MASTER OF COMPUTER APPLICATION

MCA-35

THEORY OF COMPUTATION

Directorate of Distance Education
Guru Jambheshwar University of Science & Technology, Hisar – 125001

CONTENTS

SUBJECT: Theory of Computation	
COURSE CODE: MCA- 35	AUTHOR: RAVIKA GOEL
LESSON NO. 1	
TOC AN INTRODUCTION	

STRUCTURE

1.0 OBJECTIVE

The main objective of this lesson is to understand the basic Concept of Automaton (Finite state machine). How NDFA and DFA Works. What is the Basic Difference between NDFA and DFA. Find out the equivalence Power of NDFA and DFA. How a string is accepted by NDFA and DFA.

1.1 INTRODUCTION

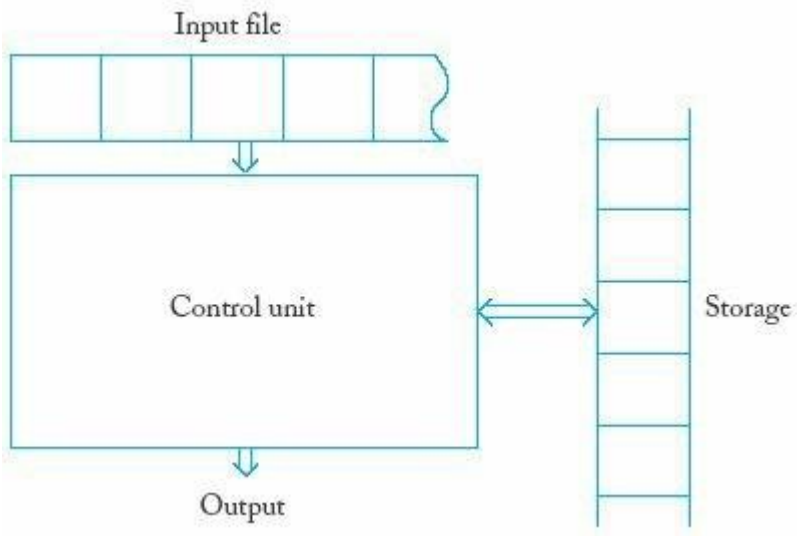
The theory of computation, includes several topics: automata theory, formal languages and grammars, computability, and complexity. To model the hardware of a computer, we introduce the notion of an automaton (plural, automata). An automaton is a construct that possesses all the indispensable features of a digital computer. It accepts input, produces output, may have some temporary storage, and can make decisions in transforming the input into the output. A formal language is an abstraction of the general characteristics of programming languages. A formal language consists of a set of symbols and some rules of formation by which these symbols can be combined into entities called sentences. A formal language is the set of all sentences permitted by the rules of formation. Although some of the formal languages we study here are simpler than programming languages, they have many of the same essential features. We can learn a great deal about programming languages from formal languages. Finally, we will formalize the concept of a mechanical computation by giving a precise definition of the term algorithm and study the kinds of problems that are (and are not) suitable for solution by such mechanical means.

We have only a general understanding of what an automaton is and how it can be represented by a graph. This type of automaton is characterized by having no temporary storage. Since an input file cannot be rewritten, a finite automaton is severely limited in its capacity to "remember" things during the computation. A finite amount of information can be retained in the control unit by placing the unit into a specific state. But since the number of such states is finite, a finite automaton can only deal with situations in which the information to be stored at any time is strictly bounded.

1.2 Definition of Automaton

An automaton is an abstract model of a digital computer. As such, every automaton includes some essential features. It has a mechanism for reading input. It will be assumed that the input is a string over a given alphabet, written on an input file, which the automaton can read but not change. The input file is divided into cells, each of which can hold one symbol. The input mechanism can read the input file from left to right, one symbol at a time. The input mechanism can also detect the end of the input string (by sensing an end-of-file condition). The automaton can produce output of some form. It may have a temporary storage device, consisting of an unlimited number of cells, each capable of holding a single symbol from an alphabet (not necessarily the same one as the input alphabet). The automaton can read and change the contents of the storage cells. Finally, the automaton has a control unit, which can be in any one of a finite number of internal states, and which can change state in some defined manner.

Figure shows a schematic representation of a general automaton



An automaton is assumed to operate in a discrete timeframe. At any given time, the control unit is in some internal state, and the input mechanism is scanning a particular symbol on the input file. The internal state of the control unit at the next time step is determined by the next-state or transition function. This transition function gives the next state in terms of the current state, the current input symbol, and the information currently in the temporary storage. During the transition from one time interval to the next, output may be produced or the information in the temporary storage changed. The term configuration will be used to refer to a particular state of the control unit, input file, and temporary

storage. The transition of the automaton from one configuration to the next will be called a **move.**

1.3 Finite State Machine

A deterministic automaton is one in which each move is uniquely determined by the current configuration. If we know the internal state, the input, and the contents of the temporary storage, we can predict the future behavior of the automaton exactly. In a nondeterministic automaton, this is not so. At each point, a nondeterministic automaton may have several possible moves, so we can only predict a set of possible actions. The relation between deterministic and nondeterministic automata of various types will play a significant role in our study.

An automaton whose output response is limited to a simple "yes" or "no" is called an accepter. Presented with an input string, an accepter either accepts the string or rejects it. A more general automaton, capable of producing strings of symbols as output, is called a transducer. The finite state machines **(FSMs**) are significant for understanding the decision making logic as well as control the digital systems. In the FSM, the outputs, as well as the next state, are a present state and the input function. This means that the selection of the next state mainly depends on the input value and strength lead to more compound system performance. As in sequential logic, we require the past inputs history for deciding the output. Therefore FSM proves very cooperative in understanding sequential logic roles. Basically, there are two methods for arranging The definition of a finite state machine **is**, the term finite state machine (FSM) is also known as finite state automation. FSM is a calculation model that can be executed with the help of hardware otherwise software. This is used for creating sequential logic as well as a few computer programs. FSMs are used to solve the problems in fields like mathematics, games, linguistics, and artificial intelligence. In a system where specific inputs can cause specific changes in state that can be signified with the help of FSMs.a sequential logic design namely mealy machine as well as more machine.

Finite Automata (FA) is the simplest machine to recognize patterns. The finite automata or finite state machine is an abstract machine that has five elements or tuples. It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. Basically, it is an abstract model of a digital computer. The following figure shows some essential features of general automation.

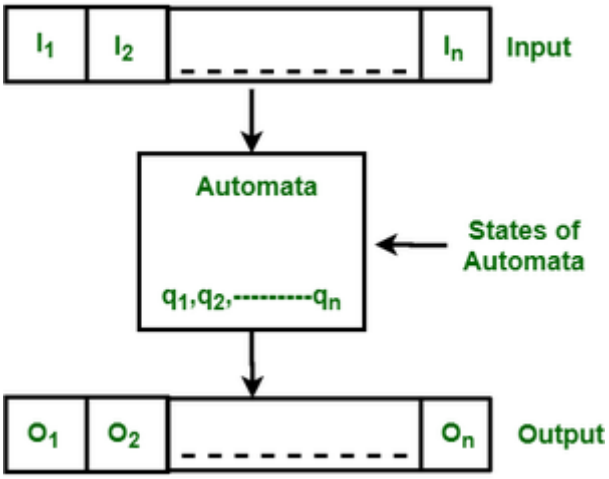


Figure : Features of Finite Automata

The above figure shows the following features of automata:

1. Input
2. Output
3. States of automata
4. State relation
5. Output relation

A Finite Automata consists of the following :

Q : Finite set of states.

Σ : set of Input Symbols.

q : Initial state.

F : set of Final States.

δ : Transition Function.

Formal specification of machine is

{ Q, Σ, q, F, δ }

Finite Automaton can be classified into two types −

- Deterministic Finite Automaton (DFA)

- Non-deterministic Finite Automaton (NDFA / NFA)

1.4 Deterministic Finite Automata

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton.**

Formal Definition of a DFA

A DFA can be represented by a 5-tuple $(Q, \sum, \delta, q_0, F)$ where −

- **Q** is a finite set of states.
- $\sum$ is a finite set of symbols called the alphabet.
- **δ** is the transition function where $\delta: Q \times \sum \rightarrow Q$
- $\mathbf{q_0}$ is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

Graphical Representation of a DFA

A DFA is represented by digraphs called **state diagram.**

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

A transition diagram or state transition diagram is a directed graph which can be constructed as follows:

- There is a node for each state in Q, which is represented by the circle.
- There is a directed edge from node q to node p labeled a if $\delta(q, a) = p$.
- In the start state, there is an arrow with no source.
- Accepting states or final states are indicating by a double circle.

Some Notations that are used in the transition diagram:

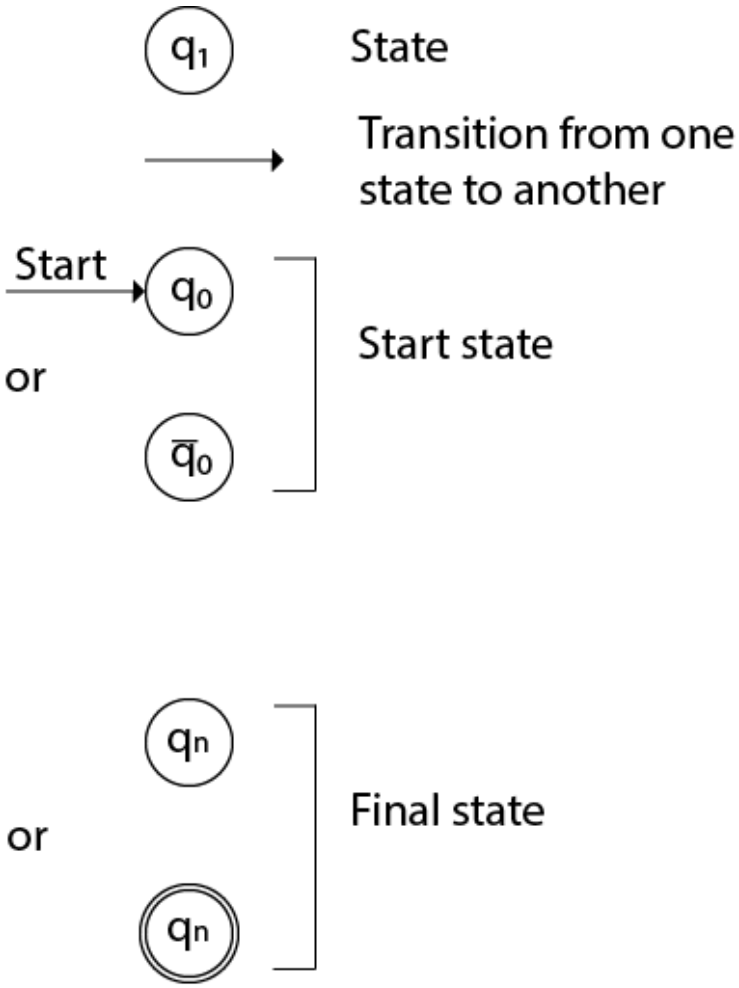


Fig:- Notations

There is a description of how a DFA operates:

1. In DFA, the input to the automata can be any string. Now, put a pointer to the start state q and read the input string w from left to right and move the pointer according to the transition function, δ. We can read one symbol at a time. If the next symbol of string w is a and the pointer is on state p, move the pointer to δ(p, a). When the end of the input string w is encountered, then the pointer is on some state F.

2. The string w is said to be accepted by the DFA if r ∈ F that means the input string w is processed successfully and the automata reached its final state. The string is said to be rejected by DFA if r ∉ F.

Example 1:

DFA with ∑ = {0, 1} accepts all strings starting with 1.

Solution:

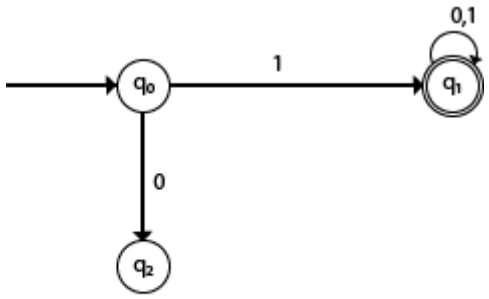


Fig: Transition diagram

The finite automata can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q0 on receiving 0, the machine changes its state to q2, which is the dead state. From q1 on receiving input 0, 1 the machine changes its state to q1, which is the final state. The possible input strings that can be generated are 10, 11, 110, 101, 111....., that means all string starts with 1.

Example

Let a deterministic finite automaton be →

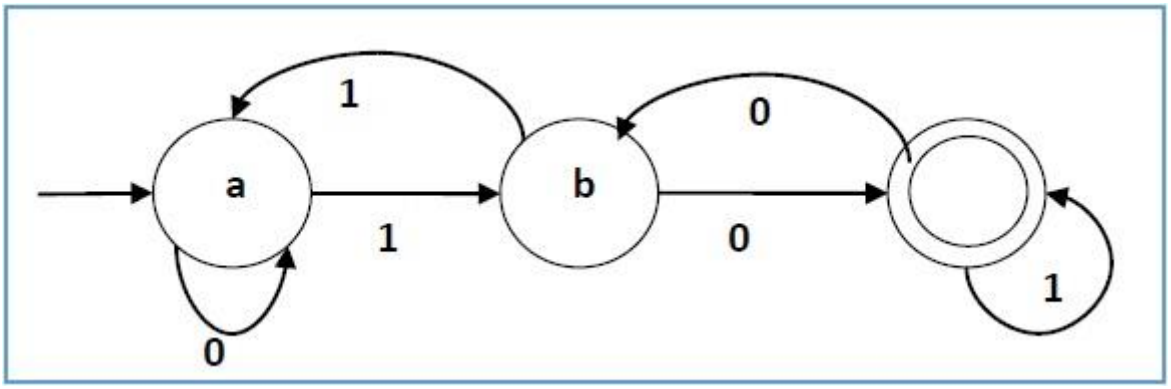
- Q = {a, b, c},
- ∑ = {0, 1},
- q_0 = {a},
- F = {c}, and

Transition function δ as shown by the following table –

Present State	Next State for Input 0	Next State for Input 1
A	A	B
B	C	A

C	B	C

Its graphical representation would be as follows −



1.5 Non Deterministic Finite State Automaton

Non-Deterministic Finite Automata is defined by the quintuple-

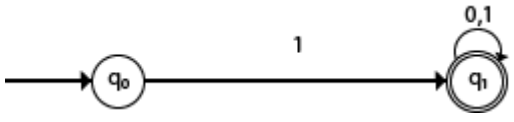
$$M = (Q, \sum, \delta, q_0, F)$$

where-

- Q = finite set of states
- $\sum$ = non-empty finite set of symbols called as input alphabets
- $\delta : Q \times \sum \rightarrow 2^Q$ is a total function called as transition function
- $q0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of final states

Example: NFA with $\sum$ = {0, 1} accepts all strings starting with 1.

Solution:

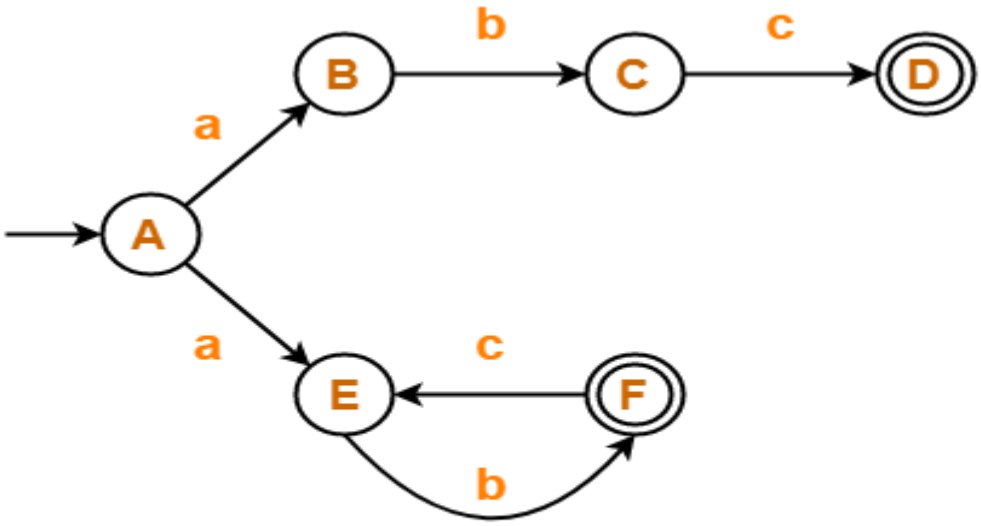


The NFA can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q1 on receiving input 0, 1 the machine

changes its state to q1. The possible input string that can be generated is 10, 11, 110, 101, 111....., that means all string starts with 1.

Example of Non-Deterministic Finite Automata Without Epsilon-

Following automata is an example of Non-Deterministic Finite Automata without epsilon-



Example of Non-Deterministic Finite Automata

(Without Epsilon)

The above NFA can be defined in form of five tuples as-

$$\{ \{A, B, C, D, E, F\}, \{a, b, c\}, \delta, A, \{D, F\} \}$$

where-

- {A, B, C, D, E, F} refers to the set of states
- {a, b, c} refers to the set of input alphabets
- δ refers to the transition function
- A refers to the the initial state
- {D, F} refers to the set of final states

Transition function δ is defined as-

- δ (A, a) = B

- δ (A, a) = E
- δ (B, b) = C
- δ (C, c) = D
- δ (E, b) = F
- δ (F, c) = E

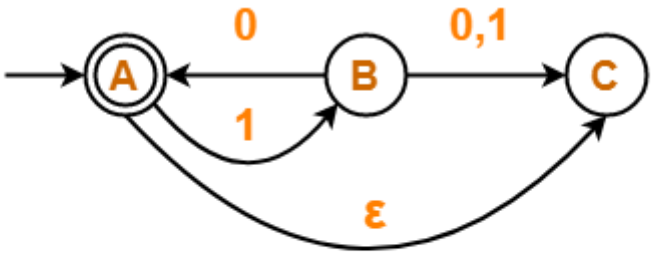
Transition Table for the above Non-Deterministic Finite Automata is-

States / Alphabets	A	B	C
A	{B, E}	–	–
B	–	C	–
C	–	–	D
D	–	–	–
E	–	F	–
F	–	–	E

1.6 NDFA with null moves

Example of Non-Deterministic Finite Automata With Epsilon-

Following automata is an example of Non-Deterministic Finite Automata with epsilon-



Example of Non-Deterministic Finite Automata

(With Epsilon)

The above NFA can be defined in form of five tuples as-

$$\{ \{A, B, C\}, \{0, 1\}, \delta, A, \{A\} \}$$

where-

- {A, B, C} refers to the set of states
- {0, 1} refers to the set of input alphabets
- δ refers to the transition function
- A refers to the the initial state
- {A} refers to the set of final states

Transition function δ is defined as-

- δ (A, 1) = B
- δ (A, ∈) = C
- δ (B, 0) = A
- δ (B, 0) = C
- δ (B, 1) = C

Transition Table for the above Non-Deterministic Finite Automata is-

States / Alphabets	0	1	∈
A	–	B	C
B	{A, C}	C	–
C	–	–	–

Dead Configuration or Trap State-

In Non-Deterministic Finite Automata,

- The result of a transition function may be empty.
- In such a case, automata gets stopped forcefully after entering that configuration.
- This type of configuration is known as dead configuration.
- The string gets rejected after entering the dead configuration.

1.7 Equivalence of NDFA and DFA

The following table lists the differences between DFA and NDFA.

DFA	NDFA
The transition from a state is to a single particular next state for each input symbol. Hence it is called deterministic.	The transition from a state can be to multiple next states for each input symbol. Hence it is called non-deterministic.
Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.
Backtracking is allowed in DFA	In NDFA, backtracking is not always possible.
Requires more space.	Requires less space.
A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state.

Equivalence of DFA and NFA-

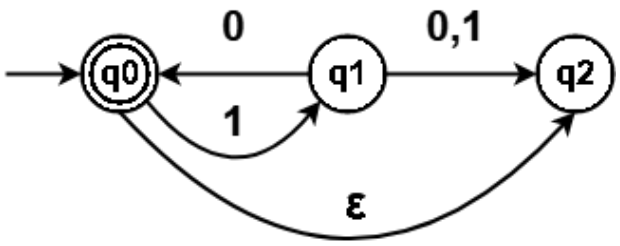
Two finite accepters are said to be equal in power if they both accepts the same language.

DFA and NFA are both exactly equal in power.

Example-

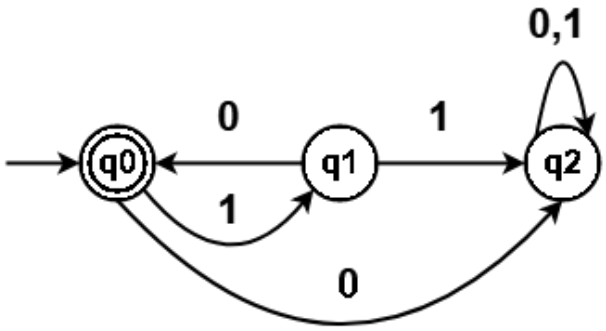
Consider a language L(M) = { $(10)^n$: n >= 0 }

Equivalent NFA for the language L(M) is-



Equivalent NDFA

Equivalent DFA for the language L(M) is-



Equivalent DFA

Both the above automata accepts the same language L(M).

- Thus, both are equal in power.

Important Points

It is important to note the following points-

- Both DFA and NFA are exactly same in power.
- For any regular language, both DFA and NFA can be constructed.
- There exists an equivalent DFA corresponding to every NFA.
- Every NFA can be converted into its equivalent DFA.
- There exists no NFA that can not be converted into its equivalent DFA.
- Every DFA is a NFA but every NFA is not a DFA.

1.8 Acceptability by NDFA and DFA

A string 'w' is said to be accepted by a NFA if

there exists at least one transition path on which we start at initial state and ends at final state.

$\delta^* (q_0, w) = F$

Acceptability by DFA and NDFA

A string is accepted by a DFA/NDFA iff the DFA/NDFA starting at the initial state ends in an accepting state (any of the final states) after reading the string wholly.

A string S is accepted by a DFA/NDFA $(Q, \sum, \delta, q_0, F)$, iff

$\mathbf{\delta^*(q_0, S) \in F}$

The language **L** accepted by DFA/NDFA is

$\mathbf{\{S \mid S \in \sum^* \text{ and } \delta^*(q_0, S) \in F\}}$

A string S′ is not accepted by a DFA/NDFA $(Q, \sum, \delta, q_0, F)$, iff

$\mathbf{\delta^*(q_0, S') \notin F}$

The language L′ not accepted by DFA/NDFA (Complement of accepted language L) is

$\mathbf{\{S \mid S \in \sum^* \text{ and } \delta^*(q_0, S) \notin F\}}$

Let $\mathbf{X = (Q_x, \sum, \delta_x, q_0, F_x)}$ be an NDFA which accepts the language L(X). We have to design an equivalent DFA $\mathbf{Y = (Q_y, \sum, \delta_y, q_0, F_y)}$ such that **L(Y) = L(X)**. The following procedure converts the NDFA to its equivalent DFA −

Algorithm

Input − An NDFA

Output − An equivalent DFA

Step 1 − Create state table from the given NDFA.

Step 2 − Create a blank state table under possible input alphabets for the equivalent DFA.

Step 3 − Mark the start state of the DFA by q0 (Same as the NDFA).

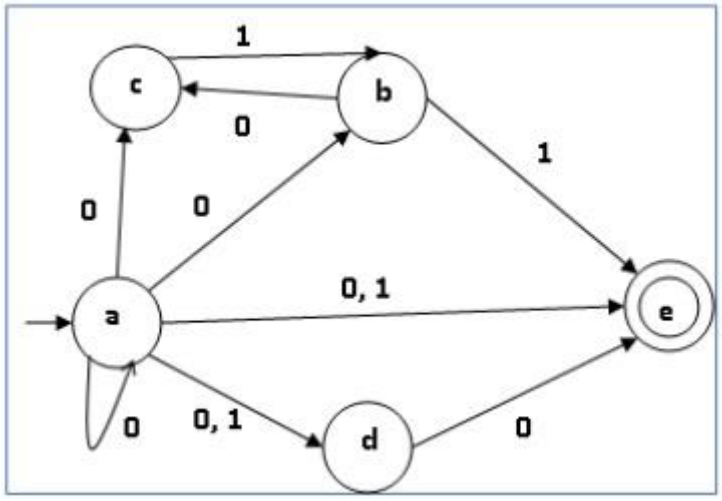
Step 4 − Find out the combination of States $\{Q_0, Q_1, \ldots, Q_n\}$ for each possible input alphabet.

Step 5 − Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

Step 6 − The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

Example

Let us consider the NDFA shown in the figure below.



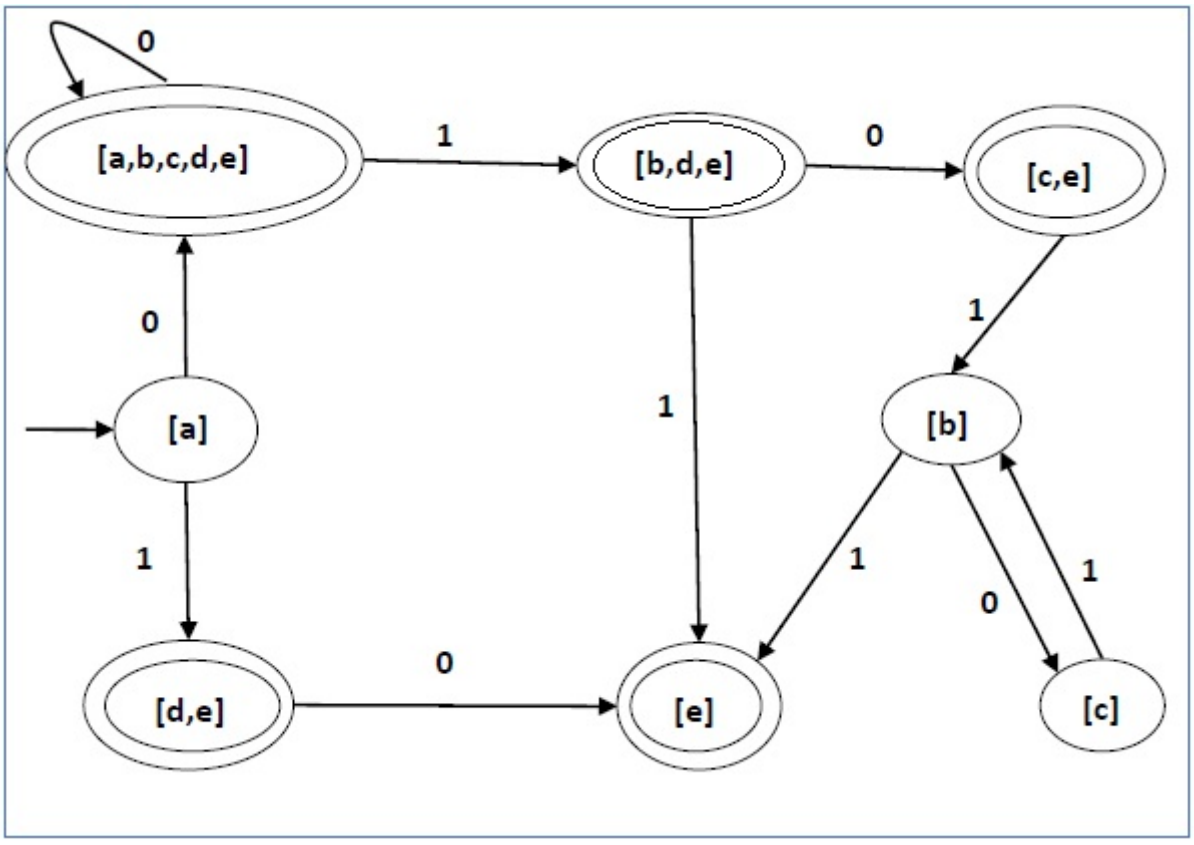
Q	δ(q,0)	δ(q,1)
A	{a,b,c,d,e}	{d,e}
B	{c}	{e}
C	∅	{b}
D	{e}	∅
E	∅	∅

Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

Q	δ(q,0)	δ(q,1)

[a]	[a,b,c,d,e]	[d,e]
[a,b,c,d,e]	[a,b,c,d,e]	[b,d,e]
[d,e]	[e]	∅
[b,d,e]	[c,e]	[e]
[e]	∅	∅
[c, e]	∅	[b]
[b]	[c]	[e]
[c]	∅	[b]

The state diagram of the DFA is as follows −



1.9 Regular Language and Minimization of Finite Automaton

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We will call such a set of languages a **family.** The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the momentwe will simply attach a name to this family.

A language L is called **regular** if and only if there exists some deterministic finite accepter M suchthat

$$L= L(M).$$

Minimization of DFA

Minimization of DFA means reducing the number of states from given FA. Thus, we get the FSM(finite state machine) with redundant states after minimizing the FSM.

We have to follow the various steps to minimize the DFA. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1. 1. δ (q, a) = p
2. 2. δ (r, a) = p

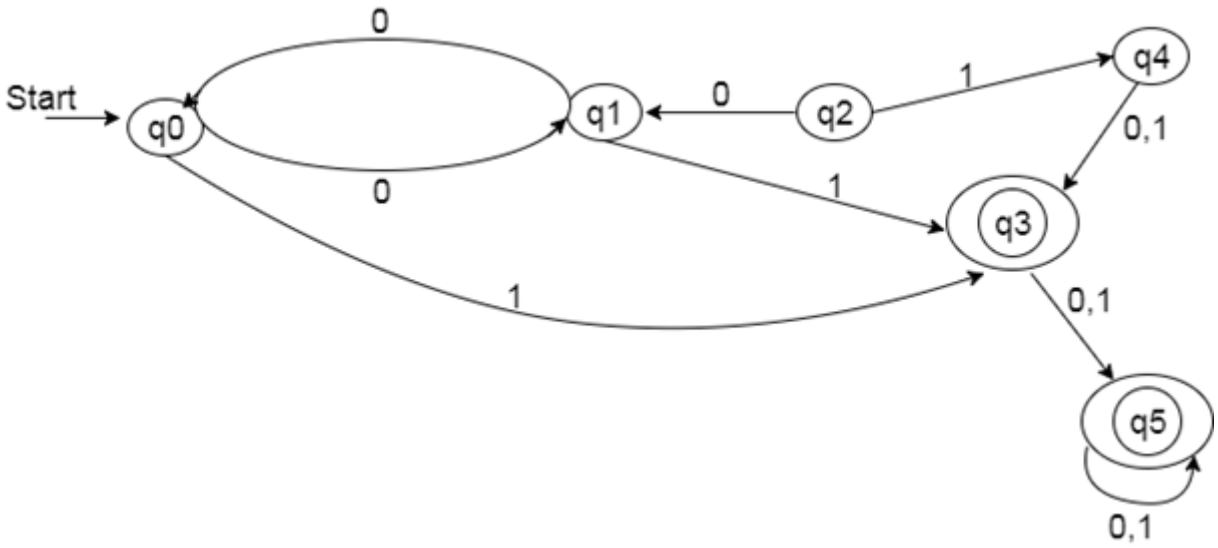
That means, find the two states which have the same value of a and b and remove one of them.

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA

Example:



Solution:

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states.

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q5	q5
*q5	q5	q5

Step 3: Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

State	0	1
q0	q1	q3
q1	q0	q3

3. Another set contains those rows, which starts from final states.
4.

State	0	1
q3	q5	q5
q5	q5	q5

Step 4: Set 1 has no similar rows so set 1 will be the same.

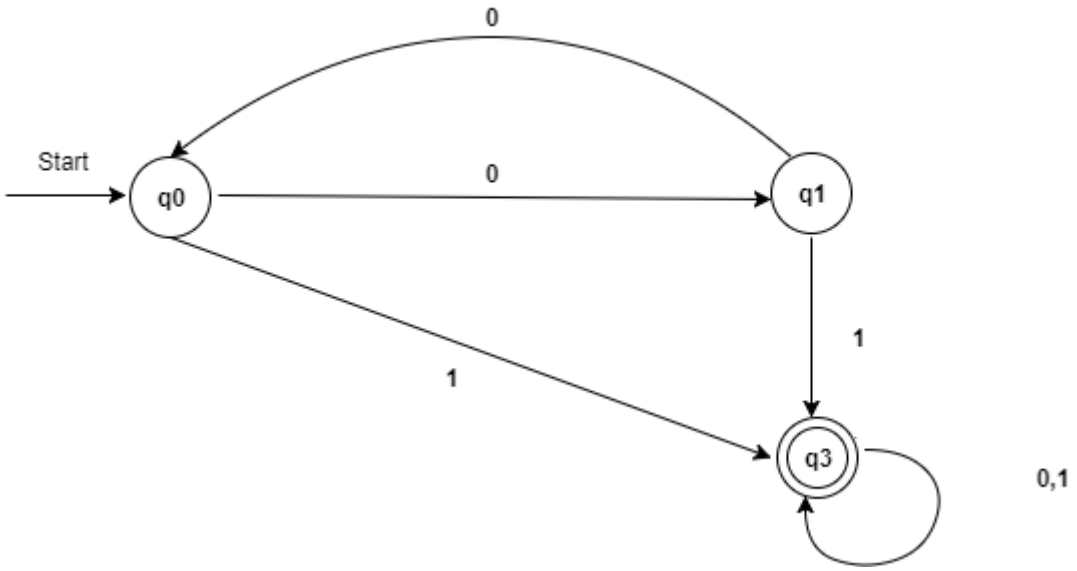
Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

State	0	1
q3	q3	q3

Step 6: Now combine set 1 and set 2 as:

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q3	q3

Now it is the transition table of minimized DFA.



1.10 CHECK YOUR PROGRESS

1. The finite Automaton accept the following Language:
   a) Context free Language b) Regular Language
   c) Context Sensitive Language d) All of the above
2. Which of the following is an application of finite automata?
   a) Text editors b) Lexical analyzers
   c) operating systems d) only (a) and (b)
3. Minimization of finite automaton:
   a) Increase the number of states of finite automaton
   b) Reduce the number of states of finite automaton
   c) convert the NFA to DFA
   d) All of the above
4. In NFA there is
   a) one move to next state for each input symbol
   b) one or more moves to next state for any input symbol
   c) Both (a) and (b)
   d) None
5. Which of the following statement is wrong?
   a) A finite automata has an infinite memory
   b) A finite automata is powerful than all other automata
   c) A finite automata accepts recursively enumerable Languages
   d) All of the above
6. Any transition diagram has an equivalent:
   a) DFA b) NFA

c) Regular Expression d) All of these

7. The Limitation of finite automata is:

a) It is difficult to manage stack in this model

b) It cannot remember arbitrary large amount of information

c) Both (a) and (b)

d) None of these

8. How we can Prove that a given Language is not regular?

a) By using Pumping Lemma

b) We can never prove the regularity of the language accepted by finite automata

c) By using crossing sequence

d) None of these

9. Which of the following are not regular?

a) Strings of a's whose length is perfect square

b) String of palindromes over { a,b}

c) String of a's whose length is a prime no

d) All of the above

10. A transition diagram is also known as?

a) Transition Diagram b) Transition System

c) Both (a) and (b) d) None of these

1.11 SUMMARY

An automaton (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically .An automaton with a finite number of states is

called a **Finite Automaton** (FA) or **Finite State Machine** (FSM). In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton.** In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called **Non-deterministic Automaton**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**. An automaton that computes a Boolean function is called an **acceptor**. All the states of an acceptor is either accepting or rejecting the inputs given to it.

Classifier

A **classifier** has more than two final states and it gives a single output when it terminates.

Transducer

An automaton that produces outputs based on current input and/or previous state is called a **transducer**. Transducers can be of two types −

- **Mealy Machine** − The output depends both on the current state and the current input.
- **Moore Machine** − The output depends only on the current state.

1.12 KEYWORDS

1. **Minimization :- Reduces no of states**
2. **Transition Function :-** Transition function(∂) is a function which maps Q * ∑ into Q . Here 'Q' is set of states and '∑' is input of alphabets..

3.Transition Table:- To show this transition function we use table called transition table. The table takes two values a state and a symbol and returns next state.

A transition table gives the information about –

1. Rows represent different states.
2. Columns represent input symbols.
3. Entries represent the different next state.

4. The final state is represented by a star or double circle.
5. The start state is always denoted by an small arrow.

1.13 SELF ASSESSMENT TEST

Q 1. What is the difference between NDFA and DFA?

Q 2. What are the Limitations of finite Automaton?

Q 3. How we can find a string is accepted or rejected by Finite Automaton?

Q 4. Equivalence Power of NDFA and DFA?

Q 5. Procedure to Convert NDFA to DFA?

Q 6. What do you mean by NFA with null transitions?

Q 7. Write down the difference between Mealy and Moore machine?

Q 8. Describe Dead or Trap state?

Q 9. Write down the algorithm to Minimization of DFA?

Q 10. Explain Regular Language?

1.14 ANSWER TO CHECK YOUR PROGRESS

1. B

2. D

3. B

4. B

5. D

6. D

7. C

8. A

9. A

10. C

1.15 REFERENCES/SUGGESTED READINGS

1.Hopcroaft & O. D. Ullman, R Mothwani, Introduction to automata theory, language & computations, AW, 2001.

2. K. L. P. Mishra & N. Chandrasekaran, . Theory of Computer Sc.(Automata, Languages and computation), PHI, 2000.

3. Peter Linz, Introduction to formal Languages & Automata, Narosa, Publication, 2001.

4. Ramond Greenlaw and H. James Hoover, Fundamentals of the Theory of ComputationPrinciples and Practice, Harcourt India Pvt. Ltd., 1998.

5. H. R. Lewis & C. H. Papaditriou, Elements of theory of Computation, PHC, 1998.

6. John C. Martin, Introduction to Languages and the Theory of Computation, T.M.H., 2003

SUBJECT: Theory of Computation	
COURSE CODE: MCA- 35	AUTHOR: RAVIKA GOEL
LESSON NO. 2	
Regular Expression	

STRUCTURE

2.0 OBJECTIVE

The main objective of this lesson is to define Regular Expression. According to our definition, a language is regular if there exists a finite accepter for it. Therefore, every regular language can be described by some DFA or some NFA. Such a description can be very useful, for example, if we want to show the logic by which we decide if a given string is in a certain language. But in many instances, we need more concise ways of describing regular languages. In this chapter, we look at other ways of representing regular languages. These representations have important practical applications, a matter that is touched on in some of the examples and exercises.

2.1 INTRODUCTION

We first define regular expressions as a means of representing certain subsets of strings over input Alphabet and prove that regular sets are precisely those accepted by finite automata or transition systems. We use pumping lemma for regular sets to prove that certain sets are not regular. We then discuss closure properties of regular sets. we give the relation between regular sets and regular grammars. Finally regular expression is given we have to draw a Finite Automata according to Regular Expression. Draw NFA To DFA conversion by Arden's Method.

2.2 Regular Expression

A **Regular Expression** can be recursively defined as follows −

- **ε** is a Regular Expression indicates the language containing an empty string. **(L (ε) = {ε})**
- **φ** is a Regular Expression denoting an empty language. **(L (φ) = { })**
- **x** is a Regular Expression where **L = {x}**
- If **X** is a Regular Expression denoting the language **L(X)** and **Y** is a Regular Expression denoting the language **L(Y)**, then
    - **X + Y** is a Regular Expression corresponding to the language **L(X) ∪ L(Y)** where **L(X+Y) = L(X) ∪ L(Y)**.

- **X . Y** is a Regular Expression corresponding to the language **L(X) . L(Y)** where **L(X.Y) = L(X) . L(Y)**
- **R*** is a Regular Expression corresponding to the language **L(R*)**where **L(R*) = (L(R))***

- If we apply any of the rules several times from 1 to 5, they are Regular Expressions.

Some RE Examples

Regular Expressions	Regular Set
(0 + 10*)	L = { 0, 1, 10, 100, 1000, 10000, … }
(0*10*)	L = {1, 01, 10, 010, 0010, …}
(0 + ε)(1 + ε)	L = {ε, 0, 1, 01}
(a+b)*	Set of strings of a's and b's of any length including the null string. So L = { ε, a, b, aa , ab , bb , ba, aaa…….}
(a+b)*abb	Set of strings of a's and b's ending with the string abb. So L = {abb, aabb, babb, aaabb, ababb, ……..}
(11)*	Set consisting of even number of 1's including empty string, So L= {ε, 11, 1111, 111111, …….}
(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's , so L = {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, ……..}
(aa + ab + ba + bb)*	String of a's and b's of even length can be obtained by concatenating any combination of the strings aa, ab, ba and bb including null, so L = {aa, ab, ba, bb, aaab, aaba, ……..}

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

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

Expression.

Any set that represents the value of the Regular Expression is called a **Regular Set.**

2.3 Closure Properties of Regular set

Properties of Regular Sets

Property 1. The union of two regular set is regular.

Proof −

Let us take two regular expressions

RE_1 = a(aa)* and RE_2 = (aa)*

So, L_1 = {a, aaa, aaaaa,.....} (Strings of odd length excluding Null)

and L_2 ={ ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 \cup L_2$ = { ε, a, aa, aaa, aaaa, aaaaa, aaaaaa,.....}

(Strings of all possible lengths including Null)

RE ($L_1 \cup L_2$) = a* (which is a regular expression itself)

Hence, proved.

Property 2. The intersection of two regular set is regular.

Proof −

Let us take two regular expressions

RE_1 = a(a*) and RE_2 = (aa)*

So, L_1 = { a,aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 \cap L_2$ = { aa, aaaa, aaaaaa,.....} (Strings of even length excluding Null)

RE ($L_1 \cap L_2$) = aa(aa)* which is a regular expression itself.

Hence, proved.

Property 3. The complement of a regular set is regular.

Proof −

Let us take a regular expression −

RE = (aa)*

So, L = {ε, aa, aaaa, aaaaaa, ...} (Strings of even length including Null)

Complement of **L** is all the strings that is not in **L**.

So, L' = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

RE (L') = a(aa)* which is a regular expression itself.

Hence, proved.

Property 4. The difference of two regular set is regular.

Proof −

Let us take two regular expressions −

RE_1 = a (a*) and RE_2 = (aa)*

So, L_1 = {a, aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 – L_2$ = {a, aaa, aaaaa, aaaaaaa, ...}

(Strings of all odd lengths excluding Null)

RE ($L_1 – L_2$) = a (aa)* which is a regular expression.

Hence, proved.

Property 5. The reversal of a regular set is regular.

Proof −

We have to prove $\mathbf{L^R}$ is also regular if **L** is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

L^R = {10, 01, 11, 01}

RE (L^R) = 01 + 10 + 11 + 10 which is regular

Hence, proved.

Property 6. The closure of a regular set is regular.

Proof −

If L = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)*

L* = {a, aa, aaa, aaaa , aaaaa,……} (Strings of all lengths excluding Null)

RE (L*) = a (a)*

Hence, proved.

Property 7. The concatenation of two regular sets is regular.

Proof −

Let RE_1 = (0+1)*0 and RE_2 = 01(0+1)*

Here, L_1 = {0, 00, 10, 000, 010, ...} (Set of strings ending in 0)

and L_2 = {01, 010,011,.....} (Set of strings beginning with 01)

Then, $L_1 L_2$ = {001,0010,0011,0001,00010,00011,1001,10010,.....}

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)*001(0 + 1)*

Hence, proved

Given R, P, L, Q as regular expressions, the following identities hold −

- ∅* = ε
- ε* = ε
- RR* = R*R
- R*R* = R*
- (R*)* = R*
- RR* = R*R
- (PQ)*P =P(QP)*
- (a+b)* = (a*b*)* = (a*+b*)* = (a+b*)* = a*(ba*)*
- R + ∅ = ∅ + R = R (The identity for union)
- R ε = ε R = R (The identity for concatenation)

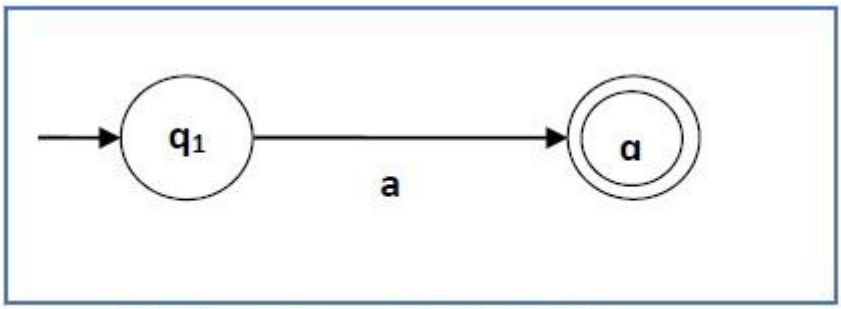
- ∅ L = L ∅ = ∅ (The annihilator for concatenation)
- R + R = R (Idempotent law)
- L (M + N) = LM + LN (Left distributive law)
- (M + N) L = ML + NL (Right distributive law)
- ε + RR* = ε + R*R = R*

2.4 Designing Finite Automaton from Regular Expression

We can use Thompson's Construction to find out a Finite Automaton from a Regular Expression. We will reduce the regular expression into smallest regular expressions and converting these to NFA and finally to DFA.

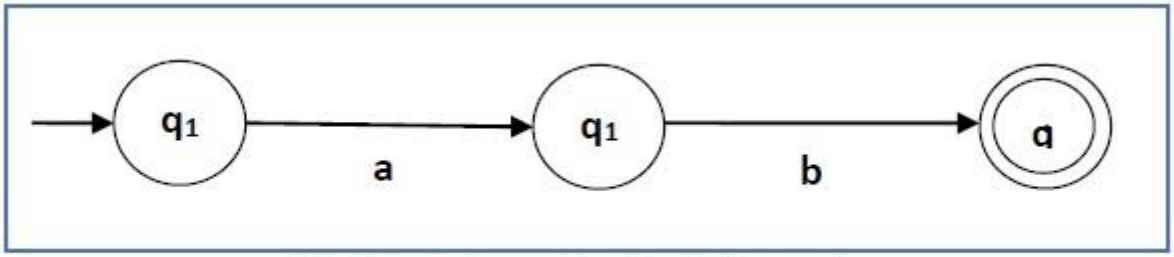
Some basic RA expressions are the following −

Case 1 − For a regular expression 'a', we can construct the following FA −



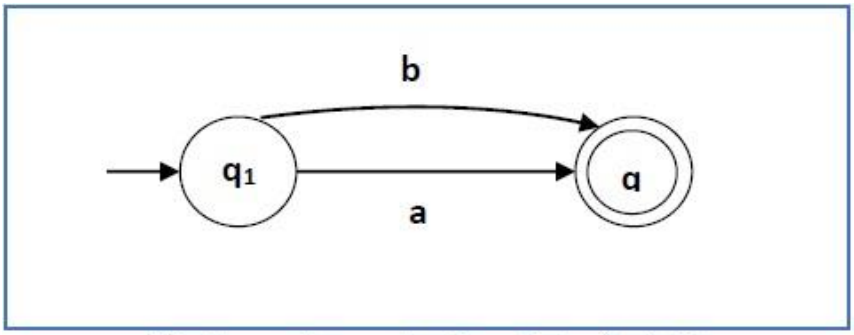
Finite automata for RE = a

Case 2 − For a regular expression 'ab', we can construct the following FA −



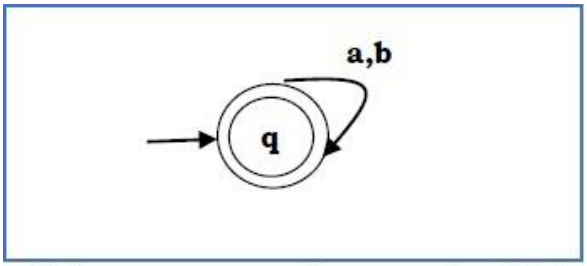
Finite automata for RE = ab

Case 3 − For a regular expression (a+b), we can construct the following FA −



Finite automata for RE= (a+b)

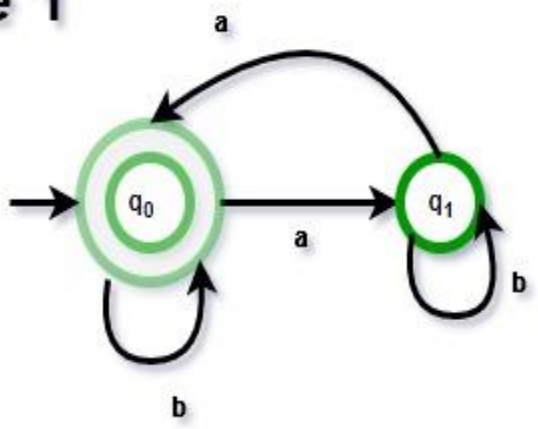
Case 4 − For a regular expression (a+b)*, we can construct the following FA −



Finite automata for RE= (a+b)*

- **Even number of a's :** The regular expression for even number of a's is **(b|ab*ab*)***. We can construct a finite automata as shown in Figure

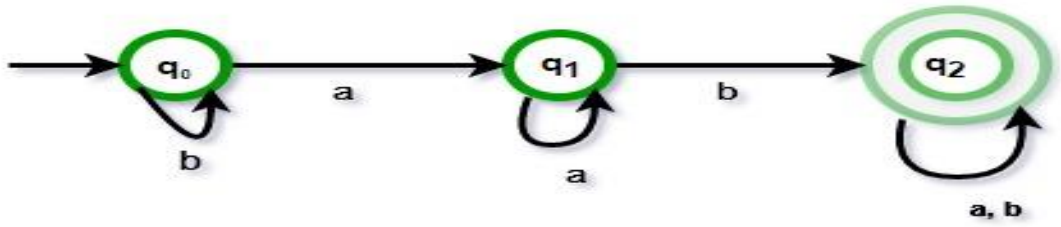
Figure 1



The above automata will accept all strings which have even number of a's. For zero a's, it will be in q0 which is final state. For one 'a', it will go from q0 to q1 and the string will not be accepted. For two a's at any positions, it will go from q0 to q1 for 1st 'a' and q1 to q0 for second 'a'. So, it will accept all strings with even number of a's.

- **String with 'ab' as substring :** The regular expression for strings with 'ab' as substring is **(a|b)*ab(a|b)***. We can construct finite automata as shown in Figure 2.

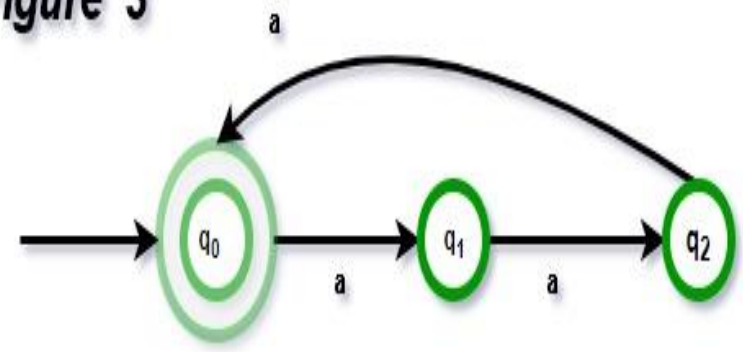
Figure 2



The above automata will accept all string which have 'ab' as substring. The automata will remain in initial state q0 for b's. It will move to q1 after reading 'a' and remain in same state for all 'a' afterwards. Then it will move to q2 if 'b' is read. That means, the string has read 'ab' as substring if it reaches q2.

- **String with count of 'a' divisible by 3 :** The regular expression for strings with count of a divisible by 3 is $\{a^{3n} \mid n >= 0\}$. We can construct automata as shown in Figure 3.

Figure 3



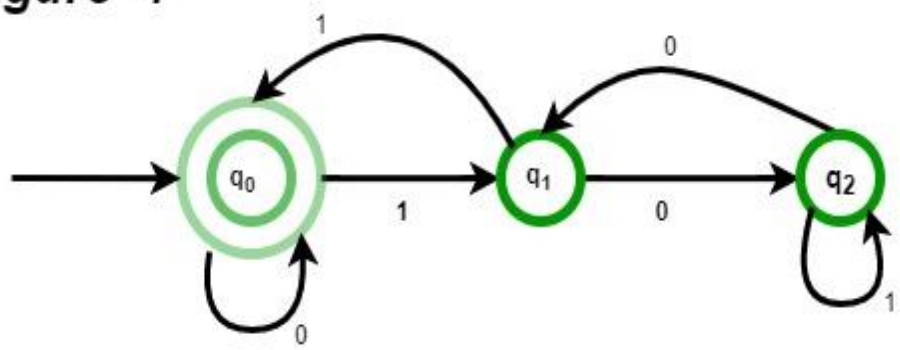
The above automata will accept all string of form a^{3n}. The automata will remain in initial state q0 for ε and it will be accepted. For string 'aaa', it will move from q0 to q1 then q1 to q2 and then q2 to q0. For every set of three a's, it will come to q0, hence accepted. Otherwise, it will be in q1 or q2, hence rejected.

Note : If we want to design a finite automata with number of a's as 3n+1, same automata can be used with final state as q1 instead of q0.
If we want to design a finite automata with language $\{a^{kn} \mid n >= 0\}$, k states are required. We have used k = 3 in our example.

- **Binary numbers divisible by 3 :** The regular expression for binary numbers which are divisible by three is **(0|1(01*0)*1)***. The examples of binary number divisible by 3 are 0, 011, 110, 1001,

1100, 1111, 10010 etc. The DFA corresponding to binary number divisible by 3 can be shown in Figure 4.

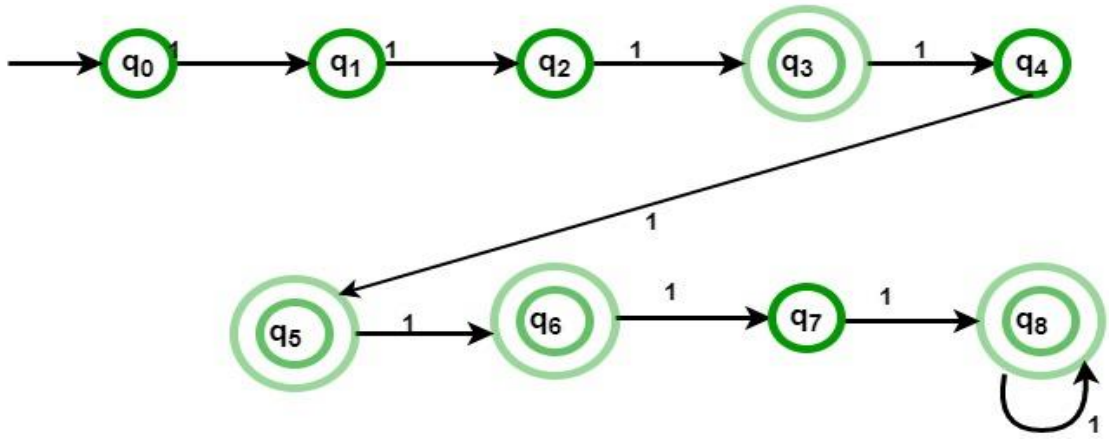
Figure 4



The above automata will accept all binary numbers divisible by 3. For 1001, the automata will go from q0 to q1, then q1 to q2, then q2 to q1 and finally q2 to q0, hence accepted. For 0111, the automata will go from q0 to q0, then q0 to q1, then q1 to q0 and finally q0 to q1, hence rejected.

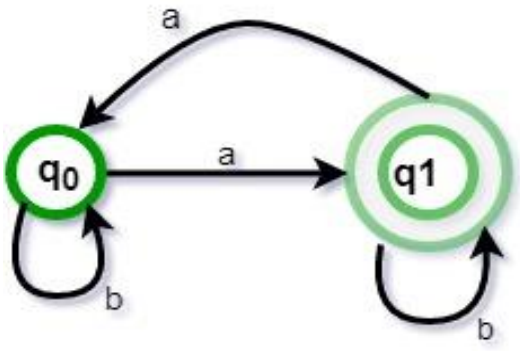
- **String with regular expression (111 + 11111)* :** The string accepted using this regular expression will have 3, 5, 6(111 twice), 8 (11111 once and 111 once), 9 (111 thrice), 10 (11111 twice) and all other counts of 1 afterwards. The DFA corresponding to given regular expression is given in Figure 5.

Figure 5



. What will be the minimum number of states for strings with odd number of a's?

Solution : The regular expression for odd number of a is b*ab*(ab*ab*)* and corresponding automata is given in Figure 6 and minimum number of states are 2.



2.5 Generating Regular Expression from Finite Automaton

In order to find out a regular expression of a Finite Automaton, we use Arden's Theorem along with the properties of regular expressions.

Statement −

Let **P** and **Q** be two regular expressions.

If **P** does not contain null string, then **R = Q + RP** has a unique solution that is **R = QP***

Proof −

R = Q + (Q + RP)P [After putting the value R = Q + RP]

= Q + QP + RPP

When we put the value of **R** recursively again and again, we get the following equation −

$R = Q + QP + QP^2 + QP^3$…..

$R = Q (\varepsilon + P + P^2 + P^3 + \ldots.)$

R = QP* [As P* represents (ε + P + P2 + P3 + ….)]

Hence, proved.

Assumptions for Applying Arden's Theorem

- The transition diagram must not have NULL transitions
- It must have only one initial state

Method

Step 1 − Create equations as the following form for all the states of the DFA having n states with initial state q_1.

$q_1 = q_1R_{11} + q_2R_{21} + \ldots + q_nR_{n1} + \varepsilon$

$q_2 = q_1R_{12} + q_2R_{22} + \ldots + q_nR_{n2}$

……

……

……

……

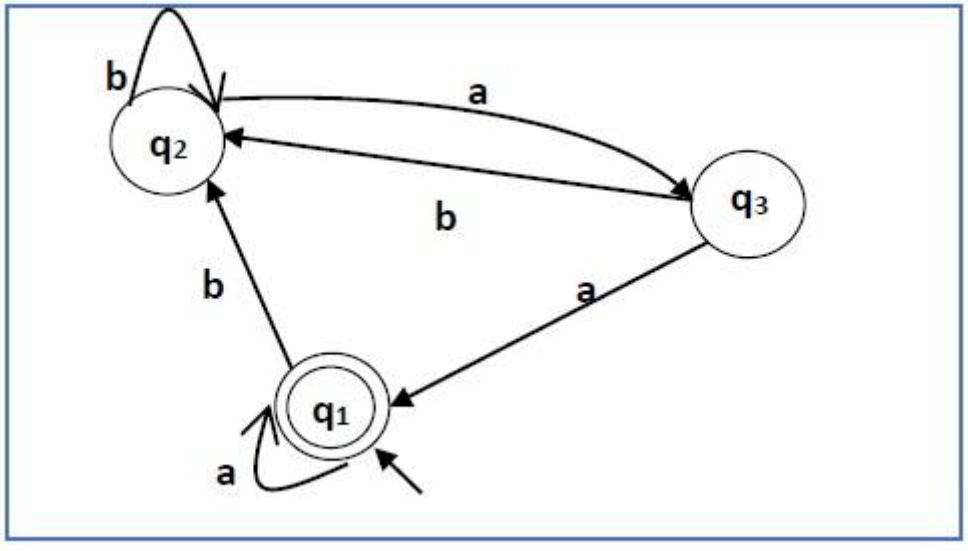
$q_n = q_1R_{1n} + q_2R_{2n} + \ldots + q_nR_{nn}$

$\mathbf{R_{ij}}$ represents the set of labels of edges from $\mathbf{q_i}$ to $\mathbf{q_j}$, if no such edge exists, then $\mathbf{R_{ij} = \emptyset}$

Step 2 − Solve these equations to get the equation for the final state in terms of $\mathbf{R_{ij}}$

Problem

Construct a regular expression corresponding to the automata given below −



Solution −

Here the initial state and final state is $\mathbf{q_1}$.

The equations for the three states q1, q2, and q3 are as follows −

$q_1 = q_1a + q_3a + \varepsilon$ (ε move is because q1 is the initial state0

$q_2 = q_1b + q_2b + q_3b$

$q_3 = q_2a$

Now, we will solve these three equations −

$q_2 = q_1b + q_2b + q_3b$

$= q_1b + q_2b + (q_2a)b$ (Substituting value of q_3)

$= q_1b + q_2(b + ab)$

$= q_1b\ (b + ab)^*$ (Applying Arden's Theorem)

$q_1 = q_1a + q_3a + \varepsilon$

$= q_1a + q_2aa + \varepsilon$ (Substituting value of q_3)

$= q_1a + q_1b(b + ab^*)aa + \varepsilon$ (Substituting value of q_2)

$= q_1(a + b(b + ab)^*aa) + \varepsilon$

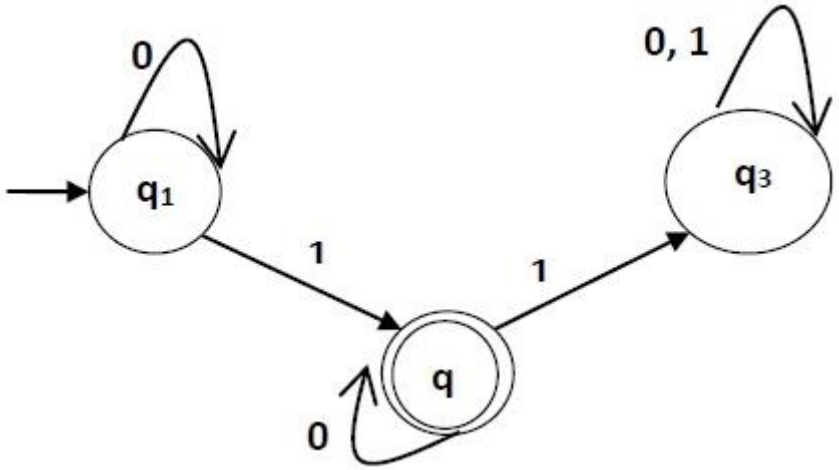
$= \varepsilon\ (a+ b(b + ab)^*aa)^*$

$= (a + b(b + ab)^*aa)^*$

Hence, the regular expression is $(a + b(b + ab)^*aa)^*$.

Problem

Construct a regular expression corresponding to the automata given below −



Solution −

Here the initial state is q_1 and the final state is q_2

Now we write down the equations −

$q_1 = q_10 + \varepsilon$

$q_2 = q_11 + q_20$

$q_3 = q_21 + q_30 + q_31$

Now, we will solve these three equations −

$q_1 = \varepsilon 0^*$ [As, $\varepsilon R = R$]

So, $q_1 = 0^*$

$q_2 = 0^*1 + q_20$

So, $q_2 = 0^*1(0)^*$ [By Arden's theorem]

Hence, the regular expression is 0^*10^*.

2.6 Generating NDFA From regular expression

Method

Step 1 Construct an NFA with Null moves from the given regular expression.

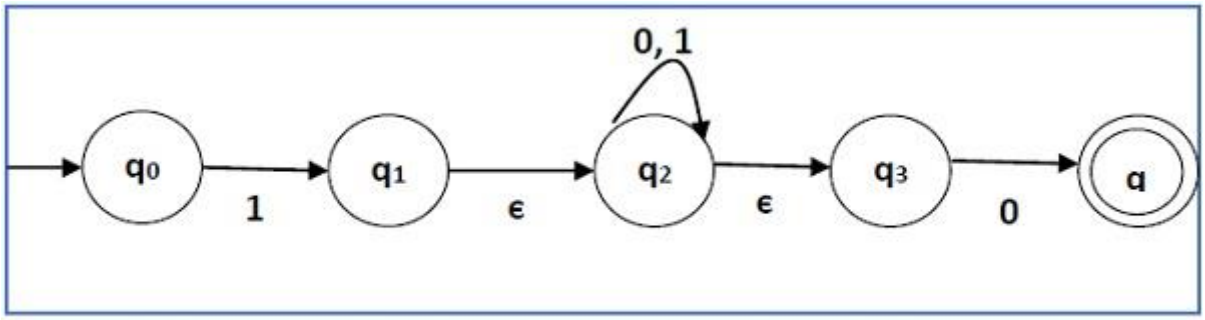
Step 2 Remove Null transition from the NFA and convert it into its equivalent DFA.

Problem

Convert the following RA into its equivalent DFA − 1 (0 + 1)* 0

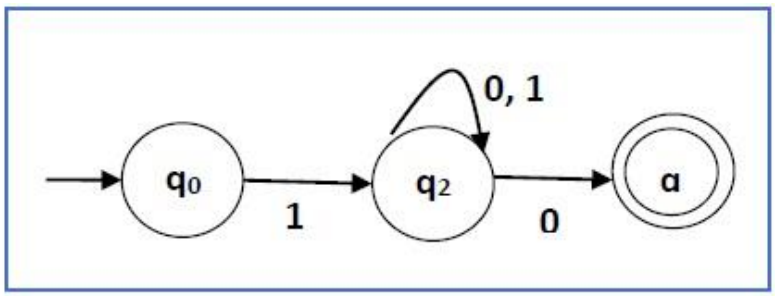
Solution

We will concatenate three expressions "1", "(0 + 1)*" and "0"



NDFA with NULL transition for RA: 1 (0 + 1)* 0

Now we will remove the **ε** transitions. After we remove the **ε** transitions from the NDFA, we get the following −



NDFA without NULL transition for RA: 1 (0 + 1)* 0

It is an NDFA corresponding to the RE − 1 (0 + 1)* 0. If you want to convert it into a DFA, simply apply the method of converting NDFA to DFA discussed in lesson plan 1.

2.7 NFA with null Transition to NFA without Null Transition

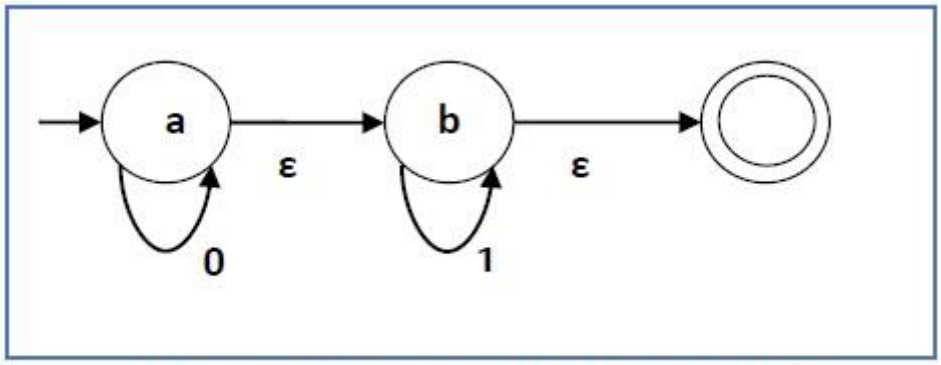
Finite Automata with Null Moves (NFA-ε)

A Finite Automaton with null moves (FA-ε) does transit not only after giving input from the alphabet set but also without any input symbol. This transition without input is called a **null move**.

An NFA-ε is represented formally by a 5-tuple (Q, ∑, δ, q_0, F), consisting of

- **Q** − a finite set of states

- $\sum$ − a finite set of input symbols
- **δ** − a transition function $\delta : Q \times (\sum \cup \{\varepsilon\}) \rightarrow 2^Q$
- $\mathbf{q_0}$ − an initial state $q_0 \in Q$
- **F** − a set of final state/states of Q ($F \subseteq Q$).



Finite automata with Null Moves

The above **(FA-ε)** accepts a string set − {0, 1, 01}

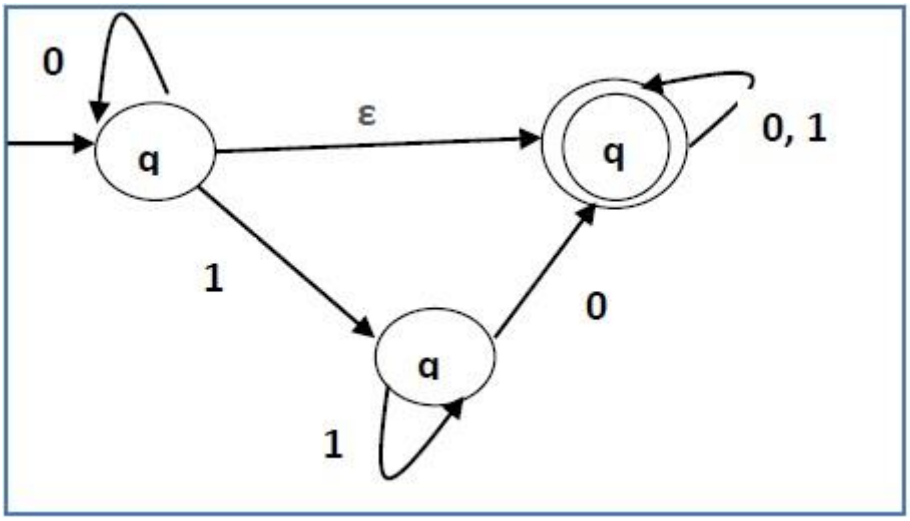
Removal of Null Moves from Finite Automata

If in an NDFA, there is ϵ-move between vertex X to vertex Y, we can remove it using the following steps −

- Find all the outgoing edges from Y.
- Copy all these edges starting from X without changing the edge labels.
- If X is an initial state, make Y also an initial state.
- If Y is a final state, make X also a final state.

Problem

Convert the following NFA-ε to NFA without Null move.



Solution

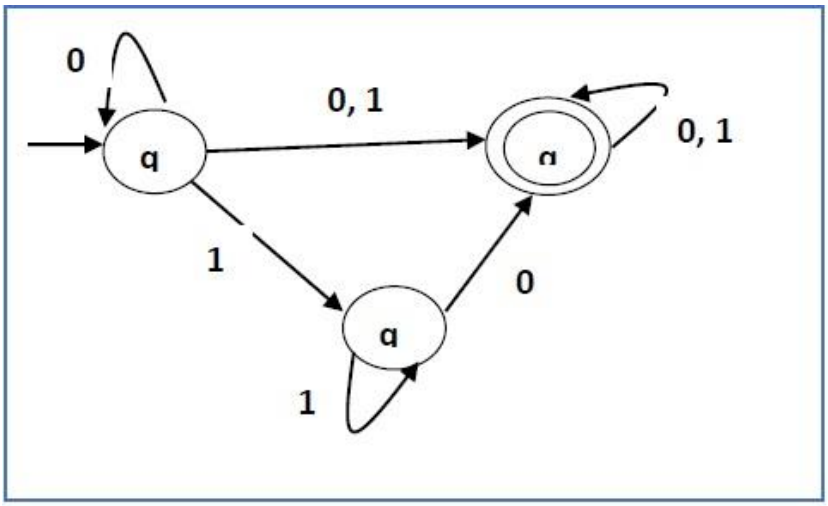
Step 1 −

Here the ε transition is between $\mathbf{q_1}$ and $\mathbf{q_2}$, so let $\mathbf{q_1}$ is **X** and $\mathbf{q_f}$ is **Y**.

Here the outgoing edges from q_f is to q_f for inputs 0 and 1.

Step 2 −

Now we will Copy all these edges from q_1 without changing the edges from q_f and get the following FA −

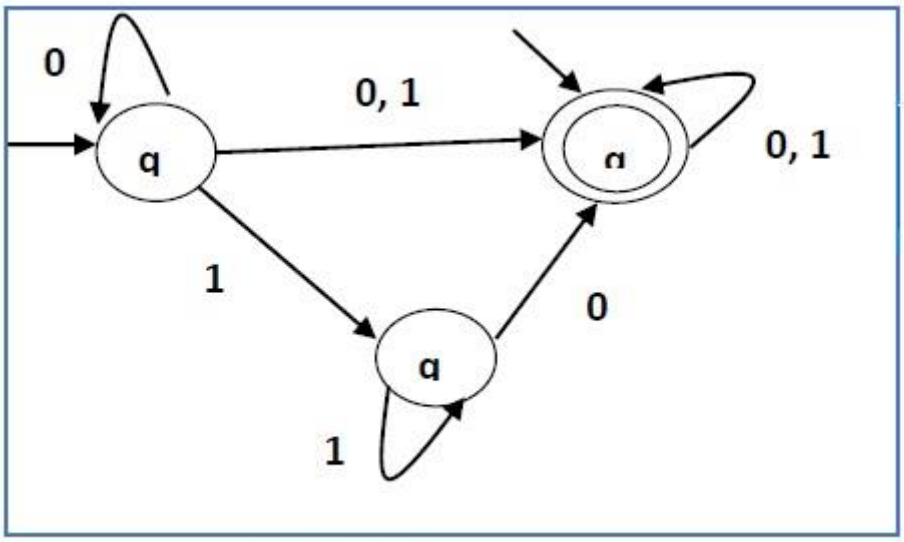


NDFA after step 2

Step 3 −

Here q_1 is an initial state, so we make q_f also an initial state.

So the FA becomes −

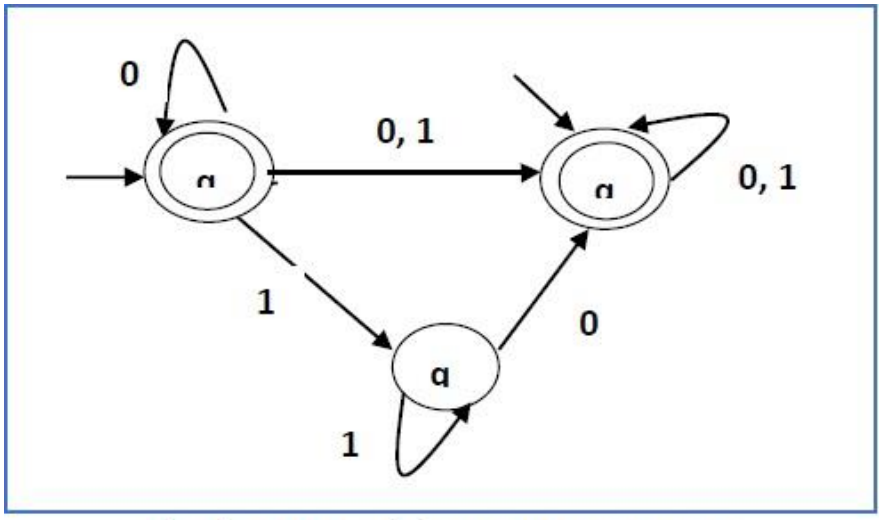


NDFA after Step 3

Step 4 −

Here q_f is a final state, so we make q_1 also a final state.

So the FA becomes −



Final NDFA without NULL moves

2.8 CHECK YOUR PROGRESS

1. The set of aJl strings over {a, b} of even length is represented by the regular expression

a) (ab + aa + bb + ba)* b) (a + b)*(a* + b)*

c) (aa + bb)* d) (ab + ba)*

2. The regular expressions denote zero or more instances of an x or y is

a) (x+y) b) (x+y)*

c) (x* + y) d) (xy)*

3. The regular expressions denote zero or more instances of an x or y is

a) (x+y) b) (x+y)*

c) (x* + y) d) (xy)*

4. The regular expression have all strings of 0′s and 1′s with no two consecutive 0′s is :

a) (0+1) b) (0+1)*

c) (0+∈) (1+10)* d) (0+1)* 011

5. The regular expression with all strings of 0′s and 1′s with atleast two consecutive 0′s, is

a) 1 + (10)* b) (0+1)*00(0+1)*

c) (0+1)*011 d) 0*1*2*

6. Let L={w ∈ (0 + 1)*|w has even number of 1s}, i.e. L is the set of all bit strings with even number of 1s. Which one of the regular expression below represents L?

a) (0*10*1)* b) 0*(10*10*)*

c) 0*(10*1*)*0* d) 0*1(10*1)*10*

7. Which of the following operation is applied on Regular expression

a) Union b) Concatenation

c) Closure d) all of these

8. Regular Expression are used to Represent which language

a) Recursive Language b) Context free Language

c) Regular Language d) all of these

9 .Which of the following String can be obtained by the language L = {ai b 2i / i >=1} a) a)aaabbbbbb

b) aabbb

c) abbabbba d) aaaabbbabb.

10. Regular expression (x/y)(x/y) denotes the set

a) {xy,xy} b) {xx,xy,yx,yy}

c) {x,y} d){x,y,xy

2.9 SUMMARY

We first define regular expressions as a means of representing certain subsets of strings over input and prove that regular sets are precisely those accepted by finite automata or transition systems. We use pumping lemma for regular sets to prove that certain sets are not regular. We then discuss closure properties of regular sets. Finally. we give the relation between regular sets and regular grammars.

2.10 KEYWORD

1. **Regular Expression** -A regular expression, often called a *pattern*, specifies a set of strings required for a particular purpose. A simple way to specify a finite set of strings is to list its elements or members.
2. **Regular set: -Any set that represents the value of the Regular Expression** is called a Regular Set. The union of two regular set is regular.
3. **Regular Language:** a regular language is a formal language (i.e., a possibly infinite set of finite sequences of symbols from a finite alphabet) that satisfies the following equivalent properties:
    - it can be accepted by a deterministic finite state machine.
    - it can be accepted by a nondeterministic finite state machine
    - it can be described by a formal regular expression.

2.11 SELF ASSESSMENT TEST

Q1. State the relations among regular expression, deterministic finite automata, nondeterministic finite automaton and finite automaton with epsilon transition.

Q 2. Finite Automata is a 5 tuples denoted by A = (Q, ∑, δ, q0, F) where • Q is a finite set of states • ∑ is the finite set of input symbols • δ is a transition function (Q X ∑ → Q) • q0 is the start state or initial state • F is a set of final or accepting states

Regular Expression for the set of strings over {0.1} that have atleast one.

Q 3. Give the regular expression for the set of all strings ending in 00.

Q 4. What are the applications of regular expression?

Q 5. Write regular expressions for the following. (i)Binary numbers that are multiple of 2. . (ii)Strings of a‟s and b‟s with no consecutive (iii) Strings of a‟s and b‟s containing consecutive a‟.

Q 6. Let P and Q be two regular expressions over ∑. If P does not contain null string ε over ∑ then R=Q+RP, it has the solution R=QP*.

Q 7. Give a regular expression for the set of all strings having odd number of 1's

2.12 ANSWER TO CHECK YOUR PROGRESS

1 **A**

2 **B**

3 **D**

4 **C**

5 **B**

6 **B**

7 **D**

8 **C**

9 **A**

10 **B**

2.13 REFRENCES/ SUGGESTED READINGS

1. Hopcroaft & O. D. Ullman, R Mothwani, Introduction to automata theory, language & computations, AW, 2001.

2. K. L. P. Mishra & N. Chandrasekaran, . Theory of Computer Sc.(Automata, Languages and computation), PHI, 2000.

3. Peter Linz, Introduction to formal Languages & Automata, Narosa, Publication, 2001.

4. Ramond Greenlaw and H. James Hoover, Fundamentals of the Theory of ComputationPrinciples and Practice, Harcourt India Pvt. Ltd., 1998.

5. H. R. Lewis & C. H. Papaditriou, Elements of theory of Computation, PHC, 1998.

6. John C. Martin, Introduction to Languages and the Theory of Computation, T.M.H., 2003

SUBJECT: Theory of Coputation	
COURSE CODE: MCA-35	AUTHOR: RAVIKA GOEL
LESSON NO. 3	
Mealy and Moore Machine	

STRUCTURE

3.0 OBJECTIVE

The main objective of this lesson is to understand the Finite State Machine. Finite State Machine is of two Types: Mealy and Moore Machines. We Discuss the Procedure of Conversion of Mealy to Moore Machine and Moore Machine to Mealy Machine. We check the equivalence Power of Mealy and Moore Machine.

3.1 INTRODUCTION

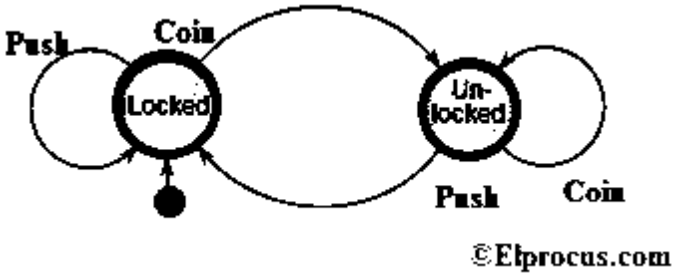
In the theory of computation, a Mealy machine is a finite-state machine whose output values are determined both by its current state and the current inputs. This is in contrast to a Moore machine, whose output values are determined solely by its current state. A Mealy machine is a deterministic finite-state transducer: for each state and input, at most one transition is possible. The Mealy machine is named after George H. Mealy, who presented the concept in a 1955 paper, "A Method for Synthesizing Sequential Circuits".

3.2 Finite State Machines

(i) The finite state machines (FSMs) are significant for understanding the decision making logic as well as control the digital systems. In the FSM, the outputs, as well as the next state, are a present state and the input function. This means that the selection of the next state mainly depends on the input value and strength lead to more compound system performance. As in sequential logic, we require the past inputs history for deciding the output. Therefore FSM proves very cooperative in understanding sequential logic roles. Basically, there are two methods for arranging a sequential logic design namely mealy machine as well as more machine. This article discusses the theory and implementation of a finite state machine or FSM, types, finite state machine examples, advantages, and disadvantages.

(ii) What is an FSM (Finite State Machine)?

(iii) The **definition of a finite state machine is**, the term finite state machine (FSM) is also known as **finite state** automation. FSM is a calculation model that can be executed with the help of hardware otherwise software. This is used for creating sequential logic as well as a few computer programs. FSMs are used to solve the problems in fields like mathematics, games, linguistics, and artificial intelligence. In a system where specific inputs can cause specific changes in state that can be signified with the help of FSMs.



(iv) Finite State Machine

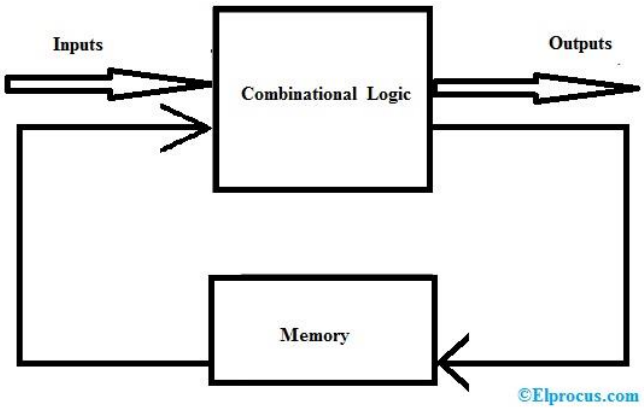
(v) This **finite state machine diagram** explains the various conditions of a turnstile. Whenever placing a coin into a turnstile will unbolt it, and after the turnstile has been pressed, it bolts gain. Placing a coin into an unbolted turnstile, otherwise pressing against a bolted turnstile will not alter its state.

(vi) Types of Finite State Machine

(vii) The finite state machines are classified into two types such as **Mealy state machine** and **Moore state machine**.

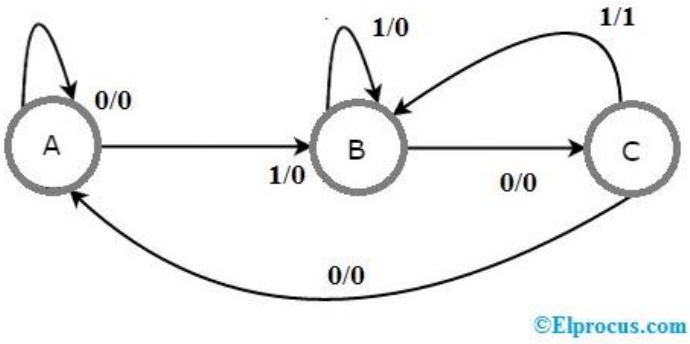
(viii) Mealy State Machine

(ix) When the outputs depend on the current inputs as well as states, then the FSM can be named to be a mealy state machine. The following diagram is the **mealy state machine block diagram**. The mealy state machine block diagram consists of two parts namely combinational logic as well as memory. The memory in the machine can be used to provide some of the previous outputs as combinational logic inputs.



(x) Mealy State Machine Block Diagram

(xi) Based on the current inputs as well as states, this machine can produce outputs. Thus, the outputs can be suitable only at positive otherwise negative of the CLK signal. The mealy state machine's state diagram is shown below.



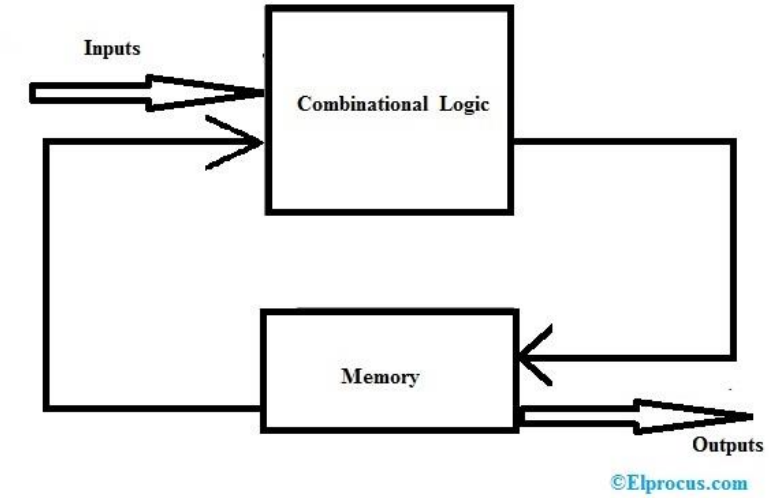
(xii) State Diagram of Mealy State Machine

(xiii) The state diagram of mealy state machine mainly includes three states namely A, B, and C. These three states are tagged within the circles as well as every circle communicates with one state. Conversions among these three states are signified by directed lines. In the above diagram, the inputs and outputs are denoted with 0/0, 1/0, and 1/1. Based on the input value, there are two conversions from every state.

(xiv) Generally, the amount of required states in the mealy machine is below or equivalent to the number of required states in Moore state machine. There is an equal Moore state machine for every Mealy state machine. As a result, based on the necessity we can employ one of them.

(xv) Moore State Machine

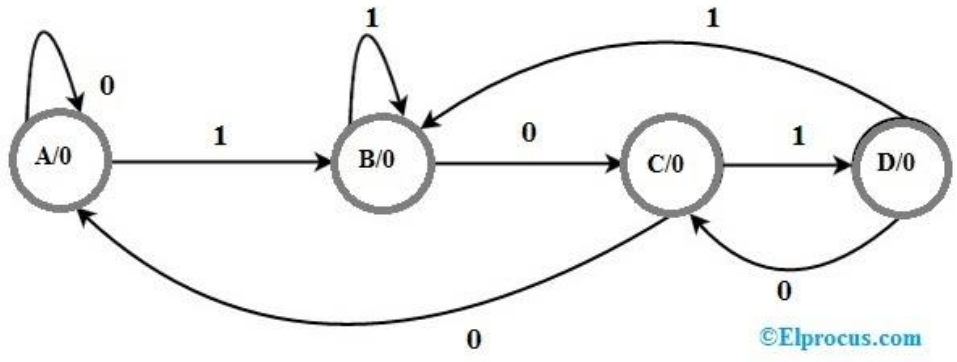
(xvi) When the outputs depend on current states then the FSM can be named as **Moore state machine**. The **Moore state machine's block diagram** is shown below. The Moore state machine block diagram consists of two parts namely combinational logic as well as memory.



(xvii) Moore State Machine Block Diagram

(xviii) In this case, the current inputs, as well as current states, will decide the next states. Thus, depending on further states, this machine will generate the outputs. So, the outputs of this will be applicable simply after the conversion of the state.

(xix) The Moore state machine state diagram is shown below. In the above state, the diagram includes four states like a mealy state machine namely A, B, C, and D. the four states as well as individual outputs are placed in the circles.



(xx) State Diagram of Moore State Machine

(xxi) In the above figure, there are four states, namely A, B, C & D. These states and the respective outputs are labeled inside the circles. Here, simply the input worth is marked on every conversion. In the above figure includes two conversions from every state depending on the input value.

(xxii) Generally, the amount of required states in this machine is greater than otherwise equivalent to the required number of states in the mealy state machine

(xxiii) Generally, the number of required states in this machine is more than otherwise equivalent to the required states in MSM (Mealy state machine). For every Moore state machine, there is a corresponding Mealy state machine. Consequently, depending on the necessity we can utilize one of them.

There is an equal mealy state machine for every Moore state machine. As a result, based on the necessity we can employ one of them.

3.3 Applications of Finite State Machines

The **finite state machine applications** mainly include the following.

FSMs are used in games; they are most recognized for being utilized in artificial intelligence, and however, they are also frequent in executions of navigating parsing text, input handling of the customer, as well as network protocols.

These are restricted in computational power; they have the good quality of being comparatively simple to recognize. So, they are frequently used by software developers as well as system designers for summarizing the performance of a difficult system. The finite state machines are applicable in vending machines, video games, traffic lights, controllers in CPU, text parsing, analysis of protocol, **recognition of speech**, language processing, etc.

3.4 Advantages and Disadvantage of FSM

Advantages of Finite State Machine

The **advantages of Finite State Machine** include the following.

- Finite state machines are flexible
- Easy to move from a significant abstract to a code execution
- Low processor overhead
- Easy determination of reachability of a state

Disadvantages of Finite State Machine

The **disadvantages of the finite state machine** include the following

- The expected character of deterministic finite state machines can be not needed in some areas like computer games
- The implementation of huge systems using FSM is hard for managing without any idea of design.
- Not applicable for all domains
- The orders of state conversions are inflexible.

3.5 Mealy and Moore Machine

Mealy Machine – A mealy machine is defined as a machine in theory of computation whose output values are determined by both its current state and current inputs. In this machine atmost one transition is possible.

It has 6 tuples: (Q, q0, ∑, O, δ, λ')

Q is finite set of states

q0 is the initial state

∑ is the input alphabet

O is the output alphabet

δ is transition function which maps Q×∑ → Q

'λ' is the output function which maps Q×∑→ O

Diagram –

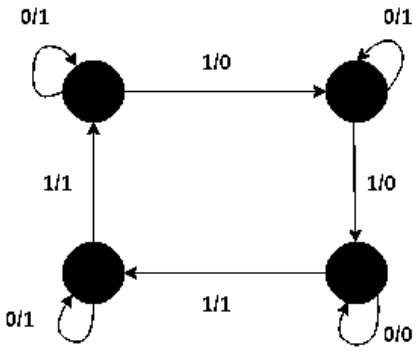


Figure - Mealy machine

Moore Machine – A moore machine is defined as a machine in theory of computation whose output values are determined only by its current state.

It has also 6 tuples: (Q, q0, ∑, O, δ, λ)

Q is finite set of states

q0 is the initial state

∑ is the input alphabet

O is the output alphabet

δ is transition function which maps Q×∑ → Q

λ is the output function which maps Q → O

Diagram –

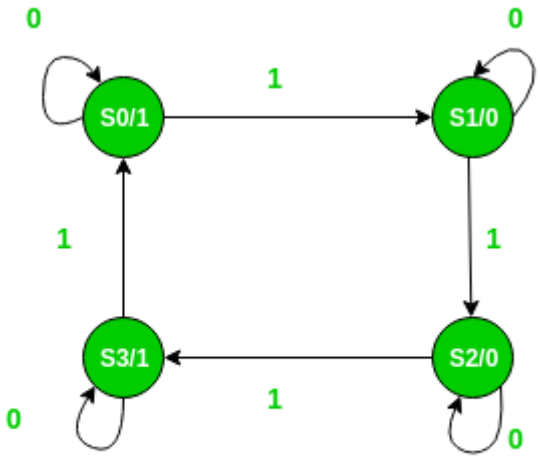


Figure - Moore machine

Moore Machine –

1. Output depends only upon present state.
2. If input changes, output does change.
3. More number of states are required.
4. There is less hardware requirement for circuit implementation.
5. They react slower to inputs(One clock cycle later).
6. Synchronous output and state generation.

7. Output is placed on states.
8. Easy to design.

Mealy Machine –

1. Output depends on present state as well as present input.
2. If input changes, output also changes.
3. Less number of states are required.
4. There is more hardware requirement for circuit implementation.
5. They react faster to inputs.
6. Asynchronous output generation.
7. Output is placed on transitions.

3.6 Conversion of Mealy to Moore Machine

Conversion from Mealy machine to Moore Machine

In Moore machine, the output is associated with every state, and in Mealy machine, the output is given along the edge with input symbol. To convert Moore machine to Mealy machine, state output symbols are distributed to input symbol paths. But while converting the Mealy machine to Moore machine, we will create a separate state for every new output symbol and according to incoming and outgoing edges are distributed.

The following steps are used for converting Mealy machine to the Moore machine:

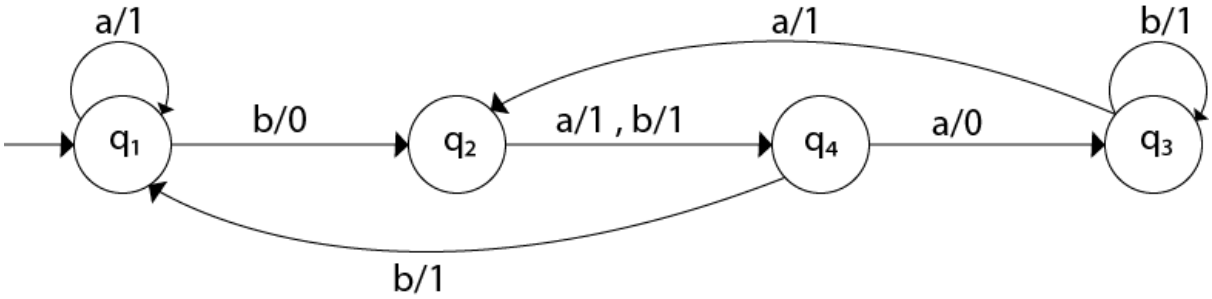
Step 1: For each state(Qi), calculate the number of different outputs that are available in the transition table of the Mealy machine.

Step 2: Copy state Qi, if all the outputs of Qi are the same. Break qi into n states as Qin, if it has n distinct outputs where n = 0, 1, 2....

Step 3: If the output of initial state is 0, insert a new initial state at the starting which gives 1 output.

Example 1:

Convert the following Mealy machine into equivalent Moore machine.



Solution:

Transition table for above Mealy machine is as follows:

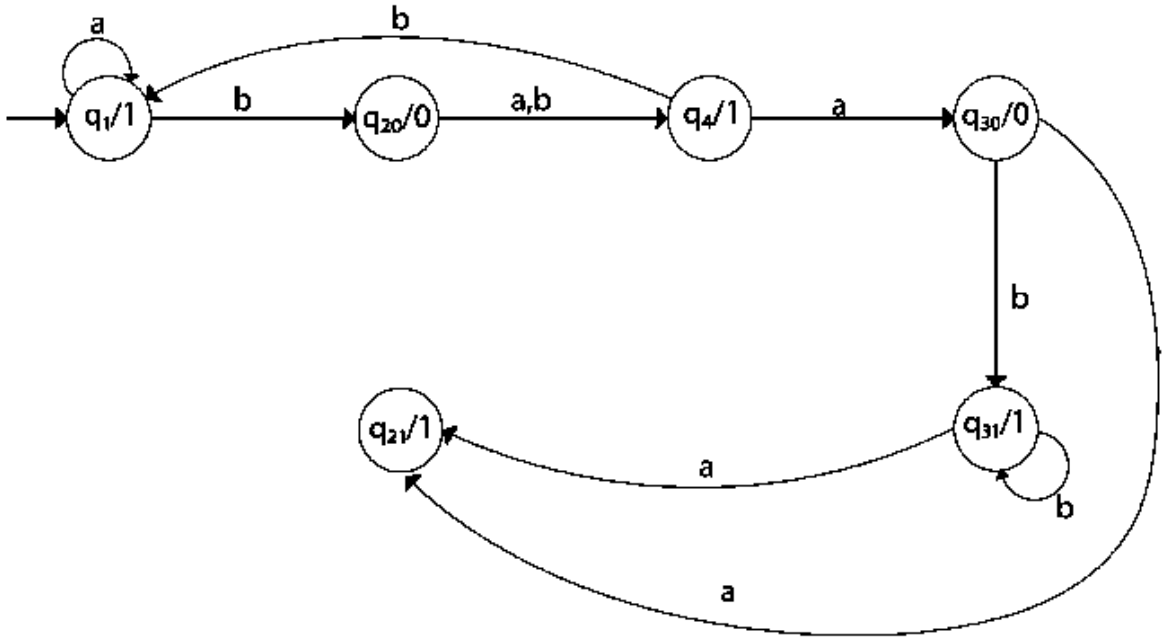
Present State	Next State			
	a		b	
	State	O/P	State	O/P
q_1	q_1	1	q_2	0
q_2	q_4	1	q_4	1
q_3	q_2	1	q_3	1
q_4	q_3	0	q_1	1

- For state q1, there is only one incident edge with output 0. So, we don't need to split this state in Moore machine.
- For state q2, there is 2 incident edge with output 0 and 1. So, we will split this state into two states q20(state with output 0) and q21(with output 1).
- For state q3, there is 2 incident edge with output 0 and 1. So, we will split this state into two states q30(state with output 0) and q31(state with output 1).
- For state q4, there is only one incident edge with output 0. So, we don't need to split this state in Moore machine.

Transition table for Moore machine will be:

Present State	Next State		Output
	a=0	a=1	
q_1	q_1	q_2	1
q_{20}	q_4	q_4	0
q_{21}	ø	ø	1
q_{30}	q_{21}	q_{31}	0
q_{31}	q_{21}	q_{31}	1
q_4	q_3	q_4	1

Transition diagram for Moore machine will be:



In the Moore machine, the output is associated with every state, and in the mealy machine, the output is given along the edge with input symbol. The equivalence of the Moore machine and Mealy machine means both the machines generate the same output string for same input string.

3.7 Conversion of Moore to Mealy Machine

We cannot directly convert Moore machine to its equivalent Mealy machine because the length of the Moore machine is one longer than the Mealy machine for the given input. To convert Moore machine to Mealy machine, state output symbols are distributed into input symbol paths. We are going to use the following method to convert the Moore machine to Mealy machine.

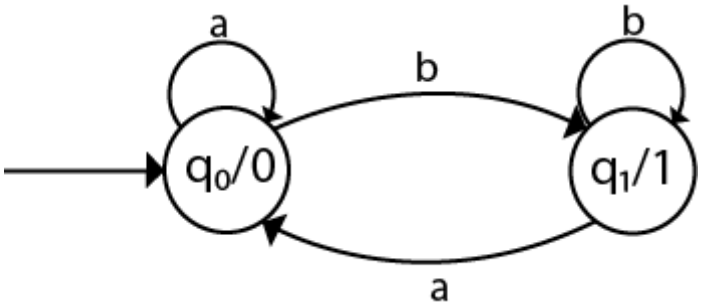
Method for conversion of Moore machine to Mealy machine

Let M = (Q, ∑, δ, λ, q0) be a Moore machine. The equivalent Mealy machine can be represented by M' = (Q, ∑, δ, λ', q0). The output function λ' can be obtained as:

1. λ' (q, a) = λ(δ(q, a))

Example 1:

Convert the following Moore machine into its equivalent Mealy machine.



Solution:

The transition table of given Moore machine is as follows:

Q	A	B	Output(λ)
q0	q0	q1	0
q1	q0	q1	1

The equivalent Mealy machine can be obtained as follows:

1. λ' (q0, a) = λ(δ(q0, a))
2. = λ(q0)
3. = 0

4.

5. λ' (q0, b) = λ(δ(q0, b))

6. = λ(q1)

7. = 1

The λ for state q1 is as follows:

1. λ' (q1, a) = λ(δ(q1, a))

2. = λ(q0)

3. = 0

4.

5. λ' (q1, b) = λ(δ(q1, b))

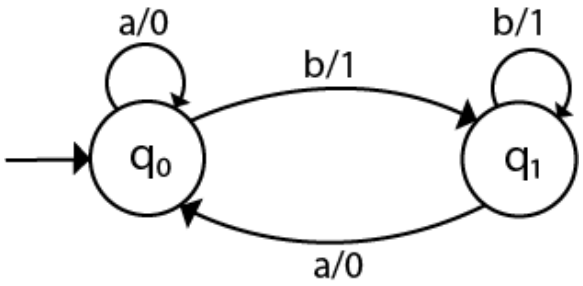
6. = λ(q1)

7. = 1

Hence the transition table for the Mealy machine can be drawn as follows:

Σ / Q	Input 0		Input 1	
	State	O/P	State	O/P
q_0	q_0	0	q_1	1
q_1	q_0	0	q_1	1

The equivalent Mealy machine will be,



Note: The length of output sequence is 'n+1' in Moore machine and is 'n' in the Mealy machine.

The equivalence of the Moore machine and Mealy machine means **both the machines generate the same output string for same input string**. We cannot directly convert Moore machine to its equivalent Mealy machine because the length of the Moore machine is one longer than the Mealy machine for the given input.

3.8 Check your Progress

1. For input null ,the output produced by a Mealy machine is

   a) Null b) Dependent on present state

   c) Depends on given machine d) Cannot decide

2. Which of the following statement is true?
   a) Moore Machine consists of six tuples.
   b) Mealy Machine Consists of six tuples.
   c) Both (a) and (b)
   d) None of these

3. In Moore Machine is of lemgth n then the output string length will be

   a) N+1 b) n

   c) n+n d) None of these

4. Mealy and Moore Machines are also called

   a) Turing machines b) Transducer

   c)Linear bounded Automaton d) All of these

5. In Moore Machine the output depends on

   a) only present state b) present state and present input

   c) nothing d) Type of input

6. In Mealy Machine the output depends on

   a) only present state b) present state and present input

c) nothing d) Type of input

7. Which of the following statement is true

a) A Mealy machine has no terminating state

b) A Moore machine has no terminating state

c) conversion from Mealy into Moore machine and vice versa is possible

d) All of the above

8. In Mealy Machine is of lemgth n then the output string length will be

a) N+1 b) n

c) n+n d) None of these

9. Finite state machine_____recognize palindromes.

a) Can b) Cannot

c) May d) May not

10. Moore machine accepts a string of length k; the length of the output string is

a) k b) 2k

c) k + 1 d) k - 1

3.9 Summary

Mealy and Moore Machines are also called Transducer. The Transducers are capable of Producing strings of symbols as output. Mealy and Moore Machines are commonly used to describe the behaviour of sequential circuits that includes flip flops and feedback electronic devices for which the output of the circuits is not only a function of the specific input but also a function of previous state. These machines differ in how they determine its output.

3.10 Keywords

1. Mealy Machine: A Mealy machine is a machine in which output symbol depends upon the present input symbol and present state of the machine.

2. Moore Machine: Moore machine is a finite state machine in which the next state is decided by the current state and current input symbol.
3. Combinational Logic: The combinational logic is the **process of combining logic gates** to process the given two or more inputs such that to generate at least one output signal based on the logic function of each logic gate.
4. Memory: refers to the psychological processes of acquiring, storing, retaining, and later retrieving information.

3.11 Self Assessment Test

Q1. Design a mealy machine which gives a output y if the input string contains a sequence which ends in 101 and output N if it ends with 110 otherwise Z.

Q2. Design a mealy machine to convert each occurrence of substring 110 by 101

Q3. Design a mealy machine which gives output Y if the input string contains the sequence 101 otherwise gives output N.

Q4. Write down the Difference between Mealy and Moore Machine.

Q5. Design a Mealy machine for the input from (0+1+2)* which prints the residue module five of the input string treated as ternary (base 0,1,2,3) number.

Q6. Design a mealy machine over the input alphabet ={0,1} which checks whether the no of 1's are even or odd.

Q7. Prove the equivalence of Moore and Mealy Machine.

Q8. Write down the algorithm for Mealy machine into Moore machine

Q9. Write down the algorithm for Moore machine into Mealy machine.

Q10. Design a Moore and Mealy machine for the input from (a+b)*, if the input ends in 'bab' output X, if the input ends in 'bba ' output y otherwise output Z

3.12 Answer to check your Progress

1. **A**
2. **C**
3. **A**

4. B

5. A

6. B

7. D

8. A

9. B

10.C

3.13 REFERENCES/SUGGESTED READINGS

1.Hopcroaft & O. D. Ullman, R Mothwani, Introduction to automata theory, language & computations, AW, 2001.

2. K. L. P. Mishra & N. Chandrasekaran, . Theory of Computer Sc.(Automata, Languages and computation), PHI, 2000.

3. Peter Linz, Introduction to formal Languages & Automata, Narosa, Publication, 2001.

4. Ramond Greenlaw and H. James Hoover, Fundamentals of the Theory of ComputationPrinciples and Practice, Harcourt India Pvt. Ltd., 1998.

5. H. R. Lewis & C. H. Papaditriou, Elements of theory of Computation, PHC, 1998.

6. John C. Martin, Introduction to Languages and the Theory of Computation, T.M.H., 2003

SUBJECT:Theory of Coputation	
COURSE CODE: MCA-35	AUTHOR: RAVIKA GOEL
LESSON NO. 4	
Pumping Lemma for Regular sets	

STRUCTURE

4.0 OBJECTIVE

The main objective of this lesson is to study the Pumping Lemma for Regular sets. What are the applications of Pumping Lemma . What are the Closure Properties of Regular sets.

Discuss the Myhill-Nerode Theorm & Minimization Algorithm.

4.1 INTRODUCTION

In the theory of formal languages, the **pumping lemma for regular languages** is a lemma that describes an essential property of all regular languages. Informally, it says that all sufficiently long strings in a regular language may be pumped—that is, have a middle section of the string repeated an arbitrary number of times—to produce a new string that is also part of the language. The pumping lemma is useful for disproving the regularity of a specific language in question. It was first proven by Michael Rabin and Dana Scott in 1959, and rediscovered shortly after by Yehoshua Bar-Hillel, Micha A. Perles, and Eli Shamir in 1961, as a simplification of their pumping lemma for context-free languages. The pumping lemma is often used to prove that a particular language is non-regular: a proof by contradiction may consist of exhibiting a string (of the required length) in the language that lacks the property outlined in the pumping lemma.

4.2 Pumping Lemma for Regular sets

- It gives a method for pumping (generating) many substrings from a given string.
- In other words, we say it provides means to break a given long input string into several substrings.
- Lt gives necessary condition(s) to prove a set of strings is not regular.

For any regular language L, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists u, v, $w \in \Sigma*$, such that x = uvw, and

(1) $|uv| \leq n$

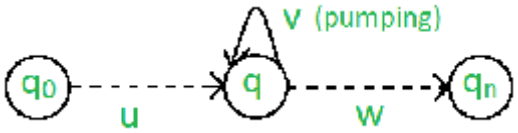
(2) $|v| \geq 1$

(3) for all $i \geq 0$: $uv^iw \in L$

In simple terms, this means that if a string v is 'pumped', i.e., if v is inserted any number of times, the resultant string still remains in L.

Pumping Lemma is used as a proof for irregularity of a language. Thus, if a language is regular, it always satisfies pumping lemma. If there exists at least one string made from pumping which is not in L, then L is surely not regular.

The opposite of this may not always be true. That is, if Pumping Lemma holds, it does not mean that the language is regular.



- For example, let us prove $L_{01} = \{0^n1^n \mid n \geq 0\}$ is irregular.
  Let us assume that L is regular, then by Pumping Lemma the above given rules follow.
  Now, let $x \in L$ and $|x| \geq n$. So, by Pumping Lemma, there exists u, v, w such that (1) – (3) hold.

  We show that for all u, v, w, (1) – (3) does not hold.
  If (1) and (2) hold then $x = 0^n1^n = uvw$ with $|uv| \leq n$ and $|v| \geq 1$.
  So, $u = 0^a$, $v = 0^b$, $w = 0^c1^n$ where : $a + b \leq n$, $b \geq 1$, $c \geq 0$, $a + b + c = n$
  But, then (3) fails for i = 0
  $uv^0w = uw = 0^a0^c1^n = 0^{a+c}1^n \notin L$, since $a + c \neq n$.

4.3 Applications of Pumping Lemma

Pumping lemma is to be applied to show that certain languages are not regular.

It should never be used to show a language is regular.

- If L is regular, it satisfies the Pumping lemma.
- If L does not satisfy the Pumping Lemma, it is not regular.

Not all languages are regular. For example, the language $L = \{a^n b^n : n \geq 0\}$ is not regular. Similarly, the language $\{a^p : p \text{ is a prime number}\}$ is not regular. A pertinent question therefore is how do we know if a language is not regular.

Question: Can we conclude that a language is not regular if no one could come up with a DFA, NFA, ε-NFA, regular expression or regular grammar so far?

- No. Since, someone may very well come up with any of these in future.

We need a property that just holds for regular languages and so we can prove that any language without that property is not regular. Let's recall some of the properties.

- We have seen that a regular language can be expressed by a finite state automaton. Be it deterministic or non-deterministic, the automaton consists of a finite set of states.
- Since the states are finite, if the automaton has no loop, the language would be finite.

  - Any finite language is indeed a regular language since we can express the language using the regular expression: S1 + S2 + ... + SN, where N is the total number of strings accepted by the automaton.
- However, if the automaton has a loop, it is capable of accepting infinite number of strings.

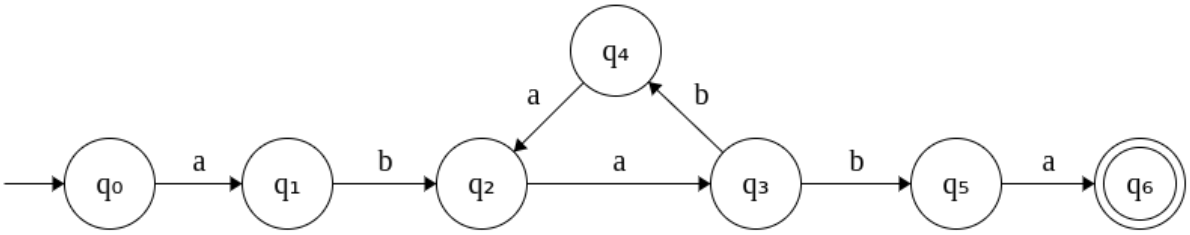
  - Because, we can loop around any number of times and keep producing more and morestrings.
  - This property is called the pumping property (elaborated below).

The pumping property of regular languages

Any finite automaton with a loop can be divided into parts three.

- Part 1: The transitions it takes before the loop.
- Part 2: The transitions it takes during the loop.
- Part 3: The transitions it takes after the loop.

For example consider the following DFA. It accepts all strings that start with aba followed by any number of baa's and finally ending with ba.

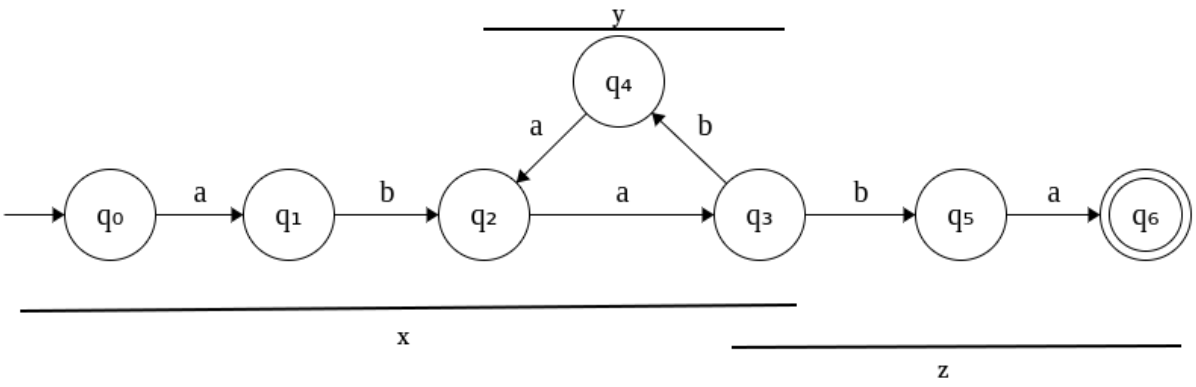


1. What strings are accepted by this DFA?

aba**baa**ba, aba**baabaa**ba, aba**baabaabaa**ba, ... so on and so forth. Thus the strings accepted by the above DFA can be divided into three parts: **aba, (baa)^i and ba**. Here, i > 0.

Investigating this further, we can say that any string w accepted by this DFA can be written as **w = x y^i z**

where y represents the part that can be pumped again and again to generate more and more valid strings. This is shown below for the given example.



Before we generalize further, let's investigate this example a little more.

2. What if the loop was at the beginning? Say a self-loop at q_0 instead of at q_2. Then x = ε or |x| = 0. In such a special case, w = yz.
3. What is the loop was at the end. Say a self loop at q_6 instead of at q_2. Then z = ε or |z| = 0. In such a special case, w = xy.
4. Can y be equal to ε ever?

No. It is impossible. If y = ε, it implies there is no loop which implies the language is finite. We have already seen that a finite language is always regular. So, we are now concerned only with infinite regular language. Hence, y can never be ε. Or |y| > 0.

5. What is the shortest string that is accepted by the DFA? ababa. Obviously, a string obtained without going through the loop. There is a catch however. See the next question.
6. What is the shortest string accepted if there are more final states? Say q_2 is final. ab of length 2.
7. What is the longest string accepted by the DFA without going through the loop even once? ababa (= xz). So, any string of length > 5 accepted by DFA must go through the loop at least once.
8. What is the longest string accepted by the DFA by going through the loop exactly once? ababaaba (= xyz) of length 8. We call this pumping length.

More precisely, pumping length is an integer p denoting the length of the string w such that w is obtained by going through the loop exactly once. In other words, |w| = |xyz| = p.

9. Of what use is this pumping length p? We can be sure that $|xy| \leq p$. This can be used to prove a language non-regular.

Now, let's define a regular language based on the pumping property.

Pumping Lemma: If L is a regular language, then there exists a constant p such that every string w $\in$ L, of length p or more can be written as w = xyz, where

1. $|y| > 0$
2. $|xy| \leq p$
3. $xy^iz \in L$ for all i

Proving languages non-regular

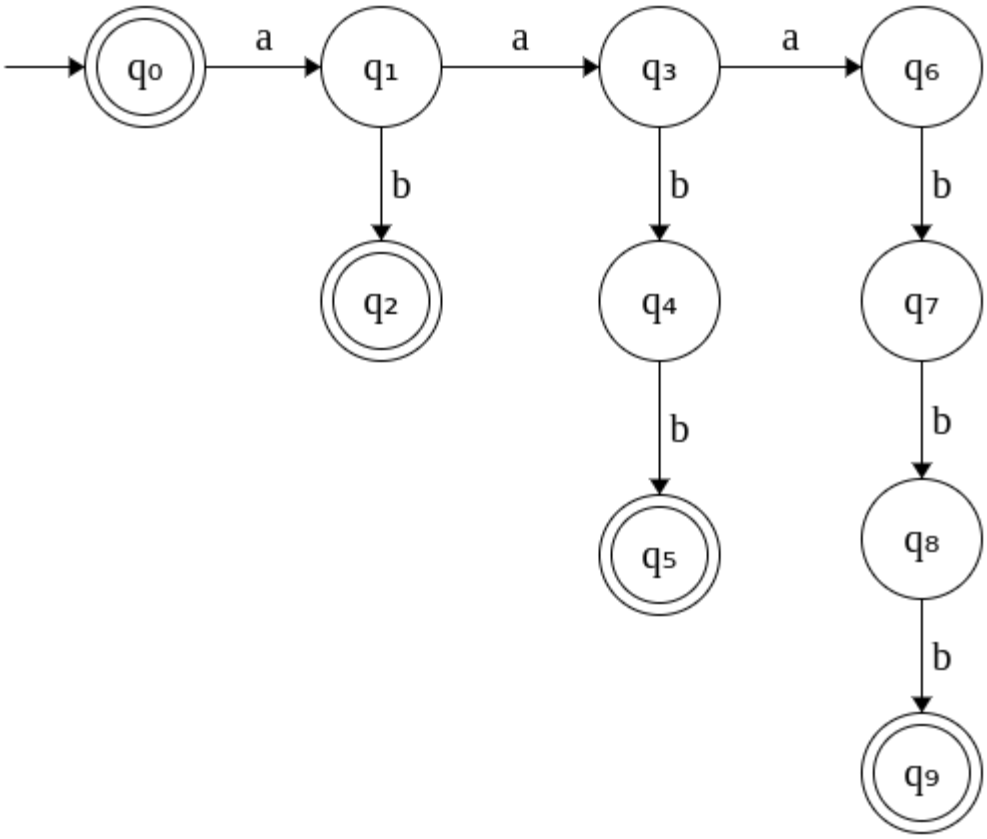
1. The language $L = \{ a^nb^n : n \neq 0 \}$ is not regular.

Before proving L is not regular using pumping property, let's see why we can't come up with a DFA or regular expression for L.

L = { ε, ab, aabb, aaabbb, ... }

It may be tempting to use the regular expression a*b* to describe L. No doubt, a*b* generates these strings. However, it is not appropriate since it generates other strings not in L such as a, b, aa, ab, aaa, aab, abb, ...

Let's try to come up with a DFA. Since it has to accept ε, start state has to be final. The following DFA can accept a^nb^n for n ≤ 3. i.e. {ε, a, b, ab, aabb, aaabbb}



The basic problem is DFA does not have any memory. A transition just depends on the current state. So it cannot keep count of how many a's it has seen. So, it has no way to match the number of a's and b's. So, only way to accept all the strings of L is to keep adding newer and newer states which makes automaton to infinite states since n is unbounded.

Now, let's prove that L does not have the pumping property.

Lets assume L is regular. Let p be the pumping length.

Consider a string w = aa....abb....b such that |w| = p.

$\Rightarrow w = a^{p/2}b^{p/2}$

We know that w can be broken into three terms xyz such that $y \neq \varepsilon$ and $xy^iz \in L$.

There are three cases to consider.

- Case 1: y is made up of only a's

  Then xy^2z has more a's than b's and does not belong to L.

- Case 2: y is made up of only b's

  Then xy^2z has more b's than a's and does not belong to L.

- Case 3: y is made up of a's and b's

  Then xy^2z has a's and b's out of order and does not belong to L.

Since none of the 3 cases hold, the pumping property does not hold for L. And therefore L is not regular.

2. The language $L = \{ uu^R : u \in \{a,b\}^* \}$ is not regular.

Lets assume L is regular. Let p be the pumping length.

Consider a string $w = a^pbba^p$.

$|w| = 2p + 2 \geq p$

Since, $xy \leq p$, xy will consist of only a's.

$\Rightarrow$ y is made of only a's

$\Rightarrow$ y^2 is made of more number of a's than y since $|y| > 0$ (Let's say y^2 has m a's more than y where $m > 1$)

$\Rightarrow xy^2z = a^{p+m}bba^p$ where $m \geq 1$

$\Rightarrow xy^2z = a^{p+m}bba^p$ cannot belong to L.

Therefore, pumping property does not hold for L. Hence, L is not regular.

3. The language $L = \{ a^n : n \text{ is prime} \}$ is not regular.

Lets assume L is regular. Let p be the pumping length. Let $q \geq p$ be a prime number (since we cannot assume that pumping length p will be prime).

Consider the string w = aa....a such that $|w| = q \geq p$.

We know that w can be broken into three terms xyz such that $y \neq \varepsilon$ and $xy^iz \in L$

$\Rightarrow$ $xy^{q+1}z$ must belong to L

$\Rightarrow |xy^{q+1}z|$ must be prime

$|xy^{q+1}z| = |xyzy^q|$

$= |xyz| + |y^q|$

$= q + q.|y|$

$= q(1 + |y|)$ which is a composite number.

Therefore, $xy^{q+1}z$ cannot belong to L. Hence, L is not regular.

IMPORTANT NOTE

Never use pumping lemma to prove a language regular. Pumping property is necessary but not sufficient for regularity.

4.4 Closure Properties of Regular Language

Union : If L1 and If L2 are two regular languages, their union L1 ∪ 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.

Intersection : If L1 and If L2 are two regular languages, their intersection L1 ∩ L2 will also be regular. For example,

$L1 = \{a^m b^n \mid n \geq 0 \text{ and } m \geq 0\}$ and $L2 = \{a^m b^n \cup b^n a^m \mid n \geq 0 \text{ and } m \geq 0\}$

$L3 = L1 \cap L2 = \{a^m b^n \mid n \geq 0 \text{ and } m \geq 0\}$ is also regular.

Concatenation : If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular. For example,

$L1 = \{a^n \mid n \geq 0\}$ and $L2 = \{b^n \mid n \geq 0\}$

$L3 = L1.L2 = \{a^m . b^n \mid m \geq 0 \text{ and } n \geq 0\}$ is also regular.

Kleene Closure : If L1 is a regular language, its Kleene closure L1* will also be regular. For example,

$L1 = (a \cup b)$

$L1^* = (a \cup b)^*$

Complement : If L(G) is regular language, its complement L'(G) will also be regular.

Complement of a language can be found by subtracting strings which are in L(G) from all possible strings. For example,

$L(G) = \{a^n \mid n > 3\}$

$L'(G) = \{a^n \mid n <= 3\}$

Note : Two regular expressions are equivalent if languages generated by them are same. For example, (a+b*)* and (a+b)* generate same language. Every string which is generated by (a+b*)* is also generated by (a+b)* and vice versa.

4.5 Myhill Nerode Theorm

is a fundamental result coming down to the theory of languages. This theory was proven by John Myhill and Anil Nerode in 1958. It is used to prove whether or not a language L is regular and it is also used for minimization of states in DFA(Deterministic Finite Automata).

To understand this theorem, first we need to understand what Indistinguishability is :

Given a language L and x,y are string over $\sum^*$, if for every string $z \in \sum^*$, $xz, yz \in L$ or $xz, yz \notin L$ then x and y are said to be indistinguishable over language L. Formally, we denote that x and y are indistinguishable over L by the following notation : $x \equiv_L y$.

$\equiv_L$ is an equivalence relation as it is :

1) Reflexive : For all string x, $xz \in L$ iff $xz \in L$ therefore $x \equiv_L x$.

2) Symmetric : Suppose $x \equiv_L y$. This means either $xz, yz \in L$ or $xz, yz \notin L$ for all $z \in \sum^*$. Equivalently this means $yz,xz \in L$ or $yz, xz \notin L$ for all $z \in \sum^*$ which implies $y \equiv_L x$

3) Transitive : Suppose $x \equiv_L y$ and $y \equiv_L w$. Then suppose for the sake of contradiction that x and w are not indistinguishable. This means there must exist some z such that exactly one of xz and wz is a member of L. Assume xz is a member of L and wz is not a member of L. $xz \in L$ implies $yz \in L$. $wz \notin L$ implies that $yz \notin L$. This is a contradiction since yz cannot both a member and not be a member of L. Therefore $x \equiv_L y$ and $y \equiv_L w \Rightarrow x \equiv_L w$.

Since $\equiv_L$ is an equivalence relation over $\sum^*$, $\equiv_L$ partitions $\sum^*$ into disjoint sets called equivalence classes.

Myhill Nerode Theorem :

A language is regular if and only if $\equiv_L$ partitions $\sum^*$ into finitely many equivalence classes. If $\equiv_L$ partitions $\sum^*$ into n equivalence classes, then a minimal DFA recognizing L has exactly n states.

Example :

To prove that $L = \{a^n b^n \mid n \geq 0\}$ is not regular.

We can show that L has infinitely many equivalence classes by showing that a^k and a^i are distinguishable by L whenever $k \neq i$. Thus, for $x = a^k$ and $y = a^i$ we let $z = b^k$. Then $xz = a^k b^k$ is in the language but $yz = a^i b^k$ is not. Thus, each equivalence class of L can contain at most one string of the form a^i so there must be infinitely many equivalence classes. That means L is not regular by the Myhill Nerode theorem.

Note : To prove whether or not a language L is regular is also done using Pumping Lemma, the distinction between this and Myhill Nerode theorem is that, there are some non-regular language satisfying the Pumping Lemma but no such non regular language is there which satisfies Myhill Nerode theorem.

Minimization of DFA using Myhill-Nerode Theorem :

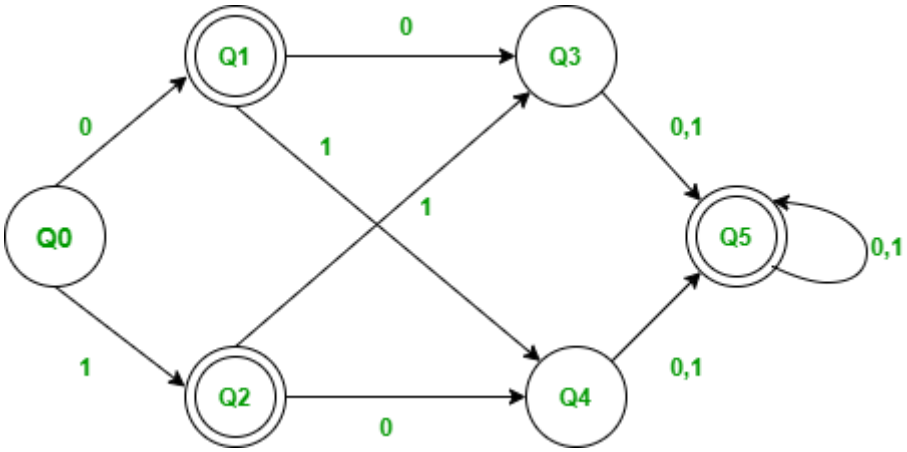
Minimization of DFA is Required to obtain the minimal and equivalent version of any DFA which consists of minimum number of states possible. Myhill-Nerode theorem can be used to convert a DFA to its equivalent DFA with minimum no of states. This method of minimization is also called Table filling method. There is also another method called Partitioning Method or Equivalence Method for the minimization of DFA.

Steps for the Minimization of DFA :

1. Create the pairs of all the states involved in the given DFA.
2. Mark all the pairs (Q_a, Q_b) such a that Q_a is Final state and Qb is Non-Final State.
3. If there is any unmarked pair (Qa,Qb) such a that δ(Qa,x) and $\delta(Q_b,x)$ is marked, then mark (Qa,Qb). Here x is a input symbol. Repeat this step until no more marking can be made.
4. Combine all the unmarked pairs and make them a single state in the minimized DFA.

Example

Consider the following DFA,

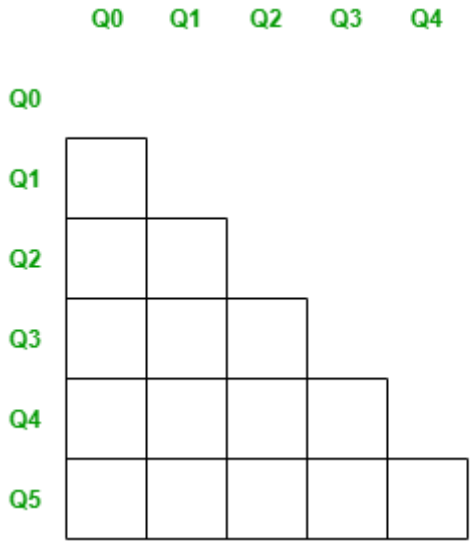


Following is the transition table for the above DFA

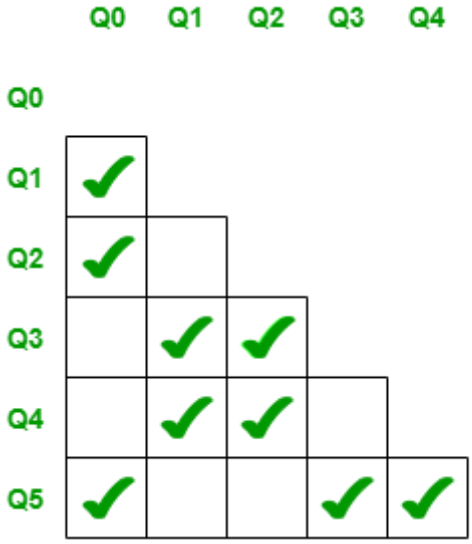
States	Inputs	
	0	1
Q0	Q1	Q2
Q1	Q3	Q4
Q2	Q4	Q3
Q3	Q5	Q5
Q4	Q5	Q5
Q5	Q5	Q5

Minimizing the above DFA using Myhill-Nerode Theorem :

Step-1: Create the pairs of all the states involved in DFA.



Step-2: Mark all the pairs (Qa,Qb) such a that Qa is Final state and Qb is Non-Final State.

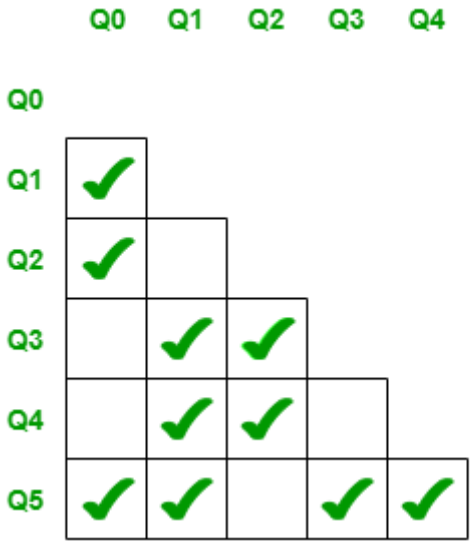


Step-3: If there is any unmarked pair (Qa,Qb) such a that δ(Qa,x) and δ(Qb,x) is marked, then mark (Qa,Qb). Here x is a input symbol. Repeat this step until no more marking can be made.

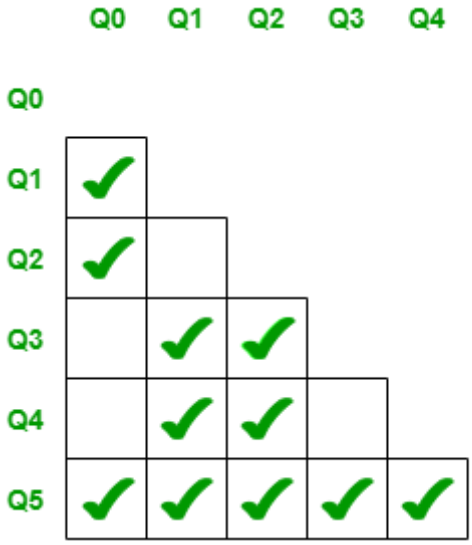
- Check for the unmarked pair Q2,Q1
    - Check when x=0 : δ(Q2,0) = Q4 and δ(Q1,0) = Q3, check if the pair Q4,Q3 is marked and no it is not marked.
    - Check when x=1 : δ(Q2,1) = Q3 and δ(Q1,1) = Q4, check if the pair Q4,Q3 is marked and no it is not marked.

- Hence we cannot mark the pair Q2,Q1.

- Check for the unmarked pair Q3,Q0
    - Check when x=0 : δ(Q3,0) = Q5 and δ(Q0,0) = Q1, check if the pair Q5,Q1 is marked and no it is not marked.
    - Check when x=1 : δ(Q3,1) = Q5 and δ(Q0,1) = Q2, check if the pair Q5,Q2 is marked and no it is not marked.
    - Hence we cannot mark the pair Q3,Q0.
- Check for the unmarked pair Q4,Q0
    - Check when x=0 : δ(Q4,0) = Q5 and δ(Q0,0) = Q1, check if the pair Q5,Q1 is marked and no it is not marked.
    - Check when x=1 : δ(Q4,1) = Q5 and δ(Q0,1) = Q2, check if the pair Q5,Q2 is marked and no it is not marked.
    - Hence we cannot mark the pair Q4,Q0.
- Check for the unmarked pair Q4,Q3
    - Check when x=0 : δ(Q4,0) = Q5 and δ(Q3,0) = Q5, Such pair of state Q5,Q5 don’t exists.
    - Check when x=1 : δ(Q4,1) = Q5 and δ(Q3,1) = Q5, Such pair of state Q5,Q5 don’t exists.
    - Hence we cannot mark the pair Q4,Q3.
- Check for the unmarked pair Q5,Q1
    - Check when x=0 : δ(Q5,0) = Q5 and δ(Q1,0) = Q3, check if the pair Q5,Q3 is marked and yes it is marked.
    - Hence we can mark the pair Q5,Q1.



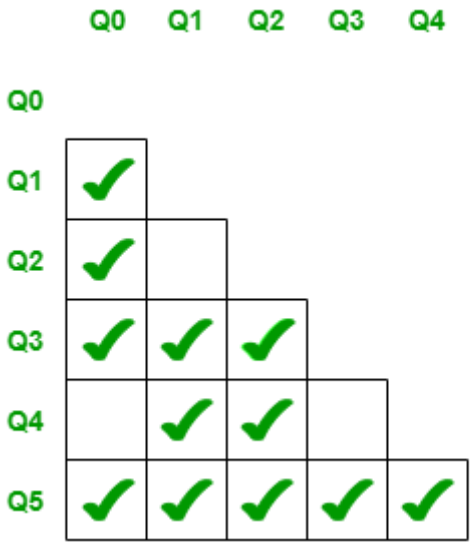
- Check for the unmarked pair Q5,Q2
    - Check when x=0 : δ(Q5,0) = Q5 and δ(Q2,0) = Q4, check if the pair Q5,Q4 is marked and yes it is marked.
    - Hence we can mark the pair Q5,Q2.



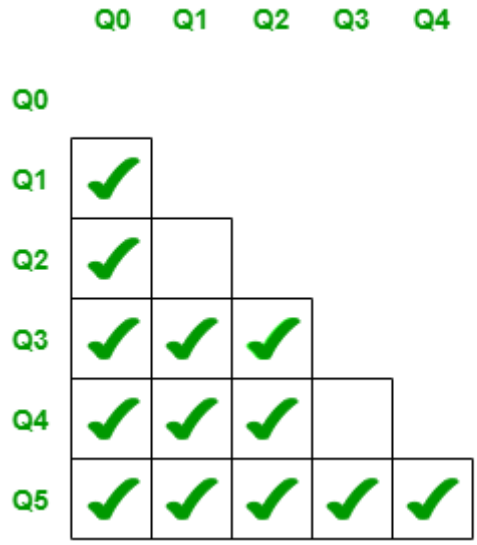
- We have checked for all the unmarked pairs but don't need to stop here we need to continue this process until no more markings can be made.
- Check for the unmarked pair Q2,Q1

- Check when x=0 : δ(Q2,0) = Q4 and δ(Q1,0) = Q3, check if the pair Q4,Q3 is marked and no it is not marked.
- Check when x=1 : δ(Q2,1) = Q3 and δ(Q1,1) = Q4, check if the pair Q4,Q3 is marked and no it is not marked.
- Hence we cannot mark the pair Q2,Q1.

- Check for the unmarked pair Q3,Q0
  - Check when x=0 : δ(Q3,0) = Q5 and δ(Q0,0) = Q1, check if the pair Q5,Q1 is marked and yes it is marked.
  - Hence we can mark the pair Q3,Q0.



- Check for the unmarked pair Q4,Q0
  - Check when x=0 : δ(Q4,0) = Q5 and δ(Q0,0) = Q1, check if the pair Q5,Q1 is marked and yes it is marked.
  - Hence we cannot mark the pair Q4,Q0.



- Check for the unmarked pair Q4,Q3
    - Check when x=0 : δ(Q4,0) = Q5 and δ(Q3,0) = Q5, Such pair of state Q5,Q5 don't exists.
    - Check when x=1 : δ(Q4,1) = Q5 and δ(Q3,1) = Q5, Such pair of state Q5,Q5 don't exists.
    - Hence we cannot mark the pair Q4,Q3.
- Now even though we repeat the procedure we cannot mark the pairs Q2,Q1(since Q4,Q3 is not marked) and Q4,Q3(since Q5,Q5 such pair of states does not exists.). Hence we stop here.

Step-4: Combine all the unmarked pairs and make them as a single state in the minimized DFA.

- The unmarked Pairs are Q2,Q1 and Q4,Q3 hence we combine them.

Following is the Minimized DFA with Q1Q2 and Q3Q4 as the combined states.

- Q0 remains as our starting state.

- Q1 and Q2 were our final states so even we combine them they will remain as the combined final state.
- Q5 is the another final state we have.
- If we check the original Transition Table
    - δ(Q0,0) was Q1 and δ(Q1,1) was Q2. As the states are combined, the transition of Q0 on both the inputs 0 and 1 will be to the state Q1Q2.
    - δ(Q1,0) was Q3, δ(Q1,1) was Q4 and δ(Q2,0) was Q4, δ(Q1,1) was Q3. As the states are combined, the transition of Q1Q2 on both the inputs 0 and 1 will be to the state Q3Q4.
    - δ(Q3,0) was Q5, δ(Q3,1) was Q5 and δ(Q4,0) was Q5, δ(Q4,1) was Q5. As the states are combined, the transition of Q3Q4 on both the inputs 0 and 1 will be to the state Q5.
    - δ(Q5,0) was Q5 and δ(Q5,1) was Q5. Hence the transition of state Q5 on both the inputs will be to the state Q5 itself.

Transition table for Minimized DFA

States	Inputs	
	0	1
Q0	Q1Q2	Q1Q2
Q1Q2	Q3Q4	Q3Q4
Q3Q4	Q5	Q5
Q5	Q5	Q5

4.6 CHECK YOUR PROGRESS

1 . The regular sets are closed under

a) Union

b) Concenteration

C) Kleen closure

d) All of the above

2. A class of language that is closed under

a)union and complementation has to be closed under intersection

b) intersection and complement has to be closed under union

C)union and intersection has to be closed under complementation

d) both (A) and (B)

3. Pumping lemma is generally used for proving that

a) given grammar is regular

b) given grammar is not regular

c) whether two given regular expressions are equivalent or not

d) None of these

4. The logic of pumping lemma is a good example of

a) pigeon-hole principle b) divide-and-conquer technique

c) Recursion d) iteration

5. Which of the following are not regular

a) String of 0's whose length is a perfect square b) Set of all palindromes made up of 0's and 1's

c) Strings of 0's, whose length is a prime number d) All of these

6.Which of the following statement is wrong

a) Recursive Languages can be recognized by finite automaton
b) The class of regular sets is closed under intersection
c) The class of regular sets is closed under substitution
d) Regular languages are recognize by finite automaton

7. Decision Properties are used to:

a) To test the emptiness of regular languages
b) To test tge whether the language is Regular or not
c) Both (a) and (b)
d) None of these

8. Let R1 and R2 be regular sets defined over alphabet ∑ then

a) R1 UNION R2 is regular b) R1 INTERSECTION R2

c) ∑ INTERSECTION R2 IS NOT REGULAR d) R2* IS NOT REGULAR

4.7 SUMMARY

We have explained Myhill Nerode Theorm and How we can Minimize the Automaton form theorm. we have discussed Pumping Lemma for Regular Languages and Closure Properties of Regular languages.Pumping Lemma is used to Prove that Language is not Regular.

4.8 KEYWORDS

1. Myhill Nerode Theorm: The Myhill Nerode theorem is a **fundamental result coming down to the theory of languages**. This theory was proven by John Myhill and Anil Nerode in 1958. It is used to prove whether or not a language L is regular and it is also used for minimization of states in DFA

2. Closure Properties: Closure Property In mathematics, a set is closed under an operation when we perform that operation on members of the set, and we always get a set member.

3. Decision Problems: In computability theory and computational complexity theory, a decision problem is **a problem that can be posed as a yes–no question of the input values.** An example of a decision problem is deciding whether a given natural number is prime.

4.9 SELF ASSESSMENT TEST

Q 1. State and prove the pumping lemma for Regular Language?

Q 2. List the closure properties of Regular Language?

Q 3. Describe the Myhill -Nerode Theorm?

Q 4. How we can Minimize the finite Automaton with the help of Myhill-Nerode Theorm?

Q 5. What is its main application? Give two examples?

4.10 ANSWER TO CHECK YOUR PROGRESS

1. D

2. B

3. B

4 A

5. D

6. A

7. A

8. A

4.11 REFERENCES/SUGGESTED READINGS

1.Hopcroaft & O. D. Ullman, R Mothwani, Introduction to automata theory, language & computations, AW, 2001.

2. K. L. P. Mishra & N. Chandrasekaran, . Theory of Computer Sc.(Automata, Languages and computation), PHI, 2000.

3. Peter Linz, Introduction to formal Languages & Automata, Narosa, Publication, 2001.

4. Ramond Greenlaw and H. James Hoover, Fundamentals of the Theory of ComputationPrinciples and Practice, Harcourt India Pvt. Ltd., 1998.

5. H. R. Lewis & C. H. Papaditriou, Elements of theory of Computation, PHC, 1998.

6. John C. Martin, Introduction to Languages and the Theory of Computation, T.M.H., 2003

SUBJECT: Theory of computation	
COURSE CODE: MCA-35	AUTHOR: RAVIKA GOEL
LESSON NO. 5	
Introduction of Context free Grammer	

STRUCTURE

5.0 OBJECTIVE

The main objective of this lesson is to study the Basic Concept of Grammer .Understand the Context free Grammer and Context Sensitive Grammer. How to Remove null and Unit Productions from Grammer. Study the Chomsky normal form and Greibach normal form.

5.1 INTRODUCTION

Grammer is a set of rules which checks whether a string belongs to a particular language a not.

A program consists of various strings of characters. But, every string is not a proper or meaningful string. So, to identify valid strings in a language, some rules should be specified to check whether the string is valid or not. These rules are nothing but make **Grammar**.

Example − In English Language, Grammar checks whether the string of characters is acceptable or not, i.e., checks whether nouns, verbs, adverbs, etc. are in the proper sequence.It is a notation used to specify the syntax of the language. Context-free Grammar is used to design parsers. As Lexical Analyzer generates a string of tokens which are given to parser to construct parse tree. But, before constructing the parse tree, these tokens will be grouped so that the results of grouping will be a valid construct of a language. So, to specify constructs of language, a suitable notation is used, which will be precise & easy to understand. This notation is Context-Free Grammar.

We define derivation trees and give methods of simplifying context-free grammars. The two normal forms-Chomsky normal form and Greibach normal form-are dealt with. Context-free languages are applied in parser design. They are also useful for describing block structures in programming languages. It is easy to visualize derivations in context-free languages as we can represent derivations using tree structures.

5.2 Grammer and Language Generated from Grammer

Grammar :

It is a finite set of formal rules for generating syntactically correct sentences or meaningful correct sentences.

Constitute Of Grammar :

Grammar is basically composed of two basic elements –

1. **Terminal Symbols –**
   Terminal symbols are those which are the components of the sentences generated using a grammar and are represented using small case letter like a, b, c etc.
2. **Non-Terminal Symbols –**
   Non-Terminal Symbols are those symbols which take part in the generation of the sentence but are not the component of the sentence. Non-Terminal Symbols are also called Auxiliary Symbols and Variables. These symbols are represented using a capital letter like A, B, C, etc.

Formal Definition of Grammar :

Any Grammar can be represented by 4 tuples – <N, T, P, S>

- **N –** Finite Non-Empty Set of Non-Terminal Symbols.
- **T –** Finite Set of Terminal Symbols.
- **P –** Finite Non-Empty Set of Production Rules.
- **S –** Start Symbol (Symbol from where we start producing our sentences or strings).

Production Rules :

A production or production rule in computer science is a rewrite rule specifying a symbol substitution that can be recursively performed to generate new symbol sequences. It is of the form α-> β where α is a Non-Terminal Symbol which can be replaced by β which is a string of Terminal Symbols or Non-Terminal Symbols.

Example-1 :

Consider Grammar G1 = <N, T, P, S>

T = {a,b} #Set of terminal symbols

P = {A->Aa,A->Ab,A->a,A->b,A-> } #Set of all production rules

S = {A} #Start Symbol

As the start symbol is S then we can produce Aa, Ab, a,b, which can further produce strings where A can be replaced by the Strings mentioned in the production rules and hence this grammar can be used to produce strings of the form (a+b)*.

Derivation Of Strings :

A->a #using production rule 3

OR

A->Aa #using production rule 1

Aa->ba #using production rule 4

OR

A->Aa #using production rule 1

Aa->AAa #using production rule 1

AAa->bAa #using production rule 4

bAa->ba #using production rule 5

Example-2 :

Consider Grammar G2 = <N, T, P, S>

N = {A} #Set of non-terminals Symbols

T = {a} #Set of terminal symbols

P = {A->Aa, A->AAa, A->a, A-> } #Set of all production rules

S = {A} #Start Symbol

As the start symbol is S then we can produce Aa, AAa, a, which can further produce strings where A can be replaced by the Strings mentioned in the production rules and hence this grammar can be used to produce strings of form (a)*.

Derivation Of Strings :

A->a #using production rule 3

OR

A->Aa #using production rule 1

Aa->aa #using production rule 3

OR

A->Aa #using production rule 1

Aa->AAa #using production rule 1

AAa->Aa #using production rule 4

Aa->aa #using production rule 3

Equivalent Grammars :

Grammars are said to be equivalent id they produce the same language.

Language generated by a grammar –

Given a grammar G, its corresponding language L(G) represents the set of all strings generated from G. Consider the following grammar,

G: S-> aSb|ε

In this grammar, using S-> ε, we can generate ε. Therefore, ε is part of L(G). Similarly, using S=>aSb=>ab, ab is generated. Similarly, aabb can also be generated.

Therefore,

$L(G) = \{a^n b^n, n>=0\}$

In language L(G) discussed above, the condition n = 0 is taken to accept ε.

Key Points –

- For a given grammar G, its corresponding language L(G) is unique.
- The language L(G) corresponding to grammar G must contain all strings which can be generated from G.
- The language L(G) corresponding to grammar G must not contain any string which can not be generated from G.

5.3 Context Free Grammer

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

- **N** is a set of non-terminal symbols.
- **T** is a set of terminals where **N ∩ T = NULL.**
- **P** is a set of rules, **P: N → (N ∪ T)***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
- **S** is the start symbol.

Example

- The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc.
- The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε
- The grammar ({S, F}, {0, 1}, P, S), P: S → 00S | 11F, F → 00F | ε

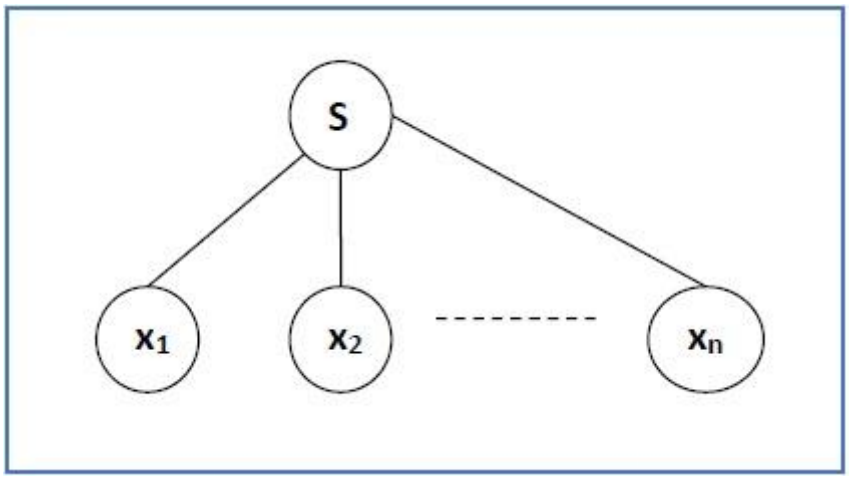
Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

Representation Technique

- **Root vertex** − Must be labeled by the start symbol.
- **Vertex** − Labeled by a non-terminal symbol.
- **Leaves** − Labeled by a terminal symbol or ε.

If $S \rightarrow x_1x_2 \ldots\ldots x_n$ is a production rule in a CFG, then the parse tree / derivation tree will be as follows



There are two different approaches to draw a derivation tree −

Top-down Approach −

- Starts with the starting symbol **S**
- Goes down to tree leaves using productions

Bottom-up Approach −

- Starts from tree leaves
- Proceeds upward to the root which is the starting symbol **S**

Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

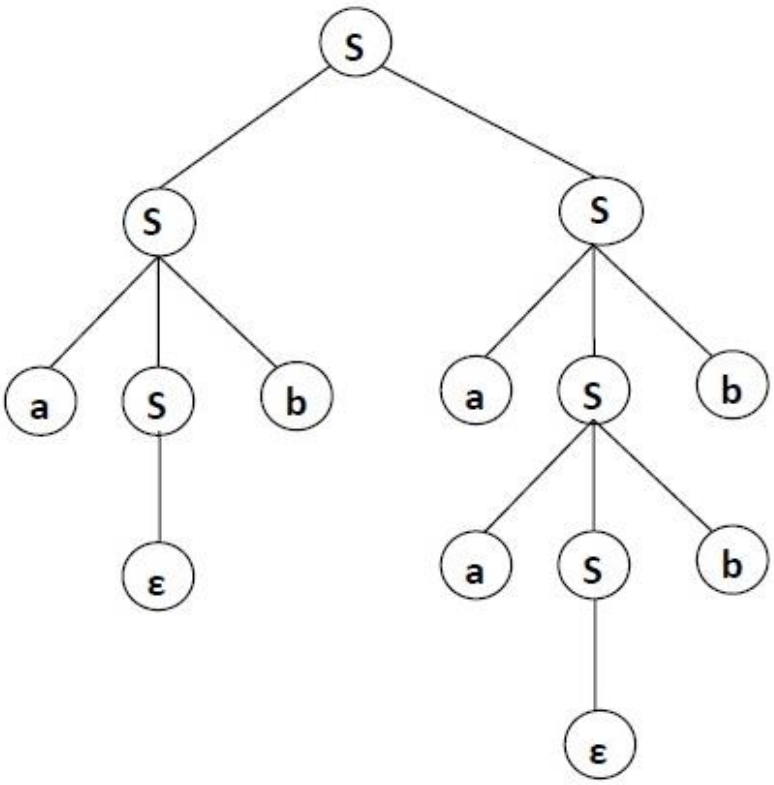
Example

Let a CFG {N,T,P,S} be

N = {S}, T = {a, b}, Starting symbol = S, P = S → SS | aSb | ε

One derivation from the above CFG is "abaabb"

S → SS → aSbS → abS → abaSb → abaaSbb → abaabb



Sentential Form and Partial Derivation Tree

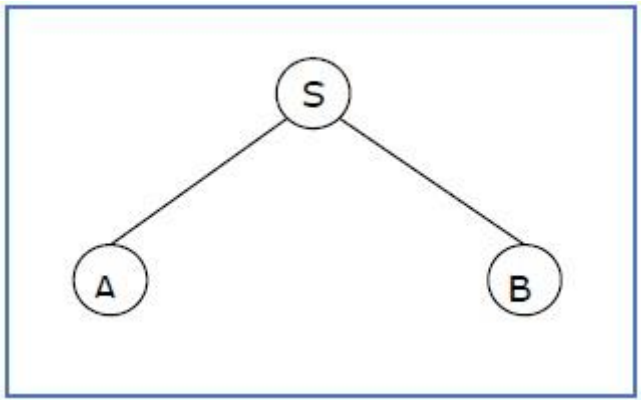
A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

Example

If in any CFG the productions are −

S → AB, A → aaA | ε, B → Bb| ε

the partial derivation tree can be the following −



If a partial derivation tree contains the root S, it is called a **sentential form**. The above sub-tree is also in sentential form.

Leftmost and Rightmost Derivation of a String

- **Leftmost derivation** − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- **Rightmost derivation** − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example

Let any set of production rules in a CFG be

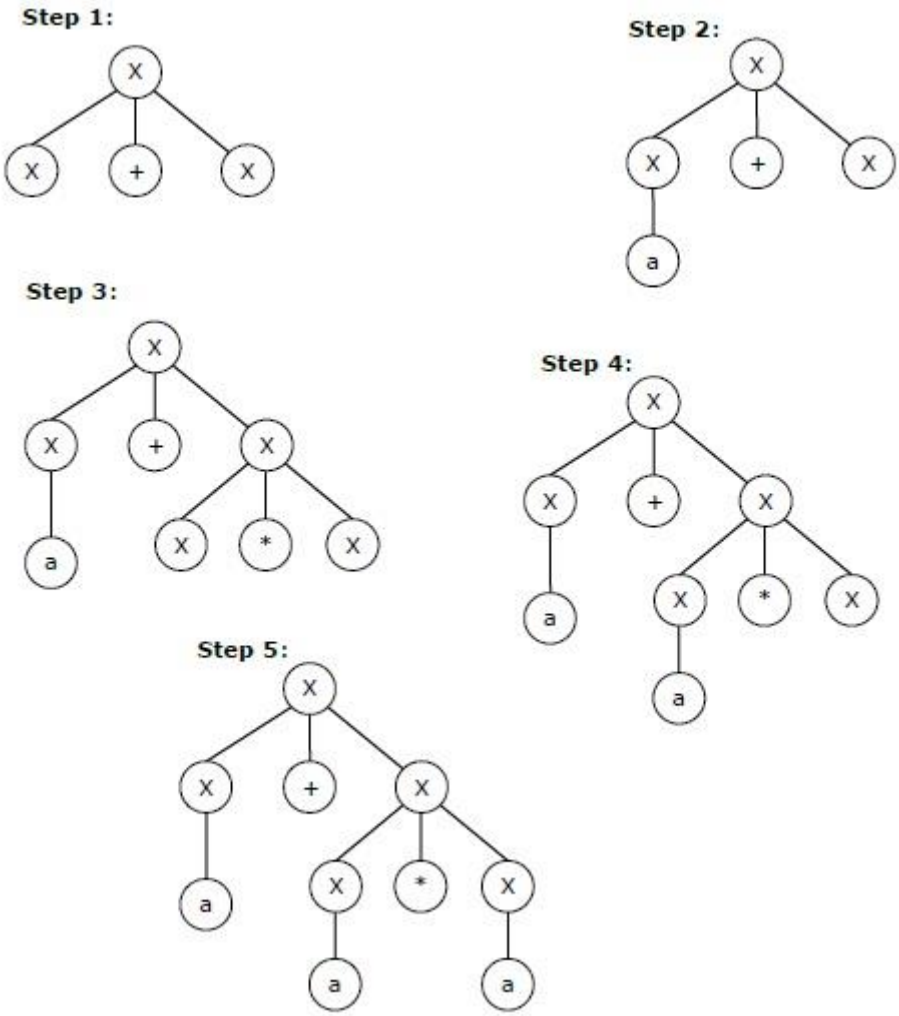
X → X+X | X*X |X| a

over an alphabet {a}.

The leftmost derivation for the string **"a+a*a"** may be −

X → X+X → a+X → a + X*X → a+a*X → a+a*a

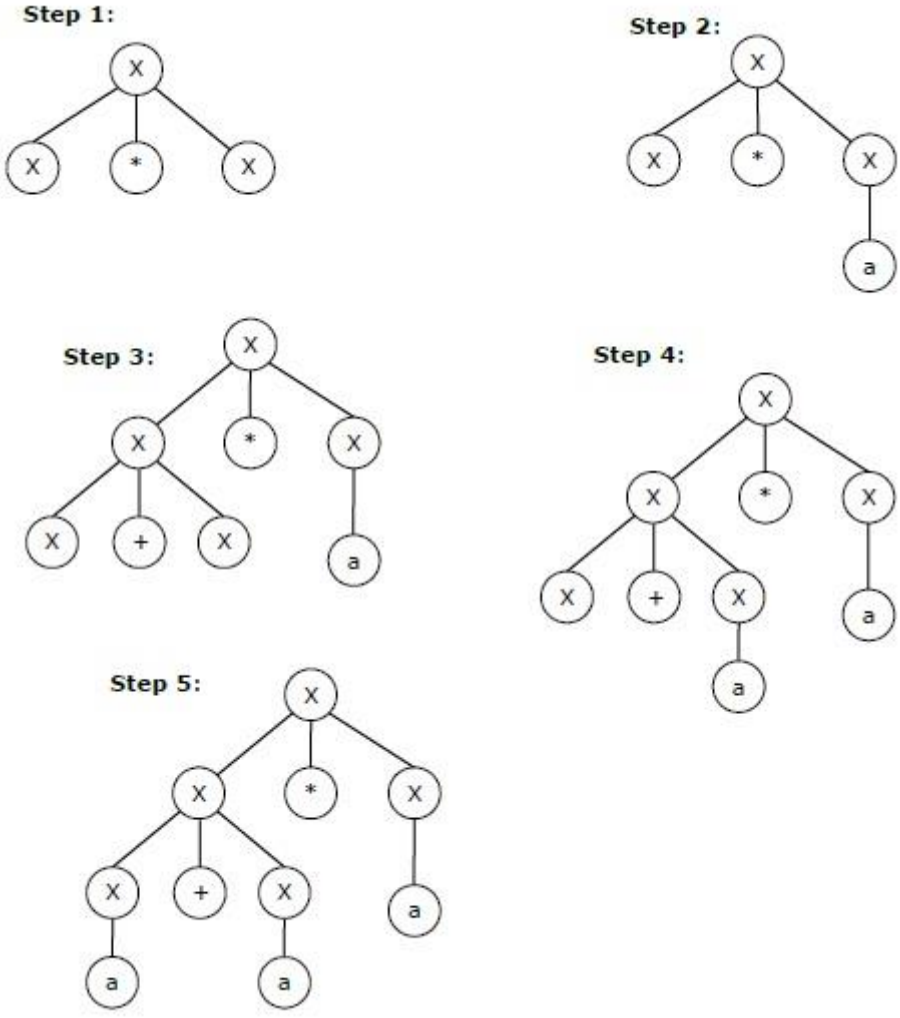
The stepwise derivation of the above string is shown as below −



The rightmost derivation for the above string **"a+a*a"** may be −

X → X*X → X*a → X+X*a → X+a*a → a+a*a

The stepwise derivation of the above string is shown as below –



Left and Right Recursive Grammars

In a context-free grammar **G**, if there is a production in the form **X → Xa** where **X** is a non-terminal and **'a'** is a string of terminals, it is called a **left recursive production**. The grammar having a left recursive production is called a **left recursive grammar**.

And if in a context-free grammar **G**, if there is a production is in the form **X → aX** where **X** is a non-terminal and **'a'** is a string of terminals, it is called a **right recursive production**. The grammar having a right recursive production is called a **right recursive grammar**.

If a context free grammar **G** has more than one derivation tree for some string **w ∈ L(G)**, it is called

5.4 Ambuiguity in Context Free Grammer

an **ambiguous grammar**. There exist multiple right-most or left-most derivations for some string generated from that grammar.

Problem

Check whether the grammar G with production rules −

X → X+X | X*X |X| a

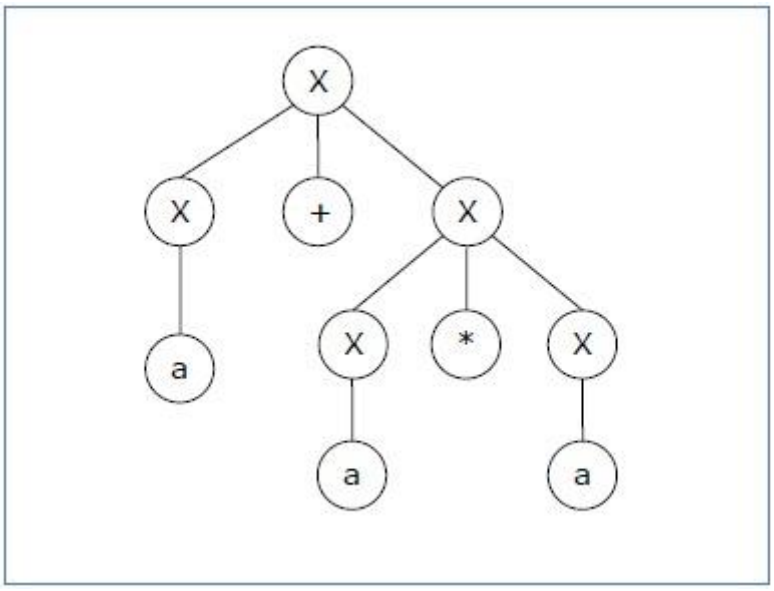
is ambiguous or not.

Solution

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

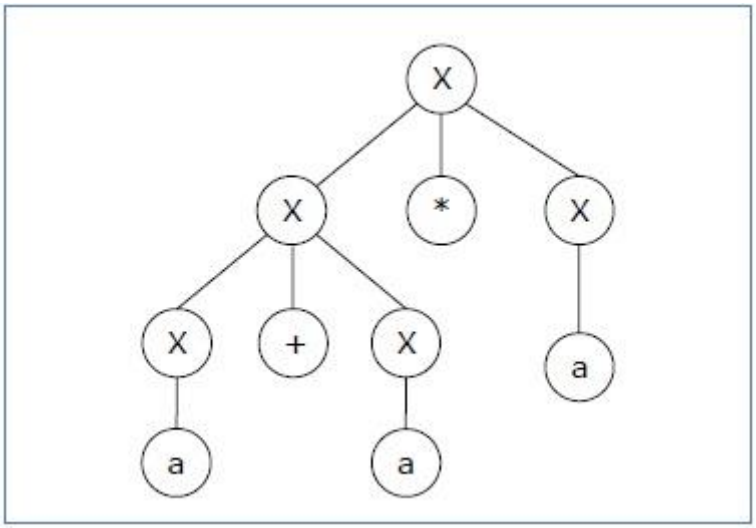
Derivation 1 − X → X+X → a +X → a+ X*X → a+a*X → a+a*a

Parse tree 1 −



Derivation 2 − X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a

Parse tree 2 −



Since there are two parse trees for a single string "a+a*a", the grammar **G** is ambiguous.

5.5 Closure Properties of Context free Grammer

Context-free languages are **closed** under −

- Union
- Concatenation
- Kleene Star operation

Union

Let L_1 and L_2 be two context free languages. Then $L_1 \cup L_2$ is also context free.

Example

Let $L_1 = \{ a^n b^n , n > 0\}$. Corresponding grammar G_1 will have P: S1 → aAb|ab

Let $L_2 = \{ c^m d^m , m \geq 0\}$. Corresponding grammar G_2 will have P: S2 → cBb| ε

Union of L_1 and L_2, $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

The corresponding grammar G will have the additional production S → S1 | S2

Concatenation

If L_1 and L_2 are context free languages, then L_1L_2 is also context free.

Example

Union of the languages L_1 and L_2, $L = L_1L_2 = \{ a^nb^nc^md^m \}$

The corresponding grammar G will have the additional production S → S1 S2

Kleene Star

If L is a context free language, then L* is also context free.

Example

Let $L = \{ a^nb^n , n \geq 0\}$. Corresponding grammar G will have P: S → aAb| ε

Kleene Star $L_1 = \{ a^nb^n \}^*$

The corresponding grammar G_1 will have additional productions $S1 \rightarrow SS_1 \mid \varepsilon$

Context-free languages are **not closed** under −

- **Intersection** − If L1 and L2 are context free languages, then L1 ∩ L2 is not necessarily context free.
- **Intersection with Regular Language** − If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.
- **Complement** − If L1 is a context free language, then L1' may not be context free.

5.6 Context Sensitive Grammer

The Context sensitive grammar (CSG) is defined as G=(V,Σ,P,S)

Where,

- V: Non terminals or variables.
- Σ: Input symbols.
- P: Production rule.
- P:{αAβ → αγβ, A ϵ V,α ϵ (V∪Σ)*, β ϵ (V∪Σ)*
- S: Starting symbol.

Example

- aS→SAa|aA
- aA→abc

In context sensitive grammar, there is either left context or right context (αAβ i.e. α is left context and β is right) with variables.

But in context free grammar (CFG) there will be no context.

For example in production rule

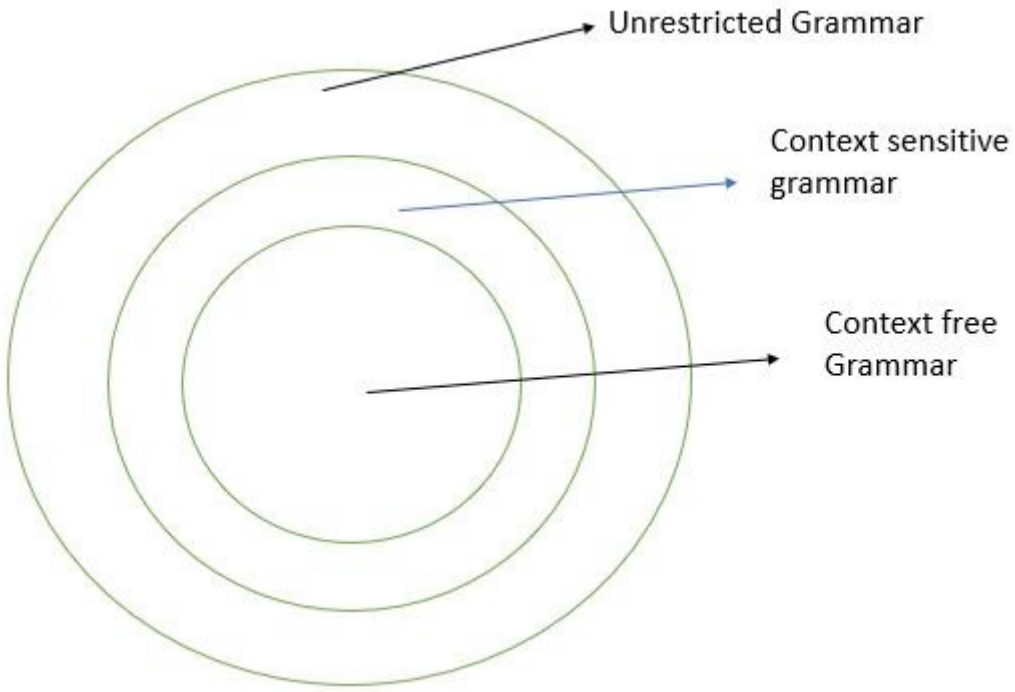
S →0 B S 2 ,

B 0 → 0 B

We cannot replace B until we get B0.

Therefore, CSG is harder to understand than the CFG.

The CFG, CSG and the unrestricted grammar are depicted below −



Context-sensitive Language: The language that can be defined by context-sensitive grammar is called CSL. Properties of CSL are :

- Union, intersection and concatenation of two context-sensitive languages is context-sensitive.
- Complement of a context-sensitive language is context-sensitive.

5.7 Removal of Useless Symbol and Unit Production

In a CFG, it may happen that all the production rules and symbols are not needed for the derivation of strings. Besides, there may be some null productions and unit productions. Elimination of these productions and symbols is called **simplification of CFGs**. Simplification essentially comprises of the following steps −

- Reduction of CFG
- Removal of Unit Productions
- Removal of Null Productions

Reduction of CFG

CFGs are reduced in two phases −

Phase 1 − Derivation of an equivalent grammar, **G'**, from the CFG, **G**, such that each variable derives some terminal string.

Derivation Procedure −

Step 1 − Include all symbols, $\mathbf{W_1}$, that derive some terminal and initialize **i=1**.

Step 2 − Include all symbols, $\mathbf{W_{i+1}}$, that derive $\mathbf{W_i}$.

Step 3 − Increment **i** and repeat Step 2, until $\mathbf{W_{i+1} = W_i}$.

Step 4 − Include all production rules that have $\mathbf{W_i}$ in it.

Phase 2 − Derivation of an equivalent grammar, **G"**, from the CFG, **G'**, such that each symbol appears in a sentential form.

Derivation Procedure −

Step 1 − Include the start symbol in $\mathbf{Y_1}$ and initialize **i = 1**.

Step 2 − Include all symbols, $\mathbf{Y_{i+1}}$, that can be derived from $\mathbf{Y_i}$ and include all production rules that have been applied.

Step 3 − Increment **i** and repeat Step 2, until $\mathbf{Y_{i+1} = Y_i}$.

Problem

Find a reduced grammar equivalent to the grammar G, having production rules, P: S → AC | B, A → a, C → c | BC, E → aA | e

Solution

Phase 1 −

T = { a, c, e }

W_1 = { A, C, E } from rules A → a, C → c and E → aA

W_2 = { A, C, E } U { S } from rule S → AC

W_3 = { A, C, E, S } U ∅

Since $W_2 = W_3$, we can derive G' as −

G' = { { A, C, E, S }, { a, c, e }, P, {S}}

where P: S → AC, A → a, C → c , E → aA | e

Phase 2 −

Y_1 = { S }

Y_2 = { S, A, C } from rule S → AC

Y_3 = { S, A, C, a, c } from rules A → a and C → c

Y_4 = { S, A, C, a, c }

Since $Y_3 = Y_4$, we can derive G" as −

G" = { { A, C, S }, { a, c }, P, {S}}

where P: S → AC, A → a, C → c

Removal of Unit Productions

Any production rule in the form A → B where A, B ∈ Non-terminal is called **unit production.**.

Removal Procedure −

Step 1 − To remove **A → B**, add production **A → x** to the grammar rule whenever **B → x** occurs in the grammar. [x ∈ Terminal, x can be Null]

Step 2 − Delete **A → B** from the grammar.

Step 3 − Repeat from step 1 until all unit productions are removed.

Problem

Remove unit production from the following −

S → XY, X → a, Y → Z | b, Z → M, M → N, N → a

Solution −

There are 3 unit productions in the grammar −

Y → Z, Z → M, and M → N

At first, we will remove M → N.

As N → a, we add M → a, and M → N is removed.

The production set becomes

S → XY, X → a, Y → Z | b, Z → M, M → a, N → a

Now we will remove Z → M.

As M → a, we add Z→ a, and Z → M is removed.

The production set becomes

S → XY, X → a, Y → Z | b, Z → a, M → a, N → a

Now we will remove Y → Z.

As Z → a, we add Y→ a, and Y → Z is removed.

The production set becomes

S → XY, X → a, Y → a | b, Z → a, M → a, N → a

Now Z, M, and N are unreachable, hence we can remove those.

The final CFG is unit production free −

S → XY, X → a, Y → a | b

Removal of Null Productions

In a CFG, a non-terminal symbol **'A'** is a nullable variable if there is a production **A → ε** or there is a derivation that starts at **A** and finally ends up with

ε: A → .....… → ε

Removal Procedure

Step 1 − Find out nullable non-terminal variables which derive ε.

Step 2 − For each production **A → a**, construct all productions **A → x** where **x** is obtained from **'a'** by removing one or multiple non-terminals from Step 1.

Step 3 − Combine the original productions with the result of step 2 and remove **ε - productions**.

Problem

Remove null production from the following −

S → ASA | aB | b, A → B, B → b | ∈

Solution −

There are two nullable variables − **A** and **B**

At first, we will remove B → ε.

After removing **B → ε**, the production set becomes −

S→ASA | aB | b | a, A ε B| b | &epsilon, B → b

Now we will remove A → ε.

After removing **A → ε**, the production set becomes −

S→ASA | aB | b | a | SA | AS | S, A → B| b, B → b

This is the final production set without null transition.

5.8 Chomsky Normal Form

A CFG is in Chomsky Normal Form if the Productions are in the following forms −

- A → a
- A → BC
- S → ε

where A, B, and C are non-terminals and **a** is terminal.

Algorithm to Convert into Chomsky Normal Form −

Step 1 − If the start symbol **S** occurs on some right side, create a new start symbol **S'** and a new production **S'→ S**.

Step 2 − Remove Null productions. (Using the Null production removal algorithm discussed earlier)

Step 3 − Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

Step 4 − Replace each production $\mathbf{A \rightarrow B_1 \ldots B_n}$ where $\mathbf{n > 2}$ with $\mathbf{A \rightarrow B_1 C}$ where $\mathbf{C \rightarrow B_2 \ldots B_n}$. Repeat this step for all productions having two or more symbols in the right side.

Step 5 − If the right side of any production is in the form **A → aB** where a is a terminal and **A, B** are non-terminal, then the production is replaced by **A → XB** and **X → a**. Repeat this step for every production which is in the form **A → aB**.

Problem

Convert the following CFG into CNF

S → ASA | aB, A → B | S, B → b | ε

Solution

(1) Since **S** appears in R.H.S, we add a new state $\mathbf{S_0}$ and $\mathbf{S_0 \rightarrow S}$ is added to the production set and it becomes −

$S_0 \rightarrow S$, S→ ASA | aB, A → B | S, B → b | ∈

(2) Now we will remove the null productions −

$B \rightarrow \in$ and $A \rightarrow \in$

After removing $B \rightarrow \varepsilon$, the production set becomes −

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB \mid a$, $A \rightarrow B \mid S \mid \in$, $B \rightarrow b$

After removing $A \rightarrow \in$, the production set becomes −

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA \mid S$, $A \rightarrow B \mid S$, $B \rightarrow b$

(3) Now we will remove the unit productions.

After removing $S \rightarrow S$, the production set becomes −

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $A \rightarrow B \mid S$, $B \rightarrow b$

After removing $S_0 \rightarrow S$, the production set becomes −

$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$

$A \rightarrow B \mid S$, $B \rightarrow b$

After removing $A \rightarrow B$, the production set becomes −

$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$

$A \rightarrow S \mid b$

$B \rightarrow b$

After removing $A \rightarrow S$, the production set becomes −

$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$

$A \rightarrow b \mid ASA \mid aB \mid a \mid AS \mid SA$, $B \rightarrow b$

(4) Now we will find out more than two variables in the R.H.S

Here, $S_0 \rightarrow ASA$, $S \rightarrow ASA$, $A \rightarrow ASA$ violates two Non-terminals in R.H.S.

Hence we will apply step 4 and step 5 to get the following final production set which is in CNF −

$S_0 \rightarrow AX \mid aB \mid a \mid AS \mid SA$

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

$A \rightarrow b \mid AX \mid aB \mid a \mid AS \mid SA$

$B \rightarrow b$

$X \rightarrow SA$

(5) We have to change the productions $S_0 \rightarrow aB$, $S \rightarrow aB$, $A \rightarrow aB$

And the final production set becomes −

$S_0 \rightarrow AX \mid YB \mid a \mid AS \mid SA$

$S \rightarrow AX \mid YB \mid a \mid AS \mid SA$

$A \rightarrow b\ A \rightarrow b \mid AX \mid YB \mid a \mid AS \mid SA$

$B \rightarrow b$

$X \rightarrow SA$

$Y \rightarrow a$

5.9 Greibach Normal Form

A CFG is in Greibach Normal Form if the Productions are in the following forms −

$A \rightarrow b$

$A \rightarrow bD_1 \ldots D_n$

$S \rightarrow \varepsilon$

where A, D_1,....,D_n are non-terminals and b is a terminal.

Algorithm to Convert a CFG into Greibach Normal Form

Step 1 − If the start symbol **S** occurs on some right side, create a new start symbol **S'** and a new production **S' → S**.

Step 2 − Remove Null productions. (Using the Null production removal algorithm discussed earlier)

Step 3 − Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

Step 4 − Remove all direct and indirect left-recursion.

Step 5 − Do proper substitutions of productions to convert it into the proper form of GNF.

Problem

Convert the following CFG into CNF

$S \rightarrow XY \mid Xn \mid p$

$X \rightarrow mX \mid m$

$Y \rightarrow Xn \mid o$

Solution

Here, **S** does not appear on the right side of any production and there are no unit or null productions in the production rule set. So, we can skip Step 1 to Step 3.

Step 4

Now after replacing

X in $S \rightarrow XY \mid Xo \mid p$

with

$mX \mid m$

we obtain

$S \rightarrow mXY \mid mY \mid mXo \mid mo \mid p.$

And after replacing

X in $Y \rightarrow X_n \mid o$

with the right side of

$X \rightarrow mX \mid m$

we obtain

$Y \rightarrow mXn \mid mn \mid o.$

Two new productions $O \rightarrow o$ and $P \rightarrow p$ are added to the production set and then we came to the final GNF as the following −

$S \rightarrow mXY \mid mY \mid mXC \mid mC \mid p$

$X \rightarrow mX \mid m$

$Y \rightarrow mXD \mid mD \mid o$

$O \rightarrow o$

$P \rightarrow p$

5.10 Pumping Lemma for CFG

If **L** is a context-free language, there is a pumping length **p** such that any string **w ∈ L** of length ≥ **p** can be written as **w = uvxyz**, where **vy ≠ ε**, **|vxy| ≤ p**, and for all **i ≥ 0, $uv^ixy^iz \in L$**.

Applications of Pumping Lemma

Pumping lemma is used to check whether a grammar is context free or not. Let us take an example and show how it is checked.

Problem

Find out whether the language **$L = \{x^ny^nz^n \mid n \geq 1\}$** is context free or not.

Solution

Let **L** is context free. Then, **L** must satisfy pumping lemma.

At first, choose a number **n** of the pumping lemma. Then, take z as $0^n1^n2^n$.

Break **z** into **uvwxy,** where

|vwx| ≤ n and vx ≠ ε.

Hence **vwx** cannot involve both 0s and 2s, since the last 0 and the first 2 are at least (n+1) positions apart. There are two cases −

Case 1 − **vwx** has no 2s. Then **vx** has only 0s and 1s. Then **uwy**, which would have to be in **L**, has **n** 2s, but fewer than **n** 0s or 1s.

Case 2 − **vwx** has no 0s.

Here contradiction occurs.

Hence, **L** is not a context-free language.

5.11 Check your Progress

1.Which of the following CFG's can't be simulated by an FSM ?

a) S --> Sa | b b) S --> aSb | ab

c) S --> abX, X --> cY, Y --> d | aX d) None of these

2. Following context free grammar S —> aB | bA A —>b | aS | bAA B —> b | bS | aBB generates strings of terminals that have

a) equal number of a's and b's b) odd number of a's and odd number b's

c) even number of a's and even number of b's d) odd number of a's and even number of a's

3. Consider the grammar : S —> ABCc | Abc BA —> AB Bb —> bb Ab —> ab Aa —> aa Which of the following sentences can be derived by this grammar

a) abc b) aab

c) abcc d) abbb

4.Which of the following statement is correct?

a) All languages can not be generated by CFG b) Any regular language has an equivalent CFG

c) Some non regular languages can't be generated by CFG d) both (b) and (c)

5. In a context-free grammar

a) ε can't be the right hand side of any production

b) terminal symbols can't be present in the left hand side of any production

c) number of grammar symbols in the left hand side is not greater than the number of grammar symbols in the right hand side

d) all of these

6. A grammar that produces more than one parse tree for some sentence is called

a) Ambiguous b) Unambiguous

c) Regular d) none of these

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

a) context free b) not context free

c) not context free but whose complement is CF d) both (b) and (c)

8. Identify the language which is not context - free.

a) L = {ωωR|ωϵ{0,1}*} b) L = {a^nb^n|n≥0}

c) L = {ωω|ωϵ{0,1}*} d) L = {a^nb^mc^md^n | n, m≥0 }

9. The CFG s---> as | bs | a | b is equivalent to regular expression

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

c) (a + b) (a + b) d) None of these

10. Which of the following statement is wrong ?

a) Any regular language has an equivalent context-free grammar.

b) Some non-regular languages can't be generated by any context-free grammar

c) Intersection of context free language and a regular language is always context-free

d) All languages can be generated by context- free grammar

5.12 Summary

We study context-free grammars and languages. We define derivation trees and give methods of simplifying context-free grammars. The two normal forms-Chomsky normal form and Greibach normal form-are dealt with. Context-free languages are applied in parser design. They are also useful for describing block structures in programming languages. We study to remove Useless Symbols and unit Productions from CFG. Pumping Lemma is used to define grammer is Context free or not.

5.13 Keywords

1. Ambiguity : Ambiguity may be used to refer either to something (such as a word) which has multiple meanings, or to a more general state of uncertainty.

2. Derivation tree: The derivation tree is also called a **parse tree**. Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

3. Useless Symbols: Useless symbols are those **non-terminals or terminals that do not appear in any derivation of a string**. A symbol, Y is said to be useful if: that is Y should lead to a set of terminals. Here Y is said to be 'generating'. b. If there is a derivation, then Y is said to be 'reachable'.

5.14 Self Assessment Test

Q 1. Let G be the grammar with S→aB|bA A→a|aS|bAA B→b|bS|aBB

What do you mean by null production and unit production? Give an example.

Q 2. Construct a CFG for set of strings that contain equal number of a's and b's over ∑ = {a,b}.

Q 3. Mention the application of CFG

Q 4. State Chomsky normal form theorem.

Q 5.What is null and unit production

Q 6. What is the purpose of normalization? Construct the CNF and GNF for the following grammar and explain the steps.

S→aAa | bBb |€ A→C|a B→C|b C→CDE | € D→A|B|ab

(i) Constuct a CFG for the regular expression (011+1)(01).

Q 7. Construct a CFG over {a,b} generating a language consisting of equal number of a's and b's.

Q 8. Consider the following grammar G with productions

S→ ABC | BaB

A→aA|BaC|aaa

B→bBb|a

C→CA|AC Give a CFG with no useless variables that generates the same language.

5.15 Answer to Check your Progress

1. **B**
2. **A**
3. **A**

4. D

5. D

6. D

7. D

8. B

9. B

10. D

5.16 REFERENCES/SUGGESTED READINGS

1.Hopcroaft & O. D. Ullman, R Mothwani, Introduction to automata theory, language & computations, AW, 2001.

2. K. L. P. Mishra & N. Chandrasekaran, . Theory of Computer Sc.(Automata, Languages and computation), PHI, 2000.

3. Peter Linz, Introduction to formal Languages & Automata, Narosa, Publication, 2001.

4. Ramond Greenlaw and H. James Hoover, Fundamentals of the Theory of ComputationPrinciples and Practice, Harcourt India Pvt. Ltd., 1998.

5. H. R. Lewis & C. H. Papaditriou, Elements of theory of Computation, PHC, 1998.

6. John C. Martin, Introduction to Languages and the Theory of Computation, T.M.H., 2003

SUBJECT: Theory of Automaton	
COURSE CODE: MCA-35	AUTHOR: RAVIKA GOEL
LESSON NO. 6	
Push Down Automata	

STRUCTURE

6.0 OBJECTIVE

The main objective of this lesson is to study the Push Down Automaton. What are the applications of Push Down Automaton. We discuss two types of acceptance of sets by pushdown automata. Finally, we prove that the sets accepted by pushdown automata are precisely the class of context-free languages.

6.1 INTRODUCTION

A push down automata is similar to deterministic finite automata except that it has a few more properties than a DFA.The data structure used for implementing a PDA is stack. A PDA has an output associated with every input. All the inputs are either pushed into a stack or just ignored. User can perform the basic push and pop operations on the stack which is use for PDA. One of the problems associated with DFAs was that could not make a count of number of characters which were given input to the machine. This problem is avoided by PDA as it uses a stack which provides us this facility also. Pushdown automata are used in theories about what can be computed by machines. They are more capable than finite-state machines but less capable than Turing machines (see below). Deterministic pushdown automata can recognize all deterministic context-free languages while nondeterministic ones can recognize all context-free languages, with the former often used in parser design.

The term "pushdown" refers to the fact that the stack can be regarded as being "pushed down" like a tray dispenser at a cafeteria, since the operations never work on elements other than the top element. A **stack automaton**, by contrast, does allow access to and operations on deeper elements. Stack automata can recognize a strictly larger set of languages than pushdown automata. A nested stack automaton allows full access, and also allows stacked values to be entire sub-stacks rather than just single finite symbols.

6.2 Introduction of Push Down automaton

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is −

"Finite state machine" + "a stack"

A pushdown automaton has three components −

- an input tape,

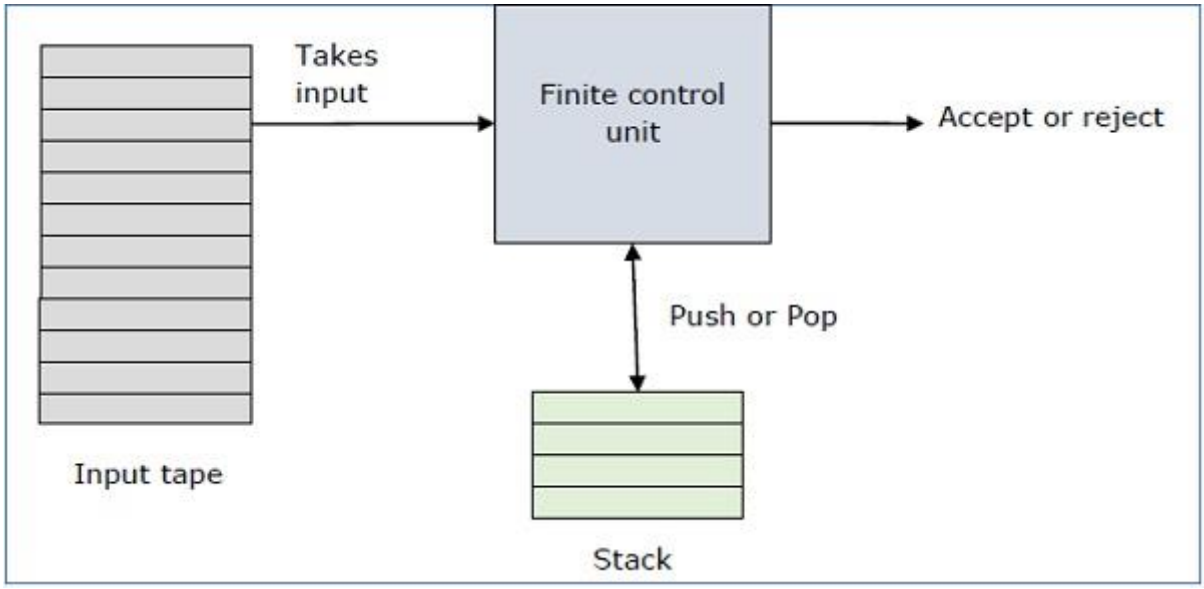
- a control unit, and
- a stack with infinite size.

The stack head scans the top symbol of the stack.

A stack does two operations −

- **Push** − a new symbol is added at the top.
- **Pop** − the top symbol is read and removed.

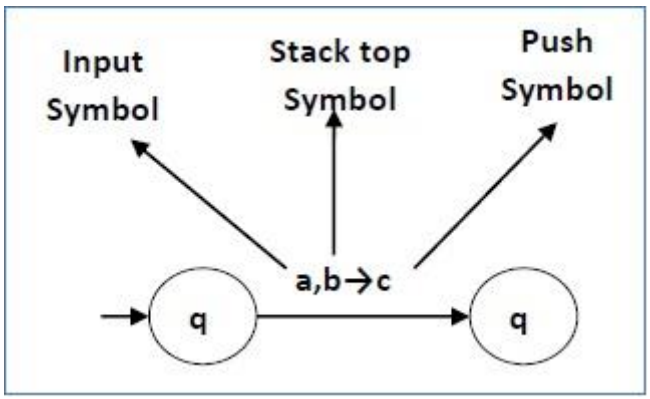
A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.



A PDA can be formally described as a 7-tuple $(Q, \sum, S, \delta, q_0, I, F)$ −

- **Q** is the finite number of states
- $\sum$ is input alphabet
- **S** is stack symbols
- **δ** is the transition function: $Q \times (\sum \cup \{\varepsilon\}) \times S \times Q \times S^*$
- $\mathbf{q_0}$ is the initial state ($q_0 \in Q$)
- **I** is the initial stack top symbol ($I \in S$)
- **F** is a set of accepting states ($F \in Q$)

The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as $a,b \rightarrow c$ −



This means at state $\mathbf{q_1}$, if we encounter an input string **'a'** and top symbol of the stack is **'b'**, then we pop **'b'**, push **'c'** on top of the stack and move to state $\mathbf{q_2}$.

Terminologies Related to PDA

Instantaneous Description

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where

- **q** is the state
- **w** is unconsumed input
- **s** is the stack contents

Turnstile Notation

The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol "⊢".

Consider a PDA (Q, ∑, S, δ, q_0, I, F). A transition can be mathematically represented by the following turnstile notation −

$$(p, aw, T\beta) \vdash (q, w, \alpha b)$$

This implies that while taking a transition from state **p** to state **q**, the input symbol **'a'** is consumed, and the top of the stack **'T'** is replaced by a new string **'α'**.

Note − If we want zero or more moves of a PDA, we have to use the symbol (⊢*) for it.

There are two different ways to define PDA acceptability.

Final State Acceptability

In final state acceptability, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. From the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

For a PDA (Q, ∑, S, δ, q_0, I, F), the language accepted by the set of final states F is −

L(PDA) = {w | (q_0, w, I) ⊢* (q, ε, x), q ∈ F}

for any input stack string **x**.

Empty Stack Acceptability

Here a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.

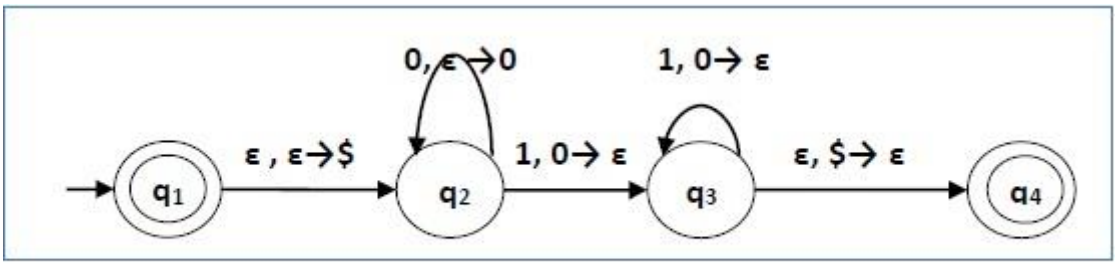
For a PDA (Q, ∑, S, δ, q_0, I, F), the language accepted by the empty stack is −

L(PDA) = {w | (q_0, w, I) ⊢* (q, ε, ε), q ∈ Q}

Example

Construct a PDA that accepts **L = {0^n 1^n | n ≥ 0}**

Solution



PDA for L= {0^n 1^n | n≥0}

This language accepts L = {ε, 01, 0011, 000111, ... }

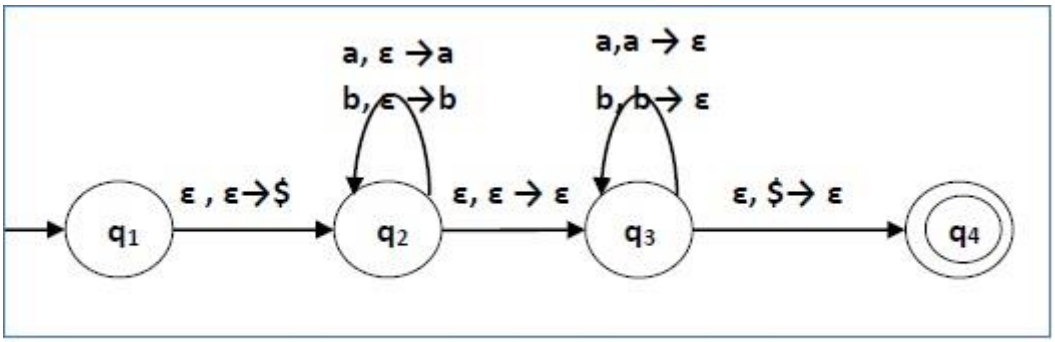
Here, in this example, the number of **'a'** and **'b'** have to be same.

- Initially we put a special symbol **'$'** into the empty stack.
- Then at state **q_2**, if we encounter input 0 and top is Null, we push 0 into stack. This may iterate. And if we encounter input 1 and top is 0, we pop this 0.
- Then at state **q_3**, if we encounter input 1 and top is 0, we pop this 0. This may also iterate. And if we encounter input 1 and top is 0, we pop the top element.
- If the special symbol '$' is encountered at top of the stack, it is popped out and it finally goes to the accepting state q_4.

Example

Construct a PDA that accepts L = { ww^R | w = (a+b)* }

Solution



PDA for L= {ww^R | w = (a+b)*}

Initially we put a special symbol '$' into the empty stack. At state $\mathbf{q_2}$, the **w** is being read. In state $\mathbf{q_3}$, each 0 or 1 is popped when it matches the input. If any other input is given, the PDA will go to a dead state. When we reach that special symbol '$', we go to the accepting state $\mathbf{q_4}$.

6.3 Algorithm to find PDA corresponding to a given CFG

If a grammar **G** is context-free, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the context-free grammar **G**. A parser can be built for the grammar **G**.

Also, if **P** is a pushdown automaton, an equivalent context-free grammar G can be constructed where

L(G) = L(P)

In the next two topics, we will discuss how to convert from PDA to CFG and vice versa.

Algorithm to find PDA corresponding to a given CFG

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F)

Step 1 − Convert the productions of the CFG into GNF.

Step 2 − The PDA will have only one state {q}.

Step 3 − The start symbol of CFG will be the start symbol in the PDA.

Step 4 − All non-terminals of the CFG will be the stack symbols of the PDA and all the terminals of the CFG will be the input symbols of the PDA.

Step 5 − For each production in the form **A → aX** where a is terminal and **A, X** are combination of terminal and non-terminals, make a transition **δ (q, a, A)**.

Problem

Construct a PDA from the following CFG.

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

where the productions are −

S → XS | ε , A → aXb | Ab | ab

Solution

Let the equivalent PDA,

P = ({q}, {a, b}, {a, b, X, S}, δ, q, S)

where δ −

δ(q, ε , S) = {(q, XS), (q, ε)}

δ(q, ε , X) = {(q, aXb), (q, Xb), (q, ab)}

δ(q, a, a) = {(q, ε)}

δ(q, 1, 1) = {(q, ε)}

Input − A CFG, G = (V, T, P, S)

6.4 Algorithm to find CFG corresponding to a given PDA

Output − Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F) such that the non- terminals of the grammar G will be $\{X_{wx} \mid w,x \in Q\}$ and the start state will be $A_{q0,F}$.

Step 1 − For every w, x, y, z ∈ Q, m ∈ S and a, b ∈ ∑, if δ (w, a, ε) contains (y, m) and (z, b, m) contains (x, ε), add the production rule $X_{wx} \rightarrow a\, X_{yz} b$ in grammar G.

Step 2 − For every w, x, y, z ∈ Q, add the production rule $X_{wx} \rightarrow X_{wy}X_{yx}$ in grammar G.

Step 3 − For w ∈ Q, add the production rule $X_{ww} \rightarrow \varepsilon$ in grammar G.

The **Applications** of these Automata are given as follows:

6.5 Applications of Push Down Automaton

1. Finite Automata (FA) –

- For the designing of lexical analysis of a compiler.
- For recognizing the pattern using regular expressions.
- For the designing of the combination and sequential circuits using Mealy and Moore Machines.
- Used in text editors.
- For the implementation of spell checkers.

2. Push Down Automata (PDA) –

- For designing the parsing phase of a compiler (Syntax Analysis).
- For implementation of stack applications.
- For evaluating the arithmetic expressions.
- For solving the Tower of Hanoi Problem.

3. Linear Bounded Automata (LBA) –

- For implementation of genetic programming.
- For constructing syntactic parse trees for semantic analysis of the compiler.

4. Turing Machine (TM) –

- For solving any recursively enumerable problem.
- For understanding complexity theory.
- For implementation of neural networks.
- For implementation of Robotics Applications
- For implementation of artificial intelligence.

6.6 Parsing

Parsing is used to derive a string using the production rules of a grammar. It is used to check the acceptability of a string. Compiler is used to check whether or not a string is syntactically correct. A parser takes the inputs and builds a parse tree.

A parser can be of two types −

- **Top-Down Parser** − Top-down parsing starts from the top with the start-symbol and derives a string using a parse tree.
- **Bottom-Up Parser** − Bottom-up parsing starts from the bottom with the string and comes to the start symbol using a parse tree.

Design of Top-Down Parser

For top-down parsing, a PDA has the following four types of transitions −

- Pop the non-terminal on the left hand side of the production at the top of the stack and push its right-hand side string.
- If the top symbol of the stack matches with the input symbol being read, pop it.
- Push the start symbol 'S' into the stack.
- If the input string is fully read and the stack is empty, go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q_0, I, F), then the top-down parsing is −

(x+y*z, I) ⊢(x +y*z, SI) ⊢ (x+y*z, S+XI) ⊢(x+y*z, X+XI)

⊢(x+y*z, Y+X I) ⊢(x+y*z, x+XI) ⊢(+y*z, +XI) ⊢ (y*z, XI)

⊢(y*z, X*YI) ⊢(y*z, y*YI) ⊢(*z,*YI) ⊢(z, YI) ⊢(z, zI) ⊢(ε, I)

Design of a Bottom-Up Parser

For bottom-up parsing, a PDA has the following four types of transitions −

- Push the current input symbol into the stack.
- Replace the right-hand side of a production at the top of the stack with its left-hand side.
- If the top of the stack element matches with the current input symbol, pop it.
- If the input string is fully read and only if the start symbol 'S' remains in the stack, pop it and go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q_0, I, F), then the bottom-up parsing is −

(x+y*z, I) ⊢ (+y*z, xI) ⊢ (+y*z, YI) ⊢ (+y*z, XI) ⊢ (+y*z, SI)

⊢(y*z, +SI) ⊢ (*z, y+SI) ⊢ (*z, Y+SI) ⊢ (*z, X+SI) ⊢ (z, *X+SI)

⊢ (ε, z*X+SI) ⊢ (ε, Y*X+SI) ⊢ (ε, X+SI) ⊢ (ε, SI)

6.7 Examples of Push Down Automaton

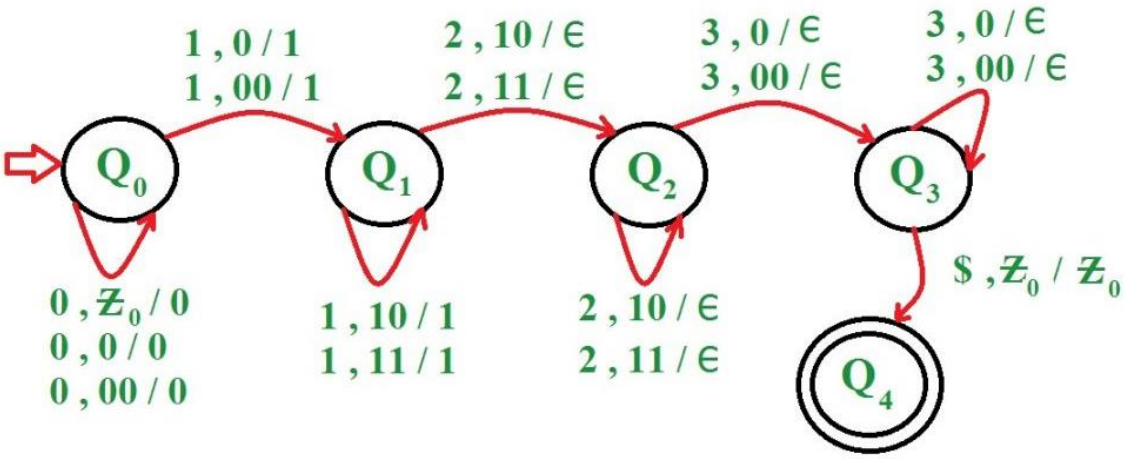
Q) Construct a PDA for language L = $\{0^n1^m2^m3^n \mid n>=1, m>=1\}$

Approach used in this PDA –

First 0's are pushed into stack. Then 1's are pushed into stack.

Then for every 2 as input a 1 is popped out of stack. If some 2's are still left and top of stack is a 0 then string is not accepted by the PDA. Thereafter if 2's are finished and top of stack is a 0 then for every 3 as input equal number of 0's are popped out of stack. If string is finished and stack is empty then string is accepted by the PDA otherwise not accepted.

- **Step-1:** On receiving 0 push it onto stack. On receiving 1, push it onto stack and goto next state
- **Step-2:** On receiving 1 push it onto stack. On receiving 2, pop 1 from stack and goto next state
- **Step-3:** On receiving 2 pop 1 from stack. If all the 1's have been popped out of stack and now receive 3 then pop a 0 from stack and goto next state
- **Step-4:** On receiving 3 pop 0 from stack. If input is finished and stack is empty then goto last state and string is accepted



Examples:

Input : 0 0 1 1 1 2 2 2 3 3

Result : ACCEPTED

Input : 0 0 0 1 1 2 2 2 3 3

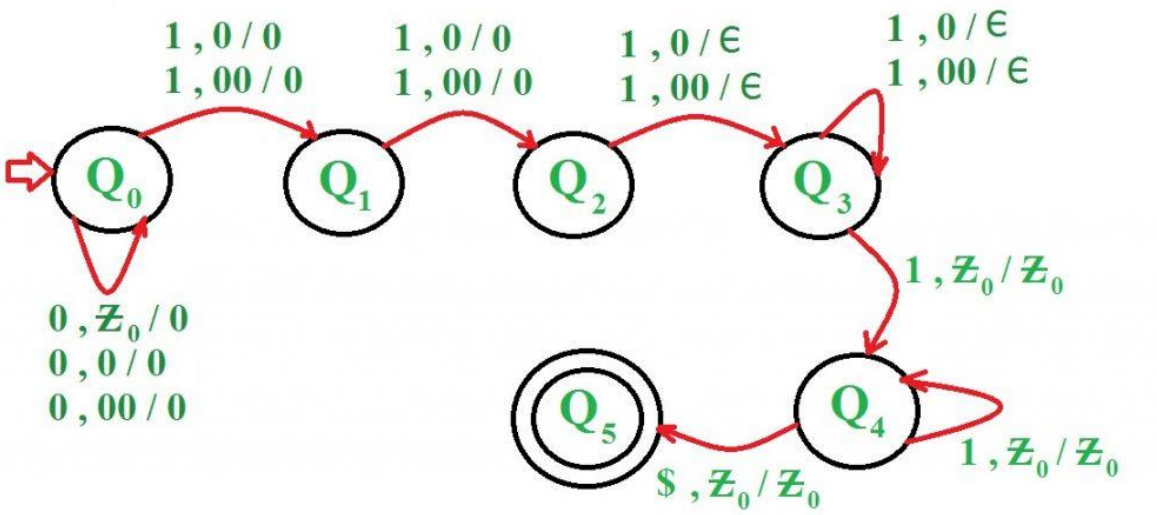
Result : NOT ACCEPTED

Q) Construct a PDA for language L = $\{0^n1^m \mid n >= 1, m >= 1, m > n+2\}$

Approach used in this PDA –

First 0's are pushed into stack.When 0's are finished, two 1's are ignored. Thereafter for every 1 as input a 0 is popped out of stack. When stack is empty and still some 1's are left then all of them are ignored.

- **Step-1:** On receiving 0 push it onto stack. On receiving 1, ignore it and goto next state
- **Step-2:** On receiving 1, ignore it and goto next state
- **Step-3:** On receiving 1, pop a 0 from top of stack and go to next state
- **Step-4:** On receiving 1, pop a 0 from top of stack. If stack is empty, on receiving 1 ignore it and goto next state
- **Step-5:** On receiving 1 ignore it. If input is finished then goto last state



Examples:

Input : 0 0 0 1 1 1 1 1 1

Result : ACCEPTED

Input : 0 0 0 0 1 1 1 1

Result : NOT ACCEPTED

Q) Construct Pushdown Automata for all length palindrome

Pushdown Automaton (PDA) is like an epsilon Non deterministic Finite Automata (NFA) with infinite stack. PDA is a way to implement context free languages. Hence, it is important to learn, how to draw PDA.

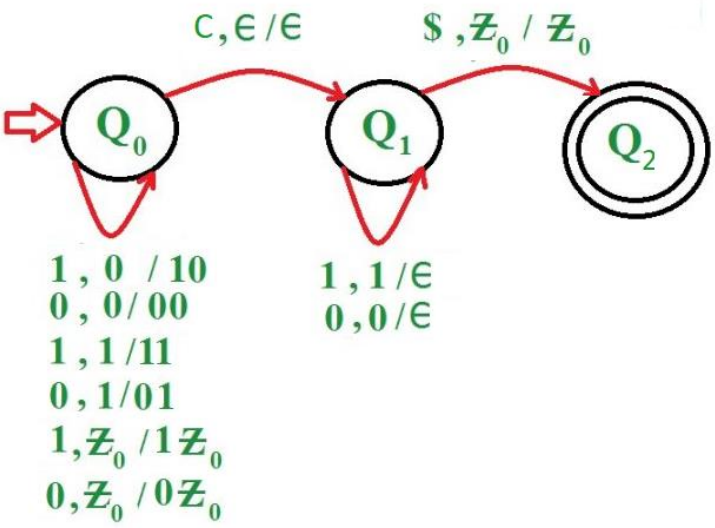
Here, take the example of odd length palindrome:

Q) Construct a PDA for language L = {wcw' | w={0, 1}*} where w' is the reverse of w.

Approach used in this PDA –

Keep on pushing 0's and 1's no matter whatever is on the top of stack until reach the middle element. When middle element 'c' is scanned then process it without making any changes in stack. Now if scanned symbol is '1' and top of stack also contain '1' then pop the element from top of stack or if scanned symbol is '0' and top of stack also contain '0' then pop the element from top of stack. If string becomes empty or scanned symbol is '$' and stack becomes empty, then reach to final state else move to dead state.

- **Step 1:** On receiving 0 or 1, keep on pushing it on top of stack without going to next state.
- **Step 2:** On receiving an element 'c', move to next state without making any change in stack.
- **Step 3:** On receiving an element, check if symbol scanned is '1' and top of stack also contain '1' or if symbol scanned is '0' and top of stack also contain '0' then pop the element from top of stack else move to dead state. Keep on repeating step 3 until string becomes empty.
- **Step 4:** Check if symbol scanned is '$' and stack does not contain any element then move to final state else move to dead state.



Examples:

Input : 1 0 1 0 1 0 1 0 1

Output :ACCEPTED

Input : 1 0 1 0 1 1 1 1 0

Output :NOT ACCEPTED

Now, take the example of even length palindrome:

Q) Construct a PDA for language L = {ww' | w={0, 1}*} where w' is the reverse of w.

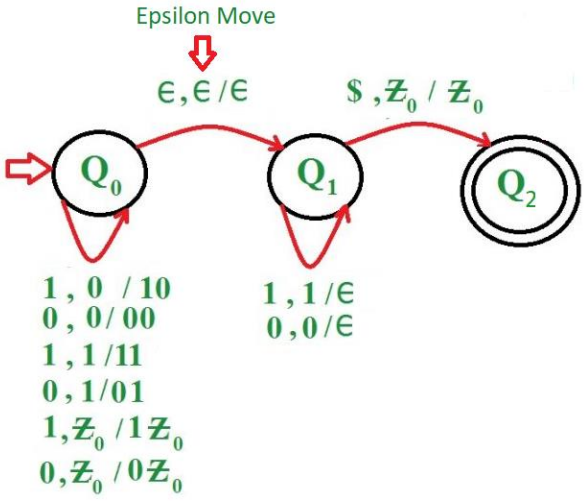
Approach used in this PDA –

For construction of even length palindrome, user has to use Non Deterministic Pushdown Automata (NPDA). A NPDA is basically an NFA with a stack added to it.

The NPDA for this language is identical to the previous one except for epsilon transition. However, there is a significant difference, that this PDA must guess when to stop pushing symbols, jump to the final state and start matching off of the stack. Therefore this machine is decidedly non-deterministic. Keep on pushing 0's and 1's no matter whatever is on the top of stack and at the same time keep a check on the input string, whether reach to the second half of input string or not. If reach to last element of first half of the input string then after processing the last element of first half of input string make an epsilon move and move to next state. Now if scanned symbol is '1' and top of stack also contain '1' then pop the element from top of stack or if scanned symbol is '0' and top of stack also contain '0' then pop the element from top of stack. If string becomes empty or scanned symbol is '$' and stack becomes empty, then reach to final state else move to dead state.

- **Step 1:** On receiving 0 or 1, keep on pushing it on top of stack and at a same time keep on checking whether reach to second half of input string or not.
- **Step 2:** If reach to last element of first half of input string, then push that element on top of stack and then make an epsilon move to next state.
- **Step 3:** On receiving an element, check if symbol scanned is '1' and top of stack also contain '1' or if symbol scanned is '0' and top of stack also contain '0' then pop the element from top of stack else move to dead state. Keep on repeating step 3 until string becomes empty.

- **Step 4:** Check if symbol scanned is '$' and stack does not contain any element then move to final state else move to dead state.



Examples:

Input : 1 0 0 1 1 1 1 0 0 1

Output :ACCEPTED

Input : 1 0 0 1 1 1

Output :NOT ACCEPTED

Now, take the example of all length palindrome, i.e. a PDA which can accept both odd length palindrome and even length palindrome:

Q): Construct a PDA for language L = {ww' | wcw', w={0, 1}*} where w' is the reverse of w.

Approach used in this PDA –

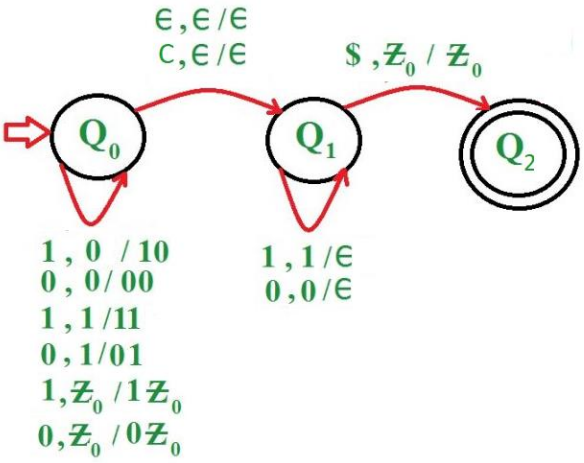
For construction of all length palindrome, user has to use NPDA.

The approach is similar to above example, except now along with epsilon move now user has to show one more transition move of symbol 'c' i.e. if string is of odd length and if reach to middle element 'c' then just process it and move to next state without making any change in stack.

- **Step 1:** On receiving 0 or 1, keep on pushing it on top of stack and at a same time keep on checking, if input string is of even length then whether reach to second half of input string or

not, however if the input string is of odd length then keep on checking whether reach to middle element or not.

- **Step 2:** If input string is of even length and reach to last element of first half of input string, then push that element on top of stack and then make an epsilon move to next state or if the input string is of odd length then on receiving an element 'c', move to next state without making any change in stack.
- **Step 3:** On receiving an element, check if symbol scanned is '1' and top of stack also contain '1' or if symbol scanned is '0' and top of stack also contain '0' then pop the element from top of stack else move to dead state. Keep on repeating step 3 until string becomes empty.
- **Step 4:** Check if symbol scanned is '$' and stack does not contain any element then move to final state else move to dead state.



Examples:

Input : 1 1 0 0 1 1 1 1 0 0 1 1

Output :ACCEPTED

Input : 1 0 1 0 1 0 1

Output :ACCEPTED

6.8 Check your Progress

1. PDA is more Powerful than

a) Turing Machine b) Finite automata

b) Both (a) and (b) d) None of these

2. Which operation is applied on stack

a) PUSH b) POP

b) Both (a) and (b) d) None of these

3. PDA can be Represented with the help of

a) Instantaneous description b) Trasition diagram

b) Trasition Table d) all of these

4. A Push Down Automaton is different than finite automata by

a) Its Memory(stack) b) number of states

c) Both (a) and (b) d) None of these

5. Which type of symbol contain in the stack of PDA

a)Variables b) Terminals

c) Both (a) and (b) d) None of these

6. Which of the following is not possible algorithmically ?

a) Regular grammar to context free grammar

b) Non-deterministic FSA to deterministic FSA

c) Non-deterministic PDA to deterministic PDA

d) None of these

7. Push Down automaton indicate the acceptance of input string in terms of

a) Final state b) empty store

c) Both (a) and (b) d) None of these

8. A PDA chooses the next move based on:

a) Current state b) Next input symbol

a) Both (a) and (b) d) none of these

6.9 Summary

We introduce pushdown automaton (pda). We discuss two types of acceptance of sets by pushdown automata. Finally, we prove that the sets accepted by pushdown automata are precisely the class of context-free languages. We have seen that the regular languages are precisely those accepted by finite automata. If M is a finite automaton accepting L, it is constructed in such a way that states act as a form of primitive memory. The states 'remember' the variables encountered in the course of derivation of a string. (In M, the states correspond to variables.) Let us consider L = {a"b"ln I}. This is a contextfree language but not regular. (S -----i aSh Iab generates L. Using the pumping lemma we can show that L is not regular, A finite automaton cannot accept L, i.e. strings of the form a"b", as it has to remember the number of a's in a string and so it will require an infinite number of states. This difficulty can be avoided by adding an auxiliary memory in the form of a 'stack' (In a stack we add the elements in a linear way. While removing the elements we follow the last-in-first-out (LIFO) basis. i.e. the most recently added element is removed first.) The a's in the given string are added to the stack. When the symbol b is encountered in the input string, an a is removed from the stack. Thus the matching of number of c's and the number of b's is accomplished. This type of arrangement where a finite automaton has a stack leads to the generation of a pushdown automaton.

6.10 Keywords

1. Palindrome: A palindrome is a word, phrase, number, or sequence of words that reads the same backward as forward.

2. NPDA: This PDA is a non-deterministic PDA because finding the mid for the given string and reading the string from left and matching it with from right (reverse) direction leads to non-deterministic moves.
3. Parsing: Parsing *is the process of converting* formatted text *into a* data structure. *A data structure type can be any suitable representation of the information engraved in the* source text.

6.11 Self Assessment Test

Q1. State the definition of Pushdown automata.

Q 2. What are the different ways of language acceptances by a PDA

Q 3. When is Push Down Automata (PDA) said to be deterministic?

Q 4. List the main application of pumping Lemma in CFL

Q 5. Compare Deterministic and Non deterministic PDA. Is it true that non deterministic PDA is more powerful than that of deterministic PDA? Justify your answer.

Q 6. Construct the PDA accepting the language

1. L={(ab)n |n>=1} by empty stack.

2. L={a2nb n |n>=1} Trace your PDA for the input with n=3.

3. L={wwR|w is in (a+b)*}

4. L={0n1 2n} by empty stack(8)

5. L={wwRw|w is in {0+1}*}

Q 7. Find the PDA equivalent to the given CFG with the following productions

1. S→A, A→BC, B→ba, C→ac

2. S→aSb|A, A→bSa|S| ε

Q 8. Prove that if there exists a PDA that accepts by final state then there exists an equivalent PDA Cthat accepts by Null state

Q 9. Construct PDA for the language L = {wwR | W in (a+b)*)}

Q 10.State pumping lemma for CFL.

6.12 Answer to Check your Progress

1. B

2. C

3. D

4. A

5. C

6. C

7. C

8. C

6.13 REFERENCES/ SUGGESTED READINGS

1.Hopcroaft & O. D. Ullman, R Mothwani, Introduction to automata theory, language & computations, AW, 2001.

2. K. L. P. Mishra & N. Chandrasekaran, . Theory of Computer Sc.(Automata, Languages and computation), PHI, 2000.

3. Peter Linz, Introduction to formal Languages & Automata, Narosa, Publication, 2001.

4. Ramond Greenlaw and H. James Hoover, Fundamentals of the Theory of ComputationPrinciples and Practice, Harcourt India Pvt. Ltd., 1998.

5. H. R. Lewis & C. H. Papaditriou, Elements of theory of Computation, PHC, 1998.

6. John C. Martin, Introduction to Languages and the Theory of Computation, T.M.H., 2003

SUBJECT: Theory of computation	
COURSE CODE: MCA-35	AUTHOR: RAVIKA GOEL
LESSON NO. 7	
Introduction of Turing Machines	

STRUCTURE

7.0 OBJECTIVE

The main objective of this lesson is to study the Concept of Turing Machine.

Find out the difference between deterministic and Non Deterministic Turing machine.

Discuss the Halting Problem and PCP Proplem of Turing Machine.

7.1 INTRODUCTION

A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing. In the early 1930s. mathematicians were trying to define effective computation. Alan Turing in 1936. Alanzo Church in 1933, S.C. Kleene in 1935, Schonfinkel in 1965 gave various models using the concept of Turing machines, JL-calculus, combinatory logic, post-systems and p-recursive functions. It is interesting to note that these were formulated much before the electro-mechanical electronic computers were devised. Although these formalisms, describing effective computations. are dissimilar, they tum to be equivalent. Among these formalisms, the Turing's formulation is accepted as a model of algorithm or computation. The Church-Turing thesis states that any algorithmic procedure that can be carried out by human beings/computer can be carried out by a Turing machine. It has been universally accepted by computer scientists that the Turing machine provides an ideal theoretical model of a computer. Turing machines are useful in several ways. As an automaton, the Turing machine is the most general model. It accepts type-O languages. It can also be used for computing functions. It turns out to be a mathematical model of partial recursive functions. Turing machines are also used for determining the undecidability of certain languages and measuring the space and time complexity of problems. These are the topics of discussion in this chapter and some of the subsequent chapters. For fonnalizing computability, Turing assumed that, while computing, a person writes symbols on a one-dimensional paper (instead of a two dimension paper as is usually done) which can be viewed as a tape divided into cells. One scans the cells one at a time and usually performs one of the three simple operations, namely (i) writing a new symbol in the cell being currently scanned, (ii) moving to the cell left of the present celL and (iii) moving to the cell light of the present cell. With these observations in mind, Turing proposed his 'computing machine.

7.2 Introduction of Turing Machine

A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing.

Definition

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of 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 TM can be formally described as a 7-tuple (Q, X, ∑, δ, q_0, B, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- ∑ is the input alphabet
- **δ** is a transition function; δ : Q × X → Q × X × {Left_shift, Right_shift}.
- **q_0** is the initial state
- **B** is the blank symbol
- **F** is the set of final states

Comparison with the previous automaton

The following table shows a comparison of how a Turing machine differs from Finite Automaton and Pushdown Automaton.

Machine	Stack Data Structure	Deterministic?
Finite Automaton	N.A	Yes
Pushdown Automaton	Last In First Out(LIFO)	No
Turing Machine	Infinite tape	Yes

Example of Turing machine

Turing machine M = (Q, X, ∑, δ, q_0, B, F) with

- Q = {q_0, q_1, q_2, q_f}
- X = {a, b}
- ∑ = {1}

- $q_0 = \{q_0\}$
- B = blank symbol
- $F = \{q_f\}$

δ is given by −

Tape alphabet symbol	Present State 'q_0'	Present State 'q_1'	Present State 'q_2'
A	$1Rq_1$	$1Lq_0$	$1Lq_f$
B	$1Lq_2$	$1Rq_1$	$1Rq_f$

Here the transition $1Rq_1$ implies that the write symbol is 1, the tape moves right, and the next state is q_1. Similarly, the transition $1Lq_2$ implies that the write symbol is 1, the tape moves left, and the next state is q_2.

Time and Space Complexity of a Turing Machine

For a Turing machine, the time complexity refers to the measure of the number of times the tape moves when the machine is initialized for some input symbols and the space complexity is the number of cells of the tape written.

Time complexity all reasonable functions −

T(n) = O(n log n)

TM's space complexity −

S(n) = O(n)

A TM accepts a language if it enters into a final state for any input string w. A language is recursively enumerable (generated by Type-0 grammar) if it is accepted by a Turing machine.

A TM decides a language if it accepts it and enters into a rejecting state for any input not in the language. A language is recursive if it is decided by a Turing machine.

There may be some cases where a TM does not stop. Such TM accepts the language, but it does not decide it.

7.3 Design of Turing Machine

Designing a Turing Machine

The basic guidelines of designing a Turing machine have been explained below with the help of a couple of examples.

Example 1

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 $\mathbf{q_1}$ 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,

$M = \{\{q_1, q_2\}, \{1\}, \{1, B\}, \delta, q_1, B, \{q_2\}\}$

where δ is given by −

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

Example 2

Design a Turing Machine that reads a string representing a binary number and erases all leading 0's in the string. However, if the string comprises of only 0's, it keeps one 0.

Solution

Let us assume that the input string is terminated by a blank symbol, B, at each end of the string.

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

- Let $\mathbf{q_0}$ be the initial state.
- If **M** is in $\mathbf{q_0}$, on reading 0, it moves right, enters the state $\mathbf{q_1}$ and erases 0. On reading 1, it enters the state $\mathbf{q_2}$ and moves right.
- If **M** is in $\mathbf{q_1}$, on reading 0, it moves right and erases 0, i.e., it replaces 0's by B's. On reaching the leftmost 1, it enters $\mathbf{q_2}$ and moves right. If it reaches B, i.e., the string comprises of only 0's, it moves left and enters the state $\mathbf{q_3}$.
- If **M** is in $\mathbf{q_2}$, on reading either 0 or 1, it moves right. On reaching B, it moves left and enters the state $\mathbf{q_4}$. This validates that the string comprises only of 0's and 1's.
- If **M** is in $\mathbf{q_3}$, it replaces B by 0, moves left and reaches the final state $\mathbf{q_f}$.
- If **M** is in $\mathbf{q_4}$, on reading either 0 or 1, it moves left. On reaching the beginning of the string, i.e., when it reads B, it reaches the final state $\mathbf{q_f}$.

Hence,

$M = \{\{q_0, q_1, q_2, q_3, q_4, q_f\}, \{0,1, B\}, \{1, B\}, \delta, q_0, B, \{q_f\}\}$

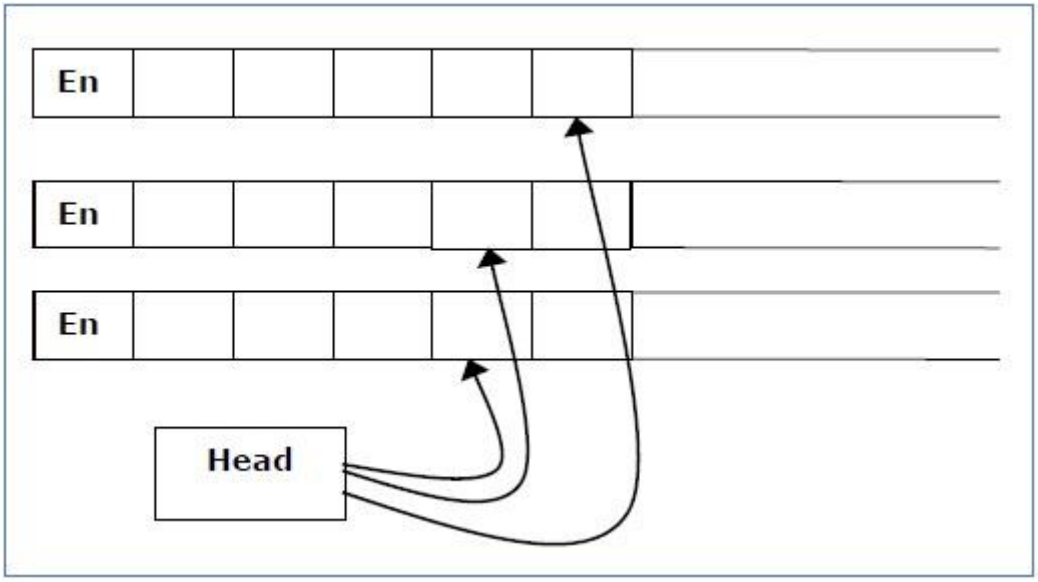
where δ is given by −

Tape alphabet symbol	Present State 'q_0'	Present State 'q_1'	Present State 'q_2'	Present State 'q_3'	Present State 'q_4'
0	BRq_1	BRq_1	ORq_2	-	OLq_4
1	$1Rq_2$	$1Rq_2$	$1Rq_2$	-	$1Lq_4$
B	BRq_1	BLq_3	BLq_4	OLq_f	BRq_f

in the system sharply increases, the mean waiting time remains nearly constant.

7.4 Types of Turing Machine

Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads.



A Multi-tape Turing machine can be formally described as a 6-tuple (Q, X, B, δ, q_0, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- **B** is the blank symbol
- **δ** is a relation on states and symbols where

  $\delta: Q \times X^k \rightarrow Q \times (X \times \{Left\_shift, Right\_shift, No\_shift\})^k$

  where there is **k** number of tapes
- **q_0** is the initial state
- **F** is the set of final states

Note − Every Multi-tape Turing machine has an equivalent single-tape Turing machine.

Multi-track Turing machines, a specific type of Multi-tape Turing machine, contain multiple tracks but just one tape head reads and writes on all tracks. Here, a single tape head reads n symbols from **n** tracks at one step. It accepts recursively enumerable languages like a normal single-track single-tape Turing Machine accepts.

A Multi-track Turing machine can be formally described as a 6-tuple (Q, X, ∑, δ, q_0, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- ∑ is the input alphabet
- **δ** is a relation on states and symbols where

  $\delta(Q_i, [a_1, a_2, a_3,....]) = (Q_j, [b_1, b_2, b_3,....]$, Left_shift or Right_shift)
- $\mathbf{q_0}$ is the initial state
- **F** is the set of final states

Note − For every single-track Turing Machine **S**, there is an equivalent multi-track Turing Machine **M** such that **L(S) = L(M)**.

In a Non-Deterministic Turing Machine, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration.

An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is called a **Decider** and if for some input, all branches are rejected, the input is also rejected.

A non-deterministic Turing machine can be formally defined as a 6-tuple (Q, X, ∑, δ, q_0, B, F) where −

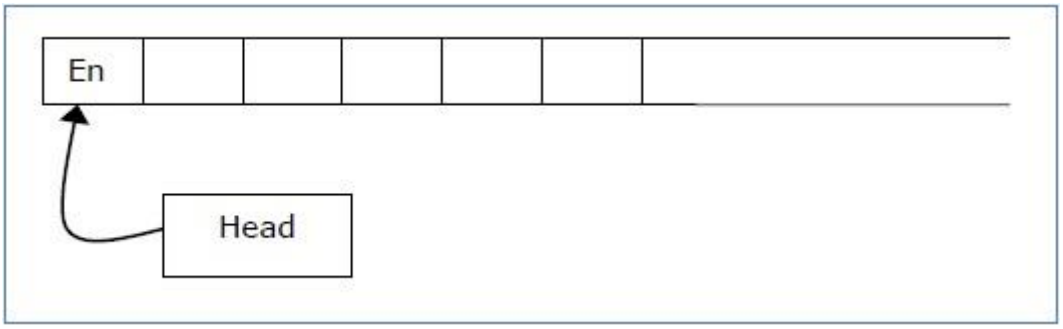
- **Q** is a finite set of states
- **X** is the tape alphabet
- ∑ is the input alphabet

- **δ** is a transition function;

  $\delta : Q \times X \rightarrow P(Q \times X \times \{Left\_shift, Right\_shift\})$.

- $\mathbf{q_0}$ is the initial state
- **B** is the blank symbol
- **F** is the set of final states

A Turing Machine with a semi-infinite tape has a left end but no right end. The left end is limited with an end marker.



It is a two-track tape −

- **Upper track** − It represents the cells to the right of the initial head position.
- **Lower track** − It represents the cells to the left of the initial head position in reverse order.

The infinite length input string is initially written on the tape in contiguous tape cells.

The machine starts from the initial state $\mathbf{q_0}$ and the head scans from the left end marker 'End'. In each step, it reads the symbol on the tape under its head. It writes a new symbol on that tape cell and then it moves the head either into left or right one tape cell. A transition function determines the actions to be taken.

It has two special states called **accept state** and **reject state**. If at any point of time it enters into the accepted state, the input is accepted and if it enters into the reject state, the input is rejected by the TM. In some cases, it continues to run infinitely without being accepted or rejected for some certain input symbols.

Note − Turing machines with semi-infinite tape are equivalent to standard Turing machines.

7.5 Linear bounded Automaton

A linear bounded automaton is a multi-track non-deterministic Turing machine with a tape of some bounded finite length.

Length = function (Length of the initial input string, constant c)

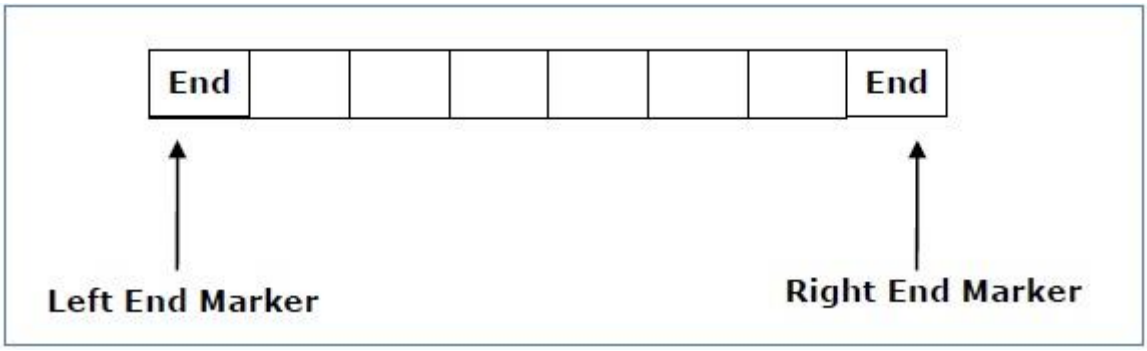
Here,

Memory information ≤ c × Input information

The computation is restricted to the constant bounded area. The input alphabet contains two special symbols which serve as left end markers and right end markers which mean the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape.

A linear bounded automaton can be defined as an 8-tuple (Q, X, ∑, q_0, ML, MR, δ, F) where −

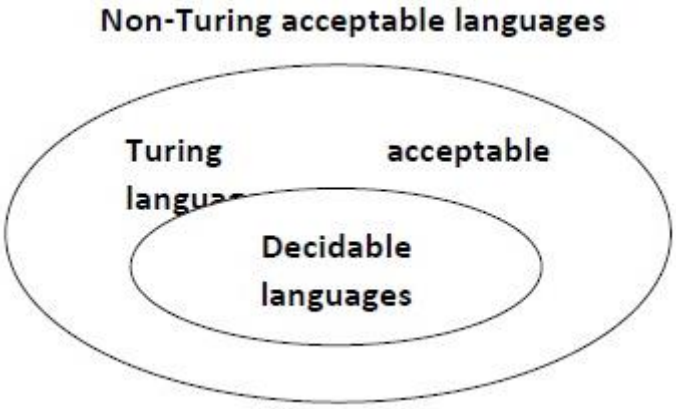
- **Q** is a finite set of states
- **X** is the tape alphabet
- ∑ is the input alphabet
- $\mathbf{q_0}$ is the initial state
- $\mathbf{M_L}$ is the left end marker
- $\mathbf{M_R}$ is the right end marker where $M_R \neq M_L$
- **δ** is a transition function which maps each pair (state, tape symbol) to (state, tape symbol, Constant 'c') where c can be 0 or +1 or -1
- **F** is the set of final states



A deterministic linear bounded automaton is always **context-sensitive** and the linear bounded automaton with empty language is **undecidable.**.

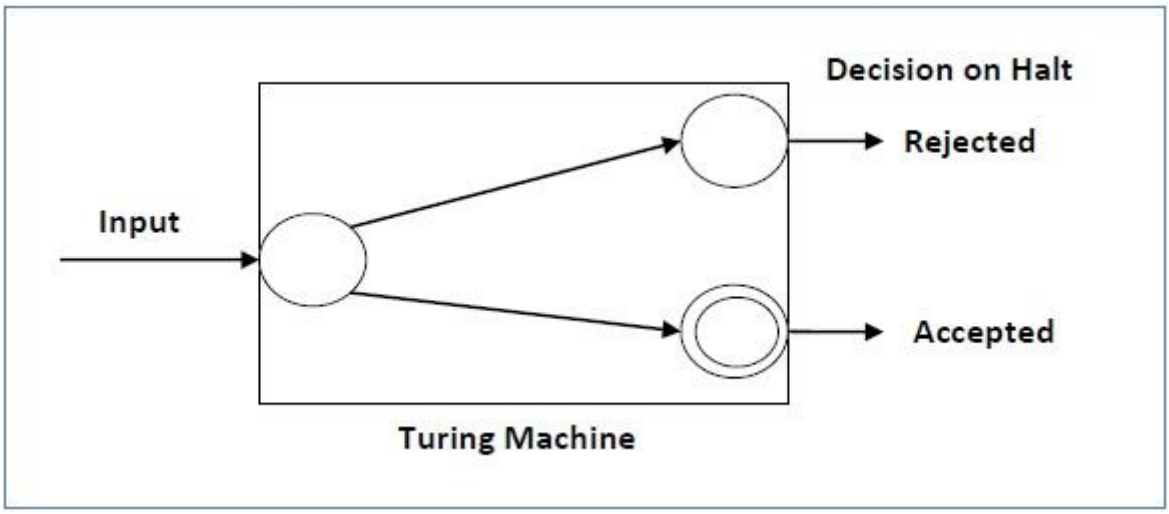
7.6 Decidability of Language

A language is called **Decidable** or **Recursive** if there is a Turing machine which accepts and halts on every input string **w**. Every decidable language is Turing-Acceptable.



A decision problem **P** is decidable if the language **L** of all yes instances to **P** is decidable.

For a decidable language, for each input string, the TM halts either at the accept or the reject state as depicted in the following diagram −



Example 1

Find out whether the following problem is decidable or not −

Is a number ‘m’ prime?

Solution

Prime numbers = {2, 3, 5, 7, 11, 13, ……..}

Divide the number **‘m’** by all the numbers between ‘2’ and ‘√m’ starting from ‘2’.

If any of these numbers produce a remainder zero, then it goes to the “Rejected state”, otherwise it goes to the “Accepted state”. So, here the answer could be made by ‘Yes’ or ‘No’.

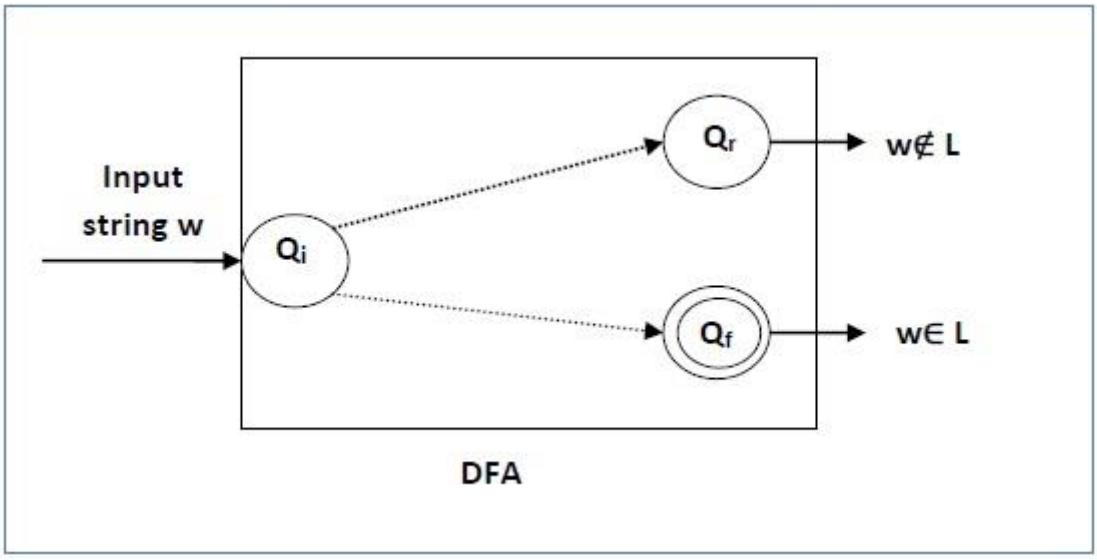
Hence, it is a decidable problem.

Example 2

Given a regular language **L** and string **w**, how can we check if **w ∈ L**?

Solution

Take the DFA that accepts **L** and check if **w** is accepted



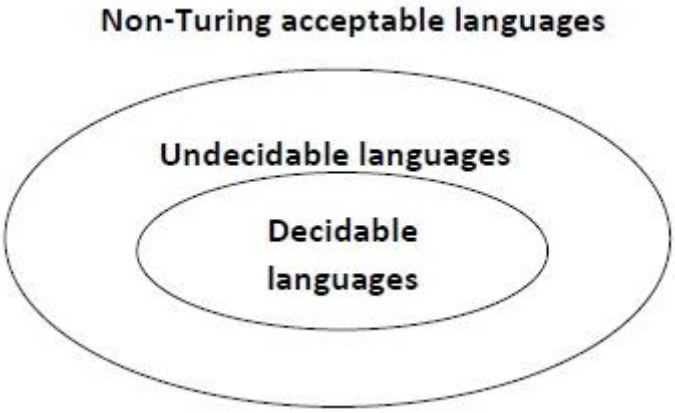
Some more decidable problems are −

- Does DFA accept the empty language?
- Is $L_1 \cap L_2 = \emptyset$ for regular sets?

Note −

- If a language **L** is decidable, then its complement **L'** is also decidable
- If a language is decidable, then there is an enumerator for it.

For an undecidable language, there is no Turing Machine which accepts the language and makes a decision for every input string **w** (TM can make decision for some input string though). A decision problem **P** is called "undecidable" if the language **L** of all yes instances to **P** is not decidable. Undecidable languages are not recursive languages, but sometimes, they may be recursively enumerable languages.



Example

- The halting problem of Turing machine
- The mortality problem
- The mortal matrix problem
- The Post correspondence problem, etc.
-

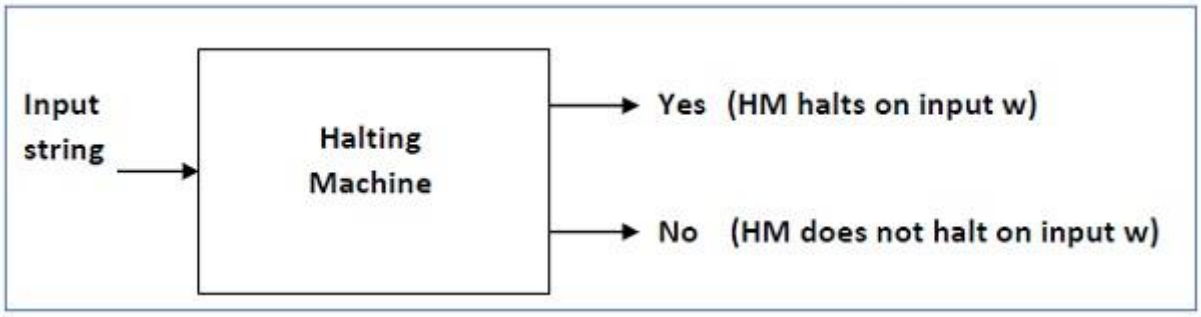
All regular, context-free, context-sensitive and recursive languages are recursively enumerable.

7.7 Halting Problem of Turing Machine

Input − A Turing machine and an input string **w**.

Problem − Does the Turing machine finish computing of the string **w** in a finite number of steps? The answer must be either yes or no.

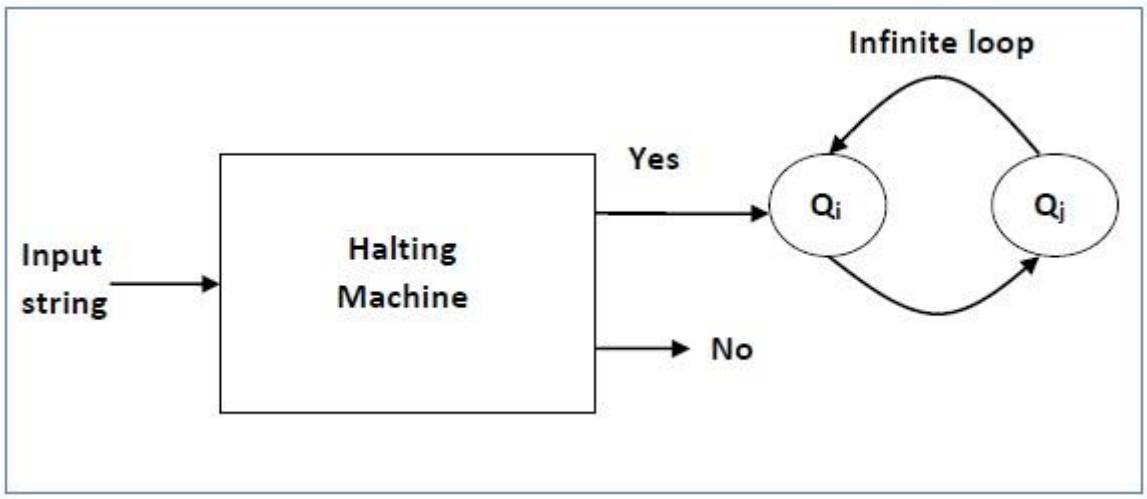
Proof − At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine −



Now we will design an **inverted halting machine (HM)'** as −

- If **H** returns YES, then loop forever.
- If **H** returns NO, then halt.

The following is the block diagram of an 'Inverted halting machine' −



Further, a machine $\mathbf{(HM)_2}$ which input itself is constructed as follows −

- If $(HM)_2$ halts on input, loop forever.
- Else, halt.

Here, we have got a contradiction. Hence, the halting problem is **undecidable**.

7.8 Post Correspondance Problem

The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem. The PCP problem over an alphabet ∑ is stated as follows −

Given the following two lists, **M** and **N** of non-empty strings over ∑ −

$M = (x_1, x_2, x_3, \ldots\ldots, x_n)$

$N = (y_1, y_2, y_3, \ldots\ldots, y_n)$

We can say that there is a Post Correspondence Solution, if for some $i_1, i_2, \ldots\ldots i_k$, where $1 \leq i_j \leq n$, the condition $x_{i1} \ldots\ldots x_{ik} = y_{i1} \ldots\ldots y_{ik}$ satisfies.

Example 1

Find whether the lists

M = (abb, aa, aaa) and N = (bba, aaa, aa)

have a Post Correspondence Solution?

Solution

	x_1	x_2	x_3
M	Abb	aa	aaa
N	Bba	aaa	aa

Here,

$x_2x_1x_3$ = 'aaabbaaa'

and **$y_2y_1y_3$ = 'aaabbaaa'**

We can see that

$x_2x_1x_3 = y_2y_1y_3$

Hence, the solution is **i = 2, j = 1, and k = 3.**

Example 2

Find whether the lists **M = (ab, bab, bbaaa)** and **N = (a, ba, bab)** have a Post Correspondence Solution?

Solution

	x_1	x_2	x_3
M	Ab	bab	bbaaa
N	A	ba	bab

In this case, there is no solution because −

$|x_2x_1x_3| \neq |y_2y_1y_3|$ (Lengths are not same)

Hence, it can be said that this Post Correspondence Problem is **undecidable**.

7.9 Check your Progress

1. Which of the following statement is wrong?

a)Turing Machine can not solve halting problem.

b) Set of recursively enumerable languages is closed under union.

c)A Finite State Machine with 3 stacks is more powerful than Finite State Machine with 2 stacks

d)Context Sensitive grammar can be recognized by a linearly bounded memory machine

2. Which of the following statement is true?

a) All languages can be generated by CFG

b) b) The number of symbols necessary to simulate a Turing Machine(TM) with m symbols and n states is mn.

c) C)Any regular languages has an equivalent CFG.

D)

d) The class of CFG is not closed under union.

3**.** Which of the following statements is (are) correct ?

a) Recursive languages are closed under complementation.

b) B) If a language and its complement are both regular, the language is recursive

c) C Set of recursively enumerable language is closed under union

d) D All of these

4.Turing machine was invented by:

a) Alan Turing b) Tuning man

b) Turing taring d) None of these

5. Turing Machine is more powerful than

a) Finite Automaton b) push down automaton

b) Both (a) and (b) d) None of these

6. In one move the turing machine :

a) May change its state

b) Write a symbol on the cell being scanned

c) Move the head one position left or right

d) All of the above

7.Turing Machine can be Represented using:

a) Transition table

b) Transition diagram

c) Instantaneous description

d) All of these

8.Which of the following is the restricted model of tuting machine

a)Turing Machine with semi infinite tape

b) Maulti stack machines

c) Offline Turing machine

d)Both (a) and (b)

9.Which of the following statement is false?

a) Turing Machine was developed by alan Turing
b) PDA is less powerful than turing machine
c) Both (a) and (b)
d) None of these

10. In Multihead Turing Machine there are

a) More than one heads of the turing machine
b) More than one input tapes of turing machine
c) Similar of basic model of Turing Machine
d) All of these

7.10 Summary

Turing Machine is a Powerful Model proposed by Alan Turing in 1936.Neither finite automata nor PDA can be considered as accurate model of general purpose computer,since they are unable to recognize simple language like L={ 0n1n2n , n>=0}

But Turing Machine is much more accurate model of General purpose computer.It has unlimited and unrestricted memory.There are some problem that can not be solved by turing machine because these problem are beyond the theoretical limits of computation.

1. Recursive Language: A proper superset of context free languages.

7.11 Keywords

- Always recognizable by pushdown automata.
- Also called type 0 languages.
- Recognizable by Turing machines.
- 2. Recursive enumerable Language:

2. A recursively enumerable language: is a formal language for which there exists a Turing machine (or other computable function) that will halt and accept when presented with any string in the language as

input but may either halt and reject or loop forever when presented with a string not in the language. Contrast this to recursive languages, which require that the Turing machine halts in all cases.

3.Halting Problem: The halting problem asks whether a given program P will halt (i.e., complete execution and return a result after a finite number of steps) when run on ...

4. PCP Problem: The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem.

7.12 Self Assessment Test

Q1. When is a Recursively Enumerable language said to be Recursive?

Q2. What is universal turing machine

Q 3. How to prove that the Post Correspondence problem is Undecidable.

Q 4. Explain the different models of Turing machines.

Q 5. Design a Turing machine for the following Reverses the given string {abb}.

Q 6. What is multitape Turing machine? Explain in one move. What are the actions take place in TM?

Q 7.What are the applications of Turing Machine?

Q 8. What are the required fields of an instantaneous description of a Turing machine?

Q 9. Design a Turing machine with no more than three states that accepts the language a(a+b)*. Assume C504.4 BTL 1 10 ∑ = {a,b}

Q 10. Design a TM that accepts the language of odd integers written in binary

Q 11. Construct a Turing machine to compute ‘n mod 2’ where n is represented in the tape in unary form consisting of only 0’s.

7.13 Answer to check your Progress

1. C

2. C

3. D

4. A

5. C

6. D

7. D

8. B

9. C

10. A

7.14 REFERENCES/SUGGESTED READINGS

1.Hopcroaft & O. D. Ullman, R Mothwani, Introduction to automata theory, language & computations, AW, 2001.

2. K. L. P. Mishra & N. Chandrasekaran, . Theory of Computer Sc.(Automata, Languages and computation), PHI, 2000.

3. Peter Linz, Introduction to formal Languages & Automata, Narosa, Publication, 2001.

4. Ramond Greenlaw and H. James Hoover, Fundamentals of the Theory of ComputationPrinciples and Practice, Harcourt India Pvt. Ltd., 1998.

5. H. R. Lewis & C. H. Papaditriou, Elements of theory of Computation, PHC, 1998.

6. John C. Martin, Introduction to Languages and the Theory of Computation, T.M.H., 2003

SUBJECT: Theory of Computation	
COURSE CODE: MCA-35	AUTHOR: RAVIKA GOEL
LESSON NO. 8	
Chomsky hierarchies of grammars	

STRUCTURE

8.0 OBJECTIVE

The main objective of this lesson is to make the students learn about Chomsky Hierarchies of Grammer. What is the relation between Language of Classes. Study the basic Primitive Recursive Function.

8.1 INTRODUCTION

The Turing machine is viewed as a mathematical model of a partial recursive function.we considered automata as the accepting devices. In this chapter we will study automata as the computing machines. The problem of finding out whether a given problem is 'solvable' by automata reduces to the evaluation of functions on the set of natural numbers or a given alphabet by mechanical means. We start with the definition of partial and total functions. A partial function f from X to Y is a rule which assigns to every element of X at most one element of Y. A total function from X to Y is a rule which assigns to every element of X a unique element of Y. For example. if R denotes the set of all real numbers, the rule f from R to itself given by fer) = +J; is a partial function since fer) is not defined as a real number when r is negative. But g(r) = 21' is a total function from R to itself. (Note that all the functions considered in the earlier chapters were total functions.) In this chapter we consider total functions from X k to X, where X = {O, 1, 2, 3, ... } or X = {a, b}*. Throughout this chapter we denote (0, 1, 2, ...) by N and (a, b) by L. (Recall that X k is the set of all k-tuples of elements of X) For example, f(m, 17) = m - 11 defines a partial function from N to itself as f(m, 11) is not defined when m - n < 0; gem, 17) = m + 11 defines a total function from N to itself.

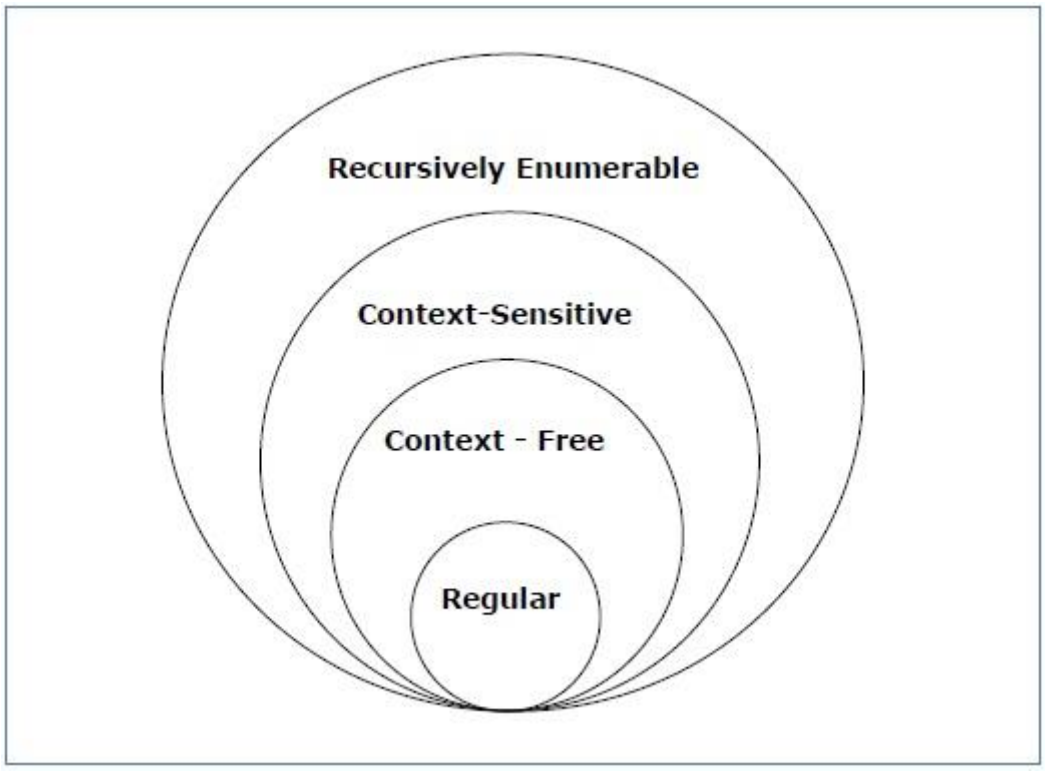
8.2 Chomsky Hierarchies of Grammer

According to Noam Chomosky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other −

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton

Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Take a look at the following illustration. It shows the scope of each type 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 → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

$X \rightarrow \varepsilon$

$X \rightarrow a \mid aY$

$Y \rightarrow b$

Type - 2 Grammar

Type-2 grammars generate context-free languages.

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

where $\mathbf{A \in N}$ (Non terminal)

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

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

Example

$S \rightarrow X\ a$

$X \rightarrow a$

$X \rightarrow aX$

$X \rightarrow abc$

$X \rightarrow \varepsilon$

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

$\boldsymbol{\alpha A \beta \rightarrow \alpha \gamma \beta}$

where $\mathbf{A \in N}$ (Non-terminal)

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

The strings $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ may be empty, but $\boldsymbol{\gamma}$ must be non-empty.

The rule $\mathbf{S \rightarrow \varepsilon}$ 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

AB → AbBc

A → bcA

B → 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 **α → β** 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

S → ACaB

Bc → acB

CB → DB

aD → Db

8.3 Unrestricted Grammer

In automaton, **Unrestricted Grammar** or **Phrase Structure Grammar** is the most general in the **Chomsky Hierarchy of classification**. This is **type0** grammar, generally used to generate **Recursively Enumerable languages**. It is called unrestricted because no other restriction is made on this except each of their left hand sides being non-empty. The left hand sides of the rules can contain terminal and non-terminal, but the condition is at least one of them must be non-terminal.

A **Turning Machine** can simulate **Unrestricted Grammar** and **Unrestricted Grammar** can simulate **Turning Machine** configurations. It can always be found for the language recognized or generated by any **Turning Machine**.

Formal Definition

The unrestricted grammar is 4 tuple -

G = (N,Σ,P,S)

N - A finite set of **non-terminal** symbols or **variables**,

Σ - It is a set of terminal symbols or the alphabet of the language being described, where **N ∩ Σ = φ**,

P - It is a finite set of "**productions**" or "**rules**",

S - It is a **start variable** or **non-terminal** symbol.

If, **α** and **β** are two strings over the alphabet **N ∪ Σ**. Then, the rules or productions are of the form **α →** **β**. The start variable **S** appears on the left side of the rule.

Example of Unrestricted Grammar

Language

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

Grammar

S→aBSc {Equal Number of a's, B's, c's}

S→ ε {Eliminate S}

Ba→aB {Move a's to Right of B's}

Bc→bc {Reduce B before first c to b}

Bb→bb {Reduce all remaining B's to b}

8.4 Context Sensitive Language

Context-Sensitive Grammar –

A Context-sensitive grammar is an Unrestricted grammar in which all the productions are of form –

$$\alpha \rightarrow \beta$$

where $\alpha, \beta \in (V \cup T)^+$ and $|\alpha| \leq |\beta|$

Where α and β are strings of non-terminals and terminals.

Context-sensitive grammars are **more powerful** than context-free grammars because there are some languages that can be described by CSG but not by context-free grammars and CSL are less powerful than Unrestricted grammar. That's why context-sensitive grammars are positioned between context-free and unrestricted grammars in the Chomsky hierarchy.

Context-sensitive grammar has 4-tuples. **G = {N, Σ, P, S}**, Where

N = Set of non-terminal symbols

Σ = Set of terminal symbols

S = Start symbol of the production

P = Finite set of productions

All rules in P are of the form α_1 A α_2 –> α_1 β α_2

Context-sensitive Language: The language that can be defined by context-sensitive grammar is called CSL. Properties of CSL are :

- Union, intersection and concatenation of two context-sensitive languages is context-sensitive.
- Complement of a context-sensitive language is context-sensitive.

Example –

Consider the following CSG.

S → abc/aAbc

Ab → bA

Ac → Bbcc

bB → Bb

aB → aa/aaA

What is the language generated by this grammar?

Solution:

S → aAbc

→ abAc

→ abBbcc

→ aBbbcc

→ aaAbbcc

→ aabAbcc

→ aabbAcc

→ aabbBbccc

→ aabBbbccc

→ aaBbbbccc

→ aaabbbccc

The language generated by this grammar is $\{\mathbf{a^n b^n c^n} \mid n \geq 1\}$.

8.5 Computability and Primitive Recursive function

Computability is the ability to solve a problem in an effective manner.

The concept of a function is fundamental to much of mathematics. As summarized a function is a rule that assigns to an element of one set, called the domain of the function, a unique value in another set, called the range of the function. This is very broad and general and immediately raises the question of how we can explicitly represent this association. There are many ways in which functions can be defined. Some of them we use frequently, while others are less common. We are all familiar with functional notation in which we write expressions like f(n) = n 2 + 1. This defines the function f by means of a recipe for its computation: Given any value for the argument n, multiply that value by itself,

and then add one. Since the function is defined in this explicit way, we can compute its values in a strictly mechanical fashion. To complete the definition of f, we also must specify its domain. If, for example, we take the domain to be the set of all integers, then the range of f will be some subset of the set of positive integers. Since many very complicated functions can be specified this way, we may well ask to what extent the notation is universal. If a function is defined (that is, we know the relation between the elements of its domain and its range), can it be expressed in such a functional form? To answer the question, we must first clarify what the permissible forms are. For this we introduce some basic functions, together with rules for building from them some more complicated ones. Primitive Recursive Functions To keep the discussion simple, we will consider only functions of one or two variables, whose domain is either I, the set of all nonnegative integers, or I × I, and whose range is in I. In this setting, we start with the basic functions: 1. The zero function z(x) = 0, for all x ∈ I. 2. The successor function s(x), whose value is the integer next in sequence to x, that is, in the usual notation, s(x) = x +1. 3. The projector functions pk (x1 , x2) = xk , k = 1, 2. There are two ways of building more complicated functions from these: 1. Composition, by which we construct f (x, y) = h (g1 (x, y), g2 (x, y)) from defined functions g1 ,g2 ,h. 2. Primitive recursion, by which a function can be defined recursively through f (x, 0) = g1 (x), f (x, y + 1) = h (g2 (x, y), f (x, y)), from defined functions g1 , g2 , and h. We illustrate how this works by showing how the basic operations of integer arithmetic can be constructed in this fashion. Example 13.1 Addition of integers x and y can be implemented with the function add (x, y), defined by add (x, 0) = x, add (x, y +1) = add (x, y)+1. To add 2 and 3, we apply these rules successively: add (3, 2) = add (3,1) + 1 = (add (3,0) + 1) + 1 = (3+1) + 1 = 4 + 1 = 5. we can now define multiplication by mult (x, 0) = 0, mult (x, y + 1) = add (x, mult (x, y)). Formally, the second step is an application of primitive recursion, in which h is identified with the add function, and g2 (x, y) is the projector function p1 (x, y). Substraction is not quite so obvious. First, we must define it, taking into account that negative numbers are not permitted in our system. A kind of subtraction is defined from usual subtraction by x y = x – y if x ≥ y, x y = 0 if x < y. The operator is sometimes called the monus; it defines subtraction so that its range is I. Now we define the predecessor function pred (0) = 0, pred (y +1) = y, and from it, the subtracting function subtr (x, 0) = x, subtr (x, y +1) = pred (subtr (x, y)). To prove that 5 – 3 = 2, we reduce the proposition by applying the definitions a number of times: In much the same way, we can define integer division, but we will leave the demonstration of it as an exercise. If we accept this as given, we see that the basic arithmetic operations are all constructible by

the elementary processes described. With the algebraic operations precisely defined, other more complicated ones can now be constructed, and very complex computations built from the simple ones. We call functions that can be constructed in such a manner primitive recursive. A function is called primitive recursive if and only if it can be constructed from the basic functions z, s, pk , by successive composition and primitive recursion. Note that if g1 , g2 , and h are total functions, then f defined by composition and primitive recursion is also a total function. It follows from this that every primitive recursive function is a total function on I or I ×I. The expressive power of primitive recursive functions is considerable, and most common functions are primitive recursive. However, not all functions are in this class, as the following argument shows. Theorem 8.1 Let F denote the set of all functions from I to I. Then there is some function in F that is not primitive recursive. Proof: Every primitive recursive function can be described by a finite string that indicates how it is defined. Such strings can be encoded and arranged in standard order. Therefore, the set of all primitive recursive functions is countable. Suppose now that the set of all functions is also countable. We can then write all functions in some order, say, f1 ,f2 ,…. We next construct a function g defined as g(i) = f i (i)+ 1, i = 1,2,…. Clearly, g is well defined and is therefore in F, but equally clearly, g differs from every f i in the diagonal position. This contradiction proves that F cannot be countable. Combining these two observations proves that there must be some function in F that is not primitive recursive. Actually, this goes even further; not only are there functions that are not primitive recursive, there are in fact computable functions that are not primitive recursive. Theorem 8.2 Let C be the set of all total computable functions from I to I. Then there is some function in C that is not primitive recursive. Proof: By the argument of the previous theorem, the set of all primitive recursive functions is countable. Let us denote the functions in this set as r1 ,r2 ,…and define a function g by g(i) = ri (i)+1 By construction, the function g differs from every ri and is therefore not primitive recursive. But clearly g is computable, proving the theorem. The nonconstructive proof that there are computable functions that are not primitive recursive is a fairly simple exercise in diagonalization. The actual construction of an example of such a function is a much more complicated matter. We will give here one example that looks quite simple; however, the demonstration that it is not primitive recursive is quite lengthy. Ackermann's Function Ackermann's function is a function from I × I to I, defined by A (0,y) = y +1, A (x, 0) = A (x – 1,1), A (x, y +1) = A (x – 1, A (x, y)). It is not hard to see that A is a total, computable function. In fact, it is quite elementary to write a recursive computer program for its computation. But in spite of its apparent simplicity,

Ackermann's function is not primitive recursive. Of course, we cannot argue directly from the definition of A. Even though this definition is not in the form required for a primitive recursive function, it is possible that an appropriate alternative definition could exist. The situation here is similar to the one we encountered when we tried to prove that a language was not regular or not context-free. We need to appeal to some general property of the class of all primitive recursive functions and show that Ackermann's function violates this property. For primitive recursive functions, one such property is the growth rate. There is a limit to how fast a primitive recursive function f(n) can grow as $n \rightarrow \infty$, and Ackermann's function violates this limit. That Ackermann's function grows very rapidly is easily demonstrated; see, for example, Exercises 9 to 11 at the end of this section. How this is related to the limit of growth for primitive recursive functions is made precise in the following theorem. Its proof, which is tedious and technical, will be omitted. Theorem 8.3 Let f be any primitive recursive function. Then there exists some integer n such that f(i) < A (n,i), for all i = n, n +1,…. Proof: For the details of the argument, see Denning, Dennis, and Qualitz (1978, p. 534). If we accept this result, it follows easily that Ackermann's function is not primitive recursive. Theorem 13.4 Ackermann's function is not primitive recursive. Proof: Consider the function g(i) = A (i, i). If A were primitive recursive, then so would g. But then, according to Theorem13.3, there exists an n such that g(i) < A (n, i), for all i. If we now pick i = n, we get the contradiction g (n) = A(n, n) < A(n, n), proving that A cannot be primitive recursive. μ Recursive Functions To extend the idea of recursive functions to cover Ackermann's function and other computable functions, we must add something to the rules by which such functions can be constructed. One way is to introduce the μ or minimalization operator, defined by μy (g (x, y)) = smallest y such that g (x, y) = 0. In this definition, we assume that g is a total function. Example 8.4 Let g (x,y) = x + y 3, which is a total function. If $x \leq 3$, then y = 3 – x is the result of the minimalization, but if x > 3, then there is no $y \in I$ such that x + y – 3 = 0. Therefore, μy(g (x, y)) = 3 – x, for $x \leq 3$, = undefined, for x > 3. We see from this that even though g (x,y) is a total function, μy(g (x,y)) may only be partial. As the previous example shows, the minimalization operation opens the possibility of defining partial functions recursively. But it turns out that it also extends the power to define total functions so as to include all computable functions. Again, we merely quote the major result with references to the literature where the details may be found. A function is said to be μ-recursive if it can be constructed from the basis functions by a sequence of applications of the μ-operator and the

operations of composition and primitive recursion. A function is μ-recursive if and only if it is computable.

8.6 CHECK YOUR PROGRESS

1. The production Grammar is {S->aSbb,S->abb} is

a) type-3 grammar b)type-2 grammar

c) type-1 grammar d) type-0 grammar

2. Which of the following statement is wrong?

a) Turing Machine can not solve halting problem.

b)Set of recursively enumerable languages is closed under union.

c)A Finite State Machine with 3 stacks is more powerful than Finite State Machine with 2 stacks

d) Context Sensitive grammar can be recognized by a linearly bounded memory machine.

3. Consider a grammar : G = ({ x , y) , { s , x , y } , p , s) where elements of parse : S--> x y S -->y x x--> x z x--> x y--> y z-->Z

a) Chomsky type 0 b) Chomsky type 1

c)Chomsky type 2 d) Chomsky type 3

4. What is the highest type number which can be applied to the following grammar ? S —> Aa, A —> Ba, B —> abc

a) Type 0 b) Type 1

c) Type 2 d)Type 3

5. Unrestricted Grammer is also known as

a) Type 0 b) semi-thue Grammer

b) Phase Structure Grammer d) all of these

6. Which of the following is more powerful?

a) PDA b) turing Machine

b) Finite automaton d) Context Sensitive Language

7. A Context sensitive Language is accepted by

a) Finite automaton b) Linear bounded automaton

b) Both (a) and (b) d) None of these

8.Which of the following statement is wrong?

a) Chomsky hierarchy originally define onlt two grammer

b) Type 0 grammer is called unrestricted grammer

c) Type 0 is recognized by turing Machine

d) All of these

8.7 SUMMARY

We shall discuss the class of primitive recursive functions-a subclass of partial recursive functions. The Turing machine is viewed as a mathematical model of a partial recursive function. we considered automata as the accepting devices. In this chapter we will study automata as the computing machines. The problem of finding out whether a given problem is 'solvable' by automata reduces to the evaluation of functions on the set of natural numbers or a given alphabet by mechanical means. We start with the definition of partial and total functions. A partial function f from X to Y is a rule which assigns to every element of X at most one element of Y. A total function from X to Y is a rule which assigns to every element of X a unique element of Y. For example. if R denotes the set of all real numbers, the rule f from R to itself given by f(r) = +root of r is a partial function since fer) is not defined as a real number when r is negative. But g(r) = 2 r is a total function from R to itself. (Note that all the functions considered in the earlier chapters were total functions.

8.8 KEYWORDS

1 Primitive Recursive Function: In computability theory, a primitive recursive function is roughly speaking **a function that can be computed by a computer program whose loops are all "for" loops** (that is, an upper bound of the number of iterations of every loop can be determined before entering the loop).

2 Computability: is **the ability to solve a problem in an effective manner.** It is a key topic of the field of computability theory within mathematical logic and the theory of computation within computer science.

8.9 SELF ASSESSMENT TEST

Q1. Describe the Chomsky hierarchy of languages. ?

Q2. Define Unrestricted Grammer?

Q 3. Explain Recursive , Total and partial Recursive Function?

Q 4. Explain Type 0,1,2,3 Grammer?

Q 5. Write a short note on Context Sensitive Grammer?

8.10 ANSWER TO CHECK YOUR PROGRESS

1.B

2.C

3.D

4.C

5.D

6.B

7.B

8.A

8.11 REFERENCES/SUGGESTED READINGS

1 Hopcroaft & O. D. Ullman, R Mothwani, Introduction to automata theory, language & computations, AW, 2001.

2 K. L. P. Mishra & N. Chandrasekaran, . Theory of Computer Sc.(Automata, Languages and computation), PHI, 2000.

3 Peter Linz, Introduction to formal Languages & Automata, Narosa, Publication, 2001.

4 Ramond Greenlaw and H. James Hoover, Fundamentals of the Theory of ComputationPrinciples and Practice, Harcourt India Pvt. Ltd., 1998.

5 H. R. Lewis & C. H. Papaditriou, Elements of theory of Computation, PHC, 1998.

6 John C. Martin, Introduction to Languages and the Theory of Computation, T.M.H., 2003

NOTES

NOTES

NOTES