**VIGNAN’S UNIVERSITY :: VADLAMUDI**

|  |  |  |
| --- | --- | --- |
| **Faculty** | **:** | **P.RAMADOSS** |
| **Subject** | **:** | **Formal Languages and Automata Theory** |
| **Code** | **:** | **16IT207** |
| **Year & Branch** | **:** | **III IT** |

**UNIT – I**

**FINITE STATE SYSTEMS**

* The finite Automation is a mathematical model of a system, with discrete inputs and outputs.
* The system can be in any one of the finite number of internal configuration (or) states.
* The states of the system summaries the information concerning past inputs that are needed to determine the behavior of the system on subsequent inputs.

**Example**: Automatic M/C tools, Automatic packing M/C’s and automatic photo printing M/C’s

Automation

q1 ,q2,q3,…..qn,

I1 O1

I2 O2

I3 O3

1. **Input**: At each of the discrete instants of time t1,t2….. input value I1,I­2,I3,…..Ip. Each of which can take a finite number of fixed values from the input alphabet ∑, are applied to the input side of model.
2. **Output**: Output o1,o2, …..oq are the output’s of the model, each of which can take finite numbers of fixed values from an output O.
3. **States**: At any instant of time the automation can be in one of the state’s q1,­q2,…..qn.
4. **State relation**: The next state of automation at any instant of time is determined by the present state and the present input.
5. O/P Relation: Output is related to either state only (or) to both the input and the state.

* It should be noted that at any instant of time the automation is in some state.
* On reading an input symbol, the automation moves to a next state which is given by the state relation.

**FORMAL LANGUAGES**

(Format of the string is important but meaning is not important).

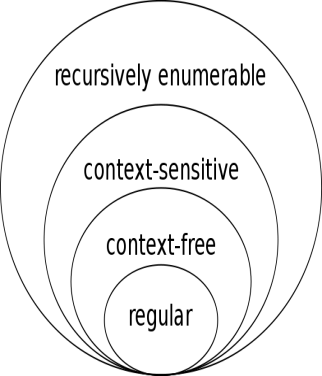
* It is also collection of strings like language but here meaning of the string is not important, but format is important.

Example :{ aaa, aabb, aaaabbbb}

**Types of formal languages**

According to Chomsky, the formal languages are of 4 types.

1. Type 3 (or) Regular Languages
2. Type 2 (or) Context Free Languages
3. Type 1 (or) Context Sensitive Languages
4. Type 0 (or) Recursively Enumerable sets.



* To recognize or accept these formal languages, we need automated tool.
* To recognize the Regular language, we have the Finite Automaton.
* Types 3 (or) Regular Languages are accepted by Finite Automaton (FA).
* Types 2 (or) Context Free Languages (CFL) are accepted by Push Down Automaton (PDA).
* Types 1 (or) Context Sensitive Languages (CSL) are accepted by Linear Bounded Automaton (LBA).
* Type 0 (or) Recursively Enumerable sets (RE) are accepted by Turing Machine I.

|  |  |  |  |
| --- | --- | --- | --- |
| Grammar | Languages | Automaton | Production rules (constraints) |
| Type-0 | [Recursively enumerable](http://en.wikipedia.org/wiki/Recursively_enumerable_language) | [Turing machine](http://en.wikipedia.org/wiki/Turing_machine) | (no restrictions) |
| Type-1 | [Context-sensitive](http://en.wikipedia.org/wiki/Context-sensitive_grammar) | [Linear-bounded non-deterministic Turing machine](http://en.wikipedia.org/wiki/Linear_bounded_automaton) |  |
| Type-2 | [Context-free](http://en.wikipedia.org/wiki/Context-free_grammar) | Non-deterministic [pushdown automaton](http://en.wikipedia.org/wiki/Pushdown_automaton) |  |
| Type-3 | [Regular](http://en.wikipedia.org/wiki/Regular_grammar) | [Finite state automaton](http://en.wikipedia.org/wiki/Finite_state_automaton) | A🡪a and  A🡪Ab|Ba |

**PRILIMINARIES**

**Symbol:**

A symbol is an abstract entity. There is no formal definition for a symbol.

Examples: letters, digits and special characters etc.

**Alphabet:**

A finite non-empty set of symbols is called ‘alphabet’. It is denoted by ∑ .

Example**: ∑**={a,b,…z}, **∑ =**{a,b}, **∑**={0,1}

**String:**

A sentence over an alphabet is a finite sequence of the symbols from an alphabet.

**Example**:

1. In ∑={a,b} is the alphabet

ab,abb are valid strings, where as abc is not a string because ‘c’ doesn’t belongs to ∑

2. In **∑ =** {0, 1} is the alphabet

00,110,1011 are valid strings.

Note: The length of any string over the given alphabet is at least 1.

**Assumption:**

In theory of computation, there exists a string of length zero called as ‘epsilon’ i.e. ‘’ .

S. =S

.S=S

.S.=S

**Prefix**: Any sequence of leading symbols over the given string is called ‘prefix’.

*Example*: For the string ‘cse’, the possible prefixes are ,c, cs, cse.

**Suffix**: Any sequence of trailing symbols over the given string is called ‘suffix’.

*Example*: For the string ‘cse’, the possible suffixes are , e, se, cse

**Substring**: Any sequence of symbols over the given string is called ‘substring’.

*Example*: For the string ‘computer’ the possible substrings are , com, put, te, mp,….

**Power of an Alphabet:**

Assume input alphabet ∑= {a,b}

∑0 = {} i.e. the set of all strings possible over the alphabet of length Zero

Similarly,

∑1= {a,b}

∑2= {aa,ab,bb,ba}

∑3= {set of all strings possible over the alphabet of lenth ‘n’}

∑ \*= ∑0∑1∑2∑3….∑n

∑\*= set of all strings over ∑ including ‘’

Here ‘\*’ is the kleene closure operator

* ∑+=∑ \*-

=∑1∑2…..∑n

Here ‘+’ is the positive closure operator

 ∑\*=∑+

**Language**:

Language is a set of strings formed from some specific alphabet set. Language over ∑, L is any subset of ∑\*.

Example:∑ = {a, b}

∑\* = {, a, b, aa, ab, ba, bb, aaa, aab, ... }

L = {a, aa, aab} is a finite language on ∑

Star-closure of language L ⊆ \*

L\* = L0 ∪ L1 ∪ L2  ∪... Remember that L0 = {

set of all strings by concatenating strings from L ... strings are “building blocks” rather than elements from 

Positive Closure

L+ = L1 ∪ L2 ∪ ... = L\* - L0

**1. FINITE AUTOMATA**

A Finite Automaton (FA) consists of a finite set of states and set of transitions among states in response to inputs.

* Always associated with a FA is a transition diagram, which is nothing but a ‘directed graph’.
* The vertices of the graph correspond to the states of the FA.
* The FA accepts a string of symbols from ∑, x if the sequence of transitions corresponding to symbols in x leads from the state to an accepting state.

FA without output or Language Recognizers ( e.g. DFA and NFA)

Finite Automata (FA)

FA with output or Transducers ( e.g. Moore and Mealy machines)

* 1. **Deterministic Finite Automata (DFA):**

A Deterministic Finite Automaton is represented by a 5-Tuple machine:

i.e M = (Q, , , q0, F)

where

Q is finite set of states

 is finite input alphabet

 is transition mapping function i.e Q x  → Q

q0 ∈ Q Initial state

F ⊆ Q Set of final states

**Acceptance by an Automaton:**

* A string “w” is said to be accepted by a finite automation M=(Q, , , q0, F)

If  q0, w)=p for some p in F. The language accepted by M, designated L(M), is the set {w| q0, w) is in F}.

* A language is a regular set, if it is the set accepted by some automaton.
* There are two preferred notations for describing automata

1. Transition Diagram
2. Transition Table

**Examples**:

1. Give DFA for accepting the set of all strings containing even number of 0s over an alphabet {0,1}.

Transition Diagram:

****

*DFA tuples are M* = (*Q*, Σ, δ, *q0*, *F*) where

*Q* = {*S*1, *S*2}, Σ = {0, 1}, *q0* = *S*1, *F* = {*S*1}, and δ is defined by the following [state transition table](http://en.wikipedia.org/wiki/State_transition_table)

**Transition Table**: Q x  → Q

|  |  |  |
| --- | --- | --- |
|  | **0** | **1** |
| ***S*1** | *S*2 | *S*1 |
| ***S*2** | *S*1 | *S*2 |

1. Give DFA for all accepting strings over {a,b} that contains exactly 2 a’s.

Transition Diagram:

****

DFAtuples *are M* = (*Q*, Σ, δ, *q0*, *F*) where

*Q* = {0,1,2,3},Σ = {a,b},*q0* = *0*,*F* = {2}, and δ is defined by the following [state transition table](http://en.wikipedia.org/wiki/State_transition_table)

**Transition Table** : Q x  → Q

|  |  |  |
| --- | --- | --- |
|  | **a** | **b** |
| ***0*** | 1 | 0 |
| ***1*** | 2 | 1 |
| ***2*** | *3* | *2* |
| ***3*** | *3* | *3* |

**1.2 Non-Deterministic Finite Automata (NFA):**

An Non-Deterministic Finite Automata is represented by a 5 – tuple.

M = (Q, , , q0, F)

where

Q is finite set of states

 is finite input alphabet

 is transition mapping function i.e Q x  →2Q

q0 ∈ Q Initial state

F ⊆ Q Set of final states

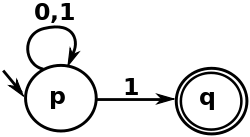
**1.2.1 Acceptance by NFA:**

An NFA accepts a string “w” if it is possible to make any sequence of choices of next state, while reading the characters of w, and go from start state to any accepting state.

**Example:**

1. Construct an NFA for the set of all strings over the alphabet {0,1}containing the string ends with a 1.

Transition Diagram:



NFA tuples are M = (Q, , , q0, F) where

*Q* = {p, q}, Σ = {0, 1}, *q0* = p, *F* = {q}, and δ is defined by the following [state transition table](http://en.wikipedia.org/wiki/State_transition_table)

**Transition Table**: Q x  → 2Q

|  |  |  |
| --- | --- | --- |
|  | **0** | **1** |
| *p* | {p} | {p,q} |
| *q* | ∅ | ∅ |

**1.3 Differences between DFA and NFA**

* In the case of DFA, the transition function gives exactly one state, when applied an input symbol.
* In the case of NFA, there can be several possible next states, and the automation ‘guesses’ (always correctly) which next state (of the set of possible next states) will lead to acceptance of the input string.
* DFA is a particular case of NFA so, transition function in NFA is Q x  →2Q

**Note:**

1. All DFAs are NFAs
2. All NFAs are not be DFAs

**1.4 Conversion of NFA to DFA**

Let N = (QN, ∑, N, q0, FN) be a NFA.

We have to construct the DFA, D= (QD, ∑, D, {q0}, FD) Such that L (D) =L (N).

 Language accepted by DFA should be same as language accepted by NFA.

Step1: Convert the given transition system into state transition table where each state corresponds to a row and each input symbol corresponds to a column.

**Step 2:** Construct the succession table which lists subsets of states reachable from the set of initial states.

**Step 3:** the transition graph given by the successor table is the required deterministic system.

* The final states contain some final state of NFA. If possible we can reduce the number of states.

1. Construct DFA equivalent to the following NFA.



**Sol:**

Equivalent DFA construction: D=(QD,∑, D,{ q0},FD)

QD={{A},{A,B},{A,C},{A,B,C}}

∑={0,1}

FD={{A,C},{A,B,C}}

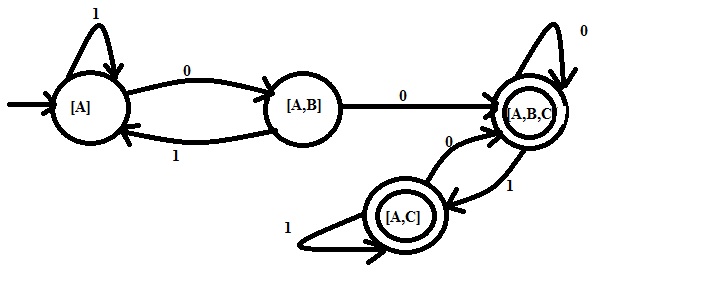
(These are final states as C is the final state of NFA and all these states contain C)

The transition function D is

**Transition Table:**

|  |  |  |
| --- | --- | --- |
| State/∑ | 0 | 1 |
| 🡪[A] | [A,B] | [A] |
| [A,B] | [A,B,C] | [A] |
| \*[A,B,C] | [A,B,C] | [A,C] |
| \*[A,C] | [A,B,C] | [A,C] |

**Transition Diagram:**

****

**Note:**

While converting NFA to DFA

1. No change in the initial state.
2. Number of final states may be changed
3. Number of states may be changed.
4. Transition function is not changed.
5. If NFA contains ‘n’ states, then the equivalent DFA contains maximum 2n states. These state names are also subset of given ‘n’ states.

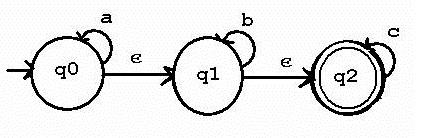
**1.5 Finite Automation with - Moves (Epsilon Transitions)**

* An NFA is allowed to make a transition spontaneously, without receiving an input symbol. These **-** NFA’s can be converted to DFA’s accepting the same language.
* Finally NFA with - **moves** can be defined to be a 5-Tuple.

M= (Q, ∑,, q0,F) with  mapping from Q x (∑{}) 🡪 2Q.

**- closure(q):**

**-** closure(q) is the set of all vertices p such that there is a path from q to p on **-** alone.



In the figure,

- closure(q0)={q0,q1,q2}

- closure(q1)={ q1,q2}

- closure(q2)={ q2}

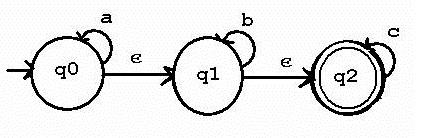
**1.5.1 Conversion of an - NFA to NFA without - moves:**

Convert a transition system with - moves into an equivalent transition system without - moves.

Suppose we want to replace an - moves from vertex V1 to vertex V2

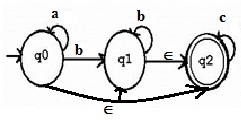
1. Find all edges starting from V2
2. Duplicate all these edges starting from V­1, without changing the edge labels.
3. If V1 is an initial state, make V2 also an initial state.
4. If V2 is a final state make V1 also a final state.

Example:

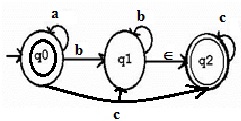


Solution:

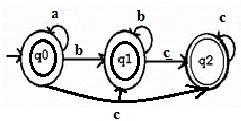
We first eliminate the - moves from q0 ­to q1

****

Then we eliminate the - moves from q­0 to q2



Next eliminate the - moves from q1,q2.



**1.6 Moore Machine**

* A Moore machine is a FA in which the output is associated with the state.
* A Moore machine is a 6- Tuple (Q, ∑,,,,q0 )

Where

Q: finite set of states.

∑: finite set of input symbols

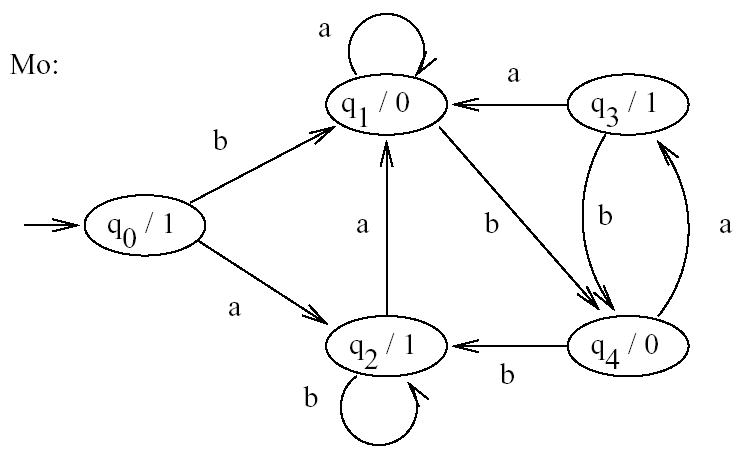
: finite set of output alphabet

: Transition function i.e. QX∑→Q

(output function): Q🡪( is a function from Q to )

q0: initial state

Example:



Note:

* Without taking an input, a Moore machine produces the output.
* If the length of the input string is ‘n’ the Moore machine produces the output string of length ‘n+1’.

**1.7 Mealy machine**

* In Mealy machine output is associated with each transition, output will be dependent on present state and present input symbol.
* A mealy machine is a 6-Tuple (Q, ∑,,,,q0 )

Where

Q: finite set of states.

∑: finite set of input symbols

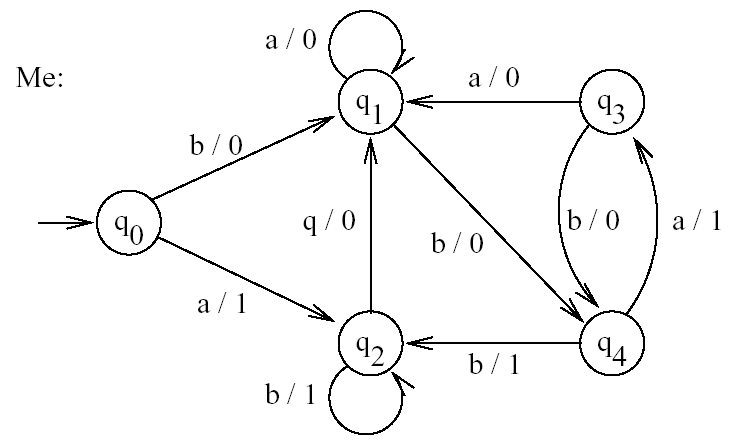
: finite set of output alphabet

: Transition function i.e. QX∑→Q

(output function): QX∑🡪(i.e. (q,a) gives the output associated with the transition from state q on input a)

q0: initial state

Example:



Note:

* Without giving any input the Mealy machine doesn’t generate output.
* If the length of the input string is ‘n’ the Mealy machine produces the output string of length ‘n’.
  1. **Equivalence of Moore and Mealy machines** 
     1. **Mealy machine equivalent to Moore machine:**

Theorem: If M1 = (Q, ∑,,,, q0 ) is a Moore machine, then there is a Mealy M2

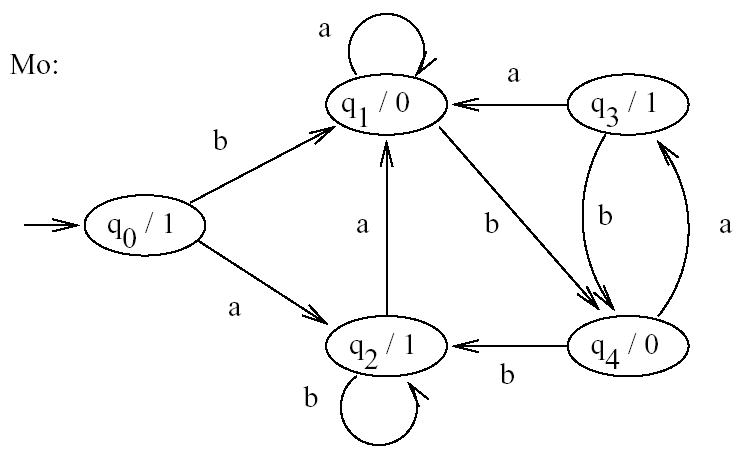
equivalent to M1.

Proof :

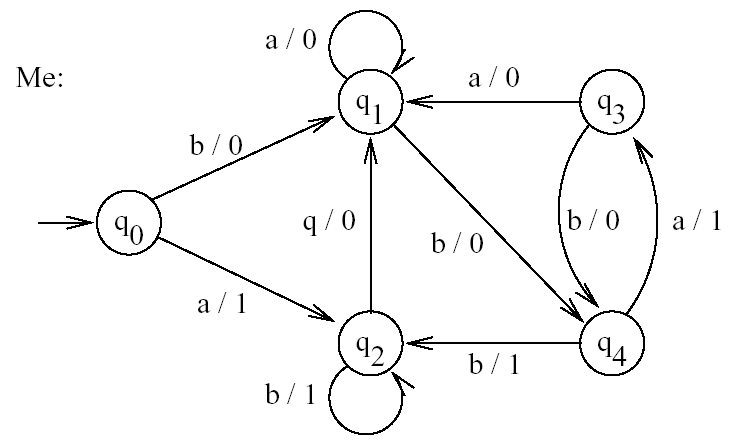
Let M2 =(Q, ∑,,, ­­­1 ,q0, ) and define ­­­1(q,a) to be ­­((q,a)) for all states q and input symbol ‘a’.

Then M1and M2enter the same sequence of states on the same input, and with each transition M2 emits the o/p that M1 associates with the state entered.

Example: Construct an equivalent an Mealy machine for the following Moore machine.



Solution:



**1.8.2 Moore machine equivalent to Mealy machine:**

Theorem: If M1 = (Q , ∑,,, ,q0 ) be a Mealy machine, then there is a Moore machine

M2 equivalent to M1**.**

Proof :

Let M1 =(Q X,∑,1, 1 ,  [q0 ,b0 ]) where b is arbitrary selected member of . That is ,the states of M2 are pairs [q,b] consisting of a state of M1 and o/p symbol.

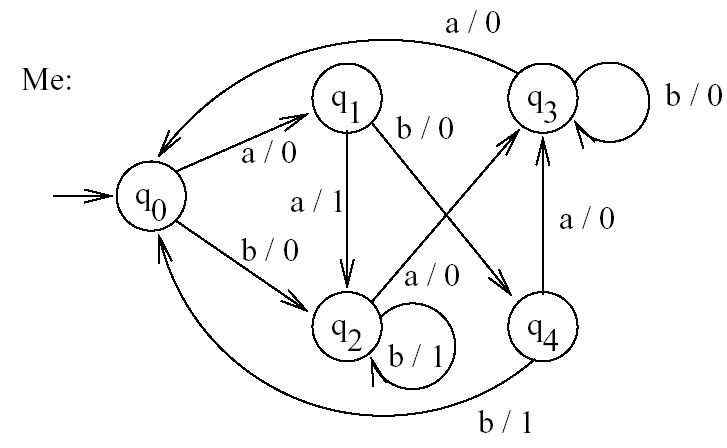
Define 1 ([q,b],a) =[(q,a),  (q,a) and 1 ([q,b]) =b.

The second component of state [q,b] of M2  is the output made by M1 on some transition into state q.

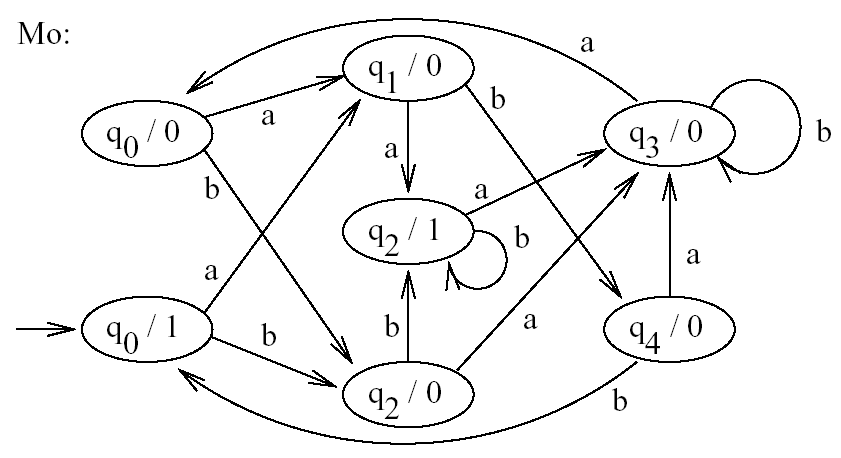
Only the first components of M2 ‘s states determine the moves made by M2  .

Every induction on ‘n’ shows that if M1  enters states q0 ,q1 ---- qn .on input a1 , a2,---aN and emits output b1 ,b2 ------bn then M2 enters states [q0,b0], [q1,b1]---, [qn,bn] and emits outputs b0,b1..bn.

**Example:** Transform the following Mealy machine into its equivalent Moore machine.



Solution:



**UNIT - II**

**1. REGULAR EXPRESSIONS**

This algebraic notation describes exactly the same language as finite automata, the regular languages. The regular expression operators are union, concatenation (or ‘dot’), and closure (or “star”).

**Definition**:

Let ∑ be a given alphabet. Then

1. ,,and ‘a’ ∑ are all Regular expressions. These are called ‘Primitive Regular expressions’
2. If r1and r2 are regular expressions, so are r1+ r2, r1r2, r1\* and (r1).
3. A string is a Regular expression, if and only if it can be derived from the primitive Regular expressions by a finite number of the rules in (ii).

**1.1 Language Associated with Regular Expressions**:

Regular expressions can be used to describe some simple languages. If r is a regular expression, we will let L(r) denote the language associated with r The language is defined as follows.

**Definition:**

The language L(r) denoted by any regular expression r is Defined by following rules.

1.  is a regular expression denoting the empty set.
2.  is a regular expression denoting the set{}
3. For every a∑, ‘a’ is a regular expression denoting set set {a}.

If r1 and r2 are regular expressions, then

1. L(r1+r2)=L(r1)L(r2)
2. L(r1.r2)=L(r1)L(r­2)
3. L(r1\*)=(L(r1))\*

**Example:**

Exhibit the language L (a\*.(a+b)) In set notation

L(a\*.(a+b))=L(a\*)L(a+b)=(L(a\*))(L(a) L(b))

={,a,aa,aaa,…..}{a,b}={a,aa,aaa,…..,b,ab,aab,…}

**1.2 Precedence of Regular Expression Operators**

(1) Kleene closure has higher precedence than concatenation operator.

(2) Concatenation has higher precedence than union operator.

**1.3 Equivalence of Regular expressions:**

* Two regular expressions are said to be equivalent if they denote the same language

**Example:** Consider the following regular expressions

r1=(1\*011\*)\*(0+)+1\*(0+) and r2=(1+01)\*(0+)

Both r1 and r2 represent the same language i.e. the language over the alphabet {0,1} with no pair of consecutive zeros. So r­1 and r­2 are said to be equal.

**1.4 Algebraic Laws For Regular Expressions:**

Let r1, r2and r3 be three regular expressions.

1. Commmmtative law for union:

* The commutative law for union, say that we take the union of two languages in either order. i.e. r1+r2=r2+r1

2 Associative laws for union:

* The association law for union says that we may take the union of three languages either by taking the union of the first two initially or taking the union of the last two initially.

(r1+r2)+r3= r1+( r2+r3)

3. Associative law for concatenation:

(r1r2) r3= r1 (r2r3)

4. Distributive laws for concatenation:

* Concatenation is left distributive over union

i.e. r1 (r2+r3) = r1r2+ r1r3

* concatenation is right distributive over union

i.e. (r1+r2) r3=r1r3+ r2r3

5. Identities For union And Concatenation:

*  is the identity for union operator

i.e. r1+=+r1=r1

*  is the identity for concatenation operator

i.e. r­1=r1=r1

6. Annihilators for Union and Concatenation:

* Annihilator for an operator is a value such that when the operator is applied to the Annihilator and other value, the result is the Annihilator**.**

 is the Annihilator for concatenation

i.e. r1= r1=

there is no Annihilator for union operator.

7. Idempotent law for Union:

* This law states that if we take the union of two identical expressions, we can replace them by one copy of the expression.

i.e. r1 + r1=r1

8. Laws involving closure

* Let ‘r’ be a regular expression ,then

1. (r\*) =r\*

2. \*=

3. \*=

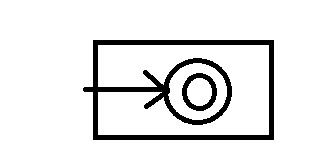
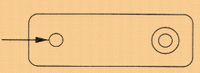
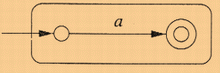
4. r+=r.r\*=r\*.r i.e. r+=rr\*=r\*r

5. r\*=r++

6. r?= +r (Unary postfix operator ? means zero or one instance)

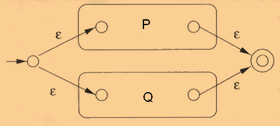
**2. Construction of** - **NFA from a regular expression**

**Basis:** Automata for ,and ‘a’ are (a),(b) and (c) respectively.

1. Accepting **∈** b) Accepting φ c) Accepting a

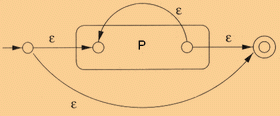
**Induction:** Automata for P+Q, PQ and P\* are (d), (e) and (f) respectively.



d) P+Q

image005

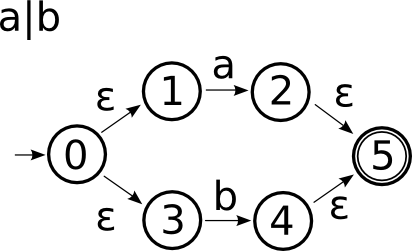
 e) PQ



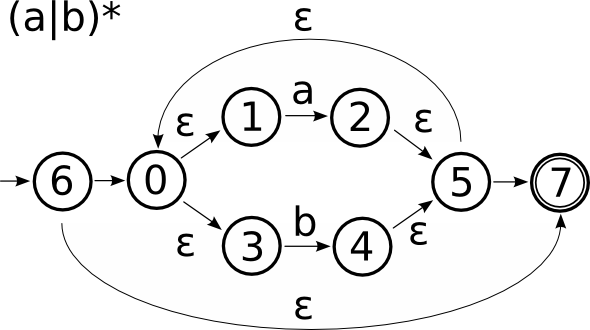
f) P\*

**Example**: Construct -NFA for the regular expression (a|b)\*|c

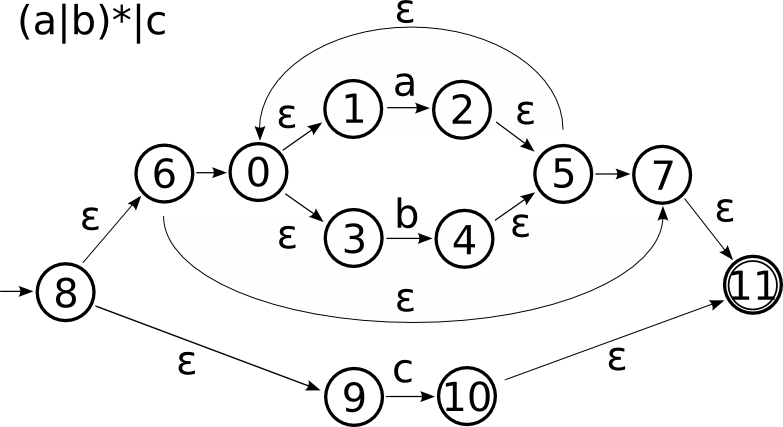
**Solution:** using Thompson's Construction. First we construct the union of a and b:



Next we construct the Kleene Star of the previous union:



Finally we create the union between this and the next symbol c:



**2.1 Construction of DFA from a regular expression:**

This procedure can be explained easily with an example.

STEP 1: Augment the given regular expression with the symbol ‘# `

Ex: if (a|b)\*abb is the given regular expression. After augmenting # the regular

expression becomes (a|b)\*abb#

STEP 2: Give positions to each symbol in the regular expression including # symbol.

Ex: (a | b)\*a b b #

1 2 3 4 5 6

STEP 3: Find Firstpos of the given regular expression. Firstpos is a set that contains the

positions of all the symbols which can be at the beginning of a valid word of the

regular expression.

Ex: The Firstpos (a|b)\*a b b #) is {1,2,3} which are corresponding to the words given 1 2 3 4 5 6

Below

‘1’ is included in the Firstpos because of the word a b a b b # or a a b b #

1 2 3 4 5 6 1 3 4 5 6

‘2’ is included in the Firstpos because of the word b a b b #

2 3 4 5 6

‘3’ is included in the Firstpos because of the word a b b #

3 4 5 6

STEP 4 :Find following of each symbol. Followpos of a symbol is a set which contains the

positions of all the symbols which can follow the current symbol.

Ex: in the regular expression (a | b)\* a b b #

1 2 3 4 5 6

Followpos(1)= {1,2,3}

Followpos(2)= {1,2,3}

Followpos(3)= {4}

Followpos(4)= {5}

Followpos(5)= {6}

Followpos(6)= 

STEP 5: construct Dstates the set of states of DFA, and Dtran,the transition table for DFA by the procedure given below .the states in Dstates are sets of positions ;initially each state is “unmarked” and state becomes “marked “just before we consider its outtransitions .the start state of DFA is Firstpos( regular expression) which is computed in step 3 ,and the start state are all those containing the position associated with the marker #.

PROCEDURE:

Initially the only unmarked state in Dstates is start state

*While* there is an unmarked state T in Dstates *do begin*

Mark T;

For each input symbol a do begin

Let U be the set of positions that are in followpos(P) for some position p in T ,such that the symbol at position p is a ;

*If* U is not empty and is not in Dstates *then*

add U as an unmarked state to Dstates;

Dtran[ T,a]:=U

*End end*

**Example:**

Root for DFA of regular expression (a/b)\* abb is {1,2,3} from step3 .

Let this set be A and consider input symbol a .positions 1 and 3 are for a ,so let B=followpos(1) U followpos(3) = {1,2,3,4}. Since this set has not yet been seen,we set

Dtran[A.a]:=B and addB to Dstates.

When we consider input b,we note that of the positions in A ,only 2 si associated with b,so we must consider the set followpos(2)={1,2,3}.Since this set has alredy been seen,we do not add it to Dstates but we add the transition Dtran[A,b]:A.

Now consider B on input ‘a’ positions 1 and 3 are for ‘a’ in B so Dtran[B,a]=followpos(1)

U followpos(3)={1,2,3,4}=B

On input ‘b’ Dtran[B,b]=follow pos(2) followpos(4)={1,2,3,4,5}

As this is the new state name it C and to Dstates .Dtran[B,b]=C.

Now we consider state C on input ‘ a’.

Dtran [C,a]=followpos(1) followpos(3)={1,2,3,4}=B

Dtran [C,b]=followpos(2) followpos(5)={1,2,3,6}

As this is the new state name it as D and add to Dstates.  Dtran[C,b]

Dtran [D,a]=followpos(1) followpos(3)={1,2,3}{4}={1,2,3,4}=B

Dtran [D,a]=followpos(2)= {1,2,3}=A

As D only contains positional number of end marker # it is the only final state.

**Transition table:** a b

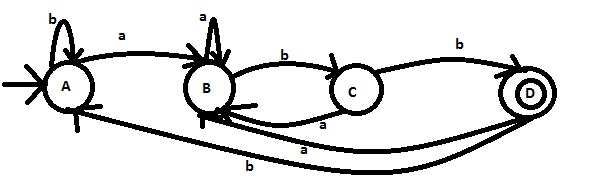
A B A

B B C

C B D

\*D B A

**Transition Diagram**:



**3. Construction of regular expression from Finite Automata:**

**Arden’s theorem:** Let P and Q be two regular expression over ∑. If ‘P’ does not contain  then the equation in R=Q+RP has unique solution (i.e only one solution) given by R=QP\*.

**Method for finding regular expression of Finite automata in transition diagram representation using Arden’s theorem:**

The following assumptions are made regarding the finite automata.

1. The finite automaton does not have  - moves.
2. It has only one initial state, say q0.
3. It’s states are q0,q1.... qn
4. Qi is the regular expression representing the set of string accepted by the automata even through qi is a final state.
5. ij denotes the regular expression representation the set of labels of edges from vi to vj when there is no such edge ij= .Consequently, we can get the following set of equation in Q1, ….Qn

Q1= Q111+ Q221+….+ Qnn1+

Q2= Q112+ Q222+….+ Qnn2

……………………………………………..

……………………………………………..

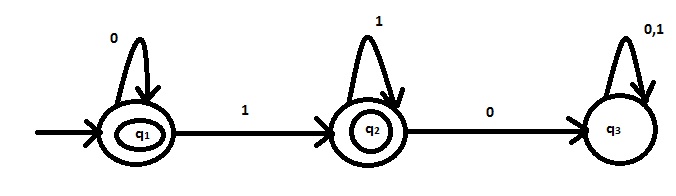
Qn= Q11n+ Q22n+…..+ Qnnn

By repeatedly applying substitutions and Arden’s theorem we can express Ri in terms of ij’s for getting the set of strings recognized by the automata, we have to take union of all Ri’s

Corresponding to final states.

**Example1:**

Derive a regular expression from the following given FA?



Sol:

q1= +q10 ……………. (1)

q2=q11+q21 …………… (2)

q3=q20+q30+q31 ………. (3)

(2) 🡪 q2=q11+q21

q2=q111\* ……….. (4)

1. 🡪 q1= +q10 (Apply Arden’s theorem)

q1=0\*

q1=0\*

(4) 🡪q2=0\*11\*

q2=0\*1\*

**4. REGULAR GRAMMARS**

**Definition of Grammar:**

A phrase-Structure grammar (or Simply a grammar) is (V, T, P, S), Where

1. V is a finite nonempty set whose elements are called variables.
2. T is finite nonempty set, whose elements are called terminals.
3. S is a special variable (i.e an element of V ) called the start symbol, and
4. P is a finite set whose elements are 🡪,where  and  are strings on V T,

 has at least one symbol from V. Elements of P are called Productions or production

rules or rewriting rules.

**4.1 Right –Linear Grammar:**

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

A🡪xB or A🡪x.

Where A,B  V and x  T\*.

**4.2 Left –Linear Grammar:**

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

A🡪Bx or A🡪x

**4.3 Regular Grammar:**

A grammar G=(V,T,P,S) is said to be Regular grammar if the productions are in either right-linear grammar or left-linear grammar.

A🡪Bx|xB|x

**Example**:

1. Construct the regular grammar for regular expression r=0(10)\*.

Sol:

Right-Linear Grammar:

S🡪0A

A🡪10A|****

Left-Linear Grammar:

S🡪S10|0

**5. Construction of -NFA from a right –linear grammar:**

Let G=(V,T,P,S) be a right-linear grammar .we construct an NFA with -moves,

M=(Q,T,,[S],[ ]) that simulates deviation in ‘G’

Q consists of the symbols [] such that  is S or a(not necessarily proper)suffix of some

Right-hand side of a production in P.

We define  by :

1 .If A is a variable ,then ([A], )={[]|A🡪 is a production}

2. If a is in T and  in T\*  T\*V, then ([a],a)={[ ]}

**Example:**

1. Construct an NFA for the right linear grammar S🡪0A , A🡪10A|****

Sol:

From the above theorem, we know that

(1)IF A is a variable, then ([A],)={[]A-> is a production }

(2) If a is in T and  in T\*T\*V,then ([a],a)={ []}.

The states Q ={[S]},[0A],[A],[10A],[E]}

These states are the suffix of right hand side of production P.

S-> 0A is a production gives a path from [S] to [0A] on reading input symbol E.

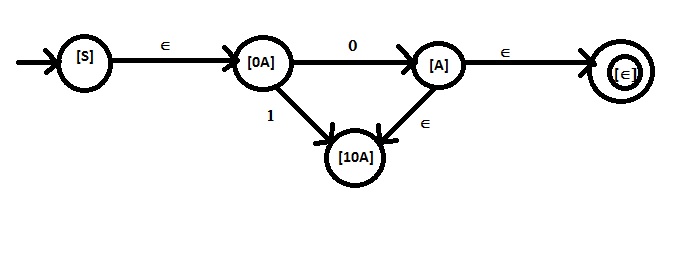
A-> 1-A production establish a path from [A] to [10A] on reading input symbol 

According to rule 1

A->  is also the direct production ,reading input  ,it gives the out put []

([0A],1)=[0A]

The Starting symbol is S and [] is the final state always.



**6. Construction of -NFA from a left-linear grammar:**

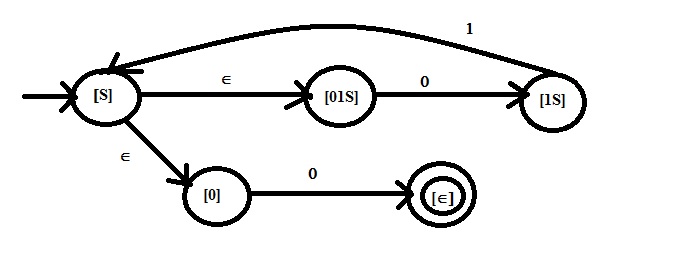
If ‘G’ is a left-linear grammar we will reverse all the right hand sider of the production then we will get right-linear grammar from which we will construct -NFA of given left- linear we will exchange initial ,final states and reverse the direction of all the edges.

Example:

1. Construct an NFA for the grammar S🡪 S10|0

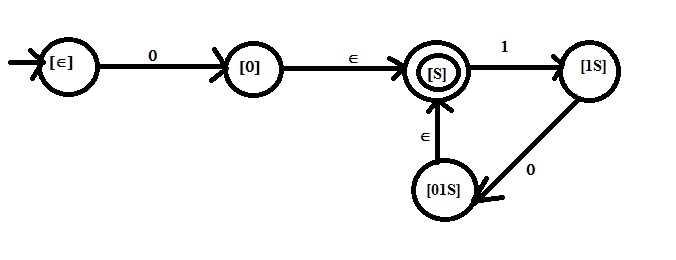
The grammar is LLG .But we can construct an NFA from the RLG Only To get the RLG from simply we will make reverse the right edges of the production

For this Grammar the NFA is



The  is derived as in the previous example. ([0,],0)= 

Now we wil reverse the edges of that NFA and exchange the initial state as final state then the ,NFA is



This is the for the given LLG

S🡪S10|0

**7. Construction of right –linear and left –linear grammar from a given Finite Automata:**

**Right –linear grammar:**

Let M=(Q,∑,,q0,F) be the given finite automata . First suppose that q0 is not a final state. Then L=L(G) for a right –linear grammar G=( Q,∑, P,q0) where P consists of production p-> aq whenever (p,a)=q and also p->q whenever (p,a) is a final state.

Now let q0 be final state,so  is in L so introduce a new start symbol S with productions.

S->q0/ 

Example:

1. Derive the right linear grammar for the following DFA for 0(10)\*

Solution:

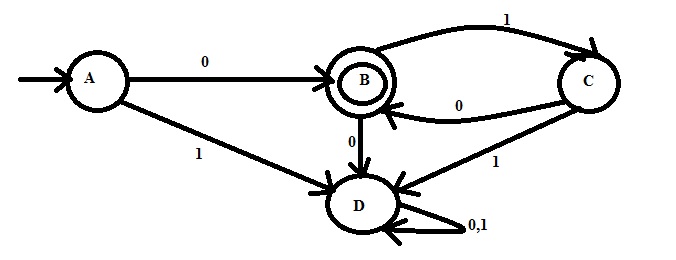
From the above Theorem

The transition fuction (p,a)= q gives a production p->aq.

If  (p,a)= final state, then

p🡪 a is one more production

DFA:

****

The starting state can made as a starting symbol of the grammar.

(A,0)= B is represented by a production A->0B.

(A,0)=D =>A 🡪 0D

(A,1)= D =>A 🡪1D

We can write

B 🡪 OD from (B,0)=D

B 🡪 1C from (B,1)=C

C 🡪 0B from (C,0)=B

C 🡪 1D from (C,1)=D

D 🡪 0D from (D,0)=D

D 🡪 1D from (D,1)=D

From the above DFA diagram

We can also write A🡪 0and C 🡪 0 as

(A,0)= Final state and

(C,0)= Final state

There are also to be added to the respective variables.

The RLG for The DFA is

A 🡪 0B|1D|0

B 🡪 0D|1C

C 🡪 0B|1d|0

D 🡪 0D|1D

Here D is useless symbol so thaht it is to be eliminated. For Applying the reduction technique the grammar can be reduced as

A 🡪 0B|0

B 🡪 1C

C 🡪 0B|0

**Left –linear grammar :**

To get left-linear grammar reverse the right-hand sides of all the production of right –linear grammar.

Example:

1. Construct left linear grammar for the above regular expression 0(10)\*

Solution:

To get left-linear grammar reverse the right-hand sides of all the production of right –linear grammar.

RLG: A 🡪 0B|0

B 🡪 1C

C 🡪 0B|0

LLG:

A 🡪 B0|0

B 🡪 C1

C 🡪 B0|0

**8. Closure Properties of Regular Languages**

1.Regular languages are closed under union, concation and kleene closure.

2. Regular languages are closed under complementation.i.e .,if L1 is a Regular language and L1 ∑\*,then L1 =∑\*-L1 is Regular language.

L1 L1

L1 L1

3. Regular languages are closed under intersection .

i.e., if L1 and L2 are Regular languages then L1 L2 is also a Regular language.

4. Regular languages are closed under difference.

i.e., if L and M are Regular languages then so is L-M

5. Regular languages are closed under Reversal operator.

6. Regular languages are closed under substitution.

7. Regular languages are closed under homomorphism andinverse homomorphism.

**Pumping lemma for Regular languages**

Non Regular Lang Non Regular Lang

Pumping lemma

**Note**

1. Pumping Lemma is used to prove non-regularity of language.
2. Pumping Lemma uses pigeon hole principle to show that certain languages are not Regular.

STATEMENT :Let ‘L’ be an infinite Regular language.then there exits some positive integer ‘n’ such that any Z L with ≥n can be decomposed as Z=uvw

With |uv| ≤n, and |v|≥1,such that z=uviw is also in L for all i=0,1,2,…….

Let us apply it on L={0n1n/n≥1}

**Proof :**

Assume L to be Regular and apply pumping lemma.

By pumping Lemma

There exists some n ≥ 0

We choose w=0n1n

Clearly wL and |w| ≥ n

 By pumping Lemma

Choose Z = uvw such that v≠ and |uv |≤n

But, since uv occurs at the beginning of the word and its total length can’t be more than n, it is bound to have only 0’s in it.

Let |u|=a ,a,|v|=b and |uv|=a+b≤ n ,b≥0

Then z=0a 0b0n-a-b.1n

If we choose k=0,

UvkwL

U,wL

0n-b1n L

But for b>0 ,this is not true ,a contradiction .

Hence L is not a Regular Language.

Example:

1. Prove or disprove the regularity if the language {aibi | i>j}

Solution :

Let the given language is regular, So it must satisfy strong form of pumping lemma, and there exists a DFA with ‘n’ states.

2n n

aaaa………….aaabbbb…………bbb

u v w

Z1 Z2  Z3 (Z3 Is )

Z1uvi wZ3  L

Choose |v| =d and i =3n.

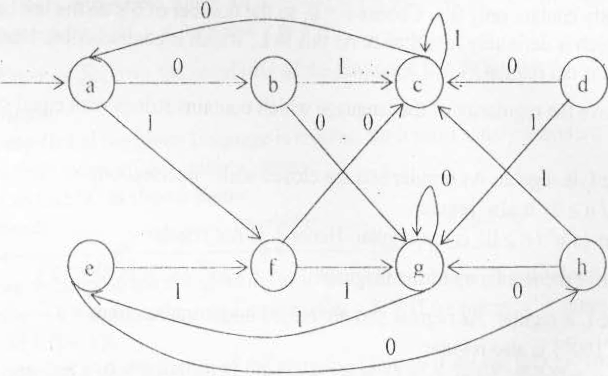
a2n b(n-d)+3nd =a2nb(3d+1) n - d

which has more number of the b’s than a’s .So the given language is not regular.

**9. MINIMIZATION OF FINITE AUTOMATA**

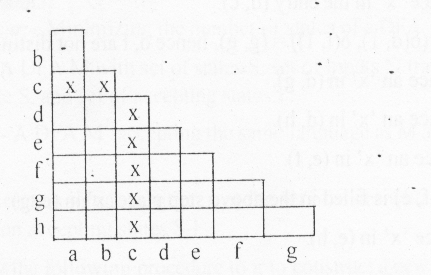
We say that a state p is distinguishable from state q if there exist a string x such that (p,x) is in F and (q .x) is not. or vice versa. Otherwise they are said to be equivalent.

Example:

Minimization can be explained easily with an example. Consider the following DFA and minimize it.

**Solution:**

Construct a table as shown below and place an ·x· in the table each time we discover a pair of states that cannot be equivalent, that is they arc distinguishable. Initially an ·x' is placed in each entry corresponding to one final state and one non final state.



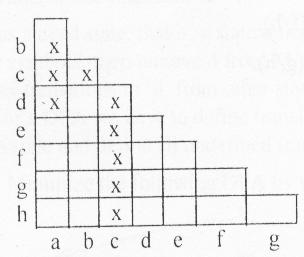
Next for each pair of states p and q that are not already known l0 be distinguishable we construct the pairs of states r = (p. a) and s = (q, a) for each input symbol a. If states r and s have been shown to be distinguishable by some string x. then p and q arc distinguishable by string x. Thus if the entry (r, s) in the table has an ·x', and 'x' is also placed at the entry (p, q). If the entry (r, s) does not yet have an 'x', then the pair (p, q) is placed on a list associated with the (r, s) entry. Al some future time if the (r. s) entry receives an 'x', then each pair on the list associated with the (r, s)-entry also receives an 'x'.

Continuing with the example, we place an 'x' in the entry (a. b) since the entry

( (b. 1), (a, 1)) = (c. f) already has an 'x'. Similarly. the (a, d)- entry receives an 'x’ since the

entry (8(a, 0), 8(d, 0) = (b, c) has an':<.".

Now the table is



( (a, 1), (e, 1))= (f, f), on input ‘1’ both are going to same state. So no string starting with '1' can distinguish states a and c. Now try on input ‘0’.

( (a, 0), (e, 0)) = (b, h), as (b, h) is not filled, so associate (a, e) with (b, h)-list.

-> (𝛿 (a, 0), (f. 0)) = (b, c), as (b,c) already has an ’x’ place 'x' in the entry (a, f).

-> (𝛿 (a, 0), (g. 0)) = (b, g), as (b, g) is not filled associate (a. g) with (b, g)-list

-> ((a, 1 ), 𝛿(h, 1)) = (f, c), as (f, c) already has an 'x' place an 'x' in the entry (a, h) also.

-> (𝛿 (b, 0), (d, 0)) =(g, c). Place ·x' in the entry (b, d).

-> (𝛿 (b, 1), (c. 1)) = (c. 1), place 'x' in the entry (b, e)

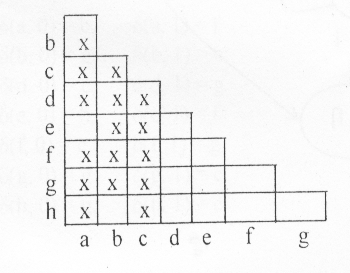
-> (𝛿 (b, 1), (f. 1)) = (c, g), place 'x' in the entry (b. f).

-> (𝛿 (b, 1), 𝛿 (g, 1)) = (c, e), place ‘x’ in the entry (b, g). As (a, g) is associated with (b. g) place and 'x' in {a, g) also.

-> ((b. 1), 𝛿(h,1)) = (c, c) and (𝛿(b. 0), 𝛿(h, 0))=(g, g). On each input symbol slates b and h are

going to the same state. Hence they are not distinguishable.

Now the table is



((d, 0), 𝛿(e, 0))= (c, h), place 'x' in the entry (d. c).

(𝛿 (d, 0), (f, 0)) = (c, c) and ((d,1 ) , 𝛿(f, 1)) = (g, g). hence d, fare not distinguishable.

-> (𝛿 (d, 0), (g, 0)) = (c, g), place an 'x' in(d, g). .

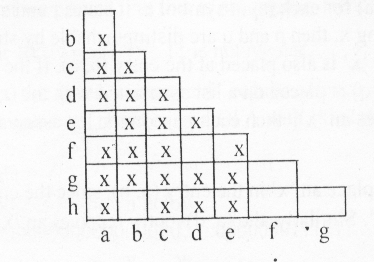
-> (𝛿 (d, 0). (h, 0)) = (c, g), place an 'x' in (d, h).

-> (𝛿 (c, 0), (f, 0)) = (h, c), place an 'x' in (e, f).

-> (𝛿 (c, 1), (g, 1))= (f. e), as (f, e) is filled in the above step place ‘x ‘ in (c, g).

-> ( (e, 1),𝛿(h,1)) = (f, c), place 'x' in (e, h).

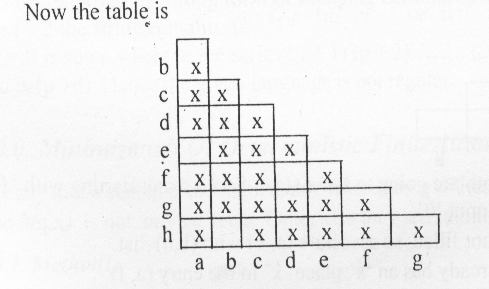
Now the table is



-> (𝛿 (f, 0), 𝛿 (g, 0)) = (c, g), place an ‘x’ in (f. g).

-> (𝛿 (f, 0), 𝛿 (h, 0)) = (c, g), place an ‘x’ in (f, h).

-> (𝛿 (g, 1), 𝛿 (h, 1)) = (e, c), place an ‘x’ in (g , h).

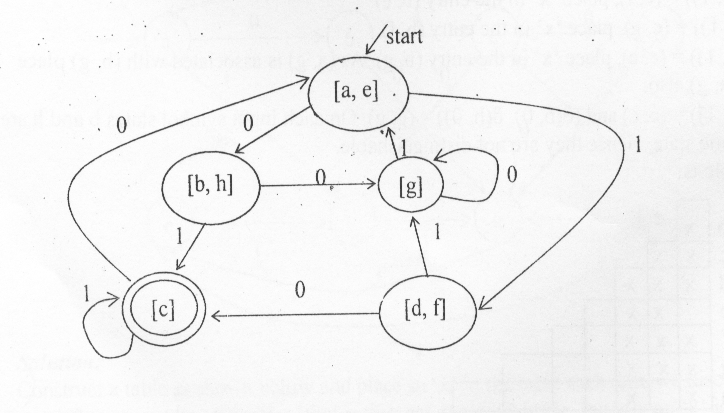


from the above table, the equivalent states(that arc not filled with 'x') are (a, e), (b, h) and



(d, f).

The minimum-state finite automaton is



**UNIT – III**

**CONTEXT FREE GRAMMAR**

A *context-free grammar* (CFG or just grammar) is defined formally as G=(V,T,P,S)

Where

V: a finite set of variables (“non-terminals”); e.g., A, B, C, …

T: a finite set of symbols (“terminals”), e.g., a, b, c, …

P: a set of production rules of the form A → α, where A ∈ V and α ∈ (V U T)\*

S: a start non-terminal; S ∈ V

A context-free grammar consists of a set of productions of the form A → α, where ‘A’ is a single non-terminal symbol and ‘α’ is a potentially mixed sequence of terminal and non-terminal symbols.

Eg: E 🡪 E+E

E 🡪 E\*E

E 🡪 (E)

E 🡪 id

In the above example, grammar tuples are defined as follows:

G=({E},{+,\*,(,),id},{ E 🡪 E+E, E 🡪 E\*E, E 🡪 (E), E 🡪 id},E).

In this chapter we use the following conventions regarding grammars.

1. The capital letters A,B,C,D,E and S denote variables; S is the start symbol unless otherwise stated.
2. The lowe-case letters a,b,c,d,e,digits,special symbols and boldface strings are terminals.
3. The capital letters X,Y and Z denote symbols that may be either terminals or varibales.
4. The lower-case letters u,v,w,x,y and z denote strings of terminals.
5. The lower-case Greek letters α,β and γ denote strings of varibles and terminals.

Generally we specify the grammar by listing the productions.

If A 🡪 α1, A 🡪 α2, A 🡪 α3, … A 🡪 αk are the prodcution then we may express then by

A 🡪 α1 | α2 | α3 | … | αk

**Context Free Language**:

* If G is a CFG, then L(G), the language of G, is {w | S  w }.

Note: ‘w’ must be a terminal string, S is the start symbol.

**Examples**:

1. Construct a CFG to generate set of palindromes over alphabet {a,b}.

**Solution**:

The productions of a grammar to generate palindromes over {a,b} are

S 🡪 aSb | bSb | ∈

Hence S ** aSa ** abSba **ab∈ba ** abba

This is the even palindrome.

Productions to generate odd palindrome are

S 🡪 aSb | bSb | a | b

Hence S ** aSa ** abSba ** ab a ba ** ababa

This is the odd palindrome.

2.Design CFG for a given language L(G)={ aibi | i ≥ 0}

**Solution**: L={ ∈,ab,aabb,aaabbb,…}

S 🡪 aSb | ∈

3.Design CFG for a given language L(G)={ aibi | i > 0}

**Solution**: L={ab,aabb,aaabbb,…}

S 🡪 aSb | ab

4.Design CFG for a given language L(G)={ wwR| w is binary}

**Solution**: L={∈,00,11,0110,1001,010010,…}

S 🡪 0S0 | 1S1 |∈

5.Design CFG for a given language L(G)={ w#wR| w is binary}

**Solution**: L={#,0#0,1#1,01#10,10#01,010#010,…}

S 🡪 0S0 | 1S1 |#

6.Design CFG for a regular expression r=(a+b)\*

**Solution**: L={∈,a,b,aa,ab,ba,bb,aaa,aab,bbb,bba,…}

S 🡪 aS | bS |∈

7. Give a CFG for the set of all well formed paranthesis.

**Solution**: S🡪SS | (S) | ( )

**2. DERIVATION**

* We derive strings in the language of a CFG by starting with the start symbol, and repeatedly replacing some variable A by the right side of one of its productions.
* That is, the “productions for A” are those that have A on the left side of the 🡪.
* αAβ  whenever there is a production A 🡪 γ
* Subscript with name of grammar, e.g.,

 if necessary.

Example: 011AS  0110A1S

* α β means string α can become β in zero or more derivation steps.

Example: 011AS  011AS (zero steps);

011AS  0110A1S (one step);

011AS  0110011 (three steps);

**Sentential Forms:**

* Any string of variables and/or terminals derived from the start symbol is called a

sentential form.

* Formally, α is a sentential form iff Sα.

**Types of Derivations:**

🡪We have a choice of variable to replace at each step.

* + Derivations may appear different only because we make the same replacements in a different order.
  + To avoid such differences, we may restrict the choice.

1. **Left Most Derivation (LMD):** If at each step in a derivation a production is applied to the leftmost variable, then the derivation is called left most derivation.
2. **Right Most Derivation (RMD):** If at each step in a derivation a production is applied to the rightmost variable, then the derivation is called right most derivation.

**🡪 ** used to indicate derivations are leftmost and rightmost.

**Derivation/Parse Trees:**

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

A derivation tree is constructed with

1) Each tree vertex is a variable or terminal or epsilon

2) The root vertex is S

3) Interior vertices are from V, leaf vertices are from T or epsilon

4) An interior vertex A has children, in order, left to right, X1, X2, ... , Xk when

there is a production in P of the form A -> X1 X2 ... Xk

5) A leaf can be epsilon only when there is a production A -> epsilon

and the leafs parent can have only this child.

**Example 1: C**onstruct parse tree for the following CFG and take input string is 0110011.

S 🡪 AS | ∈

A 🡪 0A1 | A1 | 01

**Sol:** Before constructing parse tree, first derive the given input string from the CFG.

**LMD**: S  AS  A1S  011S  011AS  0110A1S  0110011S  0110011.

**RMD**: S  AS  AAS  AA  A0A1  A0011  A10011  0110011.

**Parse tree:**



**3. Ambiguous Grammars:**

A CFG is ambiguous if one or more terminal strings have multiple leftmost derivations or multiple rightmost derivations or multiple parse trees from the start symbol.

**Example 1:** Consider the following grammar:

S 🡪 AS | ∈

A 🡪 0A1 | A1 | 01

The above CFG, the string 00111 has the following two leftmost derivations from S.

Sol:

LMD 1: S  AS  0A1S  0A11S  00111S  00111

LMD 2: S  AS  A1S  0A11S  00111S  00111

Intuitively, we can use A 🡪 A1 first or second to generate the extra 1.

**Example 2:**

Consider the following grammar:

S → SS

S → aSb

S → bSa

S → ∈

and the string w = aabb. We can draw the following 2 trees with the same string w = aabb, so we say the grammar is “ambiguous” in this case.

* If we can find either 2 leftmost / rightmost derivations or 2 different derivation trees, then we can say the grammar is ambiguous.

**4. Inherently Ambiguous context free language:**

* A context free language for which we cannot construct an unambiguous grammar is inherently ambiguous CFL.

Example:

L={anbncmdm | n≥1,m≥1} U {anbmcmdn | n≥1,m≥1}

* An operator grammar is a CFG with no ∈-productions such that no consecutive symbols on the right sides of productions are variables.
* Every CFL without ∈ has an operator grammar.
* If all productions of a CFG are of the form A🡪xB or A🡪x, then L(G) is a regular set where x is a terminal string.

**5. SIMPLIFICATION OF CFG**

* In a CFG we may not use all the symbols for deriving a sentence. So, we eliminate symbols are productions in G, which are not useful.
* We can “simplify" grammars to a great extent. Some of the things we can do are:

1. Elimination of useless symbols: those that do not participate in any derivation of a terminal string.
2. Elimination of ∈ - productions: those of the form variable 🡪 ∈.
3. Elimination of Unit productions: those of the form variable 🡪 variable.

**5.1 Eliminating Useless symbols:**

* In order for a symbol X to be useful, it must:

1. Derive some terminal string (possibly X is a terminal).

2. Be reachable from the start symbol; i.e., S αXβ

* Note that X wouldn't really be useful if α or β included a symbol that didn't satisfy (1), so it is important that (1) be tested first, and symbols that don't derive terminal strings be eliminated before testing (2).

Examples:

1.Eliminate useless symbols from the grammar

S 🡪 AB | a

A 🡪 a

Solution:

Here we find no terminal string is derivable from B. So that B is to be eliminated from

productions S 🡪 AB.

Remaking productions are

S 🡪 a

A 🡪 a

By rule 2, Here A is not useful to derive a string from starting symbol S. So we can eliminate A 🡪 a.

The final production is

S 🡪 a

2. Eliminate useless symbols from the grammar

S 🡪 aS | A | C

A 🡪 a

B 🡪 aa

C 🡪 aCb

Solution:

By rule 2, B is not useful to derive a string from starting symbol S. So we can eliminate

B 🡪 aa.

The Remaking productions are,

S 🡪 aS | A | C

A 🡪 a

C 🡪 aCb

By rule 1, C is not useful to derive some terminal string. So we can eliminate

S 🡪 C and C🡪aCb productions.

The final productions are

S 🡪 aS | A

A 🡪 a

**5.2 Eliminating** ∈ **- productions:**

* A variable A is nullable if A ∈. Find them by a recursive algorithm.

Basis: If A 🡪 ∈ is a production, then A is nullable.

Induction: If A is the head of a production whose body consists of only nullable

symbols, then A is nullable.

* Once we have the nullable symbols, we can add additional productions and then throw away the productions of the form A 🡪 ∈ for any A.
* If A 🡪 X1 X2 … Xk is a production, add all productions that can be formed by eliminating some or all of those Xi's that are nullable.
* But, don't eliminate all k if they are all nullable.

Examples:

1.Grammar G:

S 🡪 aS | bA

A 🡪 aA | ∈ , from this grammar eliminate ∈-productions.

Solution:

S 🡪 aS, S 🡪 bA gives S 🡪 bA and S 🡪 b

A 🡪 aA gives A 🡪 aA and A 🡪 a

After elimination of∈-productions, the final grammar is

S 🡪 aS | bA | b

A 🡪 aA | a

2.Grammar G:

S 🡪 AaB | aaB

A 🡪 ∈

B 🡪 bbA | ∈, from this grammar eliminate ∈ - productions and then eliminate

useless symbols.

Solution:

The given grammar is

S 🡪 AaB | aaB ……….…………………..(1)

A 🡪 ∈………………………… (2)

B 🡪 bbA | ∈ ……………….…(3)

Step 1: Elimination of ∈ - productions

The given grammar contains two ∈-proctions. i.e A 🡪 ∈ and B 🡪 ∈

1. ⇒ S 🡪 AaB | aaB | aB | Aa | a | aa {since A 🡪 ∈ and B 🡪 ∈ }
2. ⇒ B 🡪 bbA | bb

The grammar is

S 🡪 AaB | aaB | aB | Aa | a | aa

B 🡪 bbA | bb

Step 2: Elimination of useless symbols from the above grmmar

S 🡪 AaB | aaB | aB | Aa | a | aa

B 🡪 bbA | bb

In this grammar Variable A is there, but it is not producing anything. So that it can

eliminated.

The remaking productions are

S 🡪 aaB | aB | a | aa

B 🡪 bb

In this grammar no symbol is useless, then the final productions are,

S 🡪 aaB | aB | a | aa

B 🡪 bb

**5.3 Eliminating Unit productions:**

* The productions of the form A 🡪 B, where A,B ∈ V called unit production.
* Eliminate useless symbols and ∈ - productions.
* Discover those pairs of variables (A,B) such that A B.
* Because there are no ∈ - productions, this derivation can only use unit productions.
* Thus, we can find the pairs by computing reachablity in a graph where nodes = variables, and arcs = unit productions.
* Replace each combination where A  B  α and α is other than a single variable by A 🡪 α.
* i.e., “short circuit" sequences of unit productions, which must eventually

be followed by some other kind of production.

Remove all unit productions.

**Note:** Consider the grammar G is S 🡪 A, A 🡪 B, B🡪 C, C 🡪 d.

Here A, B, C are the unit variables of length one. Then the resultant grammar is S 🡪 d. This is called the chain rule.

Example:

1.Eliminate unit productions from the following grammar.

S 🡪 A | bb

A 🡪 B |b

B 🡪 S | a

Solution:

In the given grammar, the unit productions are S 🡪 A, A 🡪 B and B 🡪 S.

S 🡪 A gives S 🡪 b.

S 🡪 A 🡪 B gives S 🡪 B gives S 🡪 a.

A 🡪 B gives A 🡪 a

A 🡪 B 🡪S gives A 🡪 S gives A 🡪 bb.

B 🡪 S gives B 🡪 bb.

B 🡪 S 🡪 A gives B 🡪 A gives B 🡪 b.

The new productions are

S 🡪 bb | b | a

A 🡪 b | a | bb

B 🡪 a | bb | b

It has no unit productions. In order to get the reduced CFG, we have to eliminate the useless symbols. From the above grammar we can eliminate the A and B productions.

Then the resultant grammar is **S 🡪 bb | b | a**.

**6. NORMAL FORMS**

* In a Context Free Grammar, the right hand side of the production can be any string of variables and terminals. When productions in G satisfy certain restrictions, then G is said to be in a Normal Form.
* There are two widely useful Normal forms of CFG. They are

# Chomsky Normal Form (CNF)

1. Greibach Normal Form ( GNF )

**6.1 Chomsky Normal Form (CNF)**:

Definition: A context-free grammar G is in Chomsky normal form if any production is of the form:

A 🡪 BC or

A 🡪 a

where ‘a’ is a terminal, A,B,C are non-terminals, and B,C may not be the start variable (the axiom)

Note:

1. In CNF number of symbols on right side of production strictly limited.
2. The rule S🡪∈, where S is the start variable, is not excluded from a CFG in Chomsky normal form.

**Conversion to Chomsky normal form**:

Theorem: For every CFG, there is an equivalent grammar G in Chomsky Normal Form.

Proof:

Construction of grammar in CNF.

Step 1:

Eliminate null productions and unit productions.

Step 2:

Eliminate terminals on right hand side of productions as follows.

1. All the productions in P of the form A 🡪 a and A 🡪 BC are included.
2. Consider A 🡪 w1w2….wn will some terminal on right hand side. If wi is a

terminal say ai, add a new variable cai and cai 🡪 P. Repeat same for all terminals.

Step 3:

Restricting the number of variables on RHS as follows:

1. All the productions in P are added to P, if they are in the required form.
2. Consider A 🡪 A1A2A3 … Am, then we introduce new productions are,

A 🡪 A1C1

C1 🡪 A2C2

C2 ­🡪 A3C3

Cm-2 🡪 Am-1Cm-1

Example:

Convert the following CFG to Chomsky Normal Form (CNF):

S 🡪 aX | Yb

X 🡪 S | ∈

Y 🡪 bY | b

Solution:

Step 1 - Kill all ∈ productions:

By inspection, the only nullable nonterminal is X.

Delete all ∈ productions and add new productions, with all possible combinations of the nullable X removed.

The new CFG, without ∈ productions, is:

S 🡪 aX | a | Yb

X 🡪 S

Y 🡪 bY | b

Step 2 - Kill all unit productions:

The only unit production is X 🡪 S, where the S can be replaced with all S’s non-unit productions (i.e. aX, a, and Yb).

The new CFG, without unit productions, is:

S 🡪 aX | a | Yb

X 🡪 aX | a | Yb

Y 🡪 bY | b

Step 3 - Replace all mixed strings with solid nonterminals.

Create extra productions that produce one terminal, when doing the replacement.

The new CFG, with a RHS consisting of only solid nonterminals or one terminal is:

S 🡪 AX | YB | a

X 🡪 AX | YB | a

Y 🡪 BY | b

A 🡪 a

B 🡪 b

Step 4 - Shorten the strings of nonterminals to length 2.

All nonterminal strings on the RHS in the above CFG are already the required length, so the CFG is in CNF.

**6.2 Greibach Normal Form (GNF):**

A CFG G = (V, T, P, S) is said to be in GNF if every production is of the form A 🡪 aα, where a € T and α € V\*, i.e., α is a string of zero or more variables.

Definition: A production U € R is said to be in the form left recursion, if U : A 🡪 Aα for some A € V .

Left recursion in R can be eliminated by the following scheme:

• If A 🡪 Aα1|Aα2| . . . |Aαr|β1| β2| . . . | βs, then replace the above rules by

1. Z 🡪αi|αiZ, 1≤ i≤ r and
2. A 🡪 βi| βiZ, 1≤ i ≤ s

• If G = (V, T, P, S) is a CFG, then we can construct another CFG G1 = (V1, T, P1, S)

in Greibach Normal Form (GNF) such that L(G1) = L(G) − {∈}.

The stepwise algorithm is as follows:

1. Eliminate null productions, unit productions and useless symbols from the grammar G and then construct a G’ = (V’, T, P’, S) in Chomsky Normal Form (CNF) generating the language L(G’) = L(G) − {∈}.
2. Rename the variables like A1,A2, . . .An starting with S = A1.
3. Modify the rules in R’ so that if Ai 🡪 Aj γ € R’ then j > i.
4. Starting with A1 and proceeding to An this is done as follows:
5. Assume that productions have been modified so that for 1≤ i≤ k,

Ai 🡪 Aj γ € R’ then j > i.

1. If Ak 🡪 Ajγ is a production with j < k, generate a new set of productions

substituting for the Aj the body of each Aj production.

1. Repeating (b) at most k − 1 times we obtain rules of the form

Ak 🡪 Apγ, p≥ k

1. Replace rules Ak 🡪 Akγ by removing left-recursion as stated above.
2. Modify the Ai 🡪 Ajγ for i = n−1, n−2, ., 1 in desired form at the same time change the Z production rules.

Example: Convert the following grammar G into Greibach Normal Form (GNF).

S 🡪 XA|BB

B 🡪 b|SB

X 🡪 b

A 🡪 a

Solution:

To write the above grammar G into GNF, we shall follow the following steps:

1. Rewrite G in Chomsky Normal Form (CNF)

It is already in CNF.

2. Re-label the variables

S with A1

X with A2

A with A3

B with A4

After re-labeling the grammar looks like:

A1 🡪 A2A3|A4A4

A4 🡪 b|A1A4

A2 🡪 b

A3 🡪 a

3. Identify all productions which do not conform to any of the types listed below:

Ai 🡪 Ajxk such that j > i

Zi 🡪 Ajxk such that j ≤ n

Ai 🡪 axk such that xk € V\* and a € T

4. A4 🡪 A1A4 ................ identified

5. A4 🡪 A1A4|b.

To eliminate A1 we will use the substitution rule A1 🡪 A2A3|A4A4.

Therefore, we have A4 🡪 A2A3A4|A4A4A4|b

The above two productions still do not conform to any of the types in step 3.

Substituting for A2 🡪 b

A4 🡪 bA3A4|A4A4A4|b

Now we have to remove left recursive production A4 🡪 A4A4A4

A4 🡪 bA3A4|b|bA3A4Z|bZ

Z 🡪 A4A4|A4A4Z

6. At this stage our grammar now looks like

A1 🡪 A2A3|A4A4

A4 🡪 bA3A4|b|bA3A4Z|bZ

Z 🡪 A4A4|A4A4Z

A2 🡪 b

A3 🡪 a

All rules now conform to one of the types in step 3.But the grammar is still not in Greibach Normal Form.

7. All productions for A2,A3 and A4 are in GNF

for A1 🡪 A2A3|A4A4

Substitute for A2 and A4 to convert it to GNF

A1 🡪 bA3|bA3A4A4|bA4|bA3A4ZA4|bZA4

for Z 🡪 A4A4|A4A4Z

Substitute for A4 to convert it to GNF

Z 🡪 bA3A4A4|bA4|bA3A4ZA4|bZA4|bA3A4A4Z|bA4Z|bA3A4ZA4Z|bZA4Z

8. Finally the grammar in GNF is

A1 🡪 bA3|bA3A4A4|bA4|bA3A4ZA4|bZA4

A4 🡪 bA3A4|b|bA3A4Z|bZ

Z 🡪 bA3A4A4|bA4|bA3A4ZA4|bZA4|bA3A4A4Z|bA4Z|bA3A4ZA4Z|bZA4Z

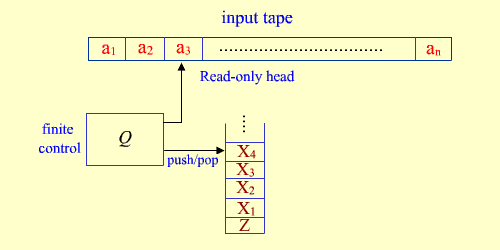
A2 🡪 b

A3 🡪 a

**UNIT - IV**

**Push Down Automata (PDA)**

* Just as the regular sets have an equivalent automaton the finite automaton, the context free grammars has their machine counterpart – the push down automation.
* The deterministic version of PDA accepts only a subset of all CFL’s where as non-deterministic version allows all CFL’s. The PDA will have an input tape, a finite control, and a stack.

****

**Formal Definition:**

* A pushdown automaton is a ε-NFA with a pushdown stack (last-in, first-out stack).
* Pushdown automata define exactly the context-free languages. There are seven components to a PDA P = (Q, Σ, Γ, δ, q0, Z0, F),

Where

* 1. Q is a finite set of states.
  2. Σ is a finite set of input symbols (the input alphabet).
  3. Γ is a finite set of stack symbols (the stack alphabet).
  4. δ is a transition function from (Q × (Σ ∪ {ε}) ∪ Γ) to subsets of (Q × Γ\*):
     + Suppose δ(q, *a*, X) contains (p, γ). Then whenever P is in state q, looking at the input symbol *a* with X on top of the stack, P may go into state q, move to the next input symbol, and replace X on top of the stack by the string γ.
     + The second component, *a*, may be ε in which case P makes the move without looking at the input symbol and does not move to the next input symbol.
     + Note that P is nondeterministic so there may be more than one pair in δ(q, *a*, X).
  5. q0 is the start state.
  6. Z0 is the start stack symbol.
  7. F is the set of final (accepting) states.

## Instantaneous Descriptions:

* We can represent a configuration of the PDA P above by a triple (q, *w*, γ) where:
  + q is the state of the finite-state control.
  + *w* is the string of remaining input symbols.
  + γ is the string of symbols on the stack. If γ = XYZ, then X is on top of the stack.
* Suppose δ(q, *a*, X) contains (p, α). Then to represent a single move of P we write

(q, *aw*, Xβ) |– (p, *w*, αβ) for all strings *w* in Σ\* and β in Γ\*.

Note that *a* may be empty.

**Language accepted by a PDA P:**

* A PDA P = (Q, Σ, Γ, δ, q0, Z0, F) can define a language two ways.
* **Acceptance by final state**: P can accept an input string w by reading all of it during a sequence of moves and entering a final state.
  + Formally, we define L(P), the language accepted by P by final state, to be the set of input strings w such that P can go from its initial ID (q0, w, Z0) in a sequence of zero or more moves to an accepting ID of the form (q, ε, α) where q is a final state and α is any stack string (perhaps empty).
* **Acceptance by empty (null) stack**: P can accept an input string by reading all of it and emptying its stack.
  + Formally, we define N(P), the language accepted by P by empty stack, to be the set of input strings w such that P can go from its initial ID (q0, w, Z0) in a sequence of zero or more moves to an accepting ID of the form (q, ε, ε) for any state q.
  + Note that the final states of a PDA accepting by empty stack are irrelevant.
* These two modes of acceptance are equivalent. That is, L has a PDA that accepts it by final state iff L has a PDA that accepts it by empty stack.

## **2.DETERMINISTIC PUSH DOWN AUTOMATA (DPDA)**

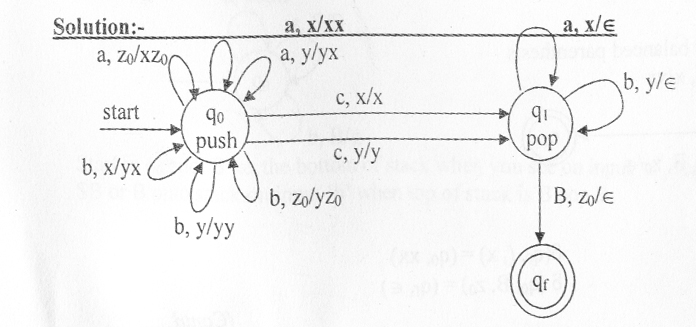
* A PDA is deterministic (DPDA) if there is never a choice for a next move in any instantaneous description.
* A PDA P=(Q, Σ, Γ, δ, q0, Z0, F) is deterministic if:
  1. For each q in Q and Z in Γ , whenever δ(q, ε, z) is nonempty, then δ(*q*, *a*, *z*) is empty for all a in Σ.
  2. For no q in Q, Z in Γ and a in ΣU{ε} does δ(*q*, a, z) contain more than one element.

## Note: For finite automata, the deterministic and non-deterministic models were equivalent with respect to the languages accepted. The same is not true for PDAs. DPDAs accept only a subset of languages accepted NPDAs. That is NPDA is more powerful than DPDA.

* If L is a CFL, then there exists a PDA, P that accepts L.

Examples:

1. Give a PDA for the language L={wcwR | w∈(a+b)+}



The transitions are δ(q0, a, z0)=(q0,xz0)

i.e In state q0 if input symbol is ‘a’ and symbol on top of stack is z0 then remain in state q0 and push ‘x’ on to the stack.

δ(q0, a, x)=(q0,xx)

δ(q0, a, y)=(q0,xy)

δ(q0, b, z0)=(q0,yz0)

δ(q0, b, x)=(q0,yx)

δ(q0, b, y)=(q0,yy)

δ(q0, c, x)=(q1,x)

δ(q0, c, y)=(q1,y)

δ(q1, a, x)=(q1, ε)

δ(q1, b, y)=(q1, ε)

δ(q1, B, z0)=(qf, ε) (B – blank space).

This is Deterministic Push Down Automata.

2. Design PDA for the following language L={0n12n|n≥1}

Solution: PDA P=(Q, Σ, Γ, δ, q0, Z0, F)

PDA P=({q0,q1,q2},{a,b},{a,z0}, δ,q0, Z0, ∅)

The transitions are

δ(q0, a, z0)={(q1,az0)}

δ(q1, a, a)={(q1,aa)}

δ(q1, b, a)={(q2,a)}

δ(q2, b, a)={(q1, ε)}

δ(q1, ε, z0)={(q1, ε)}

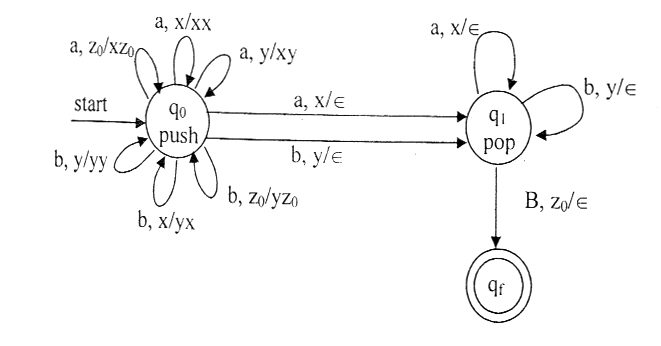
**3. NON-DETERMINISTIC PUSH DOWN AUTOMATA (NDPDA)**

A PDA is called as non-deterministic, if derivation generates more than one move in the designing of particular task.

Example:

1. Give PDA for L={wwR | w∈(a+b)+}

Solution:



The transitions are

δ(q0, a, z0)={(q0,xz0)}

δ(q0, a, x)={(q0,xx),(q1, ε)}

i.e In state q0 on input symbol ‘a’ if top of stack is ‘x’ try two possibilities,

1. Push ‘x’ on to the top on the assumption that still we have to reach the middle of the string 2. Pop ‘x’ on the assumption that we are reading the first symbol of the second half of the input string and go to the state q1.

δ(q0, a, y)={(q0,xy)}

δ(q0, b, z0)={(q0,yz0)}

δ(q0, b, x)={(q0,yx)}

δ(q0, b, y)={(q0,yy), (q1, ε)}

Explanation for this transition also is same as above explanation.

δ(q1, a, x)={(q1, ε)}

δ(q1, b, y)={(q1, ε)}

δ(q1, B, z0)={(qf, ε)} (B – blank space).

This is Non-Deterministic Push Down Automata.

2. Design NDPDA for the following L={anbn|n≥0}

Solution: PDA P=({q0,q1,q2,q3},{a,b},{R,B,G}, δ,q0,R, ∅)

The transitions are

δ(q0, a, R)={(q1,BR) ,(q3, ε)}

δ(q0, ε, R)={(q3, ε)}

δ(q1, a, B)={(q1,BB)}

δ(q1, b, B)={(q2, ε)}

δ(q2, b, B)={(q2, ε)}

δ(q2, ε, R)={(q3, ε)}

## **4. From a CFG to an equivalent PDA**

* Given a CFG *G*, we can construct a PDA *P* such that N(*P*) = L(*G*).
* The PDA will simulate leftmost derivations of G.
* Algorithm to construct a PDA for a CFG
  + Input: a CFG *G* = (V, T, P, S).
  + Output: a PDA *P* such that N(*P*) = L(*G*).
  + Method: Let *P* = ({q}, T, V ∪ T, δ, q, S) where
    1. δ(*q*, ε, *A*) = {(*q*, β) | *A* → β is in Q } for each nonterminal *A* in V.
    2. δ(*q*, *a*, *a*) = {(*q*, ε)} for each terminal *a* in *T*.
* For a given input string *w*, the PDA simulates a leftmost derivation for *w* in *G*.
* We can prove that N(*P*) = L(*G*) by showing that *w* is in N(*P*) iff *w* is in L(*G*):
  + If part: If *w* is in L(*G*), then there is a leftmost derivation
  + S = γ1 ⇒ γ2 ⇒ ... ⇒ γn = w

We show by induction on *i* that *P* simulates this leftmost derivation by the sequence of moves

(*q*, *w*, S) |–\* (*q*, *y*i, αi)

such that if γ*i* = *xi*α*i*, then *xiyi* = *w*.

* + Only-if part: If (*q*, *x*, A) |–\* (*q*, ε, ε), then A ⇒\* *x*.

We can prove this statement by induction on the number of moves made by *P*.

Example:

1. Construct PDA that accepts the language generated by grammar

S🡪aSbb|a

Solution: We first transform the grammar into GNF, changing the productions to

S🡪aSA|a

A🡪bB

B🡪b

The corresponding automata will have three states {q0,q1,q2}, with initial state q0 and final state q2. First, the start symbol S is put on the stack by

δ(q0, ε, z0)={(q1, sz0)}

The production S🡪aSA will be simulated in the PDA by removing S from the stack and replacing it with SA, while reading ‘a’ from the input. Similarly, the rule S🡪a should cause the PDA to read an ‘a’ while simply removing S. Thus, the two productions are represented in the PDA by

δ(q1, a, S)={(q1,SA),(q1, ε)}

In an analogous manner, the other productions give

δ(q1, b, A)={(q1,B)}

δ(q1, b, B)={(q1, ε)}

δ(q1, ε, z)={(q2, ε)}

## **5. From a PDA to an equivalent CFG**

* Given a PDA *P*, we can construct a CFG *G* such that L(*G*) = N(*P*).
* The basic idea of the proof is to generate the strings that cause *P* to go from state *q* to state *p*, popping a symbol X off the stack, by a nonterminal of the form [*q*X*p*].
* Algorithm to construct a CFG for a PDA
  + Input: a PDA *P* = (Q, Σ, Γ, δ, q0, Z0, F).
  + Output: a CFG *G* = (V, Σ, R, S) such that L(*G*) = N(*P*).
  + Method:
    1. Let the nonterminal S be the start symbol of *G*. The other nonterminals in V will be symbols of the form [*p*X*q*] where *p* and *q* are states in Q, and X is a stack symbol in Γ.
    2. The set of productions R is constructed as follows:
       - For all states *p*, R has the production S → [*q*0Z0*p*].
       - If δ(*q*, *a*, X) contains (*r*, Y1Y2 … Y*k*), then R has the productions

[*q*X*rk*] → *a*[*r*Y1*r*1] [*r*1Y2*r*2] … [*rk*-1Y*krk*]

for all lists of states *r*1, *r*2, … , *rk*.

* + We can prove that [*q*X*p*] ⇒\* *w* iff (*q*, *w*, X) |–\* (*p*, ε, ε).
  + From this, we have [*q0*Z0*p*] ⇒\* *w* iff (*q0*, *w*, Z0) |–\* (*p*, ε, ε), so we can conclude L(*G*) = N(*P*).

Example: 1. Let M =({q0­,q1},{0,1},{X,z0} ,q0,z0,) Where  is given by

(q0,0,z0)={(q0,Xz0)

(q1,1,X)={(q1,)

(q0,0,X)={(q0,XX0)}, (q1, ,X)={(q0,)}

(q0,1,X)={(q1,)}, (q1, ,Z0)={(q0,)}

To construct a CFG G= (V,T, P, S) generating N (M)

Let V={S,[q0,X,q0],[q0,X,q1],[q1,X,q0],[q1,X,q1],[q0,z0,q0],[q0,z0,q1],[q1,z0,q0], [q1,z0,q1]} and T={0,1}.

To construct the set of productions easily, we must realize that some variable may not appear in any derivation starting from the symbol S. thus we can save some effort if we start with the productions for S, and then add productions only for those variable that appear or the right of some production already in the set . The productions for S are

S🡪[q0,z0,q0]/ [q0,z0,q1]

Next we add production for the variables [q0,z0,q0]

These are [q0,z0,q0]🡪0[q0,X,q0][q0,z0,q0]

[q0,z0,q0]🡪0[q0,X,q1][q1,z0,q0]

There productions are required by

( q0,0,z0)={ (q0,Xz0)}

Next the productions for [q0,z0,q1] are

[q0,z0,q1]🡪0[q0,X,q0][q0,z0,q1]

[q0,z0,q1]🡪0[q0,X,q1][q1,z0,q1]

There are also required by ( q0,0,z0)={ (q0,Xz0)}

1) [q0,X,q0]🡪0[q0,X,q0] [q0,X,q0]

[q0,X,q0]🡪0[q0,X,q1] [q1,X,q0]

[q0,X,q1]🡪0[q0,X,q0] [q0,X,q1]

[q0,X,q1]🡪0[q0,X,q1] [q1,X,q1] since(q0,0,X)={( q0,XX)}

2) [q0,X,q1]🡪1 since (q0,1,X)={( q1,)}

3) [q1,z0,q1]🡪 since (q1,,z0)={( q1,)}

4) [q1,X,q1]🡪  since (q1,,X)={(q1,)}

5) [q1,X,q1]🡪 1 since (q1,1 ,X)={(q1,)}

Resulting productions are

S🡪[q0,z0,q1], [q1,z0,q1]🡪, [q1,X,q1]🡪

[q0,z0,q1]🡪0[q0,X,q1] [q1,z0,q1], [q1,X,q1]🡪1

[q1,X,q1]🡪0[q0,X,q1] [q1,X,q1], [q0,X,q1]🡪1

## **6. Closure Properties of CFL's:**

* The context-free languages are closed under
  + substitution
    - Let Σ be an alphabet and let L*a* be a language for each symbol *a* in Σ. These languages define a substitution *s* on Σ.
    - If *w* = *a*1*a*2 ... *an* is a string in Σ\*, then *s*(*w*) = { *x*1*x*2 ... *xn* | *xi* is a string in *s*(*ai*) for 1 ≤ *i* ≤ *n* }.
    - If L is a language, *s*(L) = { *s*(*w*) | *w* is in L }.
    - If L is a CFL over Σ and *s*(*a*) is a CFL for each *a* in Σ, then *s*(L) is a CFL.
  + union
  + concatenation
  + Kleene star
  + homomorphism
  + reversal
  + intersection with a regular set
  + inverse homomorphism

## **7. Non-closure Properties of CFL's:**

* The context-free languages are not closed under
  + intersection
    - L1 = { *anbnci* | *n, i* ≥ 0 } and L2 = { *aibncn* | *n, i* ≥ 0 } are CFL's. But L = L1 ∩ L2 = { *anbncn* | *n* ≥ 0 } is not a CFL.
  + complement
    - Suppose comp(L) is context free if L is context free. Since L1 ∩ L2 = comp(comp(L1) ∪ comp(L2)), this would imply the CFL's are closed under intersection.
  + difference
    - Suppose L1 – L2 is a context free if L1 and L2 are context free. If L is a CFL over Σ, then comp(L) = Σ\* - L would be context free.

## **8. Pumping Lemma for CFL's:**

Pumping Lemma for CFL’s is used to show that certain languages are non context free. There are three forms of pumping lemma.

**1. Standard form of pumping lemma**: For every non finite context-free language L, there exists a constant *n* that depends on L such that for all *z* in L with |*z*| ≥ *n*, we can write *z* as *uvwxy* where

* 1. *vx* ≠ ε,
  2. |*vwx*| ≤ *n*, and
  3. for all *i* ≥ 0, the string *uviwxiy* is in L.

One important use of the pumping lemma is to prove certain languages are not context free.

**2. Strong form of pumping lemma (Ogden’s Lemma):** Let L is an infinite CFL. Then there is a constant n such that if z is any word in L, and we mark any n or more positions of z “distinguished”, then we can write z=uvwxy such that

1. v and x together have at least one distinguished positions,
2. vwx has atmost n distinguished positions, and
3. for all i≥0, uviwxiy is in L.

**3. Weak form of pumping lemma:**Let L is an infinite CFL. When we pump the length of strings are

|uvwxy|=|uwy|+|vx|

|uv2wx2y|=|uwy|+2|vx|

...................................

|uviwxiy|=|uwy|+i|vx|.

When we pump the lengths are in arithmetic progression.

Example:

1. The language L = { *anbncn* | *n* ≥ 0 } is not context free.

Solution:

The proof will be by contradiction. Assume L is context free. Then by the pumping lemma there is a constant *n* associated with L such that for all *z* in L with |*z*| ≥ *n*, *z* can be written as *uvwxy* such that

* + 1. *vx* ≠ ε,
    2. |*vwx*| ≤ *n*, and
    3. for all *i* ≥ 0, the string *uviwxiy* is in L.

Consider the string *z* = *anbncn*.

From condition (2), *vwx* cannot contain both *a*'s and *c*'s.

* 1. Two cases arise:
     1. *vwx* has no *c*'s. But then *uwy* cannot be in L since at least one of *v* or *x* is nonempty.
     2. *vwx* has no *a*'s. Again, *uwy* cannot be in L.
  2. In both cases we have a contradiction, so we must conclude L cannot be context free.

**UNIT – V**

**TURING MACHINES**

* A Turing Machines is an automaton whose temporary storage is a tape. This tape is divided into cells, each of which is capable of holding one symbol. Associated with the tape is a read-write, read that can travel right or left on the tape and that can read and write a single symbol on each move.
* A diagram giving an intuitive visualization of a Turing Machine is

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| a1 | a2 | ... | ai | ... | an | B | ... |

|  |
| --- |
| Finite Control |

* A Turing machine is a seven tuple notation i.e M= (Q, Σ, Γ, δ, q0, B, F)
  1. Q is the finite set of states of the finite control.
  2. Σ is the finite set of input symbols.
  3. Γ is the set of tape symbols; Σ is a subset of Γ.
  4. δ is the transition function. It maps (Q × Γ) to subsets of (Q × Γ × {L,R}). If (p, Y, D) is in δ(q, X) and M is in state q reading the symbol X on the input tape, then M can
     + go from state q to state p,
     + replace the symbol X on the input tape by the symbol Y, and
     + move its input head one square in the direction D where D can be either L (for left) or R (for right).

M is deterministic if there is at most one element in δ(q, X) for any state q and tape symbol X.

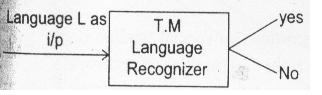
* 1. q0 is the start state.
  2. B is the blank symbol. B is in Γ but not in Σ.
  3. F, a subset of Q, is the set of final accepting states. We assume there are no transitions from a final state so that when M enters a final state it halts.

**Language Acceptance**:

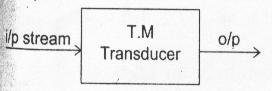
* L(M), the language accepted by M, is the set of strings w in Σ\* such that q0w |–\* αpβ for some state p in F.

**Functions of Turing Machine**

1. Turing machine as a language recognizer**.**

****

1. Turing Machine as a Transducer

****

**Halt state:**

It is a state from which no further transitions can be seen.

1. **Halt final state:**

If at all string is accepted,T.M goes to halt final state.

1. **Halt non final state:**

Turing Machine may goes to Halt non final state,if the string is rejected,when it reads invalid string.

Example:

1. Construct a TM is accept the languages {0n1n|n>1}

Solution:

The sample string is 0011 for n=2

This string is stored on the tape

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ... | B | 0 | 0 | 1 | 1 | B | ... |

First the finite control read the symbol ‘0’ and replaced by ‘x’. Now it will verify for one. Until encounter the symbol ‘1’ the tape will move towards right by reading one input symbol at a time and all the zeros are replaced by 0’s again. When the 1st ‘1’ is encountered, this is replaced by symbol ‘y’, now again head movement is towards left side unit ‘1’ encounter the ‘x’. This process is repeated.

δ(q0,B)=(q0,B,R)

δ(q0,0)=(q1,X,R)

δ(q1,0)=(q1,0,R)

δ(q1,1)=(q2,Y,L)

δ(q2,0)=(q2,0,L)

δ(q2,X)=(q0,X,R)

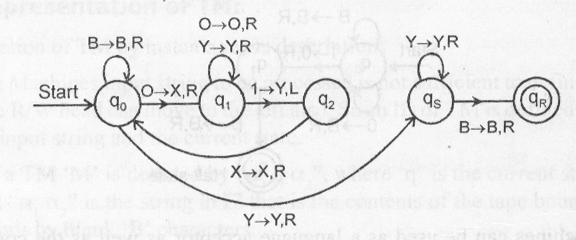
δ(q1,Y)=(q1,Y,R)

δ(q2,Y)=(q2,Y,L)

δ(q0,Y)=(q3,Y,R)

δ(q3,Y)=(q3,Y,R)

δ(q3,B)=(qf,B,R)



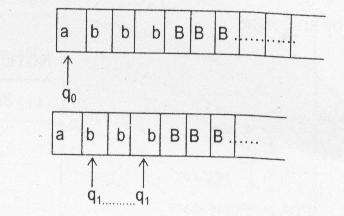
Let us consider a language

L={abn/n0}

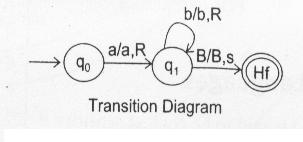
The possible strings in this language are

L={a,ab,abb,……}

If we consider the T.M for this language

****



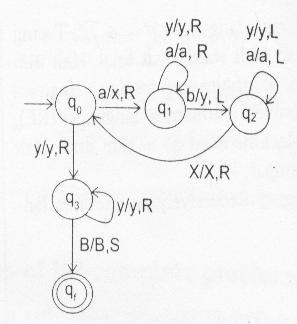
****

**Note:**

By reading an i/p symbol, Turing Machine may go to either left or right or stays at that state only.

Let us consider the language L=an bn where n1.

To construct the Turing Machine for this language,the transition diagram is

****

For the input string w=aaabbb

Instantaneous description is

1st 

2nd 

3rd 

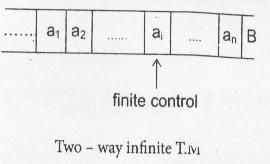
The Transaction table is

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | X | Y | B |
| → q0 | (q1,X,R) | - | - | (q3,Y,R) | - |
| q1 | (q1,a,R) | (q2,Y,L) | - | (q1,Y,R) | - |
| q2 | (q2,a,L) | - | (q0,X,R) | (q2,Y,L) | - |
| q3 | - | - | - | (q3,Y,R) | (qf,B,S) |
| \*qf | - | - | - | - | - |

**2. MODIFICATIONS FOR TURING MACHINES**

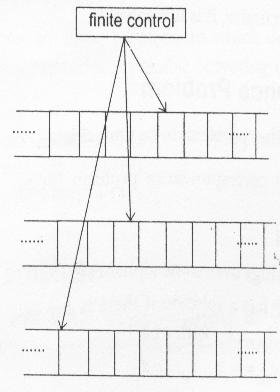
**1. Two-way Infinite Turing Machine**

* Language L is recognized by a Turing Machine with a two-way infinite tape if and only if it is recognized by a Turing Machine with a one-way infinite tape.

****

**2. Multi-tape Turing Machine**

* If a language L is accepted by a multi-tape Turing Machine,it is accepted by a single tape Machine.

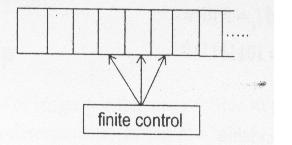
****

**3. Non-deterministic Turing Machine**

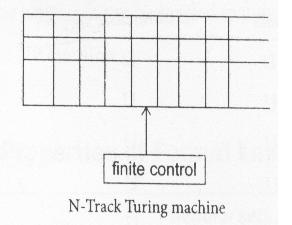
* If L is accepted by a non-deterministic Turing Machine M1,Then L is accepted by some deterministic Turing Machine,M2.

**4. Multi-head Turing Machine**

* An N-head Turing Machine has some fixed number,N,of heads.The heads are numbered I through N,and a move of the Turing Machine depends on the state and on the symbol scanned by each head.

****

**5. Multi-track Turing Machine**

****

**6. Turing Machine with Stay option**

* In these,the read-write head can stay at the current position,upon reading on i/p symbol without moving left or right.

**3. CONTEXT SENSITIVE GRAMMAR**

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

α→β

Where α,β  and 

**Note:** The length of successive sentential forms can never decrease.

**Context Sensitive Language:** A language is said to be Context Sensitive if there exist a Context Sensitive grammar G, such that L=L(G).

**Note:** Context Sensitive grammar can never generate a language containing the empty string.

Some examples of context sensitive languages

1. L={an bn cn/n1}
2. L={an bn c2n/n1}
3. L={an bm c dm/n1,m1}
4. L={ww/w{a,b}}

Context sensitive Languages are closed under the operations

1. ˜Āᄀn
2. Concatination
3. Positive clouser
4. -free homomorphism
5. Inverse homomorphism
6. Intersection with Regular sets
7. Substitution
8. Reversal
9. Intersection

**Note:**

1. Whether Context sensitive Languages are closed under complementation or not is an open question.

* If language ‘L’ is a context sensitive language and ‘w’ is a string,we can find whether

wL or not algorithmetically i.e .,there is a ‘membership algorithm’ for context sensitive languages.

**4. LINEAR BOUNDED AUTOMATION**

A Linear Bounded Automation (LBA) M accepts a string w if after starting at the initial state with R/W head reading the left-end-marker, M halts over the right-end-marker in a final state.

Otherwise ’w’ is rejected.

* A Linear Bounded Automation (LBA) is a nondeterministic Turning Machine(TM) satisfying the following two conditions.

1. Its i/p alphabet includes two special symbols $,the left and right end marker respectively.
2. The LBA has no more moves from or right from $,not it may print another symbol over $.An LBA will be defined as M={Q,

Where & $ are symbols in ,the left & right side markers.

**Note:**

1. If L is CSL,then L is accepted by some LBA.
2. If L=(M) for LBA, M=(Then L-{} is a CSL.
3. Every CSL is a recursive but converse is not true.

**5. UNDECIDABILITY**

**Recursive Languages:**

* A language L over the alphabet is called recursive language if there exist a Turing Machine M that accepts every word in L and rejects every word in L(The complement of L).

**Recursively Enumerable Languages:**

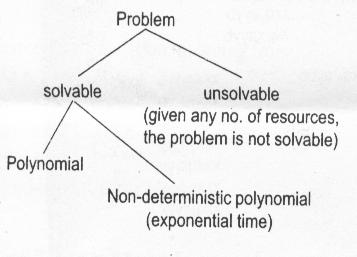
* A language L over the alphabet is called recursively enumerable if there exist a Turing Machine T that accepts every word in L and either rejects or loops for every word in the language L.
* In the case of recursive language Turing Machine must go to Halt state(Halt final,Halt non final) i.e.,no chance of infinite loop for any string.
* In the case of Recursively enumerable languages (RE),  Turing Machine must go to Halt final,Halt non final,Infinite loop.
* Recursive languages recursively enumerable languages.

**Note:**

1. Recursive languages are closed under complementation.
2. Recursively enumerable languages are not closed under complementation.
3. If language L is Recursively enumerable and L1is also Recursively enumerable then the language L should be Recursively language.

**6. UNDECIDABILITY AND NP COMPLETENESS**

For any problem P if algorithm is not available,then the problem P is undecidable.

****

**Note:**

1. If Problem is recursive, it is solvable.
2. If Problem is not recursive, it is not solvable.

**7. POST CORRESPONDENCE PROBLEM**

It is a tool in establishing other problems to be undecidable.

* An instance of post correspondence problem (PCP) consists of two lists.

A=w1,w2,…….wk and

B=x1,x2,………xk of strings over some alphabet .

* This instance of PCP has a solution if there is any sequence of integers i1,i2,….im with m1

Such that wi1,wi2,…….wim==xi1,xi2,…..xim

The sequence i1,i2,….im,ia a solution to this instance of PCP.

Let us consider the following instance of PCP.

|  |  |  |
| --- | --- | --- |
|  | **List A** | **List B** |
| i | Wi | Xi |
| 1 | 1 | 111 |
| 2 | 10111 | 10 |
| 3 | 10 | 0 |

Let M=4, I1=2,i2=1,i3=1 and i4=3 then w2w1w1w3=x2x1x1x3=101111110

**In the following table:** D-Decidable,U-Undecidable,?-open question,T-Trivially decidable.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Question | Regular sets | DCFL’s | CFL’s | CSL’s | Recursive sets | Recursively enumerable sets |
| 1. Member ship problem? | D | D | D | D | D | D |
| 1. Emptyness problem? | D | D | D | U | U | U |
| 1. Ompleteness problem is L=? | D | D | D | U | U | U |
| 1. Equality Problem? | D | ? | U | U | U | U |
| 1. Subset problem is L1 L2? | D | U | U | U | U | U |
| 1. Is L Regular? | T | D | U | U | U | U |
| 1. (7) Is the intersection of 2 languages, a lang,of the same type | T | U | U | T | T | T |
| 1. Is the complement of a lang,also a lang of the same type | T | T | U | ? | T | U |
| 1. If L is finite or infinite | D | D | D | U | U | U |

**8. P-class, NP-class, NP-Hard, NP-complete**

**P-class problem**

P class problems are those problems to which deterministic polynomial time algorithm is possible (covering every possibility).

Example: Linear search  O(n)

Bubble sort  O (n2)

Selection sort  O (n2)

**NP-class problem**

NP class problems are those problems to which non-deterministic polynomial time algorithm is possible.

**NP-Hard problem**

If there is a language L such that every language L is NP,can be polynomially reducible to L and we can’t prove that L is in NP,then L is said to be **np-hard problem.** i.e.,’L’ is polynomially reduced to all NP problems.

**NP-complete** **problem**

If we can prove that L is in NP and everyNP problem can be polynomially reducible to L then L is said to be NP-Complete problem.

**P verse NP**

NP is the class of languages that are solvable in polynomial time on a non-deterministic Turing Machine.

(or)

Equivalently,it is the class of language where the membership in the languages can be verified in polynomial time.

The P=NP question.

We are unable to prove the existence of single language in NP that is not in P.

P=NP is one of the greatest unsolved problem.

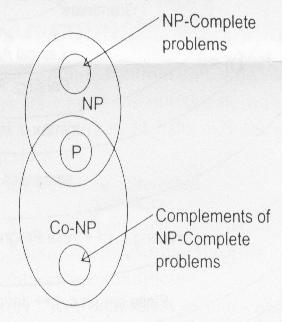
1. If P1 is NP-complete and there is a polynomial-time reduction of P1toP2 then P2 is NP-complete.
2. If some NP-complete problem is in P then P=NP.

**Examples of NP-complete problems**

1. Boolean satisfiability problem(B-SAT) is NP complete.
2. C-SAT,3-SAT
3. Travelling sales man problem.
4. Vertex cover problem.
5. Chromatic number problem.
6. The partition problem.
7. A k-clique in a graph G.
8. The edge cover problem.

**Note:**

1. P is closed under complementation,but it is not known whether NP is closed under complementation.

****

1. If P=NP then P,NP & complement NP all are same.
2. NP=Co-NP iff there is some NP-complete problem whose complement is in NP.

**9. Clousre Properties of Formal Languages**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Regular sets | DCFL’S | CFL’S | CSL’S | Recursive sets | Recursively enumerable sets |
| 1. Union | Y | N | Y | Y | Y | Y |
| 2. Concatenation | Y | N | Y | Y | Y | Y |
| 3. Kleene closure | Y | N | Y | N | N | Y |
| 4. Intersection | Y | N | N | Y | Y | Y |
| 5. Complementation | Y | Y | N | ? | Y | N |
| 6. Homomorphism | Y | N | Y | N | N | Y |
| 7. Inverse Homomorphism | Y | Y | Y | Y | Y | Y |
| 8. Reversal | Y | N | Y | Y | Y | Y |
| 9. Substitution | Y | N | Y | Y | N | Y |
| 10. Intersection with regular sets | Y | Y | Y | Y | Y | Y |

