AUTOMATA LANGUAGES AND COMPUTATION

SYLLABUS

UNIT – I

Finite automata and Regular expression: Deterministic and Non-Deterministic Finite Automata, Finite automata with ε-moves, Regular expression – Equivalence of NFA and DFA ,Two- way finite automata, Moore machine and Mealy machines , Applications of finite automata.

INDEX

UNIT –I

1. FINITE AUTOMATA AND REGULAR EXPRESSION

1.1 FINITE AUTOMATA

Finite automata is a mathematical model of a system with discrete inputs and outputs .The system can be in any one of finite number of states and the state summarizes the history of inputs and determines the behaviour of the system for subsequent input.

1.1.1 COMPONENTS OF FINITE AUTOMATA

The block diagram of the finite automata contains three components namely,

1 .Input tape

2. Reading Head Pointer

3. Finite Control

INPUT TAPE

The input tape is divided into number of squares or cells. Each square contains single symbol or alphabet from the input alphabet Σ. The left end square of the tape contains ¢ and the right end square contains $ symbol. The absence of the end markers indicate that the tape is of infinite length. The left to right sequence of symbols between the end markers is the string to be processed.

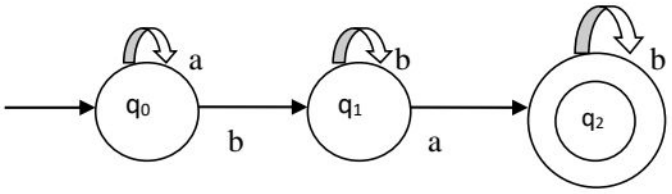
READING HEAD POINTER

The head examines only one square at a time and can move either one square to the left or to the right. We restrict the movement of the reading head pointer only to the right side.

FINITE CONTROL

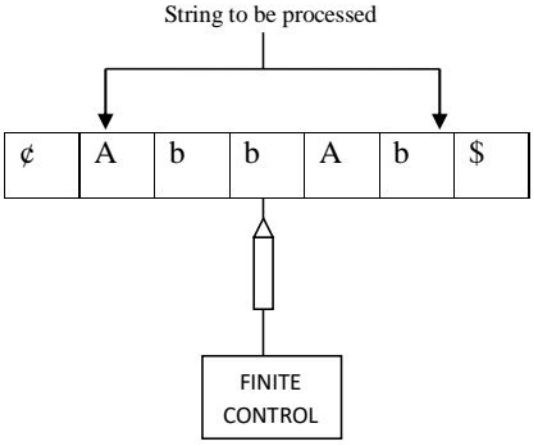
The finite control contains the routines, which instruct the reading head pointer to move from one state to the next state by recognising each and every symbol or alphabet. The transition of reading head pointer from one state to another state by recognising an alphabet is indicated by $\delta(q ,a)$.

For E.g.: Consider a Finite Automata with the transition diagram

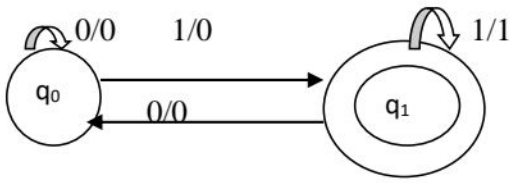


The transition graph or transition diagram is a finite labelled graph in which each vertex or node represent a state and the directed arcs indicates the transition of a state and the edges are labelled with input/output .The initial state is indicated with circle and arrow pointing towards it, and the final state is indicated by concentric circles.

Block diagram of Finite Automata



A transition graph or a transition system is a finite directed graph in which each vertex or node represents a state and the directed edge are labelled with output/input.



Automaton in which the output depends only on the input is called as an automaton without memory. An automaton in which the output depends only on the states of the machine is called as Moore machine. An automaton in which output depends only on the states as well as on the input at any instant of time is called a Mealy machine. Both the Moore and Mealy machine are Finite automaton with output.

2. TYPES OF FINITE AUTOMATA

The finite automata can be divided into

a) Deterministic Finite Automata (DFA).
b) Non-deterministic Finite Automata (NFA or NDFA)

2.1 DETERMINISTIC FINITE AUTOMATA

The deterministic finite automata can represented by 5-tuples.If M is the deterministic finite automata then,

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

Where,

M is the NFA with ε moves.

Q is the finite set of non-empty states.

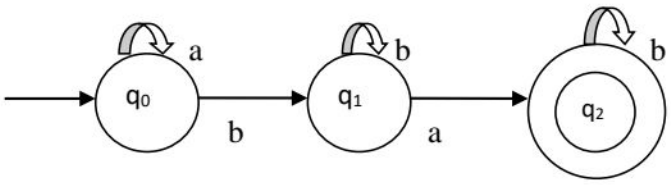
$\sum$ is the finite set of non-empty symbols.

δ is the finite set of transitions.

q_0 is the initial state.

F is the final state.

Consider an example,



Here

Q= {q_0, q_1, q_2}

∑= {a, b}

q_0= {q_0}

F= {q_2}

Transition Table

δ	a	b
{q_0}	{q_0}	{q_1}
{q_1 }	{q_2}	{q_1}
{q_2}	ϕ	{q_2}

The transition mapping is

δ: Q X∑ →Q

Where q_0 Є A and a Є ∑ and it is mapped on to q_0 Є Q and q_0 Є Q'.

2.2 NON DETERMINISTIC FINITE AUTOMATA

A non-deterministic finite automaton (NDFA) can be represented by 5 tuples namely

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

Where,

M is the NFA with ε moves.

Q is the finite set of non-empty states.

$\sum$ is the finite set of non-empty symbols.

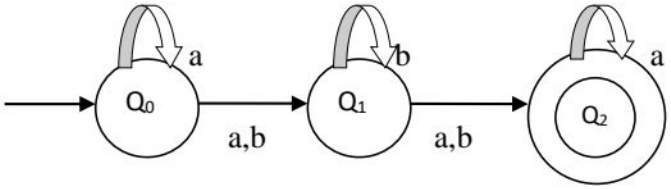
δ is the finite set of transitions.

q_0 is the initial state.

F is the final state.

i.e. "δ" is the transition mapping function from $Q \times \sum \rightarrow 2^Q$, where are mapped to subsets of the input individual state.

e.g.:



Where,

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

$\sum = \{a, b\}$

$q_0 = \{q_0\}$

$F = \{q_2\}$

Transition Table

δ	a	b
q_0	(q_0, q_1)	(q_1)
sq_1	(q_2)	(q_1, q_2)
q_2	(q_2)	ϕ

The transition mapping is

$Q \times \sum \rightarrow 2^Q$ or Q'

$\delta(q_0, a) = (q_0, q_1)$

where $q_0 \in Q$, $a \in \sum$ and $(q_0, q_1) \in Q'$ rather than $(q_0, q_1) \in Q$ since,

$Q' = 2^Q$ (subsets of individual states)

$= \{(q_0), (q_1), (q_2), (q_0, q_1), (q_1, q_2), (q_0, q_2), (q_0, q_1, q_2), \phi\}$

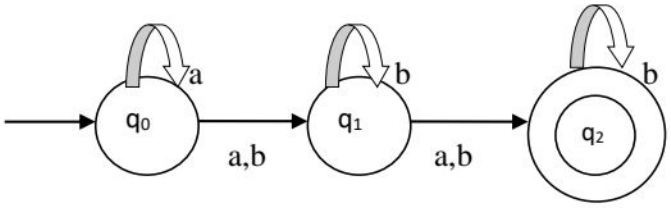
DIFFERENCE BETWEEN NFA AND DFA

The main difference between NFA and DFA is,

a). The deterministic finite automata contains only at most one transition from each and every state for each input symbol but the non-deterministic finite automata contains more than one transition for each and every symbol from each and every state.
b). The second difference is that DFA has the transition mapping $Q \times \sum \rightarrow Q$ where as the NFA contains the transition mapping $Q \times \sum \rightarrow 2^Q$ or Q'.

2.3. CONVERSION OF NFA TO DFA

Consider a NFA with the transition diagram,



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

$\sum=\{a,b\}$

$q_0=\{q_0\}$

$F=\{q_2\}$

Transition Table

δ	a	b
q_0	(q_0, q_1)	(q_1)
q_1	(q_2)	(q_1, q_2)
q_2	ϕ	(q_2)

Converting to DFA

The DFA can be expressed during conversion

$M' = (Q,\sum ,\delta ',q_0,F')$

Where $Q'=2^Q$.

$\sum= \{a, b\}$

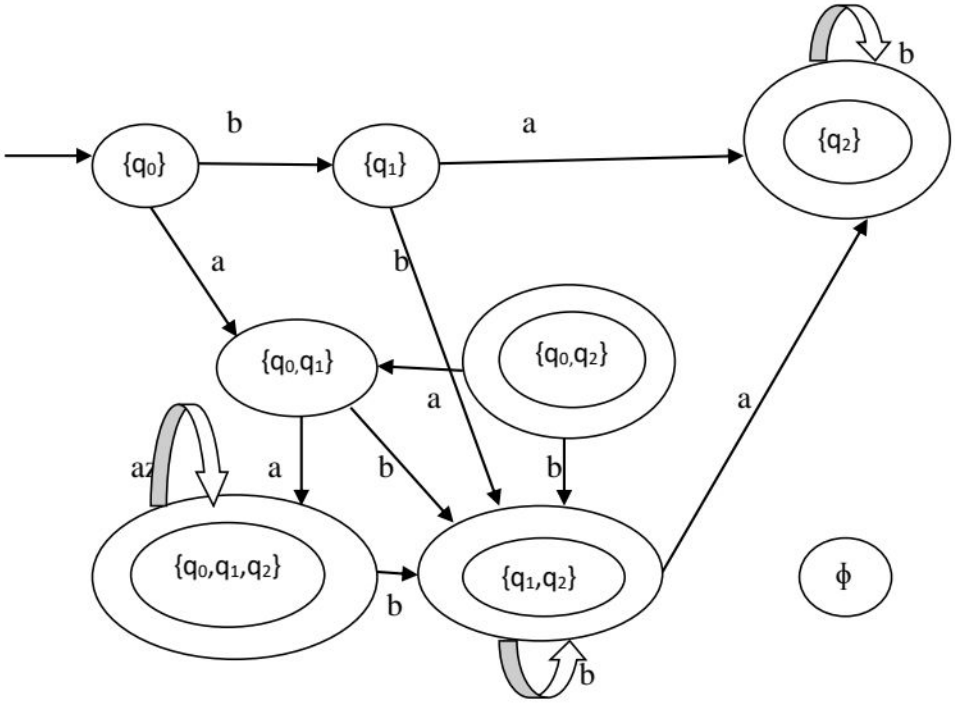
$q_0=q_0$

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

Transition Table

δ´	a	b
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_1\}$
$\{q_1\}$	$\{q_2\}$	$\{q_1, q_2\}$
$\{q_2\}$	ϕ	$\{q_2\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	$\{q_2\}$	$\{q_1, q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
ϕ	ϕ	ϕ

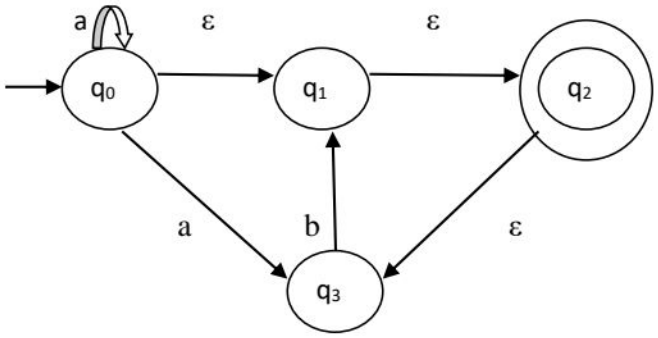
TRANSITION DIAGRAM



3. FINITE AUTOAMTA WITH ε- MOVES

The 'ε' is a character used to indicate the null string i.e. the string which is used simply for transition from one state to the other without any input . The NFA with ε-moves can be shown below:

e.g.



Let us define NFA with ε-transition as

M= (Q, ∑, δ ,q0,F)

Where,

M is the NFA with ε moves.

Q is the finite set of non-empty states.

∑ is the finite set of non-empty symbols.

δ is the finite set of transitions.

q_0 is the initial state of NFA.

F is the final state of NFA with ε −moves.

The transition mapping is given by

$$Q \times (\sum \cup \{\varepsilon\}) \rightarrow 2^Q \text{ or } Q'.$$

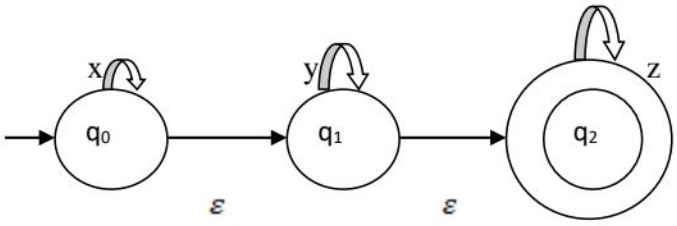
ε- closure(q) denotes the set of all states P such that there is a path from q to p labelled .The

$\hat{\delta}$ is interpreted as follows ,

$\hat{\delta}(q, \varepsilon) = \varepsilon$ −closure(q).

e.g.

Consider finite automata with ε-moves



The ε −closure for each and every state as follows:

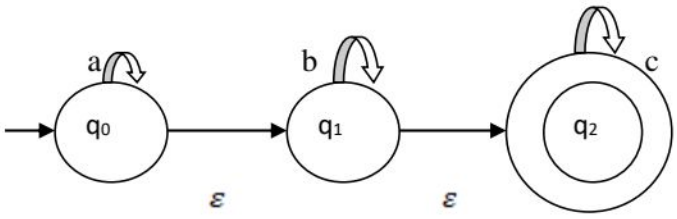
$\delta'(q_0, \varepsilon) = \varepsilon$ -closure$(q_0) = \{q_0, q_1, q_2\}$

$\delta'(q_1, \varepsilon) = \varepsilon$ -closure$(q_1) = \{q_1, q_2\}$

$\delta'(q_2, \varepsilon) = \varepsilon$ -closure$(q_2) = \{q_2\}$

4. CONVERSION OF NFA WITH ε −TRANSITION TO NFA WITHOUT ε-TRANSITION

Consider a NFA with ε-transition has 5 tuples



M= (Q,∑, δ ,q0,F)

Where,

Q={q_0,q_1,q_2}

∑={a,b,c, ε }

δ is the transition function that maps,

$$Q \times (\sum \cup \{\varepsilon\}) \rightarrow 2^Q \text{ or } Q'.$$

Transition table

Δ	a	b	c	ε
$\{q_0\}$	$\{q_0\}$	ϕ	ϕ	$\{q_1\}$
$\{q_1\}$	ϕ	$\{q_1\}$	ϕ	$\{q_2\}$
$\{q_2\}$	ϕ	ϕ	$\{q_2\}$	ϕ

$q_0 = \{q_0\}$

$F = \{q_2\}$

When converting NFA with ε-transition to NFA without ε-transition , one automaton contains

$M' = (Q, \sum, \delta', q0, F')$

Where,

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

$\sum = \{a, b, c\}$

δ′ is the transition function for NFA without transition , then we first compute the ε-closure of each and every state.

ε -closure$(q_0) = \{q_0, q_1, q_2\}$

ε -closure$(q_1) = \{q_1, q_2\}$

ε -closure$(q_2) = \{q_2\}$

The transition function for each and every input symbol can be summarised as:

$\delta'(q_0,a) = \varepsilon$ -closure($\delta(\delta'(q_0, \varepsilon),a)$)

$= \varepsilon$ -closure($\delta(\{q_0,q_1,q_2\},a)$)

$= \varepsilon$ -closure($\delta(q_0,a) \cup \delta(q_1,a) \cup \delta(q_2,a)$)

$= \varepsilon$ -closure($q_0 \cup \phi \cup \phi$)

$= \varepsilon$ -closure(q_0)

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

$\delta'(q_0,b) = \varepsilon$ -closure($\delta(\delta'(q_0, \varepsilon),b)$)

$= \varepsilon$ -closure($\delta(\{q_0,q_1,q_2\},b)$)

$= \varepsilon$ -closure($\delta(q_0,b) \cup \delta(q_1,b) \cup \delta(q_2,b)$)

$= \varepsilon$ -closure($\phi \cup q_1 \cup \phi$)

$= \varepsilon$ -closure(q_1)

$=\{q_1,q_2\}$

$\delta'(q_0,c) = \varepsilon$ -closure($\delta(\delta'(q_0, \varepsilon),c)$)

$= \varepsilon$ -closure($\delta(\{q_0,q_1,q_2\},c)$)

$= \varepsilon$ -closure($\delta(q_0,c) \cup \delta(q_1,c) \cup \delta(q_2,c)$)

$= \varepsilon$ -closure($\phi \cup \phi \cup q_2$)

$= \varepsilon$ -closure(q_2)

$=\{q_2\}$

$\delta'(q_1,a) = \varepsilon$ -closure($\delta(\delta'(q_1, \varepsilon),a)$)

$= \varepsilon$ -closure($\delta(\{q_1,q_2\},a)$)

$= \varepsilon$ -closure($\delta(q_1,a) \cup \delta(q_2,a)$)

$= \varepsilon$ -closure($\phi \cup \phi$)

$= \varepsilon$ -closure(ϕ)

$= \phi$

$\delta'(q_1,b) = \varepsilon$ -closure($\delta(\delta'(q_1, \varepsilon),b)$)

$= \varepsilon$ -closure($\delta(\{q_1,q_2\},b)$)

$= \varepsilon$ -closure($\delta(q_1,b) \cup \delta(q_2,b)$)

$= \varepsilon$ -closure($q_1 \cup \phi$)

$= \varepsilon$ -closure(q_1)

$= \{q_1,q_2\}$

$\delta'(q_1,c) = \varepsilon$ -closure($\delta(\delta'(q_1, \varepsilon),c)$)

$= \varepsilon$ -closure($\delta(\{q_1,q_2\},c)$)

$= \varepsilon$ -closure($\delta(q_1,c) \cup \delta(q_2,c)$)

$= \varepsilon$ -closure($\phi \cup q_2$)

$= \varepsilon$ -closure(q_2)

$= \{q_2\}$

$\delta'(q_2,a) = \varepsilon$ -closure($\delta(\delta'(q_2, \varepsilon),a)$)

$= \varepsilon$ -closure($\delta(\{q_2\},a)$)

$= \varepsilon$ -closure($\delta(q_2,a)$)

$= \varepsilon$ -closure(ϕ)

$= \phi$

$\delta'(q_2,b) = \varepsilon$ -closure($\delta(\delta'(q_2, \varepsilon),b)$)

$= \varepsilon$ -closure($\delta(\{q_2\},b)$)

$= \varepsilon$ -closure($\delta(q_2,b)$)

$= \varepsilon$ -closure(ϕ)

$= \phi$

$\delta'(q_2,c) = \varepsilon$ -closure($\delta(\delta'(q_2, \varepsilon),c)$)

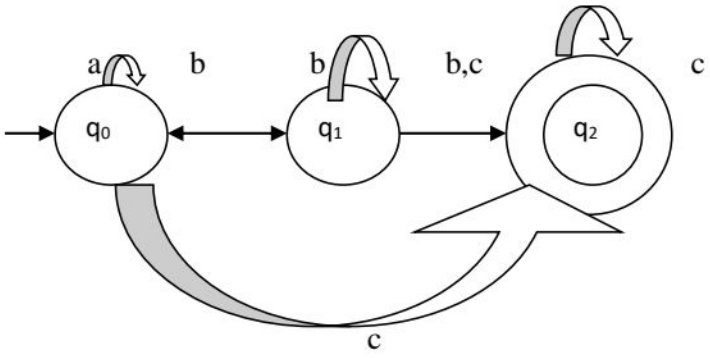
$= \varepsilon$ -closure($\delta(\{q_2\},c)$)

$= \varepsilon$ -closure($\delta(q_2,c)$)

$= \varepsilon$ -closure(q_2)

$= \{q_2\}$

The transition for NFA without ε-moves:



5. EQUIVALENCE OF NFA AND DFA

Statement:

Let 'L' be a set accepted by a non-deterministic finite automata then there exists a deterministic finite automaton that accepts "L".

To Prove:

The Language accepted by NFA is equal to the language set accepted by DFA.

i.e. L(M)=L(M´)

Where M is the NFA and M´ is the DFA.

Proof:

Let M= $(Q,\sum, \delta, q_0, F)$ be an NFA ,accepting L and we can define a DFA

M´ = $(Q,\sum, \delta', q_0, F')$ as follows .

The states of M´ are all the subsets of the set of states of M.

That is, $Q'=2^Q$

M´ will keep track in its states of all the states M could be in at any given time. F´ is the set all the states in Q´ containing the final state of M. An element of Q´ will be denoted by

$[q_1,q_2,q_3....q_i]$ are in Q .

Observe that, $[q_1,q_2,q_3....q_i]$ is a single state of DFA corresponding to the states of the NFA. Note $q_0'=[q_0]$

We define

$\delta'([q_1,q_2,q_3....q_i],a)= [P_1,P_2,P_3....P_j]$

if and only if

$\delta([q_1,q_2,q_3....q_i],a)= [P_1,P_2,P_3....P_j]$

that is

δ' applied to an element $[q_1,q_2,q_3....q_i]$ of Q´ is computed by applying δ to each state of Q represented by $[q_1,q_2,q_3....q_i]$ in applying δ to each of $[q_1,q_2,q_3....q_i]$ and taking "union" we get some new set of states $P_1,P_2,P_3....P_j$. This new set of states has a representative $P_1,P_2,P_3....P_j$ in Q´ and that element is the value of $\delta'([q_1,q_2,q_3....q_i],a)$.

It is easy to show by induction for the length of the input string x, that

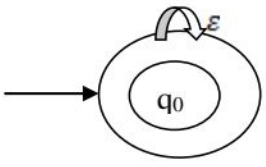
$\delta'(q_0',x) =[q_1,q_2,q_3....q_i]$

if and only if,

$\delta(q_0,x) =\{q_1,q_2,q_3....q_i\}$

Basis

Consider $|x|=0$ then x must be ε since $q_0'=[q_0]$



Then

$\delta'(q_0', \varepsilon) = (q_0)$

if and only if

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

Hence the result is trivial.

Induction

Let us assume that the hypothesis is true for inputs of length m, Let xa be a string of length m+1 with a in $\sum$, then

$\delta'(q_0', xa) = \delta'(\delta'(q_0', x), a)$

By the inductive hypothesis ,

$\delta'(q_0', x) = [P_1, P_2, P_3 \ldots P_j]$

If and only if ,

$\delta(q_0, x) = [P_1, P_2, P_3 \ldots P_j]$

But by the definition of δ',

$\delta'[(P_1, P_2, P_3 \ldots P_i), a] = [r_1, r_2, r_3 \ldots rk]$

If and only if,

$\delta[(P_1, P_2, P_3 \ldots P_i), a] = [r_1, r_2, r_3 \ldots r_k]$

Thus,

$\delta'(q_0', xa) = [r_1, r_2, r_3 \ldots r_k]$

If and only if,

$\delta(q_0, xa) = [r_1, r_2, r_3 \ldots r_k]$

Which establishes the inductive hypothesis,

To complete the proof , we have only to add that $\delta'(q_0', x)$ is in F′ exactly when $\delta(q_0, x)$ contains a state of Q that is F.

Thus,

L(M)=L(M′)

6. REGULAR EXPRESSION

The languages accepted by the finite automata are easily described as simple expression or regular expression.

Let '∑' be the alphabet. The regular expression over ∑ and the set they denote are defined recursively as follows,

a) ϕ is the regular expression and denotes the empty set.
b) ε is the regular expression and denotes the set{ ε}.
c) For each 'a' in ∑, a is a regular expression and denotes the set {a}.
d) If r and s are regular expression denoting languages R and S respectively hen (r+s) ,(rs) and (r*) are regular expression that denote the set RUS,RS,R* respectively.

7. REGULAR EXPRESSION THEOREM

Statement

Let 'r' be the regular expression then there exists an NFA with ε-transition that accepts L(r).

To Prove

L(M)=L(r).

That is the regular expression to a language 'L' can be accepted by automata with ε-moves.

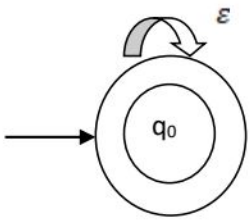
Proof

We can show by induction on the number of operators in the regular expression 'r' that is a NFA 'M' with ε-transitions, having one final state and no transition out of this final state.

Basis: (Zero Operators)

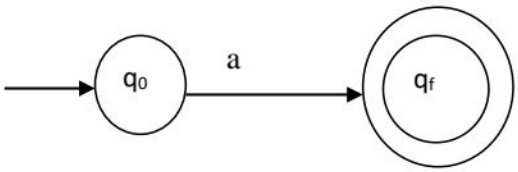
The expression 'r' must be ε, ϕ or a for some a in $\sum$. The NFA for each of the expressions are,

a) r= ε



b) r= ϕ

c) r=a



Hence if 'r' is the regular expression then there exists NFA accepting 'r'.

Induction

Assume that the theorem is true for regular expression with fewer than 'i' operators, that is i>=1. Let r have 'i' operators. There are three cases depending on the form of 'r'.

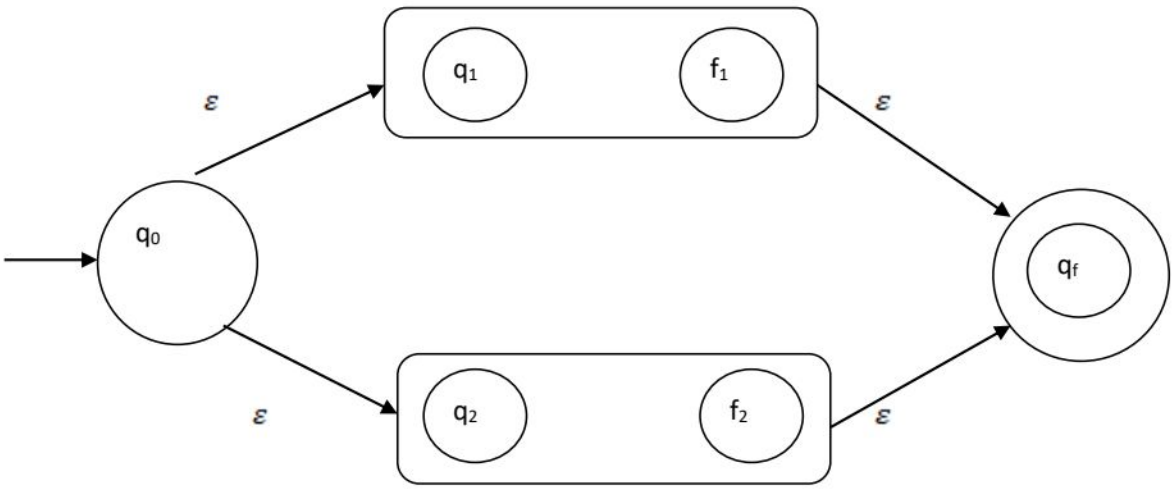
Case 1:

Union($r=r_1+r_2$)

Both r_1 and r_2 must have fewer than i operators. Thus there are NFA's $M_1=(Q_1,\Sigma_1,\delta_1,q_1,\{f_1\})$ and $M_2=(Q_2,\Sigma_2,\delta_2,q_2,\{f_2\})$ with $L(M_1)=L(r_1)$ and $L(M_2)=L(r_2)$. Since we may rename states of a NFA, we may assume Q_1 and Q_2 are disjoint. Let 'q_0' be a new initial state and f_0 a new final state.

Construct,

$M=(Q_1 U Q_2 U\{q_0,f_0\}, \Sigma_1 U \Sigma_2, \delta, q_0,\{f_0\})$



1. $\delta(q, \varepsilon) =\{ q_1,q_2\}$

2. $\delta(q,a) = \delta_1(q,a)$ For q in $Q_1-\{f_1\}$ and a in $\Sigma_1 U\{ \varepsilon \}$

3. $\delta(q,a) = \delta_2(q,a)$ For q in $Q_2-\{f_2\}$ and a in $\Sigma_2 U\{ \varepsilon \}$

4. $\delta(f_1, \varepsilon) = \delta(f_2, \varepsilon)=\{ f_0\}$

We can recall the inductive hypothesis that there are no transitions out of f_1 or f_2 in M_1 or M_2, thus the moves of M_1 and M_2 are present in M. The construction of M is depicted in the figure. Any path in the transition diagram of M from q_0 to f_0 must begin by going to either q_1 or q_2 on reading ε.If the path goes to q_1, it may follow any path $M_1$to f_1 and then goes to f_0 on

ε.Similarly, the path that begin by going to q_2 may follow any path in M_2 to f_2 and then go to f_0 on reading ε.It follows immediately that there is a path labelled x in M from q_0 to f_0,if and only if there is a path labelled x in M_1 from q_1 to f_1 or path in M_2 from q_2 to f_2. Hence $L(M)=L(M_1)UL(M_2)$ is derived.

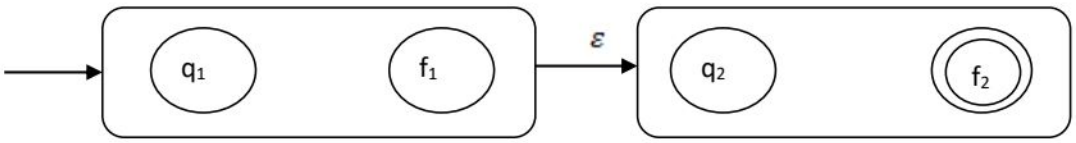
Case2:

Concatenation($r=r_1.r_2$)

Consider two NFA's $M_1=(Q_1,\Sigma_1,\delta_1,q_1,\{f_1\})$ and $M_2=(Q_2,\Sigma_2,\delta_2,q_2,\{f_2\})$ with $L(M_1)=L(r_1)$ and $L(M_2)=L(r_2)$.

Then,

$M=(Q_1UQ_2, \Sigma_1U\,\Sigma_2, \delta, q_1,\{f_2\})$



1. $\delta(q,a) = \delta_1(q,a)$ For q in $Q_1-\{f_1\}$ and a in $\Sigma_1U\{\ \varepsilon\ \}$

2. $\delta(q,a) = \delta_2(q,a)$ For q in $Q_2-\{f_2\}$ and a in $\Sigma_2U\{\ \varepsilon\ \}$

3. $\delta(f_1, \varepsilon) = \{q_2\}$

Every path in M from q_1 to f_2 is a path labelled by string x from q_1 to f_1 followed by the edge from f_1 to q_2 labelled ε, followed by a path labelled by some string y from q_2 to f_2.

Thus ,

$L(M)=\{xy/x \text{ in } L(M_1) \text{ and } y \text{ in } L(M_2) \text{ and } L(M)=L(M_1).L(M_2)\}$ is derived.

Case3:

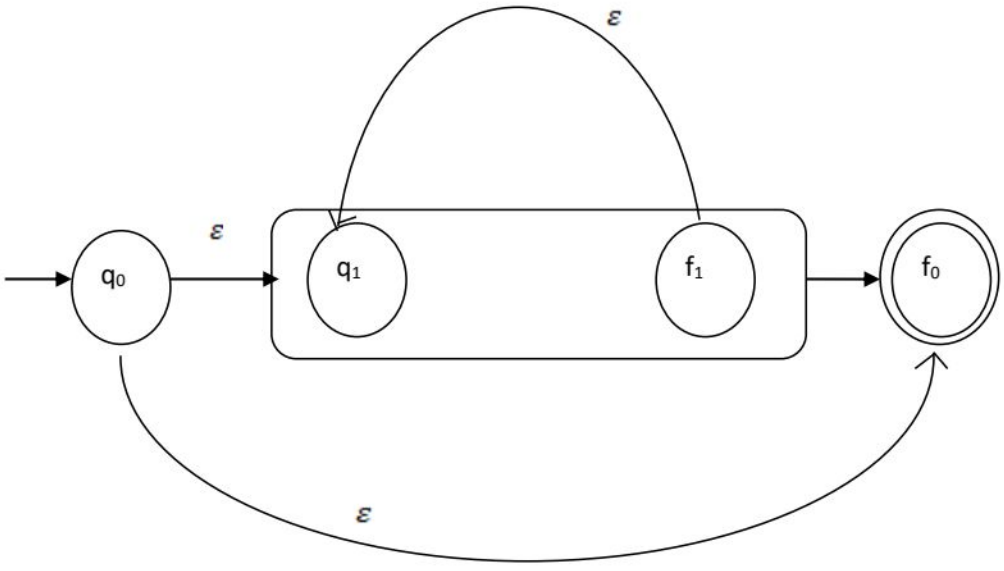
Kleene closure ($r=r_1^*$)

Let $M_1=(Q_1,\Sigma_1,\delta_1,q_1,\{f_1\})$ and $L(M_1)=L(r_1)$,Construct

$M=(Q_1 U\{q_0,f_0\}, \Sigma_1, \delta, q_0,\{f_0\})$

Where δ is given by,

$\delta(q_0, \varepsilon)= \delta(f_1, \varepsilon)=\{q,f_0\}$



$\delta(q,a) = \delta_1(q,a)$ For q in $Q_1-\{f_1\}$ and a in $\Sigma_1 U\{ \varepsilon \}$

Any path from q_0 to f_0 consists of either of a path from q_0 to f_0 on ε, followed by some number of paths from q_1 to f_1 then back to q_1 on ε each labelled by a string in $L(M_1)$ followed by a string in $L(M_1)$ followed by a path from $q_1$to f_1 on the string in $L(M_1)$ from f_0 on ε.Thus there is a path in M from q_0 to f_0 on ε.Thus there is a path in M from q_0 tof_0 labelled x if and only if we can write $x=x_1x_2x_3.....x_{j.,}$ Such that x_j is in $L(M_1)$ then $L(M)=L(M^*)$.

5. TWO-WAY FINITE AUTOMATA

We have viewed the deterministic finite automata as a control unit that reads a tape moving one square right at each move. We needed non-determinism to the model which allowed many copies of control unit to exit and scan the tape simultaneously. Next we added ε-transitions which allowed change of state without reading the input symbol or moving the tape head. Next interesting extension is to allow the tape head with the ability to move left as well as right, such a finite automaton is called as two-way finite automaton.

The two-way finite automata accepts, the input symbol where tape head must more left or right at each move. A two finite automaton (2DFA) is a quintuple.

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

Where,

M is the NFA with ε moves.

Q is the finite set of non-empty states.

$\sum$ is the finite set of non-empty symbols.

∂ is the finite set of transitions (represented by transition table).

q_0 is the initial state.

F is the final state.

The transition mapping is given by

$$Q \times \sum \rightarrow Q \times (L, R) .$$

a) If $\delta(q,a)=(P,L)$ then in state q, scanning input symbol a, the 2DFA enters state 'p' and moves the head left one square.
b) If $\delta(q,a)=(P,R)$,the 2DFA enters state 'p' and moves its head right one square .We introduce notation called "Instantaneous Description (ID)" of a 2DFA which the

input string, current state and position of the input head. Then we introduce the relation Γ_n on ID's such that $I_1 \Gamma_m I_2$ if and only if M can go from the instantaneous description I_1 to I_2 in one move.

An ID of M is a string in $\sum^* Q \sum^*$.The ID "wqx" where w and x are in $\sum^*$ and q is in Q where ,

1) Wx is the input string.
2) Q is the current state,
3) The input head is scanning the first symbol of x

we define,

$L(n)=\{w/q_0 w \Gamma^* wp$ for some path p in f$\}$

That is, w is accepted by M, if starting in state q_0 with w on the input tape and the head at the left end of w. M eventually enters a final state at the same time it falls of the right end of the input tape.

For example

Consider a 2DFA that has the transition table of

Transition Table

δ	a	b
q_0	(q_0,R)	(q_1,R)
q_1	(q_1,R)	(q_2,L)
q_2	(q_0,R)	(q_2,L)

Consider the input 101001 .Since q_0 is the initial state the first ID is q_0 101001. So,

q_0 101001 $\vdash$ 1 $q_1$01001 (Since $\delta(q_0,1) = (q_1,R)$)

$\vdash$ 10 $q_1$1001 (Since $\delta(q_1,0) = (q_1,R)$)

$\vdash$ 1 $q_2$01001 (Since $\delta(q_1,1) = (q_2,L)$)

$\vdash$ 10 $q_0$1001 (Since $\delta(q_2,0) = (q_0,R)$)

$\vdash$ 101 $q_1$001 (Since $\delta(q_2,0) = (q_0,R)$)

$\vdash$ 1010 $q_1$01 (Since $\delta(q_1,0) = (q_1,R)$)

$\vdash$ 10100 $q_1$1 (Since $\delta(q_1,0) = (q_1,R)$)

$\vdash$ 1010 $q_2$01 (Since $\delta(q_1,1) = (q_2,L)$)

$\vdash$ 10100 $q_0$1 (Since $\delta(q_2,0) = (q_0,R)$)

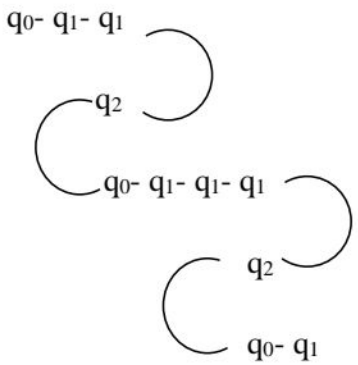
$\vdash$ 101001 q_1 (Since $\delta(q_0,1) = (q_1,R)$)

Thus the string is accepted by two-way finite automata.

CROSSING OVER SEQUENCE

A useful picture of the behaviour of 2DFA consists of the input, the path followed by the head, and the state each time the boundary between two tape squares is crossed , with the assumption that the control enters its new state prior to moving head .For example ,the behaviour of the 2DFA M of 101001 is shown as

1	0	1	0	0	1



The list of states below each boundary between squares is termed as crossing sequence. Note that if 2DFA accepts its input, no crossing sequence may have represented state with the head moving in the same direction, otherwise the 2DFA being deterministic would be in loop thus never fall off the right end.

Another important observation about crossing sequences is that the first time boundary is crossed, the head must be moving right. Subsequent crossings must be in alternate directions. Thus odd-numbered elements of a crossing sequence represent right moves and even –

numbered elements represent left moves. If the input is accepted ,it follows that crossing sequence are of odd length.

A crossing sequence q_1, , q_2.... q_k is said to be valid if it is of odd length ,and no two odd –and no two even numbered elements are identical. A 2DFA with 's' states can have valid crossing sequences of length at most 2s ,so the number of valid crossing sequence is finite. There are two types of crossing sequence

a) Right –matching sequence.
b) Left –matching sequence.

6. FINITE AUTOMATA WITH OUTPUT

The finite automata can have binary output. There are two distinct approaches,

a) Moore machine
b) Mealy machine

MOORE MACHINE

If the output is associated with each and every state, then it is called Moore machine

MEALY MACHINE

If the output is associated to each and every transition then it is called Mealy machine.

Both the machine takes the same input and returns the similar output

MOORE MACHINE

The Moore machine consists of six tuples,

$M= (Q,\sum, \Delta, \delta , \lambda ,q_0,)$

Q is the finite set of non-empty states.

$\sum$ is the finite set of non-empty input symbols.

Δ is the finite set of non-empty output symbols.

δ is the finite set of input transitions.

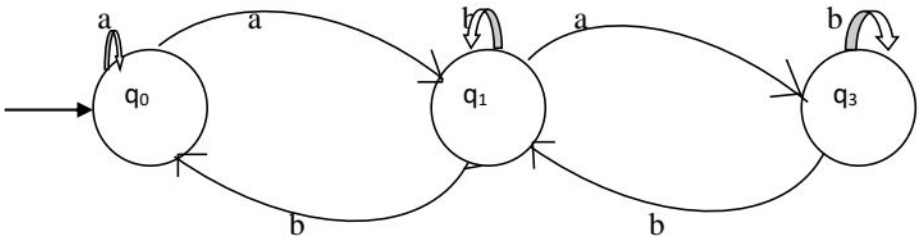
λ is the output function mapping.

q_0 is the initial state.

δ is the transition function from $Q X \sum$ into Q is the function mapping Q into Δ and 'q_0' is the initial state.

e.g.

Consider a Moore machine,



1 2 3

Here the output function $\lambda(t)$ depends on ,

$\lambda(t) = \lambda(q(t))$

This is Moore machine because the output function depends on the present state and is independent of current output of the automaton.

$\lambda: Q \rightarrow \Delta$

MEALY MACHINE

The value of the output function $\lambda(t)$ is the more general case is a function of the present state q(t) and present input x(t).

That is,

$\lambda(t)= \lambda(q(t),x(t))$.

$M= (Q,\sum, \Delta, \delta , \lambda ,q_0,)$

Q is the finite set of non-empty states.

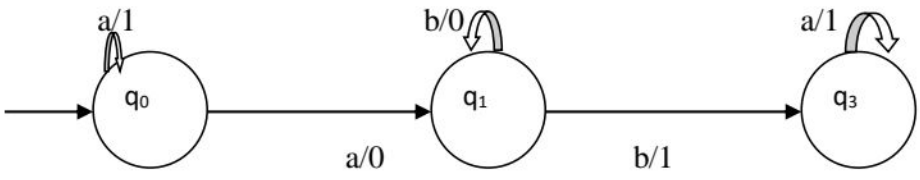
$\sum$ is the finite set of non-empty input symbols.

Δ is the finite set of non-empty output symbols.

δ is the finite set of input transitions.

λ is the output function mapping.

q_0 is the initial state.



Here,

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

$\sum= \{a,b\}$

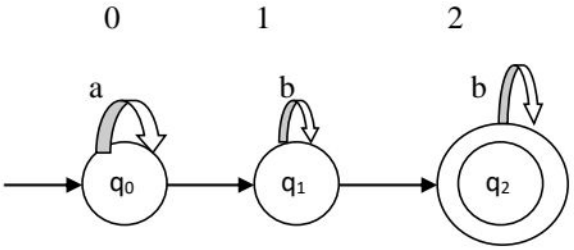
q0= { q0}

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

Since each and every transition contains an output alphabet. This automata with output is called Mealy machine.

7. CONVERSION OF MOORE TO MEALY MACHINE

Consider a Moore machine



here,

M= (Q,∑, δ ,Δ,q0,λ)

Then

Q= {q_0,q_1,q_2}

∑= {a,b}

q0= { q0}

Δ= {0,1,2}

λ (q_0)=0

λ(q_1)=1

λ(q_2)=2

Transition Table

Δ	a	b
q_0	{ q_0}	{ q_1}
q_1	{ q_2}	{ q_1}

q_2	ϕ	$\{ q_2 \}$

Converting to Mealy:

$\lambda(\delta(q_0,a)) = \lambda(q_0) = 0$

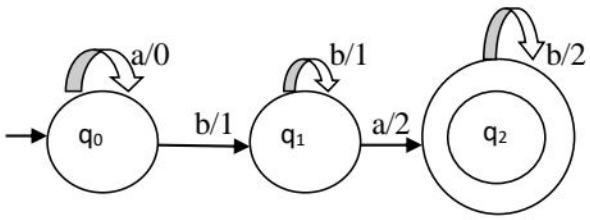
$\lambda(\delta(q_0,b)) = \lambda(q_1) = 1$

$\lambda(\delta(q_1,a)) = \lambda(q_2) = 2$

$\lambda(\delta(q_1,b)) = \lambda(q_1) = 1$

$\lambda(\delta(q_2,a)) = \lambda(q_2) = 2$

The final Mealy Transition diagram:



8.APPLICATIONS OF FINITE AUTOMATA

There are varieties of software design related problems that are simplified by automatic type conversion of regular expression notation to an efficient computer implementation to the corresponding finite automaton.

Two such applications are,

a) Lexical Analyser.
b) Text Editor.

LEXICAL ANALYSER

The tokens of the programming languages are almost with expression expressible as regular sets.

E.g. ALGOL identifiers, which are upper or lower case followed by any sequence of letter and digits with no limit on length, may be expressed as

(Letter) (letter+ digit)*

Where letter stands for A+B+.....+Z+ a + b+.....+z and digits stand for 0+1+....+9.

FORTRAN identifiers length limit six and letters restricted to uppercase and symbol $ may be expressed as

$$(\text{letter})\ (\varepsilon + \text{letter} + \text{digit})^5$$

Where "letter" now stands for ($+A+B+.....Z)

A number of lexical –analyzer generators take as input a sequence of regular expression describing the tokens and produce a finite automaton recognizing any token. Usually they convert the regular expression to an NFA with ε −transitions. Each final state indicates the particular token is found, so the automaton is really a Moore machine. The lexical analyser produced by the generator is a fixed program that interprets coded tables together with the particular table that represents the finite automata recognizing the token. The Lexical Analysis is the first phase of a compiler.

TEXT EDITOR

Certain text editors and similar programs permit the substitution of a string of any string matching a given regular expression.

E.g. The UNIX text editor allows a command such as

S/bbb*/b/

This substitute a single blank for the first string of two or more blanks found in a given line. Let "any" denote the expression $a_0 + a_1 + ...a_m$, where a_i 's are all of a computer's character except the "newline" character. We could convert a regular expression r to a DFA that accepts any *r. Note the presence of any *allows us to recognise a member of L(r) beginning anywhere in the line.

Actually what happens in the UNIX text editor is that the regular expression *r is converted to an NFA with ε –transition and NFA is stimulated directly.

AUTOMATA LANGUAGE AND COMPUTATION

UNIT-II

Regular Sets and Context Free Grammars: Properties of regular sets, Context-Free Grammars-derivation trees, Chomsky Normal Forms and Greibach Normal Forms, ambiguous and unambiguous grammars; minimization of finite automata.

1. REGULAR SETS

Regular languages can also be defined, from the empty set and from some finite number of singleton sets, by the operations of union, composition, and Kleene closure. Specifically, consider any alphabet. Then a regular set over is defined in the following way.

- The empty set Ø, the set {} containing only the empty string, and the set {a} for each symbol a in Σ, are regular sets.
- If L1 and L2 are regular sets, then so are the union L1 L2, the composition L1L2, and the Kleene closure L1*.
- No other set is regular.

1.1 The Pumping Lemma for Regular Sets

Applications:

- The pumping lemma is a powerful tool for providing and proving certain language is regular or not.
- It is also useful in the development of algorithms to answer certain questions concerning finite automata, such as whether the language accepted by a given FA is finite or infinite.

Statement:

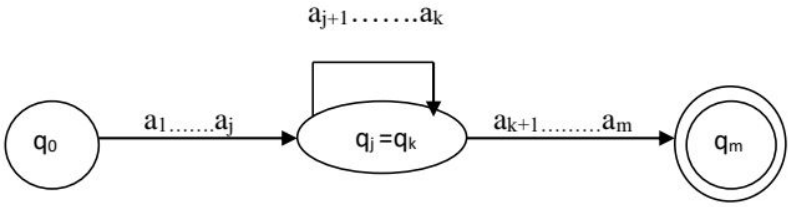
Let "L" be a regular set. Then there is a constant n such that if z is any word in L, and $|z| \geq n$. We may write z=uvw in such a way that $|uv| \leq n$, $|v| \geq 1$ and for all $i \geq 0$, then uv^iw is in L.

Proof:

If a language is accepted by a DFA M= (Q, Σ, δ, q_0, F) then it is regular with some particular number of states, say n. Consider a input of n or more symbols $a_1a_2\ldots..a_m$, $m \geq n$ and for i=1,2,…..,m.

Let $\delta (q_0, a_1a_2\ldots..a_i) = q_i$

It is not possible for each of the n+1 states $q_0, q_1, \ldots\ldots q_n$ to be distinct, since there are only different states. This there are two integers j and k, $0 \leq j < k \leq n$, such that $q_j = q_k$. The path labeled $a_1 a_2 \ldots a_m$ in the transition diagram of M. Since j<k, the string $a_{j+1} \ldots . a_k$ is of length 1.



If q_m is in F, that is $a_1 a_2 \ldots\ldots\ldots a_m$ is in L(n), then $a_1 a_2 \ldots\ldots\ldots a_j a_{k+1} a_{k+2} \ldots\ldots a_m$ is also in L(n).

$\delta(q_0, a1 \ldots\ldots\ldots a_j a_{k+1} \ldots\ldots a_m) = \delta(\delta(q_0, a1 \ldots\ldots\ldots a_j)\ a_{k+1} \ldots\ldots a_m)$

$= \delta(q_j, a_{k+1} \ldots\ldots a_m)$ (since $\delta(q_0, a1 \ldots\ldots\ldots a_j) = q_j$)

$= \delta(q_k, a_{k+1} \ldots\ldots a_m)$ (since $q_j = q_k$)

$= q_m$

which is the final state of the automata to be accepted.

Hence proved.

1.2 Properties of Regular Sets

Property 1: The regular sets are closed under union, concatenation, and Kleene closure.

Proof: Let Σ be a finite set of symbols and let L, L_1 and L_2 be sets of strings from Σ^*. The concatenation of L_1 and L_2, denoted L_1L_2, is the set {xy| x is in $L_1$and y is in L_2}. That is, the strings in L_1L_2 are formed by choosing a string L_1 and following it by a string in L_2, in all possible combinations. Define $L^0 = \{\varepsilon\}$ and $L^i = LL^{i-1}$ for $i \geq 1$. The Kleene closure of L denoted L^*, is the set

$$L^* = \sum_{i=0} L^i$$

and the positive closure of L, denoted L^+, is the set

$$L^+ = \sum_{i=1} L^i$$

That is, L^* denotes words constructed by concatenating any number of words from L. L^+ is the same, but the case of zero words, whose concatenation is defined to be ε, is excluded. Note that L^+ contains ε if and only if L does.

Property 2: The class of regular sets is closed under complementation. That is, if L is a regular set and $L \in \Sigma^*$, and then Σ^*-L is a regular set.

Proof: Let L be L (M) for DFA M= (Q, Σ_1, δ, q_0, F) and $L \in \Sigma^*$. First, we may assume $\Sigma_1 = \Sigma$, for if there are symbols in Σ_1 not in Σ, we may delete all transitions of M on symbols not in Σ. The fact

that L$\epsilon\Sigma^*$ assures us that we shall not thereby change the language of M. If there are symbols in Σ not in Σ_1, then none of these symbols appear in words of L. We may therefore introduce a "dead state" d into M with δ (d, a) = d for all *a* in Σ and δ (q, a) = d for all q in Q and *a* in Σ-Σ_1.

Now, to accept Σ^*-L, complement the final states of M. That is, let $M' = (Q, \Sigma_1, \delta, q_0, Q-F)$. Then M' accepts a word *w* if and only if δ (q_0, w) is in Q-F, that is, *w* is in Σ^*-L. Note that it is essential to the proof that M is deterministic and without ε moves.

Property 3: The regular sets are closed under intersection.

Proof: $L_1 \cap L_2 = (L_1' \cup L_2')'$, where the overbar denotes complementation with respect to an alphabet including the alphabets of L1 and L_2.Closure under intersection then follows from closure under union and complementation.

It is worth noting that a direct construction of a DFA for the intersection of two regular sets exists. The construction involves taking the Cartesian product of states, and we sketch the construction as follows:

Let $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be two deterministic finite automata. Let

$$M = (Q_1 * Q_2, \Sigma, \delta, [q_1, q_2], F_1 * F_2)$$

where for all p_1 in Q_1, p_2 in Q_2, and *a* in Σ,

$$\delta([p_1, p_2], a) = [\delta 1 (p_1, a), \delta_2 (p_2, a)]$$

Property 4: The class of regular sets is closed under substitution.

Proof: Let R $\epsilon\Sigma^*$ be a regular set and for each *a* in Σ let $R_a \epsilon \Delta^*$ be a regular set. Let *f:* $\Sigma \rightarrow \Delta^*$ be the substitutions denoting R and R_a. Replace each occurrence of the symbol **a** in the regular expression R by the regular expression for R_a. To prove that the resulting regular expression denotes *f*(R), observe that the substitution of a union, product or closure is the union, product or closure of the substitution. [Thus for example, $f(L_1 L_2) = f(L_1) \cup f(L_2)$.] A simple induction on the number of operators in the regular expression completes the proof.

A type of substitution that is of special interest is the homomorphism. A *homomorphism h* is a substitution such that *h(a)* contains a single string for each *a*. We generally take *h(a)* to be the string itself, rather than the set containing that string. It is useful to define the *inverse homomorphic image* of a language L to be

$$h^{-1}(L) = \{\lambda \mid h(x) \text{ is in } L\}$$

We also use, for string *w*;

$$h^{-1}(w) = \{\lambda \mid h(w) = w\}$$

Property 5: The class of regular sets is closed under homomorphism and inverse homomorphism.

Proof: Closure under homomorphism follows immediately from closure under substitution, since every homomorphism is a substitution, in which h(a) has one member.

To show closure under inverse homomorphism, let M= (Q, Σ, δ, q_0, F) be a DFA accepting L, and let h be a homomorphism from Δ to Σ^*. We construct a DFA M' that accepts h^{-1}(L) by reading symbol a in Δ and simulating M on h(a). Formally, let M'= (Q, Σ, δ', q_0, F) and define δ'(q, a) for q in Q and a in Δ to be δ(q, h(a)). Note that h(a) may be long string, or ε, but δ is defined on all strings by extension. It is easy to show by induction on |x| that δ'(q_0,x)=δ(q,h(x)). Therefore M' accepts x if and only if M accepts h(x). That is, L(M')= h^{-1}(L(M)).

Property 6: The class of regular sets is closed under quotient with arbitrary sets.

Proof: Let M= (Q, Σ, δ, q_0, F) be a finite automaton accepting some regular set R, and let L be an arbitrary language. The quotient R/L is accepted b a finite automaton M'= (Q, Σ, δ, q_0, F'), which behaves like M except that the final states of M' are all states q of M such that δ (q_0, xy) is in F. Thus M' accepts R/L.

2. CONTEXT FREE GRAMMARS

Context-Free Grammars are a most powerful method of describing languages. Such grammars can describe certain features that have a recursive structure, which makes them useful in a variety of applications.

A grammar consists of a collection of substitution rules, also called productions. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a ***variable***. The string consists of variables and other symbols called ***terminals***. The variable symbols often are represented by capital letters. The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols. One variable is designed as the ***start variable***. It usually occurs on the left-hand side of the topmost rule. For example, grammar G_1 contains three rules. G_1's variables are A and B, where A is the start variable. Its terminals are 0, 1 and #.

You use a grammar to describe a language by generating each string of that language in the following manner.

1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
3. Repeat step 2 until no variables remain.

For example, grammar G_1 generates the string 000#111. The sequence of substitutions to obtain a string is called a derivation. A derivation of string 000#111 in grammar G_1 is

A=>0A1=>00A11=>000A111=>000B111=>000#111

You may also represent the same information pictorially with a ***parse tree*.** An example of a parse tree is shown below

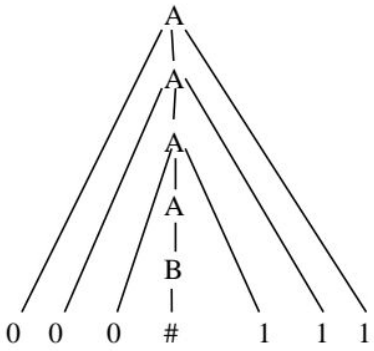


Fig: Parse tree for 000#111 in Grammar G_1

All strings generated in this way constitute the ***language of the grammar***. We write L (G_1) for the language of grammar G_1. Some experimentation with the grammar G_1 shows us that L (G_1) is $\{0^n\#1^n | n \geq 0\}$. Any language that can be generated by some context-free grammar is called a ***context-free language*** (CFL).

FORMAL DEFINITION OF A CONTEXT-FREE GRAMMAR

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. S ε V is the start variable.

3. DERIVATION TREES

The derivations in CFG can be represented using trees. Such trees representing derivations are called derivation trees.

A derivation tree for a CFG G= (V, T, P, S) is a tree satisfying the following conditions:

(a) Every vertex has a label which is a variable or terminal or ε

(b) The root has label S.

(c) The label of an internal vertex is a variable.

(d) If the vertices $n_1,n_2\ldots\ldots n_k$ written with labels $x_1,x_2\ldots\ldots x_k$ are the sons of vertex n with label A, then A-> $x_1,x_2\ldots\ldots x_k$ is a production in P.

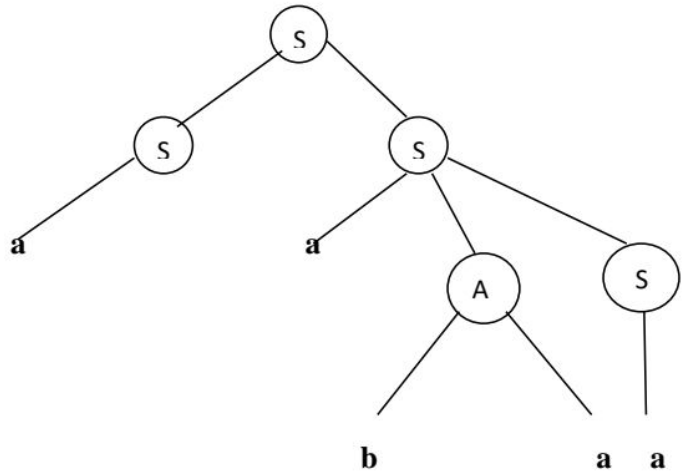
(e) A vertex n is a leaf if its label is a€Σ or ε.

For example:

Let G= ({S, A}, {a, b}, P, S) where P consists of S->aAS/a/SS

A->SbA/ba

The derivation tree is given for the string aabaa as below:



2.1 LEFTMOST AND RIGHTMOST DERIVATION

Let G be a grammar S->0B/1A

A->0/0S/1AA

B->1/1S/0BB

For the string 00110101

Leftmost derivation:

S->0B

->00BB

->001B

->0011S

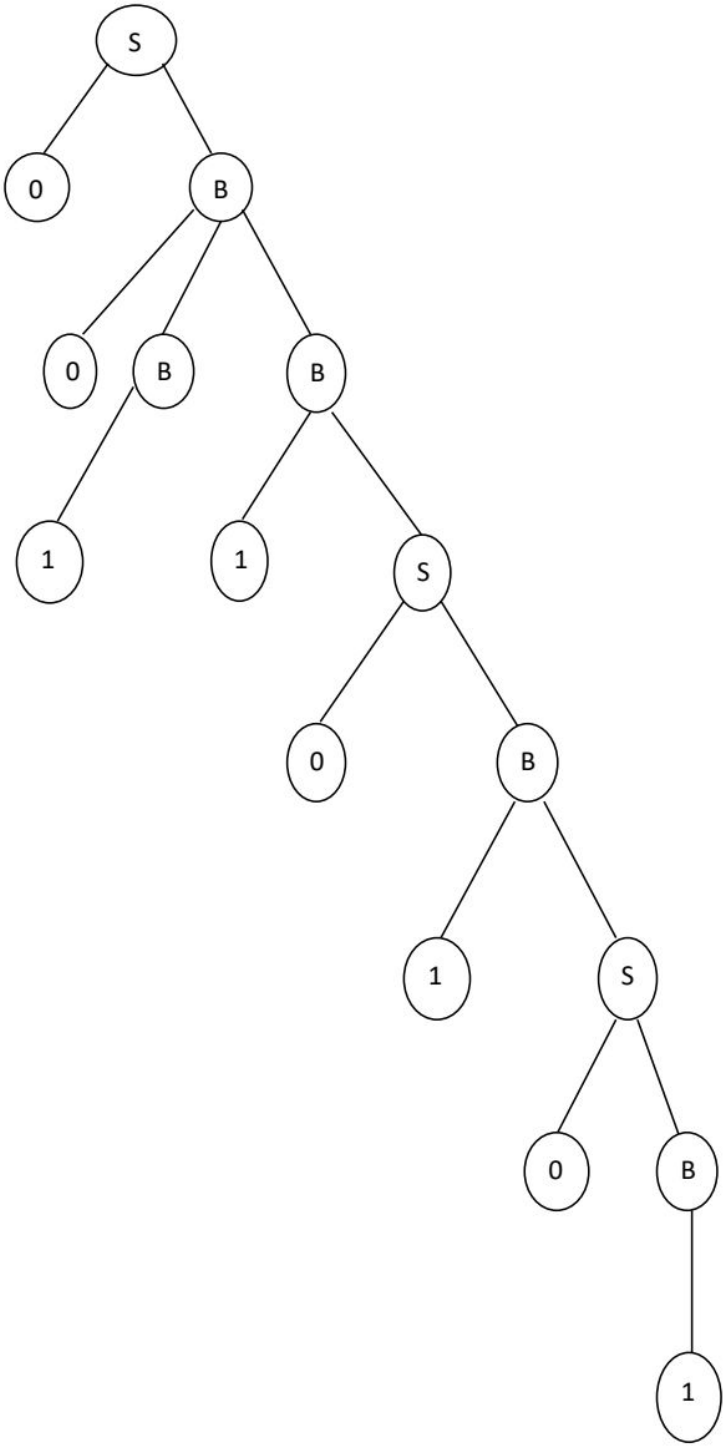
->00110B

->001101S

->0011010B

->00110101

Tree:



Rightmost derivation:

S->0B

->00BB

->00B1

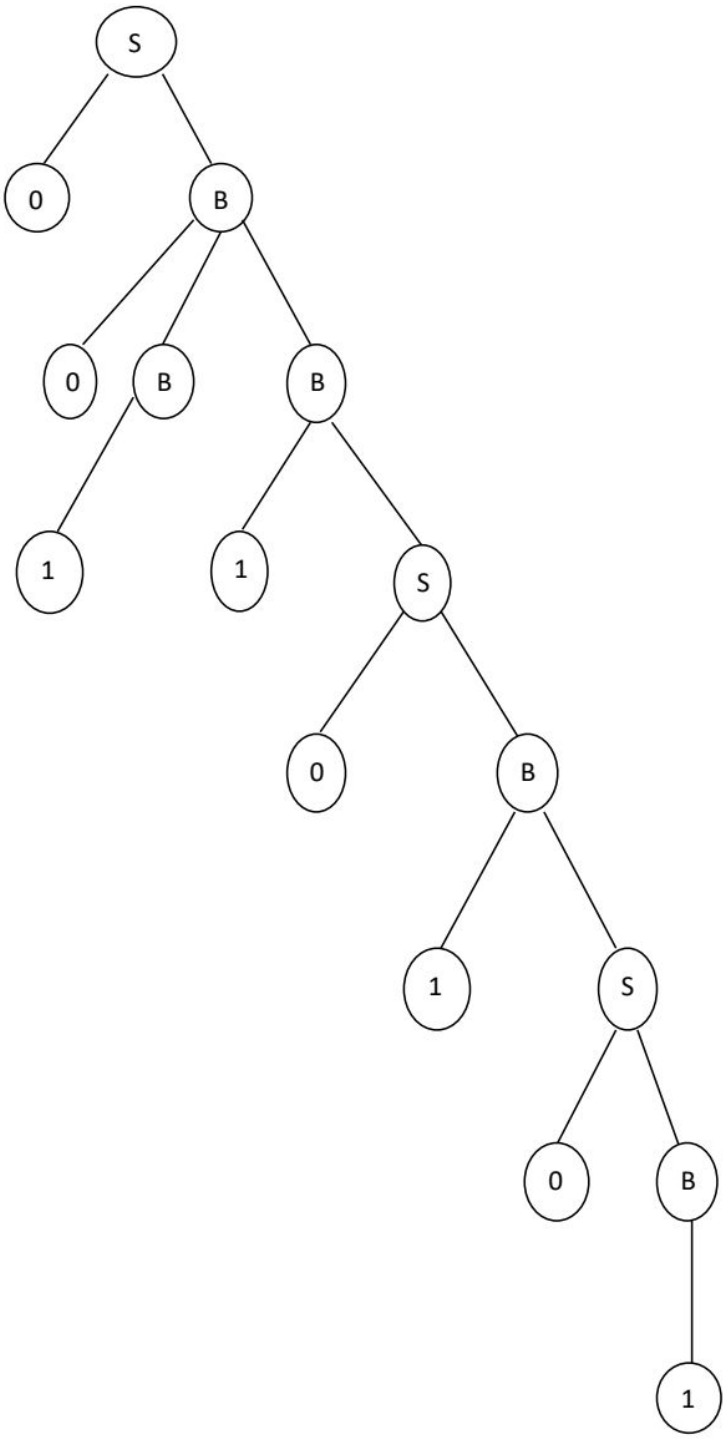
->001S1

->0011A1

->00110S1

->001101A1

->00110101



Example: Consider the following production

S->aB/bA

A->aS/bAA/a

B->bS/aBB/b

for string aaabbabbba find LMD and RMD

LMD:

S->a<u>B</u>

->aa<u>B</u>B

->aaa<u>B</u>BB

->aaab<u>B</u>B

->aaabb<u>B</u>

->aaabba<u>B</u>B

->aaabbab<u>B</u>

->aaabbabb<u>S</u>

->aaabbabbb<u>A</u>

->aaabbabbba

RMD:

S->a<u>B</u>

->aa<u>B</u>B

->aaBb<u>S</u>

->aaBbb<u>A</u>

->aa<u>B</u>bba

->aaa<u>B</u>Bbba

->aaa<u>B</u>bbba

->aaab<u>S</u>bbba

->aaabb<u>A</u>bbba

->aaabbabbba

4. CHOMSKY NORMAL FORM (CNF)

A Context-Free Grammar 'G' is in Chomsky Normal Form if every production is of the form

- A->a (Non Terminal->Terminal)
- A->BC (Non Terminal-> Non Terminal*Non Terminal)

is in G.

For Example:

Consider "G" whose productions are S->AB, A->a, B->b. Then G is in Chomsky Normal Form.

Reduction to Chomsky Normal Form

The steps involved in reduction of CFG to CNF are

Step 1: Elimination of null productions and unit productions.

Let the resultant grammar be G= (V_N, Σ, P, S)

Step 2: Elimination of terminals on RHS.

We define G= (V_N', Σ, P, S') where P and V_N' are constructed as follows:

(i)All the productions in P of the form A->a or A->BC are included in P1. All the variables in V_N are included in V_N'.

(ii) Consider A->X_1X_2…….X_n with some terminal on RHS of X_i is a terminal say a_i, add a new variable C_{ai} to V_N' and C_a->α_i to P1. In production A->X_1X_2……X_n, every terminal on RHS is replaced by the corresponding new variables on the RHS are retained. The resulting production is added to P_1. Thus we get G_1= (V_N', Σ, P_1, S).

Step 3: Restricting the number of variables on RHS.

For any production in P_1, the RHS consists of either a single terminal or two or more variables. We define G= (V_N', Σ, P_2, S) as follows:

(i)All productions in P_1 are added to P_2 if they are in the required form. All the variables in V_N''.

(ii) Consider A->A_1c_1, C_1->A_2C_2, c_{m-2}->$A_{m-1}A_m$ and new variables are $c_1, c_2, ..., c_{m-2}$.

Example 1:

Find the grammar in CNF equivalent to S->aAbB, A->aA/a, B->bB/b

Step 1: As there are no unit productions or null productions, we need not carry out step 1. We proceed to step 2.

Step 2: Let G_1= (V_N', {a, b}, P, S) where P_1 and V_N' are constructed.

Add productions A->a, B->b to P_1.

Hence S->aAbB ~ (1)

replaced as

a) S->C_aAC_bB

   Add new productions C_a->a and C_b ->b

b) A->aA becomes

   A->C_aA

c) B->bB becomes

   B->C_bB

Step 3:

a) S->C_aAC_bB is converted as

   S->$C_a$$D_1$ where

   D_1->AC_bB

   Again converted as D_1->AD_2

   where D_2->C_bB

b) A->C_aA

   They are already in CNF

c) B->C_bB

   They are already in CNF

   Hence the resultant CNF are

S->$C_a$$D_1$
D_1->AD_2
D_2->C_bB
A->C_aA
B->C_bB
A->a
B->b

Example 2:

Find the CNF equivalent to the grammar S->~S/[S∩S]/p/q

where ~, [, ∩,], p, q are terminals.

Step 1: Consider the Grammar "G".

S->~S/[S∩S]/p/q

Since there are no null productions and unit productions we can go for reduction.

Step 2: Consider the production (1),

S->~S

Add new production C_1->~

The production S->~S becomes S->C_1S

Step 3: Consider the production (2),

S-> [S∩S]

Add new productions C_2-> [

C_3->∩

C_4->]

Now S->$C_2SC_3SC_4$

S-> C_2D_1

where D_1-> SC_3SC_4

and D_2-> C_3SC_4

So D_1-> SD_2

Again D_2-> C_3SC_4

Let D3-> SC_4

And hence D_2-> C_3D_3

The resultant CNF productions are

S->C_1S

C_1->~

S-> C_2D_1

D_1-> SD_2

D_2-> C_3D_3

D3-> SC_4

S->p

S->q

Hence derived.

Solved Problems:

Note: NT-Terminal T-Terminal

1. **Convert the following grammar to Chomsky Normal Form(CNF)**

   G:

   S->aAD

   A->aB/bAB

   B->b

   D->d

Solution:

Consider S->aAD

The production is of the form NT->T*NT*NT

Replace a by C_a. We get, S->C_aAD

Let D_1->AD. Then the production becomes S->C_a D_1 which is in CNF.

Now consider A->aB

This is of the form NT->T*NT

Since C_a->a, A->aB becomes A-> C_aB which is in CNF.

Let us now consider A->bAB.

This production is of the form NT->T*NT*NT

Replace b by C_b and AB by D_2. We get, A-> C_b D_2.

The other two productions B->b and D->d are already in CNF.

Hence the resultant CNF productions are

$S \rightarrow C_a D_1$
$D_1 \rightarrow AD$
$A \rightarrow C_aB$
$A \rightarrow C_b D_2$.
D_2. $\rightarrow AB$
$B \rightarrow b$
$D \rightarrow d$

2. Convert the following grammar to Chomsky Normal Form (CNF)

G:

S->abSb/a/aAb

A->bS/aAAb

Solution:

Consider S->abSb

This production is of the form NT->T*NT*T*NT

Replacing a by C_a and b by C_b we get, $S \rightarrow C_aC_bSC_b$

This is of the form NT->NT*NT*NT*NT.

So we replace C_bSC_b by D_1. Hence the production becomes, $S \rightarrow C_a D_1$ which is of the CNF.

(Replace SC_b by D to convert D_1 into CNF. So D-> SC_b)

Since S->a is already in CNF we now consider S->aAB

The above production is of the form NT->T*NT*T. Replace a by C_a and b by C_b.

So now, S-> C_aAC_b. Replace AC_b by D_2 to convert the production to CNF.

The production now becomes $S \rightarrow C_a D_2$.

Let us now consider the production A->bS. Replacing b by C_b the grammar gets converted into CNF, $A \rightarrow C_bS$.

The last production to be converted to CNF is A->aAAb.

Replacing C_a and b by C_b we get A-> C_aAAC_b. Let us now consider $D_3 \rightarrow AD_2$ where $D_2 \rightarrow AC_b$. So the production becomes $A \rightarrow C_a D_3$ which is in CNF.

Hence the resultant CNF productions are

$D \rightarrow SC_b$

$D_1 \rightarrow C_bD$

$S \rightarrow C_a D_1$

$D_2 \rightarrow AC_b$

$S \rightarrow C_a D_2$

$A \rightarrow C_aD_3$

$D_3 \rightarrow AD_2$

$D_2 \rightarrow AC_b$

$S \rightarrow a$

3. Convert the following grammar to Chomsky Normal Form (CNF)

G:

S->ASA/bA

A->B/S

B->C

Solution:

Consider S->ASA.

The above production is of the form NT->NT*NT*NT

Let $D_1 \rightarrow SA$. Replacing SA the production becomes $S \rightarrow AD_1$ which is in CNF.

Now consider S->bA. Replace b by C_b in order to convert the production to CNF.

S->bA becomes $S \rightarrow C_bA$.

Consider the productions A->B, A->S and B->C. They can also be written as A->εB, A->εS, B->εC. If we assume $C_a \rightarrow \varepsilon$ then the productions become $A \rightarrow C_aB$, $A \rightarrow C_aS$, $B \rightarrow C_aC$ which are in CNF.

Hence the resultant CNF productions are

$S \rightarrow AD_1$

$D_1 \rightarrow SA$

$A \rightarrow C_aB$

$S \rightarrow C_b A$

$A \rightarrow C_aS$

$B \rightarrow C_aC$

4. Convert the following grammar to Chomsky Normal Form (CNF)

G:

S->1A/0B

A->1AA/0S/0

B->0BB/1S/1

Solution:

Consider S->1A. The above production is of the form NT->T*NT. In order to convert it to CNF, we replace 1 by C_a. Upon replacing we get, $S \rightarrow C_aA$ which is in CNF.

Now consider S->0B. The above production is of the form NT->T*NT. In order to convert it to CNF, we replace 0 by C_b. Upon replacing we get, $S \rightarrow C_bB$ which is in CNF.

The next production to be considered is A->1AA which is of the form NT->T*NT*NT. Let us assume that $D_1 \rightarrow AA$. Also we know $C_a \rightarrow 1$. The production becomes $A \rightarrow C_aD_1$ which is in CNF.

The next production A->0S is of the form NT->T*NT. We know $C_b \rightarrow 0$. Upon replacing we get, $A \rightarrow C_bS$ which is in CNF. The other production A->0 is already in CNF.

Now take into account the production B->0BB which is of the form NT->T*NT*NT. Let us assume that $D_2 \rightarrow BB$. Also we know $C_b \rightarrow 0$. The production becomes $B \rightarrow C_bD_2$ which is in CNF.

The next production B->1S is of the form NT->T*NT. We know $C_a \rightarrow 1$. Upon replacing we get, $B \rightarrow C_aS$ which is in CNF. The other production B->1 is already in CNF.

Hence the resultant CNF productions are

$S->C_aA$
$S->C_bB$
$D_1->AA$
$A-> C_aD_1$
$A-> C_bS$
$A->0$
$B-> C_aS$
$B->0$

5. GREIBACH NORMAL FORM (GNF)

A Context-Free Grammar 'G' is in Greibach Normal Form if every production is of the form

- $A->a\alpha$ where $\alpha \in V_N^*$ and $a \in \Sigma$

  (Non Terminal->Terminal*any number of Non Terminals)
- A->a (Non Terminal->Terminal)

For Example:

S->aAB

A->bC

B->b

C->c

are in GNF.

Greibach Normal Form Algorithm:

```
begin
        for k:=1 to m do
        begin
         for j=1 to k-1 do
          for each production of the form Ak->Ajα do
            begin
                for all productions Aj->β do
                    add production Ak->βα;
                    remove production Ak->Ajα
               end;
          for each production of the form Ak->Akα do
            begin
                add production Bk->α and Bk->α Bk;
                 remove production Ak->Akα
            end;
         for each production Ak->β, where β does
        not begin with Ak do
            add production Ak->β Bk
        end
end
```

Reduction to Greibach Normal Form

Lemma (1):

Let G= (V, Σ, P, S) be a CFG. Let A->Bγ be an A-production in P. Let the B-production be $B->B_1|B_2|.....|B_n$

Lemma (2):

Let G= (V, Σ, P, S) be a CFG. Let A->Bγ be an A-production be

$A->A\alpha_1|..... A\alpha_\gamma| B_1|.....|B_n$

then z be a new variable when P_1 is defined as:

(i)The set of A-productions in P_1 are

$A->\beta_1|\beta_2|....|\beta_n$

$A-> \beta_1 z| \beta_2 z|.......| \beta_n z$

(ii)The set of z-productions in P_1 are

$z->\alpha_1|\alpha_2|...|\alpha_n$

$z->\alpha_1 z|\alpha_2 z|...|\alpha_n z$

Example:

Construct Equivalent GNF for the CFG S->AA/a,A->SS/b

Step 1: The given grammar is in CNF. S and A are renamed as A_1 and A_2.

Hence the productions become $A_1-> A_2 A_2/a$, $A_2-> A_1 A_1/b$

There is no need for null production or unit production elimination because the production is already in CNF.

*Step 2:*The A_1 productions are in the required form. They are $A_1-> A_2 A_2/a$.

The production $A_2->b$ is also in the required form whereas we have to apply lemma 1 to convert the production $A_2-> A_1 A_1$ to the required form. Applying lemma 1 we get,

$A_2-> A_2 A_2 A_1$ $A_2-> aA_1$

Step 3: We apply lemma 2 to A_2 productions as we have $A_2-> A_2 A_2 A_1$

Let z_2 be a new variable. The resulting productions are

$A_2-> aA_1$	$A_2->b$
$A_2-> aA_1z_2$	$A_2->bz_2$
$z_2 -> A_2 A_1$	$z_2 -> A_2 A_1 z_2$

Step 4: (i) The A_2 productions are $A_2 \rightarrow aA_1/\ b/\ aA_1z_2$

(ii)Among the A_1 productions we retain $A_1 \rightarrow a$ and eliminate $A_1 \rightarrow A_2\ A_2$ using lemma 1.

The resulting productions are $A_1 \rightarrow aA_1\ A_2/bA_2$, $A_1 \rightarrow aA_1z\ A_2/bz_2A_2$.

The set of all A_1-productions is $A_1 \rightarrow a/\ aA_1\ A_2/bA_2/\ aA_1z\ A_2/bz_2A_2$

Step 5: The z_2 productions to be modified are $z_2 \rightarrow A_2\ A_1$, $z_2 \rightarrow A_2\ A_1\ z_2$.

Applying lemma 1 we get $z_2 \rightarrow aA_1A_1/bA_1\ /aA_1z_2\ A_1/\ bz_2A_1$

$z_2 \rightarrow aA_1A_1z_2/bA_1\ z_2\ /aA_1z_2\ A_1\ z_2/\ bz_2A_1\ z_2$

Hence the equivalent grammar is

$A_1 \rightarrow a/\ aA_1\ A_2/bA_2/\ aA_1z\ A_2/bz_2A_2$
$A_2 \rightarrow aA_1/\ b/\ aA_1z_2/\ bz_2$
$z_2 \rightarrow aA_1A_1/bA_1\ /aA_1z_2\ A_1/\ bz_2A_1\ z_2$
$z_2 \rightarrow aA_1A_1z_2/bA_1\ z_2\ /aA_1z_2\ A_1\ z_2/\ bz_2A_1\ z_2$

Solved Problems:

Note: NT-Terminal T-Terminal

1. **Let us convert to Greibach normal form the grammar**

   G= ({A_1, A2, A3}, {a, b}, P, A_1), where P consists of the following:

   $A_1 \rightarrow A_2A_3$

   $A_2 \rightarrow A_3A_1/b$

   $A_3 \rightarrow A_1A_2/a$

Solution:

Step 1: Since the right-hand side of the productions for A_1 and A_2 start with terminals or highest-numbered variables, we begin with the production $A_3 \rightarrow A_2A_3$ is the only production with A_1 on the left.

The resultant set of productions is: $A_1 \rightarrow A_2A_3$

$A_2 \rightarrow A_3A_1/b$

$A_3 \rightarrow A_2A_3A_2/a$

Since the right side of the production $A_3 \rightarrow A_2A_3A_2$ begins with a lower-numbered variable, we substitute for the first occurrence of A_2 both A_3A_1 and b. Thus $A_3 \rightarrow A_2A_3A_2$ is replaced by $A_3 \rightarrow A_3A_1A_3A_2$ and $A_3 \rightarrow bA_3A_2$. The new set is

$A_1 \rightarrow A_2A_3$

$A_2 \rightarrow A_3A_1/b$

$A_3 \rightarrow A_3A_1A_3A_2 / bA_3A_2/a$

We now apply lemma 2 to the productions

$A_3 \rightarrow A_3A_1A_3A_2 / bA_3A_2/a$

Let z_2 be a new variable. The resulting productions are

$A_1 \rightarrow A_2A_3$

$A_2 \rightarrow A_3A_1/b$

$A_3 \rightarrow bA_3A_2z_2/az_2/ bA_3A_2/a$

$z_2 \rightarrow A_1A_3A_2 / A_1A_3A_2z_2$

Step 2: Now all the productions with A_3 on the left have right-hand sides that start with terminals. These are used to replace A_3 in the productions $A_2 \rightarrow A_3A_1$ and then the productions with A_2 on the left are used to replace A_2 in the production $A_1 \rightarrow A_2A_3$. The result is the following.

$A_3 \rightarrow bA_3A_2z_2$ $A_3 \rightarrow bA_3A_2$

$A_3 \rightarrow az_2$ $A_3 \rightarrow a$

$A_2 \rightarrow bA_3A_2z_2A_1$ $A_2 \rightarrow bA_3A_2A_1$

$A_2 \rightarrow az_2A_1$ $A_2 \rightarrow aA_1$

$A_2 \rightarrow b$

$A_1 \rightarrow bA_3A_2z_2A_1A_3$ $A_1 \rightarrow bA_3A_2A_1A_3$

$A_1 \rightarrow b\ z_2A_1A_3$ $A_1 \rightarrow b\ A_1A_3$

$A_1 \rightarrow bA_3$

$z_2 \rightarrow A_1A_3A_2$ $z_2 \rightarrow A_1A_3A_2z_2$

Step 3: The two z_2 productions are converted to proper form, resulting in 10 more productions. That is, the productions

$z_2 \rightarrow A_1A_3A_2$ $z_2 \rightarrow A_1A_3A_2z_2$

are altered by substituting the right side of each of the five productions with A_1 on the left for the first occurrences of A_1. Thus $z_2 \rightarrow A_1A_3A_2$ becomes

$z_2 \rightarrow bA_3A_2z_2A_1A_3\ A_3A_2$ $z_2 \rightarrow az_2A_1A_3A_3A_2$

$z_2 \rightarrow bA_3A_3A_2$ $z_2 \rightarrow bA_3A_2z_2A_1A_3\ A_3A_2$

$z_2 \rightarrow aA_1A_3A_3A_2$

The other production for z_2 is replaced similarly. The final set of productions is

$A_3 \rightarrow bA_3A_2z_2$ $A_3 \rightarrow bA_3A_2$

$A_3 \rightarrow az_2$ $A_3 \rightarrow a$

$A_2 \rightarrow bA_3A_2z_2A_1$ $A_2 \rightarrow bA_3A_2A_1$

$A_2 \rightarrow az_2A_1$ $A_2 \rightarrow aA_1$

$A_2 \rightarrow b$

$A_1 \rightarrow bA_3A_2z_2A_1A_3$ $A_1 \rightarrow bA_3A_2A_1A_3$

$A_1 \rightarrow b\ z_2A_1A_3$ $A_1 \rightarrow b\ A_1A_3$

$A_1 \rightarrow bA_3$

$z_2 \rightarrow bA_3A_2z_2A_1A_3\ A_3A_2$ $z_2 \rightarrow bA_3A_2z_2A_1A_3\ A_3A_2$

$z_2 \rightarrow az_2A_1A_3A_3A_2z_2$ $z_2 \rightarrow az_2A_1A_3A_3A_2$

$z_2 \rightarrow bA_3A_3A_2z_2$ $z_2 \rightarrow bA_3A_3A_2$

$z_2 \rightarrow bA_3A_2A_1A_3\ A_3A_2z_2$ $z_2 \rightarrow bA_3A_2A_1A_3\ A_3A_2$

$z_2 \rightarrow aA_1A_3A_3A_2z_2$ $z_2 \rightarrow aA_1A_3A_3A_2$

5. AMBIGUOUS AND UNAMBIGUOUS GRAMMARS

Ambiguous Grammar: The production should be in GNF and if the production contains more than one derivation or derivation tree then it is termed as an ambiguous grammar.

For example consider the production, S->S+S/S*S/a/b

We have to derive the string a+a*b

The derivation is as follows:

S->S+S

->S+S*S

->a+a*b

Hence derived.

The same string can also be derived in the following manner

S->S*S

->S+S*S

->a+a*b

Hence from the above two derivations it is obvious that the string can be derived from more than one derivation. Therefore it is known as ambiguous grammar.

Unambiguous Grammar: If the production contains exactly one derivation or one derivation tree then it is termed as an unambiguous grammar.

Consider an example:

S->a/abSb/aAb

A->bS/aAAb, the word to be derived is W=abab

Derivation:

S->abSb

->abab

Hence derived.

Now consider

S->aAb

->abSb

->ababSbb

->abababbb

S->abSb

-> ababSbb

->abababbb

A->aAAb

->abSbSb

->abababab

A->bS

->ba

From the above derivations it is obvious that the string abab can have only one derivation. Hence the grammar can be termed as an unambiguous grammar.

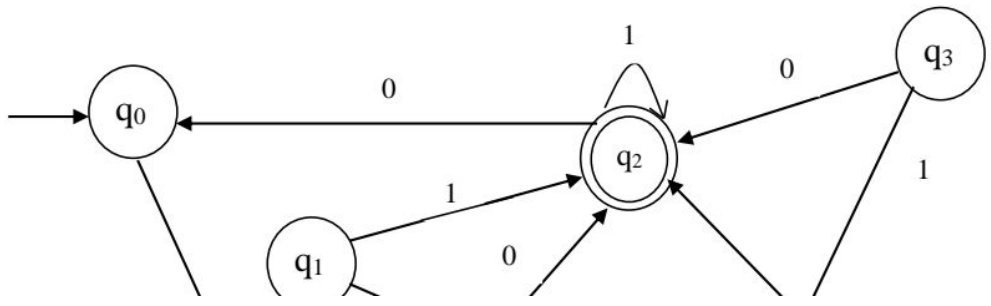
6. THE MYHILL-NERODE THEOREM

```
Begin

        for p in F and q in Q-F do mark (p,q);
        for each pair of distinct states (p,q)          in F*F
or (Q-F)*(Q-F) do
                if for some input a,(δ(p,a), δ(q,a)) is
        marked then
        begin
                Mark(p,q)
        recursively mark all unmarked pairs on the
        list for (p,q) and on the lists of the other
        pairs  that are marked at this step.
        end
        else
        for all input symbols a do
                Put(p,q) on the list for ,(δ(p,a), δ(q,a))
                unless δ(p,a)= δ(q,a)
end
```

Construction of Minimum Automaton:

Consider the following transition diagram



Now construct the transition table:

δ	0	1
q_0	q_1	q_5
q_1	q_6	q_2
q_2	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

THE MYHILL-NERODE THEOREM

```
Begin

        for p in F and q in Q-F do mark (p,q);
        for each pair of distinct states (p,q)          in F*F
or (Q-F)*(Q-F) do
                if for some input a,(δ(p,a), δ(q,a)) is
        marked then
        begin
                Mark(p,q)
        recursively mark all unmarked pairs on the
        list for (p,q) and on the lists of the other
        pairs  that are marked at this step.
        end
        else
        for all input symbols a do
                Put(p,q) on the list for ,(δ(p,a), δ(q,a))
                unless δ(p,a)= δ(q,a)
end
```

Step 1: Construction of π_0, by def 0-equivalence

$\Pi_0 = \{Q_1^0, Q_2^0\}$ where

Q_1^0 is the set of final states and

$Q_2^0 = Q - Q_1^0$

hence

$Q_1^0 = F = \{q_2\}$

$Q_2^0 = Q - Q_1^0$

So

$\Pi_0 = \{\{q_2\}, \{q_0, q_1, q_3, q_4, q_5, q_6, q_7\}\}$

The $\{q_2\}$ in π_0 cannot be partitioned.

So $Q_1' = \{q2\}$

Consider q_0 and $q_1 \in Q_2^0$.

The entries under the 0 – column corresponding to q_0 and q_1 are q_1 and q_6; they lie in Q_2^0. The entries in 1- column are q_5 and q_2, $q_2 \in Q_1^0$ and $q_5 \in Q_2^0$. Therefore q_0 and q_1 are not 1-Equivalent. Similarly q_0 is 1 – Equivalent to q_3, q_5 and q_7.

Now consider q_0 and q_4. The entries under the 0-column are q_1 and q_7 both are in Q_2^0. The entries under 1-column are q_5, so q_4 and q_0 are 1- Equivalent.

Similarly q_0 is 1- Equivalent to q_6,

So, $Q_2' = \{q_0, q_4, q_6\}$

Repeat the construction by considering q, as one of the states q_3, q_5, q_7. Now q_1 is not 1 – Equivalent to q_3 and q_5 but 1 – Equivalent to q_7 hence $Q_3' = \{q_1, q_7\}$. The left over elements in Q_2^0 are q_3 and q_5.

Therefore,

$\Pi_1 = \{\{q_2\}, \{q_0, q_4, q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$

The $\{q_2\}$ is also in Π_2 as it cannot be further partitioned. Now the entries under 0 – column to q_0 and q_4 are q_1 and q_7 and these lie in the same equivalence class in Π_1. The entries under 1-column are q_5. So q_0 and q_4 are 2- Equivalent. But q_0 and q_6 are not 2-Equivalent. Hence $\{q_0, q_4, q_6\}$ is partitioned into $\{q_0, q_4\}$ and $\{q_6\}$. q_1 and q_7 are 2-Equivalent.

$\Pi_2 = \{\{q_2\}, \{q_0, q_4\}, \{q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$

q_0 and q_4 are 3 – Equivalent

Therefore

$$\Pi_3 = \{\{q_2\}, \{q_0, q_4\}, \{q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$$

The minimum state automaton is

$$M' = (Q', \{0, 1\}, \delta', q_0', F')$$

where

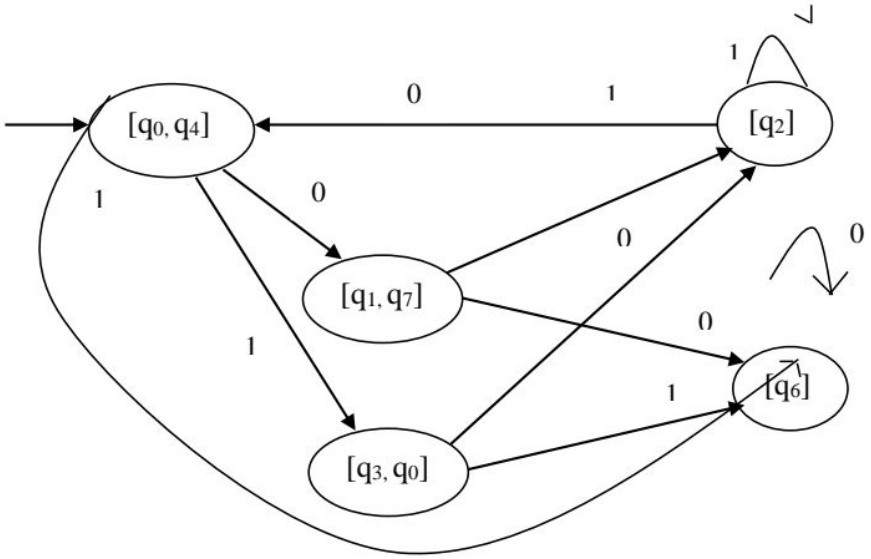
$$Q' = \{[q_2], [q_0,q_4], [q_6], [q_1,q_7], [q_3,q_5]\}$$

$$q_0' = [q_0, q_4]$$

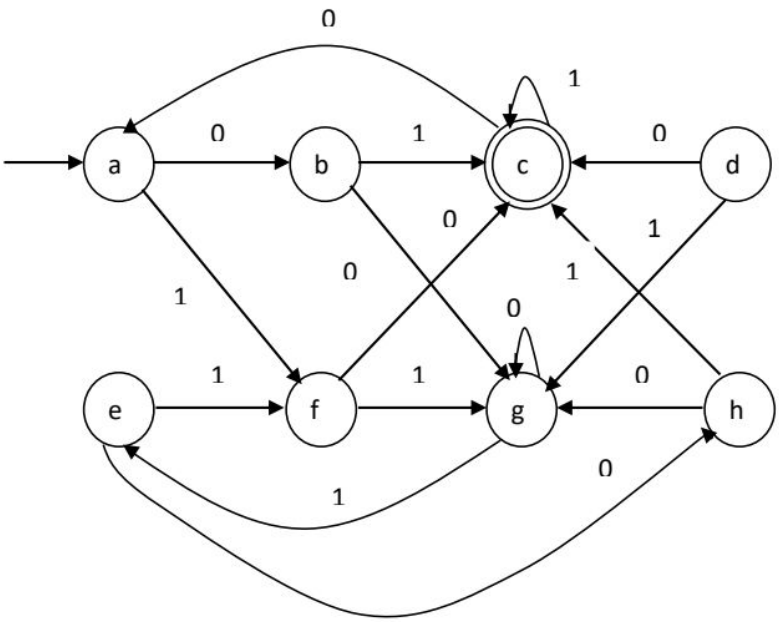
$$F' = [q_2]$$

and δ' is defined by

δ	0	1
$[q_0, q_4]$	$[q_1, q_7]$	$[q_3, q_5]$
$[q_1, q_7]$	$[q_6]$	$[q_2]$
$[q_2]$	$[q_0, q_4]$	$[q_2]$
$[q_3, q_5]$	$[q_2]$	$[q_6]$
$[q_6]$	$[q_6]$	$[q_0, q_4]$



Example 2: Minimize the finite automata given below



The transition table can be given as

δ	0	1
a	b	f
b	g	c
c	f	c
d	c	g
e	h	f
f	c	g
g	g	e
h	g	c

$\Pi_0 = \{\{c\}, \{a, b, d, e, f, g, h\}\}$

Where class A = {c} and Class B = {a, b, d, e, f, g, h}

Class B:

Case (i) : comparing a & b

$\delta(a,0) = b \in B$

$\delta(b,0) = g \in B$

$\delta(a,1) = f \in B$

$\delta(b,1) = c \in A$

So b can be partitioned from a

Case (ii) : comparing a & d

$\delta(a,0) = b \in B$

$\delta(d,0) = c \in A$

$\delta(a,1) = f \in B$

$\delta(d,1) = g \in B$

So d can be partitioned from a

Case (iii): comparing a & e

$\delta(a,0) = b \in B$

$\delta(e,0) = h \in B$

$\delta(a,1) = f \in B$

$\delta(e,1) = f \in B$

So e can't be partitioned from a

Case (iv) : comparing a & f

$\delta(a,0) = b \in B$

$\delta(f,0) = c \in A$

$\delta(a,1) = f \in B$

$\delta(f,1) = g \in B$

So f can be partitioned from a

Case (v): comparing a & g

$\delta(a,0) = b \in B$

$\delta(g,0) = g \in B$

$\delta(a,1) = f \in B$

$\delta(g,1) = e \in B$

So g can't be partitioned from a

Case (vi): comparing a & h

$\delta(a,0) = b \in B$

$\delta(h,0) = g \in B$

$\delta(a,1) = f \in B$

$\delta(h,1) = c \in A$

So h can be partitioned from a

$\Pi_1 = \{\{c\}, \{a, e, g\}, \{b, d, f, h\}\}$

Class B:

Case (i): comparing a & e

$\delta(a,0) = b \in C$

$\delta(e,0) = h \in C$

$\delta(a,1) = f \in C$

$\delta(e,1) = f \in C$

So e can't be partitioned from a

Case (i): comparing a & g

$\delta(a,0) = b \in C$

$\delta(g,0) = g \in B$

δ (a,1) = f $\in$ C

δ (g,1) = e $\in$ B

So g can't be partitioned from a

Class C

Case (i): comparing b & d

δ (b,0) = g $\in$ B

δ (d,0) = c $\in$ A

δ (b,1) = c $\in$ A

δ (d,1) = g $\in$ B

So d can be partitioned from b.

Case (i): comparing b & f

δ (b,0) = g $\in$ B

δ (f,0) = c $\in$ A

δ (b,1) = c $\in$ A

δ (f,1) = g $\in$ B

So f can be partitioned from b.

Case (iii): comparing b & h

δ (b,0) = g $\in$ C

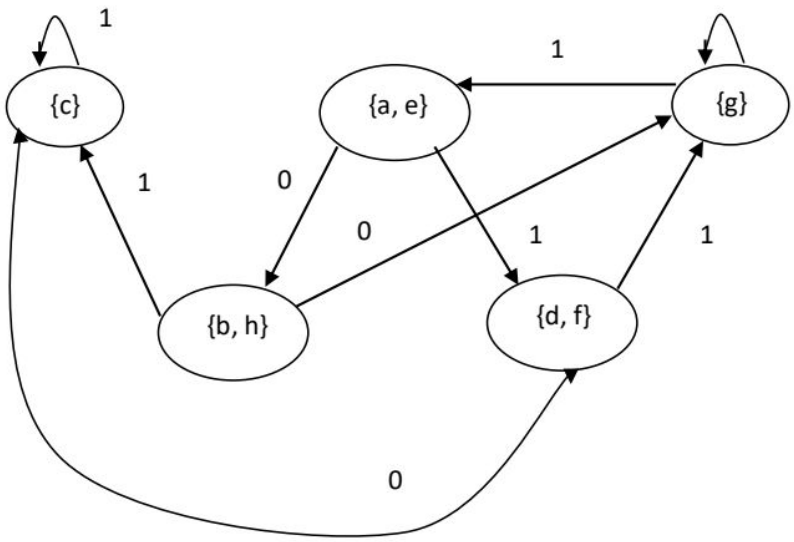
δ (h,0) = g $\in$ C

δ (b,1) = c $\in$ C

δ (h,1) = c $\in$ C

So h can't be partitioned from b.

Π_2 = {{c}, {a, e}, {g} {b, h}, {d, f}}


1
{c}
1
{a, e}
{g}
1
0
0
1
1
{b, h}
{d, f}
0

UNIT-3

PUSHDOWN AUTOMATA AND PARSING ALGORITHMS: pushdown automata and contextfree languages;top down parsing and bottomup parsing;properties of cfl;application of pumping lemma,closure properties of cfl and decision algorithms.

PUSHDOWN AUTOMATA:

APPLICATION:

Regular languages are those which are accepted by finite automata.but a language such as l=(a^nb^n/n>=1) cannot be accepted by a regular language because it has to precisely remember the number of a's and b's which would require infinite number of states.hence push down automata has been designed to accept context free languages.this is done by adding auxillary memory in the form of stack.

DEFINITIONS:

The pushdown automata formly consist of 7 tuples given by M=(q,∑,$^{\delta}$, ┌,q0,z0,f)

Where,

q:a finite non empty set of states

∑:a finite non empty set of input symbols

$:the transition function from q×(∑$^{\cup}${^})×┌ to the set of finite subsets q×┌^*

┌:finite non empty set of push down symbols

Q0:a special state called initial state

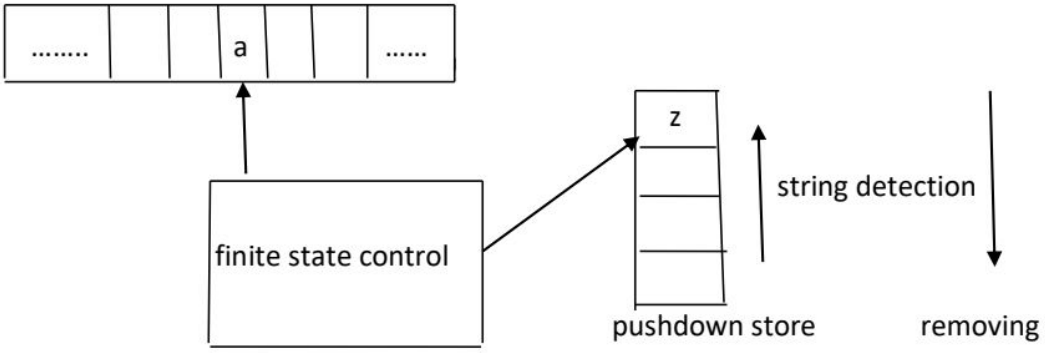
Z0:a special push down symbols called the initial symbols on the pushdown storedenoted by z0.

In otherwords indicates the top of the stack

F:the pushdown symbol denoting the final state.

COMPONENTS OF PUSH DOWN AUTOMATA:

The following diagram denotes the basic model of a pushdown automata



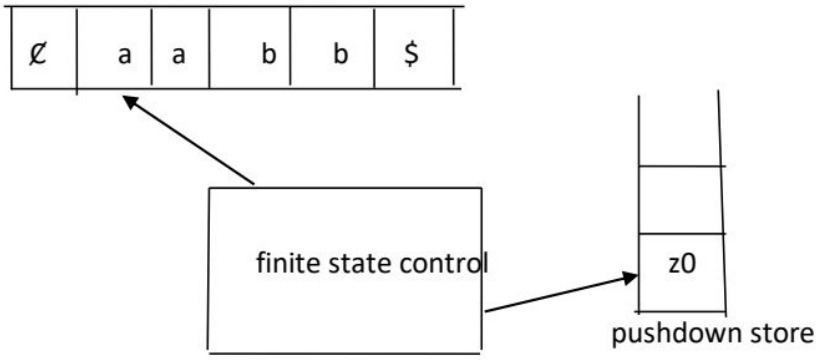
it has a read only input tape,input alphabet,finite state control,set of final states and an initial state as in the case of finite automata.

But in addition to this it has a stack called the pushdown store,it is a read write pushdown store as we add elements to the pds or remove elements from pds.a finite automata is in same state and on reading an input symbol the top most symbol in pds moves to a new state and writes a string of symbols in the pds.

DESIGN FOR PUSH DOWN AUTOMATA:

Consider the context free languages a^nb^n,n>=1,from this sample it can be concluded that the language consist of equal number of a's as equal number of b's.let us consider the case wherw n=2,then we

get the string aabb,the following figure shows the structure of pda for the above string.



there are three states to be considered in this case they are:

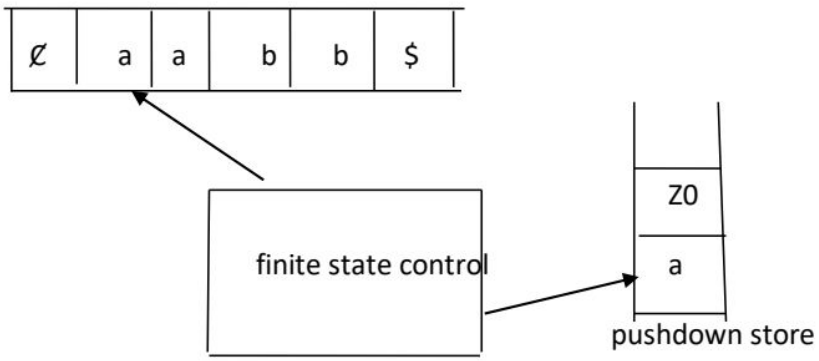
q0 it is the state that recognizes the beginning alphabets(i.e)the symbol a

q1 it recognizes the second half of the strings (i.e)the symbol b

q2 it is the state that accepts all the alphabets now we start the design of the pushdown automata.

Step1:

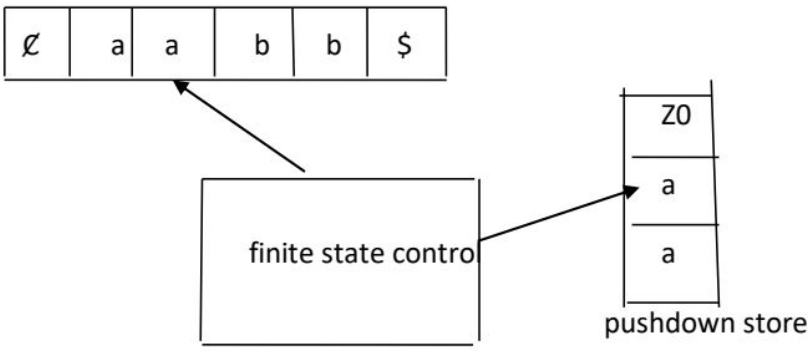
$\delta(q0,a,z0)$



in this case of the beginning alphabet a is pushed to the stack.and head pointer is incremented.

Step 2:

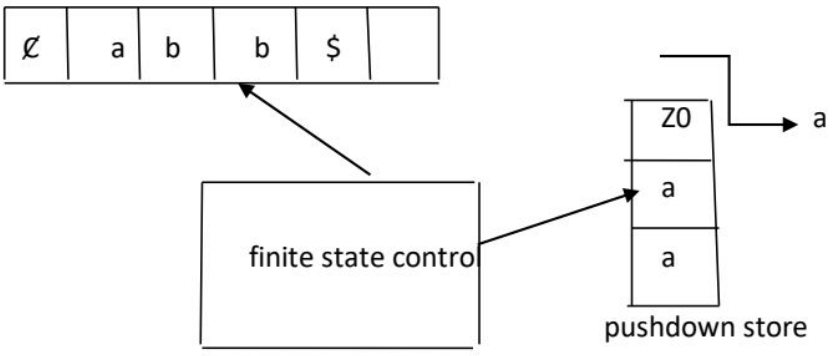
$\delta(q0,a,az0)$



the next alphabet is recognized by q0 and pushed into the stack.

Step 3:

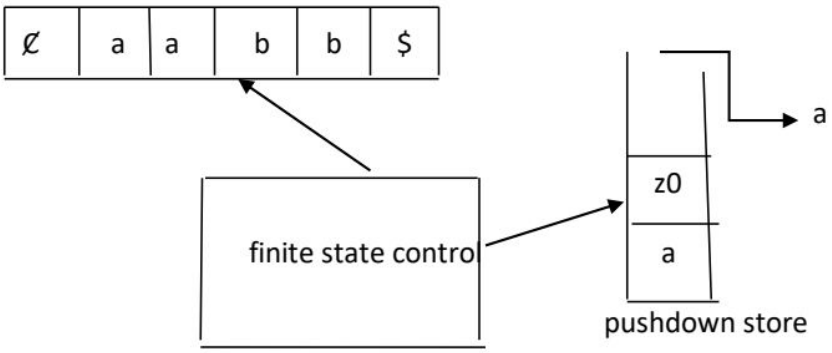
$\delta(q1,b,aaz0)$



in this case we find that we encounter a new alphabet b.upon this the alphabet a is poped from the stack.

Step 4:

$^{\delta}$(q1,b,azo)



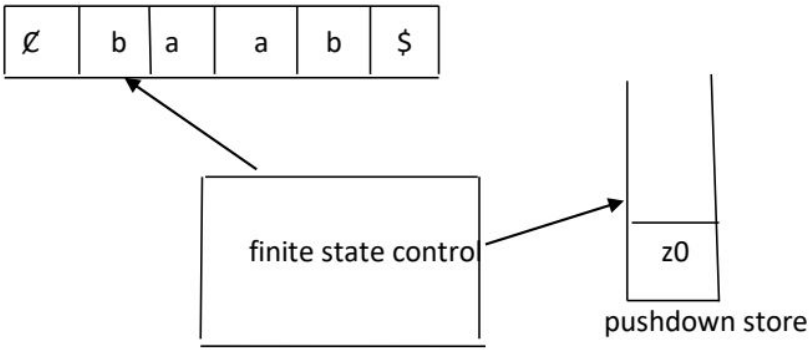
here again we encountered b,so another is poped from the stack.

this is the final state in which we again end up having the stack empty.because after all the a's in the given string are added to the stack.when the symbol b is encounter in the input string an a is removed from stack bases.this confirms the statement equal number of a's and equal number of b's in the string.

An example for palindrome checking

In this case we consider a string baab and design a push down automata to determine if the given string is palindrome or not.

The following figure shows basic PDA structure for the string.



CONTEXT FREE GRAMMERS:

P→P
P→0
P→1
P→0P0
P→1P1
A context free grammar for palindromes

There are four important components in a grammatical description of a language:

1. There is a finite set of symbols that form the strings of the language being defined. This set was {0, 1} in the palindrome example we just saw. We call this alphabet the terminals, or terminal symbols.

2. There is a finite set of variables, also called sometimes nonterminals or syntactic categories. Each variables represents a language; i.e., a set of strings. In our examples above, there was only one variable, P, which we used to represent the class of palindromes over alphabet {0, 1}.

3. One of the variables represents the language being defined; it is called the start symbol. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol. In our examples, P, the only variable, is the start symbol.

4. There is a finite set of productions or rules that represents the recursive definition of a language. Each production consists of:

(a) A variable that is being (partially) defined by the production. This variable is often called the head of the production.

(B) The production symbol→

(c) A string of zero or more terminals and variables .This string, called the body of the production, represents one way to form strings in the language of the production, represents one way to form strings in the language of the variable of the head. In so doing, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of that variable.

The four components just described form a context free grammar, or just grammar, or CFG. We shall represent a CFG G by its four components, that is, G= (V, T, P, S), where V is the set of variables the terminals the set of productions, and S the start symbol.

TOP DOWN PARSING:

A parser for grammer g is a program that takes as input as string w and produces a output either a parse tree for w .the method or the processes involved in converting a input string to a parse tree.

There are two types of parsers.

a)top down parser

b)bottom up parser

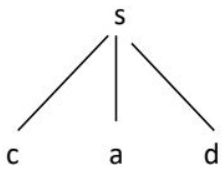
DIFFERENCE B\W TOPDOWN PARSING AND BOTTOM UP PARSING:

Bottom up parsers start with the bottom(leaves) to the top(root),while top down parsers start with the root and work down to the leaves.in both thecases the input to the parser is being scanned from left to right one symbol at a time.

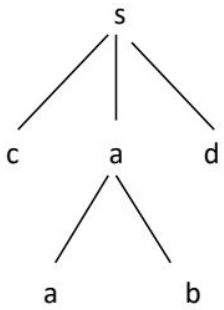
Top down parsing can be viewedas an attempt to find a left most derivation for an input string.equivalently,it can be viewed as attempting to construct a parse tree for the input starting from the root and creating the node of the power tree in preorder.

For eg.s->cAd A->ab/a and the input w=cad.

To construct a parse tree for this sentence top-down,we initially create a tree consisting of a single node labeled s. an input pointer points in c,the first symbol to c, the first symbol of w.we hen use first production for"s"to expand the tree and obtain



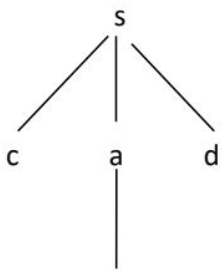
the left most leaf, labeled c, matches the first symbol of w,so we have advance the input to a,the second symbol, of w and expand the first alternate for A to obtain the tree.



using (A->ab)

we now consider d,the third input symbol and next leaf, labeled b,since b does not match d, we report failure and go back to A to see whether there is another alternate for A that we have not tried but which which produce the match.

In going back to A we must reset the input pointer to position q,the position it had when we first came to A.we now try the second alternate for A to obtain the tree.



d

the leaf matches the second symbol of w and the leaf d matches the third symbol. Since we have now produced a parse tree for w.We halt and announce successful completion of parsing.

An easy way to implement such a parser is to create a procedure for each non terminal

```
Procedure s()
Begin
If input symbol='c'then
Begin
ADVANCE()
If A ()then
Input symbol='d' then
Begin ADVANCE();return true end
End
Return false
End
Procedure A()
Begin
Isave=input-pointer;
If input symbol='a'then
Begin
ADVANCE();
If  input symbol='a'then
Begin
ADVANCE();
If input symbol='b'then begin ADVANCE();
Return true end;
End
Input-pointer-isave;
/*failure to find ab*\
If input –symbol='a'then
Begin ADVANCE();
Return true end
Else return false
End
```

The procedure ADVANCE() moves the inpit pointerto the next input symbol.”input symbol” is the one currently pointed to by the input pointer.procedure return true or false depending on whether or not they have found the non terminal.There are several difficulty with top down parsing.the first concerns left recursion.

A grammer G is said to be left recursive if it has a non terminal.a such that there is a derivation A->A* for some *.so we should eliminate left recursion.

ELIMNATION OF LEFT RECURSION:

If we have the left recursive pair of production,A->A*/β where β does not begin with on A,we can eliminate the left recursion by productions with

A->βA'

A'->*A'/ε

For eg: consider the grammer

E->E+T/T

T->T*F/F

F->(E)/id

Terminating the immediate left recursion(A->A*)

E->TE'

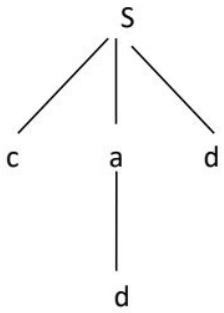
E'->+TE'/ε

T->FT'

T'->*FT'/ε

F->(E)/id

The second problem with top down back trcking parsers is the order in which alternatives are tried can affect the language accepted .for eg,if we used ’a’ and then ab as the order of the alternates for A.that is with parse trees and ca already matched the failure of the next input symbol ’b’ to match



would imply that the alternate cAd for s was wrong ,leading to rejection of cabd.

For eg:

The top down parser generate the parse tree for the string ‘abab’using grammer

S->AB/ε

A->a

B->BS

B->b

S->AB

->a B (A->a)

->a B S (B->BS)

->a b S (B->b)

->abAB (S->AB)

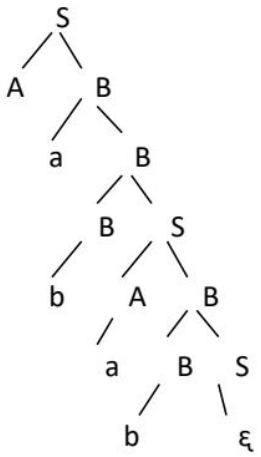
->abaB (A->a)

->abaBS (B->BS)

->ababs (B->b)

->abab (s->ε)

parse tree(root)



BOTTOM UP PARSING:-

Bottom-up parses build parse trees from the bottom (leaves) to the top (root). The bottom up parsing method we discuss is called the “shift reduce” parsing because it consists of shifting input symbols onto a stack until the right side of a production appears on top of the stack. The right side may then be replaced by (reduced to) the symbol on the left of the production, and the process repeated.

Shift-Reduce parsing:-

In this section we discuss a bottom-up parsing called shift-reduce parsing. This parsing method is bottom-up parsing because it attempts to construct a parse tree for an input string beginning at the leaves(the bottom) and working up towards the root(the top). We can think of this process as one of “reducing” a string ‘w’ to the start symbol of a grammar. In each step a string matching the right side of a production is replaced by the symbol on the left.

For example consider the grammar given below:

S → aAcBc

A → Ab/b

B → d

and the string to be derived is abbcde. We want to reduce this string S. we scan abbcde looking for substrings that match the right side of some production. The substrings b and d qualify. Let us choose the leftmost b replace it by A, the left side of the production A→b. we obtain the string aAbcde. We now find that Ab, b and d each match the right side of some production.

Suppose this time we choose to replace the substrings Ab by A, the left side of the production A→Ab. We obtain aAcde. Then replacing d by B, the left side of the production B ad, we obtain aAdBc. We can now replace this entire string by S.

Each representation of the right side of a production by the left side in the process is called a reduction. Thus, by a sequence of four reductions we were able to reduce abbcde to S. these reductions. In fact, traced out a rightmost derivation in reverse.

Informally, a substring which is the right side of production such that replacement of that string by the production, process of bottom up parsing may be viewed as of finding and reducing handles.

We must not be misled by the simplicity of this example. In many cases the leftmost substring B which matches the right side of the function.

HANDLES:-

A handle of a right-sentential form γ is a production A→β a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a right-most

derivation of γ. That is, if S → $\alpha\beta$w, then A→β in the position following α is a handle of $\alpha\beta$w. the string w to the right of the handle contains only terminal symbols.

In the above example, abbcde is a right-sentential form whose handle is A→b at position 2 likewise, aAbcde is a right-sentential form whose handle is A→Ab at position 2.

Sometimes we shall say,"The substring β is a handle of $\alpha\beta$w", if the position of B and the production A→β. We have in mind are clear. If a grammar is ambiguous, then every right-sentential form, so the grammar has exactly one handle.

The below diagram portrays the handle β in the parse tree of the right-sentential form $\alpha\beta$w the handle represents the leftmost complete sub tree consisting of the node with its children in the tree.

For example consider the grammar:-

E→E+E ; E→(E)

E→E*E ; E→id

And consider the rightmost derivation

E→E+E

→E+E*E

→E+E*id3

→E+id2*id3

→id1+id2*id3

For example id1 is a handle of the right sentential form id1+id2+id3 because id is the right most side of the production E→id, and replacing id1 by E produces the previous right-sentential form E+id2*id3. Note that the string appearing to the right of a handle contains only terminals symbols.

HANDLE PRUNING:-

A rightmost derivation in reverse, often called a canonical reduction sequence, is obtained by “handle pruning”. That is we start with a string of terminals ‘w’ which we wish to parse. If ‘w’ is a sentence of the grammar at hand, then w=γn is the nth right-sentential form of some as yet unknown rightmost derivation.

S = γ0 --> γ1 --> γ---> …..---> γn-1 ---> γn= w

To reconstruct this derivation, in reverse order. We locate the handle βn in γn and replace βn by the left side of some derivations. An→Bn to obtain the (n-1)st right-sentential form γn-1 and then repeat this process that is we locate the handle βn-1 in γn-1 and reduce this handle to obtain the right sentential form γn-2 if by considering this process we produce a right sentential form considering only of the start symbol(S).

Consider the example given below :

E→E+E

E→E*E

E→(E)

E→id

Input string is id1+id2*id3

Right-sentential form	Handle	Reducing production
id1+id2*id3	id1	E→id
E+id2*id3	id2	E→id
E+E*id3	id3	E→id

We can observe that the sequence of right-sentential forms in this example is just the reverse of the sequence in the right-most derivation.

STACK IMPLEMENTATION OF SHIFT REDUCE PARSING:-

To implement a shift-reduce parser is to use a stack and a input buffer. We shall use $ to mark the bottom of the stack and the right end of the input.

Stack	input
$ S	$

In this configuration the parser halts and announces successful completion.

Consider the given example:

Stack	Input buffer	Action
$	id1+id2*id3$	shift
$id1	+id2*id3$	reduce id1
$E	+id2*id3$	shift
$E+	id2*id3$	shift
$E+id2	*id3$	reduce id2
$E+E	*id3$	shift

y four

- Shift
- Reduce
- Accept
- Error

a)In a shift action, the next input symbol is shifted to the top of the stack.

b)In a reduce action, the parser knows the right end of the handle is at the top of the stack. It must locate the left end of the handle within the stack and decide with what non-terminal to replace the handle.

c)In a accept action, the parser announces successful completion of parsing.

d)In a error action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

APPLICATION OF PUMPING LEMA:

APPLICATION:

1. 1.check whether the given language is context free language or not.
2. 2.to check whether the string is finite or infinite.
3. Pumping level for cfl's states that ther are always two short substrings close together that can be repeated, both the same number of time as often we like.

STATEMENT:

Let "l" be any CFL (context free language).then there is a constant n,depending only on L,such that "z" is in l and |z|>=n,then we may write z=uvwxy such that,

a)|vx|>=1

b)|vwx|<=n and

for all i>0,uv^iwx^iy is in l.

TO PROVE:

To check whether uv^iwx^iy is in context free language "l" or not

PROOF:

Let G be a Chomsky normal form grammer generating L(G).observe that if Z is in L(G) and Z is long,then any parse tree for x must contain a long path.more precisely we can show by induction on "i" that if the parse tree of a word generated by a Chomsky normal form grammer has no path of length greater than i,then the word is of length no greater than 2^i-1.

BASIS:

For the path i=1,the derivation tree is of the form,

here n<=2^i-1 where n=1,i=1

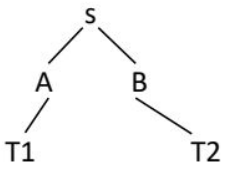
l<=2^i-1

l<=1

hence basis is proved.

INDUCTION PART:

For this consider i>1 .let the roots are



If there are no path greater than I,the words is of length no greater than 2^i-1.if there are no greater than 2^i-1.if there are no paths greater than 2^i-1,the word is of length no greater than 2^i-2 or fewer symbols.

For proof was the Chomsky normal form grammer,

G=({A,B,C},{a,b},A->BC,C->BA,A->a,B->b},A)

Where,

V={A,B,C} T={a,b}

P={A->BC,B->BA,C->BA,B->b}

S={A}

Consider a string Z=bbbaba in this grammer.the derivation and derivation tree is as follows.

A->BC

->BAC (replace B->BA)

->b AC (replace B->b)

->b BCC (replace A->BC)

->bb CC (replace B->b)

->bbBAC (replace C->BA)

->bbbAC (replace B->b)

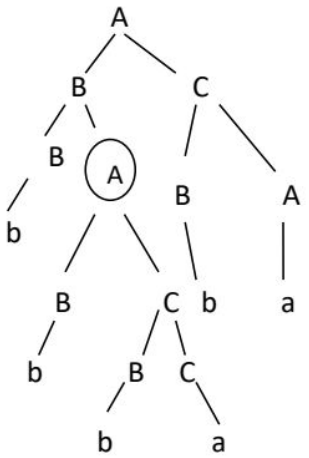
->bbbaC (replace A->C)

->bbbaBA (replace C->BA)

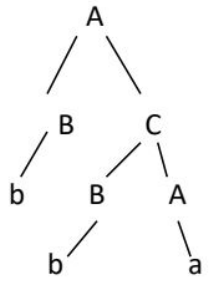
->bbbabA (replace B->b)

->bbbaba (replace A->a)

DERIVATION TREE:



a)subtree with parent vertex(a)v1:



subtree with parent vertex(a)v2:

in the parse tree,let p be a path that is long or longer than any path in the tree.then there must be a two vertices v1 and v2 on the path satisfying the following conditions,

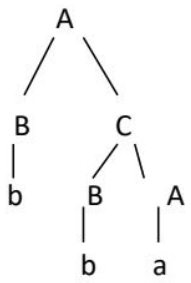
a)the vertices v1 and v2 both have same label way A.

b)vertex v1 is closer to the root than vertex v2.

c)the position of the path v1 to the leaf is of length almost k+1.

To use that V1 and V2 can be always found ,just proceed up path p from the leaf,keeping track of the labels encountered.of the first k+2 vertices only the leaf has a terminal label.

Consider a substring “bba” from the string bbbaba containing the path length i=3



path i=3

thus,

A=>bba=w

Since path length i=3

|w|<=2^i-1 |w|=3

3<=2^3-1

3<=2^2

3<=4

Which holds the condition |w|<=2^i-1 for pumping lemma for cfl.

CLOSURE PROPERTIES OF CFL:

The closure properties of CFL are

1.CFL's are closed under union, concatenation and Kleene closure.

2.CFL's are closed under homomorphism

3.CFL's are closed under substitutions

4.If L is a CFL and R is a regular language then, L/\R is a CFL.

5.CFL's are not closed under intersection

6.CFL's are closed under complementation.

Theorem 6.1 Context-free languages are closed under union, concatenation and kleene closure.

Proof : let L1 and L2 be CFL's generated by the CFG's

G1={V1,T1,P1,S1) and G2={V2,T2,P2,S2}

Respectively. since we may rename variables at will without changing the language generated, we assume that V1 and V2 are disjoint. Assume also that S2,S4 and S5 are not in V1 or V2.

For L1U L2 construct grammar G3=(V1 U V2 U {S3},T1 U T2,P3,S3), where P3 is P1 U P2 plus the production S3→S1 | S2. If w is in L1, then the derivation S3=> S1=>w is a derivation in G3,as every production of G1 is a production of G3. similarly, every word in L2 has a derivation in G3 beginning with S3=>S2. Thus L1UL2 L(G3). For the converse, let w be in L(G3). Then the derivation S3=>w begins with either S3=>S1=>w or S3=>S2=>w. In the former case, as V1 and V2 are disjoint, only symbols of G1 may appear in the derivations S1=>w. As the only productions of P3 that involve only symbols of G1 are those from P1, we conclude that only productions of P1 are used in the derivations S1=>w. Thus S1=>w, and w is in L1. Analogously, if the derivation starts S3=>S2, we may conclude w is in L2. Hence L(G2) L1UL2, so L(G3)= l1Ul2 as desired.

For concatenation, let G4=(V1UV2U{S4},T1,P4,S4), where P4 is P1UP2 plus the production S4→S1 S2. A proof that L(G4)=L(G1)L(G2) is similar to the proof for union and is omitted.

For closure, let G2=(V1UV2U{S4}, T1,P2,S3), where P3 is P1 plus the productions S3→S1 S5|Є.We again leave the proof that L(G5)=L(G1)* to the reader.

SUBSTITUTION AND HOMOMORPHISMS

Theorem 6.2 The context-free languages are closed under substitution.

Proof : Let L be a CFL, L ∑*, and for each a in ∑ let La be a CFL. Let L be L(G) and for each in ∑ let La be L(Ga). Without loss of generality assume that the variables of G and the Ga'S are disjoint. Construct a grammar G' as follows the variables of G' are all the variables of G and the Ga's, the terminals of G are the terminals of the Ga's . the start symbol of G' is the start symbol of G. The productions of G' are all the productions of the Ga's together with those productions formed by taking a production A→α of G and substituting Sa, the start symbol of Gn, for each instance of an a in Σ appearing in α.

Example 6.4 let L be the set of words with an equal number of a's and b's La={0 1 |n≥1} and L={ww r | w is in (0+2)*}. For G we may choose

S→asbs|bsas|ϵ

For Ga take Sa→0Sa1|01

For Gb take Sb→0Sb0|2Sb2|ϵ

If f is the substitution f(a)=La and f(b)=Lb, then f(L) is generated by the grammar

S→Sa SSbS | Sb SSa S|ϵ

Sa→0Sa1 | 01

Sb→0Sb0 | 2Sb2|ϵ

One should observe that since {a,b},{ab}, and CFL's the closure pf CFL's under substitute implies closure under union, concatenation, and *. The union of La and Lb is simply the substitute of La and Lb into {a,b} and similarly La Lb and La* are tge substitute into{ab} and a*, respectively. Thus theorem6.1 could be presented as a corollary of theorem6.2.

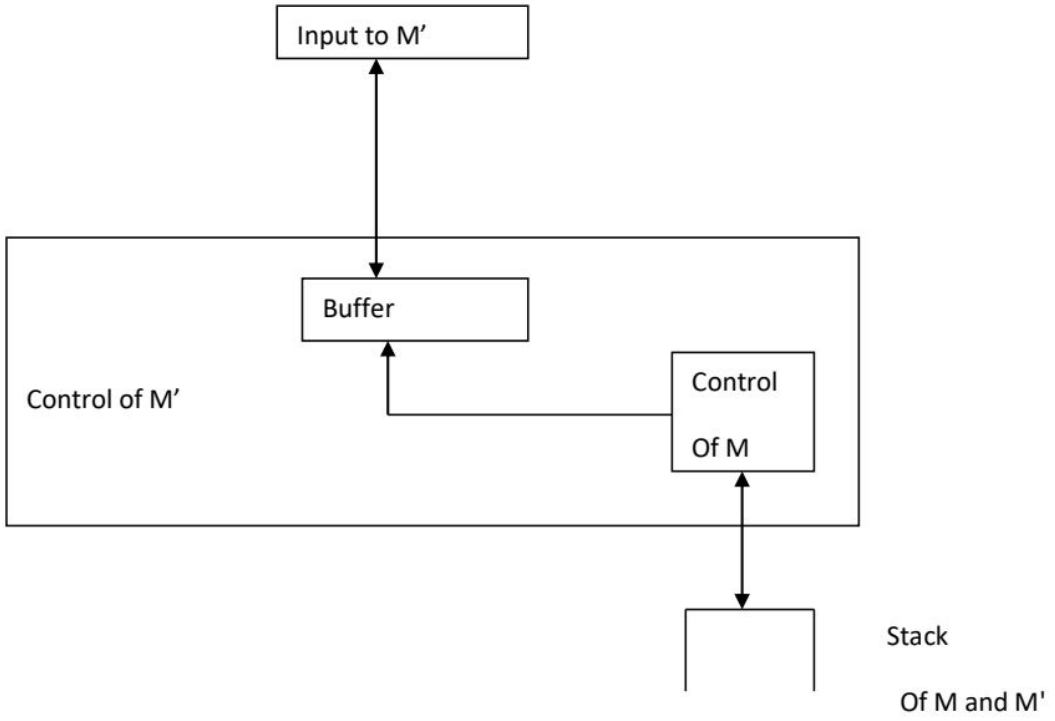
Since a homomorphism is a special type of substitution we state the following corollary.

Corollary The CFL's are closed under homomorphism.

Theorem 6.3 The context-free language are closed under inverse homomorphism .

Proof : As with regular sets, a machine-based proof for closure under inverse homomorphism is easiest to understand. Let h:∑→Δ be a homomorphism and L be a CFL. Let L=L(M), where M is the PDA (Q,Δ, ┌,δ,qo,Zo,F). In analogy with the finite-automaton construction of theorem 3.5, we construct PDA M' accepting h-1 (L) as follows. On input am M' generates the string h(a) and simulates M on h(a). if M' were finite automaton, all it could do on a string h(a) would be to change state, so M' could simulate such a composite move in one of its moves. However, in the PDA case, M could pop many symbols on a sting, or, since it is nondeterministic, make moves that push an arbitrary simulate M's moves on h(a) with one (or any finite number of) moves of its own.

What we do is give M' a buffer, in which it may store h(a). then M' may simulate any ϵ-moves of M it likes and consume the symbols of h(a) one at a time, as if they were M's input . As the buffer is a part of M's finite control, it cannot be allowed to grow arbitrarily long. We ensure that it does not, by permitting M' to read an input symbol only when the buffer is empty. Thus the buffer holds a suffix of h(a) for some a at all times. M' accepts its input w if the buffer is empty and M is in a final state. That is, M has accepted h(w). thus l(M")={w|h(w)is in L}, that is



Construction of a PDA accepting $h^{-1}(L)$

L(M')=h⁻1(L(M)).The arrangement is depicted in fig ;the formal construction follows.

Let M'=(Q',∑,Γ,δ',[q0,€],Zo,F×{€}),where Q' consists of pair [q,x] such that q is in Q and x is a (not necessarily proper)suffix of some h(a)for a in ∑ .δ' is defined as follows:

1] δ'([q,x],€,Y) contains all ([p,x],γ) such that δ(q,€,Y) contains(p,γ) .simulate €-moves of M independent of the buffer contents.

2] δ'([q,ax],€,Y) contains all([p,x],γ) such that δ(q,a,Y) contains(p,γ) . Simulate moves of M on input a in Δ, removing a from the front of the buffer.

3] δ'([q,€],a,Y) contains(q,h(a)],γ) for all a in∑ and Y in Γ . Load the buffer with h (a), reading a from M''s input ; the state of M remain unchanged.

To show that L(M')=h⁻1 (L(M))first observe that by one application of rule (3), followed by application of rules(1)and(2),if(q,h(a),α)*/M(p,€,β).

Theorem 6.4:The CFL'S are not closed under intersection.

Proof: we showed the language L1={a^ib^ic^i|i>=1 } was not a CFL. We claim that L2={ a^ib^ic^i|i>=1 and L3={ a^ib^ic^i|i>=1 and j>=1} are both CFL's. For exa,[;e. a {DA to recognize L2 stores the a's pm its stack and cancels them against b's, then accepts its input after seeing one or mors c's. Alternatively L2 is generated by the grammar

S->AB

A->aAb|ab

B->cB|c

Where A generates a a^ib^I and B generates c^i. A similar grammar

S->CD

C->aC|a

D->bDc|bc

Generates L3.

However, L2^L3 =L1.

If the CFL's were closed under intersection. L1 would thus be a CFL.

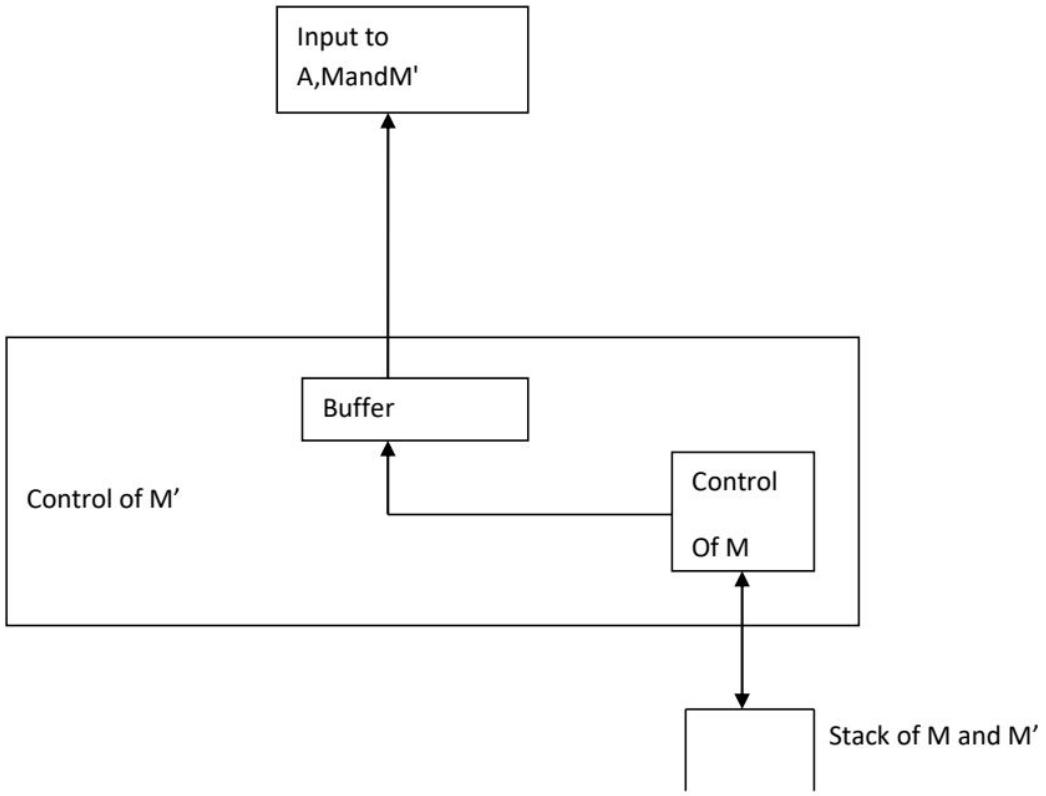
Corollary : The CFL's are not closed under complementation.

Proof: we know the CFL's are closed under union. If they were closed under complementation, they would, by DeMorgan's law, L1^L2=L1UL2, be closed under intersection.

Theorem 6.5: If L is a CFL and R is a regular set, then L^R is a CFL.

Proof: Let L be L(M) for PDA M=(Qm, $\sum, \Gamma, \delta$,[p0,qo],z0,fa×fm) , and let R be L(A)for DFA A=(qa,$\sum, \delta$a,p0,fa). We construct a PDA M' for L R by "running M and A in parallel, M' simulates moves of M on input without changing the state of A. when M' makes a move on input symbol a, M simulates that move and also simulates A's change of state on input a. M' accepts if and only if both A and M accept. Formally, let

M'=(Qa×Qm,$\sum, \Gamma, \delta$,[p0,q0],z0,Fa×Fm)



DECISION ALGORITHM FOR CFL:

STATEMENT:

Decision algorithm for cfl is mainly used to check whether the given language is finite,empty or non finite(infinite).

TO PROVE:

To check whether the language is finite,infinite or empty.

ASSUMPTIONS:

1.the grammar should be in Chomsky normal form(cnf) without ε production.

2.if 's' is the start symbol and 'r' is the rank of the symbol, then the string length will be greater then or equal to 2^r.

If 'A' is the reverse vertex associated with 's' is the root node ,then the string length will not be greater than 2^r-1.

If 'B' is the reverse vertex associated with 's' is the root node,then the string length will not be greater

than 2^r-2.

BASIS PART: (r=0)

Consider a production A->a,it is a directed acyclic graph denoted by 

.by the assumption a non terminal with rank 'r' cannot generate a string of length greater than 2^r.

Here, the rank of A is '0'.i.e(r=0)

By the condition l<=2^r,substitute ‘r’ value,

We get l<=2^0

L<=1

Hence it is proved by considering the form A->a,we may derive only a string of length ‘1’,which is finite.

INDUCTION PART:

If we use a production of the form A->a ,we may derive only a string of length 1.if we begin with A->BC, then as B and C are of rank r-1 or less,by the inductive hypothesis,they derive only strings of length 2^r-1 or less.thus BC cannot derive a string of length greater than 2^r.

Since s is of finite rank r0,and in fact,is of rank no greater than the number of variables,s derives strings oflength no greater than the number of variables,s derives strings of length no greater than 2 r^0.thus the language is finite.

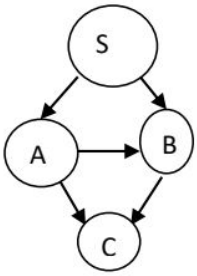
Consider an example,

S->AB

A->BC

B->CC

C->a



r (s)=3

r(A)=2

r(B)=1

r(c)=0

rank of starting symbol ‘s’ (r(s)=3).

Rank ‘A’ ,r(A) depends on root node ‘s’,it should be less than’s’ and rank of’b’,r(B) depends on both S and A ,it should be less than both S and A .

From, the figure check the condition l<=2^r

r (s)=3, l<=2^3

l<=8

r(A)=2, l<=2^2

l<=4

r(B) =1, l<=2^1

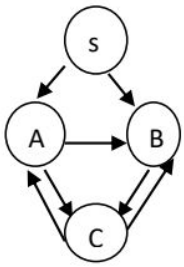
l<=2

r(c) =0, l<=2^0

l<=1

hence the finite length for each string is derived ..hence it proved.

Now,consider the directed cyclic graph,



For the figure, the rank cannot be specified. hence for the cyclic graph the language is infinite.

For cyclic graph the language is finite. hence the statement is proved.

UNIT IV

TURING MACHINE

Contents

UNIT IV

Turing machines: Turing machines(TM) – computable languages and functions – Turing machine constructions – storage in finite control - variations of TMs – recursive and recursive enumerable languages.

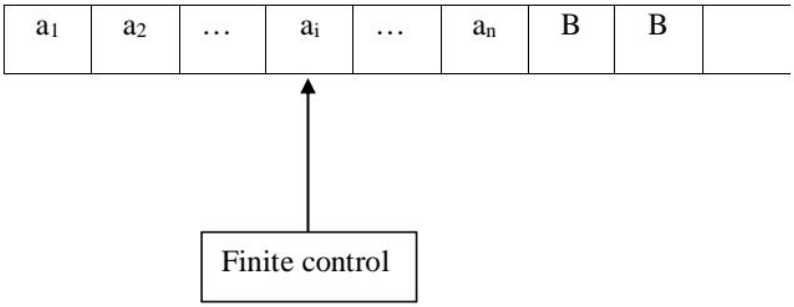
1. TURING MACHINES

It is a more powerful model than many models. It was introduced by Alan Turing in 1936. It can do everything that a real computer can do.

Configuration of the TM

It has a finite control, an input tape that is divided into cells, each of which is capable of holding one symbol. Associated with the tape head is a read write head [tape head] that can

- Travel right or left on the tape
- Scan one cell of the tape at a time.
- Read and write a single symbol on each move.



The TM's temporary storage is on the tape. Whatever input and output is necessary will be done on the tape. The tape has a leftmost cell but is infinite to its right. The tape head is initially set to scan the content of the leftmost cell.

Difference between FA and TMs

1. A TM can both write on the tape and read from it.
2. The read write head can move both left and to the right.
3. The tape is infinite.
4. The special states for rejecting and accepting take immediately effect.

Moves of the TM

Depending on the state of the infinite control the symbol on the cell scanned by the tape head, TM changes state, prints a new symbol on the tape cell scanned and moves left or right or stationary.

Mathematical definition of a TM

The basic model of a TM is defined as

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

Where,

Q – a finite set of states.

Γ – a finite set of allowable tape symbols.

B – a symbol of Γ, is the blank

∑ - a subset of Γ not including β is the set of input symbols.

δ – the next move function, if) transition function), from Q x Γ → Q x Γ x {L, R}

[δ may not be defined for some values]

q_0 – it is in Q, it is the start state

F - (Subset of Q) is the set of accepting states.

Varieties of Turing Machine (Types or modified versions of TM)

The basic model of the Turing Machine is equivalent to many other modified versions. They are

1. Non deterministic Turing machine
2. Two way infinite tape.
3. Multi tape Turing machine
4. Offline Turing machine
5. Multi head Turing machine
6. Multidimensional Turing machine
7. Restricted Turing machine

1. Non Deterministic Turing machine

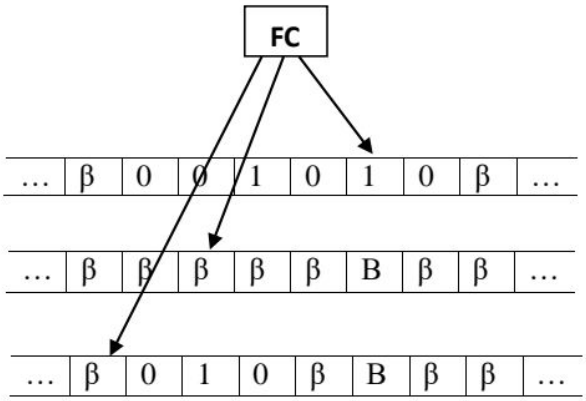
- It has a finite control and a single one way infinite tape.
- For the given state and the tape symbol scanned by the tape head the TM has a finite number of possibilities for its next move.
- Here the computation is like a tree whose branches correspond to different possibility.
- If some branch of the computation leads to the accept state the machine accepts the input.

2. Two way Infinite Turing machine

- This is similar to the basic model.
- But here the tape is infinite to the left as well as to the right.
- Infinite number of blanks will be in both sides.
- The initial ID is q_0w

3. Multi tape Turing machines

- It has a finite control with many tapes say 'k' tape heads.
- Each tape is too way infinite.
- Initially the input will be on the first tape and the other tapes are kept blank.
- Depending on the states of the PC and the symbol scanned by each of the tape heads the TM can
    i. Change the state
    ii. Print a new symbol on each of cells scanned by the tape heads.
    iii. Move each tape head independently [some to right, some to left and some other stationary]



4. Offline Turing Machine

- This is a multi tape TM whose input tape is read only.
- The input is surrounded by some end markers.
- The TM is allowed to move the input head in between the end markers.

5. Multi head Turing machine

- It has 'k' heads, k>1.
- Depending on the state and the symbol scanned by each head, the heads move independently.

6. Multidimensional Turing machine

- It has a finite control and a tape consisting of k-dimensional arrangement of the cells infinite in all directions for some 'k'.
- Initially the input string is along one axis and the tape head is input string.
- Depending on the state and the symbol scanned the TM
  - **i.** Changes state
  - **ii.** Prints a new symbol.
  - **iii.** Moves the tape head in one of the2k direction positively or negatively along the k-axes.

7. Restricted Turing machine

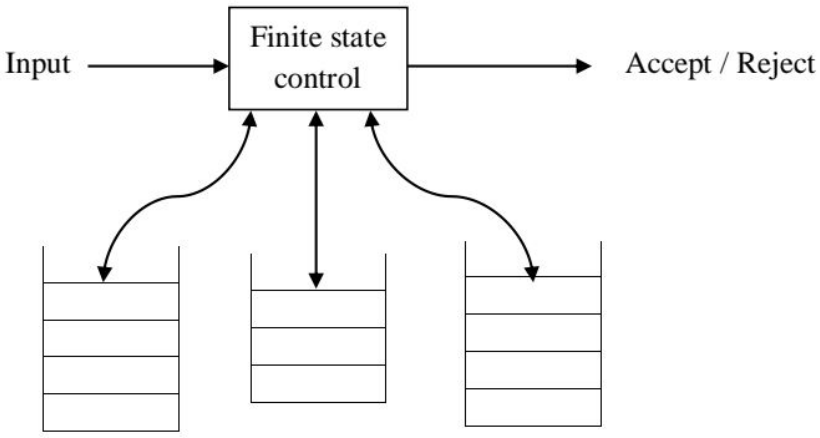
i. Counter machines
ii. Multi stack machines
iii. Turing machine with semi infinite tapes.

(i) Counter machine

- It has the same structure as the multistack machine.
- It has ability to store a finite number of the integers. (counters)
- It can able to add or subtract one from the counter but it cannot tell two different nonzero counts from each other.

(ii) Multistack machine

- A k stack machine is a deterministic PDA with k stacks.
- It has a Finite control which is in one of the state.
- It has a finite stack alphabet which is used for all stacks.
- A move of the multistack machine is based on
  - **a.** The state of the finite control.
  - **b.** The input symbol read.
  - **c.** The top stack symbol on each stack.



A multi stack machine with 3 stacks

In one move the multi stack machine can

(i) Change to a new state
(ii) Replace the top symbol of each with a string or more stack symbols.

(iii) Turing machine with semi infinite tapes

- Here the tape is semi infinite that is, there are no dells to the left of the initial position of the tape head.
- It can simulate one whether is infinite in both directions.
- It accepts any regular language.

Universal Turing machine [UTM]

It is a programmable computer. Since when it is given a program a description of another computer it makes itself as if it were that machine while processing the input.

2. COMPUTABLE LANGUAGES AND FUNCTION:

A language that is accepted by a Turing machine is said to be *recursively* enumerable (r.e). The term "enumerable" derives from the fact that is precisely these languages whose strings can be enumerated (listed) by a TM. "Recursively" is a mathematical term predating the computer, and its meaning is similar to what the computer scientist would cal "recursion". The class of r.e is very broad and properly includes the CFL's.

The class of r.e languages includes some languages for which we cannot mechanically determine membership. If L(M) is such a language, then any Turing machine recognizing L(M) must fail to halt on some input not in L(M). If w is in L(M), M eventually halts on input w. However, as long as M is still running on some input, we can never tell whether M will run forever.

It is convenient to single out a subclass of the r.e. sets called the recursive sets, which are those languages accepted by a least one TM that halts on all inputs (note that halting may or may not be preceded by acceptance).

The Turing Machine as a computer of integer functions:

In addition to being a language acceptor, the TM may be viewed as a computer of functions from integers to integers. The traditional approach is to represent integers in unary; the integer $i \geq 0$ is represented by6 the string 0^i. If a function has k arguments, $i_1, i_2, \ldots, i_k$, then these integers are initially placed on the tape separated by 1's, as $0^{i_1}10^{i_2}1\ldots 10^{i_k}$.

If the TM halts (whether or not in an accepting state) with a tape consisting of 0^m for some m, then we say that $f(i_1, i_2 \ldots, i_k) = m$, where f is the function of k arguments computed by this TM. Note that one TM may compute a function of one argument, a different function of

two arguments, and so on. Also note that if TM M computes function f of k arguments, then feed not have a value for all different k – tuples of integers $i_1,\ldots,i_k$.

If $f(i_1,\ldots,i_k)$ is defined for all $i_1, i_2,\ldots,i_k$, then we say f is a total recursive function. A function $f(i_1, i_2,\ldots,i_k)$ computed by a Turing Machine is called a partial recursive function. In a sense, the partial recursive functions are analogous to the r.e. languages, since they are computed by Turing Machine that may or may not halt on a given input. The total recursive functions correspond to the recursive languages, since they are computed by TM's that always halt. All common arithmetic functions on integers, such as multiplication, n!, $[\log_2 n]$ and 2^2 are total recursive functions.

3. TECHNIQUES FOR TURING MACHINE CONSTRUCTION

Designing Turing machines by writing out a complete set of states and a next – move function is a noticeably unrewarding task. In order to describe complicated Turing machine constructions we need some "higher – level" conceptual tools.

Storage in the finite control

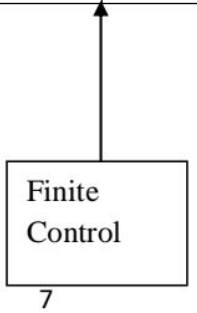
The finite control can be used to hold a finite amount of information. To do so, the state is written as a pair of elements, one exercising control and the other storing SYMBOL. It should be emphasized that this arrangement is for conceptual purposes only. No modifications in the definition of the Turing machine have been made.

In general, we can allow the finite control to have k components, all but one of which store information.

Multiple tracks

We can imagine that the tape of the Turing machine is divided into k tracks, for any finite k. This arrangement is shown in Fig. with k=3. The symbols on the tape are considered k - tuples, one component for each track.

¢	1	0	1	1	1	1	$	B	B	
B	B	B	B	1	0	1	B	B	B	…
B	1	0	0	1	0	1	B	B	B	



Checking off symbols

Checking off symbols is a useful trick for visualizing how a TM recognizes languages defined by repeated strings, such as

$\{ww \mid w \text{ in } \sum^*\}$, $\{wcy \mid w \text{ and } y \text{ in } \sum^*, w \neq y\}$ or $\{ww^R \mid w \text{ in } \sum^*\}$.

It also useful when lengths of substrings must be compared, such as in the languages

$\{a^ib^i \mid i \geq 1\}$ or $\{a^ib^ic^k \mid i \neq j \text{ or } j \neq k\}$

Shifting over

A Turing machine can make a space on its tape by shifting all non blank symbols a finite number of cells to the right. To do so, the tape head makes an excursion to the right, repeatedly storing the symbols read in its finite control and replacing them with symbols read from dells to the left. The Tm can then return to the vacated dells and print symbols of its choosing. If space is available, it can push blocks of symbols left in a similar manner.

Subroutines

As with programs, a "modular" or "top-down" design is facilitated if we use subroutines to define elementary processes. A Turing machine can simulate any type of subroutine found in programming languages, including recursive procedures and any of the known parameter-passing mechanisms.

The general idea is to write part of a TM program to serve as a subroutine; it will have a designed initial state and a designated return state which has no move and which will be used to effect a return to the calling routine. To design a TM that "calls" the subroutine, a new set of states for the subroutine is made, and a move from the return is effected by the move from the return state.

4. RECURSIVE & RECURSIVE ENUMERABLE LANGUAGES

1. Recursive enumerable language:

A language L is called a recursive enumerable language if

(i) There is a TM that accepts L and
(ii) Rejects L' or goes into an infinite loop

i.e. The TM can halt in the accepting state but need not halt in the non accepting state.

2. A language L is said to be recursive if

(i) There is a Turing machine that accepts L and
(ii) Rejects L'

i.e. Here the TM can halt (1) in the accepting states and (2) in the non accepting state.

A recursive language is recursive enumerable but not the converse.

i.e. A recursive language need not be recursive.

Properties of a recursive or recursive enumerable language:

Result 1:
If a language is recursive then it is recursive enumerable.

Result 2: Recursive languages are closed under complementation.

i.e. If L is recursive then its complement is recursive

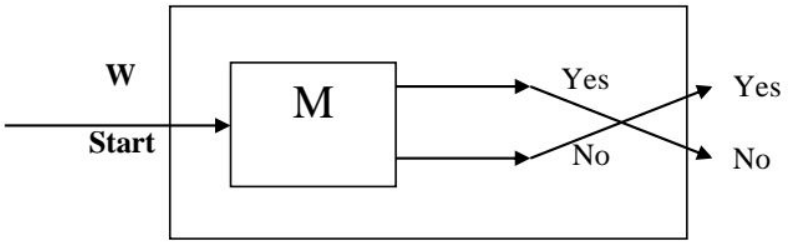
To prove: The complement of L say L' is recursive

Proof: Let L be a recursive language. Then there is a Turing Machine M that accepts L.

.

Construct a Turing machine M' from M such that : M enters a final state on the input W in L, then M' halts in a non accepting state. (Without accepting)

If M halts without accepting then M' accepts the input 'w' in L'. Hence M' is an algorithm and is given as below.



Here 'Yes' corresponds to 'accept' and 'No' corresponds to 'Reject'. The language of M' is L (M') and is the complement of L. Hence the proof.

Result 3: The union of two recursive language is recursive.

To prove: $L_1 U L_2$ is recursive.

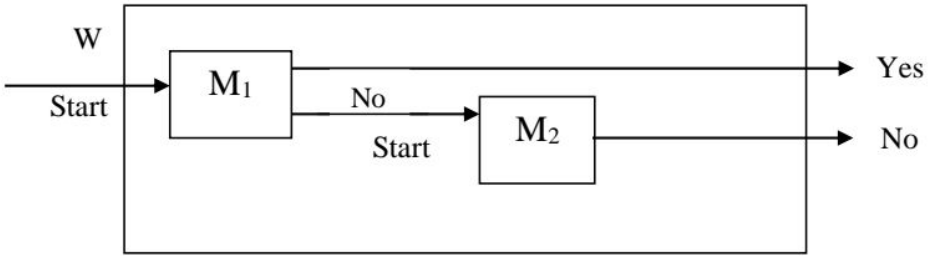
Proof: Let L_1 and L_2 two recursive languages. Then there exists Turing machine M_1 and M_2 such that M_1 and M_2 accept L_1 and L_2 respectively.

Construct a Turing machine M that accepts $L_1 U L_2$ as below:

- First simulate M on M_1.
- If M_1 accepts inputs w in $L_1 U L_2$ then M accepts.
- If mot, M simulates M_2 and accepts iff M_2 accepts.

Since M_1 and M_2 are algorithms (used to say Yes or No), M_1 and M_2 will not hang, they will halt (in accepting state or non accepting state). M will surely halt (It will not hang).

Hence M is an algorithm. M is given below:



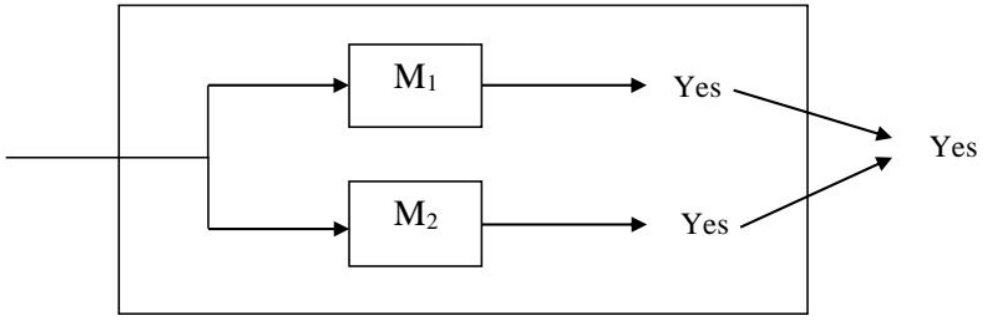
Now the language of M is $L_1 U L_2$. Hence the result.

Result 4: The union of two recursively enumerable languages enumerable.

Proof: Let L and L_2 be two recursively enumerable languages.

There are Turing machine M_1 and M_2 such that M_1 accepts L_1 and M_2 accepts L_2. Construct a Turing machine M that accepts $L_1 U L_2$ as below:

The above procedure will not work here. Since M_1 may not halt for some input W in L_1. Hence M is constructed as below:



Now the language of M is $L_1 U L_2$. Hence $L_1 U L_2$ is recursive enumerable.

Result 5: If a language L and its complement L' are both recursively enumerable, then L (and hence L' by **result 2**) is recursive.

To prove: L is recursive

Proof: Let a language L and its complement L' are recursively enumerable. Then there are Turing machine M_1 and M_2 that accepts L and L' respectively.

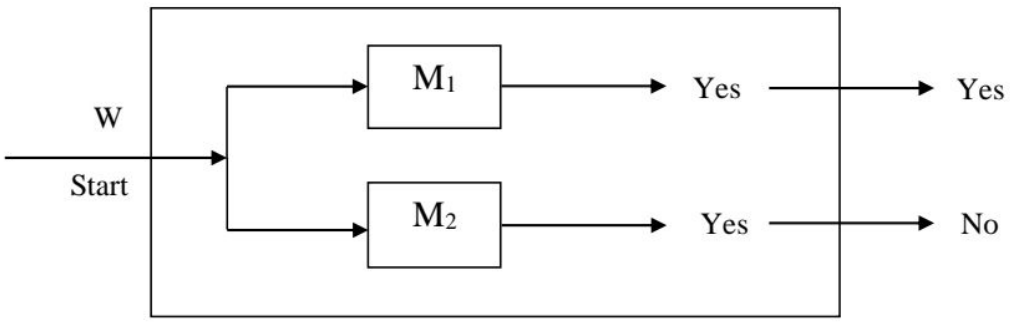
Construct a Turing machine M that accepts L and say either 'Yes' or 'No' as below: M is connected in such a way that M simulate M_1 and M_2 simultaneously.

- If M_1 accepts then M will accept it.
- If M_2 accepts then M will reject it.

Hence M will always say either 'yes' or 'no' and not both.

Hence M is an algorithm that accepts that accepts L. Hence L is recursive. {Proved Ans (1) } Hence L' is recursive. {Ans (2)}

M is shown below:



Hence we have If L and L' are a pair of complementary languages then either

i. Both L and L' are recursive (or)
ii. Neither L nor L' are recursively enumerable.

UNIT-5
INTRODUCTION TO COMPUTATIONAL COMPLEXITY

GROWTH RATE OF FUNCTIONS:

STATEMENT:

When we have two algorithms for the same problem, we may require a comparison between the running times of these two algorithms with this in mind, we study the growth rate of functions defined on the set of natural numbers.

In this section, N describes the set of natural numbers.

Let f, g : N -> R^+(R^+ being the set of all positive real numbers). We say that f (n) = O (g (n)), if there exists positive integers C and N0 such that,

f (n) <= C g (n) for all n >= N_0.

In this case we say "f" is the order of g. (Or f is "big oh" of g) .f (n) = O (g (n)) is not an equation. It expresses a relation between two functions f and g.

For example:

If f (n) = $4n^3 + 5n^2 + 7n + 3$, we have to prove that f (n) = O (n^3)

Solution:

In order to prove that f (n) = O (n^3),

Take c=5 and N_0 =10 then

f (n) = $4n^3 + 5n^2 + 7n + 3 <= 5n^3$

When n=10,

f (n) = $5n^2 + 7n + 3 < 5(10^3)$

$= 573 < 5(10^3)$

For n>10,

f (n) = $5n^2 + 7n + 3 < 5(n^3)$

---

here, n=11

$5(11)^2 + 7(11) + 3 < (11^3)$

Then,

f (n) = O (n^3).

Now we can prove that a polynomial algorithm complexity depends on the higher order power of the polynomial.

Statement: If P (n) = $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$, is a polynomial of degree K over Z and a_k an integer and positive as $a_k >= 1$,

P (n) = $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$\div n^k$

$$\frac{p(n)}{n^k} = \frac{a_k n^k}{n^k} + \frac{a_{k-1} n^{k-1}}{n^k} + \cdots + \frac{a_1 n^1}{n^k} + \frac{a_0}{n^k}$$

Since $a_{k \geq 1,}$

$$a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} > 0$$ For all n >= N_0.

Also

$$\frac{p(n)}{n^k} = a_k + \left(\frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k}\right)$$

$\frac{p(n)}{n^k} \leq a_k+1$

(Since as $a_{k-1}, a_{k-2}, \ldots, a_1, a_0$ are K fixed integers , choose N_0 such that for all n>- N_0 each of the number).

$\frac{|a_{k-1}|}{n}, \frac{|a_{k-2}|}{n^2}, \ldots, \frac{|a_1|}{n^{k-1}}, \frac{|a_0|}{n^k}$ is less than 1/k

Hence,

Hence,

$$\left|\frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \frac{a_0}{n^k}\right| < 1$$

$$\frac{p(n)}{n^k} \leq a_k$$

$P(n) \leq a_{k+1} + n^k$ where $c = a_{k+1}$.

Hence $P(n) = O(n^k)$.

Exponential function:-

An exponential function is a function q:N→N defined by q (n) = a^n for some fixed a>1.

When n grows each of n, n^2,2^n increases. But the comparison of these functions for specific values of n will indicate the vast difference between the growth rate of the functions.

Growth rate of polynomial and exponential functions:

n	**f (n) = n^2**	**g(n)=n^2+3n+9**	**q (n) = 2^n**
1	1	13	2
5	25	49	32
10	100	139	1024
50	2500	2659	$1.13X10^{15}$
100	10000	10309	$1.27X10^{30}$
1000	1000000	1003009	$1.07X10^{301}$

From the table, it is easy to say that the function q(n) grows at a very fast rate when compared to f (n) or g (n). in particular the axponential function grows at a very fast rate when compared to f (n) or g (n) . In particular the exponential function grows at a very fast rate when compared to any polynomial of large degree.

If f and g are two functions and f= O (g) but g ≠ O (f) we say that the growth rate of g is greater rate of f.

The growth rate of any exponential function is greater than that of any polynomials

Let,

P(n) = $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

And q (n)= a^n for some a>1.

As the growth rate of any polynomial is determined by its term with the highest power, it is enough to prove that:

$N^k = O(a^n)$ and $a^n \neq O(n^k)$

By L' hospital rule , $\frac{logn}{n}$ tends to 0 as N → ∞

If

$$Z(n) = e^{k\left(\frac{\log n}{n}\right)}$$

Then,

$$(Z(n))^n = e^{\left(k\frac{logn}{n}\right)^n}$$

$$= e^{k \log n}$$

$$= e^{\log n^k}$$

$$= n^k,$$

As n gets large , $k\left(\frac{\log n}{n}\right)$ tends to 0 and hence Z(n) tends to 0.

So we can choose N_0 such that Z(n) <= a for all n >= N_0. Hence $n^k = (z(n))^n <= a^b$

Proving,

$N^k = O(a^n)$

To prove that $a^n \neq O(n^k)$, it is enough to show that a^n/n^k is unbounded for large n. but we have proved that

$N^k <= a^n$ (or) $\frac{a^n}{n^{k+1}} \geq 1$.

Multiply by n, $n \times \frac{a^n}{n^{k+1}} \geq n$, which means $\frac{a^n}{n^k}$ is unbounded for large values of n.

PRIMITIVE RECURSIVE FUNCTIONS:

We consider automata as the accepting devices initially but automata can act as the computing machines. The problem of finding out whether a given problem is 'solvable' by automata reduces to the evaluation of functions on the set of natural numbers or a given alphabet by mechanical means.

PARTIAL RECURSIVE FUNCTION:

A partial function "f" from X to Y is a rule which assigns to every element of X atmost one element of Y.
For e.g. If R denotes the set of all real numbers .The rule "f" from R to itself is given by $f(r) = +\sqrt{r}$ is a partial function since f(r) is not defined as a real number when r is negative.

TOTAL RECURSIVE FUNCTION:

A total function from X and Y is a rule which assigns to every element of X a unique element of Y.

We consider total functions from X^k to X where X={0,1,2,3,….} or X={a,b}*. A partial or total function from X^k to X is also called a function of "k" variables and denoted by $f(x_1,x_2,x_3\ldots..x_k)$.for example $f(x_1,x_2)=2x_1+x_2$ is a function of two variables.

a) If f(1,2)=4,then 1 and 2 are called arguments and 4 is called the value.
b) If $g(w_1,w_2)=w_1w_2$ is a function of two variables.
   g (ab,aa) = abaa,aa are called arguments and abaa is a value of the function.
   We now construct the primitive recursive functions over N and ∑.

Initial Function over N:

The initial functions over N are,

a) Zero function: Zero function defined by z(x)=0.
b) Successor function "s" defined by s(x)=x+1.
c) Projection function U_i^n defined by $U_i^n(x_1,x_2,\ldots x_n)=x_i$

For eg:

Z(7)=0 (zero function)
S(4)=4+1 (successopr function)
=5

i. $U_2^3(2,4,7)=4$ (projection function)
ii. $U_1^3(2,4,7)=2$
iii. $U_3^3(2,4,7)=7$

Initial Function over ∑:

The initial functions over ∑ are given by
nil (abab)=&
cons a(abab)=aabab

cons b(abab)=babab

in general form ,the initial functions over $\sum$ are,

nil (x)=&
cons a(x)=ax
cons b(x)=bx

Composition of partial functions:

If $f_1,f_2,f_3\ldots.f_k$ are partial functions of n varialbles and g is a partial functions of k variables , then the composition of g with $f_1,f_2\ldots.f_k$ is a partial function of n variables defined by

$G(f_1(x_1,x_2,x_3\ldots..x_n),f_2(x_1,x_2,x_3\ldots..x_n),\ldots\ldots\ldots f_k(x_1,x_2,x_3\ldots..x_n))$

If for example f_1,f_2 and f_3 are partial functions of two varialbles , then the composition of g with f_1,f_2,f_3 is given by $g(f_1(x_1,x_2),f_2(x_1,x_2),f_3(x_1,x_2)$.

For eg:

Let $f_1(x,y)=x+y,f_2(x,y)=2x$, $f_3(x,y)=xy$ and $g(x,y,z)=x+y+z$ be the functions over N, Then

$g(f_1(x,y),f_2(x,y),f_3(x,y))=g(x+y,2x,xy)$

$=x+y+2x+xy$

Thus the composition of g with f_1,f_2 and f_3 is given by a function h:

$h(x,y)=x+y+2x+xy$

Now we will show that

$f(x,y)=x^y$ is a primitive recursive function.

Soln:

$f(x,y)=x^y$

Zero function:

$f(x,y)=x^y$

put y=0,

$f(x,0)=x^0$

$=1$

Successor function:

$f(x,y)=x^y$

put y=y+1

$f(x,y+1)=x^{y+1}$

$=x^y.x$

$=f(x,y).x$

$=x.f(x,y)$

This can be represented by projection function as

$f(x,y+1)=U_1^3(x,y,f(x,y)*U_3^3(x,y,f(x,y))$

Therefore f(x,y) is primitive recursive.

TIME AND SPACE COMPLEXITY OF COMPLETENESS

Turing machine is defined as the machine which consists of an input tape with the end markers ¢ and $ and k semi infinite tapes where the k is the number of items.
The semi infinite storage tape means only one end is closed and the other end is infinite usually at the right side.

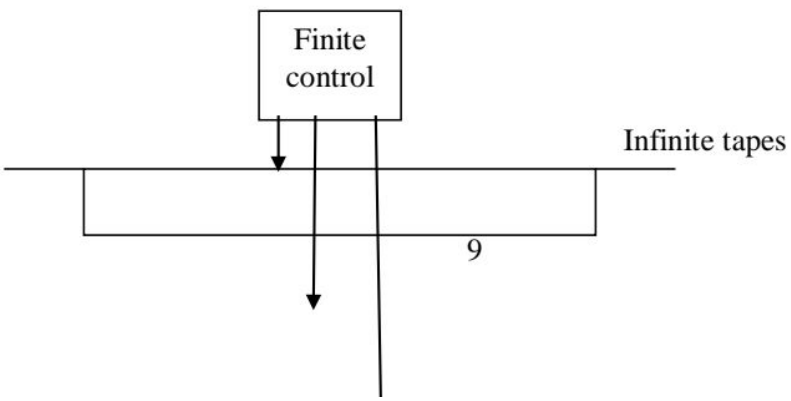
Time Complexity of Turing Machine:

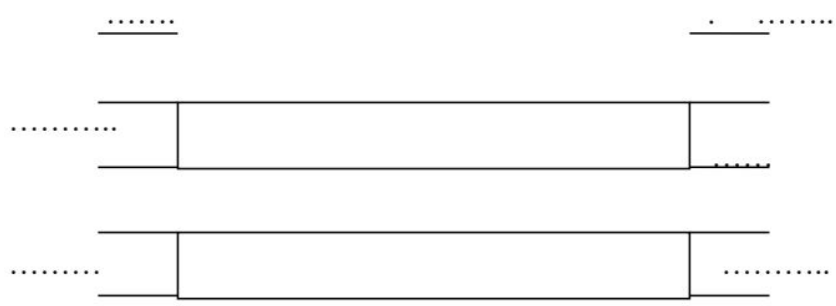
Definition:

For every input word of length n, M(Turing machine) makes at most T(n) moves before halting, then M is said to be a T(n) time-bounded Turing machine, or of time complexity halting ,then M is said to be a T(n) time –bounded Turing machine ,or of time complexity T(n).The language recognized by M is said to be of time complexity T(n).

Example:

Here the Time Complexity is calculated with respect to the Turing Machine which has k-infinite tapes.
For e.g.k=3





No of tapes = No of pointers

Here the first tape is the input tape whereas the other tapes are the storage tape

Consider an language L=abacaba where w € a, b. This language can be written in the form $L=wcw^R$. Here w=aba ,so n value is 3.Since the length of the string is 3 ,the Time complexity of the Turing machine is 3 and an extra one is needed for recognizing the intermediate c. So the total time complexity of the Turing machine is given as T(n)=O(4).

Generally the time complexity of the Turing machine is given as

$$T(n)=O(n+1)$$

Here it does not see the alphabets are equal , only the number strings in the right and the left side of the c are to be equal. If the string lengths are equal then n-time needed for recognizing the string and again 1 for the c-intermediate. So it is called as T(n+1) bounded Turing machine.

Growth Function:

Then the growth function is given as

$$f(n)= \frac{n}{n+1}$$

$$f(n)= \frac{n+1}{n+2}$$

Here n+1 is the Time Complexity for the string of length n and n+2 is the Time Complexity for the string of length n+1.

Space Complexity of Turing Machine:

Definition:

The Space Complexity of the Turing machine is indicated as S(n).For every input word of length n, M (Turing Machine) scans at most S(n) cells on any storage tape, then M is said to be an S(n) space-bounded Turing machine or of space complexity S(n). The language recognized by M is said to be of Space complexity S(n).

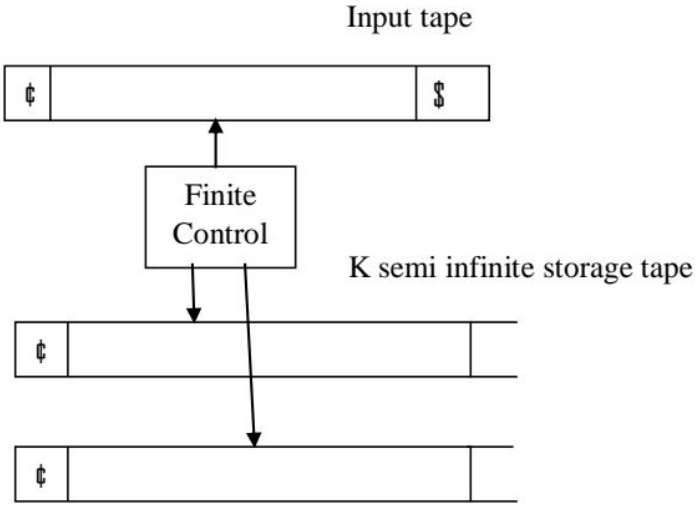
It is also defined as the time required for storing a string of input length n in a k semi infinite tape Turing machine. Here the left side of the Turing machine is bounded and the right side is unbounded.

Example

Here the space complexity of the Turing machine is calculated using the k semi infinite tape as shown above.

Consider an language L=abcba where w € a, b. This language can be written in the form L=wcwR. Here w=ab ,so n value is 2.Since the length of the string is 2 ,the Space complexity of the Turing machine is 2.

The complexity is given as log2n.Here 2 is called the binary counter. The space complexity is given as log2n.Here log is used for the comparison.



Space complexity with respect to Least Upper Bound:

Here the LUB is the suprimum.
It can written as

$$LUB = \sup_{n\to\infty} f(n)$$

$$= \sup_{n\to\infty} \left\lceil \frac{n}{n+1} \right\rceil$$

$$= \sup_{n\to\infty} \left\lceil \frac{n}{n(1+1/n)} \right\rceil$$

$$= \sup \left\lceil \frac{1}{} \right\rceil$$

$n\to\infty \quad (1+1/n)$

Substituting $n=\infty$, we get

$$\text{LUB}= \sup_{n\to\infty} \left[\frac{1}{(1+1/\infty)}\right]$$

we know that $1/\infty$ value is equal to 0,substituting we get,

$$\text{LUB}= \frac{1}{(1+0)}$$

LUB=1

Space complexity with respect to Greatest Lower Bound:

Here the GLB is the infrimum.
It can written as

$$\text{GLB}= \inf_{n\to\infty} f(n)$$

$$= \inf_{n\to\infty} \left[\frac{n}{n+1}\right]$$

$$= \inf_{n\to\infty} \left[\frac{n}{n(1+1/n)}\right]$$

$$= \inf_{n\to\infty} \left[\frac{1}{(1+1/n)}\right]$$

Substituting $n=\infty$, we get

$$\text{GLB}= \inf_{n\to\infty} \left[\frac{1}{(1+1/\infty)}\right]$$

we know that $1/\infty$ value is equal to 0,substituting we get,

$$\text{GLB}= \left[\frac{1}{(1+0)}\right]$$

GLB=1

<u>COMPLEXITY OF CLASSES:</u>

DSPACE(S(n))=deterministic Space complexity(S(n))
NSPACE(S(n))=nondeterministic Space complexity(S(n))
DTIME(T(n))=deterministic Time complexity(T(n))
NTIME(T(n))=nondeterministic Time complexity(T(n))

<u>INTRODUCTION TO NP-HARDNESS AND NP-COMPLETENESS:</u>

NP – complete problems:

In computational complexity theory, the complexity class **NP-complete** (abbreviated **NP-C** or **NPC,** with **NP** standing for nondeterministic polynomial time), is a class of problems having two properties.

- Any given solution to the problem can be verified quickly (in polynomial time); the set of problems with this property is called NP.
- If the problem can be **solved** quickly (in polynomial time), then so can every problem in NP.

Although any given solution to such a problem can be verified quickly, there is no known efficient way to locate a solution in the first place; indeed, the most notable characteristic of NP-complete problems is that no fast solution to them is kown. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows. As a result, the time required to solve even moderately large versions of many of these problems easily reaches into the billions or trillions of years, using any amount of computing power available today. As a consequence, determining whether or not it is possible to solve these problems quickly is one of the principal unsolved problems in computer science today.

While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, computer scientists and programmers still frequently encounter NP-complete problems. An expert programmer should be able to recognize an NP-complete problem so that he or she does not unknowingly waste time trying to solve a problem which so far has eluded generations of computer scientists. Instead, NP-complete problems are often addressed by using approximation algorithms.

Formal overview:

NP-complete is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time; NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a nondeterministic Turing machine. A problem p in NP is also in NPC if and only if every other problem in NP can be transformed into p in polynomial time. NP-complete can also be used as an adjective; problems in the class NP complete are known as NP-complete problems.

NP-complete problems are studied because the abilityl to quickly verify solutions to a problem (NP) seems to correlate with the ability to quickly solve that problem (P). It is not known whether every problem in NP can be quickly solved. This is called the P=NP problem. But if any single problem in NP complete can be solved quickly, then every problem in NP can also be quickly solved, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every problem in NP-complete (that is it can be reduced in polynomial time). Because of this, it is often said that the NP-complete problems are harder or more difficult than NP problems in general.

Formal definition of NP-completeness:

A decision problem C is NP-complete if:
1. C is in NP, and
2. Every problem in NP is reducible to C in polynomial time.

C can be shown to be in NP by demonstrating that a candidate solution to C can be verified in polynomial time.

A problem K is reducible to C if there is a polynomial-time many-one reduction, a deterministic algorithm which transforms any instance k□ϵ□K into an instance c□ϵ□C, such that the answer to c is yes if and only if the answer to k is yes. To prove that an NP complete C is in fact NP-complete problem it is sufficient to show that an already known NP-complete problem reduces to C.

Note that a problem satisfying condition 2 is said to be NP-hard, whether or not it satisfies condition 1.

LIST OF NP-COMPLETE PROBLEMS:-

Some NP-complete problems, indicating the reductions typically used to prove their NP-completeness.

An interesting example is the graph isomorphism problem, the graph theory problem of determining whether a graph isomorphism exists between two graphs. Two graphs are isomorphic if one can be transformed into other simply by remaining vertices.

Consider these two problems:
- Graph Isomorphism: Is graph G1 isomorphic to graph G2?
- Subgraph Isomorphism: IS graph G1 isomorphic to a subgraph of graph G2?

The subgraph isomorphism problem is NP-complete. The graph isomorphism problem is suspected to be neither in P nor NP-complete, though it is obviously in NP. This is an example of a problem that is thought to be hard, but isn't thought to be NP-complete.

The easiest way to prove that some new problem is NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems. The list below contains some well-known problems that are NP-complete when expressed as decision problems.

- Boolean satisfiability problem (Sat.).
- Knapsack problem.
- Hamiltonian path problem.
- Travelling salesman problem.
- Subgraph isomorphism problem.
- Subset sum problem.

- Clique problem.
- Vertex cover problem.
- Dominating set problem.
- Graph coloring problem.

SOLVING NP-COMPLETE PROBLEMS:-

At present, all known algorithms for NP-complete problems require time that is super polynomial in the input size, and it is unknown whether there are any faster algorithms.

The following techniques can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:

- Approximation: Instead of searching for an optimal solution search for "almost" optimal one.
- Randomization: Use randomness to get faster average running time, and allow the algorithm to fail with some small probability.
- Restriction: By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
- Parameterization: often there are algorithms if certain parameters of the input are fixed.
- Heuristic: an algorithm that works "reasonably well" in many cases, but for which there is no proof that if is both always fast and always produces a good result. Metaheuristic approaches are often used.

One example of a heuristic algorithm is a suboptimal O(n logn) greedy coloring algorithm used for graph coloring during the register allocation phase of some compliers a technique called graph-coloring global register allocation. Each vertex is a variable,edges are drawn between variables which are being used to indicate the register assigned to each variable.

BOOLEAN SATISFIABILITY PROBLEM (Sat.):-

Satisfiability is the problem determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Equally important is to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically False for all possible variable assignments. In this latter case, we would say that the function is unsatisfiable; otherwise it is satisfiable. To emphasize the binary nature of this problem, it is frequently referred to as Boolean or propositional satisfiability. The shorthand **"SAT"** is also commonly used to denote it, with the implicit understanding that the function and its variables are all binary-valued.

In complexity theory, the **Boolean Satisfiability problem (SAT)** is a decision problem, whose instance is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: given the expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true? A formula of propositional logic is said to be satisfiable if logical values can be assigned to its variables in a way that makes the formula true. The Boolean satisfiable problem is NP-complete.

A literal is either a variable or the negation of a variable, (for example, X is a positive literal and negation of X is a negative literal.)

A clause is disjunction of literals.

There are several special cases of the Boolean satisfiability problem in which the formulae are required to be conjunction of clauses (i.e. formulae in conjunctive normal form). Determining the satisfiability of a formula in conjunctive normal form where each clause ids limited to at most three literals is NP-complete; this problem is called **"3SAT", "3CNFSAT", or "3-satisfiability".** Determining the satisfiability of a formula in which each clause is limited to at most two literals is NL-complete; this problem is called **"2SAT".** Determining the satisfiability of a formula in which each clause is a horn clause (i.e. it contains at most one positive literal) is P-complete; this problem is called **Horn-satisfiability.**

NP-COMPLETENESS:-

SAT was the first known NP-complete problem, as proved by Stephen cook in 1971. Until that time, the concept of an NP-complete problem did not even exist. The problem remains NP-complete even if all expressions are written in conjunctive normal form with 3 variables per clause (3-CNF), yielding the 3SAT problem. This means the expression has the form:

(x11 OR x12 OR x13)AND
(x21 OR x22 OR x23)AND
(x31 OR x32 OR x33)AND…….
Where each x is a variable or a negation of a variable, each variable can appear multiple times in the expression.

3-satisfiability:-

3-satisfiability is a special case of k-satisfiability (k-SAT) or simply satisfiability (SAT), when each clause contains exactly k=3 literals. Here is an example, where indicates NOT:

E = (x1 or ¬x2 or ¬x3) and (x1 or x2 or x4)

E has two clauses (denoted by parentheses), four literals (x1,x2,x3,x4), and k=3 (three literals per clause).

The knapsack Problem

The knapsack problem or Rucksack problem is a problem in combinatorial optimization: given a set of items, each with a weight and a value, determine the number of each item to include in a collection, so that the total weight is less than a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items.

The decision problem from of the knapsack problem is the question "can a value of at least V be achieved without exceeding the weight W?"

Definition

In the following, we have n kinds of items, 1 through n. Each kind of item j has a value pj and a weight wj. We usually assume that all values and weights are nonnegative. The maximum weight that we can carry in the bag is W.

The most common formulation of the problem is the **0-1 Knapsack problem**, which restricts the number xj of copies of each kind of item to zero or one. Mathematically the 0-1 knapsack problem can be formulated as:

Maximize

Subject to

The **bounded Knapsack problem** restricts the number xj of copies of each kind of item to a maximum integer value bj. Mathematically the bounded Knapsack problem can be formulated as:

Maximize

Subject to

The **unbounded Knapsack problem** places no upper bound on the number of copies each kind item.

To solve this instance of the decision problem we must determine whether there is a truth value (TRUE or FALSE) we can assign to each of the literals (x1 through x4) such that the entire expression is TRUE. In this instance, there is such an assignment

(x1=TRUE,x2=TRUE,x3=TRUE,x4=TRUE), so the answer to this instance is YES. This is one of many possible assignments, with for instance, any set of assignments including x1= TRUE being sufficient. If there were no such assignment(s), the answer would be NO.

Since k-SAT reduces to 3-SAT, and 3-SAT can be proven to be NP-complete, it can be used to prove that other problems are also NP-complete.

Horn-satisfiability

A clause is Horn if it contains at most one positive literal. Such clauses are of interest because they are able to express implication of one variable from a set of other variables. Indeed, one such clause can be rewritten as, that is, if are all true, then y needs to be true as well.

The problem of deciding whether a set of horn clauses is satisfiable is in p.
This Problem can indeed be solved by a single step of the Unit propagation, which produces the single minimal (w.r.t. set of literal assigned to true) model of the set of Horn clauses.

A generalization of the clauses of the class of Horn formulae is that of renamable-Horn formulae, which is the set of formulae that can be placed in Horn form by replacing some variables with their respective negation. Checking the existence of such a replacement can be done in linear time; therefore, the satisfiablity of such formulae is in P as it can be solved by first performing this replacement and then checking the satisablity of the resulting Horn formula.

Of particular interest is the special case, of the problem with these properties:

- It is a decision problem,
- It is a 0-1 problem,
- For each kind of item, the weight equals the value: wj=pj.

Notice that in this special case, the problem is equivalent to this: Given a set of nonnegative integers, does any subset of it add up to exactly W? Or if negative weights are allowed and W is chosen to zero, the problem is: Given a set of integers, does any subset add up to exactly 0? This special case is called the subset sum problem. In the field of cryptography, the term Knapsack problem is often used to refer specifically to the subset sum problem.

COMPUTATIONAL COMPLEXITY:

The knapsack problem is interesting from the perspective of computer science because

- There is a pseudo-polynomial time algorithm using dynamic programming
- There is a fully polynomial-time approximation scheme, which uses the pseudo-polynomial time algorithm as a subroutine.
- The problem is NP-complete to solve exactly, thus it is expected that no algorithm can be both correct and fast(polynomial-time) on all cases.
- Many cases that arise in practice, and "random instances" from some distributions, can nonetheless be solved exactly.
-

UNBOUNDED KNAPSACK PROBLEM:-

If all weights (w1,…wn and W) are nonnegative integers, the knapsack problem can be solved in pseudo-polynomial time using dynamic programming. The following describes a dynamic programming solution for the unbounded knapsack problem.

To simplify thing, assume all weights are strictly positive (wj>0). We wish to maximize total value subject to the constraint that total weight is less than or equal to W. then for each Y□W, define A(Y) to be the maximum value that can be attained with total weight less than equal or equal to Y. A(W) then is the solution to the problem.

Observe that A(Y) has the following properties:

- A(0)=0 (the sum of zero items, i.e., the summation of the empty set)
- A(Y)=max{pj + A(Y-wj) | wj□Y}

Where pj is the value of the jth kind of item.

Here the maximum of the empty set is taken to be zero. Tabulating the results from A(0) up through A(W) gives the solution. Since the calculation of each A(Y) involves examining n items, and there are W values of A(Y) to calculate, the running time of the dynamic programming solution is O(nW). Dividing w1,…..wn, W by their greatest common divisor is an obvious way to improve the running time.

The O(nW) complexity does not contradict the fact that the knapsack problem is NP-complete, since W, unlike N, is not polynomial in the length of the input to the problem.

Hamiltonian path problem

In the mathematical field of graph theory, the Hamiltonian path problem and the Hamiltonian cycle problem are problems of determining whether a Hamiltonian path or a Hamiltonian cycle exists in a given graph (whether directed or undirected). Both problems are NP-complete.

Hamiltonian path

A Hamiltonian path (or unceable path), is a path in an undirected graph which visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a cycle in an undirected graph, which visits each vertex exactly once and also returns to the starting vertex. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem which is NP-complete.

Examples

- A complete graph with more than two vertices is Hamiltonian
- Every cycle graph is Hamiltonian

There is a simple relation between the two problems. The Hamiltonian path problem for graph G is equivalent to the Hamiltonian cycle problem in a graph H obtained from G by adding a new vertex and connecting it to all vertices of G.

The Hamiltonian cycle problem is a special case of the traveling salesman problem,Obtained by setting the distance between two cities to a finite constant if they are adjacent and infinity otherwise.

The directed and undirected Hamiltonian cycle problems were two of karp's 21 NP-complete problems. Garey and Johnson showed shortly afterwards in 1974 that the directed Hamiltonian cycle problem remains NP- complete for planar graphs and the Undirected Hamiltonian cycle problem remains NP-complete for cubic planar graphs.

Randomized Algorithm

A randomized algorithm for Hamiltonian path that is fast on most graphs is following:
Start from a random vertex and continue if there is a neighbor not visited. If there are no more unvisited neighbors, and the path formed isn't Hamiltonian, pick a neighbor uniformly at random, and rotate using that neighbor as a pivot. (That is,add an edge to that neighbor, and remove one of the existing edges from that neighbor so as not to form a loop.) Then, continue the algorithm at the new end of the path.

TRAVELLING SALESMAN PROBLEM

The Travelling Salesman Problem (TSP) is a problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pair wise distances, the task is to find a shortest possible tour that visits each city exactly once.

As a graph problem.

The TSP can be modeled as a graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. A TSP tour becomes a Hamiltonian cycle, and the optimal TSP tour is the shortest Hamiltonian cycle.

Often, the model is a complete graph (i.e. an edge connects each pair of vertices). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

Asymmetric and symmetric.

In the symmetric TSP, the distance between two cities is the same in each direction, forming an undirected graph. This symmetry halves the number of possible solutions. In the asymmetric TSP, paths may not exist in both directions or the distances might be different, forming a directed graph.

With metric distances.

In the metric TSP the intercity distances satisfy the triangle inequality. This can be understood as "no shortcuts", in the sense that the direct connection from A to B is never longer than the detour via C.

These edge lengths define a metric on the set of vertices. When the cities are viewed as points in the plane, many natural distance functions are metrics.

- In the Euclidian TSP the distance between two cities is the Euclidean distance between the corresponding points.
- In the Rectilinear TSP the distance between two cities is the sum of the differences of their x- and y-coordinates.

Non-metric distances.

Distance measures that do not satisfy the triangle inequality arise in many routing problems. For example, one mode of transportation, such as travel by airplane, may be faster, even though it covers a longer distance.

In its definition, the TSP does not allow cities to be visited twice, but many applications do not need this constraint. In such cases, a symmetric, non-metric instance can be reduced to a metric one. This replaces the original graph with a complete graph in which the inter-city distance C_{ij} is replaced by the shortest path between i and j in the original graph.

CLIQUE PROBLEM

In computational complexity theory, the clique problem is a graph-theoretic NP-complete problem. The problem was one of the Richard Karp's original 21 problems shown NP-complete in his 1972 paper "Reducibility among Combinatorial Problems".

A clique in a graph is a set of pair wise adjacent vertices, or in other words, an induced sub graph which is a complete graph. In the graph at the right, vertices 1, 2 and 5 form a clique, because each has an edge to all the others.

Then, the clique problem is the problem of determining whether a graph contains a clique of at least a given size k. Once we have located k or more vertices which form a clique, it's trivial to verify that they do, which is why the clique problem is

in NP. The corresponding optimization problem, the maximum clique problem, is to find the largest clique in a graph.

The NP-completeness of the clique problem follows trivially from the NP-completeness of the independent set problem, because there is a clique of size at least k if and only if there is an independent set of size at least k in the complement graph. This is easy to see, since if a sub graph is complete; its complement sub graph has no edges at all.

Algorithm

A brute force algorithm to find a clique in a graph is to examine each sub graph with at least k vertices and check to see if it forms a clique. This algorithm is polynomial if k is the number of vertices, or a constant less than this, but not if k is, say, half the number of vertices; the number of possible cliques of size k on a graph of size V is equal to combination of vertices (V) and size k.

A heuristic is to start by considering each node to be a clique of size one, and to merge cliques into larger cliques until there are no more possible merges. Two cliques A and B may be merged if each node in clique A is adjacent to each node in clique B. This requires only linear time (linear in the number of edges), but may fail to find a large clique because two or more parts of the large clique have already been merged with nodes that are not in the clique. The algorithm can be implemented most efficiently using the disjoint-set data structure.

Vertex Cover Problem:

A vertex cover of a graph is a set of vertices such that each edge of graph is incident to at least one vertex of the set.

THE COOK-LEVIN THEOREM:

Once we have one NP – complete problem, we may obtain others by polynomial time reduction from it.However ,establishing the first NP-complete problem is more difficult.Now we do so by proving that SAT is NP-complete.

STATEMENT:

SAT is NP-complete.

TO PROVE:

Showing that SAT is in NP is easy , and we do so shortly. The hard part of the proof is showing that any language in NP is polynomial time reducible to SAT.

To do so we construct a polynomial time reduction for each language A in NP to SAT. The reduction for A takes a string w and produces a Boolean formula Ø that simulates the NP machine for A on input w. If the machine accepts, Ø has a satisfying assignment that corresponds to the accepting computation. If the machine doesn't accept, no assignment satisfies Ø. Therefore w is in A if and only if Ø is satisfiable.

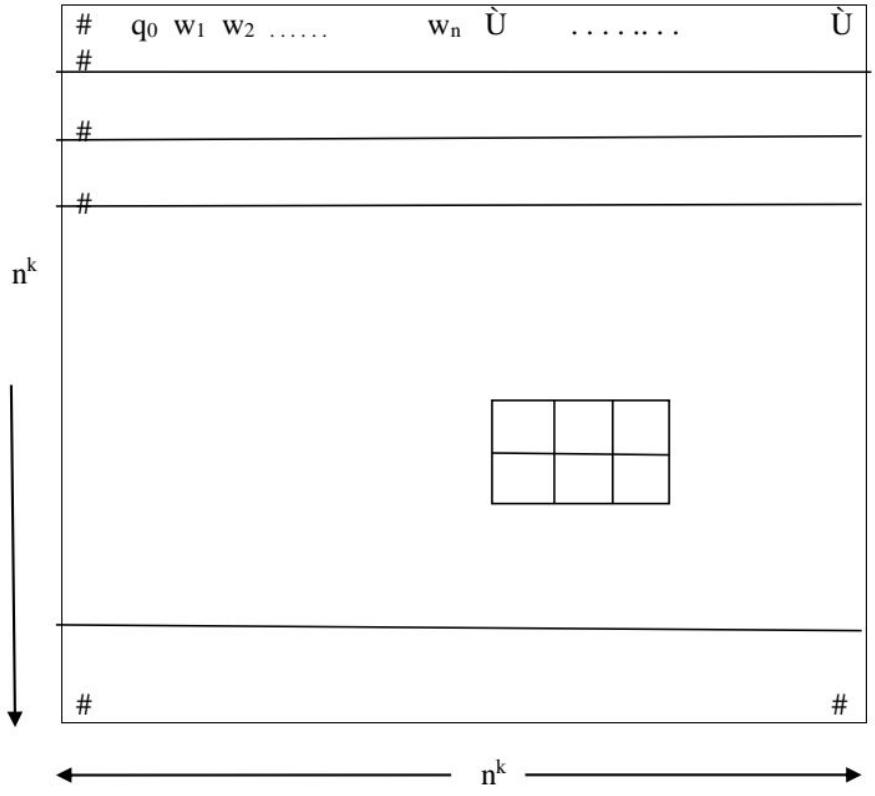
Actually constructing the reduction to work in this way is a conceptually simple task , though we must cope with many details. A Boolean formula may contain the Boolean operations AND, OR and NOT, these operations form the basis for the circuitry used in electronic computers. Hence the fact that we can design a Boolean formula to simulate a Turing machine.

PROOF:

First we show that SAT is in NP. A nondeterministic polynomial time machine can guess an assignment to a given formula Ø and accept if the assignment satisfies Ø.

Next, we take any language A in NP and show that A is polynomial time reducible to SAT. Let N be a nondeterministic Turing machine that decides A in n^k time for some constant k. (we assume that N runs in time n^k-3).The following notion helps to describe the reduction.

A tableau for N on w is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of N on input w ,as shown in the following figure.



A Tableau is an $n^k \times n^k$ table of configurations.

For convenience later we assume that each configuration starts and ends with a # symbol , so the first and last columns of a tableau are all #s. The first row of the tableau is the starting configuration of N on w, and each rows follows the previous one according to N's transition function. A tableau is accepting if any row of the tableau is an accepting configuration.

Every accepting tableau for N on w correspond to an accepting computation branch of N on w. Thus the problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tabuleau for N on w exists.

Now we get to the description of the polynomial time reduction f from A to SAT. On input w, the reduction produces a formula Ø.We begin by describing the variables of Ø. Say that Q and ┌ are the state set and tape alphabet of N. Let C= Q U ┌ U {#}. For each I and j between 1 and n^k and for each s in C we have a variable , $x_{i,j,s}$.

Each of the $(n^k)^2$ entries of a tableau is called a cell. The cell is row I and column j is called cell[i , j] and contains a symbol from C. We represent the contents of the contents of the cells with the variables of Ø. If $x_{i,j,s}$ takes on the value 1 , it means that cell[i , j] contains an s.

Now we design Ø so that a satisfying assignment to the variables does correspond to an accepting tableau for N on q . The formula Ø is the AND of four parts $Ø_{cell}$ ^ $Ø_{start}$ ^$Ø_{move}$ ^$Ø_{accept}$. We describe each part in turn.

As we mentioned previously, Turing variable $x_{i,j,s}$ on corresponds to placing symbol s in cell [i , j]. The first thing we must guarantee in order to obtain a correspondence between an assignment and a tableau is that the assignment turns on exactly one variable for each cell. Formula $Ø_{cell}$ ensures this requirement by expressing it in terms of Boolean operations:

$$Ø_{cell} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \pounds C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \pounds C \\ s \neq t}} (\overline{x_{\iota,J,s}} \, V \overline{x_{\iota,J,t}}) \right) \right]$$

The symbols ^ and v stand for iterated AND and OR . For example , the expression in the preceding formula

$$\bigvee_{s \pounds C} x_{i,j,s}$$

is shorthand for

$$x_{i,j,s1} \, V \, x_{i,j,s2} \, V \ldots V \, x_{i,j,s1}$$

where C = {s1,s2,…sl}.Hence $Ø_{cell}$ is actually a large expression that contains a fragment for each cell in the tableau because i and j range from 1 to n^k. The first part of each fragment says that at least one variable is turned on in the corresponding cell. The second part of each fragment says that no more than one variable is turned on (literally, it says

that in each pair of variables , at least one is turned off) in the corresponding cell. These fragments are connected by ^ operations .

The first part of $Ø_{cell}$ inside the brackets stipulates that at least one variable that is associated to each cell is on , whereas the second part stipulates that no more than one variables is on for each cell. Any assignment to the variables that satisfies Ø (and therefore $Ø_{cell}$) must have exactly one variable on for every cell. Thus any satisfying assignment specifies one symbol in each cell of the table. Parts Østart, $Ø_{move}$ and $ø_{accept}$ ensure that these symbols actually correspond to an accepting tableau as follows.

Formula $Ø_{start}$ ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$Ø_{start}=x_{1,1,\#} \wedge x_{1,2,q0} \wedge x_{1,3,w1} \wedge x_{1,4,w2} \wedge \ldots\ldots \wedge x_{1,n+2,wn} \wedge x_{1,n+3,Ù} \wedge x_{1,nk,\#}.$

$Ø_{accept}$ guarantees that an accepting configuration occurs in the tableau. It ensures that q_{accept}, the symbol fir the accept state, appears in one of the cells of the tableau, by stipulating that one of the corresponding variables is on :

$$Ø_{accept} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{accept}}$$

Finally, formula $Ø_{move}$ guarantees that each row of the table corresponds to a configuration that legally follows the preceding row's configuration according to N's rules. It does so by ensuring that each 2×3 window of cells is legal. We say that a 2×3 window is legal if that window does not violate the actions specified by N's transition function. In other words, a window is legal if it might appear when one configuration correctly follows another.

For example, say that a, b and c are members of the tape alphabet and q_1 and q_2 are states of N. Assume that when in state q_1 with the heading reading an a , N writes a, b stays in state q_1 and moves right , and that when in state q_1 with the heading a b , N nondeterministic ally either

1. Writes a c , enters q_2 and moves to the left , or
2. Writes an a , enters q_2 and moves to the right.

Expressed formally , $\S(q_1,a)=\{(q_1,b,R)\}$ and $\S(q_1,b)=\{(q_2,c,L),(q_2,a,R)\}$.Examples of legal windows for this machine are shown in the following figure.

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

Examples of legal windows

In the above figure , windows (a) and (b) are legal because the transition function allows N to move in the indicated way. Window (c) is legal because, with q_1 appearing on the right side of the top row, we don't know what symbol the head is over. That symbol could be an a, and q_1 might change it to a b and move to the right.That possibility would give rise to this window , so it doesn't violate N's rules. Window (d) is obviously legal because the top and bottom are identical, which would occur if the head weren't adjacent to the location of the window.Note that # may appear on the left or tight of both the top and bottom rows in a legal window. Window (e) is legal because state q_1 reading a b might have been immediately to the right of the top row, and if would then have moved to the left in state q_2 to appear on the right – hand end of the bottom row. Finally , window (f) is legal because state q_1 might have been immediately to the left of the top row and it might have changed the b to a c and moved to the left.

The window shown in the following figure aren't legal for machine N.

. (a) (b) (c)

a	b	a
a	a	a

a	q_1	b
q_1	a	a

b	q_1	b
q_2	b	q_2

Examples of illegal windows

In window (a) the central symbol in the top row can't change because a state wasn't adjacent to it. Window (b) isn't legal because the transition function that the b gets changed to a c but to an a. Window (c) isn't legal; because two states appear in the bottom row.

Part 2:

If the top row of the table is the start configuration and every window in the table is legal, each row of the table is a configuration that legally follows the preceding one.

We prove this claim by considering any two adjacent configurations in the table, called the upper configuration and the lower configuration. In the upper configuration, every cell that isn't adjacent to a state symbol and that doesn't contain the boundary symbol #, is the center top cell in a window whose top row contain no states. Therefore that symbol must appear unchanged in the center bottom of the window. Hence it appears in the same position in the bottom configuration

The window containing the state symbol in the center top cell guarantees that the corresponding three positions are updated consistently with the transition function. Therefore, if the upper configuration is a legal configuration, so is the lower configuration , and the lower configuration follows the upper one according to N's rules.

Now we return to the construction of $Ø_{move}$. It stipulates that all the windows in the tabuleau are legal. Each window contains six cells, which may be set in a fixed number of ways to yield a legal window. Formula $Ø_{move}$ says that the settings of those six cells must be one of these ways , or

$$\emptyset_{move} = \bigwedge_{1<i\leq n^k, 1<j<n^k} (the(i,j)window\ is\ legal)$$

we replace the text "the (I,j) window is legal " in this formula with the following formula. We write the contents of six cells of a window as $a_1,\ldots\ldots,a_6$.

$$\bigvee_{a_1\ldots,a_6\ is\ a\ legal\ window} (x_{i,j-1,a_1} \wedge x_{i,j+1,a_3} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

Next we analyze the complexity of the reduction to show that it operates in polynomial time. To do so we examine the size of Ø. First , we estimate the number of variables it has. Recall that the tabuleau is an $n^k \times n^k$ table, so it contains n^{2k} cells. Each cell has l variables with it, where l is the number of variables in C. Because l depends only on the TM N and not on the length of the input n, the total number of variables is $O(n^{2k})$.

We estimate the size of each of the parts of Ø. Formula $\emptyset_{cell}$ contains a fixed-size fragment of the formula for each of the tabuleau, so its size is $O(n^{2k})$. Formula $\emptyset_{start}$ has a fragment for each cell in the top row, so its size is $O(n^{2k})$. Formula $\emptyset_{start}$ has a fragment for each cell in the top row, so its size is $O(n^k)$. Formulas $\emptyset_{move}$ and $\emptyset_{accept}$ each contain a fixed-size fragment of the formula for each cell of the tabuleau, so size is $O(n^{2k})$. Thus Ø's total size is $O(n^{2k})$. That bound is sufficient for our purposes because it shows that the size of Ø is polynomial in n. If it were more than polynomial , the reduction wouldn't have any change of generating it in polynomial time. (Actually our estimates are low by a factor of O(log n) because each variable has indices that can range up to n^k and so may require O(log n) symbols to write into the formula, but this additional factor doesn't change the polynomial of the result.)

To see that we can generate the formula in polynomial time, observe its higly repetitive nature. Each component of the formula is composed of many nearly identical fragments, which differ only at the indices in a simple way. Therefore we may easily construct a reduction that produces Ø in polynomial time from the input w.

Thus we have concluded the proof of the Cook-Levin theorem, showing that SAT is NP-complete.

University Question Bank.

Unit - I

1) Construct NFA equivalent to the Regular Expression: (0+1) 01.

2. Distinguish NFA & DFA.

3. Define DFA.

4. Give the purpose of Transition diagram.

5. List out the applications of Automata Theory.

6. What is Regular Language?

7. Define ε-closure property.

8. Differentiate b/w Moore & Mealy Machines.

9. What is Regular Expression?

b. Construct two finite automata for the Regular Expression (0+1)* (0+1)*.

Appointments | Phones

11. Write Regular Expression for a set of strings over alphabet {a, b, c} containing atleast one a & atleast one c.

2019
17
Wk 16 • 107-258
Wednesday
April

Unit-II

1. List the closure properties of Regular langua
2. When a grammar is in CNF?
3. When is a grammar Ambiguous? Give Exampl
4. Define GNF.
5. What is a Context free grammar?
6. Define Induction principle.
7. What is the closure property of Regular sets?
8. What is Homomorphism?
9. List any two applications of Regular Expression.
10. State the Relations among Regular Expression deterministic finite automata, NDFA & FA with ϵ-transitions.
11. Find out the CFL $S \to aSb \mid aAb$; $A \to bAa \mid ba$
12. Define Left Most derivation.
13. What is a production? Give Example.
14. What is a derivation tree?

Notes Appointments Phones

Unit - III

What is a push down automata?

What is Recursively Enumerable Language?

Does the CFL have an equivalent Deterministic & Non-Deterministic push down automata for the same Language?

State the pumping Lemma for context free languages.

Define Sentential forms.

What is Deterministic push down automata.

Write the PDA for empty stack?

How can we prove that a given Language is not a CFL?

What is Bottom up parsing?

Appointments

Phones

2019
19
Friday
April
Wk 16 • 109-256

Unit-IV

1. What is Recursive Language?
2. Compare Total & partial Recursive function.
3. What is a Turing Machine?
4. What is a primitive Recursive fn?
5. Define Recursive Enumerable language.
6. What is the Storage in FC?
7. Does a NTM accept a language, accepted by TM (or) DTM. Justify.
8. Give the instantaneous description of TM.
9. State the properties of that satisfy recursively enumerable language.
10. Define Multitrack Machines.
11. What is offline Turing Machine.

April 20

Unit - V

1) What is Space Complexity?

2) What do you mean by Conjuctive Normal form?

3) What is NP Complete class?

4) Define Exponential form & polynomial time Reduction.

5) When do you say a problem is undecidable? Mention any two undecidable problems.

6) Is the travelling Salesman problem NP Complete? If so prove.

2019
Wk 17 • 111-254
Sunday
April
21

7) What is meant by Time Complexity?

8) What is meant by NP class?

9) What is Running time of Turing Machine?

10) Which pblms belongs to class P & NP?

11) Compare time & Space Complexity.

12) State the halting pblm of TM.

13) What are NP class?

14) State the Satisfiability pblm?

8.00

Unit-2

9.00 1) Equivalence of NFA with & without ϵ - Moore.

10.00 2) If there is a language L, such that $L = L(A)$ for some DFA A, then there is a Regular Expression 'r' such that $L = L(R)$.

11.00

12.00 3) $(0+1)^*(0+1)$

1.00 4) Minimization of DFA.

2.00 5) Conversion of NFA to DFA.

6) Moore & Mealy M/c.

3.00

Unit-II

4.00 1) properties of Regular sets in Detail.

5.00 2) $S \to SS/0S1/01$ into CNF with theorem

6.00 3) $S \to aSA/aAA/b$

$A \to bBBB$

7.00 $B \to b/\epsilon$ into CNF with theorem.

4) parse trees.

Notes Appointments Phones

Unit - III

1) properties of cfl with Example.
2) pumping lemma for cfl.
3) If V is context free Language, then proved that there exists a pda M such that $L = N(M)$.
4) push down Automata 5) Decision Algorithms

Unit - IV

1) $L = \{a^n b^n a^n b^n : n > 1\}$ and for binary Language for palindrome strings.
2) Various Types of Turing M/c.
3) Recursive Enumerable Languages.
4) Undecidable pblms

Unit - V

1) Time & Space Complexity of TM
2) Np problems
   - Vertex cover
   - SAT
   - clique
3) Np Hardness & Np Completeness. Also explain when a pblm is Np-Hard & when it is Np-complete