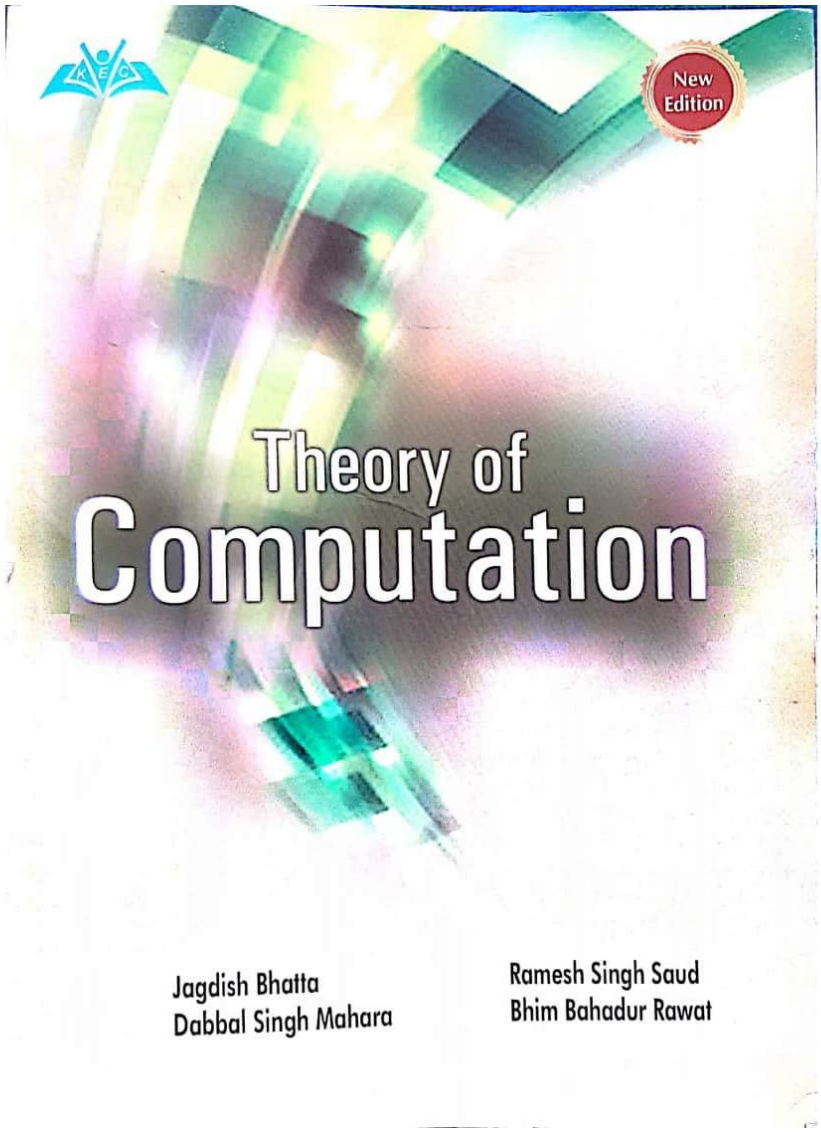


Theory of Computation

Course Title: Theory of Computation
Course No: CSC257
Nature of the Course: Theory + Lab
Semester: IV

Full Marks: 60 + 20 + 20
Pass Marks: 24 + 8 + 8
Credit Hrs: 3

Course Objectives

This course presents a study of Finite State Machines and their languages. It covers the details of finite state automata, regular expressions, context free grammars. More, the course includes design of the Push-down automata and Turing Machines. The course also includes basics of undecidabilty and intractability.

Course Description

The main objective of the course is to introduce concepts of the models of computation and formal language approach to computation. The general objectives of this course are to, introduce concepts in automata theory and theory of computation, design different finite state machines and grammars and recognizers for different formal languages, identify different formal language classes and their relationships, determine the decidability and intractability of computational problems.

Course Details **(3 Hrs.)**

Unit I: Basic Foundations

1.1. Review of Set Theory, Logic, Functions, Proofs
1.2. Automata, Computability and Complexity: Complexity Theory, Computability Theory, Automata Theory
1.3. Basic concepts of Automata Theory: Alphabets, Power of Alphabet, Kleen Closure Alphabet, Positive Closure of Alphabet, Strings, Empty String, Substring of a string, Concatenation of strings, Languages, Empty Language **(8 Hrs.)**

Unit II: Introduction to Finite Automata

2.1 Introduction to Finite Automata, Introduction of Finite State Machine
2.2 Deterministic Finite Automata (DFA), Notations for DFA, Language of DFA, Extended Transition Function of DFA Non-Deterministic Finite Automaton (NFA), Notations for NFA, Language of NFA, Extended Transition
2.3 Equivalence of DFA and NFA, Subset-Construction
2.4 Method for reduction of NFA to DFA, Theorems for equivalence of Language accepted by DFA and NFA
2.5 Finite Automaton with Epsilon Transition (ε - NFA), Notations for ε - NFA, Epsilon Closure of a State, Extended Transition Function of ε – NFA, Removing Epsilon Transition using the concept of Epsilon Closure, Equivalence of NFA and ε –NFA, Equivalence of DFA and ε – NFA
2.6 Finite State Machines with output: Moore machine and Mealy Machines **(6 Hrs.)**

Unit III: Regular Expressions

3.1 Regular Expressions, Regular Operators, Regular Languages and their applications, Algebraic Rules for Regular Expressions
3.2 Equivalence of Regular Expression and Finite Automata, Reduction of Regular Expression to ε – NFA, Conversion of DFA to Regular Expression
3.3 Properties of Regular Languages, Pumping Lemma, Application of Pumping Lemma, Closure Properties of Regular Languages over (Union, Intersection, Complement) Minimization of Finite State Machines: Table Filling Algorithm **(9 Hrs.)**

Unit IV: Context Free Grammar

4.1 Introduction to Context Free Grammar (CFG), Components of CFG, Use of CFG, Context Free Language (CFL)
4.2 Types of derivations: Bottomup and Topdown approach, Leftmost and Rightmost, Language of a grammar
4.3 Parse tree and its construction, Ambiguous grammar, Use of parse tree to show ambiguity in grammar

4.4 Regular Grammars: Right Linear and Left Linear, Equivalence of regular grammar and finite automata

4.5 Simplification of CFG: Removal of Useless symbols, Nullable Symbols, and Unit Productions; Chomsky Normal Form (CNF), Greibach Normal Form (GNF), Backus- Naur Form (BNF)

4.6 Context Sensitive Grammar, Chomsky Hierarchy Pumping Lemma for CFL, Application of Pumping Lemma, Closure Properties of CFL

Unit V: Push Down Automata **(7 Hrs.)**

5.1 Introduction to Push Down Automata (PDA), Representation of PDA, Operations of PDA, Move of a PDA, Instantaneous Description for PDA

5.2 Deterministic PDA, Non Deterministic PDA, Acceptance of strings by PDA, Language of PDA

5.3 Construction of PDA by Final State , Construction of PDA by Empty Stack,

5.4 Conversion of PDA by Final State to PDA accepting by Empty Stack and vice-versa, Conversion of CFG to PDA, Conversion of PDA to CFG

Unit VI: Turing Machines (10 Hrs.)

6.1 Introduction to Turing Machines (TM), Notations of Turing Machine, Language of a Turing Machine, Instantaneous Description for Turing Machine, Acceptance of a string by a Turing Machines

6.2 Turing Machine as a Language Recognizer, Turing Machine as a Computing Function, Turing Machine with Storage in its State, Turing Machine as a enumerator of stings of a language, Turing Machine as Subroutine

6.3 Turing Machine with Multiple Tracks, Turing Machine with Multiple Tapes, Equivalence of Multitape-TM and Multitrack-TM, Non-Deterministic Turing Machines, Restricted Turing Machines: With Semi-infinite Tape, Multistack Machines, Counter Machines

6.4 Curch Turing Thesis, Universal Turing Machine, Turing Machine and Computers, Encoding of Turing Machine, Enumerating Binary Strings, Codes of Turing Machine, Universal Turing Machine for encoding of Turing Machine

Unit VII: Undecidability and Intractability **(5 Hrs.)**

7.1 Computational Complexity, Time and Space complexity of A Turing Machine, Intractability

7.2 Complexity Classes, Problem and its types: Absract, Decision, Optimization

7.3 Reducibility, Turing Reducible, Circuit Satisfiability, Cook's Theorem,

7.4 Undecidability, Undecidable Problems: Post's Correspondence Problem, Halting Problem and its proof, Undecidable Problem about Turing Machines

Laboratory Works:

The laboratory work consists of design and implementation of finite state machines like DFA, NFA, PDA, and Turing Machine. Students are highly recommended to construct Tokenizers/ Lexers over/for some language. Students are advised to use regex and Perl (for using regular expressions), or any other higher level language for the laboratory works.

Text Books:

1. **John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd Edition, Pearson - Addison-Wesley.**

Reference Books:

1. Harry R. Lewis and Christos H. Papadimitriou, Elements of the Theory of Computation, 2nd Edition, Prentice Hall.
2. Michael Sipser, Introduction to the Theory of Computation, 3rd Edition, Thomson Course Technology
3. Efim Kinber, Carl Smith, Theory of Computing: A Gentle introduction, Prentice- Hall.
4. John Martin, Introduction to Languages and the Theory of Computation, 3rd Edition, Tata McGraw Hill.
5. Kenneth H. Rosen, Discrete Mathematics and its Applications to Computers Science, WCB/Mc-Graw Hill.

1 Chapter

BASIC FOUNDATIONS

2 Chapter INTRODUCTION TO FINITE AUTOMATA

3 Chapter REGULAR EXPRESSIONS

4 Chapter CONTEXT FREE GRAMMAR

5 Chapter PUSHDOWN AUTOMATA

6 Chapter TURING MACHINE

7 Chapter UNDECIDABILITY AND INTRACTABILITY

CHAPTER 1

BASIC FOUNDATIONS

CHAPTER OUTLINES

After studying this Chapter you should be able to:

- Review of Set Theory
- Logic
- Functions
- Proof Technique
- Automata, Computability and Complexity
- Basic Concepts of Automata Theory

Review of Set Theory

Set

A set is any well defined un-ordered collection of distinct objects, called as elements or members of the set. Some examples of a set are:

(a) All vowel alphabets

(b) All Zone of Nepal

(c) All odd numbers

Notations

In general capital letters **A, B, C, X, Y, Z, ...** are used to denote **sets** while the small letter **a, b, c, x, y, z,...** are used to denote the members of the sets unless other wise stated. If 'a' is an element of a set A we write this as $a \in A$ and read as "a belongs to the set A". Again if 'a' is not an element of the set A we write this as $a \notin A$ and read as "a does not belong to the set A."

If A be the set of three numbers 1, 2 and 3 then it is written as:

$$A = \{1, 2, 3\}$$

Here, the elements of the set must be enclosed within the Corley bracket { }.

Representation of a Set

A set may be specified by the following methods:

1. Description Method
2. Tabulation Method
3. Rule Method or Set-builder method

Description Method

In this method a set is specified by a verbal description.

For example, the set S of numbers 1, 2 and 3 is designated as:

S = the set of positive integers less than 4.

Tabulation Method

In this method a set is specified by listing all elements in a set. Thus, we can write the set S of numbers 1, 2 and 3 as:

$S = \{1, 2, 3\}$

Note that each of the sets {1, 2, 3}, {2, 3, 1} and {3, 1, 2} are the same.

Rule Method

In this method a set is specified by stating a characteristic property common to all elements in the set. Referring to the above example, we can express the set S as:

$S = \{x \rightarrow x \text{ is an integer and } 1 \leq x \leq 3\}$

This is read as the set of all elements x such that x is a positive integer less than 4. The vertical bar denotes 'such that'.

The representation of a set by the rule method is suitable when the set has a larger number of elements. For example, if we take all men in Kathmandu city using 'close-up tooth paste' it will be inconvenient to write the names of all persons within braces. But we can write this set briefly as:

S = {x: x is a man in Kathmandu who uses close-up tooth paste}

Definitions of Sets Terminologies

Finite Set

A set consisting of finite number of elements is called finite set. Thus, the set of days in a week is a finite set.

Infinite set

A set consisting of infinite number of elements is called infinite set. A set of all odd numbers is an infinite set.

Thus, A = {1, 3, 5, ...} is an infinite set.

Empty Set

A set without any element is called an empty set or null set or void set and is usually denoted by the symbol ϕ or, { }.

(i) P = {x: x is the male students of Padma Kanya Campus} = ϕ

(ii) B = {b: b is a married bachelor} = ϕ

Note that the set {0} is not an empty set since it contains zero as its element. Also *any two empty sets are equal.*

Unit Set

A set consisting of only one element is called a unit set or *singleton set*. Examples of a unit set are:

(i) S = {0} is a unit set with single element zero.

(ii) N = {2} is a unit set with single element 2.

Universal Set

The set of all the entities in the current context is called the **universal set**, or simply the **universe**. It is denoted by U. The context may be a integers, for example, where the Universal set is limited to the particular entities under its consideration. Also, it may be any arbitrary problem, where we clearly know where it is applied.

Subset

A set that consists of some or all elements of another set is called subset of the set. The set A is subset of the set B if and only if each elements of A is also an element of B. In symbols we write $A \subset B$ (and read as 'A is subset of B') if and only if $x \in A$ implies that $x \in B$.

Some examples of a subset are as follows:

(i) If A = {1, 2} and B = {1, 2, 3} then, $A \subset B$.

(ii) Every set is a subset of itself that is $A \subset A$.

(iii) Null set $\emptyset$ is a subset of any set S.

(iv) If A = {a, b, c} and B = {b, a, c}, then $A \subset B$ and $B \subset A$. Then A = B.

Power Set

The possible subsets in a set {a} will be ϕ and {a}. Hence, the number of subsets that can be formed out of a set consisting of one element is $2^1 = 2$.

Similarly, the possible subsets in a set {a, b} will be ϕ, {a}, {b} and {a, b} which are $4 = 2^2$ in number. If we take a set {a, b, c} with the three elements a, b and c then its possible subsets will be ϕ, {a}, {b}, {c}, {a, b}, {b, c}, {c, a} and {a, b, c} which are $8 = 2^3$ in number.

Proceeding in this manner, we conclude by induction that *a set with n elements has 2^n subsets.* This includes the null set and the given set.

So, the set of possible subsets from any set is known as power set of that set.

Let A={a, b, c} be any set then power set of A is

P(A)={ϕ, {a}, {b}, {c}, {a, b}, {b, c}, {c, a} and {a, b, c}}

Set Operations

Sets may be combined and operated in various ways to form new sets. The basic operations on sets are:

1. Union
2. Intersection
3. Complementation
4. Difference

Union

The union of two sets A and B, denoted by $A \cup B$, is the set of only those elements which belongs to either A or B or both A and B. Symbolically, we write this as:

$A \cup B = \{x \mid x \in A \text{ or } x \in B \text{ or } x \in \text{ both A and B}\}$

Here $A \cup B$ is also called the logical sum of A and B and sometimes read as 'A cup B'. The shaded portion on figure 2.2 represents $A \cup B$ which shows the set of objects consisting of the elements of at least one of the sets A and B. Some examples of the union of two sets are as follows:

Example: If A = {a, b} and B = {p, q, r},

then $A \cup B = \{a, b, p, q, r\}$

Intersection

The intersection of two sets A and B, denoted by $A \cap B$, is the set of only those elements which belongs to both A and B. In symbol, we write

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$

$A \cap B$ is sometimes read as 'A cap B' and is also known as the logical multiplication of A and B. The shaded area in figure 2.3 shows the common elements of A and B and form a new set $A \cap B$. Some examples of the intersection of two sets are as follows:

Example: If A = {2, 3, 4} and B = {3, 5} then, $A \cap B = \{3\}$

In this case the sets R and S are called disjoint sets.

Complement of a Set

Let A be the subset of a universal set U. Then the complement of A with respect to U is the set of all those elements of U which do not belong to A and is denoted by $\bar{A}$ or A' or A^c. In symbols, we write this as:

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

The shaded portion in figure 2.4 stands for the complement of A. Some examples of the complement sets are as follows:

Example: If U = {1, 2, 3, 4, 5} and A = {2, 4} then $A' = \{1, 3, 5\}$

Difference

Let A and B be two sets and each set is the subset of a universal set U. Then, A difference B denoted by A – B, is the set of all those elements which belong to A but not B. In symbols, we write this as:

$A - B = \{x \mid x \in A \text{ and } x \notin B\}$

Also, $B - A = \{x \mid x \in B \text{ and } x \notin A\}$

The difference, A – B, which is sometimes called 'A minus B' is also known a complement of B with respect to A. The shaded portion in figure 2.5 represents the difference of A and B. The following are some examples of the difference between two sets.

Example: If A = {a, b, x, y} and B = {c, d, x, y} then,

A – B = {a, b} and B – A = {c, d}

Thus, A – B ≠ B – A

LOGIC

Logic is the study of the form of valid inference, and laws of truth. A valid inference is one where there is a specific relation of logical support between the assumptions of the inference and its conclusion. In ordinary discourse, inferences may be signified by words such as *therefore, thus, hence, ergo,* and so on. There has always been a strong influence from mathematical logic on the field of computer science. From the beginning of the field it was realized that technology to automate logical inferences could have great potential to solve problems and draw conclusions from facts.

Logic is generally considered formal when it analyzes and represents the form of any valid argument type. The form of an argument is displayed by representing its sentences in the formal grammar and symbolism of a logical language to make its content usable in formal inference.

There are two main "branches" of logic, formal and informal. Informal logic is the logic that relates more to rhetoric, whereas formal logic is mainly divided into categorical logic and Boolean, or truth-valued logic. Informal logic is the study of natural language arguments including fallacies. A good way to think about formal logic is mathematical reasoning - categorical logic relates directly to set theory, and truth-valued logic relates to Boolean algebra, which is kind of a blanket term for logic relating to "trues" and "falses".

Each of the computing devices we use today has number of computations with in it. Such computations consists various logic to perform some desired operation.

Propositional Logic

Propositional Logic also known as sentential logic is the branch of logic that studies way of joining or modifying propositions to form more complicated propositions as well as logical relationships. The propositional logic represents knowledge or informal sentence in terms of propositions. In Propositional Logic, there are two types of sentences- simple sentences and compound sentences. Simple sentences express atomic propositions about the world. Compound sentences express logical relationships between the simpler sentences of which they are composed.

Propositions

Proposition is a declarative sentence that is either true or false, but not both. Example:

- 2 + 2 = 5. (False), is a proposition.
- 7 – 1 = 6. (True), is a proposition.
- odd numbers are divisible by 2 (false), is a proposition.
- Kathmandu is the capital of Nepal. (True), is a proposition.
- Open the door. Not a proposition.

The essential property of proposition is that, it is either true or false but not both.

Let us try to analyze the sentences below:

x > 15, go there, Who are you?

The above sentences are not propositions since we cannot say whether they are true or false.

Predicate Logic

Any declarative statements involving variables often found in mathematical assertion and in computer programs, which are neither true nor false when the values of variables are not specified is called predicate.

The logic involving predicates is called Predicate Logic or Predicate calculus similar to logic involving propositions is Propositional Logic or Propositional Calculus.

Let's take a statement 5>9, this is a propositional statement because it is false. Now let's take a statement "x > 4". Is this statement a proposition? The answer is no. The statement may be either true or false depending upon the value of x (variable). We can say that any statement involving variable is not proposition.

The predicate "x > 4" has two parts. The first part, variable x is subject of statement and another is relation part ">4" called "predicate" refers to a property that the subject of statement have. We can denote the statement "x>4" by P(x) where P is predicate ">4" and x is the variable. We also call P as a propositional function where P(x) gives value of P at x. Once value is assigned to the propositional function then we can tell whether it is true or false i.e. a proposition.

For e.g. if we put the value of x as 3 and 7 then we can conclude that P(3) is false since 3 is not greater than 4 and p(7) is true since 7 is greater than 4.

We can also denote a statements with more than one variable using predicate like for the statement "x = y" we can write P(x,y) such that P is the relation "equals to" . Similarly the statements with higher number of variables can be expressed.

Thus a predicate is a sentence that contains a finite number of variables and becomes a proposition when specific values are substituted for the variables.

Example 1: Let P(x): x+2<10, find the truth value of P(5)and P(9).

Solution:

Given,

P(x) : x + 2 < 10

When x = 5,

P(5) : 5 + 2 < 10

7 < 10 (true)

When x = 9,

P(9) : 9 + 2 < 10

11 < 10 (false)

Example 2: Let F(x, y) : x = y + 6. Find the truth value of F(1, 5) and F(5, 0).

Solution:

Given,

F(x, y) : x = y + 6

For F(1, 4), Set x = 1 and y = 5 we get,

F(1, 5): 1 = 5 + 6

1 = 11 (False)

Similarly,

For F(6, 0), Set x = 6, y = 0 we get,

F(6, 0) : 6 = 0 + 6

i.e. 6 = 6 (true)

∴ F(1, 5) is false and F(6, 0) is true for F(x, y) : x = y + 6.

Quantifiers

Quantifiers are the tools that change the propositional function into a proposition. These are the word that refers to quantities such as "some" or "all" and indicates how frequently a certain statement is true. Construction of propositions from the predicates using quantifiers is called quantification. The variables that appear in the statement can take different possible values and all the possible values that the variable can take forms a domain called "Universe of Discourse" or "Universal set".

There are two types of quantifier Universal quantifier and Existential quantifier.

Universal Quantifier

The phrase "for all" denoted by ∀, is called universal quantifier. The process of converting predicate into proposition using universal quantifier is called universal quantification. So, the universal quantification of P(x), denoted by ∀x P(x), is a proposition where "P(x) is true for all the values of x in the universe of discourse".

We can represent the universal quantification by using the English language like: "for all x P(x) holds" or "for every x P(x) holds" or "for each x P(x) holds".

Example

Take universe of discourse a set of all students of Kathmandu College.

P(x) represents: x takes Discrete Mathematics class.

Here universal quantification is ∀x P(x), which represent the English sentence "all students of Kathmandu college take Discrete Mathematics class", and now it is a proposition.

The universal quantification is conjunction of all the propositions that are obtained by assigning the value of the variable in the predicate. Going back to above example if universe of discourse is a set {Ram, Shyam, Hari, Sita} then the truth value of the universal quantification is given by P(ram) ∧ P(Shyam) ∧ P(Hari) ∧ P(Sita) i.e. it is true only if all the atomic propositions are true.

Existential Quantifier

The phrase "there exist", denoted by ∃, is called existential quantifier .The process of converting predicate into proposition using existential quantifier is called existential quantification. The existential quantification of P(x), denoted by ∃x P(x), is a proposition where "P(x) is true for some values of x in the universe of discourse". The other forms of representation include "there exists x such that P(x) is true" or "P(x) is true for at least one x".

Example: For the same problem given in universal quantification ∃x P(x) is a proposition is represent like " some students of Kathmandu College take Mathematics class".

The existential quantification is the disjunction of all the propositions that are obtained by assigning the values of the variable from the universe of discourse. So the above example is equivalent to P(Ram) ∨ P(Shyam) ∨ P(Hari) ∨ P(Sita), where all the instances of variable are as in example of universal quantification. Here if at least one of the students takes graphics class then the existential quantification results true.

Example 3: Let Z, the set of integer, be the universe of discourse and consider the statements

(i) $\forall x \in Z, x^2 = x$ **(ii)** $\exists x \in Z, x^3 = x$

Find the truth values of each of the statements.

Solution:

Let P(x) : $x^3 = x$ then

∀xP(x) is false because $2 \in z$, P(2) : $2^2 = 2$ is false and ∃ xP(x) is true because at least one proposition is true i.e. $1 \in z$, P(1) : $1^2 = 1$ is true.

Example 4: Let N = {1, 2, 3, 4, 5, 6, 7, 8, 9} be set of natural number. Determine the truth value of each of the following statements.

(a) $\exists x \in N, x + 5 = 12$
(b) $\forall x \in N, x + 4 < 15$
(c) $\forall x \in N, x + 5 \leq 10$

Solution:

Given,

$N = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

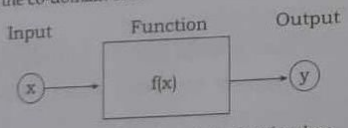
(a) $\exists x \in N, x + 5 = 12$ is true
for if $x = 7$ then $x + 5 = 12$
or, $7 + 5 = 12$
i.e. $12 = 12$ (true)

(b) $\forall x \in N, x + 4 < 15$ is true
for every $x \in N$ satisfies $x + 4 < 15$

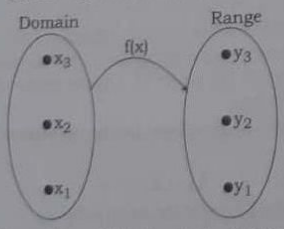
(c) $\forall x \in N, x + 5 \leq 10$ is false for $6 \in N$, $6 + 5 \leq 10$
$\Rightarrow 11 \leq 10$ (false)

FUNCTIONS

If A and B are two sets, a function f from A to B is a rule that assigns to each element x of A an element f(x) of B. For a function f from A to B, we call A the domain of f and B the co-domain of f.



(a) One way of showing what a function does



(b) A second way of showing what a function does

Every function consists of some sort of computation. The computation with in the function maps input instances to output. For example a function $f(x) = x^2$. Then for input A = {2, 3, 4}, f(A)= {4, 9, 16}.

PROOF TECHNIQUE

A theorem is a mathematical statement that can be shown to be true. A proof of given theorem is said to be well founded if its steps of mathematical statement can be present on argument that makes the theorem true.

This method of understanding correctness of statement by applying sequence of logical argument is known as proof of statement.

Problem solving or proving is not just a science so there is no hard and fast rule that is applied in problem solving. However there are some guiding methods that help us to solve different kinds of problems.

Here we will discuss different methods of proving implication $p \rightarrow q$.

Direct Proofs

The implication $p \rightarrow q$ can be proved by showing that if p is true, then q must also be true. To carry out such a proof, we assume that hypothesis p is true and using information already available if conclusion q becomes true then argument becomes valid.

Example 5: If a and b are odd integers, then a + b is an even integer.

Proof

We know the fact that if a number is even then we can represent it as 2k, where k is an integer and if the number is odd then it can be written as 2l + 1, where l is an integer. Assume that a = 2k + 1 and b = 2l + 1, for some integers k and m. then a + b = 2k + 1 + 2l + 1 = 2(k + l + 1), here (k + l + 1) is an integer. Hence a + b is even integer.

Example 6: Prove that: If n is an odd integer, then n^2 is an odd integer.

Solution:

Let $p \rightarrow q$: if n is an odd integer, then n^2 is an odd integer.

We assume that hypothesis of this implication is true i.e. suppose, n is odd then n can be expressed as n = 2k + 1

Now,

$$n^2 = (2k + 1)^2 = 4k^2 + 4k + 1$$
$$= 2(2k^2 + 2k) + 1$$

Since $2(2k^2 + 2k)$ is even but one more than even is odd

$\therefore n^2$ is an odd integer.

Example 7: Prove that sum of two rational number is rational number.

Proof:

Suppose that A and B are two rational number then by definition of rational number, it follows that there are two integers p and q where $q \neq 0$ such that $A = \frac{p}{q}$ and integers x and y with $y \neq 0$ such that $B = \frac{x}{y}$.

Now,

$$A + B = \frac{p}{q} + \frac{x}{y}$$

$$= \frac{py + ax}{ay}$$

Since: $q \neq 0$ and $y \neq 0$, it follows that $ay \neq 0$. Here, we have expressed A + B as the ratio of two integers py + qx and qy with $ay \neq 0$

$\therefore$ A + B is rational.

Example 8: Using direct proof, prove that for every positive integer n, $n^3 + n$ is even.

Proof: Case 1: Suppose, n is even, then n = 2k for some k.

Now,

$n^3 + n = (2k)^3 + 2k = 8k^3 + 2k = 2(4k^3 + k)$ which is even.

Case II: Suppose n is odd, then it can be expressed as

n = 2k + 1 for some positive integer k.

Now,

$$n^3 + n = (2k + 1)^3 + (2k + 1)$$
$$= (8k^3 + 12k^2 + 6k + 1) + (2k + 1)$$
$$= 8k^3 + 12k^2 + 8k + 2$$
$$= 2(4k^2 + 6k^2 + 4k + 1)$$

which is even. Hence, for any positive integer n, $n^3 + n$ is even.

Indirect Proofs

We have $p \rightarrow q \equiv \neg q \rightarrow \neg p$ i.e. contra positive of implication is equivalent to the implication. So, the implication $p \rightarrow q$ can be proved by showing that its contrapositive $\neg q \rightarrow \neg p$ is true. We prove the implication $p \rightarrow q$ by assuming that the conclusion (q) is false and using the known facts we show that the hypothesis (p) is also false.

Example 9: If the product of two integers a and b is even, then either a is even or b is even.

Proof

Suppose both a and b are odd, then we have a = 2k + 1 and b = 2l + 1.

So ab = (2k +1) (2l + 1) = 4kl +2k +2l +1 = 2(2kl + k + l) + 1, i.e. ab is an odd number. Hence, both a and b being odd implies ab is also odd. This is indirect proof.

Example 10: Using indirect proof, show that if 3n + 2 is odd then n is odd.

Proof: Let $p \rightarrow q$: if 3n + 2 is odd then n is odd.

Assume that conclusion of this implication is false i.e. n is even then we can express n as:

n = 2k for some integer k

It follows that

$$3n + 2 = 3 \times 2k + 2$$
$$= 6k + 2$$
$$= 2(3k + 1)$$

$\therefore$ 3n + 2 is even since 2(3k + 1) is multiple of 2.

Hence, if 3n + 2 is odd then n is odd.

Proofs by Contradiction

The steps in proof of implication $p \rightarrow q$ by contradiction are:

- Assume $p \wedge \neg q$ is true.
- Try to so that the above assumption $(p \wedge \neg q)$ is false
- When the assumption is found to be false then implication $p \rightarrow q$ is true
- Since $p \rightarrow q$ is equivalent to $\neg p \vee q$ and negation of $\neg p \vee q$ is $p \wedge \neg q$ (By De Morgan's Law), so if our assumption is false then its negation is true.

Alternately, Contradict the statement and show that this leads to the false conclusion; if this is true then the contradicted statement must be false (since $\neg p \rightarrow F$ is true only if $\neg p$ is false), hence the statement is true.

Example 11: If a^2 is an even number, then a is an even number.

Proof:

Assume that a^2 is an even number and a is an odd number. Since a is an odd number we have a = 2k + 1, for some integer k. so $a^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2(k^2 + k) + 1$, here $k^2 + k$ is some integer, say l, then $a^2 = 2l + 1$ i.e. a^2 is an odd number This contradicts our assumption that is a^2 even. Hence proved.

Example 12: Prove that if $n^3 + 5$ is odd, then n is even.

Solution:

Let: $p \rightarrow q$: if $n^3 + 5$ is odd then n is even

Suppose $n^3 + 5$ is odd and n is also odd.

$\therefore$ n can be expressed as:

n = 2k + 1 for some positive integer k.

$\therefore$ $n^3 + 5 = (2k + 1)^3 + 5$

$= 8k^3 + 12k^2 + 6k + 1 + 5$

$= 8k^3 + 12k^2 + 6k + 6$

$= 2(4k^3 + 6k^2 + 3k + 3)$

which is even.

This contradict our assumption that $n^3 + 5$ is odd.

Proofs By Counter Examples

To prove that the statement of the form xP(x) is false, we just need some value of x. So while proving for falsity we just look for counter example.

Example 13: Prove or disprove the product of two irrational numbers is irrational.

Proof:

Here we instantly try to get the product of the irrational to try it. Lets take both the number for product be $\sqrt{2}$ then we have $\sqrt{2}^*\sqrt{2} = 2$ (not rational). Hence by counter example it is shown that the product of two irrational numbers is not necessarily irrational.

Trivial and Vacuous Proofs

In the implication $p \rightarrow q$, if we can show that the consequence q is true then regardless of truth values of p, the implication $p \rightarrow q$ is true. Such type of proof technique is called **trivial proof**.

In the implication $p \rightarrow q$, if we can show that the hypothesis p is false, then regardless of truth values of q, the implication $p \rightarrow q$ is true. Such kind of proof of an implication $p \rightarrow q$ is called **vacuous proof**.

Example 14: If x is an integer, then 3 is an odd integer. (Trivial) If a black is white, then pink is blue. (Vacuous)

Existence Proofs

A proof of a proposition of the form $\exists xP(x)$ is called an existence proof. There are different ways of proving a theorem of this type. Sometime some element a is found to show P(a) to be true, this is called constructive existence proof. In other method we do not provide a such that P(a) is true but prove that $\exists xP(x)$ is true in different way, this is called non constructive existence proof.

Example 15: Prove that there are 100 consecutive positive integers that are not perfect squares.

Proof:

Lets consider 2500 this is a perfect square of 50, and take 2601 this is a perfect square of 51. in between 2601 and 2500 there are 100 consecutive positive integers. Hence the proof.

Automata, Computability and Complexity

What is computation?

Computation refers to any task that can be performed by the computer or any machine. A Computation is the definition of the set of allowable operations used in solving a problem. Simply, a computation is a function that takes an input and produces an output. If there is a mapping of input to output then compuation is likely to happen.

Theory of Computation

Theory of computation tries to answer the following questions:

- What are the mathematical properties of computer hardware and software?
- What is a computation and what is an algorithm?
- What are the limitations of computers? Can "everything" be computed?

Theory of computation teaches you about the elementary ways in which a computer can be made to think. Theory of computation includes developing formal mathematical models of computation that reflect real-world computers (machines) and study its theory i.e. determining the capabilities, problems that can be solved and limitation of that machine.

Theory of Computation can be divided into the following three areas: Complexity Theory, Computability Theory, and Automata Theory.

Complexity Theory

Complexity theory considers not only whether a problem can be solved at all on a computer, but also how efficiently the problem can be solved. Two major aspects are considered:

- Time complexity: how many steps does it take to perform a computation?
- Space complexity: how much memory is required to perform that computation?

Computability Theory

Computability theory deals primarily with the question of the extent to which a problem is solvable on a computer. The theory classifies problems as being solvable or unsolvable.

Automata Theory

Automata theory is the study of abstract computational devices. Abstract devices are (simplified) models of real computations. It is the study of abstract machine and their properties, providing a mathematical notion of "computer". Automata Theory deals with definitions and properties of different types of "computation models". Examples of such models are:

- **Finite Automata:** These are used in text processing, compilers, and hardware design.
- **Context-Free Grammars:** These are used to define programming languages and in Artificial Intelligence.
- **Turing Machines:** These form a simple abstract model of a "real" computer, such as your PC at home.

Abstract Model

An abstract model is a model of computer system (considered either as hardware or software) constructed to allow a detailed and precise analysis of how the computer system works. Such a model usually consists of input, output and operations that can be performed and so can be thought of as a processor.E.g. an abstract machine that models a banking system can have operations like "deposit", "withdraw", "transfer", etc.

Basic Concepts of Automata Theory

Alphabets (Σ)

Alphabet is a finite non-empty set of symbols. The symbols can be the letters such as {a, b, c}, bits {0, 1}, digits {0, 1, 2, 3... 9}. Common characters like $, #, etc. For example;

$\Sigma = \{0,1\}$ - Binary alphabets

$\Sigma = \{+, -, *\}$ – Special symbols

Strings

String is a finite sequence of symbols taken from some alphabet. E.g. 0110 is a string from binary alphabet, "computation" is a string over alphabet {a, b, c ... z}.

Length of String

The length of a string w, denoted by | w |, is the number of symbols in w. Example string w = computation has length |w|=11.

Empty String

It is a string with zero occurrences of symbols. It is denoted by 'ε' (epsilon). The length of empty string is zero. i.e. $|\varepsilon| = 0$.

Power of Alphabet

The set of all strings of certain length k from an alphabet is the k^{th} power of that alphabet. i.e. $\sum^k = \{w / |w| = k\}$

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

$\Sigma^0 = \{\varepsilon\}$

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

$\Sigma^2 = \{00, 01, 10, 11\}$

$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

Kleen Closure

The set of all the strings over an alphabet Σ is called kleen closure of Σ & is denoted by Σ^*. Thus, kleen closure is set of all the strings over alphabet Σ with length 0 or more.

$\therefore \Sigma^* = \Sigma 0 \cup \Sigma 1 \cup \Sigma 2 \cup \Sigma 3 \cup \ldots\ldots\ldots\ldots$

E.g. A = {0}

$A^* = \left\{\frac{0^n}{n = 0, 1, 2, \ldots}\right\}$

Positive Closure

The set of all the strings over an alphabet Σ, except the empty string is called positive closure and is denoted by Σ^+.

$\therefore \Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots\ldots\ldots\ldots$

Concatenation of Strings

If x and y are two strings over an alphabet, the concatenation of x and y is written xy and consists of the symbols of x followed by those of y.

For example;
x = aaa
y = bbb
xy = aaabbb&
yx = bbbaaa

Note: Concatenating the empty string 'ε' with another string, the result is just the other string. i.e. $\varepsilon w = w\varepsilon = w$.

Suffix of a string

String s is called a suffix of a string w if it is obtained by removing zero or more leading symbols in w.

For example;

w = abcd

s = bcd is suffix of w.

s is proper suffix if $s \neq w$.

Prefix of a string

A string s is called a prefix of a string w if it is obtained by removing zero or more trailing symbols of w.

For example;

w = abcd

s = abc is prefix of w,

Here, s is proper suffix i.e. s is proper suffix if $s \neq w$.

Substring

A string s is called substring of a string w if it is obtained by removing zero or more leading or trailing symbols in w. It is proper substring of w if $s \neq w$.

Language

A language L over an alphabet $\sum$ is subset of all the strings that can be formed out of $\sum$; i.e. a language is subset of kleen closure over an alphabet $\sum$; $L \subseteq \sum^*$. (Set of strings chosen from $\sum^*$ defines language). For example;

- Set of all strings over $\sum = \{0, 1\}$ with equal number of 0's & 1's.
  $L = \{\varepsilon, 01, 0011, 000111, \ldots\ldots\}$
- Set of all string over $\sum = \{0, 1\}$ that ends with 1.
  $L = \{1, 01, 11, 011, 0111, \ldots\ldots\}$
- English language is a set of strings taken from the alphabet $\sum = \{a, b, c, \ldots\ldots, z, A, B, C, \ldots\ldots\ldots, Z\}$.
  L = {DBMS, TOC, GRAPHICS}.

Empty Language

A language is empty if it does not have any strings within it. φ is an empty language and is a language over any alphabet. It does not contain any string.

Membership in a Language

Membership in a language defines association of some string with the language. A membership problem is the question of deciding whether a given string is a member of some particular language or not i.e. whether the string belongs to the given language or not.

In other words, if $\sum$ is an alphabet & L is a language over $\sum$, then problem is, given a string w in $\sum^*$, decide whether or not w is in L.

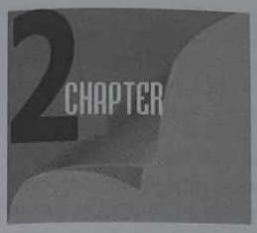
Example:

Given, L = {DBMS, TOC, GRAPHICS}, then for w = "DBMS", the membership of w is true in L and hence $w \in L$. For w'="HELLO", the membership of w' is false in Land hence $w' \notin L$.

1. What is logic ? Differentiate between propositional logic and predicate logic.
2. Define direct and indirect methods of proof with example.
3. Explain language and empty language with example.
4. What is computation?
5. Define Kleen and Positive Closure with an example.
6. Find all pairs of sets of A and B for which A.B = {ab, aaa, abab, abbb, abaaa, ababbb}.

7. For A ="hello world", compute prefix with length 3, suffix with length 4.
8. For string of size n, i.e. a_1a_2 ..... a_n. What will be the number of prefixes, suffixes, & substrings.
9. Compute the closure of Ø.
10. Given binary alphabet, $\sum = \{0, 1\}$, now compute $\sum^k$; where 0<=k<=3.
11. Given a function *Substr* (S, i, j), which returns substring of a string, S, beginning at i^{th} position and ending at j^{th} position both inclusive. Show what following instances of this function returns: S = Automata; *Substr* (S, 0, 3), Substr (S, 7, 6), Substr (S, 5, 5).

□□□



INTRODUCTION TO FINITE AUTOMATA

CHAPTER OUTLINES

After studying this chapter you should be able to:

- Automaton
- Deterministic Finite Automata
- Equivalence of Non-deterministic and Deterministic Finite Automata

AUTOMATON

An automaton is defined as a system where energy, materials and information are transformed, transmitted and used for performing some functions without direction participation of man. Examples are automatic machine tools, automatic packing machines, and automatic photo printing machines. Any automaton can be described using input, output, states, and state transition. An automaton in which the output depends only on the input is called an automaton without a memory. An automaton, in which the output depends on the state as well, is called automaton with a finite memory. Automata are plural form of automaton.

Finite Automaton / Finite State Machine

A finite automaton is a mathematical (model) abstract machine that has a set of "states" and its "control" moves from state to state in response to external "inputs". The control may be either "deterministic" meaning that the automation can't be in more than one state at any one time, or "non deterministic", meaning that it may be in several states at once. This distinguishes the class of automata as DFA or NFA.

- The DFA, i.e. Deterministic Finite Automata can't be in more than one state at any time.
- The NFA, i.e. Non-Deterministic Finite Automata can be in more than one state at a time.

Before giving the formal definition of finite automata, Let us consider an example in which such an automation shows up in a natural way. Consider the problem of designing a "machine" that controls a automatic vending machine.

The machine gives the can of coke as soon as someone pays 5 rupees. Assume that only three coin denominations: 1, 2, 5 rupees are accepted and no excess change is returned.

Now someone inserts a sequence of coins into the machine. At any moment, the machine has to decide whether or not to give a can of coke. i.e. whether or not someone has paid 5 rupees (or more). In order to decide this, the machine is in one of the following six states, at any moment during the process:

- The machine is in state q_0, if it has not collected any money yet.
- The machine is in state q_1, if it has collected exactly 1 rupee.
- The machine is in state q_2, if it has collected exactly 2 rupee.
- The machine is in state q_3, if it has collected exactly 3 rupees.
- The machine is in state q_4, if it has collected exactly 4 rupees.
- The machine is in state q_5, if it has collected 5 rupees or more.

Initially the machine is in state q_0. Assume, for example, that some person presents the sequence (1, 2, 2, 1) of coins.

- After receiving the first 1 rupee coin, the machine switches from state q_0 to state q_1.
- After receiving the second 2 rupees coin, the machine switches from state q_1 to state q_3.
- After receiving the third 2 rupees coin, the machine switches from state q_3 to state q_5.
- After receiving the Fourth 1 rupee coin, the machine does not change its state.

At this moment, when the person press the button for coke it outputs the can of coke. (Remember that no change is given).

Figure below represents the behavior of the machine for all possible sequences of coins. State q_5 is represented by two circles, because it is a special state: As soon as the machine reaches this state, gives a can of coke.

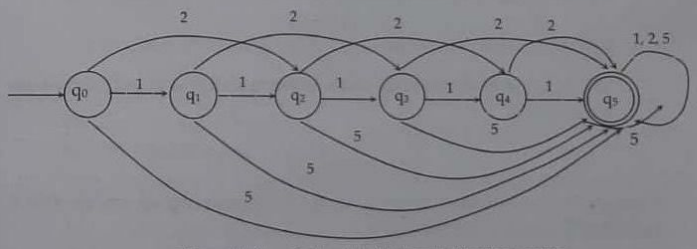


Figure: Mathematical model of Automatic Vending Machine.

DETERMINISTIC FINITE AUTOMATA

Deterministic Finite Automata is a finite-state machine that accepts or rejects strings of symbols and only produces a unique computation of the automaton for each input string. The term deterministic refers to the uniqueness of the computation. In DFA, for each input symbol, one can determine a unique state to which the machine will move.

Definition: A finite automaton is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set, whose elements are called states,
2. Σ is a finite set, called the alphabet ; the elements of Σ are called symbols,

3. $\delta : Q \times \Sigma \rightarrow Q$ is a function, called the transition function. It takes state and input symbol as a argument and returns a state as ouput. It tells machine which state to go after reading a input symbol from a particular state.
4. q_0 is an element of Q; it is called the start state,
5. F is a subset of Q; the elements of F are called accept states.

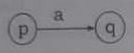
Notations for DFA's (Methods for describing Transition function)

There are two preferred notations for describing this class of automata;

- Transition Diagram (State Diagram)
- Transition Table

1. Transition Diagram (State Diagram)

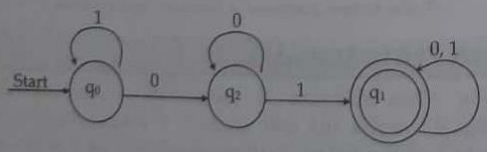
It is graphical representation in which states are represented by circles, transitions are represented by arrows with input symbols as labels,



and accepting states are designated by double instead of single circles.

The initial state will have an arrow pointing to it that doesn't come from another state.

Example 1: The transition diagram for the DFA accepting all strings with a substring 01 is shown in figure below:



The state diagram in example 1 represents the DFA $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$ where δ is given by

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

2. Transition Table

Transition table is a conventional, tubular representation of the transition function δ that takes the arguments from $Q \times \sum$ & returns a value which is one of the states of the automation. The row of the table corresponds to the states while column corresponds to the input symbol. The starting state in the table is represented by $\rightarrow$ followed by the state i.e. $\rightarrow q_i$ for q being start state, whereas final state as $*q_i$ for q being final state.

The transition table for the DFA accepting all strings over {0, 1} having substring 01 is given below:

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

How DFA process strings?

The language of DFA is the set of all strings which when processed by DFA leads its state to one of the accepting state. The DFA determines whether the given string belongs to the language of DFA with the help of transition function.

Initially, the DFA is in its start state. It then reads the input symbols of string one by one and changes its state according to the transition function. After reading all the input symbols if DFA reaches to its accepting state then the string is accepted by DFA otherwise not.

Suppose $a_1, a_2, \ldots\ldots a_n$ is a sequence of input symbols. Initially, the DFA starts its work from the starting state, q_0. It consult the transition function δ, say $\delta(q_0, a_1) = q_1$ to find the state that the DFA enters after processing the first input symbol a_1. We then process the next input symbol a_2, by evaluating $\delta(q_1, a_2)$; suppose this state be q_2. It continue in this manner, finding states $q_3, q_4, \ldots, q_n$ such that $\delta(q_{i-1}, a_i) = q_i$ for each i. if q_n is a member of F, then input $a_1, a_2, \ldots a_n$ is accepted & if not then it is rejected.

Extended Transition Function of DFA

The extended transition function is a transition function that tells the DFA, which state to go after reading a string from the particular state.

Formally, The extended transition function of DFA, denoted by $\hat{\delta}$ is a transition function that takes two arguments as input, one is the state q of Q and another is a string $w \in \sum^*$, and generates a state $p \in Q$. This state p is that the automaton reaches when starting in state q & processing the sequence of inputs w.

i.e. $\hat{\delta}(q, w) = p$

We can define $\hat{\delta}$ by induction on length of input string as follows:

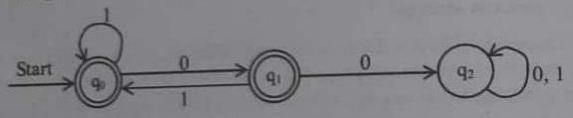
Basis step: $\hat{\delta}(q, \epsilon) = q$ i.e. from state q, reading no input symbol stays at the same state.

Induction: Let w be a string from Σ^* such that w = xa, where x is substring of w without last symbol and a is the last symbol of w, then $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$

Thus, to compute $\hat{\delta}(q, w)$, we first compute $\hat{\delta}(q, x)$, the state the automaton is in after processing all but last symbol of w. Let this state is p, i.e. $\hat{\delta}(q, x) = p$.

Then, $\hat{\delta}(q, w)$ is what we get by making a transition from state p on input a, the last symbol of w. i.e. $\hat{\delta}(q, w) = \delta(p, a)$.

Example:



Compute $\hat{\delta}(q_0, 1001)$

$= \delta(\hat{\delta}(q_0, 100), 1)$

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

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

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

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

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

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

$= \delta(q_2, 1)$

$= q_2$, so not accepted.

Acceptability of a string by a DFA

A string x is accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ if; $\hat{\delta}(q_0, x) = q \in F$.

Language of DFA

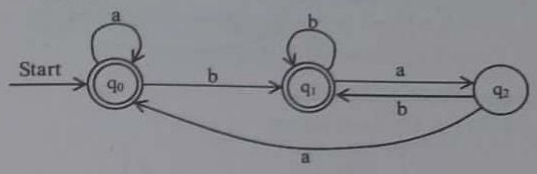
The language of DFA $M = (Q, \Sigma, \delta, q_0, F)$ denoted by L(M) is a set of strings over Σ^* that are accepted by M.

i.e; $L(M) = \{w / \hat{\delta}(q_0, w) = q \in F\}$

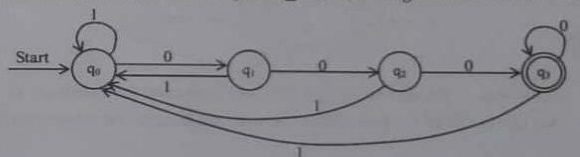
That is; the language of a DFA is the set of all strings w that take DFA starting from start state to one of the accepting states. The language of DFA is called regular language.

Examples of DFA

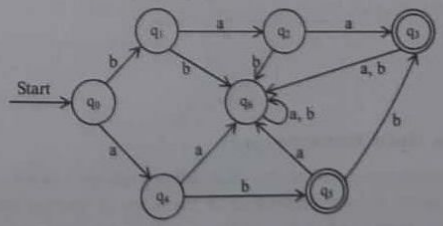
- Construct a DFA, that accepts all the strings over $\Sigma = \{a, b\}$ that do not end with ba.



- DFA accepting all string over $\Sigma = \{0, 1\}$ ending with 3 consecutive 0's.

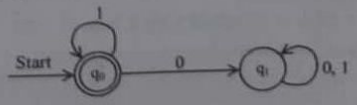


- DFA over {a, b} accepting {baa, ab, abb}

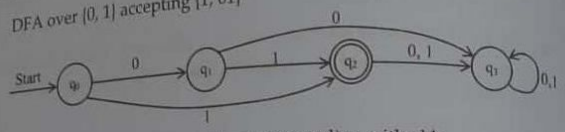


- DFA accepting zero or more consecutive 1's.

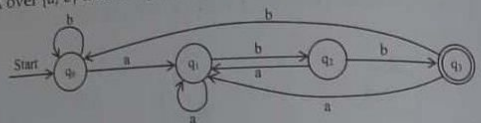
i.e. $L(M) = \{1^n / n = 0, 1, 2, \ldots\}$



- DFA over {0, 1} accepting {1, 01}

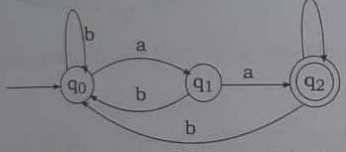


- DFA over {a, b} that accepts the strings ending with abb.



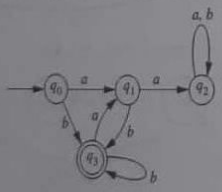
- Language of Strings ending in aa

  i.e; L = {x ∈ {a, b}* | x ends with aa}



- Language of Strings Ending in b and Not Containing the Substring aa

  i.e; L2 ={x ∈{a,b}* | x ends with b and does not contain the substring aa}



Non-deterministic Finite Automata (NFA)

NFA is an automaton machine in which for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined, hence the tern non-deterministic. The transition moves from a state can be to zero or more states taking an input.

Definition: A **NFA** is a 5-tuple **M = (Q, Σ, δ, q_0, F)**, where

1. Q is a finite set, whose elements are called states,
2. Σ is a finite set, called the alphabet ; the elements of Σ are called symbols,

3. $\delta : Q \times \Sigma \rightarrow Q$ is a function, called the transition function. It takes state in Q and input symbol in Σ as a argument and returns a subset of Q as ouput.
4. q_0 is an element of Q; it is called the start state,
5. F is a subset of Q; the elements of F are called accept states.

The difference between DFA and the NFA is in the type of transition function (δ). In NFA, δ is a function that takes a state and input symbol as arguments and returns a set of zero, one, or more states as output but in case of DFA exactly only one state is returned as output.

Example 1: Consider a NFA over {0, 1} as shown in figure below

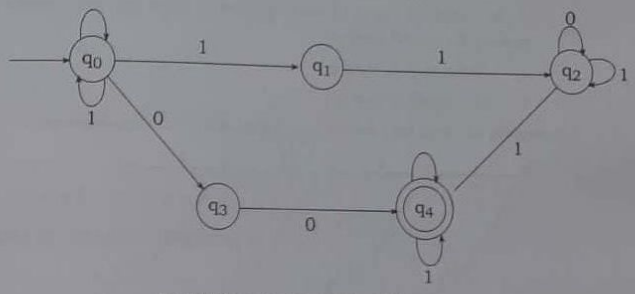


Fig: Transition system for nondeterministic automaton.

The formal description of above NFA is **M = (Q, Σ, δ, q_0, F)**, where

1. Q = { q_0, q_1, q_2, q_3, q_4,}
2. Σ = {0,1}
3. δ is given as

δ:	0	1
→q_0	{ q_0, q_3}	{ q_0, q_1}
q_1	{φ}	q_2
q_2	q_2	q_2
q_3	q_4	{φ}
*q_4	q_4	q_4

4. q_0 is the starting state
5. F = { q_4 }.

There are two differences with the DFA that we have seen until now. First, if the automaton is in state q_0 and reads the symbol 0, then it has two options: Either it

switches in state q_1, or it switches to state q_3. Second, if the automaton is in state q_1 and reads the symbol 0, then it cannot continue.

Let us consider the string 0100 then this automaton processes the string as follows:

- Initially, the automaton is in the start state q_0.
- Since the first symbol in the input string is 0, the automaton can either stays in state $q_0$or switch to state q_3.
- If the automaton switches in state q_3, then it does not have the transition after having read the second symbol.
- If the automaton stays in state q_0, then it again has two options after reading the second symbol:
  - Either stays in state q_0
  - Or, switch to state q_1

Continuing this way, the overall computation is shown in figure below:

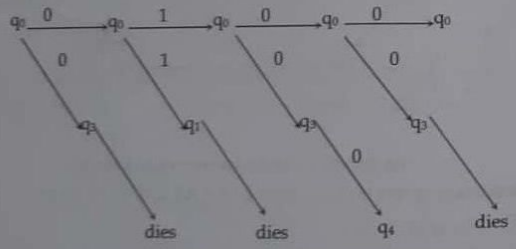
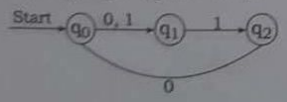


Fig: States reached while processing 0100.

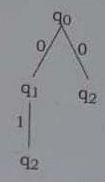
Example 2: NFA over {0, 1} accepting strings {0, 01, 11}.



Transition table:

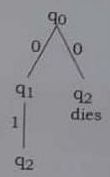
δ:	0	1
→ q0	{q0,q2}	{q1}
q1	{φ}	{q2}
*q2	{φ}	{φ}

Computation tree for 01;



Final, so 01 is accepted.

Computation tree for 0110;



dies, so 0110 is not accepted

Extended Transition Function of NFA

The extended transition function of NFA, denoted by $\hat{\delta}$ is a transition function that takes two arguments as input; a state $q \in Q$ & a string w and returns a set of states that the NFA is in, if it starts in q & processes the string w.

Definition by Induction:

Basis: $\hat{\delta}(q, \in) = \{q\}$ i.e. reading no input symbol remains into the same state.

Induction: Let w be a string from $\sum^*$ such that w = xa, where x is a substring of w without last symbol a. Also let,

$\hat{\delta}(q, x) = \{p_1, p_2, p_3, \ldots p_k\}$

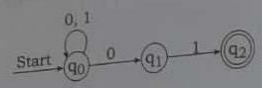
&

$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, r_3, \ldots r_m\}$

Then, $\hat{\delta}(q, w) = \{r_1, r_2, r_3, \ldots r_m\}$

Thus, to compute $\hat{\delta}(q, w)$ we first compute $\hat{\delta}(q, x)$ & then following any transition from each of these states with input a.

Consider,



Now, computing 01101;

Solution:

$\hat{\delta}(q_0, 01101)$

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

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

$\hat{\delta}(q_0, 01) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

$\hat{\delta}(q_0, 011) = \delta(q_0, 1) \cup \delta(q_2, 1) = \{q_0\} \cup \{\varphi\} = \{q_0\}$

$\hat{\delta}(q_0, 0110) = \delta(q_0, 0) = \{q_0\} \cup \{q_1\} = \{q_0, q_1\}$

$\hat{\delta}(q_0, 01101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

So, accepted.

Language of NFA

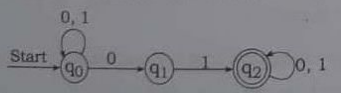
The language of NFA, $M = (Q, \Sigma, \delta, q_0, F)$, denoted by L (M) is;

$$L(M) = \{w / \hat{\delta}(q, w) \cap F \neq \varphi\}$$

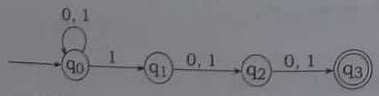
i.e. L (M) is a set of strings w in Σ^* such that $\hat{\delta}(q, w)$ contains at least one accepting state.

Examples of NFA

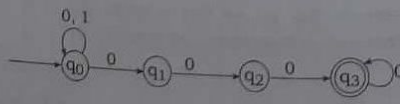
- Construct a NFA over {0, 1} that accepts strings having 11as substring.



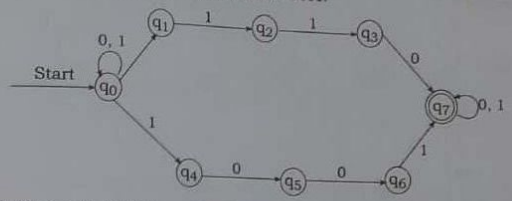
- Construct a NFA over {0, 1} that accepts strings having 1 in the third position from the end. (e.g., 00100, 010101 but not 0010).



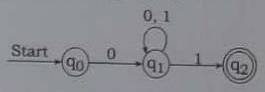
- Construct an NFA to accept those strings containing three consecutive zeroes.



- NFA for strings over {0, 1} that 0110 or 1001.



- NFA over {a, b} that accepts strings stating with *0* and ending with *1*.



Equivalence of Non-Deterministic and Deterministic Finite Automata

DFA and NFA recognise the same class of languages. That is; non-determinism does not make a finite automaton more powerful.

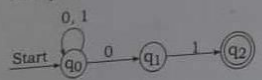
To show DFA and NFA recognise the same class of language; we will describe the method of converting an arbitrary NFA into its equivalent DFA. This method is known as **subset construction method.**

Subset Construction method

The formal conversion of a NFA $N = (Q_N, \sum, \delta_N, q_0, F_N)$ into its equivalent DFA $D = (Q_D, \sum, \delta_D, q_0, F_D)$ is done as follows:

1. The start state of D is the set of start states of N i.e. if q0 is start state of N then D has start state as $\{q_0\}$.
2. Q_D is set of subsets of Q_N i.e. $Q_D = 2^{QN}$. So, QD is power set of Q_N. So if Q_N has n states then Q_D will have 2^n states. However, all of these states may not be accessible from start state of Q_D so they can be eliminated. So Q_D will have less than 2^n states.
3. $F_D$is set of subsets S of Q_N such that $S \cap F_N \neq \varphi$ i.e. F_D is all sets of N's states that include at least one final state of N.
4. For each set $S \subseteq Q\,N$ & each input $a \in \sum$,
   $\delta_D(S, a) = \cup_{p \in S} \delta_N(p, a)$
   i.e. to compute $\delta_D(S, a)$ look at all the states p in S then find the states that N goes after reading input from p and take the union of those states.

Example 1:

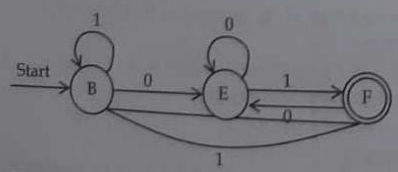


δ:	0	1
A	A	A
→B	E	B
C	A	D
-D	A	A
E	E	F
-F	E	B
-G	A	D
-H	E	F

i.e.

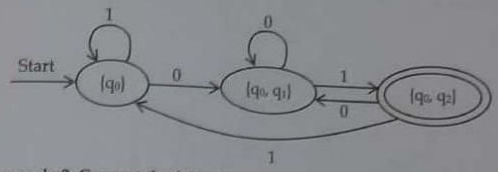
	δ:	0	1
A	ϕ	ϕ	ϕ
B	→ {q0}	{q0,q1}	{q0}
C	{q1}	ϕ	{q2}
D	-{q2}	ϕ	ϕ
E	{q0, q1}	{q0, q1}	{q0, q2}
F	-{q0, q2}	{q0, q1}	{q0}
G	-{q1, q2}	ϕ	{q2}
H	-{q0, q1, q2}	{q0, q1}	{q0, q2}

The equivalent DFA is:

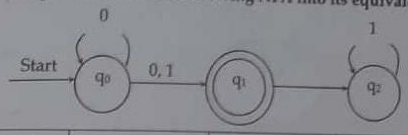


Other states are ignored as they are not reached starting from start state.

It can be rewritten as;



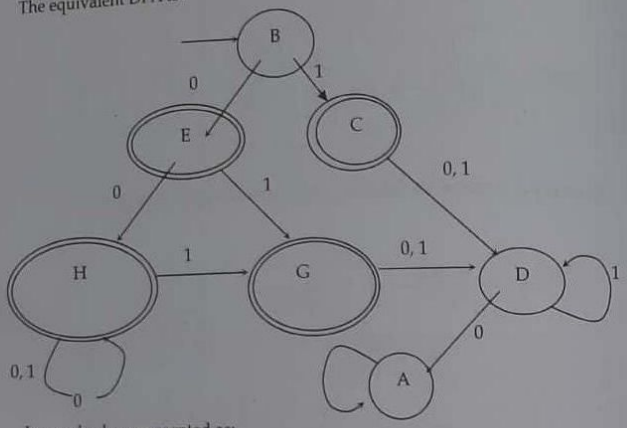
Example 2: Convert the following NFA into its equivalent DFA



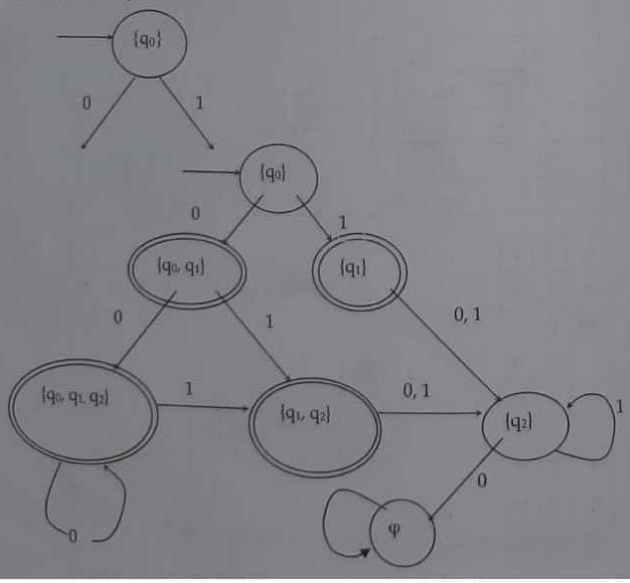
	δ:	0	1
A	φ	φ	Φ
B	→{q0}	{q0, q1}	{q1}
C	*{q1}	{q2}	{q2}
D	{q2}	φ	{q2}
E	*{q0, q1}	{q0, q1, q2}	{q1, q2}
F	{q0, q2}	{q0, q1}	{q1, q2}
G	*{q1, q2}	{q2}	{q2}
H	*{q0, q1, q2}	{q0, q1, q2}	{q1, q2}

δ:	0	1
A	A	A
*B	E	C
C	D	D
D	A	D
*E	H	G
F	E	G
*G	D	D
*H	H	G

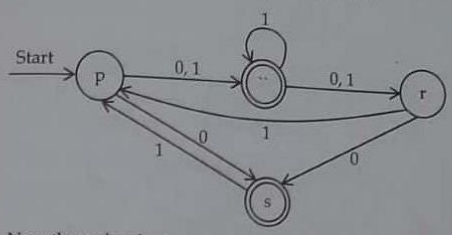
The equivalent DFA is:



It can also be represented as:



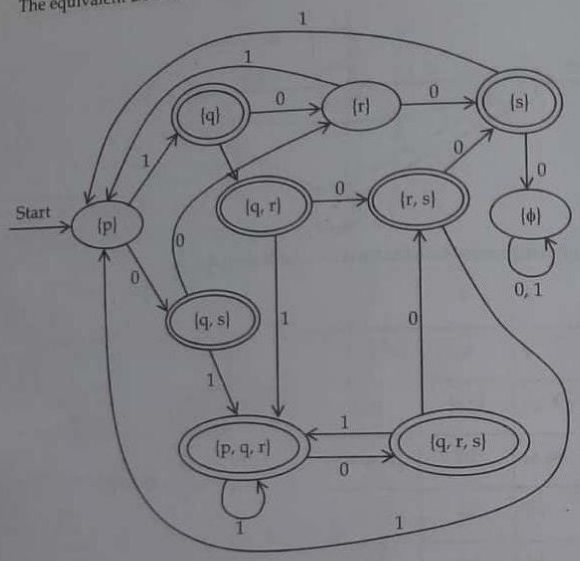
Example 3: Convert the following NFA to DFA.



Now through subset construction, we have the DFA as:

δ:	0	1
ϕ	ϕ	ϕ
→ {p}	{q, s}	{q}
*{q, s}	{r}	{p, q, r}
*{q}	{r}	{q, r}
{r}	{s}	{p}
*{p, q, r}	{q, r, s}	{p, q, r}
*{q, r}	{r, s}	{p, q, r}
*{s}	ϕ	{p}
*{q, r, s}	{r, s}	{p, q, r}
*{ r, s}	{s}	{p}

The equivalent DFA is:



Theorem 1: For any NFA, $N = (Q_N, \sum, \delta_N, q_0, F_N)$ accepting language $L \subseteq \sum^*$ there is a DFA $D = (Q_D, \sum, \delta_D, q_0', F_D)$ that also accepts L i.e. L (N) = L (D).

Proof: -

The DFA D, say can be defined as;

$QD = 2^{QN}$, $q_0' = \{q_0\}$

Let $S = \{p_1, p_2, p_3, \ldots p_k\} \in Q_D$. Then for $S \in Q_D$ & $a \in \sum$ $\delta_D(s, a) = \bigcup_{p_i \in S} \delta_N(p_i, a)$

$F_D = \{S / S \in Q_D \& S \cap F_N \neq \phi\}$

The fact that D accepts the same language as N is as;

for any string $w \in \sum^*$;

$\hat{\delta}_N(q_0, w) = \delta_D'(q_0', w)$

Thus, we prove this fact by induction on length of w.

Basis Step:

Let $|w| = 0$, then $w = \varepsilon$,

$\therefore \hat{\delta}_D(q_0', w) = \hat{\delta}_D(q_0', \epsilon) = q_0' = \{q_0\}$

Also,

$\hat{\delta}_N(q_0, w) = \hat{\delta}_N(q_0, \epsilon) = \{q_0\}$

$\therefore \hat{\delta}_D(q_0', w) = \hat{\delta}_N(q_0, w)$ is true $|w| = 0$

Induction Step:

Let $|w| = n + 1$ is a string such that $w = xa$ & $|x| = n$, $|a| = 1$; a being last symbol.

Let the inductive hypothesis is that x satisfies.

Thus, $\hat{\delta}_D(q_0', x) = \hat{\delta}_N(q_0, x)$, let these states be $\{p_1, p_2, p_3, \ldots p_k\}$

Now,

$\hat{\delta}_N(q_0, w) = \hat{\delta}_N(q_0, xa)$

$= \delta_N(\hat{\delta}_N(q_0, x), a)$

$= \delta_N(\{p_1, p_2, p_3, \ldots p_k\}, a)$ [Since, from inductive step]

$= \bigcup_{i=1}^{k} \delta_N(p_i, a)$ ———————— (I)

Also,

$\hat{\delta}_D(q_0', w) = \hat{\delta}_D(q_0', xa)$

$= \delta_D(\hat{\delta}_D(q_0', x), a)$

$= \delta_D(\hat{\delta}_N(q_0, x), a)$ [Since, by the inductive step as it is true for x]

$= \delta_D(\{p_1, p_2, p_3, \ldots p_k\}, a)$ [Since, from inductive step]

Now, we know from subset construction, it tells us that,

$\delta_D(\{p_1, p_2, p_3, \ldots p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a)$

Thus, we conclude,

$\hat{\delta}_D(q_0', w) = \ldots\ldots\ldots\ldots$ (ii)

Here, from (I) & (II),

$\hat{\delta}_N(q_0, w) = \hat{\delta}_D(q_0', w)$

Hence, if this relation is true for $|w| = n$, then it is also true for $|w| = n + 1$.

$\therefore$ DFA D & NFA N accepts the same language.

i.e. L (D) = L (N) Proved

Theorem 2: A language L is accepted by some NFA if L is accepted by some DFA.

Proof:

Consider we have a DFA $D = (Q_D, \Sigma, \delta_N, q_0, F_D)$.

This DFA can be interpreted as a NFA having the transition diagram with exactly one choice of transition for any input.

i.e. $N = (Q_N, \Sigma, \delta_N, q_0', F_N)$

where, $Q_N = Q_D$, $F_N = F_D$, $q_0' = q_0$

And δ_N defined as if $\delta_N(q, a) = p$ then $\delta_N(q, a) = \{p\}$.

Then to show if L is accepted by D then it is also accepted by N, it is sufficient to show, for any string $w \in \Sigma^*$,

$$\hat{\delta}_D(q_0, w) = \hat{\delta}_N(q_0, w)$$

We can proof this fact using induction on length of the string.

Basis step: -

Let $|w| = 0$ i.e. $w = \varepsilon$

$\therefore \hat{\delta}_D(q_0, w) = \hat{\delta}_D(q_0, \varepsilon) = q_0$

$\hat{\delta}_N(q_0, w) = \hat{\delta}_N(q_0, \varepsilon)$

$\hat{\delta}_N(q_0, w) = \hat{\delta}_N(q_0, \varepsilon) = \{q_0\}$

$\therefore \hat{\delta}_D(q_0, w) = \hat{\delta}_N(q_0, w)$ for $|w| = 0$ is true.

Induction: -

Let $|w| = n + 1$ & $w = xa$. Where $|x| = n$ & $|a| = 1$; a being the last symbol.

Let the inductive hypothesis is that it is true for $w = x$.

$\therefore$ if $\hat{\delta}D(q0, x) = p$, then $\hat{\delta}_N(q_0, x) = \{p\}$

i.e. $\hat{\delta}D(q0, x) = \hat{\delta}_N(q_0, x)$

Now,

$$\hat{\delta}D(q0, w) = \hat{\delta}_D(q_0, xa)$$
$$= \delta_D(\hat{\delta}_D(q_0, x), a)$$
$$= \delta_D(p, a) \quad [\because \text{from inductive step}]$$
$$= r, \text{ say}$$

Now,

$$\hat{\delta}N(q0, w) = \hat{\delta}_N(q_0, xa)$$
$$= \delta_N(\hat{\delta}_N(q_0, x), a)$$
$$= \delta_N(\{p\}, a) \quad [\because \text{from inductive step}]$$

As the NFA N is constructed from the DFA D, where transition diagram of NFA consists exactly one choice of transition for any input. So, taking input a, from any state, the transition of NFA will be same as DFA.

i.e. $\hat{\delta}_N(\{p\}, a) = \{r\}$

$\therefore \hat{\delta}_N(\{p\}, a) = \{r\}$

Hence, $\hat{\delta}D(q0, w) = \hat{\delta}_N(q_0, w)$ Proved.

Theorem 3: *A Language L is accepted by some DFA if and only if L is accepted by some NFA*

Proof: Here to prove the above theorem we have to prove its two parts:

(If part): If there is NFA that accepts language L then there exists some DFA that also accepts language L.i.e.for any string $w \in \Sigma^*$,

$$\hat{\delta}D(q0, w) = \hat{\delta}_N(q_0, w)$$

For this we can use same method as for theorem 1.

(Only if part): If L is accepted by DFA then there exists some NFA that also accepts language L.

$\Rightarrow$ Consider we have a DFA $D = (Q_D, \Sigma, \delta_N, q_0, F_D)$ that accepts language L we can construct its equivalent NFA $N = (Q_N, \Sigma, \delta_N, q_0', F_N)$.
where, $Q_N = Q_D$, $F_N = F_D$, $q_0' = q_0$
And δ_N defined as if $\delta_N(q, a) = p$ then $\delta_N(q, a) = \{p\}$.
Hence Proved.

NFA with ε-transition (ε-NFA)

This is another extension of finite automation called ε-NFA. The new feature of ε-NFA is that, it allows a transition on ε, the empty string which means it can switches to new states without reading an input symbol. This capability does not expand the class of languages that can be accepted by finite automata, but it does give some added "programming convenience".

Definition: A NFA with ε-transition is defined by five tuples $(Q, \Sigma, \delta, q_0, F)$, where;

Q = set of finite states
Σ = set of finite input symbols
q_0 = Initial state, $q_0 \in Q$
F = set of final states; $F \subseteq Q$

δ = a transition function that two arguments:

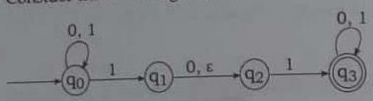
i. A state in Q, and

ii. A member of ∑ ∪ {ε}, that is , either an input symbol, or the symbol ε.

As input and produce a set of states as output. i.e. $Q \times \sum \cup \{\varepsilon\} \rightarrow 2^Q$

Examples:

1. Consider the following ε-NFA



The formal description of ε-NFA is (Q, ∑, δ, q_0, F) where

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

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

iii. δ is given as

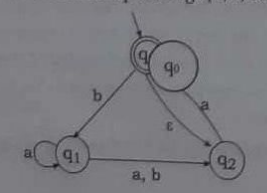
δ:	0	1	ε
→q_0	{q_0}	{q_0, q_1}	φ
q_1	{q_2}	Φ	{q_2}
q_2	φ	{q_3}	φ
q_3	{q_3}	{q_3}	φ

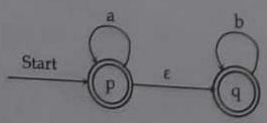
iv. q_0 is the starting state

v. $F = \{q_3\}$.

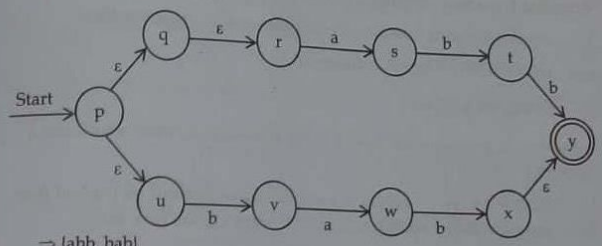
Examples

ε-NFA that accepts strings { ε, a, baba, baa}





⇒ {a, b, ab, aab, abb, ...}



⇒ {abb, bab}

Epsilon Closures

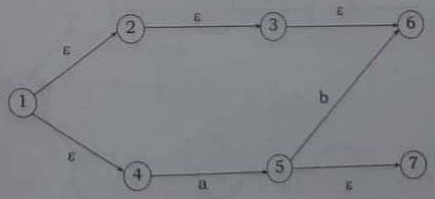
Informally Epsilon closure of a state q denoted by ε-closure (q) is the set of states that can be reached without reading any input symbol from state q. To obtain ε-closure (q), follow all transitions out of q that are labelled ε After we get to another state by following ε, follow the ε-transitions out of those states & so on, eventually finding every state that can be reached from q along any path whose arcs are all labeled ε.

Formally, we can define ε-closure of the state q as;

Basis: state q is in ε-closure (q).

Induction: If state q is reached with ε-transition from state q, p is in ε-closure (q). and if there is an arc from p to r labeled ε, then r is in ε-closure (q) and so on.

Example: Consider the ε-NFA below



Then,

ε-closure (1) = {1, 2, 3, 4, 6}

ε-closure (2) = {2, 3, 6}

ε-closure (4) = {4}

ε-closure (5) = {5, 7}

Extended Transition Function of ε-NFA: -

The extended transition function of ε-NFA denoted by $\hat{\delta}$ is defined by;

i) BASIS STEP: - $\hat{\delta}(q, \varepsilon) = \varepsilon$-closure (q)

ii) INDUCTION STEP: -

Let w = xa be a string, where x is substring of w without last symbol a and $a \in \sum$ but $a \neq \varepsilon$.

Let $\hat{\delta}(q, x) = \{p_1, p_2, \ldots p_k\}$ i.e. pi's are the states that can be reached from q following path labeled x which can end with many ε & can have many ε.

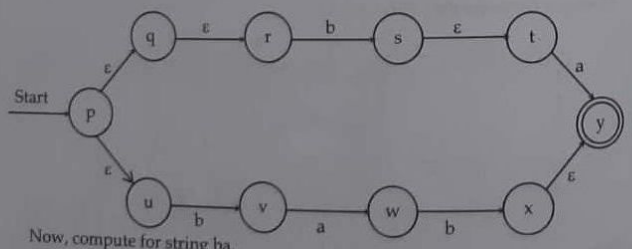
Also let,

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$$

Then, $\hat{\delta}(q, x) = \bigcup_{j=1}^{m} \varepsilon$ – closure (r_j)

To compute $\hat{\delta}$ (q, w), first find the ε-closure of starting state then read the first symbol of w and determine the states of NFA after reading first symbol and take the union of ε-closure of these states. Read the second symbol and determine the states of NFA after reading second symbol from the union of ε-closure of states found in previous step. Continue this process until the last symbol is read and finally compute the ε-closure of states that are reached after reading last symbol.

Suppose we have the ε-NFA as:



Now, compute for string ba.

1) $\hat{\delta}(p, \varepsilon) = \varepsilon$-closure (p) = {p, q, r, s}

2) $\hat{\delta}(p, b) = \delta(p, b) \cup \delta(q, b) \cup \delta(r, b) \cup \delta(s, b)$

$= \phi \cup \phi \cup \{u\} \cup \{t\} = \{u, t\}$

$\hat{\delta}(u, \varepsilon) \cup \hat{\delta}(t, \varepsilon) = \varepsilon$-closure (u) $\cup$ ε-closure (t)

$\therefore \hat{\delta}(p, b) = \{u\} \cup \{t, x\} = \{u, t, x\}$

3) $\hat{\delta}(p, ba) = \delta(u, a) \cup \delta(t, a) \cup \delta(x, a)$

$= \{v\} \cup \phi \cup \{y\}$

$\hat{\delta}(v, \varepsilon) \cup \hat{\delta}(y, \varepsilon) = \varepsilon$-closure (v) $\cup$ ε-closure (y)

$= \{v\} \cup \{y\}$

$\therefore \hat{\delta}(p, ba) = \{v, y\}$ *Accepted*

Compute for string bab

1) $\hat{\delta}(p, \varepsilon) = \varepsilon$-closure (p) = {p, q, r, s}

2) $\hat{\delta}(p, b) = \delta(p, b) \cup \delta(q, b) \cup \delta(r, b) \cup \delta(s, b)$

$= \phi \cup \phi \cup \{u\} \cup \{t\} = \{u, t\}$

$\hat{\delta}(u, \varepsilon) \cup \hat{\delta}(t, \varepsilon) = \varepsilon$-closure (u) $\cup$ ε-closure (t)

$\therefore \hat{\delta}(p, b) = \{u\} \cup \{t, x\} = \{u, t, x\}$

3) $\hat{\delta}(p, ba) = \delta(u, a) \cup \delta(t, a) \cup \delta(x, a)$

$= \{v\} \cup \phi \cup \{y\} = \{v, y\}$

$\hat{\delta}(v, \varepsilon) \cup \hat{\delta}(y, \varepsilon) = \varepsilon$-closure (v) $\cup$ ε-closure (y)

$= \{v\} \cup \{y\} = \{v, y\}$

$\therefore \hat{\delta}(p, ba) = \{v, y\}$

4) $\hat{\delta}(p, bab) = \delta(v, b) \cup \delta(y, b)$

$= \{w\} \cup \phi$

$= \{w\}$

$\hat{\delta}(w, \varepsilon) = \varepsilon$-closure (w)

$= \{y\}$ *Accepted.*

Conversion of ε-NFA into NFA & DFA

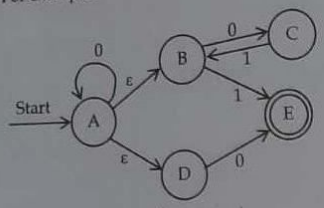
ε-NFA to NFA:

To construct NFA from a given ε-NFA, we have to eliminate the ε- transitions somehow, so that the resulting NFA will have no more ε-transitions. For this we do as below:

- Take start state of ε-NFA as start state of NFA. If ε-NFA accepts ε, then mark start state as a final state.
- Take final state of ε-NFA as final state of NFA.
- Perform $\delta_N(q, a) = \varepsilon$-closure $(\hat{\delta}(\varepsilon\text{-closure}(q), a))$

Where, δ_N = transition function of resulting NFA.

For example:



	ε	0	1
→ A	{B, D}	{A}	ϕ
B	ϕ	{C}	{E}
C	ϕ	ϕ	{B}
D	ϕ	{E}	ϕ
*E	ϕ	ϕ	ϕ

Now, the NFA is:

Start state = A

Now,

δ_N (A, 0) = ε-closure (δ (ε-closure (A), 0))
= ε-closure (δ ({A, B, D}, 0))
= ε-closure ({A, C, E})
= {A, B, C, D, E}

δ_N (A, 1) = ε-closure (δ (ε-closure (A), 1))
= ε-closure (δ ({A, B, D}, 1))
= ε-closure ({E})
= {E}

δ_N (B, 0) = ε-closure (δ (ε-closure (B), 0))
= ε-closure (δ ({B}, 0))
= ε-closure ({C})
= {C}

δ_N (B, 1) = ε-closure (δ (ε-closure (B), 1))
= ε-closure (δ ({B}, 1))
= ε-closure ({E})
= {E}

δ_N (C, 0) = ε-closure (δ (ε-closure (C), 0))
= ε-closure (δ ({C}, 0))
= ε-closure ({ϕ})
= ϕ

δ_N (C, 1) = ε-closure (δ (ε-closure (C), 1))
= ε-closure (δ ({C}, 1))
= ε-closure ({B})
= {B}

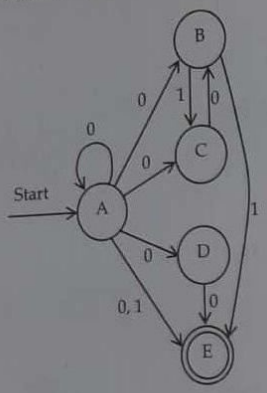
δ_N (D, 0) = ε-closure (δ (ε-closure (D), 0))
= ε-closure (δ ({D}, 0))
= ε-closure ({E})
= {E}

δ_N (D, 1) = ε-closure (δ (ε-closure (D), 1))
= ε-closure (δ ({D}, 1))
= ε-closure ({ϕ})
= ϕ

δ_N (E, 0) = ε-closure (δ (ε-closure (E), 0))
= ε-closure (δ ({E}, 0))
= ε-closure ({ϕ})
= ϕ

δ_N (E, 1) = ε-closure (δ (ε-closure (E), 1))
= ε-closure (δ ({E}, 1))
= ε-closure ({ϕ})
= ϕ

Thus, the resulting NFA is:



ε-NFA to DFA:

Given an ε-NFA $E = (Q, \Sigma, \delta, q_0, F_D)$, to construct a DFA equivalent to E, let $D = (Q', \Sigma, \delta', q_0', F')$ is a DFA equivalent to E.

Here,

Q′ = is a set of subsets of Q.

q_0' = ε-closure (q_0)

F′ = set of those states that contains at least one accepting state of E.

i.e. $F' = \{S / S \subseteq Q' \ \& \ S \cap F_D \neq \phi\}$

δ′ = is a transition function computed as;

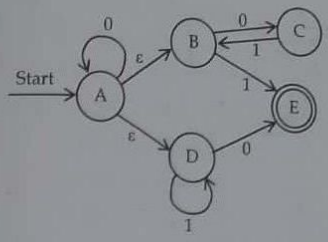
δ′ (s, a), where S is a member of Q′ & $a \in \Sigma$.

Let $S = \{p_1, p_2, \ldots p_k\}$

Compute $\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, r_3, \ldots, r_m\}$

Then, $\delta'(s, a) = \bigcup_{j=1}^{m}$ ε-closure (r_j)

Example: For the following ε-NFA, configure equivalent DFA:



The start state of given ε-NFA is A,

So start state for DFA will be;

q_0' = ε-closure (A)

= {A, B, D}

Here,

δ′ ({A, B, D}, 0) = ε-closure (δ ({A, B, D}, 0))
= ε-closure ({A, C, E})
= {A, B, C, D, E}

δ′ ({A, B, D}, 1) = ε-closure (δ ({A, B, D}, 1))
= ε-closure ({D, E})
= {D, E}

δ′ ({A, B, C, D, E}, 0) = ε-closure (δ ({A, B, C, D, E}, 0))
= ε-closure ({A, C, E})
= {A, B, C, D, E}

δ′ ({A, B, C, D, E}, 1) = ε-closure (δ ({A, B, C, D, E}, 1))
= ε-closure ({B, D, E})
= {B, D, E}

δ′ ({D, E}, 0) = ε-closure (δ ({D, E}, 0))
= ε-closure ({E})
= {E}

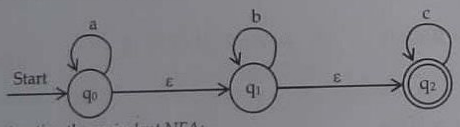
δ′ ({D, E}, 1) = ε-closure (δ ({D, E}, 1))
= ε-closure ({D})
= {D}

We calculate remaining states in same way. So that the final DFA becomes as;

	0	1
→ {A, B, D}	{A, B, C, D, E}	{E, D}
*{A, B, C, D, E}	{A, B, C, D, E}	{B, D, E}
*{D, E}	{E}	{D}
*{B, D, E}	{C, E}	{E, D}
*{E}	ϕ	ϕ
{D}	{E}	{D}
*{C, E}	ϕ	{B}
{B}	{C}	{E}
{C}	ϕ	{B}
ϕ	ϕ	ϕ

This transition table gives the final DFA of above ε-NFA.

Example: Convert following ε-NFA to NFA & DFA



1) *Constructing the equivalent NFA:*

Start state of NFA = Start state of ε-NFA

= q_0

Final state of NFA = Final state of ε-NFA

= q_2

For this, we compute as;

$\delta_N (q_0, a)$ = ε-closure (δ (ε-closure (q_0), a))

= ε-closure (δ ({q_0, q_1, q_2}, a))

= ε-closure ({q_0})

= {q_0, q_1, q_2}

$\delta_N (q_0, b)$ = ε-closure (δ (ε-closure (q_0), b))

= ε-closure (δ ({q_0, q_1, q_2}, b))

= ε-closure ({q_1})

= {q_1, q_2}

$\delta_N (q_0, c)$ = ε-closure (δ (ε-closure (q_0), c))

= ε-closure (δ ({q_0, q_1, q_2}, c))

= ε-closure ({q_2})

= {q_2}

$\delta_N (q_1, a)$ = ε-closure (δ (ε-closure (q_1), a))

= ε-closure (δ ({q_1, q_2}, a))

= ε-closure ({ϕ})

= {ϕ}

$\delta_N (q_1, b)$ = ε-closure (δ (ε-closure (q_1), b))

= ε-closure (δ ({q_1, q_2}, b))

= ε-closure ({q_1})

= {q_1, q_2}

$\delta_N (q_1, c)$ = ε-closure (δ (ε-closure (q_1), c))

= ε-closure (δ ({q_1, q_2}, c))

= ε-closure ({q_2})

= {q_2}

$\delta_N (q_2, a)$ = ε-closure (δ (ε-closure (q_2), a))

= ε-closure (δ ({q_2}, a))

= ε-closure ({ϕ})

= {ϕ}

$\delta_N (q_2, b)$ = ε-closure (δ (ε-closure (q_2), b))

= ε-closure (δ ({q_2}, b))

= ε-closure ({ϕ})

= {ϕ}

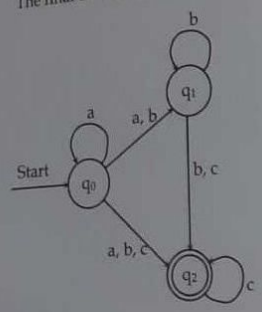
$\delta_N (q_2, c)$ = ε-closure (δ (ε-closure (q_2), c))

= ε-closure (δ ({q_2}, c))

= ε-closure ({q_2})

= {q_2}

The final DFA is as follows:



2) *Constructing the equivalent DFA:*

Start state of DFA = ε-closure (start state of ε-NFA)
= ε-closure (q_0) = {q_0, q_1, q_2}

Now,

δ' ({q_0, q_1, q_2}, a) = ε-closure (δ ({q_0, q_1, q_2}, a))
= ε-closure ({q_0}) = {q_0, q_1, q_2}

δ'({q_0, q_1, q_2}, b) = ε-closure (δ ({q_0, q_1, q_2}, b))
= ε-closure ({q_1}) = {q_1, q_2}

δ' ({q_0, q_1, q_2}, c) = ε-closure (δ ({q_0, q_1, q_2}, c))
= ε-closure ({q_2}) = {q_2}

Similarly,

δ' ({q_1, q_2}, a) = ε-closure (δ ({q_1, q_2}, a))
= ε-closure ({ϕ}) = {ϕ}

δ' ({q_1, q_2}, b) = ε-closure (δ ({q_1, q_2}, b))
= ε-closure ({q_1}) = {q_1, q_2}

δ' ({q_1, q_2}, c) = ε-closure (δ ({q_1, q_2}, c))
= ε-closure ({q_2}) = {q_2}

Similarly,

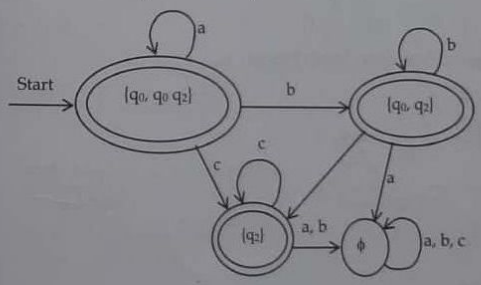
δ' ({q_2}, a) = ε-closure (δ ({q_2}, a))
= ε-closure ({ϕ}) = {ϕ}

δ' ({q_2}, b) = ε-closure (δ ({q_2}, b))
= ε-closure ({ϕ}) = {ϕ}

δ' ({q_2}, c) = ε-closure (δ ({q_2}, c))
= ε-closure ({q_2}) = {q_2}

The equivalent DFA is as follows:

δ:	a	b	c
→* {q_0, q_1, q_2}	{q_0, q_1, q_2}	{q_1, q_2}	{q_2}
*{q_1, q_2}	ϕ	{q_1, q_2}	{q_2}
*{q_2}	ϕ	ϕ	{q_2}
ϕ	ϕ	ϕ	ϕ



Finite State Machines with Output

Mealy Machines

A Mealy Machine is a finite state machine whose output depends on the present state as well as the present input. It can be described by a 6 tuple (Q, Σ, O, δ, X, q_0) where,

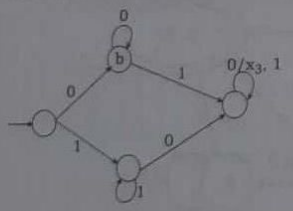
- Q is a finite set of states.
- Σ is a finite set of symbols called the input alphabet.

- O is a finite set of symbols called the output alphabet.
- δ is the input transition function where δ: $Q \times \Sigma \rightarrow Q$
- X is the output transition function where X: $Q \times \Sigma \rightarrow O$
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).

For Example

Present state	Next state			
	input = 0		input = 1	
	State	Output	State	Output
→ A	B	x_1	C	x_1
B	B	x_2	D	x_3
C	D	x_3	C	x_1
D	D	x_3	D	x_2

The state diagram of the above Mealy Machine is;



Moore Machine

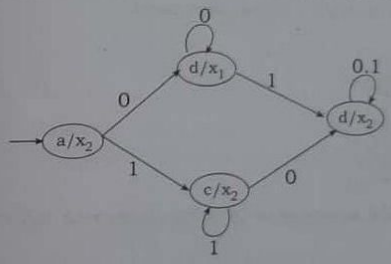
Moore machine is an finite state machine whose outputs depend on only the present state. A Moore machine can be described by a 6 tuple $(Q, \sum, O, \delta, X, q_0)$ where,

- Q is a finite set of states.
- $\sum$ is a finite set of symbols called the input alphabet.
- O is a finite set of symbols called the output alphabet.
- δ is the input transition function where δ: $Q \times \sum \rightarrow Q$
- X is the output transition function where X: $Q \rightarrow O$
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).

For Example

Present state	Next State		Output
	Input = 0	Input = 1	
→ a	B	c	x_2
b	B	d	x_1
c	c	d	x_2
d	d	d	x_3

The state diagram of the above Moore Machine is –

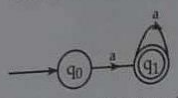


Mealy Machine vs. Moore Machine

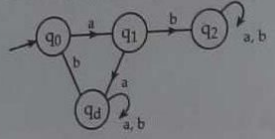
Mealy Machine	Moore Machine
Output depends both upon the present state and the present input	Output depends only upon the present state.
Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
The value of the output function is a function of the transitions and the changes, when the input logic on the present state is done.	The value of the output function is a function of the current state and the changes at the clock edges, whenever state changes occur.
Mealy machines react faster to inputs. They generally react in the same clock cycle.	In Moore machines, more logic is required to decode the outputs resulting in more circuit delays. They generally react one clock cycle later.

Some Additional Solved Examples

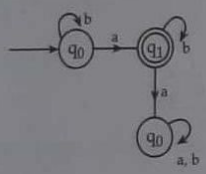
1. Draw a DFA to accept string of a's having at leat one a.



2. DFA to accept string of a's and b's starting with string ab.

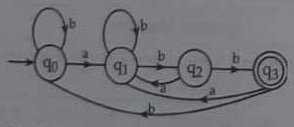


3. DFA to accept string of a's and b's having exactly one a.

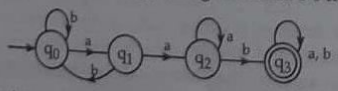


4. Construct a DFA, which accepts set of all string ending with abb, over Σ = {a, b}

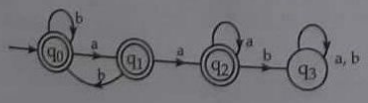
L = {abb, aabb, babb, aaabb, bbabb, bababb, . . .}



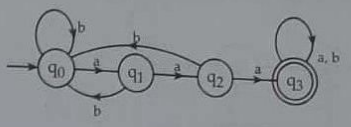
5. Design a DFA to accept string of a's and b's having substring aab.



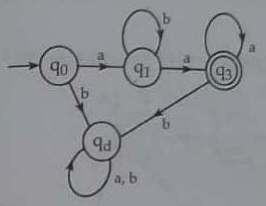
6. Design a DFA, to accept strings of a's and b's except having substring aab.



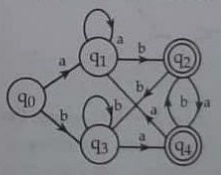
7. Construct a DFA to accept strings over Σ {a, b} having three consecutives a's.



8. Draw a DFA to accept string over Σ = {a, b} such that L {a w a | we $(a + b)^n$ Where n ≥ 0}

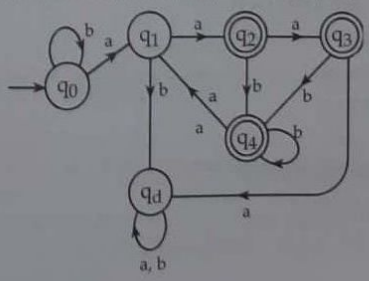


9. DFA to accept string of a's and b's ending with ab or ba or L = {w (ab + ba) /we (a, b)*} any $comb^n$ a and b

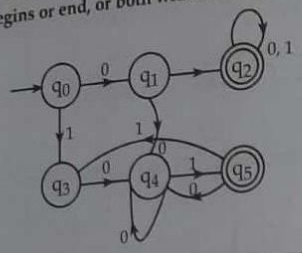


10. Design a DFA, L = {w: every string run of a's has length 20 and 3}

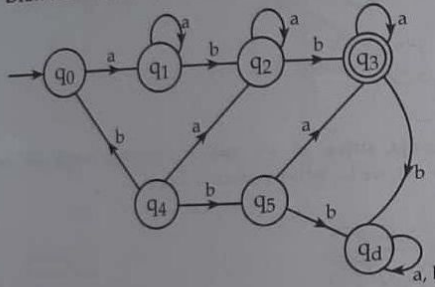
L = {aa, aaa, baaa, aab, baaa . . . aabaa, aac, baa, aabaaa}



11. Construct a DFA to accept set of all strings over the Σ = {0, 1} that either begins or end, or both with substring "01".



12. Draw a DFA to accept the language L = {w : n_a (w) ≥ 1, n_b (w) = 2}



13. Construct a DFA, which accepts strings over Σ = {0, 1} where the value of each string is represented as a binary number only the string as a binary number only the string representing zero modulo five is accepted.

i. d = {0, 1}, radix = 2, k = 5

ii. i = 0 to 4 (0 to k - 1)

let 0%5 = 0

1%5 = 1

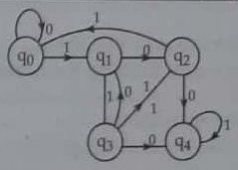
2%5 = 2

3%5 = 3

4%5 = 4

5%5 = 0

Remainder	d	J = (2 × i + d) mod 5 = 0	δ(q$_i$, d) = q$_j$
i = 0	0	J = (2 × 0 + 0) mod 5= 0	δ (q$_0$, 0) = q$_0$
	1	J = (2 × 0 + 1) mod 5 = 1	δ (q$_1$, 1) = q$_1$
i = 1	0	J = (2 × 1 + 0) mod 5 = 2	δ (q$_1$, 0) = q$_2$
	1	J = (2 × 1 + 1) mod 5 = 3	δ (q$_1$, 1) = q$_3$
i = 2	0	J = (2 × 2 + 0) mod 5 = 4	δ (q$_2$, 0) = q$_4$
	1	J = (2 × 2 + 1) mod 5 = 0	δ (q$_2$, 1) = q$_0$
i = 3	0	J = (2 × 3 + 0) mod 5 = 1	δ (q$_3$, 0) = q$_1$
	1	J = (2 × 3 + 1) mod 5 = 2	δ (q$_3$, 1) = q$_2$
i = 4	0	J = (2 × 4 + 0) mod 5 = 3	δ (q$_4$, 0) = q$_3$
	1	J = (2 × 4 + 1) mod 5 = 4	δ (q$_4$, 1) = q$_4$



1. Let us fix the alphabet Σ = {a,b}, then

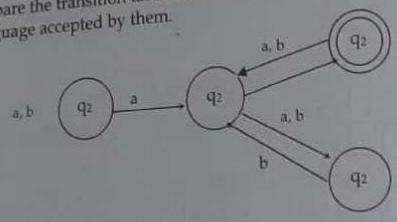
a. Construct a DFA that accepts only the language of all words with '*b*' as the second letter.

b. Construct a DFA that accepts only the strings ending with *baaa*.

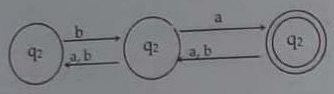
c. Construct a DFA that accepts only those strings that begin or end with a double letter like *aa*, *bb*.

d. Prepare the transition table for each of following DFAs and describe the language accepted by them.

i.



ii.

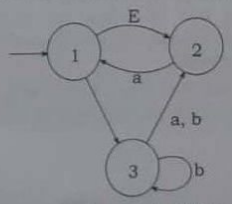


2. Let us define the alphabet set as $\Sigma = \{0,1\}$, then
   a. Build a DFA that accepts the strings that do not have 1110 as their prefix.
   b. Build a DFA that accepts those strings containing 010 as their substring but not containing the substring 0101.
   c. Build a DFA that accepts all the strings ending in 00.
3. Draw the state diagram corresponding to the DFA M= (Q, Σ, δ, q_0, F) where Q = {A, B, C, D, E}, Σ = {0, 1}, q_0= A, F={B, C, D, E} and δ is specified by following transition table,

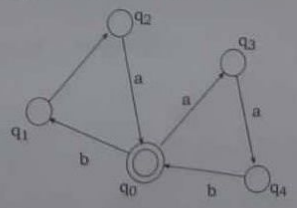
δ	0	1
A	A	B
B	C	D
C	E	A
D	B	C
E	D	E

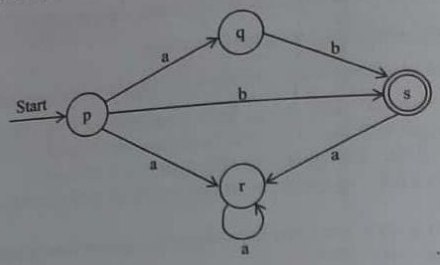
   Then for each of the following input strings, give the corresponding computation of automaton M, and say whether the computation is accepting or rejecting.
   a. 1001
   b. 0111
   c. 1110
4. Construct a DFA accepting binary strings with both the 0's and 1's are odd.
5. Construct a DFA accepting binary strings with odd number of 0's and even number of 1's.

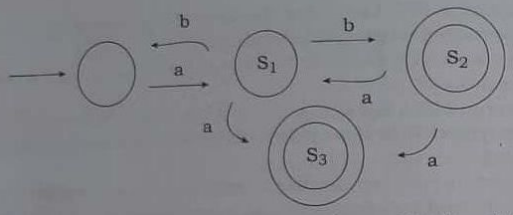
6. Construct a DFA that recognize the optionally signed integers.
7. Construct a DFA recognizing the C-identifiers.
8. Construct a DFA that recognizes the block (multiple line) comments in C.
9. DFA that accepts binary strings that are multiples of 3. For example, the machine should reject 1101 since 1101 is 13 in decimal, which is not divisible by 3. On the other hand, the machine accepts 1100 since it is 12 in decimal.
10. Construct a NFA accepting the strings over {0, 1} having at least two consecutive 0's or 1's.
11. Construct a NFA accepting the strings that accepts binary strings which starts with 1 and ends with 0.
12. Construct a NFA that accepts the set of strings over {0, 1} with at least two occurrences of substring 01 and ends with 11.
13. Construct a NFA over {a, b} that accepts the strings having *a* as one of the last three character.
14. Construct a NFA that accepts a set of strings over {a, b} containing an occurrence of the pattern *bb* or of the pattern *bab*. Show the acceptance of strings abbba and ababab.
15. Construct a NFA over {a,b} that accepts either empty strings or *ab*, or *aab*, or *aba*. Use extended transition function to determine whether the NFA accepts bbbabb and aab.
16. Use the subset construction method to convert the following nondeterministic finite automaton to an equivalent deterministic finite automaton.



17. Convert the following NFAs into their equivalent DFAs

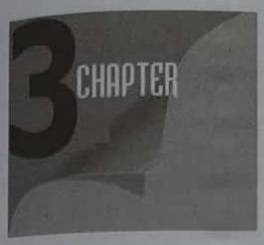






18. Construct a deterministic finite automata equivalent to $M = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_0\})$ where δ is given by $\delta(q_0, 0) = q_0$, $\delta(q_0, 1)=q_1$, $\delta(q_1, 0)=q_1$, $\delta(q_1, 1)=\{q_0, q_1\}$.
19. Construct a ϵ-NFA that accepts all strings of the form 0^k where k is a multiple of 2 or 3. (Remember that the superscript denotes repetition, not numerical exponentiation.) For example, the ϵ-NFA accepts the strings ϵ, 00, 000, 0000, and 000000, but not 0 or 00000.
20. Define extended transition function of ϵ-NFA. Convert ϵ-NFA constructed in question 19 into its equivalant NFA and DFA.

□□□



REGULAR EXPRESSIONS

CHAPTER OUTLINES

After studying this Chapter you should be able to:

- Regular Expression
- Regular Language
- Finite Automata and Regular Expression
- Pumping Lemma for Regular Expression
- Minimization of DFA

Regular Expression

Regular Expressions are those algebraic expressions used for describing regular languages, the languages accepted by finite automaton. Regular expressions offer a declarative way to express the strings we want to accept. This is what the regular expression offer that the automata do not. Regular expressions are a way of representing the generic syntax of the strings processed by finite automatons like DFA and NFA.

Any regular expression is composed of two components: symbols and operators. Symbol, $\sum$, is a set of inputs and operators are union (U), concatenation (.), Kleen Closure (*), Positive closure (+). A regular expression is built up out of simpler regular expression using a set of defining rules. Each regular expression 'r' denotes a language L(r). The defining rules specify how L(r) is formed by combining in various ways the languages denoted by the sub-expressions of 'r'.

Consider a regular expression (0 ∪ 1)01*; where 0, 1 are symbols and ∪, * are the operators. The language described by this expression is the set of all binary strings

1. that start with either 0 or 1 (this is indicated by (0 ∪1)),
2. for which the second symbol is 0 (this is indicated by 0), and
3. that end with zero or more 1s (this is indicated by 1*).

That is, the language described by this expression is {00, 001, 0011, 00111, . . ., 10, 101, 1011, 10111, . . .}.

Regular Operators

Basically, there are three operators that are used to generate the languages that are regular,

Union (U / |):

If L_1 and L_2 are any two regular languages then
$L_1 \cup L_2 = \{s \mid s \in L_1 \text{ or } s \in L_2\}$
e.g.
$L_1 = \{00, 11\}$, $L_2 = \{\epsilon, 10\}$
$L_1 \cup L_2 = \{\epsilon, 00, 11, 10\}$

Concatenation (.):

If L_1 and L_2 are any two regular languages then,
$L_1.L_2 = \{l_1.l_2 \mid l_1 \in L_1 \text{ and } l_2 \in L_2\}$
e.g. L1 = {00, 11} and L2 = {ϵ, 10}

Kleen Closure (*):

If L is any regular Language then, Kleen closure of L is;

$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup \ldots\ldots\ldots$

Positive Closure (+):

If L is any regular Language then, posotive closure of L is;

$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup \ldots\ldots\ldots$
$= L^* - L^0$

Precedence of regular operators:

- Closure (*) has highest precedence
- Concatenation (.) has next highest precedence.
- Union (U / | / +) has lowest precedence.

Formal Definition of Regular Expression

Let Σ be an alphabet, the regular expression over the alphabet Σ are defined inductively as follows;

- Φ is a regular expression representing empty language.
- ϵ is a regular expression representing the language of empty strings.
- if 'a' is a symbol in Σ, then 'a' is a regular expression representing the language {a}.
- f 'r' and's' are the regular expressions representing the language L(r) and L(s) then
  - r U s is a regular expression denoting the language L(r)UL(s).
  - r.s is a regular expression denoting the language L(r).L(s).
  - r* is a regular expression denoting the language(L(r))*.
  - (r) is a regular expression denoting the language (L(r))[L(r)] [defines parenthesis may be placed around regular expressions if we desire].

Regular Language

Let Σ be an alphabet, the class of regular language over Σ is defined inductively as;

- The regular expression Φ represents a regular language representing empty language
- The regular expression ϵ represents a regular language {ϵ} representing language of empty strings.

- For each a E Σ, the regular expression a represents a language L= {a}, which is a regular language.
- If $L_1, L_2, \ldots\ldots L_n$ are regular languages, then so is $L_1 U L_2 U \ldots\ldots\ldots UL_n$.
- If $L_1, L_2, L_3, \ldots\ldots\ldots\ldots L_n$ are regular languages, then so is $L_1.L_2.L_3.\ldots\ldots\ldots L_n$
- If L is a regular language, then so is L^*

Every language can be written as an expression using the operations of union, concatenation and Kleen closure. To simplify writing of these formulas, we adopt following conventions;

- Writing a symbol, say 'a' by itself is shorthand for {a}. That is, we promot a symbol to the singleton set containing it.
- The concatenation symbol . can be dropped as, in xy instead of x.y.
- Parentheses need be used only when it is necessary to override the normal precedence of * over . over U.

Each of these formulas are called as regular construction and by definition of the regular language are exactly those languages that are generated by regular construction.

Application of Regular Expression

- **Validation:** Determining that a string complies with a set of formatting constraints. Like email address validation, password validation etc.
- **Search and Selection:** Identifying a subset of items from a larger set on the basis of a pattern match.
- **Tokenization:** Converting a sequence of characters into words, tokens (like keywords, identifiers) for later interpretation.
- Lexer: Used as a lexer/tokenizer in lexical analysis step of compilers.

Algebraic Laws for Regular Expression

1. **Commutativity:** The union of regular expression is commutative but concatenation of regular expression is not commutative. i.e. if r and l are regular expressions representing like languages L(r) and L(l) then,

   r + l = l + r i.e.r U l = l U r

   but r.l ≠ l.r.

2. **Associativity:** The unions as well as concatenation of regular expressions are associative.

   i.e. if l, r, s are regular expressions representing regular languages L(l),L(r) and L(s) then,

   l + (r + s) = (l + r) + s

   And l.(r.s) = (l.r).s

3. **Distributive law:** Φ is identity for union. i.e. for any regular expression r representing regular language L(l), L(r) and L(s) then,

   l(m + n) = lm + ln ------ left distribution.

   (m + n)l = ml + nl ------ right distribution.

4. **Identity law:**

   Φ is identity for union. i.e. for any regular expression r representing regular expression L(r).

   r + Φ = Φ + r = r i.e. ΦUr=r.

   ∈ is identity for concatenation. i.e. ∈.r = r = r. ∈

5. **Annihilator:** An annihilator for an operator is a value such that when the operator is applied to the annihilator and some other value, the result is annihilator.

   Φ is annihilator for concatenation. i.e. Φ.r = r. Φ = Φ

6. **Idempotent law of union:** For any regular expression r representing the regular language L(r), r + r = r. This is the idempotent law of union.
7. **Law of closure:** for any regular expression r, representing the regular language L(r),

   $(r^*)^*=r^*$

   Closure of Φ = $\Phi^* = \epsilon$

   Closure of ∈ = $\epsilon^* = \epsilon$

   Positive closure of r, $r^+ = rr^*$.

Example 1: Consider Σ = {0, 1}, some regular expressions over Σ;

a. 0^*10^*={w | w contains a single 1}

b. $\Sigma^*1\ \Sigma^*$={w | w contains at least one 1}

c. $\Sigma^*001\ \Sigma^*$ = {w | w contains the string 001 as substring}

d. $(\Sigma\ \Sigma)^*$ or $((0+1)^*.(0+1)^*)$ = {w | w is string of even length}

e. $1^*(01^*01^*)^*$ = {w | w is string containing even number of zeros}

f. $0^*10^*108108$ ={w | w is a string with exactly three 1's}

g. For string that have substring either 001 or 100, the regular expression is $(1 + 0)^*.001.(1 + 0)^* + (1 + 0)^*.(100).(1 + 0)^*$

h. For strings that have at most two 0's with in it, the regular expression is $1^*.(0 + \epsilon).1^*.(0 + \epsilon).1^*$

i. For the strings ending with 11, the regular expression is $(1 + 0)^*.(11)^+$

Example 2: Regular expression that denotes the C identifiers:

(alphabet + _)(alphabet + digit + _)*

Theorem 1: If L, M and N are any languages, then L(M U N) = L(M) U L(N).

Proof:

Let w = xy be a string, now to prove the theorem we need to show that a string w is in L(M U N) if and only if it is in L(M) U L(N).

Now first consider "if part":

Let w ε L(M) U L(N)

This implies that, w ε L(M) or w ε L(N) (by union rule)

i.e. xy ε L(M) or xy ε L(N)

Also,

xy εL(M) implies x ε L and y ε M (by concatenation rule)

And,

xy ε L(N) implies x ε L and y ε N (by concatenation rule)

This implies that

x εL and y ε (M U N)

=>xy ε L(M U N) (concatenating above)

=>w ε L(M U N)

Now consider "only if" part:

Let w ε L(M U N) =>xy ε L(M U N)

Now,

xy ε L(M U N) => x ε L and y ε (M U N) (by concatenation)

y ε (M U N) => y ε M or y ε N (by union rule)

Now, we have x ε L

Here if y ε M then xy ε L(M) (by concatenation)

And if y ε N then xy ε L(N) (by concatenation)

Thus, if xy ε L(M) =>xy ε (L(M) U L(N)) (by union rule)

xy ε L(N) =>xy ε (L(M) U L(N))

i.e. w ε (L(M) U L(N)

Thus,

We have, L (M U N) = L(M) U L(N)

Finite Automata and Regular Expression

The regular expression approach for describing regular language is fundamentally different from the finite automaton approach. However, these two notations represent exactly the same set of languages, called regular languages. We can convert the finite automata to its equivalent regular expression and vice versa.

Regular Expression to finite Automata

To construct the automata from RE we first construct the automata for the basis expressions: single symbols, ϵ, and ∅.These automata are then combined into larger automata that accept union, concatenation, or closure of the language accepted by smaller automata.

Theorem 2: Every language defined by a regular expression is also defined by a finite automaton. [For any regular expression r, there is an ϵ-NFA that accepts the same language represented by r].

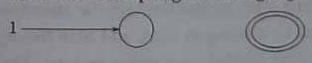
Proof:

Let L =L(r) be the language for regular expression r, now we have to show there is an ϵ-NFA E such that L(E) =L.

The proof can be done through structural induction on r, following the recursive definition of regular expressions.

For this we know Φ, ϵ, a are the regular expressions representing languages {Φ}; an empty language, language for empty strings and {a} respectively.

The ϵ-NFA accepting these languages can be constructed as;



This forms the **basis step.**

Now, **Inductive step:**

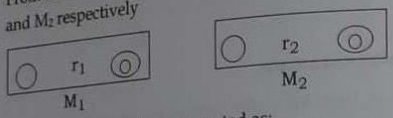
Let r be a regular expression representing language L(r) and r_1,r_2 be regular expressions for languages L(r_1) and L(r_2), such that

For $L(r) = L(r_1) \oplus L(r_2)$ we have the regular expression;

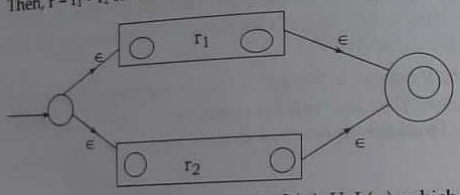
$r = r_1 \oplus r_2$ where $\oplus$ = |+ (union), . (concatenation), *(closure)|

Now Case I: $\oplus$ = + (union)

From basis step we can construct ϵ-NFA's for r_1 and r_2. Let the ϵ-NFA's be M_1 and M_2 respectively



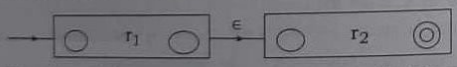
Then, $r = r_1 + r_2$ can be constructed as:



The language of this automaton is $L(r_1) \cup L(r_2)$ which is also the language represented by expression r_1+r_2.

Case II: $\oplus$ = . (Concatenation)

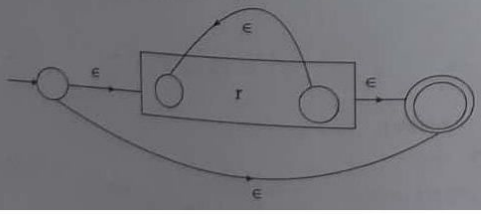
Now, $r = r_1.r_2$ can be constructed as;



Here, the path from starting to accepting state go first through the automaton for r_1, where it must follow a path labeled by a string in $L(r_1)$, and then through the automaton for r_2, where it follows a path labeled by a string in $L(r_2)$. Thus, the language accepted by above automaton is $L(r_1).L(r_2)$.

Case III: $\oplus$ = *(Kleen closure)

Now, $r = r^*$ Can be constructed as;



Clearly language of this ϵ-NFA is $L(r^*)$ as it can also just ϵ as well as string in $L(r)$, $L(r)L(r)$, $L(r)L(r)L(r)$ and so on. Thus covering all strings in $L(r^*)$.

Finally, for regular expression (r), the automaton for r also serves as the automaton for (r), since the parentheses do not change the language defined by the expression.

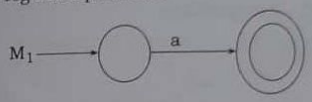
Proved.

Examples

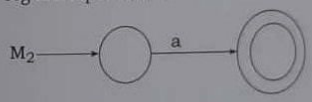
1. Construct the ϵ-NFA for the RE (ab∪ a)*, where the alphabet is |a, b|.

Solution:

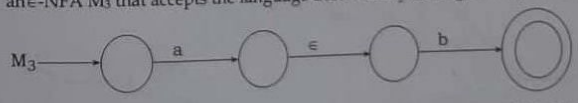
First, constructinganϵ-NFA M_1 that accepts the language described by the regular expression a:



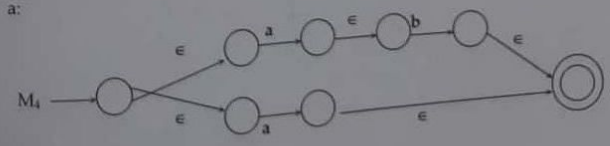
Next, constructinganϵ-NFA M_2 that accepts the language described by the regular expression b:



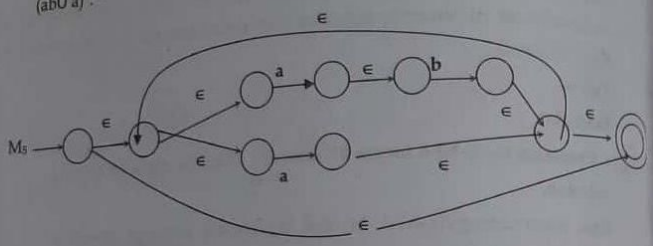
Next, applying **Case II** given in the proof of Theorem 2 to M_1 and M_2. This gives anϵ-NFA M_3 that accepts the language described by the regular expression ab:



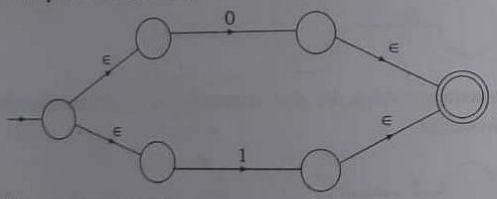
Next, applying **Case I** given in the proof of Theorem 2 to M_3 and M_1. This gives anϵ-NFA M_4 that accepts the language described by the regular expression abU a:



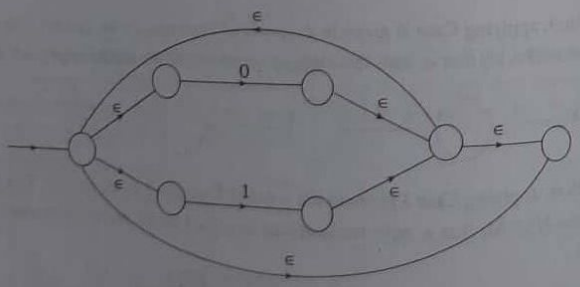
Finally, applying **Case III** given in the proof of Theorem 2 to M_3 and M_1. This gives an ∈-NFA M_5 that accepts the language described by the regular expression (abU a)*:



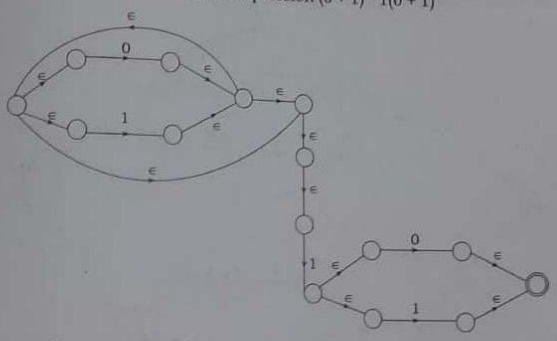
Example 2: For regular expression (1 + 0) the ∈-NFA is:



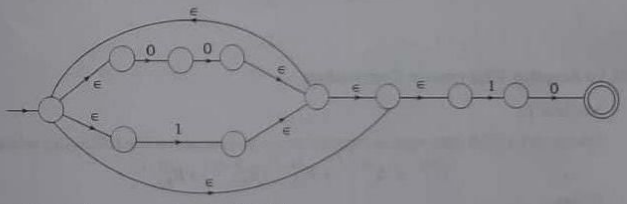
The ∈-NFA for (0 + 1)*



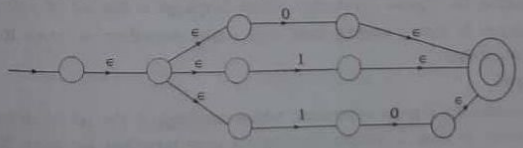
Now, ∈-NFA for whole regular expression (0 + 1) * 1(0 + 1)



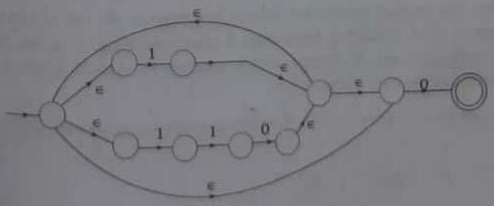
For regular expression (00+1)*10 the ∈-NFA is as



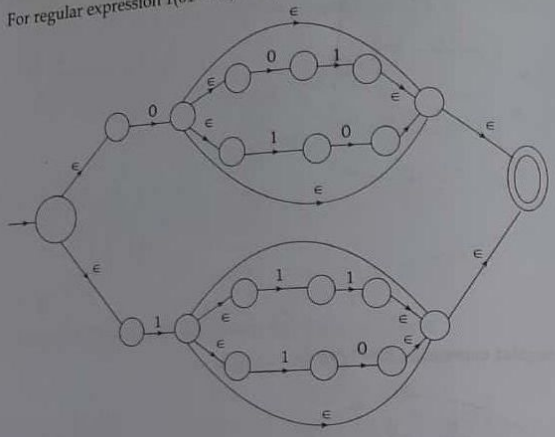
For regular expression (0 + 1 + 10)



For Regular Expression (1+110)*0



For regular expression $1(01 + 10)^* + 0(11 + 10)^*$



DFA to Regular Expression Conversion

Method 1:

To convert a DFA into regular expression we have to solve the following relation:

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

Where,

$R_{ij}^{(k)}$ represents the regular expression whose language is the set of strings w which reaches to state j starting from state i after travelling no more than k intermediate states.

For example:

$R_{12}^{(0)}$ represents the regular expression whose language is the set of strings w which reaches to state 2 starting from state 1 after travelling no more than 0 intermediate states.(i.e. set of strings which reaches to state 2 from 1 directly).

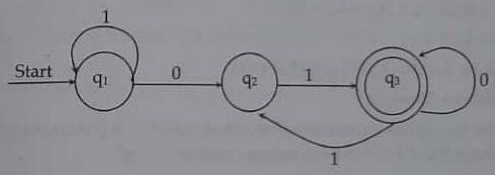
$R_{24}^{(1)}$ represents the regular expression whose language is the set of strings w which reaches to state 4 starting from state 2 after travelling no more than 1 intermediate states.(i.e. set of strings which reaches to state 4 from 2 after travelling 0 or 1 intermediate state).

Thus, if there are total n states in DFA then its equivalent regular expression $R_{1n}^{(n)}$ is given by:

$$R_{1n}^{(n)} = R_{1n}^{(n-1)} + R_{1n}^{(n-1)}(R_{nn}^{(n-1)})^* R_{nn}^{(n-1)}$$

Example 3: Convert the following DFA into its equivalent Regular expression.

$R_{11}^{(1)}$	1*
$R_{12}^{(1)}$	1*0
$R_{13}^{(1)}$	∅
$R_{21}^{(1)}$	∅
$R_{22}^{(1)}$	∫
$R_{23}^{(1)}$	1
$R_{31}^{(1)}$	∅
$R_{32}^{(1)}$	1
$R_{33}^{(1)}$	0+ ∫



$R_{11}^{(0)}$	1+ϵ
$R_{12}^{(0)}$	0
$R_{13}^{(0)}$	∅
$R_{21}^{(0)}$	∅
$R_{22}^{(0)}$	ϵ
$R_{23}^{(0)}$	1
$R_{31}^{(0)}$	∅
$R_{32}^{(0)}$	1
$R_{33}^{(0)}$	0+ ∫

$R_{11}^{(2)}$	1*
$R_{12}^{(2)}$	1*0
$R_{13}^{(2)}$	1*01
$R_{21}^{(2)}$	∅
$R_{22}^{(2)}$	∫
$R_{23}^{(2)}$	1
$R_{31}^{(2)}$	∅
$R_{32}^{(2)}$	1
$R_{33}^{(2)}$	0+ ∫ +11

The table above gives the various stages of the construction. The basis expressions are shown in the first table. For instance, $R_{11}^{(0)}$ contains the term 1 since both its beginning and ending states are state 1. It contains the term 1 because there is an arc beginning and ending at state 1 corresponding to input 1. $R_{12}^{(0)}$ is 0, since there is an arc labelled 0 beginning at state 1 and ending at state 2 and so on.

Now to calculate $R_{12}^{(0)}$ we use the formula:

$$R_{12}^{(1)} = R_{12}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)}$$
$$= 0+(1+\epsilon)(1+\epsilon)^* 0$$
$$= 1^*0$$

Similarly other values are computed.

Finally computing $R_{13}^{(3)}$ using equation:

$$R_{13}^{(3)} = R_{13}^{(2)} + R_{13}^{(2)}(R_{33}^{(2)})^* R_{33}^{(2)}$$
$$= 1^*01 + 1^*01(0+\epsilon+11)^*(0+\epsilon+11)$$
$$= 1^*01(0+11)^*$$

Hence the RE for the above DFA is $1^*01(0+11)^*$

Method 2: Arden's Theorem

Let p and q be the regular expressions over the alphabet Σ, if p does not contain any empty string then r = q + rp has a unique solution; $r + qp^*$.

Proof:

Here, r = q + rp ..... (i)

Let us put the value of r = q + rp on the right hand side of the relation (i), so;

$r = q + (q + rp)p$

$r = q + qp = rp^2$.....(ii)

Again putting value of r = q + rp in relation (ii), we get;

$r = q + qp + (q + rp)\,p^2$

$r = q + qp + qp^2 + qp^3$.....

Continuing in the same way, we will get as;

$r = q + qp + qp^2 + qp^3$.....

$r = q(\epsilon + p + p^2 + p^3 +$.....

Thus $r = qp^*$ Proved.

Use of Arden's rule to find the regular expression for DFA:

To convert the given DFA into a regular expression, here are some of the assumptions regarding the transition system:

- The transition diagram should not have the ∈-transitions.

- There must be only one initial state.
- The vertices or the states in the DFA are as;

  $q_1, q_2, \ldots\ldots\ldots\ldots q_n$ (Any qi is final state)
- W_{ij} denotes the regular expression representing the set of labels of the edjes from q_i to q_j. Thus we can write expressions as;

$q_1 = q_1w_{11} + q_2w_{12} + q_3w_{31} + \ldots\ldots\ldots\ldots q_nw_{n1} + \epsilon$

$q_2 = q_1w_{12} + q_2w_{22} + q_3w_{32} + \ldots\ldots\ldots\ldots + q_nw_{n2}$

$q_3 = q_1w_{13} + q_2w_{23} + q_3w_{33} + \ldots\ldots\ldots\ldots + q_nw_{n3}$

.....

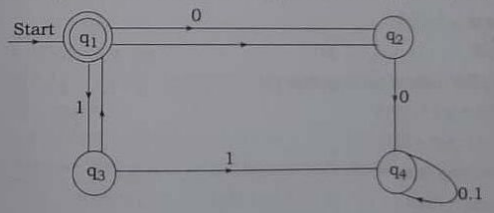
.....

.....

$q_n = q_1w_{1n} + q_2w_{n2} + q_3w_{n3} + \ldots\ldots\ldots\ldots q_nw_{nn}$

Solving these equations for q_i in terms of w_{ij} gives the regular expression.

Examples 4: Convert the following DFA'S into regular expression.



Let the equations are

$q_1 = q_21 + q_30 + \epsilon$.....(i)

$q_2 = q_10$.....(ii)

$q_3 = q_11$.....(iii)

$q_4 = q_20 + q_31 + q_40 + q_41$.....(iv)

Putting the values of q_2 and q_3 in (i)

$q_1 = q_101 + q_110 + \epsilon$

i.e. $q_1 = q_1(01 + 10) + \epsilon$

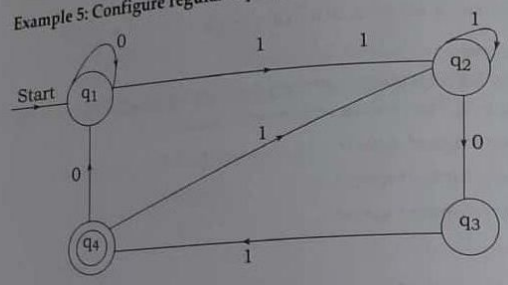
i.e. $q_1 = \epsilon + q_1(01 + 10)$ (since r = q + rp)

i.e. $q_1 = \epsilon(01 + 10)^*$ (using Arden's rule)

Since, q1 is final state, the final regular expression for the DFA is

$\epsilon(01 + 10)^* = (01+10)^*$

Example 5: Configure regular expression for following



Let the equation are:

$q_1 = q_1 0 + q_4 0 + \epsilon$ ..... (i)

$q_2 = q_1 1 + q_2 1 + q_4 1$ .....(ii)

$q_3 = q_2 0$ .....(iii)

$q_4 = q_3 1$ .....(iv)

Putting value of q_3 in (iv)

$q_4 = q_2 01$ .....(v)

Now, putting this value of q_4 in equation (ii)

$q_2 = q_1 1 + q_2 1 + q_2 011$

$= q_1 1 + q_2(1 + 011)$

Now using Arden's rule;

$q_2 = q_1 1(1 + 011)^*$

Then from equation (v)

$q_4 = q_1 1(1 + 011)^*10$ .....(vi)

Now from equation (i) &(vi), putting value of q_4 in (i0

$q_1 = q_1 0 + q_1 1(1 + 011)^*)010 + \epsilon$

$= q_1(0 + 1(1 + 011)^*010) + \epsilon$

Using Arden's rule we get,

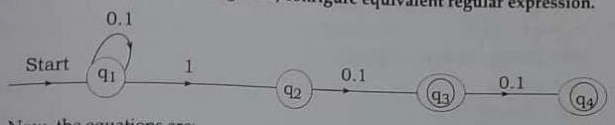
$q_1 = \epsilon(0 + 1(1 + 011)^*010)^*$

Putting the value of q1in equation (vi)

$q_4 = (0 + (1 + 011)^*010))1(1 + 011)^*01$

Since, q_4 is final state in DFA, this is the equivalent regular expression for the DFA.

Example 6: Given following NFA, configure equivalent regular expression.



Now, the equations are:

$q_1 = q_1 0 + q_1 1 + \epsilon$ .....(i)

$q_2 = q_1 1$ .....(ii)

$q_3 = q_2 0 + q_2 1$ .....(iii)

$q_4 = q_3 0 + q_3 1$ .....(iv)

From equation (i),we have,

$q_1 = q_1(0 + 1) + \epsilon$

$q_1 = \epsilon + q_1(0 + 1)$

Using Arden's rule,

$q_1 = \epsilon(0 + 1)^* = (0 + 1)^*$

Now from (ii)

$q_2 = (0 + 1)^*1$

Similarly from (iii)

$q_3 = q_2(0 + 1) = (0 + 1^*1(0 + 1)$

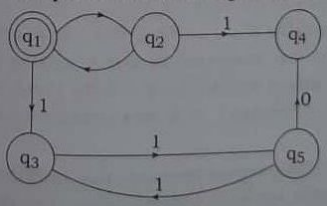
And from (iv)

$q_4 = q_3(0 + 1) = (0+1)^*1(0 + 1)(0 + 1)$

Since we have q_3 & q_4 as final state so final regular expression is

$q_3 + q_4 = (0 + 1)^*1(0 + 1) + (0 + 1)^*1(0 + 1)(0 + 1)$

Example 7: Given following finite automaton configure regular expression.



Let the equation be:

$q_1 = q_2 0 + \epsilon$ .....(i)

$q_2 = q_1 0$ .....(ii)

$q_3 = q_1 1 + q_5 1$ .....(iii)

$q_4 = q_2 1 + q_3 0 + q_4 0 + q_5 0$.....(iv)

$q_5 = q_3 1$.....(v)

Here, putting the value of q_2 in (i),

$q_1 = q_1 00 + \epsilon$

i.e. $q_1 = \epsilon + q_1 00$

Now, using Arden's rule;

$q_1 = \epsilon (00)^* = (00)^*$

Putting q_3 in q_5;

$q_5 = (q_1 1 + q_5 1)1$

$= ((00)^* 1 + q_5 1)1$

$= (00)^* 11 + q_5 11$

Using Arden's rule;

$q_5 = (00)^* 11(11)^*$

Since, here q_1 & q_5 are final states of DFA, the final regular expression is:

$q_1 + q_5 = (00)^* + (00)^* 11(11)^*$

Pumping Lemma for Regular Expression

Any regular language can be represented by each of the formalism; DFA, NFA or regular expression.

If L is any finite language, then L is regular. Because, for example, we could produce an NFA or DFA having |L| transitions, with each labeled by a different string in L.

But when L is infinite language, it must contain arbitrarily long strings. Our intuition would tell us that, in general, the longer a string x is, the more memory/states, it will take to determine if x ε L. Since DFAS have only a constant amount of memory/states, they would not be able to process long strings unless the strings had some kind of repeated patterns in the strings of L. Similarly, any regular expression that generates L must have a pattern enclosed in an *.

This leads us to suspect that any infinite regular language must contain long strings that have some type of simple repetitive pattern in them. This fact is captured by "Pumping Lemma".

Lemma: Let L be a regular language. Then, there exists an integer constant n such that for any x ε L with $|x| \geq n$, there are strings u, v, w such that $x = uvw$, $|uv| \leq n$, $|v| > 0$. Then $uv^k w \in L$ for all $k \geq 0$.

Proof:

Suppose L is a regular language, then L is accepted by some DFA M. Let M has n states. Also L is infinite so M accepts some string x of length n or greater. Let length of x, $|x| = m$ where $m \geq n$.

Now suppose;

$X = a_1 a_2 a_3 \ldots\ldots\ldots\ldots a_m$ where each $a_i \in \Sigma$ be an input symbol to M.

Now, consider for $j = 1, \ldots\ldots\ldots\ldots n$, q_j be states of M

Then,

$\hat{\delta}(q_0, x) = \hat{\delta}(q_0, a_1 a_2 \ldots\ldots\ldots a_m)$ [q_0 being start state of M]

$= \hat{\delta}(q_1, a_1 a_2 \ldots\ldots\ldots a_m)$

.....

.....

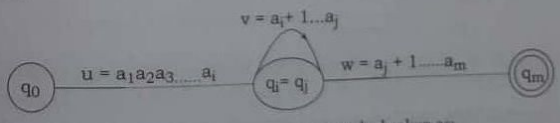
.....

$= \hat{\delta}(q_m, \epsilon)$ [q_m being final state]

Since $m \geq n$, and m has only n states, so by pigeonhole principle, there exists some I and j;

$0 \leq i < j \leq m$ such that $q_i = q_j$

So,



Now, this entire string $x = a_1 \ldots\ldots\ldots a_m$ can be broken as:

$x = uvw$ such that

$u = a_1 a_2 \ldots\ldots\ldots a_i$

$v = a_{i+1} \ldots\ldots\ldots a_j$

$w = a_{j+1} \ldots\ldots\ldots a_m$

i.e. string $a_{i+1} \ldots\ldots\ldots a_j$ takes M from state q_i back to itself since $q_i = q_j$. So we can say M accepts $a_1 a_2 \ldots\ldots a_i (a_{i+1} \ldots\ldots a_j)^k a_{j+1} \ldots\ldots\ldots a_m$ for all $k \geq 0$.

Hence, $uv^k w \in L$ for all $k \geq 0$.

Application of Pumping Lemma

The application pumping lemma is to prove any language is not a regular language.

For example:

Example 1: Show that language, $L = \{0^n 1^r \mid n \geq 0\}$ is not a regular language.

Let L is a regular language. Then by pumping lemma, there are strings u, v, w with $v \geq 1$ such that $uv^k w \in L$ for $k \geq 0$.

Case I:

Let v contain 0's only. Then, suppose $u = 0^p$, $v = 0^q$, $w = 0^r 1^s$; $p + q + r = s$ (as we have $0^r 1^r$) and $q > 0$

Now, $uv^k w = 0^p (0^q)^k 0^r 1^s = 0^{p+qk+r} 1^s$

Only these strings in $0^{p+qk+r} 1^s$ belongs to L for k=1 otherwise not.

Case II

Let v contains 1's only. Then $u = 0^p 1^q$, $v = 1^r$, $w = 1^s$

Then $p = q + r + s$ and $r > 0$

Now, $0^p 1^q (1^r)^k 1^s = 0^p 1^{q+rk+s}$

Only those strings in $0^p 1^{q+rk+s}$ belongs to L fpr k =1 otherwise not.

Case III

V contains 0's and 1's both. Then, suppose,

$u = 0^p$, $v = 0^q 1^r$, $w = 1^s$;

$p + q = r + s$ and $q + r > 0$

Now, $uv^k w = 0^p (0^q 1^r) 1^s = 0^{p+qk} 1^{rk+s}$

Only those strings in $0^{p+qk} 1^{rk+s}$ belongs to L for k=1, otherwise not. (As it contains 0 after 1 for k>1 in the string.) Thus, the language L is not a regular language.

Example 8: Show that the language L consisting of all palindromes over $(0 + 1)^*$ is not regular.

Suppose L is regular. Then there exists a constant n satisfying the conditions of the pumping lemma.

Let $x = 0^n 1 0^n$.

Using pumping lemma we can break $x = uvw$ so that $v \neq \epsilon$ and $|uv| \leq n$.

Obviously both u and v consist only of 0's. Suppose $u = 0^i$ and $v = 0^j$. Then we know by the pumping lemma that $uv^0 w$ is in L.

Thus $0^{n-j} 1 0^n \in L$. This is not possible. Hence the language is not regular.

Closure Properties of Regular Languages

Closure under Union

If L and M are regular languages, so is L UM.

Let L and M be the languages of regular expressions R and S, respectively. Then R+S is a regular expression whose language is L U M

Closure under intersection

If L and M are regular languages, so is $L \cap M$.

Let L and M be the languages of regular expressions R and S, respectively then $R \cap S$ is a regular expression whose language is $L \cap M$

Closure under Concatenation

If L and M are regular languages, so is L . M.

If L and M be the regular languages of regular expressions R and S, respectively then R . S is a regular expression whose language is L . M.

Closure under Kleene Closure

If L is the regular languages of regular expressions R then R^* is a regular expression whose language is L^*.

Closure under Complement

The complement of a language L (with respect to an alphabet Σ such that Σ^* contains L) is $\Sigma^* - L$. Since Σ^* is surely regular by the property of closure under Kleen closure, the complement of a regular language is always regular.

MINIMIZATION OF DFA

Given a DFA M, that accepts a language L. Now, DFA minimization is a problem to find an equivalent DFA M' having less number of state than M but accepts the same language L. It involves identifying the equivalent states and distinguishable states. So that after finding the equivalent states we can replace them by single state.

Equivalent States: Two states p & q are called equivalent states, denoted by $p \equiv q$ if and only if for each input string x, $\hat{\delta}(p, x)$ is a final state if and only if $\hat{\delta}(q, x)$ is a final state.

Distinguishable state: Two states p & q are said to be distinguishable states if (for any) there exists a string x, such that $\hat{\delta}(p, x)$ is a final state $\hat{\delta}(q, x)$ is not a final state.

Method 1: Table Filling Algorithm

For identifying the pairs (p, q) with $p \neq q$;

Step 1: List all the pairs of states (p,q) from the given DFA for which $p \neq q$.

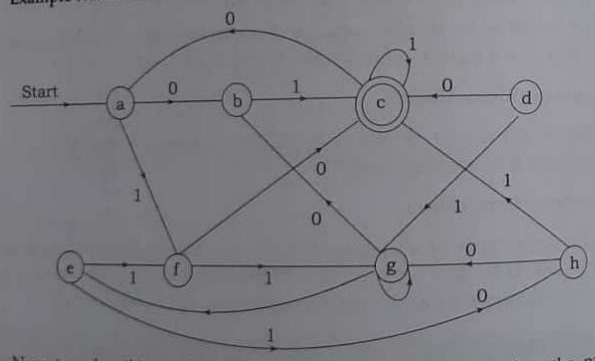
Step 2: Make a sequence of passes through each pairs.

Step 3: On first pass, mark the pair for which exactly one element is final (F). On each sequence of pass, mark the pair (r, s) if for any $a \in \Sigma$, $\delta(r, a) = p$ and $\delta(s, a) = q$ and (p, q) is already marked.

Step 4: After a pass in which no new pairs are to be marked, stop

The marked pairs (p, q) are those for which $p \neq q$ and unmarked pairs are those for which $p \equiv q$. i.e. the marked pairs are distinguishable states and unmarked pairs are equivalent states. The minimized DFA contains the equivalent states found in the table as one single state.

Example 9: Minimize following DFA;



Now to solve this problem first we should determine weather the pair is distinguishable or not.

For pair (b, a)

(δ(b, 0), δ(a, 0)) = (g, h) – unmarked

(δ(b, 1), δ(a, 1)) = (c, f) – marked

For pair (d, a)

(δ(d, 0), δ(a, 0)) = (c, b) – marked

Therefore (d, a) is distinguishable.

For pair (e, a)

(δ(e, 0), δ(a, 0)) = (h, h) – unmarked.

(δ(e, 1), δ(a, 1)) = (f, f) –unmarked.

[(e, a) is not distinguishable)]

For pair (g, a)

(δ(g, 0), δ(a, 0)) = (a, g) – unmarked.

(δ(g, 1), δ(a, 1)) = (e, f) – unmarked

For pair (h, a)

(δ(h, 0), δ(a, 0)) = (g, h) –unmarked

(δ(h, 1), δ(a 1) = (c, f) – marked

Therefore (h, a) is distinguishable.

For pair (d, b)

(δ(d, 0), δ(b,0)) = (c, g) – marked

Therefore (d, b) is distinguishable.

For pair (e, b)

(δ(e, 0), δ(b,0)) = (h, g) –unmarked

(δ(e, 1), δ(b,1) = (f, c) – marked.

For pair (f, b)

(δ(f, 0), δ(b,0)) = (c, g) – marked

For pair (g, b)

(δ(g, 0), δ(b, 0)) = (g, g) – unmarked

(δ(h, 1), δ(b, 1)) = (e, c) – marked

For pair (h, b)

(δ(h, 0), δ(b, 0)) = (g, g) – unmarked

(δ(h,1), δ(b,1)) = (c, c) - unmarked.

For pair (e, d)

(δ(e, 0), δ(d, 0)) = (h, c) – marked

(e, d) is distinguishable.

For pair (f, d)

(δ(f, 0), δ(d, 0)) = (c, c) – unmarked

(δ(f,1), δ(f,1)) = (g, g) - unmarked.

For pair (g, d)

(δ(g, 0), δ(d, 0)) = (g, c) – marked

For pair (h, d)

(δ(h, 0), δ(d, 0)) = (g, c) – marked

For pair (f, e)

(δ(f, 0), δ(e, 0)) = (c, h) – marked

For pair (g, e)

$(\delta(g, 0), \delta(e, 0)) = (g, h)$ – unmarked

$(\delta(g,1), \delta(e,1)) = (e, f)$ -marked.

For pair (h, e)

$(\delta(h, 0), \delta(e, 0)) = (g, h)$ – unmarked

$(\delta(h,1), \delta(e,1)) = (c, f)$ -marked.

For pair (g, f)

$(\delta(g, 0), \delta(f, 0)) = (g, c)$ – marked

For pair (h, f)

$(\delta(h, 0), \delta(f, 0)) = (g, c)$ – marked

For pair (h, g)

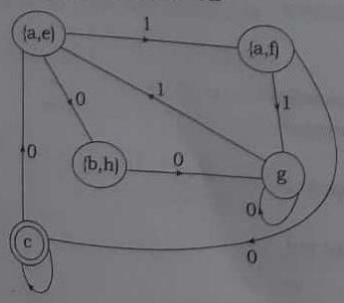
$(\delta(h, 0), \delta(g, 0)) = (g, g)$ – unmarked

$(\delta(h,1), \delta(g,1)) = (c, e)$ -marked.

Thus (a, e), (b, h) and (d, f) are equivalent pairs of states.

b	x						
c	x	x					
d	x	x	x				
e		x	x	x			
f	x	x	x		x		
g	x	x	x	x	x	x	
h	x		x	x	x	x	x
	a	b	c	d	e	f	g

Hence the minimized DFA is



Method 2: Partition Approach

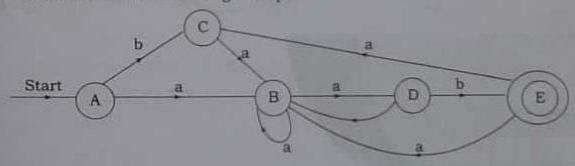
Step 1: Maintain a partition of states of DFA. Initially partition consists of two groups; the accepting states and non accepting states.

Step 2: The fundamental step is to take some group of states say $A=\{s_1, s_2, s_3, \ldots\ldots s_k\}$ and some input symbol a, and look at what transitions states $s_1, s_2, s_3, \ldots\ldots s_k$ have on input symbol a. If these transitions are the states that fall into two or more different groups of the current partition, then we must split of so that the transitions from the subsets of A are all confined to a single group of the current partition.

Suppose for example, that $s_1$and s_2 go to states t_1 and t_2 on input 'a' and t_1 and t2 are in different groups of the partition. Then we must split A into at least two subsets so that one subset contains s_1 and other s_2. Note that t_1 and t_2 are distinguished by some string w, so s_1 and s_2 are distinguished by string a^w.

Step 3: We repeat this process of splitting groups in the current partition until no more groups need to be split.

Let us consider the following example



The initial partition consists of:

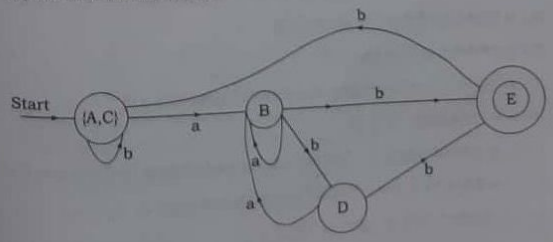
{A, B, C, D} {E}

= {A, B, C,} {D} {E} [since, D goes to E on input b]

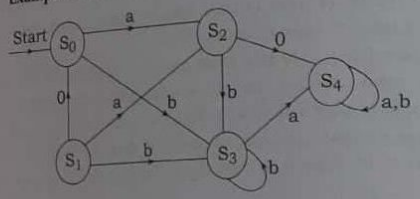
= {A, c} {B} {D} {E} {B goes to D on input b]

This is the final partition; as A and C have same transition for a and b.

So, the minimized DFA is;



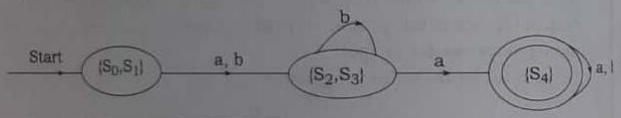
Example 10: Minimize the following DFA:



$\{S_0, S_1, S_2, S_3\}$ $\{S_4\}$

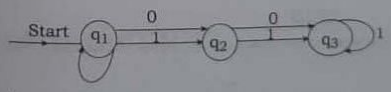
$\{S_0, S_1\}$ $\{S_2, S_3\}$ $\{S_4\}$

So, the final DFA is:

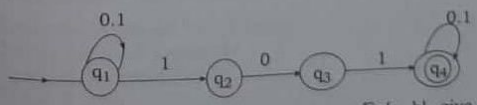


1. Write the regular expression over {0, 1} for following strings
   a. {w| w begins with 1 and ends with 0}
   b. {w| w contains at least three 1's}
   c. {w| w has length at least three and its third symbol is a 0}
   d. {w| w is not ending with 0}
   e. {w| w is a set of strings of 2 or more symbols followed by 3 or more 0's}
   f. {w| w is either no 1 preceding a 0 or no 0 preceding a 1}
   g. {w| w contains even number of 0's}

2. Describe in words the strings in each of these regular sets
   a. 1*0
   b. 1*00*
   c. 0(1+0)*
   d. (0+1)(0+1)*00
3. Write regular expressions that accepts the dates of the format
   – 12/31/2008
   – December 31, 2008
   – Wednesday, December 31, 2008
4. Write regular expressions that validate
   a. A website url
   b. An email address
   c. An IPV4 address
   d. A MAC address
   e. A Zip Code
5. Configure equivalent epsilon NFA for following regular expressions
   a. (((00)*(11)*))U01)*
   b. Ø*
   c. (ab U a)*
6. Convert following Finite Automata to equivalent regular expressions;
   a.

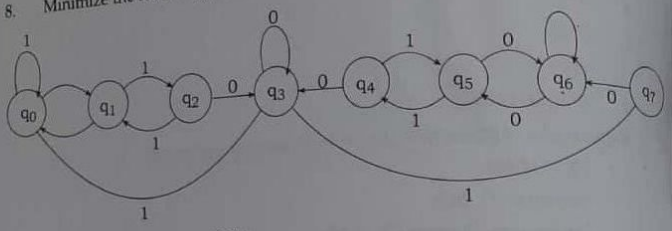


   b.

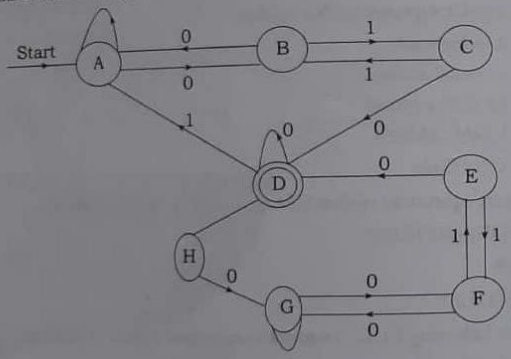


7. For each of the following languages over ∑={a, b}, give two strings that are members and two strings that are not members.
   a. ∑*a∑*b∑*a∑*
   b. (∈ U a)b
   c. (a U ba U bb) ∑*

8. Minimize the following DFAs using the table filling algorithm.



9. Minimize the following DFA:



□□□

CHAPTER 4

CONTEXT FREE GRAMMAR

CHAPTER OUTLINES

After studying this Chapter you should be able to:

- Introduction to Context Free Grammar
- Derivation using a Grammar Rule
- Parse Tree /Derivation Tree
- Regular Grammar
- Simplification of CFG
- Chomsky Normal Form
- Left recursive grammar
- Greibach Normal Form (GNF)
- Pumping Lemma for Context free languages
- Closure Property of Context Free Languages

Introduction to Context Free Grammar

The term *grammar* applied to a language refers to the mechanism for constructing phrases and sentences that belong to the language. A grammar consist a set of rules, by which strings in a language can be generated.

Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings.

Context-free grammars are studied in fields of theoretical computer science, compiler design, and linguistics. CFG's are used to describe programming languages and parser programs in compilers can be generated automatically from context-free grammars.

The CFG are more powerful than the regular expression as they have were expressive power than the regular expression. Generally regular expressions are useful for describing the structure of lexical constructs as identified, keywords, constants etc. But they do not have the capability to specify the recursive structure of the program constructs. However, the CFG are capable to define any of the recursive structures also. Thus CFG can define the language that are regular as well as that language that are not regular.

Formally, a context free grammar is defined by 4-tuples (V, T, P, S)

where,

V= set of variables or non terminals

T= Set of terminal symbols

P = Set of rules or productions. Each production rule has the form $A \rightarrow \beta$, where $A \in V$ and $\beta \in (V \cup T)^*$

S = Start symbol and $S \in V$.

Thus, CFG consist of a collection of substitution rules, also called productions with each rules being a variable and a string of variables and terminals. Each rule appears as a line in the grammar.

The symbols involved in the production may be variable or terminal symbols. The variable symbols are represented by capital letters. The terminals are analogous to the input alphabet and are often represented by lower case letters. One of the variables is designate as a start variable S. It usually occurs on the left

hand side of the topmost rule. In CFG, the left hand side of rule always consists of a variable and the right hand side consists of variables, terminals or combination of them.

For example,

Rule 1: $S \rightarrow \epsilon$

Rule 2: $S \rightarrow 0S1$

This is a CFG defining the grammar of all the strings with equal no of 0's followed by equal no of 1's.

Here, the two rules define the production P; ϵ, 0, 1 are the terminal symbols defining T; S is a variable symbol defining V; S is start symbol.

In the above example there are two rules in production set. Thus formally, the grammar is defined as $(V=\{S\}, T=\{\epsilon, 0, 1\}, P=\{S \rightarrow \epsilon, S \rightarrow 0S1\}, S=\{S\})$.

The production set can also be written as $P=\{S \rightarrow \epsilon \mid 0S1\}$ where the symbol | is used to separate two productions from the same variable S.

Recursive Structure in Grammars

Consider a CFG representing the language over $\Sigma = \{a, b\}$ which is palindrome language, defined by following productions;

$S \rightarrow \epsilon|a|,b$

$S \rightarrow aSa$

$S \rightarrow bSb$

The above grammar generates the strings like ϵ, a, b, aa, bb, aba, bab, abba, baab, bbbb etc. Each of the strings in palindrome has a structure of wwr, where wr is a reverse of string w. The regular expressions do not have capacity to represent such structure of strings but context free grammars can, as above. A significant reason behind this is that regular expressions are a way of expressing regular grammars which are processed by the state machines like DFA and NFA. Such machines do not have any storage memory associated with it so it's hard to remember the string w while processing wr. In contrast to this CFG represents context free language which is proceed by a machine called push down automaton. It has memory structure to store the history of processed inputs, thus can parse the wr by storing w in its memory and hence can determine the recursive structure easily.

Meaning of context free in CFG

Consider an example;

s→amb

m→A|B

A→∈ |Aa

B→ ∈|bB

Now, consider a string aaAb, which is an intermediate stage in the generation of aaab. It is natural to call the strings "aa" and "b"" that surround the symbol A, the "context" of A in this particular string. Now the rule A→ aA says that we can replace A by the string aA no matter what the surrounding strings are; in other words, independently of the context of A. However if there is a production of form aaAb→aBb (but not of the form A→B) , the grammar is context sensitive since A, can be replaced by B only when it is surrounded by the strings "aa" and "b". This in contraxt to context free is context sensitive in the grammar.

Example 1: CFG for language 0^n1^n where n≥0.

Solution: S→ ∈

S→ 0S1

Example 2: CFG for algebraic expression involving binary operators + and -, left and right parenthesis, and a single identifier a.

Solution:

S→ S+S

S→ S-S

S→ (S)

S→ a

Example 3: CFG for language L = {aa, ab, ba, bb}

Solution:

S→ AA

A→ a

A→ b

Example 4: CFG for language 1^n where n≥0.

Solution:

S→ 1S

S→∈

Example 5: CFG for (0+1)*

Solution:

S→ 1S

S→ 0S

S→∈

Example 6: CFG for string having atleast length two. i.e (0+1)(0+1)(0+1)*

Solution:

S→AAB

A→ 1 | 0

B→ 1S | 0S | ∈

Derivation using a Grammar Rule

The process of producing strings using the production rules of the grammar is called derivation.

A derivation of a context free grammar is a finite sequence of strings β_0 β_1 β_2……β_n such that:

- For $0 \le i \le n$, the string $\beta_i \varepsilon (V \cup T)^*$
- $B_0 = S$
- For $0 \le i \le n$, there is a production of P that applied to β_i yields β_{i+1}
- $B_n \varepsilon T^*$

There are two possible approaches of derivation:

- Body to head (Bottom Up) approach.
- Head to body (Top Down) approach.

Body to head (Bottom Up) approach

Here, we take strings known to be in the language of each of the variables of the body, concatenate them, in the proper order, with any terminals appearing in the body, and infer that the resulting string is the language of the variables in the head.

Consider grammar,

S→ S + S | S/S | (S) | S-S| S*S| a

Here to generate a + (a*a) / a - a

Now, by this approach.

S.N	String Inferred	Variable	Production	String Used
1	A	S	S→ a	a; string 1
2	a*a	S	S→S*S	a*a; string 2
3	(a*a)	S	S→(S)	String 1 and string 2; string 3
4	(a*a) / a	S	S→S/S	String 1 and string 3; string 4
5	a + (a*a) / a	S	S→S+S	String 1 and String 4; string 5
6	a + (a*a) / a - a	S	S→S-S	String 5 and string 1

Thus, in this process we start with any terminal appearing in the body and use the available rules from body to head.

Head to body (Top Down) approach

Here, we use production from head to body. We expand the start symbol using a production, whose head is the start symbol. Here we expand the resulting string until all strings of terminal are obtained. Here we have two approaches

1. Leftmost derivation
2. Rightmost derivation

Leftmost derivation

A derivation in a context-free grammar is a *leftmost* derivation (LMD) if, at each step, a production is applied to the leftmost variable occurred in the current string. It is denoted by $\underset{lm}{\rightarrow}$.

Consider a grammar defined as

S → S*S

S→ S+S

S→ S-S

S→ S/S

S→ (S)

S→ a

Now the leftmost derivation of the string $a + (a * a)$ is shown as:

$S \underset{lm}{\rightarrow} S+S$ (Rule S→ S+S)

$S \underset{lm}{\rightarrow} a+S$ (Rule S→ a)

$S \underset{lm}{\rightarrow} a+(S)$ (Rule S→ (S))

$S \underset{lm}{\rightarrow} a+(S*S)$ (Rule S→ S*S)

$S \underset{lm}{\rightarrow} a+(a*S)$ (Rule S→ a)

$S \underset{lm}{\rightarrow} a+(a*a)$ (Rule S→ a)

Rightmost derivation

A derivation in a context-free grammar is a *right most* derivation (RMD) if, at each step, a production is applied to the rightmost variable occurred in the current string. It is denoted by $\underset{rm}{\rightarrow}$.

Consider a grammar defined as

S → S*S

S→ S+S

S→ S-S

S→ S/S

S→ (S)

S→ a

Now the rightmost derivation of the string $a + (a * a)$ is shown as:

$S \underset{rm}{\rightarrow} S+S$ (Rule S→ S+S)

$S \underset{rm}{\rightarrow} S+(S)$ (Rule S→ (S))

$S \underset{rm}{\rightarrow} S+(S*S)$ (Rule S→ S*S)

$S \underset{rm}{\rightarrow} S+(S*a)$ (Rule S→ a)

$S \underset{rm}{\rightarrow} S+(a*a)$ (Rule S→ a)

$S \underset{rm}{\rightarrow} a+(a*a)$ (Rule S→ a)

#Consider a Grammar;

S→ aAS | a

S→ SbA | SS | ba

Given a string aaabaaa, give leftmost and rightmost derivation.

Leftmost Derivation:

$S \underset{lm}{\rightarrow} aAS$

$S \underset{lm}{\rightarrow} aSSS$; rule A→ SS

$S \underset{lm}{\rightarrow} aaSS$; rule S → a

$S \underset{lm}{\rightarrow} aaaSS$; rule S→ aAS

$S \underset{lm}{\rightarrow} aabaSS$; rule A→ ba

$S \underset{lm}{\rightarrow} aaabaaS$; rule S→ a

$S \underset{lm}{\rightarrow} aaabaaa$; rule S→ a

Rightmost Derivation:

$S \underset{rm}{\rightarrow} aAS$

$S \underset{rm}{\rightarrow} aAa$; rule S → a

$S \underset{rm}{\rightarrow} aSSa$; rule A→ SS

$S \underset{rm}{\rightarrow} aSaASa$; rule S→ aAS

$S \underset{rm}{\rightarrow} aSaAaa$; rule S→ a

$S \underset{rm}{\rightarrow} aSabaaa$; rule A→ ba

$S \underset{rm}{\rightarrow} aaabaaa$; rule S →a

Given CGF:

S→ aB | bA

A→ a | aS | bAA | ∈

B→ b | bS | aBB

Given string babaababbaa, shows leftmost and rightmost derivation.

Leftmost derivation:

$S \underset{lm}{\rightarrow} bA$

$S \underset{lm}{\rightarrow} baS$; rule A→ aS

$S \underset{lm}{\rightarrow} babA$; rule S→ bA

$S \underset{lm}{\rightarrow} babbAAS$; rule A→ bAA

$S \underset{lm}{\rightarrow} babbaAS$; rule A→ a

$S \underset{lm}{\rightarrow} babbaaS$; rule A→ aS

$S \underset{lm}{\rightarrow} babbaabA$; rule A→ bA

$S \underset{lm}{\rightarrow} babbaabaS$; rule A→ aS

$S \underset{lm}{\rightarrow} babbaababA$; rule A→ bA

$S \underset{lm}{\rightarrow} babbaababbAA$; rule A→ bAA

$S \underset{lm}{\rightarrow} babbaababbaA$; rule A→ a

$S \underset{lm}{\rightarrow} babbaababbaa$; rule A→ a

Rightmost Derivation:

$S \underset{rm}{\rightarrow} bA$

$S \underset{rm}{\rightarrow} baS$; rule A→ a S

$S \underset{rm}{\rightarrow} babA$; rule S→ bA

$S \underset{rm}{\rightarrow} babbAA$; rule A→ bAA

$S \underset{rm}{\rightarrow} babbAa$; rule A→ a

$S \underset{rm}{\rightarrow} babbaSa$; rule A→ a S

$S \underset{rm}{\rightarrow} babbaaBa$; rule S→ a B

$S \underset{rm}{\rightarrow} babbaabSa$; rule B→ b S

$S \underset{rm}{\rightarrow} babbaabaBa$; rule S→ a B

$S \underset{rm}{\rightarrow} babbaababSa$; rule B→ b S

$S \underset{rm}{\rightarrow} babbaababbAas$; rule S→ b A

$S \underset{rm}{\rightarrow} babbaababbaa$; rule A→ a

Direct derivation

During the derivation of a string from given productions a grammar, if there is a production $\alpha_1 \Rightarrow \alpha 2$

then α_2 can be derived directly from α_1, hence it is direct derivation.

Otherwise If α_2 can be derived from α_1 with zero or more steps of the derivation, then it is derivation and is represented as $\alpha_1 \overset{*}{\Rightarrow} \alpha_2$

For example; $S \rightarrow asa \mid a\,b \mid a \mid b \mid \in \mid$

Direct derivation;- $S \Rightarrow ab$

$S \Rightarrow aSa$

$\Rightarrow aasaa$

$\Rightarrow aaabaa$

Thus, $s \overset{*}{\Rightarrow}$ aaabaa is just a derivation.

Language of CFG

Let G = (V, T, P and S) is a context free grammar. Then the language of G denoted by L (G) is the set of terminal strings that have derivation from the start symbol in G.

i.e. $L(G) = \{x \in T^* \mid S \rightarrow^* x\}$

The language generated by a CFG is called the Context Free Language (CFL).

PARSE TREE /DERIVATION TREE

Parse tree is a tree representation of strings of terminals using the productions defined by the grammar. A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.

Parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding the replacement order.

Formally, given a Context Free Grammar G = (V, T, P and S), a parse tree is a n-ary tree having following properties:

- Root node which is labeled by the start symbol.
- Interior nodes each of which in the parse tree is variables.
- Leaf node each of which in the parse is labeled by a terminal symbol or $\in$.

If an interior node is labeled with a non terminal A and its childrens are x_1, x_2 ..... x_n from left to right there is a production P as:

$A \rightarrow x_1, x_2, \ldots\ldots\ldots\ldots x_n$ for each $x_i \in T$.

Consider the grammar

1. $S \rightarrow S*S$
2. $S \rightarrow S+S$
3. $S \rightarrow S-S$
4. $S \rightarrow S/S$
5. $S \rightarrow (S)$
6. $S \rightarrow a$

The parse tree for the string $a + (a * a)$ is shown as follows:

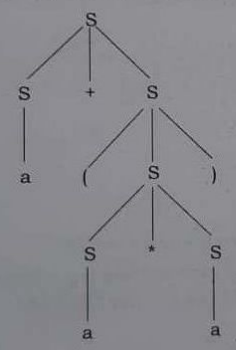


Fig: Derivation tree for $a + (a * a)$

Consider the Grammar

1. $S \rightarrow I$
2. $S \rightarrow S+S$
3. $S \rightarrow S * S$
4. $S \rightarrow (S)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$

8. I→Ib
9. I→I0
10. I→I1

Construct the parse tree for $a * (a+b00)$.

The parse tree is;

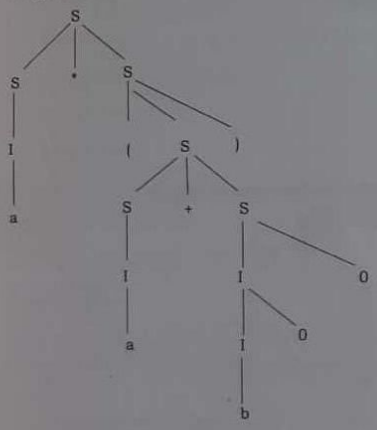


Fig: Parse tree for a * (a+b00)

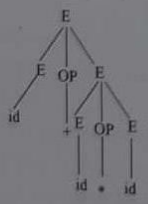
Example 7: Construct a grammer defining arithmetic expression and generate parse tree for id + id *id and (id + id) * (id + id).

Solution:

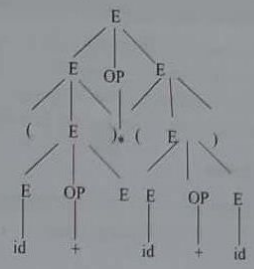
E → E OPE | (E) | id

OP → + | - | * | / | ↑

Parse tree for id + id * id



Parse tree for (id + id) * (id + id)



Ambiguity in a CFG

A grammar G = (v_1, t_1 p_1 s) is said to be ambiguity if there is a string w ∈ L (G) for which we can derive two or more distinct derivation tree rooted. at s and yielding w.

In other words, a grammar is ambiguous if it can produce more than one leftmost or more than one rightmost derivation for the same string in the language of the grammar.

Example 8: Consider the grammar

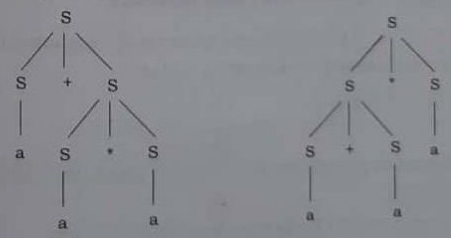
S → a | S + S | S * S | (S)

The string a + a * a has the two leftmost derivations

S → S + S → a + S → a + S * S → a + a * S → a + a * a

S → S * S → S + S * S → a + S * S → a + a * S → a + a * a

which correspond to the derivation trees



Hence the above grammar is ambiguous grammar.

Inherently ambiguity

A context free language L is said to be inherently ambiguous if all its grammars are ambiguous. E.g. L= {0^i1^j2^k | i=j or j=k}.

Regular Grammar

A regular grammar represents a language which is represented by regular expressions. The regular grammar is accepted by NFA and DFA and the language of the grammar is called regular language. A regular grammar is a subset of CFG. The regular grammar may be may be either left or right linear.

Right Linear Regular Grammar

A regular grammar is which all of the productions are of the form $A \rightarrow wB$ or $A \rightarrow w$ where $A, B \in V$ and $w \in T^*$ is called right linear.

For Example;

$S \rightarrow 00B \mid 11C$

$B \rightarrow 11C \mid S \mid 1$

$C \rightarrow 00B \mid 00$

Left Linear Regular Grammar

A grammar is which all of the production are of the form $A \rightarrow Bw$ or $A \rightarrow w$, where $A, B \in V$ and $w \in T^*$ is called left linear.

For example

$S \rightarrow B00 \mid C11$

$B \rightarrow C11 \mid S \mid 1$

$C \rightarrow B00 \mid 00$

Equivalence of Regular Grammar and Finite Automata

1. Let G = (V, T, P, S) be a right linear grammar of a language L(G), we can construct a finite automata M accepting L(G) as;

   M = (Q, T, δ, [S]. {[∈]})

Where,

Q - consists of symbols [α] such that α is either S or a suffix from right hand side of a production in P.

T - is the set of input symbols ∑ which are terminal symbols of G.

[S] - is a start symbol of grammar G which is start state of finite automata.

{[∈]} - is the set of final states in finite automata.

δ - is a transition function and is defined as:

If A is a variable then,

$\delta([A], \in) = [\alpha]$ such that $A \rightarrow \alpha$ is a production.

If 'a' is a terminal and α is in $(T \cup V)^*$ then

$\delta([a\alpha], a) = [\alpha]$

Example 9:

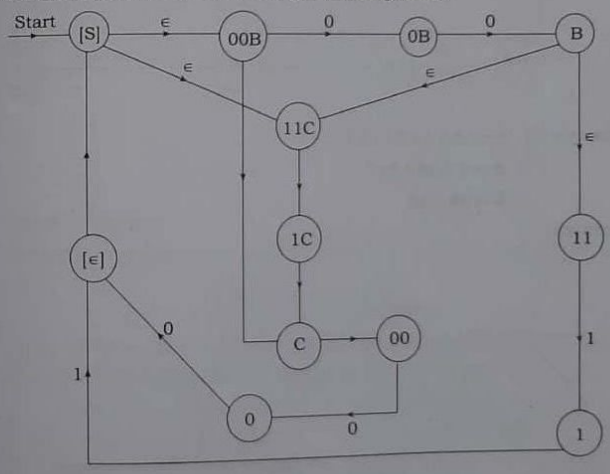
$S \rightarrow 00B \mid 11C \mid \in$

$B \rightarrow 11C \mid 11$

$C \rightarrow 00B \mid 00$

Solution:

The finite automaton for the above grammar is given as;



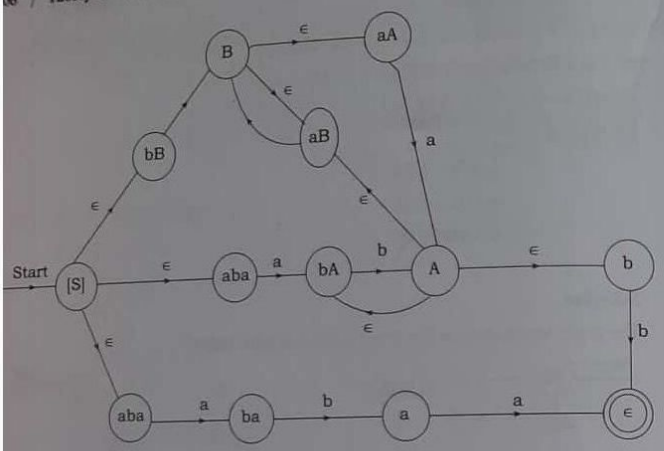
Example 10: $S \rightarrow abA \mid bB \mid aba$

$A \rightarrow b \mid aB \mid bA$

$B \rightarrow aB \mid aA$

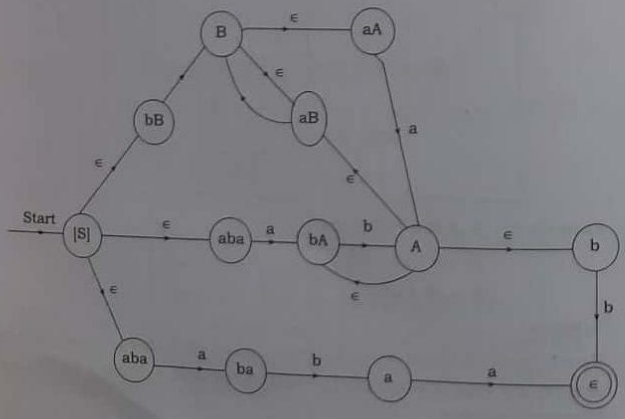
Solution:

Here ∈ does not appear on the body part of any prod ıctions, but we introduce state [∈] and select it as accepting state. Thus the finite automaton for the above grammar is given as:

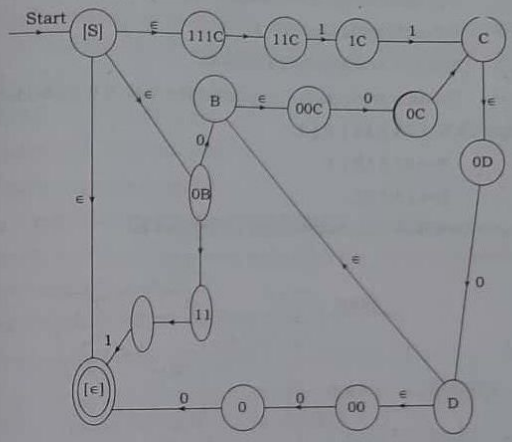


Example 11: S → abA | bB | aba
A → b | aB | bA
B → aB | aA

Solution:



Example 12: S → 0B | 111C | ϵ
B → 00C | 11
C → 00B | 0D
D → 0B | 00



Another Approach

If the regular grammar is of the form in which all the productions are in the form as;

A → aB or A → a, where A, B ε V and a ε T

Then, following steps can be followed to obtain an equivalent finite automaton that accepts the language represented by above set of productions.

a. The number of states in finite automaton will be one more than the number of variables in the grammar.(i.e. if grammar contain 4 variables then the automaton will have 5 states)
   - Such one more additional state in the final state of the automaton.
   - Each state in the automaton is a variable in the grammar.

b. The start symbol of regular grammar is the start state of the finite automaton.

c. If the grammar contains ϵ as S → *ϵ, S being start symbol of grammar, then start state of the automaton will also be final state.

d. The transition function for the automaton is defined as;

- For each production A→ aB

  $\delta(A, a) = B.0$

  So there is an arc from state A to B labeled with a.
- For each production A→ a,

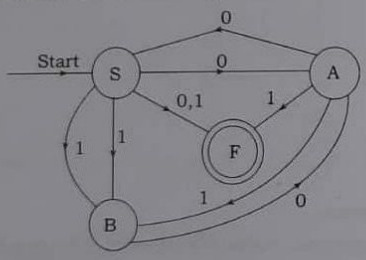
  $\delta(A, a)$ = final state of automaton
- For each production A→ ∈, make the state A as finite state.

Example 13: S→ 0A | 1B | 0 | 1

A→ 0S | 1B | 1

B→ 0A | 1S

The equivalent finite automaton can be configured as;



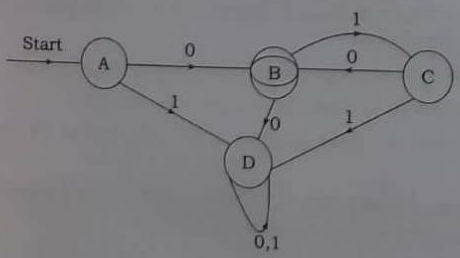
Example 14: A→ 0B | 1D | 0

B→ 0D | 1C | ∈

C→ 0B | 1D | 0

D→ 0D | 1D

The equivalent finite automaton can be configured as;



SIMPLIFICATION OF CFG

In a CFG, it may not be necessary to use all the symbols in V and T, or all the productions in P for deriving strings from the grammar. Thus simplification of CFG refers to eliminate those symbols and productions in G which are not useful or are unnecessary for the derivation of strings. The Simplification of CFG involves following;

i. Eliminating ∈-productions.

ii. Eliminating unit productions

iii. Eliminating useless symbols

Eliminating ∈-productions (Eliminating Nullable variables)

To eliminate ∈-productions we have to find nullable variables. Nullable variables are the variables that produce ∈.

Recursive Definition of Nullable variables

- Every variable A for which there is a production $A \rightarrow \in$ is nullable.
- If $A_1, A_2, \ldots, A_k$ are nullable variables (not necessarily distinct), and $B \rightarrow A_1A_2 \ldots A_k$ then B is also nullable variable.

Algorithm for elimination of ∈-productions

1. Find all the nullable variables.
2. For each production $A \rightarrow A_1A_2 \ldots A_k$, construct all productions $A \rightarrow X$ where X is obtained from '$A_1A_2 \ldots A_k$' by removing zero, one or multiple nullable variables found in step 1.
3. Combine the original productions with the result of step 2 and remove ∈-productions.

Example 15: Consider the grammar:

S→AB

A→aAA | ∈

B→bBB | ∈

Here,

A→∈	A is nullable
B→∈	B is nullable
S→AB	Since A and B are both nullable hence S is nullable.

Now, Removing of ϵ -production we get the grammar:

$S \rightarrow AB \mid A \mid B$

$A \rightarrow aAA \mid aA \mid a$

$B \rightarrow bBB \mid bB \mid b$

Example 16: Consider the grammar:

$S \rightarrow ABC$

$A \rightarrow BB \mid \epsilon$

$B \rightarrow CC \mid a$

$C \rightarrow AA \mid b$

Here,

$A \rightarrow \epsilon$ A is nullable.

$C \rightarrow AA \overset{*}{\rightarrow} \epsilon$ C is nullable

$B \rightarrow CC \overset{*}{\rightarrow} \epsilon$ B is nullable

$S \rightarrow ABC \overset{*}{\rightarrow} \epsilon$ S is nullable

Now for removal of ϵ -production;

$S \rightarrow ABC \mid AB \mid BC \mid AC \mid A \mid B \mid C$

$A \rightarrow BB \mid B$

$B \rightarrow CC \mid C \mid a$

$C \rightarrow AA \mid A \mid b$

Example 17: Consider the grammar

$S \rightarrow AACD$

$A \rightarrow aAB \mid \epsilon$

$C \rightarrow aC \mid a$

$D \rightarrow aDa \mid bDb \mid \epsilon$

Here, A and D are nullable. So, removing ϵ-productions:

$S \rightarrow AACD \mid ACD \mid AAC \mid AC \mid C$

$A \rightarrow aAb \mid ab$

$C \rightarrow aC \mid a$

$D \rightarrow aDa \mid abDb \mid bb$

Eliminating Unit Production

A unit production is a production of the form $A \rightarrow B$, where A and B are both variables.

To eliminate the unit productions, first find all of the unit pairs.

The unit pairs are;

- (A, A) is a unit pair for any variable A as $A \overset{*}{\rightarrow} A$
- If we have $A \rightarrow B$ then (A, B) is unit pair i.e. (A, B) is unit pair.
- If (A, B) is unit pair i.e. $A \rightarrow B$, and if we have $B \rightarrow C$ then (A, C) is also a unit pair.

Now, to eliminate unit pair (A , B) i.e. unit production $A \rightarrow B$ simply replace B by its body part and then remove all the unit productions.

Example 18: Consider the Grammar

$S \rightarrow S + T \mid T$

$T \rightarrow T^* F \mid F$

$F \rightarrow (S) \mid a$

Remove the unit production

Solution:

Here, $S \rightarrow T$ So, (S, T) is unit pair.

$T \rightarrow F$ So, (T, F) is unit pair.

Also, $S \rightarrow T$ and $T \rightarrow F$ So, (S, F) is unit pair.

Now removing unit production we get:

$S \rightarrow S + T \mid T^* F \mid (S) \mid a$

$T \rightarrow T^* F \mid (S) \mid a$

$F \rightarrow (S) \mid a$

Example 19: Simply the grammar G = (V, T, P, S) defined by following productions.

$S \rightarrow ASB \mid \epsilon$

$A \rightarrow aSA \mid a$

$B \rightarrow SbS \mid A \mid bb \mid \epsilon$

Solution:

Removing ϵ-productions;

Here, $S \rightarrow \epsilon$ and $B \rightarrow \epsilon$ So S and B are nullable.

Now, removing ϵ-productions, it yields;

S→ ASB | AS | AB | A

A→ aSA | aA

B→ SbS | Sb | bS | b | A | bb

Removing unit productions;

Here, S→ A, B→ A. So, (S, A) and (B, A) are two unit pairs.

Hence, removing unit productions, we have;

S→ ASB | AS | AB | aSA | aA

A→ aSA | aA

B→ SbS | Sb | bS | b | bb | aAS | aA

This grammar has no useless symbols. Hence it is the final simplified grammar.

Eliminate Useless Symbols

Useless symbols are variables or terminals that do not appear in any derivation of a terminal string from the start symbol.

Formally, we say a symbol X is useful for a grammar G = (V, T, P, S) if there is some derivation of the form $S \overset{*}{\rightarrow} \alpha X \beta \overset{*}{\rightarrow} w$, where w is in T*. Here, X may be either variable or terminal and the sentential form αXβ might be the first or last in the derivation. If X is not useful, we say it is useless.

Eliminating a useless symbol includes identifying whether or not the symbol is "generating" and "reachable".

- We say X is generating if $X \overset{*}{\rightarrow} w$ for some terminal string w: Note that every terminal is generated since w can be that terminal itself, which is derived by zero steps.
- We say X is reachable if there is derivation $S \overset{*}{\rightarrow} \alpha X \beta$ for some α and β.

Thus if we eliminate the non generating symbols first and then non-reachable we shall have only the useful symbol left.

- For eliminating non generating symbol, identifying the non-generating symbols in CFG and eliminating those productions which contain non-generating symbols.
- For eliminating non-reachable symbols in CFG, Identify non-reachable symbols and eliminating those productions which contain non-reachable symbols.

Example 20: Consider a grammar defined by following productions:

S→ aB | bX

A→ Bad | bSX | a

B→ aSB | bBX

X→ SBd | aBX | ad

Eliminate the useless symbols.

Solution:

Here, A and X are generating symbols, since they produce strings composed of terminals. i.e

A→ a

X→ ad

Also, S→ bX and X generates terminal string so S can also generate terminal string. Hence, S is also generating symbol.

But B cannot produce any terminal string, so it is non-generating.

Hence, the new grammar after removing non-generating symbols is as;

S→bX

A→bSX | a

X→ad

Here, A is non-reachable as there is no any derivation of the form $S \overset{*}{\rightarrow} \alpha A \beta$ in the grammar. i.e. A cannot be reached from the starting variable S.

Thus eliminating the non-reachable symbols, the resulting grammar is:

S→ bX

X→ ad

This is the grammar with only useful symbols.

Example 21: Consider a grammar defined by following productions:

S → AB | CA

B→ BC | AB

A → a

C→ aB | b

Eliminate useless symbols from the above grammar.

Solution:

Here A and C are generating symbols, since they produce a, b respectively.

Also, Since S→ CA and C and A are generating symbols hence S is also generating symbol.

But B is non-generating symbol, since it cannot produce any terminal string.

Hence, the new grammar after removing non-generating symbols is as;

$S \rightarrow CA$

$A \rightarrow a$

$C \rightarrow b$

Here, all symbols reachable. Hence the grammar having only useful symbols is given as follows:

$S \rightarrow CA$

$A \rightarrow a$

$C \rightarrow b$

Chomsky Normal Form

A context free grammar G = (V, T, P, S) is said to be in Chomsky's Normal Form (CNF) if every production in G are in one of the two forms;

$A \rightarrow BC$ and

$A \rightarrow a$ where A, B, C ε V and a ε T

Thus a grammar in CNF is one which should not have;

- ϵ-production
- Unit production
- Useless symbols

Theorem: Every context free language (CFL) without ϵ-production can be generated by grammar in CNF.

Proof: Let G be the grammar that represents CFL without ϵ-production. Now we can simplify this grammar into its equivalent grammar that does not contain unit productions and useless symbols. Thus its equivalent grammar is of the form:

i. $A \rightarrow a$

ii. $A \rightarrow X_1X_2X_3.....X_k$

Here we do not have to change the production (i), since it is already in CNF form.

For production (ii) if k = 2 and it contains some terminal symbol in body part then just replace this terminal symbol by introducing new variable and add a new production having head part as new variable and body part as terminal symbol, which converts it into CNF form.

For production (ii) if k > 2 and it contains some terminal symbols in body part then first just replace these each terminal symbols by introducing new variables and add a new productions having head part as new variable and body part as terminal symbol. Let the production then changes into the form:

$A \rightarrow B_1B_2B_3.....B_k$

Now we break this production into group of productions with two variables in each body. For this we introduce k-2 new variables, $C_1, C_2, ..., C_{k-2}$. The original production is then replaced by the k-1 new productions as:

$A \rightarrow B_1C_1, C_1 \rightarrow B_2C_2, C_2 \rightarrow B_3C_3, ..., C_{k-2} \rightarrow B_{k-1}B_k$

Which is in the CNF form and represents (CFL) without ϵ-production

Example 22: Consider Grammar

$S \rightarrow AAC$

$A \rightarrow aAb \mid \epsilon$

$C \rightarrow aC \mid a$

Convert it into CNF form.

Solution:

1) Now, removing ϵ- productions;

Here, A is nullable symbol as $A \rightarrow \epsilon$

So, eliminating such ϵ-productions, we have;

$S \rightarrow AAC \mid AC \mid C$

$A \rightarrow aAb \mid ab$

$C \rightarrow aC \mid a$

2) Removing unit-productions;

Here, the unit pair we have is (S, C) as $S \rightarrow C$

So, removing unit-production, we have;

$S \rightarrow AAC \mid AC \mid C$

$A \rightarrow aAb \mid ab$

$C \rightarrow aC \mid a$

Here we do not have any useless symbol. Now, we can convert the grammar to CNF. For this;

- First replace the terminal by a variable and introduce new productions for those which are not as the productions in CNF.

i.e.

$S \rightarrow AAC \mid AC \mid C_1C \mid a$

$C1 \rightarrow a$

$A \rightarrow C_1AB_1 \mid C_1B_1$

$B_1 \rightarrow b$

$C \rightarrow C_1C \mid a$

Now, replace the sequence of non-terminals by a variable and introduce new productions.

Here, replace $S \rightarrow AAC$ by $S \rightarrow AC_2$, $C_2 \rightarrow AC$

Similarly, replace $A \rightarrow C_1AB_1$ by $A \rightarrow C_1C_3$, $C_3 \rightarrow AB_1$

Thus the final grammar in CNF form will be as;

$S \rightarrow AC_2 \mid AC \mid C_1C \mid a$

$A \rightarrow C_1C_3 \mid C_1b_1$

$C_1 \rightarrow a$

$B_1 \rightarrow b$

$C_2 \rightarrow AC$

$C_3 \rightarrow AB_1$

$C \rightarrow C_1C \mid a$

Example 23: Simplify following grammars and convert to CNF;

$S \rightarrow ASB \mid \epsilon$

$A \rightarrow aAS \mid a$

$B \rightarrow SbS \mid A \mid bb$

Solutions:

First remove ϵ-productions;

Here, $S \rightarrow \epsilon$ So, S is nullable.

So, the grammar becomes:

$S \rightarrow ASB \mid AB$

$A \rightarrow aAS \mid aA \mid a$

$B \rightarrow SbS \mid Sb \mid bS \mid A \mid bb$

Removing unit productions;

Here, $B \rightarrow A$

So, (B, A) is a unit pair.

Hence, removing this production yields;

$S \rightarrow ASB \mid AB$

$A \rightarrow aAS \mid aA \mid a$

$B \rightarrow Sb \mid bS \mid SbS \mid aAS \mid aA \mid a \mid b \mid bb$

Here, we have no useless symbols.

Now, to convert into CNF, replace each terminal by variables and introduce a new production as;

$S \rightarrow ASB \mid AB$

$A \rightarrow C_1AS \mid C_1A \mid a$

$C_1 \rightarrow a$

$B \rightarrow SB_1 \mid B_1S \mid SB_1S \mid C_1AS \mid C_1A \mid a \mid B_1B_1 \mid C_1b$

$B_1 \rightarrow b$

Also, replace $S \rightarrow ASB$ by $S \rightarrow AC_2$ with $C2 \rightarrow SB$

Replace $A \rightarrow C_1AS$ by $A \rightarrow C_1C3$ with $C_3 \rightarrow AS$

Replace $B \rightarrow SB_1S$ by $B \rightarrow SB_2$ with $B_2 \rightarrow B1S$

Also, $B \rightarrow C_1AS$ by $B \rightarrow C_1C_3$

So the grammar is;

$S \rightarrow AC_2 \mid AB$

$A \rightarrow C_1C_3 \mid C_1A \mid a$

$B \rightarrow SB_2 \mid SB_1 \mid B_1 \mid C_1C_3 \mid C_1A \mid B_1B_1 \mid b$

$C_1 \rightarrow a$, $C_2 \rightarrow SB$, $B_1 \rightarrow b$, $C_3 \rightarrow AS$, $B_2 \rightarrow B_1S$

Example 24: Simplify the grammar and convert to CNF:

$S \rightarrow aaaaS \mid aaaa$

Solution:

Here, we do not have any ϵ-production, useless symbol and unit productions.

So, converting it to the CNF, it consists;

- Replace a by any variable say A, with production

  $A \rightarrow a$

  So,

  $S \rightarrow AAAAS \mid AAAA$

  Now, replace $S \rightarrow AAAAS$ by $S \rightarrow AA_1$ with $A_1 \rightarrow AAAS$

  Again $A_1 \rightarrow AAAS$ by $A_1 \rightarrow AA_2$ with $A_2 \rightarrow AAS$

Again $A_2 \rightarrow AAS$ by $A_2 \rightarrow AA_3$ with $A_3 \rightarrow AS$

Similarly replace $S \rightarrow AAAA$ by $S \rightarrow AC$ with $C \rightarrow Aa$

and $C \rightarrow AAA$ by $C \rightarrow AC_1$ with $C_1 \rightarrow AA$

Thus, the grammar in CNF is;

$S \rightarrow AA_1 \mid AC$

$A_1 \rightarrow AA_2$

$A_2 \rightarrow AA_3$

$A_3 \rightarrow AS$

$C \rightarrow AC_1$

$C_1 \rightarrow AA$

$A \rightarrow a$

LEFT RECURSIVE GRAMMAR

In a context-free grammar **G**, if there is a production in the form $\mathbf{A \rightarrow A\alpha}$ where **A** is a non-terminal and 'α' is a string composed from $(V \cup T)^*$, it is called a **left recursive production**. The grammar having a left recursive production is called a **left recursive grammar**.

For example;

$S \rightarrow AA \mid o$

$A \rightarrow AAAS(\alpha_3) \mid os\ (\beta_1) \mid 1(\beta_2 \mid$

Removal of Left Recursion

Let $A \rightarrow A\alpha \mid \beta$

Here β does not start with A. Then, the left-recursive pair of production could be replaced by the non-recursive production as;

$A \rightarrow \beta A^1$

$A^1 \rightarrow \alpha A^1 \mid \in$; without changing the set of strings derivable from A.

No matter how many A-productions there are, we can eliminate immediate left recursion from then.

So in general, if $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots\ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$; with β_i does not start with A.

Then we can resolve left recursive as;

$A \rightarrow \beta_1 A^1 \mid \beta_2 a^1 \mid \ldots\ldots \mid \beta_N a^1$

$a^1 \rightarrow \alpha_1 a^1 \mid \alpha_2 a^1 \mid \ldots\ldots \mid \alpha_M a^1 \mid \in$

Equivalently, these productions can be rewritten as;

$A \rightarrow \beta_1 A^1 \mid \beta_2 A^1 \mid \ldots\ldots\ldots \mid \beta_n A^1 \mid \beta_1 \mid \beta_2 \ldots\ldots \mid \beta_m$

$A^1 \rightarrow \alpha_1 A^1 \mid \alpha_2 A^1 \mid \ldots\ldots \mid \alpha_n A^1 \mid \alpha_1 \mid \alpha_2 \mid \ldots\ldots\ldots \mid \alpha_m$

Example 25: Remove left recursive production from the following grammar:

$S \rightarrow AA \mid 0$

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

Solution:

Here, the production $A \rightarrow AAS$ is immediate left recursive. Now comparing $A \rightarrow AAS \mid 0S \mid 1$ with (1) above we get,

$\alpha_1 = AS$

$\beta_1 = 0S$

$\beta_2 = 1$

So removing left recursion we have,

$S \rightarrow AA \mid 0$

$A \rightarrow 0SA' \mid 1A' \mid 0S \mid 1$

$A' \rightarrow ASA' \mid AS$

Example 26: Remove left recursive production from the following grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid a$

Solution:

Here, $E \rightarrow E+T$ and $T \rightarrow T*F$ are left recursive so removing the left recursive

$E \rightarrow TA' \mid T$

$A' \rightarrow +TA' \mid +T$

$T \rightarrow FB' \mid F$

$B' \rightarrow *FB' \mid *F$

$F \rightarrow (E) \mid a$

Greibach Normal Form (GNF)

A grammar G = (V, T, P and S) is said to be in Greibach Normal Form, if all the productions of the grammar are of the form:

$A \rightarrow a\alpha$, where a is a terminal, i.e. $a \in T$ and α is a string of zero or more variables. i.e. $\alpha \in V^*$

we can rewrite as;

$A \rightarrow av'$, with $a \in t$

$A \rightarrow av^*$

$A \rightarrow a$; with $a \in t$.

This form, called Greibach Normal form, other sheila Greibach, who first gave a way to construct such grammars. converting, grammar to this form is complex, even if we simplify the task by, say, starting with a CNF grammar.

To convert a grammar into GNF;

- Convert the grammar into CNF at first.
- Remove any left recursions.
- Let the left recursion free ordering is $A_1, A_2, A_3, \ldots\ldots\ldots\ldots A_p$
- Let A_p is in GNF
- Substitute A_p in first symbol of A_{p-1}, if A_{p-1} contains A_p. Then A_{p-1} is also in GNF.
- Similarly substitute first symbol of A_{p-2} by A_{p-1} production and A_p production and so on.....

Example 27: Convert the following grammar into GNF

$S \rightarrow AA \mid 0$

$A \rightarrow SS \mid 1$

This Grammar is already in CNF

Now to remove left recursion, first replace symbol of A-production by S-production (since we do not have immediate left recursion) as:

$S \rightarrow AA \mid 0$

$A \rightarrow AAS \mid 0S \mid 1$ *(Where AS is of the form = α_1, 0S of the form =β_1 and 1of the form=β_2)*

Now, removing the immediate left recursion;

$S \rightarrow AA \mid 0$

$A \rightarrow 0SA' \mid 1A' \mid 0S \mid 1$

$A' \rightarrow ASA' \mid AS$

Now we replace first symbol of S-production by A-production as;

$S \rightarrow 0SA'A \mid 1A'A \mid 0SA \mid 1A \mid 0$

$A \rightarrow 0SA' \mid 1A' \mid 0S \mid 1$

$A' \rightarrow ASA' \mid AS$

Similarly replacing first symbol of A'-production by A-production, we get the grammar in GNF as;

$S \rightarrow 0SA'A \mid 1A'A \mid 0SA \mid 1A \mid 0$

$A \rightarrow 0SA' \mid 1A' \mid 0S \mid 1$

$A' \rightarrow 0SA'SA' \mid 1A'SA' \mid 0SSA' \mid 1SA' \mid 0SA'S \mid 1A'S \mid 0SS \mid 1$

Bakus- Naur form

This is another notation used to specify the CFG .It is named so, after Jhon Bakus, who invented it, and peter Naur, who refined it. The Bajus-Naur form is used to specify the syntactic rule of many computer languages, including Java. Here, concept is similar to CFG, only the difference instead of using symbol (→) in a production, we use symbol (::=). We enclose all non terminal in only brackets, < >.

Thus A BNF grammar is a set of derivation rules, written as <symbol> ::= __expression__

Where <symbol> is a nonterminal, and the __expression__ consists of one or more sequences of symbols; more sequences are separated by the vertical bar "|", indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are terminals. On the other hand, symbols that appear on a left side are non-terminals and are always enclosed between the pair <>.

The "::=" means that the symbol on the left must be replaced with the expression on the right.

For example,

<integer>::=<digit> | <integer><digit>

<digit>::= 0 | 1 | 2 | ..... | 9

Example: The BNF for identifiers is as;

<identifier>::= <letter or underscore><identifier><symbol>

<letter or underscore>::=<letter> | -

<symbol>::<letter or underscore> | <digit>

<letter>::= a | b | ..... | z

<digit>::= 0 | 1 | 2 | ..... | 9

Context Sensitive Grammar

A context sensitive grammar (CSG) is a formal grammar in which left hand sides and right hand sides of any production may be surrounded by a context of terminal and non-terminal symbols.

Formally, CSG can be defined as;

A formal grammar, G = (V, T, p, S) is context sensitive if all the production 'P' are of the form;

$\alpha A\beta \rightarrow \alpha\gamma\beta$, where $A \in V$, $\alpha \in V$, $\alpha\beta \in (v \cup t)^*$ and $\gamma \in (v \cup t)^+$.

The name context sensitive is explained by α and β that form the context of A and determine whether A can be replaced with γ or not. Thus the production $A \rightarrow \gamma$ can only be applied in contexts where α occurs to left of A and β occurs to right of A.

Example 28: $\{a^n b^n c^n \mid n \geq 1\}$ is a context sensitive language dfined as;

$S \rightarrow aSBC \mid aBC$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bc \rightarrow bc$

$cC \rightarrow cc$

Pumping Lemma for Context Free Languages

The "pumping lemma for context free languages" says that in any sufficiently long string in a CFL, it is possible to find at most two short, nearby substrings that we can "pump" in tandem (one behind another). i.e. we may repeat both of the strings I times, for any integer I, and the resulting string will still be in the language. We can use this lemma as a tool for showing that certain languages are not context free.

Theorem: Suppose L is a context-free language. Then there exist an integer n such that for every string $z \in L$ with $|z| \geq n$, zcan be written as$z = uvwxy$, for some strings u, v, w, x, and y satisfying

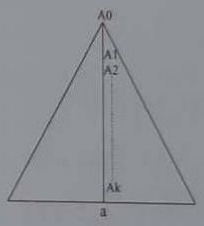
1. $|vx| > 0$
2. $|vwx| \leq n$
3. for every $i \geq 0$, $uv^iwx^iy \in L$

It means that the string can be divided into five parts so that the second and fourth parts may be repeated together any number of times and the resulting string still remains in the language.

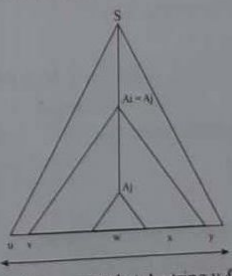
Proof: First we can find a CNF grammar for the grammar G of the CFL L, that generates L-{∈}. Choose $n = 2^m$, Next suppose z in L is of length at least n i.e. $|z| \geq b$.

any parse tree for Z must have high m +1, the if it would be less that will, i.e. say m then by the lemma for size of parse tree. $|z| = 2^{m-1} = \frac{2^m}{2} = \frac{n}{2}2$ is contracting so it should be m+.

Let us consider a path of maximum length is tree for z. as show below, where k is at least m & path is of length k.



Since, $K \geq m$, these are at least (k +1) or (M +1) occurrences of variables A_0 ..... A_k on the path. But there are only in variable in grammar, so at least two of the last m+1 variable on the path must be same, (be pigeonhole principle). Suppose $A_i = A_j$, then it is possible to divide tree

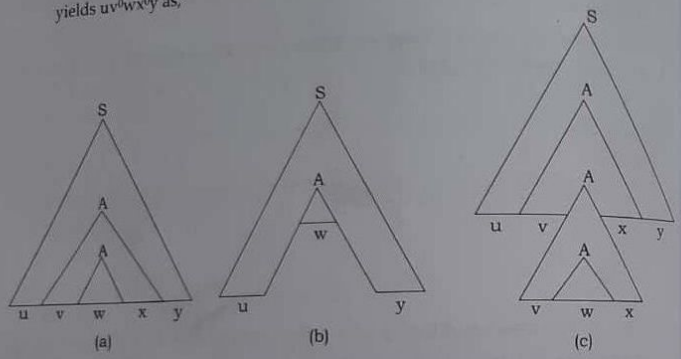


String w is the yield of sub tree rooted at A_j string v & x are to left & right of w in yield of longer sub tree rooted at A_i. If u & y represent beginning & end z. i.e. to right & left of sub tree rooted at A_i then z –uvwxy.

Since, there are not unit productions so v & x both could not be ∈ empty. Means that A_i has always two child corresponding to variables. If we let B denoted the one is not ancestor f A_j, then since w is derived from B does not overlap x . It

follows that either v or x is not empty so , Now, we know $A_i = A_j = A$ say as we found an y two variables in the tree are same.

Them, we can construct new parse tree where we can replace sub tree rooted at A_i , which has yield vwx, by sub tree rooted at A_j which has yield w. The reason we can do so is that both of these trees have root labelled A. The resulting tree yields uv^0wx^0y as;



Another option is to replace sub tree rooted at A_j by entire sub tree rooted at A_i. Then the yield can be of pattern uv^2wx^2y.

Hence, we can find any yield of the form uv^iwx^iy for all $i \geq 0$. Hence, we conclude, $uv^iwx^iy \in L$ for any $i \geq 0$.

Now, to show $|vwx| \leq n$. The A_i in the tree is the portion of o path of path which is root with height m+1. This sub tree rooted at A_1 also have height less than m+1. So by lemma for height of parse tree,

$|v\,w\,x| \leq 2^{m+1-1} = 2^m = n.$

$\therefore |v\,w\,x| \leq n.$

Thus this completes the proof of pumping lemma.

Example 29: Use the pumping lemma to show that the language $L = \{a^m b^m c^m\}$ is not context free.

Solution:

Suppose L is CFL.Then there exists a constant n satisfying the conditions of the pumping lemma. Let $z = a^n b^n c^n$. Clearly z is a member of L and of length atleast n. Using pumping lemma we can break z = uvwxy so that $|vx| > 0$ and $|vwx| \leq n$.

In this case either string vwx is composed from only one alphabet symbol or may be composed from both a's and b's or from both b's and c's but not from all three alphabet symbols.

Case I: If vwx is composed from only one alphabet symbol.

In this case the string uv^2wx^2y cannot contain equal number of a's, b's and c's. Therefore it cannot be the member of L. i.e. $uv^2wx^2y \notin L$.

Case II : If vwx is composed from two alphabet symbols.

i) Substring vwx does not contain any c.

- Consider the string uv^2wx^2y = uvvwxxy. Since $|vx| \geq 1$, this string contains more than n many a's or more than n many b's. Since it contains exactly n many c's, it follows that this string is not in the language L. Thisis a contradiction because, by the pumping lemma, the string uv^2wx^2y is inL.

ii) Substring vwx does not contain any a.

- Consider the string uv^2wx^2y = uvvwxxy. Since $|vx| \geq 1$, this string contains more than n many b's or more than n many c's. Since it contains exactly n many a's, it follows that this string is not in the language L. This is a contradiction because, by the pumping lemma, the string uv^2wx^2y is inL.

Thus, in all of the two cases, we have obtained a contradiction. Therefore, we have shown that the language A is not context-free.

Example 30: Prove that $L = \{ww : w \in \{a, b\}^*\}$ is not CFL.

Proof:

Suppose L is a context-free language. Then there exists a constant n satisfying the conditions of the pumping lemma. Let $z = a^n b^n a^n b^n$. Clearly z is a member of L and of length $4n \geq n$. Using pumping lemma we can break z = uvwxy so that $|vx| > 0$ and $|vwx| \leq n$.

Based on the location of vwx in the string z, we distinguish three cases:

Case I : The substring vwx overlaps both the leftmost half and the rightmost half of z.

Since $|vwx| \leq n$, the substring vwx is contained in the "middle" part of s, i.e., vxy is contained in the block $b^n a^n$. Consider the string uv^0wx^0y = uwy. Since $|vx| \geq 1$, we know that at least one of v and xis non-empty.

- If $v \neq \in$, then v contains at least one b from the leftmost block of b's in z, whereas x does not contain any b from the rightmost block of b's in x. Therefore, in the string uwy, the leftmost block of b's contains fewer b's than the rightmost block of b's. Hence, the string uwy is not contained in L.
- If $y \neq \in$, then y contains at least one a from the rightmost block of a's in z, whereas v does not contain any a from the leftmost block of a's in z. Therefore, in the string uwy, the leftmost block of a's contains more a's than the rightmost block of a's. Hence, the string uwy is not contained in L

Case II: The substring vwx is in the leftmost half of z. In this case, none of the strings uwy, uv^2wx^2y, uv^3wx^3y, uv^4wx^4y, etc., is contained in L. But, by the pumping lemma, each of these strings is contained in L.

Case III: The substring vwx is in the rightmost half of z. This case is symmetric to Case II: None of the strings uwy, uv^2wx^2y, uv^3wx^3y, uv^4wx^4y, etc., is contained in L. But, by the pumping lemma, each of these strings is contained in L.

To summarize, in each of the three cases, we have obtained a contradiction. Therefore, the language L is not context-free.

Closure Property of Context Free Languages

Given certain languages are context-free and a language L is formed from them by certain operation, like union of the two, then L us also context free. These theorem \ lemmas are often called closure properties of context free language, since they show whether or not the class if context free languages us closed under the operation mentioned. Closure properties express the idea that one (or several) languages are context free. Then certain related languages one also context free or not.

Here are some of the principal closure properties for context free languages;

(i) The context free language are closed under union. i.e. given any two context free languages L_1 & L_2, there union $L_1 \cup L_2$ is also context free language.

Proof:

The intuitive idea of the proof is to build a n grammar form the original two, and from start symbol of the new grammar have production to the start symbols of the original two framers.

let $G_1 = (v_1, (T), p_{11}, S_1)$ and $G_2 = (V_2(T), p_2, S_2)^b$ be two context tree grammar defining the language, L(G) & $L(G_2)$ .Without loss of generality, let us assume that they have common terminal set T, and disjoint set non-terminals i.e. $W \cap$ Because, the terminals are distinct so the production p_1 & p_2 .

Let S be a new non-terminal net in v_1 & v_2. Then, construct a new grammar G = $(v_1\ T_1\ P_1\ S)$ where,

$V = V_1\ UV_2\ U\ \{S\}$

$P = P_1\ U\ p_2\ U\ \{S \to S_1, S \to S_2\}$

G is clearly a context-free grammar because the two new productions so added are also of the correct form. We claim that $L(G) = L(G_1)\ U\ L(G_z)$. For this,

Supposed that $x \in L(G_1)$. Then these is a derivation of x is $S_1 \stackrel{*}{\Rightarrow} x_1$

But in G we have production,

$S \to S_1$

so there is a derivation of x above in G as;

$S \Rightarrow S_1 \stackrel{*}{\Rightarrow} x$

Thus, $x \in L(G)$. Therefore, $L(G_1) \leq L(G)$.

A similar argument shows $L(G_2) \leq L(G)$.

So, we have, $L(G_1) \cup L(G_2) \leq L(G)$.

Conversely, suppose that $x \in L(G)$. Then, there is a derivation x is G as;

$S \Rightarrow \beta \stackrel{*}{\Rightarrow} x$.

Because of the way in which p is constructed, β must either S_1 or S_2 suppose $\beta = S_1$. Any derivation in G of the form $S \stackrel{*}{\Rightarrow} x$ must involve only production of G_1 so, $S_1 \stackrel{*}{\Rightarrow} x$ is a derivation of c in G_1

$\therefore \beta = s_1 \Rightarrow x \in L(G_1)$. A similar argument prove that β = implies x $L(G_2)$. Thus, $L(G) \leq L(G_1) \cup LC$. It flows that $L(G) = L(G_1)\ U\ L(G_2)$.

Similarly following two close properties of CFL can be proved as for the union we have proved.

2) The CLF_s are closed under Kleen closure (star)

3) The CFL_s are closed under concatenation.

4) The CFL_s are closed under Kleen closure (star).

However, context free languages are not closed under some case live, intersection & complementation.

5) The context free language are not closed under intersection. TO prove this, the idea is to exhibit two CFL_s whose intersection result in the language which in not CFL. For this as we learned in pumping lemma that.

$L_1 = \{ a^n b^n c^n \mid n \geq 1, i \geq 1, i \geq 1 \}$

$L_2 = \{ a^i b^n c^n \mid n \geq 1, i \geq 1 \}$.

A grammar for L_1 is;

$S \rightarrow AC$

$A \rightarrow aAb \mid ab$

$C \rightarrow cC \mid c$

In this grammar, A generates all strings of the form $a^n b^n$ & C. Generates all strings of C's.

A grammar for L_2 is is;

$S \rightarrow AB$

$A \rightarrow aA \mid a$

$B \rightarrow bBc \mid bc$. In this grammar, A generates any strings and B generates strings of the form $b^n c^n$.

Clearly, $L = L_1 \cap L_2$. To see why, observe that L_2 required that three be the same number of a's & c's to be equal. A strings in both must have equal number of all three symbols & thus to be in L.

If the CFL's were closed under intersection, we could prove the false statement that L is context-free. Thus, we conclude by contradiction that the CLF's are not closed under intersection.

6) The CLF's are not closed under complementation, and then they would also be closed under the operation of intersection, contradiction above property. Let L_1 & L_2 be as defined in property then intersection of two languages can be expressed in there of complements as;

$L_1 \cap L_2 = (\overline{L_1} \cup \overline{L_2}) = \Sigma^*\text{-}L_1) \cup (\Sigma^* - L_2))$ but, we know CFL's are closed under union. So if CFL were close under complementation, then they would be also closed under intersection. Hence, it is not.

1. In Context Free Grammar, explore the meaning of the term "Context Free" with a suitable example.
2. Write a Context Free Grammar representing the "regular expressions".
3. Write a CFG representing strings that start and end with the same symbol over {0, 1}.
4. Write a CFG for a postfix expression over arithmetic operators.
5. Write Context Free Grammar for following
   a. $a^n b^n$
   b. $a^n b^{n+1}$
   c. ww^R
6. Write a Context Free Grammar that defines regular expression.
7. Write a CFG defining *if statement*.
8. What does ambiguity in grammar means? Does ambiguity exists in following grammar?

   S -> SS | aX | bY

   X -> aXX | bS | b

   Y -> bYY | aS | a
9. Write a CFG representing strings that start and end with the same symbol over 0,1
10. A CFG describing strings of letters with the word "main" somewhere in the string.

11. Identify possible useless symbols, epsilon productions and unit productions in following grammar and simplify it;

    S→ 1A | 0B | ϵ

    A→ 1AA | 0S | 0

    B→ 0BB | 1 | A

    D→DB | DD

12. When a grammar is said to be in CNF. Convert following grammar to CNF;

    S→ 1A | 0B | ϵ

    A→ 1AA | 0S | 0

    B→ 0BB | 1 | A

    C→CA | CS

13. Simplify following grammars and convert to the CNF:

    a. S→ bA | aB

       A→ bAA | aS | a

       B→ aBB | bS | a

    b. S→ abSb | a | aAb

       A→ bS | aAAb

    c. S→ 1A | 0B

       A→ 1AA | 0S | 0

       A→ 0BB | 1

14. What is left recursive grammar? How removal of left recursion from such grammar is possible.

15. What do you mean by right linear grammar? Give an example of such grammar.

16. Given following grammar, configure an equivalent finite automaton

    X→ bbY | aY | ϵ

    Y→ aYY | b | Z

    Z→ abZ | X | a

17. Write a grammar in Bakus Naur Form for the signed integers.

18. Verify that grammar defining language of palindromes over {a,b} is a CFG.

19. Define the closure properties of context free grammar.

❑❑❑

PUSHDOWN AUTOMATA

CHAPTER OUTLINES

After studying this Chapter you should be able to:

- Pushdown Automata
- Deterministic Push Down Automata (DPDA)
- Equivalence of CFG and PDA

PUSHDOWN AUTOMATA

The context free languages have a type if automation that defines them. This automaton is called "push down automaton" which can be thought as an ϵ-NFA with the addition of a stack. The presence of a stack means that, the push down automaton can "remember" an infinite amount of information. However, the pushdown automaton can only access the information on its stack in a last in first on way.

Thus, PDA is an abstract machine determined by following three things;

- Input tape
- finite state control
- A stack.

Each moves of the machine is determined by things;

- the current state
- read input symbol
- Symbol on the top of stack.

The moves consist of;

- changing state | staying on same state
- replacing the stack top by string of zero or more symbols.

Popping the top symbol off the stack means replacing it by ϵ, pushing Γ on the stack means replacing stack's top, say x, by y x s assuming the left end of string corresponds to stacks top.

The single move of the machine contain only stack operation either push or pop.

Replacing the stack symbol x lay the string α can be accomplished by a sequence of basic moves (a pop follower by a sequence of 0 or more pushes).

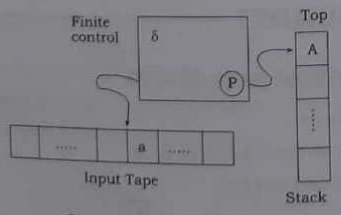


Figure: Pushdown Automaton

Formal Definition: Pushdown Automata (Non-deterministic)

A PDA is defined by seven tuples;

$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Where,

Q- Finite set of states.

Σ- Finite set of input symbols | alphabets.

Γ- Finite set of stack alphabet

q_0- Start state of PDA $q_0 \in Q$

Z_0- Initial stack symbol; $Z_0 \in \Gamma$

F- Set of final states; $F \subseteq Q$

δ- Transition function that maps;

$Q \times (\Sigma \cup \{\wedge\}) \times \Gamma \to Q \times \Gamma^*$

Thus, moves of PDA can be interpreted as;

$\delta(q, a, z) = \{(P_1, r_1) (p_2, r_2) \ldots\ldots\ldots\ldots (p_m, r_m)\}$ here $q, p_i \in Q$, $a \in \Sigma \cup \{\wedge\}$ & $z \in \Gamma$, $n \in \Gamma^*$.

Representation of PDA

PDA can be described by using two techniques:

- Transition Table
- Transition diagram

Transition Table

Consider a context free language defined by the grammar $a^n b^n$. Then the transition table used to represent this language is given as:

Move number	State	Input	Stack Symbol	Move(s)
1	q_0	A	Z_0	(q_0, aZ_0)
2	q_0	A	A	(q_0, aa)
3	q_0	ϵ	A	(q_1, a)
4	q_1	B	A	(q_1, ϵ)
5	q_1	ϵ	Z_0	(q_2, Z_0)

Here $q_0$is the starting state of PDA, q_2 is the final state of the PDA and Z_0 is the initial stack symbol.

Description

Move number 1: If the PDA is in state q_0 and read input a and top of the stack is Z_0 then it remains in the state q_0 and push the input symbol a into the top of the stack. It can be denoted as:

$$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$$

Move number 2: If the PDA is in state q_0 and read input a and top of the stack is a then it remains in the state q_0 and push the input symbol a into the top of the stack. It can be denoted as:

$$\delta(q_0, a, a) = \{(q_0, aa)\}$$

Move number 3: If the PDA is in state q_0 and read nothing (ϵ) and top of the stack is a then it switches to the state q_1 and no changed in the content of stack. It can be denoted as:

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

Move number 4: If the PDA is in state q_1 and read input b and top of the stack is a then it switches to the state q_1 and pop the symbol from the top of the stack. It can be denoted as:

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

Move number 5: If the PDA is in state q_1 and read nothing (ϵ) and top of the stack is Z_0 then it switches to the state q_2 and the stack elements remains unchanged. It can be denoted as:

$$\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}.$$

How the string aaabbb is processed by this PDA.

- Initially, the PDA is in state q_0 and stack symbol is Z_0, it then reads the first symbol a which leads it to the state q_0 and push the symbol 'a' into the stack. Hence the content of the stack is aZ_0.
- Now it reads the second symbol 'a' and top of stack is 'a' so it still remains in the state q_0 and push the element 'a' into the stack. Hence the content of the stack is aaZ_0.
- It then reads the third symbol 'a' and top of stack is 'a' so it still remains in the state q_0 and push the element 'a' into the stack. Hence the content of the stack is $aaaZ_0$.
- It then reads nothing 'ϵ' and top of stack is 'a', so it switches to the state q_1 and the content of the stack remains unchanged. Thus the content of the stack is $aaaZ_0$.

- It then reads the fourth symbol 'b' and top of stack is 'a', so it remains to the state q_1 and pop the top element from the stack. Thus the content of the stack is aaZ_0.
- It then reads the fifth symbol 'b' and top of stack is 'a', so it remains to the state q_1 and pop the top element from the stack. Thus the content of the stack is aZ_0.
- It then reads the sixth symbol 'b' and top of stack is 'a', so it remains to the state q_1 and pop the top element from the stack. Thus the content of the stack is Z_0.
- It then reads nothing (ϵ) and top of stack is Z_0, so it switches to the state q_2 and no operation is performed in stack. Thus the content of the stack is Z_0.

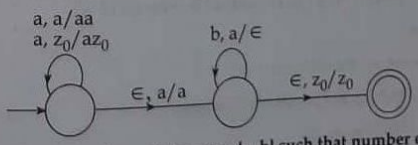
Since, after processing the whole string, PDA reaches to the state q_2 which is accepting state. Hence the string aaabbb is accepted by the PDA.

Transition Diagram

We can use transition diagram to represent a PDA, where

- Any state is represented by a node in diagram.
- Any arrowlabeled with "start" indicates the start state and doubly circled states are accepting / final states.
- The arc corresponds to transition of PDA as:
  - Arc labeled as a, X | α means;$\delta(q, a, x) = (p, \alpha)$ for arc from state p to q.

The PDA of $a^n b^n$ can represented using transition diagram as shown below:



Example 1: A PDA accepting a string over {a, b} such that number of a's and b's are equal. i.e. L={w | w ε {a, b}* and a's and b's are equal}.

Solution:

PDA to describe the above language is given by

$P = \{Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\}$ where,

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

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

$\Gamma = \{a, b, z_0\}$

$Z_0 = Z_0$

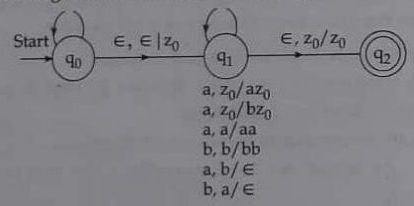
$q_0 = q_0$

$F = \{q_2\}$

And δ is defined as;

1. $\delta(q_1, a, z0) = (q_1, az_0)$
2. $\delta(q_1, b, z0) = (q_1, bz_0)$
3. $\delta(q_1, a, a) = (q_1, aa)$
4. $\delta(q_1, b, b) = (q_1, bb)$
5. $\delta(q_1, a, b) = (q_1, \epsilon)$
6. $\delta(q_1, b, a) = (q_1, \epsilon)$
7. $\delta(q_1, \epsilon, z0) = (q_2, \epsilon)$

So, the transition diagram for this PDA is shown in figure below:



Example 2: Construct a PDA accepting Language;

$L = \{ww^R \mid w \in (0+1)^*$ and w^R is reversal of $w\}$

Solution:

Let us construct the PDA for L as;

$P = (Q, \Sigma, \Gamma, \delta, q0, z0, F)$ where,

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

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

$\Gamma = \{a, b, z_0\}$

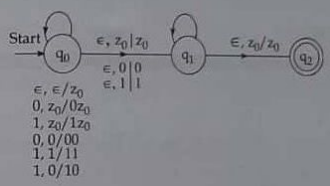
$Z_0 = Z_0$

$q_0 = q_0$

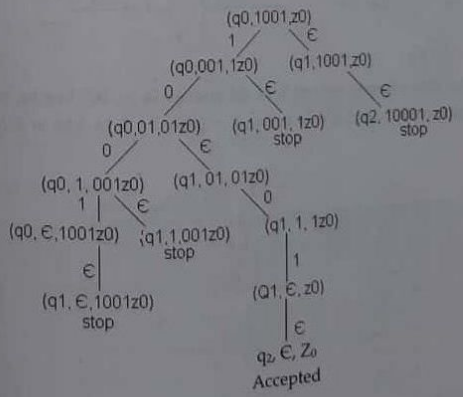
$F = \{q_2\}$

Now, δ is defined as;

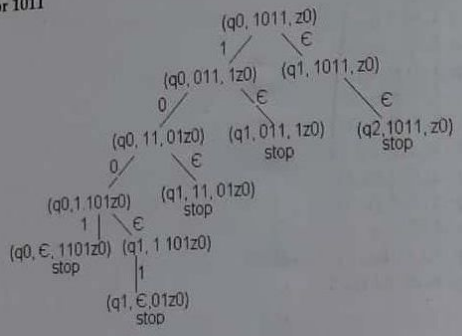
1. $\delta(q_0, 0, z_0) = (q_0, 0z_0)$
2. $\delta(q_0, 1, z_0) = (q_0, 1z0)$
3. $\delta(q_0, 0, 0) = (q_0, 00)$
4. $\delta(q_0, 1, 1) = (q_0, 11)$
5. $\delta(q_0, 0, 1) = (q_0, 01)$
6. $\delta(q_0, 1, 0) = (q_0, 10)$
7. $\delta(q_0, \epsilon, z_0) = (q_1, z_0)$
8. $\delta(q_0, \epsilon, 0) = (q_1, 0)$
9. $\delta(q_0, \epsilon, 1) = (q_1, 1)$
10. $\delta(q_1, 0, 0) = (q_1, \epsilon)$
11. $\delta(q_1, 1,1) = (q_1, \epsilon)$
12. $\delta(q_1, \epsilon, z_0) = (q_2, Z_0)$



Acceptance of any string can also be shown using parse tree. Consider any string 1001;

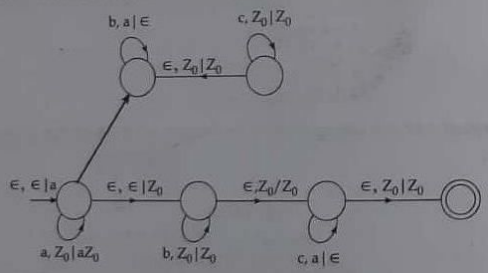


Now for 1011

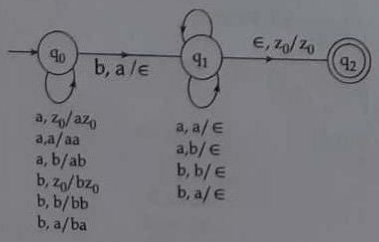


Hence, is not accepted.

Example 3: PDA that recognises the language $\{a^i b^i c^k \mid i,j,k \geq 0$ and i=j or i=k$\}$.



Example 4: PDA that accepts the set L of all strings in {a, b}* having an odd length and whose middle symbol is b, i.e., L = {vbw : v ∈ {a, b}*, w ∈ {a, b}*, |v| = |w|}.



Instantaneous Description of PDA

Any configuration of a PDA can be described by a triplet (q, w, γ).

Where,

q- is the state.

w- is the remaining input.

γ- is the stack contents.

Such a description by triple is called an instantaneous description of PDA (ID).

Instantaneous description is helpful for describing changes in the state, input, and stack of the PDA.

Let P = {Q, Σ, Γ, δ, q_0, z_0, F} be a PDA. Then, we define a relation ⊢, "yields" as;

$(q, aw, z\alpha) \vdash (p, w, \beta\alpha)$ if $\delta(q, a, z) = (p, \beta)$, and a may be ∈)

This move reflects the idea that, by consuming a (which may be ∈) from the input, and replacing z on the top of stack by β, we can go from state q to state p. Note that what remains on the input w and what is below the top of stack, β, do not influence the action the PDA.

For the PDA described earlier accepting language ww^R, [see example 2] the accepting sequence of ID's for string 1001 can be shown as;

$(q_0, 1001, Z_0) \vdash (q_0, 001, 1Z_0)$

$\vdash (q_0, 01, 01Z_0)$

$\vdash (q_1, 01, 01Z_0)$

$\vdash (q_1, 1, 1Z_0)$

$\vdash (q_1, \in, Z_0)$

$\vdash (q_2, \in, Z_0)$ Accepted

Therefore $(q_0, 1001, z_0) \vdash^* (q_1, 01, 01z_0) \vdash^* (q_2, \in, Z_0)$.

Language of a PDA

Language of a PDA refers to the set of strings that is accepted by the PDA. The acceptance of any string by a PDA can be defined in two ways;

1) **Acceptance by final state:**
   - According to this the Language of PDA is set of all string that causes PDA to enter in its accepting state after consuming all the alphabets of the string.

- Given a PDA P, the language accepted by final state, L(P) is;
  $\{w \mid (q, w, z_0) \vdash^* (q', \epsilon, \gamma)\}$ where $q' \in F$ and $\gamma \in \Gamma^*$.

2) Acceptance by empty stack:

- According to this the Language of PDA is set of all string that causes PDA to enter in its accepting state after consuming all the alphabets of the string.
- Given a PDA P, the language accepted by empty stack, L(P), is
  $\{w \mid (q, w, z_0) \vdash^* (q', \epsilon, \epsilon)\}$ where $P \in Q$.

Deterministic Push Down Automata (DPDA)

A PDA is said to be deterministic if for every input alphabet 'a' (a may be ϵ) and top of stack symbol,PDA has either unique transition(i.e moves to only one state.) or its transition is not defined.

Formally, A pushdown automata $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ is deterministic pushdown automata if the following two conditions are satisfied;

1) For any $q \in Q, X \in \Gamma$,
   If $\delta(q, a, X) \neq \phi$ for any $a \in \Sigma$ then $\delta(q, \epsilon, X) = \phi$
   i.e. if $\delta(q, a, X)$ is non-empty for some a, then $\delta(q, \epsilon, X)$ must be empty.
2) For any $q \in Q$, $a \in \Sigma \cup \{\epsilon\}$ and $X \in \Gamma$,
   $\delta(q, a, X)$ has at most one element.

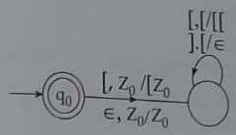
Example 5: Construct DPDA that accepts balanced parenthesis. i.e. equal no. of opening and closing braces. e.g: [[ ]], [ ][ ] etc.

Solution:

The Transition table for the above DPDA is given as:

Move number	State	Input	Stack Symbol	Move(s)
1	q_0	[	Z_0	$(q_1, [Z_0)$
2	q_1	[	[	$(q_1, [[)$
3	q_1	]	[	(q_1, ϵ)
4	q_1	ϵ	Z_0	(q_0, Z_0)

The transition diagram for this shown below:



Example 6: A DPDA accepting strings of balanced ordered parenthesis.

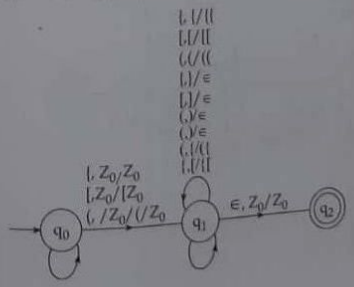
Solution:

A DPDA P can be constructed as:

$P = \{Q = (q_0, q_1, q_2), \Sigma = \{[,], (,)\}, \Gamma = \Sigma \cup \{z_0\}, \delta, q_0, z_0, \{q_2\}\}$

Where δ is defined as:

1. $\delta(q_0, [, z_0) = (q_1, [z_0)$
2. $\delta(q_0, [, z_0) = (q_1, [z_0)$
3. $\delta(q_0, (, z_0) = (q_1, (z_0)$
4. $\delta(q_1, [, [\,) = (q_1, [[\,)$
5. $\delta(q_1, [, [\,) = (q_1, [[\,)$
6. $\delta(q_1, (, (\,) = (q_1, ((\,)$
7. $\delta(q_1, [,]\,) = (q_1, \epsilon)$
8. $\delta(q_1,], [\,) = (q_1, \epsilon)$
9. $\delta(q_1,), (\,) = (q_1, \epsilon)$
10. $\delta(q_1, (, [\,) = (q_1, ([\,)$
11. $\delta(q_1, [, [\,) = (q_1, [[\,)$
12. $\delta(q_1, \epsilon, z_0) = (q_2, \epsilon)$



Evaluate [[ ]]

$(q_0, [[\,]], z_0) \vdash (q_1, [\,]], [z_0)$
$\vdash (q_1,]], [[z_0)$
$\vdash (q_1,], [z_0)$
$\vdash (q1, \epsilon, z_0)$
$\vdash (q_2, \epsilon, \epsilon)$ Accepted

Evaluate [[ ]]

$(q_0, [[\,]], z_0) \vdash (q_1, [\,]], [z_0)$
$\vdash (q_1,]], [[z_0)$

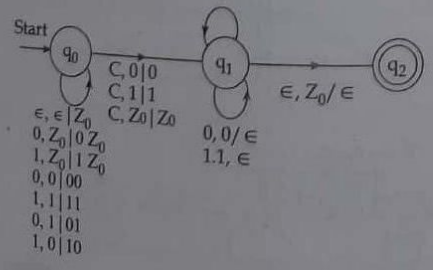
Stop (as $(q_1,], \{\,)$ is not defined) so not accepted.

Example 7: A PDA accepting language L = {w c w^R | w ε (0+1)*} is constructed as;

P = ({q_0, q_1, q_2}, {0, 1}, {0, 1, z_0}, δ, q_0, z_0, {q_2}) where δ is defined as;

1. $\delta(q_0, \epsilon, \epsilon) = (q_0, z_0)$
2. $\delta(q_0, 0, z_0) = (q_0, 0z_0)$
3. $\delta(q_0, 1, z_0) = (q_0, 1z_0)$
4. $\delta(q_0, 0, 0) = (q_0, 00)$
5. $\delta(q_0, 1, 1) = (q_0, 11)$
6. $\delta(q_0, 0, 1) = (q_0, 01)$
7. $\delta(q_0, 1, 0) = (q_0, 10)$
8. $\delta(q_0, C, 0) = (q_1, 0)$
9. $\delta(q_0, C, 1) = (q_1, 1)$
10. $\delta(q_0, C, z_0) = (q_1, z0)$
11. $\delta(q_1, 0, 0) = (q_1, \epsilon)$
12. $\delta(q_1, 1, 1) = (q_1, \epsilon)$
13. $\delta(q_1, \epsilon, z_0) = (q_2, \epsilon)$

The general notation is as:

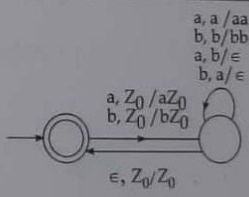


Example 8: Construct a DPDA that accepts all the string having equal no. of a's and b's.

Solution:

The transition table for above DPDA can be given as below

Move number	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	(q_1, aZ_0)
2	q_0	b	Z_0	(q_1, bZ_0)
3	q_1	a	a	(q_1, aa)
4	q_1	b	b	(q_1, bb)
5	q_1	a	b	(q_1, ϵ)
6	q_1	b	a	(q_1, ϵ)
7	q_1	∈	Z_0	(q_0, Z_0)



Finding the sequence of ID's for the string abbaaabb.

$(q_0, abbaaabb, Z_0) \vdash (q_1, bbaaabb, aZ_0)$
$\vdash (q_1, baaabb, Z_0)$
$\vdash (q_0, baaabb, Z_0)$
$\vdash (q_1, aaabb, b\,Z_0)$
$\vdash (q_1, aabb, Z_0)$
$\vdash (q_0, aabb, Z_0)$
$\vdash (q_1, abb, a\,Z_0)$
$\vdash (q_1, bb, aa\,Z_0)$
$\vdash (q_1, b, a\,Z_0)$
$\vdash (q_1, \epsilon, Z_0)$
$\vdash (q_0, \epsilon, Z_0)$ Hence accepted.

Non Deterministic PDA (NPDA)

A DFA (or NFA) is not powerful enough to recognize many context-free languages because a DFA can't count. But counting is not enough -- consider a language of palindromes, containing strings of the form WWR. Such a language requires more than an ability to count; it requires a stack.

A nondeterministic pushdown automaton (npda) is basically an nfa with a stack added to it.

We start with the formal definition of an NFA, which is a 5-tuple, and add two things to it:

Γ is a finite set of symbols called the *stack alphabet*, and

$z \in \Gamma$ is the stack start symbol

We also need to modify δ, the transition function, so that it manipulates the stack

A nondeterministic pushdown automaton or npda is a 7-tuple

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

Q is a finite set of states,

Σ is a the input alphabet,

Γ is the stack alphabet,

δ is a transition function,

$q_0 \in Q$ is the initial state,

z

$\in \Gamma$ is the stack start symbol, and

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

Transition Functions for NPDAs

The transition function for an npda has the form

$D: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow$ finite subsets of $Q \times \Gamma^*$

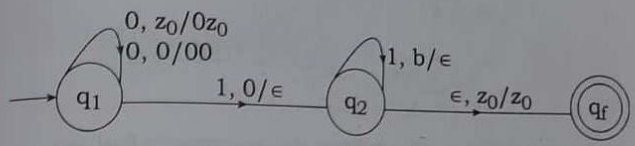
Construction of PDA by Final State

Let $P = (Q, \Sigma, \Gamma, \delta, q0, Z0, F)$

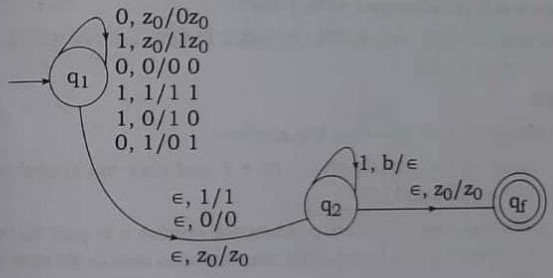
be a PDA. The language accepted by P by final state is: $L(P) = \{w \mid (q0, w, Z0) \vdash^* (q, \in, \alpha), q \in F\}$ for some state q in F and any input stack string α. Starting in

the initial ID with w waiting on the input, P consumes w from the input and enters an accepting state. The contents of the stack at that time is irrelevant.

Example 1: PDA with final state with equal number of zeros (0's) followed by equal numbers ones (1's).



Example 2: PDA with final state that accept the palindrome

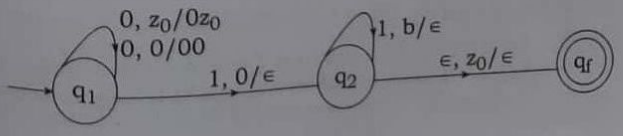


Construction of PDA by Empty Stack

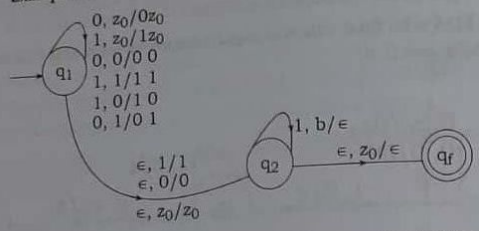
PDA- Acceptance by empty stack:

Let $P = (Q, \Sigma, \Gamma, \delta, q0, Z0, F)$ be a PDA. The language accepted by P by empty stack is: $N(P) = \{w \mid (q0, w, Z0) \vdash^* (q, \in, \in)\}$ where qis any state N(P) is the set of inputs w that P can consume an at the same time empty the stack.

Example 3: PDA with empty stack with equal number of zeros (0's) followed by equal numbers ones (1's).



Example 4: PDA with empty stack that accept the palindrome.



Conversion of PDA accepting by Empty stack to accepting by final state

Let us consider the PDA P_N that accepts a language L by empty stack and P_F is its equivalent PDA that accepts L by final state.

Theorem: If $L = L(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

Proof:

The construction of P_F from P_N is done as follows:

1. Introduce the a new symbol X_0 ($X_0 \notin \Gamma$)and place this symbol into the bottom of the stack of P_F.
2. Introduce a new start state, p_0, whose sole function is to push Z_0, the start symbol of P_N, onto the top of the stack and enter state q_0 (the start state of P_N).
3. Copy the other states and transitions functions of P_N in P_F.
4. Finally add another new state p_f, which is the accepting state of P_F ; the PDA transfers to state p_f whenever it discovers that P_N would have emptied its stack.

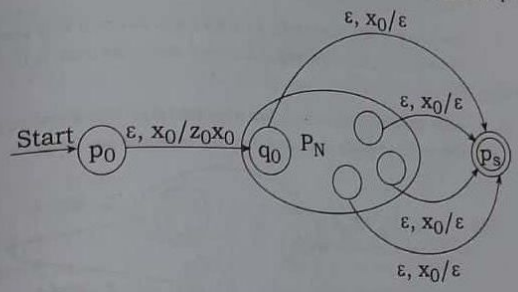
Thus the new constructed P_F has the components:

$$P_N = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

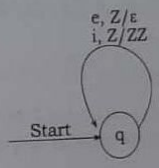
Where δ_F is defined by:

1. $\delta_F(p_0, \epsilon, X_0) = (q_0, Z_0X_0)$
2. For all states q in Q, inputs a in $\sum$ or a= ϵ, and stack symbols Y in Γ, $\delta_F(q, a, Y)$ contains all the pairs in $\delta_N(q, a, Y)$. i.e. Copy all the transition functions of P_N.

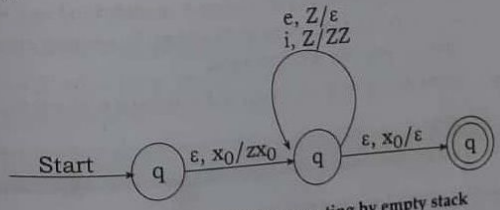
3. Add another transition function $\delta_F(q, \epsilon, X_0) = (p_f, \epsilon)$ for every state q in Q.



Example 12: Convert the following PDA that accepts the string using empty stack into its equivalent PDA that accepts the string by entering into the accepting state.



Solution:



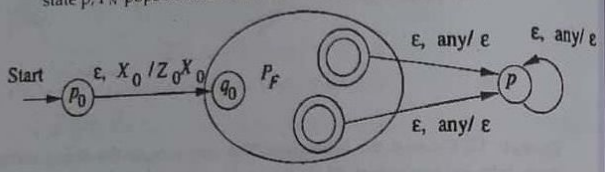
Conversion of PDA accepting by final state to accepting by empty stack

Theorem: Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

Proof:

The construction of P_N from P_F is done as follows:

1. Introduce the a new symbol X_0 ($X_0 \notin \Gamma$) and place this symbol into th bottom of the stack of P_N.
2. Introduce a new start state, p_0, whose sole function is to push Z_0, the sta symbol of P_F, onto the top of the stack and enter state q_0 (the start state P$_F$).
3. each accepting state of P_F, add a transition on ϵ to a new state p. When state p, P_N pops its stack and does not consume any input.



Thus the new constructed P_N has the components:

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

Where δ_N is defined by:

1. $\delta_N (p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$.
2. For all states q in Q, inputs a in $\sum$ or a= ϵ, and stack symbols Y in Γ, δ_N(a, Y) contains all the pairs in δ_F (q, a, Y). i.e. Copy all the transitio functions of P_N.
3. For all accepting states q in F and stack symbols Y in Γ or Y = X_0, δ_N(q, ϵ, = (p,ϵ). By this rule, whenever P_F accepts, P_N can start emptying its stac without consuming any more input.
4. For all stack symbols Y in Y in Γ or Y = X_0, δ_N(q, ϵ, Y) = (p,ϵ).

EQUIVALENCE OF CFG AND PDA

Converting CFG into its Equivalent PDA

Given a CFG, G = (V, T, P,S), we can construct a push down automaton, P whic accepts the language generated by the grammar G, i.e. L(M) = L (G).

The PDA M can be defined as;

M = ({q}, T, V U T, δ, q, S, Φ)

Where,

Q = {q} is only the state in the PDA.

Σ = T

Γ = V U T (i.e. PDA uses terminals and variables of G as stack symbols)

z_0 = S (initial stack symbol is start symbol in grammar)

F = Φ

δ can be defined as;

i. $\delta(q, \epsilon, A) = \{q, \alpha) / A \rightarrow \alpha$ is a production P of G.
ii. $\delta(q, a, a) = \{ (q, \epsilon)$, for all a ε T}

Alternatively, we can define push down automata for the CFG with two states p & q, where p being start state. Here idea is that the stack symbol initially is supposed to be ϵ, and at first PDA starts with state p and reading ϵ, it inserts start symbol 'S' of CFG into stack and changes the state to q.

Then all transitions occur in state q.

i.e. the PDA can be defined as;

P= ({p,q}, T, V U T, δ, p, {q}); stack top is ϵ.

Then δ can be defined as;

$\delta(p, \epsilon, \epsilon) = \{(q, S) / S$ is start symbol of grammar G.

$\delta(p, \epsilon, A) = \{(q, \alpha) / A \rightarrow \alpha$ is a production P of G.

$\delta(p, a, a) = \{(q, \epsilon)$, for all a ε Σ.

Example 13: Consider an example;

Let G = (V, T, P and S) where, productions in G denoted by P are:

$E \rightarrow T \mid E + T$

$T \rightarrow F \mid T^*F$

$F \rightarrow a \mid (E)$

We can define a PDA equivalent to this grammar as;

M = ({q_0}, {a, *, +, (,)},{a, *, +, (,), E, T, F}, δ, q_0, E, Φ)

Where δ can be defined as;

$\delta(q_0, \epsilon, E) = \{(q_0, T), (q_0, E + T)\}$

$\delta(q_0, \epsilon, T) = \{(q_0, F), (q_0, T^*F)\}$

$\delta(q_0, \epsilon, F) = \{(q_0, a), (q_0, (E))\}$

$\delta(q_0, a, a) = \{(q_0, \in)\}$

$\delta(q_0, *, *) = \{(q_0, \in)\}$

$\delta(q_0, +, +) = \{(q_0, \in)\}$

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

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

Now with this PDA, M, let us trace out acceptance of a + (a*a);

$(q_0, a + (a*a), E) \vdash (q_0, a + (a*a), E + T)$

$\vdash (q_0, a + (a*a), T + T)$

$\vdash (q_0, a + (a*a), F + T)$

$\vdash (q_0, a + (a*a), a + T)$

$\vdash (q_0, + (a*a), + T)$

$\vdash (q_0, (a*a), T)$

$\vdash (q_0, (a*a), F)$

$\vdash (q_0, (a*a), (E))$

$\vdash (q_0, a*a), E))$

$\vdash (q_0, a*a), T))$

$\vdash (q_0, a*a), T*F)$

$\vdash (q_0, a*a), F*F)$

$\vdash (q_0, a*a), a*F)$

$\vdash (q_0, *a), *F)$

$\vdash (q_0, a), F))$

$\vdash (q_0, a), a))$

$\vdash (q_0),))$

$\vdash (q_0, \in, \in)$ Accepted (acceptance by empty stack).

In CFG

E $\rightarrow$ E + T

$\rightarrow$ E + T

$\rightarrow$ T + T

$\rightarrow$ F + T

$\rightarrow$ a + T

$\rightarrow$ a + F

$\rightarrow$ a + (E)

$\rightarrow$ a + (T*F)

$\rightarrow$ a + (F*F)

$\rightarrow$ a + (a*F)

$\rightarrow$ a + (a*a)

Example 14: Convert the grammar defined by following production into PDA;

S$\rightarrow$ 0S1 | A

A$\rightarrow$ 1S0 | S | $\in$

Solution:

Let G = (V, T, P and S) defined by following productions;

S$\rightarrow$ 0S1 | A

A$\rightarrow$ 1S0 | S | $\in$

PDA equivalent top this grammar as

$M = (\{q_0\}, \{0, 1\}, \{0, 1, S, A\}, \delta, q_0, S, \Phi)$

Where; $Q = \{q_0\}$

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

$\Gamma = \{0, 1, S, A\}$

$z_0 = S$

$F = \Phi$

And δ is defined as;

$\delta(q_0, \in, S) = \{(q_0, 0S1), (q_0, A)\}$

$\delta(q_0, \in, A) = \{(q_0, 0S1), (q_0, S), (q_0, \in)\}$

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

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

Example 15: Construct a PDA equivalent to following grammar defined by;

S$\rightarrow$aAA

A$\rightarrow$aS | bS | a

Solution:

Let G = (V, T, P and S) be the grammar defined by the production;

S$\rightarrow$aAA

A$\rightarrow$aS | bS | a

Now, we can define the PDA equivalent to the grammar as;

$M = (\{q_0\}, \{a, b\}, \{a, b, S, A\}, \delta, q_0, S, \Phi)$

Where δ is defined as

$\delta(q_0, \epsilon, S) = \{q_0, aAA\}$

$\delta(q_0, \epsilon, A) = \{(q_0, aS), (q_0, bS), (q_0, a)\}$

$\delta(q_0, a, a) = \{q_0, \epsilon\}$

$\delta(q_0, b, b) = \{q_0, \epsilon\}$

Now, we trace acceptance of aaabaaaaa

$(q_0, aaabaaaaa, S) \vdash (q_0, aaabaaaaa, aA)$

$\vdash (q_0, aabaaaaa, A)$

$\vdash (q_0, aabaaaaa, AA)$

$\vdash (q_0, aabaaaaa, aSA)$

$\vdash (q_0, abaaaaa, SA)$

$\vdash (q_0, abaaaaa, aAAA)$

$\vdash (q_0, baaaaa, AAA)$

$\vdash (q_0, baaaaa, bSAA)$

$\vdash (q_0, aaaaa, SAA)$

$\vdash (q_0, aaaaa, aAAAA)$

$\vdash (q_0, aaaa, AAAA)$

$\vdash (q_0, aaaaa, aAAA)$

$\vdash (q_0, aaa, AAA)$

$\vdash (q_0, aaa, aAA)$

$\vdash (q_0, aa, AA)$

$\vdash (q_0, aa, aA)$

$\vdash (q_0, a, A)$

$\vdash (q_0, a)$

In CFG

$S \rightarrow aAA \rightarrow aaSA$

$\rightarrow aaaAAA$

$\rightarrow aaabSAA$

$\rightarrow aaabaAAAA$

$\rightarrow aaabaaAAA$

$\rightarrow aaabaaaAA$

$\rightarrow aaabaaaaA$

$\rightarrow aaabaaaaa$

Converting PDA into its equivalent CFG

Given a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$; $F = \Phi$, we can obtain an equivalent CFG, $G = (V, T, P \text{ and } S)$ which generates the same language as accepted by the PDA M.

The set of variables in the grammar consists of;

- The special symbol S, which is start symbol.
- All the symbols of the form $[p \times q]$; $p,q \in Q$ and $X \in \Gamma$

i.e. $V = \{S\} \cup \{[p\ X\ q]\}$

The terminal in the grammar $T = \Sigma$

The production of G is as follows;

- For all states $q \in Q$, $S \rightarrow [q_0, z_0, q]$ is a production of G.
- For any states $q, r \in Q$, $X \in \Gamma$ and $a \in \Sigma \cup \{\epsilon\}$

If $\delta(q, a, X) = (p, \epsilon)$ then $[p\ X\ q] \rightarrow a$

- For any states $q, r \in Q$, $X \in \Gamma$ and $a \in \Sigma \cup \{\epsilon\}$,

If $\delta(q, a, x) = (r, Y_1Y_2 \ldots\ldots Y_k)$; where $Y_1, Y_2, \ldots\ldots Y_k \in \Gamma$ and $k \geq 0$

Then for all lists of states $r_1, r_2, \ldots\ldots\ldots r_k$, G has the production $[p\ X\ q] \rightarrow a [rY_1r_1][r_1Y_2r_2]\ldots\ldots\ldots\ldots[r_{k-1}Y_kr_k]$

This production says that one way to pop X and go from stack q to state r_k is to read "a" (which may be ϵ), then use some input to pop Y_1 off the stack while going from state r to r_1, then read some more input that pops Y_2 off the stack and goes from r_1 to r_2 and so on.....

Note: Each of the variables [q X r] represents an event in PDA.

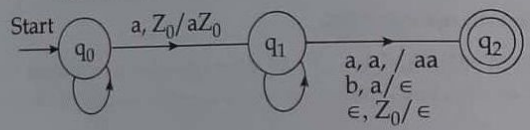
- The next popping of some symbol x from the stack,

- A change in state from some r at beginning to a q when x has finally been replaced by ϵ on the stack.

Example 16: PDA that recognizes a language

$L = \{a^n b^n \mid n > 0\}$ be defined as;

1. $\delta(q_0, a, z_0) = (q_1, az_0)$
2. $\delta(q_1, a, a) = (q_1, aa)$
3. $\delta(q_1, b, a) = (q_1, \epsilon)$
4. $\delta(q_1, \epsilon, z_0) = (q_1, \epsilon)$



Let G = (V, T, P and S) be the equivalent CFG for the given PDA where,

$V = \{S\} \cup \{[p \times q] / p, q \in Q, x \in \Gamma\}$

S= is the start state.

$T = \Sigma$

And P is defined by following production;

1. $S \rightarrow [q_0 z_0 q_0] \mid [q_0 z_0 q_1]$
   i.e. $S \rightarrow [q_0 z_0 r_2]$; for r_2 in $\{q_0, q_1\}$
2. From the fact that $\delta(q_0, a, z_0)$ contains (q_1, az_0), we get production
   $[q_0 z_0 r_2] \rightarrow a[q_1 a r_1][r_1 z_0 r_2]$; for r_i in $\{q_0, q_1\}$
3. From the fact that $\delta(q_1, a, a)$ contains (q_1, aa), we get productions
   $[q_1 a r_2] \rightarrow a[q_1 a r_1][r_1 a r_2]$ for r_i in $\{q_0, q_1\}$
4. From the fact that $\delta(q_1, b, a)$ contains (q_1, ϵ), we get
   $[q_1 a q_1] \rightarrow b$
5. From the fact that $\delta(q_1, \epsilon, z_0)$ contains (q_1, ϵ), we get
   $[q_1 z_0 q_1] \rightarrow \epsilon$

Now acceptance of aaabbb can be shown as;

$S \rightarrow [q_0 z_0 r_2]$

$\rightarrow a [q_1 a r_1] [r_1 z_0 r_2]$

$\rightarrow aa [q_1 a r_1] [r_1 a r_2] [r_1 z_0 r_2]$

$\rightarrow aaa[q_1 a r_1] [r_1 a r_2] [r_1 a r_2] [r_1 z_0 r_2]$

$\rightarrow aaab[r_1 a r_2] [r_1 a r_2] [r_1 z_0 r_2]$

$\rightarrow aaabb[r_1 a r_2] [r_1 z_0 r_2]$

$\rightarrow aaabbb[r_1 z_0 r_2]$

$\rightarrow aaabbb \epsilon = aaabbb$

Example 17: Convert the PDA $P = (\{p, q\}), \{0, 1\}, \{x, z_0\}, \delta, q, z_0)$ to a CFG if δ is given by

- **$\delta(q, 1, z_0) = (q, xz_0)$**
- **$\delta(q, 1, x) = (q, xx)$**
- **$\delta(q, 0, x) = (q, x)$**
- **$\delta(q, \epsilon, z_0) = (q, \epsilon)$**
- **$\delta(q, 1, x) = (p, \epsilon)$**
- **$\delta(q, 0, z_0) = (q, z_0)$**

Now, the equivalent grammar can be written as;

$G = (V, \Sigma, P, S)$ where P consists of productions as;

$S \rightarrow [qz_1r_2]$; r_2 in $\{p, q\}$

$[qz_0r_2] \rightarrow 1[q \times r_1] [r_1 z_0 r_2]$; for r_i in $\{p,q\}$

$[q \times r_2] \rightarrow 1[q \times r_1] [r_1 \times r_2]$; for r_i in $\{p,q\}$

$[q \times r_2] \rightarrow 0[p \times r_2]$; for r_2 in $\{p,q\}$

$[q \times q] \rightarrow \epsilon$

$[p \times p] \rightarrow 1$

$[pz_0r_2] \rightarrow 0[qz_0r_2]$; for r_2 in $\{p,q\}$

Example 18: Convert the following PDA into CFG.

$P = (\{q\}, \{i, e\}, \{X,Z\}, \delta, q, Z)$, where δ is given by:

$\delta(q, i, Z) = \{(q, XZ)\}$, $\delta(q, e, X) = \{(q, \epsilon)\}$ and $\delta(q, \epsilon, Z) = \{(q, \epsilon)\}$

Solution:

Equivalent productions are:

$S \rightarrow [qZq]$

$[qZq] \rightarrow i[qXq][qZq]$

$[qXq] \rightarrow e$

$[qZq] \rightarrow \varepsilon$

If [qZq] is renamed to A and [qXq] is renamed to B, then the CFG can be defined by:

$G = (\{S, A, B\}, \{i, e\}, \{S \rightarrow A, A \rightarrow iBA \mid \varepsilon, B \rightarrow e\}, S)$

Example 19: Convert PDA to CFG. PDA is given by P = ({p,q}, {0,1}, {X, Z}, δ, q, Z)), Transition function δ is defined by:

$\delta(q, 1, Z) = \{(q, XZ)\}$

$\delta(q, 1, X) = \{(q, XX)\}$

$\delta(q, \varepsilon, X) = \{(q, \varepsilon)\}$

$\delta(q, 0, X) = \{(p, X)\}$

$\delta(p, 1, X) = \{(p, \varepsilon)\}$

$\delta(p, 0, Z) = \{(q, Z)\}$

Solution:

Add productions for start variable

$S \rightarrow [qZq] \mid [qZp]$

For $\delta(q, 1, Z) = \{(q, XZ)\}$

$[qZq] \rightarrow 1[qXq][qZq]$

$[qZq] \rightarrow 1[qXp][pZq]$

$[qZp] \rightarrow 1[qXq][qZp]$

$[qZp] \rightarrow 1[qXp][pZp]$

For $\delta(q, 1, X) = \{(q, XX)\}$

$[qXq] \rightarrow 1[qXq][qXq]$

$[qXq] \rightarrow 1[qXp][pXq]$

$[qXp] \rightarrow 1[qXq][qXp]$

$[qXp] \rightarrow 1[qXp][pXp]$

For $\delta(q, \varepsilon, X) = \{(q, \varepsilon)\}$

$[qXq] \rightarrow \varepsilon$

For $\delta(q, 0, X) = \{(p, X)\}$

$[qXq] \rightarrow 0[pXq]$

$[qXp] \rightarrow 0[pXp]$

For $\delta(p, 1, X) = \{(p, \varepsilon)\}$

$[pXp] \rightarrow 1$

For $\delta(p, 0, Z) = \{(q, Z)\}$

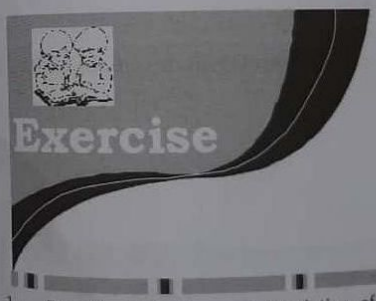
$[pZq] \rightarrow 0[qZq]$

$[pZp] \rightarrow 0[qZp]$

Renaming the variables [qZq] to A, [qZp] to B, [pZq] to C, [pZp] to D, [qXq] to E [qXp] to

F, [pXp] to G and [pXq] to H, the equivalent CFG can be defined by:

G = ({S, A, B, C, D, E, F, G, H}, {0,1}, R, S). The productions of R also are to be renamed accordingly.



1. Describe instantaneous description of PDA. Configure a PDA for balanced parentheses, i.e. $\sum = \{ [, (,),] \}$.
2. Configure a Pushdown automaton accepting the language, $L = \{wCw^R \mid w \in (a,b)^*\}$. Show instantaneous description of strings abbCbba and baCba.
3. Configure a DPDA accepting the language $L = \{ (^n C)^n \}$. Show the instantaneous description of ((C)) and (((C)).
4. Construct a PDA $L = \{w \in \{0, 1\}^*$ starts and end with same symbol $\}$.
5. Construct a PDA that accepts string of the form $\{w \in \{0, 1\}^*$ contains more 1's than 0's$\}$.
6. Construct a PDA with recognizes the language of arithmetical expression with the following $\sum = \{0, 1, +, *, (,)\}$.

7. Configure a Push Down Automaton, with 7-tuples, for the language defined by following grammar. Also draw the state diagram.

]

$S \rightarrow aSbS \mid bSaS \mid \epsilon$

8. Consider a PDA that accepts by empty stack, P= ({p, q}, {(,)}, {Z}, δ, p, z); with δ defined as $\delta(p, (, z) = (p, (z)$, $\delta(p, (, ( ) = (p, (( )$, $\delta(p,), ( ) = (p, \epsilon)$, $\delta(p, \epsilon, z) = (q, \epsilon)$. Now construct an equivalent CFG. Also show the derivation of (( )) using the productions of so constructed CFG

9. Construct a push down automaton for the following grammar. Also draw the state diagram.

$S \rightarrow 1A \mid 0B \mid \epsilon$

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

$B \rightarrow 0BB \mid 1 \mid A$

10. Construct a PDA from the following CFG

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

Where the production are

$S \rightarrow XS \mid \epsilon$

$A \rightarrow aX \mid Ab \mid ab$

11. Obtain the PDA corresponding to the grammar G = ({a, b, c, d}, {s, A, B}, S, P) with the following production rule:

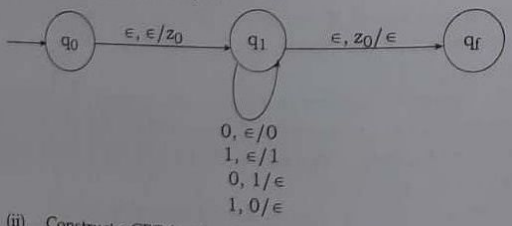
$S \rightarrow aSB \mid bA \mid b \mid d$

$A \rightarrow bA \mid b$

$B \rightarrow C$

12. Conversion PDA to CFG

(i) Construct CFG for the given PDA



(ii) Construct a CFG for the following language:

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

(b) $L = \{a^n b^{2n} \mid n \geq 0\}$

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

(d) $L = \{a^{2n+1} b^n \mid n > 0\}$

□□□

TURING MACHINE

CHAPTER OUTLINES

After studying this Chapter you should be able to:

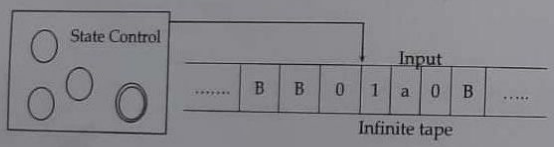
- Introduction to Turing Machine
- Instantaneous Description for TM
- Turing Machine as a Language Recognizer
- Turing Machine with Storage in the State
- Turing Machine as Enumerator of Strings of a Language
- Turing Machine with Multiple Tracks
- Equivalence of One-tape and Multi-tape TM's
- Church Thesis and Algorithm
- Encoding of Turing Machine

Introduction to Turing Machine

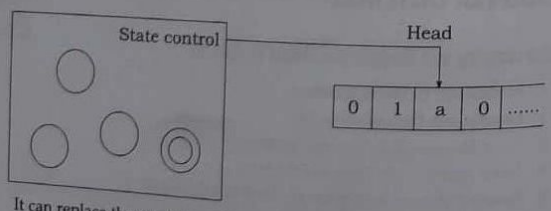
Introduction

In the previous chapters, we have seen several computational devices that can be used to accept or generate regular and context-free languages. Even though these two classes of languages are fairly large, we have seen in Section 3.8.2 that these devices are not powerful enough to accept simple languages such as $A = \{a^n b^n c^n : n \geq 0\}$. In this chapter, we introduce the Turing machine, which is a simple model of a real computer. Turing machines can be used to accept all context-free languages, but also languages such as A. Furthermore, every problem that can be solved on a real computer can also be solved by a Turing machine (this statement is known as the Church-Turing Thesis).

A Turing Machine is a finite automaton with a two-way access to an infinite tape.

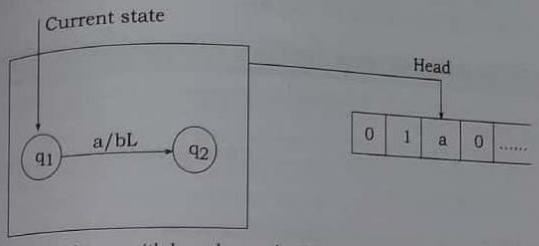


- Initially, thefew cells contain the input, and the other cells are blank.
- At each point in the computation, the machine sees its current state and the symbol at the head

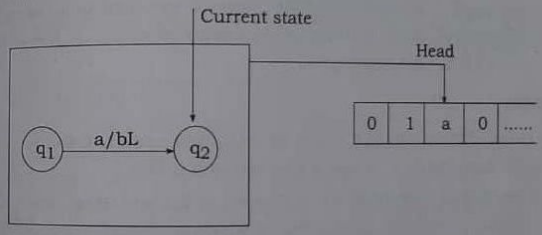


- It can replace the symbol on the tape, change state, and move the head left or right.

Example 1: Consider the following state where it reads symbol a.



Let it replace a with b, and move head left which can be shown in figure below as:



Formal Definition of Turing Machine

A Turing Machine TM is defined by the seven-tuples, $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where,

Q = the finite set of states of the finite control

Σ = the finite set of input symbols

Γ = the complete set of tape symbols Σ is always subset of Γ.

q_0 = the start state; $q_0 \varepsilon Q$

B = the blank symbol; $B \varepsilon \Gamma$ but B does not belong to Σ.

F = the set of final or accepting states; $F \subseteq Q$

δ = the transition function defined by

$Q \times \Gamma \rightarrow Q \times \Gamma \times (R, L, S)$; where R, L, S is the direction of movement ofhead-left, or right or stationary.

i.e. $\delta(q, x) = (P, Y, D)$; which means TM in state q and current tape symbol x_i moves to next state P, replacing tape symbol x with Y and move the head either direction or remains at same cell of input tape.

Instantaneous Description for TM

The configuration of a TM is described by Instantaneous description (ID) of TM as like PDA.

A string $x_1x_2\ldots\ldots\ldots\ldots x_{i-1}qx_ix_{i+1}\ldots\ldots\ldots x_n$ represents the I.D. of TM in which;

- q is the state of T M .
- the tape head scanning the i^{th} symbol from the left.
- $x_1x_2\ldots\ldots\ldots x_n$ is the portion of tape between the leftmost and rightmost non-blank.(If the head is to the left of leftmost non blank or to the right of rightmost non-blank then some prefix or suffix of $x_1x_2\ldots\ldots x_n$ will be blank and i will be 1 or n respectively.)

Moves of T M

The moves of TM, M = (Q, Σ, Γ, δ, q_0, B, F) is described by the notation $\vdash$, "yield", for single move and by $\vdash^*$ for zero or more moves as in PDA.

a. For $\delta(q, x_i) = (P, Y, L)$ i.e. next move is leftward then, $x_1x_2\ldots\ldots x_{i-1}qx_ix_{i+1}\ldots\ldots x_n \vdash x_1x_2\ldots x_{i-2}\, P\, x_{i-1}Yx_{i+1}.xn$ reflects the change of state from q to p and the replacement of symbol x_i with Y and then head is positioned at i-1 (next scan is x_{i-1}).

   i. If i=1, M moves to the left of x_1 i.e. $qx_1x_2\ldots\ldots\ldots x_n \vdash pBYx_2\ldots\ldots x_n$

   ii. If i=n, Y = B, then M moves to state p and system B written over x_n joins the infinite sequence of trailing blanks which does not appear in next ID as $x_1x_2\ldots\ldots x_{n-1}qxn \vdash x_1x_2\ldots\ldots\ldots x_{n-2}px_{n-1}$

b. If $\delta(q, x_i) = (P, Y, R)$ i.e. next move is rightward then, $x_1x_2\ldots\ldots x_{i-1}qx_i\ldots\ldots\ldots x_n \vdash x_1x_2\ldots x_{i-1}\, Y\, p\, x_{i+1}\ldots\ldots\ldots x_n$, which reflects that the symbol xi is replaced with Y and head has moved to cell i+1 with change in state from p to q.

- If i= n, then i+1 cell holds blank which is not part of previous ID; i.e. $x_1x_2\ldots\ldots\ldots x_{n-1} \vdash x_1x_2\ldots\ldots\ldots\ldots x_{n-1}\, YpB$.
- If i=1, Y=B, then the symbol B written over xi joins the infinite sequence of leading blanks and does not appear in next ID; i.e. $x_1x_2\ldots\ldots\ldots\ldots\ldots x_n \vdash px_2x_3\ldots\ldots\ldots\ldots x_n$.

Example 2: Consider an example; A TM accepting $\{0^n1^n / n \geq 1\}$

- Given finite sequence of 0's and 1's on its tape preceded and followed by blanks.
- The TM will change 0 to an X and then a 1 to Y_1 until all 0's and 1's are matches.
- Starting at left end of the input, it repeatedly changes a 0 to an X and moves to the right over whatever 0's and Y's it sees until comes to a 1.
- It changes 1 to a Y, and moves left, over Y's and 0's until it finds X. At that point, it looks for a 0 immediate to the right. If finds one 0 then changes it to X and repeats the process changing a matching 1 and Y.

Now, TM will have;

$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$

The transition rule for the move of M is described by following transition table;

	0	1	X	Y	B
q_0	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	(q_4, B, R)
q_4					

Now, the acceptance of 0011 by the TM, M1 is described by following sequence of moves;

$q_0 0011 \vdash Xq_1011 \vdash X0q_111 \vdash Xq_20Y1$
$\vdash q_2X0Y1$
$\vdash Xq_00Y1$
$\vdash XXq_1Y1$
$\vdash XXYq_11$
$\vdash XXq_2YY$
$\vdash Xq_2XYY$
$\vdash XXq_0YY$
$\vdash XXYq_3Y$
$\vdash XXYYq_3B$
$\vdash XXYYBq_4B$ Halt and accept.

For string 0110

$q_0 0110 \vdash Xq_1 110 \vdash q_2 XY10 \vdash Xq_0 Y10$

$\vdash XYq_3 10$

Halt and reject; since q_3 has no move on symbol 1.

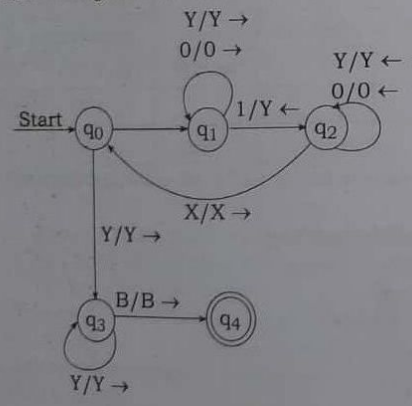
Transition diagram for a TM

A transition diagram of TM consists of,

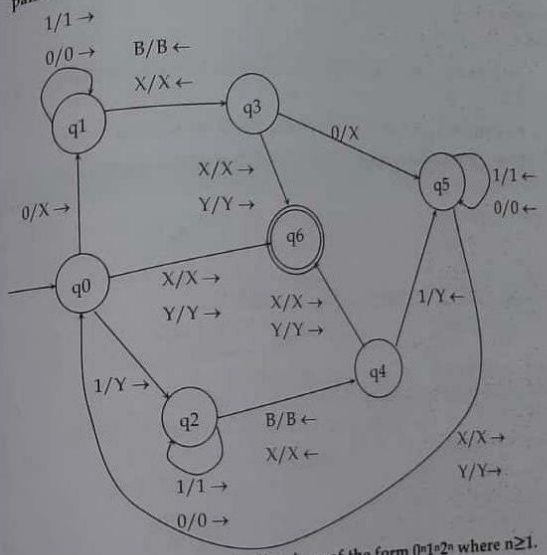
- A set of nodes representing states of TM.
- An arc from any state, q to p is labeled by the items of the form X / YD, where X and Y are tape symbols, and D is a direction, either L or R. that is, whenever $\delta(q, x) = (P, Y, D)$, we find the label X / YD on the arc from q to p.

However, in diagram, the direction D is represented by ← for left (L) and → for right (R)

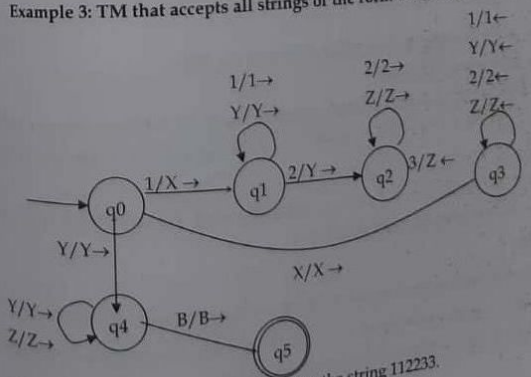
Thus, transition diagram for the TM for $L = \{0^n1^n / n \geq 1\}$ as;



Example 2: Transition diagram for the TM which accepts the set of all palindromes over {0,1}



Example 3: TM that accepts all strings of the form $0^n1^n2^n$ where n≥1.



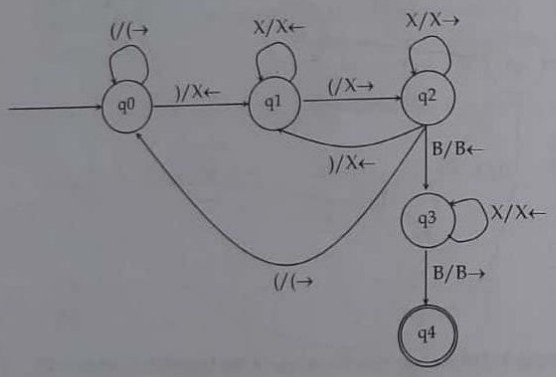
Show the instantaneous description for the string 112233.

Example 4: Design a TM that accepts well formed string of parenthesis. i.e. L = {(()),()(()),(()())(),.....}.

Solution:

Idea:

- Find the first), replace it with X and find its corresponding (replace it with X.
- Perform the above step until all the symbols are scanned.
- Finally if the tape consist of only X or B then accept otherwise not.



Show the instantaneous description for the string: (()()).

Language of Turing Machine

If T= (Q, Σ, Γ, δ, q_0, B, F) is a Turing machine and w ε Σ *, then language accepted by T, L (T) = {w | w ε Σ* and $q_0 w \vdash^* \alpha\ p\ \beta$} for some pεF and any tape string α and β.

The set of languages that can be accepted using TM are called recursively enumerable languages or RE languages.

Role of TM

The Turing Machine is designed to perform at least the following three roles;

1) As a language recognizer: TM can be used for accepting a language like Finite Automaton and Pushdown Automata.

2) As a Computer of function: A TM represents a particular function. Initial input is treated as representing an argument of the function. And the final string on the tape when the TM enters the halt state; treated as representative of the value obtained by an application of the function to the argument represented by the initial string.
3) As an enumerator of string of a language: It outputs the strings of a language, one at a time in some systematic order that is as a list.

Turing Machine as a Language Recognizer

A Turing Machine can be used a language reconizer to accept strings of certain language. For example, A TM accepting $\{0^n1^n / n \geq 1\}$ as mentioned previously.

Turing Machine as a Computing a Function:

A Turing Machine can be used to compute functions. For such TM, we adopt the following policy to input any string to the TM which is an input of the computation function.

1) The string w is presented into the form BwB, where B is a blank symbol, and placed onto the tape; the head of TM is positioned at a blank symbol which immediately follows the string w.
2) The TM is said to halt on input w if we can reach to halting state after performing some operation.

i.e. If TM = (Q, Σ, Γ, δ, q_0, B, {q_a}) is a turing machine . Then this TM is said to be halt on input w if and only if $BwqB$ yields to $B\alpha q_a B$, for some αεΓ* i.e. $(qBwB) \vdash (B\alpha q_a B)$.

Definition

A function $f(x) = y$ is said to be computable by a TM (Q, ∑, Γ, δ , $q0$, B, {q_a}) if $(q0\ ,BxB) \vdash^* (q_a,\ ByB)$ where x may be in some alphabet $\sum_1$ * and y may be in some alphabet $\sum_2$ * and $\sum_1, \sum_2 \subseteq \sum$. It means that if we give input x to the Turing Machine TM , it gives output as a string if it computes the function $f(x) = y$.

Example 5 : Design a TM which compute the function f(x) = x +1 for each x that belongs to the set of natural numbers.

Solution:

Given function f(x) = x + 1. Here we represent input x on the tape by a number of 1's on the tape.

For example x = 1, input will be B1B,

for x = 2 , input will be B11B,

for x = 3, input will be B111B and so on.

Similarly output can be seen by the number of 1s on the tape when machine halts.

Let TM = $(Q, \sum, \Gamma, \delta, q_0, B, \{q_a\})$ where Q = {q0,qa} Γ ={1,B} and halt state qa . The transition function is defined as

Q	B	1
q_0	(q_0,B,R)	$(q_1,1,R)$
q_1	$(q_a,1,R)$	$(q_1,1,R)$
q_a	-	-

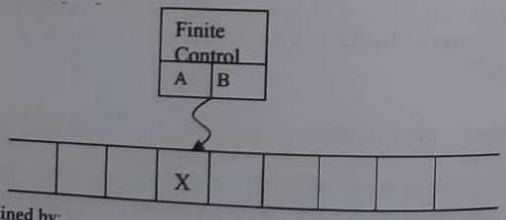
Let input x = 4, that is input tape contains input as B1111B.

So $(q_0B1111B) \vdash (Bq_01111B) \vdash (B1q_11111B) \vdash (B11q_1111B) \vdash (B111q_11B) \vdash (B1111q_1B) \vdash (B11111q_aB)$ Which means output is 5.

Turing Machine with Storage in the State

In Turing machine, any state represents the position in the computation. But a state can also be used to hold a finite amount of data. The finite control of machine consists of a state q and some data portion. In this case a state is considered as a tuple - (state,data).

Following figure illustrates the model.



δ is defined by:

$\delta([q, A], X) = ([q1, X], Y, R)$ means q is the state and data portion of q is A. The symbol scanned on the tape is copied into the second component of the state and moves right entering state q1 and replacing tape symbol by Y.

Turing Machine as Enumerator of Strings of a Language

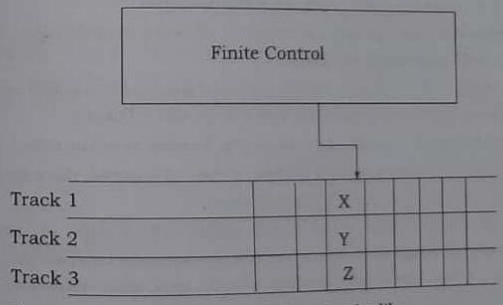
We have viewed turing machines as recognizers of languages and computers of functions on the non-negative integers. There is a third useful view of turing machines, as generating devices.

Consider a multi-tape turing machine TM that uses on tape as an output tape, which a symbol, once written can never be changed, and those whose tape head never moves left. Suppose also that on the output tape, TM writes strings over some alphabet $\sum$, separated by a marker symbol #. We can define L(TM), the language generated by TM to be set of w in $\sum^*$ such that w is eventually printed between a pair of #'s on the output tape.

Thus the language of this type of Turing machine is called Turing Enumerable languages.

Turing Machine with Multiple Tracks

The tape of TM can be considered as having multiple tracks. Each track can hold one symbol, and the tape alphabet of the TM consists of tuples, with one component for each track. Following figure illustrates the TM with multiple tracks;



The tape alphabet Γ is a set consisting of tuples like,

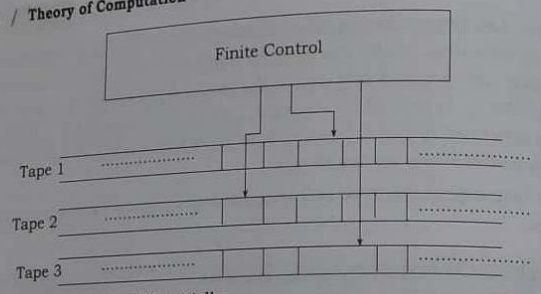
Γ = {(X, Y, Z), ...}

The tape head moves up and down scanning symbols in the tape at one position.

Multi-tape Turing Machine

Modern computing devices are based on the foundation of TM computation models. To simulate the real computers, a TM can be viewed as multi-tape machine in which there is more than one tape. However, adding extra tape adds no power to the computational model, only the ability to accept the language is concerned.

A multi-tape TM consists of finite control and finite number of tapes. Each tape is divided into cells and each cell can hold any symbol of finite tape alphabets. The set of tape symbols include a blank and the input symbols.



In the multi-tape TM, initially;

- Input (finite sequence of input symbols) w is placed on the first tape.
- All other cells of the tapes hold blanks.
- TM is in initial state
- The head of the first tape is at the left end of the input.
- All other tape heads are at some arbitrary cell. Since all other tapes except first tape consists completely blank.

A move of multi- tape TM depends on the state and the symbol scanned by each of the tape head. In one move, the multi-tape TM does the following:

- The control enters in a new state, which may be same previous state.
- On each step, a new symbol is written on the cell scanned, these symbols may be same as the symbols previously there.
- Each of the tape head make a move either left or right or remains stationary. Different head may move different direction independently i.e. if head of first tape moves leftward; at same time other head can move another direction or remains stationary.

The initial configuration (initial ID) of multi-tape TM with n-tapes is represented as;

$(q_0, ax, B, B, \ldots\ldots\ldots\ldots B)$; n+1 tuples.

Where w= ax is an input string and head first tape is scanning first symbol of w.

So, in general, it can be rewritten as;

$(q, x_1a_1y_1, x_2a_2y_2, \ldots\ldots\ldots\ldots\ldots\ldots x_na_ny_n)$

Where each x_i are the portion of string on tapes before the current head position, each a_i are the symbol currently scanning in each tapes and each y_i are the portion of string on tapes just rightward to the current head position q is the control state.

Equivalence of One-tape and Multi-tape TM's

Theorem: Every language accepted by a multitape TM is recursively enumerable.

or

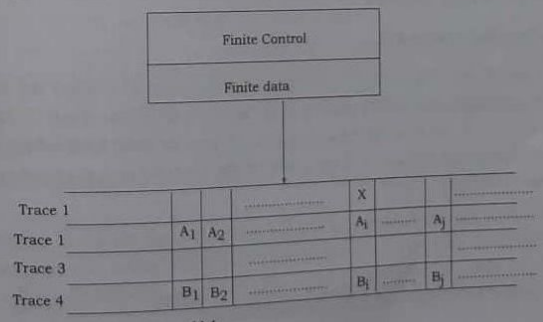
Any languages that are accepted by a multi-tape TM are also accepted by one tape Turing Machine.

(Since One tape accepts only the recursively enumerable language. Hence if multi-tape TM accepts the recursively enumerable then we can convert it into its equivalent One-tape TM . Thus to prove above theorem we have to describe the way to convert mult-tape TM to One tape TM.)

Proof:

- Let L is a language accepted by a k-tape TM, M. Now, can simulate M with a one-tape TM, N whose tape consists of 2k tracks.
- Half of these tracks holds the tapes of M, and the other half of the tracks each hold only a single marker that indicates where the head for the corresponding tape of M is currently located.

i.e. To simulate 2-tape turing machine using 1-tape turing machine we need a tape having 4-tracks. The second and fourth tracks hold the contents of the first and second tapes of M, track first holds the position of the head of tape 1, and track third holds the position of the second tape head.



Now, to simulate a move of M,

- N's head must visit the k-head markers.
- After visiting each head marker and storing the scanned symbol in a component of its finite control, N knows what tape symbols are being scanned by each of M's head.
- N also knows the state of M, which it stores in N's own finite control. Thus N knows what move M will make.

- M now revisits each of the head markers on its tape, changes the symbol in track representing corresponding tapes N and moves the head marker left or right, if necessary.

Finally, N changes the state of M as recorded in its own finite control. Hence N has simulated one move of M.

We select N's accepting states, all those states that record M's state as one of the accepting a state of M. Hence, whatever M accepts N also accepts.

Non-deterministic Turing Machine

A non-deterministic Turing Machine (NTM), $T = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is defined exactly the same as an ordinary TM, except the value of transition function δ. In NTM, the values of the transition function δ are subsets, rather than a single element of the set $Q \times \Gamma \times \{R,L,S\}$. Here, the transition function δ is such that for each state q and tape symbol x, $\delta(q, x)$ is a set of triples.

$$\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (q_k, Y_k, D_k)$$

where k is any finite integer.

The NTM can choose, at each step, any of the triples to be the next move.

Restricted Turing Machine

Linear Bounded Automaton

Linear Bound Automation is a type of turing machine wherein the tape is not permitted to move off the portion of the tape containing the input. If the machine tries to move its head off either end of the input, the head stays where it is,in the same way that the head will not move off the left-hand end of an ordinary turing machine's tape.

A Linear bounded automaton is a TM with a limited amount of memory. It can only solve problems requiring memory that can fit within the tape used for the input. Using a tape alphabet larger than the input alphabet allows the available memory to be increased up to a constant factor.

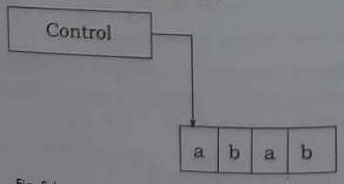
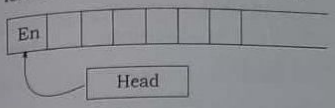


Fig: Schematic of Linear Bounded Automata

Semi-Infinite Tape Turing Machine

A Turing Machine with a semi-infinite tape has a left end but no right end. The left end is limited with an end marker.



It is a two-track tape –

- **Upper track** – It represents the cells to the right of the initial head position.
- **Lower track** – It represents the cells to the left of the initial head position in reverse order.

The infinite length input string is initially written on the tape in contiguous tape cells.

The machine starts from the initial state q_0 and the head scans from the left end marker 'End'. In each step, it reads the symbol on the tape under its head. It writes a new symbol on that tape cell and then it moves the head either into left or right one tape cell. A transition function determines the actions to be taken.

It has two special states called **accept state** and **reject state**. If at any point of time it enters into the accepted state, the input is accepted and if it enters into the reject state, the input is rejected by the TM. In some cases, it continues to run infinitely without being accepted or rejected for some certain input symbols.

Note – Turing machines with semi-infinite tape are equivalent to standard Turing machines.

Multi Stack Machines

- Generalizations of the PDAs
- TMs can accept languages that are not accepted by any PDA with one stack.
- But PDA with two stacks can accept any language that a TM can accept.

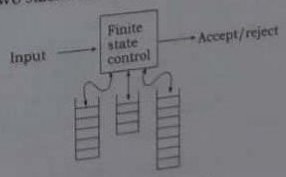


Figure: A machine with three stacks

- A k-stack machine is a deterministic PDA with k stacks.
- It obtains its input from an input source rather than having the input placed in a tape.
- It has a finite control, which is in one of a finite set of states.
- It has a finite stack alphabet, which it uses for all its stacks.
- A move of a multistack machine is based on:
- The state of the finite control
- The input symbol read, which is chosen from the finite input alphabet
- The top stack symbol on each stack
- In one move:
- a multistack machine can change to a new state $q \in Q$;
- and replace the top symbol of each stack with a string of zero or more stack symbols $X \in \Gamma^*$.
- The typical transition function of k-stack machine looks similar to:
- In state q, with Xi on top of ith stack,
- For i= 1, 2, ..., k, the machine may consume input a, go to state p, and replace Xi on top of the ith stack by string yi, for i=1,2, ..., k.
- To make it easier for a multistack machine to simulate a TM, we introduce a special symbol called the endmarker, represented by $.
- The role of the endmarker is To let us know when we have consumed all the available input.
- The endmarker appears only at the end of the input and is not part of it.

Counter Machines

Counter Machine are offline TMs (is a multi-tape TM whose input tape is read only) whose storage tapes are semi-infinite,and whose tape alphabets contain only two symbols Z and B(Blank).

Furthermore the symbol Z,which serves as a bottom of stack marker,appears initially on the cell scanned by the tape head and may never appear on any other cell.

An integer i can be stored by moving the tape head i cells to the right of Z. A stored number can be incremented or decremented by moving the tape head right or left. We can test whether a number is zero by checking whether Z is scanned by the head, but we cannot directly test whether two numbers are equal. Instantaneous description of a counter machine can be described by the state, the input tape contents, the position of the input head, and the distance of the storage heads from the symbol Z.

The counter machine can really only stores a count on each tape and tell if that count is zero.

CHURCH THESIS AND ALGORITHM

It is a mathematically un-provable hypothesis about the computability. This hypothesis simplify states that – **"Any algorithmic procedure that can be carried out at all (by human, a team of human or a computer) can be carried out by a TM."**

This statement was first formulated by Alonzo Church a logician, in 1930s and it is referred to as Church Thesis or Church-Turing thesis. It is not a mathematically precise statement so un-provable.

According to Church, **"No computational procedure will be considered an algorithm unless it can be represented by a Turing Machine."**

After adopting Church-Turing Thesis, we are giving a precise meaning of the term:

An algorithm is a procedure that can be executed on a Turing Machine.

Another use of Church-Thesis is that- When we want to describe a solution to a problem, we will often satisfied with a verbal description of the algorithm, translating it into detailed Turing Machine implementations.

Universal Turing Machine

A Turing machine is created to execute a specific algorithm. If we have a Turing machine for computing one function , then for computing different function or doing some other calculation, a different TM will be required.

Originally electronic computers were limited in a similar way , and changing the computation to be performed , requires rewriting the machine. But the modern computer are general purpose, hence to simulate modern computer Alan Turing proposed the concept of Universal Turing Machine. It can simulate arbitrary Turing machine. i.e. This single machine can perform the function of any other Turing machine.

Definition:

A *universal* Turing machine is a Turing machine T_u that works as follows.

It is assumed to receive an input string of the form $e(T)e(w)$, where T is an arbitrary TM, w is a string over the input alphabet of T , and e is an encoding function whose values are strings in $[0, 1]^*$. The computation performed by T_u on this input string satisfies these two properties:

1. T_u accepts the string $e(T)e(w)$ if and only if T accepts w.
2. If T accepts w and produces output y, then T_u produces output $e(y)$.

Encoding of Turing Machine

For the representation of any arbitrary TM T, and an input string w over an arbitrary alphabet as strings $e(T)$, $e(w)$ over some fixed alphabet , a notational system should be formulated . Encoding the TM T and input string w in to $e(T)$ and $e(w)$, it must not destroy any information , for the encoding of TM , we use the alphabet $\{0,1\}$ although the TM may have a much larger alphabet .

For encoding , we start by assigning positive integers to each state , each tape symbol and each of three directions S,L *and* R in the TM we want to encode. We assume two fixed infinite sets Q = { q_1, q_2,.....} and S = { a_1, a_2, ...} so that for any TM

$T_1 = (Q_1,\Sigma,\Gamma,\delta ,q0 , B, F)$, we have $Q_1 \subseteq Q$ and $\Gamma \subseteq S$.

Hence once we have a subscript attached to every possible state and tape symbols, we can represent a state or a symbol by a string of 0's of the appropriate length. 1's are used as separators.

The Encoding Function e

First, associate to each tape symbol, to each state and to each of the three directions , a string

of 0's. Let

$s(B) = 0$

$s(a_i) = 0^{i+1}$ *for each* $a_i \in S$

$s(q_i) = 0^{i+2}$ *for each* $q_i \in S$

$s(S) = 0$

$s(L) = 00$

$s(R) = 000$

Then each move of TM , described by formulaδ $(q, a) = (p,b, D)$ is encoded as

$e(m) = s(q)1s(a)1s(p)1s(b)1s(D)1$

and for any TM T, with initial state q, T is encoded as:

$e(T) = s(q)1e(m1)1e(m2)1.....1e(mk)1$

where $m1$, $m2$,.....mk are the distinct moves of T arranged in some arbitrary order. Finally,

any string $w = w1w2w3.....wk$, for each $wk \in S$ is encoded as:

$e(w) = 1s(w1)1s(w2)1.....1s(wk)1$

An Example: Let TM T is given as:

$T = (\{q1,q2,q3\},\{a,b\},\{a,b, B\},\delta ,q1, B, F)$ where δ is defined by the following moves

$m1 = \delta (q1,b) = (q3, a, R)$

$m2 = \delta (q3,a) = (q1,b, R)$

$m3 = \delta (q3,b) = (q2 ,a, R)$

$m4 = \delta (q3, B) = (q3,b, L)$

Encoding :

$s(q1) = 000$

$s(q2) = 0000$

$s(q3) = 00000$

$s(a1) = 00$ considering $a1 = a$ *and* $a2 = b$

$s(a2) = 000$

$s(B) = 0$

$s(S) = 0$

$s(L) = 00$

$s(R) = 000$

Now

$e(m1) = s(q1)1s(b)1s(q3)1s(a)1s(R)1 = 000100010000010010001$

$e(m2) = s(q3)1s(a)1s(q1)1s(b)1s(R)1 = 000001001000100010001$

$e(m3) = s(q3)1s(b)1s(q2)1s(a)1s(R)1 = 000001000100001010001$

$e(m4) = s(q3)1s(B)1s(q3)1s(b)1s(L)1 = 000001010000010001001$

Now The code for T will be:

$e(T)$ = $s(q1)1e(m1)1e(m2)1e(m3)1e(m4)1$

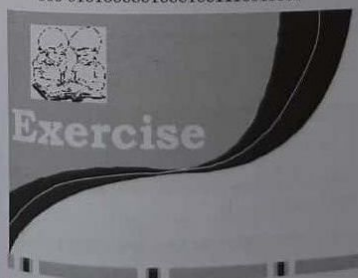
= 000100010001000001001000110000010010001000100011000001000100010001100000100010000 10010001100 000101000001000010011

for this machine the input (T,w) where w= ab , the code will be,

e(w) = 1s(a)1s(b)1 = 10010001

Code for (T,W) is:

000100010001000001001000110000010010001000100011000001000100010010001100 000 010100000100010011110010001



1. Construct a Turing Machine accepting a string ab.
2. Construct a Turing machine accepting a language of strings ending with aba.
3. Construct a Turing Machine accepting the language, L={ $\{^n\}^n$ }. Also show the transition diagram of the machine.

4. How Turing Machine can be simulated as having storage space in its finite control. Illustrate with an example.
5. How a multi-tape Turing Machine works? Is it possible to simulate a multi-tape TM with just a single tape? If yes, Justify.
6. Construct a Turing Machine accepting the language, $L=\{ a^n b^n \mid n>=1\}$. Also show the transition diagram of the machine.
7. Construct a Turing Machine accepting the language, $L= \{ a^n b^{2n} \mid n>=1\}$. Also show the transition diagram of the machine.
8. Design a TM for language $L= \{x^{2n}y^n / n>=0\}$. Show instantaneous description for xxxxyy and xxyy.
9. Construct a Turing Machine accepting the strings represented by ab^*+ba^*. Also draw the equivalent transition diagram.
10. Construct a TM for language $L= \{a^n b^n c^n / n>=0\}$. Show instantaneous description for aabbcc and abcc.
11. Design a TM that recognizes the language $L = \{0^{2n} \mid n \geq 0\}$
12. Design a Turing Machine to calculate the
    (i) 1- complement of a binary number
    (ii) 2- complement of a binary number
13. Construct a non-deterministic Turing Machine, which recognizes the language $\{ww/w \in \{a, b\}^*\}$.
14. Let M be the Turing Machine defined by

δ	B	a	b	c
q_0	q_1, B, R			
q_1	q_1, B, R	q_1, a, R	q_1, b, R	q_2, C, L
q_2		q_2, b, L	q_2, a, L	

(a) Trace the computation for the input string abcab.
(b) Trace the first six translations of the computation for the input string abab.
(c) Give the state diagram of M.
(d) Describe the result of a computation in M.

❑❑❑

UNDECIDABILITY AND INTRACTABILITY

CHAPTER OUTLINES

After studying this Chapter you should be able to:

- Computational Complexity
- Complexity Classes
- Problems and its Types
- Reducibility
- Circuit Satisfaibility
- Undecidability
- Post's Correspondence Problem (PCP)

COMPUTATIONAL COMPLEXITY

Complexity Theory is a central field of the theoretical foundations of Computer Science. It is concerned with the study of the *intrinsic complexity of computational tasks*. That is, a typical Complexity theoretic study looks at a task (or a class of tasks) and at the computational resources required to solve this task, rather than at a specific algorithm or algorithmic scheme.

The complexity of computational problems can be discussed by choosing a specific abstract machine as a model of computation and considering how much resource machine of that type require for the solution of that problem.

Complexity Measure is a means of measuring the resource used during a computation. In case of Turing Machines, during any computation, various resources will be used, such as space and time. When estimating these resources, we are always interested in growth rates rather than absolute values.

A problem is regarded as inherently difficult if solving the problem requires a large amount of resources, whatever the algorithm used for solving it. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as time and storage. Other complexity measures are also used, such as the amount of communication (used in communication complexity), the number of gates in a circuit (used in circuit complexity) and the number of processors (used in parallel computing). In particular, computational complexity theory determines the practical limits on what computers can and cannot do.

Time and Space Complexity of a Turing Machine

The model of computation we have chosen is the Turing Machine. When a Turing machine answers a specific instance of a decision problem we can measure time as number of moves and the space as number of tape squares required by the computation. The most obvious measure of the size of any instance is the length of input string. The worst case is considered as the maximum time or space that might be required by any string of that length.

The time and space complexity of a TM can be defined as; let T be a TM. The time complexity of T is the function T_t defined on the natural numbers as; for $n \in N$, $T_t(n)$ is the maximum number of moves T can make on any input string of length n. If there is an input string x such that for $|x|=n$, T loops forever on input T, $T_t(n)$ is undefined.

The space complexity of T is the function S_t defined as $S_t(n)$ is the maximum number of the tape squares used by T for any input string of length n. If T is multi-tape TM, number of tape squares means maximum of the number of individual tapes. If for some input of length n, it causes T to loop forever, $S_t(n)$ is undefined.

An algorithm for which the complexity measures $S_t(n)$ increases with n, no more rapidly than a polynomial in n is said to be *polynomially bounded*; one in which it grows exponentially is said to be *exponentially bounded*.

Intractability

Intractability is a technique for solving problems not to be solvable in polynomial time. The problems that cab be solved by any computational model, probably TM, using no more time then some slowly growing function size of the input are called "tractable:, i.e. those problems solvable within reasonable time and space constraints (polynomial time). The problems that cannot be solved in polynomial time but requires superpolynomial (exponential) time algorithm are called intractable or hard problems. There are many problems for which no algorithm with running time better than exponential time is known some of them are, traveling salesman problem, Hamiltonian cycles, and circuit satisfiability, etc.

To introduce intractability theory, the class P and class NP of problems solvable in polynomial time by deterministic and non-deterministic TM's are essential. A solvable problem is one that can be solved by particular algorithm i.e. there is an algorithm to solve this problem. But in practice, the algorithm may require a lot of space and time. When the space and time required for implementing the steps of the particular algorithm are (polynomial) reasonable, we can say that the problem is tractable. Problems are intractable if the time required for any of the algorithm is at least f(n), where f is an exponential function of n.

COMPLEXITY CLASSES

In computational complexity theory, a **complexity class** is a set of problems of related resource-based complexity. A typical complexity class has a definition of the form:

"The set of problems that can be solved by an abstract machine M using O(f(*n*)) of resource R, where *n* is the size of the input."

For example, the **class NP** is the set of decision problems that can be solved by a non-deterministic Turing machine in polynomial time, while the **class P** is the set of decision problems that can be solved by a deterministic Turing machine in polynomial space.

The simpler complexity classes are defined by the following factors:

- **The type of computational problem:** The most commonly used problems are decision problems. However, complexity classes can be defined based on function problems, optimization problems, etc.
- **The model of computation:** The most common model of computation is the deterministic Turing machine, but many complexity classes are based on nondeterministic Turing machines, Boolean circuits etc.
- **The resource (or resources) that are being bounded and the bounds:** These two properties are usually stated together, such as "polynomial time", "logarithmic space", "constant depth", etc.

The set of problems that can be solved using polynomial time algorithm is regarded as **class P**. The problems that are verifiable in polynomial time constitute the **class NP**. The class of **NP complete** problems consists of those problems that are NP as well as they are *as hard as* any problem in NP. The main concern of studying NP completeness is to understand how hard the problem is. So if we can find some problem as NP complete then we try to solve the problem using methods like approximation, rather than searching for the faster algorithm for solving the problem exactly.

Class P: The class P is the set pf problems that can be solved by deterministic TM in polynomial time. A language L is in class P if there is some polynomial time complexity T(n) such that L=L(M), for some Deterministic Turing Machine M of time complexity T(n).

Class NP: The class NP is the set of problems that can be solved by a non-deterministic TM in polynomial time. Formally, we can say a language L is in the class NP if there is a non-deterministic TM, M, and a polynomial time complexity T(n), such that L= L(M), and when M is given an input of length n, there are no sequences of more than T(n) moves of M.

Note: Since every deterministic TM is a non-deterministic TM having no choice of more than one moves, so P is subset of NP. However NP contains many problems that are not in P. The class **P** consists of all those decision problems that can be solved on a Deterministic Turing Machine in an amount of time that is polynomial in the size of the input; the class **NP** consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a Non-deterministic Turing Machine.

NP-Complete: In computational complexity theory, the complexity class **NP-complete** (abbreviated **NP-C** or **NPC**), is a class of problems having two properties:

- It is in the set of NP (nondeterministic polynomial time) problems: Any given solution to the problem can be *verified* quickly (in polynomial time).
- It is also in the set of NP-hard problems: Any NP problem can be converted into this one by a transformation of the inputs in polynomial time.

Formally;

Let L be a language in NP, we say L is NP-Complete if the following statements are true about L;

- L is in class NP
- For every language L_1 in NP, there is a polynomial time reduction of L_1 to L.

Once we have some NP-Complete problem, we can prove a new problem to be NP-Complete by reducing some known NP-Complete problem to it using polynomial time reduction.

It is not known whether every problem in NP can be quickly solved—this is called the P = NP problem. But if *any single problem* in NP-complete can be solved quickly, then *every problem in NP* can also be quickly solved, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every problem in NP-complete (that is, it can be reduced in polynomial time). Because of this, it is often said that the NP-complete problems are *harder* or *more difficult* than NP problems in general.

Some of the properties of NP-complete problems are:

- No polynomial time algorithms has been found for any of them.
- It is no established that polynomial time algorithm for these problems do not exist.
- If polynomial time algorithm is found for any one of them, there will be polynomial time algorithm for all of them.
- If it can be proved that no polynomial time algorithm exists for any of them, then it will not exist for every one of them.

Problems and its Types

Abstract Problems

Abstract problem A is binary relation on set I of problem instances, and the set S of problem solutions. For e.g. Minimum spanning tree of a graph G can be viewed as a pair of the given graph G and MST graph T.

Decision Problems

Decision problem D is a problem that has an answer as either "true", "yes", "1" or "false", "no", "0". For e.g. if we have the abstract shortest path with instances of the problem and the solution set as {0,1}, then we can transform that abstract problem by reformulating the problem as "Is there a path from u to v with at most k edges". In this situation the answer is either yes or no.

Optimization Problems

We encounter many problems where there are many feasible solutions and our aim is to find the feasible solution with the best value. This kind of problem is called optimization problem. For e.g. given the graph G, and the vertices u and v find the shortest path from u to v with minimum number of edges. The NP completeness does not directly deal with optimizations problems; however we can translate the optimization problem to the decision problem.

Function Problems

In computational complexity theory, a **function problem** is a computational problem where a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem, that is, it isn't just YES or NO. Notable examples include the *Traveling salesman problem*, which asks for the route taken by the salesman, and the *Integer factorization problem*, which asks for the list of factors. Function problems can be sorted into complexity classes in the same way as decision problems. For example FP is the set of function problems which can be solved by a deterministic Turing machine in polynomial time, and FNP is the set of function problems which can be solved by a non-deterministic Turing machine in polynomial time. For all function problems in which the solution is polynomially bounded, there is an analogous decision problem such that the function problem can be solved by polynomial-time Turing reduction to that decision problem.

Encoding

Encoding of a set S is a function e from S to the set of binary strings. With the help of encoding, we define **concrete problem** as a problem with problem instances as the set of binary strings i.e. if we encode the abstract problem, then the resulting encoded problem is concrete problem. So, encoding as a concrete problem assures that every encoded problem can be regarded as a language i.e. subset of $\{0,1\}^*$.

Reducibility

Reducibility is a way of converging one problem into another in such a way that, a solution to the second problem can be used to solve the fist one.

Many complexity classes are defined using the concept of a reduction. *A reduction is a transformation of one problem into another problem.* It captures the informal notion of a problem being at least as difficult as another problem. For instance, if a problem X can be solved using an algorithm for *Y*, X is no more difficult than *Y*, and we say that *X reduces* to *Y*. There are many different type of reductions, based on the method of reduction, such as Cook reductions, Karp reductions and Levin reductions, and the bound on the complexity of reductions, such as *polynomial-time reductions* or *log-space reductions*.

The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time. For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers. This means an algorithm for multiplying two integers can be used to square an integer. Indeed, this can be done by giving the same input to both inputs of the multiplication algorithm. Thus we see that squaring is not more difficult than multiplication, since squaring can be reduced to multiplication.

This motivates the concept of a problem being hard for a complexity class. A problem *X* is *hard* for a class of problems C if every problem in C can be reduced to *X*. Thus no problem in C is harder than *X*, since an algorithm for *X* allows us to solve any problem in C. Of course, the notion of hard problems depends on the type of reduction being used. For complexity classes larger than P, polynomial-time reductions are commonly used. In particular, the set of problems that are hard for NP is the set of NP-hard problems.

CIRCUIT SATISFAIBILITY

Cook's Theorem

Lemma: SAT is NP-hard

Proof: (This is not actual proof as given by cook, this is just a sketch)

Take a problem V ∈ NP, let A be the algorithm that verifies V in polynomial time (this must be true since V ∈ NP). We can program A on a computer and therefore there exists a (huge) logical circuit whose input wires correspond to bits of the inputs x and y of A and which outputs 1 precisely when A(x, y) returns yes.

For any instance x of V let A_x be the circuit obtained from A by setting the x-input wire values according to the specific string x. The construction of A_x from x is our reduction function. If x is a yes instance of V, then the certificate y for x gives satisfying assignments for A_x. Conversely, if A_x outputs 1 for some assignments to its input wires, that assignment translates into a certificate for x.

Theorem: SAT is NP-complete

Proof:

To show that SAT is NP-complete we have to show two properties as given by the definition of NP-complete problems. The first property i.e. SAT is in NP

Circuit satisfiability problem (SAT) is the question "Given a Boolean combinational circuit, is it satisfiable? i.e. does the circuit has assignment sequence of truth values that produces the output of the circuit as 1?" Given the circuit satisfiability problem take a circuit x and a certificate y with the set of values that produce output 1, we can verify that whether the given certificate satisfies the circuit in polynomial time. So we can say that circuit satisfiability problem is NP.

This claims that SAT is NP. Now it is sufficient to show the second property holds for SAT. The proof for the second property i.e. SAT is NP-hard is from above lemma. This completes the proof.

UNDECIDABILITY

In computability theory, an undecidable problem is a decision problem for which it is impossible to construct a single algorithm that always leads to a correct "yes" or "no" answer- the problem is not decidable. An **undecidable problem** consists of a family of instances for which a particular yes/no answer is required, such that there is no computer program that, given any problem instance as input, terminates and outputs the required answer after a finite number of steps. More

formally, an undecidable problem is a problem whose language is not a recursive set or computable or decidable.

In computability theory, **the halting problem** is a decision problem which can be stated as follows:

Given a description of a program and a finite input, decide whether the program finishes running or will run forever.

Alan Turing proved in 1936 that a general algorithm running on a Turing machine that solves the halting problem for *all* possible program-input pairs necessarily cannot exist. Hence, the halting problem is *undecidable* for Turing machines.

POST'S CORRESPONDENCE PROBLEM (PCP)

The Post's Correspondence Problem is an undecidable decision problem that was Introduced by Emil Post in 1946.

Definition: The input of the problem consists of two finite lists $U = \{u_1, u_2, \ldots, u_n\}$ and $V = \{v_1, v_2, \ldots, v_n\}$ of words over some alphabet ∑ having at lest two symbols. A solution to this problem is a sequence of indices i_k, $1 <= k <= n$, for all k, such that $u_{i1} u_{i2} \ldots u_{ik} = v_{i1} v_{i2} \ldots v_{ik}$

We say $i_1, i_2, \ldots i_k$ is a solution to this instance of PCP.

Here, the decision problem is to decide whether such a solution exits or not.

Examples:

Consider the following two lists:

U		
u_1	u_2	u_3
a	ab	bba

V		
v_1	v_2	v_3
baa	aa	bb

A solution to this problem would be the sequence (3, 2, 3, 1), because

$u_3u_2u_3u_1 = bba + ab + bba + a = bbaabbbaa$

$v_3v_2v_3v_1 = bb + aa + bb + baa = bbaabbbaa$

Furthermore, since (3, 2, 3, 1) is a solution, so are all of its "repetitions", such as (3, 2, 3, 1, 3, 2, 3, 1), etc.; that is, when a solution exists, there are infinitely many solutions of this repetitive kind.

However, if the two lists had consisted of only u_2,u_3 and v_2,v_3, then there would have been no solution (because then no matching pair would have the same last letter, as must occur at the end of a solution).

Consider another example with two lists as below

U			V		
u_1	u_2	u_3	v_1	v_2	v_3
10	011	101	101	11	011

For this instance of PCP, there is no solution !!!

Halting Problem

"Given a Turing Machine M and an input w, do M halts on w?"

Algorithms may contain loops which may be infinite or finite in length. The amount of work done in an algorithm usually depends on data input. Algorithms may consists of various numbers of loops nested or in sequence. Thus, the halting problem asks the question; *"Given a program and an input to the program, determine if the program will eventually stop when it is given that input."* The question is simply whether the given program will ever halt on a particular input.

Trial Solution

Just run the program with the given input. If the program stops we know the program halts. But if the program does not stop in reasonable amount of time, we can not conclude that it won't stop. May be we did not wait long enough!

For example, in pseudocode, the program

while True: continue

does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program

print "Hello World!"

halts very quickly.

The halting problem is famous because it was one of the first problems proven algorithmically undecidable. This means there is no algorithm which can be applied to any arbitrary program and input to decide whether the program stops when run with that input.

The Halting Problem is one of the simplest problems know to be unsolvable. You will note that the interesting problems involve loops.

Consider the following Javascript program segments (algorithm):

```
for(quarts = 1 ; quarts < 10 ; quarts++)
{
   liters = quarts/1.05671;
   alert( quarts+" "+liters);
}
```

```
limit = prompt("Max Value","");
for(quarts = 1 ; quarts < limit ; quarts++)
{
   liters = quarts/1.05671;
   alert( quarts+" "+liters);
}
```

```
green = ON
red = amber = OFF
while(true)
{
   amber = ON; green = OFF;
   wait 10 seconds;
   red = ON;  amber = OFF;
   wait 40 seconds;
   green = ON; red = OFF;
}
```

The first program clearly is a one that will terminate after printing in an alert 10 lines of output. The second program alerts as many times as indicated by the input. The last program runs forever.

Sketch of a proof that the Halting Problem is undecidable

This proof was devised by Alan Turing in 1936.

Suppose we have a solution to the halting problem called H. H takes two inputs:

1. a program P and
2. an input I for the program P.

H generates an output "***halt***" if H determines that P stops on input I or it outputs "***loop***" otherwise.

ASIDE: When an algorithm is coded, it is expressed as a string of characters which can also be interpreted as a sequence of numbers (binary). We can treat the program as data and therefore a program can be thought of as input.

For example, compilers take programs as input and generate machine code as output. Netscape takes a Javascript program and generates output.

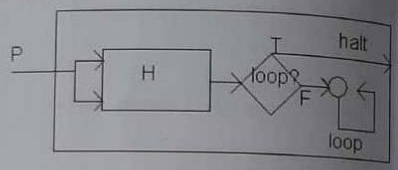
So now H can be revised to take P as both inputs (the program and its input) and H should be able to determine if P will halt on P as its input.

Let us construct a new, simple algorithm K that takes H's output as its input and does the following

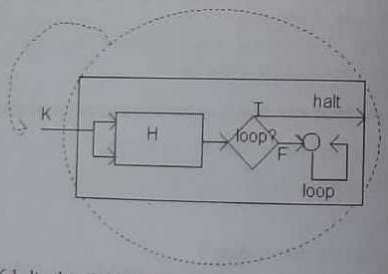
1. If H outputs "***loop***" then K halts,
2. Otherwise H's output of "***halt***" causes K to loop forever.

That is, K will do the **opposite** of H's output.

```
function K() {
  if (H()=="loop"){
    return;
  } else {
    while(true); //loop forever
  }
}
```



Since K is a program, let us use K as the input to K.



If H says that K halts then K itself would loop (that's how we constructed it). If H says that K loops then K will halt.

In either case H gives the wrong answer for K. **Thus H cannot work in all cases.**

We've shown that it is possible to construct an input that causes any solution H to fail.

Hence Proved!

1. What is Post's Corresponding Problem? What is the solution sequence, if exists, for following lists?
   a. U= (abb, a, bab, baba, aba) and V=(bbab, aa, ab, aa, a)
   b. U= (a, aa, aba, aabb, abb) and V=(b, ba, bba, bbba, bba)
2. State the Correspondence Problem given by Post? What is the solution sequence, if exists, for following lists?
   a. U= (1, 0, 010, 11) and V= (10, 10, 01, 1)
   b. U= (00, 001, 1000) and V=(0, 11, 011)
3. What does NP Completeness means? Show that SAT is NP-Complete.
4. What do you mean by decision problems? Give Turing's proof of Halting Problem as undecidable problem.

❑❑❑

LABORATORY WORKS FOR TOC

LAB 1 : Write a program to find prefixes, suffixes and substring from given string.

Code:

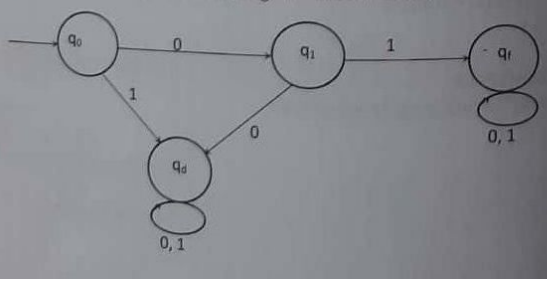
```
/* To find substring, prefix, suffix of a string */
#include<stdio.h>
#include<string.h>
void find_prefix(char string[]);
void find_suffix(char string[]);
void find_substring(char string[],int,int);

int main()
{
        char string[20];
        int i,j;
        printf("\n Enter a string\t");
        gets(string);

        printf("\n Prefixes:");
        find_prefix(string);
        printf("\n Suffixes");
        find_suffix(string);

        printf("\nEnter i and j for substring");
        scanf("%d%d",&i,&j);
        find_substring(string,i,j);

        return 0;
}

void find_prefix(char string[])
{
        int i,j;
        char prefix[20];
        for(i=strlen(string);i>=0;i--)
        {
                for( j = 0; j<i;j++)
                {
                        prefix[j]= string[j];
                }
                prefix[j]='\0';
                printf("\n %s",prefix);
        }
}
void find_suffix(char string[])
{
        int i,j,k;
        char suffix[20];
        for(i=0;i<=strlen(string);i++)
        {
                k = i;
                for( j = 0; j<strlen(string);j++)
                {
                        suffix[j]= string[k];
                        k++;
                }
                suffix[j]='\0';
                printf("\n %s",suffix);
        }
}

void find_substring(char string[],int x, int y)
{
        char substr[20];

        int k=0;
        for(int i=x-1;i<y;i++)
        {
           substr[k]=string[i];
           k++;
        }
        substr[k]='\0';
        printf("\n Substring:\n%s",substr);

}
```

OUTPUT:

```
C:\Users\c\Desktop\lab1sttring.exe

computat
computa
comput
compu
comp
com
co
c

Suffixes
computation
omputation
mputation
putation
utation
tation
ation
tion
ion
on
n

Enter i and j for substring
2  5

 Substring:
ompu
--------------------------------
Process exited after 61.09 seconds with return value 0
Press any key to continue . . .
```

LAB 2: Write program to implement following DFA's over alphabet Σ = {0, 1}.

The DFA that accepts all the strings that start with 01.



Code

```
/* Implement a DFA for L = { set of all strings over {0,1} such that string start with 01 */
#include<stdio.h>
enum states { q0, q1, qf,qd};
enum states delta(enum states, char);

int main()
{
        char input[20];
        enum states curr_state = q0;
        int i =0;

        printf("\n Enter a binary string\t");
        gets(input);
        char ch = input[i];
        while( ch !='\0')
        {
          curr_state = delta(curr_state,ch);
                ch = input[++i];
        }

        if(curr_state == qf)
          printf("\n The string %s is accepted.",input);
        else
          printf("\n The string %s is not accepted.",input);
  return 0;
}
// Transition Function
enum states delta(enum states s, char ch)
{
    enum states curr_state;
          switch(s)
                {
                        case q0:
                                if(ch=='0')
                                  curr_state = q1;
                                else
                                  curr_state = qd;
                                break;
                        case q1:
                                if(ch=='1')
                                  curr_state = qf;
                                else
```

```
                curr_state = qd;
                break;
            case qf:
                if(ch=='0')
                  curr_state = qf;
                else
                  curr_state = qf;
                break;
            case qd:
                if(ch=='0')
                  curr_state = qd;
                else
                  curr_state = qd;
                break;
        }
    return curr_state;
}
```

OUTPUT:

```
Enter a binary string  01110

The string 01110 is accepted.
--------------------------------
Process exited after 6.659 seconds with return value 0
Press any key to continue . . .
```

```
Enter a binary string  1101

The string 1101 is not accepted.
--------------------------------
Process exited after 5.392 seconds with return value 0
Press any key to continue . . .
```


LAB 3: The DFA that accepts all the strings that end with 01.

Q = { q0, q1, qf}
start state = q0,
Final state = qf
Transition function, δ is defined as:

$\delta(q0, 0) = q1$
$\delta(q0, 1) = q0$
$\delta(q1, 0) = q1$
$\delta(q1, 1) = qf$
$\delta(qf, 0) = q1$
$\delta(qf, 1) = q0$

Code:

```
// Implement a DFA for L = { set of all strings over {0,1} such that  string end with 01
#include<stdio.h>

enum states { q0, q1, qf};
enum states delta(enum states, char);

int main()
{
        char input[20];
        enum states curr_state = q0;
        int i =0;

        printf("\n Enter a binary string\t");
        gets(input);
        char ch = input[i];
        while( ch !='\0')
        {
          curr_state = delta(curr_state,ch);
                ch = input[++i];
        }

        if(curr_state == qf)
          printf("\n The string %s is accepted.",input);
        else
          printf("\n The string %s is not accepted.",input);

  return 0;
}
```

```
enum states delta(enum states s, char ch)
{
    enum states curr_state;
        switch(s)
            {
                case q0:
                    if(ch=='0')
                      curr_state = q1;
                    else
                      curr_state = q0;
                    break;
                case q1:
                    if(ch=='1')
                      curr_state = qf;
                    else
                      curr_state = q1;
                    break;
                case qf:
                    if(ch=='0')
                      curr_state = q1;
                    else
                      curr_state = q0;
                    break;
            }
        return curr_state;
}
```

OUTPUT:

```
Enter a binary string  11101

The string 11101 is accepted.
--------------------------------
Process exited after 4.533 seconds with return value 0
Press any key to continue . . .
```

```
Enter a binary string  111010

The string 111010 is not accepted.
--------------------------------
Process exited after 8.113 seconds with return value 0
Press any key to continue . . .
```


LAB 4: The DFA that accepts all the string that contains substring 001.

Q = { q0, q1, q2, qf}
start state = q0,
Final state = qf
Transition function, δ is defined as:

$\delta(q0, 0) = q1$
$\delta(q0, 1) = q0$
$\delta(q1, 0) = q2$
$\delta(q1, 1) = q0$
$\delta(q2, 0) = q2$
$\delta(q2, 1) = qf$
$\delta(qf, 0) = qf$
$\delta(qf, 1) = qf$

Code:

```
// Implement DFA that accepts strings with sub string 001 over {0,1}

#include<stdio.h>

enum states { q0,q1,q2,qf};

enum states delta(enum states, char);
int main()
{
        enum states curr_state = q0;
        char string[20], ch;
        int i=0;

        printf("\n Enter a string \t");
        gets(string);

        ch = string[i];
        while(ch!='\0')
        {
                curr_state = delta(curr_state,ch);
                ch = string[++i];
        }
        if(curr_state==qf)
          printf("\n The string %s is valid.",string);
        else
          printf("\n The string %s is not valid.",string);
        return 0;
}
```

```
enum states delta(enum states s, char ch)
{
        enum states curr_state;
        switch(s)
        {
                case q0:
                        if(ch=='0')
                          curr_state = q1;
                        else
                          curr_state = q0;
                        break;
                case q1:
                        if(ch=='0')
                          curr_state = q2;
                        else
                          curr_state = q0;
                        break;

                case q2:
                        if(ch=='0')
                          curr_state = q2;
                        else
                          curr_state = qf;
                        break;
                case qf:
                        if(ch=='0' || ch=='1')
                          curr_state = qf;
                }
                return curr_state;
}
```

OUTPUT

```
Enter a string          1100100

The string 1100100 is valid.
--------------------------------
Process exited after 9.026 seconds with return value 0
Press any key to continue . . .
```



```
Enter a string          11010

The string 11010 is not valid.
--------------------------------
Process exited after 6.237 seconds with return value 0
Press any key to continue . . .
```

LAB 5: Write a program to validate C identifiers and keywords.

C identifiers: These are the names of variables, functions, arrays, structures and pointers etc. The first character of C identifiers must be letter or underscore and remaining characters might be letters, digits or underscore.

Keywords: These are the reserved words having predefined meaning in the language. There are 32 keywords in C. They cannot be used as identifiers.

code:

```
// to identify valid identifiers and keywords in C
#include<stdio.h>
#include<string.h>
char keyword[32][10] = {"auto","double","int","struct","break","else","long",
switch","case", "enum", "register", "typedef","char", "extern", "return", "union", "const",
"float", "short", "unsigned","continue","for","signed","void","default", "goto", "sizeof",
"volatile", "do","if","static","while"} ;
enum states { q0, qf, qd};
enum states delta(enum states, char);
int iskeyword(char []);
int main()
{
        enum states curr_state = q0;
        char string[20], ch;
        int i=0;

        printf("\n Enter a string \t");
        gets(string);

        ch = string[i];
```

```
        if(iskeyword(string))
          printf("\n The string %s is keyword.",string);
        else
        {
              while(ch!='\0')
              {
                    curr_state = delta(curr_state,ch);
                    ch = string[++i];
              }
              if(curr_state==qf)
                    printf("\n The string %s is valid indentifier.",string);
              else
                    printf("\n The string %s is neither keyword nor valid
identifier.",string);
        }
        return 0;
} //end of the main

//transition function
enum states delta(enum states s, char ch)
{
        enum states curr_state;
        switch(s)
        {
              case q0:
                    if(ch>='A' && ch<='Z'||ch>='a'&&ch<='z'||ch=='_')
                      curr_state = qf;
                    else
                      curr_state = qd;
                    break;
              case qf:
                    if(ch>='A' &&
ch<='Z'||ch>='a'&&ch<='z'||ch=='_'||ch>='0'&&ch<='9')
                      curr_state = qf;
                    else
                      curr_state = qd;
                    break;
              case qd:
                    curr_state = qd;
        }
        return curr_state;
}

int iskeyword(char str[])
{
        for(int i=0;i<32;i++)
        {
              if(strcmp(str,keyword[i])==0)
                return 1;
        }
        return 0;

}
```


OUTPUT

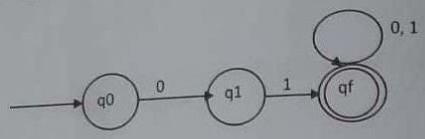
```
Enter a string          num

The string num is valid indentifier.
--------------------------------
Process exited after 10.43 seconds with return value 0
Press any key to continue . . .
```

```
Enter a string          1abc

The string 1abc is neither keyword nor valid identifier.
--------------------------------
Process exited after 8.069 seconds with return value 0
Press any key to continue . . .
```

```
Enter a string          int

The string int is keyword.
--------------------------------
Process exited after 2.508 seconds with return value 0
Press any key to continue . . .
```

LAB 6: Implement NFA over over alphabet $\Sigma = \{0, 1\}$ that accepts all the strings starting with 01.

NFA Description: The NFA has q0 as its initial state and qf as its final state.



Program Code:

```
/* TOC Lab: Implement a NFA for L = { set of all strings over {0,1} such thatstring start
with 01.
The program uses flag = 1 to indicate undefined transitions at a state.*/
#include<stdio.h>
int main()
{
    enum states { q0, q1, qf};
    char input[20];
    enum states curr_state = q0;
    inti =0;
    int flag =0;

    printf("\n Enter a binary string\t");
    gets(input);
    charch = input[i];

    while(ch !='\0')
    {
        switch(curr_state)
        {
            caseq0:
                if(ch=='0')
                    curr_state = q1;
                else
                    flag =1;
                break;
            caseq1:
                if(ch=='1')
                    curr_state = qf;
                else
                    flag=1;
                break;
            caseqf:
                if(ch=='0' || ch=='1')
                    curr_state = qf;
                break;
        }//end of the switch
        if(flag)
        break;
        ch = input[++i];
    }
    if(curr_state == qf)
    printf("\n The string %s is accepted.",input);
    else
    printf("\n The string %s is not accepted.",input);

return 0;
}
```


OUTPUT:

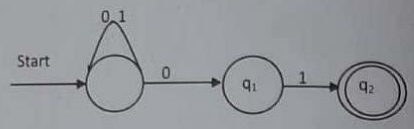
```
Enter a binary string  0111

The string 0111 is accepted.
--------------------------------
Process exited after 5.685 seconds with return value 0
Press any key to continue . . .
```

```
Enter a binary string  11011

The string 11011 is not accepted.
--------------------------------
Process exited after 6.554 seconds with return value 0
Press any key to continue . . .
```

LAB 7: Implement NFA over alphabet Σ = {0, 1} that accepts all the strings ending with 01.

NFA Description:

The NFA for the above language is given below. q0 is start state and q2 is final state.



Program Code:

```
/* Implement NFA over alphabet {0,1} that accepts all the string ending with 01.
   A state in NFA may have multiple transitions for an input symbol. Such parallel
computing can be better modelled with recursive functions. So, each transition from a
state has been defined as arecusive function. */
#include<stdio.h>
#include<string.h>

char input[20];// to store input string
int l;// to store length of the input string
int flag;// to decide accept or reject the input string

voidq2(inti)
{
        if(input[i]=='\0')
        flag =1;
}

voidq1(inti)
{
        if(i<l)
        {
                if(input[i]=='1')
                {
                        ++i;
                        q2(i);
                }
        }
}
voidq0(inti)
{
if(i<l)// i is less the length of the string
 {
int k = i;
        if(input[i]=='0')
        {
        k++;
                q0(k);
                q1(k);
        }
        else
        {
                if(input[i]=='1')
                {
                        i++;
                        q0(i);
                }
        }
}
}
int main()
{
        printf("\n Enter a string\t");
        gets(input);
        l = strlen(input);
        inti=0;
        flag=0;
        q0(i);
        if(flag==1)
        printf("\n The string is accepted.");
else
printf("\n The string is not accepted.");
        return 0;
}
```

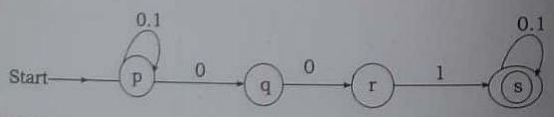
OUTPUT:

```
Enter a string 01101

The string is accepted.
--------------------------------
Process exited after 5.436 seconds with return value 0
Press any key to continue . . .
```

```
Enter a string 111011

The string is not accepted.
--------------------------------
Process exited after 4.054 seconds with return value 0
Press any key to continue . . .
```

LAB 8: Implement NFA over alphabet $\Sigma = \{0, 1\}$ that accepts all the strings containing 001 as substring.

NFA Description: The NFA has p as initial state and s as final state. This machine accepts all the strings over alphabet {0,1} containing substring 001. The state diagram is given below.



Program Code:

```
/* Implement NFA over alphabet {0,1} that accepts all the strings that contain substring
001 */

#include<stdio.h>
#include<string.h>

char input[20];
int l;
int flag;
```



```
void s(inti)
{
    flag =1;
}
void r(inti)
{
    if(i<l)
    {
        if(input[i]=='1')
        {
            ++i;
            s(i);
        }
    }
}

voidq(inti)
{
    if(i<l)
    {
        if(input[i]=='0')
        {
            ++i;
            r(i);
        }
    }
}

void p(inti)
{ if(i<l)
  {
    int k = i;
    if(input[i]=='0')
    {
        k++;
        p(k);
        q(k);
    }
```

```
        else
        {
                if(input[i]=='1')
                {
                        i++;
                        p(i);
                }
        }
        }
}

int main()
{

        printf("\n Enter a string\t");
        gets(input);
        l = strlen(input);
        inti=0;
        flag=0;
        p(i);
        if(flag==1)
                printf("\n The string is accepted.");
        else
                printf("\n The string is not accepted.");
        return 0;
}
```

OUTPUT:

```
Enter a string 1101001110

The string is accepted.
--------------------------------
Process exited after 12.23 seconds with return value 0
Press any key to continue . . .
```


LAB 9: Program to implement syntax parse tree

```
#include<stdio.h>

int main()

{
FILE *fp;
int i=0,j=0,k,l,row,col,s,x;
char a[10][10],ch,main[50],search;
fp=fopen("syntax.txt","r+");


while((ch=fgetc(fp))!=EOF)
{
if(ch=='\n')
  {
  row=i;
  col=j;
  j=0;
  i++;
  }

else
  {
  a[i][j]=ch;
  j++;
  }
}
```

```
printf("\n");
for(k=0;k<row+1;k++)
{
for(l=0;l<col;l++)
{
printf("%c",a[k][l]);
}
printf("\n");
}
i=0;
s=0;
for(k=0;k<row+1;k++)
{
    main[i]=a[k][1];
    i++;
    if(a[k][3]=='t')
    {
      search=a[k][4];
      for(l=0;l<i;l++)
      {
      if(main[l]==search)
        {
        main[i]=main[l];
        i++;
        break;
        }
```



```
    }
main[i]=a[k][5];
s=5;
i++;
}
else
{
  main[i]=a[k][3];
    //   printf("\n%c",main[i]);
  i++;
  main[i]=a[k][4];
    //   printf(",%c\n",main[i]);
  s=4;
  i++;
}

s++;

if(a[k][s]=='t')
{
s++;
search=a[k][s];

for(l=0;l<i;l++)
  {
  if(main[l]==search)
```

```
        {
        main[i]=main[l];
        i++;
        break;
        }
      }
    }
    else
    {
    main[i]=a[k][s];
    i++;
    }

}

    for(x=i-1;x>=0;x=x-4)
    {
    printf("\ntt%c: root->%c ",main[x-3],main[x-1]);
    if(main[x-2]>48 &&main[x-2]<59)
      printf("lc->t%c ",main[x-2]);
    else
      printf("lc->%c ",main[x-2]);
    if(main[x]>48 &&main[x]<59)
      printf("rc->t%c ",main[x]);
    else
      printf("rc->%c ",main[x]);
    }
return 0;
}
```


LAB 10: /* C program to implement CFG(Context Free Grammar). */

```
#include<stdio.h>
#include<string.h>

int i,j,k,l,m,n=0,o,p,nv,z=0,t,x=0;
char str[10],temp[20],temp2[20],temp3[20];

struct prod
{
  char lhs[10],rhs[10][10];
  int n;
}pro[10];

void findter()
{
  for(k=0;k<n;k++)
  {
    if(temp[i]==pro[k].lhs[0])
    {
      for(t=0;t<pro[k].n;t++)
      {
        for(l=0;l<20;l++)
          temp2[l]='\0';
        for(l=i+1;l<strlen(temp);l++)
          temp2[l-i-1]=temp[l];
        for(l=i;l<20;l++)
          temp[l]='\0';
        for(l=0;l<strlen(pro[k].rhs[t]);l++)
          temp[i+l]=pro[k].rhs[t][l];
        strcat(temp,temp2);
        if(str[i]==temp[i])
          return;
        else if(str[i]!=temp[i] && temp[i]>=65 && temp[i]<=90)
          break;
      }
      break;
    }
  }
```

```
    if(temp[i]>=65 && temp[i]<=90)
      findter();
}

int main()
{
  FILE *f;
    for(i=0;i<10;i++)
    pro[i].n=0;

  f=fopen("input.txt","r");
  while(!feof(f))
  {
    fscanf(f,"%s",pro[n].lhs);
    if(n>0)
    {
      if( strcmp(pro[n].lhs,pro[n-1].lhs) == 0 )
      {
        pro[n].lhs[0]='\0';
        fscanf(f,"%s",pro[n-1].rhs[pro[n-1].n]);
        pro[n-1].n++;
        continue;
      }
    }
    fscanf(f,"%s",pro[n].rhs[pro[n].n]);
    pro[n].n++;
    n++;
  }
  n--;

  printf("\n\nTHE GRAMMAR IS AS FOLLOWS\n\n");
  for(i=0;i<n;i++)
    for(j=0;j<pro[i].n;j++)
      printf("%s -> %s\n",pro[i].lhs,pro[i].rhs[j]);

  while(1)
  {
    for(l=0;l<10;l++)
```



```
      str[0]=NULL;

    printf("\n\nENTER ANY STRING ( 0 for EXIT ) : ");
    scanf("%s",str);
    if(str[0]=='0')
      exit(1);

    for(j=0;j<pro[0].n;j++)
    {
      for(l=0;l<20;l++)
        temp[l]=NULL;
      strcpy(temp,pro[0].rhs[j]);

      m=0;
      for(i=0;i<strlen(str);i++)
      {
        if(str[i]==temp[i])
          m++;
        else if(str[i]!=temp[i] && temp[i]>=65 && temp[i]<=90)
        {
          findter();
          if(str[i]==temp[i])
            m++;
        }
        else if( str[i]!=temp[i] && (temp[i]<65 || temp[i]>90) )
          break;
      }

      if(m==strlen(str) && strlen(str)==strlen(temp))
      {
        printf("\n\nTHE STRING can be PARSED !!!");
        break;
      }
    }
    if(j==pro[0].n)
      printf("\n\nTHE STRING can NOT be PARSED !!!");
  }
  return 0;
}
```

LAB 11: Write a program to implement PDA that accepts all strings over alphabet $\Sigma = \{0, 1\}$ that have equal number of 0s and 1s

i. by final state.

Description of PDA:

It starts at q0 and pushes $ into the empty stack and switches to new state q1. At q1, PDA pushes input symbol if stack top symbol is $ or stack top is same as input symbol, otherwise if input is 0 and stack top is 1 or input is 1 and stack top is 0, stack top is popped off. If input is finished and stack top is $ at state q1, PDA switches to final state, qf. The figure for the PDA is given below.

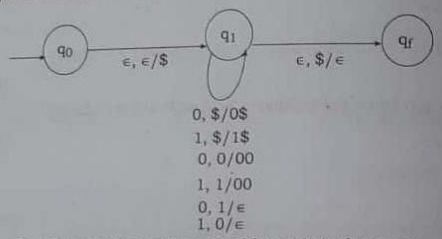


Fig. PDA accepting equal number of 0s and 1s by final state

LAB 12: /* TOC Lab : Implement a PDA for L = { set of all strings over {0,1} such that equal number of 0s and 1s , acceptance by final state */

```
#include<stdio.h>
#include<string.h>
#define MAX 100
enum states { q0, q1,qf};
void push(char ch);
void pop();
char get_stack_top();
enum states delta(enum states, char, char);

struct stack
{
        char symbols[MAX];
        int top;
};
struct stack s;

int main()
{
        char input[20];
        enum states curr_state = q0;
        s.top = -1;
        int i =0;
        char ch = 'e';   // e indicating epsilon
```



```
        char st_top = 'e';
        curr_state = delta(curr_state,ch,st_top);

        printf("\n Enter a binary string\t");
        gets(input);

        ch = input[i];
        st_top = get_stack_top();
        int c=0;

        while( c <=strlen(input))
        {
                curr_state = delta(curr_state,ch,st_top);
                ch = input[++i];
                st_top=get_stack_top();
                c++;

        }
        if(curr_state == qf)
          printf("\n The string %s is accepted.",input);
        else
          printf("\n The string %s is not accepted.",input);

 return 0;
}
enum states delta(enum states s, char ch, char st_top)
{
        enum states curr_state;
                switch(s)
                {
                        case q0:
                                if(ch=='e' && st_top=='e')
                                {
                                        curr_state = q1;
                                        push('$');          // $ is stack bottom marker
                                }
                                break;
                        case q1:
                                if(ch=='0' && (st_top=='$' ||st_top=='0'))
                                {
                                        curr_state = q1;
                                        push(ch);
                                }
                                else if(ch=='1' && (st_top=='$'||st_top=='1'))
                                {
                                        curr_state = q1;
                                        push(ch);
                                }
                                else if(ch=='1' && st_top=='0'||ch=='0'&&st_top=='1')
```

```
                        {
                                curr_state = q1;
                                pop();
                        }
                        else if(ch=='\0' && st_top=='$')
                        {
                                curr_state = qf;
                                pop();
                        }
                        break;
                }
        return curr_state;
}

//function to get stack top symbol
char get_stack_top()
{
        return (s.symbols[s.top]);
}

//push function
void push(char ch)
{
        if(s.top<MAX-1 )
         {
                s.symbols[++s.top] = ch;
         }
        else
        {
                printf("\n Stack Full.");
        }
}

//pop function
void pop()
{
        if(s.top>-1)
        {
                s.symbols[s.top]=' ';
                s.top--;
        }
        else
        printf("\n Stack Empty.");
}
```


OUTPUT:

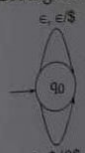
```
Enter a binary string  001101

The string 001101 is accepted.
--------------------------------
Process exited after 23.09 seconds with return value 0
Press any key to continue . . .
```

```
Enter a binary string  01110

The string 01110 is not accepted.
--------------------------------
Process exited after 4.202 seconds with return value 0
Press any key to continue . . .
```

LAB 13: PDA accepting equal number of 0s and 1s with empty stack

PDA description:

It starts at q0 and pushes $ into the empty stack. At the same state q0, PDA pushes input symbol if stack top symbol is $ or stack top is same as input symbol, otherwise if input is 0 and stack top is 1 or input is 1 and stack top is 0, stack top is popped off. If input is finished and stack top is $, PDA accepts the input string. The figure for the PDA is given below.



0, $/0$
0, 0/00
1, 1/11
0, 1/ϵ
0, 0/ϵ
ϵ, $/ϵ

Code for PDA Accepting by Empty Stack

```
/* TOC Lab : Implement a PDA for L = { set of all strings over {0,1} such that equal
number of 0s and 1s , acceptance by empty stack */
```

```
#include<stdio.h>
#include<string.h>
#define MAX 100

enum states { q0};
void push(char ch);
void pop();
char get_stack_top();
enum states delta(enum states, char, char);

struct stack
{
        char symbols[MAX];
        int top;
};

struct stack s;

int main()
{
        char input[20];
        enum states curr_state = q0;
        s.top = -1;
        int i =0;
        char ch = 'e';
        char st_top = 'e';
        curr_state = delta(curr_state,ch,st_top);

        printf("\n Enter a binary string\t");
        gets(input);

        ch = input[i];
        st_top = get_stack_top();
        int c=0;

        while( c <=strlen(input))
        {
                curr_state = delta(curr_state,ch,st_top);
                ch = input[++i];
                st_top=get_stack_top();
                c++;
    }
        if(s.symbols[s.top] == '$')
```



```
        printf("\n The string %s is accepted.",input);
    else
        printf("\n The string %s is not accepted.",input);

 return 0;
}

enum states delta(enum states s, char ch, char st_top)
{
        enum states curr_state;
            switch(s)
            {
                case q0:
                    if(ch=='e' && st_top=='e')
                    {
                        curr_state = q0;
                        push('$');
                    }
                  else  if(ch=='0' && (st_top=='$' ||st_top=='0'))
                    {
                        curr_state = q0;
                        push(ch);
                    }
                 else if(ch=='1' && (st_top=='$' ||st_top=='1'))
                    {
                        curr_state = q0;
                        push(ch);
                    }
                else if(ch=='1' && st_top=='0' ||ch=='0'&&st_top=='1')
                    {
                        curr_state = q0;
                        pop();
                    }
                else if(ch=='\0' && st_top=='$')
                    {
                        curr_state = q0;
                    //pop();
                    }
                    break;
                }
    return curr_state;
}
```

```
char get_stack_top()
{
        return (s.symbols[s.top]);
}

void push(char ch)
{
        if(s.top<MAX-1 )
        {
                s.symbols[++s.top] = ch;
        }
        else
        {
                printf("\n Stack Full.");
        }
}

void pop()
{
        if(s.top>-1)
        {
                s.symbols[s.top]=' ';
                s.top--;
        }
        else
        printf("\n Stack Empty.");
}
```

OUTPUT

```
Enter a binary string  011010

The string 011010 is accepted.
--------------------------------
Process exited after 8.807 seconds with return value 0
Press any key to continue . . .
```

```
Enter a binary string  00101

The string 00101 is not accepted.
--------------------------------
Process exited after 9.817 seconds with return value 0
Press any key to continue . . .
```


LAB 14: Write a program to implement PDA that accepts all strings over alphabet $\Sigma = \{0, 1\}$ that have contain number of 0's followed by equal number of 1s by final state.

PDA Description:

The PDA starts at q0 state. At this state, it simply pushes $ into the empty stack and switches to next state q1. At state q1, PDA pushes input 0 seen on the top of $ or 0 on the stack and if it sees 1 when stack top is 0, it pops the stack top switching to next state q2. At q2, it pops 0's for each 1 seen on the input. If no input symbol is present and $ on the top of stack, PDA switches to final state qf.

Code:

```
/* TOC Lab : Implement a PDA for L = { set of all strings over {0,1} such that  0^n1^n,
acceptance by final state */
#include<stdio.h>
#include<string.h>
#define MAX 100

enum states { q0, q1,q2,qf,qr};
void push(char ch);
void pop();
char get_stack_top();
enum states delta(enum states, char, char);

struct stack
{
        char symbols[MAX];
        int top;
};

struct stack s;

int main()
{
        char input[20];
        enum states curr_state = q0;
        s.top = -1;
        int i =0;
        char ch = 'e';
        char st_top = 'e';
        curr_state = delta(curr_state,ch,st_top);

        printf("\n Enter a binary string\t");
```

```
        gets(input);

        ch = input[i];
        st_top = get_stack_top();
        int c=0;

        while( c <=strlen(input))
        {
                curr_state = delta(curr_state,ch,st_top);
                ch = input[++i];
                st_top=get_stack_top();
                c++;
    }
        if(curr_state==qf)
          printf("\n The string %s is accepted.",input);
        else
          printf("\n The string %s is not accepted.",input);

  return 0;
}

enum states delta(enum states s, char ch, char st_top)
{
        enum states curr_state = qr;
                switch(s)
                {
                        case q0:
                                if(ch=='e' && st_top=='e')
                                {
                                        curr_state = q1;
                                        push('$');
                                }
                                break;
                        case q1:
                                if(ch=='0' && (st_top=='$' ||st_top=='0'))
                                {
                                        curr_state = q1;
                                        push(ch);
                                }
                                else if(ch=='1' && st_top=='0')
                                {
                                        curr_state = q2;
                                        pop();
```

```
                }
                else
                curr_state = qr;  // qr for undefined transition
            break;
        case q2:
                if(ch=='1' && st_top=='0')
                {
                        curr_state = q2;
                        pop();
                }
                else if(ch=='\0' and st_top=='$')
                {
                        curr_state = qf;
                        pop();
                }
                else
                 curr_state = qr;


                break;
        }
    return curr_state;
}


char get_stack_top()
{
    return (s.symbols[s.top]);
}

void push(char ch)
{
    if(s.top<MAX-1 )
     {
          s.symbols[++s.top] = ch;
     }
    else
     {
          printf("\n Stack Full.");
     }
}
```

```
void pop()
{
        if(s.top>-1)
        {
                s.symbols[s.top]=' ';
                s.top--;
        }
        else
        printf("\n Stack Empty.");
}
```

Output:

```
Select C:\Users\c\Downloads\pda0n1nbyfinal.exe

Enter a binary string  000111

The string 000111 is accepted.

--------------------------------
Process exited after 4.313 seconds with return value 0
Press any key to continue . . .
```

```
C:\Users\c\Downloads\pda0n1nbyfinal.exe

Enter a binary string  00011

The string 00011 is not accepted.

--------------------------------
Process exited after 8.545 seconds with return value 0
Press any key to continue . . .
```

LAB 15: Lab: Implement the TM accepting the language $\{0^n1^n / n \geq 1\}$ over alphabet, $\Sigma = \{0, 1\}$.

Turing Machine Description:

Given finite sequence of 0's and 1's on tape and followed by blanks. The TM starts at state q0 and changes 0 to an X and moves to the right changing its state to q1.

At stateq1, TM expects 1 and changes a 1 to Y and moves to the left changing the state to q2. If any number of 0s and Ys are seen, it remains on the state q1 and leaving these symbols unchanged and moving the head position to the right.

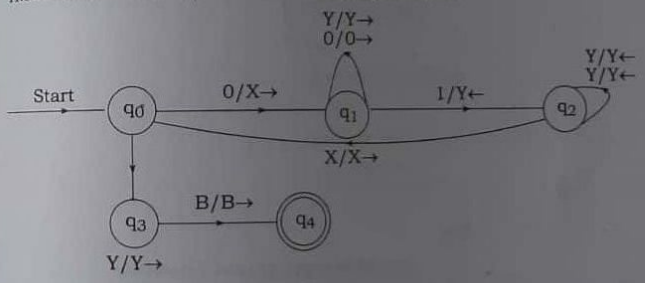
At state q2, if 0s or Y's are seen, it leaves them as it is and moves to the left staying at the same state q2. If it sees X at state q2, the tape symbol is left unchanged moves to right switching its state to q0.

At state q0, if it sees Y then the symbol is left unchanged and head is moved right changing to the state q3.

At state q3, if Y is seen, it is left unchanged and head is moved to the right. If BLANK (here, '\0') is seen at state q3, the string is accepted by switching the state to q4.

At any state, if the machine seen other than the defined symbols, it rejects the string.

The state transition diagram is shown in the figure below.



Code:

```
/* TOC Lab: Implement a TM for L = { set of all strings over {0,1} such thatthe string
have number of 0s followed by same number of 1s. */
#include<stdio.h>
enum states { q0, q1, q2,q3,q4,qr};
int main()
{
        char input[100];
        enum states curr_state = q0;
        inti;
        for(i=0;i<100;i++)
        input[i] = '\0';
```

```
printf("\n Enter a binary string\t");
gets(input);
i=0;
while(1)
{
        switch(curr_state)
        {
                caseq0:
                        if(input[i]=='0')
                        {
                                curr_state = q1;
                                input[i]='x';
                                i++;
                        }
                        else if(input[i]=='y')
                        {
                                curr_state=q3;
                                i++;
                        }
                        else
                                curr_state = qr;// for invalid transition
                        break;
                caseq1:
                        if(input[i]=='0')
                        {
                                curr_state = q1;
                                i++;
                        }
                        else if(input[i]=='y')
                        {
                                curr_state = q1;
                                i++;
                        }
```

```
        else if(input[i]=='1')
        {
                curr_state = q2;
                input[i]='y';
                i--;
        }
        else
                curr_state = qr;
        break;
caseq2:
        if(input[i]=='0')
        {
                curr_state = q2;
                i--;
        }
        else if(input[i]=='y')
        {
                curr_state = q2;
                i--;
        }
        else if(input[i]=='x')
        {
                curr_state = q0;
                i++;
        }
        else
                curr_state = qr;
        break;
caseq3:
        if(input[i]=='y')
        {
                curr_state = q3;
                i++;
```

```
                }
                else if(input[i]=='\0')
                {
                        curr_state=q4;
                }
                else
                        curr_state = qr;
                break;
        }//end of switch
        if(curr_state == qr ||curr_state==q4)
        break;
    }//end of while loop

if(curr_state == q4)
        printf("\n The string is accepted.");
else
        printf("\n The string is not accepted.");
return 0;
}
```

OUTPUT

```
Enter a binary string  000111

The string is accepted.
--------------------------------
Process exited after 7.121 seconds with return value 0
Press any key to continue . . .
```

```
Enter a binary string  00110

The string is not accepted.
--------------------------------
Process exited after 4.432 seconds with return value 0
Press any key to continue . . .
```