
BSC-CSIT
Complete notes
with
past question
solution
TOC PREPRATION
KIT
OPEN SOURCE!!!
TECHNOVATIVE
an IT Hub

Contents

Theory of computation

INTRODUCTION

Set

The term set is the collection of well-defined objects, which are called the elements of the sets. Georg Cantor (1845-1915) developed set theory. Sets are used throughout the theory of computation.

Some examples of sets are given below:

- The set of vowels in English alphabet
- The set of natural numbers less than 12.
- The set of odd numbers less than 20.

There are three methods to describe a set:

a. Enumeration method-listing
b. Standard method or description method
c. Set builder method

a. *Enumeration method:* When the elements of a set are enumerated or listed, we enclose them in braces.
e.g. A={1, 2, 3, ……, 100}

b. *Standard method*: Frequently used sets are usually given symbols that are reserved for them only.
e.g. N (Natural Numbers)= {1, 2, 3, ……., n}

c. *Set builder method*: Another way of representing set is to use set builder method.
e.g. we could define the rational numbers as
Q={x/y: x, yϵZ; y≠0}

Types of sets

a. *Empty set*: A set, which has no element, is called as empty set or null set or void set. It is denoted by Φ
b. *Singleton set*: A set, which has single element, is called as singleton set.
c. *Disjoint sets:* Two or more sets are said to be disjoints, if there are no common elements among.
d. *Overlapping sets*: Two or more sets are said to be disjoints, if there are at least one common element among them.
e. *Finite sets*: A set having specified number of elements is called as a finite set.
f. *Infinite sets*: A set is called infinite set, if it is not finite set.
g. *Universal set:* The set of all objects or things under consideration in discussion is called the universal set.

Relation

A relation is a correspondence between two sets (called the *domain* and the *range*) such that to each element of the *domain*, there is assigned one or more elements of the *range*.

State the domain and range of the following relation. Is the relation a function? {(2, –3), (4, 6), (3, –1), (6, 6), (2, 3)}

The above list of points, being a relationship between certain x's and certain y's, is a relation. The domain is all the x-values, and the range is all the y-values. To give the domain and the range, I just list the values without duplication: domain: {2, 3, 4, 6} range: {–3, –1, 3, 6}

Types of relation

Identity relation: In an identity relation "R", every element of the set "A" is related to itself only. Note the conditions conveyed through words "every" and "only". The word "every" conveys that identity relation consists of ordered pairs of element with itself - all of them. The word "only" conveys that this relation does not consist of any other combination.
Consider a set
A={1,2,3} Then, its identity relation is: R={(1,1), (2,2),(3,3)}

Reflexive relation

In reflexive relation, "R", every element of the set "A" is related to itself. The definition of reflexive relation is exactly same as that of identity relation except that it misses the word "only" in the end of the sentence. The implication is that this relation includes identity relation and permits other combination of paired elements as well.
Consider a set
A={1,2,3} Then, one of the possible reflexive relations can be:
R={(1,1), (2,2), (3,3), (1,2), (1,3)}
However, following is not a reflexive relation: R1={(1,1), (2,2), (1,2), (1,3)}

Symmetric relation

In symmetric relation, the instance of relation has a mirror image. It means that if (1,3) is an instance, then (3,1) is also an instance in the relation. Clearly, an ordered pair of element with itself like (1,1) or (2,2) is themselves their mirror images. Consider some of the examples of the symmetric relation,
R1={(1,2),(2,1),(1,3),(3,1)}
R2={(1,2),(1,3),(2,1),(3,1),(3,3)}
We have purposely jumbled up ordered pairs to emphasize that order of elements in relation is not important. In order to decide symmetry of a relation, we need to identify mirror pairs. We state the condition of symmetric relation as: Iff$(x,y)\in R \Rightarrow (y,x)\in R$ for all $x,y\in A$
The symbol "Iff" means "If and only if". Here one directional arrow means "implies". *Alternatively, the condition of symmetric relation can be stated as: $xRy \Rightarrow yRx$ for all $x,y\in A$*

Transitive relation

If "R" be the relation on set A, then we state the condition of transitive relation as: Iff$(x,y)\in R$ and $(y,z)\in R \Rightarrow (x,z)\in R$ for all $a,b,c\in A$
Alternatively, xRy and $yRz \Rightarrow xRz$ for all $x,y,z\in A$

Equivalence relation

A relation is equivalence relation if it is reflexive, symmetric and transitive at the same time. In order to check whether a relation is equivalent or not, we need to check all three characterizations.

Function

A function is a correspondence between two sets (called the domain and the range) such that to each element of the domain, there is assigned exactly one element of the range.

Determine the domain and range of the given function: $y = -\sqrt{-2x+3}$

The domain is all values that x can take on. The only problem I have with this function is that I cannot have a negative inside the square root. So I'll set the insides greater-than-or-equal-to zero, and solve. The result will be my domain:

$-2x + 3 > 0 \quad -2x > -3 \quad 2x < 3 \quad x < 3/2 = 1.5$

Then the domain is "all $x < 3/2$".

Alphabets

The symbols are generally letters and digits. *Alphabets are defined as a finite set of symbols*. It is denoted by '∑' symbol.

E.g.: An alphabet of set of decimal numbers is given by ∑= {0, 1,……., 9).

The alphabet for binary number is ∑={0, 1}.

Strings

A string or word is a finite sequence of symbols selected from some alphabets. E.g. if ∑={a, b} then '*abab'* is a string over ∑. A string is generally denoted by 'w'. The empty string is the string with 0 (zero) occurrence of symbols. This string is represented by ε or e or ^

Closure of an alphabet

Closure of an alphabet is defined as the set of all strings over an alphabet ∑ including empty string and is denoted by ∑*

e.g.

Let ∑ = {0, 1} then

$\sum^* = \{\varepsilon, 0, 1, 00, 10, 01, 11, \ldots.\}$

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

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

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

$\sum^+ = \sum^1 U \sum^2 U \sum^3 U \ldots.$

Therefore, $\sum^* = \sum^+ U \{\varepsilon\}$

Concatenating of string

Let w_1 and w_2 be two strings, then w_1w_2 denotes the concatenation of w_1 and w_2.

e.g.

if w_1 = abc, w_2 = xyz, then

w_1w_2 = abcxyz

Languages

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

Let ∑ = {0, 1} then,
L = {all strings over ∑ with equal number of 0's and 1's}
= {01, 0011, ……}
α ⊆ ∑*

Exercises

1. What is set? What are its types?
2. How can we represent set?
3. What is relation? Give examples.
4. Explain the types of relations.
5. What do you mean function?
6. Define the following terms with examples:
    a. String
    b. Alphabet
    c. Symbol
    d. Keene closure
    e. Function
    f. Union
    g. Concatenation

Methods of proof

Theorem

A **theorem** is a mathematical proposition that is true. Many theorems are conditional propositions. For example, if f(x) and g(x) are continuous then f(x)± g(x) are also continuous.
If theorem is of the form "if p then q", the p is called **hypothesis** and q is called **conclusion**.

Proof

A proof of a theorem is a logical argument that establishes the theorem to be true. There are different types of proofs of a theorem. Some of them are given below:

- Trivial proof
- Vacuous proof
- Direct proof
- Indirect proof
- Proof by contradiction
- Proof by cases
- Proof by mathematical induction
- Proof by counter examples

Trivial proof

We say is trivially true if q is true, and this kind of proof (i.e. showing q is true for without referring to p) is called a trivial proof.
Consider an implication: p→q
If it can be shown that q is true, then the implication is always true by definition of an implication.

Vacuous proofs
Consider an implication: p→q
If it can be shown that p is false, then the implication is always true by definition of an implication.
Note that you are showing that the antecedent is false

Direct Proofs
To prove p→q, we start assuming hypothesis p is true and we use information already available to prove q is true, and if q is true then the argument is valid. This is called direct proof.

E.g. If a and b are odd integers, then a+b is an odd integers.
Here a and b are odd integers. Since every odd numbers can be written by 2l+1 where l is any integer.
So, a = 2m+1
b = 2n+1 for some integers m and n
Now, a+b = 2m+1+2n+1 = 2m+2n+2 = 2(m+n+1) = 2*k where k = m+n+1 is any integer. This shows a+b is even.

Indirect proof (proof by contraposition)
Since, p→q is equivalent to ¬q→¬p. To prove p→q, we assume the conclusion is false; using the fact if p becomes false, original implication is true.

e.g. if the product of two integers a and b is even, then either a is even or b is even.
Suppose, if possible both a and b are odd integers. So, a = 2m+1 and b = 2n+1.
And axb = (2m+1)(2n+1) = 4mn+2m+2n+1 = 2(2mn+m+n)+1 = 2l+1 where l = 2mn+m+n, which is not true. So, our original implication is true.

Proof by contradiction
The following proof proceeds by contradiction. That is, we will assume that the claim we are trying to prove is wrong and reach a contradiction. If all the derivations along the way are correct, then the only thing that can be wrong is the assumption, which was that the claim we are trying to prove does not hold. This proves that the claim does hold.

Eg: For all integers n, if n2 is odd, then n is odd.
Suppose not. [We take the negation of the given statement and suppose it to be true.] Assume, to the contrary, that ∃ an integer n such that n2 is odd and n is even. [We must deduce the contradiction.] By definition of even, we have

n = 2k for some integer k.

So, by substitution we have

n . n = (2k) . (2k)

$$= 2\ (2.k.k)$$

Now (2.k.k) is an integer because products of integers are integer; and 2 and k are integers. Hence,

$$n \cdot n = 2 \cdot (\text{some integer})$$

or $$n2 = 2.\ (\text{some integer})$$

and so by definition of n2 even, is even.

So the conclusion is since n is even, n2, which is the product of n with itself, is also even. This contradicts the supposition that n2 is odd. [Hence, the supposition is false and the proposition is true.]

Proof by cases

You can sometimes prove a statement by:

1. Dividing the situation into cases which exhaust all the possibilities; and
2. Showing that the statement follows in all cases.

It's important to cover all the possibilities. And don't confuse this with trying examples; an example is not a proof.

Theorem. If n is a positive integer then n7 - n is divisible by 7.

Proof:

First we factor n7 - n = n(n6 - 1) = n(n3 - 1)(n3 + 1) = n(n-1)(n2 + n + 1)(n+1)(n2 - n + 1). Now there are 7 cases to consider, depending on n = 7 q + r where r = 0, 1, 2, 3, 4, 5, 6, 7.

Case 1: n = 7q. Then n7 - n has the factor n, which is divisible by 7.

Case 2: n = 7q + 1. Then n7 - n has the factor n-1 = 7q.

Case 3: n = 7q + 2. Then the factor n2 + n + 1 = (7q + 2)2 + (7q+2) + 1 = 49 q2 + 35 q + 7 is clearly divisible by 7.

Case 4: n = 7q + 3. Then the factor n2 - n + 1 = (7q + 3)2 - (7q+3) + 1 = 49 q2 + 35 q + 7 is clearly divisible by 7.

Case 5: n = 7q + 4. Then the factor n2 + n + 1 = (7q + 4)2 + (7q+4) + 1 = 49 q2 + 63 q + 21 is clearly divisible by 7.

Case 6: n = 7q + 5. Then the factor n2 - n + 1 = (7q + 5)2 - (7q+5) + 1 = 49 q2 + 63 q + 21 is clearly divisible by 7.

Case 7: n = 7q + 6. Then the factor n + 1 = 7q +7 is clearly divisible by 7.

Poof by mathematical induction

Mathematical induction is a powerful, yet straightforward method of proving statements whose "domain" is a subset of the set of integers. Usually, a statement that is proven by induction is based on the set of natural numbers. This statement can often be thought of as a function of a number n, where n = 1,2,3...

Proof by induction involves three main steps: proving the base of induction, forming the induction hypothesis, and finally proving that the induction hypothesis holds true for all numbers in the domain.

Proving the base of induction involves showing that the claim holds true for some base value (usually 0, 1, or 2). There are sometimes many ways to do this, and it can require some ingenuity. We will outline this with a simple example.

Theorem. A formula for the sequence an defined above, is an = (1 - 1/22n)/2 for all n greater than or equal to 0.

Proof. (By Mathematical Induction.)

Initial Step. When n = 0, the formula gives us (1 - 1/22n)/2 = (1 - 1/2)/2 = 1/4 = a0. So the closed form formula ives us the correct answer when n = 0.

Inductive Step. Our inductive assumption is: Assume there is a k, greater than or equal to zero, such that ak = (1 - 1/22k)/2. We must prove the formula is true for n = k+1.

First we appeal to the recurrsive definition of ak+1 = 2 ak(1-ak). Next, we invoke the inductive assumption, for this k, to get

ak+1 = 2 (1 - 1/22k)/2 (1 - (1 - 1/22k)/2) = (1 - 1/22k)(1 + 1/22k)/2 = (1 - 1/22k+1)/2. This completes the inductive step.

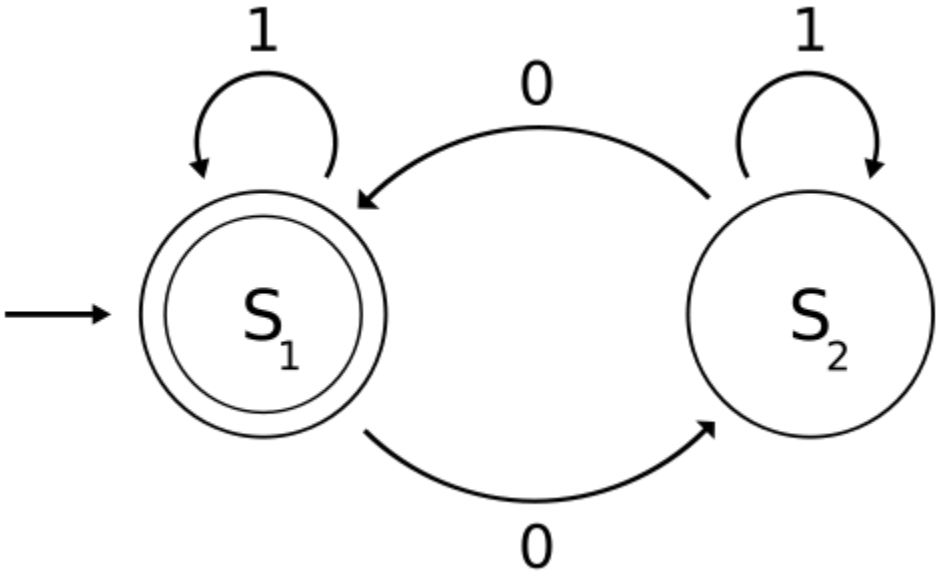
Proof by Counterexample

Consider a statement of the form
$\forall x \in M$, if P(x) then Q(x).
Suppose that we wish to prove that this statement is false. In order to disprove this statement, we have to find a value of x in M for which P(x) is true and Q(x) is false. Such an x is called a **counterexample**.
Furthermore, proving that this statement is false is equivalent to showing that its negation is true. The negation of the above statement is
$\exists x$ in M such that P(x) and not Q(x).
$\exists x \ M \mid P(x) \wedge \sim Q(x)$.
Finding an x that makes the above statement true will disprove the original statement.

Automata Theory

→The study of the mathematical properties of abstract machine or automata is automata theory.

→In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems, as they are described in mathematical terms) and the computational problems that can be solved using these machines. These abstract machines are called automata. Automata come from the Greek word, which means "self-acting".



→The figure above illustrates a finite state machine, which belongs to one well-known variety of automatons. This automaton consists of states (represented in the figure by circles), and transitions (represented by arrows). As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its *transition function* (which takes the current state and the recent symbol as its inputs).

→Automata theory is also closely related to formal language theory. An automaton is a finite representation of a formal language that may be an infinite set. Automata are often classified by the class of formal languages they are able to recognize.

→Automata play a major role in theory of computation, compiler design, parsing and formal verification.

Finite-state machine

A finite-state machine (FSM) or finite-state automaton (plural: *automata*), or simply a state machine, is a mathematical model used to design computer programs and digital logic circuits. It is conceived as an abstract machine that can be in one of a finite number of *states*. The machine is in only one state at a time; the state it is in at any given time is called the *current state*. It can change from one state to another when initiated by a triggering event or condition, this is called a *transition*. A particular FSM is defined by a list of the possible transition states from each current state, and the triggering condition for each transition. Finite-state machines can model a large number of problems, among which are electronic design automation, communication protocol design, parsing and other engineering applications. In biology and artificial intelligence research, state machines or hierarchies of state machines are sometimes used to describe neurological systems and in linguistics—to describe the grammars of natural languages.

Finite Automata

An automaton with a set of states, and its "control" moves from state to state in response to external "inputs" is called a finite automaton.

A finite automaton, FA, provides the simplest model of a computing device. It has a central processor of finite capacity and it is based on the concept of state. It can also be given a formal mathematical definition. Finite automata are used for pattern matching in text editors, for compiler lexical analysis.

Another useful notion is the notion of nondeterministic automaton.

We can prove that deterministic finite automata, DFA, recognize the same class of languages as NDFA, ie. They are equivalent formalisms.

It is also possible to prove that given a language L there exists a unique (up to isomorphism) minimum finite state automaton that accepts it, i.e. an automaton with a minimum set of states.

The automata in the examples are deterministic, that is, once their state and input are given, their evolution is uniquely determined.

Formal definition

An automaton is represented formally by a 5-tuple (Q,Σ,δ,q_0,F), where:

- Q is a finite set of *states*.
- Σ is a finite set of *symbols*, called the *alphabet* of the automaton.
- δ is the transition function, that is, $\delta: Q \times \Sigma \rightarrow Q$.
- q_0 is the *start state*, that is, the state of the automaton before any input has been processed, where $q_0 \in Q$.
- F is a set of states of Q (i.e. $F \subseteq Q$) called accept states.

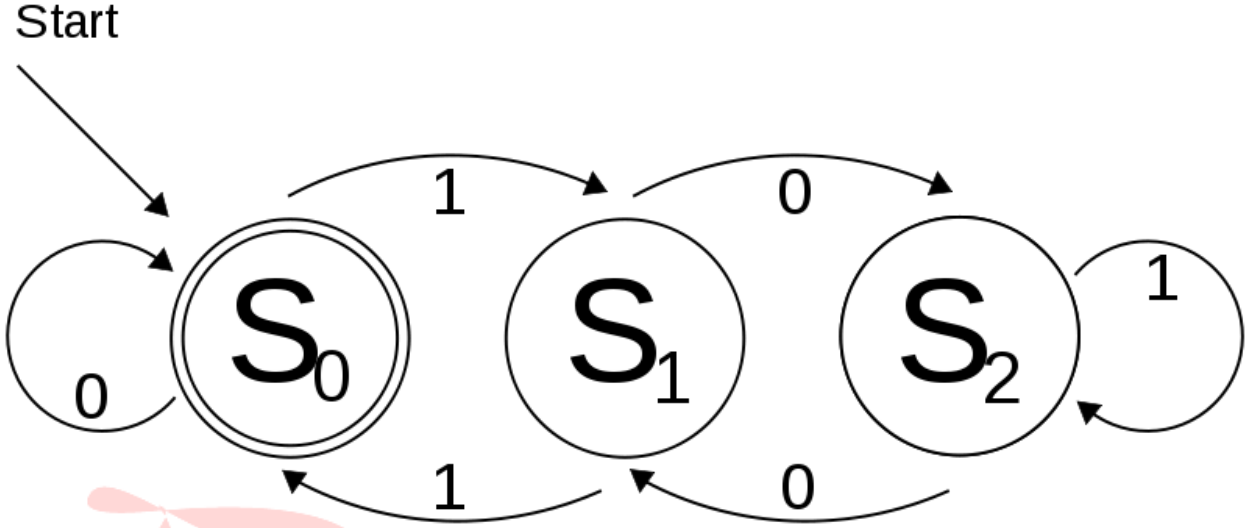
Applications

Each model in automata theory plays an important roles in several applied areas.

- Finite automata are used in text processing, compilers, and hardware design.
- Context-free grammar (CFGs) are used in programming languages and artificial intelligence. Originally, CFGs were used in the study of the human languages.
- Cellular automata are used in the field of biology, the most common example being John Conway's Game of Life.
- Some other examples which could be explained using automata theory in biology include mollusk and pine cones growth and pigmentation patterns.
- Going further, a theory suggesting that the whole universe is computed by some sort of a discrete automaton, is advocated by some scientists.

Deterministic finite automata(DFA)

In the automata theory, a branch of theoretical computer science, a deterministic finite automaton (DFA)—also known as deterministic finite state machine.
It is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string.



The figure at above illustrates a deterministic finite automaton. In the automaton, there are three states: S0, S1, and S2 (denoted graphically by circles). The automaton takes finite sequence of 0s and 1s as input. For each state, there is a transition arrow leading out to a next state for both 0 and 1. Upon reading a symbol, a DFA jumps *deterministically* from a state to another by following the transition arrow. For example, if the automaton is currently in state S0 and current input symbol is 1 then it deterministically jumps to state S1. A DFA has a *start state* (denoted graphically by an arrow coming in from nowhere) where computations begin, and a set of *accept states* (denoted graphically by a double circle) which help define when a computation is successful.

A DFA is defined as an abstract mathematical concept, but due to the deterministic nature of a DFA, it is implementable in hardware and software for solving various specific problems.
DFAs can be built from nondeterministic finite automata through the power set construction.

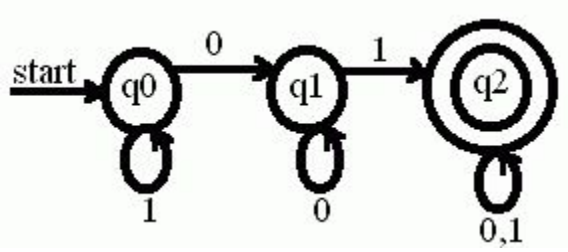
Formal definition

A deterministic finite automaton M is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, consisting of

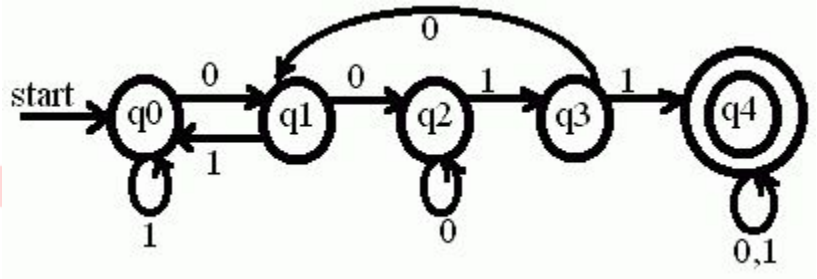
- a finite set of states (Q)
- a finite set of input symbols called the alphabet (Σ)
- a transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- a start state ($q_0 \in Q$)
- a set of accept states ($F \subseteq Q$)

Examples & Exercises of DFA

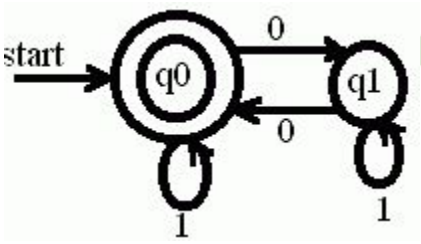
1. Construct a DFA to accept a string containing a zero followed by a one.



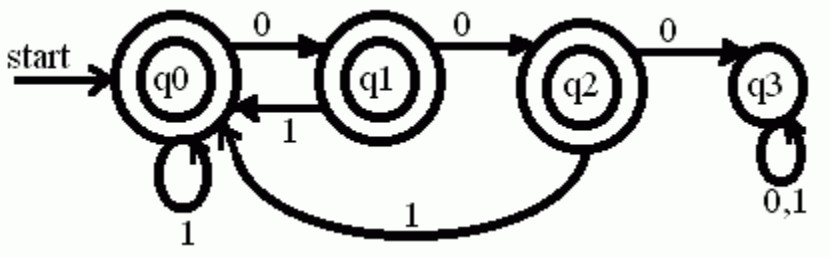
2. Construct a DFA to accept a string containing two consecutive zeroes followed by two consecutive ones .



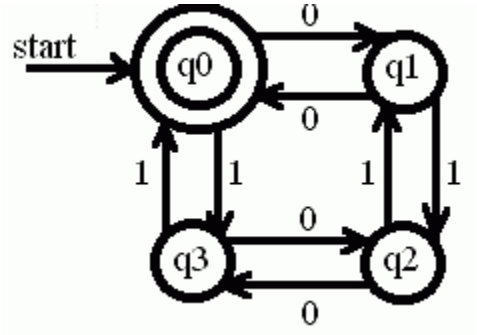
3. Construct a DFA to accept a string containing even number of zeroes and any number of ones.



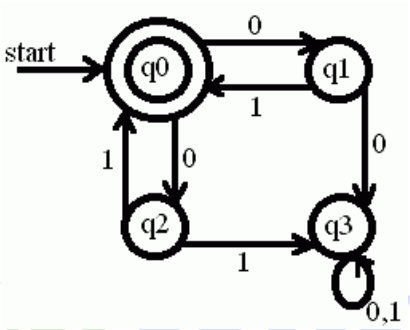
4. Construct a DFA to accept all strings which do not contain three consecutive zeroes.



5. Construct a DFA to accept all strings containing even number of zeroes and even number of ones.



6. Construct a DFA to accept all strings (0+1)* with an equal number of 0's & 1's such that each prefix has at most one more zero than ones and at most one more one than zeroes



Nondeterministic finite automata

In the automata theory, a nondeterministic finite automaton (NFA) or nondeterministic finite state machine is a finite state machine where from each state and a given input symbol the automaton may jump into several possible next states. This distinguishes it from the deterministic finite automaton (DFA), where the next possible state is uniquely determined. Although the DFA and NFA have distinct definitions, a NFA can be translated to equivalent DFA using power set construction, i.e., the constructed DFA and the NFA recognize the same formal language. Both types of automata recognize only regular languages.
Non-deterministic finite state machines are sometimes studied by the name sub shifts of finite type. Non-deterministic finite state machines are generalized by probabilistic automata, which assign a probability to each state transition.

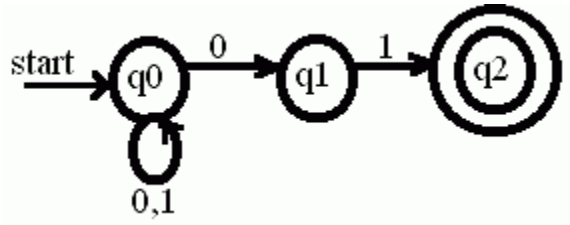
Formal definition

An *NFA* is represented formally by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, consisting of

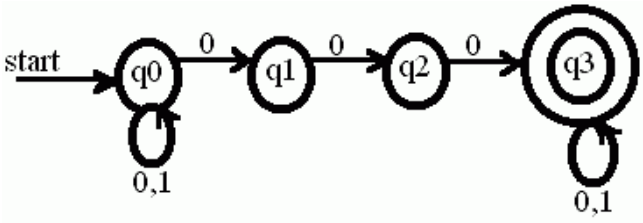
- a finite set of states Q
- a finite set of input symbols Σ
- a transition relation $\Delta : Q \times \Sigma \rightarrow P(Q)$.
- an *initial* (or *start*) state $q_0 \in Q$
- a set of states F distinguished as *accepting* (or *final*) *states* $F \subseteq Q$.

Examples and Exercises of NFA

1.Construct an NFA to accept all strings terminating in 01



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



NFA with epsilon moves(ε-NFA)

→The *NFA-ε* (also sometimes called *NFA-λ* or *NFA with epsilon moves*) replaces the transition function with one that allows the empty string ε as a possible input, so that one has instead

$\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$.

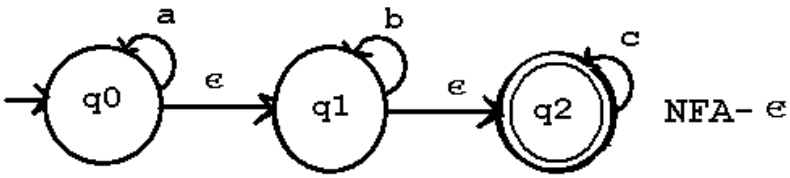
→It can be shown that ordinary NFA and NFA-ε are equivalent, in that, given either one, one can construct the other, which recognizes the same language.

→We can extend an NFA by introducing a "feature" that allows us to make a transition on , the empty string. All the transition lets us do is spontaneously make a transition, without receiving an input symbol. This is another mechanism that allows our NFA to be in multiple states at once. Whenever we take an edge, we must fork off a new "thread" for the NFA starting in the destination state.

→Just as non-determinism made NFA's more convenient to represent some problems than DFA's but were not more powerful; the same applies to εNFA's. While more expressive, anything we can represent with an εNFA we can represent with a DFA that has no ε transitions.

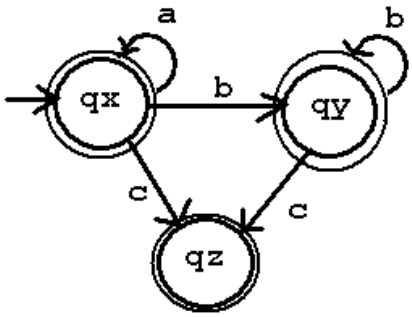
→The ε (epsilon) transition refers to a transition from one state to another without the reading of an input symbol (ie without the tape containing the input string moving). Epsilon transitions can be inserted between any states. There is also a conversion algorithm from a NFA with epsilon transitions to a NFA without epsilon transitions.

Consider the NFA-epsilon move machine M = { Q, Σ, δ, q0, F}
Q = { q0, q1, q2 }
Σ= { a, b, c } and ε moves
q0 = q0
F = { q2 }



NFA- ϵ

regular expression a* b* c*



NFA

Note: add an arc from qz to qz labeled "c" to figure above.

The language accepted by the above NFA with epsilon moves is the set of strings over {a,b,c} including the null string and all strings with any number of a's followed by any number of b's followed by any number of c's.

Now convert the NFA with epsilon moves to a NFA M = (Q', Σ, δ', q0', F') First determine the states of the new machine, Q' = the epsilon closure of the states in the NFA with epsilon moves. There will be the same number of states but the names can be constructed by writing the state name as the set of states in the epsilon closure. The epsilon closure is the initial state and all states that can be reached by one or more epsilon moves.
Thus q0 in the NFA-epsilon becomes {q0,q1,q2} because the machine can move from q0 to q1 by an epsilon move, then check q1 and find that it can move from q1 to q2 by an epsilon move.

q1 in the NFA-epsilon becomes {q1,q2} because the machine can move from

q1 to q2 by an epsilon move.

q2 in the NFA-epsilon becomes {q2} just to keep the notation the same. q2 can go nowhere except q2, that is what phi means, on an epsilon move. We do not show the epsilon transition of a state to itself here, but, beware, we will take into account the state to itself epsilon transition when converting NFA's to regular expressions.

The initial state of our new machine is {q0,q1,q2} the epsilon closure of q0

The final state(s) of our new machine is the new state(s) that contain a state symbol that was a final state in the original machine.

The new machine accepts the same language as the old machine, thus same sigma.
Exercises:

Conversion of NFA to DFA

Let $M_2 = < Q_2 , \Sigma, q_{2,0} , \delta_2 , A_2 >$ be an NFA that recognizes a language L. Then the DFA $M = < Q, \Sigma, q_0 , \delta , A >$ that satisfies the following conditions recognizes L:

$Q = 2^{Q_2}$, that is the set of all subsets of Q_2 ,
$q_0 = \{ q_{2,0} \}$,
$\delta(q, a) = \bigcup_{p \in q} \delta(p, a)$ for each state q in Q and each symbol a in Σ and
$A = \{ q \in Q \mid q \cap A_2 \neq \emptyset \}$

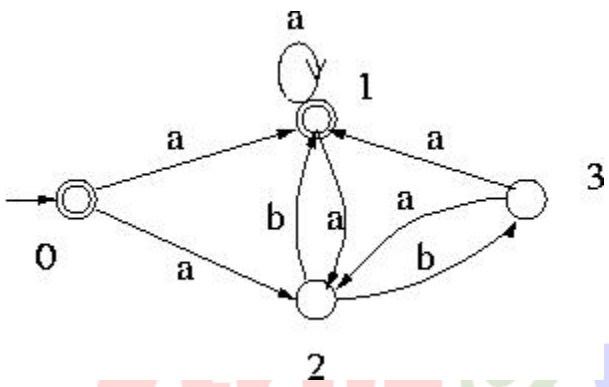
To obtain a DFA $M = < Q, \Sigma, q_0 , \delta , A >$ which accepts the same language as the given NFA $M_2 = < Q_2 , \Sigma, q_{2,0} , \delta_2 , A_2 >$ does, you may proceed as follows:

Initially $Q = \emptyset$.
First put $\{ q_{2,0} \}$ into Q. $\{ q_{2,0} \}$ is the initial state of the DFA M.
Then for each state q in Q do the following:
add the set $\bigcup_{p \in q} \delta(p, a)$, where δ here is that of NFA M_2, as a state to Q if it is not already in Q for each symbol a in Σ.
For this new state, add $\delta(q, a) = \bigcup_{p \in q} \delta(p, a)$ to δ, where the δ on the right hand side is that of NFA M_2.

When no more new states can be added to Q, the process terminates. All the states of Q that contain accepting states of M_2 are accepting states of M.

Note: The states that are not reached from the initial state are not included in Q obtained by this procedure. Thus the set of states Q thus obtained is not necessarily equal to 2^{Q_2} .

Example 1: Let us convert the following NFA to DFA.



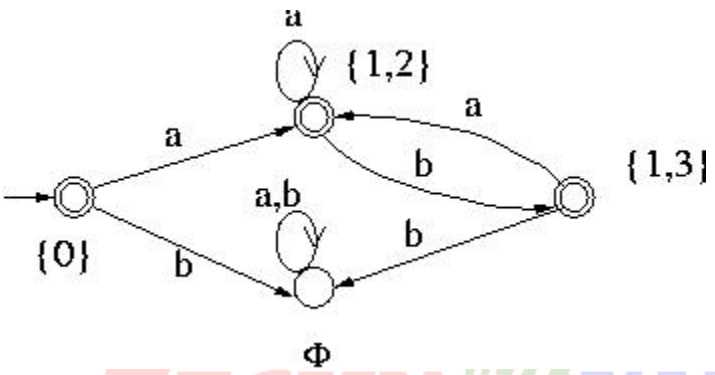
Initially Q is empty. Then since the initial state of the DFA is {0} , {0} is added to Q.
Since $\delta_2(0, a) = \{1, 2\}$, $\{1, 2\}$ is added to Q and $\delta(\{0\}, a) = \{1, 2\}$.
Since $\delta_2(0, b) = \emptyset$, $\emptyset$ is added to Q and $\delta(\{0\}, b) = \emptyset$.
At this point Q = { {0} , { 1 , 2 }, $\emptyset$ } .
Then since { 1 , 2 } is now in Q, the transitions from { 1 , 2 } on symbols a and b are computed.
Since $\delta_2(1, a) = \{1, 2\}$, and $\delta_2(2, a) = \emptyset$, $\delta(\{1, 2\}, a) = \{1, 2\}$. Similarly $\delta(\{1, 2\}, b) = \{1, 3\}$. Thus { 1 , 3 } is added to Q .
Similarly $\delta(\{1, 3\}, a) = \{1, 2\}$ and $\delta(\{1, 3\}, b) = \emptyset$. Thus no new states are added to Q . Since the transitions from all states of Q have been computed and no more states are added to Q, the conversion process stops here.

Note that there are no states of Q_2 in $\emptyset$. Hence there are no states that M_2 can go to from $\emptyset$. Hence

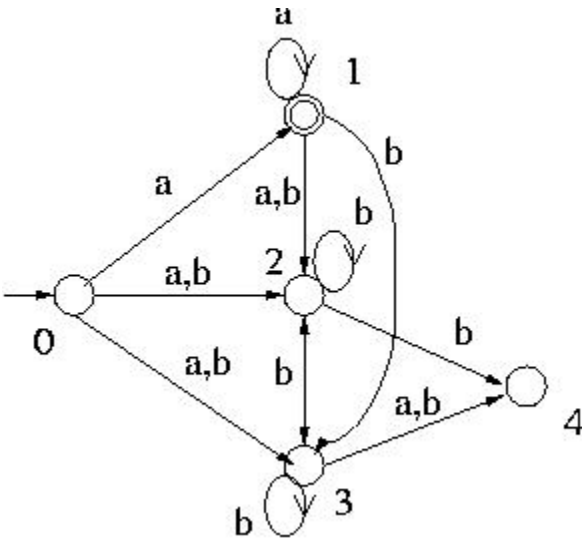
$\delta(\emptyset, a) = \emptyset$ $\delta(\emptyset, b) = \emptyset$.

For the accepting states of M, since states 0 and 1 are the accepting states of the NFA, all the states of Q that contain 0 and/or 1 are accepting states. Hence { 0 }, { 1 , 2 } and { 1 , 3 } are the accepting states of M.

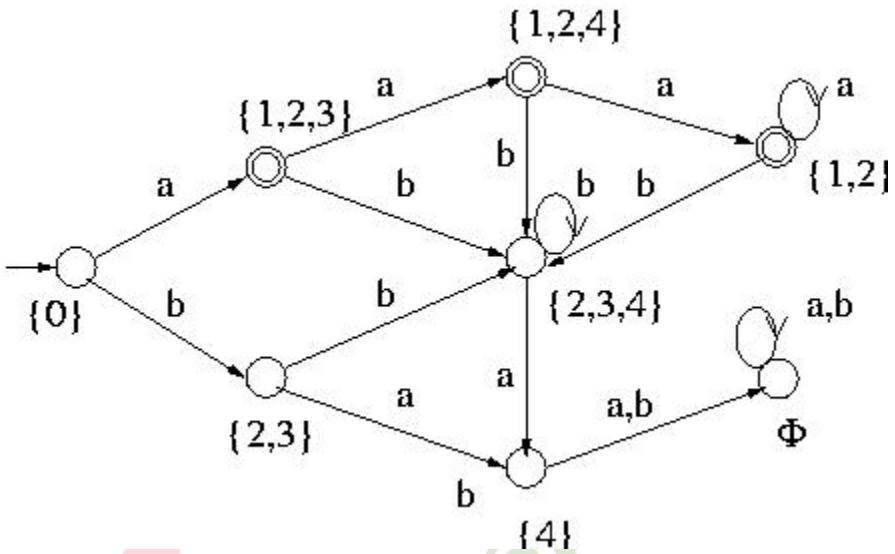
The DFA thus obtained is shown below.



Example 2: Similarly the NFA



is converted to the following DFA:



Regular Expression and Grammar

Regular operator

There are three regular operators used to generate a language which as mentioned below:-

1. Union (U): $L_1UL_2=\{S|S \varepsilon L_1 \text{ or } S \varepsilon L_2\}$
2. Concatenation (.):$L_1.L_2=\{S.t|S \varepsilon L_1 \text{ and } t \varepsilon L_2\}$
3. Kleene closure (*): $L^*=\textit{i=0}\infty \mathit{L}^i(L_0 \upsilon L_1 \upsilon L_2\ldots\ldots\ldots\ldots..)$
4. Positive closure (+): $L^+=\textit{i=1}\infty \mathit{L}^i(L_1 \upsilon L_2 \upsilon L_3\ldots\ldots\ldots\ldots..)$

Example:-

If $L_1=\{11,00\}, l_2=\{01,10\}$ over $\varepsilon=\{0,1\}$

then,

$L_1UL_2=\{11, 00, 01, 10\}$

$L_1.L_2=\{1101, 1110, 0001, 0010\}$

$L^*=\{\varepsilon, 11, 00, 1111, 11011\ldots\ldots\ldots\ldots\ldots\}$

L^+= {11, 00, 1111, 11011……}

The basic language

The simple language is of the form {a} where a ε Σ and the empty language ϕ and the language ε.

REGULAR LANGUAGE=> Basic language + Regular operator

The set of regular languages over an alphabet Σ is defined recursively as below. Any language belonging to this set is a regular language over Σ.

Definition of Set of Regular Languages:

Basis Clause: Ø, {Λ} and {a} for any symbol a ε Σ are regular languages.
Inductive Clause: If L_r and L_s are regular languages, then $L_r \cup L_s$, L_rL_s and L_r^* are regular languages.

Extremal Clause: Nothing is a regular language unless it is obtained from the above two clauses.

For example, let Σ= {a, b}. Then since {a} and {b} are regular languages, {a, b} (= {a} $\cup$ {b}) and {ab} (= {a}{b}) are regular languages. Also since {a} is regular, {a}* is a regular language which is the set of strings consisting of a's such as Λ, a, aa, aaa, aaaa etc. Note also that Σ^*, which is the set of strings consisting of a's and b's, is a regular language because {a, b} is regular.

Regular expression

Regular expressions are used to denote regular languages. They can represent regular languages and operations on them succinctly.
The set of regular expressions over an alphabet Σ is defined recursively as below. Any element of that set is a regular expression.

Basis Clause: Ø, Λ and **a** are regular expressions corresponding to languages Ø, {Λ} and {a}, respectively, where a is an element of Σ.
Inductive Clause: If **r** and **s** are regular expressions corresponding to languages L_r and L_s , then **(r + s)** , **(rs)** and **(r*)** are regular expressions corresponding to languages $L_r \cup L_s$, L_rL_s and L_r^* , respectively.
Extremal Clause: Nothing is a regular expression unless it is obtained from the above two clauses.

Examples of regular expression and regular languages corresponding to them

- $(a + b)^2$ corresponds to the language {aa, ab, ba, bb}, that is the set of strings of length 2 over the alphabet {a, b}. In general $(a + b)^k$ corresponds to the set of strings of length k over the alphabet {a, b}. $(a + b)^*$ corresponds to the set of all strings over the alphabet {a, b}.
- a^*b^* corresponds to the set of strings consisting of zero or more a's followed by zero or more b's.
- $a^*b^+a^*$ corresponds to the set of strings consisting of zero or more a's followed by one or more b's followed by zero or more a's.
- $(ab)^+$ corresponds to the language {ab, abab, ababab, ... }, that is, the set of strings of repeated ab's.

Note: A regular expression is not unique for a language. That is, a regular language, in general, corresponds to more than one regular expression. For example $(a + b)^*$ and $(a^*b^*)^*$ correspond to the set of all strings over the alphabet {a, b}.

Definition of Equality of Regular Expressions

Regular expressions are **equal** if and only if they correspond to the same language.

Thus for example $(a + b)^* = (a^*b^*)^*$, because they both represent the language of all strings over the alphabet {a, b}.

In general, it is not easy to see by inspection whether or not two regular expressions are equal.

Examples and Exercises related to R.E

Ex. 1: *Find the shortest string that is not in the language represented by the regular expression* $a^*(ab)^*b^*$.

Solution: It can easily be seen that Λ, a, b, which are strings in the language with length 1 or less. Of the strings wiht length 2 aa, bb and ab are in the language. However, ba is not in it. Thus the answer is ba.

Ex. 2: For the two regular expressions given below,
(a) find a string corresponding to r_2 but not to r_1 and
(b) find a string corresponding to both r_1 and r_2.

$r_1 = a^* + b^*$ $r_2 = ab^* + ba^* + b^*a + (a^*b)^*$

Solution: (a) Any string consisting of only a's or only b's and the empty string are in r_1. So we need to find strings of r_2 which contain at least one a and at least one b. For example ab and ba are

such strings.
(b) A string corresponding to $\mathbf{r_1}$ consists of only a's or only b's or the empty string. The only strings corresponding to $\mathbf{r_2}$ which consist of only a's or b's are a, b and the strings consiting of only b's (from $\mathbf{(a^*b)^*}$).

Ex. 3: Let r1 and r2 be arbitrary regular expressions over some alphabet. Find a simple (the shortest and with the smallest nesting of * and +) regular expression which is equal to each of the following regular expressions.

(a) $\mathbf{(r_1 + r_2 + r_1r_2 + r_2r_1)^*}$
(b) $\mathbf{(r_1(r_1 + r_2)^*)^+}$

Solution: One general strategy to approach this type of question is to try to see whether or not they are equal to simple regular expressions that are familiar to us such as $\mathbf{a}$, $\mathbf{a^*}$, $\mathbf{a^+}$, $\mathbf{(a + b)^*}$, $\mathbf{(a + b)^+}$ etc.
(a) Since $\mathbf{(r_1 + r_2)^*}$ represents all strings consisting of strings of $\mathbf{r_1}$ and/or $\mathbf{r_2}$, $\mathbf{r_1r_2 + r_2r_1}$ in the given regular expression is redundant, that is, they do not produce any strings that are not represented by $\mathbf{(r_1 + r_2)^*}$. Thus $\mathbf{(r_1 + r_2 + r_1r_2 + r_2r_1)^*}$ is reduced to $\mathbf{(r_1 + r_2)^*}$.
(b) $\mathbf{(r_1(r_1 + r_2)^*)^+}$ means that all the strings represented by it must consist of one or more strings of $\mathbf{(r_1(r_1 + r_2)^*)}$. However, the strings of $\mathbf{(r_1(r_1 + r_2)^*)}$ start with a string of $\mathbf{r_1}$ followed by any number of strings taken arbitrarily from $\mathbf{r_1}$ and/or $\mathbf{r_2}$. Thus anything that comes after the first $\mathbf{r_1}$ in $\mathbf{(r_1(r_1 + r_2)^*)^+}$ is represented by $\mathbf{(r_1 + r_2)^*}$. Hence $\mathbf{(r_1(r_1 + r_2)^*)}$ also represents the strings of $\mathbf{(r_1(r_1 + r_2)^*)^+}$, and conversely $\mathbf{(r_1(r_1 + r_2)^*)^+}$ represents the strings represented by $\mathbf{(r_1(r_1 + r_2)^*)}$. Hence $\mathbf{(r_1(r_1 + r_2)^*)^+}$ is reduced to $\mathbf{(r_1(r_1 + r_2)^*)}$.

Ex. 4: Find a regular expression corresponding to the language L over the alphabet { a , b } defined recursively as follows:

Basis Clause: $\Lambda \varepsilon L$
Inductive Clause: If x ε L , then aabx ε L and xbb ε L .
Extremal Clause: Nothing is in L unless it can be obtained from the above two clauses.

Solution: Let us see what kind of strings are in L. First of all $\Lambda \varepsilon L$. Then starting with Λ, strings of L are generated one by one by prepending aab or appending bb to any of the already generated strings. Hence a string of L consists of zero or more aab's in front and zero or more bb's following them. Thus $\mathbf{(aab)^*(bb)^*}$ is a regular expression for L.

Ex. 5: Find a regular expression corresponding to the language L defined recursively as follows:

Basis Clause: $\Lambda \varepsilon L$ and a ε L .
Inductive Clause: If x ε L , then aabx ε L and bbx ε L .

Extremal Clause: Nothing is in L unless it can be obtained from the above two clauses.

Solution: Let us see what kind of strings are in L. First of all Λ and a are in L . Then starting with Λ or a, strings of L are generated one by one by prepending aab or bb to any of the already generated strings. Hence a string of L has zero or more of aab's and bb's in front possibly followed by a at the end. Thus $\mathbf{(aab + bb)^*(a + \Lambda)}$ is a regular expression for L.

Ex. 6: Find a regular expression corresponding to the language of all strings over the alphabet { a, b } that contain exactly two a's.

Solution: A string in this language must have at least two a's. Since any string of b's can be placed in front of the first a, behind the second a and between the two a's, and since an arbitrasry string of b's can be represented by the regular expression $\mathbf{b^*}$, $\mathbf{b^*a\ b^*a\ b^*}$ is a regular expression for this language.

Ex. 7: Find a regular expression corresponding to the language of all strings over the alphabet { a, b } that do not end with ab.

Solution: Any string in a language over { a , b } must end in a or b. Hence if a string does not end with ab then it ends with a or if it ends with b the last b must be preceded by a symbol b. Since it can have any string in front of the last a or bb, $\mathbf{(a + b)^*(a + bb)}$ is a regular expression for the language.

Ex. 8: Find a regular expression corresponding to the language of all strings over the alphabet { a, b } that contain no more than one occurence of the string aa.

Solution: If there is one substring aa in a string of the language, then that aa can be followed by any number of b. If an a comes after that aa, then that a must be preceded by b because otherwise there are two occurences of aa. Hence any string that follows aa is represented by $\mathbf{(b + ba)^*}$. On the other hand if an a precedes the aa, then it must be followed by b. Hence a string preceding the aa can be represented by $\mathbf{(b + ab)^*}$. Hence if a string of the language contains aa then it corresponds to the regular expression $\mathbf{(b + ab)^*aa(b + ba)^*}$.
If there is no aa but at least one a exists in a string of the language, then applying the same argument as for aa to a, $\mathbf{(b + ab)^*a(b + ba)^*}$ is obtained as a regular expression corresponding to such strings.
If there may not be any a in a string of the language, then applying the same argument as for aa to Λ, $\mathbf{(b + ab)^*(b + ba)^*}$ is obtained as a regular expression corresponding to such strings.
Altogether $\mathbf{(b + ab)^*(\Lambda + a + aa)(b + ba)^*}$ is a regular expression for the language.

Ex. 9: Find a regular expression corresponding to the language of strings of even lengths over the alphabet of { a, b }.

Solution: Since any string of even length can be expressed as the concatenation of strings of length 2 and since the strings of length 2 are aa, ab, ba, bb, a regular expression corresponding to the language is **(aa + ab + ba + bb)*** . Note that 0 is an even number. Hence the string Λ is in this language.

Ex. 10: Describe as simply as possible in English the language corresponding to the regular expression a*b(a*ba*b)*a* .

Solution: A string in the language can start and end with a or b, it has at least one b, and after the first b all the b's in the string appear in pairs. Any numbe of a's can appear any place in the string. Thus simply put, it is the set of strings over the alphabet { a, b } that contain an odd number of b's

Ex. 11: Describe as simply as possible in English the language corresponding to the regular expression ((a + b)3)*(Λ + a + b) .

Solution: ((a + b)^3) represents the strings of length 3. Hence **((a + b)^3)*** represents the strings of length a multiple of 3. Since **((a + b)^3)*(a + b)** represents the strings of length 3n + 1, where n is a natural number, the given regular expression represents the strings of length 3n and 3n + 1, where n is a natural number.

Ex. 12: Describe as simply as possible in English the language corresponding to the regular expression (b + ab)*(a + ab)*.

Solution: (b + ab)* represents strings which do not contain any substring aa and which end in b, and **(a + ab)*** represents strings which do not contain any substring bb. Hence altogether it represents any string consisting of a substring with no aa followed by one b followed by a substring with no bb.

Properties of Regular Expressions (R.E)

1. Commutative:

The union of Regular expression is commutative, let L and K are two languages represented by R.E L and R.

2. Associativity:

The union and concatenation operation of R.E are associative. Let L,R,S are RE's represented of languages L,R and s then,

L+ (R+S) = (L+R) + S

L (RS) = (LR) S

3. Identities:

ϕ is the identity from union i.e. ϕ + R= R + ϕ =R

ε is the identity for concatenation εR=Rε=R.

4. Annihilator:

An annihilator for an operation is a value such that when operator is applied with that value and another value, the result of operation is annihilator.
ϕ is the annihilator for concatenation then,
ϕR=Rϕ=R.

5. Idempotent law:

If R is R.E then R+R=R

6. Law of closure:

If R is R.E the $((R)^*)^*=R^*$
ϕ= closure of ϕ= $\phi^*=\phi$
ε=closure of ε= $\varepsilon^*=\varepsilon$

Theorem1

If L, M and N are any language then prove: L (M ∪ N) = LM ∪ LN

Proof:
Let w is a string such that w=xy we have to show that w ε L (M ∪ N) iff w ε LM ∪ LN
Solution:
if w ε LM ∪ LN then,w ε LM and w ε LN (by union rule).
xy ε LM then, x ε L or y ε M (by concatenation rule).
xy ε LN then, x ε L and y ε N (by concatenation rule).

Hence this implies;
xy ε L (M ∪ N)

I.e. w ε L (M ∪ N)
Proved.
onlyif (iff):
w ε L (M ∪ N) then, xy ε L(M ∪ N)
x ε L and y ε (M ∪ N). (by concatenation rule)

if y ε M then xy ε LM
if y ε N then xy ε LN

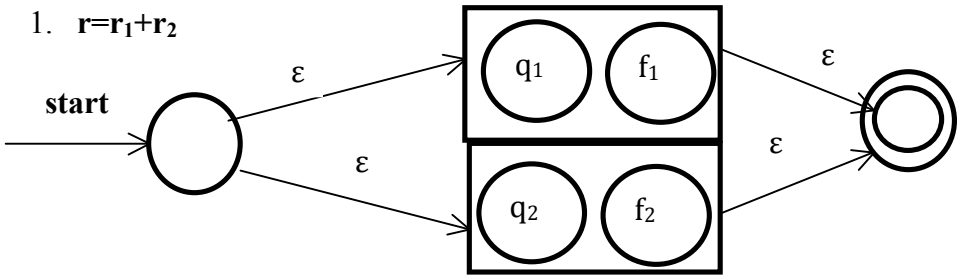
so,xy ε LM ∪ LN
Hence, w ε LM ∪ LN
Proved.

Theorem 2

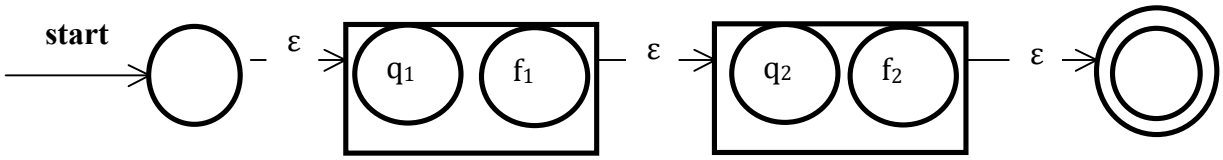
For any R.E r,there is and ε-NFA that accepts the same language represented by r.

Proof:-

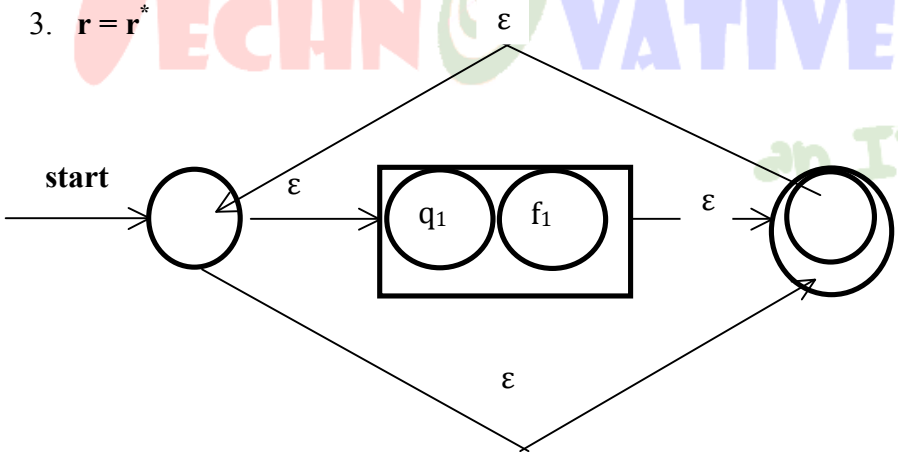
1. **$r=r_1+r_2$**



2. **$r=r_1.r_2$**



3. **$r = r^*$**



Conversion from DFA to R.E

Arden's Theorem

Let p and q be two regular expression over alphabet ε,if p doesn't contain empty string then r= q + rp has a unique solution.

i.e.

$$\boxed{r=qp^*}$$

Proof:

r=q + rp

r= qr (q +rp) p=q + qp + rp^2

Substituting r= q + rp again and again

$r = q + qp + qp^2 + qp^3 + \ldots\ldots\ldots\ldots\ldots\ldots$
$= q(\varepsilon + p + p^2 + p^3 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots)$
$= qp^*$
Proved.

Use of Arden's rule:

To convert DFA into R.E there are certain assumption regarding the transition system. They are as follow;

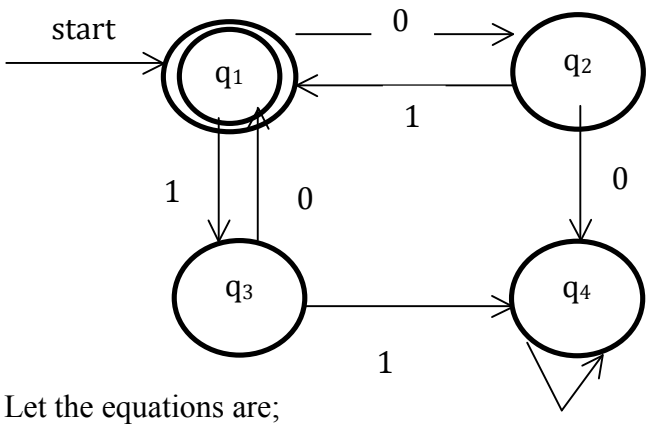
i) The transition diagram should have ε-transition.
ii) It must have only one single starting state.
iii) Its vertices are $q_1, q_2, q_3, \ldots\ldots\ldots\ldots\ldots q_n$.
iv) q_i is a final state.
v) w_{ij} denotes the regular expressions representating the set of labels of edges from q_i to q_j.

We can get the following condition.

$q_1 = q_1w_{11} + q_2w_{21} + q_3w_{31} + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + q_nw_{n1} + \varepsilon$
$q_2 = q_1w_{12} + q_2w_{22} + q_3w_{32} + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + q_nw_{n2} + \varepsilon$
.
.
.
$q_n = q_1w_{1n} + q_2w_{2n} + q_3w_{3n} + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + q_nw_{nn} + \varepsilon$

Hence, solving these equations for q_1 in terms of w_{ij}'s gives R.E.

Example:
Convert the following DFA to R.E



Let the equations are;

$q_1 = q_21 + q_30 + \varepsilon$ ……..(i)

$q_2 = q_10$ …….. (ii)

$q_3 = q_11$ …….. (iii)

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

Now put q_2 and q_3 in eq^n(i)

$q_1 = q_1 01 + q_1 10 + \varepsilon$

$= \varepsilon + q_1 (01+10)$

where,

$q=\varepsilon$

$r=q$

$p=01+10$

Therefore, $q_1 = \varepsilon (01+10)^*$

since, q_1 is the final state.

so, R.E $= \varepsilon (01+10)^*$

$= (01+10)^*$ is the required R.E from given diagram.

Pumping lemma for regular languages

The pumping lemma for regular languages describes an essential property of all regular languages. Informally, it says that all sufficiently long words in a regular language may be *pumped* that is, have a middle section of the word repeated an arbitrary number of times to produce a new word which also lies within the same language.
Specifically, the pumping lemma says that for any regular language L there exists a constant p such that any word w in L with length at least p can be split into three substrings, $w = xyz$, where the middle portion y must not be empty, such that the words xz, xyz, $xyyz$, $xyyyz$, … constructed by repeating y an arbitrary number of times (including zero times) are still in L. This process of repetition is known as "pumping". Moreover, the pumping lemma guarantees that the length of xy will be at most p, imposing a limit on the ways in which w may be split. Finite languages trivially satisfy the pumping lemma by having p equal to the maximum string length in L plus one.

Here's what the pumping lemma says:

- If an infinite language is regular, it can be defined by a DFA.
- The DFA has some finite number of states (say, n).
- Since the language is infinite, some strings of the language must have length > n.
- For a string of length > n accepted by the DFA, the walk through the DFA must contain a cycle.
- Repeating the cycle an arbitrary number of times must yield another string accepted by the DFA.

The pumping lemma for regular languages is another way of proving that a given (infinite) language is not regular. (The pumping lemma *cannot* be used to prove that a given language *is* regular.)

The Pumping Lemma is generally used to prove a language is not regular.

If a DFA, NFA or NFA-epsilon machine can be constructed to exactly accept a language, then the language is a Regular Language. If a regular expression can be constructed to exactly generate the strings in a language, then the language is regular.

If a regular grammar can be constructed to exactly generate the strings in a language, then the language is regular.

To prove a language is not regular requires a specific definition of the language and the use of the Pumping Lemma for Regular Languages.

A note about proofs using the Pumping Lemma:

Given: Formal statements A and B.
A implies B.
If you can prove B is false, then you have proved A is false.

For the Pumping Lemma, the statement "A" is "L is a Regular Language",
The statement "B" is a statement from the Predicate Calculus.
(This is a plain text file that uses words for the upside down A that reads 'for all' and the backwards E that reads 'there exists')

Applying the Pumping Lemma

Here's a more formal definition of the pumping lemma:

If L is an infinite regular language, then there exists some positive integer m such that any string w ∈ L whose length is m or greater can be decomposed into three parts, xyz, where

- |xy| is less than or equal to m,
- |y| > 0,
- $w_i = xy^iz$ is also in L for all i = 0, 1, 2, 3, ...

Here's what it all means:

- m is a (finite) number chosen so that strings of length m or greater *must* contain a cycle. Hence, m must be equal to or greater than the number of states in the dfa. Remember that we *don't know the dfa,* so we can't actually choose m; we just know that such an m must exist.

- Since string w has length greater than or equal to m, we can break it into two parts, xy and z, such that xy must contain a cycle. We don't know the dfa, so we don't know exactly where to make this break, but we know that |xy| can be less than or equal to m.
- We let x be the part before the cycle, y be the cycle, and z the part after the cycle. (It is possible that x and z contain cycles, but we don't care about that.) Again, we don't know exactly where to make this break.
- Since y is the cycle we are interested in, we must have |y| > 0, otherwise it isn't a cycle.
- By repeating y an arbitrary number of times, xy*z, we must get other strings in L.
- If, despite all the above uncertainties, we can show that the dfa has to accept some string that we know is not in the language, then we can conclude that the language is not regular.

Formal statement of the Pumping Lemma:

L is a Regular Language implies
(there exists n)(for all z)[z in L and |z|>=n implies
{(there exists u,v,w)(z = uvw and |uv|<=n and |v|>=1 and
i
(for all i>=0)(uv w is in L))}]

The two commonest ways to use the Pumping Lemma to prove a language is NOT regular are:

a) show that there is no possible n for the (there exists n),
this is usually accomplished by showing a contradiction such as (n+1)(n+1) < n*n+n

b) show there is no way to partition z into u, v and w such that
i
uv w is in L, typically for a value i=0 or i=2.
Be sure to cover all cases by argument or enumerating cases.

Examples and Exercises related to pumping lemma for regular language.

1. Prove that L = {0^i | i is a perfect square} is not a regular language.
Proof: Assume that L is regular and let m be the integer guaranteed by the pumping lemma. Now, consider the string w = 0^{m^2}. Clearly w ∈ L, so w can be written as w = xyz with |xy| ≤ m and y ≠ λ (or |y| > 0). Consider what happens when i = 2. That is, look at xy^2z. Then, we have $m^2 = |w| < |xy^2z| \leq m^2 + m = m(m + 1) < (m + 1)^2$. That is, the length of the string xy^2z lies between two consecutive perfect squares. This means $xy^2z \notin L$ contradicting the assumption that L is regular.

2. Prove that L = {ww | w ∈ {a, b}*} is not regular.

Assume L is regular and let m be the integer from the pumping lemma. Choose w = a^mba^mb. Clearly, w ∈ L so by the pumping lemma, w = xyz such that |xy| ≤ m. |y| > 0 and $xy^iz \in L$ for all i ≥ 0. Let p = |y|. Consider what happens when i = 0. The resulting string, xz = $a^{m-p}ba^mb$. Since p

≥ 1, the number of a's in the two runs are not the same, and thus this string is not in L. Therefore L is not regular.

3. Prove that L = {$a^n b^n$: n >0} is not regular.

1. We don't know m, but assume there is one.
2. Choose a string w = $a^n b^n$ where n > m, so that any prefix of length m consists entirely of a's.
3. We don't know the decomposition of w into xyz, but since |xy| ≤m, xy must consist entirely of a's. Moreover, y cannot be empty.
4. Choose i = 0. This has the effect of dropping |y| a's out of the string, without affecting the number of b's. The resultant string has fewer a's than b's, hence does not belong to L. Therefore L is not regular.

4. Prove that L = {$a^n b^k$: n > k and n≠ 0} is not regular.

1. We don't know m, but assume there is one.
2. Choose a string w = $a^n b^k$ where n > m, so that any prefix of length m consists entirely of a's, and k = n-1, so that there is just one more a than b.
3. We don't know the decomposition of w into xyz, but since |xy| ≤m, xy must consist entirely of a's. Moreover, y cannot be empty.
4. Choose i = 0. This has the effect of dropping |y| a's out of the string, without affecting the number of b's. The resultant string has fewer a's than before, so it has either fewer a's than b's, or the same number of each. Either way, the string does not belong to L, so L is not regular.

5. Prove that L = {a^n: n is a prime number} is not regular.

1. We don't know m, but assume there is one.
2. Choose a string w = a^n where n is a prime number and |xyz| = n > m+1. (This can always be done because there is no largest prime number.) Any prefix of w consists entirely of a's.
3. We don't know the decomposition of w into xyz, but since |xy| ≤m, it follows that |z| > 1. As usual, |y| > 0,
4. Since |z| > 1, |xz| > 1. Choose i = |xz|. Then $|xy^i z| = |xz| + |y||xz| = (1 + |y|)|xz|$. Since (1 + |y|) and |xz| are each greater than 1, the product must be a composite number. Thus $|xy^i z|$ is a composite number.

Context-free grammar

Definition

Context Free Grammar is defined by four tuple, G = (T, N, S, P) where,

- T is set of terminals (lexicon)

• N is set of non-terminals For NLP, we usually distinguish out a set P ⊂ N of pre-terminals which always rewrite as terminals.
• S is start symbol (one of the non-terminals)
• R is rules/productions of the form X → γ, where X is a non-terminal and γ is a sequence of terminals and non-terminals (may be empty).
• A grammar G generates a language L.

An example context-free grammar

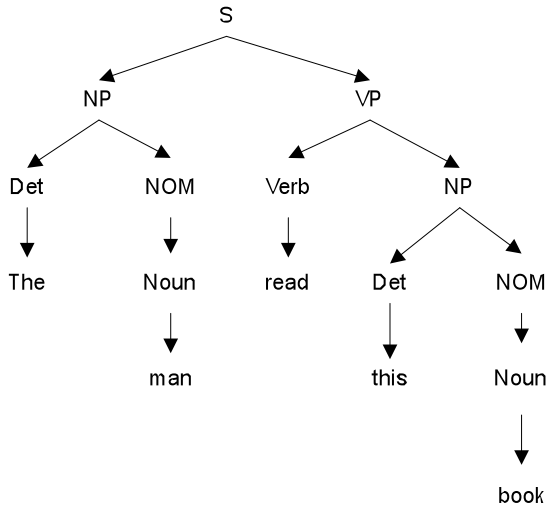
G = (T, N, S, R)
T = {that, this, a, the, man, book, flight, meal, include, read, does}
N = {S, NP, NOM, VP, Det, Noun, Verb, Aux}
S = S
R = {
S → NP VP Det → that | this | a | the
S → Aux NP VP Noun → book | flight | meal | man
S → VP Verb → book | include | read
NP → Det NOM Aux → does
NOM → Noun
NOM → Noun NOM
VP → Verb
VP → Verb NP
}

Application of grammar rules

S → NP VP Det → that | this | a | the
S → Aux NP VP Noun → book | flight | meal | man
S → VP Verb → book | include | read
NP → Det NOM Aux → does
NOM → Noun
NOM → Noun NOM
VP → Verb
VP → Verb NP

S → NP VP
→ Det NOM VP
→ The NOM VP
→ The Noun VP
→ The man VP
→ The man Verb NP
→ The man read NP
→ The man read Det NOM
→ The man read this NOM
→ The man read this Noun
→ The man read this book

Parse tree



Why such grammars are called `context free'? *Because all rules contain only one symbol on the left hand side --- and wherever we see that symbol while doing a derivation, we are free to replace it with the stuff on the right hand side.* That is, the `context' in which a symbol on the left hand side of a rule occurs is unimportant --- we can *always* use the rule to make the rewrite while doing a derivation.

A language is called *context free* if it is generated by some context free grammar. For example, the language $a^n b^n$ is context free. Not all languages are context free. For example, $a^n b^n c^n$ is not.

BNF(Backus Normal Form)

BNF (**Backus Normal Form** or **Backus–Naur Form**) is one of the two main notation techniques for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols.

Backus-Naur Form is the name of many closely related Meta Languages for describing the syntax of a Programming Language.

A BNF specification is a set of derivation rules, written as

<symbol> **::=** __expression__

where <symbol> is a *nonterminal*, and the __expression__ consists of one or more sequences of symbols; more sequences are separated by the vertical bar, '|', indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are *terminals*. On the other hand, symbols that appear on a left side are *non-terminals* and are always enclosed between the pair <>.

The meta-symbols of BNF are:

::=
: meaning "is defined as"

|
: meaning "or"

< >
: angle brackets used to surround category names.

The angle brackets distinguish syntax rules names (also called non-terminal symbols) from terminal symbols which are written exactly as they are to be represented. A BNF rule defining a nonterminal has the form:

nonterminal **::=** sequence_of_alternatives consisting of strings of
terminals or nonterminals separated by the meta-symbol |

For example, the BNF production for a mini-language is:

<program> ::= program
<declaration_sequence>
begin
<statements_sequence>
end ;

This shows that a mini-language program consists of the keyword "program" followed by the declaration sequence, then the keyword "begin" and the statements sequence, finally the keyword "end" and a semicolon.

Examples of CFG

Example 1

We have shown that $L = \{a^n b^n : n \geq 0\}$ is not regular. Here is a context-free grammar for this language.

$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow \lambda\}$

Example 2

We have shown that $L = \{a^n b^k : k > n \geq 0\}$ is not regular. Here is a context-free grammar for this language.

$G = (\{S, B\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow B, B \rightarrow bB, B \rightarrow b\})$.

Example 3

The language $L = \{ww^R : w \in \{a, b\}^*\}$, where each string in L is a palindrome, is not regular. Here is a context-free grammar for this language.

$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \lambda\})$.

Example 4

The language $L = \{w : w \in \{a, b\}^*, n_a(w) = n_b(w)\}$, where each string in L has an equal number of a's and b's, is not regular. Consider the following grammar:

$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow bSa, S \rightarrow SS, S \rightarrow \lambda\})$.

Example 5

The language L, consisting of balanced strings of parentheses, is context-free but not regular. The grammar is simple, but we have to be careful to keep our symbols (and) separate from our meta-symbols (and).

$G = (\{S\}, \{(,)\}, S, \{S \rightarrow (S), S \rightarrow SS, S \rightarrow \lambda\})$

Sentential Forms

A sentential form is the start symbol S of a grammar or any string in $(V \cup T)^*$ that can be derived from S.

Consider the linear grammar

$(\{S, B\}, \{a, b\}, S, \{S \rightarrow aS, S \rightarrow B, B \rightarrow bB, B \rightarrow \lambda\})$.

A derivation using this grammar might look like this:

$S \Rightarrow aS \Rightarrow aB \Rightarrow abB \Rightarrow abbB \Rightarrow abb$

Each of {S, aS, aB, abB, abbB, abb} is a sentential form.

Because this grammar is linear, each sentential form has at most one variable. Hence there is never any choice about which variable to expand next.

Pumping Lemma for context free Language

Size of Parse Tree: Any grammar in CNF produces a parse tree for any string that is binary tree.

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

Proof: By Induction

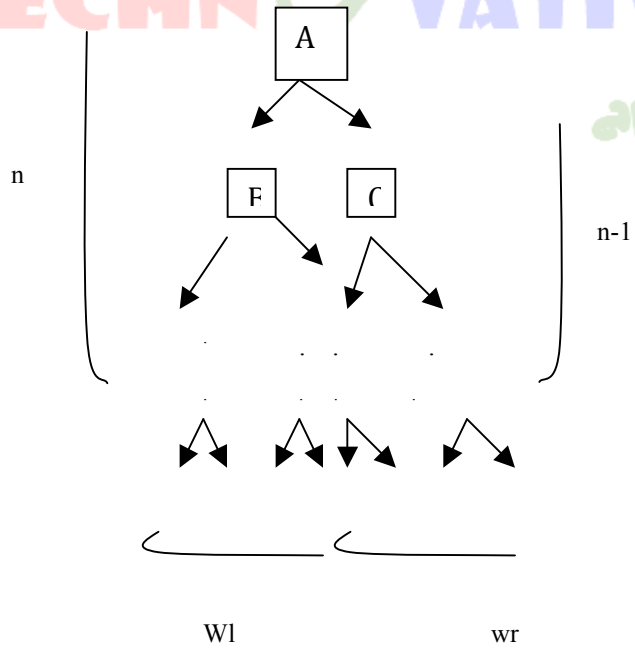
Basis: If n=1 then there consists of only a root and a leaf labeled with terminal.

So, string w is a single terminal.

$|w| = 1 = 2^{1-1} = 2^0 = 1$ which is true.

Inductive: Suppose n is the length longest path and n>1. The root of the tree uses a production A→ BC since n>1

No path in the sub trees rooted at B and C can have length greater than n-1. Since B and C are child of A.



By inductive hypothesis, the yield of these sub trees are of length at most 2^{n-2} (since n=n-1)

So the yield of entire tree is the concatenation of these two yields i.e.

$|w| \leq 2^{n-2} + 2^{n-2}$

$|w| \leq 2.2^{n-2}$

$|w| \leq 2^{1+ n-2}$

$|w| \leq 2^{n-1}$ .Hence Proved

Statement of Pumping Lemma: Let L be a CFL. There exist a constant n such that if z is any string L such that $|z| \geq n$, there are strings uvwxy satisfying

z = uvwxy

$|vx| > 0$

$|vwx| \leq n$

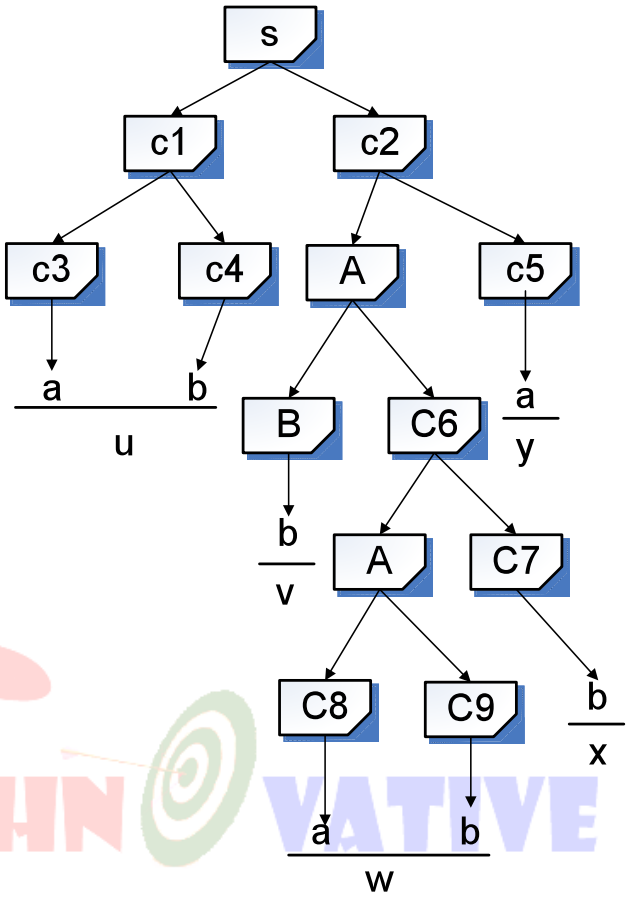
For all $i \geq 0$, $uv^iwx^iy \in L$

Proof: First step, we can find a CNF grammar for grammar G of L that generate L-{€}

- Let P be the number of variables in the grammar and also $n=2^P$.

  Since the derivation tree for z is binary tree being CNF grammar, it must have height at least P+1. Since any parse tree whose lowest path is P must have a yield of length at most 2^{P-1}.

- Let us consider a path of maximum length and look at the portion bottom, consisting of a leaf node and P nodes above it.



Each of these p nodes corresponds to a variable and since there are only P distinct variables. Some variable A must appears twice in this portion.

- Let w be the portion of z derived from A closest the leaf and t=vwx be the portion of z derived from other A. if u and y represent the beginning and ending portion of z, we have z= uvwxy.

- The A closest to the root in this portion of the path is the root of a binary derivation tree for vwx. Since we begin with a path of maximum length, this tree has height $\leq p+1$ and so, $|vwx| \leq 2^P$
  $|vwx| \leq n$

- The node containing this A has two children both correspond to a variable. If we let B denote the one that is not ancestor of the other A. The string of terminals derived from B doesn't overlap x. It follows that either v or x is not null. So, $|vx| > 0$

- Finally, $S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy$

First A in one for higher node, the second in lowers and so on. Third and successive derivation concludes the theorem i.e. $uv^iwx^jy \in L$ For all $i \geq 0$.

What is derivation tree?
Given a grammar with the usual representation G = (V, T, P, S) with variables V, terminal symbols T, set of productions P and the start symbol from V called S.

A derivation tree is constructed with
1) Each tree vertex is a variable or terminal or epsilon
2) The root vertex is S
3) Interior vertices are from , leaf vertices are from T or epsilon
4) An interior vertex A has children, in order, left to right, X1, X2, ... , Xk when there is a production in P of the form A -> X1 X2 ... Xk
5) A leaf can be epsilon only when there is a production A -> epsilon and the leafs parent can have only this child.

- A grammar may have an unbounded number of derivation trees. It just depends on which production is expanded at each vertex.
- For any valid derivation tree, reading leafs from left to right gives one string in the language defined by the grammar. There may be many derivation trees for a single string in the language.
- If the grammar is a CFG then a leftmost derivation tree exists for every string in the corresponding CFL. There may be more than one leftmost derivation trees for some string. If the grammar is a CFG then a rightmost derivation tree exists for every string in the corresponding CFL. There may be more than one rightmost derivation tree for some string.

Ambiguous Grammar
The grammar is called "ambiguous" if the leftmost (rightmost) derivation tree is not unique for every string in the language defined by the grammar. The leftmost and rightmost derivations are usually distinct but might be the same.

Leftmost derivation Tree
Given a grammar and a string in the language represented by the grammar, a leftmost derivation tree is constructed bottom up by finding a production in the grammar that has the leftmost character of the string (possibly more than one may have to be tried) and building the tree towards the root. Then work on the second character of the string. After much trial and error, you should get a derivation tree with a root S.

Examples: Construct a grammar for L = { x 0^n y 1^n z n>0 }
Recognize that 0^n y 1^n is a base language, say B
B -> y | 0B1 (The base y, the recursion 0B1)
Then, the language is completed S -> xBz using the prefix, base language and suffix.
(Note that x, y and z could be any strings not involving n)

G = (V, T, P, S) where
V = { B, S } T = { x, y, z, 0, 1 } S = S
P = S -> xBz
B -> y | 0B1

Now construct an arbitrary derivation for $S \overset{*}{\underset{G}{\Rightarrow}} x00y11z$

A derivation always starts with the start variable, S. The "=>", "*" and "G" stand for "derivation", "any number of steps", and "over the grammar G" respectively.

The intermediate terms, called **sentential form**, may contain variable and terminal symbols.

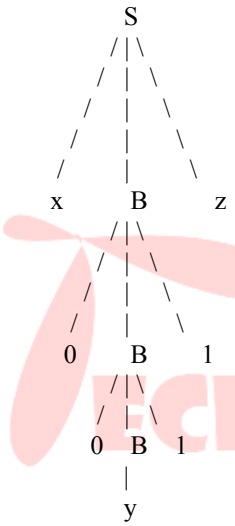
Any variable, say B, can be replaced by the right side of any production of the form B -> <right side>

A leftmost derivation always replaces the leftmost variable in the sentential form.
One possible derivation using the grammar above is

S => xBz => x0B1z => x00B11z => x00y11z

The derivation must obviously stop when the sentential form has only terminal symbols. (No more substitutions possible.) The final string is in the language of the grammar. But, this is a very poor way to generate all strings in the grammar!

A "derivation tree" sometimes called a "**parse tree**" uses the rules above: start with the starting symbol, expand the tree by creating branches using any right side of a starting symbol rule, etc.

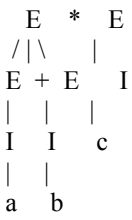


Derivation ends x 0 0 y 1 1 z with all leaves terminal symbols, a string in the language generated by the grammar.

Example 2:
Given G = (V, T, P, S) V={S, E, I} T={a, b, c} S=S
P=

I -> a | b | c
E -> I | E+E | E*E
S -> E (a subset of grammar from book)

Given a string a + b * c



Example 3: Leftmost and rightmost derivation
Now consider the grammar
G = ({S, A, B, C}, {a, b, c}, S, P) where = {S→ABC, A→aA, A→λ, B→bB, B→λ ,C→cC, C→λ }.
With this grammar, there is a choice of variables to expand. Here is a sample derivation:

S ⇒ ABC ⇒ aABC ⇒ aABcC ⇒ aBcC ⇒ abBcC⇒ abBc⇒ abbBc ⇒ abbc
If we always expanded the leftmost variable first, we would have a leftmost derivation:

S ⇒ ABC⇒ aABC⇒ aBC ⇒ abBC⇒ abbBC ⇒ abbC ⇒ abbcC⇒ abbc
Conversely, if we always expanded the rightmost variable first, we would have a rightmost derivation:

S ⇒ ABC ⇒ ABcC ⇒ ABc ⇒ AbBc ⇒ AbbBc ⇒ Abbc ⇒ aAbbc⇒ abbc

There are two things to notice here:

- ***Different derivations result in quite different sentential forms, but***
- ***For a context-free grammar, it really doesn't make much difference in what order we expand the variables.***

Example:

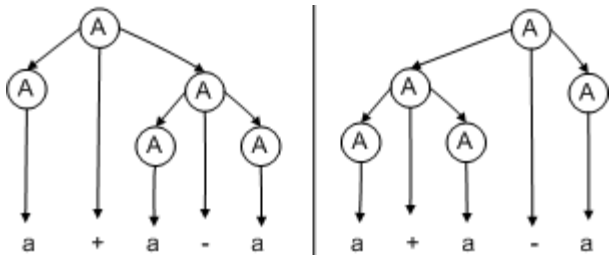
The context free grammar

A → A + A | A − A | a

is ambiguous since there are two leftmost derivations for the string a + a + a:

A → A + A	A → A + A
→ a + A	→ A + A + A (First A is replaced by A+A. Replacement of the second A would yield a similar derivation)
→ a + A + A	→ a + A + A
→ a + a + A	→ a + a + A
→ a + a + a	→ a + a + a

As another example, the grammar is ambiguous since there are two parse trees for the string a + a − a:



The language that it generates, however, is not inherently ambiguous; the following is a non-ambiguous grammar generating the same language:

$A \rightarrow A + a \mid A - a \mid a$

Chomsky Normal Form

It is convenient to assume that every context-free grammar can without loss of generality be put into a special format, called a normal form. One such format is Chomsky Normal Form. In CNF, we expect every production to be of the from A → BC or D → d, where A, B, C and D are non-terminal symbols and d is a non-lambda terminal symbols. If lambda (the empty string) is actually part of the language, then S → λ is allowed.

We will use CNF in three different places:

- A proof of a pumping lemma for CFG's.
- A proof that every language generated by a CFG can be accepted by a non-deterministic pushdown machine.
- An algorithm (dynamic programming style) for determining whether a given string is generated by a given context free grammar.

There are many steps needed to turn an arbitrary CFG into CNF. The steps are listed below:

- Get rid of Useless symbols.
- Get rid of lambda-productions.
- Get rid of Unit Productions.
- Get rid of Long Productions.
- Get rid of Terminal symbols.

The steps are completely algorithmic with step one repeatable after each of the other steps if necessary. We will do a complete example in class.

Useless Symbols –

Delete all productions containing non-terminal symbols that cannot generate terminal strings.
Delete all productions containing non-terminal symbols that cannot be reached by S.

The details of the two steps for finding useless symbols are very similar and each is a bottom-up style algorithm. To find all non-terminal symbols that generate terminal strings, we do it inductively starting with all non-terminal symbols that generate a single terminal string in one step. Call this set T. Then we iterate again looking for productions whose right sides are combinations of terminal symbols and non-terminal from T. The non-terminals on the left sides of these productions are added to T, and we repeat. This continues until T remains the same through an iteration.

To find all non-terminal symbols that can be reached by S, we do a similar thing but we start from S and check which non-terminals appear on the right side of its productions. Call this set T. Then we check which non-terminals appear on the right side of productions whose left side is a non-terminal in T. This continues until T remains the same through an iteration.

The steps need to be done in this order. For example, if you do it in the opposite order, then the grammar S →AB, S → 0, A → 0, would result in the grammar S → 0, A → 0. If we do it in the correct order, then we get the right answer, namely just S → 0.

Subsequent steps may introduce new useless symbols. Hence Useless symbol removal can be done after each of the upcoming steps to ensure that we don't waste time carrying Useless symbols forward.

Lambda-Production Removal

The basic idea is to find all non-terminals that can eventually produce a lambda (nullable non-terminals), and then take every production in the grammar and substitute lambdas for each subset of such non-terminals. We add all these new productions. We can then delete the actual lambda productions. For example A → 0N1N0 N → λ. We add A → 0N10 | 01N0 | 010, and delete N → λ

The problem with this strategy is that it must be done for all nullable non-terminals simultaneously. If not, here is a problem scenario: S → 0 | X1 | 0Y0 X → Y | λ Y → 1 | X. In this case, when we try to substitute for X → λ, we add S → 1 and Y → λ, and then we sub for Y → λ, we add S → 00 and X → λ. The trick is to calculate all nullable non-terminals at the start which include X and Y, and then sub for all simultaneously, deleting all resulting lambda productions, except perhaps for S → λ. If lambda is actually in the language, then we add a special start symbol S' → S | λ, where S remains the old start symbol.

Unit Productions

To get rid of Unit productions like A → B, we simply add A→ anything, for every production of the form B → anything, and then delete A → B. The only problem with this is that B might itself have Unit productions. Hence, like we did in lambda productions, we first calculate all Unit non-terminals that A can generate in one or more steps. Then the A → anything productions are added for all the Unit non-terminals X in the list, where X → anything, as long as anything is not a Unit production. The original Unit productions are then deleted. The Unit non-terminals that can be generated by A can be computed in a straightforward bottom-up manner similar to what we did earlier for lambda productions.

For example, consider S → A | 11 A → B | 1 B → S | 0. The Unit non-terminals that can be generated by S, A and B are A,B and B,S and A,S respectively. So we add S → 1 and S → 0; A → 11 and A → 0; and B → 1 and B → 11. Then we delete S → A, A → B and B → S.

Long Productions

Now that we have gotten rid of all length zero and length one productions, we need to concentrate on length > 2 productions. Let A → ABC, then we simply replace this with A → XC and X → AB, where X is a new non-terminal symbol. We can do this same trick inductively (recursively) for longer productions.

If we have long terminal productions or mixed terminal/non-terminal productions, then for each terminal symbol, say our alphabet is {0,1}, we add productions M → 0 and N → 1. Then all 0's and 1's are replaced by M's and N's, where M and N are new non-terminal symbols.

Pushdown automata

Definition

A pushdown automaton *is a system A* = (*Q*, Σ, Γ, *s*, Δ, *F*) *where*

- *Q is a finite set of states,*
- · Σ *is an input alphabet,*
- · Γ *is a stack alphabet,*
- · *s* ∈ *Q is an initial state,*

- · Δ *is a transition relation:*
- $\Delta \subset (Q \times (\Sigma \cup \{e\}) \times \Gamma*) \times (Q \times \Gamma*)$,
- · $F \subset Q$ *is a set of final states.*

It is useful to imagine that the automaton has a "stack" which can be filled by letters in Γ at one end, and the letters can be extracted from the stack at the same end, according to the principle "first in – last out". Consider a typical transition $((p, a, \alpha) \times (q, \beta)) \in \Delta$, where $p, q \in Q$, $a \in \Sigma \cup \{e\}$, $\alpha, \beta \in \Gamma*$. How it works: if in a state p the observed letter is a and the word at the top of the stack is α. Then the transition instructs the automaton to adopt the state q, move one position to the right along the word, and replace in the stack the word α by β. In particular, the transition $((p, a, e), (q, b))$ pushes b into the stack, while $((p, a, b), (q, e))$ "pops" b out of the stack.

Example 1

We construct a pushdown automaton accepting the language $\{a^n b^n \in \{a, b\}* \mid n \geq 0\}$.
Let Q = {s, p, f}, Σ = {a, b}, Γ = {a}, F = {s, f},
Δ= {
(1) ((s, a, e), (p, a)),
(2) ((p, a, e), (p, a)),
(3) (p, b, a), (f, e)),
(4) (f, b, a), (f, e))
}
We represent a computation with the input word *aabb* in the form of a table:

State	Input remaining	Stack	Transition
s	aabb	e	-
p	abb	a	1
p	bb	aa	2
f	b	a	3
f	e	e	4

Thus the string is accepted.

Example 2:
We construct a pushdown automaton accepting the language $\{w \in \{a, b\}* \mid w$ has the same number of a's as b's$\}$.
The idea: push into the stack the current excess of a's or b's (as the case may be).
To decide whether there is excess, there should be a way of deciding whether the stack is empty or not. This is dine by introducing a new stack letter, say c, to serve as the bottom of the stack.
Let Σ = {a, b}, Γ = {a, b, c}, Q = {s, q, f}, F = {f},
Δ= {
(1) ((s, e, e), (q, c)),
(2) ((q, a, c), (q, ac)),
(3) ((q, a, a), (q, aa)),
(4) ((q, a, b), (q, e)),
(5) ((q, b, c), (q, bc)),
(6) ((b, b, b), (q, bb)),
(7) ((q, b, a), (q, e)),
(8) ((q, e, c), (f, e))}.

State	Input remaining	Stack	Transition
s	*aabbba*	*e*	—
q	*aabbba*	*c*	1
q	*abbba*	*ac*	2
q	*bbba*	*aac*	2
q	*bba*	*ac*	7
q	*ba*	*c*	7
q	*a*	*bc*	5
q	*e*	*c*	4
f	*e*	*e*	8

We see that the string aabbba is accepted by the grammar.

From grammars to automata

Theorem. There exists an algorithm which for any context-free grammar G constructs a pushdown automaton A such that L(G) = L(A).

Sketch of a proof: Let G = (Σ,NT,R,S) be a context-free grammar. Define the automaton A as ({s, f }, Σ, Γ = Σ ∪ N T , Δ, {f }), where

Δ={
(1) ((s, e, e), (f, S)),
(2) ((f,e,C),(f,w)) for each rule C → w in R, (3) ((f, a, a), (f, e)) for each a ∈ Σ}.

Informally, the automaton A simulates a "leftmost derivation" of the input word. A transformation of the type (2) simulates one step of the derivation, transformation of the type (3), maybe repeated several times, prepares the stack for another step of the derivation.

Example 3:
Consider the language of even-length palindromes: L = {ww^R| w ∈ {a, b}∗}. The grammar G generating L is determined by the following set of rules:

{S → aSa, S → bSb, S → e}.

According to the theorem, the transitions of a pushdown automaton accepting L can be obtained from this list of rules as follows:
(1) ((s, e, e), (f, S)),
(2) ((f, e, S), (f, aSa)),
(3) ((f, e, S), (f, bSb)),
(4) ((f, e, S), (f, e)),
(5) ((f, a, a), (f, e)),
(6) ((f, b, b), (f, e)).
Apply A to the input abba:

State	Input remaining	Stack	Transition
s	$abba$	e	—
f	$abba$	S	1
f	$abba$	aSa	2
f	bba	Sa	5
f	bba	$bSba$	3
f	ba	Sba	6
f	ba	ba	4
f	a	a	6
f	e	e	5

Turing Machine

A Turing Machine (TM) is an abstract, mathematical model that describes what can and cannot be computed. A Turing Machine consists of a tape of infinite length, on which input is provided as a finite sequence of symbols. A head reads the input tape. The Turing Machine starts at "start state" S0. On reading an input symbol it optionally replaces it with another symbol, changes its internal state and moves one cell to the right or left.

Notation for the Turing Machine:

TM = {Q, Σ, Γ, δ, q0, h}
Q = a set of TM states
Σ= set of symbols
Γ = a set of tape symbols
q0 = the start state
h =halting state
δ = transition functions

Some of the characteristics of a Turing machine are:

1. The symbols can be both read from the tape and written on it.
2. The TM head can move in either direction – Left or Right.
3. The tape is of infinite length
4. The special states, Halting states and Accepting states, take immediate effect.

Design a TM that erases all non-blank symbols on the tape, where the sequence of non-blank symbols does not contain any blank symbol # in between:

TM = {Q, Σ, Γ, δ, q0, h}
Q={q0, h}
Σ={a, b}
Γ={a, b, #}
q0 is initial state

q: states	σ: input symbol	δ (q, σ)
q0	a	{qo, #, L}
q0	b	{qo, #, L}
q0	#	{h, #, N}
h	#	ACCEPT

Design a TM that recognizes the language of all strings of even length over alphabet {a, b}

TM = {Q, Σ, Γ, δ, q0, h}
Q={q0, q1, h}
Σ={a, b}
Γ={a, b, #}
q0 is initial state

q: states	a	b	#
q0	(q1, a, L)	(q1, b, L)	{qo, #, L}
q1	(q0, a, L)	(q0, b, L)	*
h	*	*	ACCEPT

Design a TM that recognizes the language of all strings, which contains 'aba' as a substring.

TM = {Q, Σ, Γ, δ, q0, h}
Q={q0, q1,q2, h}
Σ={a, b}
Γ={a, b, #}
q0 is initial state

q: states	a	b	#
q0	(q1, a, L)	(q0, b, L)	*
q1	(q1, a, L)	(q2, b, L)	*
q2	(h, a, L)	(q0, b, L)	(h, #, N)
h	*	*	ACCEPT

Design a TM which compute the function f(m) = m + 1 for each m that belongs to the set of natural numbers.
Given the function f(m) = m + 1. Here we represent the input m on the tape by a number of I's on the tape.
For, example, if m =1, input will be #I# and If m = 2, input will be #II#
TM = {Q, Σ, Γ, δ, q0, h}
Q={q0, h}
Γ={I, #}
q0 is initial state

q: states	σ=0	σ=#
q0	(q1, I, R)	(q1, #, R)
q1	*	(h, I, R)
h	*	*

Design a TM that replaces every 0 with 1 and 1 with 0 in a binary string.
TM = {Q, Σ, Γ, δ, q0, h}
Q={q0, q1,h}
Σ={0, 1}
Γ={0, 1, #}
q0 is initial state

q: states	0	1	#
q0	(q0, 1, L)	(q0, 0, L)	(q1, #, R)
q1	(q1, 1, R)	(q1, 0, R)	(h, #, N)
h	*	*	ACCEPT

Given a string of 1s on a tape (followed by an infinite number of 0s), add one more 1 at the end of the string.
Input: #111100000000…….
Output: #1111100000000…….
Initially the TM is in Start state S0. Move right as long as the input symbol is 1. When a 0 is encountered, replace it with 1 and halt.
Transitions:
(S0, 1)→ (S0, 1, R)
(S0, 0)→ (S0, 1, STOP)

P AND NP CLASS PROBLEMS

Up to now we were considering on the problems that can be solved by algorithms in worst-case polynomial time. There are many problems and it is not necessary that all the problems have the apparent solution. This concept, somehow, can be applied in solving the problem using the computers. The computer can solve: some problems in limited time e.g. sorting, some problems requires unmanageable amount of time e.g. Hamiltonian cycles, and some problems cannot be solved e.g. Halting Problem. In this section we concentrate on the specific class of problems called NP complete problems.

Tractable and Intractable Problems
We call problems as tractable or easy, if the problem can be solved using polynomial time algorithms. The problems that cannot be solved in polynomial time but requires super-polynomial 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.

P and NP classes and NP completeness

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 (more on this later). 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.

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.

Complexity Class P

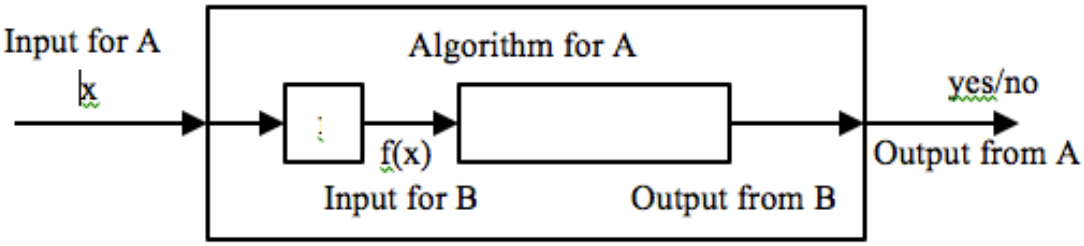
Complexity class P is the set of concrete decision problems that are polynomial time solvable by deterministic algorithm. If we have an abstract decision problem A with instance set I mapping the set {0,1}, an encoding e: I→{0,1}* is used to denote the concrete decision problem e(A). We have the solutions to both the abstract problem instance i∈I and concrete problem instance e(i) ∈{0,1}* as A(i)∈{0,1}. It is important to understand that the encoding mechanism does greatly vary the running time of the algorithm for e.g. take some algorithm that runs in O(n) time, where the n is size of the input. Say if the input is just a natural number k, then its unary encoding makes the size of the input as k bits as k number of 1's and hence the order of the algorithm's running time is O(k). In other situation if we encode the natural number k as binary encoding then we can represent the number k with just logk bits (try to represent with 0 and 1only) here the algorithm runs in O(n) time. We can notice that if n = logk then O(k) becomes O(2n) with unary encoding. However in our discussion we try to discard the encoding like unary such that there is not much difference in complexity.

We define polynomial time computable function f:{0,1}*→{0,1}* with respect to some polynomial time algorithm PA such that given any input x ∈{0,1}*, results in output f(x).

For some set I of problem instances two encoding e1 and e2 are polynomially related if there are two polynomial time computable functions f and g such that for any i ∈I, both f(e1(i)) = e2(i) and g(e2(i)) = e1(i) are true i.e. both the encoding should computed from one encoding to another encoding in polynomial time by some algorithm.

Polynomial time reduction

Given two decision problems A and B, a polynomial time reduction from A to B is a polynomial time function f that transforms the instances of A into instances of B such that the output of algorithm for the problem A on input instance x must be same as the output of the algorithm for the problem B on input instance f(x) as shown in the figure below. If there is polynomial time computable function f such that it is possible to reduce A to B, then it is denoted as A ≤p B. The function f described above is called reduction function and the algorithm for computing f is called reduction algorithm.



Complexity Class NP

NP is the set of decision problems solvable by nondeterministic algorithms in polynomial time. When we have a problem, it is generally much easier to verify that a given value is solution to the problem rather than calculating the solution of the problem. Using the above idea we say the problem is in class NP (nondeterministic polynomial time) if there is an algorithm for the problem that verifies the problem in polynomial time.

PREVIOUS YEAR QUESTION SOLUTIONS

CSC-251-2067 (FIRST BATCH)

Attempt all the questions

Group A

1. **Define Finite Automata with ε moves. Is ε NFA has more computation power than DFA?**

   An ε NFA can be represented as:
   A=(Q, ∑, δ, q_0, F) where,
   Q=a finite set of states.
   ∑=a set of input symbols
   δ=a function that takes as arguments:
   i. A state in Q and
   ii. A member of ∑ U{ε}, that is either an input symbol or the symbol ε.

   q_0=an initial state that ϵ Q

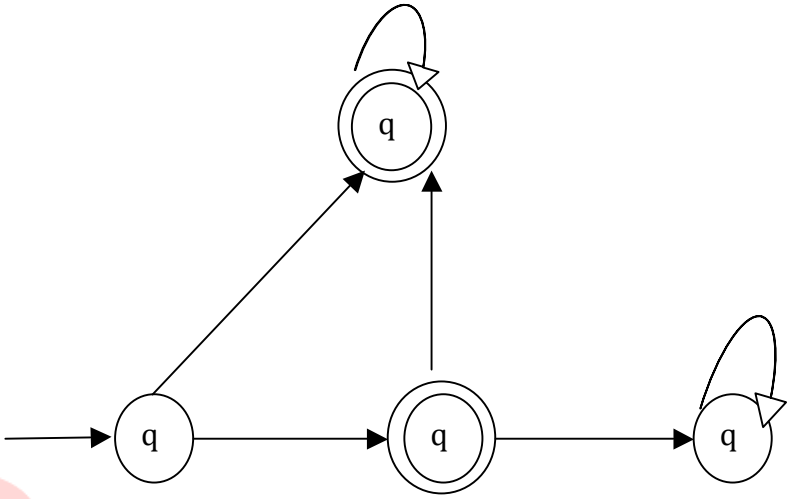
   F= a set of final states or accepting state.

   It appears that ε-NFA is more powerful than DFA but this is not the case. Because, say we have two machine A and B, then
   B is less powerful than A if and only if:
   - Some A can recognize every language a B can recognize.

- There is some language that can be recognized by an A but not by any B.

Since, every language accepted by ε-NFA is also accepted by DFA, they are of equal powers.

2. **Give the DFA accepting the strings over {a, b} such that each string does not start with ab.**



3. **Give a regular expression for the following languages:**

a) L = {S|S ϵ {a, b}* and S starts with aa or b and does not contain substring bb.
Ans: (aa+b)(a*ab)*

b) L = {S|S ϵ {0, 1}* and 0 occurs in pairs if any and ends with 1.
Ans: 1*(00)*1*(00)*1

4. **Convert following regular grammar into Finite Automata**

S→ aaB | aB | ε
B→ bb | bS | aBB
See note

5. **Convert following grammar into an equivalent PDA.**

S→AAC A→aAb | ε C→aC | b | ab
See Note

6. **What is a multi track Turing Machine? How it differs with single-track machine?**

A **Multi-track Turing Machine i**s a specific type of Multi-tape Turing Machine. In a standard n-tape Turing machine, n heads move independently along n tracks. In an n-track Turing machine, one head reads and writes on all tracks simultaneously. A tape position in an n-track Turing Machine contains n symbols from the tape alphabet. It is equivalent to the standard Turing machine and therefore accepts precisely the recursively enumerable languages.

Formal definition

A multitape Turing machine can be formally defined as a 6-tuple, where

- *Q* is a finite set of states
- Σ is a finite set of symbols called the *tape alphabet*
- q0 is the *initial state*

- $F \subseteq Q$ is the set of *final* or *accepting states*.
- $\delta \subseteq (Q \setminus A \times \Sigma) \times (Q \times \Sigma \times d)$ is a relation on states and symbols called the *transition relation*.
- $\Gamma \in Q$ is the set of tape symbols

A single track TM machine is a TM with single tape with only one track.

7. Construct a Turing Machine that accepts the language of palindrome over {a, b}* with each string of odd length.

See Chapter Turing Machine examples

8. What is an algorithm? Explain on the basis of Church Hypothesis.

According to Church, "No computational procedure will be considered an algorithm unless it can be represented by a Turing Machine."

In other word, an algorithm is a module that is accepted by some Turing machine. This connection between the informal notion of algorithm and the precise definition is known as the Church-Turing thesis. More precisely, Turing proposed to adopt the Turing machine that halts on all inputs as the precise formal notion corresponding to our intuitive notion of an "**algorithm**".

Note that the Church-Turing thesis is not a theorem, but a "thesis", as it asserts that a certain informal concept (algorithm) corresponds to a certain mathematical object (Turing machine). Not being a mathematical statement, the Church-Turing thesis cannot be proved! What are the implications of this fact?
Well, it is in principle possible that the Church-Turing thesis can be disproved.

Group B

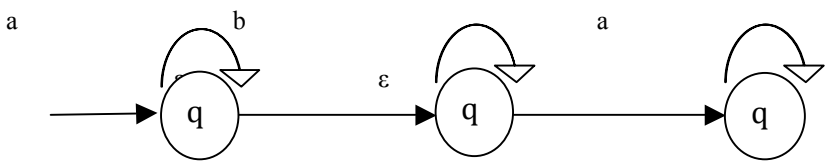
9. How a ε-NFA can be converted into NFA and DFA? Explain with a suitable example.

Conversion of ε-NFA to corresponding DFA:

In order to convert ε-NFA to corresponding DFA, following steps should be taken into consideration:

i. The ε-closure of starting state should be determined & union of ε-closure of each transitions for every symbols of $\sum$ should be calculated.

ii. The same process should be repeated till new states appear.



Firstly,

ε-closure of starting state is

ε-closure(q_0)=$\{q_0,q_1,q_2\}$ = ε(q_0)

Similarly, other closures as:

$\varepsilon(q_1)=\{ q_1,q_2\}$

$\varepsilon(q_2)=\{q_2\}$

Then, $\delta_c(\{q_0,q_1,q_2\}, a) = \varepsilon(q_0) \cup \varphi \cup \varepsilon(q_2)$

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

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

$\delta_c(\{q_0,q_1,q_2\}, b) = \varphi \cup \varepsilon(q_1) \cup \varphi = \{ q_1,q_2\}$

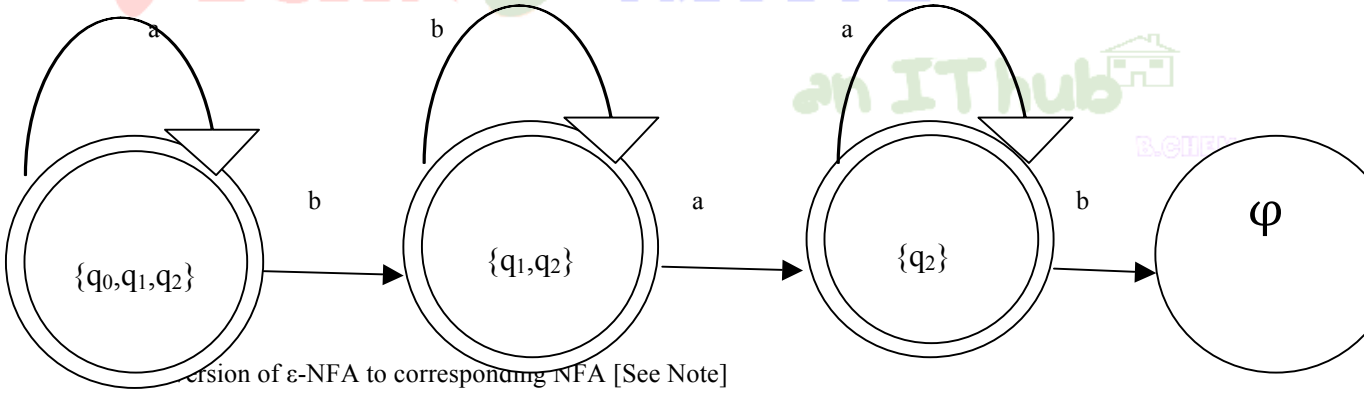
$\delta_c(q_1,q_2\}, a) = \varphi \cup \varepsilon(q_2) = \{q_2\}$

$\delta_c(q_1,q_2\}, b) = \varepsilon(q_1) \cup \varphi = \{ q_1,q_2\}$

$\delta_c(q_2\}, a) = \varepsilon(q_2) = \{q_2\}$

$\delta_c(q_2\}, b) = \varphi$

Then,



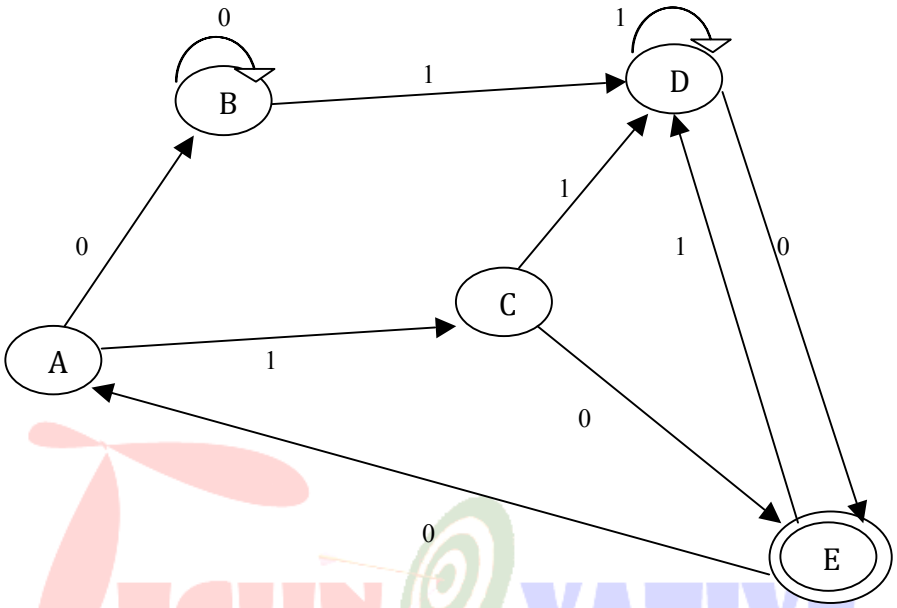
ersion of ε-NFA to corresponding NFA [See Note]

10. Find the minimum state DFA equivalent to the following DFA.

State	0	1
→A	B	C
B	B	D
C	E	D
D	E	D

*E	A	D

We have,



Here, we don't have any unreachable state. Then, separating final and non-final states as

State	0	1	State	0	1
→A	B	C	E*	A	D
B	B	D			
C	E	D			
D	E	D			

Then eliminating equivalent states as:

State	0	1	State	0	1
A	B	C	C	E	C
B	B	C			

Again,

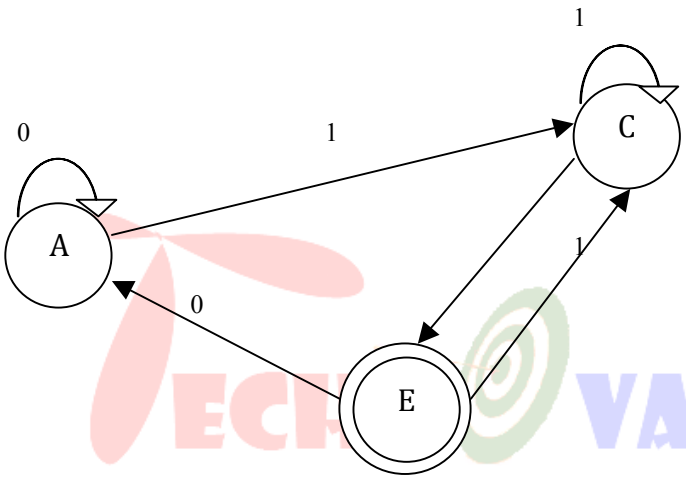
State	0	1	State	0	1
A	A	C	E*	A	C

C	E	C

Now, merging these two tables as

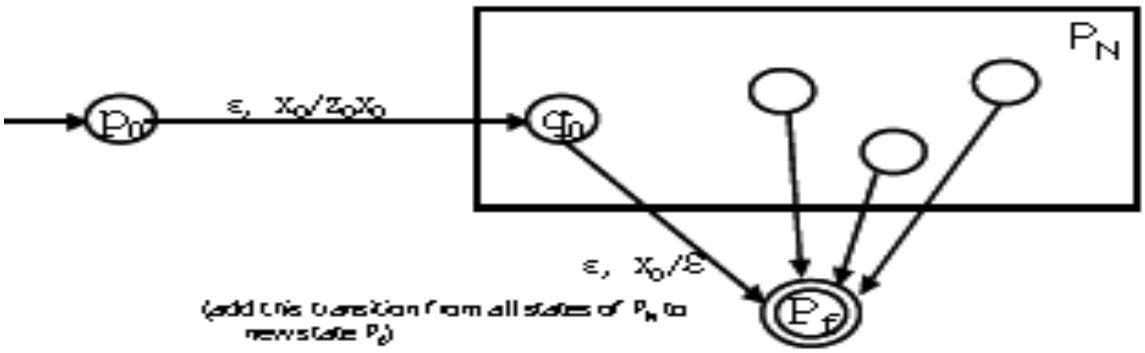
State	0	1
A	A	C
C	E	C
E*	A	C

Now, final diagram is



11. **Show that a language L is accepted by some DFA if and only if L is accepted by some NFA.**
    See the chapters Finite Automata

12. **Define the language of PDA that accepts by Final State. Explain, how a PDA accepting by empty stack can be converted into a PDA by final state.**
    From Empty Stack to Final State:
    Theorem: If L = N(PN) for some PDA PN= (Q, Σ, Γ , δN, q0, Z0), then there is a PDA PF such that L = (PF)



The method of conversion is given in figure.
We use a new symbol X0, which must be not symbol of Γ to denote the stack start symbol for PF. Also add a new start state p0 and final state pf for PF. Let PF = (Q∪{p0, pf}, Σ, Γ∪{X0}, δF, p0, X0, {Pf}), where δF is defined by

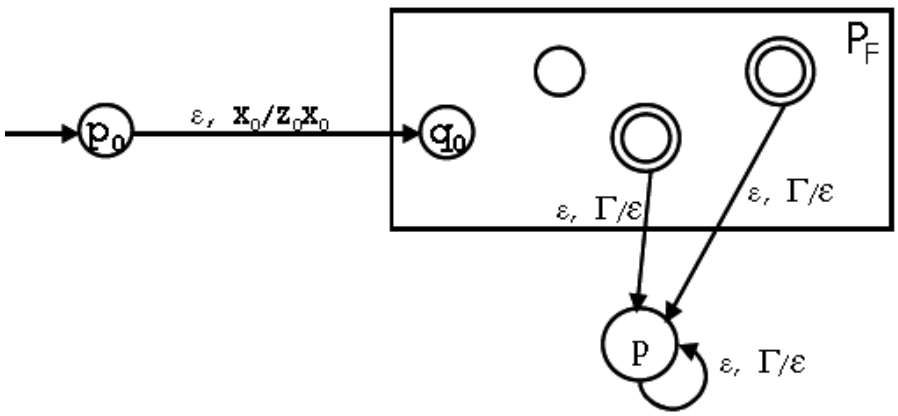
δF(p0,ε,X0) = {(q0, Z0 X0)} to push X0 to the bottom of the stack

δF(q, a, y) = δN(q, a, y) a ∈ Σ or a =ε (and y ∈ Γ, same for both PN and PF.

δF(q, ε, X0) = {(Pf, ε,)} to accept the string by moving to final state.

The moves of PF to accept a string w can be written like:
(p0, w, X0) |-PF (p0, w, Z0X0) |-*PF (q, ε,, X0) |- (Pf , ε,, ε,)

From Final State to Empty Stack:
Theorem: If L = L(PF) for some PDA PF= (Q, Σ, Γ, δF, q0, Z0, F), then there is a PDA PN such that L = N(PN)



The method of conversion is given in figure.
To avoid PF accidentally empting its stack, initially change the stack start content from Z0 to Z0X0. Also add a new start state p0 and final state p for PN. Let PN =
(Q∪{p0, p}, Σ, Γ∪{X0}, δN, p0, X0)
where δN is defined by:

$\delta N(p0, \varepsilon, X0) = \{(q0, Z0X0)\}$ to change the stack content initially
$\delta N(q, a, y) = \delta F(q, a, y)$, $a \in \Sigma$ or $a = \varepsilon$ and $y \in \Gamma$, same for both
$\delta N(q, \varepsilon, y) = \{(p, \varepsilon)\}$, $q \in F$, $y \in \Gamma$ or $y = X0$, same for both
$\delta N(p, \varepsilon, y) = \{(p, \varepsilon)\}$, $y \in \Gamma$ or $y = X0$, to pop the remaining stack contents.
The moves of PN to accept a string w can be written like:
$(p0, w, X0) \vdash_{PN} (q0, w, Z0X0) \vdash^{*}_{PN} (q, \varepsilon, X0) \vdash (p, \varepsilon, \varepsilon)$

13. **Explain about multi tape TM. Show that every language accepted by a multi-tape Turing Machine is also accepted by one tape Turing Machine.**
    For solution see the respective chapters

14. **Write short notes on:**
    (a) Decidable vs. Un-decidable problems.
    (b) Unrestricted Grammar.
    (c) NP-Completeness.
    (d) CNF-SAT Problem.

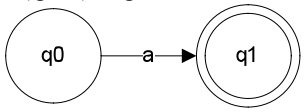
    For solution see the respective chapters

TOC (2067) Second Batch

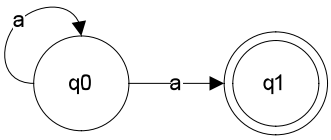
Group A

1. **What is DFA? How it differ with a NFA? Explain.**
   A DFA is defined as a five tuples $(Q, \sum, \delta, q_o, F)$ where,
   Q = finite set of states
   $\sum$ = finite set of input symbols
   δ = transition function that maps $Q^{*}\sum \rightarrow Q$
   q_o = starting state, $q_o \epsilon Q$
   F = final state (accepting state) F is a subset of Q

   DFA differs with a NFA by the type of value that transition function returns. In DFA, the transition function takes a state in Q and an input symbol in $\sum$ as arguments and returns a set of states. It means that the transition function takes DFA from one state to another state while it takes NFA from one state to several other states.
   For DFA,
   $\delta (q0, a) = q1$

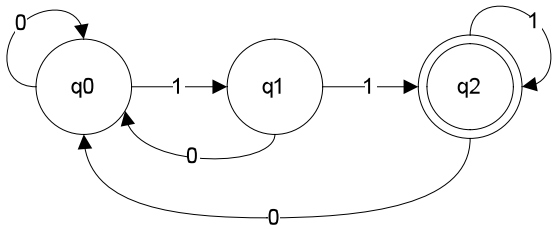


   For NFA,
   $\delta (q0, a) = q0$
   $\delta (q0, a) = q1$

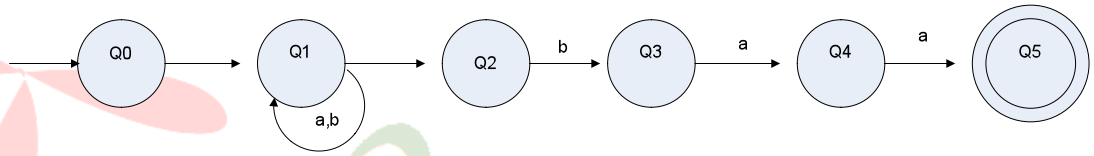


Here, from the above figure in the DFA the state q1 is reached when the input symbol 'a' is given to the state q0. And in NFA the state q0 and q1 is reached when the input symbol 'a' is given to the state q0. It shows that the DFA can have only one state as output for any input symbol and the NFA can have more than one output for any input symbol.

2. Give the DFA for language of strings over {0,1} in which each strings end with 11.



3. For a regular expression (a+b)*baa, construct ε-NFA.



4. Define the term parse tree, regular grammar, sentential form and ambiguous grammar.

Parse tree

Parse tree is the tree representation of production defined by a grammar. It is very important for syntax analysis of any programming languages. The parse tree is the tree with the following condition:

- Each interior node of a parse tree are variables
- Each leaf node is labeled with E or a terminal strings. If labeled with t then it is only child of its parents.

Regular grammar

A regular grammar is a CFG which may be left linear or right linear. A grammar in which all productions are of the form A→wB or A→w where, A, B ϵ V and w ϵ T* is called right linear grammar.
If all the productions are of the form A→Bw or A→w where, A, B ϵ V and w ϵ T* is left linear grammar.
Regular grammar always represents a language that is accepted by finite automata which is called regular language.
S → 0S | 1S | 0 (right linear grammar)
S → S0 | S1 | 0 (left linear grammar)

Sentential form

The derivation from the start symbol produce strings that have a special rule. We call these the "sentential form". That is, G = (V, T, P, S) is a CFG, then any string α in (V U T)* such that S *→ α is a sentential form.

Ambiguous grammar

A CFG is called ambiguous if for at least one word in the language that it generates there are two or more possible derivations of the word that correspond to different syntax trees.

5. Give the formal definition of NPDA. How it differs with DPDA? Explain.

A *nondeterministic pushdown automaton (npda)* is basically an nfa with a stack added to it.
A *nondeterministic pushdown automaton* or *npda* is a 7-tuple

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

where

- Q is a finite set of *states,*
- $\sum$ is a the *input alphabet,*
- Γ is the *stack alphabet,*
- δ is a *transition function,*
- $q_0 \in Q$ is the *initial state,*
- $z \in \Gamma$ is the *stack start symbol,* and
- $F \subseteq Q$ is a set of *final states.*

The NPDA differs with DPDA by the following points:
A PDA is deterministic if there is never a choice of move in any situation. It means that if δ (q, a, X) contains more than one pair, then surely the PDA is nondeterministic because we can choose among these pairs when deciding on the next move. However, even if δ (q, a, X) is always a singleton, we could still have a choice between using a real input symbol, or making a move on ϵ. Hence, PDA is deterministic if and only if the following conditions are met:

— δ (q, a, X) has at most one member for any q in Q, a in ∑ or a = ϵ, and X in Γ.

— If δ (q, a, X) is nonempty, for some a in ∑, then δ (q, ϵ, X) must be empty.

6. **Construct a Turing Machine that accepts a language of string over (a, b) with each string of even length. Show how it accepts string abab.**

Design a TM that recognizes the language of all strings of even length over alphabet {a, b}
TM = {Q, Σ, Γ, δ, q0, h}
Q={q0, q1, h}
Σ={a, b}
Γ={a, b, #}
q0 is initial state

q: states	a	b	#
q0	(q1, a, L)	(q1, b, L)	{qo, #, L}
q1	(q0, a, L)	(q0, b, L)	*
h	*	*	ACCEPT

7. **Give the formal definition of Turing Machine. How it differs from PDA?**

A Turing Machine has a 7-tuple M = (Q, ∑, Γ, δ, q_o, B, F) where
Q : The finite set of states of the finite control
∑: The finite set of input symbols
Γ : The complete set of tape symbols
δ : The transition function defined by Q*Γ→Q*Γ*(R,L,S) where R,L,S is the direction of head-left or right or stationery i.e. δ (q, X) = δ (p, Y, D)
q_o = the start state
B = the blank symbol BϵT
F = the set of final accepting state
The Turing Machine differs from PDA from the following points:
— The cell of the tape of PDA are not read/scanned but are never changed or written into whereas the cells of the tape of TM are written also.
— The tape head of a PDA always moves from left to right however the tape head TM can move in both the direction.

8. **Explain about the Unrestricted Grammar.**

An unrestricted grammar is used to describe he languages that are not context free. The language described by unrestricted grammar can be accepted by TM. The example shows the unrestricted grammar generating $\{a^n b^n c^n \mid n>1\}$

S → FS1
S1 → ABCS1
S1 → ABC
BA → AB
CA → AC
CB → BC
FA → a
aA →aa
aB → ab
bB →bb
bC →bc
cC → cc

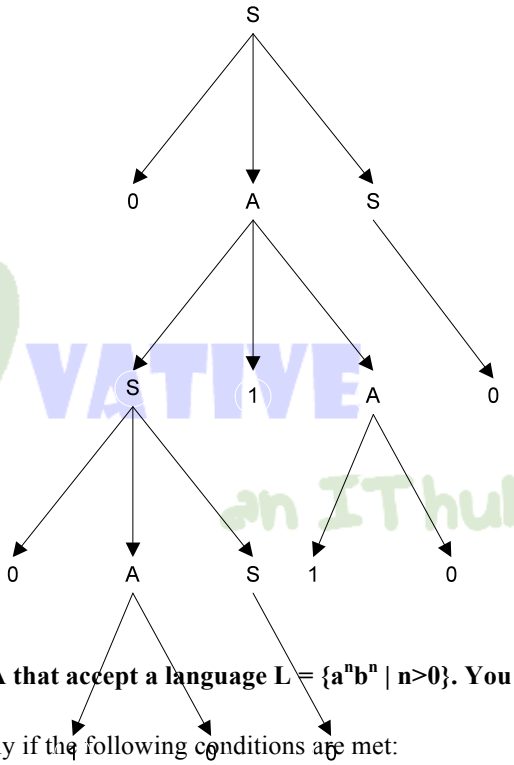
S = FS1
= FABCS1
= FABCABC
= aBCABC
= aBACBC
= aABCBC
= aaBCBC
= aabCBC
= aabBCC
= aabbCC
= aabbcC
= aabbcc

9. **Show that a language L is accepted by some DFA if and only if L is accepted by some NFA.**

A language L is acceptance of some NFA if L is accepted by DFA (NFA=DFA).
Proof:
Let $D = (Q_D, \sum, \delta_D, q_o, F_D)$ be DFA, define a NFA $N = (Q_N, \sum, \delta_N, q_o, F_N)$ equivalently where, $Q_D=Q_N$, $F_D=F_N$
δ_N is defined by if $\delta_D (q,a) = p$ then, $\delta_N = \{p\}$.
We have to show that, if $\delta_D (q_o, w) = p$ then $\delta_N (q_o, w) = \{p\}$.
Using Induction:
1. Basis:
Let $|w| = 0$ i.e. $w = \epsilon$
$\delta_D\hat{} (q_o, w) = \delta_D\hat{} (q_o, \epsilon) = q_o$
$\delta_N\hat{} (q_o, w) = \delta_N\hat{} (q_o, \epsilon) = \{q_o\}$
Since, $q_o' = q_o$
2. Inductive:
Let $|w|=n+1$ & $w=xa$ where x is a substring of w without last symbol then $|x|=n$ $|a|=1$
The inductive hypothesis,
if $\delta_D (q_o, xa) = p$ then $\delta_N\hat{} (q_o, xa) = \{p\}$
Now, $\delta_D\hat{} (q_o, xa) = \delta_D (\delta_D\hat{} (q_o, x), a) = \delta_D (p, a) = r$ (say)
$\delta_N\hat{} (q_o, xa) = \delta_N (\delta_D\hat{} (q_o, x), a) = \delta_N (\{p\}, a) = r$ (say)
Therefore, $\delta_D\hat{} (q_o, xa) = \delta_D\hat{} (q_o, xa) = r$.

10. **State and prove pumping lemma for regular language. Show by example how it can be used to prove a language is not a regular.**
Statement: Let L be regular language then there exist an integer constant n so that for any xϵL with |x|≥n there are string u,v and w such that x=uvw |uv|≥n, |v|>0 then uvkwϵL A k≥0.
For proof see the chapter CFG and examples

11. **Define Context Free Grammar. Given the following CFG**
S→0AS | 0, A→S1A | SS | 10
For the string 001001100, give the left most and right most derivation and also construct a parse tree.
A context free grammar is defined by 4 tuples (V, T, P, S) where,
V = set of vertices
T = set of terminal symbols
P = set of rules or production
S = set of start symbols SϵV
For the left most derivation
S → 0AS
→ 0S1AS
→ 00AS1AS
→ 0010S1AS
→ 001001AS
→ 00100110S
→ 001001100
For the right most derivation
S → 0AS
→ 0A0
→ 0S1A0
→ 0S1100
→ 00AS1100
→ 00A01100
→ 001001100



12. **Define deterministic PDA. Design a PDA that accept a language L = {$a^n b^n$ | n>0}. You may accept either by empty stack or by final state.**
The PDA is a deterministic PDA if and only if the following conditions are met:
— δ (q, a, X) has at most one member for any q in Q, a in ∑ or a = ϵ, and X in Γ.
— If δ (q, a, X) is nonempty, for some a in ∑, then δ (q, ϵ, X) must be empty.
For poof see the chapter PDA

13. **Describe a Universal Turing Machine and its operations. What types of languages are accepted by Universal TM?**
A **universal Turing machine** (**UTM**) is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of the machine to be simulated as well as the input thereof from its own tape.
The operation of a Turing machine proceeds as follows:

1. The Turing machine reads the tape symbol that is under the Turing machine's tape head. This symbol is referred to as the current symbol.

2. The Turing machine uses its transition function to map the current state and current symbol to the following: the next state, the next symbol and the movement for the tape head. If the transition function is not defined for the current state and current symbol, then the Turing machine crashes.
3. The Turing machine changes its state to the next state, which was returned by the transition function.
4. The Turing machine overwrites the current symbol on the tape with the next symbol, which was returned by the transition function.
5. The Turing machine moves its tape head one symbol to the left or to the right, or does not move the tape head, depending on the value of the 'movement' that is returned by the transition function.
6. If the Turing machine's state is a halt state, then the Turing machine halts. Otherwise, repeat sub-step #1.

The type of language accepted by Universal TM is the class of Type 0 languages.

14. Explain about the Chomsky Hierarchy of the language.

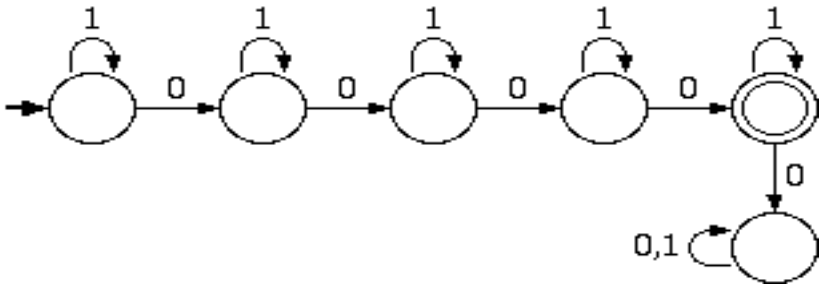
Noam Chomsky defined four classes of grammars, which define four classes of languages. These are arranged in a hierarchy: each class includes the one below it. The hierarchy is *strict*, meaning that there exist languages of each type that do not belong to the next higher type.

- Type 0: *recursively enumerable* languages (unrestricted grammars)
- Type 1: *context-sensitive* languages (context-sensitive grammars)
- Type 2: *context-free* languages (context-free grammars)
- Type 3: *regular* languages (right-linear and left-linear grammars)

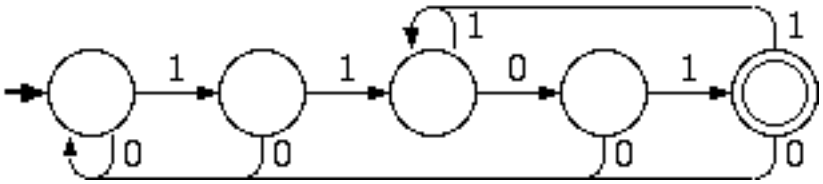
QUESTION ANSWERS

Construct DFA of the following problems

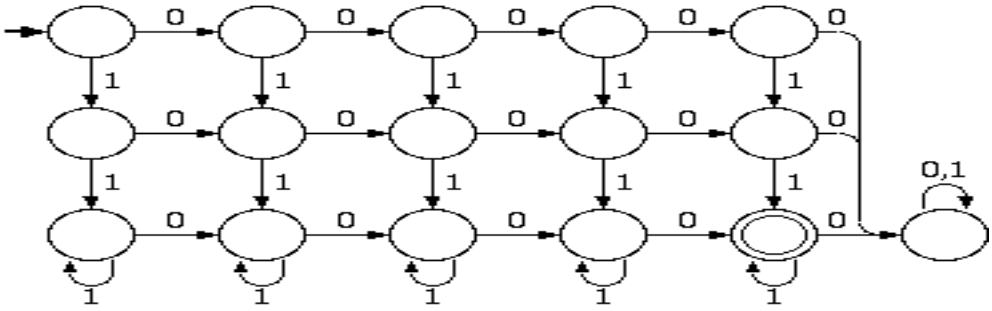
All strings that contain exactly 4 0s



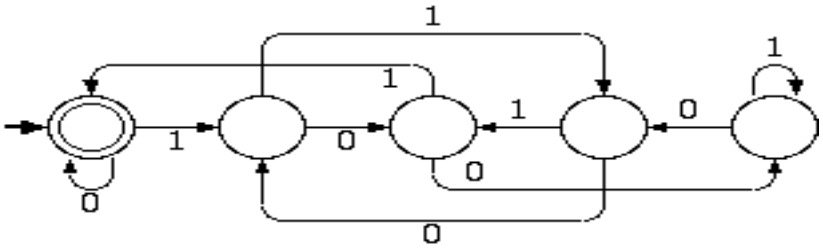
All strings ending in 1101



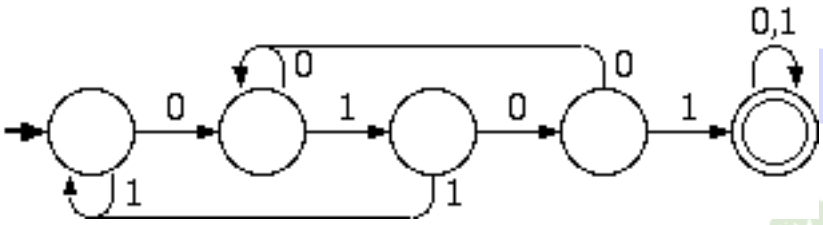
All strings containing exactly 4 0s and at least 2 1s



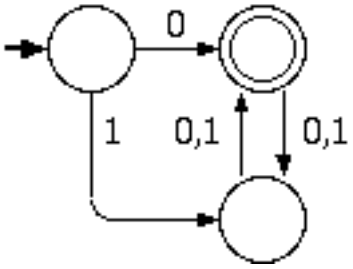
All strings whose binary interpretation is divisible by 5



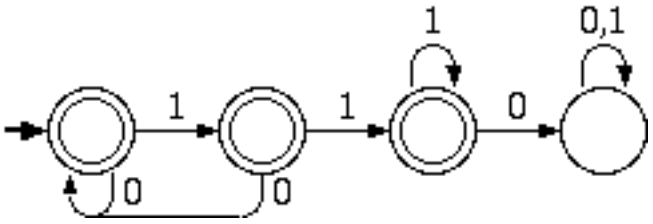
All strings that contain the substring 0101



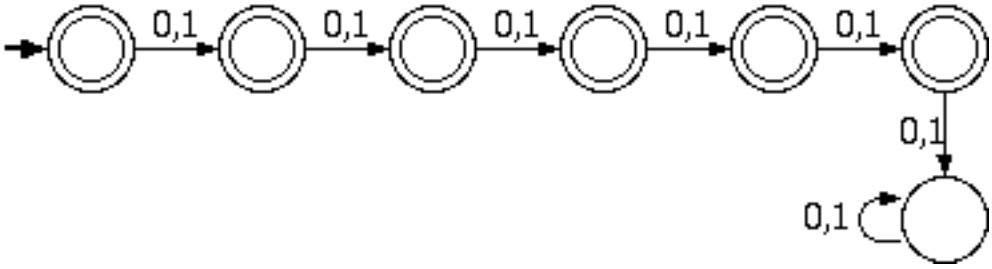
All strings that start with 0 and have odd length or start with 1 and have even length



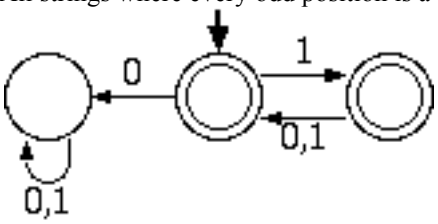
All strings that don't contain the substring 110



All strings of length at most 5

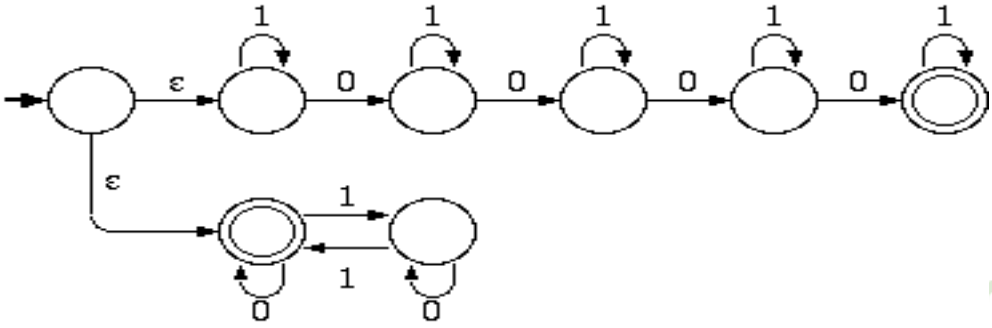


All strings where every odd position is a 1

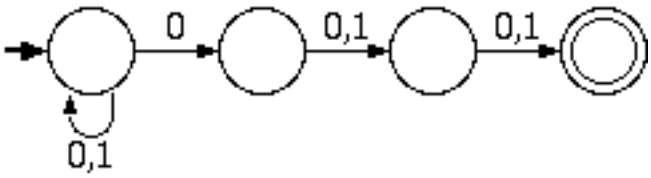


Construct NFA of the following problems

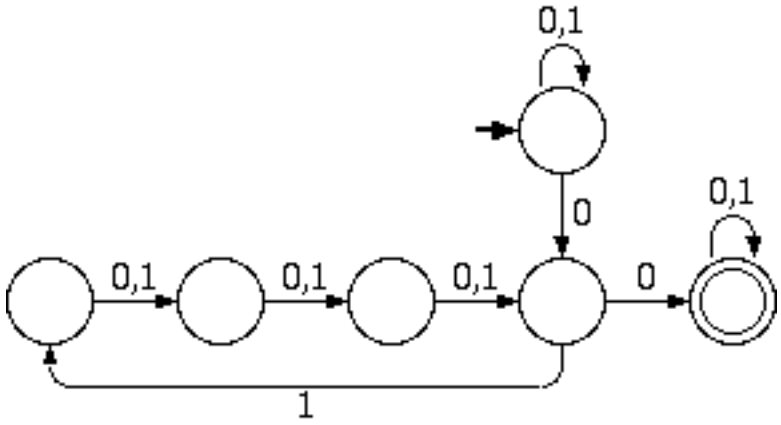
All strings containing exactly 4 0s or an even number of 1s



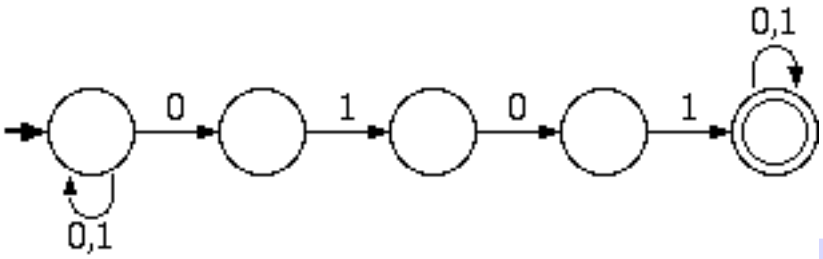
All strings such that the third symbol from the right end is a 0



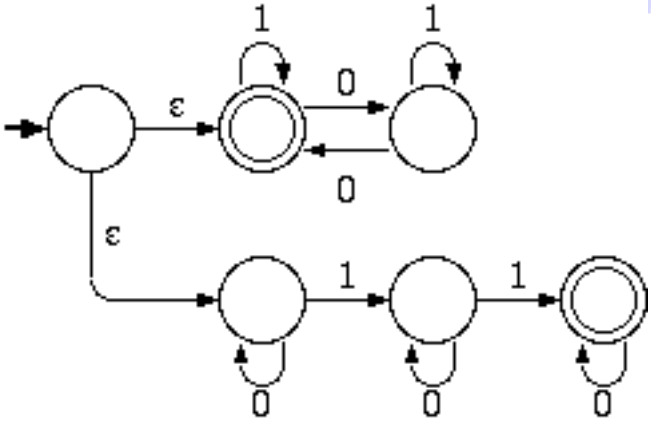
All strings such that some two zeros are separated by a string whose length is 4i for some i>=0



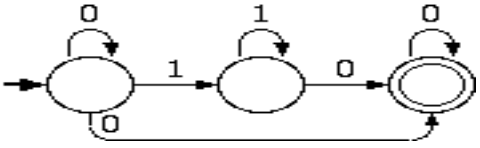
All strings that contain the substring 0101



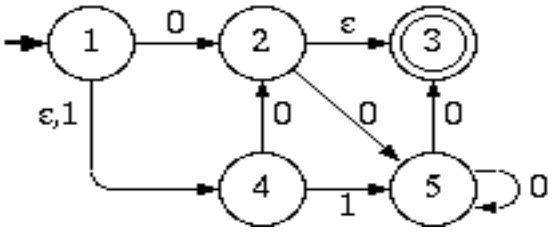
All strings that contains an even number of 0s or exactly two 1s



The language 0*1*0*0

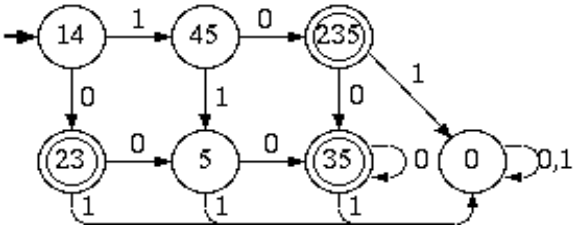


Consider the Finite Automaton below. Construct the smallest Deterministic Finite Automaton, which accepts the same language. Finally, draw a regular expression that represents the language accepted by your machine and draw a Regular Grammar that generates it.

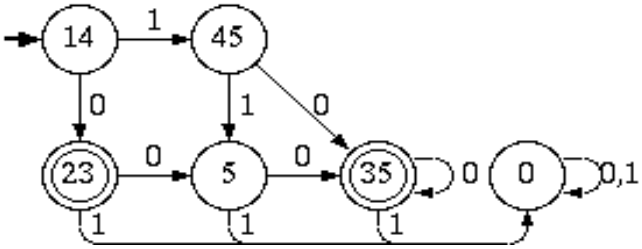


Solution

a. Converting to a DFA



b. Minimizing DFA



c. Converting to a regular expression

0 + (00 + 1 + 11) 00*

d. Converting to a regular grammar using the NFA

A -> 0 | 0B | 1C | 1D
B -> 0D
C -> 1D
D -> 0 | 0D

Using the DFA:
S -> 0A | 1B
A -> 0C | e
B -> 0D | 1C
C -> 0D

Using the regular expression:
S -> 0 | 110A | 000A | 10A
A -> 0 | 0A | e

Give CGF of the following languages

a. The language {w | w starts and ends with the same symbol}
S -> 0A0 | 1A1
A -> 0A | 1A | e

b. The language {w | the length of w is odd}
S -> 0A | 1
A -> 0S | 1S | e

c. The language {w | the length of w is odd and its middle symbol is a zero}
S -> 0 | 0S0 | 0S1 | 1S0 | 1S1

d. $\{0^n 1^n \mid n > 0\} \cup \{0^n 1^{2n} \mid n > 0\}$
S -> 0A1 | 0B11

A -> 0A1 | e
B -> 0B11 | e

e. $\{0^i 1^j 2^k \mid i \neq j \text{ or } j \neq k\}$

S -> AC | BC | DE | DF
A -> 0 | 0A | 0A1
B -> 1 | B1 | 0B1
C -> 2 | 2C
D -> 0 | 0D
E -> 1 | 1E | 1E2
F -> 2 | F2 | 1F2

f. Binary strings with twice as many ones as zeros

S -> e | 0S1S1S | 1S0S1S | 1S1S0S

Explain why the grammar below that generates strings with an equal number of 0's and 1's is ambiguous.

S →0A | 1B

A →0AA | 1S | 1

B→1BB | 0S | 0

The grammar is ambiguous because we can find strings, which have multiple derivations:

S	***S***
0A	***0A***
00AA	***00AA***
OO1S1	***0011S***
OO11B1	***00110A***
001101	***001101***

Put the following grammar into Chomsky Normal Form. Show all work.

S -> A | AB0 | A1A
A -> A0 | e
B -> B1 | BC
C -> CB | CA | 1B

Remove all e rules
S -> e | A | AB0 | A1A | B0 | A1 | 1A
A -> A0 | 0
B -> B1 | BC
C -> CB | CA | 1B

Remove unit rules
S -> e | A0 | 0 | AB0 | A1A | B0 | A1 | 1A
A -> A0 | 0
B -> B1 | BC
C -> CB | CA | 1B

Convert remaining rules into proper form
S -> e | A0 | 0 | AS1 | B0 | A1 | 1A
A -> A0 | 0
B -> B1 | BC
C -> CB | CA | 1B
S_1 -> B0 | 1A

S -> e | AN_0 | AS_1 | BN_0 | AN_1 | N_1A

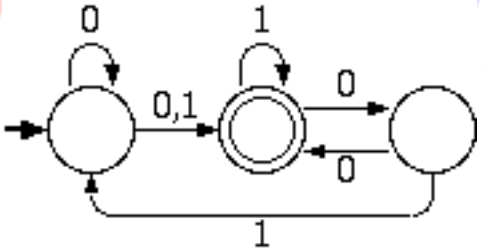
A -> AN_0 | 0
B -> BN_1 | BC
C -> CB | CA | N_1B
S_1 -> BN_0 | N_1A
N_0 -> 0
N_1 -> 1

Convert the following grammar to an equivalent one with no unit productions and no useless symbols. Show that the original grammar had NO useless symbols. What useless symbols are there after getting rid of unit productions?

S -> A | CB
A -> C | D
B -> 1B | 1
C -> 0C | 0
D -> 2D | 2

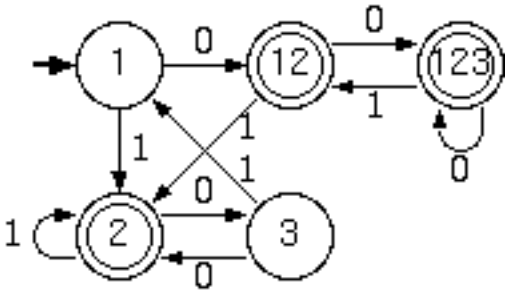
Converts to
S -> 0C | 0 | 2D | 2 | CB
A -> C | D
B -> 1B | 1
C -> 0C | 0
D -> 2D | 2
A is now useless and can be removed.

Consider the following NFA over the alphabet {0,1}:



- ***Convert this NFA to a minimal DFA.***
- ***Write a regular expression for the set the machine accepts.***
- ***Write a linear grammar where each right side is of the form aB or a. ("a" a terminal and "B" a non-terminal) to generate the set.***

Solution
a.



b. [0+(0+1)(1+00)*01]*(0+1)(1+00)*

c.
A -> 0A | 0B | 1B

B -> 1B | 0C | e
C -> 0B | 1A

MODEL QUESTIONS

SET-1

1. Define DFA and its language. Design a DFA that accepts all strings over ∑ = {a, b} in which both a's and b's are odd. Show the acceptance of string abaa sign its extended transition function.
2. State and prove Pumping Lemma for regular language. Show that L= $\{a^m b^m\}$ | m≥1} is not a regular language.
3. What is regular expression? Give the regular expression for the following languages.
   a) RE over {0, 1} whose first and fifth symbol is 1.
   b) RE of strings over {0, 1} in which 0 appears 3 times of any.
4. Find the minimum state DFA equivalent to following DFA.

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

5. Define Regular Grammar. Construct equivalent FA accepting the language generated by the following grammar.
   S→aabB | aaC | aba
   A→abA | aA | bB | ε
   B→aB | baA
   C→aC | abb
6. Show that any CFL without ε can be generated by a grammar in CNF. Convert following CFG into CNF.
   S→ASB | ε
   A→aAS | a
   B→SbS | A | bb
7. Define CFG and PDA. What is the relationship between them? Convert the following CFG into PDA.

E→I | E*E | E-E | E/E | (E)

I→a | b | Ia | Ib | I0 | I1

8. What is Turing Machine? How does it differ from PDA? Describe the different variations of Turing Machine.
9. Describe Universal Turing Machine and its operation.
10. Write short notes (any three)
    a) Epsilen-Closure of a state.
    b) CNF-SAT problem
    c) Church-Turing Thesis
    d) Parse-Tree

SET-2

1. What is the difference between DFA and NFA? Explain with their formal definition and examples. Design a DFA that accepts the set {abc, abd, aacd} over ∑={a, b, e, d}.
2. Define epsilon-closure of a state of an ε-NFA. Construct the following ε-NFA into equivalent DFA.

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

3. Show that for any regular expression r, there is a ε-NFA that accepts the same language represented by r. convert the following regular expression into ε-NFA.
   1(1+10)* + 10(0 + 10)* 0
4. Find the minimum state DFA equivalent to following DFA.

Q	0	1
→A	B	A
B	A	C
C	D	B

*D	D	A
E	D	F
F	G	E
G	F	G
H	G	D

5. Write a CFG that generates the following set over {a, b}: $\{a^i b^{i+1}, i \geq 1\}$. Convert the grammar into Chomsky's Normal Form.
6. Define parse tree of a string generated by a grammar. What is its nature if the grammar is in CNF? Show that if the longest path of the parse tree with yield w from its root to leaf is n, then $|w| <= 2^{n-1}$.
7. Define deterministic PDA. Design a PDA to accept a language $L=\{0^n 1^n \mid n>=1\}$. You may accept either by empty stack or by final state.
8. What is Universal Turing Machine and its Language. Explain its operation.
9. Consider the Turing Machine M,

   $M= (\{q_0, q_1, q_2, f\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{f\})$,

   Whose transitions are defined below.

   $\delta(q_0, 0) = \{(q_1, 1, R)\}$ $\quad\quad$ $\delta(q_1, 1) = \{(q_2, 0, L)\}$

   $\delta(q_2, 1) = \{(q_0, 1, R)\}$ $\quad\quad$ $\delta(q_1, B) = \{(f, B, R)\}$

   a) Provide the execution trace of this machine on the input 011.
   b) Describe the language accepted by M.
   c) Encode the above Turing Machine.
10. Write short note (any two):
    a. Universal Turing Machine
    b. Regular Grammar
    c. PDA vs CFG
    d. Unrestricted Grammar

SET 3

Attempt *all* questions

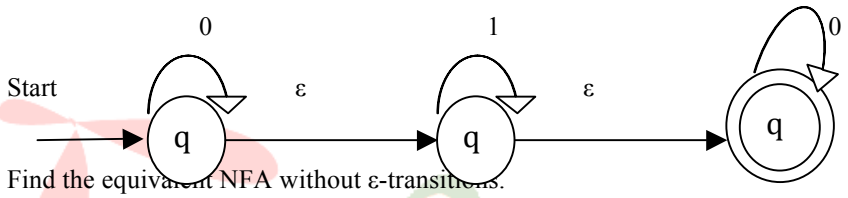
Group 'A' [8*4 = 32]

1. Prove the following by induction $2^x \geq x^2$ if $x \geq 4$.
2. Write the advantages/disadvantages of NFA over DFA.
3. Design a NFA for the language L = all strings over {0, 1} that have at least two consecutive 0's or 1's.
4. Write about the application of the Pumping Lemma.

5. Describe the closure properties of Regular Languages.
6. Describe in brief about ambiguity in grammars and languages.
7. Write in short about Chomsky Hierarchy.
8. Define P and NP class. Describe any two problems that lie in P and NP class.

Group 'B' [6*8 = 48]

9. Find the regular expression representing the following sets:
   (i) The set of all strings over {0, 1} having at most one pair of 0's or at most one pair of 1's.
   (ii) The set of all strings over {a, b} in which there are at least two occurrence of b between any two occurrences of a.
10. Write the CFG for the language L = $\{a^{2n}b^m / n > 0, m>=0\}$
11. Consider the NFA given by following diagram



Find the equivalent NFA without ε-transitions.

12. Design PDA for the grammar G = (V_n, V_t, P, S)
where
V_n = {S}
V_t = {a, b, c}
and P is defined as

S→aSa

S→bSb

S→c

13. Design a Turing Machine which accepts the language L = {w ϵ (a, b)*/w has equal number of a's and b's}.

Or

Prove that PCP with {(01, 011), (1, 10), (1, 11)} has no solution.

14. Write short notes (*any two*)
   (i) Church Thesis
   (ii) Halting Problem
   (iii) Universal Turing Machine
   (iv) Parse Tree

SET -4

Group-A [8*4=32]

1. Define the finite Automata. Describe its application in the field of Computer science.
2. Define ε-closure of a state.
3. Differentiate between Moore and Mealy Machine.
4. Define regular operators and regular language with example.
5. What do you mean by ambiguity in grammar and languages? Describe with suitable example.
6. Define a FA which accepts set of strings containing four 1's in every string over alphabet Σ= {0, 1}.
7. What is Regular expression? Give regular expression for the following languages.
   a) Strings over {0, 1} begin with 00 and end with 00.
   b) Strings over {0, 1} that begin or end with 00 or 11.
8. Describe Recursive and recursively enumerable language.

Group-B [6*8=48]

9. State and prove Pumping Lemma for regular language. Show that L= $\{a^m b^m | m>=1\}$ is not a regular language.

   Or,

   State and prove Pumping Lemma for CFL.By using CFL Pumping Lemma show that the language $\{0^n 1^n | n>=1\}$ is not a context free.

10. Define CNF.Change the following grammar into CNF.
    S→abSb/a/aAb.
    A→bS/aAAb.
11. Give the formal definition of Push Down Automata.Convert the given grammar into PDA.Also show complete sequences of ID's of PDA to accept string a+a.

    S→S+T|T
    T→T*F|F
    F→(S) |a

12. Define Turing Machine. Write down its application.Also build a Turing machine that accepts the language of all words that contain the substring bbb

    Or,

    Define Turing Machine with its block diagram.Also bulid a turing machine that accepts the language ODD PALINDROME.

13. Define P,NP and NP complete problem.Explain CROOK theorem.
14. What is Chomsky Hierarchy of the language? Differentiate the type-1 and type-2 of this language?

Technovative is a newly established organization with an aim of "dedicated to customer satisfaction. " The organization is established by a team of IT veteran experts in the field of programming, system development, web programming, networking, database and other computer related offshoots. The company seeks to use the human power produced in Nepal in its own country with an aid of promoting the standard of living of Nepalese people.

ALSO AVAILABLE

- Programming books (C, C++, JAVA, etc)
- Computer fundamentals for school level
- Professional course books
- E-Books on all Academic courses
- Preparatory Kits of all subjects

Contact Us

- http://www.facebook.com/Technovatives
- http://www.technovative.org
- technovative.anithub@gmail.com
- 9849709357; 9841293896; 9849182179; 9841225751;9803740363

Please “Like” OUR facebook fan page to get the recent updates and e-books on all academic courses.