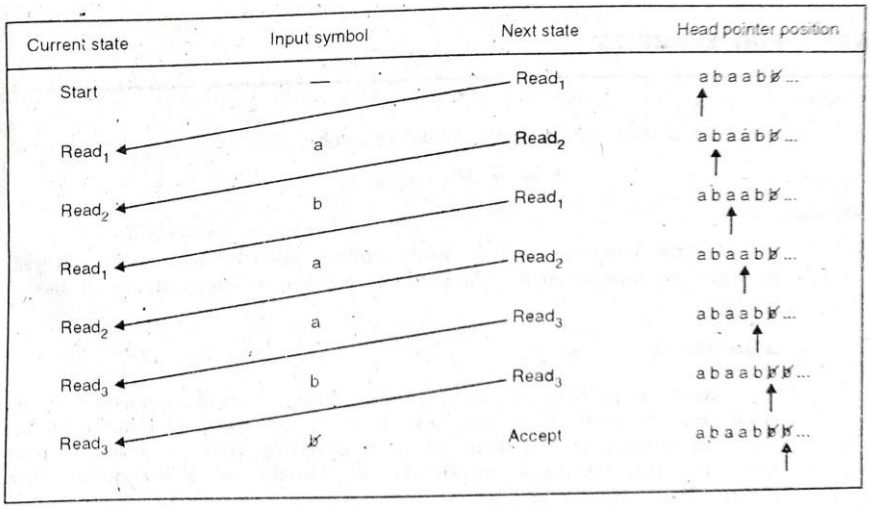
An example DFA

PDA can be constructed as shown in above diagram. We can see that state 'Read$_1$' of PDA is analog our to state '1' at given DFA. Similarly, 'Read$_2$' is analogous to state '2' and 'Read$_3$' is with state '3' of the given DFA. Obviously 'Read$_1$' and 'Read$_2$' are non-final states and reading blank 'b' indicating end of input string in that state causes machine to move to the reject indicating the input string given is rejected by the PDA.



PDA equivalent to DFA in above figure.

Let us simulate the working of the PDA for input given as "abaab" The acceptance of the input can be shown in diagram:



Thus designed PDA is accepting the given string

For construction for PDA as above accepting regular languages, it is observed that no stack is being used and therefore the design is not containing any 'push' or 'pop' states. As we have not used stack in the above designs for PDA's we can say that above are the finite automata represented using the notations. Thus FA is nothing but a special case of PDA and hence we can said that it is less powerful than PDA.

5.2.4Construction of PDA:

From the meaning of PDA any transition function of PDA is defined as

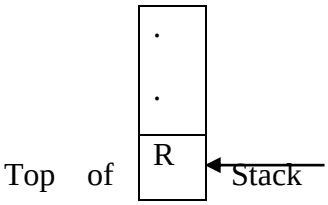
$$(Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$$

We define 'δ' transition function since q_0 as start state.

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

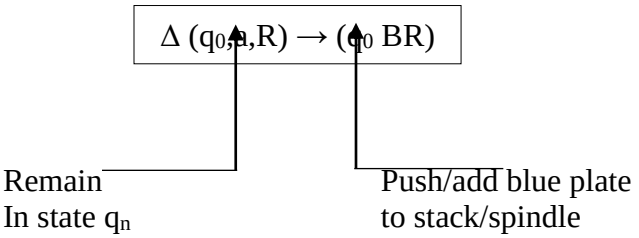
$T = \{R, B\}$

i. At first we are in state q_0 initial stack symbol is R,i.e.



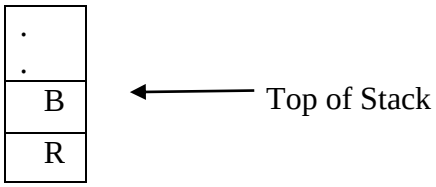
$\therefore \delta (q_0, \Sigma R)$ here Σ can be a, b, or ϵ.

Let the string is aaabbb.
From the above conversation, we write 3 transitions for q_o as start state and R on early stack symbol.
∴



Here B becomes new top of stack.

i.e.



$\delta(q_0, b, R)$ → (Step v)
$\delta(q_0, \epsilon, R)$ → (Step vi)

ii. After we read 1st 'a' we don't change the state, as we have to read next all 'a's and add 'B' to stack. The variation between 1st 'a' and remaining all 'a's can be made by observing at top of stack (TOS)

For 1st 'a' - TOS is R

For residual 'a's - TOS is B

∴ We write transition for residual a's as

$$\delta(q_o, a, B) \rightarrow (q_0,$$

i.e. we add 1'B' to stack for each residual 'a'.

iii. Now in the same state, if 'b' befalls on input tape i.e. the situation is

$\delta(q_0, b, B)$

Then we have to pop one 'B' plate from stack which matches the current 'b' with beforeadded 1 'a' i.e. (q_1, ϵ).

∴ The transition becomes

$$\delta(q_o, b, B) \rightarrow (q_1, \epsilon)$$

Specifies POP the

Symbol from stack

Here we have to change the state because we have to make difference that accepting numberof 'a's in loop and accepting first 'b'.

If we stay in the same state, i.e., in q_o, then in q_o we have the transition

$\delta(q_o, a, B)$ and (q_0, b, B)

Both will transit to q_o. This will lead to taking of the strings a*b*a* ….. Which leads toaccepting invalid strings. So we distinguish it by altering the state to q_1.

iv. If at go present state, and B on TOS if input string has 'e', it is error state.

See step (vi)

i.e. $\delta (q_0, \epsilon, B) \rightarrow$ ERROR state

v. In state q_1, probable i/ps are a, b, Δ and TOS is B.

a. i.e. $\delta (q_1, a, B) \rightarrow$ ERROR state

b. $\delta (q_1, b, B) \rightarrow (q_1, \epsilon)$

Here we pop all 'B's for all 'b's occurring in input tape and be in self loop as the same actionis to be recurrent for (n-1) times so no need to change the state.

c. $\delta (q_1, \epsilon, B) \rightarrow$ ERROR

vi. After step (v) is over, we may have the condition that current state is q_1 top of stack is R andinput symbols can be a, b, or ϵ

a. If (step i) $\delta (q_1, \epsilon, R)$ Accept

b. If $\delta(q_1, b, R) \rightarrow$ ERROR

c. if $\delta(q_1, a, R) \rightarrow$ ERROR

So from (i) to (vi) we write whole PDA as

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

Where, δ is

$\delta (q_0, a, R) = (q_0, BR)$, $\delta (q_0, b, R) =$ ERROR,

$\delta (q_o, \epsilon, R)$ ERROR, $\delta (q_o, a, B) = (q_0, BB)$,

$\delta (q_o, b, B)$ (91, E), $\delta (q_o, \epsilon, B) =$ ERROR,

$\delta (q_1, a, B)$ ERROR, $\delta (q_1, b, B) = (q_1, \epsilon)$,

$\delta (q_1, \epsilon, B)$ ERROR, $\delta (q_1, \epsilon, R) =$ ACCEPT,

$\delta (q_1, b, R)$ ERROR, $\delta (q_1, a, R) =$ ERROR

1. Construct PDA for the language

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

Solution:

The language set for L

L ={ε, ab, aabb, aaabbb....)

This language is same as $L = \{a^n b^n \mid n \geq 1\}$

In example 2 except the extra string ϵ in this example.

So the same 8 can be written as in 2 example except the transition which accepts 'ϵ' as a string andgoes to error state in 2 example, so in its place that transition, we have to write the transition as

$$\delta (q_0, \epsilon, R) \rightarrow \text{Accept state}$$

∴ PDA can be given as

$M = (\{q_o, q_1\}, \{a, b, c\}, \delta, q_0\{q_1\})$

Where, & is

$\delta (q_0, a, R) =$ (q_0, BR), $\delta (q_0, b, R) =$ ERROR,

$\delta (q_0, \epsilon, R) =$ Accept, $\delta (q_0, a, B) =$ (q_0, BB),

$\delta (q_0, b, B) =$ (q_1, ϵ), $\delta (q_0, \epsilon, B) =$ ERROR,

$\delta (q, a, B) =$ ERROR, $\delta (q_1, b, B) =$ (q_1, ϵ),

$\delta (q_1, \epsilon, B) =$ ERROR, $\delta (q_1, \epsilon, R) =$ ACCEPT,

$\delta (q_1, b, R) =$ ERROR, $\delta (q_1, a, R) =$ ERROR

The Languages of PDA (Construction ofPDA using empty stack and final statemethod). We have expected that a PDA receives its input by consuming it andentering an accepting state. We call this method "acceptance byfinal state".

There is a second method to defining the language of a PDA, we accepted by PDAcall it. Language "accepted by empty stack", i.e., the set of stringsthat cause the PDA to empty its stack, initial from early ID.

These two methods are equal, in the sense that the language L has a PDA that receives it by finalstate if and only if L has a PDA that receives it by empty stack.

However for a given PDA P, the languages that P accept by final state and by empty stack areusually different.

5.3 ACCEPTANCE BY PDA

PDA accepts its input by consuming it and entering an accepting state. We call this approach "acceptance by final state".There is a second approach to defining the language of PDA, we call it. Language "accepted by empty stack", i.e. the set of strings that cause the PDA to empty its stack, starting from initial ID.

5.3.1 PDA acceptance by Final State:

The way of defining a language accepted is similar to the way a finite automaton receives inputs. Thatis, we designate some states as final states and define the accepted language as the set of all inputsfor which some choice of moves causes the Pushdown automaton to enter a final state,

Formal Definition: Language acceptance by Final State

Let P = (Q, Σ,Γ, δ, q_0, Z_0, F) be a PDA then L(P), the language accepted by P by final state is

$\{w \mid (q_0, w, Z_0) \quad * \quad (q, \epsilon, \alpha)\} \vdash\!\!\!\!- \quad P$

For some state q in F and any stack string a. That is, starting in the new ID with w waiting on theinput, P consumes w from the input and enters an accepting state. The contents of the stack at thattime are irrelevant.

Examples:

1. Let P to accept the language L_{ww^R}.

The language set

L = {00, 11, 0110, 011110, 1001, 110011, 101101,}

The PDA is described as

P = ({q_0, q_1, q_2}, {0,1}, {0, 1, z_o}, δ, q_0, z_0, {q_2}).

Where, 8 is defined as

$\delta(q_o, 0, Z_0) \rightarrow (q_o, 0Z_0)$, $\delta(q_o, 1, Z_0) \rightarrow (q_0, 1Z_0)$,

$\delta(q_o, 0, 0) \rightarrow (q_o,00)$, $\delta(q_o,0,1) \rightarrow (q_o,01)$,

$\delta(q_o, 1, 0) \rightarrow (q_o, 10)$, $\delta(q_o, 1, 1) \rightarrow (q_0, 11)$,

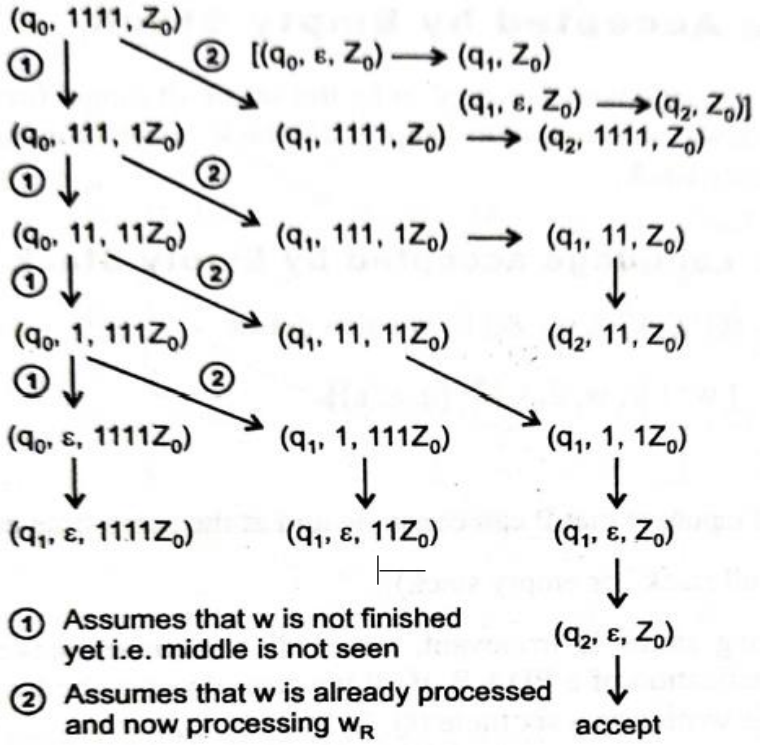
$\delta(q_o, \epsilon, Z_0) \rightarrow (q_1, Z_0)$, $\delta(q_o, \epsilon, 0) \rightarrow (q:,0)$,

$\delta(q_o, \epsilon, 1) \rightarrow (q, 1)$, $\delta(q_0, 0, 0) \rightarrow (q_1, \epsilon)$,

$\delta(q_1, 1, 1) \rightarrow (q_1, \epsilon)$, $\delta(q_1, \epsilon, Z_0) \rightarrow (q_2, Z_0)$

and accept

Let the string be 1111 where, w = 1,w_R= 1



5.3.2 PDA acceptance by Empty Stack:

To define the language known to be the set of all inputs for which some order ofmoves causes the pushdown automaton to empty its stack. This language is referred to as thelanguage recognized by empty stack.

Formal Definition: Language accepted by Empty Stack

For each PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ we also define

$N(P) = \{w \mid (q_0, w, Z_0) * (q, \epsilon, \epsilon)\}$ ⊢

for any state q.

That is, N(P) is the set of inputs w that P can consume and at the same time empty its stack.

(N in N(P) stands for “null stack" or empty stack).

Since the set of accepting states is unconnected, we shall sometimes leave off the last (seventh)component from the specification of a PDA P, if all we care about is the language that P accepts byempty stack. Thus the P is written as a six tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$

Here no constraint is placed on the uncertain state q. When acceptance is defined by empty stack, it isessential to require at least one transition to permit the acceptance of languages that do not comprisethe null string.

Examples:

1. Consider the language in above example and consider the PDA in thatexample.This example never empties the stack.

$\therefore N(P)\phi$.

$\therefore$ If we modify P to accept L_{wwr} by empty stack as well as by final state.

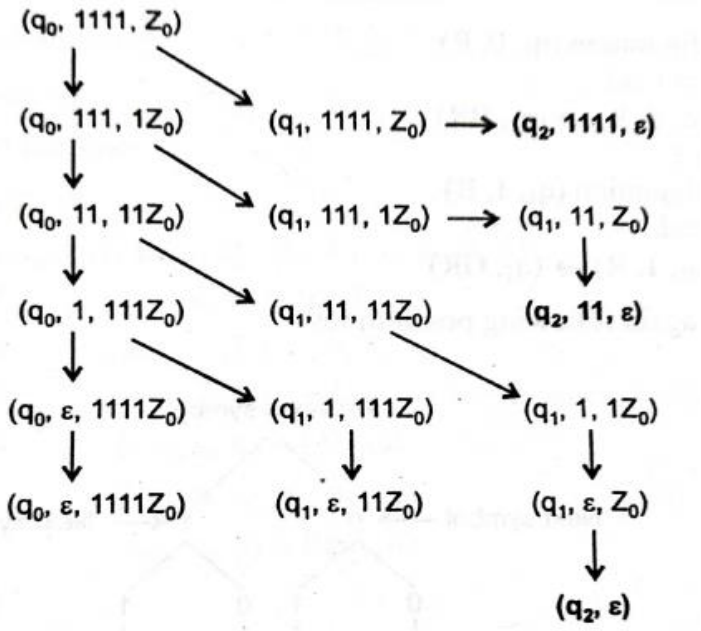
$\therefore$ Instead of transition

$\delta\ (q_1, \epsilon, Z_0)\ (q_2, Z_0)$ we add

$$\delta\ (q_1, \epsilon, Z_0) = (q_2, \epsilon)$$

Now P pops the last symbol off its stack as it accepts and L(P) = N(P) = L_{wwr}.

Consider the same string 1111.



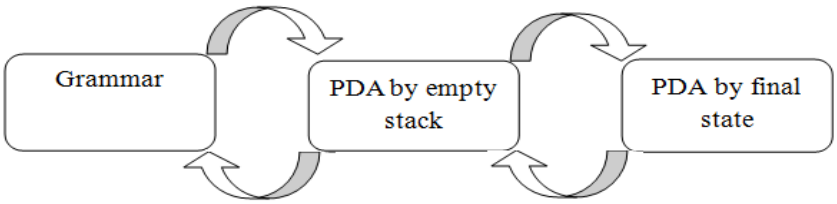
The Stack is end at here.

5.4 PDA AND CFG

Context-free languages are languages defined by PDA's. The following 3 classes of languages all are of same class:

i. Context Free Languages (CFL)
ii. The languages which are accepted in the final state by some PDA.

iii. The languages which are accepted in empty stack by some PDA.



Figure

Conversion of a Context FreeLanguage (in GNF) to Push down Automata (PDA)

Theorem:

If L is a context free language, then we can construct a PDA a accepting L by empty stack

i.e. L = N(A).

We can construct A by making use of Productions in G.

Step 1: Construction of A

Let L = L(G), where G = (V_N ,∑, P, S) is a CFG. In GNF we construct PDA A as

$$A = (\{q\}, \sum, V_N \cup \sum, \delta, q, S, \phi)$$

where, transition functionδ is defined by the following rules:

R1: δ(q, a, A) = {(q, α) A →aa is in P}

R2: δ(q, a, A) = {(q, ε)} for every A → a in ∑

The PDA can be constructed as:

1. The Pushdown symbols in A are variables and terminals.
2. If the PDA reads a variable A on the top of PDS, it makes 'a' move by placing the RHS of any
3. ε -Production (after erasing A).
4. If the PDA reads a terminal a on PDS and if it matches with the current input symbol, then the
5. PDA erase a.
6. In other cases the PDA halts.
7. If w ∈ L(G) is obtained by a left most derivation,

$$S \Rightarrow u_1 A_1 \alpha_1 \Rightarrow u_1 u_2 A_2 \alpha_2 \alpha_1 \Rightarrow \ldots. \Rightarrow w,$$

ThenA can empty the PDS on application ofi/p string w. The first move of A is by a ε -move corresponding to S →u_1 A_1 α_1. The PDA erasers S and stores u_1 A_1 α_1, then using R2, the PDA erasers the symbols in u_l by processing a prefix of w. Now the topmost symbol in PDS is A_1.

Once again by applying the ε -move corresponding to A1 → u_2A_2 α_2, the PDA erases A_2 and stores u_2A_2 α_2 above α_1. Proceeding in this way, thePDS empties the PDS by processing the entire string w.

Example 1:

Construct a PDA A equivalent to the following context free grammar:

S → OBB, B → OS | 1 | 0.

Test whether 010^4 is in N(A).

Solution:

A is defined as:

({q}, {0, 1}, {S, B, 0, 1}, δ, q, S, ϕ)

The transition functionδ is

R_1: δ(q, 0, S) = (q, BB)

R_2: δ(q, 0, B) = {(q, 0S), (q, ε)} and δ(q, 1, B) → (q, S)

R3: δ (q, 0, B) = {(q, ε)}

$(q_0, 010^4, S)$ ⊢ $(q, 010^4, BB)$ by Rule R_1

⊢ $(q, 10^4, BB)$ by Rule R_3

⊢ $(q, 10^4, SB)$ by Rule R_2 since

(q, 1S) ∈**δ** (q, ε, B)

⊢ $(q, 0^4, SB)$ by Rule R_4

⊢ $(q, 0^4, BBB)$ by Rule R_1

⊢ $(q, 0^3, BBB)$ by Rule R_3

⊢* $(q, 0^3, 000)$ by Rule R_3 as (q, 0) ∈**δ** (q, ε, B)

⊢* (q, ε, ε)

Example 2:

Convert the following CFG to a PDA

S → aAA

A → aS|bS|a

Solution:

The PDA P = (Q, Σ, Γ, δ, q_0, Z_0, F) is defined as

Q = {q}

Σ = {a, b}

Γ = {a, b, S, A}

q_0 = q

Z_0 = S

F = {}

And the transition function is defined as:

δ(q, ∈, S) = {(q, aAA)}

δ(q, ∈, I) = {(q, aS),(q, bS),(q, a)}

δ(q, a, a) = {(q, ∈)}

δ(q, b, b) = {(q, ∈)}

5.5 SUMMARY:

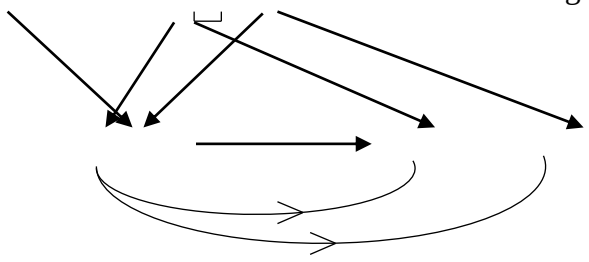
- PDA is mathematical model of PDM (Pushdown Memory - Stack Machine).
- The PDA is accepter for context free languages.
- PDA have 7 tuple - (Q, Σ, Γ, δ, q_0, Z_0, {F}) where, δ: Q x (Σ U({ϵ} x Γ→ Q x Γ*)
- Languages accepted by PDA

a) PDA accepting Final State

b) PDA accepting empty stack (final state isϕ)

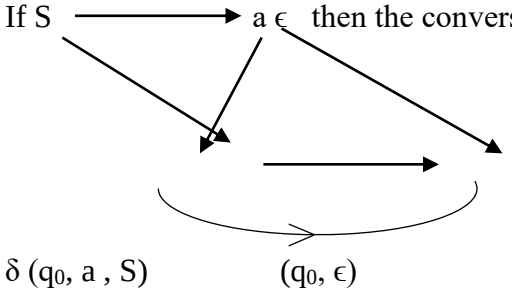
- Changing CFG (in GNF) to PDA. Simplified steps are

If S ⟶ aSB/aB then the conversion is given as



δ (q_0, a , S) {(q_0, SB), (q_0 ,B)}

If S $\rightarrow$ a ϵ then the conversion is given as



$\delta (q_0, a , S)$ (q_0, ϵ)

5.6 REFERENCES:

1. Theory of Computer Science, K. L. P Mishra, Chandrasekharan, PHI,3rd Edition
2. Introduction to Computer Theory, Daniel Cohen, Wiley,2nd Edition
3. Introductory Theory of Computer Science, E.V. Krishnamurthy, Affiliated East-West Press.
4. Introduction to Languages and the Theory of Computation, John E Martin, McGraw-Hill Education.

5.7 REVIEW QUESTIONS:

1. Define PDA.
2. Differentiate between FA and PDA.
3. Construct a PDA for language L = $\{0^n 1^m 2^m \mid n, m \geq 1\}$.
4. Construct a PDA for L =$\{a^n b^{2n} c^n \mid n \geq 1\}$.

Unit III

6

LINEAR BOUND AUTOMATA

Unit Structure

6.0 OBJECTIVES

This chapter would make you to understand the following concepts:

- To study the concept of Linear Bound Automata.
- To learn Linear Bound Automata Model.
- To Study Linear Bound Automata and Languages.

6.1 INTRODUCTION

An Automata is an abstract computing device or machine, help to check weather a string is belonging to the language or not. Theory of computation contains Finite Automata (FA), Push Down Automata (PDA), Linear Bound Automata and(LBA) and Turing Machine(TM).

Finite Automata use to recognize regular language. It is mathematical model with discrete inputs, outputs, states and set of transition functions from one state to another state over a input symbols from alphabet $\sum$. Push DownAutomata work like Finite Automata with additional stack. It is powerful than FA. PDA is used to implement Context Free Grammar. It has more memory than FA. Linear Bound Automata (LBA) accepts 'Context Sensitive Grammar (CSG)' known as 'Type-1' Grammar. LBA is a Turing Machine with limited size tape. Is powerful than PDA but less powerful as compare to Turing Machine.

6.2 LINEAR BOUND AUTOMATA

Linear bounded automata (LBA) are accepts **context-sensitive languages**. In LBA computation is restricted to the constant bounded area. It has limited size input output tape. This tape is limiting based on input size we are bounding the tape using two end markers i.e. left end marker M_L and right end markerM_R

which assure the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape. It is a restricted form of TM in which input tape is finite. In terms of computational capability FA < PDA < LBA < TM. There are two types in each of FA, PDA and TM which deterministic and non-deterministic. But in LBA there is no such classification.

Halting Problem

The halting problem is solvable for linear bounded automata.

Halt (LBA) = {< M,w > |M is an LBA and M halts on w} is decidable.

An LBA that stops on input w must stop in at most $\alpha(|w|)$ steps.

Membership problem

The membership problem is solvable for linear bounded automata.

A(LBA) = {< M, w > |M is an LBA and M accepts w} is decidable.

Emptiness Problem

The emptiness problem is unsolvable for linear bounded automata. For every Turing machine there is a linear bounded automaton which accepts the set of strings which are valid halting computations for the Turing machine.

Definition: A Non-deterministic Turing Machine with a finite length tape space fill by the input is called Linear Bound Automata(LBA).

LBA is defined as :

$$M=(Q,\sum, \Gamma,\delta,q_{0,}M_L,M_R,,F)$$

Where,

Q is a nonempty finite set of states

$\sum \subseteq \Gamma$ is a finite set of input alphabets

Γ is the finite set of input tape alphabets

δ is transition function

$q_0 \in Q$ is initial state

$M_L$is left end marker

M_R is right end marker

¸F∈ Q is finite set of final states

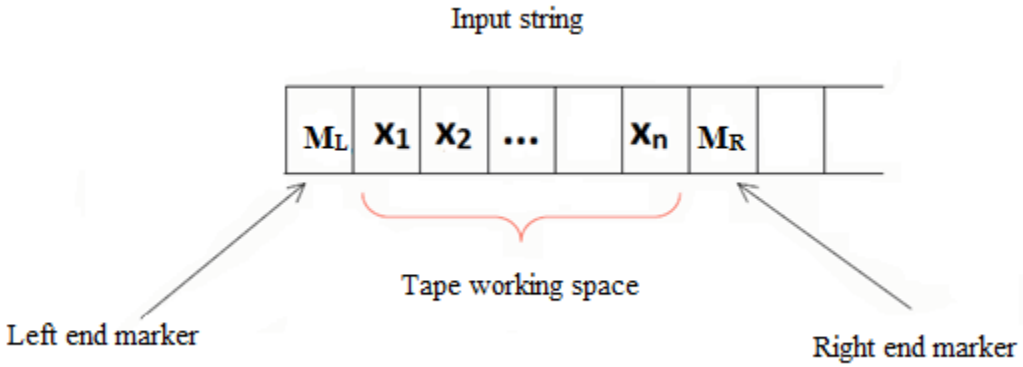


Figure 6.1

M_L and M_R are boundaries for a tape. M_L is entered in leftmost end of the input tape and avoids the Read/Write head from getting off the left end of the tape. M_R is entered in rightmost end of the input tape and avoids the Read/Write head from getting off the right end of the tape. Both end markers should be at their respective ends, and Read/Write head should not write any other symbol over both end-markers.

Examples:

1) $\{a^n b^n c^n / n \geq 1\}$

2) $\{ww / w \in \{a, b\}^+\}$

LBAs and CSLs

The development stages of LBA:

Myhillin 1960 considered deterministic LBAs.

In Landweber 1963 showed that they produce only context-sensitive languages.

And Kuroda in 1964 generalized to nondeterministic LBAs and showed that this produces precisely the context-sensitive languages.

Theorem 1 (Landweber-Kuroda)

'A language is accepted by an LBA iff it is context sensitive.

Proof :

If L is a CSL, then L is accepted by some LBA.

Let G = (N, Σ, S, P) be the given grammar such that L(G) = L.

Construct LBA M with tape alphabet Σ × {N ∪ Σ}(2- track machine)

First track holds input string w.

Second track holds a sentential form α of G, initialized to S

If w = , M halts without accepting.

Repeat :

- Non-deterministically select a position i in α.
- Non-deterministically select a production β → γ of G.
- If β appears beginning in position i of α, replace β by γ there. If the sentential form is longer than w, LBA halts without accepting.
- Compare the resulting sentential form with w on track 1. If they match, accept. If not go to step 1.

Theorem 2

If there is a linear bounded automaton M accepting the language L, then there is a context sensitive grammar generating L − {ε}.

Proof :

Derivation imitate moves of LBA

Three types of productions

- Productions that can generate two copies of a string in Σ^* , along with some symbols that work as markers to keep the two copies separate.
- Productions that can replicate a sequence of moves of M. During this portion of a derivation, one of the two copies of the original string is left unchanged; the other, representing the input tape to M, is modified accordingly.
- Productions that can erase everything but the unmodified copy of the string, provided that the simulated moves of M applied to the other copy cause M to accept.

Linearly bounded memory machine:

The linearly bounded memory machine is similar to a Turing machine, except that it has, instead of a potentially infinite tape forcomputation, only the portion of the tape containing the input string plus two squares on the tape to hold the end markers. Such a restriction reduces the machine's

power to recognize certain types of strings. It has been shown that even when the length of the tape is increased as a linear function of the length of the input string, the computational ability of the machine remains the same. Hence, such a machine is called a linearly bounded memory machine. It recognizes a class of languages known as context-sensitive languages.

6.3 THE LINEAR BOUND AUTOMATA MODEL

This model is important because (a) the set of context-sensitive languages is accepted by the model and (b) the infinite storage is restricted in size but not in accessibility to the storage in comparison with the Turing machine model. It is called the *linear bounded automaton* (LBA) because a linear function is used to restrict (to bound) the length of the tape.

In this section we define the model of linear bounded automaton and develop the relation between the linear bounded automata and context-sensitive languages. It should be noted that the study of context-sensitive languages is important from practical point of view because many compiler languages lie between context-sensitive and context-free languages.

A linear bounded automaton is a non-deterministic Turing machine which has a single tape whose length is not infinite but bounded by a linear function of the length of the input string.

Originally Linear Bound Automata were developed as models for actual computers not as computational process models. They are very important in computation theory. LBA is a multitrack non-deterministic Turing Machine with only one tape and having the length same as input string.

Example:

a. Consider a input string w, where |w| = n-1.

b. If the input string w is recognized by an LBA if it is also be recognized by a Turing machine using no more than k_n cells of input tape, where k is a constant specified in the description of LBA.
c. 'k' is a property of the machine; value of k does not depend on the input string.
d. For processing a string in LBA, the string must be enclosed in M_L and M_R.
e. The model of LBA contains two tapes:
   i) Input tape: On input tape the head never prints and it just move only in right direction, never moves left.
   ii) Working tape: On working tape head modify the contents of working tape, without any restriction.

ID of LBA

In LBA, the ID is denoted by (q, w, k) where q ∈ Q, w ∈ Γ and k is some integer between 1 and n.The transition of the IDs is similar except that if k changes to (k – 1), then Read/Write head moves to the left and if move to (k + 1) then head moves to the right.

Languages accepted by LBA is:

The language accepted by LBA is defined as the set :

$\{w \in \{\sum - \{M_L, M_R\}\}^* \mid (q_0, M_L, w, M_R, 1) \overset{*}{\vdash} (q, \alpha, i)$ for some $q \in F$ and for some integer i between 1 and n.

6.4 LINEAR BOUND AUTOMATA AND LANGUAGES

A string 'w' is accepted bylinear bounded automaton M if,

- First it start at initial state with Read/Write head reading the left end marker (M_L), M halt over the right-end marker (M_R) in final state, otherwise w is rejected.
- The production rules for the generative grammar are constructed as in the case of TM. The following additional productions are needed in the case of LBA:

  $a_i q_f M_R \rightarrow q_f M_R$, for all $a_i \in \Gamma$

  $M_L\ q_f M_R \rightarrow M_L q_R$, $M_L\ q_f \rightarrow q_f$

  The class of recursive languages does not show up explicitly in below Table, because there is no known way to characterize these languages using grammars.

Relation between LBA and Context-SensitiveLanguages

The set of strings accepted byLBA(non-deterministic TM) is the set of strings generatedby the context-sensitive grammars, excluding the null strings, Now we can conclude:

'If L is a context-sensitive language, then L is accepted by a linear bounded automaton. The converse is also true.'

The construction and the proof are similar to those for Turing machines with some modifications.

The Chomsky Hierarchy

Languages Form of Productions Type (Grammars) in Grammar Device	Accepting
3 Regular $A \rightarrow aB, A \rightarrow \lambda$ ($A,B \in V$, $a \in \sum$)	Finite automaton

2	Context-free Pushdown $(A \in V, \alpha \in (V \cup \sum)^*)$	$A \rightarrow \alpha$ automaton	
1	Context-sensitive $(\alpha, \beta \in (V \cup \sum)^*, \lvert\beta\rvert \geq \lvert\alpha\rvert,$ αcontains a variable)	$\alpha \rightarrow \beta$ automaton	Linear-bounded
0	Recursively enumerable$(\alpha, \beta \in (V \cup \sum)$ (unrestricted)	$\alpha \rightarrow \beta$ Turing machine α contains a variable)	

Example

Example 1: Construct an LBA for $\{a^n b^n c^n \mid n \geq 1\}$

Solution:

The tape alphabet for an LBA,is finite, but it may be considerably bigger than the input alphabet. So we can store more information on a tape than just an input string or related sentential form.

Consider following,

let, Γ = {a, b, c, **a, b, c**}, and Σ = {a, b, c}. Occurrences of bold letters can serve as markers for positions of interest.

To test whether an input string has the form $a^n b^n c^n$

1) Scan string to ensure it has form $a^k b^m c^n$for k, m, n ≥ 1.

Along the way, mark leftmost a, b, c.like**a**aa**b**bb**c**cc.

2) Scan string again to see if it's the **rightmost** a, b, c that are marked.

If yes for all three, ACCEPT.

If yes for some but not all of a, b, c, REJECT.

3) Scan string again moving the 'markers' one position to the right.

Like **a**aa**b**bb**c**cc becomes a**a**ab**b**bc**c**c. Then Go to 2. All this can be done by a

LBA.

Example 2. Give an LBA that accepts the language $\{a^ib^ic^i|i \in \mathbb{N}\}$.

Solution:

Logic:

- The automaton rewrites the first a to A, and changes its state, looks for the first b.
- The automaton rewrites the first b to B, and changes its state, looks for the first c.
- The automaton rewrites the first c to C, and changes its state, looks (backward) for the first a.
- The capital letters A,B,C are read without changing them.
- The above movements are repeated.
- If finally only capital letters remain between the border ♯ signs, then the automaton accepts (the input).

Formally, letM=$(Q,\sum, \Gamma,\delta,q_{0,}M_L,M_R,,F)$

LBA = $(\{q_0, q_1, q_2, q_3, q_4, q_f\}, \{a,b,c\}, \{a,b,c,A,B,C\},\delta, q_0, M_L, ,M_R, \{q_f\})$

be a deterministic LBA, where δ consists of the next transitions:

1. $\delta (q_0, M_R) = (q_f, M_R, Halt)$ – the empty word is accepted by LBA.
2. $\delta (q_0, a) = (q_1, A, Right)$ – the first (leftmost) a is rewritten to A and LBA changes its state.
3. $\delta (q_0, B) = (q_0, B, Left)$ – the capital letters B and C are skipped in state q_0,
4. $\delta (q_0, C) = (q_0, C, Left)$ – by moving the head to the left.
5. $\delta (q_1, a) = (q_1, a, Right)$ – letter a is skipped in state q_1 to the right.
6. $\delta (q_1, B) = (q_1, B, Right)$ – capital B is also skipped.
7. $\delta (q_1, b) = (q_2, B, Right)$ – the leftmost b is rewritten by B and the state becomes q_2.
8. $\delta (q_2, b) = (q_2, b, Right)$ – letter b is skipped in state q_2 to the right.
9. $\delta (q_2, C) = (q_2, C, Right)$ – capital C is also skipped in this state.
10. $\delta (q_2, c) = (q_3, C, Left)$ – the leftmost c is rewritten by C and LBA changes its state to q_3.
11. $\delta (q_3, a) = (q_3, a, Left)$ – letters a,b are skipped in state q_3

12. $\delta (q_3, b) = (q_3, b, Left)$ – by moving the head of the automaton to the left.

13. $\delta (q_3, C) = (q_3, C, Left)$ – capital letters C,B are skipped in state q_3

14. $\delta (q_3, B) = (q_3, B, Left)$ – by moving the head of the automaton to the left.

15. $\delta (q_0, A) = (q_3, A, Right)$ – the head is positioned after the last A and the state is changed to q_0.

16. $\delta (q_4, B) = (q_3, B, Right)$ – if there is a B after the last A the state is changed to q_4.

17. $\delta (q_4, B) = (q_4, B, Right)$ – in state q_4 capital letters B and C are skipped

18. $\delta (q_4, C) = (q_4, C, Right)$ – by moving the head to the right.

19. $\delta (q_f, M_R) = (q_f, M_R, Accept)$ – if in q_4 there were only capital letters on the tape, LBA accepts.

6.5 REVIEW QUESTIONS

1. Define Linear Bound Automata.
2. Write note on ID of LBA.
3. Which type of language is accepted by Linear Bound Automata?
4. Justify. Is the language accepted by LBA is accept by Turing Machine.

6.6 SUMMARY

1. Linear Bound Automata design to accept context-sensitive languages.
2. End markers (M_L and M_R) is the safety feature of LBA.

6.7 REFERENCES

1) John Martin," Introduction to Languages and the Theory of Computation", Tata McGraw-Hill, Third Edition.

2) Introduction to Languages and The Theory of Computation, Fourth Edition by John C. Martin

3) Theory of computer science Automata, languages and computation, Third edition by K.L.P. Mishra and N. Chandrasekaran

❖❖❖❖

7

TURING MACHINES

Unit Structure

7.0 OBJECTIVES

This chapter would make you to understand the following concepts:

- To study Turing Machines.
- Learn to design Turing Machine and it’s representation.
- Turing Machine construction.

7.1 INTRODUCTION

Turing machine (TM) was invented by Alan Turing in Turing 1936, which is big achievement in the field of finite-state computing machines. Initially was specially design for the computing of real numbers.TM is very powerful than Linear Bound Automata. Today this machine use as a foundational model for computability in theoretical computer science.TM can effectively represent its configuration using a simple notation like as ID's of PDA.TM has a infinite size tape, use to accept Recursive Enumerable Languagegenerated by Type-0 Grammar. The machine can read/ write and also move in both (left and write) directions. The machine producesdesired output based on its input.

Why Turing Machines?

- It isRobust model in the computation world.
- Equivalence with other such models of computation, with reasonableassumptions (e.g., only finite amount of work is possible in 1 step).
- Thus, though there are several computational models, the class of algorithms.
- They can do everything a computer can do and vice versa. But takes a lot more time. Is not practical and indeed its not what is implemented in today's computer.
- So then again, It is the top-most and powerful computational Model.

7.2 TURING MACHINE DEFINITION

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape and tape is divided into cells on which input is given. It has a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A Turing Machine can be formally described by 7-tuple $(Q, \sum, \Gamma, \delta, q_0, B, F)$ where −

- Q is a finite set of states
- $\sum$ is the finite set of input alphabets, $\sum \subseteq \Gamma$ and $B \notin \sum$
- Γ is tape input symbol
- q_0 is the initial state
- B is the blank symbol
- F is the set of final states

δ is a transition function; $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{Left, Right\}$.

i.e.value of $\delta(q_1\ x)$; if defined, is a triple(P,Y,D) where

i) P is the next state in Q.

ii) $Y \in \Gamma$, scanned and written in the cell, replacing whatever symbol is there

iii) D- direction; Left or Right, tell the moving direction for head.0

7.3 REPRESENTATIONS

Turing Machine can be presented using:

i) Instantaneous descriptions using move relations.

ii) Transition table.

iii) Transition diagram or transition graph.

i) Instantaneous descriptions using move relations: An ID of a TM is defined over entire input string and current state.

Definition: An ID of TM is a string α β γ, where β- current state of TM.

The αγ is a input string partition as:

α is substring of input string formed by the symbols available to the left of 'a', where 'a' is current symbol.

γ is the first symbol in the current symbol 'a' pointing by Read/Write head and γ has all the remaining symbols of input string.

Example: Consider following TM. Obtain its ID.

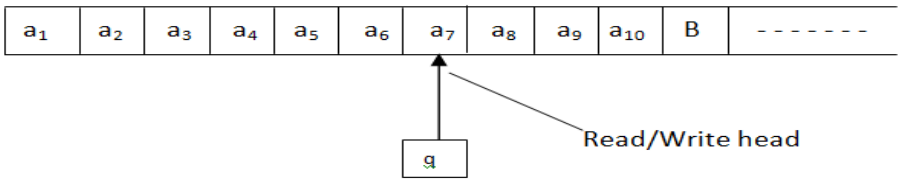


Figure 7.1

Current symbol under Read/Write head is a_7.

Suppose, current state: q_2

By definition of ID - α β γ, where β is state = q_2

∴$a_7$is written to the right of q_2 and

Symbol a_1 to a_6 to the left of q_4

∴ The ID is

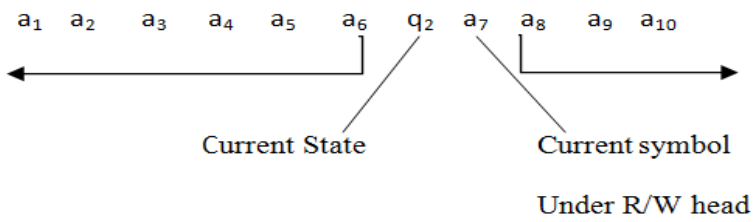


Figure 7.2

Moves in Turing machine

The $\delta(q, x_i)$ changes the ID of TM. This change is called move.

If $\delta(q, x_i) = (P, y, L)$

Take input string $x_1, x_2, \ldots X_n$.

Currentsymbol under Read/Write head- x_i

ID before processing symbol x_i

$x_1x_2\ldots\ldots x_{i-1}\ q\ \ x_{i+1}\ldots..X_n$

ID after processing symbol x_i

$x_1x_2\ldots\ldots\ x_{i-2}\ P\ x_{i-1}\ y\ \ x_{i+1}\ldots..X_n$

∴It is represented as

$x_1x_2\ldots\ldots x_{i-1}\ q\ \ x_i\ldots..X_n \vdash\ x_1\ldots\ x_{i-2}\ P\ x_{i-1}\ \ y\ \ x_{i+1}\ldots..X_n.$

If $\delta(q, x_i) = (P, y, R)$

Then the ID become:

$x_1x_2\ldots\ldots x_{i-1}\mathbf{q}\ \ x_i\ldots..X_n \vdash\ x_1x_2\ldots\ x_{i-1}\ \mathbf{y}\ \mathbf{P}\ x_{i+1}\ \ldots..X_n.$

Thus $I_i \vdash I_k$define the relationship between IDs.

$\vdash^*$ denotes reflexive-transitive clousure of relation $\vdash$.

∴ If $I_1 \vdash^* I_n$ then we split it as:

If $I_1 \vdash I_2 \vdash \ldots.. \vdash I_n$ for some IDs $I_2, \ldots I_{n-1}$

ii)Transition table:

a) The transition function δ

$$Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$$

States Q stored in table rows and table column shows each of the tape symbols Γ.

b) Each pair (Q , Γ) is represented by a triple (Q, Γ, {L,R}) as:

If $\delta(q_i, a) = (\alpha, \beta, \gamma)$ then we write (α, β, γ) under q_i^{th} row and a^{th} column.

In transition table we get entry as:

State q_i on input symbol 'a' goes to or changes to state ' γ ', by replacing the input symbol 'a' by ' α' and moves the tape head one cell in direction of ' β'.

iii)Representation by Transition Diagram

a) Every state implies to one vertex.

b) Transition of states represented by directed edges.

c) Each label of the edge is in the form (α, β, γ) where$\alpha, \beta \in \Gamma$, $\gamma \in \{L, R\}$.

Here, the directed edge from state q_i to q_j with label (α, β, γ) it indicates transition

$\delta(q_i, \alpha) = (q_i, \beta, \gamma)$

The symbol α will replaced with β and tape head moves to L , or R direction according to the value ofγ.

Every Edge in the transition diagram is represented by 5-tuple $(q_i, \alpha, \beta, \gamma, q_j)$.

7.4 ACCEPTABILITY BY TURING MACHINES

A language isaccept by Turing Machines if it enters into a final state for any input string w. If input not present in the language TM enters it into a rejecting state. A recursively enumerable language is which isgenerated by Type-0 grammar is accepted by a Turing machine. The set of languages accepted using a turing machine is often called Recursively Enumerable languages or RE languages. Recursive means for any number of times repeating the same set of rules and enumerable means a list of elements.

Formal Definition ofTuring Machines,

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

L(M) is the set of strings $w \in \sum^*$, such that $q_0 w \vdash \alpha\ p\ \beta$ for some state p in F and any tape strings α and β. The TM 'M' does not accept w if the machine M either halts in a non-accepting state.

Example : Design a TM to recognize all strings consisting of an odd number of α's.

Solution:

The Turing machine **M** can be constructed by the following moves −

- Let $\mathbf{q_0}$ be the initial state.
- If **M** is in $\mathbf{q_1}$; on scanning α, it enters the state $\mathbf{q_2}$ and writes **B** (blank).
- If **M** is in $\mathbf{q_2}$; on scanning α, it enters the state $\mathbf{q_1}$ and writes **B** (blank).
- From the above moves, we can see that **M** enters the state $\mathbf{q_1}$ if it scans an even number of α's, and it enters the state $\mathbf{q_2}$ if it scans an odd number of α's. Hence $\mathbf{q_2}$ is the only accepting state.

Hence,

M = {{q_1, q_2}, {1}, {1, B}, δ, q_0, B, {q_2}}

where δ is given by −

Tape alphabet symbol	Present State 'q_1'	Present State 'q_2'
A	BRq_2	BRq_1

7.5 DESIGNING AND DESCRIPTION OF TURING MACHINES

7.5.1 Turing Machine designing guidelines are:

i) Scan the symbol by Read / Write head to take the moving action(to move in Left/Right direction). The machine must remember the past scanned symbols. It can be remembering this by going to the next unique state.

ii) By changing the state if there is a change in the written symbol or when there is a change in the movement of Read/Write head we can minimize the number of states.

7.5.2 Description of Turing Machines:

i) At the beginning of the alphabet lower case lettersshows input symbols.

ii) Near the end of alphabet as....X, Y, Z, Capital letters are used for tape symbols that mayor may not be input symbols.

iii) B is generally represents Blank symbol.

iv) At the end of alphabet lower case letters are strings of input symbols.

v) Greek letters are used for strings of tape symbols.

vi) Letterslike q, p and nearby letters are states of machine.

7.6 TURING MACHINE CONSTRUCTION

Example 1: Construct a Turing Machine for language L = {$a^nb^nc^n$ | n≥1}**,**

where Language = {abc, aabbcc, Aaabbbccc, ……}

Solution: The language L = {$a^nb^nc^n$ | n≥1} represents a language in which we use only 3 character, i.e., a, b and c. At the beginning, language has some number of a's followed by equal number of b's and then followed by equal number of c's. Any such string which falls in this category will be accepted by this language.

i) We use 2 tape symbols X and Y for a and b respectively.

ii) We replace a by X. Move right replace first b found by Y. move right find last c replace it Blank `B'.

iii) Repeat step 2 until all a's, b's and c's. If no more a's, b's and c's are present – Accept otherwise Reject the string.

Replacement Logic:

- Mark 'a' then move right.
- Mark 'b' then move right
- Mark 'c' then move left
- Come to far left till we get 'X'
- Repeat above steps till all 'a', 'b' and 'c' are marked
- At last if everything is marked that means string is accepted.

Let's consider the string,

Direction aabbccB (String)

→ XaYbccB

← XaYbcBB

→ XYYcB

← XYYBB

→ XYYBA accept

Representation by Transition table method

	A	B	c	X	Y	Z
q_0	(q_1, X, R)	ERROR	ERROR		(q_5, Y, R)	
q_1	(q_1, a, R)	(q_2, Y, R)			(q_2, Y, R)	
q_2		(q_2, b R)	(q_2, c, R)			(q_3, B, L)
q_3			(q_4, B, L)			
q_4	(q_4, a, L)	(q_4, b, L)	(q_4, c, L)	(q_0, X, R)	(q_4, Y, L)	
q_5			ERROR		(q_5, Y, R)	(q_6, B, -)
q_6	Accept state					

7.7 VARIANTS OF TURING MACHINE

We have discussed about Turing Machines.Now let's see it's variants (types).

1) Multitrack Turing Machine:

Multitrack Turing Machine is a specific type of Multi-tape TM with only one unified tape head. It is equivalent to the standard Turing Machine and therefore accepts precisely the recursively enumerable languages. Standard Turing machines have k-tape, k heads move independently along k tracks. In ak-track Turing Machine, one head reads and writes on all tracks simultaneously one by one. A tape position in ak-track Turing Machine contains k symbols from the tape alphabet.

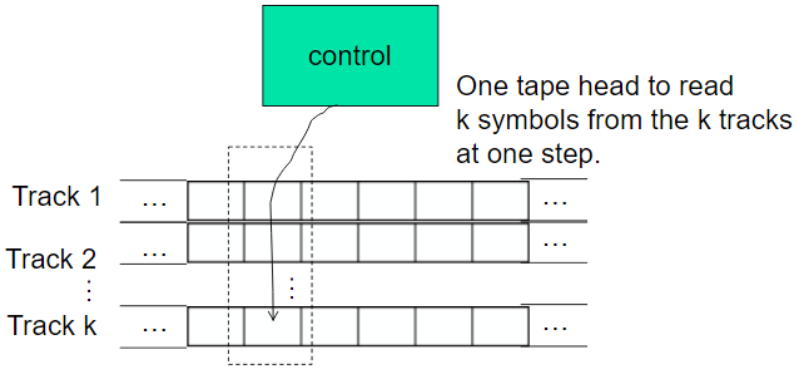


Figure 7.3

2) Two-way Turing Machine:

A two-way Turing Machine is an infinite tape with its input tape infinite in both directions, other components (Tuple) are same as that of the basic model.

Figure 7.4 (6.3)

Here the input string is a_1, a_2,a_3, ... a_n. Whenthe input string placed on tape, it is loaded with all blank symbols (B) to the left of a_1 and to the right of a_n. So if q_0 is the initial state of the TM, ID corresponding to the initial state will be q_0, a_1, a_2, s….. a_n,B.

3) MultitapeTuringMachine(MTT):

For every Multitape Turing Machine there is an equivalent single tape Turing Machine. In MTT number of tapes and read/write heads increased. All the indivisual tape will have there own respective tape heads. For this assumption is that all the tapes are two way infinite. It contains finite control with k number of heads.

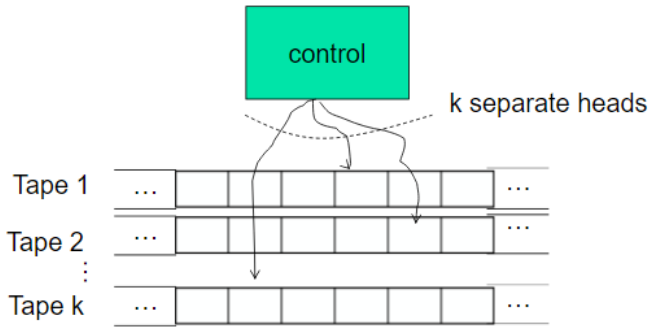


Figure 7.4

Depending upon the state of finite control, in single move the symbols scanned by each of the tape heads, the machine can

a) Change the state.

b) Print a new symbol on each of the cells scanned by its tape head.

c) Then independently move each tape head, one cell to the left or right or keep it stable.

4) Single tapeTuring Machine:

A Turing Machine consists of only one tape with infinite length on which read and write head can be performedoperation. The tape consists of infinite cells on which each cell either contains input symbol or a special symbol called blank (B).

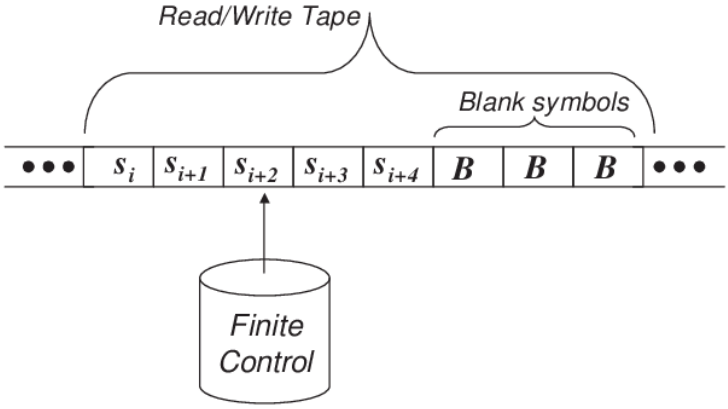


Figure 7.5

5) Non-deterministic Turing Machine(NTM):

Non-deterministic TM plays an important role in FAs and PDAs. It is convenient but not essential in caseofFA.

Deterministic Turing machines have enough computing power that nondeterministic fails to add any more. Non-deterministic TM differs from deterministic TM, we have seen earlier, in the function δ, such that for each state q and tape symbol X, δ (q, x) is a set of triples like,

$((q_1, Y_1, D_1), (q_2, Y_2, D_2) ..... (q_k, Y_k, D_K)\}$

here, k is finite integer.

At each step, aNTM chooses any of the triples to be the next move,but it cannot pick a state from one triple, a tape symbol from another and the direction from yet another triple. It take triples from entire triple group like (q_1, Y_1, D_1) at a time.

From this we can say that transitions in NTM are defined by a function from

$\delta: Q \times \Gamma \rightarrow$ subsets of Q x Γ x {L, R}.

String Acceptability by NTM

If there is any sequence of choices of move that leads from the initial ID with w as input to an ID with an accepting state then NTM, M accepts an input w.

The existence of other computations that halt in non-accepting states or fail to halt altogether is irrelevant.

Language Acceptedby NTM

The language accepted by a machine is the set of strings accepted by the M as mention in above point .

The acceptance in NTM can be defined by final state or by halting state alone. A NTM accepts a string u by halting if there is at least one computation that halts normally when run with u.

Thecomputational capability of TM does not increase by non-determinism: The languages accepted by NTM are those accepted by deterministic TMs. To accomplish the transformation of a NTM to an equivalent deterministic machine, we show that the multiple computations for a single input string can be sequentially generated and examined.

Examples

Example 1:Construct a TM machine for checking the palindrome of the string of even length.

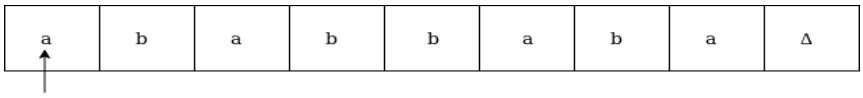
Solution:

Firstly we read the first symbol from the left and then we compare it with the first symbol from right to check whether it is the same.

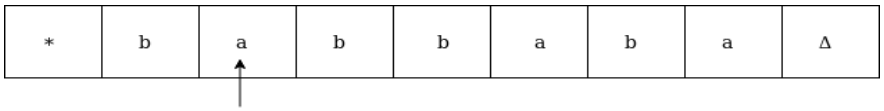
Again we compare the second symbol from left with the second symbol from right. We repeat this process for all the symbols. If we found any symbol not matching, we cannot lead the machine to HALT state.

Suppose the string is ababbabaΔ. The simulation for ababbabaΔ can be shown as follows:

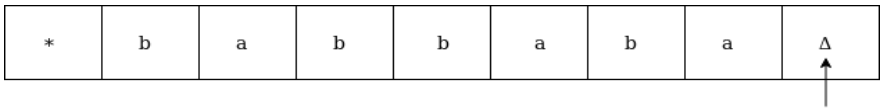
Now, we will see how this Turing machine will work for ababbabaΔ. Initially, state is q0 and head points to a as:



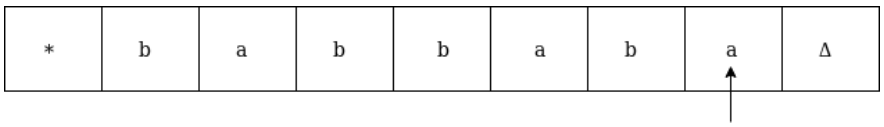
We will mark it by * and move to right end in search of a as:



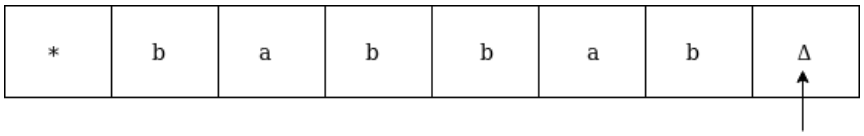
We will move right up to Δ as:



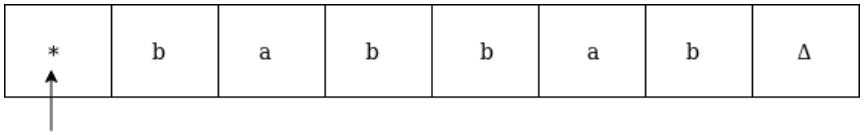
We will move left and check if it is a:



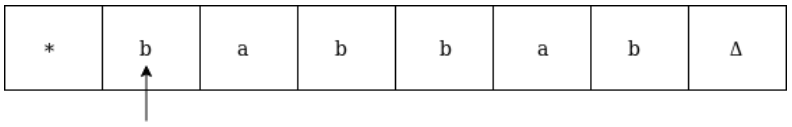
It is 'a' so replace it by Δ and move left as:



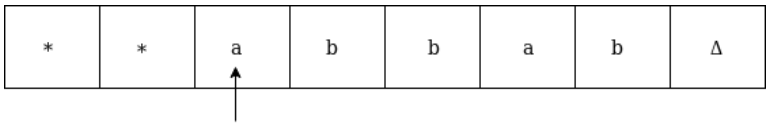
Now move to left up to * as:



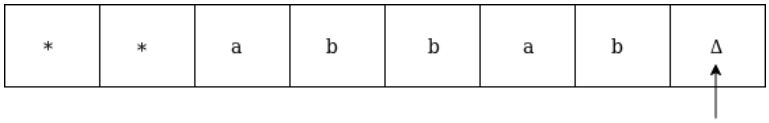
Move right and read it



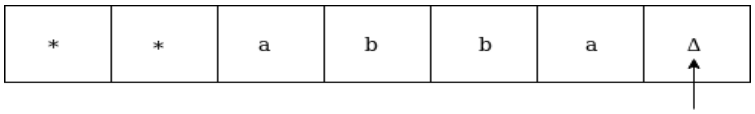
Now convert b by * and move right as:



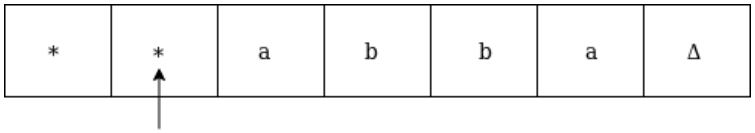
Move right up to Δ in search of b as:



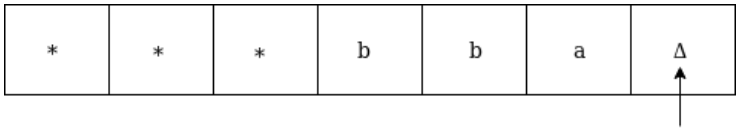
Move left, if the symbol is b then convert it into Δ as:



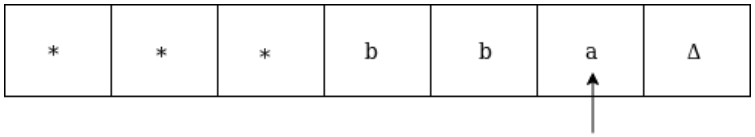
Now move left until * as:



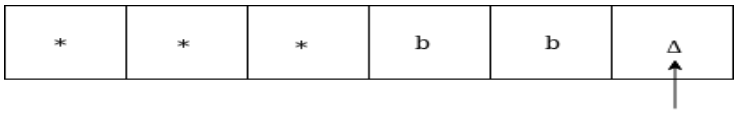
Replace a by * and move right up to Δ as:



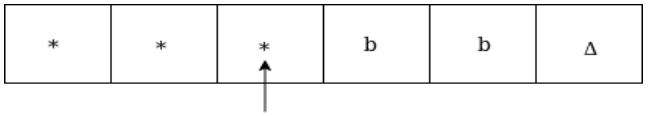
We will move left and check if it is a, then replace it by Δ as:



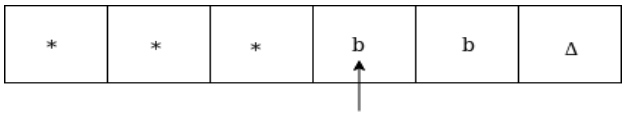
It is 'a' so replace it by Δ as:



Now move left until *

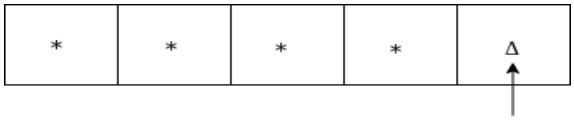


Now move right as:

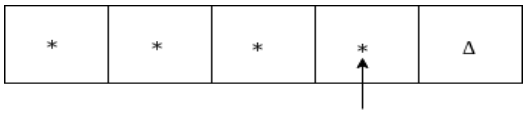


Replace b by * and move right up to Δ as:

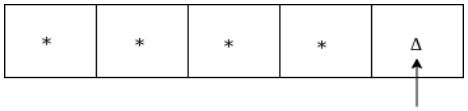
Move left, if the left symbol is b, replace it by Δ as:



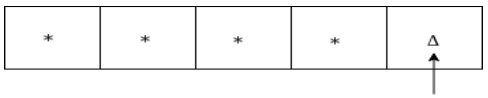
Move left till *



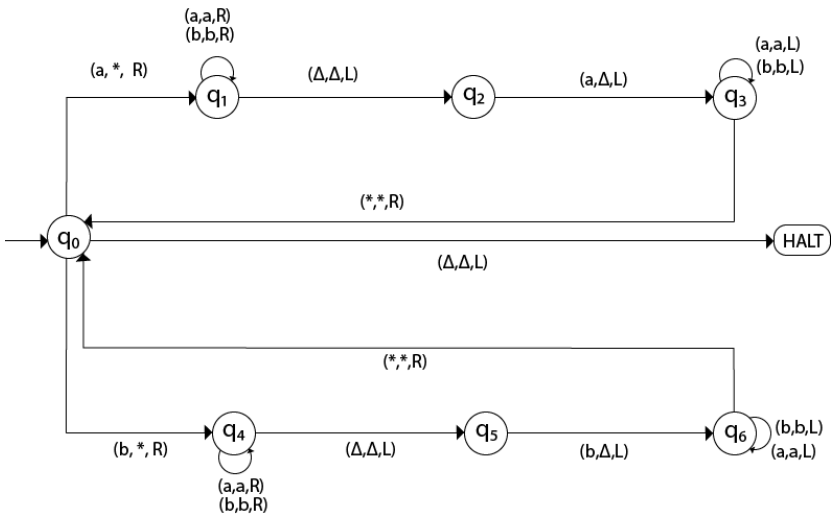
Move right and check whether it is Δ



Go to HALT state



The same TM can be represented by Transition Diagram:



Example 2:Design a Turing Machine to implement 1's complement .

Solution:

Logic for 1's complement

1. Scan input string from left to right
2. Convert '1' into '0'
3. Convert '0' into '1'
4. When BLANK is reached move towards left(i.e.start of input string).

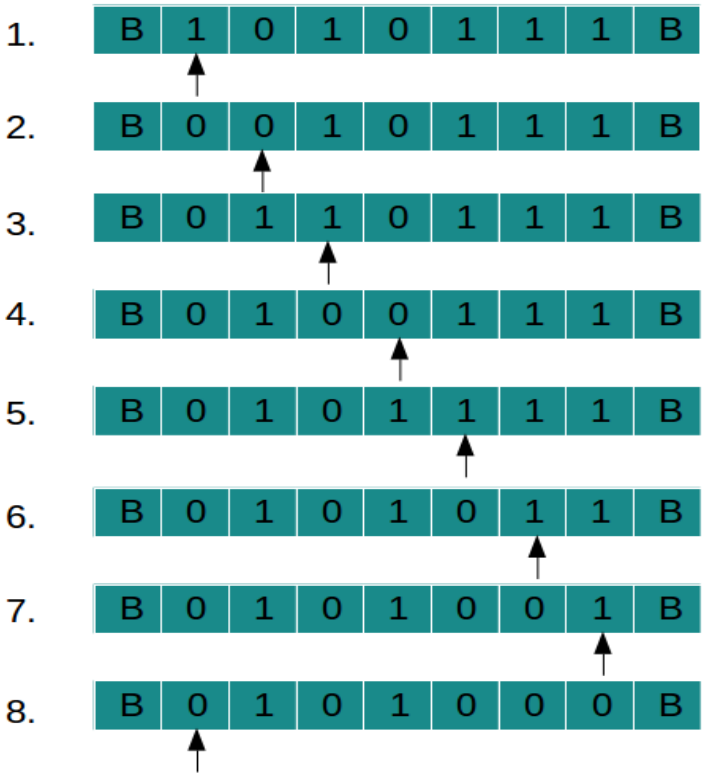
Consider, TAPE movement for string "1010111"

Sequential explanation of TAPE movement

1. Input is given as "1010111" (scan string from left to right)
2. Convert '1' into '0' and move one step right
3. Convert '0' into '1' and move one step right
4. Convert '1' into '0' and move one step right
5. Convert '0' into '1' and move one step right
6. Convert '1' into '0' and move one step right
7. Convert '1' into '0' and move one step right
8. Convert '1' into '0' and move one step right

When BLANK (in right) is reached, string is finished. So move to start of string (optional).

Tape movements are shown below.

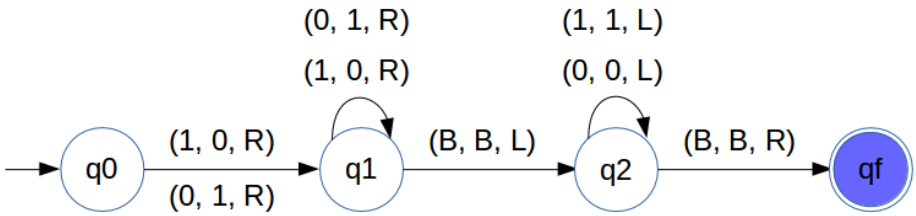


Let,

1's complement is written into the TAPE in place of input string.
Input String : 1010111
Output String : 0101000

State Transition Diagram

We have designed state transition diagram for 1's complement as follows:
1. Replace '1' with '0' and vice versa.
2. When BLANK is reached move towards left
3. Using state 'q2' we reach start of the string.
4. When we reach to BLANK in left we move one step right to point start of string.
5. qf is final state



Example 3:

Construct a Turing Machine for language L = {ww | w ∈ {0,1}}

Solution:

The w is a string. If w = 10110, so the Turing machine will accept the string

z = 1011010110.

Logic:

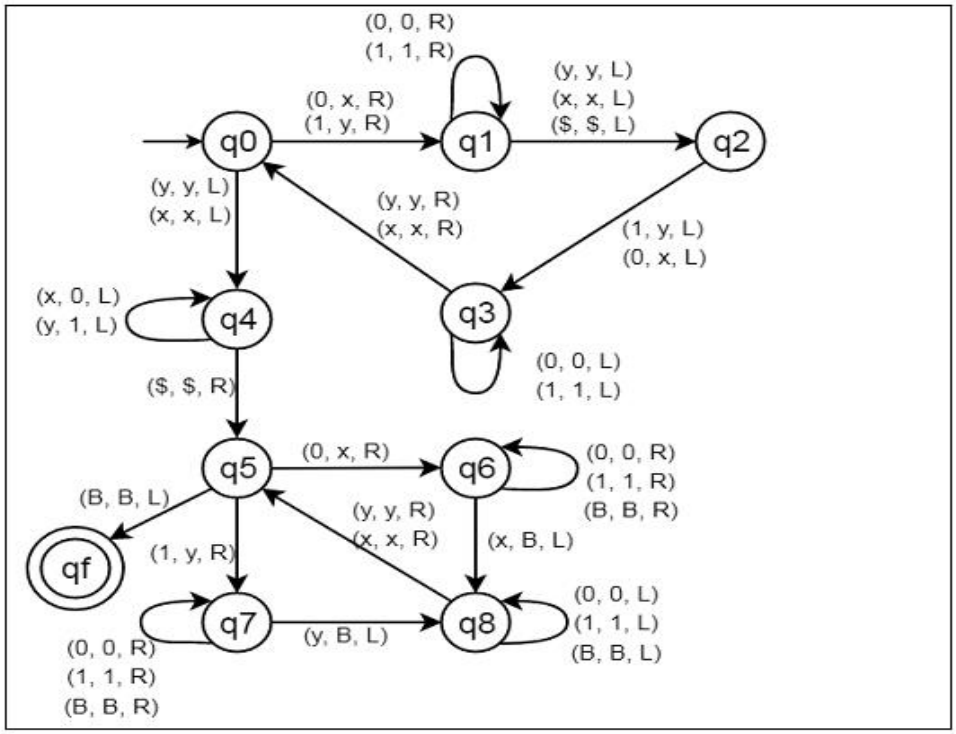
we will convert a 0 to x and 1 to y. After continuously doing it a point is reached when all 0's and 1's has been converted into x and x respectively. Now, we are on the midpoint of the string.

Now, convert all x's and y's on the left of the midpoint into 0's and 1's. Now the first half the string is in the form of 0 and 1. The second half of the string is in the form of x and y.

Now, again start from the beginning of the string. If you have a 0 then convert it into x and move right till reaching the second half, here if we find x then convert it into a blank(B). Then traverse back till find an x or a x. We convert the 0 or 1 at the right of it into x or y respectively and correspondingly, convert its x or y in the second half of string into a blank(B).

Continue this till converted all symbols on the left part of the string into x and y and all symbols on the right of string into blanks. When one part is completely converted but still some symbols in the other half are left unchanged then the string will not be accepted. If we did not find an x or y in the second half for a corresponding 0 or 1 respectively in the first half. Then also string will not be accepted.

State Transition Diagram



Example 4:

Construct a Turing Machine for language L = $\{0^{2n}1^n \mid n>=0\}$

Solution:

Stepwise Working :

- **Step-1:**

Given language contains twice number of 0's as compare with 1's. So, we will first make the first two zeros Blank and go from state Q0 to Q1 and from state Q1 to Q2.

- **Step-2:**

After making them Blank we will traverse to the end of the string till we get the rightmost 1 in state Q3 and make it Blank reaching state Q4.

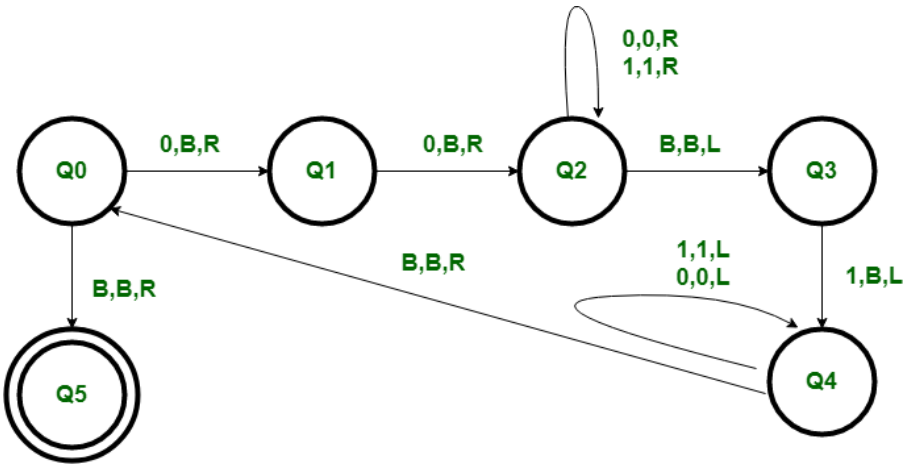
- **Step-3:**

Now we will traverse back till we get the left most zero in the string and return to the state Q0 from Q4.

- **Step-4:**

We are just reducing the string by making left most two zeros Blank and rightmost 1 Blank and if the string belongs to the language then it will be left empty and hence get accepted at state Q5 which is Final state. If the string is empty it will also get accepted at Q5.

State Transition Diagram



7.8 REVIEW QUESTIONS

1) Define Turing Machine.
2) Describe Turing Machine representation.
3) What type of language or grammar is accepted by Turing machine?
4) Design (construct) a TM for language $L=\{a^n b^{2n} \mid n \geq 1\}$
5) Construct a TM for language $L = \{0^n 1^n 2^n \mid n \geq 1\}$
6) Write note on variants of TM.
7) Construct TM to accept the language $0^* 1^*$

7.9 SUMMARY

1) Tuples of Turing Machine are:$M = (Q, \sum, \Gamma, \delta, q_0, B, F)$ $\delta : Q \times \Gamma \times \{L, R\}$

2) Language accepted by Turing Machines, $M = (Q, \sum, \Gamma, \delta, q_0, B, F)$, L(M) is the set of strings $w \in \sum^*$, such that $q_0 w \vdash \alpha p \beta$ for some state p in F and any tape strings α and β.

3) Variants of TM:

i) Multitrack TM

ii) Two-way TM

iii) Multiple tape TM

iv) Single tape TM

v) Non-deterministic TM

7.10 REFERENCES

1) Introduction to Computer Theory, Daniel Cohen, Wiley,2nd Edition

2) Introduction to Theory of Computation, Michel Sipser, Thomson.

8

UNDECIDABILITY

Unit Structure

8.0 OBJECTIVES

This chapter would make you to understand the following concepts:

- To study The Church-Turing thesis.
- Understand the Universal Turing Machine.
- What is Halting Problem?
- Introduction to Unsolvable Problems.

8.1 INTRODUCTION

In computing and mathematics there are many problems that are unsolvable. The deterministic Turing Machine has capability to compute whatever computational work is performed by the computer. Both are equally powerful for computation, and any of their variations do not exceeds the computational power of deterministic TM.

There are some problems which cannot be solved by TM and therefore by computer also, such a problem is consider asundecidable problem.

8.2 THE CHURCH-TURING THESIS

Alonzo Church wasAmerican mathematician and logician who made major contributions to mathematical logic and the foundations of theoretical computer science. His most renowned accomplishments were

Church's theorem, the Church-Turing thesis, and the creation of λ-calculus, or the Church λ operator.

There are various equivalent formulations of the Church-Turing thesis.

Formulation of the Church-Turing thesis is: " Anything that can be computed on any computational device can also be computed on a Turing machine".

or

"a problem can be solved by an algorithm iff it can be solved by a Turing Machine"

or

"A common one is that every effective computation can be carried out by a Turing machine. "

In the 1930s, two researchers– Alan Turing from England and Alonzo Church from theUS – started analyzing the question of what can be computed. They used two different approaches to answer this question:

- Alan Turing, with hiscomputational analysis of Turing machines, what we would now consider computer engineering and computer architecture;
- On the other hand, Church focused on what can be described – i.e., on what we would now consider programming languages.

Initially, they came up with two different answers to this question:

- Turing states that a function is computable if and only if it can be computed on a Turing machine, while
- Church statesthat a function is computable if and only it be described by a program in his programming language (which is similar to LISP and Scheme).

These, two statements areshows different answers – until Church proved that these definitions are actually equivalent:

- if a function can be computed on a Turing machine, then it can also be described by a program in Church's programming language, and
- if a function can be described by a program in Church's programming language, then it can also be computed on a Turing machine.

Later on, their two statements were merged into one statement, which we now call Church-Turing thesis.

Turing gave a very convincing argument that a human computer (performing symbolic manipulations with pen and paper) can be simulated by an appropriate Turing machine. Clearly, every Turing machine can be simulated by a human (who will just follow the program of the Turing

machine). Thus, we have an equivalence between the intuitive notion of an algorithm (of the symbolic-manipulation kind, as discussed above) and the formal notion of a Turing machine. This equivalence is usually called the "Church-Turing thesis"

8.3 UNIVERSAL TURING MACHINE

The Church-Turing thesis tells us that Turing machine is more powerful thanall effective models of computation.TM is create to execute specific algorithm.For every new computation the different Turing Machines are constructed for every input output relation. So this need is solved by introducing Universal Turing machine in which input on the tape takes the description of a machine M. It means if we have TM for computing one calculation, then computing a different calculation or another task we requires a different machine. Electronic computers have same limitations and if we determine to change the performance of machine ones need to rewrite the machine.

So we can design a Turing machine which gives us all computing capabilities like any other TM can do, this machine is called Universal Turing Machine(UTM). It can simulate the behaviour of any TM, which capable of running any algorithm.

This machine should have the capability of imitating any TM T, given the following information in its tape:

1. The description of T in terms of its program area or operation of the tape.

2. The Starting state or initial configuration of T and the symbol scanned (state area of the tape).

3. The data to be given to T (data area of the tape).

The Universal Turing Machine

● **Theorem1**: There is a Turing machine UTM called the universal Turing machine that, when run on ⟨M, w⟩, where M is a Turing machine and w is a string, simulates M running on w.

● Conceptually:

U_{TM} = "On input ⟨M, w⟩, where M is a TM and w ∈ Σ*:

Set up the initial configuration of M running on w.

while (true) {

If M accepted w, then UTM accepts ⟨M, w⟩.

If M rejected w, then UTM rejects ⟨M, w⟩.

Otherwise, simulate one more step of M on w.

}"

● **Theorem2**: There is a Turing machine UTM called the universal Turing machine that, when run on ⟨M, w⟩, where M is a Turing machine and w is a string, simulates M running on w.

- The observable behavior of UTM is the following:
- If M accepts w, then UTM accepts ⟨M, w⟩.
- If M rejects w, then UTM rejects ⟨M, w⟩.
- If M loops on w, then UTM loops on ⟨M, w⟩.

Design of UTM

The UTM should have the capability to correctly interprets the rules of operations of T using algorithm.

A UTM is designed to simulate the computations of an arbitrary turing machine M. For this computation the input of the universal machine must contain a representation of the machine M and the string w to be processed by M.

To achieve this we assume that M is a standard TM which accepts by halting. The action of UTM, U is represented by:

Let R(M) be the representation of machine M with string w.

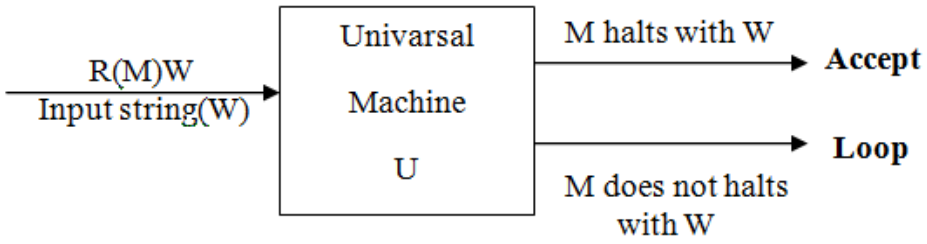


Figure 8.1

The output labeled 'loop' shows that the computation of U does not terminate. If M halts and accepts input w, U does the same.

If M does not halt with w, neither does U. The machine U is called universal since the computationof any TM M can be simulated by U.

In universal machineconstruction is for to design the string representation of a turing machine. Because of the ability to encode arbitrary symbols as strings over {0, 1}, we consider truing machines with input alphabet {0, 1} and tape alphabet {0, 1, B}. The states of turing machine are assumed to be named {qo, q1 ..... qn} whereqo as start state.

Formal Definition

Turing machine M is defined by its transition function(δ). A transition of a standard turing machine is given as:

$\delta (q_i, x) = [q_j) y, d]$

Where $q_i, q_j \in Q$, $x, y \in \Gamma$ and $d \in \{L, R)$

We encode the elements of M using strings of 1's:

Symbol	Encoding
0	1
1	11
B	111
q_0	1
q_1	11
q_n	n+1
L	1
R	11

Consider the following TM:

$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B), \delta, q_1, B, \{q_2\})$

The moves are $\delta (q_1, 1) = (q_3, 0, R)$

$\delta (q_3, 0) = (q1, 1,R)$

$\delta (q_3, 1) = (q_2, 0, R)$

$\delta (q_3, B)= (q_3, 1, L)$

Then the machines M can be coded as:

111**0100100010100110001010100100 11**

0001001001010011000100010001001 0111

The code begins and ends with 111. The bold portion represents one move of TM.

UTM Simulation of T

UTM can simulate T, one step at a time. Steps are as follows:

Step 1: Scan the square (cell) on the state area of the tape and read the symbol that T reads and initial state of T.

Step 2: Move the tape over program area containing the description of T find out the row pointed by the symbol, read in step 1.

Step 3: Find the column pointed by the statesymbol in which T resides and then read the triple (new state ,symbol to be replace and direction of the tape movement) in the intersection of this column with the row found in step 2.

Step 4: Move the tape to the appropriate cell in the data area, replace the symbol, move the head in required direction, read next symbol and finally read the state area, replace the state and scanned symbol. And go to step 1.

Let end(z) denote the encoding of a symbol z. A transition $\delta (q_i, x) = [qj, y, d]$ isencodedbythestring $end(q_i)$ 0end(x) $0end(q_j)$ 0end(y) 0end(d)

The components of transitions are separated by 0's. Two of consecutive 0's are used to show separate transition. The beginning and ending of the representation are designed by 3 0's.

Example: Let $M = (Q, \{0, 1\}, \{0, 1, B), \delta, q_1, B, \{q_2\})$ be a TM.

Assume $Q = \{q_1, q_2, ...q_n\}$

Let's consider 0, 1, and B as X1, X2, and x3 respectively.

D_1 and $D_2$are head movements for directions L and R respectively.

Consider the following move:

$\delta (q_i, X_j) = (q_k, X_l, D_m)$

The move can be encoded by the binary string:

$0^i 1 0^j 1 0^k 1 0^l 1 0^m$

A turing machine M with binary code is

$111\ code_1\ 11\ code_2\ 11\ .....\ 11\ code_n 111,$ …..Form(1)

Where each $code_i$ is of each above form, this code is beginning and ending with 111. Each move of M is encoded by one of the $code_i$. Each $code_i$is separated by two consecutive 11's. Thus the encoding will be unique.

Above imitation algorithm have problem. since here we have only one-dimensional linear tape on the UTM, we require two-dimensional description of T unless we use some coding to convert the two-dimensional information into one-dimensional.

:. So we can design a UTM in following way.

The Turing machine is described as a multitape. TM as shown:

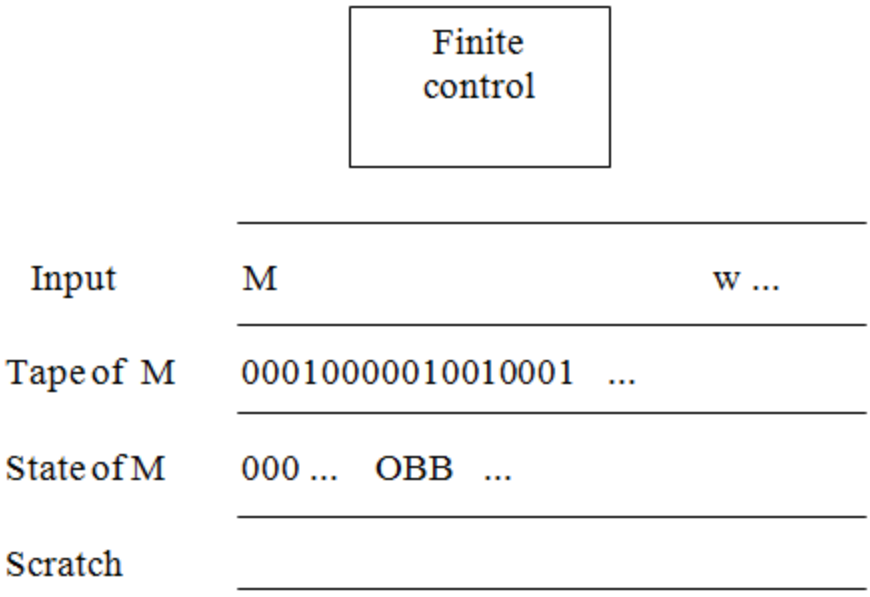


Figure 8.2

Operation of UTM is as follows:

i) Examine the input to make sure that the code for M is valid code for some TM. If not U halts without accepting. For invalid codes are assumed to represent the TM with no moves, and such a TM accepts no inputs, this action is correct.

ii) Initialize the second tape to contain the input w, in its encoded form. i.e. for each 0 of w, place 10 on the second tape, and for each 1 of w, place 100 there.

(Blanks on the simulated tape of M, which are represented by 1000, will not actually appear on that tape, all cells beyond those used for w will hold the blank of U.

However, U knows that, should it look for a simulated symbol of M and find its blank, it must replace that blank by the sequence 1000 to simulate the blank of M)

iii) Place simulated 0,at thestart state of M, on the third tape and move the head of U's second tape to the first simulated cell.

iv) For simulate the move of M, U search it on its first tape for the transition of $0^i\ 1\ 0^j\ 1\ 0^k\ 1\ 0^l\ 1\ 0^m$ where 0^i is state on third tape, and 0^j is tape symbol of M which begins at the position on tape 2 scanned by U. This is the transition of one M which create next one. U should:

a) Change the contents of third tape to 0^k. Means it should able simulate the state change of M, for that change , U first changes all the 0's on tape 3 to blanks and then copies 0^k from tape 1 to tape 3.

b) Replace 0^j on the tape 2 by 0^l means make change in tape symbol of M. If more or less space is needed (i.e. $I \neq 1$), then use the scratch tape and using shifting over technique manage the space.

c) Now, move the head on tape2 to the position of the next 1 to the left or right respectively, depending on weather m = 1 (move left) or m = 2 (move right) . Therefore U makes move of M to the left or to the right.

v) If there is no match in M for transition that matches the simulated state and tape symbol, then in (iv), no transition will be found. Thus M halts in the simulated

Configurationand U must do likewise.

vi) If M enters into its accepting state, then U accept.

8.4 HALTING PROBLEM

Before start to study Halting Problem, let's learn some concepts:

- **Computability theory –**It is the branch of theory of computation which studies the problems of computationally solvable using different model. In computer science, the computational complexity, or complexity of an algorithm is the amount of resources required for running it.
- **Decision problems –**A decision problem has only two possible outputs (i.e. yes or no) on any given input. In terms of computability theory and computational complexity theory, a decision problem is a problem that can be posed as a yes-no question for the input values. Like is there any solution to a particular problem? The answer would be either yes or no. Simply a decision problem is any arbitrary yes/no question on an infinite set of inputs.
- **Turing machine –**Now we know very well that, a Turing machine is a mathematical model of computation. A Turing machine is a general example of a CPU which controls all data manipulation work, performed by a computer. Turing machine can be of halting as well as non halting type and it depends on algorithm and input associated with the algorithm.
- **Decidable language** –

  A decision problem P is said to be decidable (i.e., have an algorithm) if the language L of all yes instances to P is decidable. Example-

(I) (Acceptance problem for DFA) Given a DFA does it accept a given word?

(II) (Emptiness problem for DFA) Given a DFA does it accept any word?

(III) (Equivalence problem for DFA) Given two DFAs, do they accept the same language?

If answer to above examples is yes then language generated by above is decidable.

- **Undecidable language** –

  A decision problem P is said to be undecidable if the language L of all yes instances to P is not decidable or a language is undecidable if it is not decidable. An undecidable language maybe a partially decidable language or something else but not decidable. If a language is not even partially decidable , then there exists no Turing machine for such language.

Halting is a situation in the program that tells us on certain input will accept it and halt or reject it and halt and it would never go into an infinite loop. Basically halt shows terminating the current program or algorithm execution on particular input condition.

So can we have an algorithm which will tell us that is given program will halt or not. In terms of Turing machine, will it terminate when run on some machine with some particular given input string.

The answer is no we cannot design a generalized algorithm which can appropriately say that given a program will ever halt or not?The only way is to run the program and check whether it halts or not.We can solve the halting problem question in such a way also: Given a program written in some programming language(c/c++/java) will it ever get into an infinite loop(loop never stops) or will it always terminate(halt)?

It is an undecidable problem because we cannot have an algorithm which will tell us whether a given program will halt or not in a generalized way at certain point in executioni.e by having specific program/algorithm.In general we can't always know that's we can't have a general algorithm. The best possible way is to run the program and see whether it halts or not.In this way for many programs we can see that it will sometimes loop and always halt.

The halting problem is a decision problem about properties of computer program on a Turing Machine computation model. The problem is to determine, for a given program and an input to a program, whether the program will halt when run with that input.

In this abstract environment, there is no resource limitations of memory or time on the program's execution, before halting it can take arbitrary long storage space. The question is that whether the given program will ever halt on a particular input.

Example

i. The following program having segment:

```
while (1) {continue; }
```

does not halt. It goes in infinite loop.

ii. The following program

```
printf ( " Hello" ) ;
```

halts very soon:

The halting problem is undecidable: This means that there is no algorithm which can be applied to any arbitrary program and input to determine whether the program stops when run with that input.

Definition of Halting Problem

The halting problem for Turing machines is defined as follows:

Given a TM M = (Q, ∑, Γ, δ, q_o, B, F) and an input string $x \in \Gamma^*$, will M eventually halt?

Halting Problem representing as a Set

The conventional representation of decision problems is the set of objects processing or satisfying the property in question:

The halting set:K = {(i, x) program i will eventually halt if run with input x} represents the halting problem.

This set is recursively enumerable i.e. there is a computable function that lists all of the pairs (i, x) it contains. However, the complement of this set is not recursively enumerable.

The halting problem would be solvable of a TM H which behaves like shown below can be constructed:

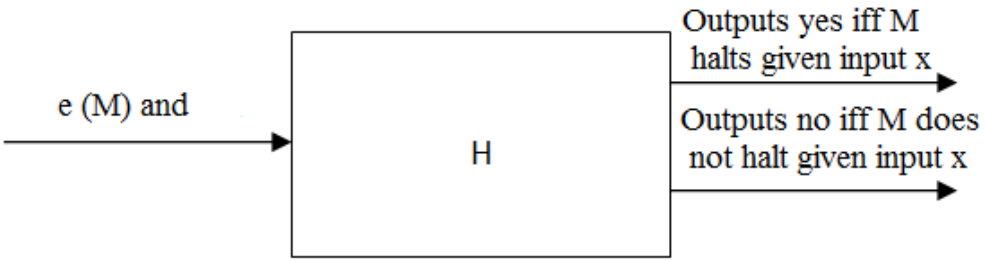


Figure 8.3

where,

e(M): Encoding of M

i.e. e(M) is for example a set of 5-tuples (q, X_1, p, r, R) that describe the TM.

Then the halting problem is:

Thus there exist an effective procedure (i.e.) an computable function for deciding, for every pair (e(M), x); does M halt for x?

In this section we introduce the reduction technique. This technique is used to prove the undecidability of halting problem of Turing machine.

We say that problem A is reducible to problem B if a solution to problemB can be used to solve problem A.

For example, ifA is the problem of finding some root of $x^4 - 3x^2 + 2 = 0$

and Bis the problem of finding some root of $x^2 - 2 = 0$, then A is reducible

toB. As $x^2 - 2$ is a factor of $x^4 - + - 3x^2 + 2$, a root of $x^2 - 2 = 0$ is also a root

of $x^4 - + - 3x^2 + 2$.

Note: IfA is reducible to B and Bis decidable then A is decidable. If Ais reducible to B and A is undecidable, then B is undecidable.

Theorem

$HALT_{TM}$={(M, w) | The Turing machine M halts on input w} is undecidable.

Proof :

We assume that $HALT_{TM}$is decidable, and get a contradiction. Let M_1

be the TM such that $T(M_1) = HALT_{TM}$ and let M_1halt eventually on all

(M, w).We construct a TM M_2 as follows:

1. For M_2, (M, w) is an input.

2. The TM M_1 acts on (M, w).

3. If M_1 rejects (M, w) then M_2rejects (M, ,w).

4. If M_1 accepts (M, w),simulate the TM M on the input string w until M halts.

5. If M has accepted w, M_2 accepts (M, w); otherwise M_2rejects (M, w).

When M_1 accepts (M, w) (in step 4), the Turing machine M halts on w.

In this case either an accepting state q or a state q' such that δ(q', a) is undefined till some symbol a in w is reached. In the first case (the first alternative of step 5) M_2accepts (M. w). In the second case (the second alternative of step 5) M_2 rejects (M, w).

It follows from the definition of M_2that M_2halts eventually.

Also,$T(M_2)$ = {(M, w) | The Turing machine accepts w} = A_{TM}

This is a contradiction since A_{TM} isundecidable.

8.5 INTRODUCTION TO UNSOLVABLE PROBLEMS

The deterministic Turing machines are capable of doing any computation that computers cando,i.e computationally they are equally powerful and any of their variations do not exceed the computational power of deterministic TM.

There are problems that cannot be solved by TMs and hence by computers so the concept of Unsolvability comes in front.

Solving a problem can be viewed as recognizing a language. The unsolvability can be seen in terms of language recognition.

Suppose that a language is acceptable but not decidable. Then given a string a TM that accept the language starts the computation. At any point of time if TM is running, there is no way of telling whether it is in an infinite loop or along the way to a solution and it needs more time.

This if a language is not decidable, the question of whether or not a string is in the language may not be answered in any finite amount of time. Since we cannot wait forever for an answer, the question is unanswerable i.e. the problem is unsolvable or undecidable.

Examples Unsolvable Problems (UndecidableProblems)

a)Unsolvable Problems about Turing Machines:

Using the reduction technique, thefollowing problems can be shown unsolvable:

1. If M is a TM, and x is a string of input symbols to M, does M halt on x?

2. If M is a TM that computes a function f with no arguments, does M halt on a blank tape?

3 .Given a TM M, halt for any string of input symbols?

4. If M is a TM, is the set of input strings on which M does halt regular? Is it context free? Is it X input?

5. Given two TMs, M_1 and M_2, over the same alphabet, do M_1 and M_2 halt on the same set of input strings?

b) Unsolvable Problems about (General) Grammars:

Unsolvability results can also be shown about grammars, using reductions.These

problems are unsolvable.

1. Given a grammar G and a string w, is w ∈L(G)?
2. Given a grammar G, is ∈∈L(G)?
3. Given grammars G1 and G2, is L(G1) = L(G2)?
4. Given a grammar G, is L(G) = φ?
5. There is a fixed grammar G such that it is undecidable given w whether w ∈L(G)

c) Unsolvable Problems about Context-Free Grammars(CFG)

The following are undecidable:

1. Given a context-free grammar G, is L(G) = Σ∗ ?
2. Given two CFG G1 and G2, is L(G1) = L(G2)?
3. Given two push-down automata M1 and M2, is L(M1) = L(M2)?
4. Given a push-down automaton M, find an equivalent push-down automaton with as few states as possible.

Here are some more unsolvable problems about context-free grammars:

Given a context-free grammar G, is G ambiguous?

Given context-free grammars G1 and G2, is L(G1) ∩ L(G2) = φ?

Some of the preceding problems are ones that we would very much like to be able to solve. Some problems are in areas not related to grammars or Turing machines at all. For example, Hilbert's Tenth problem has to do with Diophantine equations, and was shown to be unsolvable in the 1970's by a very complicated series of reductions. Hilbert's tenth problem is to give a computing algorithm which will tell of a given polynomial Diophantine equation with integer coefficients whether or not it has a solution in integers. Matiyasevic proved that there is no such algorithm.

Decidable problems related regular languages

Let, begin with certain computational problems concerning finite automata. We give algorithms for testing whether a finite automaton accepts a string, whether the language of a finite automaton is empty, and whether two finite automata are equivalent.

For convenience we use languages to represent various computational problems because we have already set up terminology for dealing with languages. For example, the acceptance problem for DFAs of testing whether a particular finite automaton accepts a given string can be expressed as a language, A(DFA). This language contains the encodings of all DFAs together with strings that the DFAs accept. Let

A(DFA) = { (B, w) B is a DFA that accepts input string w}.

The problem of testing whether a DFA B accepts an input w is the same as the problem of testing whether (B, w) is a member of the language A(DFA). Similarly, we can formulate other computational problems in terms of testing membership in a language. Showing that the language is decidable is the same as showing that the computational problem is decidable.

In the following theorem we show that A(DFA) is decidable. Hence this theorem shows that the problem of testing whether a given finite automaton accepts a given string is decidable.

Theorem

A(REX) is a decidable language.

Proof:

The following TM P decides AREX.

P = "On input (R, w) where R is a regular expression and w is a string:

a) Convert regular expressionR to an equivalent DFA.
b) Run TM M on input (A, w).
c) If M accepts, accept; if M rejects, reject."

8.8 REVIEW QUESTIONS

1) Define the Church-Turing thesis.
2) Define Universal Turing Machine.
3) Write note on HaltingProblem.
4) What is Unsolvable Problems? Give examples.

8.9 SUMMARY

- The Church-Turing thesis.

  " Anything that can be computed on any computational device can also be computed on a Turing machine".

- A transition of a standard turing machine is given as:
  $\delta (q_i, x) = [q_j) y, d]$
  Where $q_i, q_j \in Q$, $x, y \in \Gamma$ and $d \in \{L, R)$

- The halting problem for Turing machines: Given a TM
  $M = (Q, \sum, \Gamma, \delta, q_o, B, F)$ and an input string $x \in \Gamma^*$, will M ventually halt?

- There are problems that cannot be solved by TMs and hence by computers so they termed as Unsolvable Problems.

8.10 REFERENCES

1) Theory Of Computer ScienceAutomata, Languages and Computation Third Edition by K.l.P. Mishra and N. Chandrasekaran.

2) Introduction to Theory of Computation, Michel Sipser, Thomson.