IRUNGATTUKOTTAI, SRIPERUMBUDUR, CHENNAI – 602 117

DEPARTMENT OF INFORMATION TECHNOLOGY

STUDY MATERIAL

CS3452 THEORY OF COMPUTATION

REG. NUMBER :

NAME :

YEAR / SEM :

CS3452 THEORY OF COMPUTATION L T P C 3 0 0 3

COURSE OBJECTIVES:

- To understand the foundations of computation including automata theory
- To construct models of regular expressions and languages.
- To design context-free grammar and push down automata
- To understand Turing machines and their capability
- To understand Undecidability and NP class problems

UNIT I AUTOMATA AND REGULAR EXPRESSIONS 9

Need for automata theory - Introduction to formal proof – Finite Automata (FA) – Deterministic Finite Automata (DFA) – Non-deterministic Finite Automata (NFA) – Equivalence between NFA and DFA –Finite Automata with Epsilon transitions – Equivalence of NFA and DFA- Equivalence of NFAs with and without ε-moves- Conversion of NFA into DFA – Minimization of DFAs.

UNIT II REGULAR EXPRESSIONS AND LANGUAGES 9

Regular expression – Regular Languages- Equivalence of Finite Automata and regular expressions – Proving languages to be not regular (Pumping Lemma) – Closure properties of regular languages.

UNIT III CONTEXT FREE GRAMMAR AND PUSH DOWN AUTOMATA 9

Types of Grammar - Chomsky‘s hierarchy of languages -Context-Free Grammar (CFG) and Languages – Derivations and Parse trees – Ambiguity in grammars and languages – Push Down Automata (PDA): Definition – Moves - Instantaneous descriptions -Languages of pushdown automata – Equivalence of pushdown automata and CFG-CFG to PDA-PDA to CFG – Deterministic Pushdown Automata.

UNIT IV NORMAL FORMS AND TURING MACHINES 9

Normal forms for CFG – Simplification of CFG- Chomsky Normal Form (CNF) and Greibach Normal Form (GNF) – Pumping lemma for CFL – Closure properties of Context Free Languages –Turing Machine : Basic model – definition and representation – Instantaneous Description – Language acceptance by TM – TM as Computer of Integer functions – Programming techniques for Turing machines (subroutines).

UNIT V UNDECIDABILITY 9

Unsolvable Problems and Computable Functions –PCP-MPCP- Recursive and recursively enumerable languages – Properties - Universal Turing machine -Tractable and Intractable problems - P and NP completeness – Kruskal’s algorithm – Travelling Salesman Problem- 3-CNF SAT problems.

COURSE OUTCOMES:

At the end of this course, the students will be able to:

CO1: Construct automata theory using Finite Automata

CO2: Write regular expressions for any pattern

CO3: Design context free grammar and Pushdown Automata

CO4: Design Turing machine for computational functions

CO5: Differentiate between decidable and undecidable problems

TOTAL:45 PERIODS

TEXT BOOKS:

1. Hopcroft J.E., Motwani R. & Ullman J.D., "Introduction to Automata Theory, Languages and Computations", 3rd Edition, Pearson Education, 2008.

2. John C Martin , "Introduction to Languages and the Theory of Computation", 4th Edition, Tata McGraw Hill, 2011.

REFERENCES

1. Harry R Lewis and Christos H Papadimitriou , "Elements of the Theory of Computation", 2nd Edition, Prentice Hall of India, 2015.

2. Peter Linz, "An Introduction to Formal Language and Automata", 6th Edition, Jones & Bartlett, 2016.

3. K.L.P.Mishra and N.Chandrasekaran, “Theory of Computer Science: Automata Languages and Computation”, 3rd Edition, Prentice Hall of India, 2006.

CS3452 THEORY OF COMPUTATION

UNIT I

AUTOMATA AND REGULAR EXPRESSIONS

Need for automata theory - Introduction to formal proof – Finite Automata (FA) – Deterministic Finite Automata (DFA) – Non-deterministic Finite Automata (NFA) – Equivalence between NFA and DFA –Finite Automata with Epsilon transitions – Equivalence of NFA and DFA- Equivalence of NFAs with and without ε-moves- Conversion of NFA into DFA – Minimization of DFAs.

INTRODUCTION TO THEORY OF COMPUTATION

- The theory of computation describes the basic ideas and models underlying computing.
- Computation involves taking some inputs and performing the required operations on it, using syntactic procedures and algorithms and producing the outputs accordingly.
- The term "Theory of Computation" suggests various abstract models of computation, represented mathematically.
- Some of these models are as powerful as real computers.
- Each abstract computing machine recognizes a formal language at a particular complexity.
- The simplest type of abstract machine is, Finite Automata (FA) or Finite State Machine, which recognizes only the regular languages and used to solve decision making problems.
- Software implementation of the corresponding Finite Automaton is used to construct lexical analyzer of a compiler and text editors, pattern or language recognizer.

BASIC DEFINITIONS OF AUTOMATA THEORY

1. **Alphabets:**

   An alphabet is a finite, non empty set of symbols. It is denoted by 'Σ'. It includes,

   Σ = {0, 1} ; the binary alphabet.

   Σ = {a, b, c, ..z} ; the set of all lower case letters.

2. **Strings:**

   A string is a finite sequence of symbols chosen from some alphabet.

For ex:

Given an alphabet Σ = {a, b}, a string formed from Σ is "aba, ba, aab, bba, abab,……"

3. **The Empty String:**

   The empty string is the string with zero occurrences of symbols. It is denoted as 'ε'.

4. **Length of a string:**

   Length of a string is number of positions for symbols in a string. The standard notation for the length of a string 'w' is |w|. For ex:

   | 011 | = 3

   | ε | = 0

5. **Powers of an Alphabet:**

   If 'Σ' is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an exponential notation. We define Σ^k to be the set of strings of length k, each of whose symbols is in Σ.

For ex:

- $\Sigma^0 = \{ \varepsilon \}$
- If $\Sigma = \{0, 1\}$ then,

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

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

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

6. **Concatenation of Strings:**

Let 'x' and 'y' be strings, then 'xy' denotes the concatenation of 'x' and 'y' that is, the string formed by making a copy of 'x' and following it by a copy of 'y'.

For ex:

Let x = 01101, y = 110

xy = 01101110

yx = 11001101

7. **Languages:**

A set of strings all of which are chosen from some Σ^*, where Σ is a particular alphabet is called a language. If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a language over Σ.

INTRODUCTION TO FORMAL PROOF

The automata theory lends itself to natural and interesting proofs, both of the **deductive kind** and the **inductive kind**.

1. **Deductive Proofs:**

A deductive proof consist of a sequence of statements whose truth leads us from some 'initial statement' called 'Hypothesis' or the 'Given statement' to a conclusion statement.

Format: A theorem that is proved when we go from a 'Hypothesis H to a Conclusion C' is a statement "If H then C". We say that C is deduced from H.

Theorem: If $x \geq 4$ then $2^x \geq x^2$.

Proof:

First notice that the hypothesis H is "$x \geq 4$". This hypothesis has a parameter x, and thus is neither true or false. Rather, its truth depends on the value of the parameter x.

Ex: H is true for x = 6 and false for x = 2.

The conclusion C is "$2^x \geq x^2$". This statement also uses parameter x and is true for certain values of x and not others.

Ex: (i) C is false for x = 3.

ie) $2^3 \geq 3^2 \Rightarrow 8 \geq 9 \Rightarrow$ false

(ii) C is true for x = 4.

ie) $2^4 \geq 4^2 \Rightarrow 16 \geq 16 \Rightarrow$ true.

We take x = 5

$2^5 \geq 5^2 \Rightarrow 32 \geq 25 \Rightarrow$ true.

The statement "If $x \geq 4$ then $2^x \geq x^2$", for all integers x. In fact, we do not need to assume x is an integer, but the proof talked about repeatedly increasing x by 1, stating at x = 4. So we really addressed only the situation where x is an integer.

Finally the conclusion is, $2^x \geq x^2$ will be true whenever $x \geq 4$.

2. **Reduction to Definitions:**

In many theorems of Automata theory, the terms used in the statement may have implications that are less obvious. A useful way to proceed in many proof is, "If you are not sure how to start a proof", convert all terms in the hypothesis to their definitions.

Theorem:

Let S be a finite subset of some infinite set U. Let T be the complement of S with respect to U. Then T is infinite.

Proof:

This theorem uses two definitions,

(a) A set S is finite if there exists an integer n such that S has exactly n elements. We write ||S|| = n, where ||S|| is used to denote the number of elements in a set S. If the set S is not finite, we say S is infinite. An infinite set is a set that contains more than any integer number of elements.

(b) If S and T are both subsets of some set U, then T is the complement of S(with respect to U) if S $\cup$ T = U and S $\cap$ T = $\emptyset$ $\emptyset$. That is, each element of U is in exactly one of S and T. Put another way, T consists of exactly those elements of U that are not in S.

Original Statement	**New Statement**
S is finite	There is a integer *n* such that ||S|| = *n*
U is infinite	For no integer *p* is ||U|| = *p*
T is the complement of S	S $\cup$ T = U and S $\cap$ T = $\emptyset$ $\emptyset$

Fig. 1.1. Restarting the givens of theorem

3. **Other Theorem Forms:**

The "if-then" form of theorem is most common. There are some other forms of statements proved as theorem also.

(i) **Way of Saying "If-then"**

"If H then C' may appear as,

(a) H implies C.
(b) H only if C.
(c) C if H.
(d) Whenever H holds, C follows.

(ii) **If-And-Only-If Statements:**

A statement of the form "A if and only if B". Other forms of this statement are,

(a) A iff B
(b) A is equivalent to B Or
A exactly when B

In formal logic, may see the operator $\leftrightarrow$ or $\equiv$ to denote an "if-and-only-if" statement.

(ie) A $\equiv$ B and A $\leftrightarrow$ B mean the same as "A if-and-only-if B".

To prove "A if-and-only-if B" statement, First prove "A if-and-only-if C", and then prove "C if-and-only-if B". Each if-and-only-if step must be proved in both directions.

4. **Theorems that appear not to be If-Then Statements:**

In some cases, we encounter a theorems that appears not to have hypothesis.

Ex: $\text{Sin}^2\theta\,\theta + \cos^2\theta\,\theta = 1$

From the definitions and the functions sin and cos values we can prove the theorem.

ADDITIONAL FORMS OF PROOFS

Several additional forms of proofs are there,

(1) Proofs about sets
(2) Proofs by contradiction
(3) Proofs by counterexample

(1) **Proofs about sets:**

In automata theory, set theory plays an important role. If E and F are two expressions representing sets, the statement E = F means that the two sets represented are the same. More precisely, every element in the set represented by E is in the set represented by F, and every element in the set represented by F is in the set represented by E.

Ex:

Commutative law of union says, for two sets R and S.

ie) $R \cup\cup S = S \cup\cup R$

Here, $R \cup\cup S = E$

$S \cup\cup R = F$

$\therefore\therefore$ The commutative law of union says that E = F.

Theorem:

$R \cup\cup (S \cap\cap T) = (R \cup\cup S) \cap\cap (R \cup\cup T)$

Proof:

The two set-expressions involved are $E = R \cup\cup (S \cap\cap T)$ and $F = (R \cup\cup S) \cap\cap (R \cup\cup T)$. In this theorem to prove two parts, that is,

(i) If part
(ii) Only-if part

(i) **To prove if part:**

We assume element 'x' is in E and show it is in F.

Take L.H.S. $R \cup\cup (S \cap\cap T)$ and show it is in $(R \cup\cup S) \cap\cap (R \cup\cup T)$.

$x \in\in R \cup\cup (S \cap\cap T)$

$x \in\in R$ or $x \in\in (S \cap\cap T)$

$x \in\in R$ or $x \in\in S$ and $x \in\in R$ or $x \in\in T$ [$\therefore\therefore$ x is in R or x is in both S and T]

x $\in\in$ R $\cup\cup$ S and x $\in\in$ R $\cup\cup$ T

x $\in\in$ (R $\cup\cup$ S) $\cap\cap$ (R $\cup\cup$ T)

(ii) **Only-if part:**

Here we assume 'x' is in F and show it is in E.

Take R.H.S. (R $\cup\cup$ S) $\cap\cap$ (R $\cup\cup$ T) and show it is in R $\cup\cup$ (S $\cap\cap$ T).

x $\in\in$ (R $\cup\cup$ S) $\cap\cap$ (R $\cup\cup$ T)

x $\in\in$ R $\cup\cup$ S and x $\in\in$ R $\cup\cup$ T

x $\in\in$ R or x $\in\in$ S and x $\in\in$ R and x $\in\in$ T

x $\in\in$ R or x $\in\in$ S and x $\in\in$ T

x $\in\in$ R or x $\in\in$ S $\cap\cap$ T

x $\in\in$ R $\cup\cup$ (S $\cap\cap$ T)

2. Proofs by Contradiction:

Another way to prove a statement of the form "if H then C" is, "H and not C implies falsehood". By assuming both the hypothesis H and the negation of the conclusion C. Complete the proof by showing that something known to be false follows logically from H and C. This form of Proof is called Proof by Contradiction.

Ex:

Hypothesis H = U is an infinite set, S is a finite subset of U, and T is the complement of S with respect to U. The conclusion C was "T is infinite". But we assumed T was finite. It is contradiction.

3. Proofs by Counterexample:

In real life, to resolve the question, we may alternately try to prove the theorem, and if we cannot try to prove that its statement is false. But it is easier to prove that a statement is not a theorem than to prove it is a theorem.

Alleged Theorem: All primes are odd.

Disproof: The integer 2 is a prime, but 2 is even.

The Contrapositive :

The contrapositive of the statement "if H then C" is "if not C then not H". A statement and its contrapositive are either both true or both false. So we can prove either to prove the other.

Ex:

If $x \geq 4$ then $2^x \geq x^2$.

The contrapositive of this statement is "if not $2^x \geq x^2$ then not $x \geq 4$.

<u>INDUCTIVE PROOFS</u>

There is a special form of proof called "inductive", which is essential when dealing with recursively defined objects. Many of the inductive proofs deal with integers, trees, expressions of various sorts etc.

Inductions on Integers:

Suppose we are given a statement S(n), about an integer n, to prove. One common approach is to prove two things,

(1) The **basis**, where we show S(i) for a particular integer 'i'. Usually i=0 or i=1.

(2) The **inductive**, where we assume n ≥ i, where 'i' is the basis integer, and we show that "if S(n) then S(n+1)".

Theorem:

For all n ≥ 0. $\sum_{i=1}^{n} i \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \frac{n(n+1)(2n+1)}{6}$

Proof:

The proof is in two parts, the basis and the inductive step, we prove each in turn.

Basis:

Let n = 0,

First, To prove, L.H.S. $\sum_{i=1}^{n} i \sum_{i=1}^{n} i^2$

L.H.S. $\Rightarrow \sum_{i=1}^{0} i \sum_{i=1}^{0} i^2 = 0$

To prove, R.H.S, $\frac{n(n+1)(2n+1)}{6} \frac{n(n+1)(2n+1)}{6}$

R.H.S. $\Rightarrow \frac{0(0+1)(2*0+1)}{6} \frac{0(0+1)(2*0+1)}{6} = 0$

∴∴ L.H.S = R.H.S

Thus the equation is true for n = 0.

Induction:

Assume that, it is true for n. To prove it is true for n+1.

To prove L.H.S. $\sum_{i=1}^{n} i \sum_{i=1}^{n} i^2$, assume n= n+1

L.H.S. $\Rightarrow (\sum_{i=1}^{n} i \ (\sum_{i=1}^{n} i^2) + (n+1)^2$

$\Rightarrow \frac{n(n+1)(2n+1)}{6} \frac{n(n+1)(2n+1)}{6} + (n+1)^2$

$\Rightarrow \frac{(n^2+n)(2n+1)}{6} \frac{(n^2+n)(2n+1)}{6} + n^2+2n+1$

$\Rightarrow \frac{2n^3+n^2+2n^2+n}{6} \frac{2n^3+n^2+2n^2+n}{6} + n^2+2n+1$

$\Rightarrow \frac{2n^3+3n^2+n}{6} \frac{2n^3+3n^2+n}{6} + n^2+2n+1$

$\Rightarrow \frac{2n^3+3n^2+n+6n^2+12n+6}{6} \frac{2n^3+3n^2+n+6n^2+12n+6}{6}$

$\sum_{i=1}^{n} i \sum_{i=1}^{n} i^2$ $\Rightarrow \frac{2n^3+9n^2+13n+6}{6} \frac{2n^3+9n^2+13n+6}{6}$

To prove, R.H.S. for n = n+1,

R.H.S. $\Rightarrow \frac{n(n+1)(2n+1)}{6} \frac{n(n+1)(2n+1)}{6}$

$$\Rightarrow \frac{(n+1)(n+1+1)(2(n+1)+1)}{6} \quad \frac{(n+1)(n+1+1)(2(n+1)+1)}{6}$$

$$\Rightarrow \frac{(n+1)(n+2)(2n+3)}{6} \quad \frac{(n+1)(n+2)(2n+3)}{6}$$

$$\Rightarrow \frac{(n^2+2n+n+2)(2n+3)}{6} \quad \frac{(n^2+2n+n+2)(2n+3)}{6}$$

$$\Rightarrow \frac{2n^3+3n^2+4n^2+6n+2n^2+3n+4n+6}{6} \quad \frac{2n^3+3n^2+4n^2+6n+2n^2+3n+4n+6}{6}$$

$$\Rightarrow \frac{2n^3+9n^2+13n+6}{6} \quad \frac{2n^3+9n^2+13n+6}{6}$$

∴∴ L.H.S = R.H.S

ie) $\sum_{i=1}^{n} i \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \quad \frac{n(n+1)(2n+1)}{6}$

∴∴ The given theorem is proved.

Structural Inductions:

The examples of structural inductions are, trees and expressions.

Ex: Here is the recursive definition of a tree.

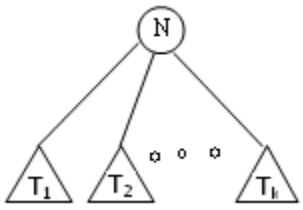
Basis:

A single node is a tree, and that node is the root of the tree.

Induction:

If T_1, T_2,…T_k are trees, then we can form a new tree as follows,

1. Begin with a new node, N which is the root of the tree.
2. Add copies of all the trees T_1, T_2,…T_k.
3. Add edges from node N to the roots of each of the trees T_1, T_2,…T_k.



Mutual Inductions:

To prove a group of statements is no different from proving the **conjunction** (logical AND) of all the statements. For instance the group of statements $S_1(n)$, $S_2(n)$,…..$S_k(n)$ could be replaced by the single statement $S_1(n)$ AND $S_2(n)$ AND…..AND $S_k(n)$.

However when there are really several independent statements to prove, it is generally less confusing to keep the statements separate and to prove them all in their own parts of the basis and inductive steps. We call this sort of proof is **mutual induction**.

FINITE AUTOMATA(FA)

A finite automata has a finite set of states and its control moves from one state to another in response to external inputs. There are two types of finite automata; They are,

1. Deterministic Finite Automata(DFA)
2. Non-deterministic Finite Automata(NFA)

1. **Deterministic Finite Automata:**
   The automaton cannot be in more than one state at any one time.
2. **Non-deterministic Finite Automata:**
   In NFA is used to several states at once.

Formal Definition of Finite Automata:

A Finite Automata can be defined as 5 – tuples.

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

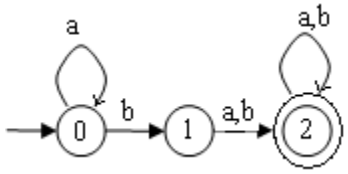
Where,

- Q - None empty, finite set of states.
- Σ - Finite set of symbols.
- δ - Transition function.
- q_0 - Initial state.
- F - Finite set of final states.

DETERMINISTIC FINITE AUTOMATA (DFA)

DFA refers the fact that on each input there is one and only one state to which the automaton can transition from its current state.

Ex:



Definition of a DFA:

A DFA consists of 5 – tuples.

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

Simpler Notations for DFA's:

There are two preferred notations for describing automata,

(1) A Transition Diagram: Which is a graph.
(2) A Transition Table: Which is a tabular listing of the δ function, which by implication tells us the set of states and the input alphabet.

PROBLEMS:

Ex. 1:

Construct the transition diagram and the transition table for the following language,

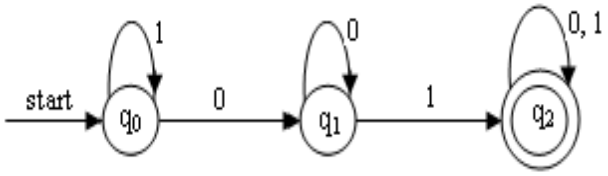
L = { x01y | x and y are any strings of 0's and 1's }

Soln:

The language include the strings are { 01, 100011, 11010, 101001,…..}

The strings not in the language are { 0, 1, 111000, 00, 11,….. }

Transition diagram:



Transition table:

State	Input	
	0	1
→ q_0	q_1	q_0
q_1	q_1	q_2
* q_2	q_2	q_2

Theorem: Extending the transition function to strings on DFA:

Extended transition function describes, when we start in any state and follow any sequence of inputs. If δ is our transition function, then the extended transition function constructed from δ will be called $\hat{\delta}$.

δ consists of a state & a single symbol.

$\hat{\delta}$ consists of a state & a string.

Proof: To compute $\hat{\delta}(q, w) = r$

Basis:

$\hat{\delta}(q, \varepsilon) = q$, That is, if we are in state q and read no inputs, then we are still in state q.

Induction:

Suppose 'w' is a string of the form 'xa' that is,

a - is the last symbol of 'w'.

x - is the string consisting of all but except the last symbol.

For ex.,

w = 1101, is broken into xa,

x = 110 and a = 1

To compute $\hat{\delta}(q, w)$,

First compute $\hat{\delta}(q, x)$, it states that the automaton is in after processing all but the last symbol of 'w'. Suppose this state is 'p', that is; $\hat{\delta}(q, x) = p$.

ie) $\hat{\delta}(q, w) = \hat{\delta}(q, xa)$ [∴∴ w = xa]

$= \delta(\hat{\delta}(q, x), a)$

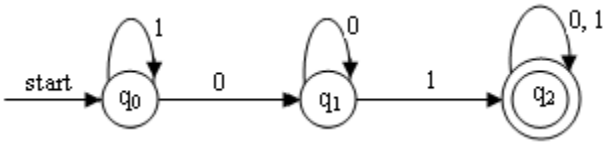
$= \delta(p, a)$ [∴∴ $\hat{\delta}(q, x) = p$]

$\hat{\delta}(q, w) = r$ [$\therefore\therefore$ $\delta(p, a) = r$]

PROBLEMS:

Ex. 1:

Consider the following DFA, check whether the input strings w1 = 1011 and w2 = 1100 is accepted by the finite automata or not?



Soln:

- w1 = 1011

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

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

$\hat{\delta}(q_0, 10) = \delta(\hat{\delta}(q_0, 1), 0) = \delta(q_0, 0) = q_1$

$\hat{\delta}(q_0, 101) = \delta(\hat{\delta}(q_0, 10), 1) = \delta(q_1, 1) = q_2$

$\hat{\delta}(q_0, 1011) = \delta(\hat{\delta}(q_0, 101), 1) = \delta(q_2, 1) = q_2$

Here answer is q_2, that is the accepting state,

$\therefore\therefore$ w1 = 1011 is **accepted** by finite automata.

- w2 = 1100

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

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

$\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_0, 1) = q_0$

$\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_1$

$\hat{\delta}(q_0, 1100) = \delta(\hat{\delta}(q_0, 110), 0) = \delta(q_1, 0) = q_1$

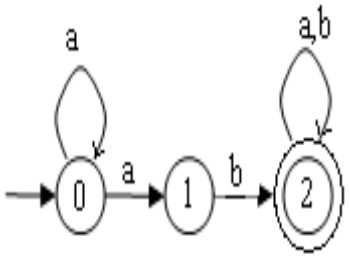
Here answer is q_1, that is not the accepting state,

$\therefore\therefore$ w2 = 1100 is **not accepted** by finite automata.

NON DETERMINISTIC FINITE AUTOMATA (NFA)

A Non Deterministic Finite Automata has the power to be in several states at once.

Ex.,



Definition of NFA:

A DFA consists of 5 – tuples.

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

Where,

- Q - None empty, finite set of states.
- Σ - Finite set of symbols.
- δ - Transition function.
- q_0 - Initial state.
- F - Finite set of final states.

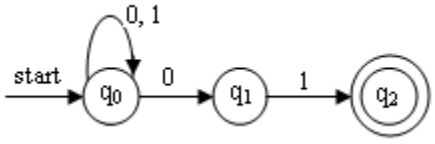
Difference between DFA and NFA:

DFA	NFA
1. In DFA δ is a transition function.	1. In NFA δ is a transition function that takes a state and input symbol as arguments.
2. DFA must returning exactly one state.	2. NFA returns a set of zero, one or more states.
3. The language of a DFA is defined by, $L(A) = (w \mid \hat{\delta}(q_0, w)$ is in F}	3. The language of an NFA is defined by, $L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$

PROBLEMS:

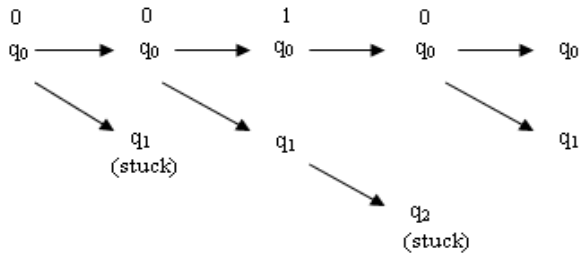
Ex.1:

Consider the following NFA, check whether the input strings w1 = 0010 and w2 = 00101 is accepted by the finite automata or not?



Soln:

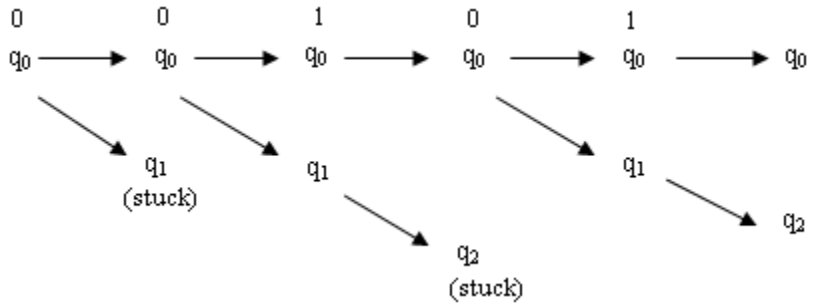
- w1 = 0010



Here answer is{ q_0, q_1}, both states are not accepting state.

∴∴ w1 = 0010 is **not accepted** by finite automata.

- w2 = 00101



Here answer is $\{ q_0, q_2 \}$, q_2 is the accepting state.

$\therefore\therefore$ w2 = 00101 is **accepted** by finite automata.

Theorem: Extending the transition function to strings on NFA:

Proof: To compute $\hat{\delta}(q, w) = \{ r_1, r_2, \ldots. r_m \}$

Basis:

$\hat{\delta}(q, \varepsilon) = \{q\}$. That is without reading any input symbols, we are only in the state we began it.

Induction: To compute $\hat{\delta}(q, w)$

Suppose 'w' is a string of the form 'xa' that is,

a - is the last symbol of 'w'.

x - is the string consisting of all but except the last symbol.

First compute $\hat{\delta}(q, x)$, it states that the automaton is in after processing all but the last symbol of 'w'. Suppose that is; $\hat{\delta}(q, x) = \{ p_1, p_2, \ldots\ldots\ldots, p_k \}$

ie) $\hat{\delta}(q, w) = \hat{\delta}(q, xa)$ $\quad [\therefore\therefore w = xa]$

$= \delta(\hat{\delta}(q, x), a)$

$= \delta(\{ p_1, p_2, \ldots\ldots\ldots, p_k \}, a)$ $\quad [\therefore\therefore \hat{\delta}(q, x) = \{ p_1, p_2, \ldots\ldots\ldots, p_k \}]$

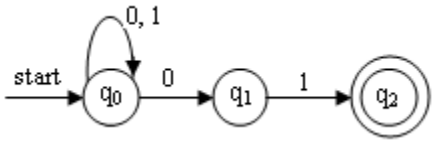
$= \delta(p_i, a)$ $\quad [\therefore\therefore \bigcup_{i=1}^{k} \delta(p_i, a) \bigcup_{i=1}^{k} \delta(p_i, a)]$

$= \{ r_1, r_2, \ldots\ldots\ldots, r_m \}$

PROBLEMS:

Ex.1:

Consider the following NFA, check whether the input string w1 = 0010 and w2 = 00101 is accepted by the finite automata or not?



Soln:

- w1 = 0010

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

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

$\hat{\delta}(q_0, 00) = \delta(\hat{\delta}(q_0, 0), 0) = \delta(\{q_0, q_1\}, 0) = \{q_0, q_1\}$

$\hat{\delta}(q_0, 001) = \delta(\hat{\delta}(q_0, 00), 1) = \delta(\{q_0, q_1\}, 1) = \{q_0, q_2\}$

$\hat{\delta}(q_0, 0010) = \delta(\hat{\delta}(q_0, 001), 0) = \delta(\{q_0, q_2\}, 0) = \{q_0, q_1\}$

Here answer is $\{q_0, q_1\}$, both the states are not accepting state,

$\therefore\therefore$ w1 = 0010 is **not accepted** by finite automata.

- w2 = 00101

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

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

$\hat{\delta}(q_0, 00) = \delta(\hat{\delta}(q_0, 0), 0) = \delta(\{q_0, q_1\}, 0) = \{q_0, q_1\}$

$\hat{\delta}(q_0, 001) = \delta(\hat{\delta}(q_0, 00), 1) = \delta(\{q_0, q_1\}, 1) = \{q_0, q_2\}$

$\hat{\delta}(q_0, 0010) = \delta(\hat{\delta}(q_0, 001), 0) = \delta(\{q_0, q_2\}, 0) = \{q_0, q_1\}$

$\hat{\delta}(q_0, 00101) = \delta(\hat{\delta}(q_0, 0010), 1) = \delta(\{q_0, q_1\}, 1) = \{q_0, q_2\}$

Here answer is $\{q_0, q_2\}$, q_2 is the accepting state,

$\therefore\therefore$ w2 = 00101 is **accepted** by finite automata.

Equivalence of Deterministic & Non-Deterministic Finite Automata:

The DFA's can do whatever NFA's can do involves an important "construction" called the **subset construction** because it involves constructing all subsets of the set of states of the NFA.

The subset construction starts from an NFA $N = \{Q_N, \Sigma, \delta_N, q_0, F_N\}$. Its goal is the description of a DFA $D = \{Q_D, \Sigma, \delta_D, q_0, F_D\}$, such that L(D) = L(N). If there are n elements in NFA there are 2^n elements in DFA.

Theorem:

If $D = \{Q_D, \Sigma, \delta_D, q_0, F_D\}$ is the DFA constructed from NFA $N = \{Q_N, \Sigma, \delta_N, q_0, F_N\}$ by the subset construction, then L(D) = L(N).

Proof:

To prove, by induction on |w| is that, $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$. Notice that each of the $\hat{\delta}$ functions returns a set of states from Q_N, but $\hat{\delta}_D$ interprets this set as one of the states of Q_D (which is the power set of Q_N), while $\hat{\delta}_N$ interprets this set as a subset of Q_N.

Basis:

Let |w| = 0, that is , w = ε. By the basis definitions of $\hat{\delta}$ for DFA's and NFA's both $\hat{\delta}_D(\{q_0\}, \varepsilon)$ and $\hat{\delta}_N(q_0, \varepsilon)$ are $\{q_0\}$.

Induction:

Let 'w' be of length n+1, and assume the statement for length 'n'. Break 'w' up as w = xa, where 'a' is the final symbol of 'w'. By the inductive hypothesis, $\hat{\delta}(\{q_0\}, x) = \hat{\delta}(q_0, x)$. Let both these sets of N's states be

$\{p_1, p_2, \ldots\ldots, p_k\}$. The inductive part of the definition of $\hat{\delta}$ for NFA's tell us,

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

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

$= \delta_N(\{p_1, p_2, \ldots p_k\}, a)$

$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \bigcup_{i=1}^{k} \delta_N(p_i, a)$ ---------- (1)

The subset construction on the other hand tell us that,

$\delta_D(\{p_1, p_2, \ldots p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \bigcup_{i=1}^{k} \delta_N(p_i, a)$ ---------- (2)

Now, let us use equation (2) and the inductive part of the definition of $\hat{\delta}$ for DFA's tell us,

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

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

$= \delta_D(\{p_1, p_2, \ldots p_k\}, a)$

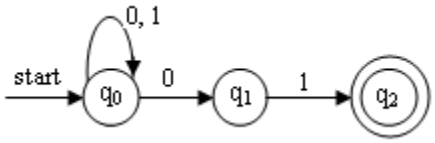
$\hat{\delta}_D(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \bigcup_{i=1}^{k} \delta_N(p_i, a)$ ---------- (3)

Thus equations (1) and (3) demonstrate that $\hat{\delta}_D(\{q_0\}, w\} = \hat{\delta}_N(q_0, w)$. When we observe that D and N both accept 'w' if and only if $\hat{\delta}_D(\{q_0\}, w\}$ or $\hat{\delta}_N(q_0, w)$, respectively contain a state in F_N, we have a complete proof that L(D) = L(N).

PROBLEMS:

Ex.1:

Convert the following NFA to its equivalent DFA.



Soln:

Step 1: To find subset construction:

P(N) = 2^n, where n is the total number of states. n =3,

P(N) = $2^3 = 8$, To find 8 subsets of the set of states.

P(N) = ({}, $\{q_0\}$, $\{q_1\}$, $\{q_2\}$, $\{q_0, q_1\}$, $\{q_0, q_2\}$, $\{q_1, q_2\}$, $\{q_0, q_1, q_2\}$)

Step 2: To construct transition table:

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

*$\{q_0, q_1, q_2\}$		

Step 3: Renaming the States:

State	Input	
	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

Step 4: To find transition function:

$\{q_0\}$ ---------- (A)

$\delta_D(\{q_0\}, 0) = \{q_0, q_1\}$ ---------- (B)

$\delta_D(\{q_0\}, 1) = \{q_0\}$ ---------- (A)

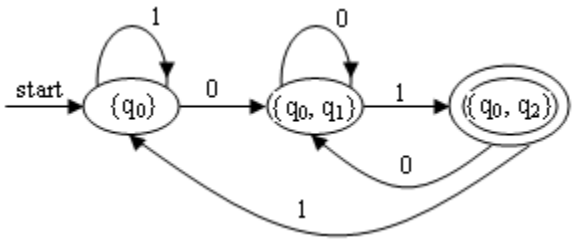
$\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ ---------- (B)

$\delta_D(\{q_0, q_1\}, 1) = \{q_0, q_2\}$ ---------- (C)

$\delta_D(\{q_0, q_2\}, 0) = \{q_0, q_1\}$ ---------- (B)

$\delta_D(\{q_0, q_2\}, 1) = \{q_0\}$ ---------- (A)

Step 5: To construct transition diagram for DFA:



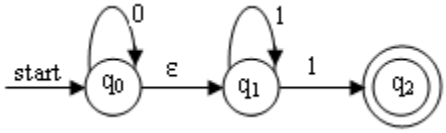
FINITE AUTOMATA WITH EPSILON TRANSITIONS

ε – transition only in NFA not in DFA. ε – NFA is move from one state to another state without the input symbol.

Use of ε – transitions:

We shall begin with an informal treatment of ε – NFA's, using transition diagrams with ε allowed as a label.

Ex:



The Formal Notation for an ε – NFA:

- An ε – NFA exactly do an NFA, with one exception; the transition function must include information about transitions on ε.
- Formally, represent an ε – NFA A by, A = (Q, Σ, δ, q_0, F), where all components have their same interpretation as for an NFA, except that δ is now a function that takes as arguments;
  (1) A state in Q.
  (2) A number of Σ ∪ ∪ {ε} that is, either an input symbol or the symbol ε.

Epsilon – Closures:

ε-close a state q by following all transitions out of q that are labeled ε. However, when we get to other states by following ε, we follow the ε – transitions out of those states, and so on, eventually finding every state that can be reached from q along any path whose arc's are all labeled ε.

Theorem: Extended Transitions and Languages for ε – NFA:

Suppose that E = (Q, Σ, δ, q_0, F) is an ε – NFA. We first define $\hat{\delta}$, the extended transition function, to reflect what happens on a sequence of inputs. The appropriate recursive definition of $\hat{\delta}$ is;

Basis:

$\hat{\delta}(q, \varepsilon)$ = ECLOSE(q)

Induction:

Suppose 'w' is of the xa, where 'a' is the last symbol of 'w'. Note that 'a' is a member of Σ; it cannot be ε, which is not in Σ. To compute $\hat{\delta}(q, w)$ as follows:

1. Let {p_1, p_2,……p_k} be $\hat{\delta}(q, x)$. That is, the P_i's are all and only the states that can reach from 'q' following a path labeled 'x'.
2. Let $\bigcup_{i=1}^{k} \delta(p_i, a)$ $\bigcup_{i=1}^{k} \delta(p_i, a)$ be the set {r_1, r_2, ……r_m}, that is follow all transitions labeled 'a' from states we can reach from 'q' along paths labeled 'x'. The r_j's are some of the states can reach from 'q' along paths labeled 'w'.
3. The $\hat{\delta}(q, w) = \bigcup_{j=1}^{m} ECLOSE(r_j)$ $\bigcup_{j=1}^{m} ECLOSE(r_j)$

   ie) $\hat{\delta}(q, w)$ = ECLOSE($\hat{\delta}$ (q, xa)) [∴∴ w = xa]

   = ECLOSE(δ ($\hat{\delta}$ (q, x), a))

   = ECLSOE(δ ({ p_1, p_2,……, p_k }, a))[∴∴ $\hat{\delta}$ (q, x) = { p_1, p_2,…, p_k }]

= ECLOSE(δ (p_i, a)) [∴∴ $\bigcup_{i=1}^{k} \delta(p_i, a)$ $\bigcup_{i=1}^{k} \delta(p_i, a)$]

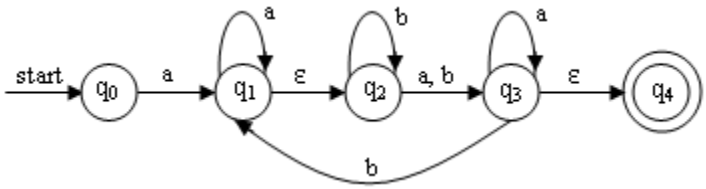
= ECLOSE({ r_1, r_2, ……., r_m })

$\hat{\delta}$ (q, w) = $\bigcup_{j=1}^{m} ECLOSE(r_j)$ $\bigcup_{j=1}^{m} ECLOSE(r_j)$.

PROBLEMS:

Ex.1:

Consider the following ε – NFA, check whether the following input string w = abab is accepted or not?



Soln:

w = abab

$\hat{\delta}$ (q_0, ε) = ECLOSE({q_0}) = {q_0}

$\hat{\delta}$ (q_0, a) = ECLOSE(δ($\hat{\delta}$ (q_0, ε), a)) = ECLOSE(δ({q_0}, a))

= ECLOSE ({q_1}) = {q_1, q_2}

$\hat{\delta}$ (q_0, ab) = ECLOSE(δ($\hat{\delta}$ (q_0, a), b)) = ECLOSE(δ({q_1, q_2}, b)) =

ECLOSE({q_2, q_3}) = {q_2, q_3, q_4}

$\hat{\delta}$ (q_0, aba) = ECLOSE(δ($\hat{\delta}$ (q_0, ab), a)) = ECLOSE(δ({q_2, q_3, q_4}, a)) =

ECLOSE({q_3}) = {q_3, q_4}

$\hat{\delta}$ (q_0, abab) = ECLOSE(δ($\hat{\delta}$ (q_0, aba), b)) = ECLOSE(δ({q_3, q_4}, b)) =

ECLOSE({ q_1}) = {q_1, q_2}

Here answer is { q_1, q_2 }, both the states are not accepting states,

∴∴ w = abab is **not accepted**.

Eliminating ε – transitions:

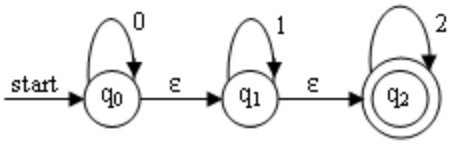
Given any ε – NFA E, we can find a DFA D that accepts the same language as E. Let E = (Q_E, Σ, δ_E, q_0, F_E). Then the equivalent DFA, D = (Q_D, Σ, δ_D, q_0, F_D) is defined as follows;

1. Q_D is the set of subsets of Q_E.
2. q_0 = ECLOSE(q_0)
3. δ_D(S, a) is computed, for all 'a' in Σ and sets 'S' in Q_D by:
   (a) Let S = {p_1, p_2, …..p_k}
   (b) Compute $\bigcup_{i=1}^{k} \delta(p_i, a)$ $\bigcup_{i=1}^{k} \delta(p_i, a)$; let this set be {$r_1$, r_2, …… r_m}.
   (c) Then δ_D(S, a) = $\bigcup_{j=1}^{m} ECLOSE(r_j)$ $\bigcup_{j=1}^{m} ECLOSE(r_j)$

PROBLEMS:

Ex.1:

Consider the following ε – NFA, compute the ε-closure of each state and find it's equivalent DFA.



Soln:

Step 1: To find ε-closure:

ECLOSE($\{q_0\}$) = $\{q_0, q_1, q_2\}$ ---------- (A)

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

$\delta(A, 1) = \{q_1\}$

$\delta(A, 2) = \{q_2\}$

ECLOSE($\{q_1\}$) = $\{q_1, q_2\}$ ---------- (B)

$\delta(B, 0) = \phi$

$\delta(B, 1) = \{q_1\}$

$\delta(B, 2) = \{q_2\}$

ECLOSE($\{q_2\}$) = $\{q_2\}$ ---------- (C)

$\delta(C, 0) = \phi$

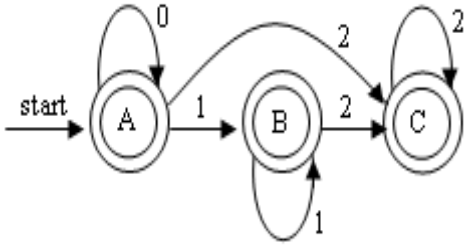
$\delta(C, 1) = \phi$

$\delta(C, 2) = \{q_2\}$

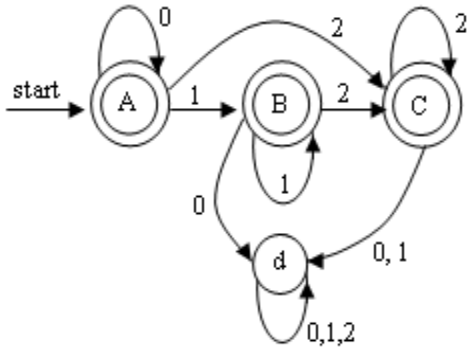
Step 2: To construct transition table:

State	Input		
	0	1	2
→*A	A	B	C
*B	ϕ	B	C
*C	ϕ	ϕ	C

Step 3: To construct transition diagram for DFA:



This is not DFA, in DFA each state read all the input's exactly once. So here, we construct one more transition diagram, for that we can add one more state, that is known as **"dead state"**, which is represented by 'd'.



UNIT II
REGULAR EXPRESSIONS AND LANGUAGES

Regular expression – Regular Languages- Equivalence of Finite Automata and regular expressions – Proving languages to be not regular (Pumping Lemma) – Closure properties of regular languages.

.

REGULAR EXPRESSION

The language accepted by finite automata are easily described by simple expressions called regular expressions.

Definition:

Let Σ be an alphabet. The regular expressions over Σ and the regular sets are defined as follows,

1. Ø is a regular expression, and the regular set is denoted as empty set {}.
2. ε is a regular expression and the regular set is denoted as {ε}.
3. For each 'a' in Σ. 'a' is a regular expression and the regular set is denoted as {a}.
4. If 'r' and 's' are regular expressions denoting the languages R and S then r+s, rs and r* are regular expressions that denotes the set R S, RS and R* respectively.

Languages associated with the regular expressions 'r' is denoted as L(r). If 'r_1' and 'r_2' are regular expressions, then

$L(r_1 + r_2) = L(r_1) + L(r_2)$

$L(r_1 . r_2) = L(r_1) . L(r_2)$

$L(r_1)^* = (L(r_1))^*$

The Operators of Regular Expressions:

Before describing the regular expression notation, we need to learn the three operations on languages that the operators of regular expressions represent. These operations are,

1. **Union(∪):**

   If L and M are regular expressions then LU M is the set of strings that are in either L or M or both.

   Ex: If L = {001, 10, 111} and M = {ε, 001} then,

   L∪M = {ε, 10, 001, 111}.

2. **Concatenation(.):**

   If L and M are regular expressions then, the set of strings that can be formed by taking any stings in L and concatenating it with any string in M. We denote the concatenation operator is frequently called 'dot'.

   If L = {001, 10, 111} and M = {ε, 001} then,

   L.M (or) LM = {001, 10, 111, 001001, 10001, 111001}.

3. **Closure(*):**

   A language L is denoted L* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions and concatenating all of them.

Building Regular Expressions:

Basis:

It consists of three parts,

1. The constants ε and Ø are regular expressions, denoting the languages {ε} and φ respectively.

That is, L(ε) = {ε} and L(Ø) = Ø.

2. If 'a' is any symbol, then 'a' is a regular expression. It denotes the language {a}. That is, L(a) = {a}.
3. A variable usually capitalized such as L, is a variable representing any language.

Induction:

It consists of four parts,

1. If E and F are regular expressions, then E+F is regular expression denoting the union of L(E) and L(F). That is, L(E+F) ⊎ L(E) L(F).
2. If E and F are regular expressions, then EF is a regular expression denoting the concatenation of L(E) and L(F). That is, L(E.F) = L(E).L(F)
3. If E is a regular expression, then E* is a regular expression denoting the closure of L(E). That is , L(E)*=(L(E))*.
4. If E is a regular expression, then (E), a parenthesized E, is also a regular expression denoting the same language as E. That is, L((E)) = L(E).

Precedence of Regular Expression Operators:

The regular expression operators have an assumed order or "precedence", which means that operators are associated with their operands in a particular order. For regular expression the following is the order of precedence for the operators,

1. The star(*) operator is of highest precedence.
2. Next in precedence comes the concatenation or dot(.) operator.
3. Finally use the '+' and '–' operators. This '+' and '-' are lowest precedence than other two.

Let Σ = {a, b, c, d}, check whether (a+b)*(cd) is a regular expression?

Let r = (a+b)*cd Let r_1 = a and r_2 = b

$r_3 = r_1 + r_2$

r_3 = a + b is a regular expression.

$(r_3)^*$ = (a + b)* is also a regular expression.

Let r_4 = c, r_5 = d

r_6 = cd is also a regular expression. [$r_6 = r_4 . r_5$] Hence, (a + b)*(cd) is regular expression.

Ex.2:

Describe the following sets by regular expressions,

(a) The set of all strings of 0's and 1's ending in 00.
(b) Set of all strings of 0's and 1's beginning with 11 and ending with '0'.
(c) The set of all strings over {a, b} with three consecutive b's.
(d) The set of all strings with atleast one pair of consecutive 0's and atleast one pair of consecutive 1's.
(e) All strings that end with '1' and does not contains the substring '00'.

Soln:

(a) r = (0+1)*00
(b) r = 11(0+1)*0
(c) r = (a+b)*bbb(a+b)*

(d) r = (1+01)*00(1+01)*(0+10)*11(0+10)*

(e) r = (1+01)*(10+11)*1

FA AND REGULAR EXPRESSIONS

The regular expressions define the same class, it shows that,

1. Every language defined by one of these automata is also defined by a regular expression. For this proof, we can assume the language is accepted by some DFA.
2. Every language defined by one of these automata. For this part of the proof, the easiest is to show that there is an NFA with ε – transitions accepting the same language.

From DFA's to Regular Expressions:

Theorem:

If L = L(A) for some DFA A, then there is a regular expression R such that L = L(R).

Basis:

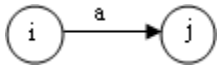
The basis is k=0. Then the regular expression is $R_{ij}^{(0)}$

Case 1:

If there is not such symbol 'a', then $R_{ij}^{(0)}$ = ø, That is,

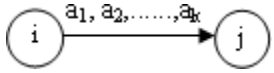
Case 2:

If there is exactly one such symbol 'a', then $R_{ij}^{(0)}$ = **a,** That is,



Case 3:

If there are symbols $a_1, a_2, \ldots\ldots, a_k$ that label arcs from state 'i' to state 'j', then = $\mathbf{a_1+a_2+\ldots..+a_k}$, That is,



Case 4:

If i = j then the legal paths are the path of length '0' and all loops from 'i' to itself. The path of length '0' is represented by the regular expression 'ε', since that path has no symbols along it.

If there is no such symbol 'a', the expression becomes ε, then $R_{ii}^{(0)}$ = ε + ø, That is,

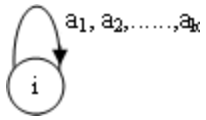
Case 5:

If there is a symbol 'a', the expression becomes ε, then $R_{ii}^{(0)}$ = ε + **a,** That is,

(or)

- = $\varepsilon+\mathbf{a_1+a_2+\ldots..+a_k}$, That is,

Induction:



Suppose there is a path from state 'i' to state 'j' that goes through no state higher than 'k'. There are two possible cases to consider,

Case 1:

The path does not go through state 'k' at all. In this case, the label of the path is in the language of $R_{ij}^{(k-1)}$.

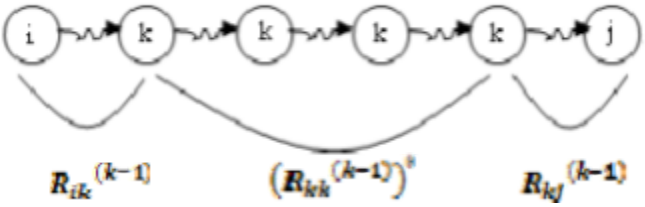
Case 2:

The path goes through state 'k' atleast once. The we can break the path into several pieces. That is,

(i) The first goes from state 'i' to state 'k' without passing through 'k'. That is, $R_{ik}^{(k-1)}$

(ii) The last piece goes from 'k' to 'j' without passing through 'k'. That is, $R_{kj}^{(k-1)}$

(iii)All the pieces in the middle go from 'k' to itself without passing through 'k'. That is, $(R_{kk}^{(k-1)})^*$



$\therefore R_{ij}^{(k)}$ = **case1 + case2**

$R_{ij}^{(k)}$

Minimization Rules for Regular Expression:

1. Ø + R = R
2. ØR = RØ = Ø
3. εR = Rε = R
4. ε* = ε and Ø* = ε
5. R + R = R
6. R*R* = R*
7. RR* = R*R
8. (R*)* = R*
9. ε + RR* = R* = ε + R*R
10. R* + ε = R*
11. (R + ε)* = R*
12. R.R = R
13. ε + R = R
14. R*(a + b) + (a + b) = R*(a + b)
15. Ø + ε = ε

16. R*R + R = R*R
17. (R + ε) (R + ε)* (R + ε) = R*
18. (R + ε) R* = R*(R + ε) = R*
19. Ø(ε + R)* = Ø
20. (ε + R) (ε + R)* = R*
21. ε + R* = R*

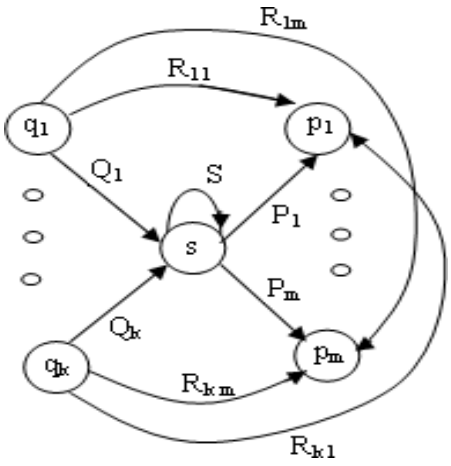
Converting DFA's to Regular Expressions by Eliminating States:

For converting DFA's to Regular Expression by avoids duplicating work at some points.

PROBLEMS:

Ex.1:

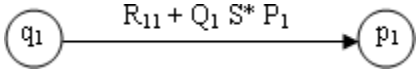
Convert the following DFA to regular expression by eliminating states.



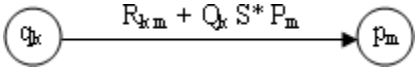
Soln:

To eliminate state 's'. So all arcs involving state 's' are deleted.

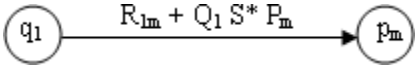
1. Find $q_1 \rightarrow p_1$



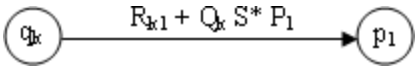
2. Find $q_k \rightarrow p_m$



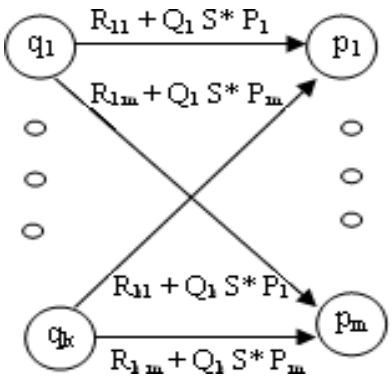
3. Find $q_1 \rightarrow p_m$



4. Find $q_k \rightarrow p_1$



Result of Eliminating State 's' is;



Converting Regular Expressions to Automata:

All of the automata we construct are ε – NFA's with a single accepting state.

Theorem:

Every language defined by a regular expression is also defined by a finite automaton.

Proof:

Suppose L = L(R) for a regular expression R. We show that L = L(E) for some ε – NFA E with;

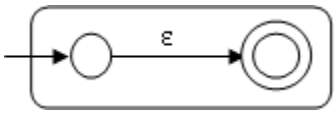
1. Exactly one accepting state.
2. No arcs into the initial state.
3. No arcs out of the accepting state.

Basis:

There are three parts to the basis,

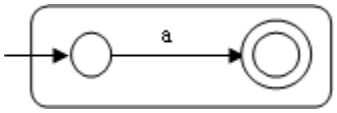
a. How to handle the expression ε.

The language of the automaton is easily seen to be {ε}, since the only path from the start state to an accepting state is labeled ε.



b. It shows the construction for Ø. Clearly there are no paths from start state to accepting state. So Ø is the language of this automaton.

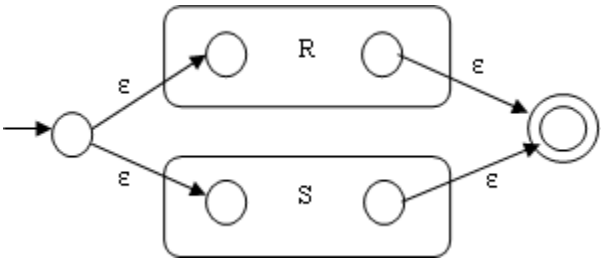
c. The language of this automaton evidently consists of the one string ‘a’, which is also L(a).



Induction:

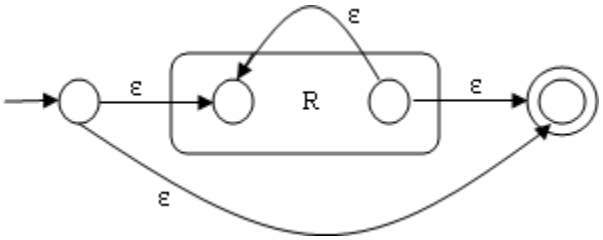
There are three parts to the induction,

1. The expression is R + S for some smaller expressions R and s. Thus the language of the automaton is L(R) L(S). The R + S equivalent ε – NFA is,
2.



3. The expression is R.S for some smaller expressions R and S. Thus the language of automaton is L(R)L(S). The automaton for the concatenation is shown in fig.

4. The expression is R* for some smaller expression R. The R* equivalent ε – NFA is,



PROBLEMS:

Ex.1:

Convert the following regular expressions to NFA’s with ε – transitions.

(i) 01*

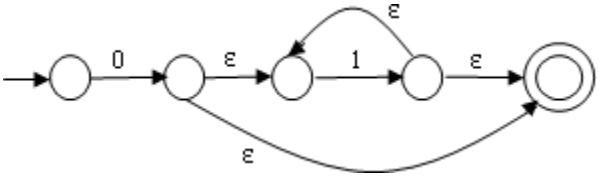
(ii) (0 + 1) 01

(iii) 00 (0 + 1)*

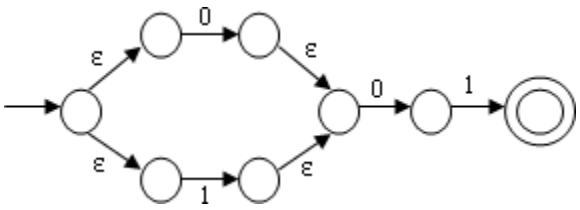
(iv) (0 + 1)* 1 (0 + 1)

Soln:

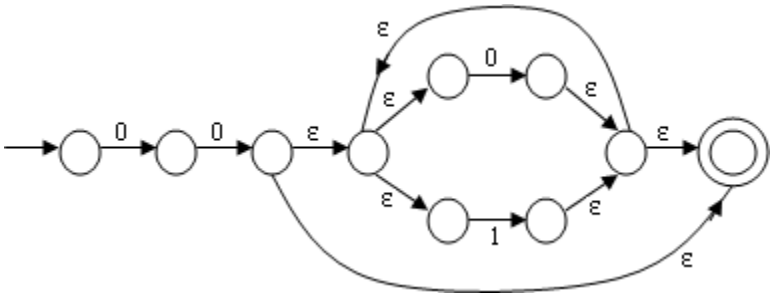
(i) 01*



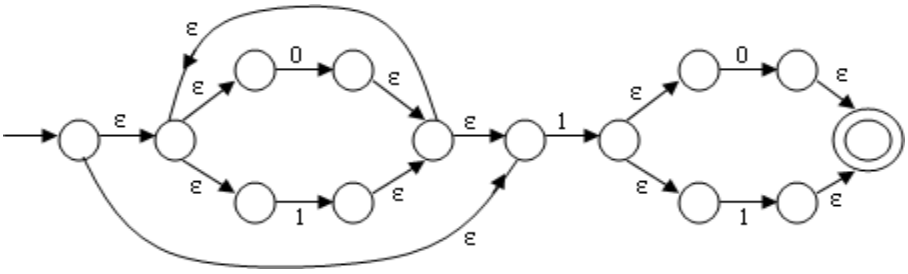
ii) (0 + 1) 01



(iii) 00 (0 + 1)*



(iv) (0 + 1)* 1 (0 + 1)



PROVING LANGUAGES NOT TO BE REGULAR

Pumping Lemma:

It is a powerful technique, which is used to prove that certain languages are not regular.

Principle:

- For a string of length > n accepted by DFA (n, number of states) the walk through of a DFA must contain a cycle.
- Repeating the cycle an arbitrary number of times, it should yield another string accepted by the DFA.

Theorem:

Let L be a regular language. Then there exists a constant 'n'(which depends on L) such that for every string 'w' in L such that $|w| \geq n$, we can break 'w' into three strings, w = xyz, such that;

(1) $| y | \geq 1$

(2) $| xy | \leq n$

(3) For all $k \geq 0$, the string xy^kz is also in L.

Proof:

Suppose L is regular. Then L = L(A) for some DFA A. Suppose A has 'n' states. Now, consider any

string 'w' of length 'n' or more, say $w = a_1a_2 \ldots a_m$, where $m \geq n$ and each 'a_i' is an input symbol. For i=0,1,….n define state P_i to be,

$$\hat{\delta}(q_0, a_1a_2 \ldots a_i)$$

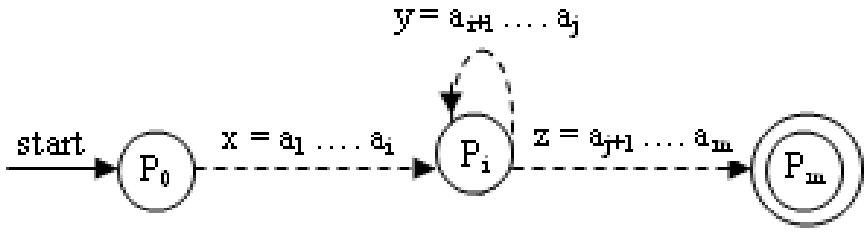
Where δ is the transition function of A, and q_0 is the start state of A. That is P_i is the state A is in after reading the first 'i' symbols of 'w'. Note that $P_0 = q_0$.

By the Pigeonhole principle, it is not possible for the n+1 different P_i's for i = 0,1, …. n to be distinct, since there are only 'n' different states. Thus we can find two different integers 'i' and 'j', with $0 \leq i \leq j \leq n$, such that $P_i = P_j$. Now, we can break w = xyz as follows,

(1) $x = a_1a_2 \ldots a_i$

(2) $y = a_{i+1}\, a_{i+2} \ldots\ldots a_j$

(3) $z = a_{j+1}\, a_{j+2} \ldots\ldots a_m$



That is, A receives the input xy^kz for any $k \geq 0$.

- If k = 0, then the automaton goes from the start state q_0 to P_i on input 'x'. Since, P_i is also P_j, it must be that A goes from P_i to the accepting state on input 'z'. Thus, A accepts xz.
- If k > 0, then A goes from q_0 to P_i on input 'x', circles from P_i to P_{ik} times on input y_k, and then goes to the accepting state on input z. Thus for any $k \geq 0$, xy^kz is also accepted by A, that is xy^kz is in L.

Applications of the Pumping Lemma:

To prove certain language is not regular.

1. Assume language L is regular.
2. Let 'n' be the constant of pumping lemma and is finite.
3. Select a string 'w' in L with $|w| \geq n$.
4. Show that for every decomposition of 'w' into 'xyz'(such that $|y| \geq 1$, $|xy| \leq n$) there exists k ≥ 0, such that $xy^kz \notin L$.
5. Conclude the assumption in (1) is false, that is, the language is not regular.

CLOSURE PROPERTIES OF REGULAR LANGUAGES

If certain languages are regular, and a language L is formed from them by certain operations, then L is also regular. These theorems are often called closure properties of the regular languages. Some of the closure properties from regular languages are,

(1) The Union of two regular languages is regular.

(2) The Intersection of two regular languages is regular.
(3) The Complement of a regular language is regular.
(4) The Difference of two regular languages is regular.
(5) The Reversal of a regular language is regular.
(6) The Closure (star) of a regular language is regular.
(7) The Concatenation of regular language is regular.
(8) The Homomorphism of a regular language is regular.

1. Closure Under Union:

Let L and M be languages over alphabet Σ. Then L ∪ M is the language that contains all strings that are in either or both or L and M.

Theorem:

If L and M are regular languages then L ∪ M is also regular.

Proof: Since L and M are regular languages the regular expressions say, L = L(R), M = L(S)

Then L ∪ M = L(R + S)

For ex: Consider two languages L and M,

$L = \{0^n1^n \mid n \geq 1\}$ and $M = \{0^i1^j \mid i \geq j\}$

ie) $L = \{01, 0011, 000111, 00001111, \ldots\ldots\}$ and $M = \{01, 001, 0011, 00011, \ldots\ldots\}$

$\therefore L \cup M = \{01, 0011, 000111, 00001111, 001, 00011, \ldots\}$

This is present in both the languages L and M. So the Union of two regular languages is regular.

2. Closure Under Intersection:

Let L and M be languages over alphabet Σ. Then L ∩ M is the language that contains all strings in both the languages L and M.

Theorem:

If L and M be regular languages, then L∩M is regular.

Proof:

Since L and M are regular languages the regular expressions say, L = L(R), M = L(S)

Then L∩M = L(R . S)

For ex: Consider two languages L and M,

$L = \{0^n1^n \mid n \geq 1\}$ and $M = \{0^i1^j \mid i \geq j\}$

ie) $L = \{01, 0011, 000111, 00001111, \ldots\ldots\}$ and $M = \{01, 001, 0011, 00011, \ldots\ldots\}$

∴ L∩M = {01, 0011, ……}

This is present in both the languages L and M. So the Intersection of two regular languages is regular.

3. <u>Closure Under Complementation:</u>

Steps for converting a regular expression to its complement is,

(i) Convert the regular expression to NFA. (ii)Convert NFA to a DFA by subset construction.
(iii) Complement the accepting states of that DFA.

Theorem:

If L is a regular language over Σ then $\bar{L} = \Sigma^* - L$ is also regular.

Proof:

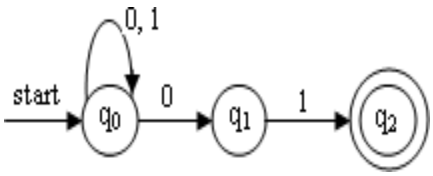
Let L = L(A) for DFA, A = (Q, Σ, δ, q_0, F) then, $\bar{L}$ = L(B) where B = (Q, Σ, δ, q_0, Q-F). B is similar to Abut accepting states of A have become non accepting states of B and accepting states of B have become accepting states of A. Then 'w' is in L(B) iff (q_0, w) is in Q-F which occurs iff 'w' is not in L(A).

Ex.

Soln:

Find the complement of (0+1)*01.

Step 1: Convert the regular expression to NFA.



Step 2: Convert NFA to a DFA by subset construction:

$P(N) = 2^3 = 8$, To find 8 subsets of the set of states.

$P(N) = (\{\}, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\})$

$\{q_0\}$ ---------- (A)

$\delta_D(\{q_0\}, 0) = \{q_0, q_1\}$ ---------- (B)

$\delta_D(\{q_0\}, 1) = \{q_0\}$ ---------- (A)

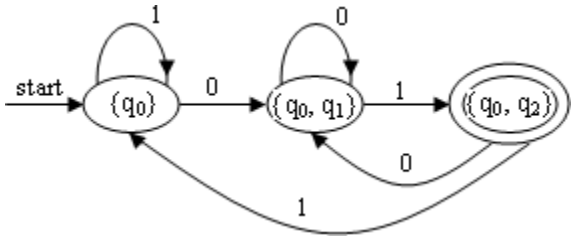
$\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ ---------- (B)

$\delta_D(\{q_0, q_1\}, 1) = \{q_0, q_2\}$ ---------- (C)

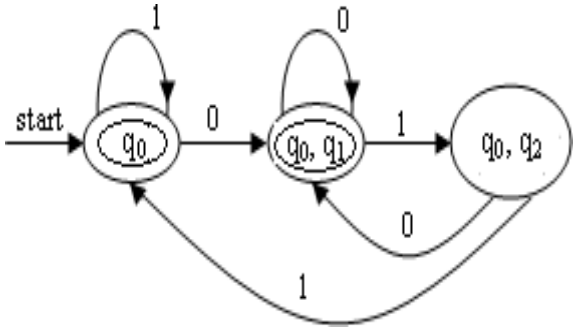
$\delta_D(\{q_0, q_2\}, 0) = \{q_0, q_1\}$ ---------- (B)

$\delta_D(\{q_0, q_2\}, 1) = \{q_0\}$ ---------- (A)

Transition Diagram:



Step 3: Complement the accepting states of that DFA.



4. **Closure Under Difference:**

Let L and M be languages over alphabet Σ. Then L-M is the language that contains all strings in L which is not in M.

Theorem:

If L and M be regular languages, then L-M is regular.

Proof:

Since L and M are regular languages the regular expressions say, L = L(R), M = L(S)

Intersection : Let L1, L2 be any two regular languages accepted by two DFSA's M1 = (K1;_1; _1; q1; F1) and M2 = (K2;Σ2; δ2; q2; F2). Then the DFSA M constructed as below accepts L1 \ L2. Let M = (K; Σ; δ;q0; F) where K = K1 x K2, q0 = (q1; q2), F = F1 x F2, δ : K→ K is defined by δ((p1; p2); a) = (δ1(p1; a); δ2(p2; a)). One can see that for each input word w, M runs M1 and M2 parallel, starting from q1; q2 respectively. Having finished reading the input, M accepts only if both M1; M2 accept. Hence L(M) = L(M1) \ L(M2).

Complementation : Let L1 be a regular language accepted by DFSA M = (K;_; _; q0; F). Then clearly the complement of L is accepted by the DFSA Mc = (K; Σ; δ; q0;K; F).

Concatenation : We prove this property using the concept of regular grammar. Let L1and L2 and G1 and G2 be defined as in proof of union of this theorem. Then the type 3 grammar G constructed as below satisfies the requirement that L(G) = L(G1):L(G2). G = (N1 [N2; T1 [T2; S1; P2 [P) clearly L(G) = L(G1):L(G2) because any derivation starting from S1 derives a word w 2 L1 and for G,

Catenation Closure : Here also we prove the closure using regular grammar. Let L1 be a regular grammar generated by G1 = (N1; T1; P1; S1). Then the type 3 grammar G = (N1 [fS0g; T1; S0; fS0 ! _; S0 ! S1g [fA! aS1jA ! a 2 P1g [P1). Clearly G generates L1*.

Reversal : The proof is given using the NFSA model. Let L be a language accepted by a NFSA with ε-transitions which has exactly one final state.

EQUIVALENCE AND MINIMIZATION OF AUTOMATA

Equivalence of two states:

The language generated by a DFA is unique. But, there can exist many DFA's that accept the same language. In such cases, the DFA's are said to be equivalent.

Definition of Equivalent and Inequivalent States: Equivalent (Indistinguishable) State:

Two states 'p' and 'q' of a DFA are equivalent if and only if δ(p, w) and δ(q, w) are final states or both δ(p, w) and δ(q, w) are non-final state for all w Σ* that is, if δ(p, w) F and δ(q, w) F then the states 'p' and 'q' are equivalent. If δ(p, w) F and δ(q, w) F then also the states 'p' and 'q' are equivalent.

Inequivalent (Distinguishable) State:

Two states 'p' and 'q' of a DFA are inequivalent, if there is atleast one string 'w' such that one of δ(p, w) and δ(q, w) is final state and the other is non-final state, then the states 'p' and 'q' are called inequivalent states.

The equivalent and inequivalent states can be obtained using **table filling algorithm**(also called mark procedure).

Table Filling Algorithm:

1. For each pair(p, q) where p Q and q Q. Find q F or vice versa then, the pair (p, q) is inequivalent and mark the pair (p, q) [by putting 'X' for the pair (p, q)]

2. For each pair (p, q) and for each a Σ find δ(p, a) = r and δ(q, a) = s. If the pair (r, s) is already marked at inequivalent then the pair (p, q) is also inequivalent and mark it as say 'X'.

Ex.1:

Soln:

Obtain the inquivalent table for the automaton shown below,

State	Input	
	0	1
→A	B	F
B	G	C
*C	A	C
D	C	G
E	B	F
F	C	G
G	G	E
H	G	C

Construct a table with an entry for each pair of states. An 'X' is placed in the table each time we discover a pair of state that cannot be equivalent. Initially an 'X' is placed in each entry corresponding to one final state and one non-final state.

X						
X	X					
X	X	X				
	X	X	X			
X	X	X		X		
X	X	X	X	X	X	
X		X	X	X	X	X

B
C
D
E
F
G
H

A B C D E F G

Minimization of Automata:

Once we have found the distinguishable and indistinguishable pairs we can easily minimize the number of states of a DFA and accepting the same language which is accepted by original DFA. It is required to reduce the number of states for storage efficiency.

Minimization of DFA:

Let M = (K;Σ; δ; q0; F) be a DFSA. Let R be an equivalence relation on K such that pRq, if and only if for each input string x, δ(p; x) € F if and only if δ (q; x) € F. This essentially means that if p and q are equivalent, then either δ(p; x) and δ(q; x) both are in F or both are not in F for any string x. p is

distinguishable from q if there exists a string x such that one of δ(q; x), δ(p; x) is in F and the other is not. x is called the distinguishing string for the pair < p; q >.

If p and q are equivalent δ p; a) and δ q; a) will be equivalent for any a. If δ(p; a) = r and δ(q; a) = s and r and s are distinguishable by x, then p and q are distinguishable by ax.

We get a partition of the set of states of K as follows:

Step 1: Consider the set of states in K. Divide them into two blocks F and K - F. (Any state in F is distinguishable from a state in K - F by ε) Repeat the following step till no more split is possible.

Step 2: Consider the set of states in a block. Consider the a-successors of them for a € Σ. If they belong to different blocks, split this block into two or more blocks depending on the a-successors of the states.

For example if a block has {q1; ……. ; qk}. δ(q1; a) = p1, δ(q2; a) = p2, …. , δ(qk; a) = pk and p1; …… ; pi belong to one block, pi+1; …….; pj belong to another block and pj+1; : : : ; pk belong to third block, then split {q1; ….. ; qk} into {q1; ….. ; qi} {qi+1; ….. ; qj} {qj+1; ….. ; qk}.

Step 3: For each block Bi, consider a state bi.

UNIT III
CONTEXT FREE GRAMMAR AND PUSH DOWN AUTOMATA

Types of Grammar - Chomsky's hierarchy of languages -Context-Free Grammar (CFG) and Languages – Derivations and Parse trees – Ambiguity in grammars and languages – Push Down Automata (PDA): Definition – Moves - Instantaneous descriptions -Languages of pushdown automata – Equivalence of pushdown automata and CFG-CFG to PDA-PDA to CFG – Deterministic Pushdown Automata.

CONTEXT FREE GRAMMAR(CFG)

Definition:

A context free grammar is a finite set of variables each of which represents a language. The languages represented by the variable are described recursively in terms of each other and primitive symbols called terminals. The rules relating the variables are called production. A Context Free Grammar(CFG) is denoted by,

$$G = (V, T, P, S)$$

Where, V – Variables

T – Terminals

P – Finite set of Productions of the form $A \rightarrow \alpha$ where, A is a variable and α is a string of symbols.

S – Start symbol.

A CFG is represented in Backus-Naur Form(BNF). For example consider the grammar,

<expression> → <expression> + <expression>

<expression> → <expression> * <expression>

<expression> → (<expression>)

<expression> → id

Here <expression> is the variable and the terminals are +, *, (,), id. The first two productions say that an expression can be composed of two expressions connected by an addition or multiplication sign. The third production says that an expression may be another expression surrounded by paranthesis. The last says a single operand is an expression.

Ex.1:

A CFG, G = (V, T, P, S) whose productions are given by,

$A \rightarrow Ba$

$B \rightarrow b$

Soln:

$A \rightarrow Ba$

$\rightarrow ba$ $\quad \therefore$ which produces the string ba.

Derivations Using a Grammar:

While inferring whether the given input string belongs to the given CFG, we can have two approaches,

- Using rules from body of head.
- Using rules from head to body.

First approach is called by the **name recursive inference**. Here we take strings from each variable, concatenate them in proper order and infer that the resulting strings is in the language of the variable in the head.

Another approach is called **derivation.** Here we use the productions from head to body. We expand the start symbol using one of its productions till it reached the given string.

PROBLEMS:

Ex.1:

Obtain the derivation for L, with production for the string 01c10.

$E \rightarrow c$

$E \rightarrow 0E0$

$E \rightarrow 1E1$

Soln:

$E \rightarrow 0E0$

$\rightarrow 01E10$ [$E \rightarrow 1E1$]

$\rightarrow 01c10$ [$E \rightarrow c$]

Ex.2:

Obtain the derivation for L, with production for the strings aaababbb and abbaab.

$S \rightarrow aSb \mid aAb$

$A \rightarrow bAa \mid ba$

Soln:

(i) $S \rightarrow aSb$

$\rightarrow aaSbb$ [$S \rightarrow aSb$]

$\rightarrow aaaAbbb$ [$S \rightarrow aAb$]

$\rightarrow aaababbb$ [$A \rightarrow ba$]

(ii) $S \rightarrow aAb$

$\rightarrow abAab$ [$A \rightarrow bAa$]

$\rightarrow abbaab$ [$A \rightarrow ba$]

Leftmost and Rightmost Derivations:

Leftmost Derivation:

If at each step in a derivation, a production is applied to the leftmost variable then it is called leftmost derivation.

Ex.1:

Let G = (V, T, P, S), V = {E}, T = {+, *, id}, S = E, P is given by,

E → E + E | E * E | id

Construct leftmost derivation for id+id*id.

Soln:

$E \overset{lm}{\Rightarrow} E+E$

$\Rightarrow$ id+E [E → id]

$\Rightarrow$ id+E*E [E → E*E]

$\Rightarrow$ id+id*E [E → id]

$\Rightarrow$ id+id*id [E → id]

Rightmost Derivation:

If at each step in a derivation a production is applied to the rightmost variable, then it is called rightmost derivation.

Ex.1:

Let G = (V, T, P, S), V = {E}, T = {+, *, id}, S = E, P is given by,

E → E + E | E * E | id

Construct rightmost derivation for id+id*id.

Soln:

$E \overset{rm}{\Rightarrow} E*E$

$\Rightarrow$ E*id [E → id]

$\Rightarrow$ E+E*id [E → E+E]

$\Rightarrow$ E+id*id [E → id]

$\Rightarrow$ id+id*id [E → id]

PARSE TREES or (Derivation Trees)

The derivations can be represented by trees using "parse trees".

Constructing Parse Trees:

Let G = (V, T, P, S) be a grammar. The parse trees for 'G' is a trees with following conditions.

1. Each interior node is labeled by a variable in V.
2. Each leaf is labeled by either a variable, a terminal or ε.
3. If an interior node is labeled A, and its children are labeled $X_1, X_2, \ldots\ldots X_k$ respectively from left, then $A \rightarrow X_1, X_2, \ldots\ldots\ldots X_k$ is a production in P.
4. If A → ε then A is considered to be the label.

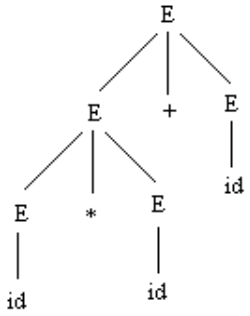
Ex.1:

Construct a parse tree for the grammar, E → E + E | E * E | (E) | id, for the string id*id+id

Soln:

Derivation: Parse Tree:

$E \Rightarrow E+E$
$\Rightarrow E*E+E$ $[E \Rightarrow E*E]$
$\Rightarrow id*E+E$ $[E \Rightarrow id]$
$\Rightarrow id*id+E$ $[E \Rightarrow id]$
$\Rightarrow id*id+id$ $[E \Rightarrow id]$



Yield of a Parse Tree:

The string obtained by concatenating the leaves of a parse tree from the left is called yield of a parse tree. The yield is always derived from the root. The root is the start symbol.

Ex.1:

For the grammar G is defined by the production,

$S \rightarrow A \mid B$
$A \rightarrow 0A \mid \varepsilon$
$B \rightarrow 0B \mid 1B \mid \varepsilon$

Find the parse tree for the yields (i) 1001 (ii) 00101

Soln:

(i) 1001

Derivation:

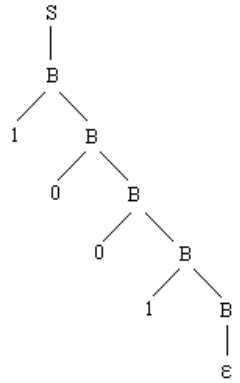
$S \Rightarrow B$
$\Rightarrow 1B$ $[B \rightarrow 1B]$
$\Rightarrow 10B$ $[B \rightarrow 0B]$
$\Rightarrow 100B$ $[B \rightarrow 0B]$
$\Rightarrow 1001B$ $[B \rightarrow 1B]$
$\Rightarrow 1001$ $[B \rightarrow \varepsilon]$

(ii) 00101

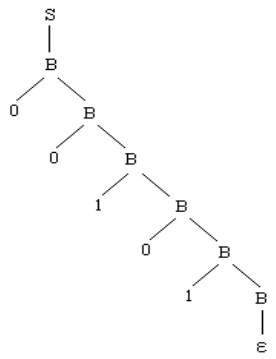
Derivation:

$S \Rightarrow B$
$\Rightarrow 0B$ $[B \rightarrow 0B]$
$\Rightarrow 00B$ $[B \rightarrow 0B]$
$\Rightarrow 001B$ $[B \rightarrow 1B]$
$\Rightarrow 0010B$ $[B \rightarrow 0B]$
$\Rightarrow 00101B$ $[B \rightarrow 1B]$
$\Rightarrow 00101$ $[B \rightarrow \varepsilon]$

Parse Tree:



Parse Tree:



Relationship Between the Derivation Trees and Derivation:

Theorem:

Let G = (V, T, P, S) be a CFG. Then $S \overset{*}{\Rightarrow} \alpha$ if and only if there is a derivation tree in grammar G with yield α.

Proof:

Suppose there is a parse tree with root S and yield α, then there is a leftmost derivation,

$$S \overset{*}{\underset{lm}{\Rightarrow}} \alpha \text{ in G.}$$

To prove, $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$ in G. Let us prove this theorem by induction on height of the tree.

Basis:

If the height of parse tree is '1' then the tree must be of the form given in figure below with root 'S' and yield 'α'.

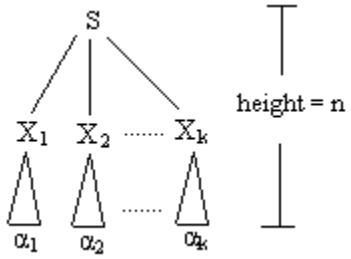
This means that 'S' has only leaves and no subtrees. This is possible only with the production.

$$S \Rightarrow \alpha \text{ in G}$$

$S \overset{}{\underset{lm}{\Rightarrow}} \alpha$ is an one step leftmost derivation.

Induction:

If the height of the parse tree is in the parse tree must look like in.



$\alpha = \alpha_1\alpha_2 \ldots\ldots\ldots \alpha_k$. Where $X_1, X_2, \ldots.. X_k$ are all the subtrees of S. Assume there exist a leftmost derivation $S \overset{*}{\Rightarrow} \alpha$ for every parse tree of height less than 'n'. Consider a parse tree of height 'n'. Let the leftmost derivation be,

$$S \overset{*}{\underset{lm}{\Rightarrow}} X_1X_2, \ldots.. X_k$$

The X_i's may be either terminals or variables.

(i) If X_i is a terminal then $X_i = \alpha_i$

(ii) If X_i is a variable then it must be the root of some sub-tree with yield α_i of height less than 'n'. By applying inductive hypothesis, there is a leftmost derivation,

$$X_i \overset{*}{\underset{lm}{\Rightarrow}} \alpha_i$$

$$S \overset{}{\underset{lm}{\Rightarrow}} X_1X_2, \ldots\ldots X_k$$

If X_i is a terminal then no change.

$$S \overset{*}{\underset{lm}{\Rightarrow}} \alpha_1\alpha_2 \ldots\ldots\ldots \alpha_i X_{i+1} \ldots\ldots X_k$$

If X_i is a variable then derive the string X_i to α_i as,

$$X_i \overset{}{\underset{lm}{\Rightarrow}} w_1 \Rightarrow w_2 \Rightarrow \ldots\ldots \Rightarrow \alpha_i$$

Therefore,

$$S \overset{*}{\underset{lm}{\Rightarrow}} \alpha_1\alpha_2 \ldots\ldots\ldots \alpha_{i-1} X_i X_{i+1} \ldots\ldots X_k$$

$$\overset{*}{\underset{lm}{\Rightarrow}} \alpha_1\alpha_2 \ldots\ldots\ldots \alpha_{i-1} w_1 X_{i+1} \ldots\ldots X_k$$

$$\overset{*}{\underset{lm}{\Rightarrow}} \alpha_1\alpha_2 \ldots\ldots\ldots \alpha_{i-1} w_2 X_{i+1} \ldots\ldots X_k$$

$$\vdots$$

$$\overset{*}{\underset{lm}{\Rightarrow}} \alpha_1\alpha_2 \ldots\ldots\ldots \alpha_{i-1} \alpha_i X_{i+1} \ldots\ldots X_k$$

By repeating the process we can get,

$$S \overset{*}{\underset{lm}{\Rightarrow}} \alpha_1\alpha_2 \ldots\ldots\ldots \alpha_k \qquad [\ \because\ \alpha = \alpha_1\alpha_2 \ldots\ldots\ldots \alpha_k\ ]$$

Thus proved.

AMBIGUITY IN GRAMMARS AND LANGUAGES

Sometimes there is an occurrence of ambiguous sentence in a language we are using. Like that in CFG there is a possibility of having two derivations for the same string.

Ambiguous Grammars:

A CFG, G = (V, T, P, S) is said to be ambiguous, if there is atleast one string 'w' has two different parse trees.

PROBLEMS:

Ex.1:

Construct ambiguous grammar for the grammar, $E \rightarrow E + E \mid E * E \mid (E) \mid id$, and generate a string id+id*id.

Soln:

Derivation1:

$E \Rightarrow E+E$

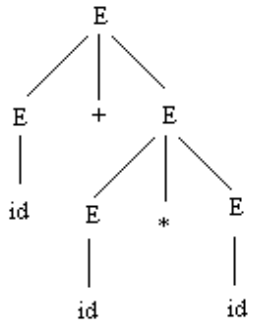
Derivation2:

$E \Rightarrow E*E$

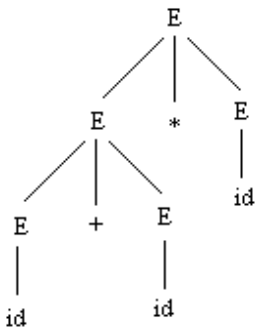
⇒ id+E [E → id]
⇒ id+E*E [E → E*E]
⇒ id+id*E [E → id]
⇒ id+id*id [E → id]

⇒ E+E*E [E → E+E]
⇒ id*E+E [E → id]
⇒ id*id+E [E → id]
⇒ id*id+id [E → id]

Parse Tree1:



Parse Tree2:



Ex.2:

Construct ambiguous grammar for the grammar,

E → I | E+E | E*E | (E)

I → a | b | Ia | Ib | I0 | I1

and generate a string a+a*a.

Soln:

Derivation1:

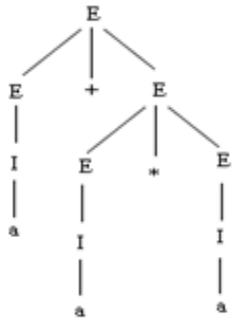
E ⇒ E+E
⇒ I+E [E → I]
⇒ a+E [I → a]
⇒ a+E*E [E → E*E]
⇒ a+I*E [E → I]
⇒ a+a*E [I → a]
⇒ a+a*I [E → I]
⇒ a+a*a [I → a]

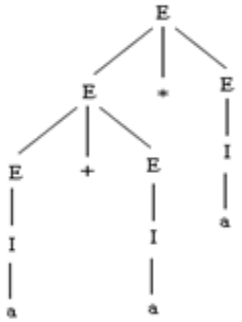
Derivation2:

E ⇒ E*E
⇒ E*I [E → I]
⇒ E*a [I → a]
⇒ E+E*a [E → E+E]
⇒ E+I*a [E → I]
⇒ E+a*a [I → a]
⇒ I+a*a [E → I]
⇒ a+a*a [I → a]

Parse Tree1:

Parse Tree2:





Unambiguous:

If each string has atmost one parse tree in the grammar, then the grammar is unambiguous.

Leftmost Derivations as a way to Express Ambiguity:

A grammar is said to be ambiguous, if it has more than one leftmost derivation.

Ex:

Construct ambiguous grammar for the grammar,

$E \rightarrow I \mid E+E \mid E*E \mid (E)$

$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

and generate two leftmost derivation for a string a+a*a.

Soln:

Derivation1:

$E \overset{rm}{\Rightarrow} E+E$

$\Rightarrow I+E \quad [E \rightarrow I]$

$\Rightarrow a+E \quad [I \rightarrow a]$

$\Rightarrow a+E*E \quad [E \rightarrow E*E]$

$\Rightarrow a+I*E \quad [E \rightarrow I]$

$\Rightarrow a+a*E \quad [I \rightarrow a]$

$\Rightarrow a+a*I \quad [E \rightarrow I]$

$\Rightarrow a+a*a \quad [I \rightarrow a]$

Derivation2:

$E \overset{rm}{\Rightarrow} E*E$

$\Rightarrow E+E*E \quad [E \rightarrow E+E]$

$\Rightarrow I+E*E \quad [E \rightarrow I]$

$\Rightarrow a+E*E \quad [I \rightarrow a]$

$\Rightarrow a+I*E \quad [E \rightarrow I]$

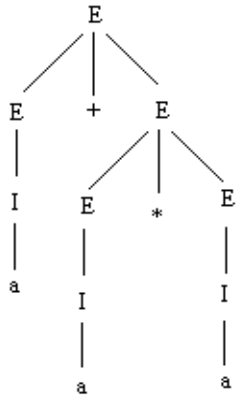
$\Rightarrow a+a*E \quad [I \rightarrow a]$

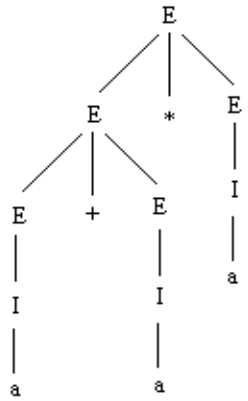
$\Rightarrow a+a*I \quad [E \rightarrow I]$

$\Rightarrow a+a*a \quad [I \rightarrow a]$

Parse Tree1:

Parse Tree2:





Inherent Ambiguity:

A CFL L is said to be inherently ambiguous, if every grammar for the language must be ambiguous.

Ex:

Show that the language is inherent ambiguous $L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$ then the production P is given by,

$S \rightarrow AB \mid C$

$A \rightarrow aAb \mid ab$ $\qquad$ $C \rightarrow aCd \mid aDd$

$B \rightarrow cBd \mid cd$ $\qquad$ $D \rightarrow bDc \mid bc$

Soln:

L is a context free language. It separate sets of productions to generate two kinds of strings in L. This grammar is ambiguous For ex, the string aabbccdd has two leftmost derivations.

Derivation1:

$S \overset{rm}{\Rightarrow} AB$

$\Rightarrow aAbB$

$\Rightarrow aabbB$

$\Rightarrow aabbcBd$

$\Rightarrow aabbccdd$

Derivation2:

$S \overset{rm}{\Rightarrow} C$

$\Rightarrow aCd$

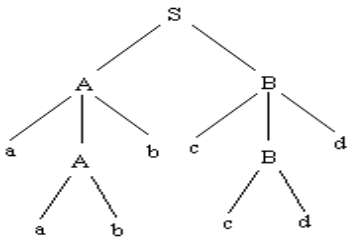
$\Rightarrow aaDdd$

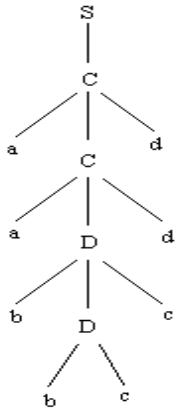
$\Rightarrow aabDcdd$

$\Rightarrow aabbccdd$

Parse Tree1:

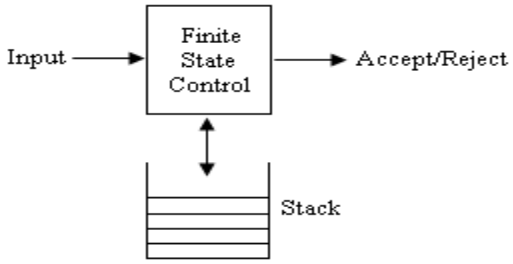
Parse Tree2:





DEFINITION OF THE PUSH DOWN AUTOMATA

A PushDown Automata(PDA) is essentially a finite automata with control of both an input tape and a stack on which it can store a string of stack symbols. With the help of a stack the pushdown automata can remember an infinite amount of information.



Model of PDA:

- The PDA consists of a finite set of states, a finite set of input symbols and a finite set of pushdown symbols.
- The finite control has control of both the input tape and pushdown store.
- In one transition of the PDA,
  - The control head reads the input symbol, then goto the new state.
  - Replaces the symbol at the top of the stack by any string.

Definition of PDA:

A PDA consists of seven tuples.

$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

where, Q – A finite set of states.

Σ – A finite set of input symbols.

Γ – A finite set of stack symbols.

δ - The transition function. Formally, δ takes a argument a triple $\delta(q, a, X)$,

where, - 'q' is a state in Q

- 'a' is either an input symbol in Σ or $a = \varepsilon$.
- 'X' is a stack symbol, that is a member of Γ.

Q_0 – The start state.

Z_0 – The start symbol of the stack.

F – The set of accepting states or final states.

Ex:

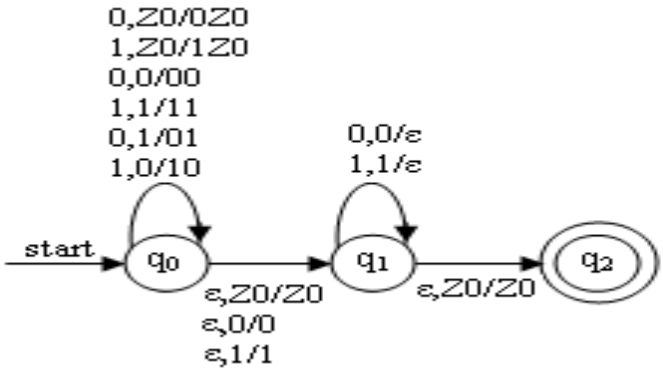
Mathematical model of a PDA for the language, L = {wwR | w is in (0+1)* }, then PDA for L can be described as, P = ({q_0, q_1, q_2}, {0, 1}, {0, 1, Z_0}, δ, q_0, Z_0, {q_2}), where δ is defined by the following rules;

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ and $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$. One of these rules applies initially, when we are in state q_0 and we see the start symbol Z_0 at the top of the stack. We read the first input, and push it onto the stack, leaving Z_0 below to mark the bottom.
2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$ and $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. These four similar rules allow us to stay in state q_0 and read inputs, pushing each onto the top of the stack and leaving the pervious top stack symbol alone.
3. $\delta(q_0, \varepsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \varepsilon, 0) = \{(q_1, 0)\}$, and $\delta(q_0, \varepsilon, 1) = \{(q_1, 1)\}$. These three rules allow P to go from state q_0 to state q_1 spontaneously (on ε input), leaving intact whatever symbol is at the top of the stack.
4. $\delta(q_1, 0, 0) = \{(q_1, \varepsilon)\}$, and $\delta(q_1, 1, 1) = \{(q_1, \varepsilon)\}$. Now in state q1we can match input symbols against the top symbols on the stack, and pop when the symbols match.
5. $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$. Finally, if we expose the bottom-of-stack marker Z_0 and we are in state q_1, then we have found an input of the form ww^R. We go to state q_2 and accept.

A Graphical Notation for DFA:

Sometimes, a diagram generalizing the transition diagram of a finite automaton, will make aspects of the behavior of a given PDA clearer. A transition diagram for PDA indicates,

(a) The nodes correspond to the states of the PDA.
(b) An arrow label start indicates, the start state and doubly circled states are accepting, as for finite automata.
(c) An arc labeled a, X/α from state q to state 'p' means that δ(q, a, X) contains the pair (p,α).



<u>Instantaneous Descriptions(ID) of a PDA:</u>

The ID is defined as a triple (q, w, γ), where,

q – Current state
w – String of input symbols
γ – String of stack symbols

Let P = (Q, Σ, Γ, δ, q_0, Z_0, F) be a PDA. Suppose δ(q, a, X) contains (p,α). Then for all strings 'w' in Σ* and β in Γ*; (q, aw, β) ⊢ (p, w, αβ).

PROBLEMS:

Ex.1:

Construct PDA on the input strings 001010c010100, 001010c011100. PDA can be described as, P = ($\{q_1, q_2\}$, {0, 1, c}, {R, B, G}, δ, q_1, R, $\{q_2\}$), where δ is,

$\delta(q_1, 0, R) = (q_1, BR)$ $\quad\quad$ $\delta(q_1, c, R) = (q_2, R)$

$\delta(q_1, 1, R) = (q_1, GR)$ $\quad\quad$ $\delta(q_1, c, B) = (q_2, B)$

$\delta(q_1, 0, B) = (q_1, BB)$ $\quad\quad$ $\delta(q_1, c, G) = (q_2, G)$

$\delta(q_1, 1, B) = (q_1, GB)$ $\quad\quad$ $\delta(q_2, 0, B) = (q_2, \varepsilon)$

$\delta(q_1, 0, G) = (q_1, BG)$ $\quad\quad$ $\delta(q_2, 1, G) = (q_2, \varepsilon)$

$\delta(q_1, 1, G) = (q_1, GG)$ $\quad\quad$ $\delta(q_2, \varepsilon, R) = (q_2, \varepsilon)$

check whether the string is accepted or not?

Soln:

- w1 = 001010c010100

$(q_1, 001010c010100, R) \vdash (q_1, 01010c010100, BR)$
$\vdash (q_1, 1010c010100, BBR)$
$\vdash (q_1, 010c010100, GBBR)$
$\vdash (q_1, 10c010100, BGBBR)$
$\vdash (q_1, 0c010100, GBGBBR)$
$\vdash (q_1, c010100, BGBGBBR)$
$\vdash (q_2, 010100, BGBGBBR)$
$\vdash (q_2, 10100, GBGBBR)$
$\vdash (q_2, 0100, BGBBR)$
$\vdash (q_2, 100, GBBR)$
$\vdash (q_2, 00, BBR)$
$\vdash (q_2, 0, BR)$
$\vdash (q_2, \varepsilon, R)$
$\vdash (q_2, \varepsilon, \varepsilon)$

$\therefore$ The string is accepted.

- w2 = 001010c011100

$(q_1, 001010c011100, R) \vdash (q_1, 01010c011100, BR)$
$\vdash (q_1, 1010c011100, BBR)$
$\vdash (q_1, 010c011100, GBBR)$
$\vdash (q_1, 10c011100, BGBBR)$
$\vdash (q_1, 0c011100, GBGBBR)$
$\vdash (q_1, c011100, BGBGBBR)$
$\vdash (q_2, 011100, BGBGBBR)$
$\vdash (q_2, 11100, GBGBBR)$
$\vdash (q_2, 1100, BGBBR)$

$\therefore$ There is no transition for $(q_2, 1, B)$. So the string is not accepted.

LANGUAGES OF A PUSH DOWN AUTOMATA

There are two ways to accept a string a PDA,

(a) Accept by final state that is, reach the final state from the start state.
(b) Accept by an empty stack that is, after consuming input, the stack is empty and current state could be a final state or non-final state.

Both methods are equivalent. One method can be converted to another method and vice versa.

Acceptance by final state:

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. The languages accepted by a final state is defined as,

$$L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha), \text{ where } q \in F \text{ and } \alpha \in \Gamma^*.$$

It means that, from the current station q_0 after scanning the input string 'w', the PDA enters into a final state 'q' leaving the input tape empty. Here contents of the stack is irrelevant.

Acceptance by Empty Stack:

For each PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, language accepted by empty stack can be defined as,

$$N(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon), \text{ where } q \in Q.$$

It means that when the string 'w' is accepted by an empty stack, the final state is irrelevant, the input tape should be empty and stack also should be empty.

From Empty Stack to Final State:

Theorem:

If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

Proof:

Initially, change the stack content from Z_0 to Z_0X_0. So consider a new stack start symbol X_0 for the PDA P_F. Also need a new start state P_0, which is the initial state of P_F. It is to push Z_0 the start symbol of P_N, onto the top of the stack and enter state q_0. Finally, we need another new state P_f, that is the accepting state of P_F.

The specification of P_F is as follows:

$$P = (Q \cup \{P_0, P_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, P_0, X_0, \{P_f\})$$

where δ_F is defined by,

1. $\delta_F(P_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$. In its start state, P_F makes a spontaneous transition to the start state of P_N, pushing its start symbol Z_0 onto the stack.
2. For all states 'q' in Q, inputs 'a' in Σ or $a = \varepsilon$, and stack symbols Y in Γ, $\delta_F(q, a, Y) = \delta_N(q, a, Y)$.
3. $\delta_F(q, \varepsilon, X_0) = (P_f, \varepsilon)$ for every state 'q' in Q.

We must show that 'w' is in $L(P_F)$ if and only if 'w' is in $N(P_N)$. The moves of the PDA P_F to accept a string 'w' can be written as,

$$(P_0, w, X_0) \vdash^{P_F} (q_0, w, Z_0X_0) \vdash^{*P_F} (q_0, \varepsilon, X_0) \vdash^{P_F} (P_f, \varepsilon, \varepsilon).$$

Thus P_F accepts 'w' by final state.

From Final State to Empty Stack:

Theorem:

Let L be $L(P_F)$ for some PDA, $P_F = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

Proof:

Initially, change the stack content from Z_0 to Z_0X_0. So we also need a start state P_0, and final state P, which is the start and final of P_N.

The specification of P_N is as follows:

$$P_N = (Q \cup \{P_0, P\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, P_0, X_0)$$

where δ_N is defined by,

1. $\delta_N(P_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$ to change the stack content initially.
2. $\delta_N(q, a, Y) = \delta_F(q, a, Y)$, for all states 'q' in Q, inputs 'a' in Σ or a = ε, and stack symbols Y in Γ.
3. $\delta_N(q, \varepsilon, Y) = (P, \varepsilon)$, for all accepting states 'q' in F and stack symbols Y in Γ or Y = X_0.
4. $\delta_N(P, \varepsilon, Y) = (P, \varepsilon)$, for all stack symbols Y in Γ or Y = X_0, to pop the remaining stack contents.

Suppose $(Q_0, w, Z_0) \vdash^* {}^{P_F} (q, \varepsilon, \alpha)$ for some accepting state 'q' and stack string α. Then P_N can do the following:

$$(P_0, w, X_0) \vdash {}^{P_N} (q_0, w, Z_0X_0) \vdash^* {}^{P_N} (q, \varepsilon, \alpha X_0) \vdash^* {}^{P_N} (P, \varepsilon, \varepsilon).$$

PROBLEMS:

Ex.1:

Construct a PDA that accepts the given language, $L = \{x^m y^n \mid n<m\}$.

Soln:

Language L accepted by the strings are, L = {xxy, xxxy, xxxyy, xxxxyy,……}

First find the grammar for that language. The grammar for the language can be,

$S \rightarrow xSy \mid xS \mid x$

The corresponding PDA for the above grammar is,

$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

where, $Q = \{q\}$

$\Sigma = \{x, y\}$

$\Gamma = \{S, x, y\}$

$q_0 = \{q\}$

$Z_0 = \{S\}$

$F = \phi$

and δ is defined as,

$\delta(q, \varepsilon, S) = \{(q, xSy), (q, xS), (q, x)\}$

$\delta(q, x, x) = (q, \varepsilon)$

$\delta(q, y, y) = (q, \varepsilon)$

To prove the string xxxyy is accepted by PDA,

$(q, xxxyy, S) \vdash (q, xxxyy, xSy) \vdash (q, xxyy, Sy) \vdash (q, xxyy, xSyy) \vdash (q, xyy, Syy)$
$\vdash (q, xyy, xyy) \vdash (q, yy, yy) \vdash (q, y, y) \vdash (q, \varepsilon, \varepsilon)$

Hence the string is accepted.

EQUIVALENCE OF PUSHDOWN AUTOMATA AND CFG

From Grammars to PushDown Automata:

It is possible to convert a CFG to PDA and vice versa.

Input:

Context Free Grammar 'G'.

Output:

PDA – P that simulates the leftmost derivations of G. Stack contains all the symbols (variables as well as terminals) of CFG.

Let G = (V, T, P, S) be a CFG. The PDA which accepts L(G) is given by,

P = ({q}, T, V ∪ T, δ, q, S, ϕ) where δ is defined by,

1. For each variable 'A' include a transition δ(q, ε, A) = (q, b) such that A → b is a production of P.
3. For each terminal 'a' include a transition δ(q, a, a) = (q, ε).

PROBLEMS:

Ex.1:

Construct a PDA that accepts the language generated by the grammar,

S → aSbb | abb

Soln:

PDA – P is defined as follows:

P = (Q, Σ, Γ, δ, q_0, Z_0, F)

Where, Q = {q}

Σ = {a, b}

Γ = {S, a, b}

q_0 = {q}

Z_0 = {S}

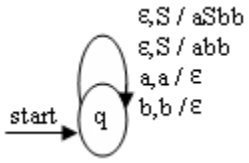
F = ϕ

and δ is defined as,

δ(q, ε, S) = {(q, aSbb), (q, abb)}

δ(q, a, a) = (q, ε)

δ(q, b, b) = (q, ε)



From PDA's to Grammars:

Theorem:

If L is N(M) for some PDA M, then L is a context free grammar.

Construction:

Let M = (Q, Σ, Γ, δ, q_0, Z_0, F) be the PDA . Let G = (V, T, P, S) be a CFG.

Where, - V is the set of objects of the form [q, A, p], 'q' and 'p' in Q and A in Γ.

- New symbol S.
- P is the set of productions.

The productions are,

(1) S → [q_0, Z_0, q], q in Q

(2) If δ(q, a, A) = (q_1, B_1B_2 …… B_n) then,

$[q, A, q_{m+1}] = a[q_1, B_1, q_2][q_2, B_2, q_3] \ldots\ldots\ldots\ldots [q_m, B_m, q_{m+1}]$, for each 'a' in $\Sigma \cup \{\varepsilon\}$ and A, $B_1B_2 \ldots\ldots B_m$ in Γ.

Proof:

If m = 0; $\delta(q, a, A) = (q_1, \varepsilon)$

$[q, A, q_1] \rightarrow a$

Let 'x' be the input string, to show that, $[q, A, p] \overset{*}{\Rightarrow} x$, iff $(q, x, A) \vdash^* (p, \varepsilon, \varepsilon)$

We show by induction on 'i' that, if $(q, x, A) \vdash^i (p, \varepsilon, \varepsilon)$ then $[q, A, p] \overset{*}{\Rightarrow} x$

Basis: when i = 1,

$\delta(q, x, A) = (p, \varepsilon)$

here 'x' is a single input symbol. Thus $[q, A, p] \rightarrow x$ is a production of G.

Induction: when i > 1, let x = ay.

$(q, ay, A) \vdash (q1, ay, B_1B_2 \ldots\ldots..B_n)$

The string 'y' can be written as $y = y_1y_2 \ldots..y_n$, where y_j has the effect of popping B_j from the stack possibly after a long sequence of moves.

Let y_1 be the prefix of 'y' at the end of which the stack first becomes short as n-1 symbols. Let y_2 be the symbols of 'y' following y_1 such that at the end of y_2 the stack is a short as n-2 symbols and so on. That is,

$(q_1, y_1y_2\ldots\ldots y_n, B_1B_2\ldots\ldots B_n) \vdash (q_2, y_2y_3\ldots\ldots y_n, B_2B_3\ldots\ldots B_n) \vdash (q_3, y_3y_4\ldots\ldots y_n, B_3B_4\ldots\ldots B_n)$

There exist states $q_2, q_3, \ldots\ldots..q_{n+1}$.

Where $q_{n+1} = p$

$(q_1, y_1, B_1) \vdash (q_2, \varepsilon, \varepsilon)$

$(q_2, y_2, B_2) \vdash (q_3, \varepsilon, \varepsilon)$

. . .

. . .

. . .

$(q_j, y_j, B_j) \vdash (q_j, \varepsilon, \varepsilon)$

In CFG, $[q_j, B_j, q_j+1] \overset{*}{\Rightarrow} y_j$

The original move,

$(q, ay, A) \vdash (q_1, y, B_1B_2\ldots\ldots B_n)$

$(q, ay_1y_2 \ldots\ldots. y_n, A) \vdash (q_1, y_1y_2\ldots\ldots y_n, B_1B_2\ldots\ldots..B_n)$

CFG is,

$[q, A, p] \Rightarrow a[q_1, B_1, q_2] [q_2, B_2, q_3] \ldots\ldots.. [q_n, B_n, q_{n+1}]$

$[q, A, p] \overset{*}{\Rightarrow} ay_1y_2 \ldots\ldots. y_n$

$[q, A, p] \overset{*}{\Rightarrow} ay$

$[q, A, p] \overset{*}{\Rightarrow} x$ iff $(q, x, A) \vdash^* (p, \varepsilon, \varepsilon)$, where $q_{n+1} = p$.

Algorithm for getting production rules of CFG:

1. The start symbol production can be, $S \rightarrow [q_0, Z_0, q]$

where, q indicates the next state.

q_0 is a start state

Z_0 is a stack symbol

q and $q_0 \in Q$

2. If there exist a move of PDA, $\delta(q, a, Z) = \{(q', \varepsilon)\}$, then the production rule can be written as, $[q, Z, q'] \rightarrow a$
3. If there exist a move of PDA as, $\delta(q, a, Z) = \{(q_m, Z_1Z_2\ldots\ldots Z_n)\}$, then the production rule can be written as, $[q, Z, q_m] \rightarrow a[q_1, Z_1, q_2]\ [q_2, Z_2, q_3]\ [q_3, Z_3, q_4]\ \ldots\ldots\ [q_{m-1}, Z_{n-1}, q_m]$

PROBLEMS:

Ex.1:

Construct a CFG for the PDA, $P = (\{q_0, q_1\}, \{0, 1\}, \{S, A\}, \delta, q_0, S, \{q_1\})$, where δ is,

$\delta(q_0, 1, S) = \{(q_0, AS)\}$ $\quad$ $\delta(q_0, 0, A) = \{(q_1, A)\}$

$\delta(q_0, \varepsilon, S) = \{(q_0, \varepsilon)\}$ $\quad$ $\delta(q_1, 1, A) = \{(q_1, \varepsilon)\}$

$\delta(q_0, 1, A) = \{(q_0, AA)$ $\quad$ $\delta(q_1, 0, S) = \{(q_0, S)\}$

Soln:

CFG, G is defined as, $G = (V, T, P, S)$

Where, $V = \{[q_0, S, q_0], [q_0, S, q_1], [q_1, S, q_0], [q_1, S, q_1], [q_0, A, q_0], [q_0, A, q_1], [q_1, A, q_0], [q_1, A, q_1]\}$

$T = \{0, 1\}$

$S = \{S\}$ [Start stack symbol]

To find production, P;

(1) Production for S,

$S \rightarrow [q_0, S, q_0]$

$S \rightarrow [q_0, S, q_1]$ $\quad$ [q_0 – Start state, S – Initial stack symbol]

(2) $\delta(q_0, 1, S) = \{(q_0, AS)\}$ we get,

For q_0, $[q_0, S, q_0] \rightarrow 1[q_0, A, q_0]\ [q_0, S, q_0]$

$[q_0, S, q_0] \rightarrow 1[q_0, A, q_1]\ [q_1, S, q_0]$

For q_1, $[q_0, S, q_1] \rightarrow 1[q_0, A, q_0]\ [q_0, S, q_1]$

$[q_0, S, q_1] \rightarrow 1[q_0, A, q_1]\ [q_1, S, q_1]$

(3) $\delta(q_0, \varepsilon, S) = \{(q_0, \varepsilon)\}$

$[q_0, S, q_0] \rightarrow \varepsilon$

(4) $\delta(q_0, 1, A) = \{(q_0, AA)$

For q_0, $[q_0, A, q_0] \rightarrow 1[q_0, A, q_0]\ [q_0, A, q_0]$

$[q_0, A, q_0] \rightarrow 1[q_0, A, q_1]\ [q_1, A, q_0]$

For q_1, $[q_0, A, q_1] \rightarrow 1[q_0, A, q_0]\ [q_0, A, q_1]$

$[q_0, A, q_1] \rightarrow 1[q_0, A, q_1]\ [q_1, A, q_1]$

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

For q_0, $[q_0, A, q_0] \rightarrow 0[q_1, A, q_0]$

For q_1, $[q_0, A, q_1] \rightarrow 0[q_1, A, q_1]$

(6) $\delta(q_1, 1, A) = \{(q_1, \varepsilon)\}$

$[q_1, A, q_1] \rightarrow 1$

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

For q_0, $[q_1, S, q_0] \rightarrow 0[q_0, S, q_0]$

For q_1, $[q_1, S, q_1] \rightarrow 0[q_0, S, q_1]$

Since $[q_1, A, q_0]$, $[q_1, A, q_1]$ does not have any productions we can leave them.

After eliminating the unwanted productions,

$S \rightarrow [q_0, S, q_0]$

$[q_0, S, q_0] \rightarrow 1[q_0, A, q_1]\ [q_1, S, q_0]$

$[q_0, S, q_0] \rightarrow \varepsilon$

$[q_0, A, q_1] \rightarrow 1[q_0, A, q_1]\ [q_1, A, q_1]$

$[q_0, A, q_1] \rightarrow 0[q_1, A, q_1]$

$[q_1, A, q_1] \rightarrow 1$

$[q_1, S, q_0] \rightarrow 0[q_0, S, q_0]$

Finally P is given by,

$S \rightarrow [q_0, S, q_0]$
$[q_0, S, q_0] \rightarrow 1[q_0, A, q_1]\ [q_1, S, q_0] \mid \varepsilon$
$[q_0, A, q_1] \rightarrow 1[q_0, A, q_1]\ [q_1, A, q_1] \mid 0[q_1, A, q_1]$
$[q_1, A, q_1] \rightarrow 1$
$[q_1, S, q_0] \rightarrow 0[q_0, S, q_0]$

DETERMINISTIC PUSHDOWN AUTOMATA (DPDA)

Definition of a Deterministic PDA:

A PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ to be deterministic, if and only if the following conditions are met;

(1) $\delta(q, a, X)$ has at most one member for any 'q' in Q, 'a' in Σ or $a=\varepsilon$ and X in Γ.

(2) If $\delta(q, a, X)$ is nonempty, for some 'a' in Σ then $\delta(q, \varepsilon, X)$ must be empty.

Regular Languages and DPDA's:

The DPDA's accept a class of language that is between the regular languages and the CFL's. We shall first prove that the DPDA languages include all the regular languages.

Theorem:

If L is a regular language, then L = L(P) for some DPDA P.

Proof:

A DPDA can simulate a deterministic finite automaton. The PDA keeps some stack symbol Z_0 on its stack because a PDA has to have a stack, but really the PDA ignores its stack and just uses its state. Formally, let $A = (Q, \Sigma, \delta_A, q_0, F)$ be a DFA construct DPDA.

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

By defining $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ for all states 'p' and 'q' in Q, such that $\delta_A(q, a) = p$.

$$\therefore (q_0, w, Z_0) \vdash^{*}{}_{P} (p, \varepsilon, Z_0) \text{ if and only if } \delta A(q_0, w) = p.$$

DPDA's and Context Free Languages:

The languages accepted by DPDA's by final state properly include the regular languages, but are properly included in the CFL's.

UNIT IV
NORMAL FORMS AND TURING MACHINES

Normal forms for CFG – Simplification of CFG- Chomsky Normal Form (CNF) and Greibach Normal Form (GNF) – Pumping lemma for CFL – Closure properties of Context Free Languages –Turing Machine : Basic model – definition and representation – Instantaneous Description – Language acceptance by TM – TM as Computer of Integer functions – Programming techniques for Turing machines (subroutines).

INTRODUCTION

If a language is a context free language, then it should have a grammar in some specific form. In order to obtain the form, we need to simplify the context free grammars.

NORMAL FORMS FOR CFG

Simplification of CFG:

In a CFG, it may not be necessary to use all the symbols in V T on all the productions in P for deriving sentences. So we try to eliminate the symbols and productions in G which are not useful for derivation of sentences. To simplify a CFG we need to eliminate,

(a) Eliminating Useless Symbols.
(b) Eliminating ε – Productions.
(c) Eliminating Unit Productions.

(a) Eliminating Useless Symbols:

The productions from a grammar that can never take part in any derivation is called useless symbols.

Definition:

Let G = (V, T, P, S) be a grammar, A grammar 'X' is useful, if there is a derivation S αXβ

w, where 'w' is in T. A symbol 'X' is not useful, we say it is useless. There are two ways to eliminating useless symbols,

(1) First, eliminate non generating symbols.
(2) Second, eliminate all symbols that are not reachable in the grammar G.

Ex.1:

Eliminate useless symbols from the given grammar,

$S \rightarrow AB \mid a$

$A \rightarrow a$

Soln:

We find that no terminal string is derivable from B. So, to eliminate the symbol B and the production S→AB. After eliminating useless symbols and the productions are,

$S \rightarrow a$

$A \rightarrow a$

(b) Eliminating ε – Productions:

Any productions of CFG of the form $A \rightarrow \varepsilon$ is called ε – production. Any variable A for which the derivations A ε is possible, is called as nullable.

Steps:

(i) Find the set of nullable variables of G, for all productions of the form A→ε. Put A to Vnullable, if $B \rightarrow A_1A_2 \ldots\ldots A_n$, where $A_1, A_2, \ldots\ldots A_n$ are in Vnullable then put B also in Vnullable.

(ii) Construct a new set of production P'. For each production in P, put that production and all the production generated by replacing nullable variables for all possible combinations into P'.

Ex.1:

Eliminate ε – productions from the grammar,

$S \rightarrow abB$

$B \rightarrow Bb \mid \varepsilon$

Soln:

(i) $B \rightarrow \varepsilon$ is a null production. Vnullable = {B}

(ii) Construct the production in P'

$S \rightarrow abB \mid ab$

$B \rightarrow Bb \mid b$

(c) Eliminating Unit Productions:

A production of the form $A \rightarrow B$, where A, B V is called unit productions.

Steps:

(1) Add all non – unit productions (or) to eliminate all unit productions of P into P'.

(2) For each unit production $A \rightarrow B$, add $A \rightarrow \alpha$, to new production set P', where $B \rightarrow \alpha$ is a non – unit production in P.

Ex.1:

Eliminate all unit productions from the grammar,

$S \rightarrow AaB \mid C$ $\quad\quad$ $B \rightarrow A \mid bb$

$A \rightarrow a \mid bC \mid B$ $\quad\quad$ $C \rightarrow a$

Soln:

(1) To eliminate all unit productions to P'.

$S \rightarrow AaB$ $\quad\quad$ $B \rightarrow bb$

$A \rightarrow a \mid bC$ $\quad\quad$ $C \rightarrow a$

(2) $S \rightarrow C$, $A \rightarrow B$ and $B \rightarrow A$ are unit productions, they are derivable, hence removing unit productions to get,

$S \rightarrow AaB \mid a$ $\quad\quad$ $B \rightarrow a \mid bC \mid bb$

$A \rightarrow a \mid bC \mid bb$ $\quad\quad$ $C \rightarrow a$

Types of Normal Forms:

There are two normal forms, they are,

(1) Chomsky Normal Form (CNF)

(2) Greibach Normal Form (GNF)

(1) Chomsky Normal Form:

Every CFL is generated by a CFG in which all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B, C are variables, and 'a' is a terminal. This form is called Chomsky Normal Form(CNF).

Rules for Converting a Grammar into CNF:

(i) Simplify the grammar by, eliminating ε – productions, unit productions and useless symbols.
(ii) Add all the productions of the form $A \rightarrow BC$ and $A \rightarrow a$ to the new production set P'.
(iii) Consider a production $A \rightarrow A_1A_2 \ldots\ldots A_n$, if A_i is a terminal say a_i then add a new variable C_{ai} to the set of variables, say V' and a new production $C_{ai} \rightarrow a_i$ to the new set of production P'. Replace A_i and A production of P by C_{ai}.
(iv) Consider $A \rightarrow A_1A_2 \ldots\ldots\ldots A_n$, where $n \geq 3$ and all A_i's are variables then introduce new productions. $A \rightarrow X_1C_1$, $C_1 \rightarrow X_2C_2$, …….. $C_{n-2} \rightarrow X_{n-1}C_n$ to the new set of productions P' and the new variables $C_1, C_2, \ldots\ldots\ldots C_{n-2}$ into new set of variables V'.

PROBLEMS:

Ex.1:

Convert the following grammar into CNF,

$S \rightarrow AAC$
$A \rightarrow aAb \mid \varepsilon$
$C \rightarrow aC \mid a$

Soln:

Step 1: Simplify the Grammar:

- **Eliminate ε – Productions:**
  - Find nullable variables Vnullable = {A}
  - Construct the production in P'.

  P' : $S \rightarrow AAC \mid AC \mid C$
  $A \rightarrow aAb \mid ab$
  $C \rightarrow aC \mid a$

- **Eliminate Unit Productions:**

  $S \rightarrow C$ is a Unit Production. Replace C by its productions,

  $S \rightarrow AAC \mid AC \mid aC \mid a$
  $A \rightarrow aAb \mid ab$
  $C \rightarrow aC \mid a$

- **Eliminate Useless Symbols:**

  There is no useless symbols. All the variables are generating terminal string.

Step 2: Reduce the given grammar into CNF:

Add all the productions of the form $A \rightarrow BC$ or $A \rightarrow a$.

- $S \rightarrow AAC$

$S \rightarrow AD_1$; $D_1 \rightarrow AC$

- $S \rightarrow aC$
  $S \rightarrow C_aC$; $C_a \rightarrow a$
- $A \rightarrow ab$
  $A \rightarrow C_aC_b$
- $A \rightarrow aAb$
  $A \rightarrow C_aAC_b$; $C_b \rightarrow b$
  $A \rightarrow C_aD_2$; $D_2 \rightarrow AC_b$
- $C \rightarrow aC$
  $C \rightarrow C_aC$

The Resultant Grammar is,

$S \rightarrow AD_1 \mid AC \mid C_aC \mid a$
$D_1 \rightarrow AC$
$C_a \rightarrow a$
$A \rightarrow C_aD_2 \mid C_aC_b$
$D_2 \rightarrow AC_b$
$C_b \rightarrow b$
$C \rightarrow C_aC \mid a$

(2) Greibach Normal Form (GNF)

A context free grammar G is reduced to GNF of every productions of the form A→aα., where 'a' is terminal, $\alpha \in \in (V \cup T)$ also 'α' can be empty if α→ε then A→α. The construction of GNF depends on two lemmas.

Lemma 1:

Let G = (V, T, P, S) be a CFG.

Let $A \rightarrow \alpha_1 B \alpha_2$ is a production in G.

Let $B \rightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots\ldots \mid \beta_m$.

Then a new grammar G_1 can be constructed by replacing 'B' by its productions.

$\therefore A \rightarrow \alpha_1\beta_1\alpha_2 \mid \alpha_1\beta_2\alpha_2 \mid \ldots\ldots\ldots \alpha_1\beta_m\alpha_2$

Lemma 2:

Let G = (V, T, P, S) be a CFG. Let the set of A – productions be,

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

$A \rightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots\ldots \mid \beta_n$

(or)

$A \rightarrow A\alpha_i \mid \beta_i$

Introduce a new variable say 'B', 'P_1' can be formed by replacing the A – production by,

$A \rightarrow A\alpha \mid \beta$
$A \rightarrow \beta \mid \beta B$
$B \rightarrow \alpha \mid \alpha B$

Reducing a CFG to GNF:

To reduce a grammar into GNF form, the following actions have to be performed.

(1) Construct a grammar CFG, G in CNF generating the CFL L. Rename the variables in V as $\{A_1, A_2, \ldots A_n\}$ with start symbol as A_1.

(2) Modify the productions, such that, $A_i \rightarrow A_j\gamma$, where $i < j$.

(3) If $A_k \rightarrow A_j\gamma$ is a production with $j < k$, generated a new set of productions by substituting for A_j.

(4) By repeating this we obtain the productions of the form $A_k \rightarrow A_l\gamma$, $l \geq k$. The production with l=k are replaced according to lemma2 by introducing a new variable.

PROBLEMS:

Ex.1:

Construct a GNF for the following grammar,

$S \rightarrow AA \mid a$

$A \rightarrow SS \mid b$

Soln:

Step 1:

The given grammar is in CNF. Rename the variables 'S' and 'A' as 'A_1' and 'A_2'. Modify the productions,

$A_1 \rightarrow A_2A_2 \mid a$

$A_2 \rightarrow A_1A_1 \mid b$

Step 2:

Check $A_i \rightarrow A_j$, where $i < j$.

- A_1 productions are in the required form. (ie, $i < j$, where $i = 1, j = 2$)
- A_2 productions are not in the required form.

  ie) $A_2 \rightarrow A_1A_1 \mid b$ ($i > j$, where $i = 2, j = 1$)

∴ Replace the leftmost symbol by its productions,

$A_2 \rightarrow (A_2A_2 \mid a)\ A_1 \mid b$

$A_2 \rightarrow A_2A_2A_1 \mid aA_1 \mid b$

Now, the production $A_2 \rightarrow A_2A_2A_1$ is of the form $A_k \rightarrow A_k$.

Step 3:

By introducing a new variable 'B', ie, in the form of;

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta \mid \beta B$

$B \rightarrow \alpha \mid \alpha B$

$A_2 \rightarrow A_2A_2A_1 \mid aA_1 \mid b$

Here, $\alpha = A_2A_1$

$\beta = aA_1 \mid b$

$A_2 \rightarrow aA_1 \mid b \mid (aA_1 \mid b)\ B$

$A_2 \rightarrow aA_1 \mid b \mid aA_1B \mid bB$

$B \rightarrow A_2A_1 \mid A_2A_1B$

Now the productions are,

$$\begin{bmatrix} A_1 \rightarrow A_2A_2 \mid a \\ A_2 \rightarrow aA_1 \mid b \mid aA_1B \mid b \\ B \rightarrow A_2A_1 \mid A_2A_1B \end{bmatrix}$$

Step 4:

- $A_1 \rightarrow A_2A_2 \mid a$, replace leftmost A_2 by,

  $A_1 \rightarrow (aA_1 \mid b \mid aA_1B \mid bB)\ A_2 \mid a$

  $A_1 \rightarrow aA_1A_2 \mid bA_2 \mid aA_1BA_2 \mid bBA_2 \mid a$

- $B \rightarrow A_2A_1 \mid A_2A_1B$, replace leftmost A_2 by,
  $B \rightarrow (aA_1 \mid b \mid aA_1B \mid bB)\ A_1 \mid (aA_1 \mid b \mid aA_1B \mid bB)\ A_1B$
  $B \rightarrow aA_1A_1 \mid bA_1 \mid aA_1BA_1 \mid bBA_1 \mid aA_1A_1B \mid bA_1B \mid aA_1BA_1B \mid bBA_1B$

  ∴ The resultant GNF is,

$A_1 \rightarrow aA_1A_2 \mid bA_2 \mid aA_1BA_2 \mid bBA_2 \mid a$
$A_2 \rightarrow aA_1 \mid b \mid aA_1B \mid bB$
$B \rightarrow aA_1A_1 \mid bA_1 \mid aA_1BA_1 \mid bBA_1 \mid aA_1A_1B \mid bA_1B \mid aA_1BA_1B \mid bBA_1B$

PUMPING LEMMA FOR CFL

Pumping lemma for CFL states that in any sufficiently long string in a CFL, it is possible to find at most two short substrings close together that can be repeated, both of the strings same number of times.

Statement of the Pumping Lemma:

The pumping lemma for CFL's is similar to the pumping lemma for regular language, but we break each string 'z' in the CFL, L into five parts.

Theorem:

Let L be any CFL. Then there is a constant 'n', depending only on L, such that if 'z' is in L and $|z| \geq n$, then we may write z=uvwxy such that,

(1) $|vx| \geq 1$
(2) $|vwx| \leq n$ and
(3) For all $i \geq 0$ uv^iwx^iy is in L.

Proof:

If 'z' is in L(G) and 'z' is long then any parse tree for 'z' must contain a long path. If the parse tree of a word generated by a Chomsky Normal Form grammar has no path of length greater than 'i' then the word is of length no greater than 2^{i-1}.

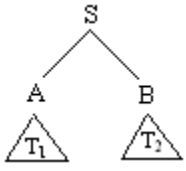
To prove this,

Basis:

Let i=1, the tree must look like,

Induction:

Let i > 1, the root and its sons be,



If there are no paths of length greater than i-1 in trees T_1 and T_2 then the trees generate words of 2^{i-2}. Let G have 'k' variable and let $n=2^k$. If 'z' is in L(G) and $|z| \geq n$ then since $|z| > 2^{k-1}$ any parse for 'z' must have a path of length atleast k+1. But such a path has atleast k+2 vertices. Then there must be some variables that appear twice in a path since there are only 'k' variables.

Let 'P' be a path that is a long than any path in the tree. Then there must be 2 vertices V_1 and V_2 on the path satisfying following conditions,

(1) The vertices V_1 and V_2 both have same label say 'A'.
(2) Vertex V_1 is closer to the root than vertex V_2.
(3) The portion of the path from V_1 to leaf is of length atmost k+1.

The subtree T_1 with root 'r_1' represents the derivation of length atmost 2^k. There is no path in T_1 of length greater than k+1, since 'P' was the longest path.

Applications of the Pumping Lemma:

Pumping Lemma can be used to prove a variety of languages not to be context free. To show that a language L is not context free, we use the following steps;

(i) Assume L is context free. Let 'n' be the natural number obtained by using pumping lemma.
(ii) Choose $|z| \geq n$, write z = uvwxy using the lemma.
(iii) Find a suitable integer 'i' such that $uv^iwx^iy \notin L$. This is a contradiction, and so L is not context free.

PROBLEMS:

Ex.1:

Show that $L = \{a^nb^nc^n \mid n \geq 1\}$ is not context free.

Soln:

Assume L is context free.
L = {abc, aabbcc, aaabbbccc, …….. }
Let z = uvwxy.
Take a string in L = aabbcc [Take any string in L]
To prove, aabbcc is not regular.

Case 1:

z = aabbcc ; n = 6
Now, divide 'z' into uvwxy.
Let u = aa, v = b, w = b, x = ε, y = cc [$\therefore |vx| \geq 1, |vwx| \leq n$]
Find, uv^iwx^iy. When i = 2,
uv^iwx^iy = aabbbcc
$\therefore$ aabbbcc $\notin$ L.
So L is not context free.

Case 2:

z = aabbcc ; n = 6
Now, divide 'z' into uvwxy.
Let u = a, v = a, w = b, x = ε, y = bcc [$\therefore |vx| \geq 1, |vwx| \leq n$]
Find, uv^iwx^iy. When i = 2,
uv^iwx^iy = aaabbcc
$\therefore$ aaabbcc $\notin$ L.
So L is not context free.

CLOSURE PROPERTIES OF CFL

We now consider some operations that preserve context free languages. The operations are useful not only in constructing or proving that certain languages are context free but also in proving certain languages not to be context free.

Closure Under Union:

Theorem:

Context Free Languages are closed under union.

Proof:

Let 'L_1' and 'L_2' be CFL's generated by CFG's.

$G_1 = \{V_1, T_1, P_1, S_1\}$, $G_2 = \{V_2, T_2, P_2, S_2\}$

For $L_1 \cup L_2$, construct grammar G_3,

$G_3 = \{V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, P_3, S_3\}$

Where, P_3 is $P_1 \cup P_2$ plus the production $S_3 \rightarrow S_1 \mid S_2$.

If a string 'w' is in L_1 then the derivation, $S_3 \overset{G_3}{\Rightarrow} \overset{G_3}{\Rightarrow} S_1 \overset{G_1}{\Rightarrow} \overset{G_1}{\Rightarrow}{}^{*}$ w is in derivation in G_3, as every production of G_1 is a production of G_1 is a production of G3.

Thus $L_1 \subset L(G_3)$.

Similarly for a string w_1 in L_2, $S_3 \overset{G_3}{\Rightarrow} \overset{G_3}{\Rightarrow} S_2 \overset{G_2}{\Rightarrow} \overset{G_2}{\Rightarrow}{}^{*}$ w_1 is a derivation in G_3, as every of G_2 is a production of G_3.

Thus $L_2 \subset L(G_3)$. Hence $L(G_3) = L_1 \cup L_2$

Closure Under Concatenation:

Theorem:

Context Free Languages are closed under concatenation.

Proof:

Let 'L_1' and 'L_2' be CFL's generated by CFG's.

$G_1 = \{V_1, T_1, P_1, S_1\}$, $G_2 = \{V_2, T_2, P_2, S_2\}$

For $L_1 . L_2$, construct grammar G_1,

$G_1 = \{V_1 \cup V_2 \cup \{S_4\}, T_1 \cup T_2, P_4, S_4\}$

Where, P_4 is $P_1 \cup P_2$ plus the production $S_4 \rightarrow S_1 S_2$.

$\therefore L(G_4) = L(G_1) . L(G_2)$

Closure Under Kleene Closure:

Theorem:

Context Free Languages are closed under kleene closure.

Proof:

Let 'L_1' and 'L_2' be CFL's generated by CFG's.

G1 = $\{V_1, T_1, P_1, S_1\}$

For closure, let $G_5 = \{V_1 \cup \{S_5\}, T_1, P_5, S_5\}$

Where, P_5 is P_1 plus the production $S_5 \rightarrow S_1 S_5 \mid \varepsilon$.

If a string 'w' is in L then the derivation, $S_5 \Rightarrow S_1 S_5 \Rightarrow wS_5 \Rightarrow w$ $[S_5 \Rightarrow \varepsilon]$

$\therefore L(G_5) = L(G_1)^*$

Closure Under Substitution:

Theorem:

Context Free Languages are closed under substitution.

Proof:

Let L be a CFL, L $\subset$ Σ* and for each 'a' in Σ, let L_a be a CFL. Let L be a L(G) and for each 'a' in Σ, let L_a be $L(G_a)$. Construct a grammar G' as follows;

- The variables of G' are all the variables of G and G_a's.
- The terminals of G' are the terminals of the G_a's.
- The start symbols of G' are all the production of the G_a's.
- The productions of G' are all the productions of the G_a's together with those productions formed by taking a production $A \rightarrow \alpha$ of G and substituting S_a, the start symbol of G_a, for each instance of an 'a' in Σ appearing in α.

Ex:

Let L be the set of words with an equal number of a's and b's,

$L_a = \{0^n1^n \mid n \geq 1\}$ and $L_b = \{ww^R \mid w \text{ in } (0+2)^*\}$

For G we may choose, $S \rightarrow aSbS \mid bSaS \mid \varepsilon$.

For G_a take, $S_a \rightarrow 0S_a1 \mid 01$

For G_b take, $S_b \rightarrow 0S_b0 \mid 2S_b2 \mid \varepsilon$

If 'f' is the substitution $f(a) = L_a$ and $f(b) = L_b$, then, f(L) is generated by grammar G' as,

$S \rightarrow S_aSS_bS \mid S_bSS_aS \mid \varepsilon$

$S_a \rightarrow 0S_a1 \mid 01$

$S_b \rightarrow 0S_b0 \mid 2S_b2 \mid \varepsilon$

Closure Under Intersection:

Theorem:

Context Free Languages are closed under intersection.

Proof:

Let L_1 and L_2 be CFL's, then $L_1 \cap L_2$ is a language, where it satisfies both the properties of L_1 and L_2 which is not possible in CFL.

Ex:

Let $L_1 = \{a^mb^n \mid m \geq 1, n \geq 1\}$, $L_2 = \{a^nb^m \mid n \geq 1, m \geq 1\}$

$L = L_1 \cap L_2$ is not possible.

Because L_1 requires that there be 'm' number of a's and 'n' mumber of b's.

So CFL's are not closed under intersection.

Closure Under Homomorphism:

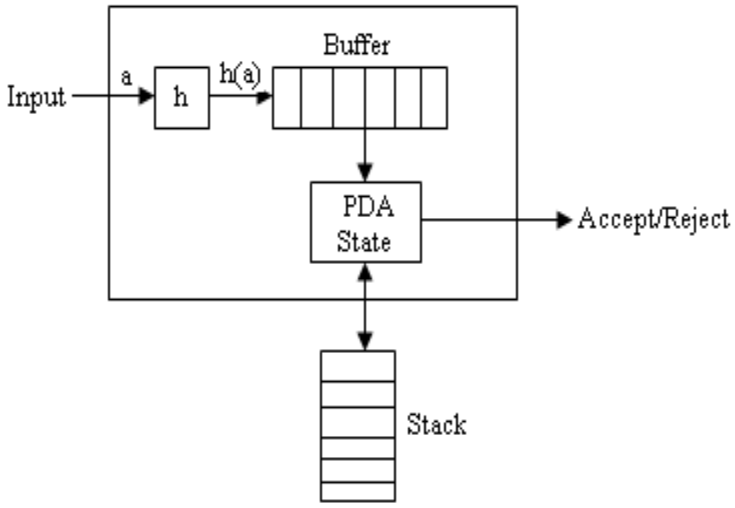
Let L be a CFL over the alphabet Σ and 'h' is a homomorphism on Σ. Let 'S' be the substitution that replaces each symbol 'a' in Σ by the language consisting of one string h(a).

$S(a) = \{h(a)\}$ for all 'a' in Σ.

Thus h(L) = S(L).

Closure Under Inverse Homomorphism:

If 'h' is a homomorphism and 'L' is any language then $h^{-1}(L)$ is the set of strings 'w' such that h(w) is in L. Thus CFL's are closed under inverse homomorphism. The following fig., shows the inverse homomorphism of PDA's.



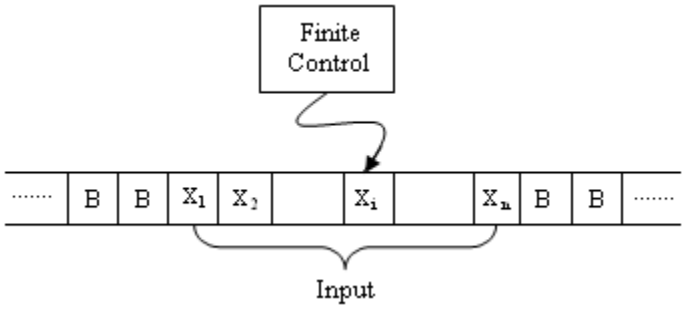
After getting the input 'a', h (a) is placed in a buffer. The symbols of h(a) are used one at a time and fed to the PDA being simulated. While applying homomorphism the PDA checks whether the buffer is empty. If it is empty, then the PDA read the input symbols and applies the homomorphism.

TURING MACHINES (TM)

- A Turing Machine is a simple, abstract mathematical model of a computer.
- It is developed by "Alan Turing", during the year 1936.
- Turing Machine is introduced as a tool for studying computability of mathematical functions.
- Turing Machine is mostly used to define languages and to compute integer functions.

Notation for the Turing Machine:

- The Turing Machine consists of a finite control, and a input tape.
- Finite control has a finite set of states.
- Input tape is divided into cells, each cell can hold any one of a finite number of symbols.



- Initially, the input, which is a finite length of symbols, that is placed on the tape.
- All other tape cells, extending infinitely to the left and, right can hold a special symbol called **Blank**.
- The blank is a tape symbol, but not an input symbol.
- The model of the turing machine also has a **tape head**, that is always positioned at one of the tape cells.

- Initially the tape head is pointing the leftmost cell that holds the input.

Move of the Turing Machine:

In one move, the turing machine depending upon the symbol scanned by the tape head and the state of finite control.

(1) It changes the state.
(2) Writes a tape symbol.
(3) Moves tape head one cell, to its left or right.

Formal Definition of Turing Machine:

A turing machine M has 7 – tuples,

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

Where, Q – The finite set of states.

Σ – The finite set of input symbols.

Γ – Finite set of tape symbols.

δ - The transition function. The arguments of δ is,

$\delta(q, A) = (p, B, L)$

where, 'q' is the current state.

A, B is the input symbols.

L is the direction (L → Left, R → Right)

'p' is the next state.

q_0 – The start state.

B – The blank symbol, blank symbol is in tape symbol, but not in input symbols.

F – The set of final or accepting states.

Instantaneous Descriptions(ID) for Turing Machines:

Instantaneous Description (ID) of the turing machine M is denoted by $\alpha_1 q \alpha_2$. Here 'q' is the current state, α_1 and α_2 is the string. We define a move of M as follows;

Let $X_1 X_2 \ldots\ldots X_{i-1} q X_i X_{i+1} \ldots\ldots X_n$ be an ID, suppose $\delta(q, X_i) = (p, Y, L)$, then, we write move as,

$X_1 X_2 \ldots\ldots X_{i-1} q X_i X_{i+1} \ldots\ldots X_n \vdash X_1 X_2 \ldots\ldots p X_{i-1} Y X_{i+1} \ldots\ldots X_n$

Alternatively, $\delta(q, X_i) = (p, Y, R)$, then, we write move as,

$X_1 X_2 \ldots\ldots X_{i-1} q X_i X_{i+1} \ldots\ldots X_n \vdash X_1 X_2 \ldots\ldots X_{i-1} Y p X_{i+1} \ldots\ldots X_n$

PROBLEMS:

Ex.1:

Design a TM to accept the language $L = \{0^n 1^n \mid n \geq 1\}$

Soln:

Step 1: Place $0^n 1^n$ on the tape following by infinite blank symbols.

Step 2: Replace leftmost '0' by 'X' and move right to find the leftmost '1'.

Step 3: The leftmost '1' is replaced by 'Y', moves left to find rightmost 'X', then moves one cell right to the leftmost '0' and repeats the cycle.

Step 4: While searching, for a '1', if M finds a blank, then M halts without accepting.

Step 5: If after changing '1' to 'Y', M finds no more 0's then M checks whether there is any more 1's left out.

Step 6: If none, then it accepts the string else not.

Let M = (Q, Σ, Γ, δ, q_0, B, F), assume the set of states Q = {q_0, q_1, q_2, q_3, q_4}, Σ = {0, 1}, Γ = {0,1,X,Y,B}, q_0 = {q_0}, F = {q_4}.

Assume n= 2 then the input string is: 0011 [ie., $0^2 1^2$]

(i) 0011BBB ⊢ X011BBB ⊢ X011BBB ⊢ X0Y1BBB ⊢ X0Y1BBB ⊢ X0Y1BBB ⊢ XXY1BBB

$\uparrow q_0$ $\uparrow q_0$ $\uparrow q_1$ $\uparrow q_1$ $\uparrow q_1$ $\uparrow q_1$ $\uparrow q_2$ $\uparrow q_2$ $\uparrow q_2$ $\uparrow q_2$ $\uparrow q_0$

$\uparrow q_0$ $\uparrow q_1$ $\uparrow q_1$

⊢ XXY1BBB ⊢ XXYYBBB ⊢ XXYYBBB ⊢ XXYYBBB ⊢ XXYYBBB ⊢ XXYYBBB

$\uparrow q_1$ $\uparrow q_1$ $\uparrow q_2$ $\uparrow q_2$ $\uparrow q_2$ $\uparrow q_2$ $\uparrow q_0$ $\uparrow q_0$

$\uparrow q_3$ $\uparrow q_3$ $\uparrow q_3$ $\uparrow q_3$

⊢ XXYYBBB

$\uparrow q_4$ $\uparrow q_4$ ∴ Accepted.

(ii) Suppose the input string is 0010 then,

0010BBB ⊢ X010BBB ⊢ X010BBB ⊢ X0Y0BBB ⊢ X0Y0BBB ⊢ X0Y0BBB ⊢ XXY0BBB

$\uparrow q_0$ $\uparrow q_0$ $\uparrow q_1$ $\uparrow q_1$ $\uparrow q_1$ $\uparrow q_1$ $\uparrow q_2$ $\uparrow q_2$ $\uparrow q_2$ $\uparrow q_2$ $\uparrow q_0$

$\uparrow q_0$ $\uparrow q_1$ $\uparrow q_1$

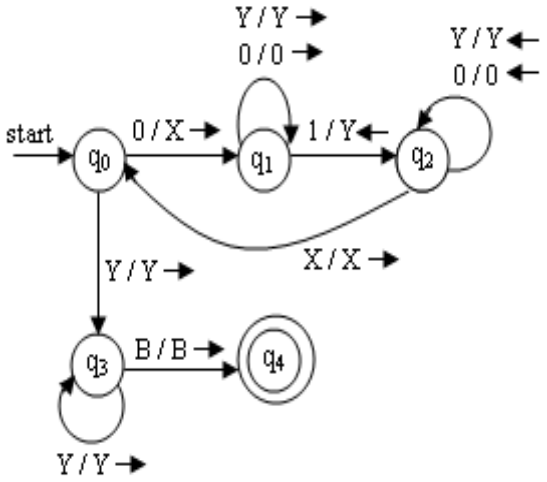
⊢ XXY0BBB ⊢ XXY0BBB

$\uparrow q_1$ $\uparrow q_1$ $\uparrow q_1$ $\uparrow q_1$ ∴ Not accepted.

Transition Table:

State	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	-	-	-	-	-

Transition Diagram:



Ex.1:

Construct a TM that performs addition.

Soln:

Procedure:

- The function is defined as f(x, y) = x+y.
  'x' is given by 0^x.
  'y' is given by 0^y.
- The input is placed on tape as $0^x|0^y$, where '|' is the separator.
- Then the output will be 0^{x+y}.
- Starting from the first zero in the 0^x, the tape head moves till it finds a separator '|' and replaces it by '0', move right to find the blank symbol.
- Then moves left one cell and replace the zero in that cell by a blank symbol.

Let M = (Q, Σ, Γ, δ, q_0, B, F), assume the set of states Q = {q_0, q_1, q_2, q_3}, Σ = {0, 1}, Γ = {0,1,B}, q_0 = {q_0}, F = {q_3}.

Assume x = 3, y = 2 then, input string is: $0^3|0^2$ ==> 000|00BBB

000|00BBB ⊢ 000|00BBB ⊢ 000|00BBB ⊢ 000|00BBB ⊢ 000000BBB ⊢ 000000BBB ⊢ 000000BBB

$\uparrow_{q_0}\uparrow_{q_0}$ $\uparrow_{q_0}\uparrow_{q_0}$ $\uparrow_{q_0}\uparrow_{q_0}$ $\uparrow_{q_0}\uparrow_{q_0}$ $\uparrow_{q_1}\uparrow_{q_1}$ $\uparrow_{q_1}\uparrow_{q_1}$ $\uparrow_{q_1}\uparrow_{q_1}$

⊢ 000000BBB ⊢ 00000BBBB

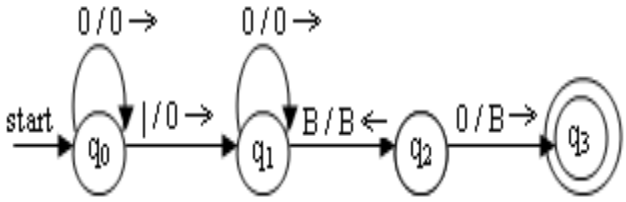
$\uparrow_{q_2}\uparrow_{q_2}$ $\uparrow_{q_3}\uparrow_{q_3}$

Transition Table:

State	0	|	B
→q_0	(q_0,0,R)	(q_1,0,R)	-
q_1	(q_1,0,R)	-	(q_2,B,L)
q_2	(q_3,B,R)	-	-

*q_3	-	-	-

Transition Diagram:



Ex.2:

Construct a TM to compute the function, f(x) = x+1

Soln:

- 'x' is given by 0^x.
- ∴ $f(x) = x+1 = 0^{x+1}$.
- The output contains one more '0' than the input.
- Initially the TM is at q_0.
- At q_0 if it reads a blank symbol by skipping 0's, replace it with '0' and enters final state.

Let M = (Q, Σ, Γ, δ, q_0, B, F), assume the set of states Q = {q_0, q_1}, Σ = {0}, Γ = {0, B}, q_0 = {q_0}, F = {q_1}.

Assume x = 3 then, input string is: 0^3 ==> 000BBB

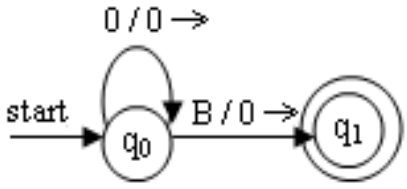
000BBB ⊢ 000BBB ⊢ 000BBB ⊢ 000BBB ⊢ 0000BB

↑q_0 ↑q_0 ↑q_0 ↑q_0 ↑q_0 ↑q_0 ↑q_0 ↑q_0 ↑q_1 ↑q_1

Transition Table:

State	0	B
→q_0	(q_0,0,R)	(q_1,0,R)
*q_1	-	-

Transition Diagram:



Ex.3:

Design a TM to compute proper subtraction.

Soln:

Proper subtraction is defined by $m \dot{-} n$.

ie) $m \dot{-} n = \max(m - n, 0)$

$$m \dot{-} n \begin{cases} m - n & \text{if } m > n \\ 0 & \text{if } m < n \end{cases}$$

Procedure:

- The TM start its operation with $0^m|0^n$ on its input tape.
- Initially the TM is at state q_0.
- At q_0, it replaces the leading '0' by blank and search right looking for first '|'.
- After finding it, the TM searches right for '0' and change it to '|'.
- Then move the tape head to left till reaches the blank symbol. And then enter state q_0 to repeat the cycle.

The repetition ends if:

(1) Searching right for a '0', TM encounters a blank. Then n 0's in $0^m|0^n$ have all been changed to B. Replace the $(n+1)^{th}$ '|' by '0' and n B's. Leaving m-n 0's on its tape.

(2) TM cannot find a '0' to change it to blank during the beginning of the cycle. Change all zero's and 1's to blank and the result in zero.

Let M = (Q, Σ, Γ, δ, q_0, B, F), assume the set of states Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6}, Σ = {0, 1}, Γ = {0,1,B}, q_0 = {q_0}, F = {q_6}.

(i) Assume m=2, n=1 then, Input string is: 00|0

00|0BBB ⊢ B0|0BBB ⊢ B0|0BBB ⊢ B0|0BBB ⊢ B0|1BBB ⊢ B0|1BBB ⊢ B0|1BBB ⊢ B0|1BBB

$\uparrow_{q_0}\uparrow_{q_0}$ $\uparrow_{q_1}\uparrow_{q_1}$ $\uparrow_{q_1}\uparrow_{q_1}$ $\uparrow_{q_2}\uparrow_{q_2}$ $\uparrow_{q_3}\uparrow_{q_3}$ $\uparrow_{q_3}\uparrow_{q_3}$ $\uparrow_{q_3}\uparrow_{q_3}$ $\uparrow_{q_0}\uparrow_{q_0}$

⊢ BB|1BBB ⊢ BB|1BBB ⊢ BB|1BBB ⊢ BB|1BBB ⊢ BB|BBBB ⊢ BBBBBBB

$\uparrow_{q_1}\uparrow_{q_1}$ $\uparrow_{q_2}\uparrow_{q_2}$ $\uparrow_{q_2}\uparrow_{q_2}$ $\uparrow_{q_4}\uparrow_{q_4}$ $\uparrow_{q_4}\uparrow_{q_4}$ $\uparrow_{q_4}\uparrow_{q_4}$

⊢ B0BBBBB

$\uparrow_{q_6}\uparrow_{q_6}$

(ii) Assume m=1, n=2 then, input string is: 0|00

0|00BB ⊢ B|00BB ⊢ B|00BB ⊢ B|10BB ⊢ B|10BB ⊢ B|10BB ⊢ BB10BB ⊢ BBB0BB

$\uparrow_{q_0}\uparrow_{q_0}$ $\uparrow_{q_1}\uparrow_{q_1}$ $\uparrow_{q_2}\uparrow_{q_2}$ $\uparrow_{q_3}\uparrow_{q_3}$ $\uparrow_{q_3}\uparrow_{q_3}$ $\uparrow_{q_0}\uparrow_{q_0}$ $\uparrow_{q_3}\uparrow_{q_3}$ $\uparrow_{q_3}\uparrow_{q_3}$

⊢ BBBBBB ⊢ BBBBBB

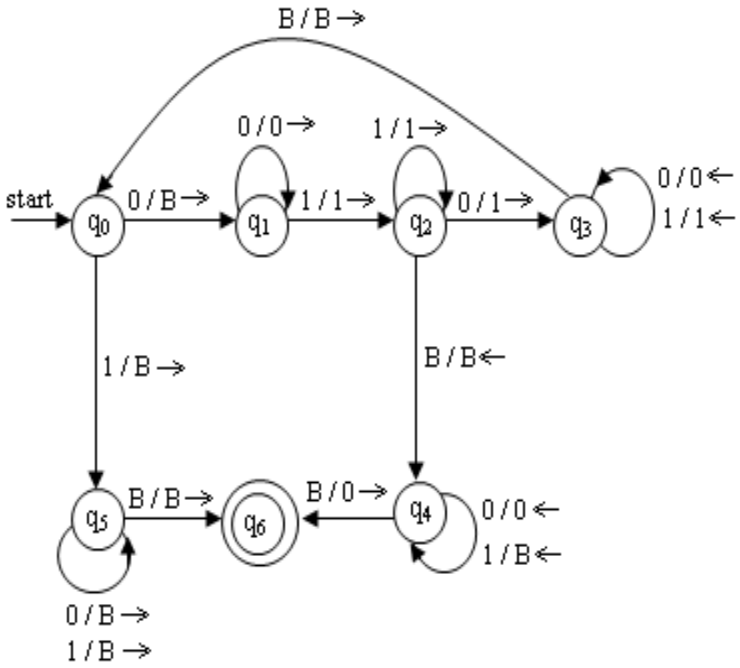
$\uparrow_{q_3}\uparrow_{q_3}$ $\uparrow_{q_6}\uparrow_{q_6}$

Transition Table:

State	0	1	B

$\rightarrow q_0$	(q_1,B,R)	(q_5,B,R)	-
q_1	$(q_1,0,R)$	$(q_2,1,R)$	-
q_2	$(q_3,1,L)$	$(q_2,1,R)$	(q_4,B,L)
q_3	$(q_3,0,L)$	$(q_3,1,L)$	(q_0,B,R)
q_4	$(q_4,0,L)$	(q_4,B,L)	$(q_6,0,R)$
q_5	(q_5,B,R)	(q_5,B,R)	(q_6,B,R)
$*q_6$	-	-	-

Transition Diagram:



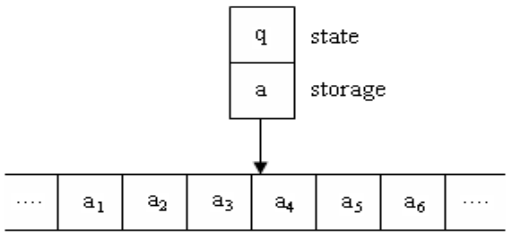
PROGRAMMING TECHNIQUES FOR TM

There are different techniques are used for constructing Turing Machine. They are,

(1) Storage in the state.
(2) Multiple Tracks
(3) Subroutines

(1) **Storage in the state:**

The finite control holds a finite amount of information. Then the state of the finite control is represented as a **pair of elements**. The first element represents the **state** and the second element represents **storing a symbol**.



Ex.1:

Construct a TM, M=(Q, {0,1}, {0,1,B}, δ, [q_0,B], Z0, [q_1,B]), that look at the first input symbol records in the finite control and checks that symbol does not appear else where on its input.

Soln:

For the states of Q as, $Q \times \{0,1,B\} = \{q_0, q_1\} \times \{0,1,B\}$

$Q = \{[q_0, 0], [q_0, 1], [q_0, B], [q_1, 0], [q_1, 1], [q_1, B]\}$

In this, the finite control holds a pair of symbol, that is, both the state and the symbol.

(i) $\delta([q_0, B], a) = ([q_1, a], a, R)$; where, a=0 (or) 1

At 'q_0', the TM reads the first symbol 'a' and goes to state 'q_1'. The input symbol is coped into the second component of the state and moves right.

(ii) $\delta([q_1, a], \bar{a}) = ([q_1, a], \bar{a}, R)$; where, $\bar{a}$ is the complement of 'a'.

ie) if a = 0 then $\bar{a} = 1$

if a = 1 then $\bar{a} = 0$

At q_1, if the TM reads the other symbols, M skips over and moves right.

(iii) $\delta([q_1, a], B) = ([q_1, B], B, R)$

If M reaches the same symbol, it halts without enters accepting.

(iv) $\delta([q_1, a], B) = ([q_1, B], B, R)$

If M reaches the first blank, then it enters the accepting state.

Input String: 011BBB

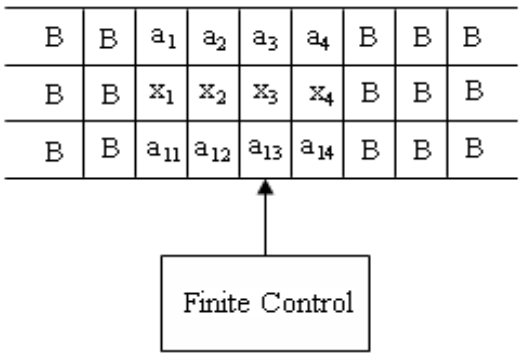
011BBB ⊢ 011BBB ⊢ 011BBB ⊢ 011BBB ⊢ 011BBB

↑ ↑ ↑ ↑ ↑

$[q_0, B]$ $[q_1, 0]$ $[q_1, 0]$ $[q_1, 0]$ $[q_1, B]$

(2) <u>Multiple Tracks:</u>

It is possible that a Turing Machine's input tape can be divided into several tracks. Each track can hold symbols.



Ex.1:

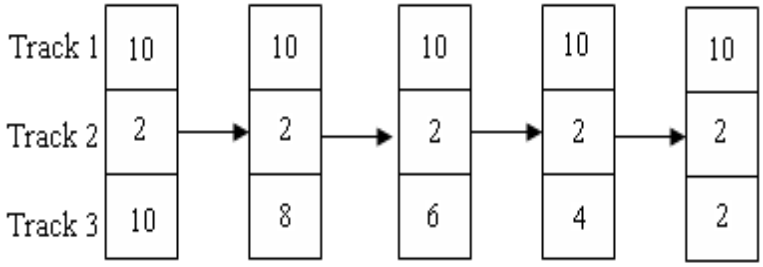
Construct a TM that takes an input greater than 2 and checks whether it is even or odd.

Soln:

Procedure:

(1) The input is placed into first tape or track.
(2) The integer 2 is placed on the second track.
(3) The input on the first track is copied into third track.
(4) The number on the second track is subtracted from the third track.
(5) If the remainder is same as the number in the second track then the number on the first track is even.
(6) If it is greater than 2, then continue this process until the remainder in the third track is <= 2, if it is equal to 2 then the number is even otherwise it is odd.

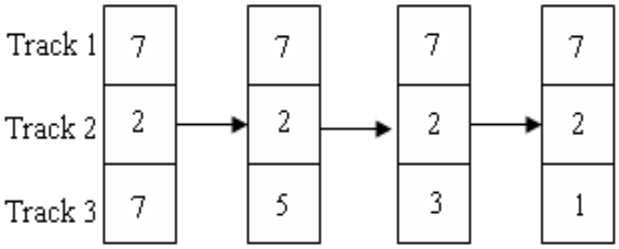
(i) Take the input ==> 10



Finally second track number and the third track number is equal.

∴ The given number is even.

(ii) Assume the input ==> 7



∴ The given number is odd.

Ex.2:

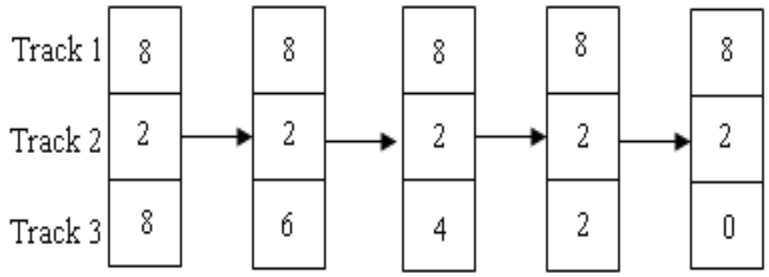
Design a TM that takes an input greater than 2 and checks whether the given input is prime or not.

Soln:

Procedure:

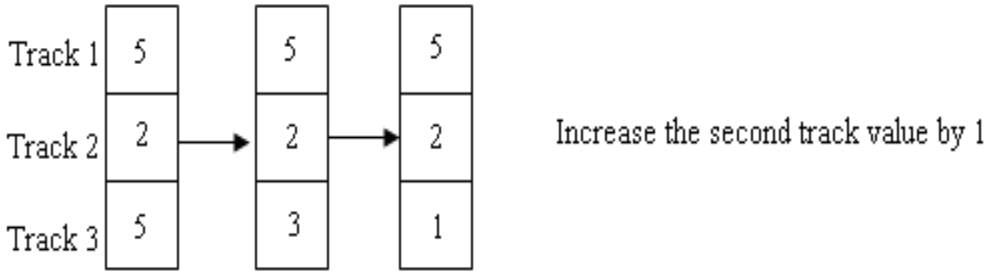
(1) The input is placed into first track.
(2) The integer 2 is placed on the second track.
(3) The input on the first track is copied into third track.
(4) The number on the second track is subtracted from the third track.
(5) If the remainder is zero, then the number on the first track is not a prime.
(6) If the remainder is non – zero, then increase the number on the second track by one.
(7) If the second track equals the first track, then the given number is prime.

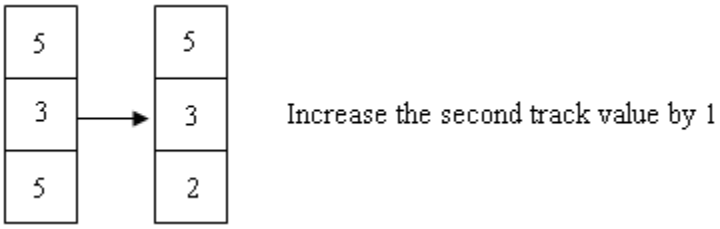
(i) Assume the input ==> 8

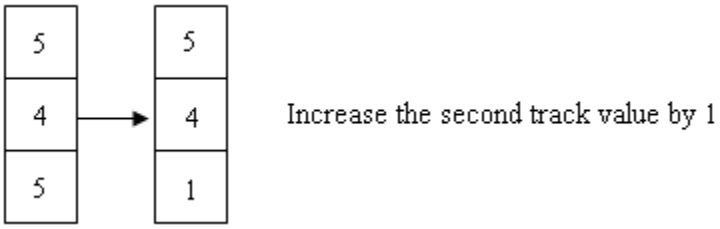


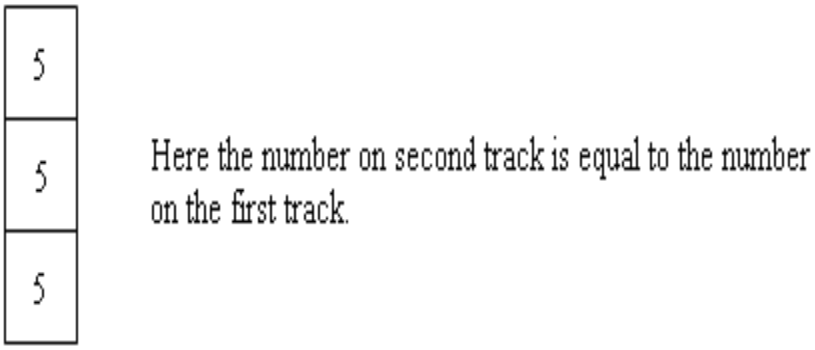
∴ The given number is not a prime number.

(ii) Assume the input ==> 5









∴ The given number is prime.

(3) <u>Subroutines:</u>

Subroutines are used in computer languages, which performs some task repeatedly. A turing machine can simulate any type of subroutine found in programming languages. A part of the TM program can be used as subroutine. This subroutine can be called for any number of times in the main TM program.

Ex:

Design a TM to implement multiplication function, f(m,n) = m*n

Soln:

'm' is given by 0^m

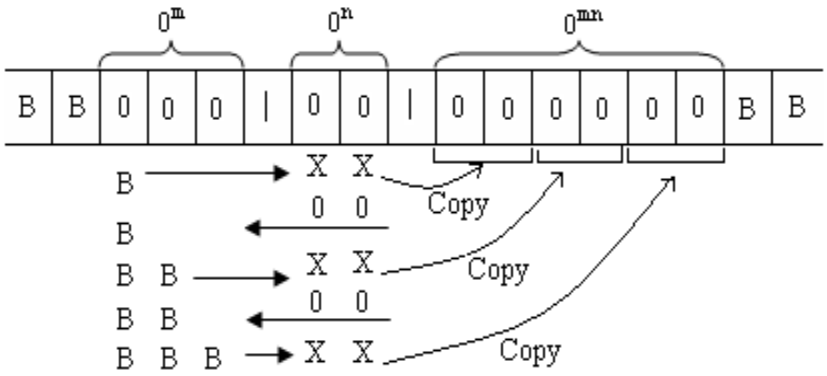
'n' is given by 0^n

Input is : $o^m \mid o^n$

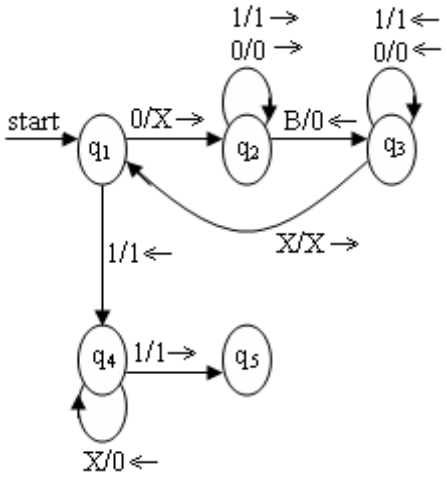
Output is : o^{mn}

Input and output is placed into the tape, that is, $o^m \mid o^n \mid o^{mn}$

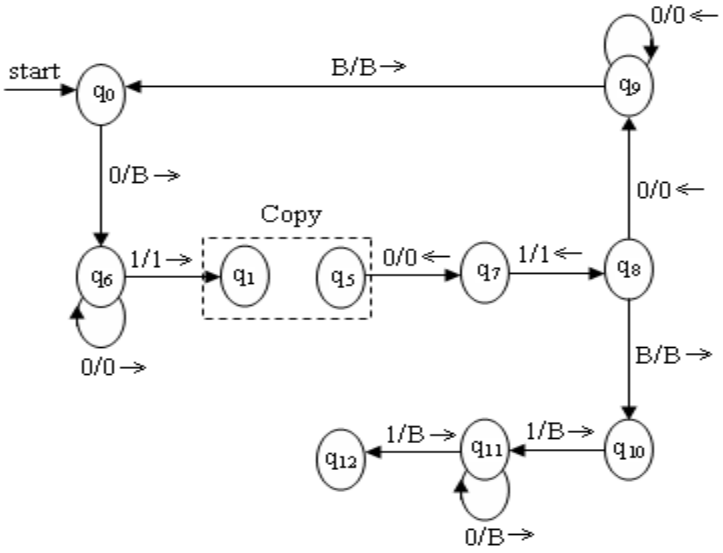
i/p o/p



The main concept is, it copy 'n' zero's 'm' times.

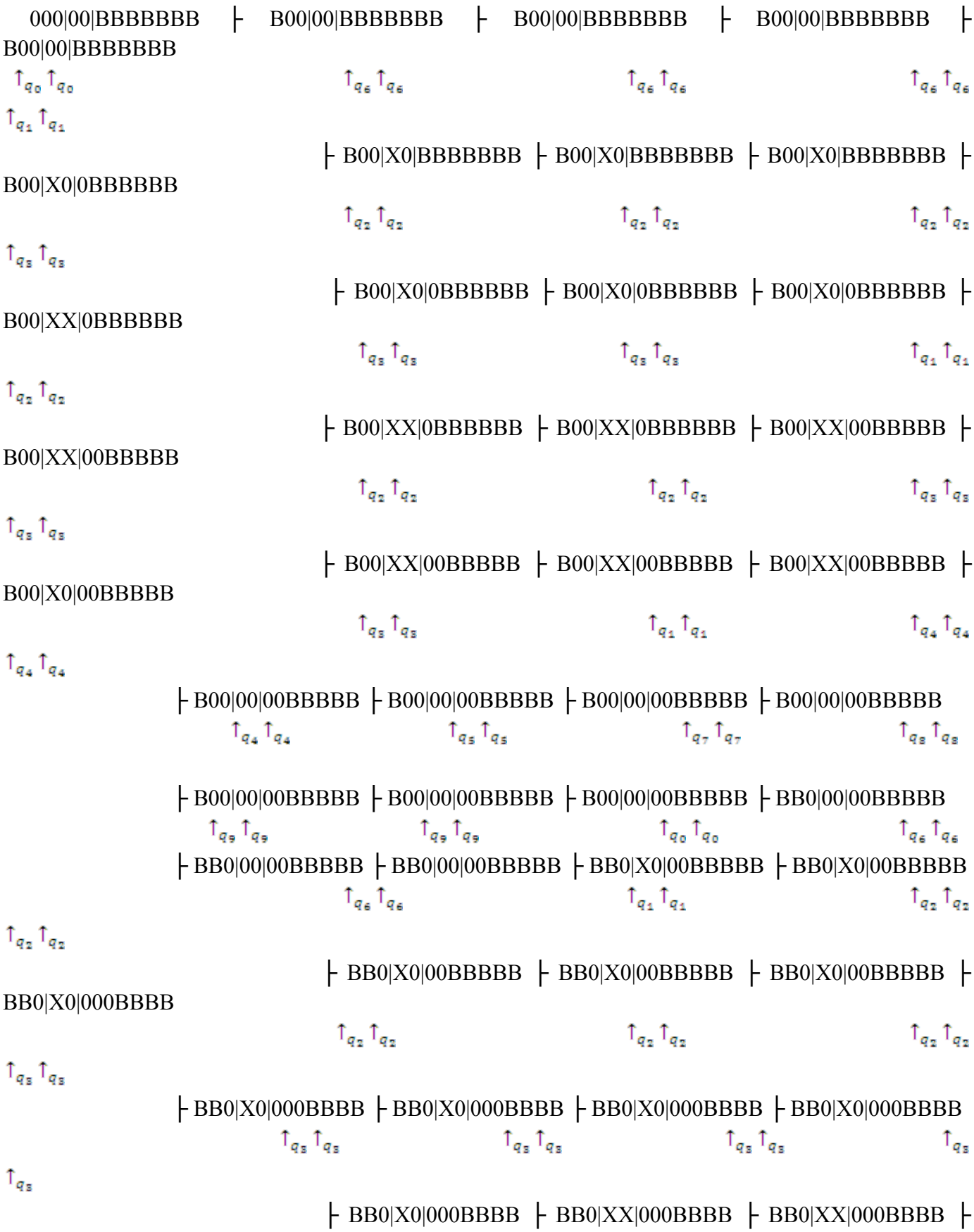


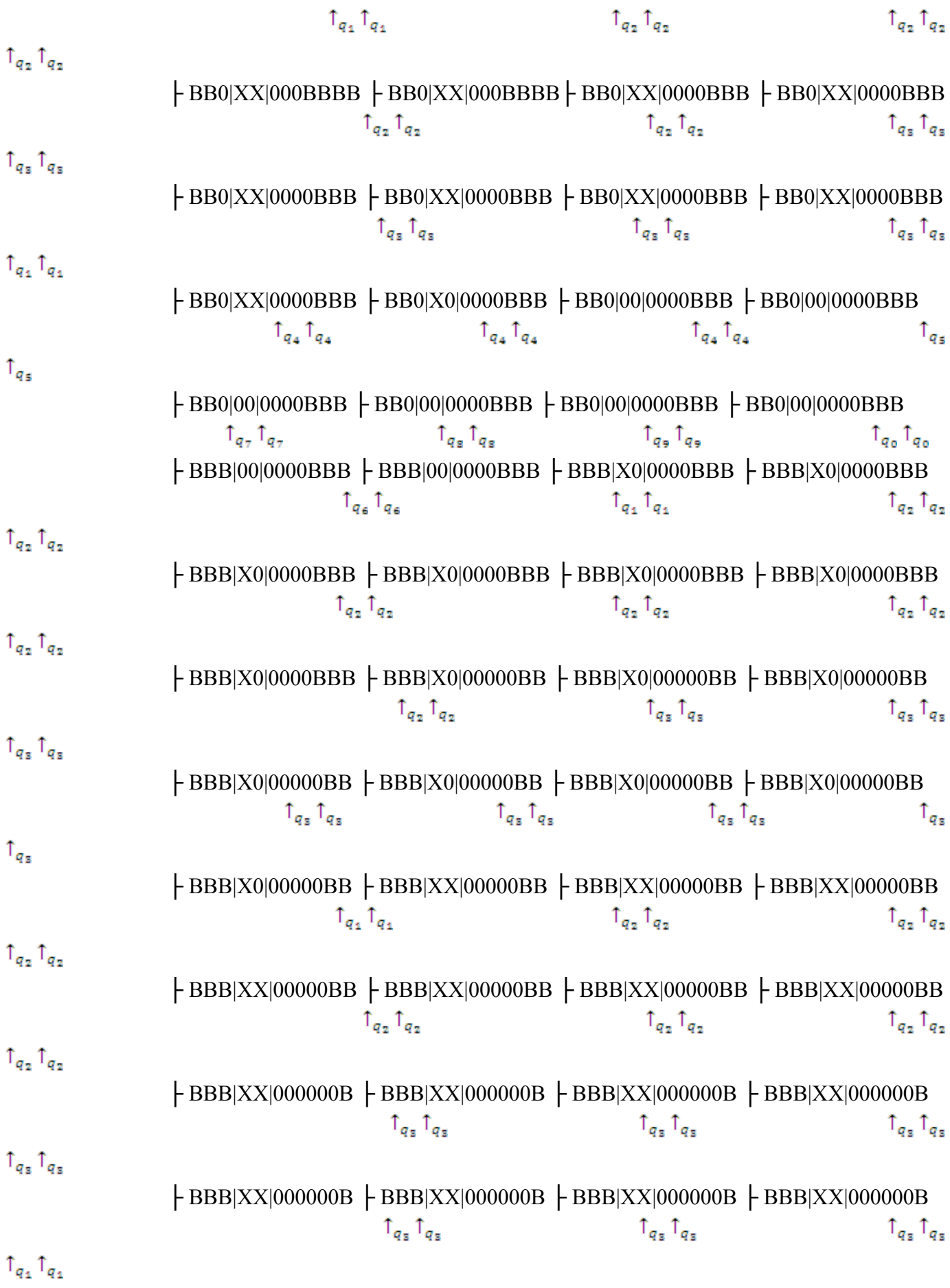
The Subroutine Copy



The Complete Multiplication Program Uses in Subroutine Copy

Assume m = 3, n = 2 then, input is $0^3|0^2$, that is placed on the input tape as,


000|00|BBBBBBB ⊢ B00|00|BBBBBBB ⊢ B00|00|BBBBBBB ⊢ B00|00|BBBBBBB ⊢ B00|00|BBBBBBB
↑q0 ↑q0 ↑q6 ↑q6 ↑q6 ↑q6 ↑q6 ↑q6
↑q1 ↑q1
⊢ B00|X0|BBBBBBB ⊢ B00|X0|BBBBBBB ⊢ B00|X0|BBBBBBB ⊢ B00|X0|0BBBBBB
↑q2 ↑q2 ↑q2 ↑q2 ↑q2 ↑q2
↑q3 ↑q3
⊢ B00|X0|0BBBBBB ⊢ B00|X0|0BBBBBB ⊢ B00|X0|0BBBBBB ⊢ B00|XX|0BBBBBB
↑q3 ↑q3 ↑q3 ↑q3 ↑q1 ↑q1
↑q2 ↑q2
⊢ B00|XX|0BBBBBB ⊢ B00|XX|0BBBBBB ⊢ B00|XX|00BBBBB ⊢ B00|XX|00BBBBB
↑q2 ↑q2 ↑q2 ↑q2 ↑q3 ↑q3
↑q3 ↑q3
⊢ B00|XX|00BBBBB ⊢ B00|XX|00BBBBB ⊢ B00|XX|00BBBBB ⊢ B00|X0|00BBBBB
↑q3 ↑q3 ↑q1 ↑q1 ↑q4 ↑q4
↑q4 ↑q4
⊢ B00|00|00BBBBB ⊢ B00|00|00BBBBB ⊢ B00|00|00BBBBB ⊢ B00|00|00BBBBB
↑q4 ↑q4 ↑q5 ↑q5 ↑q7 ↑q7 ↑q8 ↑q8
⊢ B00|00|00BBBBB ⊢ B00|00|00BBBBB ⊢ B00|00|00BBBBB ⊢ BB0|00|00BBBBB
↑q9 ↑q9 ↑q9 ↑q9 ↑q0 ↑q0 ↑q6 ↑q6
⊢ BB0|00|00BBBBB ⊢ BB0|00|00BBBBB ⊢ BB0|X0|00BBBBB ⊢ BB0|X0|00BBBBB
↑q6 ↑q6 ↑q1 ↑q1 ↑q2 ↑q2
↑q2 ↑q2
⊢ BB0|X0|00BBBBB ⊢ BB0|X0|00BBBBB ⊢ BB0|X0|00BBBBB ⊢ BB0|X0|000BBBB
↑q2 ↑q2 ↑q2 ↑q2 ↑q2 ↑q2
↑q3 ↑q3
⊢ BB0|X0|000BBBB ⊢ BB0|X0|000BBBB ⊢ BB0|X0|000BBBB ⊢ BB0|X0|000BBBB
↑q3 ↑q3 ↑q3 ↑q3 ↑q3 ↑q3 ↑q3
↑q3
⊢ BB0|X0|000BBBB ⊢ BB0|XX|000BBBB ⊢ BB0|XX|000BBBB ⊢ BB0|XX|000BBBB


↑q1 ↑q1 ↑q2 ↑q2 ↑q2 ↑q2
↑q2 ↑q2
├ BB0|XX|000BBBB ├ BB0|XX|000BBBB ├ BB0|XX|0000BBB ├ BB0|XX|0000BBB
↑q2 ↑q2 ↑q2 ↑q2 ↑q3 ↑q3
↑q3 ↑q3
├ BB0|XX|0000BBB ├ BB0|XX|0000BBB ├ BB0|XX|0000BBB ├ BB0|XX|0000BBB
↑q3 ↑q3 ↑q3 ↑q3 ↑q3 ↑q3
↑q1 ↑q1
├ BB0|XX|0000BBB ├ BB0|X0|0000BBB ├ BB0|00|0000BBB ├ BB0|00|0000BBB
↑q4 ↑q4 ↑q4 ↑q4 ↑q4 ↑q4 ↑q5
↑q5
├ BB0|00|0000BBB ├ BB0|00|0000BBB ├ BB0|00|0000BBB ├ BB0|00|0000BBB
↑q7 ↑q7 ↑q8 ↑q8 ↑q9 ↑q9 ↑q0 ↑q0
├ BBB|00|0000BBB ├ BBB|00|0000BBB ├ BBB|X0|0000BBB ├ BBB|X0|0000BBB
↑q6 ↑q6 ↑q1 ↑q1 ↑q2 ↑q2
↑q2 ↑q2
├ BBB|X0|0000BBB ├ BBB|X0|0000BBB ├ BBB|X0|0000BBB ├ BBB|X0|0000BBB
↑q2 ↑q2 ↑q2 ↑q2 ↑q2 ↑q2
↑q2 ↑q2
├ BBB|X0|0000BBB ├ BBB|X0|00000BB ├ BBB|X0|00000BB ├ BBB|X0|00000BB
↑q2 ↑q2 ↑q3 ↑q3 ↑q3 ↑q3
↑q3 ↑q3
├ BBB|X0|00000BB ├ BBB|X0|00000BB ├ BBB|X0|00000BB ├ BBB|X0|00000BB
↑q3 ↑q3 ↑q3 ↑q3 ↑q3 ↑q3 ↑q3
↑q3
├ BBB|X0|00000BB ├ BBB|XX|00000BB ├ BBB|XX|00000BB ├ BBB|XX|00000BB
↑q1 ↑q1 ↑q2 ↑q2 ↑q2 ↑q2
↑q2 ↑q2
├ BBB|XX|00000BB ├ BBB|XX|00000BB ├ BBB|XX|00000BB ├ BBB|XX|00000BB
↑q2 ↑q2 ↑q2 ↑q2 ↑q2 ↑q2
↑q2 ↑q2
├ BBB|XX|000000B ├ BBB|XX|000000B ├ BBB|XX|000000B ├ BBB|XX|000000B
↑q3 ↑q3 ↑q3 ↑q3 ↑q3 ↑q3
↑q3 ↑q3
├ BBB|XX|000000B ├ BBB|XX|000000B ├ BBB|XX|000000B ├ BBB|XX|000000B
↑q3 ↑q3 ↑q3 ↑q3 ↑q3 ↑q3
↑q1 ↑q1

$\vdash$ BBB|XX|000000B $\vdash$ BBB|X0|000000B $\vdash$ BBB|00|000000B $\vdash$ BBB|00|000000B

$\uparrow_{q_4}\uparrow_{q_4}$ $\uparrow_{q_4}\uparrow_{q_4}$ $\uparrow_{q_4}\uparrow_{q_4}$ $\uparrow_{q_5}$

$\uparrow_{q_5}$

$\vdash$ BBB|00|000000B $\vdash$ BBB|00|000000B $\vdash$ BBB|00|000000B $\vdash$ BBBB00|000000B

$\uparrow_{q_7}\uparrow_{q_7}$ $\uparrow_{q_8}\uparrow_{q_8}$ $\uparrow_{q_{10}}\uparrow_{q_{10}}$ $\uparrow_{q_{11}}$

$\uparrow_{q_{11}}$

$\vdash$ BBBBB0|000000B $\vdash$ BBBBBB|000000B $\vdash$ BBBBBBB000000B

$\uparrow_{q_{11}}\uparrow_{q_{11}}$ $\uparrow_{q_{11}}\uparrow_{q_{11}}$ $\uparrow_{q_{12}}\uparrow_{q_{12}}$

UNIT V
UNDECIDABILITY

Unsolvable Problems and Computable Functions –PCP-MPCP- Recursive and recursively enumerable languages – Properties - Universal Turing machine -Tractable and Intractable problems - P and NP completeness – Kruskal's algorithm – Travelling Salesman Problem- 3-CNF SAT problems.

BASIC DEFINITIONS

Decidable Problem:

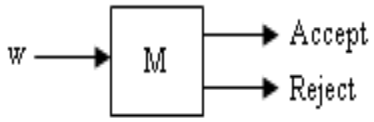
If and only if there exists an algorithm to solve the problem in finite time and determine whether the answer is 'yes' or 'no'.

Undecidable Problem:

If and only if there exists no algorithm to solve the problem in infinite time and determine whether the answer is 'yes' or 'no'.

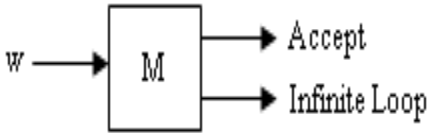
Recursive Language:

A language is recursive if there exists a TM that accepts every string of the language and rejects every string that is not in the language.



Recursively Enumerable Language:

A language is recursively enumerable if there exists a TM that accepts every string of the language, and does not accept strings that are not in the language. The strings that are not in the language may be rejected and it may cause the TM to go to an infinite loop.



NON RECURSIVE ENUMERABLE (RE) LANGUAGE

A language L is recursively enumerable if L = L(M) for some TM M. Recursive or decidable languages that are not only recursively enumerable, but are accepted by a TM.

To prove undecidable, the language consisting of pairs(M, w) such that;

(1) M is a Turing Machine (suitably coded, in binary) with input alphabet {0, 1}.
(2) 'w' is a string of 0's and 1's.
(3) M accepts input 'w'.

If this problem with inputs restricted to the binary alphabet is undecidable, then surely the more general problem, where TM's may have any alphabet, is undecidable.

Enumerating the Binary Strings:

To assign integers to all the binary strings so that each string corresponds to one integer, and each integer corresponds to one string.

Codes for Turing Machines:

A binary code for Turing Machines so that each TM with input alphabet {0, 1} may be thought of as a binary string.

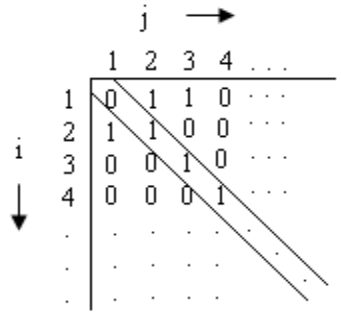
To represent a TM M = (Q, {0, 1}, Γ, $\delta\,\delta$, q_1, B, F) as a binary string, we must first assign integers to the states, tape symbols and directions L and R.

- We shall assume the states are q_1, q_2, ……., q_k for some 'k'. The start state will always be 'q_1' and 'q_2' will be the only accepting state. Note that, since we may assume the TM halts whenever it enters an accepting state, there is never any need for more than one accepting state.
- We shall assume the tape symbols are X_1,X_2,…..,X_m for some 'm'. X_1 always be the symbol '0', X_2 will be '1' and X_3 will be B. However, other tape symbols can be assigned to the remaining integers arbitrarily.
- We shall refer to direction L as D_1 and direction R as D_2.

Once we have established an integer to represent each state, symbol and direction, we can encode the transition function 'δ'. Suppose one transition rule is $\delta(q_i, X_j) = (q_k, X_l, D_m)$, for some integers i, j, k, l and m. We shall code this rule by the string $0^i1o^j1o^k1o^l1o^m$.

The Diagonalization Language (L_d):

The undecidable problem can be proved by the method of diagonalization. Construct a list of words over (0, 1)* in canonical order, where 'w_i' is the i^{th} word and M_j is TM. This can be represented as a table.



Construct a language L_d using the diagonal entries. The value'0' means, w_i is not in $L(M_j)$ and '1' means w_i is in $L(M_j)$.

To construct L_d, we complement the diagonal. For instance, the complemented diagonal would begin 1,0,0,0,…… The trick of complementing the diagonal to construct the characteristic vector of a language that cannot be the language that appears in any row, is called diagonalization.

Proof that L_d is not Recursively Enumerable:

Theorem:

L_d is not a recursively enumerable language. That is, there is no Turning Machine that accepts L_d.

Proof:

Suppose Ld were L(M) for some TM M. Since Ld is a language over alphabet {0,1}, M would be in the list of TM's we have constructed, since it includes all TM's with input alphabet {0,1}. Thus, there is atleast one code for M, say 'i', that is, $M=M_i$.

- If w_i is in L_d, then M_i accepts w_i. But then, by definition of L_d, w_i is not in L_d, because L_d contains only those w_i such that M_j does not accept w_j.
- Similarly, if w_i is not in L_d, then M_i does not accept w_i. Thus, by definition of L_d, w_i is in L_d.

Since w_i can neither be in L_d nor fail to be in L_d, we conclude that there is a contradiction of our assumption that M exists. That is, L_d is not a recursively enumerable language.

PROBLEM:

Ex:

Obtain the code for (M, 1011) where $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0,1,B\}, \delta, q_1, B, \{q_2\})$, δ is,

$\delta(q_1, 1) = (q_3, 0, R)$ $\quad\quad$ $\delta(q_3, 1) = (q_2, 0, R)$

$\delta(q_3, 0) = (q_1, 1, R)$ $\quad\quad$ $\delta(q_3, B) = (q_3, 1, L)$

Soln:

1 – represented as two zero's

0 – represented as one zero

L – represented as one zero

R – represented as two zero's

B – represented as threee zero's

(M, 1011) = 0 | 00 | 000 | 0 | 00 || 000 | 0 | 0 | 00 | 00 || 000 | 00 | 00 | 0 | 00 || 000 | 000 | 000 | 00 | 0

UNDECIDABLE PROBLEM WITH RE

Recursive Language:

A language L recursive if L=L(M) for some TM M such that;

1. If 'w' is in L, then accepts.

2. If ‘w’ is not in L, then M eventually halts, although it never enters an accepting state.

If we think of the language L as a “problem” as will be the case frequently, then problem L is called **decidable** if it is a recursive language, and it is called **undecidable** if it is not a recursive language.

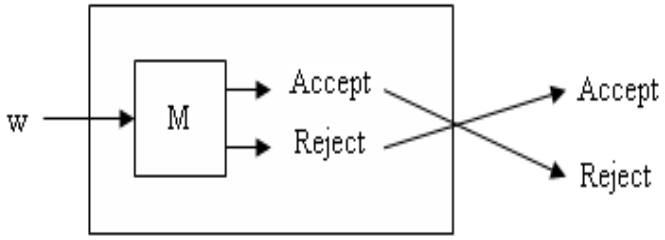
Complements of Recursive and RE languages:

Theorem:

If L is a recursive language, so is $\overline{L}$.

Proof:

Let L = L(M) for some TM M that always halts. We construct a TM M such that $\overline{L}$ = L($\overline{M}$) by the construction.



That is, $\overline{M}$ behaves just like M. However, M is modified as follows to create $\overline{M}$;

(1) The accepting states of M are made non accepting states of $\overline{M}$ with no transitions, ie., in these states $\overline{M}$ will halt without accepting.

(2) $\overline{M}$ has a new accepting state ‘r’; there are no transitions from ‘r’.

(3) For each combination of a non accepting state of M and a tape symbol of M such that M has no transition (ie., M halts without accepting), add a transition to the accepting state ‘r’.

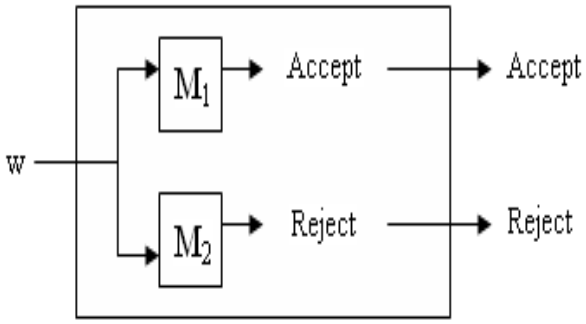
Theroem:

If L and $\overline{L}$ are recursively enumerable then L is recursive.

Proof:

Let M1 be the L and M2 be the $\overline{L}$. Construct M to simulate M1 and M2 simultaneously, since ‘w’ is either in L or $\overline{L}$.

M accepts ‘w’ if M1 accepts it and M rejects ‘w’ if M2 accepts it. Thus M will always say either “Accept” or “reject”. Since M is accepts L. Thus L is recursive.



The important consequences of the properties that we have seen are,

If L and $\overline{L}$ are complementary then only one of the following should be true.

(1) Both L and $\overline{L}$ are recursive.

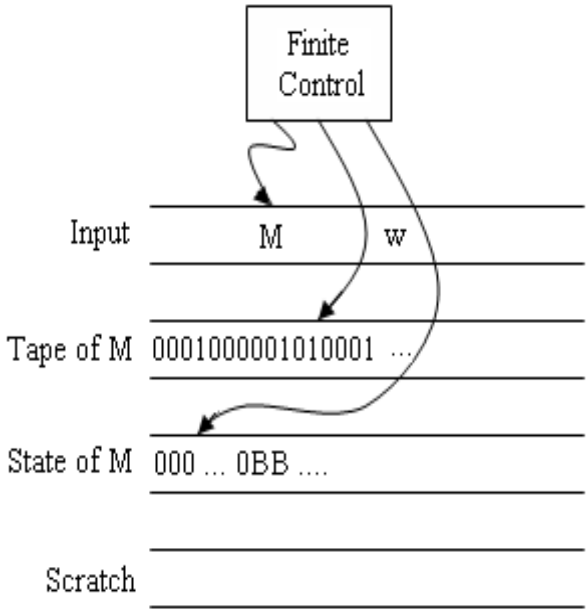
(2) Neither L nor $\overline{L}$ is recursively enumerable.

(3) One of the L is recursively enumerable and other is not recursively enumerable.

The Universal Language:

We define L_u, the universal language, to be the set of binary strings that encode in the pair (M,w), where M is a TM with the binary input alphabet, and 'w' is a string in (0, 1)*, such that 'w' is in L(M). That is, L_u is the set of strings representing a TM and an input accepted by that TM. We shall show that there is a TM U, often called the universal Turing Machine such that L_u = L(U).

- It is easiest to describe U as a multitape TM.
- In the case of U, the transitions of M are stored initially on the first tape, doing with the string 'w'.
- A second tape will be used to hold the simulated tape of M, using the same format as for the code of M.
- That is, tape symbol Xi of M will be represented by 0i, and tape symbols will be separated by single 1's.
- The third tape of U holds the state of M, with state qi represented by 'i', 0's.



The operation of U can be summarized as follows;

(1) Examine the input to make sure that the code for M is a legitimate code for some TM. If not, U halts without accepting. Since invalid codes are assumed to represent the TM with no moves, and such a TM accepts to inputs.

(2) Initialize the second tape to contain the input 'w', in its encoded form. That is, for each '0' of 'w', place 10 on the second tape, and for each '1' of 'w', place 100 there. Note that the blanks on the simulated tape of M, which are represented by 1000.

(3) Place '0', the start state of M, on the third tape, and move the head of U's second tape to the first simulated cell.

(4) To simulate a move of M.
(5) If M has no transition that matches the simulated state and tape symbol, then no transition will be found. Thus, M halts in the simulated configuration.
(6) If M enters its accepting state, then U is accepts.

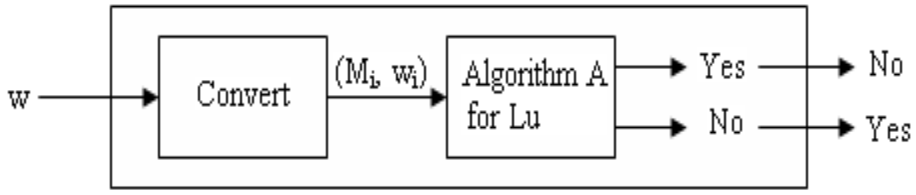
Undecidability of the Universal Language:

Theorem:

Universal Language L_u is not recursive.

Proof:

We know that $\overline{L_d}$ is not recursive, by reducing $\overline{L_d}$ to L_u.



- Assume that L_u is recursive. Then $\overline{L_d}$ must be recursive too.
- Since we know that $\overline{L_d}$ is not recursive we can conclude that L_u is not recursive.
- Construct an algorithm for L_u which accepts (M_i, w_i) if w_i is in $L(M_i)$. Thus we have an $\overline{L_d}$. This is contradiction. Hence L_u is not recursive.

UNDECIDABLE PROBLEMS ABOUT TURING MACHINES

Turing Machines that accept the Empty language:

In this, we are using two languages, called L_e and L_{ne}. Each consists of binary strings. If 'w' is a binary string, then it represents some TM, M_i.

If $L(M_i) = \phi$, that is, Mi does not accept any input, then 'w' is in L_e. Thus, L_e is the language consisting of all those encoded TM's whose language is empty. On the other hand, if $L(M_i)$ is not the empty language, then 'w' is in L_{ne}. Thus, L_{ne} is the language of all codes for Turing Machines that accept atleast one input string. Define the two languages are,

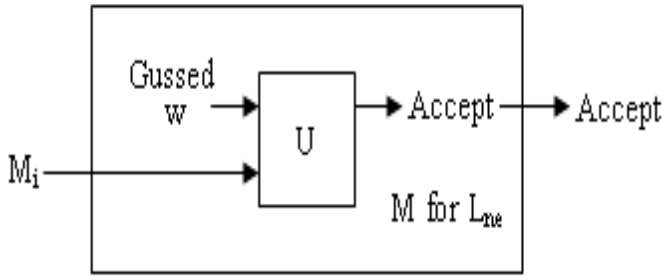
- $L_e = \{M \mid L(M) = \phi\}$
- $L_{ne} = \{M \mid L(M) \neq \phi\}$

Theorem:

L_{ne} is recursively enumerable.

Proof:

In this, a TM that accepts L_{ne}. It is easiest to describe a non deterministic TM M.



The operation of M is as follows;

(1) M takes as input a TM code M_i.
(2) Using its nondeterministic capability, M guesses an input 'w', that Mi might accept.
(3) M test whether M_i accepts 'w'. For this part, M can simulate the Universal TM U that accepts L_u.
(4) If M_i accepts 'w', then M accepts its own input, which is M_i.

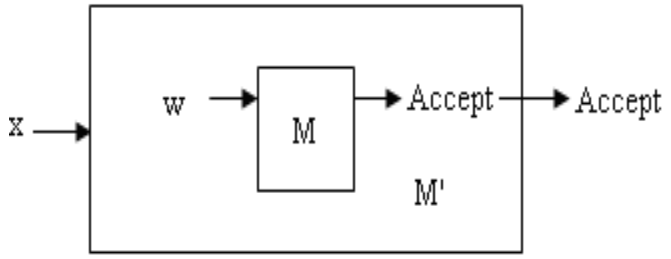
If M_i accepts even one string M will guess that string and accept M_i. However, if $L(M_i) = \phi$, then no guess 'w' leads to acceptance by M_i, so M does not accept M_i. Thus, $L(M) = L_{ne}$.

Theorem:

Lne is not recursive.

Proof:

In this, we must design an algorithm that converts an input that is a binary coded pair(M, w) into a TM M' such that $L(M') \neq \phi$ if and only if M accepts input 'w'. The construction of M' is shown in following fig.



If M does not accept 'w', then M' accepts none of its input's, ie., $L(M') = \phi$. However, if M accepts w, then M' accepts every input, and thus L(M') surely is not ϕ. M' is designed to do the following;

(1) M' ignores its own input 'x'. Rather it replaces its input by the string that represents TM M and input string 'w'. Since M' is designed for a specific pair (M, w) which has some length n, we may construct M' to have a sequence of states $q_0, q_1, \ldots\ldots, q_n$, where q_0 is the start state.
    a. In state q_i, for i=0, 1, ……, n-1, M' writes the (i+1), bit of the code for (M, w) goes to state q_{i+1}, and moves right.
    b. In state q_n, M' moves right, if necessary replacing any nonblanks by blanks.
(2) When M' reaches a blank in state qn, it uses a similar collection of states to reposition its head at the left end of the tape.
(3) Now, using additional states, M' simulates a universal TM U on its present tape.
(4) If U accepts, then M' accepts. If U never accepts, then M' never accepts either.

Rice's Theorem and Properties of the RE Languages:

All nontrivial properties of the RE languages are undecidable, that is, it is impossible to recognize by a TM. The property of the RE languages is simply a set of RE languages. The property of being empty is the set {ϕ) consisting of only the empty language.

A property is trivial if it is either empty, or is all RE languages. Otherwise, it is nontrivial.

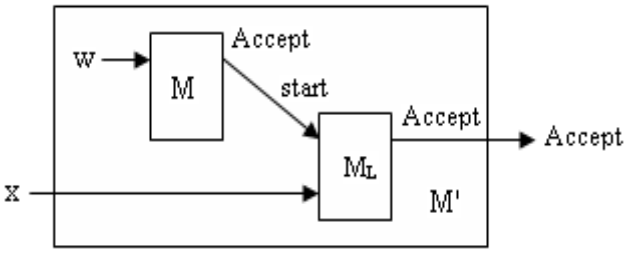
Theorem: (Rice's Theorem)

Every nontrivial property of the RE languages is undecidable.

Proof:

Let P be a nontrivial property of the RE languages. Assume to begin that ϕ, the empty language, is not in P. Since, P is nontrivial, there must be some nonempty language L that is in P. Let M_L be a TM accepting L.

We shall reduce Lu to Lp, thus proving that Lp is undecidable , since Lu is undecidable. The algorithm to perform the reduction takes as input a pair (M, w) and produces a TM M'. The design of M' is given by the following fig.



L(M') is ϕ if M does not accept 'w', and L(M') = L if M accepts 'w'. L(M') = L if M accepts w. The TM M' is constructed to do the following;

(1) Simulate M on input 'w'. Note that 'w' is not the input to M'; rather M' writes M and 'w' onto one of its tapes and simulates the universal TM U on that pair.

(2) If M does not accept 'w', then M' does nothing else. M' never accepts its own input, x, so L(M')=ϕ. Since we assume ϕ is not in property P, that means the code for M' is not in L_P.

(3) If M accepts w, then M' begins simulating M_L on its own input 'x'. Thus M' will accept exactly the language L. Since L is in P, the code for M' is in L_P.

We observe that constructing M' from M and 'w' can be carried out by an algorithm. Since this algorithm turns (M, w) into an M' that is in L_P if and only if (M, w) is in L_u, this algorithm is a reduction of L_u to L_p and proves that the property P is undecidable.

<u>POST's CORRESPONDENCE PROBLEM</u>

In this section, we will discuss the undecidability of strings and not of Turing Machines. The undecidability of strings is determined with the help of Post's Correspondence Problem(PCP).

Our goal is to prove this problem about strings to be undecidable, and then use its undecidability to prove other problems undecidable by reducing PCP to those.

Definition of Post's Correspondence Problem:

An instance of Post's Correspondence Problem(PCP) consists of two lists of strings over some alphabet Σ; the two lists must be of equal length. We generally refer to the A and B lists, and write A

= $w_1, w_2, \ldots\ldots w_k$ and B = $x_1, x_2, \ldots\ldots x_k$, for some integer 'k'. For each 'i', the pair (w_i, x_i) is said to be a corresponding pair.

We say this instance of PCP has a solution, if there is a sequence of one or more integers $i_1,i_2\ldots..i_m$ that, when interpreted as indexes for strings in the A and B lists, yield the same string. That is, $w_{i1}, w_{i2},\ldots\ldots..w_{im} = x_{i1}, x_{i2},\ldots\ldots\ldots x_{im}$. We say the sequence $i_1,i_2, \ldots.., i_m$ is a solution to this instance of PCP.

Ex.1:

Consider the correspondence system as given bellows,

	List A	List B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Does this PCP have a solution?

Soln:

In this case, PCP has a solution. We find, $w_{i1}, w_{i2},\ldots\ldots..w_{im} = x_{i1}, x_{i2},\ldots\ldots\ldots x_{im}$

Let m = 4, ie) $i_1 = 2, i_2 = 1, i_3 = 1, i_4 = 3$

(2) (1) (1) (3)

List A: 10111 1 1 10

List B: 10 111 111 0

ie) $w_{i1}, w_{i2},\ldots\ldots..w_{im} = x_{i1}, x_{i2},\ldots\ldots\ldots x_{im}$

101111110 = 101111110

$\therefore$ The solution list is {2, 1, 1, 3}

The Modified PCP (MPCP):

- It is easier to reduce Lu to PCP, if we first introduce an intermediate version of PCP, which we call the Modified Post's Correspondence Problem or MPCP.
- In the Modified PCP, there is the additional requirement on a solution that the first pair on the A and B lists must be the first pair in the solution.
- An instance of MPCP is two lists A = $w_1, w_2, \ldots\ldots w_k$ and B = $x_1, x_2, \ldots\ldots x_k$, and a solution is a list of '0' or more integers $i_1,i_2\ldots..i_m$ such that, $\mathbf{w_1w_{i1}w_{i2},\ldots\ldots..w_{im} = x_1x_{i1}x_{i2},\ldots\ldots\ldots x_{im}}$.
- Notice that the pair (w_1, x_1) is forced to be at the beginning of the two strings, even though the index '1' is not mentioned at the front of the list that is the solution.
- Unlike PCP, where the solution has to have atleast one integer on the solution list, in MPCP the empty list could be a solution, if $w_1 = x_1$.

Ex:

Consider the correspondence system as given bellows,

	List A	List B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Does this MPCP have a solution?

Soln:

In this case, MPCP has no solution. In proof, observe that any partial solution has to begin with index 1, so the two strings of a solution would begin;

List A: 1 …...

List B: 111…..

Thus, the next index would have to be '1' yielding;

List A: 11 …...

List B: 111111 …..

ie) $w_{i1}, w_{i2},\ldots\ldots..w_{im} = x_{i1}, x_{i2},\ldots\ldots\ldots x_{im}$

11 $\neq$ 111111

$\therefore$ The solution list is {1, 1}

$\therefore$ The B string remains three times as long as the A string, and the two strings can never become equal.

Reducing MPCP to PCP:

An instance of MPCP with alphabet Σ, we construct an instance of PCP $C=y_0,y_1,\ldots\ldots y_{k+1}$, and $D=z_0,z_1,\ldots\ldots..z_{k+1}$ as follows;

(1) First, we introduce a new symbol '*' that, in the PCP instance, goes between every symbol in the strings of the MPCP instance. For i=1,2,…..k, let y_i be w_i with a '*' after each symbol of w_i, and let z_i be xi with a '*' before each symbol of x_i.

(2) $y_0 = *y_1$, and $z_0 = z_1$. That is, the o^{th} pair looks like pair1, expect that there is an extra '*' at the beginning of the string from the first list.

(3) A final pair ($, *$) is added to the PCP instance, $y_{k+1} = \$$ and $z_{k+1} = *\$$

Ex:

An instance of MPCP is defined as,

	List A	List B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Construct an instance of PCP?

Soln:

	List A	List B
i	y_i	z_i
0	*1*	*1*1*1
1	1*	*1*1*1
2	1*0*1*1*1*	*1*0
3	1*0*	*0
4	$	*$

Theorem:

MPCP reduces to PCP.

Proof:

First, suppose that $i_1,i_2…..i_m$ is a solution to the given MPCP instance with lists A and B. Then we know $w_1w_{i1}w_{i2},……..w_{im} = x_1x_{i1}x_{i2},……x_{im}$. If we were to replace the w's by y's and the x's by z's, we would have two strings that were almost the same, $y_1,y_{i1}, y_{i2},……y_{im}$, and $z_1,z_{i1}, z_{i2},……..z_{im}$.

The difference is that the first string would be missing a '*' at the beginning, and the second would be missing a '*' at the end. That is,

$$*y_1,y_{i1}, y_{i2},……y_{im} = z_1,z_{i1}, z_{i2},……..z_{im}*$$

However, $y_0 = *y_1$ and $z_0 = z_1$, so we can fix the initial '*' by replacing the first index by '0'.

$$*y_0,y_{i1}, y_{i2},……y_{im} = z_0,z_{i1}, z_{i2},……..z_{im}*$$

We can take care of the final '*' by appending the index k+1. Since $y_{k+1} = \$$, and $z_{k+1} = *\$$, we have:

$$y_0,y_{i1}, y_{i2},…… y_{im},y_{k+1} = z_0,z_{i1}, z_{i2},…….. z_{im},z_{k+1}$$

We show that $0,i_1,i_2, ….. i_m,k+1$ is a solution to the instance of PCP. We claim that $i_1,i_2…..i_m$ is a solution to the MPCP instance. The reason is that if we remove the *'s and the final $ from the strings,

$$y_0y_{i1}\ y_{i2},…… y_{im}y_{k+1} = z_0z_{i1}\ z_{i2},…….. z_{im}\,z_{k+1}$$

we get $w_1w_{i1}w_{i2},……..w_{im} = x_1x_{i1}x_{i2},……x_{im}$.

Completion of the Proof of PCP Undecidability:

To complete the chain of reductions by reducing Lu to MPCP. That is, given a pair (M, w), we construct an instance (A, B) of MPCP such that TM M accepts input 'w' if and only if (A, B) has a solution.

The essential idea is that MPCP instance (A, B) simulates, in its partial solutions, the computation of M on input 'w'. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM and let w in Σ^* be an input string. We construct an instance of MPCP as follows:

(1) The first pair is:

List A	**List B**
#	$\#q_0w\#$

This pair, which must start any solution according to the rules of MPCP, begins the simulation of M on input 'w'.

(2) Tape symbols and the separator # can be appended to both lists. The pairs,

List A	List B	
X	X	
#	#	for each X in Γ.

(3) To simulate a move of M, we have certain pairs that reflect those moves. For all 'q' in Q-F, p in Q, and X, Y and Z in Γ we have;

List A	List B	
qX	Yp	if $\delta(q, X) = (p, Y, R)$
zqX	pZY	if $\delta(q, X) = (p, Y, L)$; 'Z' is an tape symbol
q#	yp#	if $\delta(q, B) = (p, Y, R)$
zq#	pZY#	if $\delta(q, B) = (p, Y, L)$; 'Z' is an tape symbol

(4) If the ID at the end of the B string has an accepting state, then we need to allow the partial solution to become a complete solution. Thus, if 'q' is an accepting state, then for all tape symbols 'X' and 'Y' there are pairs;

List A	List B
XqX	q
XqY	q
YqX	q
YqY	q
Xq	q
Yq	q
qX	q
qY	q

(5) Finally, once the accepting state has consumed all tape symbols, it stands alone as the last ID on the string. That is the remainder of the two strings is q#. We use the final pair;

List A	List B
q##	#

Ex:

Consider the TM M and w=01, where $M=(\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_3\})$ and δ is given by,

q_i	$\delta(q_i, 0)$	$\delta(q_i, 1)$	$\delta(q_i, B)$
$\rightarrow q_1$	$(q_2, 1, R)$	$(q_2, 0, L)$	$(q_2, 1, L)$
q_2	$(q_3, 0, L)$	$(q_1, 0, R)$	$(q_2, 0, R)$
$*q_3$	-	-	-

Reduce the above problem to PCP and find whether that PCP has a solution or not?

Soln:

Rule	List A	List B	Source
(1)	#	$\#q_1 01\#$	

(2)	0 1 #	0 1 #	
(3)	$q_1 0$ $0 q_1 1$ $1 q_1 1$ $0 q_1 \#$ $1 q_1 \#$ $0 q_2 0$ $1 q_2 0$ $q_2 1$ $q_2 \#$	$1 q_2$ $q_2 00$ $q_2 10$ $q_2 01 \#$ $q_2 11 \#$ $q_3 00$ $q_3 10$ $0 q_1$ $0 q_2 \#$	From $\delta(q_1, 0) = (q_2, 1, R)$ From $\delta(q_1, 1) = (q_2, 0, L)$ From $\delta(q_1, 1) = (q_2, 0, L)$ From $\delta(q_1, B) = (q_2, 1, L)$ From $\delta(q_1, B) = (q_2, 1, L)$ From $\delta(q_2, 0) = (q_3, 0, L)$ From $\delta(q_2, 0) = (q_3, 0, L)$ From $\delta(q_2, 1) = (q_1, 0, R)$ From $\delta(q_2, B) = (q_2, 0, R)$
(4)	$0 q_3 0$ $0 q_3 1$ $1 q_3 0$ $1 q_3 1$ $0 q_3$ $1 q_3$ $q_3 0$ $q_3 1$	q_3 q_3 q_3 q_3 q_3 q_3 q_3 q_3	
(5)	q3##	#	

Now find the sequence of partial solutions, that mimics this computation of M and eventually leads to a solution.

(1) A: #
B: $\#q_1 01\#$

(2) A: $\#q_1 01\#$
B: $\#q_1 01\#1q_2 1\#$

(3) A: $\#q_1 01\#1q_2 1\#$
B: $\#q_1 01\#1q_2 1\#10q_1\#$

(4) A: $\#q_1 01\#1q_2 1\#10q_1\#$
B: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#$

(5) A: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#$
B: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#$

(6) A: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#$
B: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#$

(7) A: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#$
B: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#$

(8) A: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#$
B: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q3\#$

With only q_3 left in the ID, we can use the pair (q_3##, #) from rule(5) to finish the solution.

A: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#\#$
B: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#\#$

∴ The PCP has a solution. Because it produces same string on list A and list B.

Theorem:

Post's Correspondence Problem is undecidable.

Proof:

The construction of this section shows how to reduce L_u to MPCP. Thus, we complete the proof of undecidability of PCP by proving that the construction is correct, that is;

M accepts 'w' if and only if the constructed MPCP instance has a solution. If 'w' is in L(M), then we can start with the pair(M, w) from rule1 to rule5, allow the A string to catch up to the B string and form a solution.

In particular, as long as M does not enter an accepting state, the partial solution is not a solution the B string is longer than the A string. Thus, if there is a solution, M must at some point enter an accepting state. That is M accepts w.

THE CLASSES P AND NP

The classes P and NP of problems solvable in **polynomial time by deterministic and nondeterministic TM's** and the technique of polynomial time reduction. Also define the notion of "NP-Completeness".

Problems Solvable in Polynomial Time:

A TM M is said to be of time complexity T(n). If whenever M is given an input 'w' of length 'n', M halts after making atmost T(n) moves, regardless of whether or not M accepts. Then the language L is in class P if there is some polynomial T(n) such that L = L(M) for some deterministic TM M of time complexity T(n).

An Example: Kruskal's Algorithm:

Kruskal's algorithm focus on finding a Minimum-Weight Spanning Tree(MWST) for a graph.

Spanning Tree:

A spanning tree is a subset of the edges such that all nodes are connected through these edges, yet there are no cycles.

MWST:

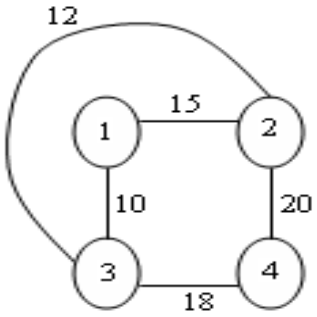
A Minimum-Weight Spanning Tree has the least possible total edge weight of all spanning trees. There is a well-known "Greedy Algorithm", called Kruskal's algorithm, for finding a MWST.

Procedure:

(1) Identify the minimal edge connected component.
(2) Find the minimal edge if both the vertices belong to different connected component then add the edge.
(3) Identify minimal edge, if that edge connects both the vertices in the same component then leave the edge. (Otherwise it will form a cycle).
(4) Repeat the process until all the vertices are found.

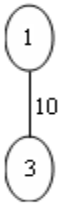
Ex:

Find the MWST using Kruskal's algorithm.

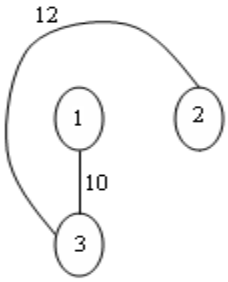


Soln:

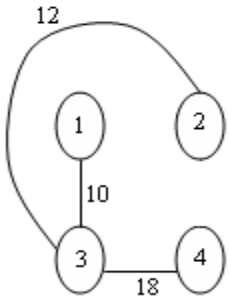
(1) First consider the edge (1, 3) because it has the lowest weight 10.



(2) The next minimal edge is (2, 3), with weight 12. Since 2 and 3 are in different components, we accept this edge and add the node 2 into first connected component. ie;



(3) The third edge is (1, 2) with weight 15. However, 1 and 2 are now in the same component, so we reject this edge and proceed to the fourth edge (3. 4). ie;



(4) Now, we have three edges for the spanning tree of a 4-node graph and so may stop.

When we translate the above ideas to TM's, we face several issues:

- When we deal with TM's, we may only think of problems as languages, and the only output is Yes or No(ie., Accept or Reject). For instance, the MWST problem could be couched as: "Given this graph G and limit weight 'W' or less?
- While we might think informally of the "size" of a graph as the number of its nodes or edges, the input to a TM is a string over a finite alphabet. Thus, problem elements such as nodes and edges must be encoded suitably.

Nondeterministic Polynomial Time:

A fundamental class of problems can be solved by a nondeterministic TM that runs in polynomial time. A language L is in the class NP (Nondeterministic Polynomial) if there is a nondeterministic 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.

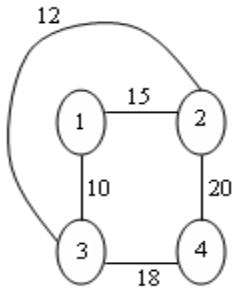
An NP Example: The Traveling Salesman Problem:

The input to Traveling Salesman Problem TSP) is the same as to MWST, a graph with integer weights on the edges and a weight limit W.

A **Hamilton Circuit** is a set of edges that connect the nodes into a single cycle, with each node appearing exactly once. Note that the number of edges on a Hamilton Circuit must equal the number of nodes in the graph.

Ex:

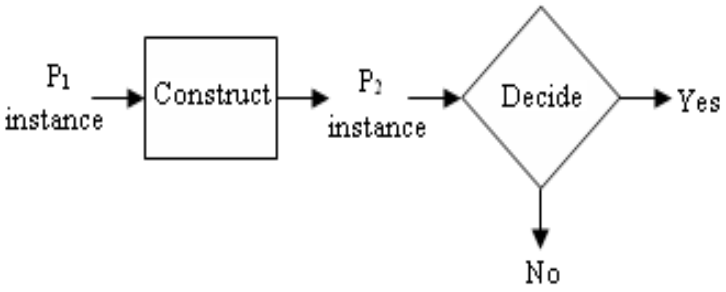
Find Travelling Salesman Problem for the graph,



Soln:

Hamilton Circuit: The cycle (1, 2, 4, 3, 1). The total weight of this cycle is 15+20+18+10 = 63. Thus if 'w' is 63 or more, the answer is "yes", and if w<63 the answer is "no".

Polynomial Time Reductions:



In this, to prove the statement "if P_2 is in P, then so is P_1".

- For the proof, suppose that we can decide membership in P_2 of a string of length 'n' in time $O(n^k)$. Then we can decide membership in P1 of a string of length 'm' in time $O(m^j + (cm^j)^k)$ time; the term m^j accounts for the time to do the translation, and the term $(cm^j)^k$ accounts for the time to decide the resulting instance of P_2.
- Simplifying the expression, we see that P_1 can be solved in time $O(m^j + (cm^j)^k)$. Since c, j and k are all constants, this time is polynomial in 'm', and we conclude P_1 is in P.
- A reduction from P_1 to P_2 is polynomial time if it takes time that is some polynomial in the length of the P_1 instance. Note that as a consequence, the P_2 instance will be of a length that is polynomial in the length of the P_1 instance.

NP – Complete Problems:

Let L be a language(problem) in NP, we say L is NP-Complete if the following statements are true about L;

(1) L is in NP.

(2) For every language L' in NP there is a polynomial – time reduction of L' to L.

An example of NP- Complete problem is the Travelling Salesman Problem. Since it appears that P≠NP, all the NP-Complete problems are in NP – P, we generally view a proof of NP – Completeness for a problem as a proof that the problem is not in P.

Theorem:

If P_1 is NP-Complete, and there is a Polynomial – time reduction of P_1 to P_2, then P_2 is NP – Complete.

Proof:

To show that every language L in NP Polynomial – time reduces to P_2. We know that there is a polynomial time reduction of L to P_1, this reduction takes some polynomial time P(n). Thus, a string 'w' in L of length 'n' is converted to a string 'x' in P_1 of length atmost P(n).

Also know that there is a polynomial time reduction of P_1 to P_2; Let this reduction take polynomial time q(m). Then this reduction transforms 'x' to some string 'y' in P_2, taking time atmost q(p(n)).

Thus, the transformation of 'w' to 'y' takes time atmost p(n)+q(p(n)), which is a polynomial. We conclude that L is polynomial time reducible to P_2. Since L could be any language in NP, we have shown that all of NP polynomial – time reduces to P_2; that is, P_2 is NP-Complete.