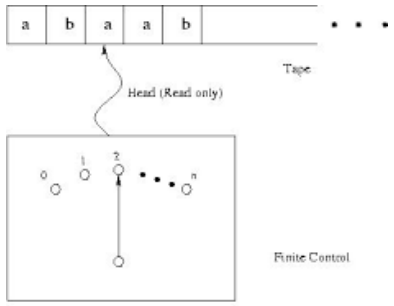
FINITE AUTOMATA

- A Finite Automata is the mathematical model of a digital computer. Finite Automata are used as string or language acceptors. They are mainly used in pattern matching tools like LEX and Text editors.
- The Finite State System represents a mathematical model of a system with certain input. The model finally gives a certain output. The output given to the machine is processed by various states. These states are called intermediate states.
- A good example of finite state systems is the control mechanism of an elevator. This mechanism only remembers the current floor number pressed, it does not remember all the previously pressed numbers.
- The finite state systems are useful in design of text editors, lexical analyzers and natural language processing. The word “automaton” is singular and “automata” is plural.
- An automaton in which the output depends only on the input is called an automaton without memory.
- An automaton in which the output depends on the input and state is called as automation with memory.

1. FINITE AUTOMATON MODEL

- Informally, a FA – Finite Automata is a simple machine that reads an input string -- one symbol at a time -- and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far.
- The Finite Automata can be represented as,



Finite Automaton

i) **Input Tape:** Input tape is a linear tape having some cells which can hold an input symbol from ∑.

ii) **Finite Control:** It indicates the current state and decides the next state on receiving a particular input from the input tape. The tape reader reads the cells one by one from left to right and at any instance only one input symbol is read. The reading head examines read symbol and the head moves to the right side with or without changing the state. When the entire string is read and if finite control is in final state then the string is accepted otherwise rejected. The finite automaton can be represented by a transition diagram in which the vertices represent the states and the edges represent transitions.

- 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.
- Finite Automata can be classified into two type:
  1. FA without output or Language Recognizers (e.g. DFA and NFA)
  2. FA with output or Transducers (e.g. Moore and Mealy machines)

Deterministic Finite Automata (DFA):

i. In Deterministic Finite Automata no input symbol causes to move more than one state or a state does not contain more than one transition from same input symbol.

ii. Each and every state has to consume all the input symbols present in ∑.

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

i.e M = (Q, Σ, δ, q0, F)

where

Q is finite set of states

Σ is finite input alphabet

δ is transition mapping function i.e Q x Σ □ Q q0 □ Q

Initial state

F □ Q Set of final states

(State) Transition diagram:

A state transition diagram or simply a transition diagram is a directed graph which can be constructed as follows:

1. For each state in Q there is a node.
2. There is a directed edge from node *q* to node *p* labeled *a* iff $\delta(q, a) = p$. (If there are several input symbols that cause a transition, the edge is labeled by the list of these symbols).
3. There is an arrow with no source into the start state.
4. Accepting states are indicated by double circle.

Transition Table:

It is basically a tabular representation of the transition function that takes two arguments (a state and a symbol) and returns a value (the "next state").

1. Rows correspond to states.
2. Columns correspond to input symbols.
3. Entries correspond to next states.
4. The start state is marked with an arrow.
5. The accept states are marked with a star (*).

Acceptance by an Automaton:

- A string "w" is said to be accepted by a finite automation M=(Q, Σ, δ, 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

Language Accepted or Recognized by a DFA:

- The language accepted or recognized by a DFA M is the set of all strings accepted by M , and is denoted by $L(M)$ i.e. $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.
- The notion of acceptance can also be made more precise by extending the transition function δ.

Extended transition function :

Extend $\delta : Q \times \Sigma \to Q$ (which is function on symbols) to a function on strings, i.e. $\delta^\wedge : Q \times \Sigma^* \to Q$.

That is, $\delta^\wedge (q, w)$ is the state the automation reaches when it starts from the state q and finish processing the string w. Formally, we can give an inductive definition as follows:

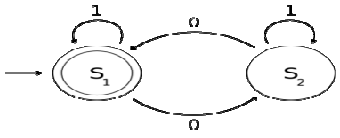
The language of the DFA M is the set of strings that can take the start state to one of the accepting states i.e.

$$L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$
$$= \{ w \in \Sigma^* \mid \delta^*(q_0, w) \in F \}$$

Examples:

1. Construct 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

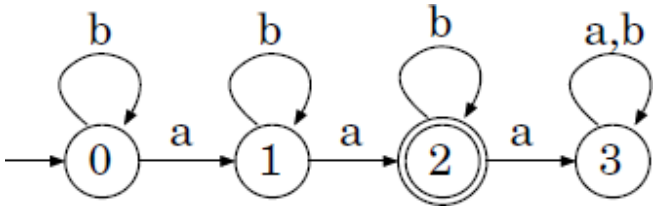
   Q = {*S*1, *S*2}, Σ = {0, 1}, *q0* = *S*1, *F* = {*S*1}, and

   δ is defined by the following state transition table

	0	**1**
□***S1**	*S*2	*S*1
S2	*S*1	*S*2

2. Construct DFA for all accepting strings over {a,b} that contains exactly 2 a's.

   Transition Diagram:



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

where

$Q = \{0,1,2,3\}, \Sigma = \{a,b\}, q0 = 0, F = \{2\}$, and

δ is defined by the following state transition table

Transition Table : □□Q x □ □ Q

	a	**b**
□***0***	1	0
1	2	1
****2***	*3*	*2*
3	*3*	*3*

Non-Deterministic Finite Automata (NFA):

- Non-Determinism is an important abstraction in computer science. Importance of Non-Determinism is found in the design of algorithms. For examples, there are many problems with efficient nondeterministic solutions but no known efficient deterministic solutions. (Travelling salesman, Hamiltonian cycle, clique, etc). Behaviour of a process is in a distributed system is also a good example of nondeterministic situation. Because the behaviour of a process might depend on some messages from other processes that might arrive at arbitrary times with arbitrary contents.

- It is easy to construct and comprehend an NFA than DFA for a given regular language. The concept of NFA can also be used in proving many theorems and results. Hence, it plays an important role in this subject.

- In the context of FA, Nondeterminism can be incorporated naturally. That is, an NFA is defined in the same way as the DFA but with the following two exceptions:

  - Multiple next states.
  - ∈- transitions.

- In NFA, one input symbol causes to move more than one state or a state may contain more the one transition from same input symbol.
- It is not compulsory that all the states have to consume all input symbols in ∑.
- A Non-Deterministic Finite Automata is represented by a 5 – tuple. M = (Q, Σ, δ, q0, F)

Where

Q is finite set of states

Σ is finite input alphabet

δ is transition mapping function i.e Q x Σ □ 2^Q q0 □ Q

Initial state

F □ Q Set of final states

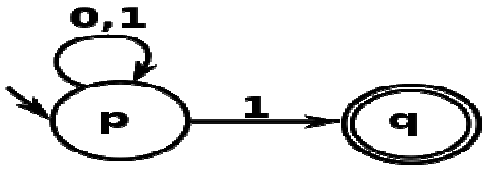
1.2.1 Acceptance by NFA:

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, Σ, δ, q0, F)

where

Q = {p, q}, Σ = {0, 1}, *q0* = p, *F* = {q}, and

δ is defined by the following state transition table

Transition Table: □□Q x □ □ 2^Q

	0	**1**
p	{p}	{p,q}
q	∅	∅

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 □ □2^Q

Note:

1) All DFAs are NFAs.
2) All NFAs are not being DFAs.

2. Equivalence between NFAs and DFAs

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. **Step 1**: Convert the given transition system into state transition table where each state corresponds to a row and each input symbol corresponds to a column.

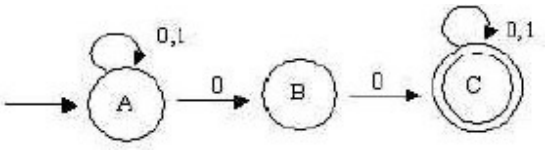
Step 2: Construct the succession table which lists subsets of states reachable from the set of initial states.

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.

Example:

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

}}

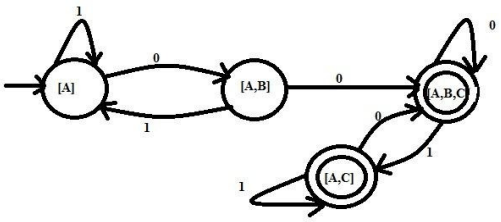
(These are final states as C is the final state of NFA and all these states contain C)

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.

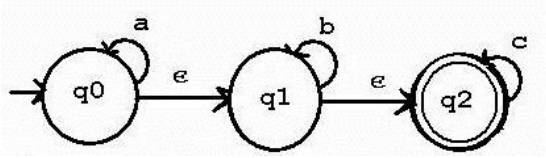
(4) Transition function is not changed.

(5) If NFA contains 'n' states, then the equivalent DFA contains maximum 2^n states. These state names are also subset of given 'n' states.

3. Finite Automation with □- Moves (Epsilon Transitions)

- In an NFA with □ -transition, the tape head doesn't do anything- it doesnot read and it doesnot move. However, the state of the automata can be changed - that is can go to zero, one or more states. This is written formally as $\delta(q,\in) = \{q_1, q_2, \cdots, q_k\}$ implying that the next state could by any one of $q_1, q_2, \cdots, q_k$ w/o consuming the next input symbol.
- An NFA is allowed to make a transition spontaneously, without receiving an input symbol. These □- NFA's can be converted to DFA's and NFA'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 (∑ □ {□}) □ 2^Q.

□- 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}

3.1 Conversion of an □- NFA to NFA without □- moves:

Refer Class notes

3.1 Conversion of an □- NFA to DFA:

Refer Class notes

4. Finite Automaton with Output

- The FA designed so far gives the output, for the given input string as accepted or rejected. It can say they give output which has only two possibilities. But if it is required to have more than two output like "yes" or "no". It can associate the output either with state or transition. If the output is associated with state then such a machine is called a Moore machine and if the output is associated with transition then it is called a Mealy machine.
- Moore and Mealy machines are special cases of DFA.
- Both the machines act as output generators rather than language acceptors.
- No need to define the final states, because no concept of final & dead state in Moore and Mealy machine.

Moore Machine:

- A Moore machine is a FA in which the output is associated with the state itself.

- A Moore machine is a 6- Tuple (Q, ∑, □ , □ , □ ,q0)

Where

Q: finite set of states.

∑: finite set of input symbols

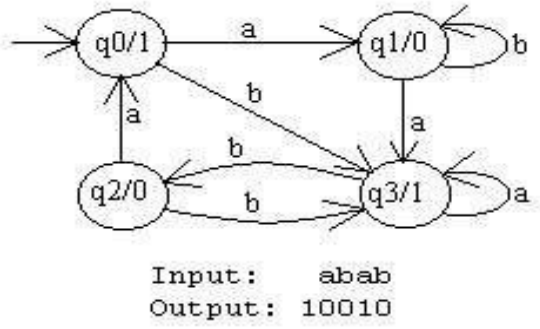
□ : finite set of output alphabet

□ : Transition function i.e. QX∑→Q

□ (output function): Q□ □ (□ is a function from Q to □) q0: initial state

Example: M=(Q, ∑, □ , □ , □ ,q0)

M=({q0,q1,q2,q3},{a,b},{0,1},□ , □ ,q0)



Present State	Next State a	Next State b	Output λ
-> q0	q 1	q3	1
q 1	q 3	q 1	0
q 2	q 0	q 3	0
q 3	q 3	q 2	1

The Moore machines in the same way as finite automata but the label in a state is composed both of the name of the state and the output character that the state produces. Run the string abab through the following machine and you will find that the output produced is 10010.

Note:

- Moore machine is DFA.
- In Moore machine the output is depends on state itself.
- In Moore machine, without reading any input symbol, it produces output alphabet.
- If the length of the input string is 'n' the Moore machine produces the output string of length 'n+1'.

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

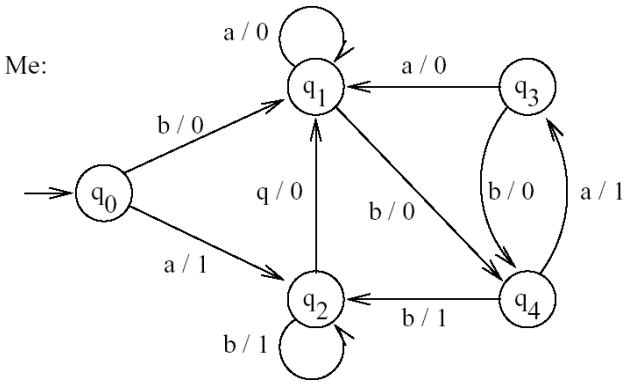
□ : Transition function i.e. QX∑→Q

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

q0: initial state

Example:

Me:



Present State	Next State		Output	
	a	b	a	b
->q0	q2	q1	1	0
q1	q1	q4	0	0
q2	q1	q2	0	1
q3	q1	q4	0	0
q4	q3	q2	1	1

Note:

- Mealy machine is DFA.
- In Mealy machine the output is depends on state and transition.
- 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'.

5. Equivalence of Moore and Mealy machines

Mealy machine equivalent to Moore machine:

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

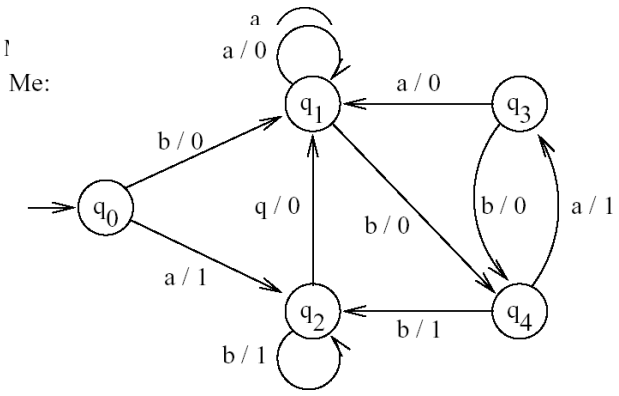
Proof :

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:



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.

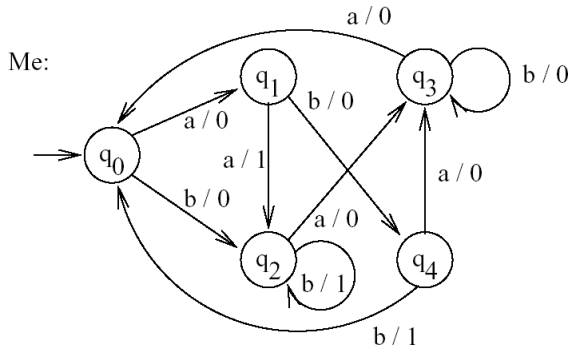
Only the first components of M2 ‘s states determine the moves made by

M2 .

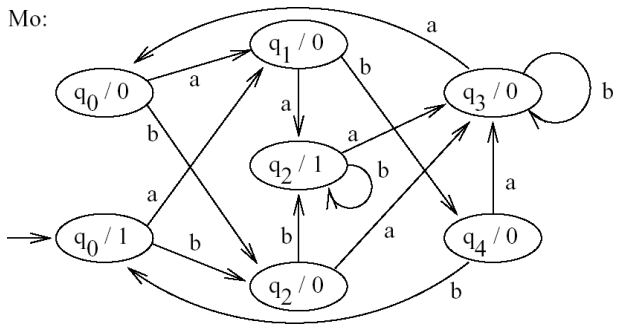
Every induction on ‘n’ shows that if M1 enters states q0 ,q1 ---- qn .on input a1 , a2,--- aN and emits output b1 ,b2 ------bn then M2 enters states [q0,b0], [q1,b1]---, [qn,bn] and emits outputs b0,b1..bn.

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

Solution:



Mo:

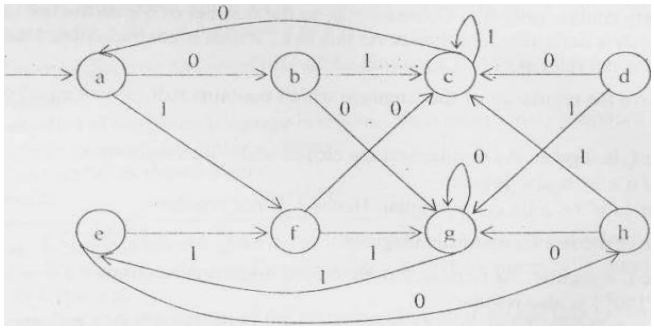


7. MINIMIZATION OF FINITE AUTOMATA

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

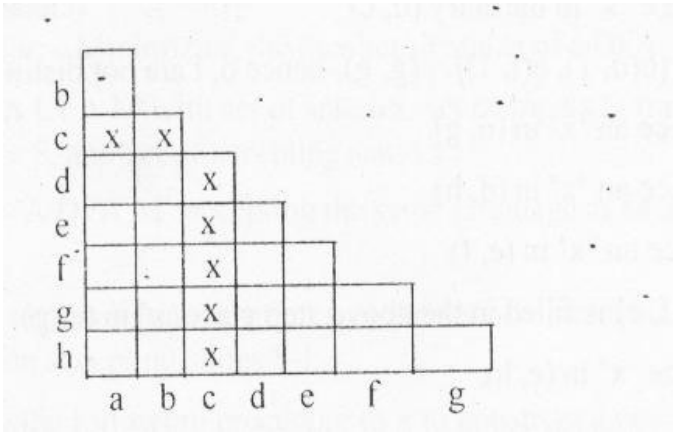
Example:

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



Solution: Refer Class notes

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.

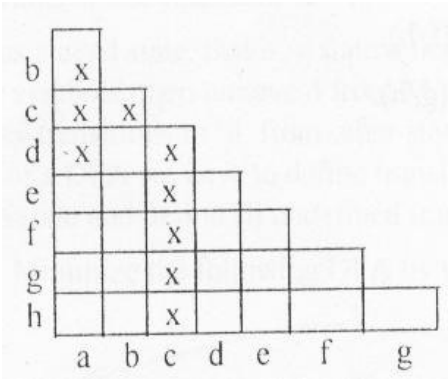


Next for each pair of states p and q that are not already known l0 be distinguishable we construct the pairs of states r = $\boldsymbol{\delta}$ (p. a) and s = $\boldsymbol{\delta}$ (q, a) for each input symbol a. If states r and s have been shown to be distinguishable by some string x. then p and q 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 (δ (b. 1), δ (a, 1)) = (c. f) already has an 'x'. Similarly the (a, d)- entry receives an 'x' since the entry (δ (a, 0), δ (d, 0) = (b, c) has an':<.".

Now the table is



(δ (a, 1), δ (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'.

(δ (a, 0), δ (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).

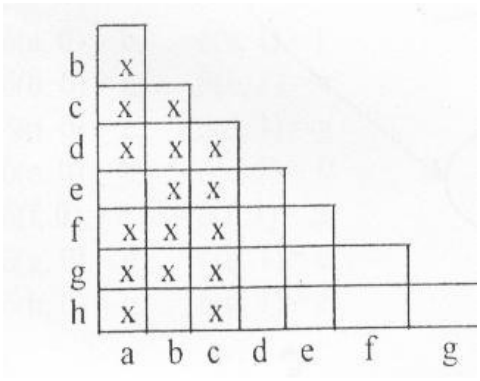
-> (δ (b, 1), (c. 1)) = (c. 1), place 'x' in the entry (b,e)

-> (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.

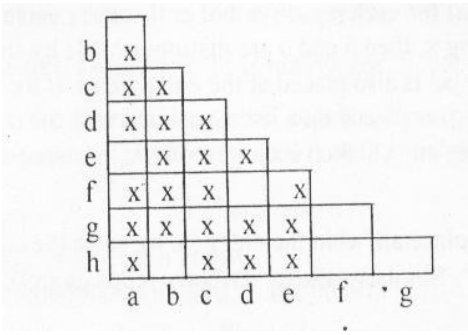
-> (δ (d, 0), (g, 0)) = (c, g), place an 'x'in(d, g).

-> (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

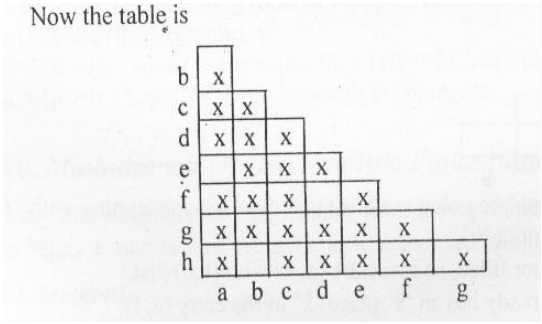


-> (δ (f, 0), δ (g, 0)) = (c, g), place an ‘x’ in (f. g).

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

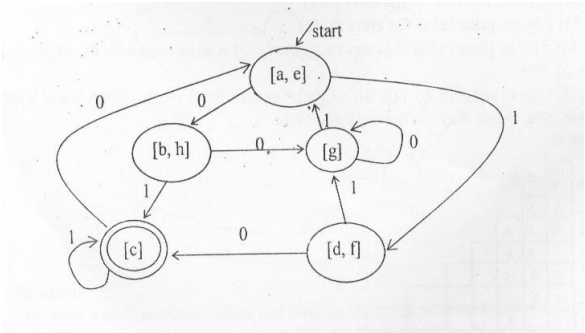
-> (δ (g, 1), δ (h, 1)) = (e, c), place an ‘x’ in (g , h).

Now the table is



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



Applications of Finite Automata:

- Finite State Programming
- Event Driven Finite State Machine (FSM)
- Virtual FSM
- DFA based text filter in Java
- Acceptors and Recognizers
- Transducers
- UML state diagram
- Hardware Applications

Limitations of Finite Automata:

The defining characteristic of FA is that they have only a finite number of states. Hence, a finite automata can only "count" (that is, maintain a counter, where different states correspond to different values of the counter) a finite number of input scenarios.

There is no finite automaton that recognizes these strings:

- The set of binary strings consisting of an equal number of 1's and 0's
- The set of strings over '(' and ')' that have "balanced" parentheses

The 'pumping lemma' can be used to prove that no such FA exists for these examples.

UNIT - II

1. REGULAR EXPRESSIONS

- Regular expressions are shorthand notations to describe a language. They are used in many programming languages and language tools like lex, vi editor etc. They are used as powerful tools in search engines.
- Regular expressions (RE) are useful for representing certain sets of strings in an algebraic fashion. RE describes the language accepted by finite state automata.

Definition:

Let $\sum$ be a given alphabet. Then

i. ϕ ,$\in$,and 'a' $\in \sum$ are all Regular expressions. These are called 'Primitive Regular expressions'

ii. If r1 and r2 are regular expressions, so are $r_1 + r_2$, $r_1 r_2$, r_1^* and (r_1).

iii. 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).

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. ϕ is a regular expression denoting the empty set.
2. $\in$ is a regular expression denoting the set $\{\in\}$
3. For every $a \in \sum$, 'a' is a regular expression denoting set set {a}.

If r_1 and r_2 are regular expressions, then

4. $L(r_1+r_2)=L(r_1) \cup L(r_2)$

5. $L(r_1.r_2)=L(r_1)L(r_2)$
6. $L(r_1^*)=(L(r_1))^*$

Example:

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

$$L(a^*.(a+b))=L(a^*)L(a+b)=(L(a^*))(L(a) \cup L(b))$$

$$=\{\in,a,aa,aaa,.....\}\{a,b\}=\{a,aa,aaa,.....,b,ab,aab,...\}$$

Precedence of Regular Expression Operators

Consider the RE *ab + c*. The language described by the RE can be thought of either $L(a)L(b+c)$ or $L(ab) \cup L(c)$ as provided by the rules (of languages described by REs) given already. But these two represents two different languages lending to ambiguity. To remove this ambiguity we can either

1) Use fully parenthesized expression - (cumbersome) or

2) Use a set of precedence rules to evaluate the options of REs in some order. Like other algebras mod in mathematics.

For REs, the order of precedence for the operators is as follows:

i. The star operator precedes concatenation and concatenation precedes union (+) operator.
ii. It is also important to note that concatenation & union (+) operators are associative and union operation is commutative.

Using these precedence rule, we find that the RE ab+c represents the language $L(ab) \cup L(c)$ i.e. it should be grouped as ((ab)+c).

Of course change the order of precedence by using parentheses. For example, the language represented by the RE $a(b+c)$ is $L(a)L(b+c)$.

Example**:**

The RE ab*+b is grouped as $((a(b^*))+b)$ which describes the language $L(a)(L(b))^* \cup L(b)$

Example**:**

The RE $(ab)^*+b$ represents the language $(L(a)L(b))^* \cup L(b)$.

Example**:**

It is easy to see that the RE (0+1)*(0+11) represents the language of all strings over {0,1} which are either ended with 0 or 11.

Equivalence of Regular expressions:

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

Example: Consider the following regular expressions

$r_1=(1^*011^*)^*(0+\in)+1^*(0+\in)$ and $r_2=(1+01)^*(0+\in)$

Both r_1 and r_2 represent the same language i.e. the language over the alphabet {0,1} with no pair of consecutive zeros. So r_1 and r_2 are said to be equal.

Algebraic Laws For Regular Expressions:

Let r_1, r2 and r_3 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. $r_{1+}r_2=r_2+r_1$
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.

$$(r_{1+}r_2)+r_3= r_{1+(}r_2+r_3)$$

3. Associative law for concatenation:

$$(r_1r_2)\ r3= r_1\ (r_2r_3)$$

4. Distributive laws for concatenation:
   - Concatenation is left distributive over union

$$\text{i.e. } r_1\ (r_2+r_3) = r_1r_2+ r_1r_3$$

   - concatenation is right distributive over union

i.e. $(r_1+r_2)\ r3=r_1r_3+ r_2r_3$

5. Identities For union And Concatenation:
   - ϕ is the identity for union operator

     i.e. $r_1+\phi=\phi+r_1=r_1$
   - $\in$ is the identity for concatenation operator

     i.e. $r_1\in=\in r_1=r_1$
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**.**

     ϕ is the Annihilator for concatenation

     i.e. $\phi r_1= r_1\ \phi=\phi$

     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. $r_1 + r_1=r_1$

8. Laws involving closure
   - Let 'r' be a regular expression ,then

     1. $(r^*) =r^*$
     2. $\phi^*=\in$
     3. $\in^*=\in$
     4. $r^+=r.r^*=r^*.r$ i.e. $r^+=rr^*=r^*r$
     5. $r^*=r^++\in$
     6. r?= $\in$+r (Unary postfix operator ? means zero or one instance)

2. Equivalence between REs and FA

The language that is accepted by some FAs is known as Regular language. The two concepts: REs and Regular language are essentially same i.e. for every regular language can be developed by a RE, and for every RE there is a Regular Language. This fact is rather surprising, because RE approach to describing language is fundamentally different from the FA approach. But REs and FA are equivalent in their descriptive power.

Construction of $\in$- NFA from a regular expression

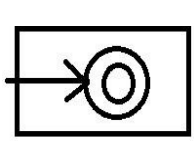
Lemma: If $L(r)$ is a language described by the RE r, then it is regular i.e. there

is a FA such that $L(M) \cong L(r)$.

Proof: To prove the lemma, we apply structured index on the expression r. First, we show how to construct FA for the basis elements: ϕ, $\in$ and for any $a \in \Sigma$. Then we show how to combine these Finite Automata into Complex Automata that accept the Union, Concatenation, Kleen Closure of the languages accepted by the original smaller automata.

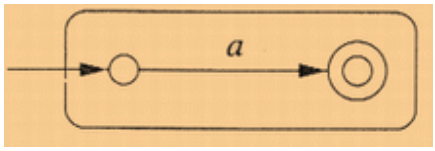
Use of NFAs is helpful in the case i.e. we construct NFAs for every REs which are represented by transition diagram only.

Basis: Automata for $\in$, ϕ and 'a' are (a),(b) and (c) respectively.



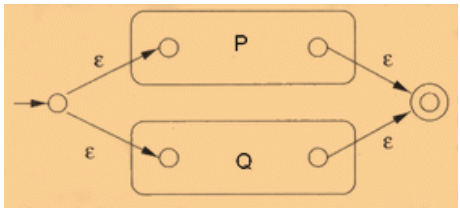
a) Accepting ∈

b) Accepting ϕ

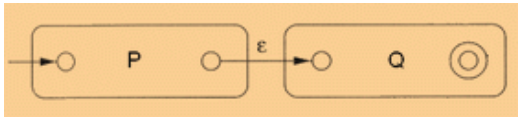


c) Accepting a

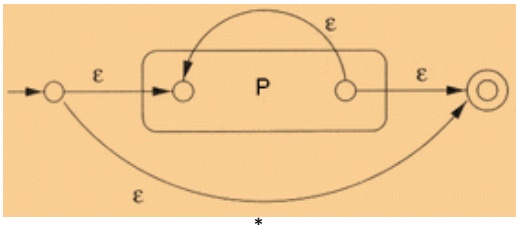
Induction: Automata for P+Q, PQ and P^* are (d), (e) and (f) respectively.



d) P+Q



e) PQ

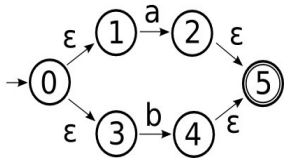


f) P^*

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

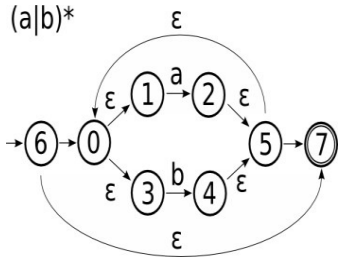
Solution: using Thompson's Construction. First we construct the union of a and b:

a|b



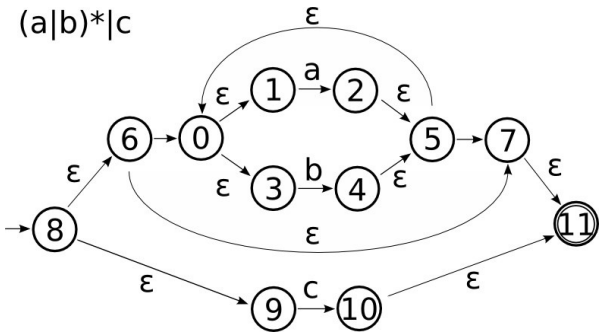
Next we construct the Kleene Star of the previous union:

(a|b)*



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

(a|b)*|c



Construction of DFA from a regular expression:

Refer Class notes

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.

i. The finite automaton does not have ∈ - moves.
ii. It has only one initial state, say q_0.
iii. It's states are $q_0, q_1 \ldots q_n$.
iv. Q_i is the regular expression representing the set of string accepted by the automata even through q_i is a final state.
v. α_{ij} denotes the regular expression representation the set of labels of edges from v_i to v_j when there is no such edge $\alpha_{ij}=\phi$.

Consequently, we can get the following set of equation in $Q_1, \ldots Q_n$

$Q_1= Q_1 \alpha_{11}+ Q_2 \alpha_{21}+\ldots.+ Q_n \alpha_{n1}+\in$

$Q_2= Q_1 \alpha_{12}+ Q_2 \alpha_{22}+\ldots + Q_n \alpha_{n2}$

……

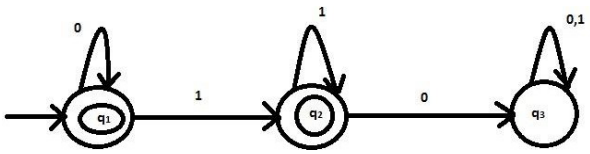
……

$Q_n= Q_1 \alpha_{1n}+ Q_2 \alpha_{2n}+\ldots..+ Q_n \alpha_{nn}$

By repeatedly applying substitutions and Arden's theorem we can express Ri in terms of α_{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:

$q_1=\in +q_10$ .....(1)

$q_2=q_11+q_21$ ..... (2)

$q_3=q_20+q_30+q_31$.....(3)

(2) → $q_2=q_11+q_21$

$q_2=q_1 11^*$ ..... (4)

(1) → $q_1=\in +q_10$ (Apply Arden's theorem)

$q_1=\in 0^*$

$q_1=0^*$

(4) →$q_2=0^*11^*$

$q_2=0^*1^*$

3. Pumping lemma for Regular languages

It can prove that a certain language is non regular by using a theorem called "Pumping Lemma". According to this theorem every regular language must have a special property. If a language does not have this property, than it is guaranteed to be not regular. The idea behind this theorem is that whenever a FA process a long string (longer than the number of states) and accepts, there must be at least one state that is repeated, and the copy of the sub string of the input string between the two occurrences of that repeated state can be repeated any number of times with the resulting string remaining in the language.

Pumping Lemma:

STATEMENT :Let 'L' be an infinite Regular language, then there exits some positive integer 'n' such that any Z∈ L with $|z| \geq n$ can be decomposed as Z=uvw

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

Let us apply it on $L=\{0^n1^n/n\geq 1\}$

Proof : Refer Class notes

Assume L to be regular and apply pumping lemma.

By pumping Lemma

There exists some n ≥ 0

We choose $w=0^n1^n$

Clearly w∈L 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 $z=0^a\,0^b0^{n-a-b.}\,1^n$

If we choose k=0,

$Uv^iw\in L$

U,w∈L

$0^{n-b}1^n \in L$

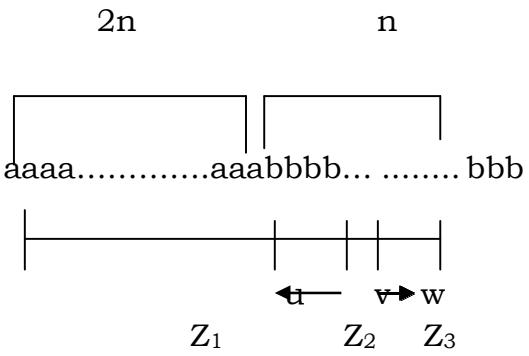
But for b>0 ,this is not true ,a contradiction .

Hence L is not a Regular Language.

Example:

1. Prove or disprove the regularity if the language $\{a^ib^i \mid i>j\}$

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



$Z_1uv^i wZ_3 \in L$

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

$a^{2n}\, b^{(n-d)+3nd} = a^{2n} b^{(3d+1)\ n\ -\ d}$

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

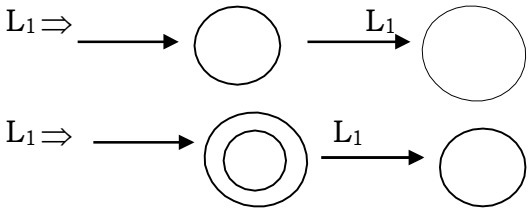
Applications of Pumping Lemma

The pumping lemma is extremely useful in proving that certain sets are non-regular. The general methodology followed during its applications is:

1. Select a string z in the language L.
2. Break the string z into u, v and w in accordance with the above conditions imposed by the pumping lemma.
3. Now check if there is any contradiction to the pumping lemma for any value of i.

4. Closure Properties of Regular Languages

1. Regular languages are closed under union, concatenation and kleene closure.
2. Regular languages are closed under complementation. i.e., if L_1 is a Regular language and $L_1 \subseteq \overline{\sum}^*$, then $L_1 = \sum^* - L_1$ is Regular language.



3. Regular languages are closed under intersection.

i.e., if L_1 and L_2 are Regular languages then $L_1 \cap L_2$ 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 and inverse homomorphism.

UNIT - III

GRAMMARS FORMALISM

- Last chapters we have discussed finite automata (DFA's, NFA's, NFA-Λ's). We will take a look at other "machines" that model computation. Before we do that, we will examine methods of generating languages: Regular expressions, Grammars. We have already discussed regular expressions. Now we will examine several kinds of grammars.
- Generally, we are familiar with natural language grammars. For example, a sentence with a transitive verb has the "creation" rule:

  <sentence>→<subject> <predicate>

  Then there are other rules that we apply before getting to the actual words:

  <subject>→<article> <adjective> <noun>

  <predicate>→<verb> <object>

  <object>→<article> <noun>
- In grammar theory, we call <sentence>, <subject>, <predicate>, <verb>, <object>, <article>, <adjective>, <noun>, etc., variables or non-terminals. The variable <sentence> is special and it is the "start variable", i.e. where we start constructing a sentence.
- Then there are other rules that allow us to replaces variables by actual dictionary words, which we call terminals:

  <noun>→dog,<noun>→cat,<verb>→chased, etc.

  We sometimes diagram sentences using these substitution rules, and thus build a parse tree, with the start variable at the root.

Definition of Grammar:

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

(i) V is a finite nonempty set whose elements are called variables.

(ii) T is finite nonempty set, whose elements are called terminals.

(iii) S is a special variable (i.e an element of V) called the start symbol, and

(iv) P is a finite set whose elements are $\alpha \rightarrow \beta$,where α and β are strings on V $\cup$ T, α has at least one symbol from V. Elements of P are called Productions or production rules or rewriting rules.

1. 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

- There are two types of Regular grammars:
  i. Right Linear Grammar (RLG)
  ii. Left Linear Grammar (LLG)

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^*.

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

  Where A,B ∈ V and x ∈ T^*.

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

2. 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 $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$

S: a start non-terminal; $S \in V$

A context-free grammar consists of a set of productions of the form $A \rightarrow \alpha$, where 'A' is a single non-terminal symbol and 'α' is a potentially mixed sequence of terminal and non-terminal symbols.

Eg:

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

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

G=({E},{+,*,(,),id},{ $E \rightarrow E+E$, $E \rightarrow E*E$, $E \rightarrow (E)$, $E \rightarrow id$},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 $\overset{*}{\Rightarrow}_G$ 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)={ a^ib^i | i ≥ 0}

Solution: L={ ∈,ab,aabb,aaabbb,…}

S → aSb | ∈

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

Solution: L={ab,aabb,aaabbb,…}

S → aSb | ab

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

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

S → 0S0 | 1S1 |∈

5. Design CFG for a given language L(G)={ $w\#w^R$ | 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) | ( )

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., $\Rightarrow_G$ if necessary.

  Example: 011AS ⇒ 0110A1S
- $\alpha \overset{*}{\Rightarrow} \beta$ means string α can become β in zero or more derivation steps.

  Example: 011AS $\overset{*}{\Rightarrow}$ 011AS (zero steps);

  011AS $\overset{*}{\Rightarrow}$ 0110A1S (one step);

011AS $\overset{*}{\Rightarrow}$ 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 S $\overset{*}{\Rightarrow}$ α.

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.

→ $\overset{}{\underset{lm}{\Rightarrow}}$, $\overset{}{\underset{rm}{\Rightarrow}}$, etc., 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: Construct parse tree for the following CFG and take input string is 0110011.

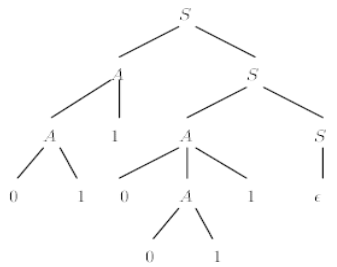
S → AS | ∈
A → 0A1 | A1 | 01

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

LMD: S $\overset{\Rightarrow}{lm}$ AS $\overset{\Rightarrow}{lm}$ A1S $\overset{\Rightarrow}{lm}$ 011S $\overset{\Rightarrow}{lm}$ 011AS $\overset{\Rightarrow}{lm}$ 0110A1S $\overset{\Rightarrow}{lm}$ 0110011S $\overset{\Rightarrow}{lm}$ 0110011.

RMD: S $\overset{\Rightarrow}{rm}$ AS $\overset{\Rightarrow}{rm}$ AAS $\overset{\Rightarrow}{rm}$ AA $\overset{\Rightarrow}{rm}$ A0A1 $\overset{\Rightarrow}{rm}$ A0011 $\overset{\Rightarrow}{rm}$ A10011 $\overset{\Rightarrow}{rm}$ 0110011.

Parse tree:



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: $S \overset{lm}{\Rightarrow} AS \overset{lm}{\Rightarrow} 0A1S \overset{lm}{\Rightarrow} 0A11S \overset{lm}{\Rightarrow} 00111S \overset{lm}{\Rightarrow} 00111$

LMD 2: $S \overset{lm}{\Rightarrow} AS \overset{lm}{\Rightarrow} A1S \overset{lm}{\Rightarrow} 0A11S \overset{lm}{\Rightarrow} 00111S \overset{lm}{\Rightarrow} 00111$

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

Example 2:

Consider the following grammar:

$S \rightarrow SS$

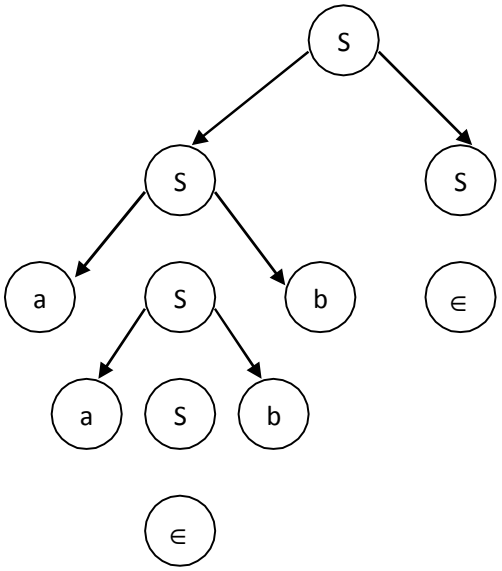
$S \rightarrow aSb$

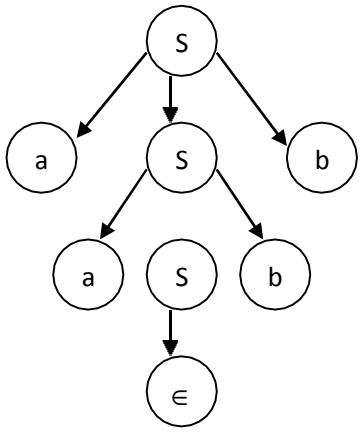
$S \rightarrow bSa$

$S \rightarrow \in$

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.





Inherently Ambiguous context free language:

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

Example:

L={$a^n b^n c^m d^m$ | n≥1,m≥1} U {$a^n b^m c^m d^n$ | 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.

3. 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: (Simplification Order is)

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

Eliminating ∈ - productions:

- A variable A is nullable if A $\Rightarrow^{*}$ ∈. 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

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 $\overset{*}{\Rightarrow}$ B.
    - Because there are no ∈ - productions, this derivation can only use unit productions.
    - Thus, we can find the pairs by computing reachability in a graph where nodes = variables, and arcs = unit productions.
- Replace each combination where A $\overset{*}{\Rightarrow}$ 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**.

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., $S \overset{*}{\Rightarrow} \alpha X \beta$
- 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

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
  i. Chomsky Normal Form (CNF)
  ii. Greibach Normal Form (GNF)

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.

i. All the productions in P of the form A → a and A → BC are included.

ii. Consider A → $w_1w_2 \ldots w_n$ will some terminal on right hand side. If w_i is a terminal say a_i, add a new variable c_{ai} and c_{ai} → P. Repeat same for all terminals.

Step 3:

Restricting the number of variables on RHS as follows:

i. All the productions in P are added to P, if they are in the required form.

ii. Consider A → $A_1A_2A_3 \ldots A_m$, then we introduce new productions are,

$A \rightarrow A_1C_1$

$C_1 \rightarrow A_2C_2$

$C_2 \rightarrow A_3C_3$

$C_{m-2} \rightarrow A_{m-1}C_{m-1}$

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 non terminal 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 non terminals.

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

The new CFG, with a RHS consisting of only solid non terminals 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 non terminals to length 2.

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

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

(i) Z →αi|αiZ, 1≤ i≤ r and

(ii) 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:
   (a) Assume that productions have been modified so that for 1≤ i≤ k, Ai → Aj γ € R' then j > i.
   (b) 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.
   (c) Repeating (b) at most k − 1 times we obtain rules of the form Ak → Apγ, p≥ k
   (d) Replace rules Ak → Akγ by removing left-recursion as stated above.
5. 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

7. Closure Properties of CFL's:

- The context-free languages are closed under
  - substitution
  - union
  - concatenation
  - Kleene star
  - homomorphism
  - reversal
  - intersection with a regular set
  - inverse homomorphism

8. Non-closure Properties of CFL's:

- The context-free languages are not closed under
  - intersection
    - $L_1 = \{a^n b^n c^i \mid n, i \geq 0\}$ and $L_2 = \{a^i b^n c^n \mid n, i \geq 0\}$ are CFL's. But $L = L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ is not a CFL.
  - complement
    - Suppose comp (L) is context free if L is context free. Since $L_1 \cap L_2$ = comp (comp (L_1) $\cup$ comp (L_2)), this would imply the CFL's are closed under intersection.
  - difference
    - Suppose $L_1 - L_2$ is a context free if L_1 and L_2 are context free. If L is a CFL over Σ, then comp (L) = Σ^* - L would be context free.

9. 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 $|z| \geq n$, we can write z as $uvwxy$ where

   1. $vx \neq \varepsilon$,
   2. $|vwx| \leq n$, and
   3. for all $i \geq 0$, the string $uv^i wx^i y$ 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

   i. v and x together have at least one distinguished positions,
   ii. vwx has atmost n distinguished positions, and
   iii. for all i≥0, $uv^i wx^i y$ is in L.

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

$|uvwxy| = |uwy| + |vx|$

$|uv^2wx^2y| = |uwy| + 2|vx|$

.....

$|uv^iwx^iy| = |uwy| + i|vx|$.

When we pump the lengths are in arithmetic progression.

Example:

1. The language L = { $a^nb^nc^n \mid n \geq 0$ } is not context free.

Solution: **Refer Class Notes**

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 $|z| \geq n$, z can be written as $uvwxy$ such that

1. $vx \neq \varepsilon$,
2. $|vwx| \leq n$, and
3. for all $i \geq 0$, the string uv^iwx^iy is in L.

Consider the string $z = a^nb^nc^n$.

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

- 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.
- In both cases we have a contradiction, so we must conclude L cannot be context free.