Unit 1

Mathematical Preliminaries: Set Functions and Relations etc.

Sets

A set is a collection of well defined objects. Usually the element of a set has common properties. e.g. all the student who enroll for a course “theory of computation” make up a set.

Examples

The set of even positive integer less than 20 can be expressed by

E = {2,4,6,8,10,12,14,16,18}

Or E = {x|x is even and 0<x<20}

Finite and Infinite Sets

A set is finite if it contains finite number of elements. And infinite otherwise.The empty set has no element and is denoted by ϕ.

Cardinality of set:

It is a number of element in a set. The cardinality of set E is

|E|=9.

Subset :

A set A is subset of a set B if each element of A is also element of B and is denoted by $A \subseteq B$.

Set operations

Union:

The union of two set has elements, the elements of one of the two sets and possibly both. Union is denoted by .

Intersection:

The intersection of two sets os the collection of all elements of the two sets which are common and **is** denoted by.

Differences:

The difference of two sets A and B, denoted by A-B, is the set of all elements that are in the set A but not in the set B.

Sequences and Tuples

A sequence of objects is a list of objects in some order. For example, the sequence 7,4,17 would be written as (7,4,17).In set the order does not matter but in sequence it does. Also, repetition is not permitted in a set but is allowed in a sequence. Like set , sequence may be finite or infinite.

Relations A nd Functions

A binary relation on two sets A and B is a subset of A×B. for example, if A={1,3,9}, B={x,y}, then {(1,x),(3,y),(9,x)} is a binary relation on 2- sets. Binary relations on K-sets A_1,A_2,……..A_k can be similarly defined.

A function is an object that setup an input- output relationship i.e. a function takes an input and produces the required output. For a function f , with input x, the output y, we write f(x)=y. We also say that f maps x to y.

A binary relation r is an equivalence relation if R satisfies :

R is reflexive.i.e. for every x,(x,x)ϵR.

R is symmetric i.e. for every x and y , (x,y)ϵR imlies (y,x) ϵR.

R is transitive i..e. for every x,y, and z, (x,y) ϵR and (y,z) ϵR imples (x,z) ϵR.

Closures

Closures is an important relationship among sets and is a general tool for dealing with sets and relationship of many kinds. Let R be a binary relation on a set A. Then the reflexive closure of R is arelation R’ such that :

1. R’ is reflexive (symmetric, transitive)
2. R’$\sqsupseteq$ R.
3. If R’’ is a reflexive relation containing R then R$\sqsupseteq$ R

Method of proofs:

Mathematical Induction

Let A be a set of natural numbers such that :

i. 0ϵA

ii. For each natural number *n*, if {0,1,2,3,…….n}ϵA. Then A=N. In particular, induction is used to prove assertions of the form " for all nϵN, the property is valid". i.e.

In the basis step, one has to show that P(0) us true. i.e. the property is true for 0.

P holds for n will be the assumption.

Then one has to prove the validity of P for n+1.

Strong mathematical Inductions

Another form of proof by induction over natural numbers is called strong induction. Suppose we want to prove that P(n) is true for all $n \geq t$. Then in the induction step, we assume that P(j) us true for all j, $t \leq j \leq k$. Then using this, we prove P(k). in ordinary induction in the induction step, we assume P(k-1) to prove P(k). There are some instances, where the result can be proved easily using strong induction. In some cases, it will not be possible to use weak induction and one use strong induction.

Computation:

If it involves a computer, a program running on a computer and numbers going in and out then computation is likely happening.

Theory of computation:

- It is a Study of power and limits of computing. It has three interacting components:
    - Automata Theory
    - Computability Theory
    - Complexity Theory

Computability Theory: -
- What can be computed?
- Are there problems that no program can solve?

Complexity Theory: -

- What can be computed efficiently?
- Are there problems that no program can solve in a limited amount of time or space?

Automata Theory: -

- Study of abstract machine and their properties, providing a mathematical notion of "computer"
- Automata are abstract mathematical models of machines that perform computations on an input by moving through a series of states or configurations. If the computation of an automaton reaches an accepting configuration it accepts that input.

Study of Automata

- For software designing and checking behavior of digital circuits.
- For designing software for checking large body of text as a collection of web pages, to find occurrence of words, phrases, patters (i.e. pattern recognition, string matching, …)
- Designing “lexical analyzer” of a compiler, that breaks input text into logical units called “tokens

Abstract Model

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

Brief History:

Before 1930’s, no any computer were there and Alen Turing introduced an abstract machine that had all the capabilities of today’s computers. This conclusion applies to today’s real machines.

Later in 1940’s and 1950’s, simple kinds of machines called finite automata were introduced by a number of researchers.

In late 1950’s the linguist N. Chomsky begun the study of formal grammar which are closely related to abstract automata.

In 1969 S. Cook extended Turing’s study of what could and what couldn’t be computed and classified the problem as:

-Decidable
-Tractable/intractable

The basic concepts of Languages

The basic terms that pervade the theory of automata include “alphabets”, “strings”, “languages”, etc.

Alphabets: - (Represented by ‘Σ’)

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

{0,1} – Binary alphabets
{+, −, *} – Special symbols

Strings: - (Strings are denoted by lower case letters)

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

Empty String: -
It is a string with zero occurrences of symbols. It is denoted by 'ε' (epsilon).

Length of String

The length of a string w, denoted by | w |, is the number of positions for symbols in w. we have for every string s, length (s) ≥ 0.
$| \varepsilon | = 0$ as empty string have no symbols.
$| 0110 | = 4$

Power of alphabet

The set of all strings of certain length k from an alphabet is the kth power of that alphabet.
i.e. $\Sigma_k = \{w / |w| = k\}$
If $\Sigma = \{0, 1\}$ then,

$\Sigma^0 = \{\varepsilon\}$
$\Sigma^1 = \{0, 1\}$
$\Sigma^2 = \{00, 01, 10, 11\}$
$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

Kleen Closure

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

$\therefore \Sigma_* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup$ ……………
E.g. A = {0}
$A_* = \{0^n / n = 0, 1, 2, \ldots\}$.

Positive Closure: -
The set of all the strings over an alphabet Σ, except the empty string is called positive closure and is denoted by Σ_+.

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

Language:

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

- Set of all strings over Σ = {0, 1} with equal number of 0's & 1's.
  L = {ε, 01, 0011, 000111, ………}
- φ is an empty language & is a language over any alphabet.

- {ε} is a language consisting of only empty string.
- Set of binary numbers whose value is a prime:
  L = {10, 11, 101, 111, 1011, ……}

Concatenation of Strings

Let *x* & *y* be strings then *xy* denotes concatenation of *x* & *y*, i.e. the string formed by making a copy of *x* & following it by a copy of *y*.

More precisely, if *x* is the string of *i* symbols as $x = a_1a_2a_3\ldots a_i$ & *y* is the string of *j* symbols as $y = b_1b_2b_3\ldots b_j$ then *xy* is the string of *i* + *j* symbols as $xy = a_1a_2a_3\ldots a_ib_1b_2b_3\ldots b_j$.

For example;

x = 000
y = 111
xy = 000111 &
yx = 111000

Note: 'ε' is identity for concatenation; i.e. for any *w*, ε*w* = *w*ε = *w*.

Suffix of a string

A string *s* is called a suffix of a string *w* if it is obtained by removing 0 or more leading symbols in *w*. For example;

w = abcd
s = bcd is suffix of *w*.

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

Prefix of a string

A string *s* is called a prefix of a string *w* if it is obtained by removing 0 or more trailing symbols of *w*. For example;

w = abcd
s = abc is prefix of *w*,

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

Substring

A string *s* is called substring of a string *w* if it is obtained by removing 0 or more leading or trailing symbols in *w*. It is proper substring of *w* if $s \neq w$.
If *s* is a string then *Substr (s, i, j)* is substring of *s* beginning at *i*th position & ending at *j*th position both inclusive.

Problem

A problem is the question of deciding whether a given string is a member of some particular language.
In other words, if Σ is an alphabet & L is a language over Σ, then problem is;
- Given a string w in Σ*, decide whether or not *w* is in L.

Exercises:

1. Let A be a set with n distinct elements. How many different binary relations on A are there?

2. If ∑= {a,b,c} then find the followings

   a. $\sum^1, \sum^2, \sum^3$.

3. If ∑={0,1}. Then find the following languages
   a. The language of string of length zero.
   b. The language of strings of 0's and 1's with equal number of each.
   c. The language $\{0^n1^n \mid n \geq 1\}$
   d. The language $\{0^i0^j \mid 0 \leq i \leq j\}$.
   e. The language of strings with odd number of 0's and even number of 1's.

4. Define the Kleen closure and power of alphabets.

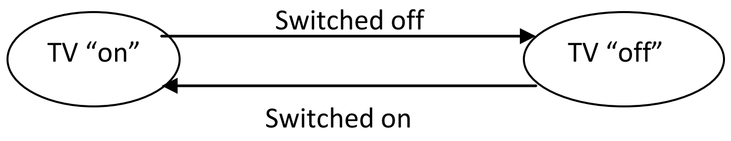
Readings: Plz, read the *chapter 1 sections* 1.1, 1.1.3, 1.5 of Text book. And solve some numerical example given in Text book.

Chapter 2

Finite Automata (DFA and NFA, epsilon NFA)

Intuitive example

Consider a man watching a TV in his room. The TV is in "on" state. When it is switched off, the TV goes to "off" state. When it is switched on, it again goes to "on" state. This can be represented by following picture.



The above figure is called state diagram.

A language is a subset of the set of strings over an alphabet. A language can be generated by grammar. A language can also be recognized by a machine. Such machine is called recognition device. The simplest machine is the finite state automaton.

Finite Automata

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

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

Applications:

The finite state machines are used in applications in computer science and data networking. For example, finite-state machines are basis for programs for spell checking, indexing, grammar checking, searching large bodies of text, recognizing speech, transforming text using markup languages such as XML & HTML, and network protocols that specify how computers communicate.

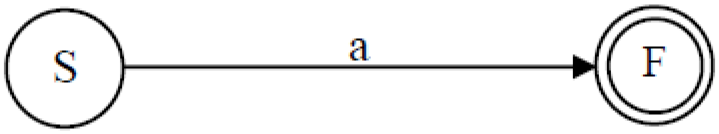
Definition (Deterministic finite state automata [DFSA])

A deterministic finite automaton is defined by a quintuple (5-tuple) as $(Q, \Sigma, \delta, q_0, F)$.

Where,
Q = Finite set of states,
Σ = Finite set of input symbols,
δ = A transition function that maps Q × Σ -> Q
q_0 = A start state; $q_0 \in Q$
F = Set of final states; $F \subseteq Q$.

A transistion function δ that takes as arguments a state and an input symbol and returns a state. In our diagram, δ is represented by arcs between states and the labels on the arcs.

For example



If s is a state and a is an input symbol then δ(p,a) is that state q such that there are arcs labled 'a' from p to q.

General Notations of DFA

There are two preferred notations for describing this class of automata;
- Transition Table
- Transition Diagram

1) Transition Table: -

Transition table is a conventional, tabular representation of the transition function δ that takes the arguments from Q × Σ & returns a value which is one of the states of the automation. The row of the table corresponds to the states while column corresponds to the input symbol. The starting state in the table is represented by -> followed by the state i.e. ->q, for q being start state, whereas final state as *q, for q being final state.
The entry for a row corresponding to state q and the column corresponding to input *a*, is the state δ (q, a).

For example:

I. Consider a DFA;
$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{0, 1\}$
$q_0 = q_0$
$F = \{q_0\}$
$\delta = Q \times \Sigma \rightarrow Q$

Then the transition table for above DFA is as follows:

δ	0	1
* -> q_0	q2	q1
q1	q3	q0
q2	q0	q3
q3	q1	q2

This DFA accepts strings having both an even number of 0's & even number of 1's.

Transition Diagram:

A transition diagram of a DFA is a graphical representation where; (or is a graph)

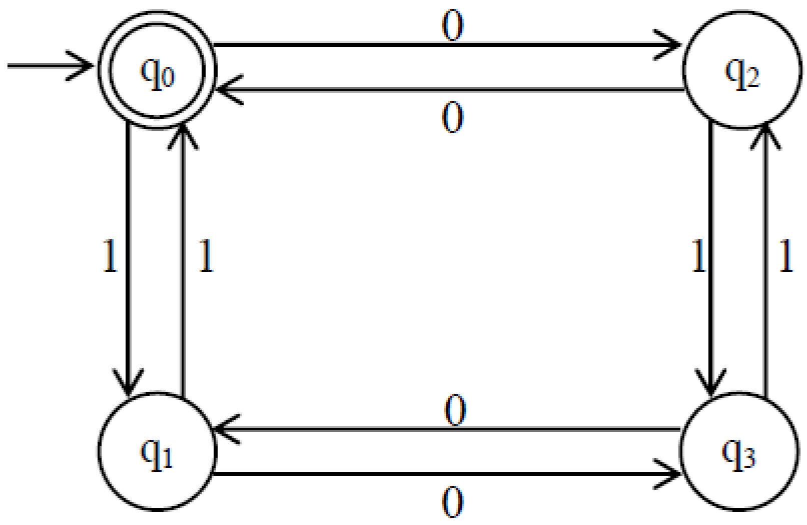
- For each state in Q, there is a node represented by circle,

- For each state q in Q and each input a in Σ, if δ (q, a) = p then there is an arc from node q to p labeled a in the transition diagram. If more than one input symbol cause the transition from state q to p then arc from q to p is labeled by a list of those symbols.

- The start state is labeled by an arrow written with "start" on the node.

- The final or accepting state is marked by double circle.

- For the example I considered previously, the corresponding transition diagram is:



How a DFA process strings?

The first thing we need to understand about a DFA is how DFA decides whether or not to "accept" a sequence of input symbols. The "language" of the DFA is the set of all symbols that the DFA accepts. Suppose $a_1, a_2, \ldots\ldots a_n$ is a sequence of input symbols. We start out with the DFA in its start state, q_0. We consult the transition function δ also for this purpose. Say $\delta(q_0, a_1) = q_1$ to find the state that the DFA enters after processing the first input symbol a_1. We then process the next input symbol a_2, by evaluating $\delta(q_1, a_2)$; suppose this state be q_2. We continue in this manner, finding states $q_3, q_4, \ldots, q_n$. such that $\delta(q_{i-1}, a_i) = q_i$ for each i. if q_n is a member of F, then input $a_1, a_2, \text{---}\ a_n$ is accepted & if not then it is rejected.

Extended Transition Function of DFA($\hat{\delta}$): -

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

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

Let us define by induction on length of input string as follows:

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

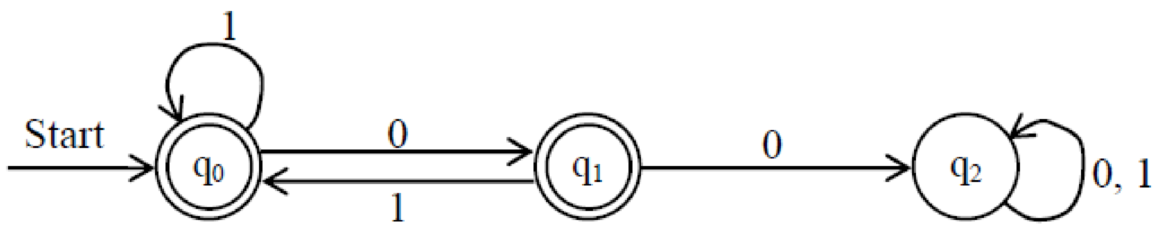
Induction: Let w be a string from Σ^* such that $w = xa$, where x is substring of w without last symbol and a is the last symbol of w, then $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$.

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

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

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

For Example



Now compute $\hat{\delta}$ (q_0,1001)

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

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

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

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

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

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

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

$$= \delta(q_2, 1)$$

$$= q_2, \textit{ so accepted.}$$

2) **Compute** $\hat{\delta}$ (q_0,101) yourself.(ans : Not accepted by above DFA)

String accepted by a DFA

A string x is accepted by a DFA (Q, Σ, δ, q_0, F) if; $\hat{\delta}$ (q, x) = p ∈ F.

Language of DFA

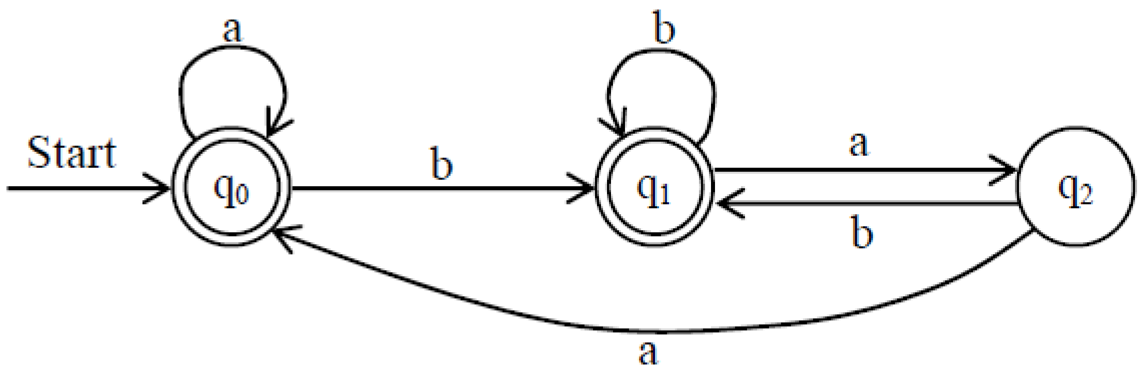
The language of DFA M = (Q, Σ, δ, q_0, F) denoted by L(M) is a set of strings over Σ* that are accepted by M.

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

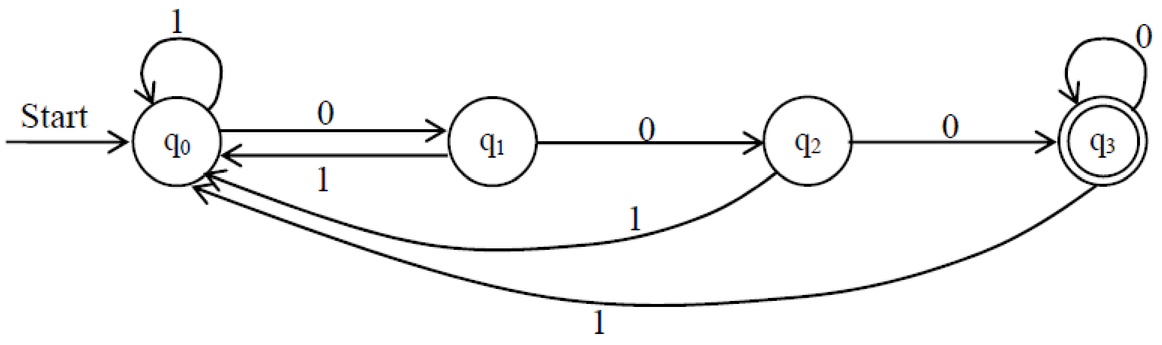
That is; the language of a DFA is the set of all strings w that take DFA starting from start state to one of the accepting states. The language of DFA is called regular language.

Examples (DFA Design for recognition of a given language)

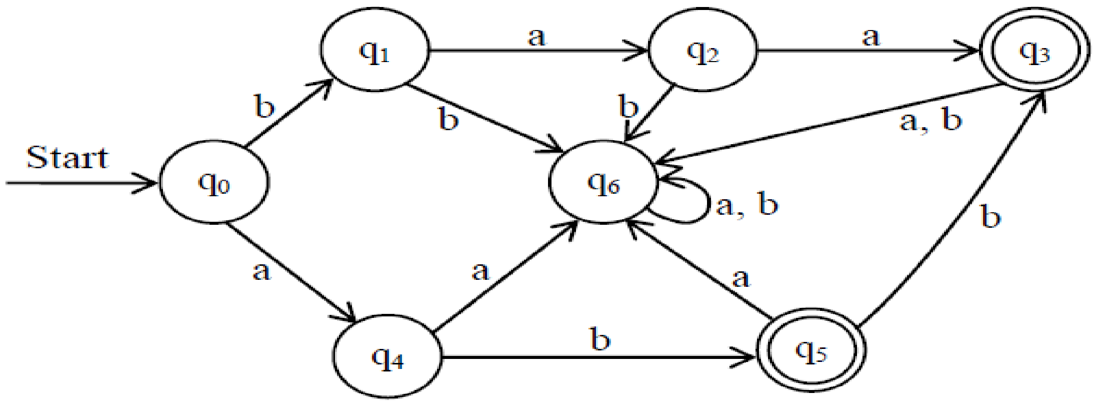
1. Construct a DFA, that accepts all the strings over $\Sigma = \{a, b\}$ that do not end with ba.



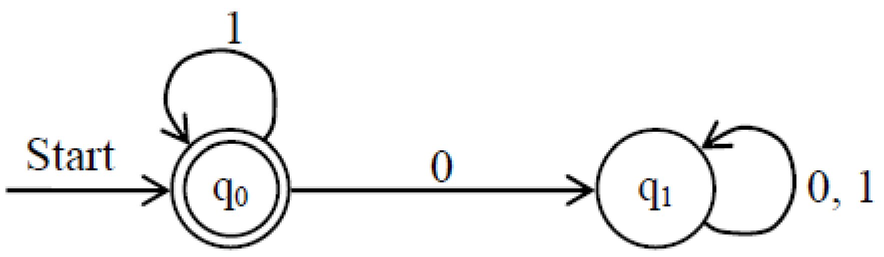
2. DFA accepting all string over $\Sigma = \{0, 1\}$ ending with 3 consecutive 0's.



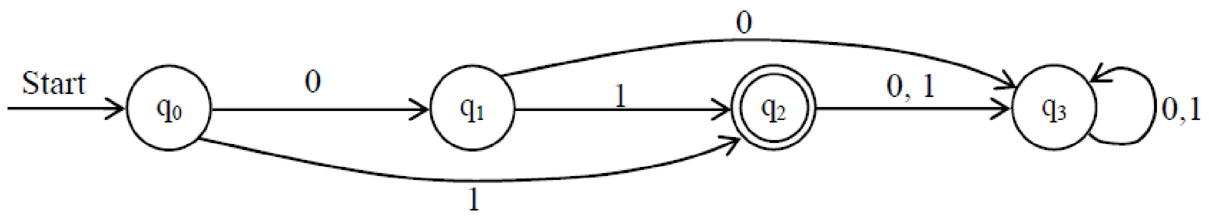
3. DFA over {a, b} accepting {baa, ab, abb}



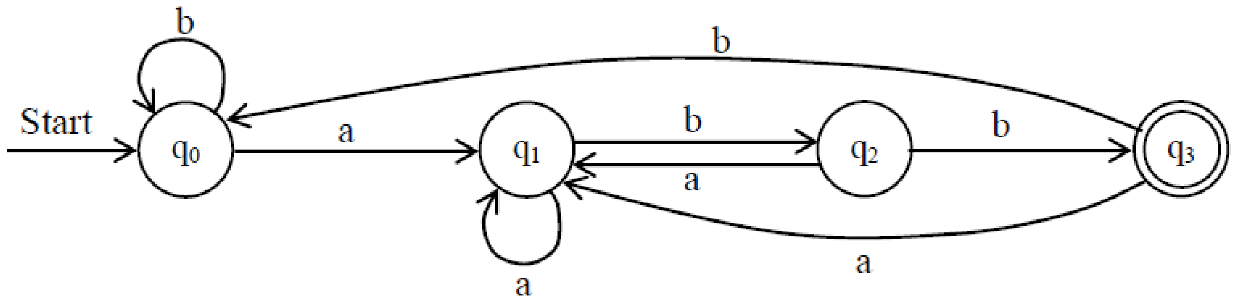
4. DFA accepting zero or more consecutive 1's.
i.e. L (M) = {1^n / n = 0, 1, 2, ……}



5. DFA over {0, 1} accepting {1, 01}



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



Exercises: (Please do this exercise as homework problems and the question in final exam will be of this patterns)

1. **Give the DFA for the language of string over {0.1} in which each string end with 11. [2067,TU BSc CSIT]**

2. **Give the DFA accepting the string over {a,b} such that each string does not end with ab.[2067, TU B.Sc CSIT]**

3. Give the DFA for the language of string over {a,b} such that each string contain aba as substring.
4. Give the DFA for the langague of string over {0,1} such that each string start with 01.
5. The question from book: 2.2.4, 2.2.5 of chapter 2.

Non-Deterministic Finite Automata (NFA)

A non-deterministic finite automaton is a mathematical model that consists of:

- A set of states Q, (finite)
- A finite set of input symbols Σ, (alphabets)
- A transition function that maps state symbol pair to sets of states.
- A state $q_0 \in Q$, that is distinguished as a start (initial) state.
- A set of final states F distinguished as accepting (final) state. $F \subseteq Q$.

Thus, NFA can also be interpreted by a quintuple; $(Q, \Sigma, \delta, q_0, F)$ where δ is $Q \times \Sigma = 2^Q$. Unlike DFA, a transition function in NFA takes the NFA from one state to several states just with a single input.

For example;

1.

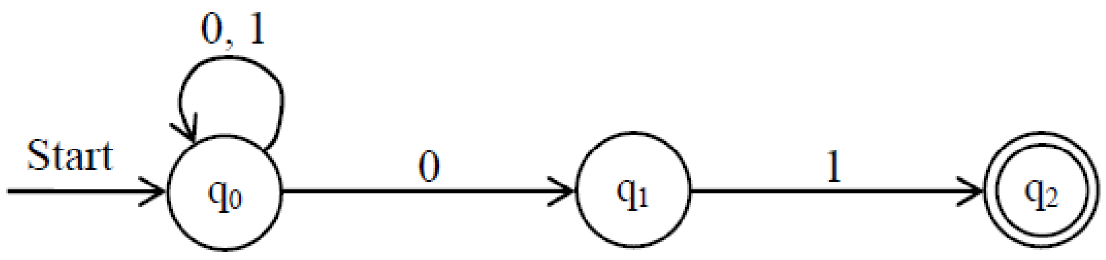
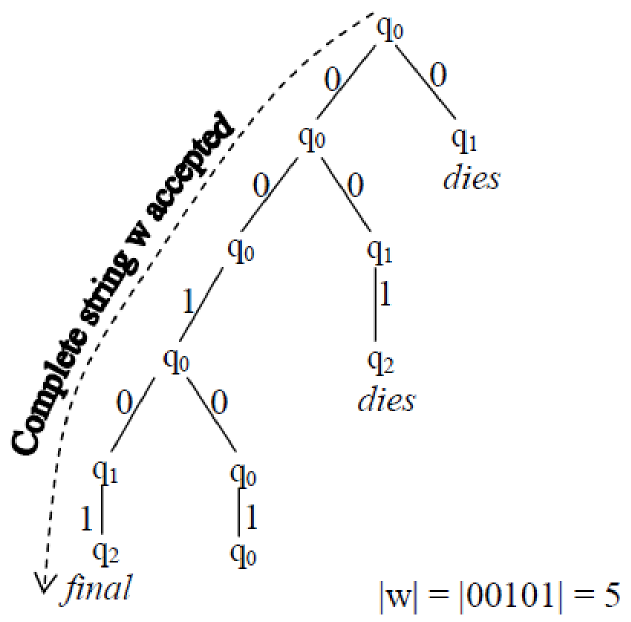


Fig: - NFA accepting all strings that end in 01.

Here, from state q_1, there is no any arc for input symbol 0 & no any arc out of q_2 for 0 & 1. So, we can conclude in a NFA, there may be zero no. of arcs out of each state for each input symbol. While in DFA, it has exactly one arc out of each state for each input symbol.

δ, the transition function is a function that takes a state in Q and an input symbol in ∑ as arguments and returns a subset of Q The only difference between an NFA and DFA is in type of value that δ returns. In NFA, δ returns a set of states and in case of DFA it returns a single state.

For input sequence w = 00101, the NFA can be in the states during the processing of the input are as:



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

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

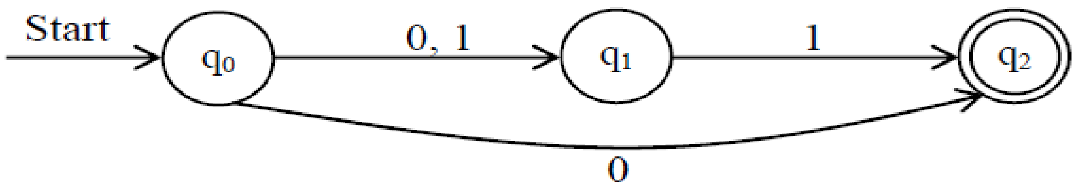
$q_0 = \{q_0\}$

$F = \{q_2\}$

Transition table:

δ:	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{\phi\}$	$\{q_2\}$
$*q_2$	$\{\phi\}$	$\{\phi\}$

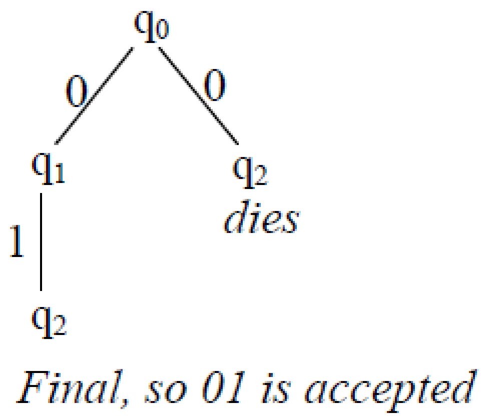
2. NFA over {0, 1} accepting strings {0, 01, 11}.



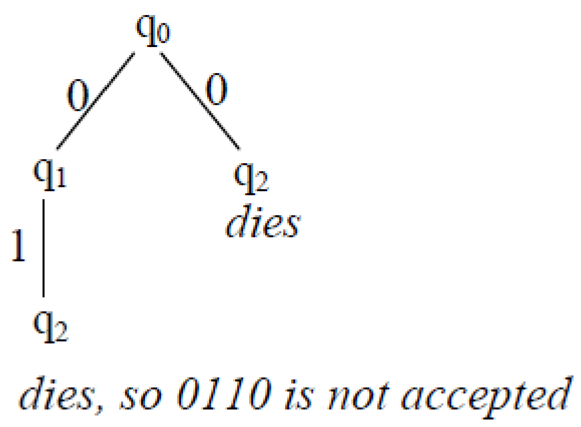
Transition table:

δ:	0	1
$\rightarrow q_0$	$\{q_0, q_2\}$	$\{q_1\}$
q_1	$\{\phi\}$	$\{q_2\}$
$*q_2$	$\{\phi\}$	$\{\phi\}$

Computation tree for 01;



Computation tree for 0110



The Extended transition function of NFA

As for DFA's, we need to define the extended transition function $\hat{\delta}$ that takes a state q and a string of input symbol w and returns the set of states that is in if it starts in state q and processes the string w.

Definition by Induction:

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

Induction: Let w be a string from Σ* such that w = xa, where x is a substring of without last symbol a.

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

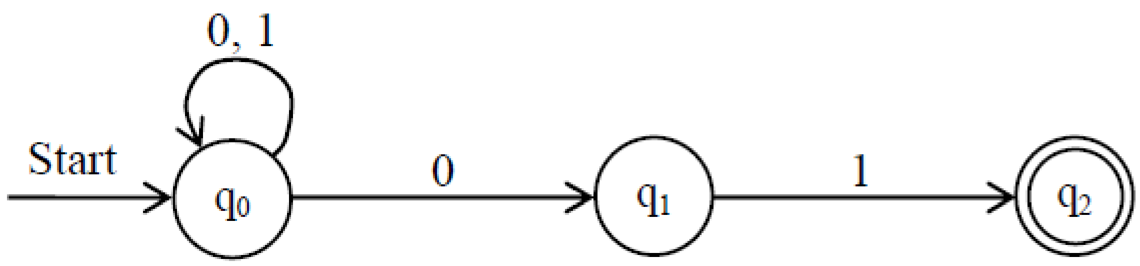
and

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

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

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

Consider, a NFA,



Now, computing for $\hat{\delta}$ (q_0, 01101)

Solution:

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

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

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

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

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

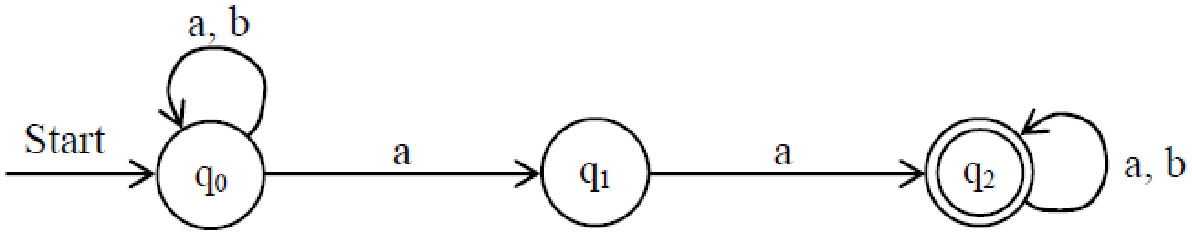
$\hat{\delta}(q_0, 0110) = \delta(q_0, 0) = \{q_0\} \cup \{q_1\} = \{q_0, q_1\}$

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

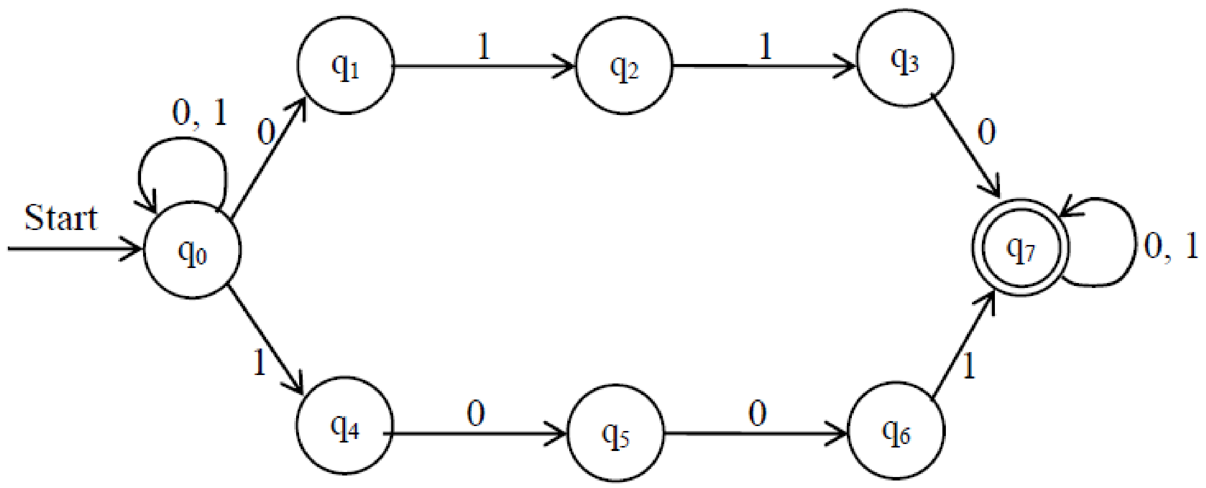
Since the result of above computation returns the set of state {q_0,q_2) which include the accepting state q2 of NFA so the string 01101 is accepted by above NFA.

Examples (Design NFA to recognize the given language)

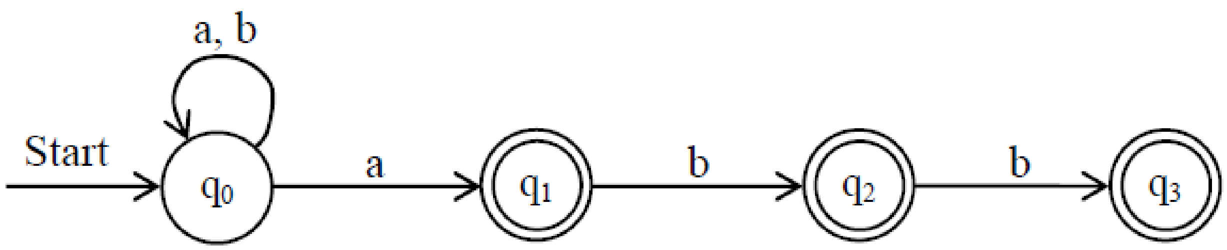
1. **Construct a NFA over {a, b} that accepts strings having *aa* as substring.**



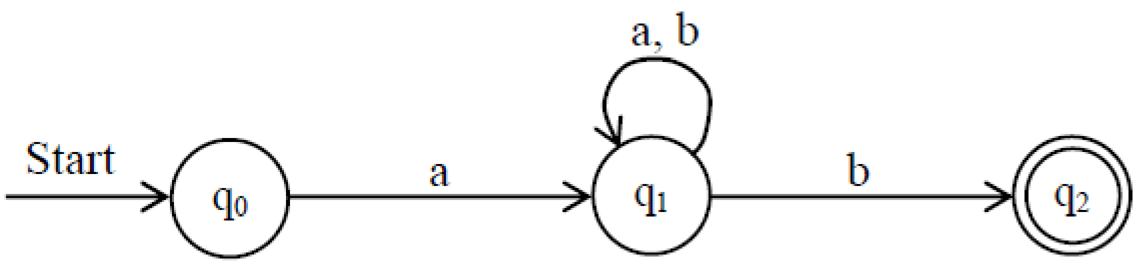
2. **NFA for strings over {0, 1} that contain substring 0110 or 1001**



3. NFA over {a, b} that have "a" as one of the last 3 characters.



4. NFA over {a, b} that accepts strings starting with *a* and ending with *b*.



Language of NFA

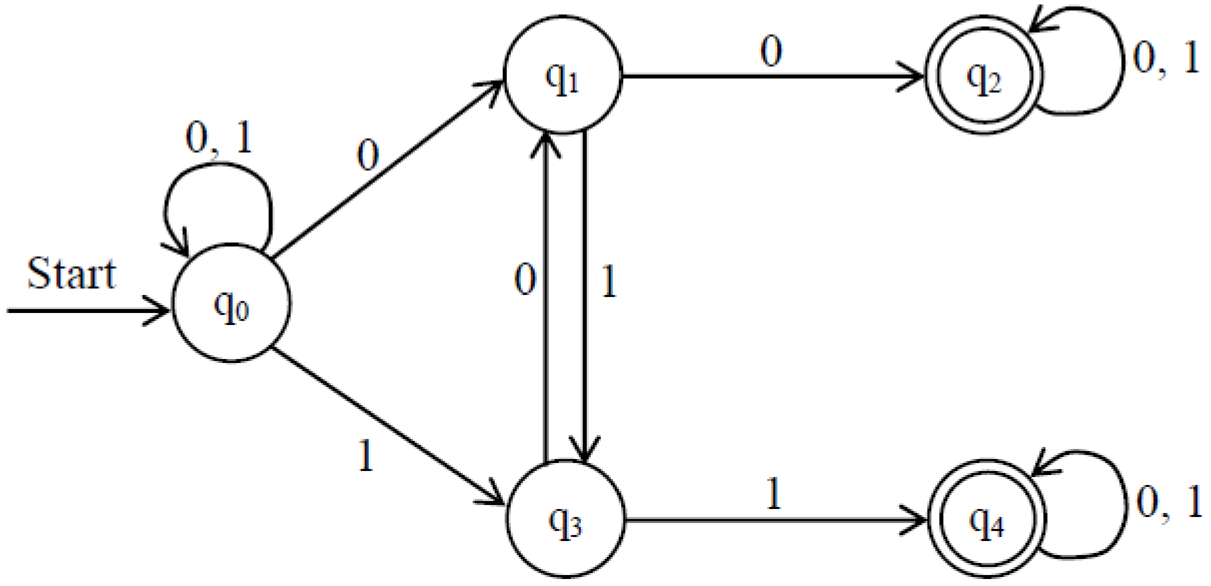
The language of NFA, M = (Q, Σ, δ, q0, F), denoted by L (M) is;

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

i.e. L(M) is the set of strings w in $\sum^*$ such that $\hat{\delta}(q_0,w)$ contains at least one state accepting state.

Examples

1. **Design a NFA for the language over {0, 1} that have at least two consecutive 0's or1's**



Now, compute for acceptance of string 10110;

Solution

$\hat{\delta}$ (q_0, 10110)

Start with starting state as
$\hat{\delta}$ (q_0, ε) = {q_0}
$\hat{\delta}$ (q_0, 1) = {q_0, q_3}
(q0, 10) = δ (q_0, 0) ∪ δ (q_3, 0) = {q_1,q_0} ∪ {q1} = {q1, q_0}
(q0, 101) = δ (q_1, 1) ∪ δ (q_0, 1) = {q3} ∪ {q_3} = {q3}
(q0, 1011) = δ (q3, 1) = {q4}
(q0, 10110) = δ (q4, 0) = {q4} = {q4}

So accepted (since the result in final state)

Exercise

1. **Question from book: 2.3.4 of chapter 2**
2. **Give a NFA to accept the language of string over{a.b} in which each string contain abb as substring.**
3. **Give a NFA which accepts binary strings which have at least one pair of '00' or one pair of '11'.**

Equivalence of NFA & DFA

Although there are many languages for which NFA is easier to construct than DFA, it can be proved that every language that can be described by some NFA can also be described by some DFA.

The DFA has more transition than NFA and in worst case the smallest DFA can have 2^n state while the smallest NFA for the same language has only n states.

We now show that DFAs & NFAs accept exactly the same set of languages. That is non-determinism does not make a finite automaton more powerful.

To show that NFAs and DFAs accept the same class of language, we show;

Any language accepted by a NFA can also be accepted by some DFA. For this we describe an algorithm that takes any NFA and converts it into a DFA that accepts the same language. The algorithm is called “subset construction algorithm”.

The key idea behind the algorithm is that; the equivalent DFA simulates the NFA by keeping track of the possible states it could be in. Each state of DFA corresponds to a subset of the set of states of the NFA, hence the name of the algorithm. If NFA has n-states, the DFA can have 2^n states (at most), although it usually has many less.

The steps are:

To convert a NFA, $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ into an equivalent DFA $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$, we have following steps.

1.The start state of D is the set of start states of N i.e. if q0 is start state of N then D has start state as {q0}.

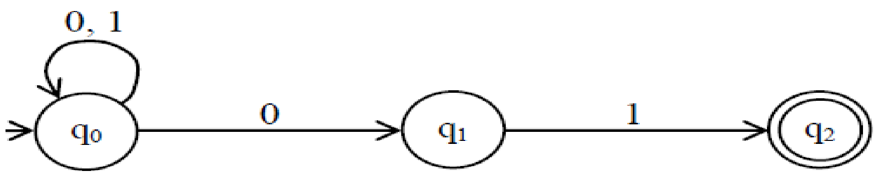
2.Q_D is set of subsets of Q_N i.e. $Q_D = 2^{QN}$. So, Q_D is power set of Q_N. So if Q_N has n states then Q_D will have 2^n states. However, all of these states may not be accessible from start state of Q_D so they can be eliminated. So Q_D will have less than 2^n states.

3.F_D is set of subsets S of Q_N such that $S \cap F_N \neq \phi$ i.e. F_D is all sets of N’sstates that include at least one final state of N.

For each set $S \subseteq Q_N$ & each input $a \in \Sigma$, $\delta_D (S, a) = \bigcup_{p\ in\ s} \delta N(p, a)$

i.e. for any state {q0, q1, q2, … qk} of the DFA & any input a, the next state of the DFA is the set of all states of the NFA that can result as next states if the NFA is in any of the state’s q0, q1, q2, … qk when it reads a.

For Example

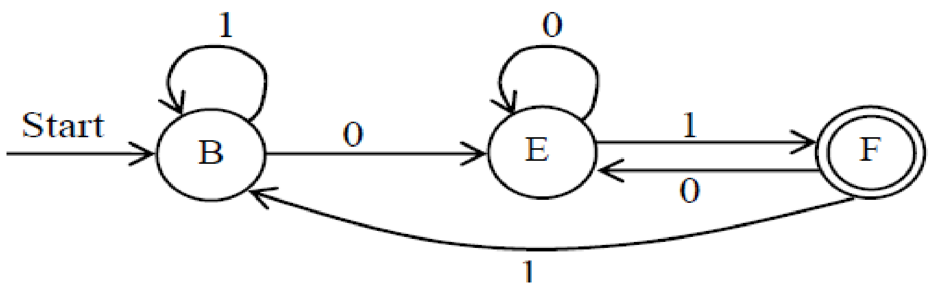


	δ:	0	1
A	ϕ	ϕ	ϕ
B	$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
C	$\{q_1\}$	ϕ	$\{q_2\}$
D	$*\{q_2\}$	ϕ	ϕ
E	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
F	$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
G	$*\{q_1, q_2\}$	ϕ	$\{q_2\}$
H	$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

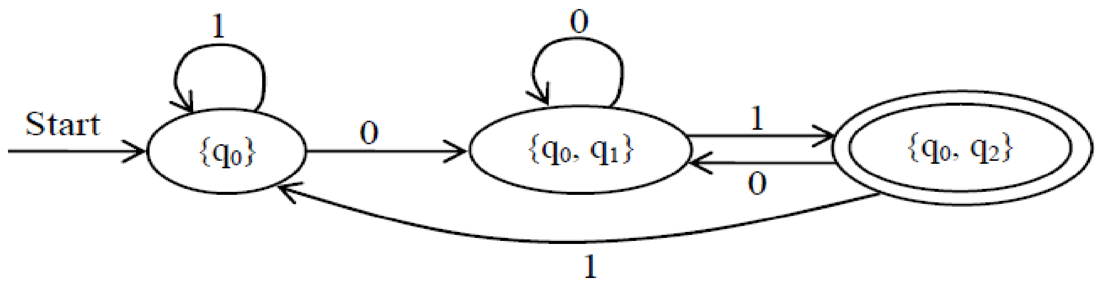
The same table can be represented with renaming the state on table entry as

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

The equivalent DFA is



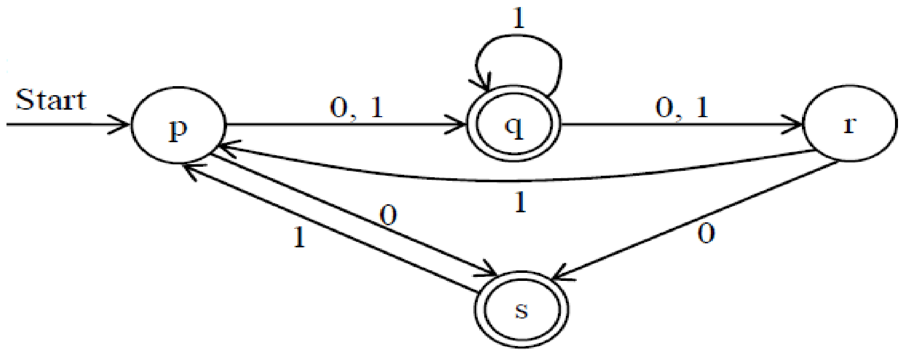
Or



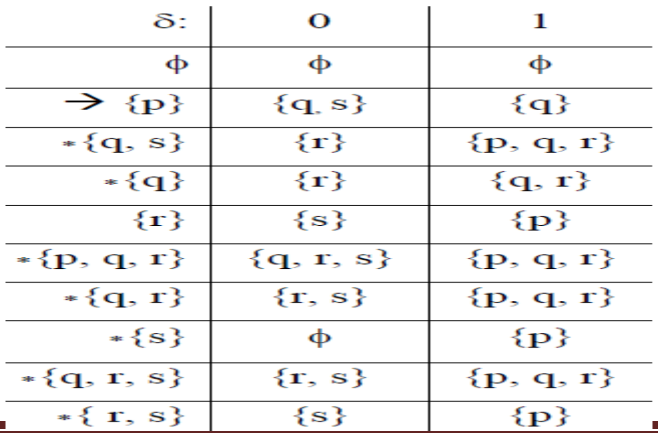
The other state are removed because they are not reachable from start state.

Example2

Convert the NFA to DFA



Solution: using the subset construction we have the DFA as

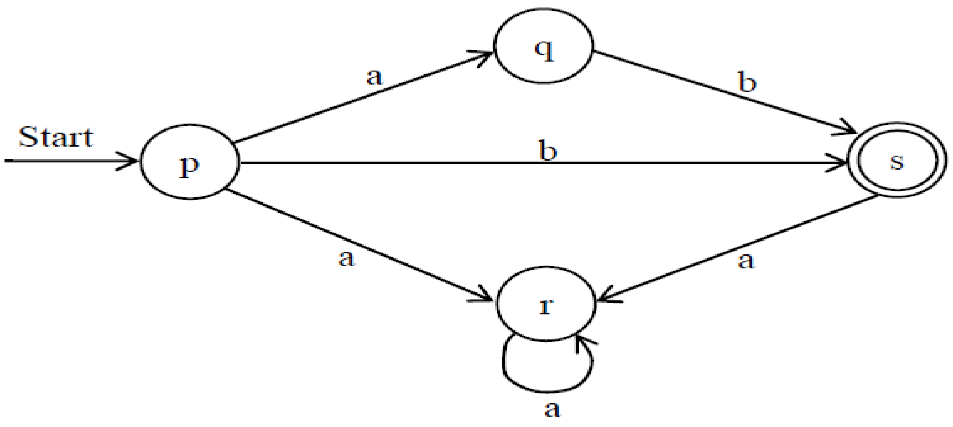


δ:	0	1
φ	φ	φ
→ {p}	{q, s}	{q}
*{q, s}	{r}	{p, q, r}
*{q}	{r}	{q, r}
{r}	{s}	{p}
*{p, q, r}	{q, r, s}	{p, q, r}
*{q, r}	{r, s}	{p, q, r}
*{s}	φ	{p}
*{q, r, s}	{r, s}	{p, q, r}
*{ r, s}	{s}	{p}

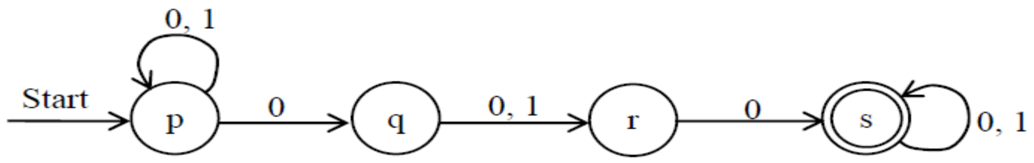
Draw the DFA diagram from above transition table yourself.

Exercises

1. Convert the following NFA to DFA



2.



3.Question from text book: 2.3.1, 2.3.2

Theorem 1:

For any NFA, N = (Q_N, Σ, δ_N, q0, FN) accepting language L ⊆ Σ there is a DFA D = (Q_D, Σ, δ_D, q0',F_D) that also accepts L i.e. L (N) = L (D).*

Proof: -

The DFA D, say can be defined as;
$Q_D = 2^{QN}$, q0 = {q0}

Let S = { p1, p2, p3, … pk} ∈ Q_D. Then for S ∈ Q_D & a ∈ Σ,

$$\delta_D(s, a) = \bigcup_{p_i \in S} \delta_N(p_i, a)$$

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

The fact that D accepts the same language as N is as;
for any string $w \in \Sigma^*$;
$\hat{\delta}_N(q0, w) = \hat{\delta}_D(q0, w)$

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

Basis Step:

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

$\hat{\delta}_N(q0, \varepsilon) = \{q0) = q0 = \hat{\delta}_D(q0, \varepsilon)$

Induction step;

Let $|w| = n + 1$ is a string such that $w = xa$ & $|x| = n$, $|a| = 1$; a being last symbol.
Let the inductive hypothesis is that x satisfies.
Thus,
$\hat{\delta}_D(q0', x) = \hat{\delta}_N(q0, x)$, let these states be $\{p1, p2, p3, \dots pk\}$

Now,
$\hat{\delta}_N(q0, w) = \hat{\delta}_N(q0, xa)$
$= \delta_N(\hat{\delta}_N(q0, x), a)$
$= \delta_N(\{p1, p2, p3, \dots pk\}, a)$ [Since, from inductive step]
$= \cup\, \delta_N(p_i, a)$……………………(1)

Also
$\hat{\delta}_D(q0', w) = \hat{\delta}_D(q0', xa)$
$= \delta_D(\hat{\delta}_D(q0', x), a)$
$= \delta_D(\hat{\delta}_N(q0, x), a)$ [Since, by the inductive step as it is true for x]
$= \delta_D(\{p1, p2, p3, \dots pk\}, a)$ [Since, from inductive step]

Now, from subset construction, we can write,
$\delta_D(\{p1, p2, p3, \dots pk\}, a) = \cup\delta_N(p_i, a)$

so, we have
$\hat{\delta}_D(q0', w) = \cup\delta_N(p_i, a)$………………………..(2)

Now we conclude from 1 and 2 that
$\hat{\delta}_N$ (q0, w) = $\hat{\delta}_D$ (q0', w).
Hence, if this relation is true for |x| = n, then it is also true for |w| = n + 1.

∴DFA D & NFA N accepts the same language.
i.e. L (D) = L (N) **Proved.**

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

Proof:

'if' part (A language is accepted by some DFA if L is accepted by some NFA):

It is the subset construction and is proved in previous theorem. In exam, you should write the proof of previous theorem here.

Only if part (a language is accepted by some NFA if L is accepted by some DFA):
Here we have to convert the DFA into an identical NFA.

Consider we have a DFA D = (Q_D, Σ, δ_D, q0, F_D).

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

Let NFA N = (Q_N, Σ, δ_N, q0', F_N) to be equivalent to D.
Where $Q_N = Q_D$, $F_N = F_D$, q0' = q0 and δ_N is defined by the rule
If δ_D(p,a)=q then δ_N(p,a)= {q}.
.
Then to show if L is accepted by D then it is also accepted by N, it is sufficient to show, for any string w ∈ Σ*, $\hat{\delta}_D$ (q0, w) = $\hat{\delta}_N$ (q0, w)

We can proof this fact using induction on length of the string.
Basis step: -

Let |w| = 0 i.e. w = ε
∴$\hat{\delta}_D$ (q0, w) =$\hat{\delta}_D$ (q0, ε) = q0
$\hat{\delta}_N$ (q0, w) = $\hat{\delta}_N$ (q0, ε) = {q0}
∴$\hat{\delta}_D$ (q0, w) = $\hat{\delta}_N$ (q0, w) for |w| = 0 is true.
Induction: -

Let $|w| = n + 1$ & $w = xa$. Where $|x| = n$ & $|a| = 1$; a being the last symbol.
Let the inductive hypothesis is that it is true for x.
$\therefore$ if $\hat{\delta}_D$ (q0, x) = p, then $\hat{\delta}_N$ (q0, x) = {p}

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

Now,
$\hat{\delta}_D$ (q0, w) = $\hat{\delta}_D$ (q0, xa)
= δ_D ($\hat{\delta}_D$ (q0, x), a)
= δ_D (p, a) [from inductive step $\hat{\delta}_D$ (q0, x,=p]
= r, say

Now,
$\hat{\delta}_N$ (q0, w) = $\hat{\delta}_N$ (q0, xa)
= δ_N ($\hat{\delta}_N$ (q0, x), a) [from inductive steps]
= δ_N({p}, a)
=r [from the rule that define δ_N)

Hence proved. i.e. $\hat{\delta}_D$ (q0, w)= $\hat{\delta}_N$ (q0, w)

NFA with ε-transition (ε-NFA)

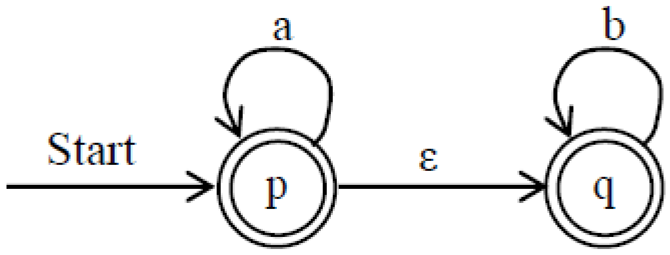
This is another extension of finite automation. The new feature that it incorporates is, it allows a transition on ε, the empty string, so that a NFA could make a transition spontaneously without receiving an input symbol.

Like Non-determinism added to DFA, this feature does not expand the class of language that can be accepted by finite automata, but it does give some added programming connivance. This are very helpful when we study regular expression (RE) and prove the equivalence between class of language accepted by RE and finite automata.

A NFA with ε-transition is defined by five tuples (Q, Σ, δ, q0, F), where;

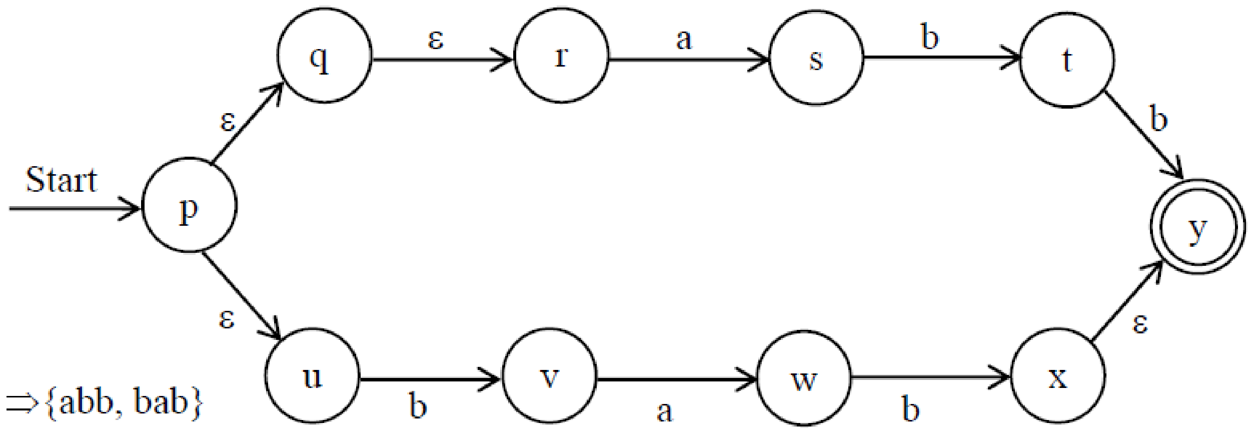
Q = set of finite states
Σ = set of finite input symbols
q0 = Initial state, $q0 \in Q$
F = set of final states; $F \subseteq Q$
δ = a transition function that maps;
$Q \times \Sigma \cup \{\varepsilon\} \dashrightarrow 2^Q$

For examples:
1.



This accepted the language {a,aa,ab,abb,b,bbb,…………………}

2.



ε-closure of a state:

ε-closure of a state 'q' can be obtained by following all transitions out of q that are labeled ε. After we get to another state by following ε, we follow the ε-transitions out of those states & so on, eventually finding every state that can be reached from q along any path whose arcs are all labeled ε.

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

Basis: state q is in ε-closure (q).

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

Extended Transition Function of ε-NFA: -

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

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

ii) INDUCTION STEP: -

Let w = xa be a string, where x is substring of w without last symbol a and a ∈ Σ but a ≠ ε.

Let $\hat{\delta}$ (q, x) = {p1, p2, … pk} i.e. pi's are the states that can be reached from q following path labeled x which can end with many ε & can have many ε.

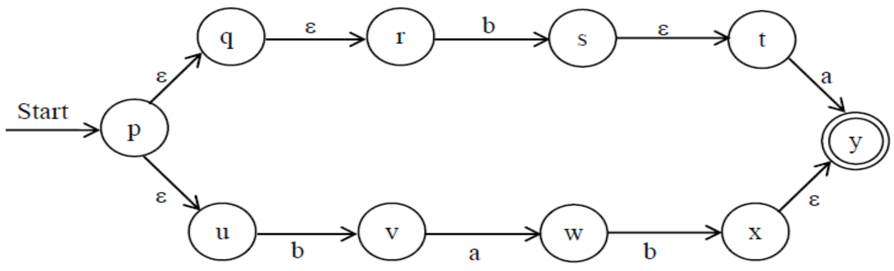
Also let,

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

then

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

Example



Now compute for string ba

-$\hat{\delta}$ (p, ε)= ε-closure(p)={p,q,r,u}

Compute for b i.e.
- δ(p,b)U δ(q,b)U δ(r,b)U δ(u,b)={s,v}
- ε-colsure(s)U ε-closure(v)={s,t,v}

Computer for next input 'a'
- δ(s,a)U δ(t,a)U δ(v,a)={y,w}
- ε-closure(y)U ε-closure(w)={y,w}

The final result set contains the one of the final state so the string is accepted.

Unit 1: Chapter 3

(Regular Expression (RE) and Language)

In previous lectures, we have describe the languages in terms of machine like description-finite automata(DFA or NFA). Now we switch our attention to an algebraic description of languages, called regular expression.

Regular Expression are those algebraic expressions used for representing regular languages, the languages accepted by finite automaton. Regular expressions offer a declarative way to express the strings we want to accept. This is what the regular expression offer that the automata do not. Many system uses regular expression as input language. Some of them are:

- Search commands such as UNIX grep.
- Lexical analyzer generator such as LEX or FLEX. Lexical analyzer is a component of compiler that breaks the source program into logical unit called tokens.

Defining Regular expressions

A regular expression is built up out of simpler regular expression using a set of defining rules. Each regular expression ‘r’ denotes a language L(r). The defining rules specify how L(r) is formed by combining in various ways the languages denoted by the sub expressions of ‘r’.

Here is the method:

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

Basic steps:

-Φ is a regular expression representing empty language.
-Є is a regular expression representing the language of empty strings. i.e.{ Є}
- if ‘a’ is a symbol in Σ, then ‘a’ is a regular expression representing the language {a}.

Now the following operations over basic regular expression define the complex regular expression as:

-if ‘r’ and ‘s’ are the regular expressions representing the language L(r) and L(s) then

- r U s is a regular expression denoting the language L(r) U L(s).
- r.s is a regular expression denoting the language L(r).L(s).
- r* is a regular expression denoting the language (L(r))*.
- (r) is a regular expression denoting the language (L(r)). (this denote the same language as the regular expression ‘r’ denotes.

Note: any expression obtained from Φ, Є, a using above operation and parenthesis where required is a regular expression.

Regular operator:

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

Union (U / | /+): If L_1 and L_2 are any two regular languages then

$L_1 U L_2 = \{s \mid s \varepsilon L_1, \text{ or } s \varepsilon L_2\}$

For Example:
$L_1 = \{00, 11\}$, $L_2 = (Є, 10\}$
$L_1 U L_2 = \{Є, 00, 11, 10\}$

Concatenation (.): If L_1 and L_2 are any two regular languages then,
$L_1.L_2 = \{l_1.l_2 | l_1 \varepsilon L_1 \text{ and } l_2 \varepsilon L_2\}$

For examples: L1 = {00, 11} and L2 = {Є, 10}
L1.L2={00,11,0010,1110}
L2.L1={1000,1011,00,11}
So L1.L2 !=L2.L1

Kleen Closure (*):

If L is any regular Language then,
$L_* = L_i = L_0 U L_1 U L_2 U$…………

Precedence of regular operator:

1. The star operator is of highest precedence. i.e it applies to its left well formed RE.
2. Next precedence is taken by concatenation operator.
3. Finally, unions are taken.

Examples: Write a RE for the set of string that consists of alternating 0's and 1's over {0,1}.

First part: we have to generate the language {01,0101,0101,…………………}
Second part we have to generate the language {10,1010,101010……………}

So lets start first part.
Here we start with the basic regular expressions 0 and 1 that represent the language {0} and {1} respectively.

Now if we concatenate these two RE, we get the RE 01 that represent the language {01}.
Then to generate the language of zero or more occurrence of 01, we take Kleen closure. i.e. the RE (01)* represent the language {01,0101,…………..}
Similarly, the RE for second part is (10)*.
Now finally we take union of above two first part and second part to get the required RE. i.e. the RE (01)*+(10)* represent the given language.

Regular language:

Let Σ be an alphabet, the class of regular language over Σ is defined inductively as;
- Φ is a regular language representing empty language
- {Є} is a regular language representing language of empty strings.
- For each a ε Σ, {a} is a regular language.
- If L_1, L_2............ L_n is regular languages, then so is L_1U L_2U………..UL_n.
- If L_I,L_2,L_3,…………..L_n are regular languages, then so is L_1.L_2.L_3………L_n

-If L is a regular language, then so is L*

Note: strictly speaking a regular expression E is just expression, not a language. We should use L(E) when we want to refer to the language that E denotes. However it is to common to refer to say E when we really mean L(E).

Application of regular languages:

Validation: Determining that a string complies with a set of formatting constraints. Like email address validation, password validation etc.

Search and Selection: Identifying a subset of items from a larger set on the basis of a pattern match.

Tokenization: Converting a sequence of characters into words, tokens (like keywords, identifiers) for later interpretation.

Algebraic Rules/laws for regular expression

1. ***Commutativity:*** *Commutative of oerator means we can switch the order of its operands and get the same result. The union of regular expression is commutative but concatenation of regular expression is not commutative. i.e. if r and s are regular expressions representing like languages L(r) and L(s) then, r+s =s+r i.e. r U s = s U r but r.s ≠s.r.*

2. *Associativity: The unions as well as concatenation of regular expressions are associative. i.e. if t, r, s are regular expressions representing regular languages L(t), L(r) and L(s) then,*

   t+(r+s) = (t+r)+s

   And t.(r.s) = (t.r).s

3. ***Distributive law:** For any regular expression r, s, t representing regular language L(r), L(s) and L(t) then,*

   r(s+t) = rs+rt ------ left distribution.

   (s+t)r = sr+tr ------ right distribution.

4. ***Identity law:** Φ is identity for union. i.e. for any regular expression r representing regular expression L(r).*

   r + Φ = Φ + r = r i.e. ΦUr=r.

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

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

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

   -(r)*=r**

   -Closure of Φ = Φ = Є*

   -Closure of Є = Є = Є*

   -Positive closure of r, r_+ = rr.*

Examples

Consider Σ = {0, 1}, then some regular expressions over Σ are ;

- 0*10* is RE that represents language {w|w contains a single 1}
- $\Sigma^*1\Sigma^*$ is RE for language{w|w contains at least single 1}
- Σ*001 Σ* = {w|w contains the string 001 as substring}
- (Σ Σ)* or ((0+1)*.(0+1)*) is RE for {w|w is string of even length}
- 1*(01*01*)* is RE for {w|w is string containing even number of zeros}
- 0*10*10*10* is RE for {w|w is a string with exactly three 1's}

- For string that have substring either 001 or 100, the regular expression is $(1+0)^*.001.(1+0)^*+(1+0)^*.(100).(1+0)^*$
- For strings that have at most two 0's with in it, the regular expression is $1^*.(0+\epsilon).1^*.(0+\epsilon).1^*$
- For the strings ending with 11, the regular expression is $(1+0)^*.(11)^+$
- Regular expression that denotes the C identifiers:

  (Alphabet + _)(Alphabet + digit + _)*

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

Proof:
Let w = xy be a string, now to prove the theorem it is sufficient to show that 'w' ε LM U LN.

Now first consider "if part":

Let w ε LM U LN This implies that, w ε L(M) or w ε L(N) (by union rule)
i.e. xy ε LM or xy ε LN

Also,
xy ε LM implies x ε L and y ε M (by concatenation rule)

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

This implies that
x εL and y ε (M U N) then xy ε L(M U N) (concatenating above)

This implies that w ε L(M U N)

Now consider "only if" part:
Let w ε L(M U N) => xy ε L(M U N)

Now,
xy ε L(M U N) => x ε L and y ε (M U N) (by concatenation)
y ε (M U N) => y ε M or y ε N (by union rule)

Now, we have x ε L
Here if y ε M then xy ε L(M) (by concatenation)
And if y ε N then xy ε L(N) (by concatenation)

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

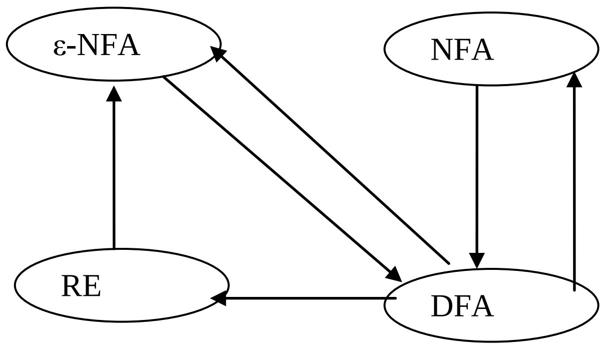
Thus,
We have, L (M U N) = L(M) U L(N)

Finite Automata and Regular expression

The regular expression approach for describing language is fundamentally different from the finite automaton approach. However, these two notations turn out to represent exactly the same set of languages, which we call regular languages. In order to show that the RE define the same class of language as Finite automata, we must show that:

1)Any language define by one of these finite automata is also defined by RE.
2)Every language defined by RE is also defined by any of these finite automata.

We can proceed as:



1. RE to NFA conversion

   We can show that every language L(R) for some RE R, is also a language L(E) for some epsilon NFA. This say that both RE and epsilon-NFA are equivalent in terms of language representation.

Theorem 1
Every language defined by a regular expression is also defined by a finite automaton. [For any regular expression r, there is an Є-NFA that accepts the same language represented by r].

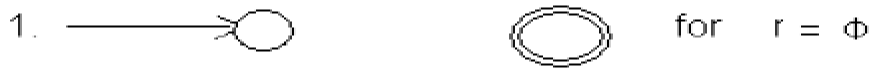
Proof:

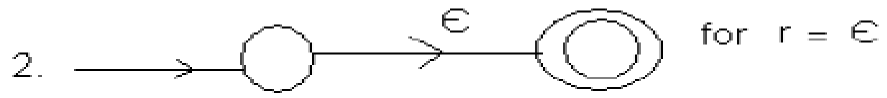
Let L =L(r) be the language for regular expression r, now we have to show there is an Є-NFA E such that L (E) =L.

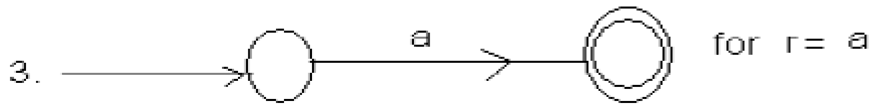
The proof can be done through structural induction on r, following the recursive definition of regular expressions.

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

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





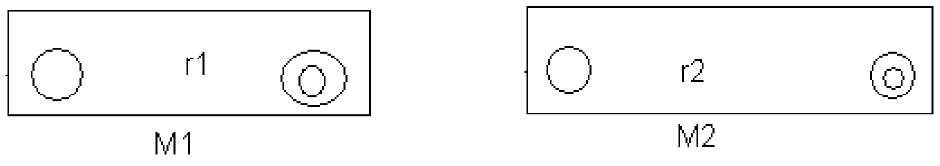


This forms the basis steps.

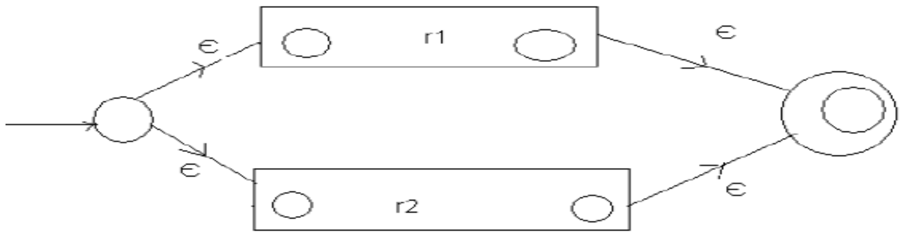
Now the induction parts are shown below

Let r be a regular expression representing language L(r) and r1,r2 be regular expressions for languages L(r1) and L(r2),

1. For union '+': From basis step we can construct Є-NFA's for r1 and r2. Let the Є-NFA's be M1 and M2 respectively

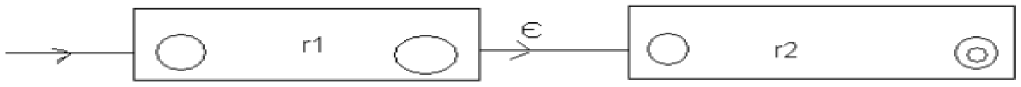


Then, $r=r_1+r_2$ can be constructed as:



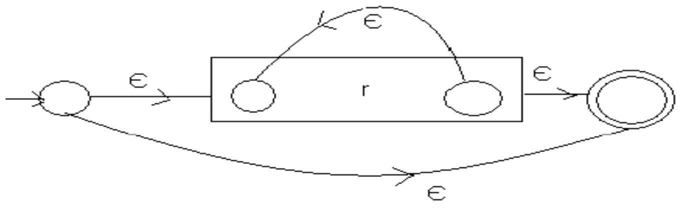
The language of this automaton is L(r1) U L(r2) which is also the language represented by expression r1+r2.

2. For concatenation '.': Now, $r = r_1.r_2$ can be constructed as;



Here, the path from starting to accepting state go first through the automaton for r1, where it must follow a path labeled by a string in L(r1), and then through the automaton for r2, where it follows a path labeled by a string in L(r2). Thus, the language accepted by above automaton is L(r1).L(r2).

3. For *(Kleen closure)
Now, r* Can be constructed as;

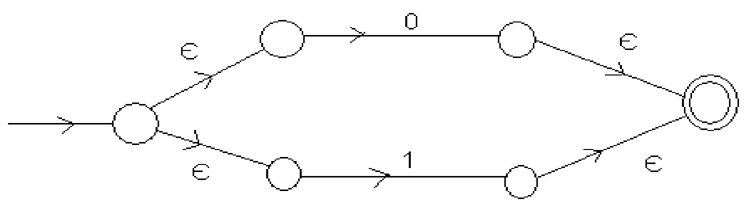


Clearly language of this Є-NFA is L(r*) as it can also just Є as well as string in L(r), L(r)L(r), L(r)L(r)L(r) and so on. Thus covering all strings in L(r*).

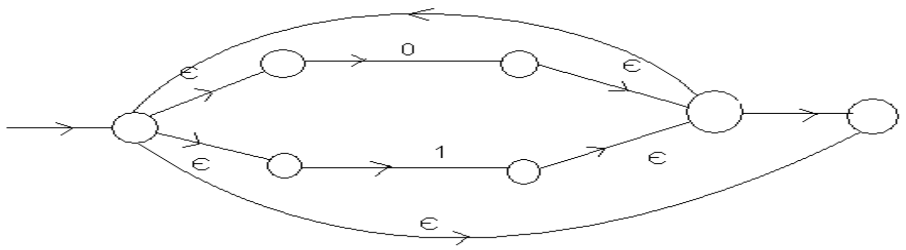
Finally, for regular expression (r), the automaton for r also serves as the automaton for (r), since the parentheses do not change the language defined by the expression.

This completes the proof.

Examples (Conversion from RE to Є-NFA)
1. For regular expression (1+0) the Є-NFA is:

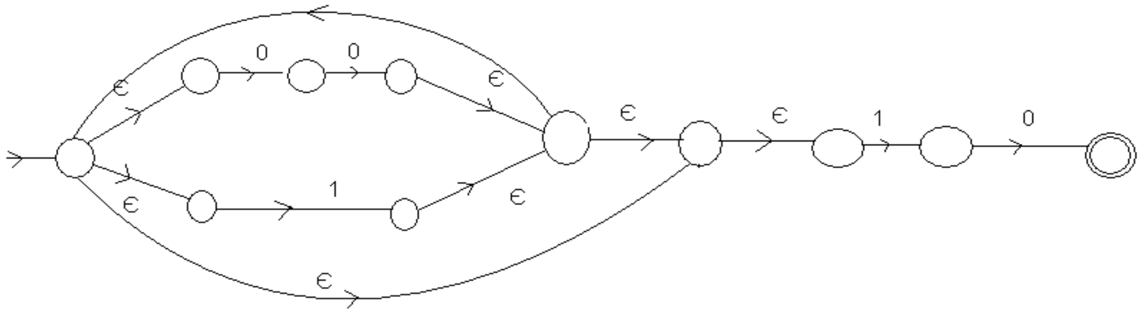


2. for (0+1)*, the ϵ-NFA is:



3. Now, ϵ-NFA for whole regular expression (0+1)*1(0+1)

4. For regular expression (00+1)*10 the ϵ-NFA is as:



Conversion from DFA to Regular Expression(DFA to RE)

Arden's Theorem

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

Proof:

Here, r = q + rp ……………… (i)

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

r = q + (q + rp)p

$r = q + qp + rp^2$………………(ii)

Again putting value of r = q + rp in relation (ii), we get;
$r = q + qp + (q + rp)\, p^2$
$r = q + qp + qp^2 + qp^3$………………

Continuing in the same way, we will get as;
$r = q + qp + qp^2 + qp^3$………………..
$r = q(\epsilon + p + p^2 + p^3 +$…………………..

Thus r = qp* Proved.

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

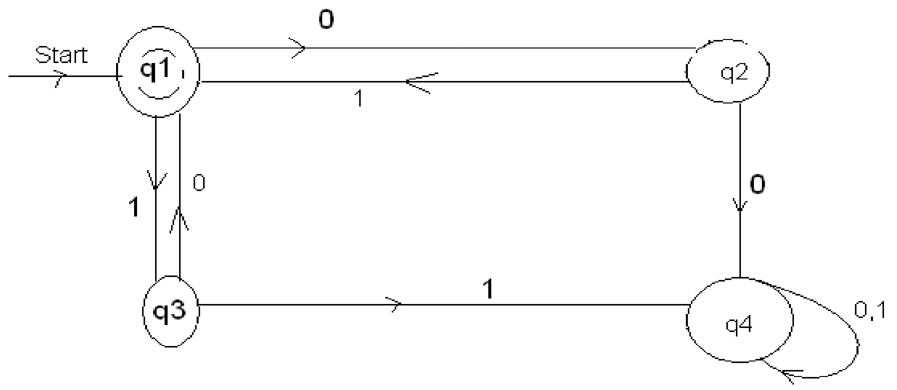
To convert the given DFA into a regular expression, here are some of the assumptions regarding the transition system:

- The transition diagram should not have the Є-transitions.
- There must be only one initial state.
- The vertices or the states in the DFA are as;
  $q_1, q_2,$……………..q_n (Any qi is final state)
- W_{ij} denotes the regular expression representing the set of labels of the edjes from q_i to q_j.
  Thus we can write expressions as;

$q_1=q_1w_{11}+q_2w_{12}+q_3w_{31}+$……………… $q_nw_{n1}+\epsilon$
$q_2=q_1w_{12}+q_2w_{22}+q_3w_{32}+$……………… $+q_nw_{n2}$
$q_3=q_1w_{13}+q_2w_{23}+q_3w_{33}+$……………… $+q_nw_{n3}$
…………………………………………………
…………………………………………………
…………………………………………………
$q_n=q_1w_{1n}+q_2w_{n2}+q_3w_{n3}+$……………………… q_nw_{nn}

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

Examples: Convert the following DFA into regular expression.



Let the equations are:
q1=q21+q30+ Є……….(i)
q2=q10…………………(ii)
q3=q11…………………..(iii)
q4=q20+q31+q40+ q41……(iv)

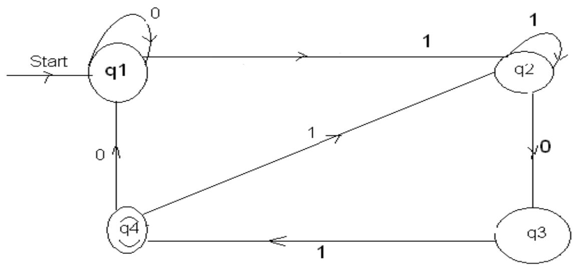
Putting the values of q2 and q3 in (i)

q1=q101+q110+ Є
i.e.q1=q1(01+10)+ Є
i.e.q1= Є+q1(01+10) (since r = q+rp)
i.e. q1= Є(01+10)* (using Arden's rule)

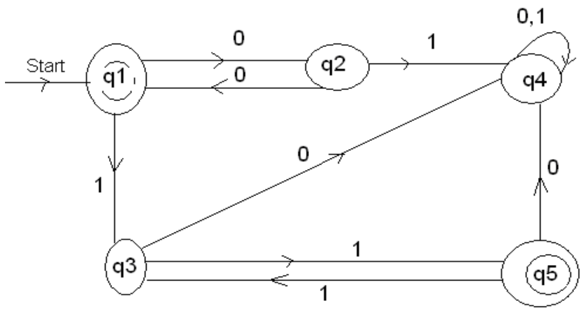
Since, q1 is final state, the final regular expression for the DFA is
Є(01+10)* = (01+10)*

Exercise: Try some Question from text book as well as from Adesh Kumar Pandey's book.Here I have maintion some of them.

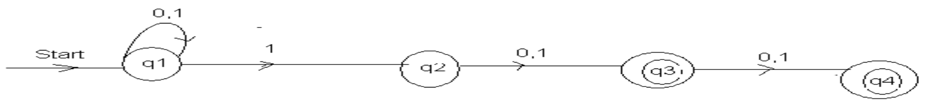
1. Convert this DFA into RE.



2.Convert this DFA into RE



3.Convert this NFA into RE



Proving Langauge not to be Regular

It is shown that the class of language known as regular language has at least four different descriptions. They are the language accepted by DFA's, by NFA's, by Є-NFA, and defined by RE.
Not every language is Regular. To show that a langauge is not regular, the powerfull technique used is known as Pumping Lemma.

Pumping Lemma

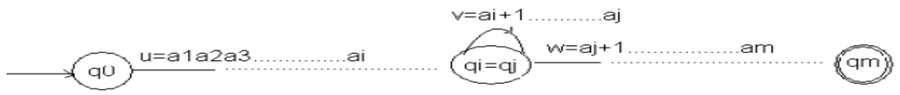
Statement: Let L be a regular language. Then, there exists an integer constant n so that for any $x \varepsilon L$ with $|x| \geq n$, there are strings u, v, w such that $x = uvw$, $|uv| \leq n$, $|v| > 0$.
Then $uv^kw \varepsilon L$ for all $k \geq 0$.

Note: Here y is the string that can be pumped i.e repeating y any number of times or deleting it, keeps the resulting string in the language.

Proof:

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

Now suppose;
$X = a_1a_2a_3\ldots\ldots\ldots a_m$ where each $a_i \varepsilon \Sigma$ be an input symbol to M. Now, consider for j = 1,…………n, qj be states of M
Then,
$\hat{\delta}(q_0,x) = \hat{\delta}(q_0,a_1a_2\ldots\ldots a_m)$ [q0 being start state of M]
$= \hat{\delta}(q_1,a_2\ldots\ldots a_m)$
=…………………
=…………………
=…………………
$= \hat{\delta}(q_m,Є)$ [qm being final state]
Since $m \geq n$, and DFA M has only n states, so by pigeonhole principle, there exists some i and j; $0 \leq i < j \leq m$ such that $q_i = q_j$.



Now we can break x=uvw as
u = a1a2…………..ai
v =ai+1……………aj
w =aj+1……………am

i.e. string ai+1 ………………aj takes M from state qi back to itself since qi = qj. So we can say M accepts $a_1a_2\ldots\ldots a_i(a_{i+1}\ldots\ldots a_j)^K a_{j+1}\ldots\ldots a_m$ for all $k \geq 0$.

Hence, uvkw ε L for all k≥0.

Application of Pumping Lemma:

To prove any language is not a regular language.

For example:

1. Show that language, $L=\{0_r\ 1_r|n \geq 0\}$ is not a regular language.

Solution:

Let L is a regular language. Then by pumping lemma, there are strings u, v, w with v≥1 such that uv^kw ε L for k≥0.

Case I:

Let v contain 0's only. Then, suppose $u = 0^p$, $v = 0^q$,$w = 0^r1^s$;
Then we must have p+q+r = s (as we have 0^r1^r) and q>0

Now, $uv^kw = 0^p(0^q)^k0^r1^s=0^{p+qk+r}1^s$
Only these strings in $0^{p+qk+r}1^s$ belongs to L for k=1 otherwise not.
Hence we conclude that the language is not regular.

Case II

Let v contains 1's only. Then $u= 0^p1^q$, $v= 1^r$, $w=1^s$
Then p= q+r+s and r>0

Now, $0^p1^q(1^r)^k1^s=0^p1^{q+rk+s}$
Only those strings in 0^p1^{q+rk+s} belongs to L fpr k =1 otherwise not.
Hence the language is not regular.

Case III

V contains 0's and 1's both. Then, suppose,
$u = 0^p$, $v = 0^q1^r$, $w = 1^s$;
p+q = r+s and q+r>0

Now, $uv^kw = 0^p(0^q1^r)^k1^s= 0^{p+qk}1^{rk+s}$
Only those strings in $0^{p+qk}1^{rk+s}$ belongs to L for k=1, otherwise not. (As it contains 0 after 1 for k>1 in the string.)
Hence the language is not regular.

Minimization of DFA

Given a DFA M, that accepts a language L (M). Now, configure a DFA M '. During the course of minimization, it involves identifying the equivalent states and distinguishable states.

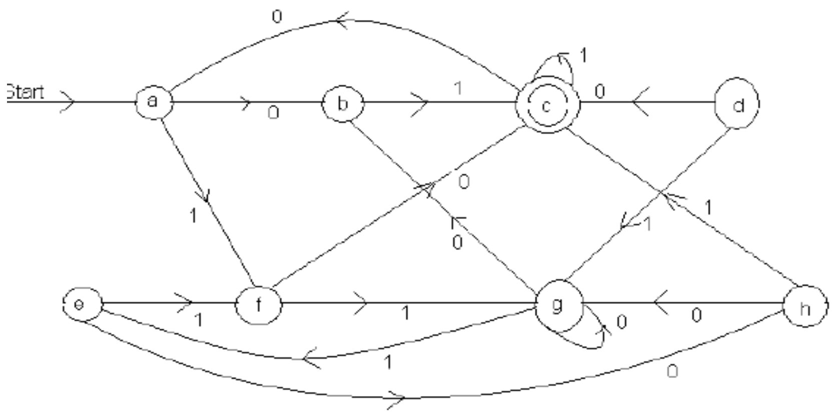
Equivalent States: Two states p & q are called equivalent states, denoted by p ≡ q if and only if for each input string x, $\hat{\delta}$ (p, x) is a final state if and only if $\hat{\delta}$(q, x) is a final state.

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

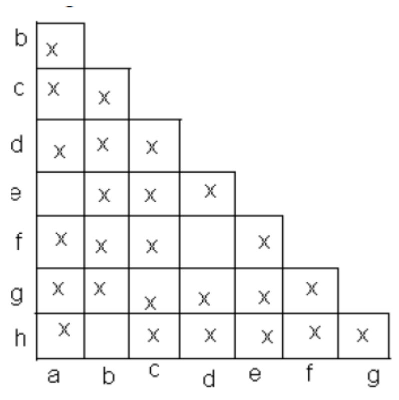
For minimization, the table filling algorithm is used. The steps of the algorithm are;
For identifying the pairs (p, q) with p ≠ q;

- List all the pairs of states for which p ≠ q.
- Make a sequence of passes through each pairs.
- On first pass, mark the pair for which exactly one element is final (F).
- On each sequence of pass, mark the pair (r, s) if for any a ε Σ, δ(r, a) = p and δ(s, a) = q and (p, q) is already marked.
- After a pass in which no new pairs are to be marked, stop
- Then marked pairs (p, q) are those for which p q and unmarked pairs are those for which p ≡ q.

Example



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



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

For pair (b, a)
(δ(b, 0), δ(a, 0)) = (g, h) – unmarked
(δ(b, 1), δ(a, 1)) = (c, f) – marked

For pair (d, a)
(δ(d, 0), δ(a, 0)) = (c, b) – marked
Therefore (d, a) is distinguishable.

For pair (e, a)
(δ(e, 0), δ(a, 0)) = (h, h) – unmarked.
(δ(e, 1), δ(a, 1)) = (f, f) –unmarked.
[(e, a) is not distinguishable)]

For pair (g, a)
(δ(g, 0), δ(a, 0)) = (a, g) – unmarked.
(δ(g, 1), δ(a, 1)) = (e, f) – unmarked

For pair (h, a)
(δ(h, 0), δ(a, 0)) = (g, h) –unmarked
(δ(h, 1), δ(a 1) = (c, f) – marked
Therefore (h, a) is distinguishable.

For pair (d, b)
(δ(d, 0), δ(b,0)) = (c, g) – marked
Therefore (d, b) is distinguishable.

For pair (e, b)
(δ(e, 0), δ(b,0)) = (h, g) –unmarked
(δ(e, 1), δ(b,1) = (f, c) – marked.

For pair (f, b)
(δ(f, 0), δ(b,0)) = (c, g) – marked

For pair (g, b)
(δ(g, 0), δ(b, 0)) = (g, g) – unmarked
(δ(h, 1), δ(b, 1)) = (e, c) – marked

For pair (h, b)
(δ(h, 0), δ(b, 0)) = (g, g) – unmarked
(δ(h,1), δ(b,1)) = (c, c) - unmarked.

For pair (e, d)
(δ(e, 0), δ(d, 0)) = (h, c) – marked
(e, d) is distinguishable.

For pair (f, d)
(δ(f, 0), δ(d, 0)) = (c, c) – unmarked
(δ(f,1), δ(f,1)) = (g, g) - unmarked.

For pair (g, d)
(δ(g, 0), δ(d, 0)) = (g, c) – marked

For pair (h, d)
(δ(h, 0), δ(d, 0)) = (g, c) – marked

For pair (f, e)
(δ(f, 0), δ(e, 0)) = (c, h) – marked

For pair (g, e)
(δ(g, 0), δ(e, 0)) = (g, h) – unmarked
(δ(g,1), δ(e,1)) = (e, f) -marked.

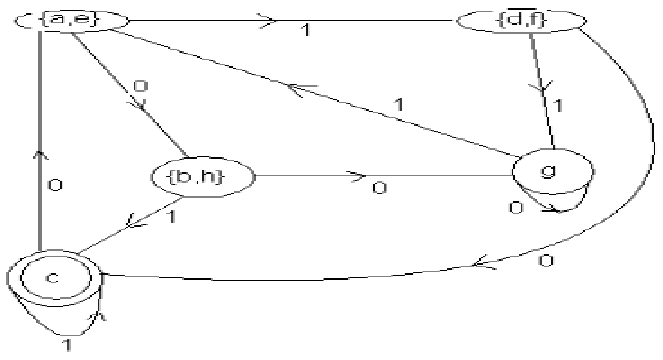
For pair (h, e)
(δ(h, 0), δ(e, 0)) = (g, h) – unmarked
(δ(h,1), δ(e,1)) = (c, f) -marked.

For pair (g, f)
(δ(g, 0), δ(f, 0)) = (g, c) – marked

For pair (h, f)
(δ(h, 0), δ(f, 0)) = (g, c) – marked

For pair (h, g)
(δ(h, 0), δ(g, 0)) = (g, g) – unmarked
(δ(h,1), δ(g,1)) = (c, e) -marked.
Thus (a, e), (b, h) and (d, f) are equivalent pairs of states.

Hence the minimized DFA is



Another simple approaches from Asesh Kumar Pandey's Book is

Context Free Grammars and Languages

Grammars are used to generate the words of a language and to determine whether a word is in a language. Formal languages, which are generated by grammars, provide models for both natural languages, such as, English, and for programming languages, such as C, JAVA.

Context free grammars are used to define syntax of almost all programming languages, in context of computer science. Thus an important application of context free grammars occurs in the specification and compilation of programming languages.

Formal Description of Context Free Grammar:

A context free grammar is defined by 4-tuples (V, T, P,S) where,
V= set of variables
T= set of terminal symbols
P= set of rules and productions
S= start symbol and S ε V

There are four components in CFG.

1. There is a finite set of symbol that form the strings of language being defined. We call this alphabet the terminal or terminal symbols. In above tuples, it is represented by T
2. There is a finite set of variables, also called sometimes non-terminals or non-terminal symbols or syntactic categories'. Each variable represents a language i.e a set of strings.
3. One of the variables represents the language being defined. It is called the start symbol.
4. There is a finite set of productions or rules that represent the recursive definition of a language. Each production consists of :
    a. A variable that is being defined by the production. This is called head of production.
    b. The production symbol $\rightarrow$
    c. A string of Zero or more terminals and Variables. This string is called the body of the production, represents one way to form the string in the language of the head.

Note: The variable symbols are after represented by capital letters. The terminals are analogous to the input alphabet and are often represented by lower case letters. One of the variables is designed as a start variable. It usually occurs on the left hand side of the topmost rule.

For example,
$S \rightarrow \epsilon$
$S \rightarrow 0S1$

This is a CFG defining the grammar of all the strings with equal no of 0's followed by equal no of 1's.

Here,

the two rules define the production P,
Є, 0, 1 are the terminals defining T,
S is a variable symbol defining V
And S is start symbol from where production starts.

CFG Vs RE

The CFG are more powerful than the regular expressions as they have more expressive power than the regular expression. Generally regular expressions are useful for describing the structure of lexical constructs as identical keywords, constants etc. But they do not have the capability to specify the recursive structure of the programming constructs. However, the CFG are capable to define any of the recursive structure also. Thus, CFG can define the languages that are regular as well as those languages that are not regular.

Compact Notation for the Productions
It is convenient to think of a production as "belonging" to the variable of head. we may use the terms like A-production or the production of A to refer to the production whose head is A. We write the production for a grammar by listing each variable once and then listing all the bodies of the production for that variable, separated by |. This is called compact Notation.

Example
CFG representing the language over Σ = {a, b} which is palindrome language.
S→ Є | a | b ……………………………..Grammar (1)
S→a S a
S→b S b

Here the first production is written in compact notation. The detail of this production look like as
S→ Є
S→a
S→b

Meaning of context free:

Consider an example:
S → a M b
M→ A | B
A→ Є | aA
B→ Є | bB

How, consider a String aaAb, which is an intermediate stage in the generating of aaab. It is natural to call the strings "aa" and "b" that surround the symbol A, the "context" of A in this particular string. Now, the rule A→ aA says that we can replace A by the string aA no matter what the surrounding strings are; in other words, independently of the context of A.

When there is a production of form $lw_1r \rightarrow lw_2r$ (but not of the form $w_1 \rightarrow w_2$), the grammar is context sensitive since w_1 can be replaced by w_2 only when it is surrounded by the strings “l” and”r”.

Derivation Using Grammar Rule:

We apply the production rule of a CFG to infer that certain strings are in the language of a certain variable. This is the process of deriving strings by applying productions rules of the CFG.

General Convention used in Derivation

- Capital letters near the beginning of the alphabet A, B and so on are used as variables.
- Capital letters near the end of the alphabet such as X, Y are used as either terminals or variables.
- Lower case letters near the beginning of the alphabet, a, b and so on are used for terminals.
- Lower case letters near the end of alphabet, such as ‘w’ or ‘z’ are used for string of terminals.
- Lower case Greek letters such as α and β are used for string of terminal or variables.

Derivation:

A derivation of a context free grammar is a finite sequence of strings β0 β1 β2………βn such that:

- For $0 \leq i \leq n$, the string βi ε (V U T)*
- B0 = S
- For $0 \leq i \leq n$, there is a production of P that applied to βi yields βi+1
- Bn ε T*

There are two possible approaches of derivation:

- Body to head (Bottom Up) approach.
- Head to body (Top Down) approach.

Body to head

Here, we take strings known to be in the language of each of the variables of the body, concatenate them, in the proper order, with any terminals appearing in the body, and infer that the resulting string is the language of the variables in the head.

Consider grammar,

$S \rightarrow S + S$
$S \rightarrow S/S$ ………………………..Grammer(2)
$S \rightarrow (S)$
$S \rightarrow S-S$
$S \rightarrow S*S$
$S \rightarrow a$

Here given a + (a*a) / a – a

Now, by this approach.

SN	String Inferred	Variable(For Language of)	Production	String Used
1	a	S	S→a	
2	a*a	S	S→S*S	String 1
3	(a*a)	S	S→(S)	String 2
3	(a*a)/a	S	S→S/S	String 3 and string 1
4	a+(a*a)/a	S	S→S+S	String 1 and 3
5	a+(a*a)/a-a	S	S→S-S	String 4 and 1

Thus, in this process we start with any terminal appearing in the body and use the available rules from body to head.

Head to Body
Here, we use production from head to body. We expand the start symbol using a production, whose head is the start symbol. Here we expand the resulting string until all strings of terminal are obtained. Here we have two approaches:

Leftmost Derivation: Here leftmost symbol (variable) is replaced first.
Rightmost Derivation: Here rightmost symbol is replaced first.

For example: consider the previous example of deriving string
a+ (a*a)/a-a with the grammar(2)

Now leftmost derivation for the given string is:

S→ S + S	rule → S + S
S→a + S	rule S→ a [Note here we have replaced left symbol of the body]
S→a + S – S	rule S→S - S
S→a + S / S – S	rule S→ S / S
S→a + (S) /S – S	rule S→(S)
S→(S*S) / S – S;	rule S→S*S
S→ a + (a*S) / S – S	rule S→a
S→ a + (a*a) / S – S	" "
S→ a + (a*a) / a – S	" "
S→a + (a*a) / a – a	" "

And rightmost derivation is;

S→ S – S;	rule S→S - S
S→ S – a;	rule S→ a
S→S + S – a;	rule S→ S + S
S→S + S / S – a;	rule S→S / S
S→S + S / a –a;	rule S→ a
S→S + (S) / a – a;	rule S→ (S)
S→S + (S*S) / a – a;	rule S→ S*S

$S \rightarrow S + (S*a) / a - a;$ rule $S \rightarrow a$
$S \rightarrow S + (a*a) / a - a;$
$S \rightarrow a + (a*a) / a - a;$

Direct Derivation:
$\alpha_1 \rightarrow \alpha_2$: If α_2 can be derived directly from α_1, then it is direct derivation.
$\alpha_1 \rightarrow^* \alpha_2$: If α_2 can be derived from α_1 with zero or more steps of the derivation, then it is just derivation.

Example
$S \rightarrow aSa \mid ab \mid a \mid b \mid \epsilon$

Direct derivation:
$S \rightarrow ab$
$S \rightarrow aSa$
$S \rightarrow aaSaa$
$S \rightarrow aaabaa$

Thus S□* aaabaa is just a derivation.

Language of Grammar (Context Free Grammar):

Let G = (V, T, P and S) is a context free grammar. Then the language of G denoted by L(G) is the set of terminal strings that have derivation from the start symbol in G.
i.e. $L(G) = \{x \varepsilon T^* \mid S \rightarrow^* x\}$
The language generated by a CFG is called the Context Free Language (CFL).

Sentential forms
Derivations from the start symbol produce strings that have a special role. We call these "sentential forms". i.e. if G=(V,T,P,S) is a CFG, then any string α is in $(V \cup T)^*$ such that $S \rightarrow^* \alpha$ is a sentential form. If $S \rightarrow_{lm}^* \alpha$, then α is a left sentential forms, and if $S \rightarrow_{rm}^* \alpha$, then α is a right sentential form.

Note: The language L(G) is those sentential forms that are in T* i.e. they consist solely of terminals.

Derivation Tree / Parse Tree:
Parse tree is a tree representation of strings of terminals using the productions defined by the grammar. A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.

Parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding the replacement order.

Formally, given a Context Free Grammar G = (V, T, P and S), a parse tree is a n-ary tree having following properties:

- The root is labeled by the start symbol.
- Each interior node of parse tree are variables.
- Each leaf node of parse is labeled by a terminal symbol or Є.
- If an interior node is labeled with a non terminal A and its childrens are x_1, x_2 ……………x_n from left to right then there is a production P as:
  A→ x_1, x_2, ………………x_n for each x_i ε T.

Example

Consider the grammer

S→ aSa | a | b| Є

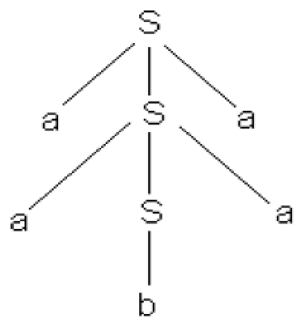
Now for string S→* aabaa

We have,

S→ aSa

S→aaSaa

S→aabaa So the parse tree is→:



Exercise

a)Consider the Grammar G:

S→ A1B

A→0A | Є

B→0B | 1B | Є

1.Construct the parse tree for 00101

2.Construct the parse tree for 1001

3.Construct the parse tree for 00011

b) Consider the grammer for the arithematic expression

E→ EOPE | (E) |id

OP→ + | - | * | / | ↑

1. **Construct the parse tree for id + id *id**
2. **Construct the parse tree for (id +id)*(id + id)**

c) Construct a CFG that generates language of balanced parentheses:

-Show the parse tree computation for

1.( ) ( )

2.((( ))) ( )

Solution:

The CFG is G=(V,T,P,S)
S→ SS
S→(S)
S→ Є
Where,
V = {S}
S = {S}
T = {(,)}
& P = as listed above

Now the parse tree for ( ) ( ) is:

And the parse tree for (((( )))) ( )

Ambiguity in Grammar:
A Grammar G = (V, T, P and S) is said to be ambiguous if there is a string w ε L (G) for which we can derive two or more distinct derivation tree rooted at S and yielding w. In other words, a grammar is ambiguous if it can produce more than one leftmost or more than one rightmost derivation for the same string in the language of the grammar.

Example
S→ AB |aaB
A→ a | Aa
B → b

For any string aab;

We have two leftmost derivations as;
S→AB the parse tree for this derivation is:
→AaB
→aaB
→ aab

Also,
S→ aaB the parse tree for this derivation is:
→aab

Normal forms and Simplification of CFG

The goal of this section is to show that every context free language (without Є) is generated by a CFG in which all production are of the form A□ BC or A□ a, where A,B and c are variables, and a is terminal. This form is called Chomsky Normal Form. To get there, we need to make a number of preliminary simplifications, which are themselves useful in various ways;

1. We must eliminate "useless symbols": Those variables or terminals that do not appear in any derivation of a terminal string from the start symbol.

2. We must eliminate "Є-production": Those of the form A→ Є for some variable A.
3. We must eliminate "unit production": Those of the form A□ B for variables A and B.

Eliminating Useless Symbols:

We say a symbol x is useful for a grammar G = (V, T, P, S) if there is some derivation of the form S →* αxβ →*w, where w is in T*. Here, x may be either variable or terminal and the sentential form αxβ might be the first or last in the derivation. If x is not useful, we say it is useless.

Thus useful symbols are those variables or terminals that appear in any derivation of a terminal string from the start symbol. Eliminating a useless symbol includes identifying whether or not the symbol is "generating" and "reachable".

<u>Generating Symbol:</u> We say x is generating if x→*w for some terminal string w: Note that every terminal is generated since w can be that terminal itself, which is derived by zero steps.

<u>Reachable symbol</u>: We say x is reachable if there is derivation S →* αxβ for some α and β.

Thus if we eliminate the non generating symbols and then non-reachable, we shall have only the useful symbols left.

Example
Consider a grammar defined by following productions:
S→aB | bX
A→ Bad | bSX | a
B→ aSB | bBX
X→SBd | aBX | ad

Here;
A and X can directly generate terminal symbols. So, A and X are generating symbols. As we have the productions A→ a and X→ ad.

Also,
S→ bX and X generates terminal string so S can also generate terminal string. Hence, S is also generating symbol.
B can not produce any terminal symbol, so it is non-generating.
Hence, the new grammar after removing non-generating symbols is:

S→ bX
A→ bSX | a
X→ ad

Here,

A is non-reachable as there is no any derivation of the form S→* α A β in the grammar. Thus eliminating the non-reachable symbols, the resulting grammar is:

S→ bX
X→ ad

This is the grammar with only useful symbols.

Exercise

1) **Remove useless symbol from the following grammar:**

S→ xyZ | XyzZ
X→Xz | xYZ
Y→yYy | XZ
Z→ Zy | z

2) Remove useless symbol from the following grammar

S→ aC | SB
A→ bSCa
B→aSB | bBC
C→ aBc | ad

Eliminating Є-productions:

A grammar is said to have Є-productions if there is a production of the form A□ Є. Here our strategy is to begin by discovering which variables are "nullable". A variable 'A' is "nullable" if A→* Є.

Algorithm (Steps to remove Є-production from the grammar):

- If there is a production of the form A□ Є, then A is "nullable".
- If there is production of the form B□ X1, X2………. And each Xi's are nullable then B is also nullable.
- Find all the nullable variables.
- If B→X1, X2……………. Xn is a production in P then add all productions P' formed by striking out some subsets of there Xi's that are nullable.
- Do not include B□ Є if there is such production.

Example:

Consider the grammar:

S→ABC
A→ BB | Є
B→ CC | a
C→ AA | b

Here,

A→ Є A is nullable.
C→AA→* Є, C is nullable
B→ CC→* Є, B is nullable
S→ABC→* Є, S is nullable

Now for removal of Є –production:
In production
S→ABC, all A, B and C are nullable. So, striking out subset of each the possible combination of production gives new productions as:

S→ ABC | AB | BC | AC | A | B | C

Similarly for other can be done and the resulting grammar after removal of e-production is:

S→ABC | AB | BC | AC | A | B | C
A→BB | B
B→CC | C | a
C→AA | A | b

Exercise:
Remove Є-productions for each of grammar;
1)

S→ AB
A→ aAA | Є
B → bBB | Є

Eliminating Unit Production:

A unit production is a production of the form A→ B, where A and B are both variables.

Here, if A→ B, we say B is A-derivable. B→ C, we say C is B-derivable.
Thus if both of two A→ B and B→C, then A→* C, hence C is also A-derivable.
Here pairs (A, B), (B, C) and (A, C) are called the unit pairs.

To eliminate the unit productions, first find all of the unit pairs. The unit pairs are;
(A, A) is a unit pair for any variable A as A→* A
If we have A→ B then (A, B) is unit pair.
If (A, B) is unit pair i.e. A→B, and if we have B→ C then (A, C) is also a unit pair.

Now, to eliminate those unit productions for a, gives grammar say G = (V, T, P, S), we have to find another grammar G' = (V, T, P', S) with no unit productions. For this, we may workout as below;

- ✓ Initialize P' = P
- ✓ For each A ε V, find a set of A-derivable variables.
- ✓ For every pair (A, B) such that B is A-derivable and for every non-unit production B→ α, we add production A → α is P' if it is not in P' already.
- ✓ Delete all unit productions from P'.

Example
Remove the unit production for grammar G defined by productions:

P = {S→ S + T | T
T→ T* F | F
F→ (S) | a
};

Initialize

1) P’ = {S→ S + T | T
T→ t*F | F
F→ (S) | a }
2) Now, find unit pairs;

Here, S→ T So, (S, T) is unit pair.
T→ F So, (T, F) is unit pair.
Also, S→T and T→ F So, (S, F) is unit pair.

3) Now, add each non-unit productions of the form B →α for each pair (A, B);
P’ = {
S→S + T |T * F| (S) | a
T→T * F | (S) | a | F
F→ (S) | a
}
4) Delete the unit productions from the grammar;
P’ = {
S→ S + T | T * F | (S) | a
T→ T * F | (S) | a
F→ (S) | a
}

Exercise

1) Simply the grammar G = (V, T, P, S) defined by following productions.
S→ ASB | Є
A→ aAS | a
B→ SbS | A | bb | Є

Note: Here simplify means you have to remove all the useless symbol, Unit production and **Є**-productions.

2) Simplify the grammar defined by following production:
S→ 0A0 | 1B1 | BB
A→ C
B→ S | A
C→S | Є

Chomsky Normal Form

A context free grammar G = (V, T, P, S) is said to be in Chomsky's Normal Form (CNF) if every production in G are in one of the two forms;

$A \rightarrow BC$ and

$A \rightarrow a$ where A, B, C ε V and a ε T

Thus a grammar in CNF is one which should not have;

Є-production

Unit production

Useless symbols.

Theorem: Every context free language (CFL) without Є-production can be generated by grammar in CNF.

Proof:

If all the productions are of the form $A \rightarrow a$ and $A \rightarrow BC$ with A, B, C ε V and a ε T, we have done.

Otherwise, we have to do two task as:

1. Arange that all bodies of length 2 or more consist only of variable
2. Break bodies of length 3 or more into a cascade of production , each with a body consisting of two variable.

The construction for task (1) is as follows

if the productions are of the form: $A \rightarrow X1, X2, \ldots\ldots Xm$, m>2 and if some X_i is terminal a, then we replace the X_i by C_a having $C_a \rightarrow a$ where C_a is a variable itself.

Thus as result we will have all productions of the form:

$A \rightarrow B_1B_2\ldots\ldots B_m$, m>2; where all B_i's are non-terminal.

The construction for task (2) is as follows

We break those production $A \rightarrow B_1B_2\ldots\ldots B_m$ for m>=3, into group of production with two variables in each body. We introduce m-2 new variables $C_1, C_2, \ldots\ldots C_{m-2}$. The original production is replaced by the m-1 productions

$A \rightarrow B_1C_1$,

$C_1 \rightarrow B_2C_2$,

……..

……..

…

$C_{k-2} \rightarrow B_{k-1}B_k$

Finally, all of the productions are achieved in the form as:

$A \rightarrow BC$ or $A \rightarrow a$

This is certainly a grammar in CNF and generates a language without Є-productions.

Consider an example:

$S \rightarrow AAC$

$A \rightarrow aAb \mid Є$

$C \rightarrow aC \mid a$

1) Now, removing Є- productions;

Here, A is nullable symbol as $A \rightarrow \epsilon$
So, eliminating such ϵ-productions, we have;
$S \rightarrow AAC \mid AC \mid C$
$A \rightarrow aAb \mid ab$
$C \rightarrow aC \mid a$

2) Removing unit-productions;

Here, the unit pair we have is (S, C) as $S \rightarrow C$
So, removing unit-production, we have;
$S \rightarrow AAC \mid AC \mid aC \mid a$
$A \rightarrow aAb \mid ab$
$C \rightarrow aC \mid a$
Here we do not have any useless symbol. Now, we can convert the grammar to CNF. For this;

→First replace the terminal by a variable and introduce new productions for those which are not as the productions in CNF.
i.e.
$S \rightarrow AAC \mid AC \mid C_1C \mid a$
$C_1 \rightarrow a$
$A \rightarrow C_1AB_1 \mid C_1B_1$
$B1 \rightarrow b$
$C \rightarrow C_1C \mid a$

Now, replace the sequence of non-terminals by a variable and introduce new productions.
Here, replace $S \rightarrow AAC$ by $S \rightarrow AC_2$, $C_2 \rightarrow AC$
Similarly, replace $A \rightarrow C_1AB_1$ by $A \rightarrow C_1C_3$, $C_3 \rightarrow AB_1$

Thus the final grammar in CNF form will be as;
$S \rightarrow AC_2 \mid AC \mid C_1C \mid a$
$A \rightarrow C_1C_3 \mid C_1b_1$
$C_1 \rightarrow a$
$B_1 \rightarrow b$
$C_2 \rightarrow AC$
$C_3 \rightarrow AB_1$
$C \rightarrow C_1C \mid a$

Exercise
Simplify following grammars and convert to CNF;
1)
$S \rightarrow ASB \mid \epsilon$
$A \rightarrow aAS \mid a$
$B \rightarrow SbS \mid A \mid bb$

2) $S \rightarrow AACD$
$A \rightarrow aAb \mid \epsilon$
$C \rightarrow aC \mid a$
$D \rightarrow aDa \mid bDa \mid \epsilon$

3)
S→ aaaaS | aaaa

Left recursive Grammar

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation (A →* A α) for some string x.
The top down parsing methods can not handle left recursive grammars, so a transformation that eliminates left recursion is needed.

Removal of immediate left recursion:
Let A→ Aα | β, where β does not start with A. Then, the left-recursive pair of productions could be replaced by the non-left-recursive productions as;
A→ βA'
A'→ α A' | Є; without changing the set of strings derivable from A.

Or equvalnetly, these production can be rewritten as with out Є- production
A→ βA'| β
A'→ α A' | α;

No matter how many A-productions there are, we can eliminate immediate left recursion from them.

So, in general,
If A→ A α1 |A α2 | …………………….|A αm | β1 | β2 | ……………| βn|; With βi does not start with A**.**

Then
we can remove left recursion as:
A→ β1A' | β2A'| …………………. | βnA'
A'→ α1A' | α2A' | ………………... | αmA' | Є

Equivalently, these productions can be rewritten as:
A→ β1A' | β2A'| …………………. | βnA' | β1 | β2 |………..| βn
A'→ α1A' | α2A' | ………………... | αmA' | α1 | α2| ………..| αm

Consider an example

S→AA | 0
A→ AAS | 0S | 1 *(Where AS is of the form =* α1, *0S of the form =β1 and 1of the form=β2)*

Here, the production A→ AAS is immediate left recursive. So removing left recursion,

we have;
S→AA | 0
A→0SA' | 1A'
A' → ASA' | Є

Equivalently, we can write it as:

$S \rightarrow AA \mid 0$
$A \rightarrow 0SA' \mid 1A' \mid 0S \mid 1A'$
$A' \rightarrow ASA' \mid AS$

Exercise

For the following grammar
$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid a$
Remove the Left recursion.

Greibach Normal Form (GNF)

A grammar G = (V, T, P and S) is said to be in Greibach Normal Form, if all the productions of the grammar are of the form:

$A \rightarrow a\alpha$, where a is a terminal, i.e. $a \in T^*$ and α is a string of zero or more variables. i.e. $\alpha \in V^*$
So we can rewrite as:

$A \rightarrow aV^*$ with $a \in T^*$
Or,

$A \rightarrow aV^+$
$A \rightarrow a$ with $a \in T$

This form called Greibach Normal Form, after Sheila Greibach, Who first gave a way to construct such grammars. Converting a grammar to this form is complex, even if we simplify the task by, say, starting with a CNF grammar.

To convert a grammar into GNF;
- Convert the grammar into CNF at first.
- Remove any left recursions.
- Let the left recursion tree ordering is $A_1, A_2, A_3, \ldots\ldots\ldots\ldots\ldots A_p$
- Let A_p is in GNF
- Substitute A_p in first symbol of A_{p-1}, if A_{p-1} contains A_p. Then A_{p-1} is also in GNF.
- Similarly substitute first symbol of A_{p-2} by A_{p-1} production and A_p production and so on…………

Consider an Example;
$S \rightarrow AA \mid 0$
$A \rightarrow SS \mid 1$
This Grammar is already in CNF.
Now to remove left recursion, first replace symbol of A-production by S-production (since we do not have immediate left recursion) as:

$S \rightarrow AA \mid 0$
$A \rightarrow AAS \mid 0S \mid 1$ *(Where AS is of the form =* $\alpha 1$, *0S of the form =$\beta 1$ and 1of the form=$\beta 2$)*

Now, removing the immediate left recursion;

S→AA | 0
A→0SA' | 1A'
A'→ ASA' | Є

Equivalently, we can write it as:
S→ AA | 0
A→ 0SA' | 1A' | 0S | 1
A'→ASA' | AS

Now we replace first symbol of S-production by A-production as;
S→AASA' | 1A'A | 0SA | 1A | 0
A→ 0SA' | 1A' | 0S | 1
A'→ ASA' | AS

Similarly replacing first symbol of A'-production by A-production, we get the grammar in GNF as;
S→0SA'A | 1A'A | 0SA | 1A | 0
A→ 0SA' | 1A' | 0S | 1
A'→ 0SA'SA' | 1A'SA' | 0SSA' | 1SA'| 0SA'S | 1A'S | 0SS | 1

Regular Grammar:

A regular grammar represents a language that is accepted by some finite automata called regular language. A regular grammar is a CFG which may be either left or right linear. A grammar in which all of the productions are of the form A→ wB (*wB is of the form α)* or A→ w for A, B ε V and w ε T* is called left linear.

Equivalence of Regular Grammar and Finite Automata

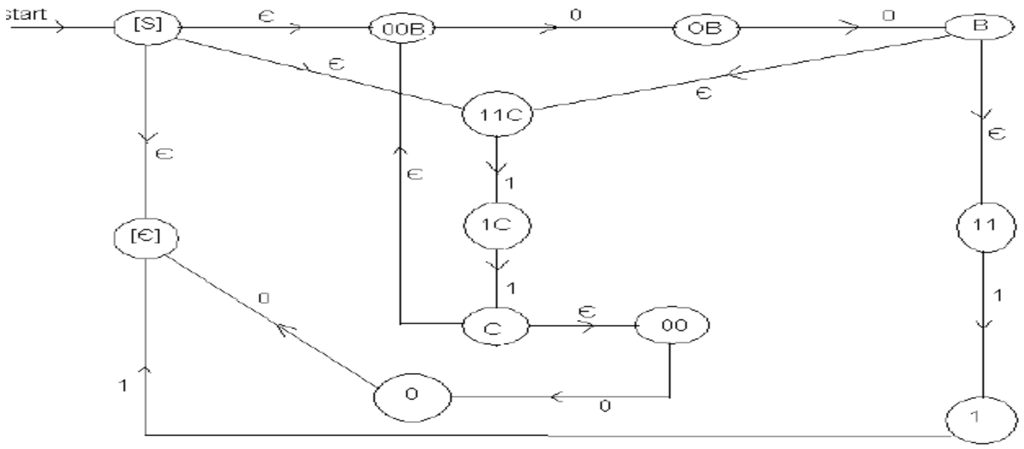
1. Let G = (V, T, P and S) be a right linear grammar of a language L(G). we can construct a finite automata M accepting L(G) as;

M = (Q, T, δ, [S]. {[Є]})

Where,
Q – consists of symbols [α] such that α is either S or a suffix from right hand side of a production in P.
T - is the set of input symbols which are terminal symbols of G.
[S] – is a start symbol of G which is start state of finite automata.
{[Є]} – is the set of final states in finite automata.

And δ is defined as:
If A is a variable then, δ([A], Є) = [α] such that A→ α is a production.
If 'a' is a terminal and α is in T* U T*V then δ([aα], a) = [α].

For example;
S→ 00B | 11C | Є
B→ 11C | 11
C→00B | 00
Now the finite automata can be configured as;



Another way:

If the regular grammar is of the form in which all the productions are in the form as;
A→ aB or A→ a, where A, B ε V and a ε T Then, following steps can be followed to obtain an equivalent finite automaton that accepts the language represented by above set of productions.

a. The number of states in finite automaton will be one more than the number of variables in the grammar.(i.e. if grammar contain 4 variables then the automaton will have 5 states)

- Such one more additional state in the final state of the automaton.
- Each state in the automaton is a variable in the grammar.

b. The start symbol of regular grammar is the start state of the finite automaton.
c. If the grammar contains Є as S→*Є, S being start symbol of grammar, then start state of the automaton will also be final state.

d. The transition function for the automaton is defined as;

- For each production A→ aB, we write δ(A, a) = B. So there is an arc from state A to B labeled with a.
- For each production A→ a, δ(A, a) = final state of automaton
- For each production A→ Є, make the state A as finite state.

Exercise
Write the equivalent finite automata

1.
S→ abA | bB | aba
A→ b | aB | bA
B→aB | aA

2.
S→0B | 111C | Є
B→ 00C | 11
C→ 00B | 0D
D→0B | 00

3.
S→ 0A | 1B | 0 | 1
A→ 0S | 1B | 1
B→0A | 1S

4.
A→ 0B | 1D | 0
B→0D | 1C | €
C→0B | 1D | 0
D→0D | 1D

5.
A→0B | E | 1C
B→0A | F | €
C→0C | 1A | 0
E→0C | 1A | 1
F→0A | 1B | €

Pumping lemma for context free Language:
The "pumping lemma for context free languages" says that in any sufficiently long string in a CFL, it is possible to find at most two short, nearby substrings that we can "pump" in tandem (one behind another). i.e. we may repeat both of the strings I times, for any integer I, and the resulting string will still be in the language. We can use this lemma as a tool for showing that certain languages are not context free.

Size of Parse tree: Our first step in pumping lemma for CFL's is to examine the size of parse trees, During the lemma, we will use the grammar in CNF form. One of the uses of CNF is to turn parses into binary trees. Any grammar in CNF produces a parse tree for any string that is binary tree. These trees have some convenient properties, one of which can be exploited by following theorem.

Theorem: Let a terminal string w be a yield of parse tree generated by a grammar G = (V, T, P and S) in CNF. If length path is n, then $|w| \leq 2^{n-1}$

Proof: Let us prove this theorem by simple induction on n.

Basis Step: Let n = 1, then the tree consists of only the root and a leaf labeled with a terminal.
So, string w is a simple terminal.
Thus $|w| = 1 = 2^{1-1} = 2^0$ which is true.

Inductive Step: Suppose n is the length of longest path and n>1. Then the root of the tree uses a production of the form; A→BC, since n>1

No path is the sub-tree rooted at B and C can have length greater than n-1 since B&C are child of A. Thus, by inductive hypothesis, the yield of these sub-trees are of length almost 2^{n-2}.
The yield of the entire tree is the concatenation of these two yields and therefore has length at most,

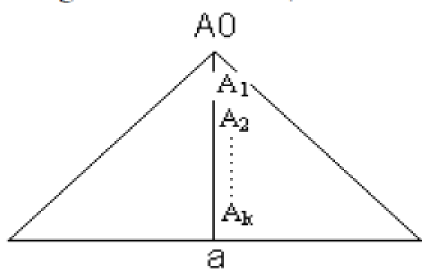
$2^{n-2} + 2^{n-2} = 2^{n-1}$
$|w| <= 2^{n-1}$. Hence proved.

Pumping Lemma

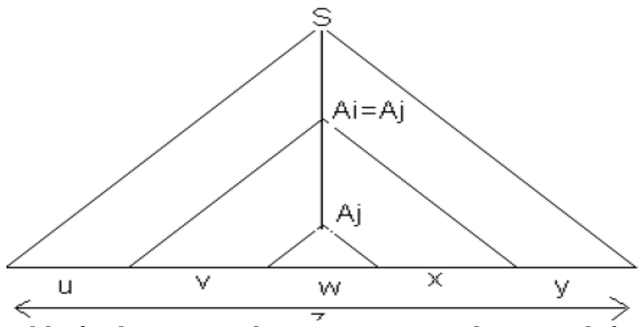
It states that every CFL has a special value called pumping length such that all longer strings in the language can be pumped. This time the meaning of pumped is a bit more complex. It means the string can be divided into five parts so that the second and the fourth parts may be repeated together any number of times and the resulting string still remains in the language.

Theorem: Let L be a CFL. Then there exists a constant n such that if z is any string in L such that |z| is at least n then we can write z=uvwxy, satisfying following conditions.
I) $|vx| > 0$
II) $|vwx| \leq n$
III) For all $i \geq 0$, $uv^iwx^iy \in L$. i.e. the two strings v & x can be “pumped” nay number of times, including ‘0’ and the resulting string will be still a member of L.

Proof: First we can find a CNF grammar for the grammar G of the CFL L that generates L-{Є}.
Let m be the number of variables in this grammar choose $n= 2^m$. Next suppose z in L is of length at least n. i.e. $|z| \geq n$
Any parse tree for z must have height m+1, other if it would be less than m+1, i.e. say m then by the lemma for size of parse tree, $|z| = 2^{m-1} = 2^m/2= n/2$ is contradicting. So it should be m+1.
Let us consider a path of maximum length in tree for z, as shown below, where k is the least m and path is of length k.

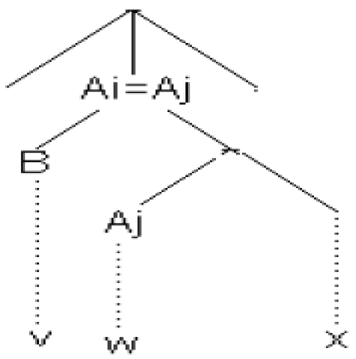


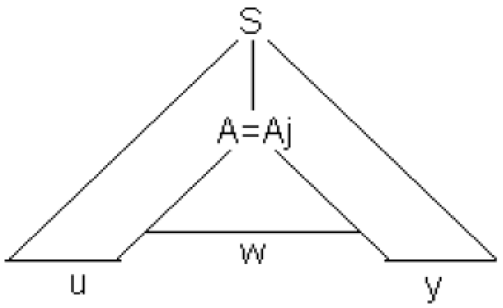
Since $k \geq m$. there are at least m+1 occurrence of variables A0, A1, A2………. Ak on the path. But there are only m variables in the grammar, so at least two of the last m+1 variable on the path must be same, (by pigeonhole principle).
Suppose Ai = Aj, then it is possible to divide tree,



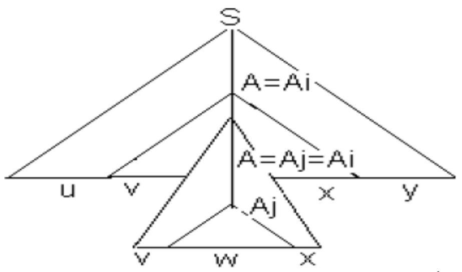
String w is the yield of sub-tree noted at Aj, string v and x are to left and right of w in yield of larger sub-tree rooted at Ai. If u and y represent beginning and end z i.e. to right and left of sub-tree rooted at Ai, then
Z= uvwxy
Since, there are not unit productions so v & x both could not be Є, i.e. empty. Means that Ai has always two children corresponding to variables. If we let B denotes the one tree is not ancestor of Aj, then since

w is derived from Aj then strings of terminal derived from B does not overlap x. it follows that either v or x is not empty. So,
$|vx| > 0$
Now, we know Ai = Aj = A say, as we found any two variables in the tree are same. Then, we can construct new parse tree where we can replace sub-tree rooted at Ai, which has yield vwx, by sub-tree rooted at Aj, which has yield w. the reason we can do so is that both of these trees have root labeled A. the resulting tree yields uv^0wx^0y as;





Another option is to replace sub-tree rooted at Aj by entire sub-tree rooted at Ai. Then the yield can be of pattern uv^2wx^2y as

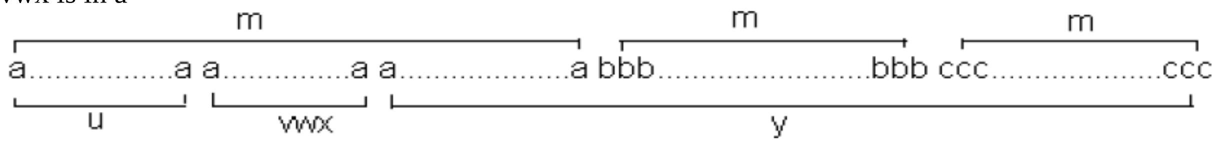


Hence, we can find any yield of the form $uv^iwx^iy \in L$ for any $i \geq 0$ Now, to show $|vwx| \leq n$. the Ai in the tree is the portion of path which is root of vwx.
Since we begin with maximum path of length with height m+1. this sub-tree rooted at Ai also have height of less than m+1.
So by lemma for height of parse tree,
$|vwx| \leq 2^{m+1-1} = 2^m = n$
$|vwx| \leq n$
Thus, this completes the proof of pumping lemma.

Example
Show that $L = \{a^nb^nc^n : n \geq 0\}$ is not a CFL

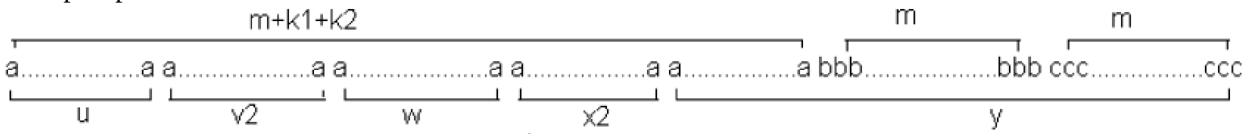
To prove L is not CFL, we can make use of pumping lemma for CFL. Let L be CFL. Let any string in the language is $a^mb^mc^m$
$z = uvwxy$, $|vwx| \leq m$, $|vx| > 0$

Case I:
vwx is in a^m


m
m
m
a................a a..............a a...................a bbb........................bbb ccc...................ccc
u
vwx
y

$v = a^{k1}$, $x = a^{k2}$; $k1+k2 > 0$ i.e. $k1+k2 \geq 1$

Now pump v and x then


m+k1+k2
m
m
a................a a..................a a...................a a................a a................a bbb..................bbb ccc................ccc
u
v2
w
x2
y

But by pumping lemma, $uv^2xy^2z = a^{m+K1+k2}b^mc^m$ does not belong to L as, k1+k2. Hence, it shows contradiction that L is CFL. So, L is not CFL.
Note: other cases can be done similarly.

Bakus-Naur Form

This is another notation used to specify the CFG. It is named so after John Bakus, who invented it, and Peter Naur, who refined it. The Bakus-Naur form is used to specify the syntactic rule of many computer languages, including Java. Here, concept is similar to CFG, only the difference is instead of using symbol "$\rightarrow$'' in production, we use symbol ::=. We enclose all non-terminals in angle brackets, < >.

For example:

The BNF for identifiers is as;
<identifier> ::= <letter or underscore> | <identifier> | <symbol>
<letter or underscore> ::= <letter> | <_>
<symbol> ::= <letter or underscore> | <digit>
<letter> ::= a | b | ……………….. | z
<digit> ::= 0 | 1| 2 | ……………….|3

Closure Property of Context Free Languages

Given certain languages are context free, and a language L is formed from them by certain operation, like union of the two, then L is also context free. These theorem lemma are often called closure properties of context free languages, since they show whether or not the class of context free language is called under the operation mentioned.
Closure properties express the idea that one (or several) languages are context free, and then certain related languages are also context free or not.

Here are some of the principal closure properties for context free languages;

A. The context free language are closed under union
i.e. Given any two context free languages L1 and L2, their union L1 U L2 is also context free language.

Proof
The inductive idea of the proof is to build a new grammar from the original two, and from start symbol of the new grammar have productions to the start symbols of the original two grammars.

Let G1 = (V1, T1, P1 and S1) and G2 = (V2, T2, P2 and S2) be two context free grammars defining the languages L(G1) and L(G2). Without loss of generality, let us assume that they have common terminal set T, and disjoint set of nonterminals. Because, the non-terminals are distinct so the productions P_1 and P_2.
Let S be a new non-terminal not in V1 and V2. Then, construct a new grammar G = (V, T, P, S) where;
V = V1 U V2 U {S}
P = P1 U P2 U {$S\rightarrow S1$, $S\rightarrow S2$}
G is clearly a context free grammar because the two new productions so added are also of the correct form, we claim that L (G) = L (G1) U L (G2).

For this, Suppose that x ε L (G1). Then there is a derivation of x ;
$S_1\rightarrow^* x$
But in G we have production, $S\rightarrow S1$
So there is a derivation of x also in G as:
$S\rightarrow S_1\rightarrow^* x$
Thus, x ε L (G). Therefore, L (G1)$\subseteq$ L (G).A similar argument shows L (G2)$\subseteq$L (G).

So, we have,

L (G1) U L (G2) is subset of L (G)

Conversely, suppose that x ε L (G). Then there is a derivation of x in G as:

$S\rightarrow \beta \rightarrow^* x$

Because of the way in which P is constructed, β must be either S1 or S2.

Suppose β =S1. Any derivation in G of the form $S1\rightarrow^* x$ must involve only productions of G1 so, $S1\rightarrow^* x$ is a derivation of x in G1.

Hence, β = S1 → x ε L(G1). A simpler argument proves that β = S1 implies x ε L (G1).

Thus L(G) is subset of L(G1) U L(G2).

It follows that L(G) = L(G1) U L(G2)

Similarly following two closure properties of CFL can be proved as for the union we have proved.

B. The CFLs are closed under concatenation

C. The CFLs are closed under kleen closure [prove yourself]

However context free languages are not closed under some cases like, intersection and complementation.

The context free languages are not closed under intersection.

To prove this, the idea is to exhibit two CFLs whose intersection results in the language which is not CFL.

For this as we learned in pumping lemma that

L = $\{a^n b^n c^n \mid n \geq 1\}$ is not CFL

However following two languages are context free;

L1 = $\{a^n b^n c^i \mid n \geq 1\ i \geq 1\}$

L2 = $\{a^i b^n c^n \mid n \geq 1\ i \geq 1\}$

Clearly,

L = L1∩L2. to see why, observe that L1 requires that there be the same number of a's + b's, while L2 requires the number of b's and c's to be equal. A string in both must have equal number of all three symbols and thus to be in L.

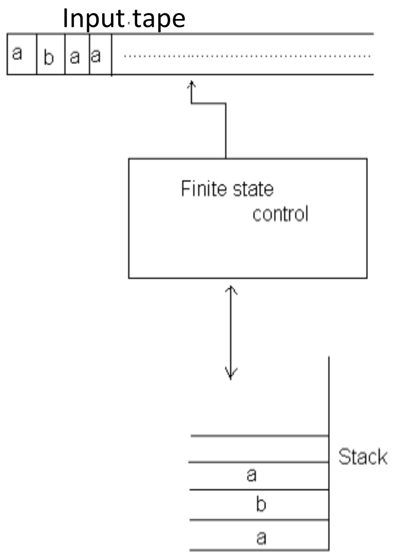
If the CFL's were closed under intersection, then we could prove the false statement that L is context free. Thus, we conclude by contradiction that the CFL's are not closed under intersection.

Pushdown Automata (PDA)

Informal Introduction

The context free languages have a type of automaton that defines them. This automaton is called "pushdown automaton" which can be thought as a Є-NFA with the addition of stack. The presence of a stack means that, the pushdown automata can remember infinity amount of information. However, the pushdown automaton can only access the information on its stack in a Last-in-first-out way.

We can define PDA informally as device shown in figure:



PDA is an abstract machine determined by following three things:

- Input table
- Finite stack control
- A stack

Each moves of the machine is determined by three things;

- The current state
- Next input symbol
- Symbol on the top of stack

The moves consist of;

- Changing state | staying on same state
- Replacing the stack top by string of zero or more symbols.

Popping the top symbol off the stack means replacing it by Є.
Pushing Y on the stack means replacing stack's top, say X, by YX.
of stack corresponds to stack's top. The single move of the machine contains only one stack operation either push or pop.

Formal Definition:
A PDA is defined by seven tuples (Q, Σ, Γ, δ, q0, z0, F)
Where,

Q- Finite set of states.
Σ- Finite set of input symbols | alphabets.
Γ- Finite set of stack alphabet
q0- Start state of PDA q0 ε Q
z0- Initial stack symbol; z0 ε Γ
F- Set of final states; F is subset of Q
δ- Transition function that maps $Q \times (\Sigma \cup \{Є\}) \times \Gamma \rightarrow Q \times \Gamma^*$

as for finite automata, δ governs the behavior of PDA. Formally, δ takes as argument a triple δ (q,a,X) where

- q is a state.
- 'a' is either an input symbol in Σ or Є. (note Є does not belongs to Σ)
- X is a stack symbol.

The output of δ is a finite set of pairs (p, γ), where p is the new state and γ is the string on the stack after transition.

i.e the moves of PDA can be interpreted as;

$\delta(q, a, z) = \{(P_1, \gamma_1)\ (p_2, \gamma_2)\ \ldots\ldots\ldots\ldots\ldots(p_m, \gamma_m)\}$ here q, p_i ε Q, a ε Σ U Є & z ε Γ, γ_i ε Γ*

Graphical notation of PDA

We can use transition diagram to represent a PDA, where

- Any state is represented by a node in graph (diagram)
- Any arc labeled with "start" indicates the start state and doubly circled states are accepting / final states.
- The arc corresponds to transition of PDA as:
  Arc labeled as **a, x | α** means the transition δ(q, a, x) = (p,α) for arc from state p to q.

Example: A PDA accepting a string over {a, b} such that number of a's and b's are equal.
i.e. L={w | w ε {a, b}* and a's and b's are equal}.

The PDA that accepts the above language can be constructed using the idea that the PDA should push the input symbol if the top of the stack symbol is same as it otherwise Pop the stack.
For this, let us construct a PDA as;

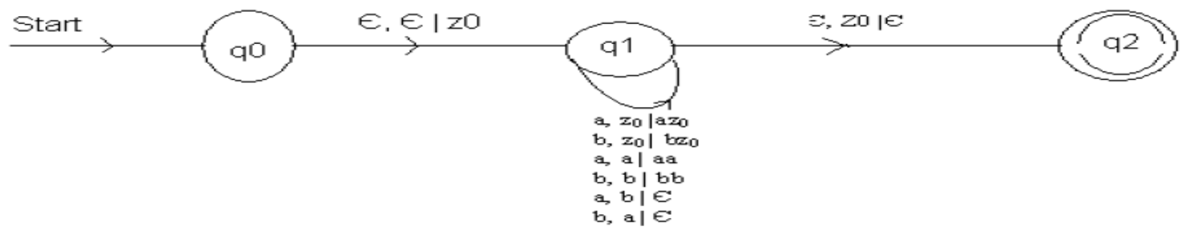
P = {Q, Σ, Γ, δ, q0, z0, F} be the PDA recognizing the given language. where, let us suppose

Q = {q0, q1, q2}
Σ = {a, b}
Γ = {a, b, z0}
z0 = z0
q0 = q0
F = {q2}

Now δ is defined as;

1. δ(q0, Є, Є) = (q1, z0) //initialize the stack to indicate the bottom of stack.
2. δ(q1, a, z0) = (q1, az0)
3. δ(q1, b, z0) = (q1, bz0)
4. δ(q1, a, a) = (q1, aa)
5. δ(q1, b, b) = (q1, bb)
6. δ(q1, a, b) = (q1, Є)
7. δ(q1, b, a) = (q1, Є)
8. δ(q1, Є, z0) = (q2, Є) // indicate the acceptance of string.

So the graphical notation for the PDA constructed in example 1 can be constructed as;



Let us trace for w= aabbbaab

The execution of PDA on the given string can be traced as shown on the table;

S.N	State	unread string	Stack	Transition Used
1	q_0	aabbbaab	Є	--
2	q_1	aabbbaab	z_0	1
3	q_1	abbbaab	az_0	2
4	q_1	bbbaab	aaz_0	4
5	q_1	bbaab	az_0	7
6	q_1	baab	z_0	7
7	q_1	aab	bz_0	3
8	q_1	ab	z_0	6
9	q_1	b	az_0	2
10	q_1	Є	z_0	7
11	q_2	Є	Є	8

Finally we have q2 state. Hence accepted.

Exercise
1) Trace as above for string aababb
2) Trace as above for string ababa (note this string will be not accepted.i.e. PDA will not reach to final state)

Instantaneous Description of PDA (ID)

Any configuration of a PDA can be described by a triplet (q, w, r).

Where,

q- is the state.

w- is the remaining input.

γ- is the stack contents.

Such a description by triple is called an instantaneous description of PDA (ID). For finite automaton, the $\hat{\delta}$ notation was sufficient to represent sequences of instantaneous description through which a finite automaton moved, since the ID for a finite automaton is just its state. However, for PDA's we need a notation that describes changes in the state, the input and stack. Thus, we adopt the notation of instantaneous description.

Let P = {Q, Σ, Γ, δ, q0, z0, F} be a PDA. Then, we define a relation ├, "yields as (q, aw, zα) ├ (p, w, βα) if δ(q, a, z) contains (p, β, and may be Є). This move reflects the idea that, by consuming "a" from the input, and replacing z on the top of stack by β, we can go from state q to state p.

Note that what remains on the input w, and what is below the top of stack, β, do not influence the action of the PDA.

For the PDA described earlier accepting language of equal a's and b's, [see example 1] the accepting sequence of ID's for string 1001 can be shown as;

(q0, 1001, Є) ├ (q1, 1001, z_0)
├ (q1, 001, 1z_0)
├ (q1, 01, z_0)
├ (q1, 1, 0z_0)
├ (q1, Є, z_0)
|- (q2, Є , Є) Accepted

Therefore (q_0, 1001, z_0) ├* (q_2, Є, Є)

Language of a PDA

We can define acceptance of any string by a PDA in two ways;
1) Acceptance by final state:
- Given a PDA P, the language accepted by final state, L(P) is;
{w | (q, w, z0) ├* (P, Є, γ)} where P ε F and γ ε Γ*.

2) Acceptance by empty stack:
- Given a PDA P, the language accepted by empty stack, L(P), is
{w | (q, w, z0) ├* (P, Є, Є)} where P ε Q.

Deterministic Pushdown Automata (DPDA)

While the PDAs are by definition, allowed to be Non-deterministic, the deterministic PDA is quite important. In practice the parsers generally behave like Deterministic PDA, so the class of language accepted by these PDAs are practically important. DPDA are suitable for use in programming language.

A pushdown automata P = (Q, Σ, Γ, δ, q0, z0, F) is deterministic pushdown automata if there is no configuration for which P has a choice of more than one moves. i.e. P is deterministic if following two conditions are satisfied;
1) For any q ε Q, x ε Γ, If δ(q, a, x) ≠Φ for some a ε Σ then δ(q, Є, x) = Φ
i.e. if δ(q, a, x) is non-empty for some a, then δ(q, Є, x) must be empty.
2) For any q ε Q, a ε Σ U {Є} and x ε Γ, δ(q, a, x) has at most one element.

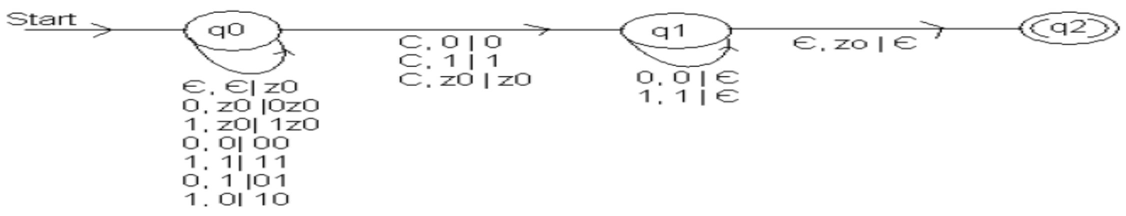
Example: A DPDA accepting language L = {w c w^R | w ε (0+1)*} is constructed as;

P = ({q0, q1, q2}, {0, 1}, {0, 1, z0}, δ, q0, z0, {q2}) where δ is defined as;

1. δ(q0, Є, Є) = (q0, z0) //initialize the stack
2. δ(q0, 0, z0) = (q0, 0z0)
3. δ(q0, 1, z0) = (q0, 1z0)
4. δ(q0, 0, 0) = (q0, 00)
5. δ(q0, 1, 1) = (q0, 11)
6. δ(q0, 0, 1) = (q0, 01)

7. δ(q0, 1, 0) = (q0, 10)
8. δ(q0, C, 0) = (q1, 0) //change the state when centre is reached.
9. δ(q0, C, 1) = (q1, 1) // change the state when centre is reached
10. δ(q0, C, z0) = (q1, z0)// change the state when centre is reached
11. δ(q1, 0, 0) = (q1, Є)
12. δ(q1, 1, 1) = (q1, Є)
13. δ(q1, Є, z0) = (q2, Є)

The graphical notation for the above PDA is:



Example: A DPDA accepting strings of balanced ordered parenthesis P;
Q = (q0, q1, q2), Σ = {{,}, (,), [,]}, Γ = Σ U {z0}, δ, q0, z0, {q2}

Where δ is defined as:
1. δ(q0, Є, Є) = (q0, z0)
2. δ(q0, {, z0) = (q1, {z0)
3. δ(q0, [, z0) = (q1, [z0)
4. δ(q0, (, z0) = (q1, (z0)
5. δ(q1, {, {) = (q1, {{)
6. δ(q1, [, [ ) = (q1, [[ )
7. δ(q1, (, ( ) = (q1, (( )
8. δ(q1, }, {) = (q1, Є)
9. δ(q1,], [ ) = (q1, Є)
10. δ(q1,), ( ) = (q1, Є)
11. δ(q1,), {) = (q1, ({)
12. δ(q1, {, [ ) = (q1, {[ )
13. δ(q1, Є, z0) = (q2, Є)

Evaluate or trace for the string [{ }]
(q0, [{ }], z0) ├ (q1, { }], [z0)
├ (q1,}], {[z0)
├ (q1,], [z0)
├ (q1, Є, z0)
├ (q2, Є, Є) Accepted

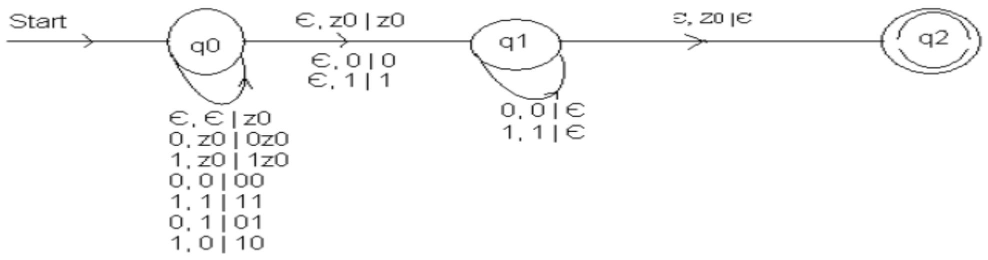
1.Try for the string [{( ) }].

2.try for the string [( ) ( )]

3.Try for the string {[}]. Note: this should be rejected.

Exercise

2. Construct a PDA accepting Language L = {ww^R | w ε (0+1)* and w^R is reversal of w}



Equivalence of CFG and PDA
Now we will show that the language defined by PDA's are exactly the context free language. The goal is to prove that the CFG and PDA accept the same class of languge. For this we show:

1) From CFG to PDA:
Given a CFG, G = (V, T, P and S), we can construct a push down automaton, M which accepts the language generated by the grammar G, i.e. L(M) = L (G).

The machine can be defined as;
M = ({q}, T, V U T, δ, q, S, Φ)
Where,
Q = {q} is only the state in the PDA.
Σ = T
Γ = V U T (i.e. PDA uses terminals and variables of G as stack symbols)
z0 = S (initial stack symbol is start symbol in grammar)
F = Φ
q0=q
And
δ can be defined as;

i. δ(q, Є, A) = {q, α) / A→ α is a production P of G.
ii. δ(q, a, a) = { (q, Є), for all a ε T}

Here all transition occurs on the only one state q.

Alternatively, we can define push down automata for the CFG with two states p & q, where p being start state. Here idea is that the stack symbol initially is supposed to be Є, and at first PDA starts with state P and reading Є, it inserts start symbol 'S' of CFG into stack and changes the state to q. Then all transitions occur in state q.
i.e. the PDA can be defined as;
M = ({p, q}, T, V U T, δ, p, {q}); stack top is Є.

Then δ can be defined as;
δ(p, Є, Є) = {(q, S) / S is start symbol of grammar G.
δ(q, Є, A) = {(q, α) / A→ α is a production P of G.
δ(q, a, a) = {(q, Є), for all a ε Σ.

Consider an example;

Let G = (V, T, P and S) where,
P is

E→T | E + T
T→ F | T*F
F→ a | (E)

We can define a PDA equivalent to this grammar as;
M = ({q0}, {a, *, +, (,)},{a, *, +, (,), E, T, F}, δ, q0, E, Φ}

Where δ can be defined as;

δ(q0, Є, E) = {(q0, T), (q0,E + T)}
δ(q0, Є, T) = {(q0, F), (q0,T*F)}
δ(q0, Є, F) = {(q0, a), (q0,(E))}
δ(q0, a, a) = {(q0, Є)}
δ(q0, *, *) = {(q0, Є)}
δ(q0, +, +) = {(q0, Є)}
δ(q0, (, ( ) = {(q0, Є)}
δ(q0,),)) = {(q0, Є)}

Now with this PDA, M, let us trace out acceptance of a + (a*a);

(q0, a + (a*a), E) ├ (q0, a + (a*a), E + T)
├ (q0, a + (a*a), T + T)
├ (q0, a + (a*a), F + T)
├ (q0, a + (a*a), a + T)
├ (q0, + (a*a), + T)
├ (q0, (a*a), T)
├ (q0, (a*a), F)
├ (q0, (a*a), (E))
├ (q0, a*a), E))
├ (q0, a*a), T))
├ (q0, a*a), T*F)
├ (q0, a*a), F*F)
├ (q0, a*a), a*F)
├ (q0, *a), *F)

├ (q0, a), F))

├ (q0, a), a))

├ (q0,),))

├ (q0, Є, Є) Accepted (acceptance by empty stack).

In CFG

E→ E + T
→ E + T
→ T + T
→F + T
→a + T
→a + F
→a + (E)
→a + (T*F)
→ a + (F*F)
→a + (a*F)
→a + (a*a)

Exercise
Convert the grammar defined by following production into PDA;
S→0S1 | A
A→ 1S0 | S | . Also we trace acceptance of aaabaaaaa.

3) From PDA to CFG:

Given a PDA M = (Q, Σ, Γ, δ, q0, z0, F); F = Φ, we can obtain an equivalent CFG, G = (V, T, P and S) which generates the same language as accepted by the PDA M.
The set of variables in the grammar consists of;
- The special symbol S, which is start symbol.
- All the symbols of the form [p x q]; p, ε Q and X ε Γ, where p and q are state in Q
i.e. V = {S} U {[p x q]}
The terminal in the grammar T = Σ

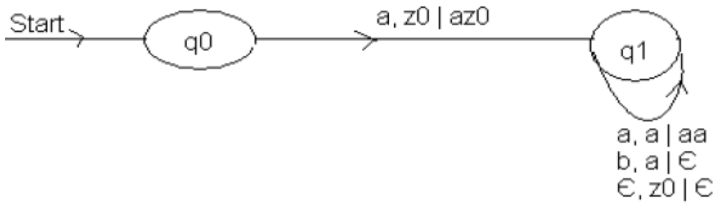
The production of G is as follows;
- For all states q ε Q, S→ [q0, z0, q] is a production of G.
- For any states p, q ε Q, x ε Γ and a ε Σ U {Є} if δ(q, a, x) = (p, Є) then [p × q]→ a
- For any states p, q ε Q, x ε Γ and a ε Σ U {Є}, if δ(q, a, x) = (p, Y1, Y2,……..Yk); where Y1, Y2, ……Yk ε Γ and k ≥ 0, Then for all lists of states P1, P2, …………..Pk, G has the production [p x q] → a [pY_1P_1][$P_1Y_2P_2$]…………..[$P_{k-1}Y_kP_k$].

This last production says that one way to pop x and go from stack q to state P_k is to read "a" (which may be Є), then use some input to pop Y1 off the stack while going from state P to P1, then read some more input that pops Y2 off the stack and goes from P1 t oP2 and so on………

For Example: Let a PDA that recognizes a language

L = {$a^n b^n$ | n >0} be defined as;
1. δ(q0, a, z0) =(q1, az0)
2. δ(q0, a, a) =(q1, aa)
3. δ(q0, b, a) =(q1, Є)
4. δ(q0, Є, z0) =(q1, Є)

The graphical representation is



Let G = (V, T, P and S) be the equivalent CFG for the given PDA where,
V= {S} U {[p × q] / p, q ε Q, x ε Γ}
S= is the start state.
T= Σ
And P is defined by following production;
1. S→[q0 z0 q0] | [q0 z0 q1]
    i.e. S□ [q0 z0 r2]; for r2 in {q0, q1}
2. From the fact that δ(q0, a, z0) contains (q1, az0), we get production
    [q0 z0 r2]→ a[q1 ar1][r1 z0r2]; for r_i in{q0, q1}
3. From the fact that δ(q1, a, a) contains (q1, aa), we get productions
    [q1 ar2]→ a[q1 ar1][r1 a r2] for ri in {q0, q1}
4. From the fact that δ(q1, b, a) contains (q1, Є), we get
    [q1aq1]→b
5. From the fact that δ(q1, Є, z0) contains (q1, Є), we get
    [q1 z0 q1]→ Є

Now acceptance of aaabbb can be shown as;

S→ [q0 z0 r2]
→a [q1 a r1] [r1 z0 r2]
→ aa [q1 z0 r1] [r1 a r2] [r1 z0 r2]
→ aaa[q1 a r1] [r1 ar2] [r1 ar2] [r1 z0 r2]
→ aaab[r1 ar2] [r1 ar2] [r1 z0 r2]
→ aaabb[r1 ar2] [r1 z0 r2]
→ aaabbb[r1 z0 r2]
→ aaabbb Є
= aaabbb

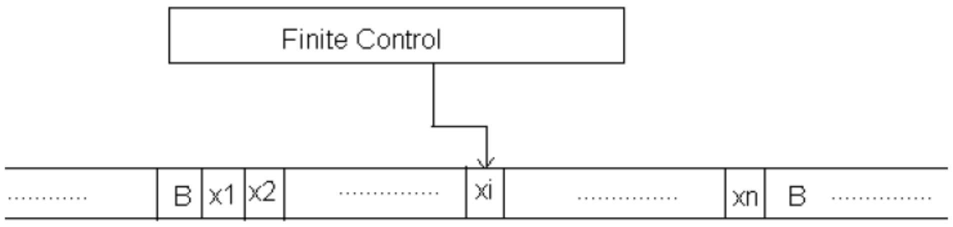
Exercise

Convert the PDA P = ({p, q}), {0, 1}, {x, z0}, δ, q ,z0) to a CFG if δ is given by
• δ(q, 1, z0) = (q, xz0)
• δ(q, 1, x) = (q, xx)
• δ(q, 0, x) = (q, x)
• δ(q, Є, z0) = (q, Є)
• δ(q, 1, x) = (p, Є)
• δ(q, 0, z0) = (q, z0)

Turing Machine

Turing machine is an abstract machine developed by an English Mathematician Alan Turing in 1936. The model of computation provides a theoretical foundation for modern computers. A Turing machine will have;

- A finite set of alphabets
- A finite set of states
- A linear tape which is potentially infinite to both end.



The tape is marked off into squares, each of which can hold one symbol from the alphabet. If there is no symbol in the square then it contains blank. The reading and writing is done by a tape head. The tape serves as:

- Input device (input is simply the string assumed to this)
- The memory available for use during computations
- The output device (output is the string of symbols left on the tape at the end of computation).

A single move of Turing machine is function of the state of TM and the current tape symbol and it consists of three things;

- Replacing the symbol in the current squared by another, possibly different symbol.
- Moving the tape head one square right or left or leaving it where it is.
- Moving from current state to another, possibly different state.

Difference between TM and Other Automata (FSA and PDA)

The most significant difference between the TM and the simpler machine (FSA or PDA) is that; in a Turing Machine, processing a string is no longer restricted to a single left to right pass through input. The tape head can move in both directions and erase or modify any symbol it encounters. The machine can examine part of the input, modify it, take time execute some computation in a different area of the tape, return to re-examine the input, repeat any of these actions and perhaps stop the processing before it has locked at all input.

Formal Description of Turing Machine

A Turing Machine T M is defined by the seven-tuples, M = (Q, Σ, Γ, δ, q0, B, F) where,

Q = the finite set of states of the finite control
Σ = the finite set of input symbols

Γ = the complete set of tape symbols Σ is always subset of Γ.
q0 = the start state; q0 ε Q
B = the blank symbol; B ε Γ but B does not belong to Σ.
F = the set of final or accepting states; F is subset of Q
δ = the transition function defined by
Q × Γ→ Q × Γ × (R, L, S); where R, L, S is the direction of movement of head left, or right or stationary. i.e. δ(q, x) = δ(p, Y, D); which means T M in state q and current tape symbol x, moves to next state P, replacing tape symbol x with Y and move the head either direction or remains at same cell of input tape.

Instantaneous Description for T M

The configuration of a T M is described by Instantaneous description (ID) of T M as like PDA.
A string x_1x_2……………..$x_{i-1}qx_ix_{i+1}$………..x_n represents the I.D. of T M in which;

- q is the state of T M .
- the tape head scanning the ith symbol from the left.
- x_1x_2……….. x_n is the portion of tape between the leftmost and rightmost non-blank.(If the head is to the left of leftmost non blank or to the right of rightmost non-blank then some prefix or suffix of x_1x_2……x_n will be blank and I will be 1 or n respectively.)

Moves of T M

The moves of T M, M = (Q, Σ, Γ, δ, q_0, B, F) is described by the notation ├, "yield", for single move and by ├* for zero or more moves as in PDA.

a) For δ(q, x_i) = (P, Y, L) i.e. next move is leftward then, x_1x_2………$x_{i-1}qx_ix_{i+1}$……..x_n ├x_1x_2….x_{i-1} P $x_{i-1}Yx_{i+1}$.x_n reflects the change of state from q to p and the replacement of symbol x_i with Y and then head is positioned at i-1 (next scan is x_{i-1}).

i. If i = 1, M moves to the left of x_1 i.e. qx_1x_2………..x_n ├ pBYx_2…….x_n

ii. If i=n, Y = B, then M moves to state p and system B written over x_n joins the infinite sequence of trailing blanks which does not appear in next ID as x_1x_2……..$x_{n-1}qx_n$├ x_1x_2………$x_{n-2}px_{n-1}$

b) If δ(q, x_i)= (P, Y, R) i.e. next move is rightward then, x_1x_2…….$x_{i-1}qx_i$………..x_n ├ x_1x_2….x_{i-1} Y p x_{i+1}……..x_n, which reflects that the symbol xi is replaced with Y and head has moved to cell i+1 with change in state from p to q.

If i= n, then i+1 cell holds blank which is not part of previous ID; i.e. x_1x_2………x_{n-1} ├ x_1x_2…………… x_{n-1} YpB.

If i=1, Y=B, then the symbol B written over xi joins the infinite sequence of leading blanks and does not appear in next ID; i.e. x_1x_2………………x_n ├px_2x_3…………x_n.

Note: Write equivalent definition from Adesh Kumar.

Consider an example; A TM accepting {0^n1^n / n ≥ 1}

- Given finite sequence of 0's and 1's on its tape preceded and followed by blanks.
- The TM will change 0 to an X and then a 1 to Y until all 0's and 1's are matches.

- Starting at left end of the input, it repeatedly changes a 0 to an X and moves to the right over whatever 0's and Y's it sees until comes to a 1.
- It changes 1 to a Y, and moves left, over Y's and 0's until it finds X. At that point, it looks for a 0 immediate to the right. If finds one 0 then changes it to X and repeats the process changing a matching 1 and Y.

Now, TM will have;

M = ({q_0, q_1, q_2, q_3, q_4}, {0, 1}, {0, 1, X, Y, B}, δ, q_0, B, {q_4})

The transition rule for the move of M is described by following transition table:

	0	1	X	Y	B
q0	(q1,X,R)			(q3, Y, R)	
q1	(q1,0,R)	(q2,Y,L)		(q1, Y, R)	
q2	(q2,0,L)		(q0, X, R)	(q2, Y, L)	
q3				(q3, Y, R)	
q4					(q4, B, R)

Now, the acceptance of 0011 by the TM, M1 is described by following sequence of moves;

q_0 0011 ├ X $q_1$011
├X0$q_1$11
├X$q_2$0Y1
├X$q_0$0Y1
├XXq_1Y1
├XXY$q_1$1
├XXq_2YY
├Xq_2XYY
├XXq_0YY
├XXYq_3Y
├XXYYq_3B
├XXYYBq_4B Halt and accept.

For string 0110 try yourself.

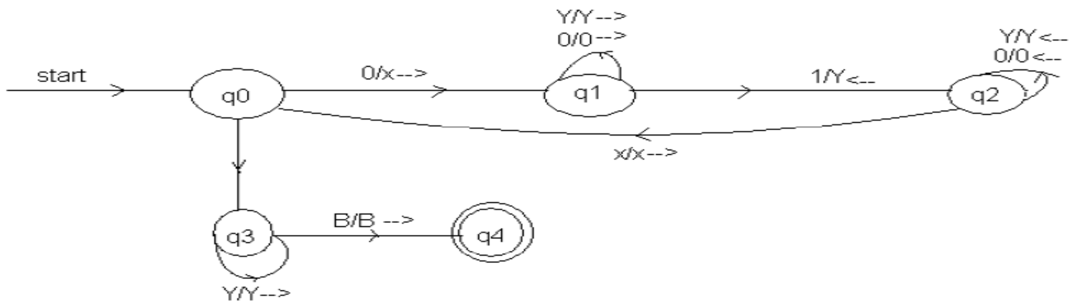
Transition diagram for a TM:

A transition diagram of TM consists of,

- A set of nodes representing states of TM.
- An arc from any state, q to p is labeled by the items of the form X / YD, where X and Y are tape symbols, and D is a direction, either L or R. that is, whenever δ(q, x) = (P, Y, D), we find the label X / YD on the arc from q to p.

However, in diagram, the direction D is represented by ← for left (L) and → for right (R)

Thus, transition diagram for the TM for L = $\{0^n1^n / n \geq 1\}$ as;



The language of Turing Machine:

If T= (Q, Σ, Γ, δ, q0, B, F) is a Turing machine and w ε Σ *, then language accepted by T, L (T) = {w | w ε Σ* and q0w $\vdash^*$ α p β} for some pεF and any tape string α and β.

The set of languages that can be accepted using TM are called recursively enumerable languages or RE languages.

The Turing Machine is designed to perform at least the following three roles;

1) As a language recognizer: TM can be used for accepting a language like Finite Automaton and Pushdown Automata.

2) As a Computer of function: A TM represents a particular function. Initial input is treated as representing an argument of the function. And the final string on the tape when the TM enters the halt state; treated as representative of the value obtained by an application of the function to the argument represented by the initial string.

3) As an enumerator of string of a language: It outputs the strings of a language, one at a time in some systematic order that is as a list.

Turing Machine for Computing a Function:

A Turing Machine can be used to compute functions. For such TM, we adopt the following policy to input any string to the TM which is an input of the computation function.

1) The string w is presented into the form BwB, where B is a blank symbol, and placed onto the tape; the head of TM is positioned at a blank symbol which immediately follows the string w.

2) We can show by underlining that symbol to the current position of machine head in the tape as (q, BwB) or, we can represent it by ID of TM as BwqB.

3) The TM is said to halt on input w if we can reach to halting state after performing some operation.

i.e. If TM = (Q, Σ, Γ, δ, q0, B, {qa}) is a turing machine . Then this TM is said to be halt on input w if and only if BwqB yields to BαqaB, for some αεΓ* i.e.(q,BwB) ├(qa, B α B)

Formal Definition:
A function f(x) = y is said to be computable by a TM and defined as; (Q, Σ, Γ, δ, q0, B, {q0})
If (q0, B x B) ├* (qa, ByB); where x may be in some Σ*, and y may be in some Σ2* and Σ1Σ2…………
It means that if we give input x to the Turing machine; it gives output as a string if it computes the function f(x) = y.
Example: Design a TM which computes the function f(x) = x+1 for each x belonging to set of natural numbers.
Given the function, f(x) = x+1. Here, we represent the input x on the tape by a number of 1's on the tape.
i.e. for x=1, input tape will have B1B,
for x=2 input tape will have B11B,
and so on…………..

Similarly, output can be seen by the number of 1'son the tape when machine halts.
Let us configure a TM as;
TM = (Q, Σ, Γ, δ, q0, B, {qf});
Where Q = {q0, qf}
Γ = {1, B}
Halt state = {q0}
Then δ can be simulated as;

	B	1
q0	(q0, 1, S)	(qf, 1, R)
q1		

So, let the input be x = 4. So, input tape at initial step consists of B1111B.
Then (q0, B1111B) ├ (q0, B11111) ├ (qf, B11111B).Which means output is 5 accepted.

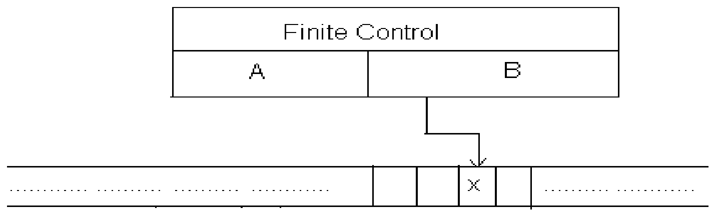
Turing Machines and Halting

There is another notion of acceptance that is commonly used for Turing machines: acceptance by Halting. We say a TM halts if it enters a state q, scanning a tape symbol X and there is no move in this situation; i.e δ(q,X) is undefined.

The Turing machine described above was not designed to accept any language rather we viewed it as computing an arithmetic function.
We always assume that a TM halts if it accepts. i.e. without changing language accepted, we can make δ(q,X) undefined whenever q is an accepting state.

Turing Machine with Storages in the state:

In Turing Machine, generally, any state represents the position in the computation. But the state can also be used to hold a finite amount of data. We can use the finite control not only to represent a position in the computation / program of the Turing Machine, but to hold a finite amount of data. In this case, a state is considered as a tuple; (state, data).
Following, it shows the model,



With this model of computation, δ is defined by:
δ([q, A], x) = ([q1, x], Y, R); means that q is the state and data portion associated with q is A. the symbol scanned on the tape is copied into the second component of the state and moves right entering state q1 and replacing tape symbol by Y.

Example: This model of TM can be used to recognize languages like 01* + 10*, where first symbol (0 or 1) that it sees, and checks that it does not appear else where in the input. For this, it remembers the first symbol in finite control.

Thus, the TM can be designed as;
M = (Q, {0, 1}, {0, 1, B}, δ, [q0, B], {[q1, B]})
The set of states Q, is {q0, q1} × {0, 1, B}. That is the states may be thought of a pair with two components;

a) A control portion, q0 or q1 that remembers the TM is doing. Control state q0 indicates that M has not yet read its first symbol, while q1 indicates that it has read the symbol, and is checking that it does not appear elsewhere, by moving right and hoping to reach a blank cell.

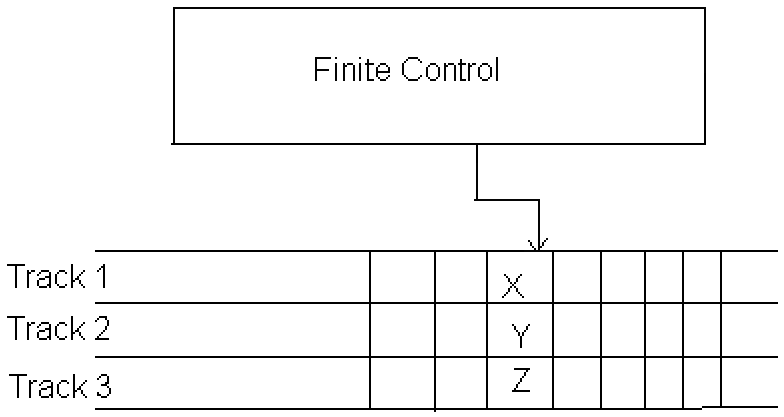
b) A data portion, which remembers the first symbol seen, which must be 0 or 1. The symbol B in this component means that no symbol has been read.

The transition function of M is defined as;
1) δ([q0, B], a) = ([q1, a], a, R); for a=0 or 1
2) δ([q1, a], a') = ([q1, a], a', R); where a' is the "complement" a, i.e. a'=0 if a=1 and a'=1 if a=0
3) δ([q1, a], B) = ([q1, B], B, R); for a=0 or 1. If M reaches the first blank, it enters the accepting state [q1, B].

Note: Since M has no definition for δ([q1, a], a); for a = 0 or 1, thus if M encounters a second occurrence of some symbol it stored initially in its control, it halts without having entered the accepting state. [eg: as 110 or 001 does not lie within the language so should be rejected.]

Turing Machine with multiple tracks:

The tape of TM can be considered as having multiple tracks. Each track can hold one symbol, and the tape alphabet of the TM consists of tuples, with one component for each track. Following figure illustrates the TM with multiple tracks;



The tape alphabet Γ is a set consisting of tuples like,
Γ = {(X, Y, Z), …………}
The tape head moves up and down scanning symbols in the tape at one position.

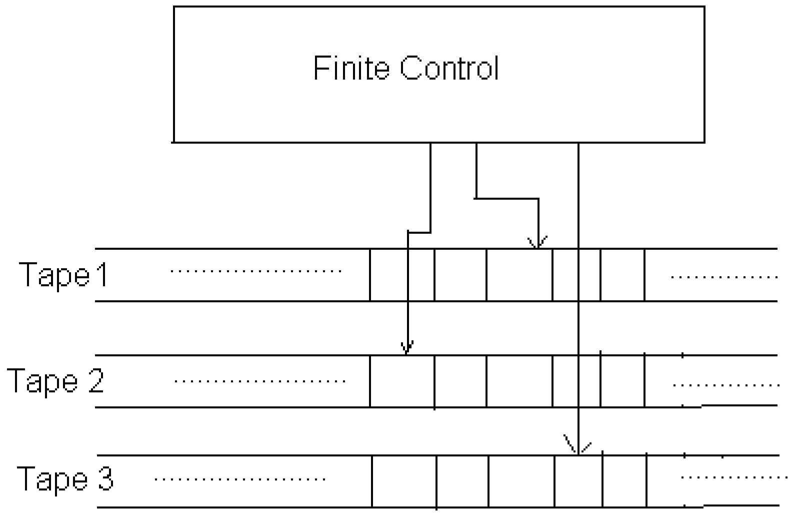
Like the technique of storage in the state, using multiple tracks does not extend what the TM can do. It is simply a way to view tape symbols and to imagine that they have a useful structure.

Sub-routines:
A complex TM can be thought as built from a collection of interacting components like general program. Such components of TM are called sub-routines. A TM subroutine is a set of states that performs some useful processes and can be called into another machine for the part of that computation.

Multi-tape Turing Machine:

Modern computing devices are based on the foundation of TM computation models. To simulate the real computers, a TM can be viewed as multi-tape machine in which there is more than one tape. However, adding extra tape adds no power to the computational model, only the ability to accept the language is concerned.

A multi-tape TM consists of finite control and finite number of tapes. Each tape is divided into cells and each cell can hold any symbol of finite tape alphabets. The set of tape symbols include a blank and the input symbols.



In the multi-tape TM, initially;

1. The Input (finite sequence of input symbols) w is placed on the first tape.
2. All other cells of the tapes hold blanks.
3. TM is in initial state q0.
4. The head of the first tape is at the left end of the input.
5. All other tape heads are at some arbitrary cell. Since all other tapes except first tape consists completely blank.

A move of multi- tape TM depends on the state and the symbol scanned by each of the tape head. In one move, the multi-tape TM does the following:

1. The control enters in a new state, which may be same previous state.
2. On each step, a new symbol is written on the cell scanned, these symbols may be same as the symbols previously there.
3. Each of the tape head make a move either left or right or remains stationary. Different head may move different direction independently i.e. if head of first tape moves leftward; at same time other head can move another direction or remains stationary.

The initial configuration (initial ID) of multi-tape TM with n-tapes is represented as;
(q0, ax, B, B, …………… B); n+1 tuples. Where w= ax is an input string and head first tape is scanning first symbol of w.
So, in general, it can be rewritten as;
$(q, x_1a_1y_1, x_2a_2y_2, \ldots\ldots x_na_ny_n)$
Where each x_i are the portion of string on tapes before the current head position, each a; are the symbol currently scanning in each tapes and each y_i are the portion of string on tapes just rightward to the current head position.
q is the control state.

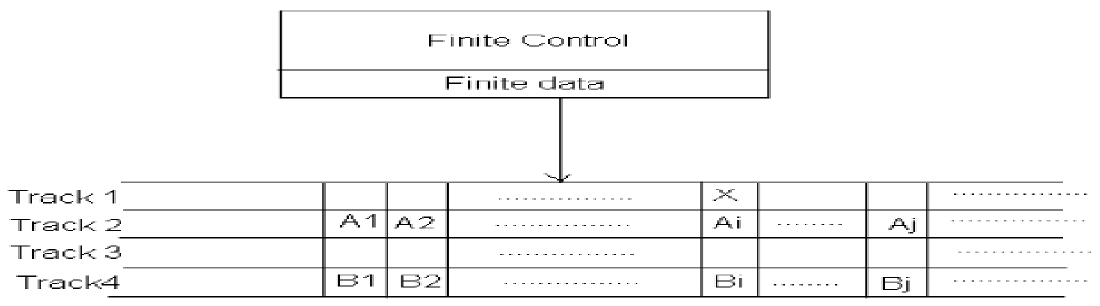
Equivalence of one-tape and Multi-tape TM

Any recursively enumerable languages that are accepted by one-tape TM are also accepted by multi-tape TM. i.e. Any n-tuples TM for $n \geq 2$, are at least as powerful as 1-tape TM's.

Simulating one tape TM with multi-tape TM:

Theorem: Every language accepted by a Multi-tape TM is recursively enumerable.
Or,
Any languages that are accepted by a multi-tape TM are also accepted by one tape Turing Machine.

Proof:
Let L is a language accepted by a n-tape TM, T1= (Q1, Σ, Γ1, δ1, q1, B, F1). Now, we have to simulate T1 with a one-tape TM, T2 = (Q2, Σ, Γ2, δ2, q2, B, F2) considering there are 2n tracks in the tape of T2. For simplicity, let us assume n = 2, then for n > 2 is the generalization of this case.
Then total number of tracks in T2 will be 4. The second and fourth tracks of T2 hold the contents of first and second tapes of T1. The track1 in T2 holds head position of tape 1 of T1 and track3 in T2 holds head position of tape2 of T1.



Now, to simulate a move of T1,
 T2's head must visit the n-head markers so that it must remember how many head markers are to its left at all times. That count is stored as a component of T2's finite control.
 After visiting each head marker and storing the scanned symbol in a component of its finite control, T2 knows what tape symbols are being scanned by each of T1's head.
 T2 also knows the state of T1, which it stores in T2's own finite control. Thus T2 knows what move T1 will make.

T2 now revisits each of the head markers on its tape, changes the symbol in track representing corresponding tapes T1 and moves the head marker left or right, if necessary.
Finally, T2 changes the state of T1 as recorded in its own finite control. Hence T2 has simulated one move of T1.
We select T2's accepting states, all those states that record T1's state as one of the accepting a state of T1. Hence, whatever T1 accepts T2 also accepts.

Non-Deterministic Turing Machine

A non-deterministic Turing Machine (NTM), T = (Q, Σ, Γ, δ, q0, B, F) is defined exactly the same as an ordinary TM, except the value of transition function δ. In NTM, the values of the transition function δ are subsets, rather than a single element of the set Q×Γ×{R,L,S}. Here, the transition function δ is such that for each state q and tape symbol x, δ(q, x) is a set of triples:

{(q1, Y1, D1), (q2, Y2, D2), …………………………(qk, Yk, Dk) where k is any finite integer.

The NTM can choose, at each step, any of the triples to be the next move. It cannot, however pick a state from one, a tape symbol from another, and the direction from yet another.

Church-Turing Thesis:

The Church-Turing Thesis states that in its most common form that "every computation or algorithm can be carried out by a Turing Machine."
This statement was first formulated by Alonzo Church. It is not a mathematically precise statement so un-provable. However, it is now generally assumed to be true.

The thesis might be replaced as saying that the notation of effective or mathematical method in logic and mathematics is captured by TM. It is generally assumed that such methods must satisfy the following requirements;
1. The method consists of a finite set of simple and precise instructions that are described with a finite number of symbols.
2. The method will always produce the result in a finite number of steps.
3. The method can in principle be carried out by a human being with only paper and pencil.
4. The execution of the method requires no intelligence of the human being except that which is needed to understand and execute the instructions.

The example of such a method is the "Euclidean algorithm" for determining the "Greatest Common Divisor" of two natural numbers.

The invention of TM has accumulated enough evidence to accept this hypothesis.
Following are the some of evidences:

1. In nature, a human normally works on 2D paper. The transfer of attention is not adjacent block like TM. However, transferring attention from one place to another during computation can be simulated by TM as one human step by multiple tapes.

2. Various extensions of TM model have been suggested to make computation efficient like doubly infinite tape, a tape with multiple tracks, multi-tape etc. In each case the computational power is reserved.

3. Other theoretical model have been suggested that are closer to modern computers in their operation (e.g. simple programming type languages, grammar and others)

4. Since the introduction of the TM, no one has suggested any type of computations that ought to be included in the category of "Algorithmic procedure."
Thus after adopting Church-Turing Thesis, we are giving a precise meaning of the term;
"An algorithm is a procedure that can be executed on a Turing Machine."

Universal Turing Machine

If a TM really is a sound model of computation, it should be possible to demonstrate that it can act as a stored program machine, where the program is regarded as an input, rather than hard-wired. We shall construct a Turing Machine Mu, that takes as input a description of a Turing Machine M and an input word x, and simulates the computation of M on input x. A machine such as Mu that can simulate the behavior of an arbitrary TM is called a Universal Turing Machine.

Thus, we can describe a Universal Turing Machine Tu as a TM, that on input <M, w>; where M is a TM and w is string of input alphabets, simulates the computation of M on input w.
specially,

- Tu accepts <M, w> iff M accepts w.
- Tu rejects <M, w> iff M rejects w.

Encoding of TM

For the representation of any arbitrary TM T_1, and an input string w over as arbitrary alphabet, as binary strings e (T_1), e (w) over some fixed alphabet, a notational system should be formulated.

Encoding the TM T1, and input w into e (T1) and e (w), it must not destroy any information. For encoding of TM, we use alphabet {0, 1}, although the TM may have a much larger alphabet.

To represent a TM T1 =(Q1,{0,1}, Γ, δ, q1,B,F) as binary string ,we first assign integers to the states, tape symbols and directions. We assume two fixed infinite sets Q = {q1, q2, q3…………} and S = {a1, a2, a3……………} so that Q1 is subset of Q and Γ is subset of S. now we have a subscript attached to every possible state and tape symbols, we can represent a state or a symbol by a string of 0's of the appropriate length. Here, 1's are used as separators.

Once we have established an integer to represent each state, symbol and direction, we can encode the transition function δ. Let one transition rule is

$\delta(q_i,a_j)=(q_k,a_l,D_m)$ for some integer I,j,k,l,m.
then we shall code this rule by the string $s(q_i)1s(a_j)1s(q_k)1s(a_j)1s(D_m)$.say this as m1. Where s is the encoding function defined below.

A code for entire TM T1 consist of all the codes for the transitions, in some order, separated by pairs of 1's:
$m_1 11 m_2 11 \ldots\ldots m_n$.

Now the code for TM and input string w will be formed by separating them by three consecutive ones i.e. 111.

The Encoding Function s
First, associate a string of 0's, to each state, to each of the three directions, and to each tape symbol. Let the function S is defined as

S(B) = 0
S(ai) =0^{i+1} for each ai ε S
S(qi) = 0^{i+2} for each qi ε Q
S(S) = 0
S(L) = 00
S(R) = 000

Consider an example, where, TM T is defined as;
T = ({q1, q2, q3}, {a, b} {a, b, B}, δ, q1, B, F),

where δ is defined as
δ(q1, b) = (q3, a, R) → m1
δ(q3, a) = (q1, b, R) → m2
δ(q3, b) = (q2, a, R) → m3
δ(q3, B) = (q3, b, L) → m4

Now, using the encoding function s defined above, as the rule, we have
S(q1) = 000
S(q2) = 0000
S(q3) = 00000
S(a1) = 00 considering a1 = a & a2 = b
S(a2) = 000
S(B) = 0
S(R) = 000
S(L) = 00
S(S) = 0

Now, encoding for rules
e (m1) = S(q1) 1 S(b)1 S(q3)1 S(a)1 S(R)
= 0001000100000100l000

e (m2) = S(q3) 1 S(a)1 S(q1)1 S(b)1 S(R)
= 00000100100010001000

e (m3) = S(q3) 1 S(b)1 S(q2)1 S(a)1 S(R)
= 000001000100001001000

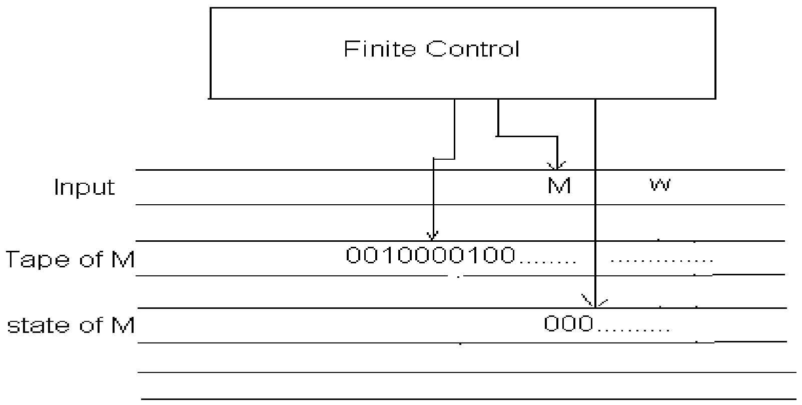
e (m4) = S(q3) 1 S(B)1 S(q3)1 S(b)1 S(L)
= 00000101000001000100

Now the code for TM T is
e(m1)11e(m2)11e(m3)11e(m4)=00010001000001001000
1100000100100010001000110000010001000010010001
1100000101000001000100

For this machine T, for any input w, where w= ab, the code will be e(T)111e(w)
Where e(w)= s(a)1s(b)=001000

Operation of Universal Turing Machine

Now, the Universal Turing Machine Tu can be described as a multi-tape Turing Machine in which the transitions of any other Turing Machine M are stored initially on the first tape, along with string w. The second tape holds the simulated tape of M, using the same format as for the code of M. The third tape holds the state of M, with suitable encoding. The sketch for Tu can be shown as;



The operation of universal Turing Machine Tu can be described as;

1. Examine the input to make sure that the code for M is valid code for some TM. If not, Tu halts without accepting. Any invalid codes represent TM with no moves.
2. Initialize the second tape to contain the input w in encoded form (simulated tape of M).
3. Place code of q1 (the start state of M) on the third tape and move head of Tu's second tape to the first simulated sell.
4. To simulate a move of M, Tu searches on its first tape for a transition $0^i10^j10^k10^l10^m$ such that o^i is the state on tape 3, o^j is the tape symbol of M that begins at the position on tape 2 scanned by Tu. This transition is the one move of M. Tu should;
   a) Change the content of tape 3 to 0^k, i.e. simulate state change of M.
   b) Replace 0^j on tape 2 by 0^l i.e. change the tape symbol of M. If more or less space is needed ($j \neq l$) use the scratch tape and shifting over technique as;
      - Copy onto a scratch tape, the entire non-blank tape to the right of where new value goes.
      - Write the new value using correct amount of space for that value.
      - Recopy the scratch onto tyape2, immediately to right of new value
   c) Move head on tape 2 to the position of next 1 to the left or right or stationary. If m = 1 (stationary), if m = 2 (move left) and m = 3 (move right).

   Thus, Tu simulates one move of M.
5. If M has no transition that matches the simulated state and tape symbol, then in 4 , no transition will be found. Thus Tu halts in the simulated configuration and Tu must do likewise.
6. If M enters its accepting state then Tu accepts.

Computational Complexity:

Complexity Theory is a central field of the theoretical foundations of Computer Science. It is concerned with the study of the *intrinsic complexity of computational tasks*. That is, a typical Complexity theoretic study looks at a task (or a class of tasks) and at the computational resources required to solve this task, rather than at a specific algorithm or algorithmic scheme.

The complexity of computational problems can be discussed by choosing a specific abstract machine as a model of computation and considering how much resource machine of that type require for the solution of that problem.

Complexity Measure is a means of measuring the resource used during a computation. In case of Turing Machines, during any computation, various resources will be used, such as space and time. When estimating these resources, we are always interested in growth rates rather than absolute values.

A problem is regarded as inherently difficult if solving the problem requires a large amount of resources, whatever the algorithm used for solving it. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as time and storage. Other complexity measures are also used, such as the amount of communication (used in communication complexity), the number of gates in a circuit (used in circuit complexity) and the number of processors (used in parallel computing). In particular, computational complexity theory determines the practical limits on what computers can and cannot do.

Time and Space Complexity of a Turing Machine:

The model of computation we have chosen is the Turing Machine. When a Turing machine answers a specific instance of a decision problem we can measure time as number of moves and the space as number of tape squares, required by the computation. The most obvious measure of the size of any instance is the length of input string. The worst case is considered as the maximum time or space that might be required by any string of that length.

The time and space complexity of a TM can be defined as;

Let T be a TM. The time complexity of T is the function Tt defined on the natural numbers as; for n ε N, Tt(n) is the maximum number of moves T can make on any input string of length n. If there is an input string x such that for |x|=n, T loops forever on input T, Tt(n) is undefined.

The space complexity of T is the function St defined as St (n) is the maximum number of the tape squares used by T for any input string of length n. If T is multi-tape TM, number of tape squares means maximum of the number of individual tapes. If for some input of length n, it causes T to loop forever, St (n) is undefined.

An algorithm for which the complexity measures St(n) increases with n, no more rapidly than a polynomial in n is said to be *polynomially bounded*; one in which it grows exponentially is said to be *exponentially bounded*.

Intractability:

Intractability is a technique for solving problems not to be solvable in polynomial time. The problems that cab be solved by any computational model, probably TM, using no more time then some slowly growing function size of the input are called “tractable:, i.e. those problems solvable within reasonable time and space constraints (polynomial time). The problems that cannot be solved in polynomial time but requires super polynomial (exponential) time algorithm are called intractable or hard problems. There are many problems for which no algorithm with running time better than exponential time is known some of them are, traveling salesman problem, Hamiltonian cycles, and circuit satisfiability, etc.

To introduce intractability theory, the class P and class NP of problems solvable in polynomial time by deterministic and non-deterministic TM’s are essential. A solvable problem is one that can be solved by particular algorithm i.e. there is an algorithm to solve this problem. But in practice, the algorithm may require a lot of space and time. When the space and time required for implementing the steps of the particular algorithm are (polynomial) reasonable, we can say that the problem is tractable. Problems are intractable if the time required for any of the algorithm is at least f(n), where f is an exponential function of n.

Complexity Classes:

In computational complexity theory, a **complexity class** is a set of problems of related resource-based complexity. A typical complexity class has a definition of the form:

“The set of problems that can be solved by an abstract machine M using O(f(*n*)) of resource R, where *n* is the size of the input.”

For example, the **class NP** is the set of decision problems that can be solved by a non-deterministic Turing machine in polynomial time, while the **class P** is the set of decision problems that can be solved by a deterministic Turing machine in polynomial space.

The simpler complexity classes are defined by the following factors:

- *The type of computational problem:* The most commonly used problems are decision problems. However, complexity classes can be defined based on function problems, optimization problems, etc.

- *The model of computation:* The most common model of computation is the deterministic Turing machine, but many complexity classes are based on nondeterministic Turing machines, Boolean circuits etc.

- *The resource (or resources) that are being bounded and the bounds:* These two properties are usually stated together, such as "polynomial time", "logarithmic space", "constant depth", etc.

The set of problems that can be solved using polynomial time algorithm is regarded as **class P**. The problems that are verifiable in polynomial time constitute the **class NP**. The class of **NP complete** problems consists of those problems that are NP as well as they are *as hard as* any problem in NP. The main concern of studying NP completeness is to understand how hard the problem is. So if we can find some problem as NP complete then we try to solve the problem using methods like approximation, rather than searching for the faster algorithm for solving the problem exactly.

Class P: The class P is the set pf problems that can be solved by deterministic TM in polynomial time. A language L is in class P if there is some polynomial time complexity T(n) such that L=L(M), for some Deterministic Turing Machine M of time complexity T(n).

Class NP: The class NP is the set of problems that can be solved by a non-deterministic TM in polynomial time. Formally, we can say a language L is in the class NP if there is a non-deterministic TM, M, and a polynomial time complexity T(n), such that L= L(M), and when M is given an input of length n, there are no sequences of more than T(n) moves of M.

Note: *Since every deterministic TM is a non-deterministic TM having no choice of more than one moves, so P is subset of NP. However NP contains many problems that are not in P. The class **P** consists of all those decision problems that can be solved on a Deterministic Turing Machine in an amount of time that is polynomial in the size of the input; the class **NP** consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a Non-deterministic Turing Machine.*

NP-Complete: In computational complexity theory, the complexity class **NP-complete** (abbreviated **NP-C** or **NPC**), is a class of problems having two properties:

→ It is in the set of NP (nondeterministic polynomial time) problems: Any given solution to the problem can be *verified* quickly (in polynomial time).

→It is also in the set of NP-hard problems: Any NP problem can be converted into this one by a transformation of the inputs in polynomial time.

Formally;

Let L be a language in NP, we say L is NP-Complete if the following statements are true about L;

→L is in class NP

→ For every language L1 in NP, there is a polynomial time reduction of L1 to L.

Once we have some NP-Complete problem, we can prove a new problem to be NP-Complete by reducing some known NP-Complete problem to it using polynomial time reduction.

It is not known whether every problem in NP can be quickly solved—this is called the P = NP problem. But if *any single problem* in NP-complete can be solved quickly, then *every problem in NP* can also be quickly solved, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every problem in NP-complete (that is, it can be reduced in polynomial time).

Because of this, it is often said that the NP-complete problems are *harder* or *more difficult* than NP problems in general.

Some of the properties of NP-complete problems are:

- No polynomial time algorithms has been found for any of them.

- It is no established that polynomial time algorithm for these problems do not exist.

- If polynomial time algorithm is found for any one of them, there will be polynomial time algorithm for all of them.

- If it can be proved that no polynomial time algorithm exists for any of them, then it will not exist for every one of them.

Problems:

Abstract Problems:

Abstract problem A is binary relation on set I of problem instances, and the set S of problem solutions. For e.g. Minimum spanning tree of a graph G can be viewed as a pair of the given graph G and MST graph T.

Decision Problems:

Decision problem D is a problem that has an answer as either "true", "yes", "1" or "false", "no", "0". For e.g. if we have the abstract shortest path with instances of the problem and the solution set as {0,1}, then we can transform that abstract problem by reformulating the problem as "Is there a path from u to v with at most k edges". In this situation the answer is either yes or no.

Optimization Problems:

We encounter many problems where there are many feasible solutions and our aim is to find the feasible solution with the best value. This kind of problem is called optimization problem. For e.g. given the graph G, and the vertices u and v find the shortest path from u to v with minimum number of edges. The NP completeness does not directly deal with optimizations problems; however we can translate the optimization problem to the decision problem.

Function Problems:

In computational complexity theory, a **function problem** is a computational problem where a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem, that is, it isn't just YES or NO. Notable examples include the *Traveling salesman problem*, which asks for the route taken by the salesman, and the *Integer factorization problem*, which asks for the list of factors. Function problems can be sorted into complexity classes in the same way as decision problems. For example FP is the set of function problems which can be solved by a deterministic

Turing machine in polynomial time, and FNP is the set of function problems which can be solved by a non-deterministic Turing machine in polynomial time. For all function problems in which the solution is polynomially bounded, there is an analogous decision problem such that the function problem can be solved by polynomial-time Turing reduction to that decision problem.

Encoding:

Encoding of a set S is a function e from S to the set of binary strings. With the help of encoding, we define **concrete problem** as a problem with problem instances as the set of binary strings i.e. if we encode the abstract problem, then the resulting encoded problem is concrete problem. So, encoding as a concrete problem assures that every encoded problem can be regarded as a language i.e. subset of {0,1}*.

Reducibility:

Reducibility is a way of converging one problem into another in such a way that, a solution to the second problem can be used to solve the fist one.

Many complexity classes are defined using the concept of a reduction. *A reduction is a transformation of one problem into another problem*. It captures the informal notion of a problem being at least as difficult as another problem. For instance, if a problem *X* can be solved using an algorithm for *Y*, *X* is no more difficult than *Y*, and we say that *X reduces* to *Y*. There are many different type of reductions, based on the method of reduction, such as Cook reductions, Karp reductions and Levin reductions, and the bound on the complexity of reductions, such as *polynomial-time reductions* or *log-space reductions*.

The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time. *For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers. This means an algorithm for multiplying two integers can be used to square an integer*. Indeed, this can be done by giving the same input to both inputs of the multiplication algorithm. Thus we see that squaring is not more difficult than multiplication, since squaring can be reduced to multiplication.

This motivates the concept of a problem being hard for a complexity class. A problem *X* is *hard* for a class of problems *C* if every problem in *C* can be reduced to *X*. Thus no problem in *C* is harder than *X*, since an algorithm for *X* allows us to solve any problem in *C*. Of course, the notion of hard problems depends on the type of reduction being used. For complexity classes larger than P, polynomial-time reductions are commonly used. In particular, the set of problems that are hard for NP is the set of NP-hard problems.

Circuit Satisfaibility:

Cook's Theorem
Lemma: SAT is NP-hard

Proof: (This is not actual proof as given by cook, this is just a sketch)

Take a problem V ε NP, let A be the algorithm that verifies V in polynomial time (this must be true since V ε NP). We can program A on a computer and therefore there exists a (huge) logical circuit whose input wires correspond to bits of the inputs x and y of A and which outputs 1 precisely when A(x, y) returns yes.
For any instance x of V let A_x be the circuit obtained from A by setting the x-input wire values according to the specific string x. The construction of A_x from x is our reduction function. If x is a yes instance of V, then the certificate y for x gives satisfying assignments for A_x. Conversely, if A_x outputs 1 for some assignments to its input wires, that assignment translates into a certificate for x.

Theorem: SAT is NP-complete

Proof:

To show that SAT is NP-complete we have to show two properties as given by the definition of NP-complete problems. The first property i.e. SAT is in NP

Circuit satisfiability problem (SAT) is the question "Given a Boolean combinational circuit, is it satisfiable? i.e. does the circuit has assignment sequence of truth values that produces the output of the circuit as 1?" Given the circuit satisfiability problem take a circuit x and a certificate y with the set of values that produce output 1, we can verify that whether the given certificate satisfies the circuit in polynomial time. So we can say that circuit satisfiability problem is NP.

This claims that SAT is NP. Now it is sufficient to show the second property holds for SAT. The proof for the second property i.e. SAT is NP-hard is from above lemma. This completes the proof.

Note:

(Refer Chapter "NP-Completeness" of the Book "Introduction to Algorithms": By Cormen, Leiserson, Rivest and Stein. It's a reference book of DA!!!!)

(From Page 966 to…., In 2nd Ed.) (You will find detail concepts of SAT, BSAT)

Undecidability:

In computability theory, an undecidable problem is a decision problem for which it is impossible to construct a single algorithm that always leads to a correct "yes" or "no" answer- the problem is not decidable. An **undecidable problem** consists of a family of instances for which a particular yes/no answer is required, such that there is no computer program that, given any problem instance as input, terminates and outputs the required answer after a finite number of steps. More formally, an undecidable problem is a problem whose language is not a recursive set or computable or decidable.

In computability theory, **the halting problem** is a decision problem which can be stated as follows:

Given a description of a program and a finite input, decide whether the program finishes running or will run forever.

Alan Turing proved in 1936 that a general algorithm running on a Turing machine that solves the halting problem for *all* possible program-input pairs necessarily cannot exist. Hence, the halting problem is *undecidable* for Turing machines.

Post's Correspondence Problem: (PCP)

The Post's Correspondence Problem is an undecidable decision problem that was Introduced by Emil Post in 1946.

Definition: The input of the problem consists of two finite lists U= {u1, u2, ….., un} and V= {v1, v2,……., vn} of words over some alphabet Σ having at lest two symbols. A solution to this problem is a sequence of indices ik; 1<=k<= n, for all k, such that

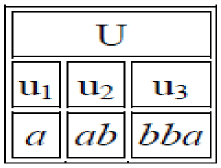
u_{i1} u_{i2} ………….. $u_{ik} = v_{i1}$ v_{i2} ………….. v_{ik}

We say i1, i2,…….ik is a solution to this instance of PCP.

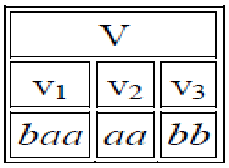
Here, the decision problem is to decide whether such a solution exits or not.

Examples:

Consider the following two lists:



U		
u_1	u_2	u_3
a	*ab*	*bba*



V		
v_1	v_2	v_3
baa	*aa*	*bb*

A solution to this problem would be the sequence (3, 2, 3, 1), because

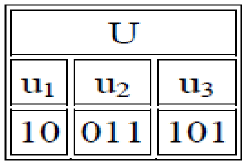
u3u2u3u1 = *bba* + *ab* + *bba* + *a* = *bbaabbbaa*

v3v2v3v1 = *bb* + *aa* + *bb* + *baa* = *bbaabbbaa*

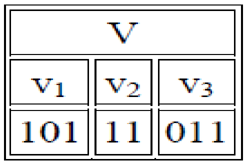
Furthermore, since (3, 2, 3, 1) is a solution, so are all of its "repetitions", such as (3, 2, 3, 1, 3, 2, 3, 1), etc.; that is, when a solution exists, there are infinitely many solutions of this repetitive kind.

However, if the two lists had consisted of only u2,u3 and v2,v3, then there would have been no solution (because then no matching pair would have the same last letter, as must occur at the end of a solution).

Consider another example with two lists as below



U		
u_1	u_2	u_3
10	011	101



V		
v_1	v_2	v_3
101	11	011

For this instance of PCP, there is no solution !!!

Halting Problem:

"Given a Turing Machine M and an input w, do M halts on w?"

Algorithms may contain loops which may be infinite or finite in length. The amount of work done in an algorithm usually depends on data input. Algorithms may consists of various numbers of loops nested or in sequence. Thus, the halting problem asks the question; *"Given a program and an input to the program, determine if the program will eventually stop when it is given that input."* The question is simply whether the given program will ever halt on a particular input.

Trial Solution: Just run the program with the given input. If the program stops we know the program halts. But if the program does not stop in reasonable amount of time, we can not conclude that it won't stop. May be we did not wait long enough!

For example, in pseudocode, the program

while True: continue

does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program

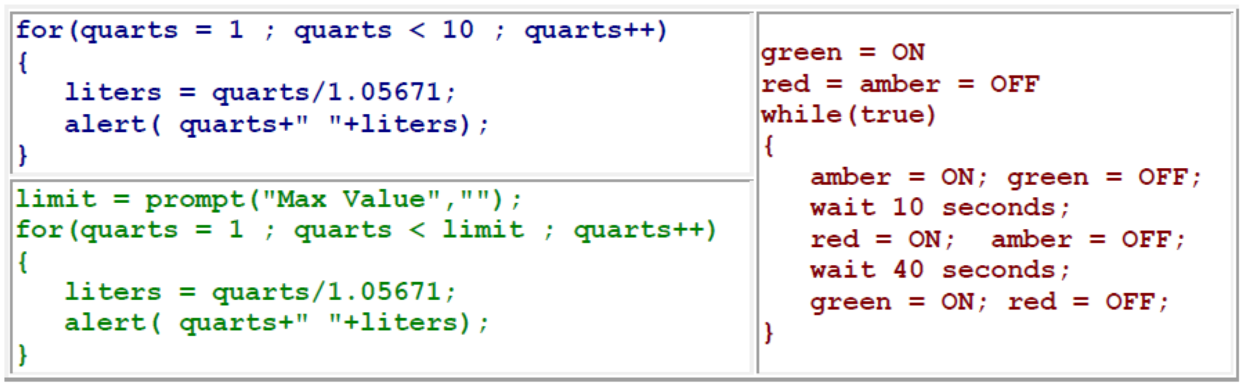
print "Hello World!"

halts very quickly.

The halting problem is famous because it was one of the first problems proven algorithmically undecidable. This means there is no algorithm which can be applied to any arbitrary program and input to decide whether the program stops when run with that input.

The Halting Problem is one of the simplest problems know to be unsolvable. You will note that the interesting problems involve loops.

Consider the following Javascript program segments (algorithm):



```
for(quarts = 1 ; quarts < 10 ; quarts++)
{
   liters = quarts/1.05671;
   alert( quarts+" "+liters);
}
```

```
limit = prompt("Max Value","");
for(quarts = 1 ; quarts < limit ; quarts++)
{
   liters = quarts/1.05671;
   alert( quarts+" "+liters);
}
```

```
green = ON
red = amber = OFF
while(true)
{
   amber = ON; green = OFF;
   wait 10 seconds;
   red = ON;  amber = OFF;
   wait 40 seconds;
   green = ON; red = OFF;
}
```

The first program clearly is a one that will terminate after printing in an alert 10 lines of output. The second program alerts as many times as indicated by the input. The last program runs forever.

Sketch of a proof that the Halting Problem is undecidable:

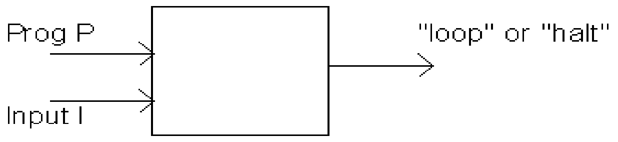
This proof was devised by Alan Turing in 1936.

Suppose we have a solution to the halting problem called H. H takes two inputs:

1. a program P and

2. an input I for the program P.

H generates an output "***halt***" if H determines that P stops on input I or it outputs "***loop***" otherwise.



Note: When an algorithm is coded, it is expressed as a string of characters which can also be interpreted as a sequence of numbers (binary). We can treat the program as data and therefore a program can be thought of as input.

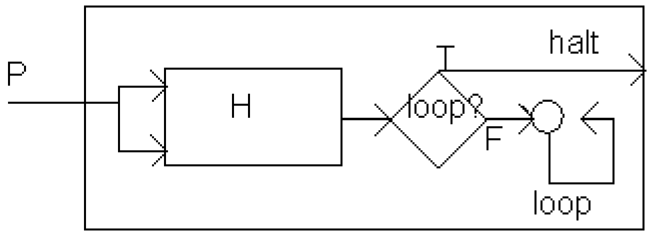
For example, compilers take programs as input and generate machine code as output. Netscape takes a Javascript program and generates output.

So now H can be revised to take P as both inputs (the program and its input) and H should be able to determine if P will halt on P as its input.

Let us construct a new, simple algorithm K that takes H's output as its input and does the following

1. If H outputs "***loop***" then K halts,

2. Otherwise H's output of "***halt***" causes K to loop forever.

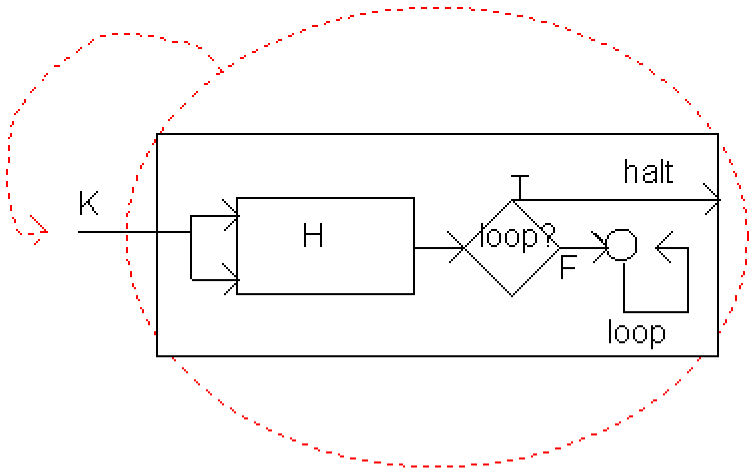


That is, K will do the **opposite** of H's output.

function K() {

```
if (H()=="loop"){
return;
} else {
while(true); //loop forever
}
}
```

Since K is a program, let us use K as the input to K.



If H says that K halts then K itself would loop (that's how we constructed it). If H says that K loops then K will halt.

In either case H gives the wrong answer for K. **Thus H cannot work in all cases.**

We've shown that it is possible to construct an input that causes any solution H to fail.

Hence Proved!