17BECS405	Formal Languages and Automata	L	T	P	C
		3	0	0	3

COURSE OBJECTIVES:

- To have an introductory knowledge of automata, formal language theory
- and computability.
- To have an understanding of finite state and pushdown automata.
- To have a knowledge of regular languages and context free languages.
- To know the relation between regular language, context free language and corresponding recognizers.
- To study the Turing machine and classes of problems

LEARNING OUTCOMES:

- To have a introductory knowledge of automata, formal language theory and
- computability.
- To have an understanding of finite state and pushdown automata.
- To have a knowledge of regular languages and context free languages.
- To know the relation between regular language, context free language and
- corresponding recognizers.
- To study the Turing machine and classes of problems.

UNIT- I Introduction To Automata (9)

Basics of String and Alphabets - Finite Automata (FA) – Deterministic Finite Automata (DFA)– Non-deterministic Finite Automata (NFA) – Conversion of NFA to DFA- Finite Automata with Epsilon transition-Equivalence and Minimization of Automata

UNIT- II Regular Expressions And Languages (9)

Regular Expression – FA and Regular Expressions – Proving languages not to be regular – Pumping lemma for regular sets - Closure properties of regular languages- Decision Properties of Regular Languages

UNIT- III Context-Free Grammar And Languages (9)

Context-Free Grammar (CFG) – Parse Trees – Ambiguity in grammars and languages – Definition of the Pushdown automata – Languages of a Pushdown Automata – Equivalence of Pushdown automata and CFG, Deterministic Pushdown Automata- Pumping Lemma for CFL - Closure Properties of CFL- Context Sensitive Grammar (CSG) & Languages

UNIT IV Properties of Context Free Grammar
(9)

Normal forms for Context Free Grammar- Chomsky Normal Form- The Pumping lemma for Context free Languages- Closure properties of Context Free Languages-Inverse Homomorphism-Decision Properties of CFL

UNIT- V Turing Machine
(9)

Turing Machines – Introduction- Definition – Turing machine construction- Storage in Finite control-Multiple tracks- Subroutines-Checking of Symbols – Two way infinite tape-Undecidability .

Total Hours: 45

TEXT BOOKS:

1. Hopcroft J.E, R.Motwani and J.D.Ullman, Introduction to Automata Theory, Languages and Computations, Pearson Education, 2011.

REFERENCES:

1. Lewis H.R and C.H.Papadimitriou, Elements of The theory of Computation, Pearson Education, PHI, 2009.
2. Martin J, Introduction to Languages and the Theory of Computation, TMH, 2010
3. Micheal Sipser, Introduction of the Theory and Computation, Edition,Thomson Brokecole,2012.
4. An Introduction to Formal Languages and Automata, 5th Edition, Peter Linz, 2011

WEBSITES:

1. http://www.regular-expressions.info/tutorial.html
2. http://www.cs.duke.edu/csed/jflap/tutorial/fa/nfa2dfa/index.html
3. http://web.cecs.pdx.edu/~harry/compilers/slides/LexicalPart3.pdf

KARPAGAM ACADEMY OF HIGHER EDUCATION

Faculty of Engineering

Department of Computer Science and Engineering

Lecture Plan

Subject Name: FORMAL LANGUAGES AND AUTOMATA Subject Code: 17BECS405

S.No	Topic Name	No.of Periods	Supporting Materials	Teaching Aids
	UNIT- I Introduction To Automata			
1	Basics of String and Alphabets	1	R[1]-1	BB
2	Finite Automata (FA)	1	R[1]-1	BB
3	Deterministic Finite Automata (DFA)	1	R[1]-5	PPT
4	Non-deterministic Finite Automata (NFA)	1	R[1]-6	PPT
5	Non-deterministic Finite Automata (NFA)	1	R[1]-6	PPT
6	Conversion of NFA to DFA	1	T[1]-95	PPT
7	Conversion of NFA to DFA	1	T[1]-95	PPT
8	Finite Automata with Epsilon transition	1	T[1]-68	BB
9	Finite Automata with Epsilon transition	1	Web	PPT
10	Equivalence and Minimization of Automata	1	T[1]-12	BB
11	Tutorial: Basic Applications	1	Web	BB
Total		**11**		
	UNIT- Regular Expressions and Languages			
12	Regular Expression	1	T[1]-200	PPT
13	Finite Automata (FA)	1	web	PPT
14	FA and Regular Expressions	1	T[1] 201	BB
15	Proving languages not to be regular	1	T[1]214	PPT
16	Pumping lemma for regular sets	1	T[1]214	PPT
17	Closure properties of regular languages	1	T[1]218	PPT
18	Closure properties of regular languages	1	R[1]218	PPT
19	Decision properties of regular langauages	1	R[1]218	PPT
20	Decision properties of regular langauages	1	R[1]221	BB
21	Revision	1	R[1]221	PPT
Total		**10**		
	UNIT- III Context-Free Grammar And Languages			
22	Context-Free Grammar (CFG)	1	web	PPT

23	Parse Trees	1	web	PPT
24	Ambiguity in grammars and languages	1	web	PPT
25	Definition of the Pushdown automata	1	T[1]-488	BB
26	Languages of a Pushdown Automata	1	T[1]-193	PPT
27	Equivalence of Pushdown automata and CFG	1	T[1]-266	BB
28	Deterministic Pushdown Automata	1	T[1]-305	PPT
29	Pumping Lemma for CFL	1	T[1]-343	BB
30	Closure Properties of CFL	1	web	PPT
31	Context Sensitive Grammar (CSG) & Languages	1	web	PPT
Total		**10**		
	UNIT- IV Properties of Context Free Grammar			
32	Properties of Context Free Grammar	1	R[1]-139	PPT
33	Normal forms for Context Free Grammar	1	R[1]-139	PPT
34	Chomsky Normal Form	1	T[1]-140	PPT
35	The Pumping lemma	1	R[1]-152	BB
36	The Pumping lemma for Context free Languages	1	R[1]-159	PPT
37	Closure properties of Context Free Languages	1	R[1]-162	BB
38	Inverse Homomorphism	1	R[1]-163	PPT
39	Decision Properties of CFL	1	R[1]-133	PPT
40	Properties of CFL	1	web	PPT
41	Revision	1	R[1]-133	BB
Total		**10**		
	UNIT- V Turing Machine			
42	Turing Machines	1	R[1]-248	PPT
43	Introduction to TM	1	R[1]-465	BB
44	Definition – Turing machine construction	1	R[1]-465	BB
45	Storage in Finite control	1	R[1]-255	PPT
46	Multiple tracks	1	R[1]-248	PPT
47	Subroutines	1	T[1]-1087	PPT
48	Checking of Symbols	1	T[1]-1087	PPT
49	Two way infinite tape	1	T[1]-690	BB
50	Undecidability Problem	1	T[1]-690	PPT
51	Revision	1	T[1]-752	BB
52	**Discussion on Previous University Question Papers**			
Total		**10**		
	Total Hours	**52**		

TEXT BOOKS

S.NO	Title of the book			Year of publication
1	Hopcroft J.E, R.Motwani and J.D.Ullman, Introduction to Automata Theory, Languages and Computations, Pearson Education			2011

REFERNCE BOOKS

S.NO	Title of the book			Year of publication
1	Lewis H.R and C.H.Papadimitriou, Elements of The theory of Computation, Pearson Education, PHI, 2009.			2009

WEBSITES

1. http://www.regular-expressions.info/tutorial.html
2. http://www.cs.duke.edu/csed/jflap/tutorial/fa/nfa2dfa/index.html
3. http://web.cecs.pdx.edu/~harry/compilers/slides/LexicalPart3.pdf

KARPAGAM UNIVERSITY

Karpagam Academy of Higher Education

(Established under Section 3 of UGC Act 1956)

Eachanari, Coimbatore-641 021. TAMILNADU

Faculty of Engineering

Department of Computer Science and Engineering

13BECS505

THEORY OF COMPUTATION

LECTURE NOTES

Prepared by

K.Sundareswari,AP

SYLLABUS

UNIT- I AUTOMATA

Introduction to formal proof – Additional forms of proof – Inductive proofs –Finite Automata (FA) – Deterministic Finite Automata (DFA)– Non-deterministic Finite Automata (NFA) – Finite Automata with Epsilon transitions.

UNIT- II REGULAR EXPRESSIONS AND LANGUAGES

Regular Expression – FA and Regular Expressions – Proving languages not to be regular – Closure properties of regular languages – Equivalence and minimization of Automata.

UNIT- III CONTEXT-FREE GRAMMAR AND LANGUAGES

Context-Free Grammar (CFG) – Parse Trees – Ambiguity in grammars and languages – Definition of the Pushdown automata – Languages of a Pushdown Automata – Equivalence of Pushdown automata and CFG, Deterministic Pushdown Automata.

UNIT- IV PROPERTIES OF CONTEXT-FREE LANGUAGES

Normal forms for CFG – Pumping Lemma for CFL - Closure Properties of CFL – Turing Machines – Programming Techniques for TM.

UNIT- V UNDECIDABILITY

A language that is not Recursively Enumerable (RE) – An undecidable problem that is RE – Undecidable problems about Turing Machine –Post's Correspondence Problem-The classes P and NP.

UNIT- I AUTOMATA

Introduction to formal proof – Additional forms of proof – Inductive proofs –Finite Automata (FA) – Deterministic Finite Automata (DFA)– Non-deterministic Finite Automata (NFA) – Finite Automata with Epsilon transitions.

Theory of computations is based on mathematical computations. These computations are used to represent various mathematical models. In this subject we will study many interesting models such as finite automata, push down automata, turning machines. This subject is a fundamental subject, that is very close to the subjects like compilers, operating system, sytem software and pattern recognition sytem. The automata theory is the base of this subject. The automata theory is theory of models. Working of every process can be represented by means of model. The model can be a theoretical or mathematical model. The model helps in representing the concpet of every activity. The logic or behaviour of such models can be well understood with the help of proofs. In this chapter we will learn the basic types of mathematical proofs. We will learn how automata are important in theory of computation. Hence we will study concept of finite automata.

1.1 Introduction to formal Proofs

The formal proof can be using deductive proof and inductive proof. The deductive proof consists of sequence of statements given with logical reasoning in order to prove the first or initial statement. The initial statement is called hypothesis.

The inductive proof is a recursive kind of proof which consists of sequence of parameterized statements that use the statement itself with lower values of its parameter.

In short, formal proofs are the proofs in which we try to prove that statement B is true because statement A is true. The statement A is called hypothesis and B is called conclusion statement. In other words, "if A then B" we say that B is deduced from A.

Let us see some additional forms of proofs.

1.2 Additional forms of Proof

We will discuss additional forms of proofs with the help of some examples. We will discuss

1. Proofs about sets
2. Proofs by contradiction
3. Proofs by counter example

1.2.1 A Proof about Sets

The set is a collection of elements or items. By giving proofs about the sets we try to prove certain properties of the sets.

For example if there are two expressions A and B and we want to prove that both the expressions A and B are equivalent then we need to show that the set represented by expression A is same as the set represented by expression B. Let PUQ = Q U R if we map expression A with PUQ and expression B with QUR then to prove A = B we need to prove PUQ = QUP.

This proof is of the kind "if and only if" that means an element x is in A if and only if it is in B. We will make use of sequence of statements along with logical justification in order to prove this equivalence.

Sr. No.	Statement	Justification
1.	x is in PUQ	Given
2.	x is in P or x is in Q	1) And by definition of union
3.	x is in Q or x is in P	2) And by definition of union
4.	x is in QUP	3) (2) And by definition of union

Table 1.1

Sr. No.	Statement	Justification
1.	x is in QUP	Given
2.	x is in Q or x is in P	1) And by definition of union
3.	x is in P or x is in Q	2) And by definition of union
4.	x is in PUQ	3) (2) by And by definition of union

Table 1.2

Hence PUQ = QUP. Thus A = B is true as element x is in B if and only if x is in A.

1.2.2 Proof by Contradiction

In this type of proof, for the statement of the form if A then B we start with statement A is not true and thus by assuming false A we try to get the conclusion of statement B. When it becomes impossible to reach to statement B we contradict our self and accept that A is true.

For example –

Prove PUQ = QUP.

Proof. Assume PUQ ≠ QUP. Now consider x is in P or x is in Q that means x is in PUQ (by definition of union), in other words x is in Q or a is in P which can also be written as x is in QUP (by definition of union). This contradicts our assumption PUQ ≠ QUP. hence the assumption which we made initially is false. And therefore PUQ = QUP is proved.

1.2.3 Proof by Counter Example

In order to prove certain statements, we need to see all possible conditions in which that statement remains true. There are some situations in which the statement can not be true. For example

There is no such pair of integers such that a mod b = b mod a

Proof – Here the possibilities are either a > b or a < b. For example if a = 2 and b = 3 then 2 mod 3 ≠ 3 mod 2.

Similarly, if a = 3 and b = 2 then also 3 mod 2 ≠ 2 mod 3. Thus the given statement is true for any pair of integers but if a = b then naturally, a mod b = b mod a. Thus we need to change the statement of theorem slightly and it will be a mod b = b mod a is true only when a = b.

This type of proof is called counter example. Such proof are true only at some specific conditions.

1.3 Inductive Proofs

Inductive proofs are special type of proofs used to prove recursively defined objects.

The inductive proofs can be carried out using two steps

1. Basis of induction – In this step we start with the lowest possible value. For example to show – S(i) we will consider i = 0 or i = 1
2. Inductive step – In this step we try to show S(n) is since S(n + 1) is true. Let us discuss some example to understand the inductive proof.

Example 1.1

Prove

$1 + 2 + 3 + \ldots + n = n\,(n + 1)/2$ using method of induction

Proof :

Consider the two step approach for a proof by method of induction

1. Basis of induction – Let n = 1 then LHS = 1 and RHS = 1 + 1/2 = 1. Hence LHS = RHS.
2. Induction hypothesis – To prove $1 + 2 + 3 \ldots + n = n\,(n + 1)\,/\,2$ consider $n = n + 1$

then $1 + 2 + 3 + \ldots + n + (n + 1) = n\,(n + 1)/2 + (n + 1)$

$= n^2 + 3n + 2/2$

$= (n + 1)\,(n + 2)/2$

Thus it is proved that $1 + 2 + 3 \ldots + n = n\,(n + 1)$

Example 1.2

Prove $n! > = 2^{n-1}$

Proof : Consider

1. Basis of induction – Let n = 1 then LHS = 1 RHS = $2^{1-1} = 2^0 = 1$ hence $n! > = 2^{n-1}$ is proved.
2. Induction hypothesis. Let n = n + 1 then

$k! = 2^{k-1}$ where $k > = 1$

then $(k + 1)! = (k + 1)k!$ by definition of n!

$= (k + 1)\,2^{k-1}$

$= 2 \times 2^{k-1}$

$= 2^k$

1.4 Basic Concepts of Automata Theory

1.4.1 Set

Set is defined as a collection of various objects. These objects are called the elements of the set.

For example : A set of vowels. This set has the elements such as *a*, e, i, o, u.

The set is denoted by capital letter. Let us see how the set of vowels be represented,

$$A = \{ a, e, i, o, u \}$$

The elements of the set are grouped into the curly brackets and separated by commas. The set can be finite or infinite.

Finite set : A finite set is a set of finite number of elements. As you have seen set of vowels is a finite set of elements.

Infinite set : The infinite set is a collection of all the elements which are infinite in number.

For example : Set of natural numbers.

If we want to describe any element of a set we can write it as,

$$\{ a \mid a \text{ is always even number} \}$$

This can be read as the element a of set A such that A (a) i.e. element a is always even number.

If we want to show that b is any element in set A then we can denote it as $b \in A$ i.e. b belongs to A.

Subset : The subset A is called subset of set B if every element of set A is present in set B. But reverse is not true. It is denoted as $A \subseteq B$.

Empty set : The set having no elements in it is called empty set. It is denoted by $A = \{ \}$, the empty set is written as ϕ.

Null string : The null element is denoted by ε or ^ character. Null element means no value character. Remember $\varepsilon \neq \phi$.

Power set : The power set is a set of all the subsets of its elements.

For example : $A = \{ 1, 2, 3 \}$

Then the power set $Q = \{\phi, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\},\}$

The number of elements are always equal to 2^n where n is number of elements in original set. As in set A there are 3 elements, therefore in powerset Q there are $2^3 = 8$ elements.

1.4.2 Operations on Set

On the sets following operations are possible.

i) $A \cup B$ is union operation – If

$$A = \{ 1, 2, 3 \} \quad B = \{ 1, 2, 4 \} \quad \text{then}$$

$$A \cup B = \{ 1, 2, 3, 4 \} \text{ i.e. combination of both the sets.}$$

ii) $A \cap B$ is intersection operation – If

$A = \{ 1, 2, 3 \}$ and $B = \{ 2, 3, 4 \}$ then

$A \cap B = \{ 2, 3 \}$ i.e. collection of common elements from both the sets.

iii) $A - B$ is the difference operation – If

$A = \{ 1, 2, 3 \}$, and $B = \{ 2, 3, 4 \}$ then

$A - B = \{ 1 \}$ i.e. elements which are there in set A but not in set B.

1.4.3 Cardinality of Sets

The cardinality of the set is nothing but the number of members in the set.

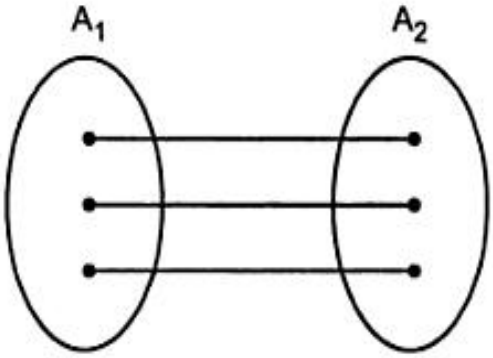


Fig. 1.1 Cardinality of two sets

These sets A_1 and A_2 have the same cardinality as there is one to one mapping of the elements of A_1 and A_2.

1.4.4 Relations

Relationship is a major aspect between two objects, even this is true in our real life. One object can be related with the other object by a 'mother of' relation. Then those two objects form a pair based on this certain relationship.

Definition : The relation R is a collection for the set S which represents the pair of elements.

For example : (*a*, b) is in R. We can represent their relation as a R b. The first component of each pair is chosen from a set called domain and second component of each pair is chosen from a set called range.

Properties of Relations

A relation R on set S is

1. Reflexive if iRi for all i in S.
2. Irreflexive if iRi is false for all i in S.
3. Transitive if iRj and jRk imply iRk.
4. Symmetric if iRj implies jRi.
5. Asymmetric if iRj implies that jRi is false.

Every asymmetric relation must be irreflexive.

For example : if A = { *a*, b } then

reflexive relation R can be = { (*a*, *a*), (b, b) }

irreflexive relation R can be = { (*a*, b) }

transitive relation R can be = { (*a*, b), (b, *a*), (*a*, *a*) }

symmetric relation R can be = { (*a*, b), (b, *a*) }

asymmetric relation R can be = { (*a*, b) }

Equivalent Relation

A relation is said to be equivalence relation if it is reflexive, symmetric and ransitive, over some set S.

Suppose R is a set of relations and S is the set of elements.

For example : S is the set of lines in a plane and R is the relation of lines ntersecting to each other.

Example 1.3 : *Determine whether R is equivalence relation or not where*

$$A = \{ 0, 1, 2 \}, \quad R = \{ (0, 0), (1, 0), (1, 1), (2, 2), (2, 1) \}$$

Solution : The R is reflexive because (0, 0), (1, 1), (2, 2) $\in$ R.

Where as R is not symmetric because (0, 1) is not in R whereas (1, 0) is in R.

Hence the R is not a equivalence relation.

Closures of Relations

Sometimes when the relation R is given, it may not be reflexive or transitive.

By adding some pairs we make the relation as reflexive or transitive.

For example : Let { (*a*, b), (b, c), (*a*, *a*), (b, b) }

Now this relation is not transitive because (*a*, b), (b, c) is there in relation R but (*a*, c) is not there in R so we can make the transitive closure as

{ (*a*, *a*), (b, b), (*a*, b), (b, c), (*a*, c) }

We can even define reflexive closure and symmetric closure in the same way.

1.4.5 Graphs

A graph, denoted G = (V, E) consists of finite set of vertices (or nodes) V and set of edges, the edges are nothing but pair of vertices.

For example :

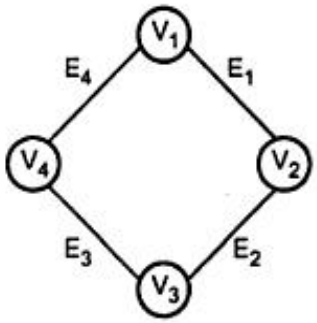


Fig. 1.2

Here the E1 is a edge connecting the vertices V1 and V2.

The set V = {V1, V2, V3, V4 } and E = { E1, E2, E3, E4 }

1.4.6 Directed Graph

The directed graph is also a collection of vertices and edges where the directions are mentioned along the edges.

For example :

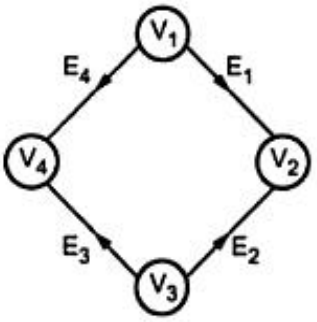


Fig. 1.3

The edge E1 shows the direction to V2 vertex from vertex V1.

We will see the directed graphs in further chapters and we will call those graphs as transition graphs.

1.4.7 Trees

Trees is a collection of vertices and edges with following properties.

1. There is one vertex, called root which has no predecessors.
2. From this root node all the successors are ordered.

For example :

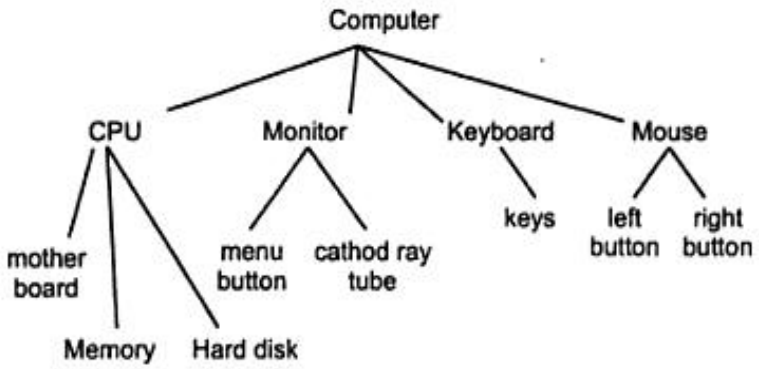


Fig. 1.4

In the above example, at the leaf node all the components of computer are given, when motherboard, memory and harddisk these components are configured they are meant for CPU always. Keys are the prominent component for keyboard. When CPU, monitor, keyboard and mouse are taken as one set, then it constitutes a device called computer.

Computer is a root node in the above figure. CPU, Monitor, keyboard, mouse are the interior nodes. Motherboard, memory harddisk are the leaf nodes for the CPU. Monitor is a parent node or father of Menu button and cathode ray tube whereas menu button is a left child of monitor and cathode ray tube is a right child of monitor.

The only difference between graphs and trees is that graphs do not have special node called root node.

1.4.8 Alphabets, String and Languages

Alphabet is a finite set of symbols we can not define symbol formally.

We can give supporting example for definition of alphabets.

For example : S = { *a*, b, c, ... z }

The elements *a*, b, c, ... z are the alphabets or S = { 0, 1 }

Here 0 and 1 are alphabets.

Strings is a collection of alphabets.

Theory of computer science is based on a basic unit recognized as language. Basically language is a collection of alphabets - the alphabets which form the strings or words. Everybody of us, develop our own ideas when we think in our language. Thus language becomes a fundamental aspect for building any idea or a thought.

In theory of computations there are two types of languages regular language and irregular language. Let us define the language.

2.1 Definition of Language

The language can be defined as a collection of strings or words over the certain input set.

For example : Rama is a boy.

This above statement indicates certain language and the input set is L = {A, B, C, ... Z, *a*, b, c ... z}. The words 'Rama', 'boy', is '*a*' are collected together in certain manner to form a sentence of the language. Many such statements help us to speak a language. In this chapter we will discuss a mathematical model called finite state machine.

2.2 Finite State System

The finite state system represents a mathematical model of a system with certain input. The model finally gives certain output. The input when is given to the machine it is processed by various states, these states are called as intermediate states.

The very good example of finite state system is a control mechanism of elevator. This mechanism only remembers the current floor number pressed, it does not remember all the previously pressed numbers.

The finite state system is a very good design tool for the programs such as text editors and lexical analysers (which is used in compilers). The lexical analyzer is a program which scans your program character by character and recognizes those words as tokens.

Let us take some number 010 which is equivalent to 2. We will scan this number from MSB to LSB.

$$\begin{array}{ccc} 0 & 1 & 0 \\ \uparrow & \uparrow & \uparrow \\ S_0 & S_1 & S_2 \end{array}$$

Then we will reach to state S_2 which is remainder 2 state. Similarly for input 1001 which is equivalent to 9 we will be in final state after scanning the complete input.

$$\begin{array}{cccc} 1 & 0 & 0 & 1 \\ \uparrow & \uparrow & \uparrow & \uparrow \\ S_1 & S_2 & S_1 & \boxed{S_0} \end{array}$$

Thus the number is really divisible by 3.

Example 2.8 : *Design FA which accepts even number of 0's and even number of 1's.*

Solution : This FA will consider four different stages for input 0 and 1. The stages could be

even number of 0 and even number of 1,

even number of 0 and odd number of 1,

odd number of 0 and even number of 1,

odd number of 0 and odd number of 1

Let us try to design the machine

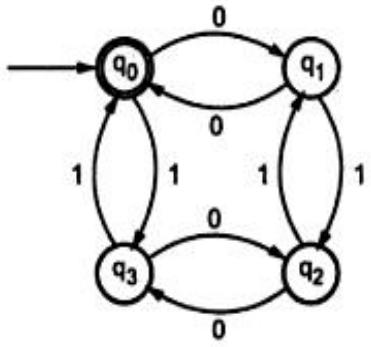


Fig. 2.13

Here q_0 is a start state as well as final state. Note carefully that a symmetry of 0's and 1's is maintained. We can associate meanings to each state as :

q_0 : State of even number of 0's and even number of 1's.

q_1 : State of odd number of 0's and even number of 1's.

q_2 : State of odd number of 0's and odd number of 1's.

q_3 : State of even number of 0's and odd number of 1's.

The transition table can be as follows -

	0	1
→ q_0	q_1	q_3
q_1	q_0	q_2
q_2	q_3	q_1
q_3	q_2	q_0

Example 2.9 : *Design FA to accept the string that always ends with 00.*

Solution :

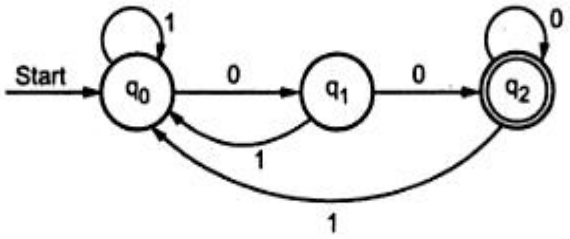


Fig. 2.14

If the input is 01001100 it will be processed as

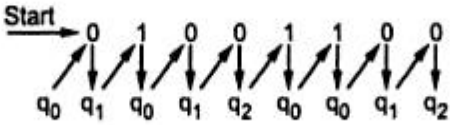


Fig. 2.15

The q_2 is a final state, hence the input is accepted.

Example 2.10 : *Construct the transition graph for a FA which accepts a language L over* $\Sigma\{0, 1\}$ *in which every string start with 0 and ends with 1.*

Solution :

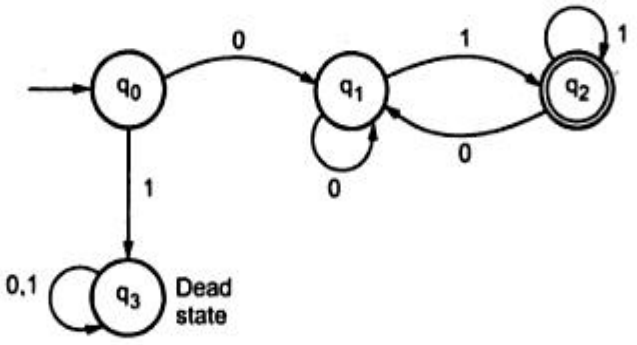


Fig. 2.16

In the above transition graph we have handled the case as if the input starts with 1 then it will be in q_3 state which is a dead state and never lead to final state. Thus this machine strictly handles the strings starting with 0 and ending with 1.

Example 2.11 : *Design FA to accept L, where L = {Strings in which a always appears trippled} over the set Σ = {a, b}.*

Solution : For this particular language the valid strings are *aaa*b, b*aaaaaa*, bb*aaa*b and so on. The *a* always appears in a clump of 3. The TG (transition graph) will look like this -

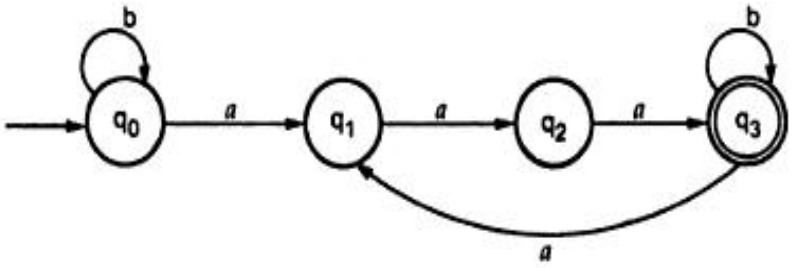


Fig. 2.17 For Ex. 2.11

Note that the sequence of tripple *a* is maintained to reach to the final state.

Example 2.12 : *Design FA to accept L where all the strings in L are such that total number of a's in them are divisible by 3.*

Solution : As we have seen earlier, while testing divisibility by 3, we group the input as remainder 0, remainder 1 and remainder 2.

Hence

S_0 : State of remainder 0

S_1 : State of remainder 1

S_2 : State of remainder 2

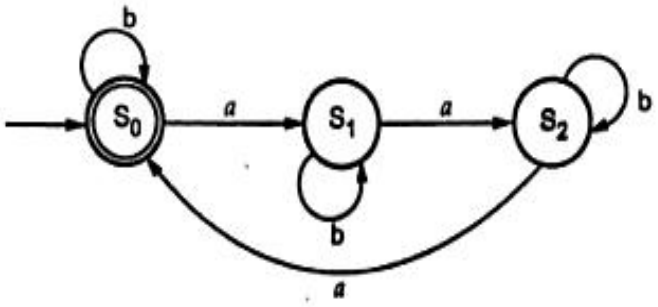


Fig. 2.18

Note the difference between previous example and this, here there is no condition as *a* should be in clump but total number of *a*'s in a string are divisible by 3. Hence b's are allowed in between.

Example 2.13 : *Design a FA that reads strings made up of letters in the word CHARIOT and recognize those strings that contain the word 'CAT' as a substring.*

Solution : To design this FA we will first consider the input set which will be Σ = {C, H, A, R I, O, T}. From this input set we have to recognize the word 'CAT'. We can do that as follows –

Step 1 :

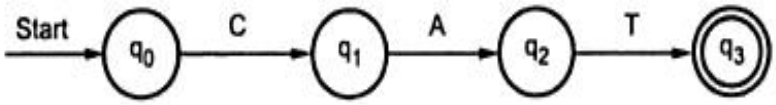


Fig. 2.19

Step 2 :

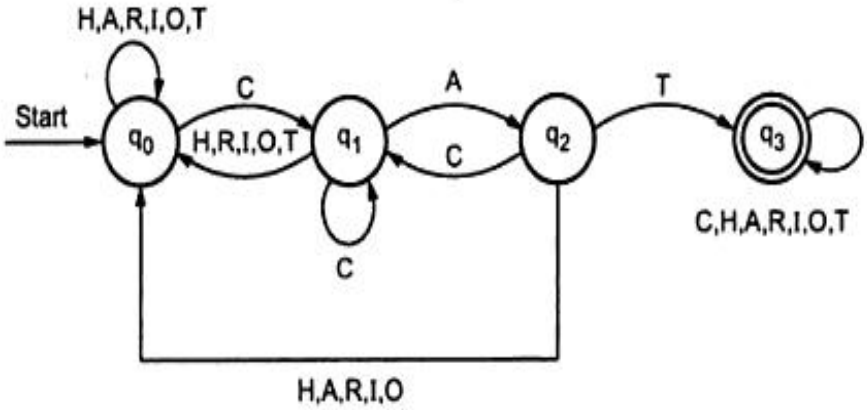


Fig. 2.20 Required finite automata

From above figure it is clear that on recognizing C initially from Σ = {C, H, A, R, I, O, T} we are moving to state q_1, other than C all the characters are remaining in state q_0 only. From q_1 the character A will be identified and from q_2 character T will be identifier and from q_2 character T will be identified. The transition table can be drawn as follows

State \ Input	C	H	A	R	I	O	T
$\rightarrow q_0$	q_1	q_0	q_0	q_0	q_0	q_0	q_0
q_1	q_1	q_0	q_2	q_0	q_0	q_0	q_0
q_2	q_1	q_0	q_0	q_0	q_0	q_0	q_3
(q_3)	q_3	q_3	q_3	q_3	q_3	q_3	q_3

The substring may appear anywhere in the input string for any number of times. For example 'HACCATHA'. From this 'CAT' can be recognized and we should reach to q_3 state finally.

2.3 Deterministic Finite Automata (DFA)

The finite Automata is called Deterministic Finite Automata if there is only one path for a specific input from current state to next state. For example, the DFA can be shown as below.

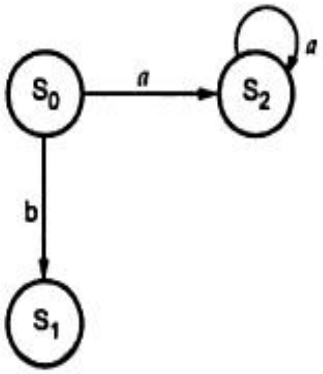


Fig. 2.21 Deterministic finite automata

From state S_0 for input '*a*' there is only one path, going to S_2. Similarly from S_0 there is only one path for input b going to S_1.

The DFA can be represented by the same 5-tuples described in the definition of FSM. All the above examples are actually the DFAs.

Definition of DFA

A deterministic finite automation is a collection or following things -

1) The finite set of states which can be denoted by Q.

2) The finite set of input symbols Σ

3) The start state q_0 such that $q_0 \in Q$

4) A set of final states F such that $F \in Q$

5) The mapping function or transition function denoted by δ. Two parameters are passed to this transition function : one is current state and other is input symbol. The transition function returns a state which can be called as next state.

For example $q_1 = \delta\ (q_0, a)$ means from current state q_0, with input a the next state transition is q_1.

In short, the DFA is a five tuple notation denoted as :

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

The name of DFA is A which is a collection above described five elements.

Method of Specifying DFA

This 5-tuple DFA can be represented by two ways – 1) Transition diagram 2) Transition table.

1) Transition Diagram : It is a 5-tuple graph used state and edges represent the transitions from one state to another.

For example –

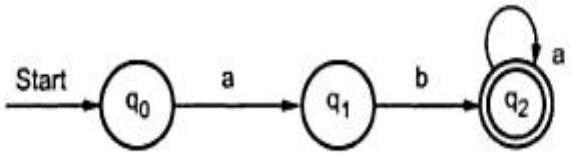


Fig. 2.22

2) Transition table - This is the tabular representation of the DFA. For a transition table the transition function is used.

For example

State \ Input	a	b
q_0	q_1	–
q_1	–	q_2
q_2	q_2	–

The rows of the table correspond to states and columns of the table correspond to inputs.

2.4 Non-Deterministic Finite Automata (NFA or NDFA)

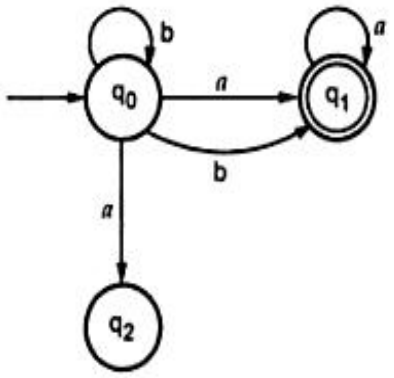


Fig. 2.23 Non-deterministic finite automata

The concept of Nondeterministc Finite Automata is exactly reverse of Deterministic Finite Automata. The Finite Automata is called NFA when there exists many paths for a specific input from current state to next state. The NFA can be shown as in Fig. 2.23.

Note that the NFA shows from q_0 for input a there are two next states q_1 and q_2. Similarly, from q_0 for input b the next states are q_0 and q_1.

Thus it is not fixed or determined that with a particular input where to go next. Hence this FA is called Nondeterministic finite Automata.

Consider the input string bb*a*. This string can be derived as

	Input	b	b	*a*
	Path	q_0	q_0	q_1
or	Input	b	b	*a*
	Path	q_0	q_0	q_2
or	Input	b	b	*a*
	Path	q_0	q_1	q_1

Thus you can not take the decision of which path has to be followed for deriving the given string.

Definition of NFA

The NFA can be formally defined as a collection of 5 tuples,

Q is a finite set of states

Σ is a finite set of inputs

δ is called next state or transition function

q_0 is initial state

F is a final state where $F \subseteq Q$.

There can be multiple final states. Thus the next question might be what is the use of NFA. The NFA is basically used in theory of computations because they are more flexible and easier to use than the DFAs.

Difference between NFA and DFA

The DFA is a deterministic finite automata whereas NFA is a non deterministic finite automata. In DFA, for a given state, on a given input we reach to a deterministic and unique state. On the other hand, in NFA we may lead to more than one states for given input. The DFA is a subset of NFA. We need to convert NFA to DFA in the design of compiler.

2.5 The Equivalence of DFA and the NFA

As we have discussed, the finite automata can either be DFA or NFA. You might be thinking now, who is better NFA or DFA. Which has more power ? Here is a theorem which tell you that any NFA can be converted to its equivalent DFA. That is any language L acceptable by NFA can be acceptable by its equivalent DFA. The basic idea in this theorem is that, DFA keeps track of all the states, that NFA could be in reading the same input as the DFA has read.

Let us see the theorem.

Theorem : Let L be a set accepted by non deterministic finite automation. Then there exists a deterministic finite automation that accepts L.

Proof : Let

$M = (Q, \Sigma, \delta, q_0, F)$ be an NFA for language L. Then define DFA M′ such that

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

The states of M′ are all the subset of M′. The $Q' = 2^Q$.

F′ be the set of all the final states in M.

The elements in Q′ will be denoted by $[q_1, q_2, q_3, \ldots q_i]$ and the elements in Q are denoted by $\{q_0, q_1, q_2, \ldots\}$ The $[q_1, q_2, \ldots q_i]$ will be assumed as one state in Q′ if in the NFA q_0 is a initial state it is denoted in DFA as $q_0' = [q_0]$ We define,

$\delta'([q_1, q_2, q_3, \ldots q_i], a) = [p_1, p_2, p_3, \ldots p_j]$

if and only if,

$\delta(\{q_1, q_2, q_3, \ldots q_i\}, a) = \{p_1, p_2, p_3, \ldots p_j\}$

This means that whenever in NFA, at the current states $\{q_1, q_2, q_3, \ldots q_i\}$ if we get input a and it goes to the next states $\{p_1, p_2, \ldots p_j\}$ then while constructing DFA for it the current state is assumed to be $[q_1, q_2, q_3, \ldots q_i]$ At this state, the input is a and the next is assumed to be $[p_1, p_2, \ldots p_j]$. On applying δ function on each of the states $q_1, q_2, q_3, \ldots q_i$ the new states may be any of the states from $[p_1, p_2, \ldots p_j]$. The theorem can be proved with the induction method by assuming length of input string x

$$\delta'(q_0', x) = [q_1, q_2, \ldots q_i]$$

if and only if,

$$\delta(q_0, x) = \{q_1, q_2, q_3, \ldots q_i\}$$

Basis : If length of input string is 0

i.e. $|x| = 0$, that means x is ε then $q_0' = [q_0]$

Induction : If we assume that the hypothesis is true for the input string of length m or less than m. Then if x a is a string of length m+1. Then the function δ' could be written as

$$\delta'(q_0', xa) = \delta'(\delta'(q_0, x), a)$$

By the induction hypothesis,

$$\delta'(q_0', x) = [p_1, p_2, \ldots p_j]$$

if and only if,

$$\delta(q_0, x) = \{p_1, p_2, p_3, \ldots p_j\}$$

By definition of δ'

$$\delta'([p_1, p_2, \ldots p_j], a) = [r_1, r_2, \ldots r_k]$$

if and only if,

$$\delta(\{p_1, p_2, \ldots p_j\} a) = \{r_1, r_2, \ldots r_k\}$$

Thus

$$\delta'(q_0', xa) = [r_1, r_2, \ldots r_k]$$

if and only if

$$\delta(q_0, x\ a) = \{r_1, r_2, \ldots r_k\}$$

is shown by inductive hypothesis.

Thus $L(M) = L(M')$

With the help of this theorem, let us try to solve few examples.

Example 2.14 : *Let* $M = (\{q_0, q_1\}, \{0,1\}, \delta, q_0, \{q_1\})$

be NFA where $\delta(q_0, 0) = \{q_0, q_1\}, \delta(q_0, 1) = \{q_1\}$,

$\delta(q_1, 0) = \phi, \delta(q_1, 1) = \{q_0, q_1\}$.

Construct its equivalent DFA.

Solution : Let the DFA $M' = (Q', \Sigma, \delta', q_0', F')$

Now, the δ' function will be computed as follows -

As $\delta(q_0, 0) = \{q_0, q_1\}$ $\delta'([q_0], 0) = [q_0, q_1]$

As in NFA the initial state is q_0, the DFA will also contain the initial state $[q_0]$

Let us draw the transition table for δ function for a given NFA.

		0	1		
$\delta(q_0, 0)$	$\Rightarrow$	$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_1\}$	$\Leftarrow$ $\delta(q_0, 1)$
$\delta(q_1, 0)$	$\Rightarrow$	q_1	ϕ	$\{q_0, q_1\}$	$\Leftarrow$ $\delta(q_1, 1)$

δ Function for NFA

From the transition table we can compute that there are $[q_0]$, $[q_1]$, $[q_0, q_1]$ states for its equivalent DFA. We need to compute the transition from state $[q_0, q_1]$

$$\delta(\{q_0, q_1\}, 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$$
$$= \{q_0, q_1\} \cup \phi$$
$$= \{q_0, q_1\}$$

So, $\delta'([q_0, q_1], 0) = [q_0, q_1]$

Similarly,

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

So, $\delta'([q_0, q_1], 1) = [q_0, q_1]$

As in the given NFA q_1 is a final state, then in DFA where ever q_1 exists that state becomes a final state. Hence in the DFA final states are $[q_1]$ and $[q_0, q_1]$ Therefore set of final states $F = \{[q_1], [q_0, q_1]\}$

The equivalent DFA is

	0	1
$\rightarrow q_0$	$[q_0, q_1]$	$[q_1]$
$[q_1]$	ϕ	$[q_0, q_1]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$

Transition table for equivalent DFA

2.2.1 Basic Definition of Finite Automata (FA)

A finite automata is a collection of 5 - tuples $(Q, \Sigma, \delta, q_0, F)$ where

Q is a finite set of states, which is non empty.

Σ is input alphabet, indicates input set

q_0 in Q is the initial state

F is a set of final states

δ is a transition function or a function defined for going to next state.

The finite automation can be represented as

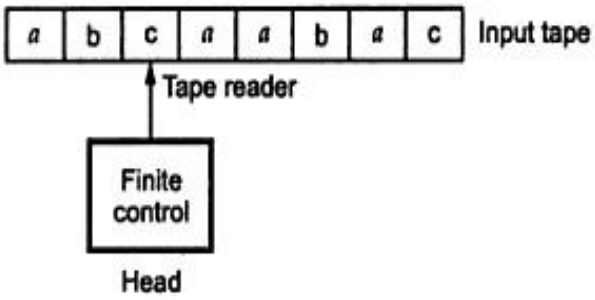


Fig. 2.1 Finite Automation

Usually the finite automation is a mathematical representation of finite state machine.

From Fig. 2.1 we can see that, the machine has a input tape on which the input can be placed. The tape reader is for pointing the character which is to be read from input tape.

The finite control is always one of internal states which decides what will be the next state after reading the input tape by a reader. For example suppose current state is q_1 and suppose now the reader is reading c, it is a finite control which decides what will be the next state at input c.

Transition graph : The FA is associated with a directed graph called a transition diagram or transition graph. The vertices of this graph corresponds to the states.

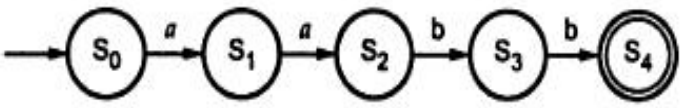


Fig. 2.2

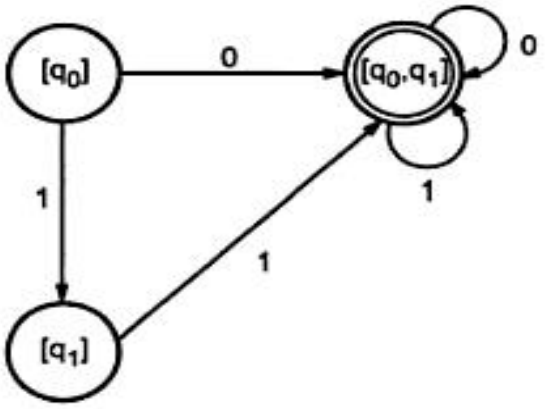


Fig. 2.24

Even, we can change the names of the states of DFA

$$A = [q_0]$$

$$B = [q_1]$$

$$C = [q_0, q_1]$$

With these new names the DFA will be as follows -

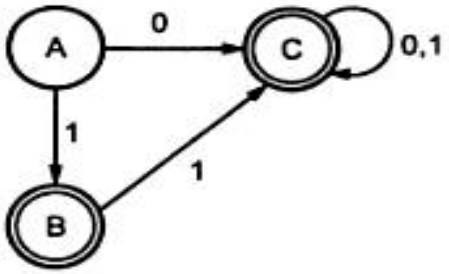


Fig. 2.25 An equivalent DFA

Example 2.15 : *Convert the given NFA to DFA*

	0	1
→ q_0	$\{q_0, q_1\}$	q_0
q_1	q_2	q_1
q_2	q_3	q_3
(q_3)	ϕ	q_2

Solution : As we know, first we will compute δ′ function

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

Hence $\delta'([q_0], 0) = [q_0, q_1]$

Similarly,

$$\delta(\{q_0\}, 1) = \{q_0\}$$

Hence $\delta'([q_0], 1) = [q_0]$

Thus we have got a new state $[q_0, q_1]$.

Let us check how it behaves on input 0 and 1.

So,

$$\begin{aligned}\delta'([q_0, q_1], 0) &= \delta([q_0], 0) \cup \delta([q_1], 0) \\ &= \{q_0, q_1\} \cup \{q_2\} \\ &= \{q_0, q_1, q_2\}\end{aligned}$$

Hence a new state is generated i.e. $[q_0, q_1, q_2]$

Similarly

$$\begin{aligned}\delta'([q_0, q_1], 1) &= \delta([q_0], 1) \cup \delta([q_1], 1) \\ &= \{q_0\} \cup \{q_1\} \\ &= \{q_0, q_1\}\end{aligned}$$

No new state is generated here.

Again δ′ function will be computed for $[q_0, q_1, q_2]$, the new state being generated.

State	0	1
$[q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_0, q_1]$
$[q_0, q_1, q_2]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_3]$

As, you have observed in the above table for a new state $[q_0, q_1, q_2]$ the input 0 will give a new state $[q_0, q_1, q_2, q_3]$ and input 1 will give a new state $[q_0, q_1, q_3]$ because

$$\begin{aligned}\delta'([q_0, q_1, q_2], 0) &= \delta'([q_0], 0) \cup \delta'([q_1], 0) \cup \delta'([q_2], 0) \\ &= \{q_0, q_1\} \cup \{q_2\} \cup \{q_3\}\end{aligned}$$

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

$= [q_0, q_1, q_2, q_3]$

Same procedure for input 1. Thus the final DFA is as given below.

State	0	1
→ $[q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_1]$	$[q_2]$	$[q_1]$
$[q_2]$	$[q_3]$	$[q_3]$
$[q_3]$	ϕ	$[q_2]$
$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_0, q_1]$
$[q_0, q_1, q_2]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_3]$
$[q_0, q_1, q_3]$	$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3]$

DFA for example 14

Example 2.16 : *Construct DFA equivalent to the given NFA*

	0	1
→ p	{p, q}	p
q	r	r
r	s	–
s	s	s

The NFA M = {{p, q, r, s}, {0, 1}, δ {p}, {s}}

The equivalent DFA will be constructed.

	0	1
→ [p]	[p, q]	[p]
[q]	[r]	[r]
[r]	[s]	–
[s]	[s]	[s]
[p, q]	[p, q, r]	[p, r]

Continuing with the generated new states.

	0	1
→ [p]	[p, q]	[p]
[q]	[r]	[r]
[r]	[s]	–
([s])	[s]	[s]
[p, q]	[p, q, r]	[p, r]
[p, q, r]	[p, q, r, s]	[p, r]
[p, r]	[p, q, s]	[p]
([p, q, r, s])	[p, q, r, s]	[p, r, s]
([p, q, s])	[p, q, r, s]	[p, r, s]
([p, r, s])	[p, q, s]	[p, s]
([p, s])	[p, q, s]	[p, s]

The final state F' = { [s], [p, q, r, s], [p, q, s], [p, r, s], [p, s] }

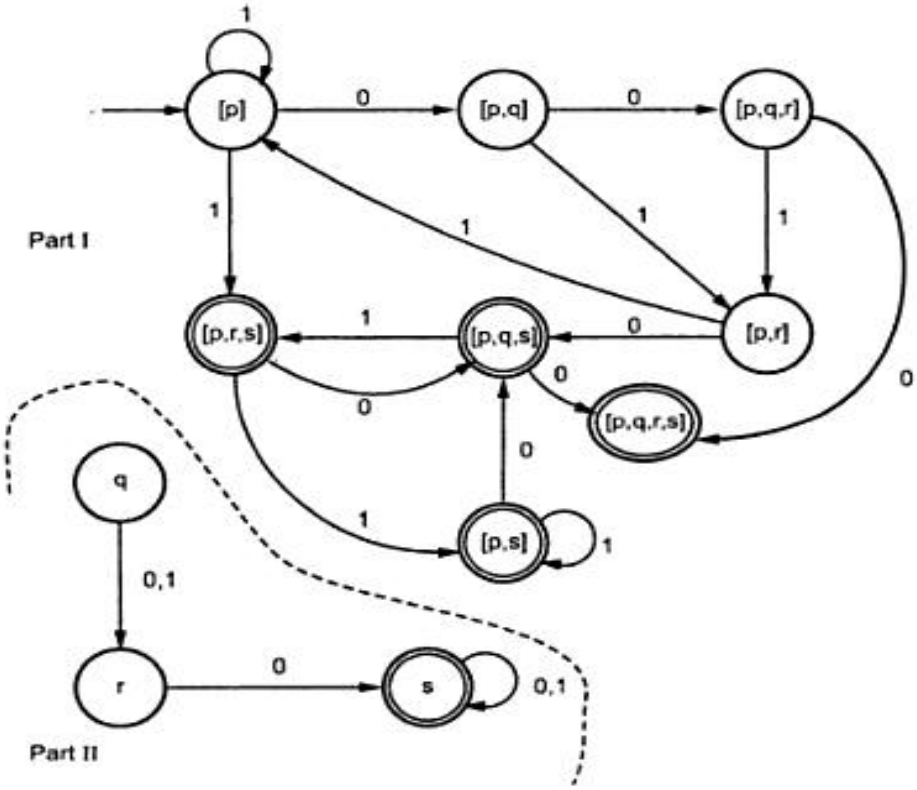


Fig. 2.26

The transition graph shows two disconnected parts. But part I will be accepted as final DFA because it consists of start state and final states, in part II there is no start state.

2.6 NFA with ε Moves

The ε is a character used to indicate null string i.e. the string which is used simply for transition from one state to other without any input. The NFA with ε moves can be shown as below.

For example

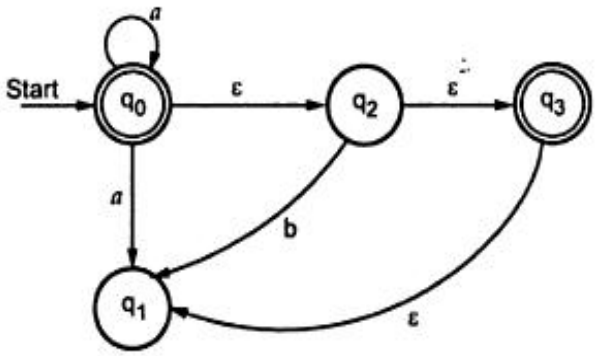


Fig. 2.27 NFA with ε moves

We can convert the given NFA with ε moves to the NFA without ε moves. Let us take one example to see how this happens.

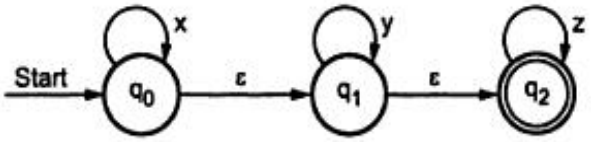


Fig. 2.28 Finite Automata with ε moves

The important aspect for building the transition table is to compute the δ function. The δ the transition function, maps $Q \times (\Sigma \cup \{\varepsilon\})$ to 2^Q. The intention is that δ (q, *a*) will consist of all states p such that there is a transition labelled a from q to p where a is either ε or the symbol in ε. Let us design the transition table with δ function for Fig. 2.28.

	x	y	z	ε
q_0	$\{q_0\}$	ϕ	ϕ	$\{q_1\}$
q_1	ϕ	$\{q_1\}$	ϕ	$\{q_1\}$
q_2	ϕ	ϕ	$\{q_2\}$	ϕ

Let δ" (q, w) will be all states p such that one can go from q to p along a path labelled w, sometime including edges labelled ε. While constructing δ" we have to

compute the set of states reachable from a given state q using ε transitions only. Let us use ε CLOSURE (q) to denote the set of all vertices p such that there is a path from q to p labelled ε.

The δ" can be interpreted as follows -

$$\delta''(q, \varepsilon) = \varepsilon \text{ - CLOSURE } (q)$$

For w in Σ^* and a in Σ, $\delta''(q, wa) = \varepsilon$ CLOSURE (p)

where $p = \{p \mid \text{for some } r \text{ in } \delta''(q, w)\ p \text{ is in } \delta(r, a)\}$

For the transition table

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

Thus

$$\delta''(q_0, x) = \varepsilon\text{-CLOSURE } (\delta(\delta''(q_0, \varepsilon), x))$$
$$= \varepsilon\text{-CLOSURE } (\delta(\{q_0, q_1, q_2\}, x))$$
$$= \varepsilon\text{-CLOSURE } (\delta(q_0, x) \cup \delta(q_1, x) \cup \delta(q_2, x))$$
$$= \varepsilon\text{-CLOSURE } (\{q_0\} \cup \phi \cup \phi)$$
$$= \varepsilon\text{-CLOSURE } [\{q_0\}]$$
$$\delta''(q_0, x) = \{q_0, q_1, q_2\}$$
$$\delta''(q_0, xy) = \varepsilon\text{-CLOSURE } (\delta(\delta''(q_0, x), y))$$
$$= \varepsilon\text{-CLOSURE } (\delta(\{q_0, q_1, q_2\}, y))$$
$$= \varepsilon\text{-CLOSURE } (\{q_1\})$$
$$= \{q_1, q_2\}$$

Thus the transition table can be drawn as -

	x	y	z
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	ϕ	$\{q_1, q_2\}$	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_2\}$

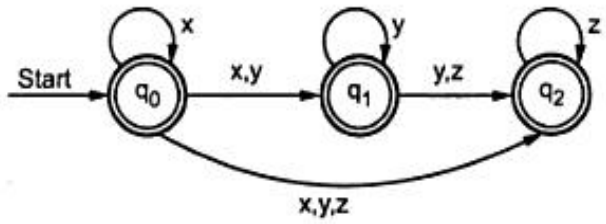


Fig. 2.29 NFA without ε transitions

Conversion of NFA with ε transitions to NFA without ε transitions -

As we have seen one example of conversion of NFA with ε transitions. Let us try to prove this with the induction method.

Theorem : If L is accepted by NFA with ε transitions, then there exist L which is accepted by NFA without ε transitions.

Proof :

Let, $M = (Q, \sum, \delta, q_0, F)$ be an NFA with ε transitions.

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

$$F' = \begin{cases} F \cup \{q_0\} & \text{if } \varepsilon\text{ - CLOSURE contains state off} \\ F & \text{otherwise} \end{cases}$$

M' is a NFA without ε moves. The δ' function can be denoted by δ" with some input. For example, δ' (q, *a*) = δ" (q, *a*) for some q in Q and a from Σ . We will apply the method of induction with input x. The x will not be ε because

$\delta'(q_0, \varepsilon) = \{q_0\}$.

$\delta''(q_0, \varepsilon) = \varepsilon$-CLOSURE (q_0). Therefore we will assume length of string to be 1.

Basis : $|x| = 1$. Then x is a symbol *a*

$$\delta'(q_0, a) = \delta''(q_0, a)$$

Induction : $|x| > 1$ Let $x = wa$

$$\delta'(q_0, w\ a) = \delta'(\delta'(q_0, w), a)$$

By inductive hypothesis,

$$\delta'(q_0, w) = \delta''(q_0, w) = p$$

Now we will show that $\delta'(p, a) = \delta(q_0, w\ a)$

But $$\delta'(p, a) = \bigcup_{q \text{ in } p} \delta'(q, a) = \bigcup_{q \text{ in } p} \delta''(q, a)$$

as $p = \delta''(q_0, w)$ we have,

$$\cup \quad \delta''(q, a) = \delta''(q_0, wa)$$

q in p

Thus by definition δ''

$$\delta'(q_0, wa) = \delta''(q_0, wa)$$

2.7 NFA with ε Moves to DFA

As we have seen that NFA with ε moves can be converted to NFA without ε moves. To gain the DFA, we can follow the following diagram.

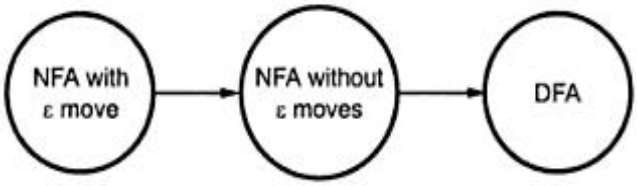


Fig. 2.30 Moves to DFA

Example 2.17 : *Construct an equivalent NFA without ε from given NFA with ε .*

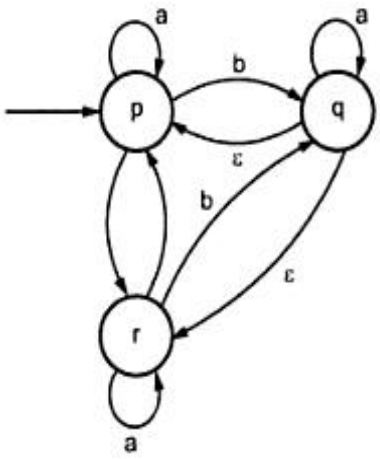


Fig. 2.31

Solution : We will find out ε- closure function of every state for given input Σ = {a, b, c}.

$$\varepsilon - \text{closure } \{P\} = \{P\}$$

$$\delta \{p, a\} = \delta \{\varepsilon - \text{closure } \{p, a\}$$

$$= \{p\}$$

$$\delta \{p, b\} = \delta \{\varepsilon - \text{closure } \{p, b\}$$

$$= \{q\}$$

$$\delta \{p, c\} = \delta \{\varepsilon - \text{closure } \{p, c\}$$

$$= \{r\}$$

ε - closure {q} = {pq}

δ {q, a} = {ε - closure {p, a} $\cup$ ε closure {q, a}

= {p,q}

δ {a, b} = δ {ε - closure {p, b} $\cup$ ε closure {q, b}

= q $\cup$ r

= {q,r}

δ {q, c} = δ {ε - closure {p, c} $\cup$ ε closure {q, c}

= {r}

ε - closure {r} = }p, q}

δ {r, a} = δ {ε - closure {p, a} $\cup$ ε closure {q, a}

= {p, q}

δ {r, b} = δ {ε - closure {p, b} $\cup$ ε closure {q, b}

= {q, r}

δ {r, c} = δ {ε - closure {p, c} $\cup$ ε closure {q, c}

= {r}

Hence the NFA can be drawn as -

		a	b	c
→	{p}	{p}	{q}	{r}
	{q}	{p, q}	{q, r}	{r}
•	{r}	{p, q}	{q, r}	{r}

The transition diagram can be drawn as –

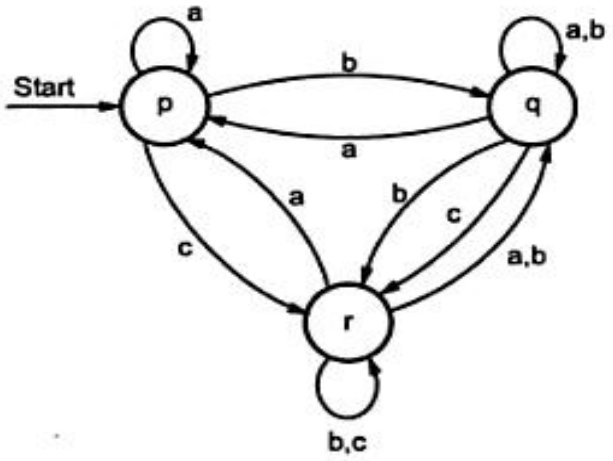


Fig. 2.32

2.8 Minimization of DFA

The minimization of DFA means reducing the number of states from given FA. Thus we get the DFA with redundant states after minimizing the DFA.

While minimizing DFA we first find out which two states are equivalent we then replace those two states by one representative state. Now for finding the equivalent states we will apply the following rule.

"The two states S_1 and S_2 are equivalent if and only if both the states are final states or both are non-final states."

Let us understand the rule with the help of some examples.

Example 2.18 : *Construct a minimized DFA for*

		0	1
	→ *a*	b	f
•	b	g	c
	Ⓒ	*a*	c
⇒	d	c	g
	e	h	f
⇒	f	c	g
	g	g	e
•	h	g	c

Solution : From the given table, the state *a* is a initial state, state c is a final state. Now, we can notice that states b and h gives the same output states on receiving the respective input. And both the states are non final states. So we can reduce those states we can say, b = h i.e. wherever h occurs we will replace it by b, similarly d = f and hence we will replace every f by d.

		0	1
•	→ *a*	b	d
	b	g	c
	Ⓒ	*a*	c
	d	c	g
•	e	b	d
	g	g	e

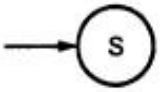
For example - We draw transition diagram for the input *a a* b b as follows

Note that the start state is S_0 and final state is S_4. The input set here is $\Sigma = \{a, b\}$ S_1, S_2 and S_3 all are the intermediate states.

For a transition diagram following notations are used -

State : q_0 is a name of the state (you can give any name to a state)

transition from one state to another



(or)

denotes the start or initial state

(or)

denotes the final state

Mapping functions : The FSM has two forms of mapping functions.

Machine function and state function.

The machine function is a function which denotes, with certain input what will be the output from current state. It can be written as

$$S \times I = O$$

where S is a current state

I is the input

O is the output

The state function is a function which denotes with certain input from current state what will be the next state. It can be written as -

$$S \times I = S_1$$

Now *a* = e and both are non final states. So we will replace every e by *a*.

	0	1
→ *a*	b	d
b	g	c
c	*a*	c
d	c	g
g	g	*a*

Now there is no reduction possible further. Hence this is a minimized DFA.

Example 2.19 : *Construct a minimized DFA for the given transition table.*

	0	1
→ q_0	q_1	q_5
q_1	q_6	q_2
((q_2))	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

Solution : From given transition table we can get two non final states which are equivalent and those are $q_3 = q_5$. These two states have similar output on receiving input 0 and 1. So we will replace q_5 by q_3 every where.

	0	1
→ q_0	q_1	q_3
q_1	q_6	q_2
((q_2))	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_3
q_6	q_6	q_4
q_7	q_6	q_2

Now again we find $q_1 = q_7$. These two non final states are equivalent.

	0	1
→ q_0	q_1	q_3
q_1	q_6	q_2
((q_2))	q_0	q_2
q_3	q_2	q_6
q_4	q_1	q_3
q_6	q_6	q_4

Again, $q_0 = q_4$

	0	1
→ q_0	q_1	q_3
q_1	q_6	q_2
((q_2))	q_0	q_2
q_3	q_2	q_6
q_6	q_1	q_3

Again, $q_0 = q_6$

	0	1
→ q_0	q_1	q_3
((q_1))	q_0	q_2
q_2	q_0	q_2
q_3	q_2	q_0

Now again we can see q_1 and q_2 gives the same output states for input 0 and 1. But these can not be the equivalent states because q_2 is final state and q_1 is a nonfinal state. Hence this is a final minimized DFA.

Solved Examples

Example 2.20 : *Design FSM that accepts the string with exactly two zeros.*

Solution :

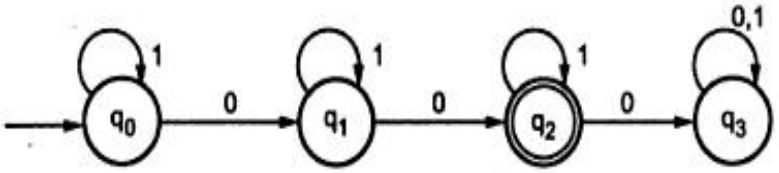


Fig. 2.33

At every state there is a loop of 1 that means there can be possibility of occurance of 1's in between 0's. The count of 0 has to be maintained as exactly two. There is no restriction on number of 1's. You can try out a valid string '101011' will lead to the final state q_2. The state q_3 is a dead state. For the invalid inputs you will fell in deadstate like q_3.

Example 2.21 : *Design FSM for the language that accepts the string containing at least two zero's over the* $\Sigma = \{0, 1\}$.

Solution : The condition is for at least two zeros and there is no restriction on number of 1's. At least two zeros means minimum two zeros are necessary.

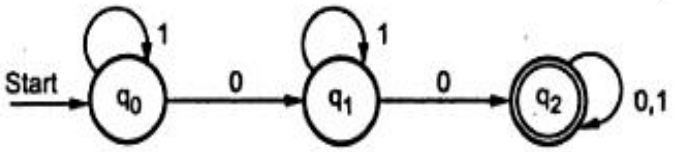


Fig. 2.34

You are suggested to note the difference between the FSM drawn in example 18 and this FSM.

Example 2.22 : *Design FSM for the language L consisting of the strings containing at the most two zeros.*

Solution : In this problem, at the most two 0's are allowed. This means no zero, single zero or two zeros will do ! But not more than two zeros are allowed. The TG for this will look like this -

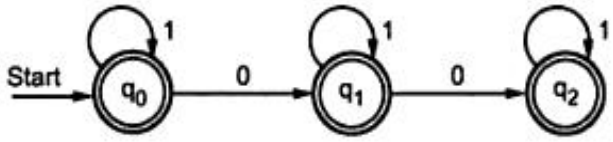


Fig. 2.35

Here q_0 is a start and final state. This state indicates that there is no zero in the input string. As there is no restriction on number 1's, the input 1 may occur at any number of times at any place in the string.

➠ **Example 2.23 :** *Design FSM which contains, the string containing double zeros always over* $\Sigma = \{0, 1\}$

Solution : Here, the condition is that whenever zero will appear it should be double. No single zero is allowed in the string. The 1 will be flexible i.e. it may appear any number of times.

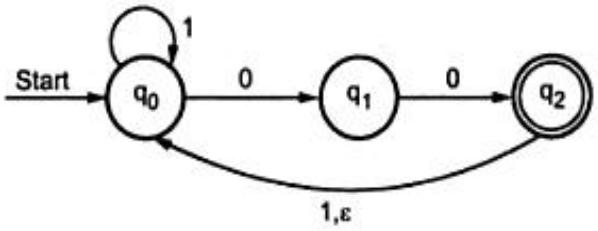


Fig. 2.36

The edge from q_2 to q_0 is followed for input either 1 or ε. The ε indicates no string.

For example if wish to derive the above FSM for 0000, then ε edge will be important. The FSM will derive it as 00ε00, but which is equal to 0000. Thus because of ε the recursion is possible.

➠ **Example 2.24 :** *Design FSM which accepts the language in which every string starts with 0 followed by any number of 1's but having no two consecutive zeros.*

Solutin : Here the strings are starting with 0 and to have the situation as not to have the consecutive zeros, we have shown the transition between q_1 and q_2 as alternate 0 and 1. The machine can accept the input as 0111 i.e. starting with 0 and followed by no (absolutely no) consecutive zeros.

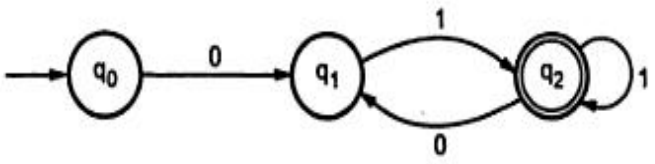


Fig. 2.37

Example 2.25 : *Describe in English the string accepted by following FA.*

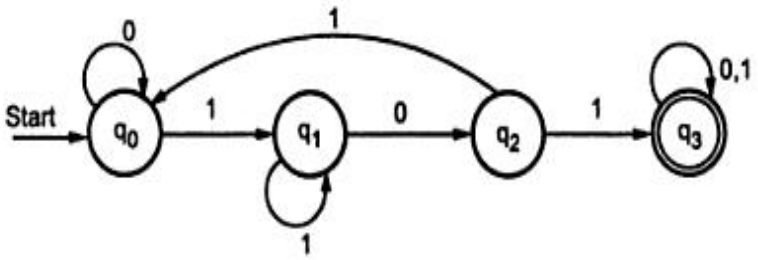


Fig. 2.38

Solution : If we follow the above FA, in straight, we will get a string 101. If the state q_1 is a current state then even after many iterations to q_1 with input 1 we will reach to final state q_3 after following 01, latter on it may come across any number of 0's and 1's. Thus we can predict that the above machine is for the language containing 101 as a substring, over the input set $\Sigma = \{0,1\}$.

Example 2.26 : *Construct an equivalent DFA for following NFA.*

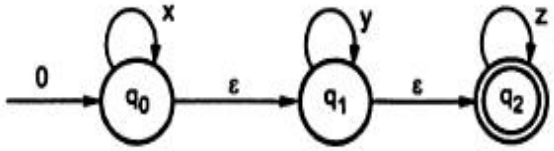


Fig. 2.39

Solution : The NFA given in the problem statement is a NFA with ε moves. We will first convert it to NFA without ε moves. From NFA without ε moves the equivalent DFA can be constructed.

We have discussed conversion of NFA with ε moves to NFA with cut ε moves in previous section. And diagram for NFA without εmoves can be

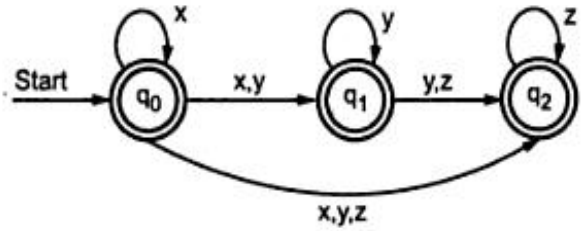


Fig. 2.40

Now we will convert this to equivalent DFA. The given NFA can be written using transition table as -

	x	y	z
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	–	$\{q_1, q_2\}$	$\{q_2\}$
q_2	–	–	$\{q_2\}$

We can see that $\delta\ (q_0, x) = \{q_0, q_1, q_2\}$ hence we will assume $\{q_0, q_1, q_2\}$ as a state $[q_0, q_1, q_2]$.

For this new state $[q_0, q_1, q_2]$ assume input x,y and z.

$\delta\ \{[q_0, q_1, q_2], x\} = [q_0, q_1, q_2]$

$\delta\ \{[q_0, q_1, q_2], y\} = [q_1, q_2]$

$\delta\ \{[q_0, q_1, q_2], z\} = [q_2]$

The new state generated is $\{q_1, q_2\}$. Hence we will check the input transitions for state $\{q_1, q_2\}$ (Note that square bracket is used for state representation)

$\delta\ \{[q_1, q_2], x\} = -$

$\delta\ \{[q_1, q_2], y\} = [q_1, q_2]$

$\delta\ \{[q_1, q_2], z\} = [q_2]$

Hence finally the DFA will be –

State \ Input	x	y	z
$[q_0]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$	$[q_2]$
$[q_1]$	–	$[q_1, q_2]$	$[q_2]$
$[q_2]$	–	–	$[q_2]$
$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$	$[q_2]$
$[q_1, q_2]$	–	$[q_1, q_2]$	$[q_2]$

Here q_0 is a start state and q_0, q_1, q_2, $[q_0, q_1, q_2]$, $[q_1, q_2]$ are all final states. The transition diagram can be drawn.

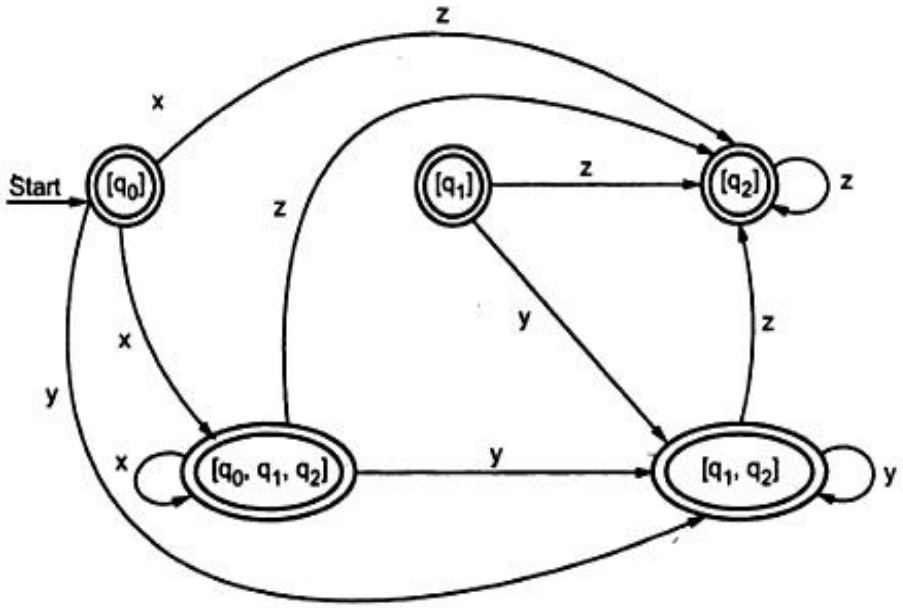


Fig. 2.41

Review Questions

1. *Describe the language represented by following DFA*

	0	1
$\rightarrow * q_0$	q_1	q_0
$* q_1$	q_2	q_0
q_2	q_2	q_2

2. *Design a finite automaton for the language consisting of all the strings with 011 as substring over* $\Sigma = \{0, 3\}$

3. *Convert the following NFA to DFA and also describe the language accepted by it*

	0	1
$\rightarrow p$	{p, q}	{p}
q	{r, s}	{t}
r	{p, r}	{t}
(s)	ϕ	ϕ
(t)	ϕ	ϕ

4. *Explain the significance of ε transitions in finite state system.*

where S_1 indicates the next state.

Thus at any stage the machine will act as either on arrival of an input symbol it will change the state using state function or it will give output using machine function.

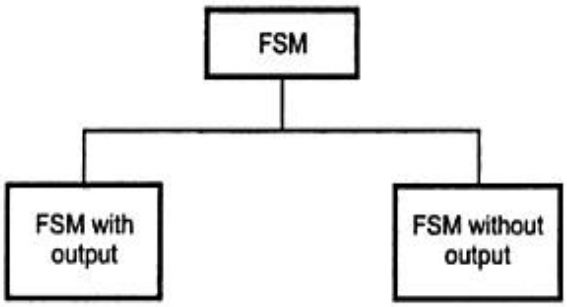


Fig. 2.3 Finite state machine

Now, we will solve some examples based on FSM without output.

➠ **Example 2.1 :** *Design a FA which accepts the only input 101 over the input set* $Z = \{0, 1\}$

Solution :

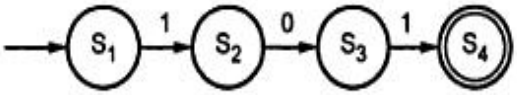


Fig. 2.4 For Ex. 2.1

Note that in the problem statement it is mentioned as only input 101 will be accepted. Hence in the solution we have simply shown the transitions, for input 101. There is no other path shown for other input.

➠ **Example 2.2 :** *Design a FA which checks whether the given binary number is even.*

Solution : The binary number is made up of 0's and 1's when any binary number ends with 0 it is always even and when a binary number ends with 1 it is always odd. For example

0100 is a even number, it is equal to 4.

0011 is a odd number, it is equal to 3.

and so while designing FA we will assume one start state, one state ending in 0 and other state for ending with 1. Since we want to check whether given binary number is even or not, we will make the state for 0, the final state.

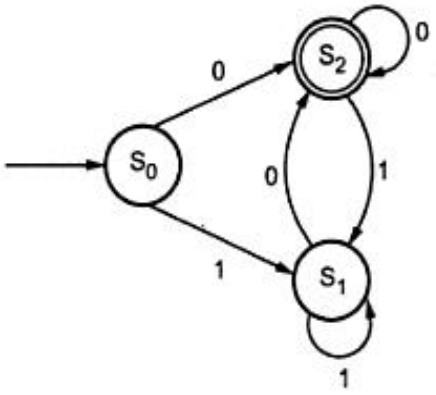


Fig. 2.5 For Ex. 2.2

The FA indicates clearly S_1 is a state which handles all the 1's and S_2 is a state which handles all the 0's. Let us take some input

$$01000 \Rightarrow 0\underline{S_2}1000$$

$$01\underline{S_1}000$$

$$010\underline{S_2}00$$

$$0100\underline{S_2}0$$

$$01000S_2$$

Now at the end of input we are in final or in accept state so it is a even number. Similarly let us take another input.

$$1011 \Rightarrow 1\underline{S_1}011$$

$$10\underline{S_2}11$$

$$101\underline{S_1}1$$

$$1011S_1$$

Now we are at the state S_1 which is a non final state.

Another idea to represent FA with the help of transition table.

State \ Input	0	1
→ S_0	S_2	S_1
S_1	S_2	S_1
(S_2)	S_2	S_1

Transition table

Thus the table indicates in the first column all the current states. And under the column 0 and 1 the next states are shown.

The first row of transition table can be read as : When current state is S_0, on input 0 the next state will be S_2 and on input 1 the next state will be S_1. The arrow marked to S_0 indicates that it is a start state and circle marked to S_2 indicates that it is a final state.

Example 2.3 : *Design FA which accepts only those strings which start with 1 and ends with 0.*

Solution : The FA will have a start state A from which only the edge with input 1 will go to next state.

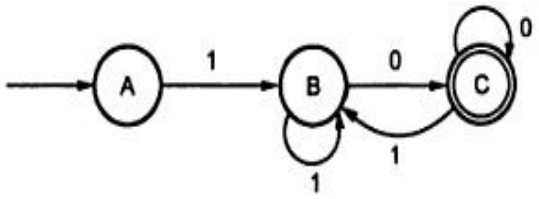


Fig. 2.6 For Ex. 2.3

In state B if we read 1 we will be in B state but if we read 0 at state B we will reach to state C which is a final state. In state C if we read either 0 or 1 we will go to state C or B respectively. Note that the special care is taken for 0, if the input ends with 0 it will be in final state.

Example 2.4 : *Design FA which accepts odd number of 1's and any number of 0's.*

Solution : In the problem statement, it is indicated that there will be a state which is meant for odd number of 1's and that will be the final state. There is no condition on number of 0's.

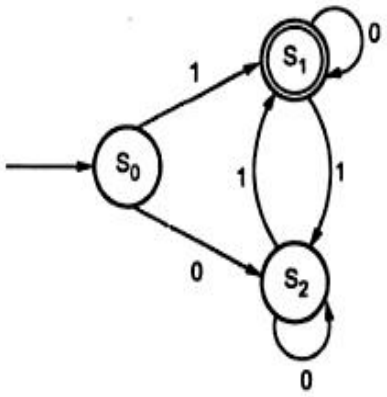


Fig. 2.7 For Ex. 2.4

At the start if we read input 1 then we will go to state S_1 which is a final state as we have read odd number of 1's. There can be any number of zeros at any state and therefore the self loop is applied to state S_2 as well as to state S_1. For example if the

input is 10101101, in this string there are any number of zeros but odd number of ones. The machine will derive this input as follows -

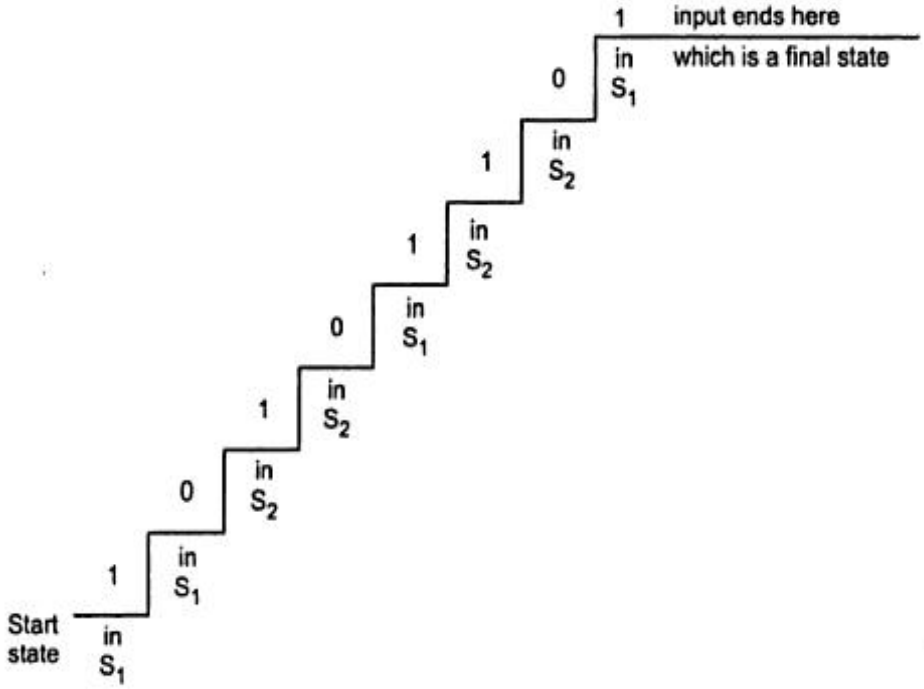


Fig. 2.8 Ladder diagram of processing the input

Example 2.5 : *Design FA which checks whether the given unary number is divisible by 3.*

Solution :

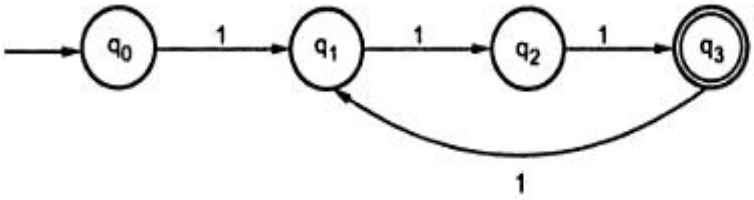


Fig. 2.9 For Ex. 2.5

The unary number is made up of ones. The number 3 can be written in unary form as 111, number 5 can be written as 11111 and so on. The unary number which is divisible by 3 can be 111 or 111111 or 111111111 and so on. The transition table is as follows

Input	1
→ q_0	q_1
q_1	q_2
q_2	q_3
(q_3)	q_1

Consider a number 111111 which is equal to 6 i.e. divisible by 3. So after complete scan of this number we reach to final state q_3.

Start 111111

State q_0

$1\underline{q_1}11111$

$11\underline{q_2}1111$

$111q_3111$

$1111\underline{q_1}11$

$11111\underline{q_2}1$

$111111q_3$ → Now we are in final state.

Example 2.6 : *Design FA to check whether given decimal number is divisible by three.*

Solution : To determine whether the given decimal number is divisible by three, we need to take the input number digit by digit. Also, while considering its divisibility by three, we have to consider that the possible remainders could be 1, 2 or 0. The remainder 0 means, it is divisible by 3. Hence from input set {0, 1, 2, ..., 9} (since decimal number is a input), we will get either remainder 0 or 1 or 2 while testing its divisibility by 3. So we need to group these digits according to their remainders. The groups are as given below -

remainder 0 group : S_0 : (0, 3, 6, 9)

remainder 1 group : S_1 : (1, 4, 7)

remainder 2 group : S_2 : (2, 5, 8)

We have named out these states as S_0, S_1 and S_2. The state S_0 will be the final state as it is remainder 0 state.

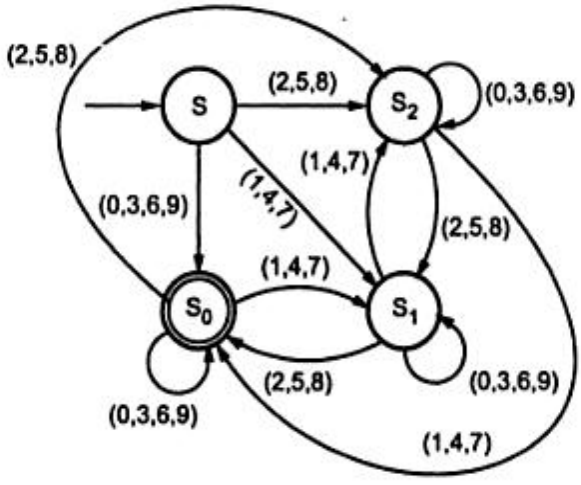


Fig. 2.10 For Ex. 2.6

Let us test the above FA, if the number is 36 then it will proceed by reading the number digit by digit.

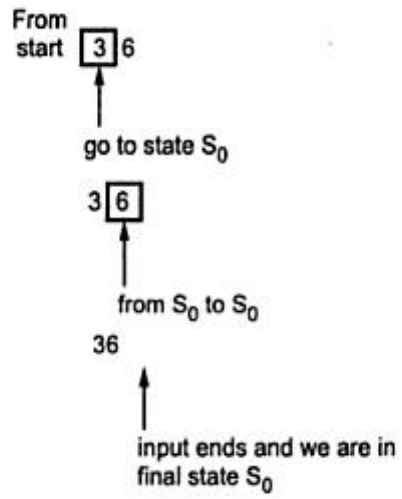


Fig. 2.11

Hence the number is divisible 3, isn't it ? Similarly if number is 121 which is not divisible by 3, it will be processed as

S 121

$1\underline{S_1}\,21$

$12\underline{S_0}\,1$

$121\,S_1$ which is remainder 1 state.

➠ **Example 2.7 :** *Design FA which checks whether a given binary number is divisible by three.*

Solution : As we have discussed in previous example, the same logic is applicable to this problem even ! The input number is a binary number. Hence the input set will be $\Sigma = \{0, 1\}$. The start state is denoted by S, the remainder 0 is by S_0, remainder 1 by S_1 and remainder 2 by S_2.

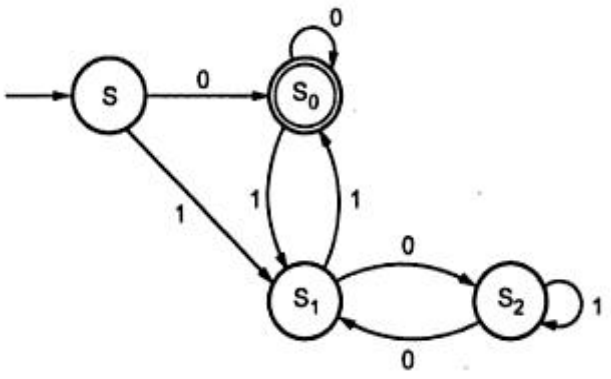


Fig. 2.12 For Ex. 2.7

TWO MARKS

1. Define: (i) Finite Automaton(FA) (ii)Transition diagram April /May 2008

   FA consists of a finite set of states and a set of transitions from state to state that occur on input symbols chosen from an alphabet _. Finite Automaton is denoted by a 5- tuple(Q,Σ, δ,q0,F), where Q is the finite set of states , _ is a finite input alphabet, q0 in Q is the initial state, F is the set of final states and _ is the transition mapping function Q * _ to Q.

   Transition diagram is a directed graph in which the vertices of the graph correspond to the states of FA. If there is a transition from state q to state p on input a, then there is an arc labeled ' a ' from q to p in the transition diagram.

2. What is a : (a) String (b) Regular language

   A string x is accepted by a Finite Automaton M=(Q, Σ, δ.q0,F) if δ (q0,x)=p, for some p in F.FA accepts a string x if the sequence of transitions corresponding to the symbols of x leads from the start state to accepting state.

   The language accepted by M is L(M) is the set {x | _(q0,x) is in F}. A language is regular if it is accepted by some finite automaton.

3. Why are switching circuits called as finite state systems?

   A switching circuit consists of a finite number of gates, each of which can be in any one of the two conditions 0 or 1.Although the voltages assume infinite set of values,the electronic circuitry is designed so that the voltages orresponding to 0 or 1 are stable and all others adjust to these value. Thus control unit of a computer is a finite statesystem.

4. What is Deductive proofs?

   A deductive proof consists of a sequence of statements which starts from a hypothesis or a given statement to a conclusion. Each step is satisfying some logical principle.

5. Define proof by contrapositive.

   It is the other form of if then statement. The contrapositive of the statement " if H then C " is If not C then not H ".

6. Define the concatenation of two strings.

   Suppose x and y are two strings then the concatenation of x and y is xy.

   Ex: if x = 0011 and y = 1100 then xy = 00111100.

7. What are the applications of automata theory?

   _ In compiler construction.
   _ In switching theory and design of digital circuits.
   _ To verify the correctness of a program.
   _ Design and analysis of complex software and hardware systems.
   _ To design finite state machines such as Moore and mealy machines.

8. What is Moore machine and Mealy machine?

   A special case of FA is Moore machine in which the output depends on the state of the machine. An automaton in whch the output depends on the transition and current input is called Mealy machine.

9. Define Language.

   A set of strings all of which are chosen from some Σ^*, where Σ is a particular alphabet is called a language.

   Ex: The language L over {0,1} where set of strings with an equal number 0's and 1's.

   L = { ε, 01, 10, 0011, 0101, 1001, 1100,……}

10. What are the components of Finite automaton model?
    The components of FA model are Input tape, Read control and finite control.
    (a)The input tape is divided into number of cells. Each cell can hold one i/p symbol
    (b)The read head reads one symbol at a time and moves ahead.
    (c)Finite control acts like a CPU. Depending on the current state and input symbol read from the input tape it changes state.

11. Differentiate NFA and DFA

    NFA or Non Deterministic Finite Automaton is the one in which there exists many paths for a specific input from current state to next state. NFA can be used in theory of computation because they are more flexible and easier to use than DFA .

    Deterministic Finite Automaton is a FA in which there is only one path for a specific input from current state to next state. There is a unique transition on each input symbol.(Write examples with diagrams).

12. What is ε-closure of a state q0?

    ε -closure(q0) denotes a set of all vertices p such that there is a path from q0 to p labeled ε. Example :closure(q0)={q0,q1}

13. Give the examples/applications designed as finite state system.

    Text editors and lexical analyzers are designed as finite state systems. A lexical analyzer scans the symbols of a program to locate strings corresponding to identifiers, constants etc, and it has to remember limited amount of information .

14. Define automaton.

    Automaton is a abstact computing device. It is a mathematical model of a system,with discrete inputs, outputs, states and set of transitions from state to state that occurs on input symbols from alphabet Σ.

15. What is the principle of mathematical induction. April/May 2008

    Let P(n) be a ststement about a non negative integer n. Then the principle of mathematical induction is that P(n) follows from

    P(1) and P(n-1) implies P(n) for all n>1.

    Condition(i) is called the basis step and condition (ii) is called the inductive step. P(n-1) is called the induction hypothesis.

16. List any four ways of theorem proving
    Deductive
    If and only if
    Induction
    Proof by contradiction.

17. Define TOC

    TOC describes the basic ideas and models underlying computing. TOC suggests various abstract models of computation, represented mathematically.

18. What are the applications of TOC?
    Compiler Design
    Robotics
    Artificial Intelligence
    Knowledge Engineering.

19. Define a Deterministic Finite Automaton.
    A Determinstic finite automaton consists of :
    A finite set of states, often denoted by Q
    A finite set of input symbols, often denoted by Σ
    A transition function that takes as arguments a state and an input symbol and returns a state.
    A start state, one of the states in Q
    A set of final or accepting states F.

20. Define a Non Deterministic Finite Automaton
    A Non Deterministic Finite Automaton consists of
    A finite set of states, often denoted by Q
    A finite set of input symbols, often denoted by Σ
    A transition function that takes as arguments a state and an input symbol in Σ, and returns a subset of Q.
    A start state, one of the states in Q
    A set of final or accepting states F.

21. Define Transition Diagram.
    Transition Diagram associated with DFA is a directed graph whose vertices correspond to states of DFA, The edges are the transitions from one state to another.

22. What are the properties of Transition Function(δ)
    $\delta(q.\varepsilon)=q$ For all strings w and input symbol a
    $\Delta(q,aw)= \delta(\delta(q.a),w)$
    $\Delta(q,wa)= \delta(\delta(q,w).a)$
    The transition function δ can be extended that operates on states and strings.

23. Lists the operations on Strings.
    a. Length of a string
    b. Empty string
    c. Concatenation of string
    d. Reverse of a string
    e. Power of an alphabet
    f. Kleene closure
    g. Substring
    h. Palindrome

24. Lists the operations on Languages.
    a. Product
    b. Reversal
    c. Power
    d. Kleene star
    e. Kleene plus
    f. Union
    g. Intersection

UNIT II
REGULAR EXPRESSION

In the previous chapter, we have learnt how to represent any language by a machine or model. In this chapter we will discuss how these models can be converted into expressions. The languages accepted by finite automata are easily described by simple expressions, called regular expressions. The regular expression is the very effective way to represent any language.

In this chapter, we will also see how to design the finite automata from given regular expression and vice versa. The regular expressions are mainly to represent the set of regular language. We will also see the language which is not regular. Let us start with the regular expression.

3.1 Definition of Regular Expression

Let Σ be an alphabet which is used to denote the input set. The regular expression over Σ can be defined as follows.

1. ϕ is a regular expression which denotes the empty set.

2. ε is a regular expression and denotes the set $\{\varepsilon\}$.

3. For each '*a*' in Σ '*a*' is a regular expression and denotes the set $\{a\}$.

4. If r and s are regular expressions denoting the languages L_1 and L_2 respectively, then

r+s is equivalent to $L_1 \cup L_2$ i.e. union.

rs is equivalent to $L_1 L_2$ i.e. concatenation

r^* is equivalent to L_1^* i.e. closure.

The r^* is known as kleen closure or closure which indicates occurrence of r for ∞ number of times.

For example if $\Sigma = \{a\}$ and we have regular expression $R = a^*$, then R is a set denoted by $R = \{\varepsilon, a, aa, aaa, aaaa, ...\}$

That is R includes any number of *a*'s as well as empty string which indicates zero number of *a*'s appearing, denoted by ε character.

Similarly there is a positive closure of L which can be shown as L^+. The L^+ denotes set of all the strings except the ε or null string. The null string can be denoted by ε or ^

If $\Sigma = \{a\}$ and if we have regular expression $R = a^+$ then R is a set denoted by

$$R = \{a, aa, aaa, aaaa, ...\}$$

We can construct L^* as

$$L^* = \varepsilon \cdot L^+$$

Let us try to use regular expressions with the help of some examples.

➠ **Example 3.1 :** *Write the regular expression for the language accepting all combinations of a's over the set* $\Sigma = \{a\}$.

Solution : All combinations of *a*'s means *a* may be single, double, tripple and so on. There may be the case that *a* is appearing for zero times, which means a null string. That is we expect the set of {ε, *a*, *aa*, *aaa*, ...}. So we can give regular expression for this as

$$R = a^*$$

That is kleen closure of *a*.

➠ **Example 3.2 :** *Design the regular expression (r.e.) for the language accepting all combinations of a's except the null string over* $\Sigma = \{a\}$

Solution : The regular expression has to be built for the language

$$L = \{a, aa, aaa, ...\}$$

This set indicates that there is no null string. So we can denote r.e. as

$$R = a^+$$

As we know, positive closure indicates the set of strings without a null string.

➠ **Example 3.3 :** *Design regular expression for the language containing all the strings containing any number of a's and b's.*

Solution : The regular expression will be

$$\text{r.e.} = (a + b)^*$$

This will give the set as L = {ε, *a*, *aa*, *ab*, b, b*a*, b*ab*, *abab*, ... any combination of *a* and b} .

The $(a + b)^*$ means any combination with *a* and b even a null string.

TWO MARKS:

1. **What is a regular expression?**

A regular expression is a string that describes the whole set of strings according to certain syntax rules. These expressions are used by many text editors and utilities to search bodies of text for certain patterns etc. Definition is: Let _ be an alphabet. The regular expression over _ and the sets they denote are:

i. _ is a r.e and denotes empty set.

ii. _ is a r.e and denotes the set {_}

iii. For each 'a' in _ , a+ is a r.e and denotes the set {a}.

iv. If 'r' and 's' are r.e denoting the languages R and S respectively then (r+s),

(rs) and (r*) are r.e that denote the sets RUS, RS and R* respectively.

2. **Differentiate L* and L+**

L* denotes Kleene closure and is given by L* =U Li i=0

example : 0* ={_ ,0,00,000,……}

Language includes empty words also.

L+ denotes Positive closure and is given by L+= U Li i=1 q0 q1

3. **What is Arden's Theorem?**

Arden's theorem helps in checking the equivalence of two regular expressions. Let P and Q be the two regular expressions over the input alphabet _. The regular expression R is given as : R=Q+RP Which has a unique solution as R=QP*.

4. **Write a r.e to denote a language L which accepts all the strings which begin or end with either 00 or 11.**

The r.e consists of two parts:

L1=(00+11) (any no of 0's and 1's) =(00+11)(0+1)*

L2=(any no of 0's and 1's)(00+11) =(0+1)*(00+11)

Hence r.e R=L1+L2 =[(00+11)(0+1)*] + [(0+1)* (00+11)]

5. **Construct a r.e for the language over the set _={a,b} in which total number of a's are divisible by 3**

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

6. **What is: (i) (0+1)* (ii)(01)* (iii)(0+1) (iv)(0+1)+**

(0+1)*= { _ , 0 , 1 , 01 , 10 ,001 ,101 ,101001,……}

Any combinations of 0's and 1's.

(01)*={_ , 01 ,0101 ,010101 ,……..}

All combinations with the pattern 01.

(0+1)= 0 or 1,No other possibilities.

(0+1)+= {0,1,01,10,1000,0101,…….}

7. **Reg exp denoting a language over _ ={1} having (i) even length of string (ii) odd length of a string**

(i) Even length of string R=(11)*

(ii) Odd length of the string R=1(11)*

8. **Reg exp for: (i) All strings over {0,1} with the substring '0101' (ii) All strings beginning with '11 ' and ending with 'ab' (iii) Set of all strings over {a,b}with 3 consecutive b's. (iv) Set of all strings that end with '1'and has no substring '00'**

(i)(0+1)* 0101(0+1)*

(ii)11(1+a+b)* ab

(iii)(a+b)* bbb (a+b)*

(iv)(1+01)* (10+11)* 1

9. **Construct a r.e for the language which accepts all strings with atleast two c's over the set Σ={c,b}**

(b+c)* c (b+c)* c (b+c)*

10. **What are the applications of Regular expressions and Finite automata Lexical analyzers and Text editors are two applications.**

Lexical analyzers:

The tokens of the programming language can be expressed using regular expressions. The lexical analyzer scans the input program and separates the tokens.For eg identifier can be expressed as a regular expression as: (letter)(letter+digit)*

If anything in the source language matches with this reg exp then it is recognized as an identifier.The letter is{A,B,C,……..Z,a,b,c….z} and digit is {0,1,…9}.Thus reg exp identifies token in a language.

Text editors:

These are programs used for processing the text. For example UNIX text editors uses the reg exp for substituting the strings such as: S/bbb*/b/

Gives the substitute a single blank for the first string of two or more blanks in a given line. In UNIX text editors any reg exp is converted to an NFA with Єtransitions, this NFA can be then simulated directly.

11. .Reg exp for the language that accepts all strings in which 'a' appears tripled overthe set Σ ={a}

reg exp=(aaa)*

12. .What are the applications of pumping lemma?

Pumping lemma is used to check if a language is regular or not.

(i) Assume that the language(L) is regular.
(ii) Select a constant 'n'.
(iii) Select a string(z) in L, such that |z|>n.
(iv) Split the word z into u,v and w such that |uv|<=n and |v|>=1.
(v) You achieve a contradiction to pumping lemma that there exists an 'i' Such that uvi w is not in L.Then L is not a regular language.

13. What is the closure property of regular sets?

The regular sets are closed under union, concatenation and Kleene closure.

r1Ur2= r1 +r2

r1.r2= r1r2

(r)*=r*

The class of regular sets are closed under complementation, substitution, homomorphism and inverse homomorphism.

14. .Reg exp for the language such that every string will have atleast one 'a' followed by atleast one 'b'.

$R=a^+b^+$

15. Write the exp for the language starting with and has no consecutive b's .

reg exp=(a+ab)*

16. Construct a regular expression denoting odd numbers in their binary representation

$\{0/1\}^*1$

17. Construct a regular expression denoting even numbers in their binary representation

$\{0/1\}^*0$

18. Construct a regular expression denoting the set of all strings over {a,b} such that all starts with a and ends with b

$a\{a/b\}^*b$

19. Construct a regular expression denoting the set of all strings over {a,b} such that all starts with a and ends with ab

$a\{a/b\}^*ab$

20. Construct a regular expression denoting the set of all strings over {a,b} such that all ends with abb

$\{a/b\}^*abb$

UNIT- III CONTEXT-FREE GRAMMAR AND LANGUAGES

Context-Free Grammar (CFG) – Parse Trees – Ambiguity in grammars and languages – Definition of the Pushdown automata – Languages of a Pushdown Automata – Equivalence of Pushdown automata and CFG, Deterministic Pushdown Automata.

Grammar

A grammar is a mechanism used for describing languages. This is one of the most simple but yet powerful mechanism. There are other notions to do the same, of course. In everyday language, like English, we have a set of symbols (alphabet), a set of words constructed from these symbols, and a set of rules using which we can group the words to construct meaningful sentences. The grammar for English tells us what are the words in it and the rules to construct sentences. It also tells us whether a particular sentence is well-formed (as per the grammar) or not. But even if one follows the rules of the english grammar it may lead to some sentences which are not meaningful at all, because of impreciseness and ambiguities involved in the language. In english grammar we use many other higher level constructs like noun-phrase, verb-phrase, article, noun, predicate, verb etc. A typical rule can be defined as

<sentence>→< noun-phrase >< predicate >

meaning that "a sentence can be constructed using a 'noun-phrase' followed by a predicate". Some more rules are as follows:

<noun-phrase>→< article >< noun >

<predicate>→< verb >

with similar kind of interpretation given above. If we take {a, an, the} to be <article>; cow, bird, boy, Ram, pen to be examples of <noun>; and eats, runs, swims, walks, are associated with <verb>, then we can construct the sentence- a cow runs, the boy eats, an pen walks- using the above rules. Even though all sentences are well-formed, the last one is not meaningful. We observe that we start with the higher level construct <sentence> and then reduce it to <noun-phrase>, <article>, <noun>, <verb> successively, eventually leading to a group of words associated with these constructs.

These concepts are generalized in formal language leading to formal grammars. The word 'formal' here refers to the fact that the specified rules for the language are explicitly stated in terms of what strings or symbols can occur. There can be no ambiguity in it.

FORMAL DEFINITIONS OF A GRAMMAR

A grammar *G* is defined as a quadruple.

G={N,∑,P,S)

N is a non-empty finite set of non-terminals or variables,

∑ is a non-empty finite set of terminal symbols such that N ∩∑=Φ

S□ N, is a special non-terminal (or variable) called the start symbol, and P□ (N U ∑)*

x(N U ∑)* is a finite set of production rules.

The binary relation defined by the set of production rules is denoted by →, i.e.α→βiff(α,β)□ P. In other words, P is a finite set of production rules of the form .α→β, where and α□ (N U ∑)* and β□ (N U ∑)*

PRODUCTION RULES:

The production rules specify how the grammar transforms one string to another. Given a string □αy, we say that the production rule α→β is applicable to this string, since it is possible to use the rule α→β to rewrite the α(in □αy) to β obtaining a new string □βy. We say that □αy derives □βy and is denoted as

□αy==> □βy

Successive strings are dervied by applying the productions rules of the grammar in any arbitrary order. A particular rule can be used if it is applicable, and it can be applied as many times as described. We write α==>β if the string βcan be derived from the string α in zero or more steps; α==>β if β can be derived from α in one or more steps.

By applying the production rules in arbitrary order, any given grammar can generate many strings of terminal symbols starting with the special start symbol, *S*, of the grammar. The set of all such terminal strings is called the language generated (or defined) by the grammar. Formaly, for a given grammar G={N,∑,P,S) the language generated by *G* is

L(G)={ w□ ∑*|s => w}

That is w □ L(G) iffs=>w.Ifw □ L(G) , we must have for some n≥0, S=α1=>a2=>a3=>….=>a n=w, denoted as aderivation sequence of *w*, The strings S= α 1, α 2, α 3,…, α n=w are denoted as sentential forms of the derivation.

Example : Consider the grammar G={N,∑,P,S), where *N* = {*S*}, ∑={*a, b*} and P is the set of the followingproduction rules

{ *S→ab, S→aSb*}

Some terminal strings generated by this grammar together with their derivation is given below.

S =>ab

S =>aSb=>aabb

S =>aSb=>aaSbb=>aaabbb

It is easy to prove that the language generated by this grammar is

L(G)={a□ □b□| i≥1}

By using the first production, it generates the string ab(for*i* =1).

To generate any other string, it needs to start with the production *S→aSb*and then the non-terminal *S* in theRHS can be replaced either by *ab* (*in which we get the string aabb*) or the same production *S→aSb*can be *used one or more times. Every time it adds an 'a' to the left and a 'b' to the right of S, thus giving the sentential* forma□Sb□,i≥1. When the non-terminal is replaced by *ab* (which is then only possibility for generating a terminal string) we get a terminal string of the form a□Sb□,i≥1.

There is no general rule for finding a grammar for a given language. For many languages we can devise grammars and there are many languages for which we cannot find any grammar.

Example: Find a grammar for the language L={$a^n b^{n+1}$|n≥1}.

It is possible to find a grammar for L by modifying the previous grammar since we need to generate an extra *b* at the end of the string . We can do this by adding a production *S→Bb* where the non-terminal *B generates* a□Sb□,i≥1 as given in the previous example.

Using the above concept we devise the follwoing grammar for *L*.
G={N,∑,P,S),where, *N* = { *S, B* }, *P* = { *S-->Bb, B→ab, B→aBb*}

PARSE TREES:

There is a tree representation for derivations that has proved extremely usetul. This tree shows us clearly how the symbols of a terminal string are grouped into substrings, each of which belongs to the language of one of the variables of the grammar. But perhaps more importantly, the tree, known as a "parse tree"

CONSTRUCTION OF PARSER TREE:

Let us fix on a grammar $G = (V, T, P, S)$. The *parse trees* for G are trees with the following conditions:

1. Each interior node is labeled by a variable in V.
2. Each leaf is labeled by either a variable, a terminal, or ϵ. However, if the leaf is labeled ϵ, then it must be the only child of its parent.
3. If an interior node is labeled A, and its children are labeled

$$X_1, X_2, \ldots, X_k$$

respectively, from the left, then $A \to X_1 X_2 \cdots X_k$ is a production in P. Note that the only time one of the X's can be ϵ is if that is the label of the only child, and $A \to \epsilon$ is a production of G.

Example 5.10: Figure 5.5 shows a parse tree for the palindrome grammar of Fig. 5.1. The production used at the root is $P \to 0P0$, and at the middle child of the root it is $P \to 1P1$. Note that at the bottom is a use of the production $P \to \epsilon$. That use, where the node labeled by the head has one child, labeled ϵ, is the only time that a node labeled ϵ can appear in a parse tree. □

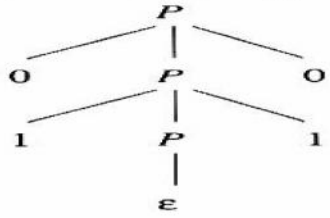


Figure 5.5: A parse tree showing the derivation $P \Rightarrow^{*} 0110$

If we look at the leaves of any parse tree and concatenate them from the left, we get a string, called the *yield* of the tree, which is always a string that is derived from the root variable. The fact that the yield is derived from the root will be proved shortly. Of special importance are those parse trees such that:

1. The yield is a terminal string. That is, all leaves are labeled either with a terminal or with ϵ.
2. The root is labeled by the start symbol.

When a grammar fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. Unfortunately, sometimes we cannot do so. That is, there are some CFL's that are "inherently ambiguous"; every grammar for the language puts more than one structure on some strings in the language.

grammar lets us generate expressions with any sequence of $*$ and $+$ operators, and the productions $E \to E + E \mid E * E$ allow us to generate these expressions in any order we choose.

Example 5.25: For instance, consider the sentential form $E + E * E$. It has two derivations from E:

1. $E \Rightarrow E + E \Rightarrow E + E * E$
2. $E \Rightarrow E * E \Rightarrow E + E * E$

Notice that in derivation (1), the second E is replaced by $E * E$, while in derivation (2), the first E is replaced by $E + E$. Figure 5.17 shows the two parse trees, which we should note are distinct trees.

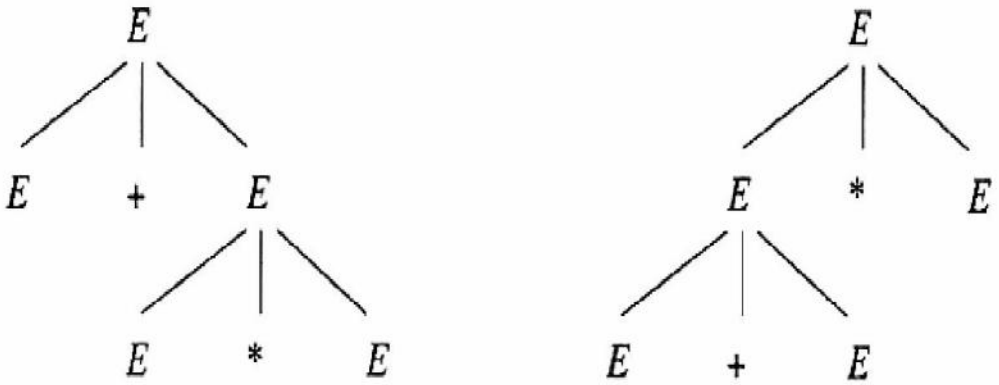
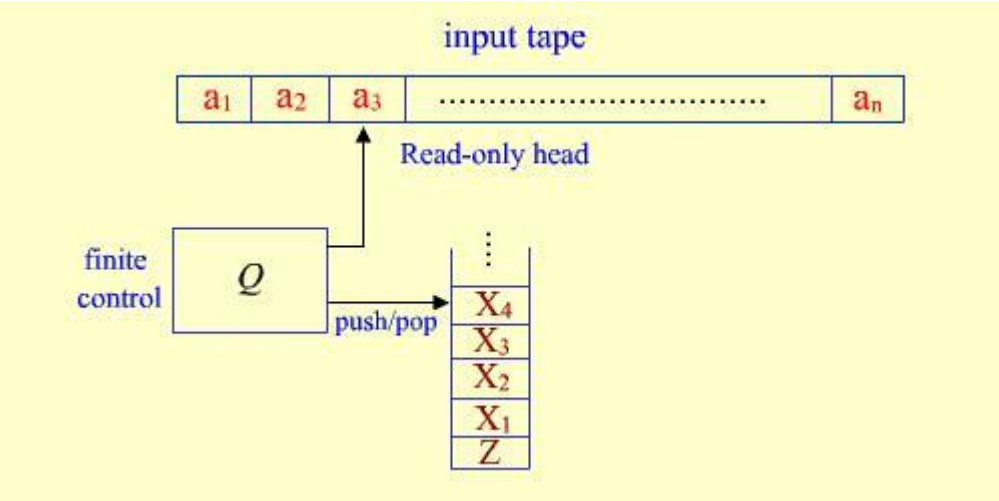


Figure 5.17: Two parse trees with the same yield

we say a CFG $G = (V, T, P, S)$ is *ambiguous* if there is at least one string w in T^* for which we can find two different parse trees, each with root labeled S and yield w. If each string has at most one parse tree in the grammar, then the grammar is *unambiguous*.

PUSH DOWN AUTOMATA:

Regular language can be charaterized as the language accepted by finite automata. Similarly, we can characterize the context-free language as the langauge accepted by a class of machines called "PushdownAutomata" (PDA). A pushdown automation is an extension of the NFA. It is observed that FA have limited capability. (in the sense that the class of languages accepted or characterized by them is small). This is due to the "finite memory" (number of states) and "no external memory" involved with them. A PDA is simply an NFA augmented with an "external stack memory". The addition of a stack provides the PDA with a last-in, first-out memory management cpapability. This "Stack" or "pushdownstore" can be used to record a potentially unbounded information. It is due to this memory management capability with the help of the stack that a PDA can overcome the memory limitations that prevents a FA toaccept many interesting languages like $\{a^n b^n | n \geq 0\}$. Although, a PDA can store an unbounded amount of information on the stack, its access to the information on the stack is limited. It can push an element onto the top of the stack and pop off an element from the top of the stack. To read down into the stack the top elements must be popped off and are lost. Due to this limited access to the information on the stack, a PDA still has some limitations and cannot accept some other interesting languages.



As shown in figure, a PDA has three components: an input tape with read only head, a finite control and a pushdown store. The input head is read-only and may only move from left to right, one symbol (or cell) at a time. In each step, the PDA pops the top symbol off the stack; based on this symbol, the input symbol it is currently reading, andits present state, it can push a sequence of symbols onto the stack, move its read-only head one cell (or symbol) to the right, and enter a new state, as defined by the transition rules of the PDA.

PDA are nondeterministic, by default. That is, □- transitions are also allowed in which the PDA can pop and push, and change state without reading the next input symbol or moving its read-only head. Besides this, there may be multiple options for possible next moves.

Formal Definitions : Formally, a PDA*M*is a 7-tuple*M* =*(Q,∑, ┌, ,q0,z0,F)*where,

- Qis a finite set of states,
- ∑is a finite set of input symbols (input alphabets),
- ┌is a finite set of stack symbols (stack alphabets),
- □ is a transition function from Q□ (∑ U {□ })□ to subset of Q□ ┌*
- q0□ Qis the start state
- z0□ ┌, is the initial stack symbol, and
- F*□ Q, is the final or accept states.

Explanation of the transition function, □:

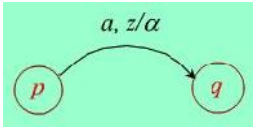
If, for any a□ ∑, □ (q,a,z)={(p1β1), (p2,β2)…..(pk,βk). This means intitutively that whenever the PDA is in state *q* reading input symbol *a* and *z* on top of the stack, it can nondeterministically for any $i \leq t \leq k$

- go to state pi
- pop *z* off the stack
- push βionto the stack (where β1□ ┌*) (The usual convention is that if βi=x1x2….xn, then x1 will be at the top and xn at the bottom.)
- move read head right one cell past the current symbol *a*.

If *a* = □, then □ (q,□ ,z)={(p1β1),(p2,β2),… (pk,βk)],(means intitutively that whenver the PDA is in state *q* with *z* on the top of the stack regardless of the current input symbol, it can nondeterministically for any *i*, $i \leq t \leq k$,

- go to state Pi
- pop *z* off the stack
- push β1 onto the stack, and
- leave its read-only head where it is.

State transition diagram :A PDA can also be depicted by a state transition diagram. The labels on the arcsindicate both the input and the stack operation. The transition □ (p,a,z)={(qα)}for a□ ∑ U {□ } ,p,q □ Q,z □ and α □ ┌* is depicted by



Final states are indicated by double circles and the start state is indicated by an arrow to it from nowhere.

CONFIGURATION OR INSTANTANEOUS DESCRIPTION (ID) :

A configuration or an instantaneous description (ID) of PDA at any moment during its computation is an
element of Q□ ∑*□ ┌*describing the current state, the portion of the input remaining to be

read (i.e. under and to the right of the read head), and the current stack contents. Only these three elements can affect the computation from that point on and, hence, are parts of the ID.

The start or initalconfiguartion (or ID) on input ωis(q0,ω,z0) . That is, the PDA always starts in its start state, q0with its read head pointing to the leftmost input symbol and the stack containing only the start/initial stack symbol,z0. The "next move relation" one figure describes how the PDA can move from one configuration to another in one step.

Formally, (q,aω,z) M(p,ω,βα)

Iff (p,β) (q,a,z)
'*a*' may be or an input symbol.

Let *I, J, K* be IDs of a PDA. We define we write $I \vdash^{*} M,K$, if ID *I* can become *K* after exactly*i*moves. Therelations ⊢M and ⊢M define as follows

I ⊢*K*

I ⊢*J* if such that *I* ⊢*K* and *K* ⊢ *J*

I ⊢*J* if such that *I* ⊢*J*.
That is, ⊢Mis the reflexive, transitive closure of ⊢N. We say that *I* ⊢*NJ* if the ID *J* follows from the ID *I* inzero or more moves.

(Note**:** subscript *M* can be dropped when the particular PDA *M* is understood.)

Language accepted by a PDA *M*

There are two alternative definiton of acceptance as given below.

1. Acceptance by final state**:**

Consider the PDA *M =(Q,∑, ┌, ,q0,z0,F)* . Informally, the PDA *M* is said to accept its input ωby final state if it enters any final state in zero or more moves after reading its entire input, starting in the start configuration on input ω.

Formally, we define *L*(*M*), the language accepted by final state to be

{ ω ∑*| (q0,ω,z0)|- N (pβ)for some p F and β ┌*}

2. Acceptance by empty stack (or Null stack)**:** The PDA *M*accepts its input ω by empty stack if starting in the start configuration on inputω, it ever empties the stack w/o pushing anything back on after reading the entire input. Formally, we define *N*(*M*), the language accepted by empty stack, to be

{ ω ∑*| (q0,ω,z0)|- N (p f)for some p Q}

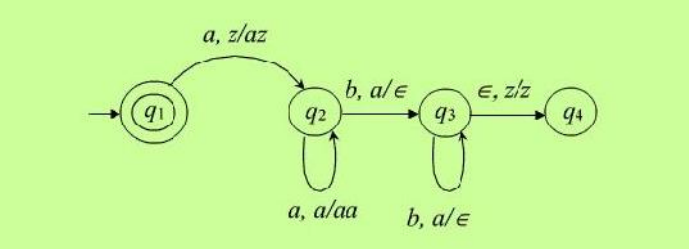
Note that the set of final states, *F* is irrelevant in this case and we usually let the *F* to be the empty set i.e. *F = Q* .

Example 1 : Here is a PDA that accepts the language $\{a^n b^n | n \geq 0\}$.

M =(Q,∑, ┌, ,q0,z0,F)
Q={q1,q2,q3.q4}
∑={a,b}
┌={a,b,z}
F={q1,q4} , and □ consists of the following transitions

1. ∈(q1,a,z)={{q2.az)}
2. ∈ (q2,a,a)={{q2.aa)}
3. ∈ (q2,b,a)={{q3.□)}
4. ∈ (q3,b,a)={{q3.□)}
5. ∈ (q3,□ ,z)={{q4,z)}

The PDA can also be described by the adjacent transition diagram.



Informally, whenever the PDA *M* sees an input *a* in the start state q1with the start symbol *z* on the top of the stack it pushes *a* onto the stack and changes state to q2. (to remember that it has seen the first '*a*'). On state q2if it sees anymore *a*, it simply pushes it onto the stack. Note that when *M* is on stateq2, the symbol on the top of the stack can only be *a*. On state q2 if it sees the first *b* with *a* on the top of the stack, then it needs to start comparison of numbers of *a*'s and *b*'s, since all the *a*'s at the begining of the input have already been pushed onto the stack. It start this process by popping off the a from the top of the stack and enters in state q3 (to remember that the comparison process has begun). On stateq3 , it expects only b's in the input (if it sees any more a in the input thus the input will not be in the proper form of anbn).

Hence there is no more on input a when it is in state q3. On state q3it pops off an a from the top of the stack for every b in the input. When it sees the last b on state q3 (i.e. when the input is exaushted), then the last a from the stack will be popped off and the start symbol z is exposed. This is the only possible case when the input (i.e. on □-input) the PDA *M* will move to state q4 which is an accept state. We can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

Let the input be aabb. we start with the start configuration and proceed to the subsequent IDs using the transition function defined

(q1,aabb,z) |-- (q2,aabb,z) (using transition 1)

|-(q1,bb,zz) (using transition 2)
|-(q3,b,z) (using transition 3)
|-(q3,□ ,z)(using transition 4), |-(q4,□ ,z)(using transition 5) , q4is final state. Hence , accept.
So the string *aabb* is rightly accepted by *M*

we can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

i) Let the input be *aabab*.

(q1,aabab,z) |-- (q2,abab,z)
|-(q2,bab,aaz)
|-(q3,ab,az)

No further move is defined at this point. Hence the PDA gets stuck and the string *aabab* is not accepted.

Example 2 : We give an example of a PDA*M*that accepts the set of balanced strings of parentheses [] byempty stack.
The PDA *M* is given below.

M =({q}{[,]},{z,[}, ,q,z,Φ) where □is defined as

□ (q,[,z)={(q,[z)}
□ (q,[,])={(q,[[)}
□ (q,[,])={(q,□)}
□ (q,□ ,z)={(q,□)}

Informally, whenever it sees a [, it will push the] onto the stack. (first two transitions), and whenever it sees a] and the top of the stack symbol is [, it will pop the symbol [off the stack. (The third transition). The fourth transition is used when the input is exhausted in order to pop z off the stack (to empty the stack) and accept. Note that there is only one state and no final state. The following is a sequence of configurations leading to the acceptance of the string [[ ] [ ]] [ ].

(q,[[][]][],z) |- (q,[][]][],[z) |-(q,[]][],[z) |-(q,]][][[z)

|-(q,]][],[[z) |-(q,][],[z)|-(q,[],z) |-(q,],[z) |-(q,□ ,z)

Equivalence of acceptance by final state and empty stack.

It turns out that the two definitions of acceptance of a language by a PDA - accpetance by final state and empty stack- are equivalent in the sense that if a language can be accepted by empty stack by some PDA, it can also be accepted by final state by some other PDA and vice versa. Hence it doesn't matter which one we use, sinceeach kind of machine can simulate the other.Given any arbitrary PDA *M* that accpets the language *L* by final state or empty stack, we can always construct an equivalent PDA *M* with a single final state that accpets exactly the same language *L*. The construction process of *M'* from *M* and the proof of equivalence of *M*&*M'* are given below.

There are two cases to be considered.

CASE I : PDA*M*accepts by final state, Let *M =(Q,∑, ┌, ,q0,z0,F)* Let *q*f be a new state not in *Q*. Consider the PDA *M'=(Q U{q1}∑., ,q0,z0,{q}*)where □ ' as well as the following transition.

□ '(q□ ,X)contains (q1,X)□ q□ Fand X□ ┌. It is easy to show that *M* and *M'* are equivalent i.e.

L(*M*) = *L*(M')

Let ω□*L*(*M*). Then (q0,ω,z0)|- M(q,□ ɣ)for some q□ Fandɣ□ ┌*

Then (q0,ω,z0)|- M*(q,□ ɣ)|-M*(q,□ ɣ)|

Thus M' accepts ω

Conversely, let M'acceptsωi.e. ω□*L*(M'), then(q0,ω,z0)|- M*(q,□ ɣ)|-M*(q,□ ɣ)| for q□ F M' inherits all other moves except the last one from *M*. Hence (q0,ω,z0)|- M*(q,□ ɣ)|-for some

q□ F.

Thus *M* accepts ω. Informally, on any input M' simulate all the moves of *M* and enters in its own final state q1whenever *M* enters in any one of its final status in *F*. Thus M'accepts a string ω iff*M* accepts it.

CASE II :PDA*M*accepts by empty stack.

We will construct M'from *M* in such a way that M' simulates *M* and detects when *M* empties its stack.

M' enters its final state q1 when and only when *M* empties its stack.ThusM'will accept a string ω iff*M* accepts.

Let M'=(Q U{q0'q1},∑, ┌ U {X},□ ',q0',X,{q1})where q0',q1 not □ Q and *X* □ ┌and □ contains all the transition of □, as well as the following two transitions.

1.□ '(q0,□ ,X)={(q0,z0,X)}and
2. □ '(q0,□ ,X)={(q1,□)} ,□ q□ Q

Transitions 1 causes M' to enter the initial configuration of *M* except that M'will have its own bottom-of-stack marker *X* which is below the symbols of *M*'s stack. From this point onward M' will simulate every move of *M* since all the transitions of *M* are also in M'

If *M* ever empties its stack, then M'when simulating *M* will empty its stack except the symbol *X* at the bottom.

At this point, M' will enter its final state q1by using transition rule 2, thereby (correctly) accepting the input. We will prove that *M*and M'are equivalent.

Let *M* accepts ω. Then

(q0,ω,zo) |- M* (q,□ ,□) for some (q □ Q). But then
(q0,ω,X)|- M'¹(q0,ω,z0,X) (by transition rule 1)
|- M*(q1,□ ,X) (Since M' includes all the moves of *M*)
|-- M¹ |-(q1,□ ,□) (by transition rule 2)

Hence, M' also accepts ω. Conversely, let M' accepts ω.
Then (q0',ω,X) |- M*(q0',ω,zoX) |- M* (q0',□ ,X) ⊦ M* (q0',□ ,□) for some q □ Q
Every move in the sequence, (q0',ω,zoX) |- M* (q0',□ ,X) were taken from *M*.
Hence, *M* starting with its initial configuration will eventually empty its stack and accept the input i.e.

(q0,ω,z0)|- M* (q0,□ ,□)

EQUIVALENCE OF PDA'S AND CFG'S:

We will now show that pushdown automata and context-free grammars are equivalent in expressive power, that is, the language accepted by PDAs are exactly the context-free languages. To show this, we have to prove each of the following:

i) Given any arbitrary CFG *G* there exists some PDA *M* that accepts exactly the same language generated by *G*.
ii) Given any arbitrary PDA *M* there exists a CFG *G* that generates exactly the same language accpeted by *M*.

(i) CFA to PDA

We will first prove that the first part i.e. we want to show to convert a given CFG to an equivalent PDA.
Let the given CFG is G={N,∑,P,S). Without loss of generality we can assume that *G* is in Greibach Normal Form i.e. all productions of *G* are of the form .

A→cB1B2…Bn where c□ ∑ U{□ } and k≥0.

From the given CFG *G* we now construct an equivalent PDA *M* that accepts by empty stack. Note that there is only one state in *M*. Let
M={{q},∑,N,□ ,q,S,Φ) where

- *q* is the only state
- ∑ is the input alphabet,
- *N* is the stack alphabet ,
- *q* is the start state.
- *S* is the start/initial stack symbol, and □, the transition relation is defined as follows

For each production A→cB1B2…Bk □ P, (q,B1,B2,…Bk)□ □ (q,cA). We now want to show that *M* and *G* are equivalent i.e. *L*(*G*)=*N*(*M*). i.e. for any w□ ∑*, w□ L(G) iff w□ N(M)
If w□ L(G), then by definition of *L*(*G*), there must be a leftmost derivation starting with *S* and deriving *w*.

i.e. S=>w

Again if w□ N(M), then one sysmbol. Therefore we need to show that for any w□ ∑*S=>w iff (q,w,s) |- (q,□ ,□).

But we will prove a more general result as given in the following lemma. Replacing *A* by *S*

(the start symbol) and γ by gives the required proof.
Lemma For anyx,y ∑*, γ N* and A N, A=>xy via a leftmost derivative iff
(q,xy,A)|- M*(q,y,γ)
Proof : The proof is by induction on*n*.
Basis :*n*= 0
A=>xyiffA=xy i.e. x= and y=A
Iff (q,xy,A)=(q,y,γ)
iff(q,xy,a) |- M*(q,y,γ)
Induction Step :
First, assume that A=>xy via a leftmost derivation. Let the last production applied in their derivation is B→cβ for some c ∑U{ }and β N*.
Then, for some ω ∑*, α N*

A=>ωBα=>ωc βα=xy

Where x=ωc and y=βα
Now by the indirection hypothesis, we get,

(q,ωy,A)|- M*(q,cy,Bα).....(1)
Again by the construction of *M*, we get
(q,β) (q,c,B)
so, from (1), we get
(q,ωy,A)|- M*(q,cy,Bα) |- M' *(q,y,Bα)
since x=ωc and y=βα, we get (q,ωy,A)|- M*(q,y,γ)
That is, ifA=>xy , then(q,ωy,A)|- M*(q,y,γ) . Conversely, assume that
(q,ωy,A)|- M*(q,y,γ) and let
(q,c,β)=(q,β) be the transition used in the last move. Then for some ω ∑*, c ∑ U { }and α Γ*
(q,ωy,A)|- M*(q,cy,Bα) |- M' *(q,y,Bα) where x=ωc and y=βα,

Now, by the induction hypothesis, we get A=>ωBα via a leftmost derivation. Again, by the construction of *M*, B->cβ must be a production of *G*. [Since(q,β) (q,c,B)]. Applying the production to the sentential form ωBα kwe get
A=>ωβα=>ωc βα=xy
i.e. A=>xy
via a leftmost derivation.
Hence the proof.

Example : Consider the CFG*G*in GNF

S->aAB
A→a / aA
B→a / bB

The one state PDA *M* equivalent to *G* is shown below. For convenience, a production of *G* and the corresponding transition in *M* are marked by the same encircled number.

(1) *S →aAB*

(2) $A \rightarrow a$
(3) $A \rightarrow aA$
(4) $B \rightarrow a$
(5) $B \rightarrow bB$

M={{q},{a,b},{S,A,B},∑,□ ,q,S)We have used the same construction discussed earlier

Some Useful Explanations :

Consider the moves of M on input *aaaba* leading to acceptance of the string.

1.(*q*,*aaaba*,*s*) →(*q*, *aaba*, *AB*)

2. →(*q*, *aba*, *AB*)

3. →(*q*, *ba*, *B*)

4. →(*q*, *a*, *B*)

5. →(*q*, a , B) Accept by empty stack.

Note :encircled numbers here shows the transitions rule applied at every step.

Now consider the derivation of the same string under grammar G. Once again, the production used at every step is shown with encircled number.

S =>aAB=>aaAB=>aaaB=>aaabB=>aaaba

Steps 1 2 3 4 5

Observations:

- There is an one-to-one correspondence of the sequence of moves of the PDA *M* and the derivation sequence under the CFG *G* for the same input string in the sense that - number of steps in both the cases are same and transition rule corresponding to the same production is used at every step (as shown by encircled number).
- considering the moves of the PDA and derivation under *G* together, it is also observed that at every step the input read so far and the stack content together is exactly identical to the corresponding sentential form i.e.

<what is Read><stack> = <sentential form>

Say, at step 2, Read so far
= *a* stack = *AB*

Sentential form = *aAB* From this property we claim that (q,x,S)|- M*(q,□ α)iff S=>xα. If the claim is true, then apply with α=□and we get (q,x,S)|- M*(q,□ ,□)iffS=>x or x□ N(M)iffx□ L(G)(by definition)

Thus *N*(*M*) = *L*(*G*) as desired. Note that we have already proved a more general version of the claim PDA and CFG:
We now want to show that for every PDA *M* that accpets by empty stack, there is a CFG *G* such

that $L(G) = N(M)$ we first see whether the "reverse of the construction" that was used in part (i) can be used here to construct an equivalent CFG from any PDA *M*.

It can be show that this reverse construction works only for single state PDAs.

- That is, for every one-state PDA *M* there is CFG *G* such that $L(G) = N(M)$. For every move of the PDA *M* (q,B1,B2,…Bk)we introduce a production A→ cB1,B2,…Bkin the grammar G=(N,∑,P,S) where *N* = *T* and S=z0.

we can now apply the proof in part (i) in the reverse direction to show that $L(G) = N(M)$.
But the reverse construction does not work for PDAs with more than one state. For example, consider the PDA *M* produced here to accept the langauge$\{a^n b a^n | n \geq 1\}$

M=({p,q},{a,b},{z0,A},□ ,p,z0,Φ)

Now let us construct CFG G=(N,∑,P,S) using the "reverse" construction.
(NoteN={z0,A},S=z0).

	Transitions in *M*	Corresponding Production in *G*
a,z0/A	z0→aA	
a,A/AA	A→aAA	
b,A/A		A→bA
a,A/□		A→a

We can drive strings like aabaa which is in the language.

s=>z0=>aA=>aaAA=>aabAA=>aabaA=>aabaa

But under this grammar we can also derive some strings which are not in the language. e.g

s=>z0=>aA=>aaAA=>aabAA=>aabaA=>aabaa

ands=>z0=>aA=>aa But aa,not□ L(M)

Therefore, to complete the proof of part (ii) we need to prove the following claim also.

Claim: For every PDA *M* there is some one-state PDA M'such that N(M)=N(M').
It is quite possible to prove the above claim. But here we will adopt a different approach. We start with any arbitrary PDA *M* that accepts by empty stack and directly construct an equivalent CFG *G*.

PDA to CFG

We want to construct a CFG *G* to simulate any arbitrary PDA *M* with one or more states. Without loss of generality we can assume that the PDA *M* accepts by empty stack. The idea is to use nonterminal of the form <*PAq*> whenever PDA *M* in state *P* with *A* on top of the stack goes to state q0. That is, for example, for a given transition of the PDA corresponding production in the grammar as shown below, And, we would like to show, in general, that <PAq>=>ω iff the PDA *M*, when started from state *P* with *A* on the top of the stack will finish processing ω, arrive at state *q* and remove *A* from the stack. we are now ready to give the construction of an equivalent CFG *G* from a given PDA *M*. we need to introduce two kinds of producitons in the grammar as given below. The reason for introduction of the first kind of production will be justified at a later point. Introduction of the second type of production has been justified in the above discussion.

Let M={Q, ┌,∑,N,□ ,q,S,Φ) be a PDA. We construct from M a equivalent CFG G=(N,∑,P,S) Where

- *N* is the set of nonterminals of the form<*PAq*>for p,q □ Q and A□ ┌ and *P* contains the following two kind of production

1.S→<q0z0q>□ q□ Q
2. If (q1,B1B2…..Bn)□ □ (q,a,A) then for every choice of the sequence q2,q3,…qn+1, q1□ Q, 2≤i≤n+1.

Include the follwoing production
<qn,qn+1>→a<q1B1q2><q2B2q3>……<qnBnqn>

If *n* = 0, then the production is <qA q1>→a.For the whole exercise to be meaningful we want <qA q1> =>ωmeans there is a sequence of transitions (for PDA *M*), starting in state *q*, ending in qn+1,during which the PDA *M* consumes the input string ω and removes *A* from the stack (and, of course, all other symbols pushed onto stack in *A*'s place, and so on.)

That is we want to claim that

<pAq> => ω iff (p,ω,A) |-(q,□ ,□)

If this claim is true, then let p=q0,A=z0 to get <q0z0q>iff (q0,ω,z0) |-(q,□ ,□)for someq□ Q. But for allq□ Q we have S→<q0z0q>as production in *G*. Therefore, S=><q0z0q>iff=>(q0,ω,z0)|-(q,□ ,□),i.e. S=>w iff PDA *M* accepts w by empty stack or *L*(*G*) = *N*(*M*). Now, to show that the above construction of CFG *G* from any PDA *M* works, we need to prove the proposed claim.

Note: At this point, the justification for introduction of the first type of production (of the form s→<q0z0q>) in the CFG *G*, is quite clear. This helps use deriving a string from the start symbol of the grammar.

Proof : Of the claim <PAq>=>w iff (P,ω,A)|-(q,□ ,□)for some w□ ∑*, A□ ┌and P,q□ Q .The proof is by induction on the number of steps in a derivation of *G* (which of course is equal to the number of moves taken by *M*). Let the number of steps taken is *n*.

The proof consists of two parts: ' *if* ' part and ' *only if* ' part. First, consider the ' *if* ' part

If (P,ω,A)|-(q,□ ,□)then <PAq>=>w.

Basis is *n* =1

Then(P,ω,A)|(q,□ ,□) . In this case, it is clear that w□ ∑ U{□ '} Hence, by construction <PAq>=>w.
is a production of *G*.

Then

Inductive Hypothesis :

∨ **i<n** (P,ω,A)|(q,□ ,□)=><PAq>=>w
Inductive Step :(P,ω,A)|(q,□ ,□)

For *n*>1, let *w* = *ax* for some a□ ∑ U {□ }and x□ ∑* consider the first move of the PDA *M* which uses the general transition (q1,B1B2….Bn) |-(q,□ ,□ ∌

(p,ax,A) |-(q1,x, B1B2….Bn)|-(q,□ ,□ .)Now *M* must remove q1,B1B2….Bnfrom stack while consuming *x* in the remaining *n*-1 moves.

Letx=x1x2…xn, where x1x2…xnis the prefix of *x* that *M* has consumed when Bi+1 first appears at top of the stack. Then there must exist a sequence of states in *M* (as per construction) q2,q3,…qn,qn+1(withqn+1=p), such that

(p,ax,A) |-(q1,x,B1B2…Bn)=(q1,x1x2….xn, B1B2…Bn)
(q2,x2x3….xn, B2B3…Bn) [this step implies (q1,x1,B1) |-(q2,□ ,□)
(q3,x3x4….xn, B3B4…Bn) [this step implies (q2,x2,B2) |-(q3,□ ,□)]
…….
|- (qn,xn,Bn)=|-(q,□ ,□)]

[Note: Each step takes less than or equal to *n* -1 moves because the total number of moves required assumed to be *n*-1.]

That is, in general

(qi,xi,Bi)|-(qi+1,□ ,□), 1≤i≤ n+1

So, applying inductive hypothesis we get

<qiBiqi+1> => xi, 1≤i≤ n+1. But corresponding to the original move

(p,w,A)= (p,ax,A)|-(q1,x,B1B2…Bn)in *M* we have added the following production in *G*.

We can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

i) Let the input be *aabb*. we start with the start configuration and proceed to the subsequent IDs using the transition function defined

(q1,aabb,z)|- (q2,abb,az)(using transition 1) ,|- (q2,abb,az) (using transition 2)

|- (q3,b,az)(using transition 3), (q3,□ ,z)(using transition 4)

|-(q4,□ ,z)(using transition 5) , q4 is final state. Hence, accept.

So the string *aabb* is rightly accepted by *M*.

we can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

i) Let the input be *aabab*

(q1,aabb,z)|- (q2,abb,az)
|- (q2,bab,aaz)
|-(q3,ab.az)

No further move is defined at this point.

Hence the PDA gets stuck and the string *aabab* is not accepted.

The following is a sequence of configurations leading to the acceptance of the string [[ ] [ ]] [ ].
(q,[[][][],z)|-(q,[][]],[z) |- (q,][]][],[[z)
|-(q,[]][],[z)|-(q,]][],[z) |- (q,][],[z)
|-(q,[],z) |-(q,].[z) |-(q,□ ,z) ⊦ (q, □ , □)

Equivalence of acceptance by final state and empty stack.

It turns out that the two definitions of acceptance of a language by a PDA - accpetance by final state and empty stack- are equivalent in the sense that if a language can be accepted by empty stack by some PDA, it can also be accepted by final state by some other PDA and vice versa. Hence it doesn't matter which one we use, since each kind of machine can simulate the other.Given any arbitrary PDA *M* that accpets the language *L* by final

state or empty stack, we can always construct an equivalent PDA *M* with a single final state that accpets exactly the same language *L*. The construction process of *M'* from *M* and the proof of equivalence of *M*&*M'* are given below

There are two cases to be considered.

CASE 1 :PDA*M*accepts by final state, Let M=(Q,∑, ┌,□ ,q0,z0,F). Let qf be a new state not in *Q*. Consider the PDAM'=(Q U{qf},∑, ┌,□ ',q0,z0,F) where □ ' as well as the following transition.

□'(q,□ ,X)contains (qf,X) □ q□ Fand X□ ┌. It is easy to show that *M*and M'are equivalent i.e. L(M)=L(M')
Let ω□ L(M). Then (q0,w,z0)|- M*(q,□ ɣ)for some q□ Fand ┌□ ∑*

Then (q0,w,z0)|- M*(q,□ ɣ)|- M*(qf,□ ɣ)

Thus M' accepts ω.

Conversely, let M'acceptsω i.e. ω□ L(M'), then (q0,w,z0)|- M*(q,□ ɣ)|- M*(qf,□ ɣ)for some q□ F. M inherits all other moves except the last one from *M*. Hence (q0,w,z0)|- M*(q,□ ɣ)for some q□ F.

Thus *M* accepts ω. Informally, on any input M' simulate all the moves of *M* and enters in its

own final state qfwhenever *M* enters in any one of its final status in *F*. Thus M' accepts a string ωiff*M* accepts it.
g
CASE 2 :PDA*M*accepts by empty stack.

we will construct M'from*M* in such a way M' simulates *M* and detects when *M* empties its stack.

M enters its final state qf when and only when *M* empties its stack.ThusM' will accept a string ωiff*M* accepts.

Let M'=(Q U{qo',qf},∑, ┌ U{x},□ ',q0',X,{qf}) where qo',qf not □ Qand □ 'and □ contains all the transition of □ , as well as the following two transitions.

1. □ '(q0,□ ,X)={(q0,z0,X)}and
2. □ '(q0,□ ,X)={(qf,□)}, □ q□ Q

TWO MARKS

1. What are the applications of Context free languages
   Context free languages are used in :
   a. Defining programming languages.
   b. Formalizing the notion of parsing.
   c. Translation of programming languages.
   String processing applications.

2. What are the uses of Context free grammars?
   Construction of compilers.
   Simplified the definition of programming languages.
   Describes the arithmetic expressions with arbitrary nesting of balanced parenthesis { (,) }.
   Describes block structure in programming languages.
   Model neural nets.

3. Define a Context Free Grammar
   A context free grammar (CFG) is denoted as G=(V,T,P,S) where V and T are finite set of variables and terminals respectively. V and T are disjoint. P is a finite set of productions each is of the form A->_ where A is a variable and _ is a string of symbols from (V U T)*.

4. What is the language generated by CFG or G?
   The language generated by G (L(G)) is {w | w is in T* and S=>w. That is a G string is in L(G) if:
   i. The string consists solely of terminals.
   ii. The string can be derived from S.

5. What is : (a) CFL (b) Sentential form
   L is a context free language (CFL) if it is L(G) for some CFG G.
   A string of terminals and variables α is called a sentential form if: S => α ,where S is the start symbol of the grammar.

6. What is the language generated by the grammar G=(V,T,P,S) where
   P={S->aSb, S->ab}?
   S=> aSb=>aaSbb=>……..=>a^nb^n

Thus the language L(G)={ $a^n b^n$ | n>=1}.The language has strings with equal number of a's and b's.

7. What is :(a) derivation (b)derivation/parse tree (c) subtree
   (a) Let G=(V,T,P,S) be the context free grammar. If A-> β is a production of P and α and γ are any strings in (VUT)* then $\alpha A \gamma \Rightarrow \alpha\beta\gamma$

   (b) A tree is a parse \ derivation tree for G if:
   a. Every vertex has a label which is a symbol of VU TU{_}.
   b. The label of the root is S.
   c. If a vertex is interior and has a label A, then A must be in V.
   d. If n has a label A and vertices n1,n2,….. nk are the sons of the vertex n in order from left with labels X1,X2,……..Xk respectively then A X1X2…..Xk must be in P.
   e. If vertex n has label _ ,then n is a leaf and is the only son of its father.

   (c) A subtree of a derivation tree is a particular vertex of the tree together with all its descendants ,the edges connecting them and their labels.The label of the root may not be the start symbol of the grammar.

8. If S->aSb | aAb , A->bAa , A->ba .Find out the CFL
   soln. S->aAb=>abab
   S->aSb=>a aAb b =>a a ba b b(sub S->aAb)
   S->aSb =>a aSb b =>a a aAb b b=>a a a ba b bb
   Thus L={$a^n b^m a^m b^n$, where n,m>=1}

9. What is a ambiguous grammar?
   A grammar is said to be ambiguous if it has more than one derivation trees for a sentence or in other words if it has more than one leftmost derivation or more than one rightmost derivation.

10. Find CFG with no useless symbols equivalent to : $S \rightarrow AB \mid CA$,
    $B \rightarrow BC \mid AB$, $A \rightarrow a$, $C \rightarrow aB \mid b$.
    S-> AB
    S->CA
    B->BC
    B->AB
    A->a
    C->aB
    C->b are the given productions.
    A symbol X is useful if S => αXβ => w. The variable B cannot generate terminals as B->BC and B->AB. Hence B is useless symbol and remove B from all productions.
    Hence useful productions are: S->CA , A->a , C->b

11. Construct CFG without Є production from : $S \rightarrow a \mid Ab \mid aBa$, $A \rightarrow b \mid Є$, $B \rightarrow b \mid A$.
    S->a
    S->Ab
    S->aBa
    A->b
    A->Є
    B->b
    B->A are the given set of production.
    A->Є is the only empty production. Remove the empty production
    S-> Ab , Put A->Є and hence S-> b.
    If B-> A and A->Є then B ->Є

Hence S->aBa becomes S->aa .
Thus S-> a | Ab | b | aBa | aa
A->b
B->b
Finally the productions are: S-> a | Ab | b | aBa | aa
A->b
B->b

12. Define a Context Free Grammar
    A CFG is a grammar whose productions are of the form A -> α where A Є V and α Є (V U T)*
13. Construct a CFG for the language L(G) = $\{0^n1^n : n \geq 1\}$.
    G = {V = {S}, T = {0, 1}, P, S} where P ={S -> 0S S -> 01}
14. Construct a CFG for the language L (G) = $\{0^n1^n: n \geq 0\}$.
    G = {V = {S}, T = {0, 1}, P, S} where P ={S -> 0S1 S ->ε}
15. Find a LM derivation for *aaabbabbba* with the productions.
    P : S → aB | bA, A → a | aS | bAA, B → b | bS | aBB
    Solution: S → aB
    S → aaBB
    S → aaaBBB
    S → aaabBB
    S → aaabbB
    S → aaabbaBB
    S → aaabbabB
    S → aaabbabbS
    S → aaabbabbbA
    S → aaabbabbba
16. Find a L(G) S → aSb, S → ab
    Solution: S → aSb
    1. aaSbb
    2. a^iSb^i
    3. a^iabb^i
    4. a^nb^n

    L(G) = $\{a^nb^n, n \geq 1\}$
    For the grammar *S → aCa, C → aCa | b*. Find L(G)
    Solution:
    S → aCa-> aaCaa-> a^nCa^n
    S → a^nba^n L(G) = $\{a^nba^n, n > 0\}$

17. Construct a CFG for the language over {a,b} which contains palindrome strings.
    G = {V = {S}, T = {a, b}, P, S}
    where
    P ={S -> aSa
    S -> bSb
    S -> a
    S -> b
    S ->ε}
18. Define the language of a Grammar.
    If G = (V, T, P, S) is a CFG, the language of G denoted by L(G), is the set of terminal strings that have derivations from the start symbol i.e. L(G) = {w in T / S -> *w}
19. What are the three ways to simplify a context free grammar?
    removing the useless symbols from the set of productions.
    By eliminating the empty productions.
    By eliminating the unit productions.

UNIT- IV PROPERTIES OF CONTEXT-FREE LANGUAGES

Normal forms for CFG – Pumping Lemma for CFL - Closure Properties of CFL – Turing Machines – Programming Techniques for TM.

PROPERTIES OF CONTEXT - FREE LANGUAGES

Empty Production Removal

The productions of context-free grammars can be coerced into a variety of forms without affecting the expressive power of the grammars. If the empty string does not belong to a language, then there is a way to eliminate the productions of the form $A \rightarrow \lambda$ from the grammar. If the empty string belongs to a language, then we can eliminate λ from all productions save for the single production $S \rightarrow \lambda$. In this case we can also eliminate any occurrences of S from the right-hand side of productions.

Procedure to find CFG with out empty Productions

Step (i): For all productions $A \rightarrow \lambda$, put A into V_N.

Step (ii): Repeat the following steps until no further variables are added to V_N.

For all productions

$$B \rightarrow A_1 A_2 \ \ldots\ldots\ A_n.$$

Step (i): For all productions $A \rightarrow \lambda$, put A into V_N.

Step (ii): Repeat the following steps until no further variables are added to V_N.

For all productions

$$B \rightarrow A_1 A_2 \ \ldots\ldots\ A_n.$$

where $A_1, A_2, A_3, \ldots\ldots, A_n$ are in V_N, put B into V_N.

To find $\hat{P}$, let us consider all productions in P of the form

$$A \rightarrow x_1 x_2 \ \ldots\ldots\ x_m, m \geq 1$$

for each $x_i \in V \cup T$.

UNIT PRODUCTION REMOVAL

Any production of a CFG of the form

$$A \to B$$

where $A, B \in V$ is called a "Unit-production". Having variable one on either side of a production is sometimes undesirable.

"Substitution Rule" is made use of in removing the unit-productions.

Given $G = (V, T, S, P)$, a CFG with no λ-productions, there exists a CFG $\hat{G} = (\hat{V}, \hat{T}, S, \hat{P})$ that does not have any unit-productions and that is equivalent to G.

Let us illustrate the procedure to remove unit-production through example 2.4.6.

Procedure to remove the unit productions:

Find all variables B, for each A such that

$$A \overset{*}{\Rightarrow} B$$

This is done by sketching a "depending graph" with an edge (C, D) whenever the grammar has unit-production $C \to D$, then $A \overset{*}{\Rightarrow} B$ holds whenever there is a walk between A and B.

The new grammar $\hat{G}$, equivalent to G is obtained by letting into $\hat{P}$ all non-unit productions of P.

Then for all A and B satisfying $A \overset{*}{\Rightarrow} B$, we add to $\hat{P}$

$$A \to y_1 \mid y_2 \mid \ldots\ldots \mid y_n$$

where $B \to y_1 \mid y_2 \mid \ldots\ldots \mid y_n$ is the set of all rules in $\hat{P}$ with B on the left.

LEFT RECURSION REMOVAL:

A variable A is left-recursive if it occurs in a production of the form

$$A \to Ax$$

for any $x \in (V \cup T)^*$.

A grammar is left-recursive if it contains at least one left-recursive variable.

Every content-free language can be represented by a grammar that is not left-recursive.

NORMAL FORMS

Two kinds of normal forms viz., Chomsky Normal Form and Greibach Normal Form (GNF) are considered here.

Chomsky Normal Form (CNF)

Any context-free language L without any λ-production is generated by a grammar is which productions are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B VN , and a V T.

Procedure to find Equivalent Grammar in CNF

(i) Eliminate the unit productions, and λ-productions if any,
(ii) Eliminate the terminals on the right hand side of length two or more.
(iii) Restrict the number of variables on the right hand side of productions to two. Proof:
For Step (i): Apply the following theorem: "Every context free language can be generated by a grammar with no useless symbols and no unit productions".
At the end of this step the RHS of any production has a single terminal or two or more symbols. Let us assume the equivalent resulting grammar as G (VN ,VT ,P ,S).
For Step (ii): Consider any production of the form

$B_a \rightarrow a$. Let V_N' be the set of variables in V_N together with $B_a's$ introduced for every terminal on RHS.

The resulting grammar $G_1 = (V_N', V_T, P', S)$ is equivalent to G and every production in P' has either a single terminal or two or more variables.

For step (iii): Consider $A \rightarrow B_1 B_2 \ldots\ldots B_m$

where B_i's are variables and $m \geq 3$.

If $m = 2$, then $A \rightarrow B_1, B_2$ is in proper form.

The production $A \rightarrow B_1 B_2 \ldots\ldots B_m$ is replaced by new productions

$$A \rightarrow B_1 D_1,$$
$$D_1 \rightarrow B_2 D_2,$$
$$\ldots\ldots \quad \ldots\ldots$$
$$\ldots\ldots \quad \ldots\ldots$$
$$D_{m-2} \rightarrow B_{m-1} B_m$$

where $D_i'S$ are new variables.

The grammar thus obtained is G_2, which is in CNF.

Example: Obtain a grammar in Chomsky Normal Form (CNF) equivalent to the grammar G with productions P given

$$S \to aAbB$$
$$A \to aA \mid a$$
$$B \to bB \mid b.$$

Solution

(i) There are no unit productions in the given set of P.

(ii) Amongst the given productions, we have

$$A \to a,$$
$$B \to b$$

which are in proper form.

For $S \to aAbB$, we have

$$S \to B_a AB_b B,$$
$$B_a \to a$$
$$B_b \to b.$$

For $A \to aA$, we have

$$A \to B_a A$$

For $B \to bB$, we have

$$B \to B_b B.$$

(iii) In P' above, we have only

$$S \to B_a AB_b B$$

not in proper form.

Hence we assume new variables D_1 and D_2 and the productions

$$S \to B_a D_1$$
$$D_1 \to AD_2$$
$$D_2 \to B_b B$$

Therefore the grammar in Chomsky Normal Form (CNF) is G_2 with the productions given by

$$S \to B_a D_1,$$
$$D_1 \to AD_2,$$
$$D_2 \to B_b B,$$
$$A \to B_a A,$$
$$B \to B_b B,$$
$$B_a \to a,$$
$$B_b \to b,$$
$$A \to a,$$

and

$$B \to b.$$

Pumping Lemma for CFG

A "Pumping Lemma" is a theorem used to show that, if certain strings belong to a language, then certain other strings must also belong to the language. Let us discuss a Pumping Lemma for CFL. We will show that , if *L* is a context-free language, then strings of *L* that are at least '*m*' symbols long can be "pumped" to produce additional strings in *L*. The value of '*m*' depends on the particular language. Let *L* be an infinite context-free language. Then there is some positive integer '*m*' such that, if *S* is a string of *L* of Length at least '*m*', then

(i) S = uvwxy (for some u, v, w, x, y)

(ii) | vwx| ≤ m

(iii) | vx| ≥1

(iv) uv iwx i y L.

for all non-negative values of *i*. It should be understood that

(i) If *S* is sufficiently long string, then there are two substrings, *v* and *x*, somewhere in *S*. There is stuff (*u*) before *v*, stuff (*w*) between *v* and *x*, and stuff (*y*), after *x*.

(ii) The stuff between *v* and *x* won't be too long, because | *vwx* | can't be larger than *m*.

(iii) Substrings *v* and *x* won't both be empty, though either one could be.

(iv) If we duplicate substring *v*, some number (i) of times, and duplicate *x* the same number of times, the resultant string will also be in *L*.

Definitions A variable is useful if it occurs in the derivation of some string. This requires that

(a) the variable occurs in some sentential form (you can get to the variable if you start from *S*), and

(b) a string of terminals can be derived from the sentential form (the variable is not a "dead end"). A variable is "recursive" if it can generate a string containing itself. For example, variable *A* is recursive if

$$S \overset{*}{\Rightarrow} uAy$$

for some values of u and y.

A recursive variable A can be either

(i) "Directly Recursive", i.e., there is a production

$$A \to x_1 A x_2$$

for some strings $x_1, x_2 \in (T \cup V)^*$, or

(ii) "Indirectly Recursive", i.e., there are variables x_i and productions

$$A \to X_1 \ldots$$
$$X_1 \to \ldots X_2 \ldots$$
$$X_2 \to \ldots X_3 \ldots$$
$$X_N \to \ldots A \ldots$$

Proof of Pumping Lemma

(a) Suppose we have a CFL given by *L*. Then there is some context-free Grammar *G* that generates *L*. Suppose
(i) *L* is infinite, hence there is no proper upper bound on the length of strings belonging to *L*.
(ii) *L* does not contain l.
(iii) *G* has no productions or l-productions.

There are only a finite number of variables in a grammar and the productions for each variable have finite lengths. The only way that a grammar can generate arbitrarily long strings is if one or more variables is both useful and recursive. Suppose no variable is recursive. Since the start symbol is non recursive, it must be defined only in terms of terminals and other variables. Then since those variables are non recursive, they have to be defined in terms of terminals and still other variables and so on.

After a while we run out of "other variables" while the generated string is still finite. Therefore there is an upper bond on the length of the string which can be generated from the start symbol. This contradicts our statement that the language is finite. Hence, our assumption that no variable is recursive must be incorrect.

(b) Let us consider a string *X* belonging to *L*. If *X* is sufficiently long, then the derivation of *X* must have involved recursive use of some variable *A*. Since *A* was used in the derivation, the derivation should have started as

$$S \overset{*}{\Rightarrow} uAy$$

for some values of u and y. Since A was used recursively the derivation must have continued as

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy$$

Finally the derivation must have eliminated all variables to reach a string X in the language.

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy = x$$

This shows that derivation steps

$$A \overset{*}{\Rightarrow} vAx$$

and

$$A \overset{*}{\Rightarrow} w$$

are possible. Hence the derivation

$$A \overset{*}{\Rightarrow} vwx$$

must also be possible.

Usage of Pumping Lemma:

The Pumping Lemma can be used to show that certain languages are not context free.

Let us show that the language

$$L = \{a^i b^i c^i \mid i > 0\}$$

is not context-free.

Proof: Suppose L is a context-free language.

If string $X \in L$, where $|X| > m$, it follows that $X = uvwxy$, where $|vwx| \leq m$.

Choose a value i that is greater than m. Then, wherever vwx occurs in the string $a^i b^i c^i$, it cannot contain more than two distinct letters it can be all a's, all b's, all c's, or it can be a's and b's, or it can be b's and c's.

Therefore the string vx cannot contain more than two distinct letters; but by the "Pumping Lemma" it cannot be empty, either, so it must contain at least one letter.

Now we are ready to "pump".

Since $uvwxy$ is in L, uv^2wx^2y must also be in L. Since v and x can't both be empty,

$$|uv^2wx^2y| > |uvwxy|,$$

so we have added letters.

Both since vx does not contain all three distinct letters, we cannot have added the same number of each letter.

Therefore, uv^2wx^2y cannot be in L.

Thus we have arrived at a "contradiction".

Hence our original assumption, that L is context free should be false. Hence the language L is not con text-free.

Example

Check whether the language given by L {a mbmcn : m ≤ n ≤ 2m} is a CFL or not. Solution

Closure properties of CFL – Substitution

Let Σ be an alphabet, and suppose that for every symbol a in Σ, we choose a language L_a. These chosen languages can be over any alphabets, not necessarily Σ and not necessarily the same. This choice of languages defines a function s (a *substitution*) on Σ, and we shall refer to L_a as $s(a)$ for each symbol a.

If $w = a_1 a_2 \cdots a_n$ is a string in Σ^*, then $s(w)$ is the language of all strings $x_1 x_2 \cdots x_n$ such that string x_i is in the language $s(a_i)$, for $i = 1, 2, \ldots, n$. Put another way, $s(w)$ is the concatenation of the languages $s(a_1)s(a_2)\cdots s(a_n)$. We can further extend the definition of s to apply to languages: $s(L)$ is the union of $s(w)$ for all strings w in L.

Theorem 7.23: If L is a context-free language over alphabet Σ, and s is a substitution on Σ such that $s(a)$ is a CFL for each a in Σ, then $s(L)$ is a CFL.

PROOF: The essential idea is that we may take a CFG for L and replace each terminal a by the start symbol of a CFG for language $s(a)$. The result is a single CFG that generates $s(L)$. However, there are a few details that must be gotten right to make this idea work.

More formally, start with grammars for each of the relevant languages, say $G = (V, \Sigma, P, S)$ for L and $G_a = (V_a, T_a, P_a, S_a)$ for each a in Σ. Since we can choose any names we wish for variables, let us make sure that the sets of variables are disjoint; that is, there is no symbol A that is in two or more of V and any of the V_a's. The purpose of this choice of names is to make sure that when we combine the productions of the various grammars into one set of productions, we cannot get accidental mixing of the productions from two grammars and thus have derivations that do not resemble the derivations in any of the given grammars.

We construct a new grammar $G' = (V', T', P', S)$ for $s(L)$, as follows:

- V' is the union of V and all the V_a's for a in Σ.
- T' is the union of all the T_a's for a in Σ.
- P' consists of:
    1. All productions in any P_a, for a in Σ.
    2. The productions of P, but with each terminal a in their bodies replaced by S_a everywhere a occurs.

Thus, all parse trees in grammar G' start out like parse trees in G, but instead of generating a yield in Σ^*, there is a frontier in the tree where all nodes have labels that are S_a for some a in Σ. Then, dangling from each such node is a parse tree of G_a, whose yield is a terminal string that is in the language $s(a)$.

Inverse Homomorphism:

Theorem 7.30: Let L be a CFL and h a homomorphism. Then $h^{-1}(L)$ is a CFL.

PROOF: Suppose h applies to symbols of alphabet Σ and produces strings in T^*. We also assume that L is a language over alphabet T. As suggested above, we start with a PDA $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$ that accepts L by final state. We construct a new PDA

$$P' = (Q', \Sigma, \delta', (q_0, \epsilon), Z_0, F \times \{\epsilon\}) \tag{7.1}$$

where:

1. Q' is the set of pairs (q, x) such that:
   (a) q is a state in Q, and
   (b) x is a suffix (not necessarily proper) of some string $h(a)$ for some input symbol a in Σ.

   That is, the first component of the state of P' is the state of P, and the second component is the buffer. We assume that the buffer will periodically be loaded with a string $h(a)$, and then allowed to shrink from the front, as we use its symbols to feed the simulated PDA P. Note that since Σ is finite, and $h(a)$ is finite for all a, there are only a finite number of states for P'.

2. δ' is defined by the following rules:
   (a) $\delta'\big((q, \epsilon), a, X\big) = \{\big((q, h(a)\big), X\big)\}$ for all symbols a in Σ, all states q in Q, and stack symbols X in Γ. Note that a cannot be ϵ here. When the buffer is empty, P' can consume its next input symbol a and place $h(a)$ in the buffer.
   (b) If $\delta(q, b, X)$ contains (p, γ), where b is in T or $b = \epsilon$, then

   $$\delta'\big((q, bx), \epsilon, X\big)$$

   contains $\big((p, x), \gamma\big)$. That is, P' always has the option of simulating a move of P, using the front of its buffer. If b is a symbol in T, then the buffer must not be empty, but if $b = \epsilon$, then the buffer can be empty.

3. Note that, as defined in (7.1), the start state of P' is (q_0, ϵ); i.e., P' starts in the start state of P with an empty buffer.

4. Likewise, the accepting states of P', as per (7.1), are those states (q, ϵ) such that q is an accepting state of P.

The following statement characterizes the relationship between P' and P:

- $(q_0, h(w), Z_0) \overset{*}{\underset{P}{\vdash}} (p, \epsilon, \gamma)$ if and only if $((q_0, \epsilon), w, Z_0) \overset{*}{\underset{P'}{\vdash}} ((p, \epsilon), \epsilon, \gamma)$.

TURING MACHINE: INFORMAL DEFINITION:

We consider here a basic model of TM which is deterministic and have one-tape. There are many variations, all are equally powerfull.

The basic model of TM has a finite set of states, a semi-infinite tape that has a leftmost cell but is infinite to the right and a tape head that can move left and right over the tape, reading and writing symbols.

For any input w with |w|=n, initially it is written on the n leftmost (contiguous) tape cells. The infinitely many cells to the right of the input all contain a blank symbol, B which is a special tape symbol that is not an input symbol. The machine starts in its start state with its head scanning the leftmost symbol of the input w.
Depending upon the symbol scanned by the tape head and the current state the machine makes a move which consists of the following:

- writes a new symbol on that tape cell,
- moves its head one cell either to the left or to the right and
- (possibly) enters a new state.

The action it takes in each step is determined by a transition functions. The machine continues computing (i.e. making moves) until

- it decides to "accept" its input by entering a special state called accept or final state or
- halts without accepting i.e. rejecting the input when there is no move defined.

On some inputs the TM many keep on computing forever without ever accepting or rejecting the input, in which case it is said to "loop" on that input

Formal Definition :

Formally, a deterministic turing machine (DTM) is a 7-tuple  , where

- Q is a finite nonempty set of states.
- is a finite non-empty set of tape symbols, call ed the tape alphabet of *M*.

- [redacted] is a finite non-empty set of input symbols, called the input alphabet of *M*.
- [redacted] is the transition function of *M*,
- [redacted] is the initial or start state.
- [redacted] is the blank symbol
- [redacted] is the set of final state.

So, given the current state and tape symbol being read, the transition function describes the next state, symbol to be written on the tape, and the direction in which to move the tape head (*L* and *R* denote left and right, respectively).

Transition function : [redacted]

- The heart of the TM is the transition function, [redacted] because it tells us how the machine gets one step to the next.
- when the machine is in a certain state q [redacted] Q and the head is currently scanning the tape symbol [redacted], and if [redacted], then the machine

1. replaces the symbol *X* by *Y* on the tape
2. goes to state *p*, and
3. the tape head moves one cell (i.e. one tape symbol) to the left (or right) if *D* is *L* (or *R*).

The ID (instantaneous description) of a TM capture what is going out at any moment i.e. it contains all the information to exactly capture the "current state of the computations".

It contains the following:

- The current state, *q*
- The position of the tape head,
- The constants of the tape up to the rightmost nonblank symbol or the symbol to the left of the head, whichever is rightmost.

Note that, although there is no limit on how far right the head may move and write nonblank symbols on the tape, at any finite time, the TM has visited only a finite prefix of the infinite tape.

An ID (or configuration) of a TM *M* is denoted by [redacted] where [redacted] and

- [redacted] is the tape contents to the left of the head
- *q* is the current state.
- [redacted] is the tape contents at or to the right of the tape head

That is, the tape head is currently scanning the leftmost tape symbol of [redacted]. (Note that if [redacted], then the tape head is scanning a blank symbol)

If  is the start state and w is the input to a TM M then the starting or initial configuration of M is denoted by 

The Halting Problem:

The input to a Turing machine is a string. Turing machines themselves can be written as strings. Since these strings can be used as input to other Turing machines. A "Universal Turing machine" is one whose input consists of a description *M* of some arbitrary Turing machine, and some input w to which machine *M* is to be applied, we write this combined input as *M* + *w*. This produces the same output that would be produced by M. This is written as
Universal Turing Machine (*M* + *w*) = *M* (*w*).

As a Turing machine can be represented as a string, it is fully possible to supply a Turing machine as input to itself, for example *M* (*M*). This is not even a particularly bizarre thing to do for example, suppose you have written a C pretty printer in C, then used the Pretty printer on itself. Another common usage is Bootstrapping—where some convenient languages used to write a minimal compiler for some new language L, then used this minimal compiler for L to write a new, improved compiler for language L. Each time a new feature is added to language L, you can recompile and use this new feature in the next version of the compiler. Turing machines sometimes halt, and sometimes they enter an infinite loop.

A Turing machine might halt for one input string, but go into an infinite loop when given some other string. The halting problem asks: "It is possible to tell, in general, whether a given machine will halt for some given input?" If it is possible, then there is an effective procedure to look at a Turing machine and its input and determine whether the machine will halt with that input. If there is an effective procedure, then we can build a Turing machine to implement it. Suppose we have a Turing machine "WillHalt" which, given an input string *M* + *w*, will halt and accept the string if Turing machine *M* halts on input *w* and will halt and reject the string if Turing machine *M* does not halt on input *w*. When viewed as a Boolean function, "WillHalt (*M*, *w*)" halts and returns "TRUE" in the first case, and (halts and) returns "FALSE" in the second.

Theorem:

Turing Machine "WillHalt (*M*, *w*)" does not exist.

Proof: This theorem is proved by contradiction. Suppose we could build a machine "WillHalt". Then we can certainly build a second machine, "LoopIfHalts", that will go into an infinite loop if and only if "WillHalt" accepts its input:

Function LoopIfHalts (*M*, *w*): if
WillHalt (*M*, *w*) then while true do { }
else
return false;

We will also define a machine "LoopIfHaltOnItSelf" that, for any given input *M*, representing a Turing machine, will determine what will happen if *M* is applied to itself, and loops if *M* will halt in this case.

Function LoopIfHaltsOnItself (*M*): return
LoopIfHalts (*M*, *M*):

Finally, we ask what happens if we try:

Function Impossible:
return LoopIfHaltsOnItself (LoopIfHaltsOnItself):

This machine, when applied to itself, goes into an infinite loop if and only if it halts when applied to itself. This is impossible. Hence the theorem is proved.

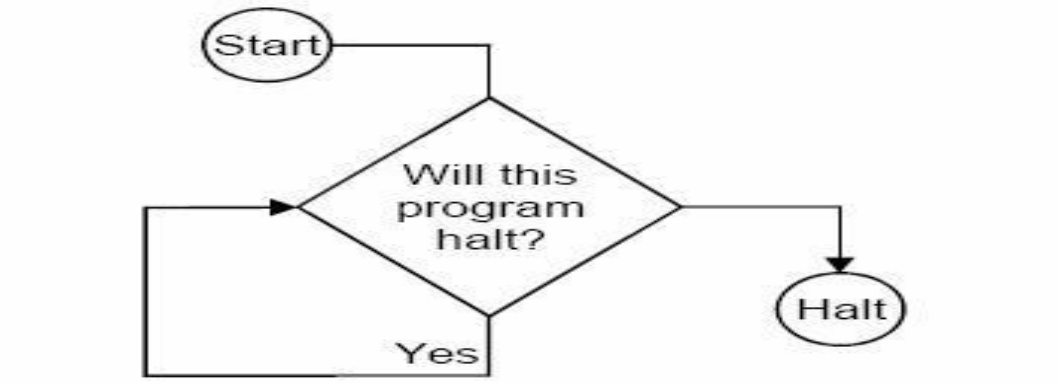
A Turing machine can be "programmed," in much the same manner as a computer is programmed. When one specifies the function which we usually call δ for a Tm, he is really writing a program for the Tm.

1. Storage in finite Control

The finite control can be used to hold a finite amount of information. To do so, the state is written as a pair of elements, one exercising control and the other storing a symbol. It should be emphasized that this arrangement is for conceptual purposes only. No modification in the definition of the Turing machine has been made.

Example

Consider the Turing machine Solution



$$T = (K, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], F),$$

where K can be written as $\{q_0, q_1\} \times \{0, 1, B\}$. That is, K consists of the pairs $[q_0, 0]$, $[q_0, 1]$, $[q_0, B]$, $[q_1, 0]$, $[q_1, 1]$, and $[q_1, B]$. The set F is $\{[q_1, B]\}$. T looks at the first input symbol, records it in its finite control, and checks that the symbol does not appear elsewhere on its input. The second component of the state records the first input symbol. Note that T accepts a regular set, but T will serve for demonstration purposes. We define δ as follows.

1. a) $\delta([q_0, B], 0) = ([q_1, 0], 0, R)$
   b) $\delta([q_0, B], 1) = ([q_1, 1], 1, R)$
   (T stores the symbol scanned in second component of the state and moves right. The first component of T's state becomes q_1.)

2. a) $\delta([q_1, 0], 1) = ([q_1, 0], 1, R)$
   b) $\delta([q_1, 1], 0) = ([q_1, 1], 0, R)$
   (If T has a 0 stored and sees a 1, or vice versa, then T continues to move to the right.)

3. a) $\delta([q_1, 0], B) = ([q_1, B], 0, L)$
   b) $\delta([q_1, 1], B) = ([q_1, B], 0, L)$
   (T enters the final state $[q_1, B]$ if T reaches a blank symbol without having first encountered a second copy of the leftmost symbol.)

If T reaches a blank in state $[q_1, 0]$ or $[q_1, 1]$, it accepts. For state $[q_1, 0]$ and symbol 0 or for state $[q_1, 1]$ and symbol 1, δ is not defined, so if T ever sees the symbol stored, it halts without accepting.

In general, we can allow the finite control to have k components, all but one of which store information.

2. Multiple Tracks

We can imagine that the tape of the Turing machine is divided into k tracks, for any finite k. This arrangement is shown in Fig., with k = 3. What is actually done is that the symbols on the tape are considered as k-tuples. One component for each track.

Example

The tape in Fig. can be imagined to be that of a Turing machine which takes a binary input greater than 2, written on the first track, and determines if it is a prime. The input is surrounded by ¢ and $ on the first track.

Thus, the allowable input symbols are [¢, B, B], [0, B, B], [1, B, B], and [$, B, B]. These symbols can be identified with ¢, 0, 1, and $, respectively, when viewed as input symbols. The blank symbol can be represented by [B, B, B]

To test if its input is a prime, the Tm first writes the number two in binary on the second track and copies the first track onto the third track. Then, the second track is subtracted, as many times as possible, from the third track, effectively dividing the third track by the second and leaving the remainder. If the remainder is zero, the number on the first track is not a prime. If the remainder is nonzero, increase the number on the second track by one.

If now the second track equals the first, the number on the first track is a prime, because it cannot be divided by any number between one and itself. If the second is less than the first, the whole operation is repeated for the new number on the second track. In Fig., the Tm is testing to determine if 47 is a prime. The Tm is dividing by 5; already 5 has been subtracted twice, so 37 appears on the third track.

3. Subroutines

VII. SUBROUTINES. It is possible for one Turing machine to be a "subroutine" of another Tm under rather general conditions. If T_1 is to be a subroutine of T_2, we require that the states of T_1 be disjoint from the states of T_2 (excluding the states of T_2's subroutine). To "call" T_1, T_2 enters the start state of T_1. The rules of T_1 are part of the rules of T_2. In addition, from a halting state of T_1, T_2 enters a state of its own and proceeds.

TWO MARKS

1. **What are the two major normal forms for context – free grammar?**

The two Normal forms are

- Chomsky Normal Form
- Greibach Normal Form

2. **What is a useless symbol? Nov/Dec 2007**

A symbol x is useful if there is a derivation.

$S \Rightarrow \alpha x \beta \Rightarrow w$ for some $\alpha, \beta, w \in T^*$

or else, it is useless.

3. **What is £ - Production rule?**

Any production rule of the form $A \rightarrow \varepsilon$ is known as ε - production.

4. Define Unit Production.

Any production rule of the form A → B is known as unit production.

5. When do you say a symbol is useful?

We say a symbol is useful either if it derives a string of terminals or it can be used in the middle of a derivation which yields a terminal or a string of terminals.

6. Define Chomsky's Normal form.

A CFG whose production rules are of the form

A → BC or A → a

where A, B, and C are variables and a is terminal.

7. Write the procedure to eliminate £ - productions.

- For all productions A →ε, put A into V^1
- Repeat the following steps until no new variables are added.
  a. For all productions
  $B \rightarrow A_1 A_2 A_3 \ldots A_n$
  where $A_1 A_2 A_3 \ldots A_n$ are in V^1
  b. Put B into $V^{1.}$

8. Write the procedure to eliminate the unit productions.

- Find all variables B, for each A such that
  $A \overset{*}{\Longrightarrow} B$
- The new grammar G' is obtained by letting into P' all non – unit productions of P.
- For all A and B satisfying A ==>, add to P'
  $A \rightarrow y_1 \mid y_2 \mid \ldots \mid y_n$
  where $B \rightarrow y_1 \mid y_2 \mid \ldots \mid y_n$ is the set of productions in P'

9. Eliminate the useless symbol from the following

S → AB | a A → b

B is an useless symbol since it doesn't derive a terminal. Eliminating it we get

S → a

A → b

10. Define Greibach Normal form. Nov/Dec 2009

A CFG whose production rules are of the form A → α where a is a terminal and α is either empty or a string of non – terminals.

11. State pumping lemma for Context free language. April/May 2008

Let L be a CFL. Then there exists a constant n such that if z is any string in L such that $|z| \leq n$ then we can write z = uvwxy subject to the following conditions.

a) $|vwx| \leq n$
b) $vx \neq \varepsilon$
c) for all $i \geq 0$ uv^iwx^iy is in L.
d)

12. What is the use of pumping lemma for CFG.

It is used to check whether a given language is context free language or not.

13. What operations that preserve CFL's.

1. Substitution

2. Union

3. Concatenation

4. Closure (star)

5. Reversal

14. What is a formal language?

Language is a set of valid strings from some alphabet. The set may be empty,finite or infinite. L(M) is the language defined by machine M and L(G) is the language defined by Context free grammar. The two notations for specifying formal languages are: Grammar or regular expression Generative approach) Automaton(Recognition approach)

15. What is Backus-Naur Form(BNF)?

Computer scientists describes the programming languages by a notation called Backus- Naur Form. This is a context free grammar notation with minor changes in format and some shorthand.

16. Let G= ({S,C} ,{a,b},P,S) where P consists of S->aCa , C->aCa |b. FindL(G).

S-> aCa => aba

S->aCa=> a aCa a=>aabaa

S->aCa=> a aCa a=> a a aCa a a =>aaabaaa

Thus L(G)= { $a^n ba^n$,where n>=1 }

17. Find L(G) where G= ({S} ,{0,1}, {S->0S1 ,S->_ },S) S->_ , _ is in L(G)

S-> 0S1 =>0_1=>01 S->0S1=>0 0S11=>0011 Thus L(G)= { 0n1n | n>=0}

18. What is a parser?

A parser for grammar G is a program that takes as input a string w and produces as output either a parse tree for w ,if w is a sentence of G or an error message indicating that w is not a sentence of G.

19. What are the closure properties of CFL?

CFL are closed under union, concatenation and Kleene closure. CFL are closed under substitution , homomorphism. CFL are not closed under intersection , complementation. Closure properties of CFL's are used to prove that certain languages are not context free.

20. State the pumping lemma for CFLs.

Let L be any CFL. Then there is a constant n, depending only on L, such that if z is in L and |z| >=n, then z=uvwxy such that :

(i) |vx| >=1

(ii) |vwx| <=n and

(iii) for all i>=0 uviwxiy is in L.

UNIT- V UNDECIDABILITY

A language that is not Recursively Enumerable (RE) – An undecidable problem that is RE – Undecidable problems about Turing Machine – Post's Correspondence Problem - The classes P and NP.

5.1 INTRODUCTION:

While there are many branches of knowledge, each having its own problems and methods, `Undecidability' is a subject dealing with the very nature of problems itself. Given any problem, does it have a solution? Is there any method to find the solution? These are the kind of questions which this subject tries to address. In what follows, we shall see the answers to these questions. But first, we need precise definitions of what is a problem, a solution, a method and a few entertaining mathematical results. Design a Turing machine to add two given integers.

Solution:

Assume that m and n are positive integers. Let us represent the input as $0^m B 0^n$.

If the separating B is removed and 0's come together we have the required output, $m + n$ is unary.

(i) The separating B is replaced by a 0.
(ii) The rightmost 0 is erased i.e., replaced by B.

Let us define $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0\}, \{0, B\}, \delta, q_0, \{q_4\})$. δ is defined by Table shown below.

	Tape Symbol	
State	0	B
q_0	$(q_0, 0, R)$	$(q_1, 0, R)$
q_1	$(q_1, 0, R)$	(q_2, B, L)
q_2	(q_3, B, L)	—
q_3	$(q_3, 0, L)$	(q_4, B, R)

M starts from ID $q_0 0^m B 0^n$, moves right until seeking the blank B. M changes state to q_1. On reaching the right end, it reverts, replaces the rightmost 0 by B. It moves left until it reaches the beginning of the input string. It halts at the final state q_4.

Some unsolvable Problems are as follows:

- Does a given Turing machine *M* halts on all input?
- Does Turing machine *M* halt for any input?
- Is the language *L*(*M*) finite?
- Does *L*(*M*) contain a string of length *k*, for some given *k*?

- Do two Turing machines *M*1 and *M*2 accept the same language?

It is very obvious that if there is no algorithm that decides, for an arbitrary given Turing machine *M* and input string *w*, whether or not *M* accepts *w*. These problems for which no algorithms exist are called "UNDECIDABLE" or "UNSOLVABLE".

Our next goal is to devise a binary code for Turing machines so that each TM with input alphabet $\{0, 1\}$ may be thought of as a binary string. Since we just saw how to enumerate the binary strings, we shall then have an identification of the Turing machines with the integers, and we can talk about "the ith Turing machine, M_i." To represent a TM $M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, F)$ as a binary string, we must first assign integers to the states, tape symbols, and directions L and R.

- We shall assume the states are $q_1, q_2, \ldots, q_r$ for some r. The start state will always be q_1, and q_2 will be the only accepting state. Note that, since we may assume the TM halts whenever it enters an accepting state, there is never any need for more than one accepting state.
- We shall assume the tape symbols are $X_1, X_2, \ldots, X_s$ for some s. X_1 always will be the symbol 0, X_2 will be 1, and X_3 will be B, the blank. However, other tape symbols can be assigned to the remaining integers arbitrarily.
- We shall refer to direction L as D_1 and direction R as D_2.

Since each TM M can have integers assigned to its states and tape symbols in many different orders, there will be more than one encoding of the typical TM. However, that fact is unimportant in what follows, since we shall show that no encoding can represent a TM M such that $L(M) = L_d$.

Once we have established an integer to represent each state, symbol, and direction, we can encode the transition function δ. Suppose one transition rule is $\delta(q_i, X_j) = (q_k, X_l, D_m)$, for some integers i, j, k, l, and m. We shall code this rule by the string $0^i10^j10^k10^l10^m$. Notice that, since all of i, j, k, l, and m are at least one, there are no occurrences of two or more consecutive 1's within the code for a single transition.

A code for the entire TM M consists of all the codes for the transitions, in some order, separated by pairs of 1's:

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

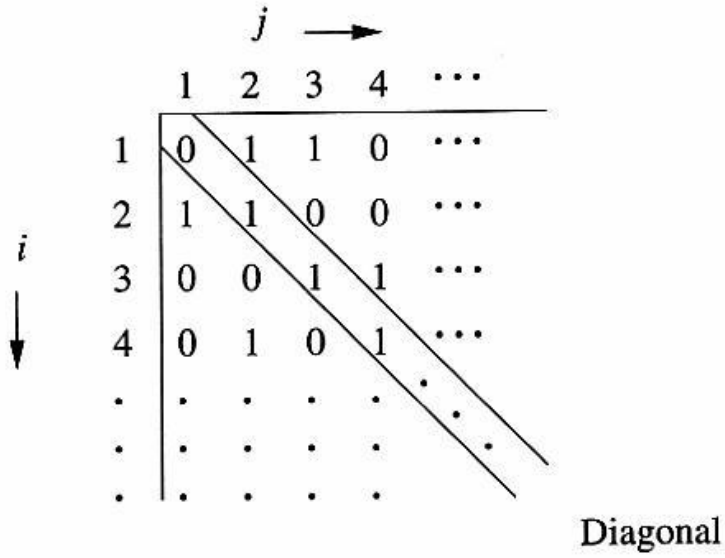
where each of the C's is the code for one transition of M.

5.1.1 Diagonalization language:

- The language L_d, the *diagonalization language*, is the set of strings w_i such that w_i is not in $L(M_i)$.

That is, L_d consists of all strings w such that the TM M whose code is w does not accept when given w as input.

The reason L_d is called a "diagonalization" language can be seen if we consider Fig. 9.1. This table tells for all i and j, whether the TM M_i accepts input string w_j; 1 means "yes it does" and 0 means "no it doesn't."[1] We may think of the ith row as the *characteristic vector* for the language $L(M_i)$; that is, the 1's in this row indicate the strings that are members of this language.



This table represents language acceptable by Turing machine

The diagonal values tell whether M_i accepts w_i. To construct L_d, we complement the diagonal. For instance, if Fig. 9.1 were the correct table, then the complemented diagonal would begin $1, 0, 0, 0, \ldots$. Thus, L_d would contain $w_1 = \epsilon$, not contain w_2 through w_4, which are 0, 1, and 00, and so on.

The trick of complementing the diagonal to construct the characteristic vector of a language that cannot be the language that appears in any row, is called *diagonalization*. It works because the complement of the diagonal is

Proof that L_d is not recursively enumerable:

Theorem 9.2: L_d is not a recursively enumerable language. That is, there is no Turing machine that accepts L_d.

PROOF: Suppose L_d were $L(M)$ for some TM M. Since L_d is a language over alphabet $\{0,1\}$, M would be in the list of Turing machines we have constructed, since it includes all TM's with input alphabet $\{0,1\}$. Thus, there is at least one code for M, say i; that is, $M = M_i$.

Now, ask if w_i is in L_d.

- If w_i is in L_d, then M_i accepts w_i. But then, by definition of L_d, w_i is not in L_d, because L_d contains only those w_j such that M_j does *not* accept w_j.
- Similarly, if w_i is not in L_d, then M_i does not accept w_i, Thus, by definition of L_d, w_i *is* in L_d.

Since w_i can neither be in L_d nor fail to be in L_d, we conclude that there is a contradiction of our assumption that M exists. That is, L_d is not a recursively enumerable language. □

5.2 RECURSIVE LANGUAGES:

We call a language L *recursive* if $L = L(M)$ for some Turing machine M such that:

1. If w is in L, then M accepts (and therefore halts).
2. If w is not in L, then M eventually halts, although it never enters an accepting state.

A TM of this type corresponds to our informal notion of an "algorithm," a well-defined sequence of steps that always finishes and produces an answer. If we think of the language L as a "problem," as will be the case frequently, then problem L is called *decidable* if it is a recursive language, and it is called *undecidable* if it is not a recursive language.

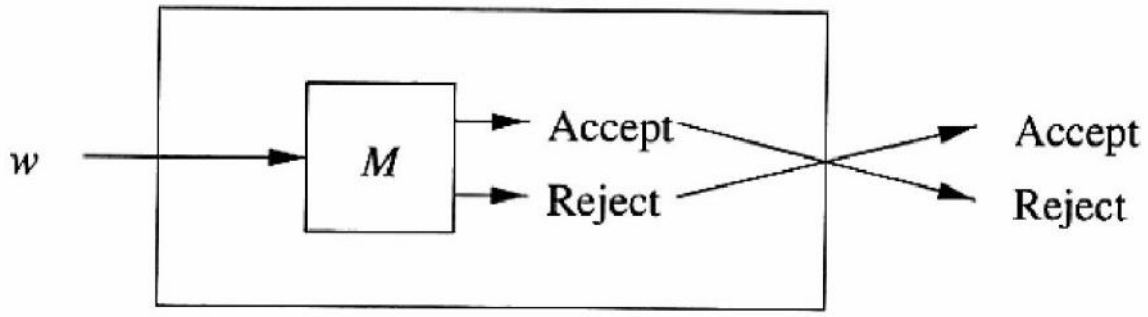
Theorem 9.3: If L is a recursive language, so is $\overline{L}$.

PROOF: Let $L = L(M)$ for some TM M that always halts. We construct a TM $\overline{M}$ such that $\overline{L} = L(\overline{M})$ by the construction suggested in Fig. 9.3. That is, $\overline{M}$ behaves just like M. However, M is modified as follows to create $\overline{M}$:

1. The accepting states of M are made nonaccepting states of $\overline{M}$ with no transitions; i.e., in these states $\overline{M}$ will halt without accepting.

2. $\overline{M}$ has a new accepting state r; there are no transitions from r.

3. For each combination of a nonaccepting state of M and a tape symbol of M such that M has no transition (i.e., M halts without accepting), add a transition to the accepting state r.



Since M is guaranteed to halt, we know that $\overline{M}$ is also guaranteed to halt. Moreover, $\overline{M}$ accepts exactly those strings that M does not accept. Thus $\overline{M}$ accepts $\overline{L}$. □

Theorem 9.4: If both a language L and its complement are RE, then L is recursive. Note that then by Theorem 9.3, $\overline{L}$ is recursive as well.

PROOF: The proof is suggested by Fig. 9.4. Let $L = L(M_1)$ and $\overline{L} = L(M_2)$. Both M_1 and M_2 are simulated in parallel by a TM M. We can make M a two-tape TM, and then convert it to a one-tape TM, to make the simulation easy and obvious. One tape of M simulates the tape of M_1, while the other tape of M simulates the tape of M_2. The states of M_1 and M_2 are each components of the state of M.

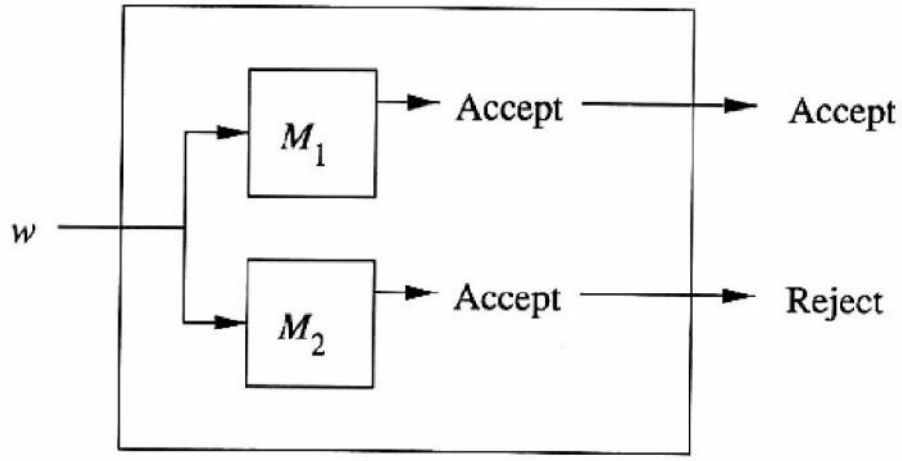


Figure 9.4: Simulation of two TM's accepting a language and its complement

If input w to M is in L, then M_1 will eventually accept. If so, M accepts and halts. If w is not in L, then it is in $\overline{L}$, so M_2 will eventually accept. When M_2 accepts, M halts without accepting. Thus, on all inputs, M halts, and $L(M)$ is exactly L. Since M always halts, and $L(M) = L$, we conclude that L is recursive. □

5.2.1 UNIVERSAL LANGUAGE:

We define L_u, the *universal language*, to be the set of binary strings that encode, in the notation of Section 9.1.2, a pair (M, w), where M is a TM with the binary input alphabet, and w is a string in $(\mathbf{0}+\mathbf{1})^*$, such that w is in $L(M)$. That is, L_u is the set of strings representing a TM and an input accepted by that TM. We shall show that there is a TM U, often called the *universal Turing machine*, such that $L_u = L(U)$. Since the input to U is a binary string, U is in fact some M_i in the list of binary-input Turing machines we developed in

5.3 UNDECIDABILITY OF UNIVERSAL LANGUAGE:

Theorem 9.6: L_u is RE but not recursive.

PROOF: We just proved in Section 9.2.3 that L_u is RE. Suppose L_u were recursive. Then by Theorem 9.3, $\overline{L_u}$, the complement of L_u, would also be recursive. However, if we have a TM M to accept $\overline{L_u}$, then we can construct a TM to accept L_d (by a method explained below). Since we already know that L_d is not RE, we have a contradiction of our assumption that L_u is recursive.

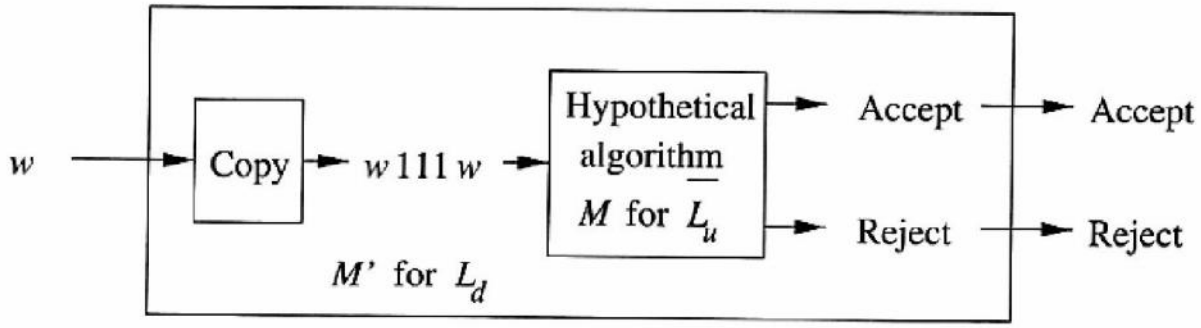


Figure 9.6: Reduction of L_d to $\overline{L_u}$

Suppose $L(M) = \overline{L_u}$. As suggested by Fig. 9.6, we can modify TM M into a TM M' that accepts L_d as follows.

1. Given string w on its input, M' changes the input to $w111w$. You may, as an exercise, write a TM program to do this step on a single tape. However, an easy argument that it can be done is to use a second tape to copy w, and then convert the two-tape TM to a one-tape TM.

2. M' simulates M on the new input. If w is w_i in our enumeration, then M' determines whether M_i accepts w_i. Since M accepts $\overline{L_u}$, it will accept if and only if M_i does not accept w_i; i.e., w_i is in L_d.

Thus, M' accepts w if and only if w is in L_d. Since we know M' cannot exist by Theorem 9.2, we conclude that L_u is not recursive. □

5.4 Post's Correspondence Problem (Pcp)

A post correspondence system consists of a finite set of ordered pairs

(x_i, y_i), $i = 1, 2, \cdots, n$, where

$x_i, y_i \in \Sigma^+$ for some alphabet $\sum$.

Any sequence of numbers $i_1, i_2, \ldots i_k$ s – t is called a solution to a Post Correspondence System. The Post's Correspondence Problem is the problem of determining whether a Post Correspondence system has a solutions.

Example 1 : Consider the post correspondence system

{(aa,aa), (bb,ba),(abb,b)}. The list 1,2,1,3 is a solution to it.

Because $x_1x_2x_1x_3=y_1y_2y_1y_3$

A post correspondence system is also denoted as an instance of the PCP) Example 2 : The following PCP instance has no solution. This can be proved as follows. cannot be chosen at the start, since than the LHS and RHS would differ in the first symbol (in LHS and in RHS). So, we must start with. The next pair must be so that the 3 rd symbol in the RHS becomes identical to that of the LHS, which is a . After this step, LHS and RHS are not matching. If is selected next, then would be mismatched in the 7 th symbol (in LHS and in RHS). If is selected, instead, there will not be any choice to match the both side in the next step.

Example3 : The list 1,3,2,3 is a solution to the following PCP instance.

The following properties can easily be proved.

Proposition The Post Correspondence System has solutions if and only if

Corollary : PCP over one-letter alphabet is decidable.

Proposition Any PCP instance over an alphabet with is equivalent to a PCP instance over an alphabet with

Proof : Consider We can now encode every as any PCP instance over will now have only two symbols, 0 and 1 and, hence, is equivalent to a PCP instance over

Theorem : PCP is undecidable. That is, there is no algorithm that determines whether an arbitrary Post Correspondence System has a solution.

Proof: The halting problem of turning machine can be reduced to PCP to show the undecidability of PCP. Since halting problem of TM is undecidable (already proved), This reduction shows that PCP is also undecidable. The proof is little bit lengthy and left as an exercise.

Some undecidable problem in context-free languages

We can use the undecidability of PCP to show that many problem concerning the context-free languages are undecidable. To prove this we reduce the PCP to each of these problem. The following discussion makes it clear how PCP can be used to serve this purpose. Let be a Post Correspondence System over the alphabet. We construct two CFG's G_x and G_y from the ordered pairs x,y respectively as follows. It is clear that the grammar generates the strings that can appear in the LHS of a sequence while solving the PCP followed by a sequence of numbers. The sequence of number at the end records the sequence of strings from the PCP instance (in reverse order) that generates the string. Similarly, generates the strings that can be obtained from the RHS of a sequence and the corresponding sequence of numbers (in reverse order). Now, if the Post Correspondence System has a solution, then there must be a sequence. Conversely, let Hence, w must be in the form w_1w_2 where and w_2 in a sequence (since, only that kind of strings can be generated by each of and).Now, the string is a solution to the Post Correspondence

System. It is interesting to note that we have here reduced PCP to the language of pairs of CFG,s whose intersection is nonempty. The following result is a direct conclusion of the above.

Theorem : Given any two CFG's G_1 and G_2 the question "Is " is undecidable.

Proof: Assume for contradiction that there exists an algorithm *A* to decide this question. This would imply that PCP is decidable as shown below.

For any Post Correspondence System, *P* construct grammars and by using the constructions elaborated already. We can now use the algorithm *A* to decide whether and Thus, PCP is decidable, a contradiction. So, such an algorithm does not exist.

If and are CFG's constructed from any arbitrary Post Correspondence System, than it is not difficult to

show that and are also context-free, even though the class of context-free languages are not closed under complementation.

and their complements can be used in various ways to show that many other questions related to CFL's are undecidable. We prove here some of those.

5.5 Class p-problem solvable in polynomial time:

A Turing machine M is said to be of *time complexity* $T(n)$ [or to have "running time $T(n)$"] if whenever M is given an input w of length n, M halts after making at most $T(n)$ moves, regardless of whether or not M accepts. This definition applies to any function $T(n)$, such as $T(n) = 50n^2$ or $T(n) = 3^n + 5n^4$; we shall be interested predominantly in the case where $T(n)$ is a polynomial in n. We say a language L is in class $\mathcal{P}$ if there is some polynomial $T(n)$ such that $L = L(M)$ for some deterministic TM M of time complexity $T(n)$.

5.5.1 Non deterministic polynomial time:

A nondeterministic TM that never makes more than p(n) moves in any sequence of choices for some polynomial p is said to be non polynomial time NTM. NP is the set of languags that are accepted by polynomial time NTM's. Many problems are in NP but appear not to be in p. One of the great mathematical questions of our age: is there anything in NP that is not in p?

5.5.2 NP-complete problems:

If We cannot resolve the "p=np question, we can at least demonstrate that certain problems in NP are the hardest , in the sense that if any one of them were in P , then P=NP. These are called NP-complete. Intellectual leverage: Each NP-complete problem's apparent difficulty reinforces the belief that they are all hard.

5.5.3 Methods for proving NP-Complete problems:

Polynomial time reduction (PTR): Take time that is some polynomial in the input size to convert instances of one problem to instances of another. If P1 PTR to P2 and P2 is in P1 the so is P1. Start by showing every problem in NP has a PTR to Satisfiability of Boolean formula.Then, more problems can be proven NP complete by showing that SAT PTRs to them directly or indirectly.

TWO MARKS QUESTIONS AND ANSWERS

1. **What is the weak-form of Turing thesis?**

A Turing Machine can pute anything that can be puted by a general purpose digital puter.

2. **What is the strong-form of Turing thesis?**

A Turing Machine can pute anything that can be puted. This is the strong form of Turing thesis.

3. **When a language is said to be recursively enumerable?**

A language is recursively enumerable if there exists a Turing Machine that accepts every string of the language and does not accept strings that are not in the language.

4. **When a language is said to be recursive?**

A language L is said to be recursive if there exists a Turing machine M that accepts L, and goes to halt state or else M rejects L. The language L_d. Which consists of all those strings w such that the Turing machine represented by w does not accept the input w.

$L_d = \{ w_i \mid w_i \quad L(M_i)\}$

6. **Define decidability (or) decidable problems?**

A problem is said to be decidable if there exists a Turing machine which gives one 'yes' or 'no' answer for every input in the language.

(or)

A problem is said to be decidable if it is a recursive language.

7. **Define Undecidable problem?**

If a problem is not a recursive language, then it is called undecidable problem.

8. **Define Universal language?**

A Universal Turing Machine Mu is an automation, that given as input the description of any Turing Machine M and a string w, can simulate the putation of M on w.

9. **What are the reasons for a TM not accepting its input?**

i) The TM may halt in a non final state.

ii) The TM may enter into an indefinite loop.

10. **Define trivial property?**

A property is trivial if it is either empty of is all RE languages.

11. **Define rice Theorem?**

Every non-trivial property of the RE languages is undecidable.

12. **Define post's correspondence problem?**

An instance of PCP consists of two lists, $A = w_1, w_2, w_3, \ldots\ldots . w_k$ $B = x_1, x_2, x_3, \ldots\ldots\ldots\ldots x_k$ of strings over some $\sum$.

This instance of PCP has a solution if there is any sequence of

integers $i_1, i_2, \ldots\ldots\ldots i_m$ with $m \geq 1$.

Such that

$w_{i1}, w_{i2}, w_{i3} \ldots\ldots\ldots\ldots\ldots\ldots\ldots w_{im} = x_{i1}, x_{i2}, x_{i3}, \ldots\ldots\ldots\ldots x_{im}$

The sequence of $i_1, i_2, \ldots\ldots\ldots\ldots\ldots i_m$ is a solution to this instance of PCP.

13. Let A and B be lists of three strings each, as defined in the following table?

	A	B
	w	**x**
1	**1**	**111**
2	**10111**	**10**
3	**10**	**0**

Find the instance of post correspondence Problem.

Solution :

Apply $w_{i1}, w_{i2}, w_{i3} \ldots\ldots\ldots\ldots\ldots\ldots\ldots w_{im} = x_{i1}, x_{i2}, x_{i3}, \ldots\ldots\ldots\ldots x_{im}$ to this problem.

Take M = 4

w 2 w 1 w 1 wi3 = x 2 x 1 x 1 xi3

10 111 111

0 =10 111 111 0

Instance = 2,1,1,3.

14. Define modified . post's correspondence problem?

Given lists A and B, of K strings each from $\sum^*$, say

A = w , w w , ... w B= x11, x2,2x3, .3 . . . . .xkk

Does there exist a sequence of integers $i_1, i_2, \ldots\ldots\ldots\ldots i_r$ such that

$w_{i1}, w_{i2}, w_{i3} \ldots\ldots\ldots\ldots\ldots\ldots\ldots w_{im} = x_{i1}, x_{i2}, x_{i3}, \ldots\ldots\ldots\ldots x_{ir}$

The sequence of $i_1, i_2, \ldots\ldots\ldots\ldots\ldots i_m$ is a solution to this instance of PCP.

15. Define problem solvable in polynomial Time?

A Turing Machine M is said to be of time plexity T(n) if whenever m given an input w of length n, m halts after making atmost T(n) moves, regardless of whether or not m accepts.

16. Define the classes P and NP?

P consists of all those languages or problems accepted by some Turing Machine that runs in some polynomial amount of time, as a function of its input length.

NP is the class of languages or problems that are accepted by Nondeterministic TM's with a polynomial bound on the time taken along any sequence of non – deterministic choices.

17. Define NP – plete Problem?

A language L is NP – plete if the following statements are true.

a. L is in NP
b. For every language L^1 in NP there is a polynomial time reduction of L^1 to L

18. What are tractable problems?

The problems which are solvable by polynomial time algorithms are called tractable problems.

19. What are the properties of recursive enumerable sets Which are undecidable?

i) Emptiness
ii) Finiteness
iii) Regularity
iv) Context – freedom

20. What are the properties of recursive and Recursively Enumerable Language?

1. The plement of a Recursive language is Recursive.
2. The union of two recursive languages are recursive.
The union of two RE languages are RE.
3. If a language L and plement L are both RE, then L is recursive. .

KARPAGAM UNIVERSITY

(Under Section 3 of UGC Act 1956)

COIMBATORE-641 021

THEORY OF COMPUTATION

III-CSE A&B

UNIT IV & V

IMPORTANT TWO MARKS

1. Define Deterministic push down automata

A PDA P = (Q, Σ, δ, Γ, q0, Z0, F) to be deterministic iff

- δ (q,a,X) has at most one member for any q in Q, a in Σ or a = ε and X in |--
- if δ (q,a,X) is not empty, for some a in Σ, then δ(q,ε,X) must be empty.

2. **What is a multi-tape Turing machine?**

A multi-tape Turing machine consists of a finite control with k-tape heads and k-tapes; each tape is infinite in both directions. On a single move depending on the state of finite control and symbol scanned by each of tape heads ,the machine can change state print a new symbol on each cells scanned by tape head, move each of its tape head independently one cell to the left or right or remain stationary.

3. Is it true that NDPA is more powerful than that od DPDA? Justify your answer.

No, NPDA is not powerful than DPDA. Because NPDA may produce ambiguous grammar by reaching its final state or by emptying its stack. But DPDA produces only unambiguous grammar.

4. **What are the two major normal forms for context-free grammar?**

The two Normal forms are

- Chomsky Normal Form (CNF)
- Greibach Normal Form (GNF)

5. What are the various representation of TM?

The TM can be represented using:

- Instantaneous description.
- Transition table.
- Transition diagram.

6. How do you simplify the context-free grammar?

- First eliminate useless symbols, where the variable or terminals that do not appear in any derivation of a terminal string from the start symbol.
- Next eliminate a ε- production which is of the form A→ ε for some variable A.

- Eliminate unit productions, which are of the form A → B for variables A, B.
- Finally use any of the normal forms to get the simplified CFG.

7. What are the required fields of an instantaneous description or configuration of a TM.

The required fields of an instantaneous description or configuration of a TM are:

- The state of the TM
- The contents of the tape.
- The position of the tape head on the tape.

8. When a language is said to be recursive?

A language L is said to be recursive if there exists a Turing machine M that accepts L, and goes to halt state or else M rejects L.

9. Define Class NP Complete problem.

A language L is NP – complete if the following statements are true.

- L is in NP
- For every language L1 in NP there is a polynomial time reduction of L1 to L

10. **What are the properties of recursive and Recursively Enumerable Language?**

- The complement of a Recursive language is Recursive.
- The union of two recursive languages is recursive.
- The union of two RE languages is RE.
- If a language L and complement L are both RE, then L is recursive.

11. State the pumping lemma for CFL

Let L be any CFL. Then there is a constant n, depending only on L, such that if z is in L and |z| >=n, then z=uvwxy such that :
(i) |vx| >=1
(ii) |vwx| <=n and
(iii) for all i>=0 uv^iwx^iy is in L.

12. Differentiate PDA and TM.

PDA	**TM**
1. PDA uses a stack for storage.	1. TM uses a tape that is infinite.
2.The language accepted by PDA is CFL.	2. Tm recognizes recursively enumerable languages.

13. What are UTMs or Universal Turing machines?

Universal TMs are TMs that can be programmed to solve any problem, that can be solved by any Turing machine. A specific Universal Turing machine U is: Input to U: The

encoding "M " of a Tm M and encoding "w" of a string w. Behavior : U halts on input "M" "w" if and only if M halts on input w.

14. Define Nullable Variable?

Nullable variable in a CFG G =(V,T,P,S) can be defined as follows.

- Any variable A for which P contains the production A→ A, is nullable.
- If P contains the production A→ B1,B2, ……. Bn and B1, B2, ….Bn are nullable variables, then A is nullable.
- No other variables in V are nullable.

15. Let G=(V,T,P,S) with the productions given by

S → aSbS/ ε

B→abB

Eliminate the useless production.

Solution:

Remove B is useless production because of Variable is not reachable. S→ aSbS / ε

16. Write the procedure to eliminate the unit productions.

i) Find all variables B, for each A such that A→* B

ii) The new grammar G is obtained by letting into P all non-unit productions of P.

iii) For all A and B satisfying A→* B, add to p

A→y_1 / y_2 / y_3 /……/ y_n , where B →y_1 / y_2 / y_3 /……/ y_n is the set of productions in P.

17. Define CNF.

A CFG without any ε–production is generated by a grammar in which the productions are of the form. A →BC or A→ a, where A,B Σ V and a Σ T.

18. Define Turing Machine.

The Turing Machine is denoted by M = (Q, Σ, δ, Γ, q_0, B, F) Where

Q – Finite set of states

Σ - Finite set of input symbols

Γ - Finite set of stack symbols

δ - Transition function - Q x |-- →Q x |-- x {L,R}, Where L,R – Directions.

q_0 – Start State

B – A Start symbol of the |--, a blank

F – Final State.

19. When a language is said to be recursively enumerable?

A language is recursively enumerable if there exists a Turing Machine that accepts every string of the language and does not accept strings that are not in the language

20. Define post's correspondence problem?

An instance of PCP consists of two lists,

A = w_1, w_2, w_3, ... w_k

B = x_1, x_2, x_3, ...x_k of strings over some Σ.

This instance of PCP has a solution if there is any sequence of integers $i_1, i_2, \ldots\ldots\ldots i_m$ with $m \geq 1$. Such that $w_{i1}, w_{i2}, w_{i3} \ldots\ldots\ldots\ldots\ldots\ldots\ldots w_{im} = x_{i1}, x_{i2}, x_{i3}, \ldots\ldots\ldots\ldots x_{im}$. The sequence of $i_1, i_2, \ldots\ldots\ldots\ldots i_m$ is a solution to this instance of PCP.

21. **Define the classes P and NP?**

- P consists of all those languages or problems accepted by some Turing Machine that runs in some polynomial amount of time, as a function of its input length.
- NP is the class of languages or problems that are accepted by Nondeterministic TM's with a polynomial bound on the time taken along any sequence of non – deterministic choices.

22. What is GNF?

Every CFL L without ε can be generated by a grammar for which every production is of the form $A \rightarrow a\alpha$, where $A \Sigma V$, $a \Sigma T$, is a string of variables.

23. What are the properties of recursive enumerable sets which are undecidable?

- Emptiness
- Finiteness
- Regularity
- Context – freedom

24. Compare NPDA and DPDA.

NPDA	DPDA
1. NPDA is the standard PDA used in automata theory.	1. The standard PDA in practical situation is DPDA.
2. Every PDA is NPDA unless otherwise specified.	2. The PDA is deterministic in the sense ,that at most one move is possible from any ID.

Fourteen marks:

1. i) Convert the following CFG to CNF

S->ASA|aB

A->B|S

B->b| ε (10)

ii) Explain about Greibach Normal Form. (4)

2. Show that halting problem of Turing machine is undecidable. (14)

3. Explain the theorem of CFL to prove that CFL are closed under union,intersection and kleen closure. Prove that CFL's are closed under homomorphism (14)

4. i) Show that the intersection of two recursive languages is also recursive (7)

ii) Show that the union of two recursively enumerable language is also recursively enumerable (7)

5. Find the CNF for the following grammar

$S \rightarrow AS / as$

$A \rightarrow aab / E$

$S \rightarrow bba$ (14)

6. i) Prove that if L is a CFL and R is a regular set, then LR is a CFL. (10)

ii) Prove that CFL are closed under substitution (4)

7. Find whether the following languages are recursive or recursively enumerable.

i) Union of two recursive languages

ii) Union of two recursively enumerable languages.

iii) If L and complement of L are recursively enumerable.

iv) L_u (14)

8. i) Explain multiple track Turing machine with an example. (7)

ii) Construct the Turing machine to compute the concatenation function (7)

9. Define Universal language Lu. Show that Lu is recursively enumerable but not recursive. (14)

10) Explain about Greibach Normal Form. (4)

Chapter 6

Automata Theory

Instructions

A. The following abbreviations are used in this chapter.

FSM	-	Finite State Machine
DFSM	-	Deterministic Finite State Machine
NDFSM	-	Non-Deterministic Finite State Machine
PDM	-	Push Down Machine
DPDM	-	Deterministic Push Down Machine
NDPDM	-	Non-Deterministic Push Down Machine
TM	-	Turing Machine
UTM	-	Universal Turing Machine
CFG	-	Context Free Grammar
CF	-	Context Free
CFL	-	Context Free Language
CSG	-	Context Sensitive Grammar

B. In Transition diagrams, states are represented by circles.
The start state is represented by a circle pointed to by an arrow.
A final state is represented by a circle encircled by another.

C. In a CFG, unless stated otherwise, grammar symbol on the left hand side of the first production, is the start symbol.

1. The word 'formal' in formal languages means
(a) the symbols used have well-defined meaning
(b) they are unnecessary, in reality
(c) only the form of the string of symbols is significant
(d) none of the above

2. Let $A = \{0, 1\}$. The number of possible strings of length 'n' that can be formed by the elements of the set A is

(a) $n!$ (b) n^2 (c) n^n (d) 2^n

3. Choose the correct statements.

(a) Moore and Mealy machines are FSM's with output capability.

(b) Any given Moore machine has an equivalent Mealy machine.

(c) Any given Mealy machine has an equivalent Moore machine.

(d) Moore machine is not an FSM.

4. The major difference between a Moore and a Mealy machine is that

(a) the output of the former depends on the present state and the current input

(b) the output of the former depends only on the present state

(c) the output of the former depends only on the current input

(d) none of the above

5. Choose the correct statements.

(a) A Mealy machine generates no language as such.

(b) A Moore machine generates no language as such.

(c) A Mealy machine has no terminal state.

(d) For a given input string, length of the output string generated by a Moore machine is one more than the length of the output string generated by that of a Mealy machine.

***6.** The recognizing capability of NDFSM and DFSM

(a) may be different (b) must be different

(c) must be the same (d) none of the above

7. FSM can recognize

(a) any grammar (b) only CFG

(c) any unambiguous grammar (d) only regular grammar

8. Pumping lemma is generally used for proving

(a) a given grammar is regular

(b) a given grammar is not regular

(c) whether two given regular expressions are equivalent

(d) none of the above

***9.** 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) String of odd number of zeroes.

***10.** Which of the following pairs of regular expressions are equivalent?

(a) `1(01)*` and `(10)*1` (b) `x(xx)*` and `(xx)*x`

(c) `(ab)*` and `a*b*` (d) x^* and x^*x^*

11. Choose the correct statements.

(a) $A = \{a^n b^n \mid n=0,1,2,3,\ldots\}$ is a regular language.

(b) The set B, consisting of all strings made up of only a's and b's having equal number of a's and b's defines a regular language.

(c) $L(A^*B^*) \cap B$ gives the set A.

(d) None of the above

***12.** Pick the correct statements.

The logic of Pumping lemma is a good example of

(a) the Pigeon-hole principle (b) the divide and conquer technique

(c) recursion (d) iteration

***13.** The basic limitation of an FSM is that

(a) it can't remember arbitrary large amount of information

(b) it sometimes recognizes grammars that are not regular

(c) it sometimes fails to recognize grammars that are regular

(d) all of the above

14. Palindromes can't be recognized by any FSM because

(a) an FSM can't remember arbitrarily large amount of information

(b) an FSM can't deterministically fix the mid-point

(c) even if the mid-point is known, an FSM can't find whether the second half of the string matches the first half

(d) none of the above

15. An FSM can be considered a TM

(a) of finite tape length, rewinding capability and unidirectional tape movement

(b) of finite tape length, without rewinding capability and unidirectional tape movement

(c) of finite tape length, without rewinding capability and bidirectional tape movement

(d) of finite tape length, rewinding capability and bidirectional tape movement

16. TM is more powerful than FSM because

(a) the tape movement is confined to one direction

(b) it has no finite state control

(c) it has the capability to remember arbitrary long sequences of input symbols.

(d) none of the above

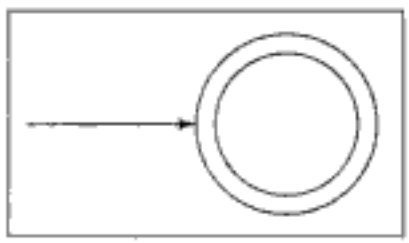


Fig. 6.1

***17.** The FSM pictured in Fig. 6.1 recognizes

(a) all strings (b) no string

(c) ε – alone (d) none of the above

18. The FSM pictured in Fig. 6.2 is a

(a) Mealy machine

(b) Moore machine

(c) Kleene machine

(d) none of the above

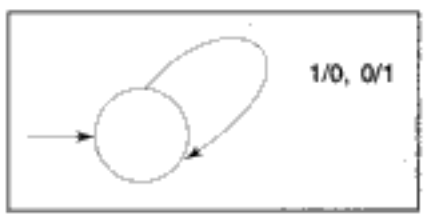


Fig. 6.2

19. The above machine

(a) complements a given bit pattern
(b) generates all strings of 0's and 1's
(c) adds 1 to a given bit pattern
(d) none of the above

20. The language of all words (made up of a's and b's) with at least two a's can be described by the regular expression

(a) `(a+b)*a(a+b)*a(a+b)*`
(b) `(a+b)*ab*a(a+b)*`
(c) `b*ab*a(a+b)*`
(d) `a(a+b)*a(a+b)*(a+b)*`

21. Which of the following pairs of regular expression are not equivalent?

(a) `(ab)*a and a(ba)*`
(b) `(a+b)* and (a*+b)*`
(c) `(a*+b)* and (a+b)*`
(d) none of the above

***22.** Consider the two FSM's in Fig. 6.3.
Pick the correct statement.

(a) Both are equivalent
(b) The second FSM accepts only ε
(c) The first FSM accepts nothing
(d) None of the above

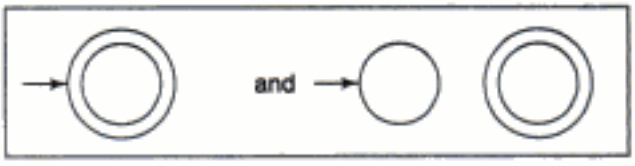


Fig. 6.3

23. Set of regular languages over a given alphabet set, is not closed under

(a) union
(b) complementation
(c) intersection
(d) none of the above

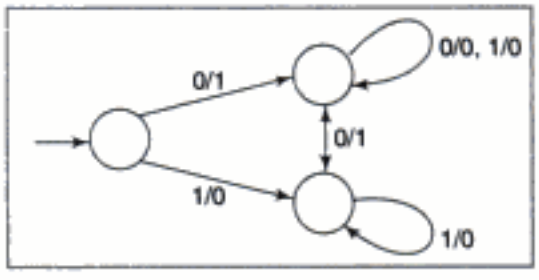


Fig. 6.4

***24.** The machine pictured in Fig. 6.4.

(a) complements a given bit pattern
(b) finds 2's complement of a given bit pattern
(c) increments a given bit pattern by 1
(d) changes the sign bit

25. For which of the following applications regular expressions can't be used?

(a) Designing compilers
(b) Developing text editors
(c) Simulating sequential circuits
(d) Designing computers

***26.** The FSM pictured in Fig. 6.5 recognizes

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

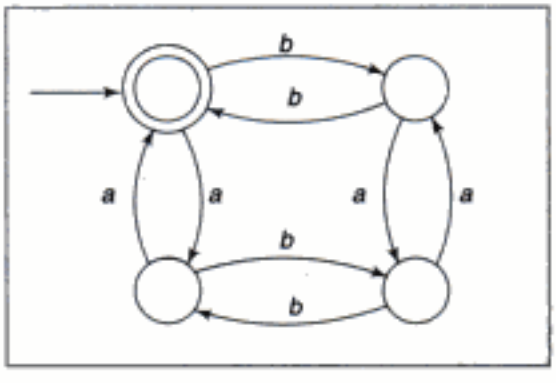


Fig. 6.5

27. Any given Transition graph has an equivalent

(a) regular expression
(b) DFSM
(c) NDFSM
(d) none of the above

28. The following CFG

$$S \rightarrow aS \mid bS \mid a \mid b$$

is equivalent to the regular expression

(a) (a*+b)* (b) (a+b)* (c) (a+b)(a+b)* (d) (a+b)*(a+b)

***29.** Any string of terminals that can be generated by the following CFG

S → XY
X → aX | bX | a
Y → Ya | Yb | a

(a) has at least one b
(b) should end in an 'a'
(c) has no consecutive a's or b's
(d) has at least two a's

***30.** The following CFG

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 a's and even number of a's

31. Let L(G) denote the language generated by the grammar G. To prove set $A = L(G)$,

(a) it is enough to prove that an arbitrary member of A can be generated by grammar G
(b) it is enough to prove that an arbitrary string generated by G, belongs to set A
(c) both the above comments (a) and (b) are to be proved
(d) either of the above comments (a) or (b) is to be proved

***32.** The set $\{a^n b^n \mid n = 1,2,3...\}$ can be generated by the CFG

(a) S → ab | aSb
(b) S → aaSbb | ab
(c) S → ab | aSb | ε
(d) S → aaSbb | ab | aabb

33. Choose the correct statements.

(a) All languages can be generated by CFG.
(b) Any regular language has an equivalent CFG.
(c) Some non-regular languages can't be generated by any CFG.
(d) Some regular languages can't be generated by any CFG.

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

(a) S → Sa | a
(b) S → abX
X → cY
Y → d | aX
(c) S → aSb| ab
(d) None of the above

35. CFG is not closed under

(a) union (b) Kleene star (c) complementation (d) product

36. The set $A = \{a^n b^n a^n \mid n = 1, 2, 3, \ldots\}$ is an example of a grammar that is

(a) regular (b) context free

(c) not context free (d) none of the above

37. Let $L1 = \{a^n b^n a^m \mid m, n = 1, 2, 3, \ldots\}$

$L2 = \{a^n b^n a^m \mid m, n = 1, 2, 3, \ldots\}$

$L3 = \{a^n b^n a^n \mid n = 1, 2, 3, \ldots\}$

Choose the correct statements.

(a) $L3 = L1 \cap L2$

(b) L1 and L2 are CFL but L3 is not a CFL

(c) L1 and L2 are not CFL but L3 is a CFL

(d) L1 is a subset of L3

38. $L = \{a^n b^n a^n \mid n = 1, 2, 3, \ldots\}$ 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) context free but whose complement is not CF

39. The intersection of a CFL and a regular language

(a) need not be regular (b) need not be context free

(c) is always regular (d) is always CF

40. A PDM behaves like an FSM when the number of auxiliary memory it has is

(a) 0 (b) 1 (c) 2 (d) none of the above

41. A PDM behaves like a TM when the number of auxiliary memory it has is

(a) 0 (b) 1 or more (c) 2 or more (d) none of the above

42. Choose the correct statements.

(a) The power of DFSM and NDFSM are the same.

(b) The power of DFSM and NDFSM are different.

(c) The power of DPDM and NDPDM are different.

(d) The power of DPDM and NDPDM are the same.

43. Which of the following is accepted by an NDPDM, but not by a DPDM?

(a) All strings in which a given symbol is present at least twice.

(b) Even palindromes (i.e. palindromes made up of even number of terminals).

(c) Strings ending with a particular terminal.

(d) None of the above

44. CSG can be recognized by a

(a) FSM (b) DPDM

(c) NDPDM (d) linearly bounded memory machine

45. Choose the correct statements.
(a) An FSM with 1 stack is more powerful than an FSM with no stack.
(b) An FSM with 2 stacks is more powerful than a FSM with 1 stack.
(c) An FSM with 3 stacks is more powerful than an FSM with 2 stacks.
(d) All of these.

46. Choose the correct statements.
(a) An FSM with 2 stacks is as powerful as a TM.
(b) DFSM and NDFSM have the same power.
(c) A DFSM with 1 stack and an NDFSM with 1 stack have the same power.
(d) A DFSM with 2 stacks and an NDFSM with 2 stacks have the same power.

47. Bounded minimalization is a technique for
(a) proving whether a primitive recursive function is Turning computable
(b) proving whether a primitive recursive function is a total function
(c) generating primitive recursive functions
(d) generating partial recursive functions

48. Which of the following is not primitive recursive but computable?
(a) Carnot function (b) Riemann function
(c) Bounded function (d) Ackermann function

49. Which of the following is not primitive recursive but partially recursive?
(a) Carnot function (b) Rieman function
(c) Bounded function (d) Ackermann function

50. Choose the correct statements.
(a) A total recursive function is also a partial recursive function.
(b) A partial recursive function is also a total recursive function.
(c) A partial recursive function is also a primitive recursive function.
(d) A primitive recursive function is also a partial recursive function.

51. A language L for which there exists a TM, T, that accepts every word in L and either rejects or loops for every word that is not in L, is said to be
(a) recursive (b) recursively enumerable
(c) NP-HARD (d) none of the above

52. Choose the correct statements.
(a) $L = \{a^n b^n a^n \mid n=1,2,3,\ldots\}$ is recursively enumerable.
(b) Recursive languages are closed under union.
(c) Every recursive language is recursively enumerable.
(d) Recursive languages are closed under intersection.

53. Choose the correct statements.
(a) Set of recursively enumerable languages is closed under union.
(b) If a language and its complement are both regular, then the language must be recursive.

(c) Recursive languages are closed under complementation.

(d) None of the above.

54. Pick the correct answers.

Universal TM influenced the concept of

(a) stored-program computers

(b) interpretive implementation of programming languages

(c) computability

(d) none of the above

55. The number of internal states of a UTM should be at least

(a) 1 (b) 2 (c) 3 (d) 4

56. The number of symbols necessary to simulate a TM with m symbols and n states is

(a) $m + n$ (b) $8mn + 4m$ (c) mn (d) $4mn + m$

57. Any TM with m symbols and n states can be simulated by another TM with just 2 symbols and less than

(a) $8mn$ states (b) $4mn + 8$ states (c) $8mn + 4$ states (d) mn states

58. The statement — "A TM can't solve halting problem" is

(a) true (b) false

(c) still an open question (d) none of the above

59. If there exists a TM which when applied to any problem in the class, terminates if the correct answer is yes, and, may or may not terminate otherwise is said to be

(a) stable (b) unsolvable (c) partially solvable (d) unstable

60. The number of states of the FSM, required to simulate the behaviour of a computer, with a memory capable of storing 'm' words, each of length 'n' bits is

(a) $m \times 2^n$ (b) 2^{mn} (c) 2^{m+n} (d) none of the above

61. The vernacular language English, if considered a formal language, is a

(a) regular language (b) context free language

(c) context sensitive language (d) none of the above

***62.** Let P, Q, and R be three languages. If P and R are regular and if PQ = R, then

(a) Q has to be regular (b) Q cannot be regular

(c) Q need not be regular (d) Q has to be a CFL

63. Consider the grammar

$S \rightarrow PQ \mid SQ \mid PS$

$P \rightarrow x$

$Q \rightarrow y$

To get a string of n terminals, the number of productions to be used is

(a) n^2 (b) $n + 1$ (c) $2n$ (d) $2n - 1$

64. Choose the correct statements.

A class of languages that is closed under

(a) union and complementation has to be closed under intersection

(b) intersection and complementation has to be closed under union

(c) union and intersection has to be closed under complementation

(d) all of the above

65. The following grammar is

$S \rightarrow a\alpha b \mid b\alpha c \mid aB$

$S \rightarrow aS \mid b$

$S \rightarrow \alpha bb \mid ab$

$b\alpha \rightarrow bdb \mid b$

(a) context free (b) regular (c) context sensitive (d) LR (k)

***66.** Which of the following definitions generates the same language as L, where

$L = \{x^n y^n,\ n \geq 1\}$?

I. $E \rightarrow xEy \mid xy$

II. $xy \mid x^+xyy^+$

III. x^+y^+

(a) **I** only (b) **I** and **II** (c) **II** and **III** (d) **II** only

***67.** A finite state machine with the following state table has a single input x and a single output z.

Present state	*Next state, z*	
	$x = 1$	$x = 0$
A	D, 0	B, 0
B	B, 1	C, 1
C	B, 0	D, 1
D	B, 1	C, 0

If the initial state is unknown, then the shortest input sequence to reach the final state C is

(a) 01 (b) 10 (c) 101 (d) 110

***68.** Let $A = \{0,1\}$ and $L = A^*$. Let $R = \{0^n1^n,\ n>0\}$. The languages L U R and R are respectively

(a) regular, regular (b) not regular, regular

(c) regular, not regular (c) not regular, not regular

***69.** Which of the following conversion 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) Non-deterministic Turing machine to deterministic Turing machine

***70.** An FSM can be used to add two given integers. This remark is

(a) true (b) false (c) may be true (d) none of the above

***71.** A CFG is said to be in Chomsky Normal Form (CNF), if all the productions are of the form `A` → `BC` or `A` → `a`. Let `G` be a CFG in CNF. To derive a string of terminals of length x, the number of productions to be used is

(a) $2x - 1$ (b) $2x$ (c) $2x + 1$ (d) 2^x

Answers

1. c	2. d	3. a, b, c	4. b	5. a, b, c, d
6. c	7. d	8. b	9. a, b, c	10. a, b, d
11. c	12. a	13. a	14. a, b, c	15. b
16. c	17. c	18. a	19. a	20. a, b, c
21. d	22. d	23. d	24. c	25. a, d
26. c	27. a, b, c	28. b, c, d	29. d	30. a
31. c	32. a, d	33. b, c	34. c	35. c
36. c	37. a, b	38. b, c	39. c, d	40. a
41. c	42. a, c	43. b	44. d	45. a, b
46. a, b, d	47. c	48. d	49. d	50. a, d
51. b	52. a, b, c, d	53. a, b, c	54. a, b, c	55. b
56. d	57. a	58. a	59. c	60. b
61. b	62. c	63. d	64. a, b	65. c
66. a	67. b	68. c	69. c	70. b
71. a				

Explanations

6. DFSM is a special case of NDFSM. Corresponding to any given NDFSM, one can construct an equivalent DFSM. Corresponding to any given DFSM, one can construct an equivalent NDFSM. So they are equally powerful.

9. Strings of odd number of zeroes can be generated by the regular expression `(00)*0`. Pumping lemma can be used to prove the non-regularity of the other options.

10. Two regular expressions `R1` and `R2` are equivalent if any string that can be generated by `R1` can be generated by `R2` and vice-versa. In option (c), `(ab)*` will generate `abab`, which is not of the form $a^n b^n$ (because a's and b's should come together). All other options are correct (check it out!).

12. Pigeon-hole principle is that if 'n' balls are to be put in 'm' boxes, then at least one box will have more than one ball if $n>m$. Though this is obvious, still powerful.

13. That's why it can't recognize strings of equal number of a's and b's, well-formedness of nested parenthesis etc.

17. Here the `final` state and the `start` state are one and the same. No transition is there. But by definition, there is an (implicit) ε-transition from any state to itself. So, the only string that could be accepted is ε.

22. Refer Qn. 17. In the second diagram, the final state is unreachable from the `start` state. So not even ε could be accepted.

24. Let `011011` be the input to the FSM and let it be fed from the right (i.e., least significant digit first). If we add `1` to `011011` we should get `011100`. But did we obtain it? Whenever we add `1` to an `1`, we make it `0` and carry `1` to the next stage (state) and repeat the process. If we add `1` to a `0`, then first make it `1` and all the more significant digits will remain the same, i.e., a `0` will be `0` and an `1` will be `1`. That's what the given machine does. Hence the answer is (c).

26. Here the initial and the final states are one and the same. If you carefully examine the transition diagram, to move right you have to consume a '*b*', to move left a '*b*', to go up an '*a*' and to go down an '*a*'. Whenever we move right, we have to move left at some stage or the other, to get back to the initial-cum-final state. This implies, a '*b*' essentially has an associated another '*b*'. Same is the case with '*a*' (since any up (down) has a corresponding down (up)). So, even number of *a*'s and *b*'s have to be present.

29. `S` is the `start` state. `X`→`a`, `Y`→`a` are the only productions that could terminate a string derivable from `X` and `Y` respectively. So at least two *a*'s have to come anyway. Hence the answer is (d).

30. We have `S` → `aB` → `aaBB` → `aabB` → `aabb`.

So (b) is wrong. We have

`S` → `aB` → `ab`

So (c) is wrong.

A careful observation of the productions will reveal a similarity. Change `A` to `B`, `B` to `A`, `a` to `b` and `b` to `a`. The new set of productions will be the same as the original set. So (d) is false and (a) is the correct answer.

32. Option (b) is wrong because it can't generate `aabb` (in fact any even power). Option (c) is wrong since it generates ε also. Both (a) and (d) are correct.

34. Option (c) generates the set $\{a^n b^n, \quad n=1,2,3,\ldots\}$ which is not regular. Options (a) and (b) being left linear and right linear respectively, should have equivalent regular expressions.

60. Totally there are *mn* bits. Each bit will be in one of the two possible states – `1` or `0`. So the entire memory made up of *mn* bits will be in one of the possible 2^{mn} states.

62. For example, $P = a^*; \quad Q = a^n b^n b^*; \quad R = PQ = a^* b^*$

64. The first two options can be proved to be correct using De Morgan's laws. Option (c) can be disproved by the following counter-example. Let the universal set U be {a, b, c, d,}. Let A = {{a}, {d}, {a, d}, {b, d}, {a, b, d}, {}}. A is closed under union and intersection but is not closed under complementation. For example complement of {a, d} is {b, c}, which is not a member of A.

66. II generates strings like `xxyyy`, which are not supposed to be. III generates strings like `xyy`, which are not supposed to be. I can be verified to generate all the strings in `L` and only those.

67. Draw the transition diagram and verify that the string `10` from `A`, leads to `C`.

68. L is the set of all possible strings made up of 0's and l's (including the null string). So, L U R is L, which can be generated by the regular expression `(a+b)*`, and hence a regular language. `R` is not a regular expression. This can be proved by using Pumping Lemma or simply by the fact that finite state automata, that recognizes regular expressions, has no memory to record the number of 0's or l's it has scanned. Without this information 0^n1^n cannot be recognized.

69. In general, a language (or equivalently the machine that recognizes it) cannot be converted to another language that is less powerful.

70. FSM is basically a language acceptor. As such, it does not have any output capability. So it cannot add and output the result.

71. This can be proved using induction.