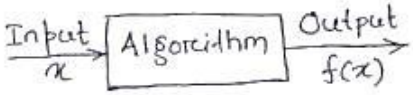
FORMAL LANGUAGE AND AUTOMATA THEORY
(THEORY OF COMPUTATION)

Module - I

Introduction

- Computation: It is a step by step solution to a problem. It is carried out by executing programs
- Theory of computation is the theory about programs.
- Program is nothing but the expression of algorithms in a particular language ie/ Program express algorithms.
- So theory of computation is about theory of algorithms
- An algorithm is a recipe (description of a process) for carrying out input to input transformation
- Function: A mapping from domain (input) to range (output)



- An algorithm computes a function
- Goal of theory: To figure out for what function we can have algorithms.
- Example of function:

is-Prime : number $\longrightarrow$ {yes, no}

func^{n} def^{n} $\longrightarrow$ $\text{is-Prime}(n) = \begin{cases} \text{Yes}, & \text{if } n \text{ is a prime} \\ \text{No}, & \text{if } n \text{ is not a prime} \end{cases}$

- Function definition does not say anything about how to come up with an answer.
- So we need algorithm which will tell us how to come up with an answer. The steps of algorithm should be simple operations which will be carried out on a computer.

AS THE DEFINITION OF A FUNCTION DOES NOT SAY ANYTHING ABOUT HOW TO COME UP WITH AN ANSWER, WE MAY HAVE SOME FUNCTION FOR WHICH WE DO NOT HAVE ALGORITHM IN FACT FOR MOST FUNCTIONS THERE IS NO ALGORITHM TO COMPUTE

- Basic Goal: To identify the class of functions which do not admit any algorithms to compute them.
- At the end of this course we will have the knowledge of the techniques to demonstrate that a function does not admit any algorithms to compute.

1 - Set Membership Problem: Given a set S and an input a. Decide whether $a \in S$ or not.

- Let us define a set which is associated to any function f as follows:

$$graph(f) = \{(a,b) \mid f(a) = b\}$$

- Suppose we show that there is no algorithm to solve the set membership problem graph(f) then we conclude that there is no algorithm to compute f.

Because if there is an algorithm to compute f then for input (a,b) we can compute f(a) using the algorithm and suppose $b = f(a) \iff (a,b) \in graph(f)$

THE ENTIRE COURSE CONCENTRATE ON SET & THEIR MEMBERSHIP FUNCTIONS RATHER THAN ONLY FUNCTIONS

- The sets are going to be the sets of finite strings.

Components of Theory of Computation

1. Complexity Theory:
   - The main question asked in this area: What makes some problems computationally hard and other problems easy?
   - We have to classify problems according to their degree of difficulty. We need to give a rigorous proof that problems that seem to be "hard" are really "hard".
2. Computability Theory:
   - In 1930 Godel Turing & church discovered that some of the mathematical problems (fundamental problems) can't be solved by a computer.
   - This theory helps us to classify the problems as being solvable or unsolvable.
3. Automata Theory:
   - This theory deals with definitions and properties of different types of computational models like
     - Finite Automata: Used in text processing, compiler etc.
     - Context-Free Grammars: Used in programming languages; AI etc.
     - Turing Machine: Model of real computer
   - Central Question: Do all these models have same power, or one model solve more problems than the other.

Central Concepts of Automata Theory

1. **Alphabet**: An alphabet is a finite, non empty symbols.
   We use the symbol Σ for an alphabet
   - $\Sigma = \{0, 1\}$, the binary alphabet
   - $\Sigma = \{a, b, \cdots, z\}$, the set of all lower case letters
   - The set of all ASCII characters
2. **Strings (word)**: A finite sequence of symbols chosen from some alphabet. Normally denoted by ω, x, y, z.

   Eg: If $\Sigma = \{0, 1\}$ then 1011, 111, 11000 are strings
   If $\Sigma = \{a, b, \cdots, z\}$ then abz, xyz " ".

Empty String: String with zero occurrences of symbols denoted by ϵ

Length of a string: The number of positions for symbols in the string

If $\omega = 1011$ then $|\omega| = 4$

Powers of an alphabet: If Σ: an alphabet
Σ^k: set of strings of length k whose symbol is in Σ

$\Sigma^0 = \epsilon$

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

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

Note: Diff between Σ & Σ^1 ← Set of strings with members as the strings 0, 1 of length 1
↑ alphabet with members 0, 1

Σ^*: set of all strings over an alphabet Σ

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$

Σ^+: set of nonempty strings from alphabet Σ

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \cdots$

$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Concatenation of Strings :

Let x, y : strings

xy : concatenation of x & y

Eg : If $x = 1101$, $y = 0011$ then $xy = 1101\,0011$, $yx = 0011\,1101$

For any string w , $\epsilon w = w\epsilon = w$

ϵ : identity for concatenation

3. **Languages** : set of strings all of which are chosen from some Σ^*, where Σ is a particular alphabet

If Σ is an alphabet & $L \subseteq \Sigma^*$, then L is a language over Σ

Examples : English language , C language

- Language of all strings consisting of n 0's followed by n 1's , for some $n \geq 0$
  $\{ \epsilon, 01, 0011, 000111, \cdots \}$
- Language of all strings of 0's and 1's with an equal no. of each
  $\{ \epsilon, 01, 10, 0011, 0110, 1001, \cdots \}$
- Language of binary nos whose value is a prime
  $\{ 10, 11, 101, 111, 1011, \cdots \}$
- Σ^* : language for any alphabet Σ
- ϕ : Empty language over any alphabet
- $\{\epsilon\}$: language consisting of only empty strings is a language over any alphabet

NOTE : $\phi \neq \{\epsilon\}$

(ϕ : contains no strings ; $\{\epsilon\}$: contains one string.)

4. Problem Given Σ: an alphabet
L: language over Σ
ω: string in Σ^*

(Set membership Problem)

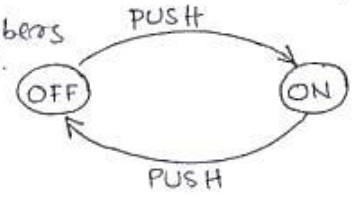
Decide whether or not ω is in L

Finite Automata (plural of automaton) — something capable of acting automatically without an external motive

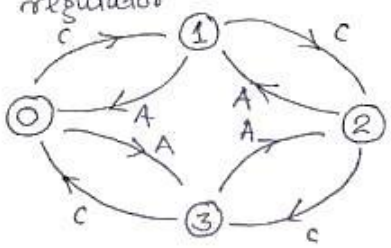
- It is a computational device (model) which has finite amount of memory. (Other computational devices: calculator, cell phone, computer etc.)
- It has a finite no of states which does not depend on input size.
- At a particular time it remembers only the current state.

Examples:

a) An electric switch



b) Fan regulator



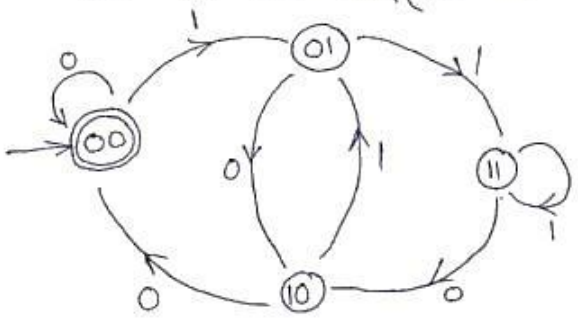
C: clockwise
A: anticlockwise

c) We are looking for a set of strings defined as

$L = \{x : x \text{ is a binary string divisible by } 4\}$

$= \{100, 1000, 1100, \ldots\}$

Note: All binary nos which are divisible by 4 have "00" at the end (00 as the suffix)



Let us store the information of last two bits in the states

Start state: 00
Accept state: 00

Eg of strings

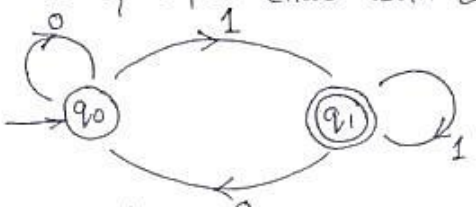
a) 110 — 00 —1→ 01 —1→ 11 —0→ 10 — string not acceptable

b) 1100 — 00 —1→ 01 —1→ 11 —0→ 10 —0→ 00 — string acceptable

5

d) Consider the following language

$L = \{ \omega \mid \omega \text{ ends with a } 1 \}$



state q_0: All strings end with 0

" q_1: " " 1

Deterministic Finite Automata (DFA)

Formal Def^n:

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

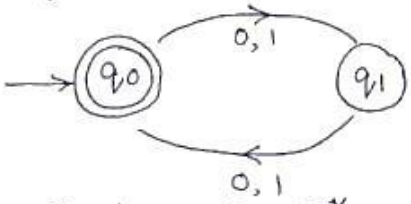
A DFA consists of:

1. Q: finite set of states
2. Σ: finite set of input symbols / alphabet
3. δ: $Q \times \Sigma \to Q$ is the transition function
4. q_0: start state $q_0 \in Q$
5. F: set of accept states $F \subseteq Q$

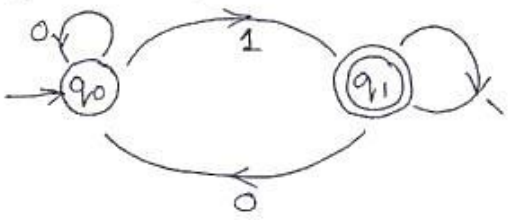
- We say that a DFA, $M = (Q, \Sigma, \delta, q_0, F)$ accepts a string ω if starting at the state q_0 the DFA ends at an accept state on reading the string ω
- The DFA M accepts a language $L \subseteq \Sigma^*$ if every string in L is accepted by M and no more which is denoted as $L(M)$.
- This is also referenced as M recognizes L

Examples of DFA

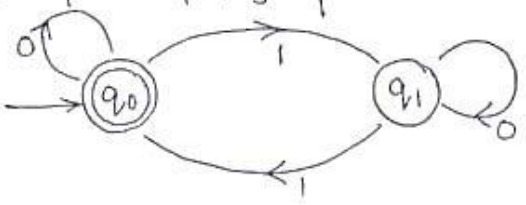
1. $L_1 = \{ \omega \mid \omega \in \{0,1\}^* \text{ and } |\omega| \text{ is even} \}$



2. $L_2 = \{ \omega \mid \omega \in \{0,1\}^* \text{ and } \omega \text{ ends with } 1 \}$

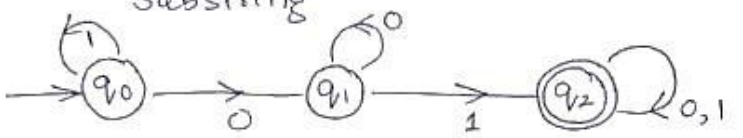


3. $L_3 = \{ \omega \in \{0,1\}^* \mid \omega$ has even no of 1's $\}$

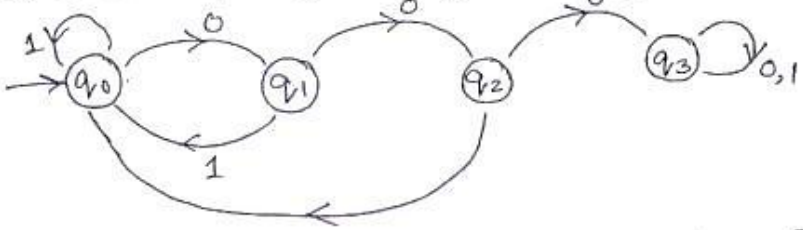


q_0 : all strings having even no of 1s
q_1 : all strings having odd no of 1s

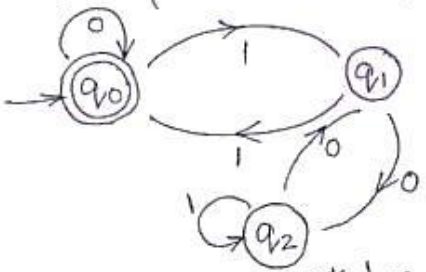
4. L_4 = set of all strings over $\{0,1\}$ having 01 as substring



5. L_5 = set of all strings having three consecutive 0's



6. $L_6 = \{ \omega \in \{0,1\}^* \mid \omega$ is divisible by 3 $\}$

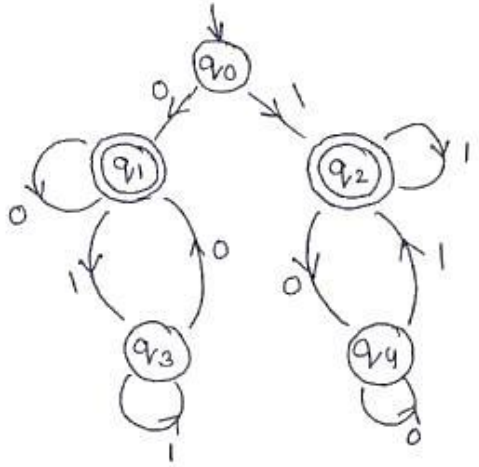


q_0 : ω such that $\omega \bmod 3 \equiv 0$
q_1 : " $\equiv 1$
q_2 : " $\equiv 2$

7. $L_7 = \{ \omega \in \{0,1\}^* \mid \omega$ begins and ends with same symbol $\}$

We have set of strings:

q_0 : empty string $\in$
q_1 : begin with 0, end with 0
q_2 : 1 1
q_3 : 0 1
q_4 : 1 0



7 NOTE:

- For a language there can be many DFAs accepting it
  But every DFA accepts exactly one language
- Given $(q, a) \to q'$ where q' is unique
  $q, q' \in Q$, $a \in \Sigma$
  This is the reason for which it is deterministic i.e/ Given a state and a symbol pair the model deterministically goes to an unique state

Transition Func^n : Transition from one internal state to another are governed by the transition function $\delta : Q \times \Sigma \to Q$.

$$\delta(q_i, a) = q_j$$

where $q_i, q_j \in Q$ and $a \in \Sigma$

State Tran. Diag / Transition Graph : Graph in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states while the labels on the edges are the current values of the input symbol.

Extended Transition Function ($\hat{\delta}$)

The extended transition function for DFA inductively defined as follows:

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

INDUCTION Let w is a string with $|w| \geq 1$.
w can be splitted as $w = xa$, where a is the last symbol of w.

$$\hat{\delta}(q, w) = \hat{\delta}(q, xa) = \delta\left(\hat{\delta}(q, x), a\right)$$

Computation of a DFA

Let $M = (Q, \Sigma, \delta, q_0, F)$ and $w = a_1 a_2 \dots a_n$ where $a_i \in \Sigma$
We say that M accepts w if $\exists$ a sequence of states $r_0, r_1, r_2, \dots, r_n$ such that:

i) $r_0 = q_0$ (initial condition)
ii) $r_i = \delta(r_{i-1}, a_i)$, $\forall i = 1, 2, \dots, n$ (transition ")
iii) $r_n \in F$ (acceptance ")

NOTE: The states r_i needs not be distinct.

Language of a DFA: Language L over Σ is said to be the language of DFA M iff

for all $\omega \in L$, M accepts ω

i.e. $\hat{\delta}(q_0, \omega) \in F$

Regular Language

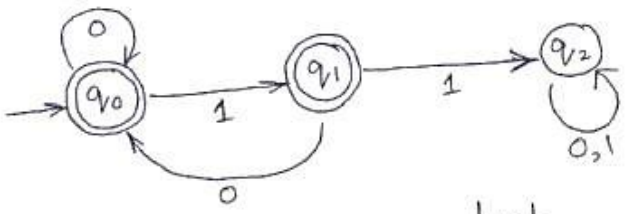
A language L is called regular iff some finite automaton M accepts / recognize it.

- Let $L \subseteq \Sigma^*$
  We say that M accepts L if $L = \{\omega \in \Sigma^* \mid M \text{ accepts } \omega\}$
- Let $L \subseteq \Sigma^*$
  We say that L is regular if $\exists$ a DFA, M such that $L = L(M)$

ARE THERE LANGUAGES WHICH ARE NOT REGULAR?
ANS: Yes
Eg: $L = \{0^n 1^n \mid n \geq 0\}$ is not regular

More Examples

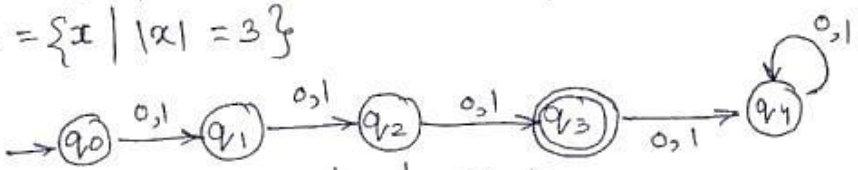
8. $L_8 = \{\omega \mid \omega$ does not contain 11 as substrings $\}$



q_0: does not contain 11 as a substring & does not end with a 1
q_1: does not contain 11 as a substring & ends with a 1
q_2: contain 11 as substrings.

Here the state q_2 is a dump state (a state from where the automaton can't reach an accept state.

9. $L_9 = \{x \mid |x| = 3\}$

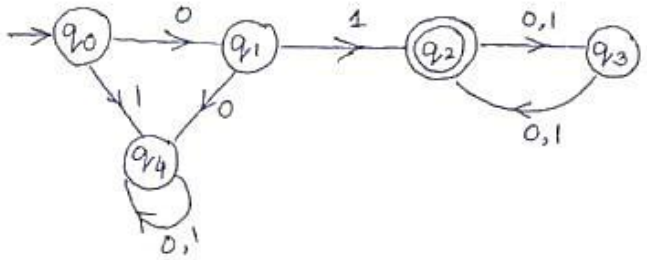


Here q_4 is a dump state.

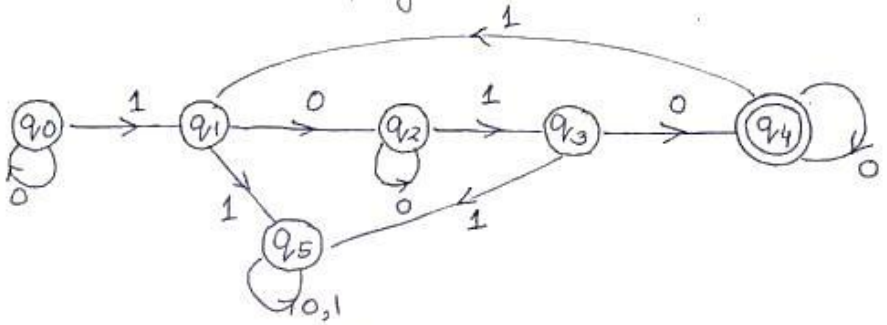
9

10. Design a DFA to accept the language

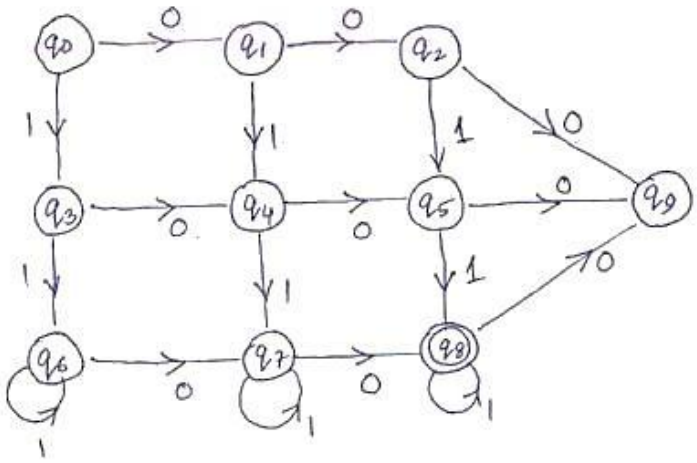
$L = \{ w \mid w \text{ is of even length and begins with } 01 \}$



11. Set of all strings over $\{0,1\}$ having even number of 1's and each 1 is followed by at least one 0.



12. Set of all strings having exactly two 0's and atleast two 1's



Transition Diagram

A transition diagram for a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

a) for each state in Q there is a node.

b) For each state $q \in Q$ & each input symbol $a \in \Sigma$, let $\delta(q,a) = p$, $p \in Q$. Then the transition diagram has an arc from node q to p labeled a. If there are multiple symbols that cause transitions from q to p, then the transition diagram can have one arc, labeled by the list of these symbols

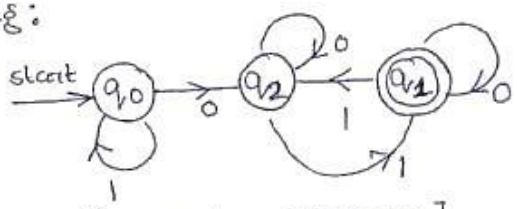
c) there is an arrow into the start state q_0, labeled start.

d) Nodes corresponding to accepting states (those in F) are marked by a double circle. States not in F have a single circle.

Transition Table

- It is the tabular representation of all the transition func^{n}s each of which takes two arguments and returns a value
- The rows of the table correspond to the states and the columns correspond to the inputs
- The entry for the row corresponding to state q and the column corresponding to input a is the state $\delta(q,a)$
- The start state is marked with an arrow
- The accepting states are marked with a star.

Eg:



[Transition Diagram]

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

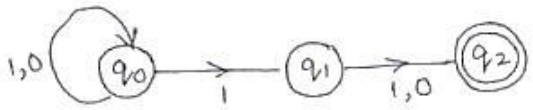
[Transition Table]

11

Nondeterministic Finite Automaton (NFA)

- It gives a machine multiple options for its moves.
- It has the power to be in several states at once.
- This ability is often expressed as an ability to guess something about its input.
- NFAs are more easier to design.

Example: Design a finite automaton which accepts strings that have the symbol 1 in the second last position.

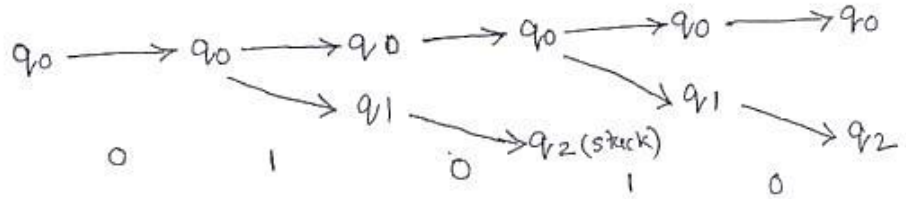


NOTE:

1. From a state q, on an input symbol a, the automaton can go to multiple states, $k \geq 0$.
2. Computation happens simultaneously along each of these paths
3. An input is accepted if there is some computation path that leads to an accept state.

Example Input string: 01010

Set of states the automaton is in	Symbol read
$\{q_0\}$ – initial state	0
$\{q_0\}$,	1
$\{q_0, q_1\}$	0
$\{q_0, q_2\}$	1
$\{q_0, q_1\}$	0
$\{q_0, q_1, q_2\}$	Input accepted.



As q_2 is one of the final state so 01010 accepted.

NFA : Formal Defn

An NFA is defined as:

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

where

1. Q : finite set of states
2. Σ : finite set of input symbols
3. q_0 : start state, $q_0 \in Q$
4. F : set of final states, $F \subseteq Q$
5. δ : transition func^n that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q.

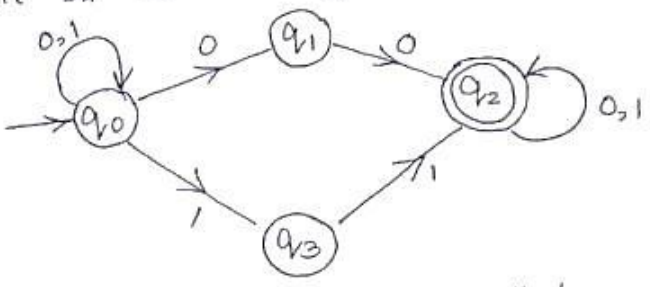
$$\delta : Q \times \Sigma \longrightarrow 2^Q$$

The transition table for the last example

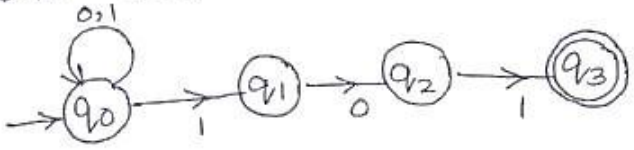
δ	0	1
$\rightarrow q_0$	q_0	q_0, q_1
q_1	q_2	q_2
$* q_2$	ϕ	ϕ

More Examples

1. Design an NFA which accepts any binary strings that contains 00 or 11 as substrings.



2. Design an NFA that accepts all binary strings that end with 101.



13 Extended Transition Function

We define the extended transition function ($\hat{\delta}$) for an NFA's transition function δ by:

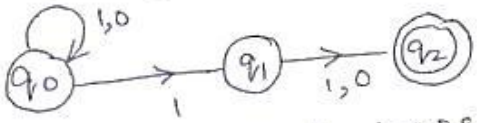
BASIS: $\hat{\delta}(q, \epsilon) = \{q\}$

INDUCTION: Suppose ω is of the form $\omega = xa$, where a is the final symbol of ω and x is the rest of ω. Also suppose that $\hat{\delta}(q, x) = \{p_1, p_2, \dots, p_k\}$

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

then $\hat{\delta}(q, \omega) = \{r_1, r_2, \dots, r_m\}$ i.e. $\hat{\delta}(q, \omega) = \bigcup_{i=1}^{k} \delta(p_i, a)$

Example Find $\hat{\delta}(q_0, 01010)$ in the following NFA



which accepts all strings whose second last symbol is 1.

Soln:

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

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

$\hat{\delta}(q_0, 01) = \hat{\delta}(q_0, 0) \cup \delta(q_0, 1)$ $\delta(q_0, 1) = \{q_0, q_1\}$

$\hat{\delta}(q_0, 010) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

$\hat{\delta}(q_0, 0101) = \delta(q_0, 1) \cup \delta(q_2, 1) = \{q_0, q_1\} \cup \phi = \{q_0, q_1\}$

$\hat{\delta}(q_0, 01010) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

Language of an NFA

If $A = (Q, \Sigma, \delta, q_0, F)$ is an NFA then

$L(A) = \{\omega \mid \hat{\delta}(q_0, \omega) \cap F \neq \phi\}$

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

Equivalence of NFA & DFA

- Every language that can be described by some NFA can also be described by some DFA.
- The proof that DFA's can do whatever NFA's can do involves an important "construction" called subset construction.

Construction Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an NFA

We need to construct a DFA, $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(D) = L(N)$.

Note that the input alphabets of the two automata are same and the start state of D is the set containing only the start state of N.

The rest of the components are constructed as follows:

- Q_D is the set of subsets of Q_N i.e. Q_D is the power set of Q_N.

  Note that not all these sets of Q_D are accessible from the start state. Inaccessible states are thrown away. So effectively no of states of D are much smaller than 2^n.
- F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \phi$. i.e. F_D is all sets of N's states that include at least one accepting state of N.
- For each set $S \subseteq Q_N$ & $a \in \Sigma$

$$\delta_D(S, a) = \bigcup_{p \text{ in } S} \delta_N(p, a)$$

Example Construct the equivalent DFA for the following NFA, $N = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$

δ	0	1
$\rightarrow q_0$	q_0	q_0, q_1
q_1	q_2	q_2
$* q_2$	ϕ	ϕ

Soln 15

Let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be the equivalent DFA.

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

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

The transition δ_D is shown in the following table

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

$\delta_D(\{q_0\},0) = \delta_N(q_0,0) = \{q_0\}$

$\delta_D(\{q_0\},1) = \delta_N(q_0,1) = \{q_0,q_1\}$

$\delta_D(\{q_1\},0) = \delta_N(q_1,0) = \{q_2\}$

$\delta_D(\{q_1\},1) = \delta_N(q_1,1) = \{q_2\}$

$\delta_D(\{q_2\},0) = \delta_N(q_2,0) = \phi$

$\delta_D(\{q_2\},1) = \delta_N(q_2,1) = \phi$

$\delta_D(\{q_0,q_1\},0) = \delta_N(q_0,0) \cup \delta_N(q_1,0) = \{q_0\} \cup \{q_2\} = \{q_0,q_2\}$

$\delta_D(\{q_0,q_1\},1) = \delta_N(q_0,1) \cup \delta_N(q_1,1) = \{q_0,q_1\} \cup \{q_2\} = \{q_0,q_1,q_2\}$

$\delta_D(\{q_1,q_2\},0) = \delta_N(q_1,0) \cup \delta_N(q_2,0) = \{q_2\} \cup \phi = \{q_2\}$

$\delta_D(\{q_1,q_2\},1) = \delta_N(q_1,1) \cup \delta_N(q_2,1) = \{q_2\} \cup \phi = \{q_2\}$

$\delta_D(\{q_0,q_2\},0) = \delta_N(q_0,0) \cup \delta_N(q_2,0) = \{q_0\} \cup \phi = \{q_0\}$

$\delta_D(\{q_0,q_2\},1) = \delta_N(q_0,1) \cup \delta_N(q_2,1) = \{q_0,q_1\} \cup \phi = \{q_0,q_1\}$

$\delta_D(\{q_0,q_1,q_2\},0) = \delta_N(q_0,0) \cup \delta_N(q_1,0) \cup \delta_N(q_2,0)$

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

$\delta_D(\{q_0,q_1,q_2\},1) = \delta_N(q_0,1) \cup \delta_N(q_1,1) \cup \delta_N(q_2,1)$

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

If we draw the transition diagram of the above constructed DFA we will find many of the states are not reachable. So instead of finding the transition func^n for all ~~set~~ state & input symbol combination we should find the transition func^n for the states which are reachable/accessible.

Def^n: Accessible state

BASIS: Set containing only N's start state is accessible

INDUCTION: Suppose we have determined that set S of states is accessible. Then for each input symbol a, compute the set of states $\delta_D(S,a)$. These sets of states will also be accessible.

Example: $\{q_0\}$ is the start state of the DFA. Hence it is accessible.

$New_\delta(\{q_0\}, 0) = \{q_0\}$ Old accessible state

$\delta(\{q_0\}, 1) = \{q_0, q_1\}$ New accessible state

$\delta(\{q_0, q_1\}, 0) = \{q_0, q_2\}$ New accessible state

$\delta(\{q_0, q_1\}, 1) = \{q_0, q_1, q_2\}$ New accessible state

$\delta(\{q_0, q_2\}, 0) = \{q_0\}$ Old

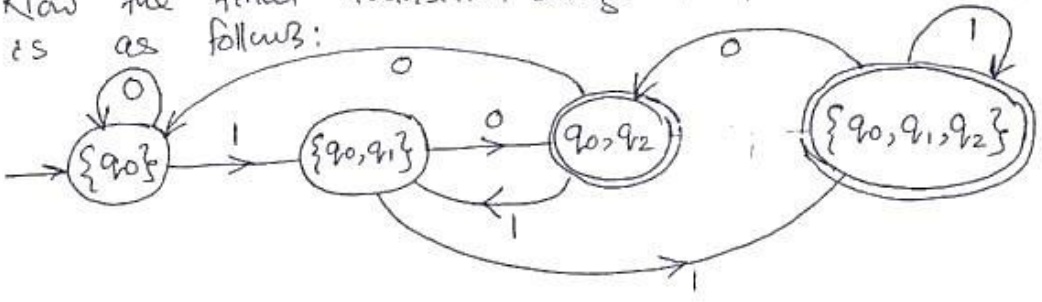
$\delta(\{q_0, q_2\}, 1) = \{q_0, q_1\}$ Old

$\delta(\{q_0, q_1, q_2\}, 0) = \{q_0, q_2\}$ Old

$\delta(\{q_0, q_1, q_2\}, 1) = \{q_0, q_1, q_2\}$ Old

~~Let us rename the states $\{q_0\}$:~~

Now the final transition diagram of the constructed DFA is as follows:



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

Proof We will prove this by induction of $|\omega|$.
We will show that $\hat{\delta}_D(\{q_0\}, \omega) = \hat{\delta}_N(q_0, \omega)$

BASIS: Let $|\omega|=0$ i.e. $\omega=\epsilon$. By the basis defn of $\hat{\delta}$ for DFA's and NFA's both $\hat{\delta}_D(\{q_0\}, \epsilon)$ and $\hat{\delta}_N(q_0, \epsilon)$ are $\{q_0\}$

INDUCTION: Let $|\omega| = n+1$ and assume that $\omega = xa$
By inductive hypothesis $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$
Let $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x) = \{p_1, \cdots, p_k\}$

As per the inductive part of the definition of $\hat{\delta}$ for NFA

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

As per the subset construction

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

As per the inductive definition of $\hat{\delta}$ for DFA

$$\hat{\delta}_D(\{q_0\}, \omega) = \delta_D(\hat{\delta}_D(\{q_0\}, x), a)$$
$$= \delta_D(\{p_1, p_2, \cdots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a)$$

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

This completes the proof $L(D) = L(N)$. □

Differences: DFA vs NFA

DFA	NFA
1. All transactions are deterministic	1. Some transitions could be non-deterministic
2. For each state, transition on all possible symbols should be defined	2. Not all transitions need to be defined explicitly
3. Accepts input if the last state visited is in F.	3. Accepts input if one of the last state is in F.
4. Sometimes harder to construct because of the no. of states	4. Generally easier than a DFA to construct.
5. Practical implementation is feasible	5. Practical implementations is limited.

Finite Automata with Epsilon-Transitions (ϵ-NFA)

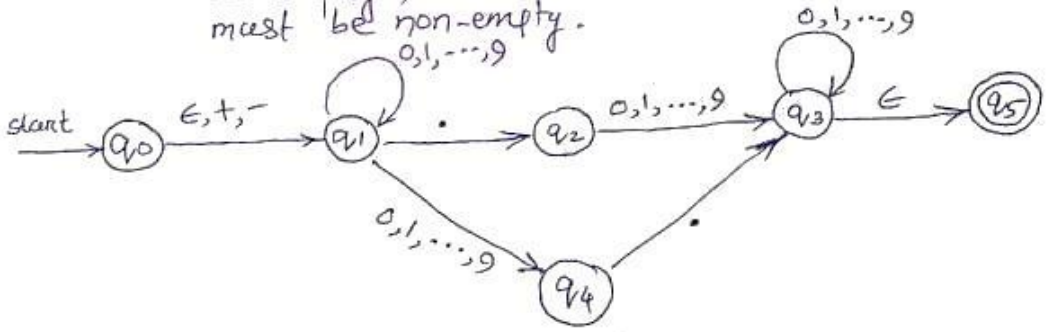
- From a state q, on an input symbol a, the automaton can go to multiple states k, $k \geq 0$
- Has ϵ-transitions: If there is a ϵ-transition then the automaton moves from q_i to q_j without reading the next input symbol. (spontaneous transition)

- Computation happens simultaneously along each of these paths
- An input is accepted if there is some computation path that leads to an accept state.

Example1: Design an ϵ-NFA that accepts decimal numbers consisting of:

1. An optional + or − sign
3. A string of digits (0,1,···,9)
3. A decimal point
4. Another string of digits.

Either of the strings of digits in (2 or 4) may be empty, but one of the two strings of digits must be non-empty.



q_0: initial state

q_1: we have seen the sign if there is one, but none of the digits or decimal point.

q_2: we have just seen the decimal point and may or may not have seen digits prior to decimal.

q_4: We have seen at least one digit, but not the decimal point.

q_3: We have seen a decimal point and at least one digit either before or after the decimal point

q_5: accepting state. We may stay in q_3 reading whatever digits there are and also have the option of guessing the string of digits is complete and going spontaneously to q_5.

19

ϵ-NFA : Formal Defn

An ϵ-NFA is the five tuple $N = (Q, \Sigma, \delta, q_0, F)$ where

Q : finite set of states

Σ : input alphabet

$\delta : Q \times \Sigma_\epsilon \to 2^Q \quad [\Sigma_\epsilon = \Sigma \cup \{\epsilon\}]$

q_0 : start state

F : set of accept states

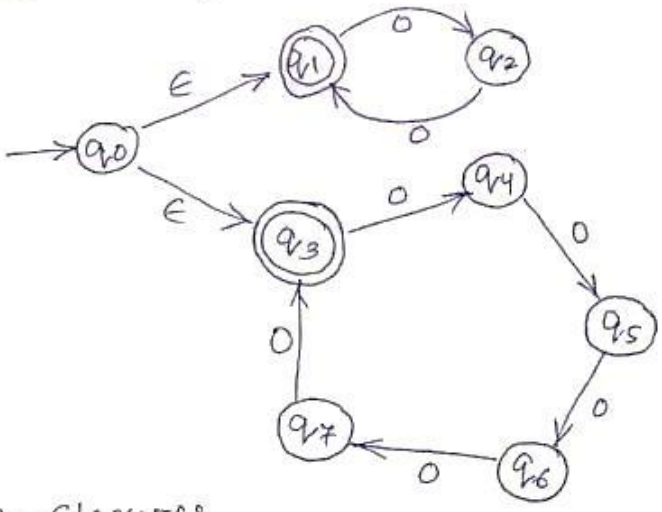
We say that N accepts an input $\omega = a_1 a_2 \cdots a_n$ if we can write ω as $\omega = b_1 b_2 \ldots b_m$ where $b_i \in \Sigma_\epsilon$ and $\exists$ a sequence of states $r_0, r_1, \ldots, r_m$ (not necessarily be distinct) such that :

i) $r_0 = q_0$ (initial condition)

ii) $r_i \in \delta(r_{i-1}, b_i) \quad \forall i = 1, 2, \ldots, m$ (transition condition)

iii) $r_m \in F$ (acceptance state)

Example 2: $L = \{\omega \in \{0\}^* \mid |\omega| \text{ is divisible by 2 or 5}\}$



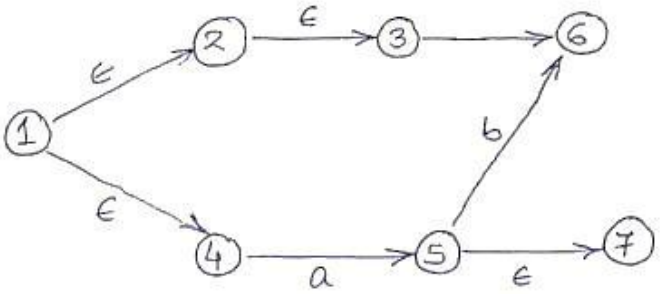
Epsilon – Closures

The ϵ-closure of a state q, ECLOSE(q) is recursively defined as follows:

BASIS : State q is in ECLOSE(q)

INDUCTION: If state p is in ECLOSE(q), and there is a transition from state p to state r labelled ϵ, then r is in ECLOSE(q)

Example:



ECLOSE(1) = {1, 2, 3, 4, 6}

ECLOSE(5) = {5, 7}

Extended Transitions

The extended transition ($\hat{\delta}$) of ε-NFA is defined as follows:

BASIS: $\hat{\delta}(q, \epsilon) = \text{ECLOSE}(q)$

INDUCTION: Suppose $w = xa$ where a is the last symbol $a \in \Sigma$

We compute $\hat{\delta}(q, w)$ as follows:

1. Let $\hat{\delta}(q, x) = \{p_1, p_2, \cdots, p_k\}$
2. Let $\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \cdots, r_m\}$
3. Then $\hat{\delta}(q, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$

Example Find $\hat{\delta}(q_0, 5.6)$ for the ε-NFA of Example 1

$\hat{\delta}(q_0, \epsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$

$\delta(q_0, 5) \cup \delta(q_1, 5) = \phi \cup \{q_1, q_4\} = \{q_1, q_4\}$

$\hat{\delta}(q_0, 5) = \text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$

$\delta(q_1, \cdot) \cup \delta(q_4, \cdot) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$

$\hat{\delta}(q_0, 5\cdot) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$

$\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \phi = \{q_3\}$

$\hat{\delta}(q_0, 5.6) = \text{ECLOSE}(q_3) = \{q_3, q_5\}$

Language of an ε-NFA, $E = (Q, \Sigma, \delta, q_0, F)$ is expressed as

$$L(E) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \phi\}$$

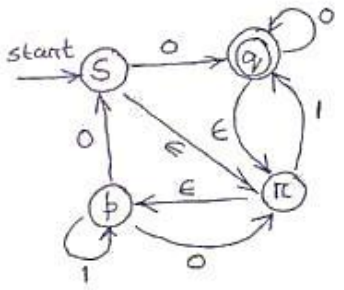
21 Eliminating ϵ-Transitions

Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ be an ϵ-NFA. Then the equivalent DFA, $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ is defined as follows:

1. Q_D contains those sets $S \subseteq Q_E$ such that $S = \text{ECLOSE}(S)$
   NOTE: ϕ is an ϵ-closed set
2. $q_D = \text{ECLOSE}(q_0)$
3. F_D is those sets of states that contain at least one accepting state of E. i.e $F_D = \{S \mid S \in Q_D \text{ and } S \cap F_E \neq \phi\}$
4. $\delta_D(S,a)$ is computed, for all $a \in \Sigma$ and sets $S \in Q_D$ by:
   a) Let $S = \{p_1, p_2, \ldots, p_k\}$
   b) Compute $\bigcup_{i=1}^{k} \delta(p_i, a)$. Let this set be $\{r_1, r_2, \ldots, r_m\}$
   c) Then $\delta_D(S,a) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$

Example Convert the following ϵ-NFA to an DFA.

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



Solu^n Let the equivalent DFA, is $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$

$q_D = \text{ECLOSE}(s) = \{s, r, p\}$

$\delta_D(\{s,r,p\}, 0) = \text{ECLOSE}(\delta_N(s,0) \cup \delta_N(r,0) \cup \delta_N(p,0))$

$= \text{ECLOSE}(\{q\} \cup \phi \cup \{s,r\}) = \text{ECLOSE}(\{q,r,s\}) = \{p,q,r,s\}$

$\delta_D(\{s,r,p\}, 1) = \text{ECLOSE}(\delta_N(s,1) \cup \delta_N(r,1) \cup \delta_N(p,1))$

$= \text{ECLOSE}(\phi \cup \{q\} \cup \{p\}) = \text{ECLOSE}(\{p,q\}) = \{p,q,r\}$

$\delta_D(\{p,q,r,s\}, 0) = \text{ECLOSE}(\delta_N(p,0) \cup \delta_N(q,0) \cup \delta_N(r,0) \cup \delta_N(s,0))$

$= \text{ECLOSE}(\{s,r\} \cup \{q\} \cup \phi \cup \{q\}) = \text{ECLOSE}(\{s,r,q\}) = \{p,q,r,s\}$

$\delta_D(\{p,q,r,s\}, 1) = \text{ECLOSE}(\delta_N(p,1) \cup \delta_N(q,1) \cup \delta_N(r,1) \cup \delta_N(s,1))$

$= \text{ECLOSE}(\{p\} \cup \phi \cup \{q\} \cup \phi) = \text{ECLOSE}(\{p,q\}) = \{p,q,r\}$

$\delta_D(\{p,q,r\},0) = ECLOSE(\delta_N(p,0) \cup \delta_N(q,0) \cup \delta_N(r,0))$

$= ECLOSE(\{s,r\} \cup \{q\} \cup \phi) = ECLOSE(\{s,r,q\}) = \{p,q,r,s\}$

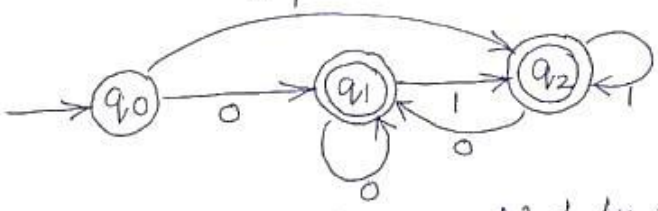
$\delta_D(\{p,q,r\},1) = ECLOSE(\delta_N(p,1) \cup \delta_N(q,1) \cup \delta_N(r,1))$

$= ECLOSE(\{p\} \cup \phi \cup \{q\}) = ECLOSE(\{p,q\}) = \{p,q,r\}$

As q was the accept state of NFA

$F_D = \{\{p,q,r\}, \{p,q,r,s\}\}$

After renaming $\{s,r,p\} : q_0$, $\{p,q,r,s\} : q_1$, $\{p,q,r\} : q_2$



Theorem A language L is accepted by some ϵ-NFA if and only if L is accepted by some DFA.

Proof **If part**

Suppose $L = L(D)$ for some DFA.

Turn D into ϵ-NFA by adding transitions $\delta(q,\epsilon) = \phi$ for all states q of D.

Convert the transition $\delta_D(q,a) = p$ into $\delta_E(q,a) = \{p\}$

Now the transitions of E and D are same so

$L = L(D) = L(E)$

Only-if part

Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ be an ϵ-NFA and after the modified subset construction the produced DFA is $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$

We need to show $L(D) = L(E)$

Formally, we need to show $\hat{\delta}_E(q_0,\omega) = \hat{\delta}_D(q_D,\omega)$

by induction on length of ω.

23

BASIS: if $|\omega| = 0$ then $\omega = \epsilon$

We know $\hat{\delta}_E(q_0, \epsilon) = ECLOSE(q_0)$ [As per the basis of $\hat{\delta}_E$]

From the subset construction we know $q_D = ECLOSE(q_0)$

For a DFA, we know $\hat{\delta}(p, \epsilon) = p$

So $\hat{\delta}_D(q_D, \epsilon) = \hat{\delta}_D(ECLOSE(q_0), \epsilon) = ECLOSE(q_0)$

Hence $\boxed{\hat{\delta}_E(q_0, \epsilon) = \hat{\delta}_D(q_D, \epsilon)}$ proved for $|\omega| = 0$

INDUCTION: Suppose $\omega = xa$

Assume that for $\omega = x$ $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(q_D, x) = \{p_1, p_2, \cdots, p_k\}$

By the defn of $\hat{\delta}$ for ϵ-NFA's we compute $\hat{\delta}_E(q_0, \omega)$ by:

1. Let $\{r_1, r_2, \cdots r_m\}$ be $\bigcup_{i=1}^{k} \delta_E(p_i, a)$
2. Then $\hat{\delta}_E(q_0, \omega) = \bigcup_{j=1}^{m} ECLOSE(r_j)$

By examining the construction of DFA, D in the modified subset construction we find that $\delta_D(\{p_1, p_2, \cdots, p_k\}, a)$ is constructed by the same two steps.

Thus $\hat{\delta}_D(q_D, \omega) = \delta_D(\{p_1, p_2, \cdots p_k\}, a) = \hat{\delta}_E(q_0, \omega)$

Now we proved that $\hat{\delta}_E(q_0, \omega) = \hat{\delta}_D(q_D, \omega)$

Minimization of DFA

- Any DFA defines a unique language, but the converse is not true.
- For a single language we may have multiple DFAs
- For storage efficiency it is desirable to reduce the no. of states as far as possible.

Defn Two states p, q of a DFA are called <u>indistinguishable</u>

if $\hat{\delta}(p, \omega) \in F \Rightarrow \hat{\delta}(q, \omega) \in F$

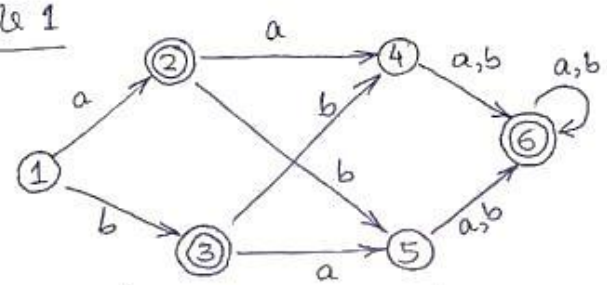
and $\hat{\delta}(p, \omega) \notin F \Rightarrow \hat{\delta}(q, \omega) \notin F$

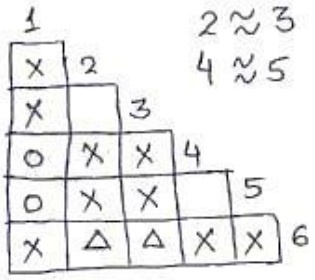
If $\exists$ some string $\omega \in \Sigma^*$ such that $\hat{\delta}(p, \omega) \in F$ and $\hat{\delta}(q, \omega) \notin F$ or vice versa then the states p & q are said to be <u>distinguishable</u>

Algorithm

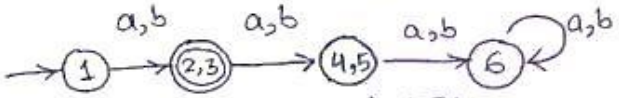
1. Remove all inaccessible states
2. Consider all pairs of states (p,q). If $p \in F$ & $q \notin F$ or vice-versa, mark the pair (p,q) as distinguishable
3. Repeat the following step until no previously unmarked pairs are marked
   a) For all pairs (p,q) and all $a \in \Sigma$, compute $\delta(p,a) = pa$ and $\delta(q,a) = qa$.
   b) If the pair (pa, qa) is marked as distinguishable, mark (p,q) as distinguishable.
4. Two or more states are equivalent if they are not marked

Example 1





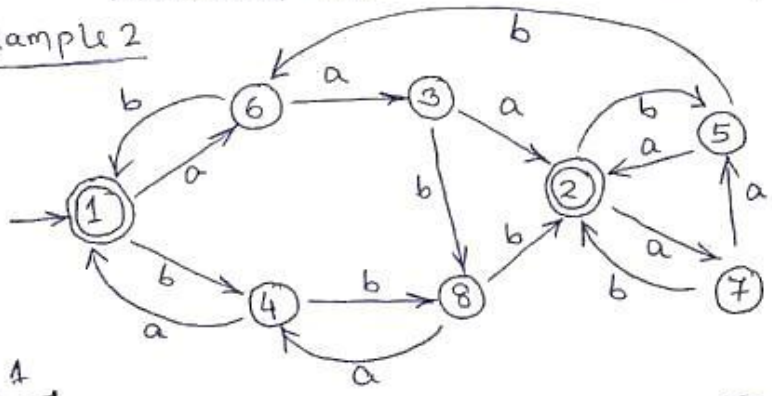
$2 \approx 3$
$4 \approx 5$

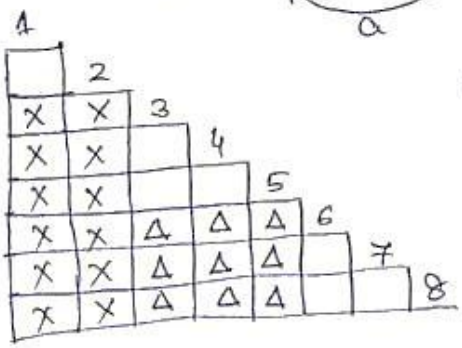


Minimized DFA

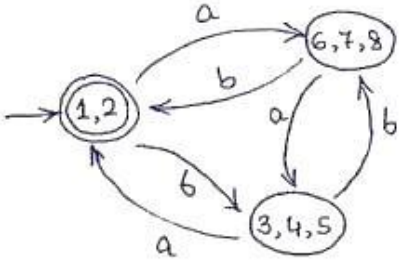
X : Marked after step 2
Δ : Marked after step 3 (1st iteration)
O : Marked after step 3 (2nd iteration)

Example 2





$1 \approx 2$
$3 \approx 4 \approx 5$
$6 \approx 7 \approx 8$



Minimized DFA

Module - II

REGULAR EXPRESSIONS

- Regular expressions are another way of defining language-notation.
- Closely related to NFA used for describing software components.
- Algebric description of languages

Operators of regular expressions

1. Union: The union of two languages L & M denoted as L∪M is the set of strings that are either in L or M or both.

   Eg: if $L = \{001, 100, 11\}$, $M = \{\epsilon, 100\}$ then $L \cup M = \{001, 100, 11, \epsilon, 100\}$

2. Concatenation: The concatenation of languages L and M is the set of strings that can be formed by taking any strings in L and concatenating it with any strings in M.

   Eg: $L.M = \{001, 001100, 100, 100100, 11, 11100\}$

3. Closure (or star, or kleene closure): Closure of a language L is denoted as L^* and represents the set of those strings that can be formed by taking any no. of strings from L, possibly with repetitions.

   Eg: if $L = \{0, 11\}$ then $L^* = \{0011, 1110, \dots\}$

   Formally $L^* = \bigcup_{i \geq 0} L^i$, where $L^0 = \{\epsilon\}$, $L^1 = L$ and L^i for $i > 1$ is $LL \dots L$ (concatenation of i copies)

   Eg: $L^0 = \{\epsilon\}$, $L^1 = L = \{0, 11\}$, $L^2 = \{00, 1111, 011, 110\}$

   $\phi^* = \{\epsilon\}$

   NOTE: ϕ is one of only two languages whose closure is not infinite

Formal Def^n of Regular Expressions

BASIS: 1) The constants ϵ and ϕ are regular expressions
$L(\epsilon) = \{\epsilon\}$, $L(\phi) = \phi$

2) If a is any symbol, then a is a regular expression
$L(a) = \{a\}$

3. A variable **L** represents any language

INDUCTION:

1. If E & F are regular expressions, then E+F is a regular expressions. If L(E) & L(F) are the languages of E, F respectively then $L(E+F) = L(E) \cup L(F)$
2. If E & F are regular expressions, then EF is a regular expressions. So $L(EF) = L(E)\,L(F)$
3. If E is a regular expressions, then E^* is a regular expression. So $L(E^*) = (L(E))^*$
4. If E is a regular expression then (E) is also a regular expression. $L((E)) = L(E)$

Example (Language → RE)

Language	Regular Expression
a) $\{w \mid w$ has a single 1$\}$	0^*10^*
b) $\{w \mid w$ has at most a single 1$\}$	$0^* + 0^*10^*$
c) $\{w \mid \lvert w\rvert$ is divisible by 3$\}$	$((0+1)(0+1)(0+1))^*$
d) $\{w \mid w$ has a 1 at every odd position and length of w is odd$\}$	$1((0+1)1)^*$
e) $\{w \mid w$ has 101 as substring$\}$	$(0+1)^*101(0+1)^*$

Precedence of Regular-Expression Operators

The precedence of operators from high to low are:
parentheses(( )), star(*), dot(·), plus(+)

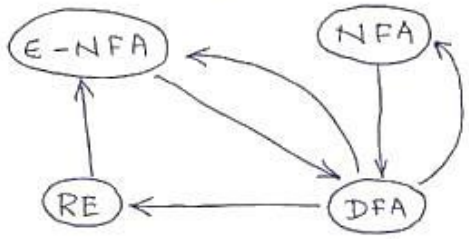
Eg: $01^* + 1$ is grouped $(0.(1^*))+1$

Examples RE → Language

RE	Language
01	$\{01\}$
01 + 1	$\{01, 1\}$
(01 + ∈)1	$\{011, 1\}$
(0+10)*(∈+1)	$\{\in, 1, 0, 01, 10, 101, 00, 001, 010, 0101 \ldots\}$

Converting Regular Expressions to Automata

The plan of showing the equivalence of four different notations for regular languages is as follows:



Theorem: Every language defined by a regular expression is also defined by a finite automaton

Proof: Suppose $L = L(R)$ for a regular expression R.
We show that $L = L(E)$ for some ϵ-NFA E with

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

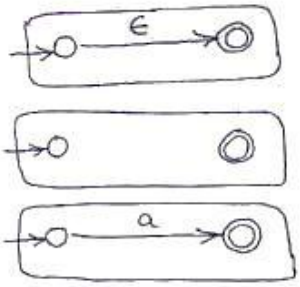
The proof is by structural induction on R, following the recursive definition of regular expressions

BASIS

a) ϵ

b) ϕ

c) a

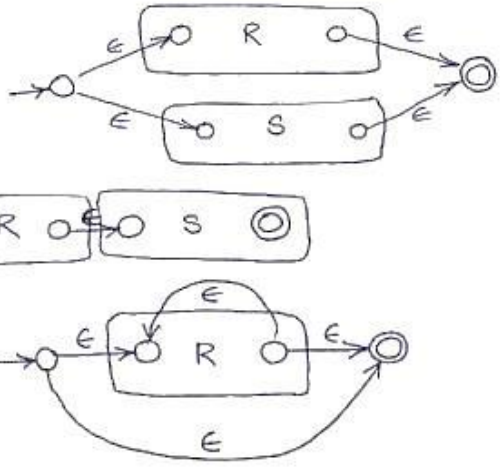


INDUCTION

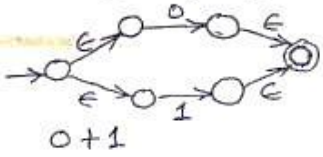
a) $R+S$

b) RS

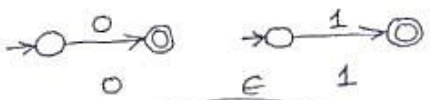
c) R^*

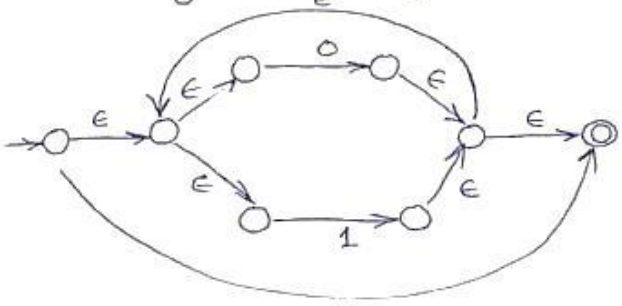


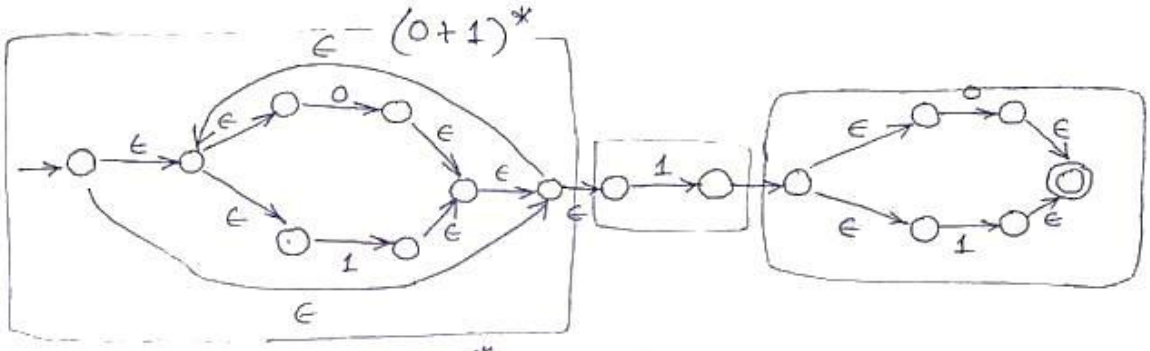
27 Example: Convert the regular expression $(0+1)^*1(0+1)$ to an ϵ-NFA.



Soln







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

Algebraic Laws for Regular Expressions

1. Associativity and Commutativity

- $L+M = M+L$ Union of two languages in either order are same (Commutative Law for Union)
- $(L+M)+N = L+(M+N)$ (Associative Law for Union)
- $(LM)N = L(MN)$ (Associative Law for Concatenation)
- $LM \neq ML$ (Concatenation is not commutative)

2. Identities and Annihilators

- $\phi + L = L + \phi = L$. [ϕ is the identity for union]
- $\epsilon L = L\epsilon = L$. [ϵ is the identity for concatenation]
- $\phi L = L\phi = \phi$ [ϕ is the annihilator for concatenation]

3. Distributive Laws

- $L(M+N) = LM + LN$
- $(M+N)L = ML + NL$

4. The Idempotent Law

- $L+L=L$ [Idempotence law of union]

5. Laws Involving Closures

- $(L^*)^* = L^*$
- $\phi^* = \epsilon$
- $\epsilon^* = \epsilon$
- $L^+ = LL^* = L^*L$ [Note: $L^+ = L+LL+LLL+\cdots$, $L^* = \epsilon+L+LL+LLL+\cdots$]
- $L^* = L^+ + \epsilon$
- $L? = \epsilon + L$

Closure Properties of Regular Languages

1. Union

If L_1 and L_2 are regular languages then $L_1 \cup L_2$ is regular

Proof: If L_1 and L_2 are regular, then there exist regular expressions r_1 and r_2 such that $L_1 = L(r_1)$ and $L_2 = L(r_2)$.
By defn $r_1 + r_2$ is a regular expressions denoting the language $L_1 \cup L_2$

2. Concatenation

- If L_1 and L_2 are regular languages then L_1L_2 is regular.

Proof: Similar to the proof of union using defn of regular expression.

3. Star; If L is a regular language then L^* is regular

Proof: Similar to the proof of union using the defn of regular expression.

4. Complement:

If L is a regular language then $\overline{L}\ (\Sigma^* - L)$ is regular

Proof: Let $D = (Q, \Sigma, \delta, q_0, F)$ be some DFA for L.
We construct a DFA D' for $\overline{L}$ where
$D' = (Q, \Sigma, \delta, q_0, Q-F)$
which will be the DFA for $\overline{L}$.

5. Intersection:

If L_1 and L_2 are regular then $L_1 \cap L_2$ is also regular

Proof: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ (De'Morgan's Law)
L_1, L_2 are regular $\Rightarrow \overline{L_1}, \overline{L_2}$ are regular
$\Rightarrow \overline{L_1} \cup \overline{L_2}$ is regular $\Rightarrow \overline{\overline{L_1} \cup \overline{L_2}}$ is regular $\Rightarrow L_1 \cap L_2$ is regular

6. Set Difference

If L_1, L_2 are regular languages then $L_1 - L_2$ is also regular

Proof: $L_1 - L_2 = L_1 \cap \overline{L_2}$

L_1, L_2 are regular $\Rightarrow$ $\overline{L_2}$ is regular $\Rightarrow$ $L_1 \cap \overline{L_2}$ is regular

$\Rightarrow$ $L_1 - L_2$ is regular.

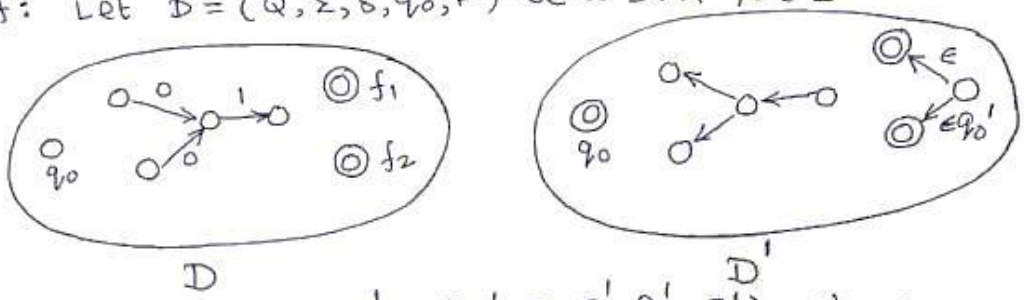
7. Reversal

Let $\omega = a_1 a_2 \cdots a_n$ be a string then $rev(\omega) = a_n a_{n-1} \cdots a_1$

For a language $L \subseteq \Sigma^*$, $rev(L) = \{\omega \in \Sigma^* \mid rev(\omega) \in L\}$

If L is regular, then $rev(L)$ is also regular

Proof: Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA for L



Define an NFA $D' = (Q', \Sigma, \delta', q_0', F')$ where

$Q' = Q \cup \{q_0'\}$

$\delta'(q, a) = \{r \mid \delta(r, a) = q\}$

$\delta'(q_0', \epsilon) = \{f \mid f \in F\}$,

$F' = \{q\}$

then $L(D') = rev(L)$

8. Homomorphism

Let Σ, Γ be two alphabets (may be not be distinct)

Then a function $h: \Sigma \to \Gamma^*$ is called a homomorphism

Let ω be a string $\omega \in \Sigma^*$ and $\omega = a_1 a_2 \cdots a_n$ then

$h(\omega) = h(a_1) h(a_2) \cdots h(a_n)$

Let $L \subseteq \Sigma^*$, $h(L) = \{h(\omega) \in \Gamma^* \mid \omega \in L\}$

If L is regular then $h(L)$ is also regular.

Example: Take $\Sigma = \{a, b\}$ and $\Gamma = \{b, c, d\}$.

Define h by $h(a) = dbcc$, $h(b) = bdc$

If L is the regular language denoted by

$r = (a + b^*)(aa)^*$

then $r_1 = (dbcc + (bdc)^*)(dbccdbcc)^*$

denotes the regular language $h(L)$

9. Inverse Homomorphism

Let $h: \Sigma \to \Gamma^*$ is a homomorphism and $L \subseteq \Gamma^*$, define $h^{-1}(L) = \{w \in \Sigma^* \mid h(w) \in L\}$

If L is regular then $h^{-1}(L)$ is also regular.

Non-regular Languages and Pumping Lemma

- Regular languages has at least four different descriptions: DFA, NFA, ϵ-NFA, regular expressions.
- There exist non-regular languages
- A powerful technique, known as "pumping lemma" is used for showing certain languages not to be regular.

Theorem: Pumping lemma for regular languages

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

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. For all $k \geq 0$, the string xy^kz is also in L.

Proof Suppose L is regular

then $L = L(A)$ for some DFA A.

Suppose A has n states, Let us consider any string $w = a_1 a_2 \ldots a_m$, where $m \geq n$ & each a_i is an input symbol

For $i = 0, 1, \ldots, n$ Let $p_i = \hat{\delta}(q_0, a_1 a_2 \ldots a_i)$

Note that δ is the transition func^n of A & q_0 is the start state of A. That means $p_0 = q_0$.

By the pigeonhole principle, it is not possible for the $n+1$ different p_i's for $i = 0, 1, \ldots, n$ to be distinct, since there are only n different states.

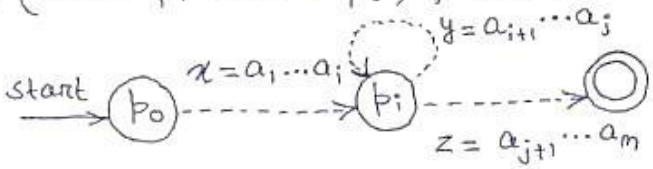
Thus, we can find two different integers i and j, with $0 \leq i < j \leq n$ such that $p_i = p_j$.

Now we can break $w = xyz$ as follows:

1. $x = a_1 a_2 \ldots a_i$
2. $y = a_{i+1} \cdots a_j$
3. $z = a_{j+1} \cdots a_m$

31

That is, x takes us to p_i once; y takes us from p_i back to p_i (since p_i is also p_j), and z is the balance of ω.



Note: x may be empty, y may be empty.
However y can not be empty since $i < j$

If the automaton receives the input xy^kz, $k \geq 0$ we need to check whether it accepts or not.

When $k=0$: Automaton A goes from the start state $q_0 (= p_0)$ to p_i on input x. Since $p_i = p_j$, it must go from p_i to accept state on input z.

When $k>0$: Automaton A goes from q_0 to p_i on input x, circles p_i to p_i, k times on input y^k and then goes to the accepting state on input z. Thus, for any $k \geq 0$, xy^kz is also accepted by A. $\therefore xy^kz \in L$

Statement of Pumping Lemma

If L is a regular language then $\exists\, n \geq 0$, such that for all string $\omega \in L$ where $|\omega| \geq n$, $\exists$ a partition of $\omega = xyz$ such that $|xy| \leq n$ and $|y| > 0$ and for all $k \geq 0$ $xy^kz \in L$.

Contrapositive: If $A \Rightarrow B$ then $\neg B \Rightarrow \neg A$

Contrapositive form of Pumping Lemma

Let $L \subseteq \Sigma^*$ if $\forall n \geq 0$ (O), $\exists\, \omega \in L$ s.t. $|\omega| \geq n$ s.t. $\forall$ partitions $\omega = xyz$ where $|xy| \leq n$ and $|y| > 0$ (O), $\exists\, k \geq 0$ (Y) s.t. $xy^kz \notin L$ then L is not regular

Game [Opponent move: O; Your move: Y]

Example

Prove that the following languages are not regular

(i) $L_1 = \{ 0^m 1^m \mid m \geq 0 \}$

(ii) $L_2 = \{ a^l b^m c^p \mid l+m \leq p \}$

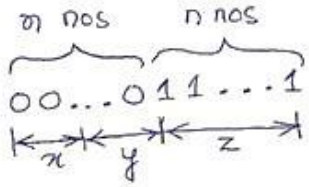
(iii) $L_3 = \{ 0^p \mid p \text{ is a prime} \}$

(iv) $L_4 = \{ \omega \in \Sigma^* \mid \text{no of } 0\text{'s in } \omega = \text{no of } 1\text{'s in } \omega \}$

Soln

(i) o[Given $n \geq 0$

Y[Choose $\omega = 0^n 1^n$ [$\because |\omega| = 2n \geq n$]



o[Now given a partition $\omega = xyz$ where $|xy| \leq n$ and $|y| > 0$

Note that the strings x & y consists of only 0's, otherwise $|xy| > n$

Let $y = 0^t$. So $x = 0^{n-t}$ and $z = 1^n$

Y[Choose $k = 0$ then $xy^0z = xz = 0^{n-t}1^n \notin L_1$

$\Rightarrow$ The language $\{ 0^m 1^m \mid m \geq 0 \}$ is not regular.

(ii) Given $n \geq 0$

Choose $\omega = a^n b^n c^{2n}$ [$|\omega| > n$ and $n+n \leq 2n$]

Now given a partition $\omega = xyz$ where $|xy| \leq n$ and $|y| > 0$.

So we have $y = a^t$ for some $t > 0$ and $t < n$

So $x = a^{n-t}$, $y = a^t$, $z = b^n c^{2n}$

Choose $k = 2$ then $xy^2z = a^{n-t} a^{2t} b^n c^{2n} = a^{n+t} b^n c^{2n}$

Here $n+t+n = 2n+t \notin 2n$ [$\because t$ is $< n$ and > 0]

$\Rightarrow xy^2z \notin L_2$

So L_2 is not regular

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

[Hint: Choose $\omega = 0^n 1 0^n$ & show that for $k=0$ $xy^kz \notin L$]

(iii) Given $n \geq 0$

Choose $\omega = 0^{\pi}$ where π is a prime $\geq n$

Now given a partition, $\omega = xyz$ where $|y| = t$, $t > 0$

Choose $k = \pi + 1$

then $xy^k z = xyz \cdot y^{k-1} = 0^{\pi} \cdot (0^t)^{k-1} = 0^{\pi + (k-1)t}$

Now $\pi + (k-1)t = \pi + \pi t$ $\left[\because k = \pi + 1 \Rightarrow k - 1 = \pi\right.$

$= \pi(1+t)$

As π is a prime no $\geq n$, $n \geq 0$ So π is at least 2

And $t \geq 1$ as $|y| > 0$. So $t+1$ is at least 2.

So $\pi(1+t)$ is a composite no.

$\Rightarrow xy^k z \notin L_3 \Rightarrow L_3$ is not regular.

(iv) $L_4 = \{ \omega \in \Sigma^* \mid$ no of 0's in ω = no of 1's in $\omega \}$

$L(0^*1^*) \cap L_4 = L_1$

0^*1^* is regular because we can construct a DFA for this.

If L_4 is regular $\Rightarrow L_1$ is regular

This is a contradiction

Hence L_4 is not regular.

NOTE

* $L \cap L' = L''$

  If L is regular and L' is not regular it does not imply that L'' is not regular

* If L and L' are not regular even then L'' may be regular

Pumping Lemma as an Adversarial Game

1. Player 1 picks the language L to be proved nonregular
2. Player 2 picks n, but doesn't reveal to player 1 what is n.
3. Player 1 picks ω such that $|\omega| \geq n$
4. Player 2 divides ω into x, y, z s.t. $y \neq \epsilon$ and $|xy| \leq n$
   Player 2 does not disclose to player 1 what x, y, z are
5. Player 1 wins by picking k such that $xy^k z \notin L$.

Arden's Theorem : Conversion of DFA into RE.

Let P and Q be two regular expressions. If P does not contain null string, then $R = Q + RP$ has a unique solution that is $R = QP^*$

Proof:

$$\begin{aligned} R &= Q + RP \\ &= Q + (Q+RP)P \\ &= Q + QP + RPP \\ &= Q + QP + (Q+RP)P^2 \\ &= Q + QP + QP^2 + RP^3 \\ &= Q(\epsilon + P + P^2 + P^3 + \dots) \end{aligned}$$

$$\Rightarrow R = QP^*$$

Assumptions for applying Arden's Theorem

- The transition diagram must not have ϵ transitions
- It must have only one initial state.

Method

Step 1: Create equations as the following form of all the states of the DFA having n states with initial state q_1

$$q_1 = q_1R_{11} + q_2R_{21} + \dots + q_nR_{n1} + \epsilon$$

$$q_2 = q_1R_{12} + q_2R_{22} + \dots + q_nR_{n2}$$

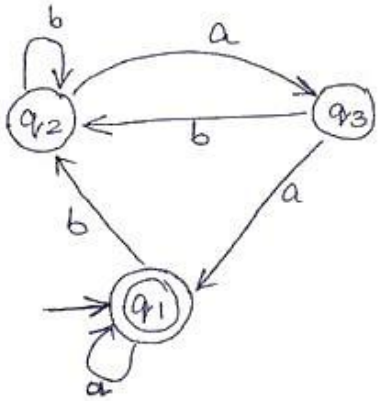
$$\dots\dots\dots$$

$$q_n = q_1R_{1n} + q_2R_{2n} + q_nR_{nn}$$

R_{ij} represents the set of labels of edges from q_i to q_j. If no such edge exists then $R_{ij} = \phi$

Step 2: Solve these equations to get the equation for the final state in terms of R_{ij}

Example: Convert the following automaton into regular expression:



The equation for the three states q_1, q_2, q_3 as follows:

$$q_1 = q_1a + q_3a + \epsilon$$

$$q_2 = q_1b + q_2b + q_3b$$

$$q_3 = q_2a$$

Solving these eqn's:

$$\begin{aligned} q_2 &= q_1b + q_2b + (q_2a)b \\ &= q_1b + q_2(b + ab) \\ &= q_1b(b+ab)^* \end{aligned}$$

[Applying Arden's Theorem]

$q_1 = q_1a + q_3a + \epsilon$

$= q_1a + q_2a.a + \epsilon$

$= q_1a + q_2a^2 + \epsilon$

$= q_1a + q_1b(b+ab)^*aa + \epsilon$

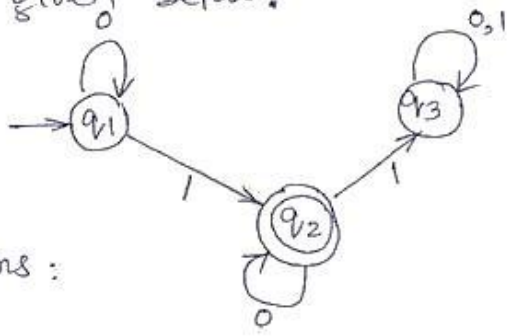
$= q_1(a + b(b+ab)^*aa) + \epsilon$

$= \epsilon(a + b(b+ab)^*aa)^*$

$= (a + b(b+ab)^*aa)^*$

Hence the regular expression is $(a + b(b+ab)^*aa)^*$

Example 2: Construct a regular expression corresponding to the automaton given below:



Soln: the equ^{n}s are

$q_1 = q_10 + \epsilon$

$q_2 = q_11 + q_20$

$q_3 = q_21 + q_30 + q_31$

Now we solve these equations:

$q_1 = q_10 + \epsilon$

$= \epsilon + q_10$

$= \epsilon.0^* = 0^*$

$q_2 = q_11 + q_20$

$= 0^*1 + q_20$

$= 0^*1.0^*$

As q_2 is the final state the regular expression is 0^*10^*

Context-Free Grammars and Languages

- Context-free language is a larger class of language than regular language.
- These languages have a natural, recursive notation called context-free grammars (CFG)
- Major application of CFG is in programming language and compiler design.

Example: Language of palindromes is defined as:

$$L_{pal} = \{ \omega \in \Sigma^* \mid \omega = \omega^R \}, \quad \omega^R \text{ is reverse of } \omega$$

Using pumping lemma we can prove that L_{pal} is not a regular language.

Inductive definition of palindrome is as follows:

BASIS: ϵ, 0, 1 are palindromes

INDUCTION: If ω is palindrome, so are $0\omega0$ and $1\omega1$.

No string of 0's and 1's is a palindrome unless it follows from this basis and induction rule

CFG is a formal notation for expressing recursive def^n of languages.

It consists of:

1. Variables: represents strings. Denoted with capital letters
2. Terminals: Similar to alphabet
3. Production Rules: Describe how the variables get replaced.
4. Start variable: Starting point of computation

The CFG for palindrome is described as

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

Terminal: $\{0, 1\}$
Variable: $\{P\}$
Production Rules:
Start variable: P

How computation happens:

1. Write down the start variable as the current string
2. Pick a variable in the current string and replace it with one of its production rules
3. Continue step 2 until no more variables are left.

Example: $P \to OPO \to O1P10 \to 01110$

Example 2: CFG for the language $L = \{ 0^n 1^{2n} \mid n \geq 0 \}$ is as follws:

Terminals: $\{0, 1\}$

Variables: $\{S, A, B\}$

Production Rules: $S \to ASB$, $A \to 0$, $B \to 11$, $S \to \epsilon$

Start Variable: S

$S \to ASB \to OSB \to OASBB \to OABB \to OAB11 \to OA1111 \to 001111$

Formal Def^n: CFG

A grammar G is defined as a 4-tuple $G = (V, T, P, S)$

where V is the finite set of variables

T is the finite set of terminals

P is the finite set of productions

$S \in V$ is a special variable called start variable

Here V and T are non-empty and disjoint

NOTE: The production rules are the heart of grammar. They specify how the grammar transforms one string to another

In our grammar all productions are of the form $\alpha \to \gamma$, where $\alpha \in (VUT)^+$ and $\gamma \in (VUT)^*$

Language Generated by a Grammar

Let $G = (V, T, P, S)$ be a grammar

Then the set $L(G) = \{ \omega \in T^* : S \overset{*}{\Rightarrow} \omega \}$ is the language generated by G.

NOTE: A language L is said to be a context-free language (CFL) if $\exists$ a CFG, G s.t. $L = L(G)$

Example 3 Consider the grammar $G = (\{S\}, \{a, b\}, P, S)$ with P given by:

$$P = \{ S \to aSb, S \to \epsilon \}$$

Then $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

We can write $S \overset{*}{\Rightarrow} aabb$

We say aabb is a string in the language generated by G.

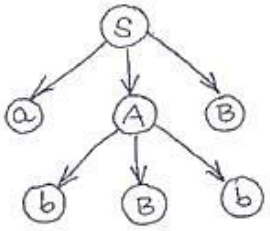
Here S, aSb, aaSbb, aabb are called the sentential form

Here $L(G) = \{ a^n b^n \mid n \geq 0 \}$

The string of symbols obtained by reading the leaves of the tree from left to right omitting any ϵ's encountered is said to be the yield of the tree.

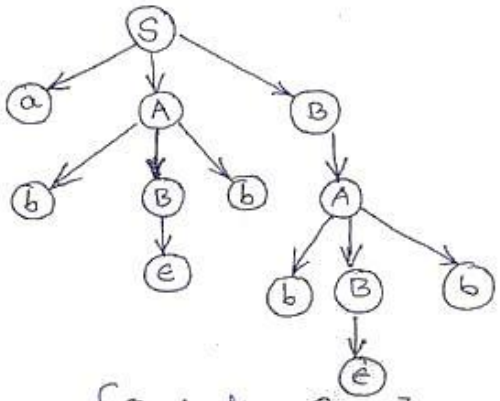
Example: Consider the grammar G, with productions

$S \rightarrow aAB$
$A \rightarrow bBb$
$B \rightarrow A \mid \epsilon$



[Partial Derivation Tree]

Yield: abBbB



[Derivation Tree]

Yield: abbbb

Parsing

- Parsing describes finding a sequence of productions by which $w \in L(G)$ is derived.

Exhaustive Search Parsing / Brute Force Parsing

Given a string $w \in L(G)$. We can parse it in the following manners:

Systematically construct all possible (Let leftmost) derivations and check whether any of them match w in the following rounds:

Round 1: Look for all productions of the form $S \rightarrow x$, finding all x that can be derived from S in one step. If none of them matches w then go to round 2.

Round 2: In this round we apply all applicable productions to the leftmost variable of every x. This gives us some sentential forms, some of them possibly w. If not go to round 3 and so on.

Example

Consider the grammar

$S \to SS \mid aSb \mid bSa \mid \epsilon$

and the string $\omega = aabb$

Round 1

$S \Rightarrow SS$
$S \Rightarrow aSb$
$S \Rightarrow bSa$
$S \Rightarrow \epsilon$

] Not considered further

Round 2

$S \Rightarrow SS \Rightarrow SSS$
$S \Rightarrow SS \Rightarrow aSbS$
$S \Rightarrow SS \Rightarrow bSaS$
$S \Rightarrow SS \Rightarrow S$

] Not considered further

$S \Rightarrow aSb \Rightarrow aSSb$
$S \Rightarrow aSb \Rightarrow aaSbb$
$S \Rightarrow aSb \Rightarrow abSab$
$S \Rightarrow aSb \Rightarrow ab$

] Not considered further

In Round 3

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

Hence $aabb$ is accepted.

Limitation of Exhaustive Search Parsing

- Tedious. Not used where efficient parsing is required.
- Although the method always parses a $\omega \in L(G)$, it is possible that it never terminates for strings not in $L(G)$

  [Eg: $\omega = abb$]

NOTE: The problem of nontermination of exhaustive search parsing can be overcome if we rule out the productions of the form:

$A \to \epsilon$ and $A \to B$

Theorem

Suppose that $G = (V, T, P, S)$ is a CFG which does not have any productions of the form $A \to \epsilon$ or $A \to B$ where $A, B \in V$. Then exhaustive search parsing method can be made into an algorithm which for any $\omega \in \Sigma^*$, either produces a parsing of ω or tells us that no parsing is possible.

Theorem

For every context-free grammar there exists an algorithm that parses any $\omega \in L(G)$ in a no. of steps proportional to $|\omega|^3$.

Simple Grammar (S-grammar) : Defn

A CFG, $G = (V, T, P, S)$ is said to be a simple grammar if all productions are of the form

$$A \rightarrow a\alpha,$$

where $A \in V$, $a \in T$, $\alpha \in V^*$, and any pair (A, a) occurs at most once in P.

Example: The grammar

$$S \rightarrow aS \mid bSS \mid c$$

is an S-grammar

The grammar

$$S \rightarrow aS \mid bSS \mid aSS \mid c$$

is not an S-grammar as the pair (S, a) occurs in the two productions $S \rightarrow aS$ and $S \rightarrow aSS$.

Ambiguity in Grammar

A context-free grammar G is said to be ambiguous if there exists some $w \in L(G)$ that has two distinct derivation trees.

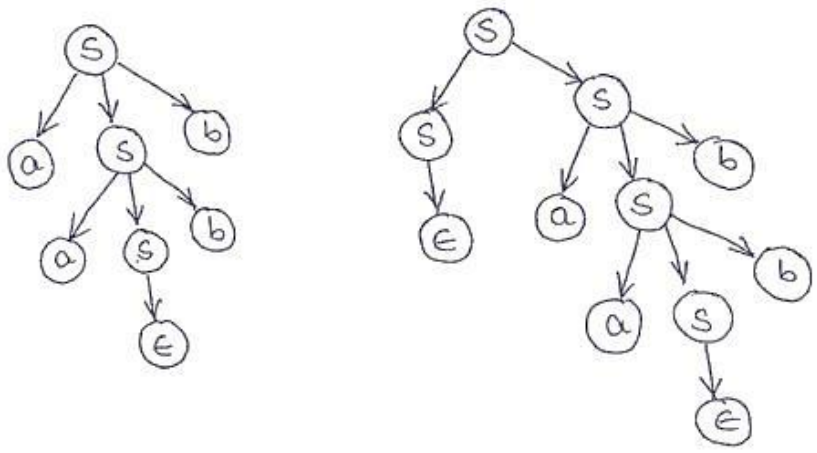
Example:

The grammar

$$S \rightarrow aSb \mid SS \mid \epsilon$$

is ambiguous

The string aabb has two derivation trees as follows:



Example

Consider the grammar $G = (V, T, P, E)$ with

$V = \{E, I\}$

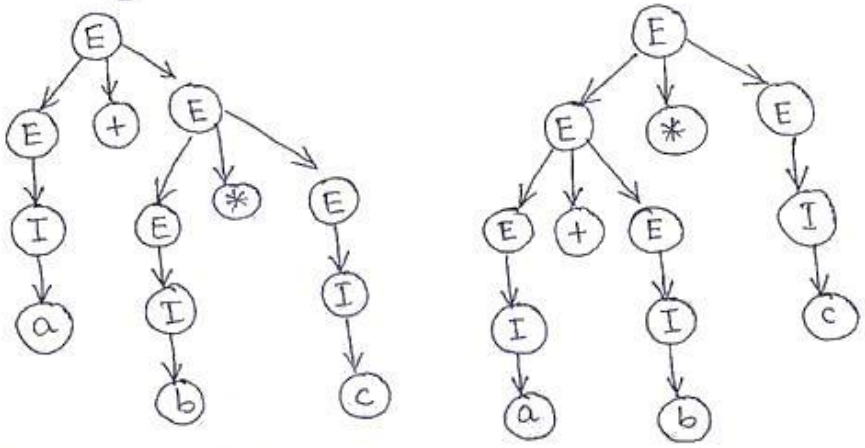
$T = \{a, b, c, +, *, (,)\}$

and P: $E \rightarrow I \mid E+E \mid E*E$

$E \rightarrow (E)$

$I \rightarrow a \mid b \mid c$

The string $a+b*c$ has two different derivation trees.



Unambiguous Language

If L is a CF language for which there exists an unambiguous grammar then L is said to be unambiguous language.

Inherently Ambiguous

If every grammar that generates L is ambiguous, then the language is called inherently ambiguous.

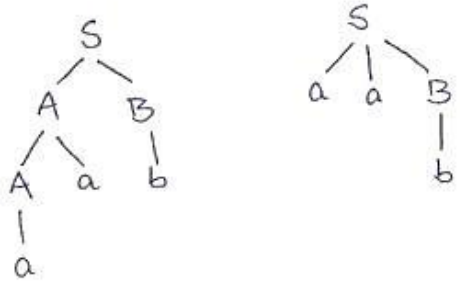
Question

Show that the following grammar is ambiguous

$S \rightarrow AB \mid aaB$

$A \rightarrow a \mid Aa$

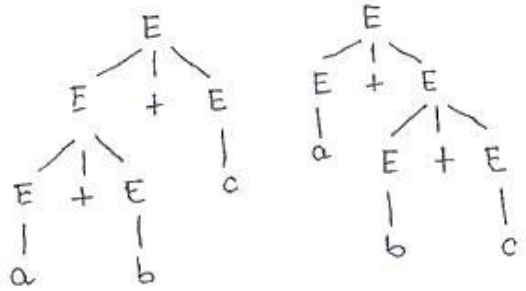
$B \rightarrow b$



Removal of Ambiguity

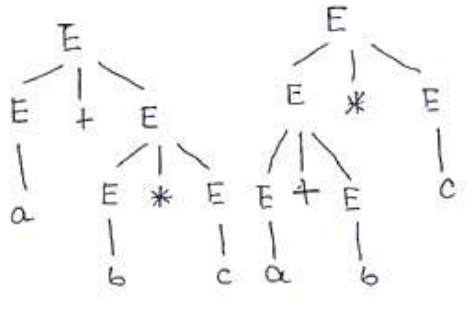
Consider a grammar $G = (V, T, P, S)$ where the production P is given as:

$$E \rightarrow E+E \mid E*E \mid a \mid b \mid c$$



[Two different parse tree for $a+b+c$]

No Associativity of operators



[Two different parse tree for $a+b*c$]

No precedence of operators

Soln

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow a \mid b \mid c$$

This grammar is unambiguous.
The operators +, * are left associative
* has more precedence than +.

More Examples

1) $E \rightarrow E+E \mid E*E \mid E \uparrow E \mid a \mid b \mid c$

Soln

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow G \uparrow F \mid G$$
$$G \rightarrow a \mid b \mid c$$

Operators precedence:

$\uparrow > * > +$

Associativity of operators:

+, * — left associativity

$\uparrow$ — right "

2) $E \rightarrow E \,||\, E \mid E \,\&\&\, E \mid !E \mid 0 \mid 1$

Soln

$$E \rightarrow E \,||\, F \mid F$$
$$F \rightarrow F \,\&\&\, G \mid G$$
$$G \rightarrow !G \mid 0 \mid 1$$

Operator precedence:

$! > \&\& > ||$

Associativity of operators

||, && : Left

! : Right

3. From the following grammar given below find the precedence of operators. Also find the associativity of each operator

$A \rightarrow A \$ B \mid B$
$B \rightarrow B \# C \mid C$
$C \rightarrow C @ D \mid D$
$D \rightarrow d$

Precedence :
$\$ < \# < @$
Here all operators are left associative

Simplification of a CFG

In a CFG, it may happen that all the production rules and symbols are not needed for the derivation of strings. Besides, there may be some null productions and unit productions. Elimination of these productions and symbols is called simplification of CFGs.

Simplification essentially comprises of the following steps :

- Eliminating Useless Productions (Reduction of CFG)
- Removal of Unit Production
- Removal of Null Production

By simplifying CFGs we remove all these redundant productions from a grammar, while keeping the transformed grammars equivalent to the original grammar.

Two grammars are equivalent if they produce same language.

Simplifying CFGs is necessary to later convert them into normal forms.

Two of the most useful normal forms are :

1. Chomsky Normal Form
2. Greibach Normal Form

Eliminating Useless productions

A variable is useless if either it cannot be reached from the start symbol or because it can't derive a terminal string

Example: In a grammar with start symbol S and productions

$S \rightarrow A$
$A \rightarrow aA \mid \epsilon$
$B \rightarrow bA$

the variable B is useless because we can't reach B from start symbol S.
So the production $B \rightarrow bA$ is also useless.

Example: Eliminate useless symbols and productions from $G = (V, T, P, S)$ where $V = \{S, A, B, C\}$ and $T = \{a, b\}$, with P consisting of

$S \rightarrow aS \mid A \mid C$
$A \rightarrow a$
$B \rightarrow aa$
$C \rightarrow aCb$

Step 1

Identify the set of variables that can lead to a terminal string. Remove other variables and their productions

Here, S, A, B are the variables that can lead to a terminal string since

$S \rightarrow A \rightarrow a$ So we eliminate C and its productions
$A \rightarrow a$
$B \rightarrow aa$

Now the grammar is

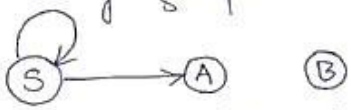
$S \rightarrow aS \mid A$
$A \rightarrow a$
$B \rightarrow aa$

Step 2 In this step eliminate the variables that can't be reached from the start variable by looking at the dependency graph.

A dependency graph has its vertices labeled with variables with an edge between vertices C and D if and only if there is a production of the form

$$C \to xDy$$

The dependency graph for the given grammar is as follows:



A variable is useful only if there is a path from the vertex labeled S to that.

So for the given grammar B is useless.

Removing B with all its productions

$$S \to aS \mid A$$
$$A \to a$$

Removing ϵ - Productions

Def^n: Any production of a context-free grammar of the form

$$A \to \epsilon$$

is called a ϵ-production

Any variable A for which the derivation

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

is possible is called nullable.

Method G: CFG with ϵ not in L(G).
$\hat{G}$: Equivalent CFG of G after removing ϵ-productions

I) First find the set V_N of all nullable variables of G using the following steps:

1. For all productions $A \to \epsilon$, put A into V_N
2. Repeat the following step until no further variables are added to V_N

   For all productions $B \to A_1 A_2 \dots A_n$

   where $A_1 A_2 \dots A_n$ are in V_N, put B into V_N

II) Next look at all productions in P of the form

$$A \to x_1 x_2 \dots x_m, \quad m \geq 1$$

where each $x_i \in VUT$

For each such production P, we put into $\hat{P}$ that production as well as those generated by replacing nullable variables with ϵ in all possible combinations

If all α_i's are nullable, the production $A \to \epsilon$ is not put into $\hat{P}$.

Example: Find a context-free grammar without ϵ-productions equivalent to the grammar defined by

$S \to ABaC$
$A \to BC$
$B \to b \mid \epsilon$
$C \to D \mid \epsilon$
$D \to d$

Soluⁿ

I) Here $V_N = \{B, C\}$ initially

Then as $A \to BC$ and $B, C \in V_N$ So $A \in V_N$

So now $V_N = \{A, B, C\}$

II) $S \to ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Aa \mid Ba \mid a$
$A \to BC \mid B \mid C$
$B \to b$
$C \to D$
$D \to d$

Removing Unit-Productions

Defⁿ: Any productions of a CFG of the form

$A \to B$

where $A, B \in V$ is called a unit-production

Method: G: CFG without ϵ-productions
$\hat{G}$: Equivalent CFG of G after removing unit-productions

I) First find for each A, all variables B such that

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

We can do this by drawing a dependency graph with an edge (C, D) whenever the grammar has a unit production $C \to D$

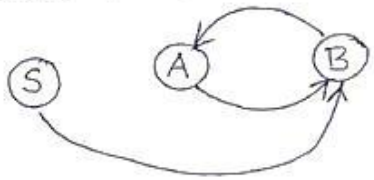
II) The new grammar $\hat{G}$ is generated by first putting into $\hat{P}$ all non-unit productions of P.
Next for all A & B satisfying $A \overset{*}{\Rightarrow} B$ we add $\hat{P}$

$A \to y_1 | y_2 | \cdots | y_n$

where $B \to y_1 | y_2 | \cdots | y_n$ is the set of rules in $\hat{P}$ with B on the left

Example: Remove all unit-productions from

$S \to Aa | B$
$B \to A | bb$
$A \to a | bc | B$



Soln

I) From the dependency graph we have

$S \overset{*}{\Rightarrow} B$, $B \overset{*}{\Rightarrow} A$, $A \overset{*}{\Rightarrow} B$. Hence $S \overset{*}{\Rightarrow} A$

II) Hence we keep the non-unit productions in $\hat{P}$

Now $\hat{P}$: $S \to Aa$
$A \to a | bc$
$B \to bb$

Next we add the following new rules to $\hat{P}$

$S \to a | bc | bb$
$A \to bb$
$B \to a | bc$

Now $\hat{P}$ becomes

$S \to a | bc | bb | Aa$
$A \to a | bb | bc$
$B \to a | bb | bc$

Step 2: Introducing additional variables for the first two productions

$S \rightarrow AD_1$
$D_1 \rightarrow BBa$
$A \rightarrow BaD_2$
$D_2 \rightarrow BaBb$
$B \rightarrow ABc$
$Ba \rightarrow a$
$Bb \rightarrow b$
$Bc \rightarrow c$

NOTE:

1. If the start symbol S occurs on some right side, create a new start symbol S' and a new production
   $S' \rightarrow S$
2. If the CFG contains null productions or unit productions first remove null productions and unit productions before converting it into CNF.

Example: Convert the following CFG into CNF.

$S \rightarrow ASA \mid aB$
$A \rightarrow B \mid S$
$B \rightarrow b \mid \in$

Solution: Since S appears in RHS, we introduce a new state S' and added the production $S' \rightarrow S$
Now the grammar is

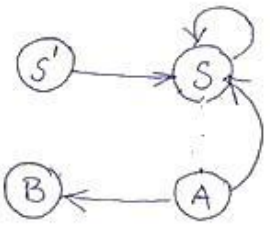
$S' \rightarrow S$
$S \rightarrow ASA \mid aB$
$A \rightarrow B \mid S$
$B \rightarrow b \mid \in$

Removing $\in$-productions

The nullable variables are A and B.
After removing $\in$-productions we have the following CFG:

$S' \rightarrow S$
$S \rightarrow AS \mid SA \mid S \mid aB \mid a \mid ASA$
$A \rightarrow B \mid S$
$B \rightarrow b$

Removing Unit productions



From the above CFG we have the dependency graph
From the dependency graph we can find out

$S' \overset{*}{\Rightarrow} S$: $A \overset{*}{\Rightarrow} S$

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

Excluding the ~~no~~ unit productions we have the following grammar

$S \rightarrow ASA \mid AS \mid SA \mid aB \mid a$
$B \rightarrow b$

Next we add the following new rules

$S' \rightarrow ASA \mid AS \mid SA \mid aB \mid a$
$A \rightarrow ASA \mid AS \mid SA \mid aB \mid a$
$A \rightarrow b$

So the grammar becomes

$S' \rightarrow ASA \mid AS \mid SA \mid aB \mid a$
$A \rightarrow ASA \mid AS \mid SA \mid aB \mid a \mid b$
$S \rightarrow ASA \mid AS \mid SA \mid aB \mid a$
$B \rightarrow b$

Converting into CNF

$S' \rightarrow ASA \mid AS \mid SA \mid BaB \mid a$
$A \rightarrow ASA \mid AS \mid SA \mid BaB \mid a \mid b$
$S \rightarrow ASA \mid AS \mid SA \mid BaB \mid a$
$B \rightarrow b$
$Ba \rightarrow a$

Changing the productions
$S' \rightarrow aB$
$S \rightarrow aB$
& $A \rightarrow aB$

Removing more than two variables from RHS we have

$S' \rightarrow AD_1 \mid AS \mid SA \mid BaB \mid a$
$A \rightarrow AD_1 \mid AS \mid SA \mid BaB \mid a \mid b$
$S \rightarrow AD_1 \mid AS \mid SA \mid BaB \mid a$
$B \rightarrow b$
$Ba \rightarrow a$
$D_1 \rightarrow SA$

which is in CNF.

Greibach Normal Form

- This normal form put restrictions on the positions in which terminals and variables can appear rather than putting restrictions on the length of the right side of production.

Def^n: A context-free grammar is said to be in Greibach normal form if all productions have the form

$$A \to ax$$

where $a \in T$ and $x \in V^*$

Example Convert the following grammar into Greibach Normal form:

$$S \to AB$$
$$A \to aA \mid bB \mid b$$
$$B \to b$$

Solu^n After replacing A in $S \to AB$ with $aA \mid bB \mid b$

we obtain

$$S \to aAB \mid bBB \mid bB$$

Now the grammar is

$$S \to aAB \mid bBB \mid bB$$
$$A \to aA \mid bB \mid b$$
$$B \to b$$

which is in Greibach Normal Form.

Example Convert the following grammar

$$S \to abSb \mid aa$$

into Greibach Normal form

Solu^n We introduce new variables A and B that are essentially synonyms for a and b respectively.

Substituting these we get

$$S \to aBSB \mid aA$$
$$A \to a$$
$$B \to b$$

Method to convert into GNF

1. It is advisible to convert the grammar into CNF before converting into GNF.
2. If the grammar contains productions in which the RHS starts with non-terminal then remove it using the following mechanism:

Mechanism 1: Removal of nonterminal

Let the grammar contains the production $X \to Y\alpha$ where $X, Y \in V$ and $\alpha \in (V \cup T)^*$

with the set of productions

$Y \to a_1 | a_2 | \cdots | a_n \cup P$, where $a_i \in T$.

Then the equivalent grammar will be with the following productions:

$$\hat{P} = P - \{X \to Y\alpha\} \cup \{X \to a_1\alpha | a_2\alpha | \cdots | a_n\alpha\}$$

3. If any of the productions of the form $A \to A\alpha$ then remove the left recursion using the following mechanism:

Mechanism 2: Removal of recursion (left)

Let the set of productions P_i contains the production

$$X \to X\alpha_1 | X\alpha_2 | X\alpha_3 | \cdots | X\alpha_n | \beta_1 | \beta_2 | \cdots | \beta_m$$

s.t. $X \in V$ and all $\alpha, \beta \in (V \cup \Sigma)^*$

the new productions will be as follows:

$$\hat{P} = P - \{X \to X\alpha_1 | X\alpha_2 | \cdots | X\alpha_n | \beta_1 | \beta_2 | \cdots | \beta_m\}$$
$$\cup \{X \to \beta_1 | \beta_2 | \cdots | \beta_m | \beta_1 Z | \beta_2 Z | \cdots | \beta_m Z$$
$$Z \to \alpha_1 | \alpha_2 | \cdots | \alpha_n | \alpha_1 Z | \alpha_2 Z | \cdots | \alpha_n Z\}$$

NOTE: Mechanism 1 & 2 can be used in any order.

Example 1 Convert the following grammar in GNF

$$S \to YY | 0$$
$$Y \to SS | 1$$

Solu^n Here $S \to 0$ and $Y \to 1$ are already in GNF

Now let us replace $S \to YY$ first

$S \to SSY \mid 1Y \mid 0$ [Using mechanism 1]

Now we find left recursion hence by applying mechanism 2

$S \to 1Y \mid 0 \mid 1YZ \mid 0Z$

$Z \to SY \mid SYZ$

Here the first production ie $S \to 1Y \mid 0 \mid 1YZ \mid 0Z$ is in GNF

Substitution S in $Z \to SY \mid SYZ$ we have

$Z \to 1YY \mid 1YYZ \mid 0Y \mid 0YZ \mid 1YZY \mid 1YZYZ \mid 0ZY \mid 0ZYZ$

which is in GNF now.

Now replace $Y \to SS$

$Y \to 0S \mid 1YS \mid 1YZS \mid 0ZS \mid 1.$

So the complete is:

$S \to 0 \mid 1Y \mid 0Z \mid YZ$

$Y \to 0S \mid 1YS \mid 1YZS \mid 0ZS \mid 1$

$Z \to 0Y \mid 1YY \mid 0ZY \mid 1YZY \mid 0YZ \mid 1YYZ \mid 0ZYZ \mid 1YZYZ$

Example 2: $S \to XY$, $X \to 0X \mid 1Y \mid 1$, $Y \to 1$ } Convert this CFG into GNF

Soln Except $S \to XY$ all are in GNF. This can be converted into GNF by replacing X

So $S \to 0XY \mid 1YY \mid 1Y$

$X \to 0X \mid 1Y \mid 1$

$Y \to 1$

Example 3: Convert the following CFG into GNF.

$S \to XY \mid X0 \mid p$

$X \to mX \mid m$

$Y \to Xn \mid 0$

Solution: No start symbol in the RHS, No Null production and No Unit production. $S \to X0$

Except the productions $S \to XY$, and $Y \to Xn$ all others are in GNF.

Replace X in $S \to XY$ by $X \to mX \mid m$

$S \to mXY \mid mY \mid p \mid mX0 \mid m0$

Also replace X in $Y \rightarrow Xn \mid o$

$$Y \rightarrow mXn \mid mn \mid o$$

So now the grammar becomes

$$S \rightarrow mXY \mid mY \mid mXo \mid mo \mid \epsilon$$
$$X \rightarrow mX \mid m$$
$$Y \rightarrow mXn \mid mn \mid o$$

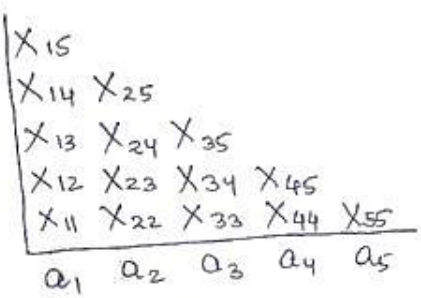
Now the production $S \rightarrow mXo$ and $Y \rightarrow mXn$ are not in GNF. So by introducing new variables O & N:

$$S \rightarrow mXY \mid mY \mid mXO \mid mO \mid \epsilon$$
$$X \rightarrow mX \mid m$$
$$Y \rightarrow mXN \mid mN \mid o$$
$$O \rightarrow o$$
$$N \rightarrow n$$

Testing Membership in a CFL / CYK Algorithm

Co ke Younger Kasami

- How to decide membership of a string ω in a CFL, L? There are many algorithms to test; all of them take exponential time in $|\omega|$
- CYK Algorithm is based on the dynamic programming algorithm design technique. It employs bottom-up parsing.
- The algorithm starts with a CNF grammar $G = (V, T, P, S)$ for a language L.
- The algorithm takes a string $\omega = a_1 a_2 \dots a_n$ in T^* as input
- It constructs a table that tells whether $\omega \in L$ or not. in $O(n^3)$ time
- The horizontal axis corresponds to the positions of the string $\omega = a_1 a_2 \dots a_n$ (Here $n = 5$)
- The table entry X_{ij} is the set of variables A such that $A \overset{*}{\Rightarrow} a_i a_{i+1} \dots a_j$
- If X_{1n} contains S then $\omega \in L$ because in that case $S \overset{*}{\Rightarrow} \omega$



- Each row corresponds to one length of substring: bottom row is for strings of length 1, second from bottom row for strings of length 2 and so on.
- Here is the algorithm for computing X_{ij}'s

BASIS: Compute the first row as follows

X_{ii} is the set of variables A such that $A \to a_i$ is a production of G.

INDUCTION: In order for A to be in X_{ij} we must find variables B and C and integer k such that

1. $i \leq k < j$
2. B is in X_{ik}
3. C is in $X_{k+1,j}$
4. $A \to BC$ is a production of G.

Finding such variable A requires us to compare at most n pairs of previously computed sets:
$(X_{ii}, X_{i+1,j}), (X_{i,i+1}, X_{i+2,j}) \dots (X_{i,j-1} X_{jj})$

Running Time: there are $\frac{n(n+1)}{2} = O(n^2)$ entries in the table
Computing each entry needs comparison of n already computed entries.
Hence running time = $O(n^3)$

Example: Test the membership of the string 110100 in L(G) where G is a CNF grammar whose productions are as follows:

$S \to AB \mid BC$
$A \to BB \mid 0$
$B \to BA \mid 1$
$C \to AC \mid AA \mid 0$

{S}					
{S,B}	{A,C}				
{S,B}	{A,C}	{S}			
{A,C}	{A}	{S}	{S,B}		
{A}	{S,B}	{S}	{S,B}	{C}	
{B}	{B}	{A,C}	{B}	{A,C}	{A,C}
1	1	0	1	0	0

To fill X_{13} we consider all bodies in $X_{11}X_{23} \cup X_{12}X_{33}$

This set of strings is $\{B\}\{S,B\} \cup \{A\}\{A,C\}$

$= \{BS, BB, AA, AC\}$

Only BB and AC are present in the RHS of some productions & their LHS are A, C respectively

Hence $X_{13} = \{A, C\}$

Continuing in similar fashion as X_{16} contains {S} we conclude that $110100 \in L(G)$

Pumping Lemma for Context-Free Languages

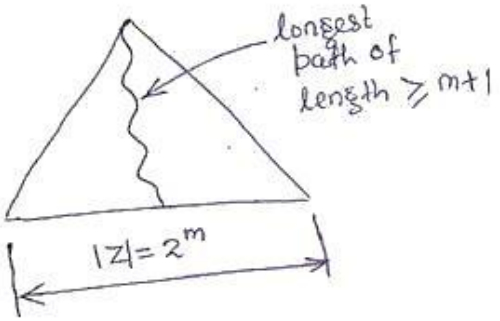
Let L be a CFL. Then there exists a constant n such that if z is any string in L such that $|z|$ is at least n, then we can write $z = uvwxy$, subject to the following conditions:

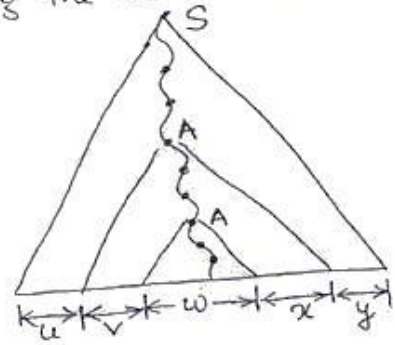
1. $|vwx| \leq n$. That is, the middle portion is not too long.
2. $vx \neq \epsilon$. Since v and x are the pieces to be "pumped", this condition says that at least one of the strings we pump must not be empty.
3. For all $i \geq 0$, uv^iwx^iy is in L. That is two strings v & x may be pumped any no of times, resulting string still be a member of L

Proof The proof of pumping lemma is based on another theorem whose statement is as follows:

> Suppose we have a parse tree according to a CNF grammar $G=(V,T,P,S)$, and suppose that the yield of the tree is a terminal string w. If the length of the longest path is n, then $|w| \leq 2^{n-1}$

Let G be a CNF grammar for the given CFL, L.
Let G has m non-terminals
Consider a derivation tree in G of a string z in L of length 2^m or more i.e. $n = 2^m$ where n is the length of z.
The derivation for z must have a path of length greater than equal to m+1. [Using the above theorem statement]





In the longest path there will be m+1 or more non terminals. But there are only m non-terminals That means in the longest path at least one of the non terminal is get repeated. Let the repeating non terminal is A.

Consider the longest path. Start from the leaf and stop when a non-terminal (A) repeats for the first time.

From the picture

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

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

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

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

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

From the above

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \Rightarrow uvvAxxy \Rightarrow \cdots \Rightarrow uv^iAx^iy \Rightarrow uv^iwx^iy$$

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

Hence $uv^iwx^iy \in L$ for all $i \geq 0$. So condition 3 is proved.

If $|vx| = 0$ i.e. both v and x is empty then

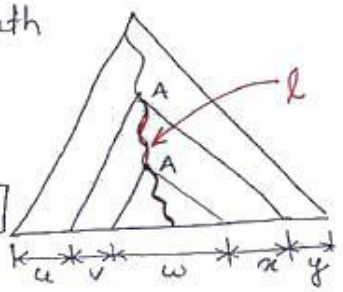
$$A \overset{*}{\Rightarrow} vAx \Rightarrow A$$

This is not possible since CNF does not contain unit productions. Hence condition 2 is proved.

Now to proove condition 1

The string vwx is the yield of the subtree rooted at A

Let the length of the longest path of that derivation tree be l.



Length of $l \leq m+1$ [$\because$ length of longest path of the whole tree $\geq m+1$]

As per the statement of the theorem stated at the begening of the proof

The string vwx can't be of length $> 2^{m+1-1} = 2^m = n$

Hence condition 1 is proved.

Application of Pumping Lemma for CFLs

We use the pumping lemma as an adversary game to proove a language L is context free or not in the following way:

1. Pick up a language L that we want to show is not a CFL.
2. Our adversary gets to pick n, which we do not know, and we therefore must plan for any possible n.
3. We get to pick Z, and may use n as a parameter when we do so.
4. Our adversary gets to break Z into $uvwxy$ such that $|vwx| \leq n$ and $vx \neq \epsilon$
5. We win the game, if we can, by picking i and showing that $uv^iwx^iy \notin L$.

Example 4: Design a TM for the following language

$L = \{\omega \mid \omega \in \{1\}^+ \text{ and } |\omega| = 3n, \forall n \geq 1\}$

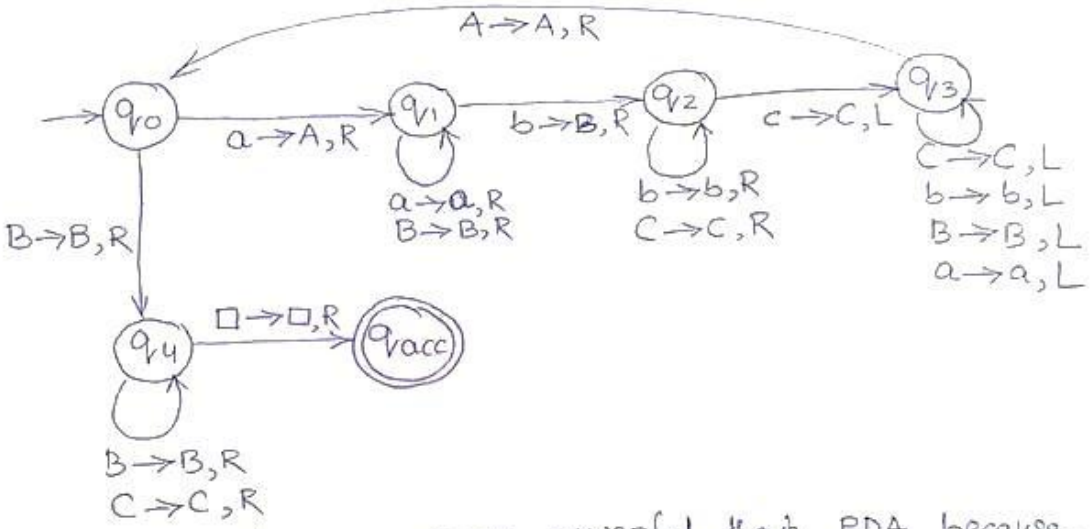
Solu^n $Q = \{q_0, q_1, q_2, q_3, q_\pi, q_f\}$

The transitions are

$\delta(q_0, 1) = (q_1, 1, R)$
$\delta(q_1, 1) = (q_2, 1, R)$
$\delta(q_2, 1) = (q_3, 1, R)$
$\delta(q_3, 1) = (q_1, 1, R)$

$\delta(q_0, B) = (q_\pi, B, R)$
$\delta(q_1, B) = (q_\pi, B, L)$
$\delta(q_2, B) = (q_\pi, B, L)$
$\delta(q_3, B) = (q_f, B, L)$

Example 5: TM for $\{a^n b^n c^n \mid n > 0\}$



NOTE Turing Machine is more powerful than PDA because it can recognize some languages that are not context free

Eg: TM can recognize $\{a^n b^n c^n \mid n > 0\}$ which is not a context free language.

Turing Computable

A function f with domain D is said to be Turing-computable or computable if there exists some Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ such that

$$q_0 \omega \overset{*}{\underset{M}{\vdash}} q_f f(\omega), \quad q_f \in F$$

for all $\omega \in D$

NOTE: All mathematical functions are turing computable.

Example 6: Given two positive integers x and y, design a TM that computes $x+y$.

Solution We use unary notation in which any positive integer x is represented by $w(x) \in \{1\}^+$, such that

$$|w(x)| = x$$

For eg 4 is represented as 1111

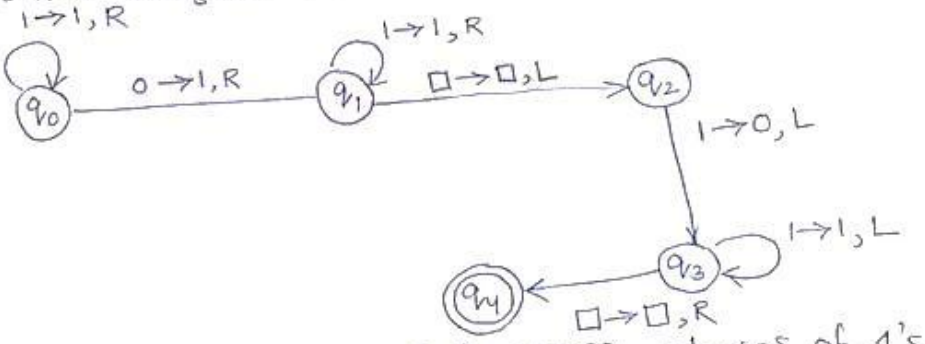
We assume that the input $w(x)$ and $w(y)$ are on the tape in unary notation separated by a single 0, and the R/W head will be positioned at the left end of the result. After the computation $w(x+y)$ will be on the tape followed by a single 0. So we want to design a TM for performing the following computation:

$$q_0 w(x) 0 w(y) \vdash^* q_f w(x+y) 0$$

where q_0 is the initial state & q_f is the final state

$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ with $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_4\}$

The transition diagram is as follows:



Example 7: Design a TM that copies string of 1's

Soln So we need a TM that performs the following computation

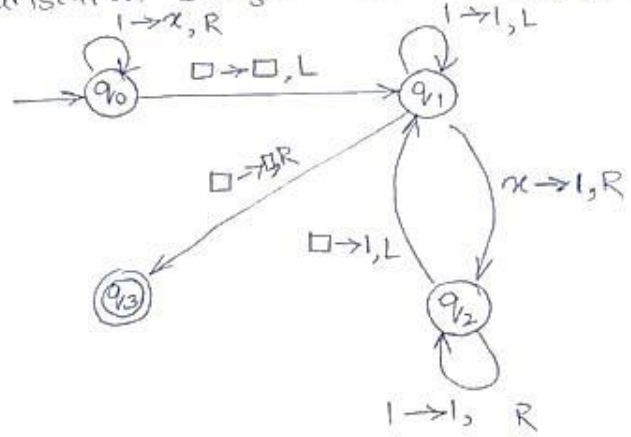
$$q_0 w \vdash^* q_f ww$$

We implement the following process:

1. Replace every 1 by an x
2. Find the rightmost x and replace it with 1
3. Travel to the right end of current nonblank region and create a 1 there
4. Repeat steps 2 and 3 until there are no more x's

$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ with $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_3\}$

Transition Diagram is as follows:



Example 8 Let x and y be two positive integers represented in unary notation. Construct a TM that will halt in a final state q_y if $x \geq y$ and will halt in a nonfinal state q_n if $x < y$.

Soln We need a TM to perform the following computation

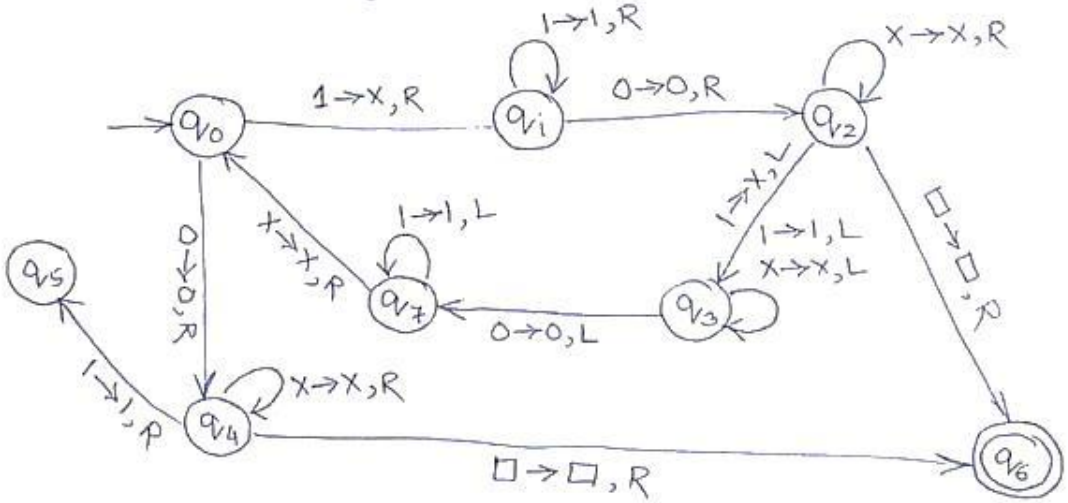
$q_0 w(x) 0 w(y) \vdash^* q_y w(x) 0 w(y)$ if $x \geq y$

$q_0 w(x) 0 w(y) \vdash^* q_n w(x) 0 w(y)$ if $x < y$

$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ with $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$

$F = \{q_6\}$

Transition Diagram is as follows:



NOTE: The above machine halts in the accepting or non accepting state without reaching the start of the input. Also it does not change X's back to 1.

Variants of TM

1. Turing Machines with a stay-Option

We define a TM with stay option by redefining δ as

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

Theorem: The class of TM with stay option is equivalent to the class of standard TMs.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a TM with stay-option and $\hat{M} = (\hat{Q}, \Sigma, \Gamma, \hat{\delta}, q_0, \square, \hat{F})$ be its equivalent standard TM

For each transition L or R type

$$\delta(q_i, a) = (q_j, b, L \text{ or } R)$$

we put into $\hat{\delta}$

$$\hat{\delta}(\hat{q}_i, a) = (\hat{q}_i, b, L \text{ or } R)$$

For each S-transition

$$\delta(q_i, a) = (q_j, b, S)$$

we put into $\hat{\delta}$ the corresponding transitions

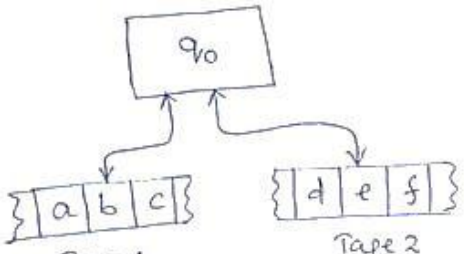
$$\hat{\delta}(\hat{q}_i, a) = (\hat{q}_{ij}, b, R)$$

and $$\hat{\delta}(\hat{q}_{ij}, c) = (\hat{q}_j, c, L)$$

for all $c \in \Gamma$

2. Multitape Turing Machine

A multitape TM is a TM with several tapes, each with its own independently controlled read-write head as shown in the figure.



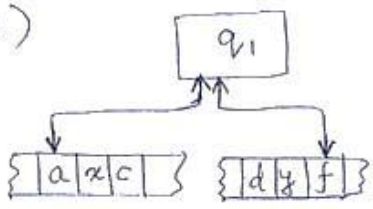
Def^n A n-tape TM, $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ where $Q, \Sigma, \Gamma, q_0, F$ are same as in the def^n of standard TM, but δ is defined as

$$\delta : Q \times \Gamma^n \to Q \times \Gamma^n \times \{L, R\}^n$$

Eg: for $n = 2$ with a current configuration shown in the diagram

$$\delta(q_0, b, e) = (q_1, x, y, L, R)$$

After the transition the machine will be a new configuration shown in the diagram shown here.



Example Consider the language $\{a^n b^n\}$. Using a two-tape machine our job will be much easier. Assume that the input string is written on tape 1 at the beginning of the computation. First read all the a's and copy them onto tape 2. When we reach the end of the a's, we match the b's on tape 1 against the copied a's on tape 2. So we don't have to repeat back-and-forth movement of the read-write head.

Theorem: Every multitape TM has an equivalent single tape TM.

3. Nondeterministic Turing Machine

A nondeterministic TM is an automaton as given by the standard def^n of TM except that δ is now a function

$$\delta : Q \times \Gamma \longrightarrow 2^{Q \times \Gamma \times \{L, R\}}$$

Theorem: The class of deterministic TM and the class of nondeterministic TM are equivalent.

Def^n A nondeterministic TM, M is said to accept a language L if, for all $\omega \in L$, at least one of the possible configurations accepts ω.

Def^n A nondeterministic TM, M is said to decide a language L if for all $\omega \in \Sigma^*$, there is a path that leads either to acceptance or rejection

4. Universal TM

A universal Turing Machine M_u is an automaton that, given as input the description of any Turing machine M and a string ω, can simulate the computation of M on ω.

Construction of M_u

To construct M_u, we choose a way to describe TM called encoding scheme. The encoding scheme is described as follows:

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be the TM we want to encode

Without loss of generality let us assume

$$Q = \{q_1, q_2, \cdots, q_n\}$$

with $q_0 = q_1$ and $F = \{q_2\}$

and $\Gamma = \{a_1, a_2, \cdots, a_n\}$ where a_1 represents $\square$

We encode q_1 as 1, q_2 as 11 and so on.

Similarly we encode a_1 as 1, a_2 as 11 and so on.

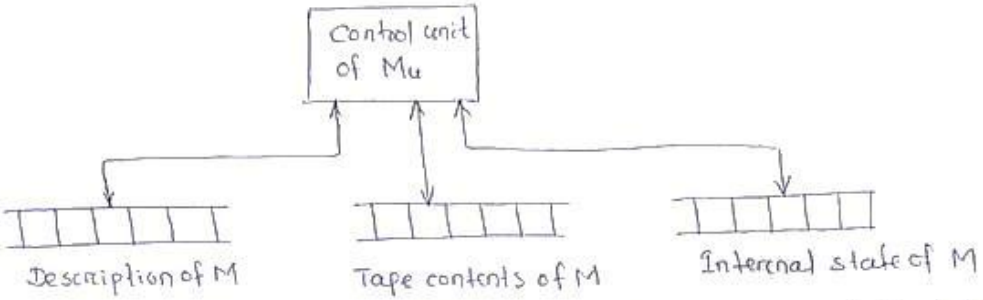
The symbol 0 is used as separator

R/W head moving direction L is encoded as 1 and R is encoded as 11

The transition functions can be described as

$\delta(q_1, a_2) = (q_2, a_3, L) \Rightarrow \ldots 10110110111010\ldots$

A universal TM, Mu has an input alphabet that includes $\{0,1\}$ and its structure is as follows:



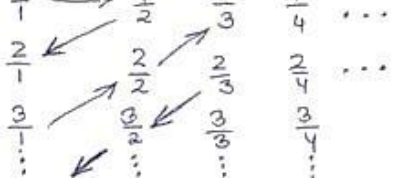
For any input M and w, tape 1 will keep an encoded definition of M. Tape 2 will contain the tape contents of M, and tape 3 the internal state of M. Mu looks first at the contents of tape 2 and 3 to determine the configuration of M. It then consults tape 1 to see what M would do in this configuration. Finally tapes 2 and 3 will be modified to reflect the result of the move.

Countable / Uncountable Set

A set is said to be countable if its elements can be put into a one-to-one correspondence with the positive integers That means the elements of the set can be written in some order, say $x_1, x_2, \ldots$, so that every element of the set has some finite index.

Eg: 1) Set of even numbers 0, 2, 4, ... are countable as the number $2n$ is present at index/position $n+1$.

2) Set of all quotients of the form p/q where p, q are positive integers is also countable as follows:



Eg: 1/3 occurs in the sixth place

The method of showing that elements can be written in some sequence is called enumeration procedure.

Linear Bounded Automata (lba)

Although it is not possible to extend the powers of standard TM by complicating the tape structure, it is possible to limit it by restricting the way in which the tape can be used

Eg: A PDA can be regarded as a nondeterministic TM with a tape that is restricted to being used like a stack.

We can limit the tape use as :

Allow the machine to use only that part of the tape occupied by the input.

This generates another class of machines called linear bounded automata
To enforce the usable part of the tape to exactly the cells taken by input we envision the input as bracketed by two special symbols, the left-end marker [and the right end marker].
So the initial configuration of TM is given by the ID $q_0[w]$
The R/W head can't move to the left of [or to the right of]

Defn A linear bounded automaton is a nondeterministic TM $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ s.t. the restriction that Σ must contain two special symbols [and], such that $\delta(q_i, [)$ can only contain elements of the form $(q_j, [, R)$ and $\delta(q_i,])$ can contain only elements of the form $(q_j,], L)$.

Defn A string w is accepted by a linear bounded automaton if there is a possible sequence of moves

$$q_0[w] \overset{*}{\vdash} [x_1 q_f x_2]$$

for some $q_f \in F$, $x_1, x_2 \in \Gamma^*$

The language accepted by the lba is the set of all such accepted strings.

Recursively Enumerable Language / Turing Recognizable Language

A language L is said to be recursively enumerable if there exists a Turing Machine that accepts it.

The definition says there exists a TM M, such that for every $\omega \in L$,

$$q_0 \omega \vdash_M^* x_1 q_f x_2,$$

with initial state q_0 and final state q_f

NOTE: The above definition says nothing about what happens for ω not in L. The machine may halts in a non final state or that it never halts and goes into an infinite loop.

Recursive Language / Turing Decidable Language

A language L on Σ is said to be recursive if there exists a TM M that accepts L and halts on every ω in Σ^+.

Ref A language is recursive if and only if there exists a membership algorithm for it.

Languages That are Not Recursively Enumerable

Theorem: Let S be an infinite countable set. Then its powerset 2^S is not countable

Proof

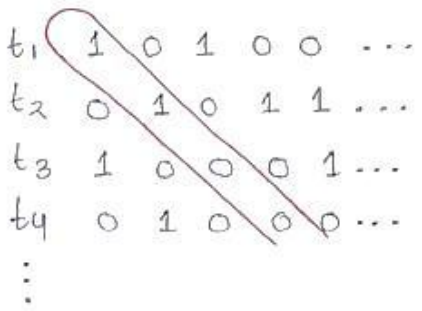
Let $S = \{s_1, s_2, s_3, \cdots\}$

We can represent the element t of 2^S by a binary string with a 1 in position i if and only if s_i is in t.

Example: Set $\{s_1, s_3, s_4\}$ is represented by 101100...

Note that any element of 2^S can be represented by a binary sequence and any such sequence represents a unique element of 2^S.

Let us assume that 2^S ~~is~~ are countable. Then the elements of 2^S can be written in some order, say $t_1, t_2, \cdots$ and let us enter these into a table as shown below.



After complement the elements in the diagonal 1100... will become 0011...

Let us take the elements in the main diagonal and complement each entry.

The new sequence along the diagonal represents some element of 2^S, say t_i for some i.

But definitely $t_i \neq t_i$ because t_i differs from t_1 through s_1, similarly t_i differs from t_2 through s_2, differs from t_3 through s_3 and so on

This is a contradiction since t_i is not equal to any of the t's.

Hence 2^S is not countable.

NOTE: This kind of argument which involves a manipulation of the diagonal elements of a table is called diagonalization.

Theorem

For any nonempty Σ, there exist languages that are not recursively enumerable

Theorem

There exists a recursively enumerable language whose complement is not recursively enumerable

Theorem

If a language L and its complement $\bar{L}$ are both recursively enumerable, then both languages are recursive

If L is recursive, then $\bar{L}$ is also recursive, and consequently both are recursively enumerable.

Proof Let L and $\bar{L}$ are both recursively enumerable languages. So there exists Turing Machine M and $\hat{M}$ for L and $\bar{L}$ respctively.

M produces $\omega_1, \omega_2, \ldots$ in L and $\hat{M}$ produces $\hat{\omega}_1, \hat{\omega}_2, \ldots$ in $\bar{L}$.

Let $\omega \in \Sigma^+$ is a given string.

First let M generate ω_1 and compare it with ω If they are not same we let $\hat{M}$ to generate $\hat{\omega}_1$ and compare with ω again. Next let M generate ω_2 then ~~Next~~ $\hat{M}$ will generate ω_2 and again we compare with ω again and the process will continue

Any $\omega \in \Sigma^+$ will be generated by either M or $\hat{M}$. so it will get a match.

If the string is produced by M, ω belongs to L otherwise it is in $\bar{L}$.

The process is a membership algorithm for both L and $\bar{L}$. Hence both L and $\bar{L}$ are recursive

To proove the converse of the theorem let us assume L is recursive. Then there exists a membership algorithm for it. But this algorithm can also be used for $\bar{L}$ by complementing its conclusion. Therefore $\bar{L}$ is recursive.

As any recursive language is recursively enumerable, L & $\bar{L}$ are recursively enumerable.

Theorem

There exists a recursively enumerable language that is not recursive. ie/ the family of recursive languages is a proper subset of the family of recursively enumerable languages.

Unrestricted Grammars

A grammar $G=(V,T,P,S)$ is called unrestricted if all productions are of the form

$u \to v$

where u is in $(VUT)^+$ and v is in $(VUT)^*$

NOTE: Unrestricted grammars are much more powerful than restricted forms like the regular and CFGs

Theorem: Any language generated by an unrestricted grammar is recursively enumerable

Theorem: For every recursively enumerable language L, there exists an unrestricted grammar G, such that $L=L(G)$

Context-Sensitive Grammar

A grammar $G=(V,T,P,S)$ is said to be context-sensitive if all productions are of the form

$x \to y$

where $x,y \in (VUT)^+$ and $|x| \leq |y|$

Context-Sensitive Language

A language L is said to be context-sensitive if there exists a context-sensitive grammar G such that $L=L(G)$ or $L=L(G) \cup \{\in\}$

Example: The language $L=\{a^n b^n c^n : n \geq 1\}$ is a context-sensitive language. For L the context-sensitive grammar is:

$S \to abc \mid aAbc$
$Ab \to bA$
$Ac \to Bbcc$
$bB \to Bb$
$aB \to aa \mid aaA$

Eg: Derivation of $a^3b^3c^3$

$S \Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc$
$\Rightarrow aBbbcc \Rightarrow aaAbbcc$
$\Rightarrow aabAbcc \Rightarrow aabbAcc$
$\Rightarrow aabbBbccc \Rightarrow aabBbbccc$
$\Rightarrow aaBbbbccc \Rightarrow aaabbbccc$

Theorem: For every context-sensitive language L not including $\in$, there exists some linear bounded automaton M such that $L=L(M)$

Theorem: If a language L is accepted by some linear bounded automaton M, then there exists a context-sensitive grammar that generates L.

Theorem: Every context-sensitive language L is recursive

Theorem: There exists a recursive language that is not context-sensitive

The Chomsky Hierarchy

We have now encountered a number of language families:

- Recursively Enumerable languages (L_{RE})
- Context-sensitive languages (L_{CS})
- Context-free languages (L_{CF})
- Regular languages (L_{REG})

Noam Chomsky, a founder of formal languages theory provided an initial classification into four language types:

Type 0 : L_{RE}, Type 1 : L_{CS}, Type 2: L_{CF}, Type 3: L_{REG}

The relationship among these type of languages given by Chomsky is shown below:

Original Chomsky Hierarchy

Including the family of deterministic context-free languages (L_{DCF}) and recursive languages (L_{REC}) we found the following extended hierarchy:

Extended Chomsky Hierarchy

We know previously the context-free language

$$L = \{ \omega : n_a(\omega) = n_b(\omega) \}$$

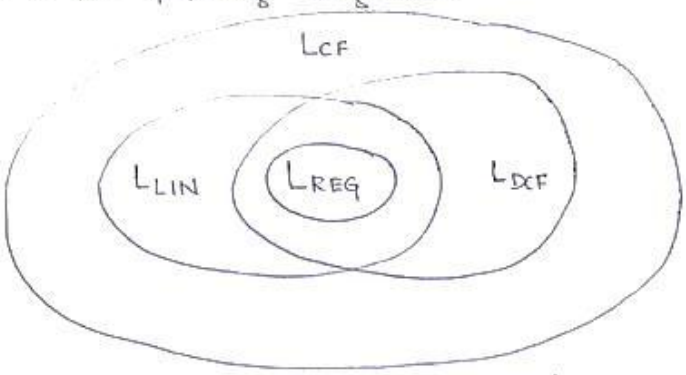
is deterministic, but not linear

Similarly, the language

$$L = \{a^n b^n\} \cup \{a^n b^{2n}\}$$

is linear but not deterministic

So the relationship between regular, linear, deterministic context free and non-deterministic context-free languages is shown in the following diagram:



Closure Properties of Recursive and Recursively Enumerable Languages

Operation	Recursive	Recursively Enumerable
Union	✓	✓
Concatenation	✓	✓
Star	✓	✓
Intersection	✓	✓
Complement	✓	X

DECIDABLE LANGUAGES

1. $A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$

Theorem A_{DFA} is a decidable language

Proof: To prove A_{DFA} is decidable we need to present a TM M that decides A_{DFA}.

We have to construct a machine M which will halt on input $\langle B, w\rangle$. It should halt in an accept state if B accepts w, otherwise in a reject state.

The description of machine M is as follows:

M = " On input $\langle B, w\rangle$, where B is a DFA & w is a string

1. Simulate B on input w
2. If the simulation ends in an accept state, accept. If it ends in a nonaccepting state, reject."

When the machine M receives an input $\langle B, w\rangle$ it first checks on whether B and w are represented properly or not. If not M rejects it.

Then M can carry out the simulation in the following way:

- It keeps track of B's current state and current position in w by writing this information down on its tape
- The state and position is updated according to the specified transition function δ
- When M finishes processing the last symbol of w, M accepts the input if B is in an accepting state M rejects the input if B is in a nonaccepting state.

2. $A_{NFA} = \{\langle B, w\rangle \mid B \text{ is a NFA that accepts input string } w\}$

Theorem: A_{NFA} is a decidable language

Proof: We present a TM N that decides A_{NFA} as follows:

N = " On input $\langle B, w \rangle$ where B is an NFA and w is a string

1. Convert NFA B to an equivalent DFA C using the procedure for NFA to DFA conversion
2. Run TM M from previous theorem on input $\langle C, w \rangle$
3. If M accepts, accept otherwise reject."

NOTE: Running TM M in step 2 means incorporating M into the design of N as a subprocedure

3. $A_{REX} = \{ \langle R, w \rangle \mid R$ is a regular expression that generates string $w \}$

Theorem: A_{REX} is a decidable language

Proof: The following TM P decides A_{REX}

P = " On input $\langle R, w \rangle$ where R is a regular expression and w is a string

1. Convert the regular expression R to an equivalent DFA by the procedure for RE to DFA conversion
2. Run TM M on input $\langle A, w \rangle$
3. If M accepts, accept; if M rejects reject"

4. $E_{DFA} = \{ \langle A \rangle \mid A$ is a DFA and $L(A) = \phi \}$

Theorem: E_{DFA} is a decidable language

Proof: A DFA accepts some string if and only if it can reach an accept state from the start state by travelling along the arrows of the DFA. To test this we design the following TM T:

T = " On input $\langle A \rangle$ where A is a DFA

1. Mark the start state of A
2. Repeat until no new stages get marked:
3. Mark any state that has a transition coming into it from any state that is already marked
4. If no accept state is marked, accept otherwise reject."

5. $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates string } w \}$

Theorem: A_{CFG} is a decidable language

Proof: The TM S for A_{CFG} follows

S = " On input $\langle G, w \rangle$, where G is a CFG & w is a string

1. Convert G to an equivalent grammar in CNF
2. List all derivations with $2n-1$ steps, where n is the length of w, except if $n=0$ then $w = \epsilon$, so list all derivations with 1 step
3. If any of these derivations generates w, accept; if not reject."

The Halting Problem

Although we believe that computer is very powerful and it can solve all problems but actually it can't. There are some ordinary problems that people want to solve turn out to be computationally unsolvable

Eg: Software verification Problem is unsolvable by computer

Given a computer program and a precise specification of what the program is supposed to do. We need to verify that the program performs as specified

Let $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$

A_{TM} is also called as halting problem.

Theorem

A_{TM} is undecidable

Proof A_{TM} is Turing-recognizable. The following TM U recognizes A_{TM}.

U = " On input $\langle M, w \rangle$, where M is a TM and w is a string

1. Simulate M on input w
2. If M ever enters its accept state, accept; if M ever enters its reject state, reject."

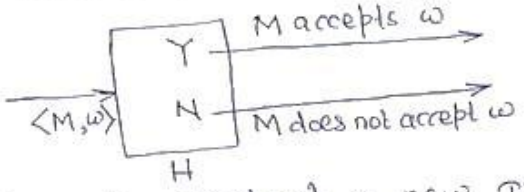
This machine loops on input $\langle M, w \rangle$ if M loops on w, which is why this machine does not decide A_{TM}

If the algorithm had some way to determine that M was

not halting on w, it could reject. Actually the algorithm has no way to make this determination

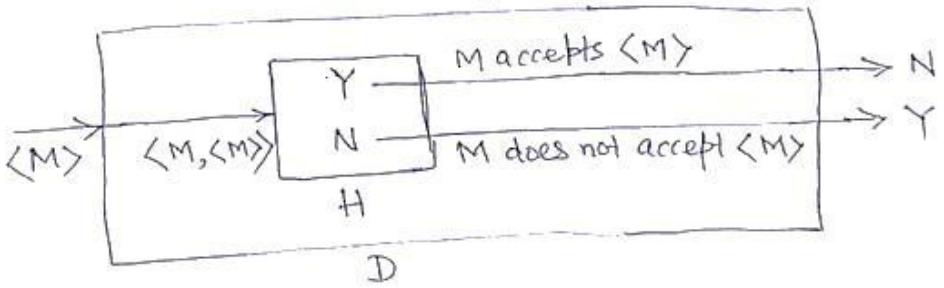
NOTE: U is an example of universal TM. U is called universal because it is capable of simulating other TM from its description.
Universal TM played an important early role in simulating the development of stored-program computers.

Let us assume that A_{TM} is decidable.
So there exists a halting TM, H such that $A_{TM} = L(H)$



Now we construct a new TM D with H as a subroutine as follows:

D = " On input <M>, where M is a TM
1. Run H on input <M, <M>>
2. If H accepts then reject
else accept.



Now consider what happens if D is given the input <D>

D accepts <D> $\Leftrightarrow$ H does not accept <D, <D>>
$\Leftrightarrow$ D does not accept <D>

which is a contradiction,
Thus neither D nor H exists.

Co-Turing Recognizable Language

A language is called co-Turing recognizable if its complement is Turing recognizable.

i.e. L is co-Turing recognizable if $\Sigma^* - L$ is Turing recognizable.

Theorem A language is decidable if it is both Turing-recognizable and co-Turing recognizable

i.e. A language is decidable if and only if it and its complement are Turing recognizable.

Proof

Else if part

Given: A language L is decidable

To show: L & $\bar{L}$ are Turing recognizable

As L is decidable $\bar{L}$ is also decidable since (decidable) recursive languages are closed under complementation.

Also we know that any decidable language is Turing-recognizable

So as L & $\bar{L}$ are decidable $\Rightarrow$ L & $\bar{L}$ are Turing-recognizable

If part

Given: L and $\bar{L}$ are Turing recognizable

To show: L is decidable.

As L and $\bar{L}$ are Turing recognizable let M_1 be the recognizer for L and M_2 be the recognizer for $\bar{L}$.

The following TM M is a decider for A

M = "On input w:

1. Run both M_1 & M_2 in parallel
2. If M_1 accept, accept; if M_2 accepts, reject"

Here M decides L. Every string w is either in L or $\bar{L}$ Therefore either M_1 or M_2 accepts w. Because M halts whenever M_1 or M_2 accepts, M always halts and so it is a decider. Also M accepts all strings in L and reject all strings not in L

So M is a decider for L and thus L is decidable.

Corollary $\overline{A_{TM}}$ is not Turing-recognizable

Proof We have already proved that A_{TM} is Turing-recognizable. If $\overline{A_{TM}}$ also were turing-recognizable then A_{TM} would be decidable but it has already been proved that A_{TM} is not decidable so $\overline{A_{TM}}$ must not be turing recognizable.

Reducibility

Defⁿ A function $f: \Sigma^* \to \Sigma^*$ is said to be computable if $\exists$ a TM M with output s.t. $\forall x \in \Sigma^*$, M halts with $f(x)$ written on its output tape

Reduction : Defⁿ

Let $L_1, L_2 \subseteq \Sigma^*$. We say that L_1 reduces to L_2 ($L_1 \leq_m L_2$) if $\exists$ a computable function f s.t. $\forall x \in \Sigma^*$, $x \in L_1 \iff f(x) \in L_2$ [In $L_1 \leq_m L_2$ m stands for many to one]

Reduction is a way of converting one problem into another problem in such a way that a solution to the second problem can be used to solve the first problem

$HALT_{TM} = \{ \langle M, w \rangle \mid M$ is a TM and halts (by accepting or rejecting) on a given input $\}$

Theorem : $HALT_{TM}$ is undecidable

Proof We will use the undecidability of A_{TM} to prove the undecidability of $HALT_{TM}$ by reducing A_{TM} to $HALT_{TM}$

By contradiction let us assume that $HALT_{TM}$ is decidable. Let TM R decides $HALT_{TM}$. We construct TM S to decide A_{TM} which operates as follows:

S = " On input $\langle M, \omega \rangle$, an encoding of a TM M and a string ω

1. Run TM R on input $\langle M, \omega \rangle$
2. If R rejects, reject
3. If R accepts, simulate M on ω until it halts
4. If M has accepted, accept; if M has rejected, reject

Clearly if R decides $HALT_{TM}$, then S decides A_{TM}. But A_{TM} is undecidable so $HALT_{TM}$ is also undecidable.

Corollary

1. If $L_1 \leq_m L_2$ and L_2 is decidable then L_1 is also decidable
2. If $L_1 \leq_m L_2$ and L_1 is undecidable then L_2 is also undecidable
3. If $L_1 \leq_m L_2$ and L_1 is not turing recognizable then L_2 is also not turing recognizable.

$E_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \phi \}$

Theorem

E_{TM} is undecidable.

Post Correspondence Problem (PCP)

An instance of the PCP is a collection P of dominos:

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \cdots \left[\frac{t_k}{b_k}\right] \right\}$$

and a match is a sequence $i_1, i_2, i_3, \ldots, i_l$ where

$$t_{i_1}, t_{i_2}, \ldots, t_{i_l} = b_{i_1}, b_{i_2}, \ldots, b_{i_l}.$$

The problem is to determine whether P has a match

$PCP = \{ \langle P \rangle \mid P \text{ is an instance of Post Correspondence Problem with a match} \}$

Example 1

$$P = \left\{ \left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right] \right\}$$

P has a match. The matching sequence is i_2, i_1, i_3, i_2, i_4

Example 2

$A = \{ \omega_1 = 11, \omega_2 = 1, \omega_3 = 100, \omega_4 = 111 \}$

$B = \{ v_1 = 111, v_2 = 00, v_3 = 001, v_4 = 11 \}$

Pair (A, B) has a match. The matching sequence is i_1, i_3, i_4

NOTE: For some collection of dominos finding a match may not be possible.

Eg: $P = \left\{ \left[\frac{abc}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{acc}{ba}\right] \right\}$

Here every top string is longer than corresponding bottom string, hence match is not possible

Theorem PCP is undecidable.

Modified Post Correspondence Problem (MPCP)

Given a collection of dominos P. We say P has a modified post correspondence solution if there exists a sequence $i_1, i_j \cdots i_k$ such that

$$t_{i_1}, t_{i_j} t_{i_j} \cdots t_{i_k} = b_{i_1}, b_{i_j} b_{i_j} \cdots b_{i_k}$$

MPCP = $\{ \langle P \rangle \mid$ P is an instance of PCP with a match that starts with the first domino$\}$

Example:

i	1	2	3	4
A	11	1	0111	10
B	1	111	10	0

This MPCP has a solution 3,2,2,4 as

$$t_1 t_3 t_2 t_2 t_4 = b_1 b_3 b_2 b_2 b_4 = 1101111110$$

Theorem MPCP is undecidable

MODULE - IV

Computable Functions :

Any func^n for which there is an effective procedure (an algorithm) is a computable func^n

- The concept of computing has been intuitively linked with concept of functions.
- A computing machine can only be designed for the functions which are computable.
- Given a recursive language L and a string ω over Σ^*, the characteristic function f is given by

$$f(\omega) = \begin{cases} 1 & , \text{if } \omega \in L \\ 0 & , \text{if } \omega \notin L \end{cases}$$

- If the language L is not recursive then the function f may not be computable
- The key factor behind designing of a computing machine is that it has to be designed for a function which is computable.

Primitive Recursive Functions

I : set of non-negative integers

Basic Func^{n}s / Initial Func^{n}s

1. Zero Function $z(x) = 0$, for all $x \in I$ Eg: $z(10) = 0$, $z(21) = 0$
2. Successor function $s(x) = x + 1$
   Eg: $s(4) = 5$, $s(10) = 11$
3. Projector Function
   $p_k(x_1, x_2) = x_k$, $k = 1, 2$

Complicated Func^{n}s

4. Composition, by which we construct
   $f(x, y) = h(g_1(x, y), g_2(x, y))$
   from defined functions g_1, g_2, h
5. Primitive recursion, by which a function can be defined recursively through
   $f(x, 0) = g_1(x)$
   $f(x, y+1) = h(g_2(x, y), f(x, y))$
   from defined functions g_1, g_2 and h

Basic func^n / Initial func^n

6. Constant function
   $c(x_1, x_2, \cdots, x_k) = a$
   Eg: $c(x) = 1$.

Example: Addition of integers x, y

$$add(x,0) = x$$
$$add(x, y+1) = add(x,y) + 1$$

Example: Multiplication of integers x, y

$$mult(x,0) = 0$$
$$mult(x, y+1) = add(x, mult(x,y))$$

Here the second step is an application of primitive recursion in which h is ~~defined~~ identified with the add function, and $g_2(x,y)$ is the projector function $p_1(x,y)$

Example: subtraction of integers x, y

As negative numbers are not permitted we define an alternative operator $\dot{-}$ called as monus as follows:

$$x \dot{-} y = x - y \text{ if } x \geq y$$
$$x \dot{-} y = 0 \text{ if } x < y$$

We also define the predecessor function

$$pred(0) = 0$$
$$pred(y+1) = y = p_1(y, pred(y)) = p_1(p_1(y), pred(y))$$

Now we define subtraction as follows:

$$subtr(x,0) = x$$
$$subtr(x, y+1) = pred(subtr(x,y))$$

Eg: subtr(3,2) = pred(subtr(3,1))
= pred(pred(subtr(3,0))
= pred(pred(3))
= pred(2)
= 1

Primitive Recursive function: defn

A function is called primitive recursive if and only if it can be constructed from the basic functions z, s, p_k by successive composition and primitive recursion.

1. addition

$add(x,0) = x$ [Here g_1 is the projector func^n $p_1(x_1,x_2)=x$

$add(x,y+1) = add(x,y)+1 = s(add(x,y))$ [Here h is the successor func^n & there is no g_2

2. Multiplication

$mult(x,0) = 0$ [Here g_1 is the constant func^n c

$mult(x,y+1) = add(x, mult(x,y))$ [Here h is add & g_2 is projector

$= add(p_1(x,y), mult(x,y))$ [func^n p_1

3. $exp(x,y) = x^y$

$exp(x,0) = 1$ [Here g_1 is the constant func^n]

$exp(x,y+1) = mult(x, exp(x,y))$

$= mult(p_1(x,y), exp(x,y))$ [Here g_2 is p_1 & h is mult

4. $fac(x) = x!$

$fac(0) = 1$ [Here g_1 is the constant func^n]

$fac(x+1) = mult(x+1, fac(x))$

$= mult(s(x), fac(x))$ [Here h is mult & g_2 is s]

NOTE: Most common functions are primitive recursive. However there are functions which are not primitive recursive. Eg: Ackermann's function

Ackermann's function

Ackermann's function is a function from $I \times I$ to I defined by

$A(0,y) = y+1$

$A(x,0) = A(x-1,1)$

$A(x,y+1) = A(x-1, A(x,y))$

Example :

$A(1,1) = A(0, A(1,0)) = A(0, A(0,1)) = A(0,2) = 3$

$A(1,2) = A(0, A(1,1)) = A(0,3) = 4$

$A(1,3) = A(0, A(1,2)) = A(0,4) = 5$

$A(2,2) = A(1, A(2,1)) = A(1,5) = A(0, A(1,4)) = A(0,6) = 7$

$A(2,1) = A(1, A(2,0)) = A(1, A(1,1)) = A(1,3)$

$= A(0, A(1,2)) = A(0,4) = 5$

$A(1,4) = A(0, A(1,3)) = A(0,5) = 6$

NOTE :

Ackermann's function is not primitive recursive because of the following reasons :

1. The definition of Ackermann's function is not in the form required by primitive recursion
2. There is a limit to how fast a primitive recursive function $f(n)$ can grow as $n \to \infty$, and Ackermann's function violates that limit. Let f be any primitive recursive function then we have $A(k,i) > f(i)$ for all $i = n, n+1, n+2, \cdots$.

When k becomes 3 the growth rate is exponential

$A(1,y) = y+2$

$A(2,y) = 2y+3$

$A(3,y) = 2^{y+3} - 3$

μ - Recursive function

- All the primitive recursive functions are computable total functions. However the reverse is not true.
- A given function is not primitive recursive does not guarantee that the function is not computable
- How to ensure that a given function is computable

To understand μ-Recursive Function we need the following functions:

Partial Function

A function f is said to be a partial function from domain X to range Y if for every argument in X, f assigns at most one value in the set Y.

Eg: Let N be the set of non-negative integers, 0, 1, 2, ...

A function $f : N \to N$

$$f(x) = x - 5$$

is a partial function because it is not defined for $x < 5$

Total Function

A function g is said to be total function from domain X to range Y if for every argument in X, g assigns exactly one value in the set Y.

Eg: The function $g : N \to N$

$$g(x) = x + 5$$

is a total function because it is defined for all $x \in N$

μ-Recursive Function

In the μ-recursive function, μ is known as the minimization operator defined on a total function $f(x,y)$ as follows:

$$\mu y(f(x,y)) = \text{smallest } y \text{ such that } f(x,y) = 0$$

The function $\mu y(f(x,y))$ may not be a total function. It may be a partial function. the function $\mu y(f(x,y))$ is also referred to as $M_f(x)$ function

Formal Def^n of μ-Recursive function

Given a total function $f : N^{k+1} \to N$ with the set of arguments $(x_1, x_2, \ldots, x_k, y)$, the function $M_f : N^k \to N$ with the set of arguments $(x_1, x_2, \ldots, x_k)$ is a μ-recursive function defined as:

$M_f(x_1, x_2, \cdots, x_k) = \mu y(f(x_1, x_2, \cdots x_k, y))$ which provides the smallest value of y for which $f(x_1, x_2, \cdots, x_k, y) = 0$

Total function

CASE 1 : When $x = 0$

CASE 2 :

Theorem

Godel Numbering

- Every positive integer can be factored into a unique set of prime factors
  Eg: $50 = 2\times5\times5$, $70 = 2\times5\times7$, $100 = 2\times2\times5\times5$
- It is possible to assign a serial number to each prime number:
  1 to 2, 2 to 3, 3 to 5, 4 to 7, 5 to 11, ...
- Formal Defn: For any finite sequence $(x_0, x_1, x_2, \cdots, x_n)$ the associated Godel number G_n is
  $$G_n(x_0, x_1, x_2, \cdots, x_n) = 2^{x_0} 3^{x_1} 5^{x_2} \cdots (P_n)^{x_n}$$
  where P_n is n^{th} prime number
  Eg: $7000 = 2^3 \times 5^3 \times 7^1 = 2^3 \times 3^0 \times 5^3 \times 7^1$
  We say the Godel number associated with the sequence $(3, 0, 3, 1)$ is 7000
- Godel number is an n-argument function where each argument is the exponent for the corresponding prime number.
- Two sequences representing a Godel number are identical if they differ only in terms of number of trailing zeros.
  Eg: $(2, 3, 5)$ and $(2, 3, 5, 0, 0)$ represents the same Godel number.
- The Godel number function $G_n : N^m \to N$ is primitive recursive. Each entry in the argument list is an exponent function $f_i(P_i, x)$ which is primitive recursive As G_n is the composition of exponent function, is also primitive recursive.
- Assuming no trailing zeros in sequences, each Godel number encodes a unique sequence and vice versa
- Godel number allows us to represent the different configurations of a multi-parameter-dependent entity/object in the form of a unique number
  Eg: Let entity E depends on three parameters x, y & z. Let x exists in 3 states, y exists in 4 states and z exists in 2 states. Then each possible state of $E(x, y, z)$ can be represented by a Godel number $G_n(x, y, z)$
- A TM at any instant, can exist in different configurations with each configuration defined by current state and location of the R/W head. It is possible to number each state and each cell and thereby encode every configuration by a Godel number. Similarly it is possible to define every transition function in the form of

Godel number.

- It is possible to assign Godel numbers to statements and formulae, thereby making it possible to manipulate numbers using numbers. We do the same thing in TM which indicates the acceptability of the binary string ω by a TM coded in binary string.

Church-Turing Thesis

- The Church-Turing thesis proposed by Alonzo Church and Alan Turing in 1936.
- It talks about mechanical calculation devices and the kind of calculations they can perform
- The statement of the thesis is
  Every function which would naturally be regarded as computable can be computed by a TM
- Some more definitions of the Church-Turing Thesis are as follows:
  1. Any mechanical computation can be performed by a TM.
  2. For every computable problem there is a corresponding TM.
  3. TMs can be used to model any mechanical computer.
  4. The set of languages that can be decided by a TM is the same as that which can be decided by any mechanical computing machine
  5. If there is no TM that decides problems P, there is no algorithm that can solve problem P.
- The Church-Turing Thesis is a conjecture which can't be proved or disproved. It is generally accepted because of the following reasons:
  - ✓ All the modifications proposed to the original TM in the form of multitape, multitrack and non deterministic versions are reducible to the original TM
  - ✓ Since the introduction of the TM, no one has proved any type of computation that ought to be included in the category of algorithmic process but can't be performed on a TM.

Introduction to Complexity Theory

Def^n: Let M be a Turing Machine

The Running Time of M is said to be $t: \mathbb{N} \to \mathbb{N}$ if $\forall x \in \Sigma^*$, M halts in at most $O(t(|x|))$ steps

The Space required by M is said to be $s: \mathbb{N} \to \mathbb{N}$ if $\forall x \in \Sigma^*$, M uses at most $O(s(|x|))$ cells in its workspace.

Remarks:

- We assume the 3-tape model of a TM and count the space used in the work tape only.
- We always count the amount of resource used as a function of the input length
- Since we use the O-notation we ignore multiplicative constants and lower order terms.

Time and Space Complexity Classes of Deterministic TM

Let $t: \mathbb{N} \to \mathbb{N}$. The time complexity class $\text{TIME}(t(n))$, is defined as

$$\text{TIME}(t(n)) = \{L : L \text{ is a language decided by an } O(t(n)) \text{ time deterministic TM}\}$$

Let $s: \mathbb{N} \to \mathbb{N}$. The space complexity class $\text{SPACE}(s(n))$ is defined as

$$\text{SPACE}(s(n)) = \{L : L \text{ is a language decided by an } O(s(n)) \text{ space deterministic TM}\}$$

Time and Space Complexity Classes of Non-deterministic TM

Def^n: Let N be a non-deterministic TM

The Running Time of N is said to be $t: \mathbb{N} \to \mathbb{N}$ if $\forall x \in \Sigma^*$, every computation path in N halts in at most $O(t(|x|))$ steps.

The Space required by N is said to be $s: \mathbb{N} \to \mathbb{N}$ if $\forall x \in \Sigma^*$, every computation of N uses at most $O(s(|x|))$ cells in its workspace.

Let $t: N \to N$. The time complexity class $NTIME(t(n))$ is defined as

$NTIME(t(n)) = \{ L : L$ is a language decided by an $O(t(n))$ time nondeterministic TM $\}$

Let $s: N \to N$. The space complexity class $NSPACE(s(n))$, is defined as

$NSPACE(s(n)) = \{ L : L$ is a language decided by an $O(s(n))$ space nondeterministic TM. $\}$

Class P

- P is the class of languages that are decidable in polynomial time on a deterministic single-tape TM.

$$P = \bigcup_{k \geq 0} TIME(n^k)$$

- P is widely recognized as the class of problems having an efficient solution.

Examples of Problems/Languages in P

1. PATH Problem: Given a graph G. Is there a path from vertex s to t?
2. RELPRIME = $\{ \langle x, y \rangle \mid x$ and y are relatively prime $\}$

   Two numbers are relatively prime if 1 is the largest integer that evenly divides both of them.
3. Every context free languages
4. Matrix Multiplication
5. Sorting an array
6. Computing Minimum Spanning Tree in a graph

Class NP (Non-deterministic Polynomial)

- NP is the class of languages that are decided by some non-deterministic polynomial time TM

$$NP = \bigcup_{k \geq 0} NTIME(n^k)$$

- NP is the class of problems whose solutions can be verified efficiently.

Alternative Def^n:

A language $L \subseteq \Sigma^*$ is said to be in NP if $\exists$ constants $c > 0$ and $k \geq 0$ and a deterministic polynomial time TM V (verifier) such that $\forall x \in \Sigma^*$, $x \in L \Leftrightarrow \exists\, y \in \Sigma^*$, $|y| \leq c \cdot x^k$, $V(x,y) = 1$

V: Verifier, a deterministic polynomial time machine that takes two strings: the input x and a string y.

y: certificate, a string whose length is polynomial in the length of x ($|x| = n$)

Examples of Problem/Languages in NP

1. HAMPATH $= \{\langle G, s, t\rangle \mid G$ is a directed graph with a hamiltonian path from s to $t\}$
2. COMPOSITES $= \{x \mid x = pq$, for integers $p, q > 1\}$
3. CLIQUE $= \{\langle G, k\rangle \mid G$ is an undirected graph with k-clique$\}$

A clique is an undirected graph is a subgraph, wherein every two nodes are connected by an edge.
A k-clique is a clique that contains k nodes.

4. SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, x_2, \cdots, x_k\}$ and for some $\{y_1, y_2 \cdots y_l\} \subseteq \{x_1, x_2, \ldots x_k\}$, we have $\Sigma y_i = t\}$

P vs NP

- P = the class of languages where membership can be decided in polynomial time
- NP = the class of languages where membership can be verified in polynomial time.
- Is P = NP? This is one of the greatest unsolved problems in theoretical Computer Science.
- If these classes were equal, any polynomially verifiable problem would be polynomially decidable.

- Researchers believe that $P \neq NP$.
- The best method known for solving languages in NP deterministically uses exponential time. In other words

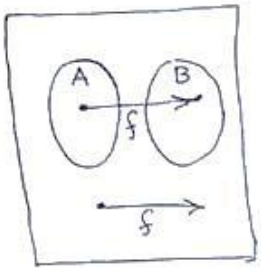
$$NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k})$$

NP-Completeness

- In 1970 Stephen Cook and Leonid Levin discovered certain problems in NP whose individual complexity is related to that of the entire class.
- If a polynomial time algorithm exists for any of these problems all problems in NP would be polynomial time solvable. These problems are called NP-Complete.
- A researcher attempting to prove that P equals to NP only needs to find a polynomial time algorithm for an NP-complete problem.
- Practically NP-Completeness may prevent wasting time in searching for a non-existent polynomial time algorithm to solve a particular problem.

Definition: Polynomial Time Computable Function

A function $f: \Sigma^* \to \Sigma^*$ is a polynomial time computable function if some polynomial time Turing Machine M exists that halts just $f(w)$ on its tape when started on any input w.



Definition: Polynomial time reducible

Language L_1 is polynomially time reducible to language L_2, written as $L_1 \leq_p L_2$, if a polynomial time computable function $f: \Sigma^* \to \Sigma^*$ exists, where for every w

$$x \in L_1 \iff f(x) \in L_2$$

The function f is called polynomial time reduction of A to B

NP-Complete

Definition: NP-Hard

A language $L \subseteq \Sigma^*$ is said to be NP-Hard if $\forall L' \in NP$, $L' \leq_p L$

Definition: NP-Complete

A language $L \subseteq \Sigma^*$ is said to be NP-complete (NPC) if $L \in NP$ and L is NP-Hard

Theorem:

If L is NP-Complete and $L \in P$ then $P = NP$

Theorem:

If $L_1 \leq_p L_2$ and L_1 is NP-Hard then L_2 is also NP-Hard.

Theorem:

If $L_1 \leq_p L_2$ and $L_1 \in P$ then $L_2 \in P$.

Examples of Problems in NP-complete

1. The satisfiability problem, SAT is the first NPC problem

$SAT = \{\langle \phi \rangle \mid \phi$ is a satisfiable boolean formula$\}$

- A boolean formula is an expression involving boolean variables and operations (AND, OR, NOT etc.)
- Example: $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
- A boolean formula is satisfiable if some assignments of 0s and 1s to the variable makes the boolean formula evaluate to 1.

2. 3SAT

$3SAT = \{\langle \phi \rangle \mid \phi$ is a satisfiable 3CNF formula$\}$

- A literal in a boolean formula is an occurrence of a variable or its negation

- A boolean formula is in conjunctive normal form (CNF) if it is expressed as AND of clauses each of which is OR of one or more literals
- A boolean formula is in 3CNF if each clause has exactly three distinct literals
- Example : $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

3. CLIQUE = $\{\langle G, k\rangle \mid G$ is an undirected graph with k-clique$\}$

4 VERTEX-COVER = $\{\langle G, k\rangle : G$ is an undirected graph that has k-node vertex cover$\}$

- A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then $u \in V'$ or $v \in V'$ or both.
- Size of a vertex cover is the number of vertices in it.

5. HAMPATH = $\{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to t$\}$

6. SUBSET-SUM = $\{\langle S, t\rangle \mid S = \{x_1, x_2, \cdots, x_k\}$ and for some $\{y_1, y_2, \cdots, y_l\} \subseteq \{x_1, x_2, \cdots, x_k\}$ we have $\sum y_i = t\}$