UNIT-1

Introduction to finite Automata —

Finite Automata — A finite Automata or finite State machine (FSM) is a mathematical model that takes String (I/P) as its Input and checks whether it belongs to the given lang or not.

A finite State machine is mainly used in designing the text editors, language processors and lexical analyzers.

A finite automata (FA) is designed using five tuples such as,

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

Where,

Q = A finite Set of States i.e, $\{q_1, q_2, q_3, \ldots\}$. (non-Empty)

Σ = A finite Set of Input Symbols. (sigma) (alphabet) (s-e)

δ = A transition function that defines rule for change of State. (delta)

q_0 = An initial State or Start State.

where $q_0 \in Q$ (belongs)

F = The Set of final State or accepting State $F \subseteq Q$.

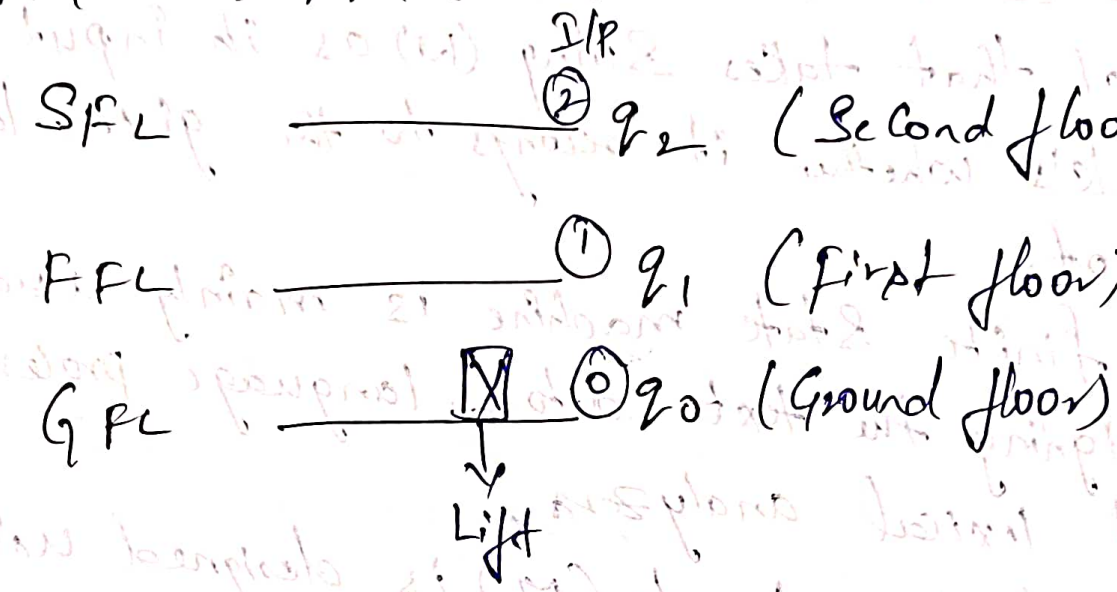
(subset)

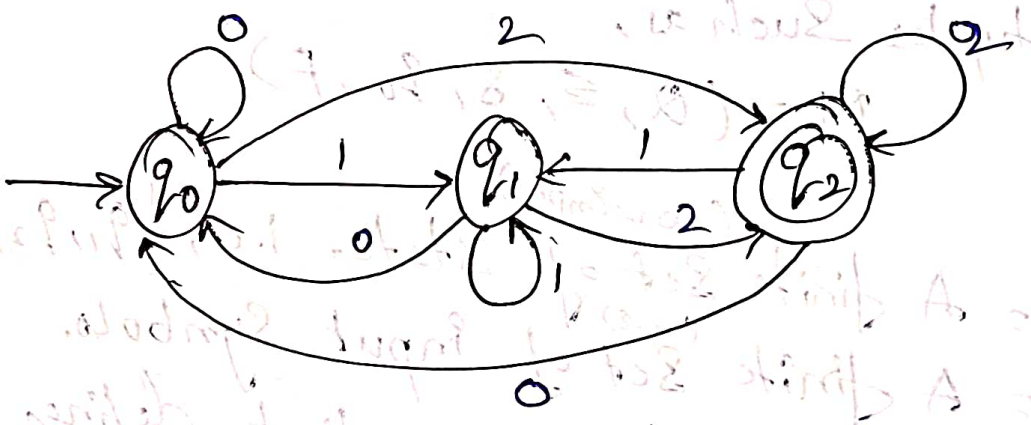
'δ' wch maps $Q \times \Sigma$ into Q, i.e $Q \times \Sigma \to Q$

(sigma)

- Mapping is usually Represented by transition table or transition function.

Example of lift control





Transition System (diagram)

→ (q0) is the initial state denoted with an →

((q2)) is the final state denoted with ⊛

There is no mandatory that q0 is an initial state based on certain conditions (situations) any input state can be an initial state. and any input state can be of final state.

where according to this light control transition diagram the tuples designed are.

$Q = \{q_0, q_1, q_2\}$ → input States

$\Sigma = \{0, 1, 2\}$ → input Symbols

q_0 = initial state, $q_0 \in Q$

$F = \{q_2\}$ (final states) where $F \subseteq Q$

(present state) p_s	Next State (NS) 0	1	2 ← i/p's
→ q_0	q_0	q_1	q_2
q_1	q_0	q_1	q_2
((q_2))	q_0	q_1	q_2

Transition Table

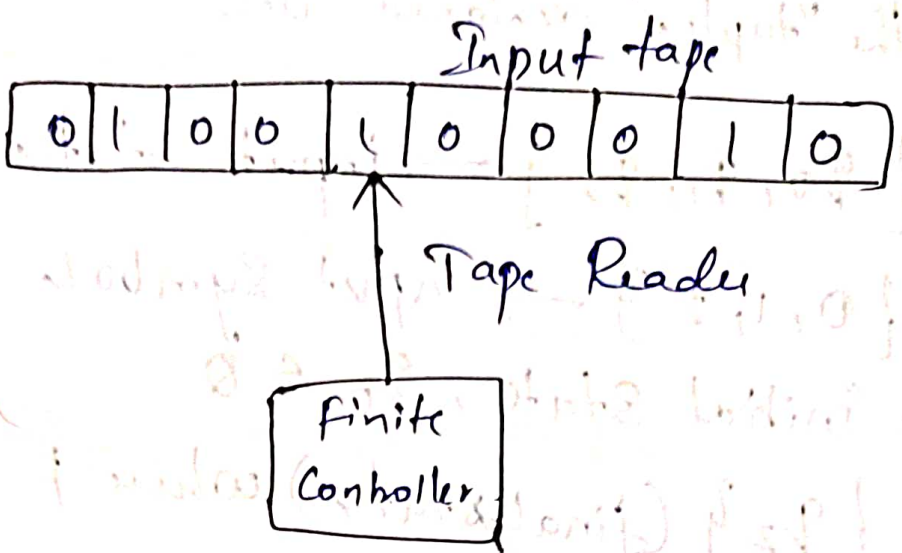
$\delta(q_0, 0) = q_0$ $\quad \delta(q_0, 1) = q_1$ $\quad \delta(q_0, 2) = q_2$

$\delta(q_1, 0) = q_0$ $\quad \delta(q_1, 1) = q_1$ $\quad \delta(q_1, 2) = q_2$

$\delta(q_2, 0) = q_0$ $\quad \delta(q_2, 1) = q_1$ $\quad \delta(q_2, 2) = q_2$.

i.e $Q \times \Sigma = 3 \times 3 = 9$

Mathematical model of FA :—



The mathematical model of finite automata consists of three types.

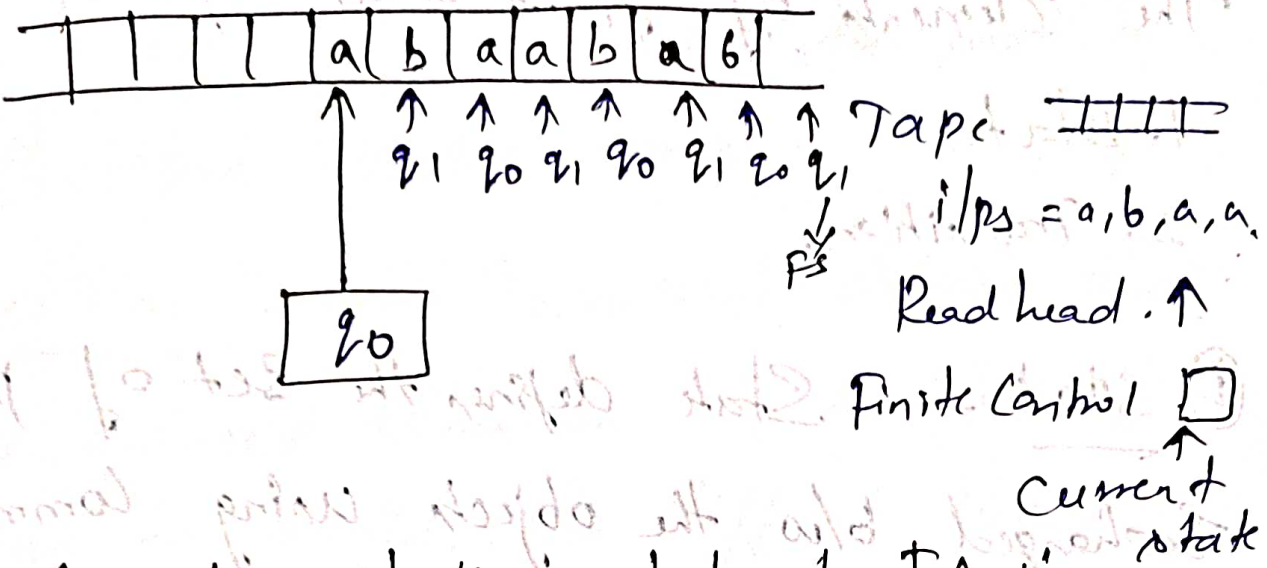
(i) Input Tape: It is a Sequential tape consisting of finite number of cells where each cell can hold only single input symbol.
(Containing Input Strings)

(ii) Tape Reader:— It performs the read operation cell-by-cell and reads only one input symbol (from each cell) at a given instance.
(it read only head starts traverse from left end of the tape).

(iii) Finite Controller:— It takes decision about wch input symbol to read next after

receiving a particular Input.

Ex:



Let's Consider starting state of FA is q_0 by reading Symbol a it may change the state to q_1 and q_1 on b it may change the state to q_0 and q_0 on a it may change the state to q_1 .. so on at last (Every time the read head will move to forward direction it doesn't go in backward direction (work).

q_0 on b will move to the next state i.e q_f but there is no Input Symbols in a tape, the System will halt. i.e q_f is a final State Control. (or machine)

i.e $q_1 \in F$ (final state) $\rightarrow$ accepted

$q_1 \notin F \rightarrow$ Not accepted.

Elements of Finite Automata —

The Elements of finite State System are

1. State
2. Transition.

① State — State defines the set of properties Exchanged b/w the objects using Communication channels. Any modification done in the property is detected by the receiving object.

Types of States — The different types of states are,

(a) Start State — It is the initial State of finite State machine (FSM).

(b) Next Accepting State — It is a State wch is immediately figured after the current State defined by transition function.

(b) Accepting State — It is a State in wch the String is accepted.

(d) Universal State —— It is the State, in wch

FSM accepts only those moves that leads to acceptance.

(c) Dead / Trap State —— It is the state in wch the input symbol terminates on itself.

(2) Transition —— Transition defines the change occured when an object moves from one state to another. The different ways of representing transition are,

(a) Transition diagram.
(b) Transition table
(c) Transition functions.

(a) Transition diagram — A transition diagram describes the behaviour of a finite automata. It consists of a collection of nodes or circles called states and directed line segments b/w each state called node edges.

The transition diagram wch associates more than one i/p symbol to an edge is called Non-deterministic finite transition diagram, whereas, the transition diagram wch associate unique i/p symbol on edge is called deterministic transition diagram.

Example & steps write same as lift control transition diagram wch i hav explained b/4 class.

(b) Transition table — It is the tabular representation of a given finite automata. It consists of rows and columns, where the rows correspond to the i/p state and the column corresponds to the i/p symbol of given i/p state. The transition table is used to define a transition function δ.

Ex! Same (write transition table Example with diagram).

(C) Transition function — A transition function defines the rules for change of the state. It takes two arguments a state and the i/p symbol. It returns a subset of o/p states i.e., more than one o/p state.

The properties of transition function are as follows,

Property! — For any transition function 'δ'.

and for any i/p symbol $\in$

$$\delta(q, \in) = q$$

Explanation: The state of the transition System is changed only by an input symbol.

property 2 → for any transition function 'δ', containing an input string s (small s) and input symbol b. (small b).

$$\delta(q, bs) = \delta(\delta(q, b), s)$$

and

$$\delta(q, sb) = \delta(\delta(q, s), b)$$

Explanation: The transition $\delta(q, bs)$ returns the next state after the first symbol of the string bs is read by the automation.

The transition $\delta(q, sb)$ returns the state after the prefix of the string sb is read by the automation.

Structural Representation of finite Automata

Structural Representations of finite Automata are used for representing the data in a systematic manner. In automata theory, two important notations are used for structural representation, those notation are mostly used in SW designing. The two types of structural representations are -

① Grammar ② Regular Expressions

① Grammar — A grammar ~~used~~ widely defines the construction of programming lang. This form of representation is needed for those SW that follow recursive structure for data processing.

For instance, a parser that uses the relatively nested aspect of programming languages including arithmetic expressions, conditional expressions etc,

Ex: Consider the grammar exp → exp + exp.

This means that addition of two expressions using plus operator can result in a new expression.

Context free grammar is an example of this form of representation.

② Regular Expression :— A Regular Expression is a concise notation for denoting regular set. It describes the language accepted by a finite automata. This form of representation is similar to that of finite automata, but it is different from grammars.

Ex — Consider the pattern " Capitalized word accompanied by a Space and two uppercase letters " UNIX — Supported regular Expression represent this pattern as

[A-Z][a-z]*[ ][A-Z][A-Z]

where

[A-Z] : Indicates all the uppercase characters from A to Z

[ ] : Indicates Space

* : Indicates any number of characters of the previous Expression.

Automata and Complexity :—

Automata plays an imp role in analyzing the boundaries of computation. It involves two issues such as follows.

① Desirability and decidability

② Intractability and tractability.

① Desirability and Decidability :— Computable at-all].

(a) Desirability :— D is the Study regarding all the functionality of the Computer. Mostly, it is a Study about wat a Computer can do, how it performs th task etc.,

(b) Decidability :— D refers to problem Solving ability of the Computer. A problem is said to be Solveable or decidable if it's Supportive language is "recursive". that is, a 'Solveable problem' can be easily Solved by using algorithmic approach. Here the word "decidable" refers to the word "recursive". Hence it can be said that for a given set of problems if there exists an algorithm which can Sucessfully produce two outputs namely "yes" or "no", Such Set of problems are said to be Set of "decidable problems".

② Intractability & Tractability

(a) Intractability :— Intractability is the Study regarding the Efficient working of a Computer. wat can a Computer do Efficiently!

(b) Tractability :— Tractability describes the problm Solving ability of the Computer within the time Specified by the polynomial functions (i.e., Slowly growing functions). A problem is said to be tractable if the time and Space Complexity is reasonable to Solve a problem.

The Central Concepts of Automata Theory,

Automata (what is Automata & what are the applications of FA).

Automata —— Automata is defined as a Self-operational System which doesn't require any Explicit human involvement in performing its function. The basic function carried out by automation include,

(i) Generating Information
(ii) Transforming Information
(iii) Transmitting Information

Applications ——

① It is used by language processors and Compilers in order to carry out the process of tokenization. In this process, the program is broken

down into Sequence of tokens like identifiers, Constants and Keywords.

(2) It is used in deleting / Eliminating Comments & white Spaces from program code.

(3) It is used by text processors in order to Search for a particular string in a text file that matches a particular pattern.

(4) It is used in various Signal & Speech processing Systems to transform incoming Signals.

(5) It is used in various industrial processing & in video game Systems.

* Central Concepts of AT ——

(a) Alphabets (b) Strings (c) Language (d) problems

for (1) st

(a) Alphabets — An Alphabet / Character is any non-Empty finite Set of Symbols.

(b) Strings — A finite Sequence of Symbol / drawn from alphabet - is known as strings

(c) Language — the language denoted by L is a Set of Strings over alphabet Σ.

(d) problem — A problem can be defined as a question which is useful to determine

whether a string belong to specific language.

5th

① Alphabets —

- It is a finite, non-Empty Sets of Symbols. (integer, alpha, binary No, operators can be brackets etc)
- Represented by Σ (Sigma)

(i) Binary Symbols : $\Sigma = \{0, 1\}$

(ii) All lower case letters : $\Sigma = \{a, b, c, \ldots z\}$

(iii) Digits : $\Sigma = \{0, 1, 2, \ldots 9\}$

(iv) Alphanumeric : $\Sigma = \{a-z, A-Z, 0-9\}$

② Strings —

- A string or word is a finite collection of Symbols selected from the alphabets (Σ).
- A string can be Empty, (if it is Empty it is denoted by ε) which is represented by ε ("epsilon"). ep-si-lon. (sai) (rep-si-lun)

Length of String: The "Length" of the string is denoted by $|w|$ and it is the number of positions for the Symbols in the string.

for Ex $w = 01101$ has length = 5 i.e $|w| = 5$

ababaa ∴ length = 6 $|w| = 6$

(representation with modulus sign.)

EMPTY STRING

The "empty" string is the string with zero occurrence of symbols, the empty string is represented by ε (Epsilon).

$\therefore \Sigma^* = \{ \underset{0}{\varepsilon}, 0, 1, \underset{L=2}{01}, \underset{L=2}{10}, \underset{L=3}{000}, \underset{L=3}{010}, \underset{L=4}{0000}, \ldots \}$.

(power i.e * is nothing but powers of Σ

$\Sigma^0, \Sigma^1 \Sigma^2 \Sigma^3 \Sigma^4 \ldots \quad * = 0, 1, 2, \ldots \infty$

i.e $\Sigma = \{a, b\}$ $\quad \Sigma^2 = \{a, b\}\{a, b\}$

$= \{aa, bb, ab, ba\}$.

$\{0\}^* = \{\varepsilon, 0, 00, 000, 0000, \ldots\}$

$\{1\}^* = \{\varepsilon, 1, 11, 111, 1111, \ldots\}$

Note that

ε is in Σ^*, i.e ε is only string whose length is 0.

(tell after completion of lang topic) demand them all

[Alphabets (Σ); Strings (w), Languages (L); power (*), Length (|w|)]

Concatenation of strings — let x & y be two strings, then xy denotes the concatenation of x and y i.e, the string formed by making a

Concatenation — of x and followed it by a copy of y. Kushagra

Ex: if x=ab & y=cd then Concatenation xy = abcd

Reverse of the string:

Reverse of the string can be achieved by simply interchanging over last symbols.

For Ex w=abc, then $(w)^R$ = cba.

LANGUAGES :

(Language(L) is a one we're going to design a machine. L → M/c

L is said to be the language over a given set of alphabets, if L belongs to Σ^*.

(If an having a string (H)

Σ^* — power (represent the universal set of strings)

String is having alphabets set of

[By using these set of strings along with certain rules we are going to design a Machine. Here,]

Ex.① — L is a language that comprises of set of even numbers over the alphabet.

Consider that length = { 0, 2, 4, 6, 8. ... } (even no's)

then language shd be.

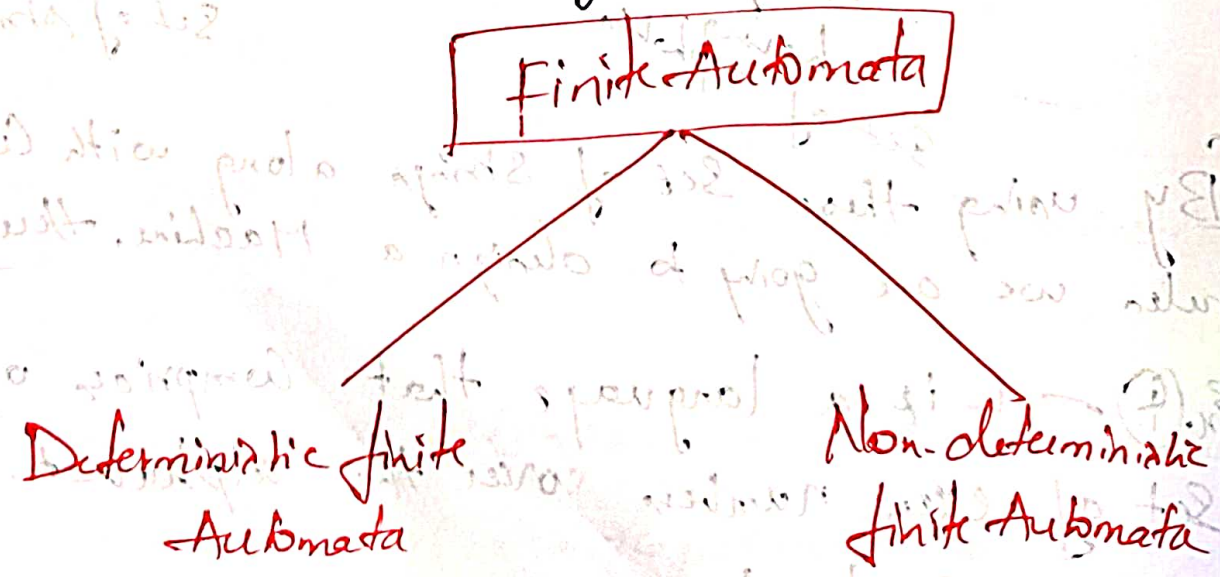
0 indicates epsilon

$L = \{ \in, \underbrace{00, 11, 01, 10}_{2}, \underbrace{0000, 1111, 1010, 0101}_{4}, \ldots \}$

(This is how we took the language in statement form & then represent into a language representation form).

(d) problems — A problem can be defined as a question which is useful to determine whether a string belong to some specific language. In other words, if L is a language over an alphabet Σ the problem is,

$L = \{ W \in \Sigma^* / \text{only if } W \in L \}$



Deterministic Finite Automata (DFA)

(one-one transition)

DFA refers to deterministic finite Automata. Deterministic refers to the uniqueness of the computation.

[The Name itself says determine i.e Consider $\Sigma = \{0, 1\}$ two i/p symbols/alphabet 0 & 1 it will determine that i will concatenate on 0 & i will also concatenate on 1 & i will send these 0 & 1 from one state — another state Exactly once not more than once,] (i.e transition) (not less than also. (for every state)

- In DFA, there is only one path for specific i/p from the current state to the next state.
- DFA does not accept the null move, i.e., the DFA cannot change state without any input character.

→ These are some characteristics of DFA →

DFA — A DFA is a collection of 5-tuples same as we described in the definition of FA = { Q, Σ, q0, F, δ }

Same as for final state F we can use X, q also etc.

q0, we can use A, B any alphabet in the place of initial state

there is no mandatory that we can use the same.

DFA = { Q, Σ, q0, F, δ }

1. Q: finite set of States
2. Σ: finite set of i/p Symbol
3. q0: initial State
4. F: final State
5. δ: Transition function,

Transition function can be defined as

$$\delta : Q \times \Sigma \to Q$$

- Total no. of states
- will be looped
- Total no. of i/p symbols
- It may go to the next state or present state

Transition Diagram —

- Each State in Q is represented by a Circle. ○
- A directed Edge from State q to state p labeled 'a' if $\delta(q,a) = p$.



that a particular Symbol/alphabet will be transfered from one state — another.

- In the initial State, there is an arrow with no source. →○
- Accepting States or final States are indicated by a double circle. ◎

Transition Table —

Same write as finite automata, transition table.

Ex: $Q = \{q_0, q_1, q_2\}$, where q_0 is the Starting state & q_2 is the final state

I/p (Σ) →	a	b
→ q_0	q_1	q_2
q_1	q_2	q_0
* q_2	q_2	q_2

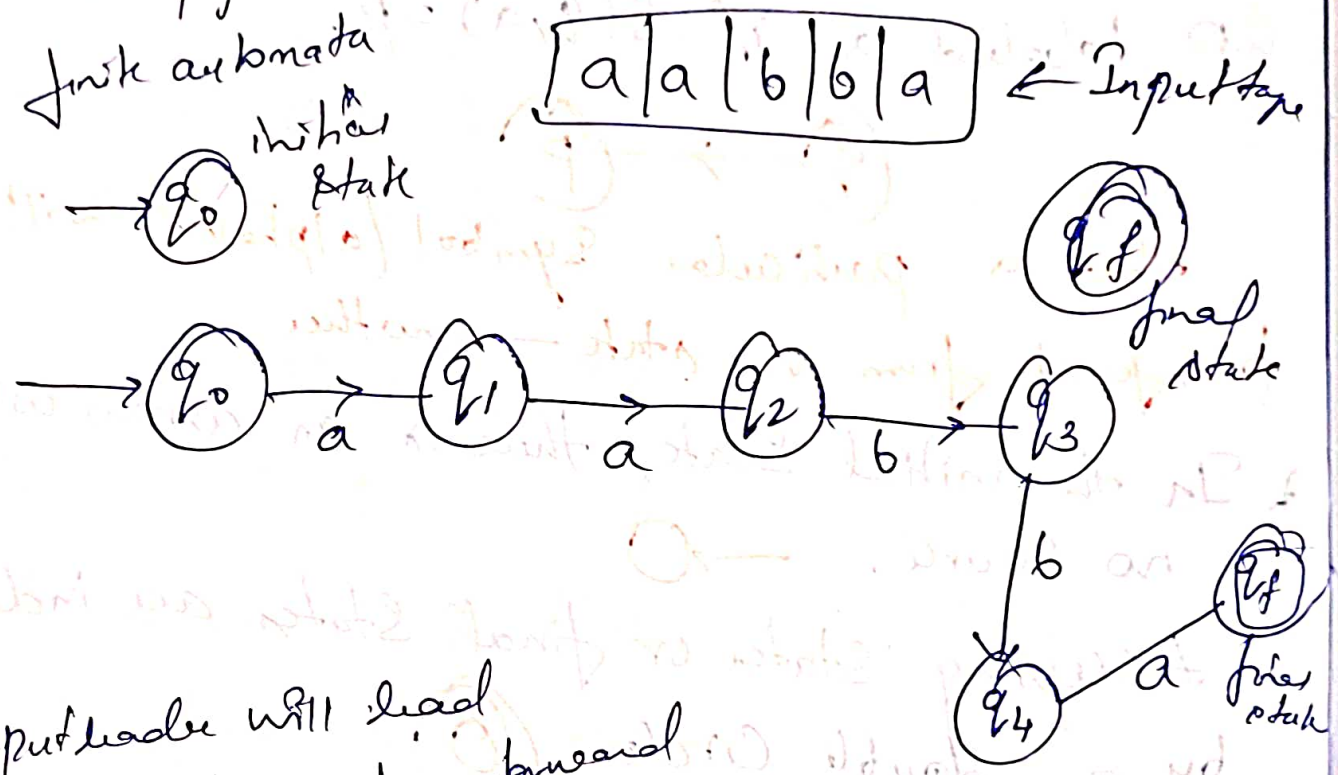
Q ↑

where $Q \times \Sigma \to Q$

Ex:

Construct a DFA that accepts a String "aabba" over $\Sigma = \{a, b\}$

first simply construct finite automata



Input header will read each i/p from a tape forward direction.

Examples (DFA)

Ex: Construct a DFA over $\Sigma = \{a, b\}$, where the length is Equal to 2 (any given no.)

Sol: the language will be: $L = \{aa, ab, ba, bb\}$

Step 1: from the given language (the language is already given to us).

Step 2: From the language we need to identify strings (which are nothing but i/p's)

Step 3: From the strings i/p's we need to analyse i.e we need to put these strings in a machine format (designing part).

Language → Strings = I/p → Machine (FA).

Construction.

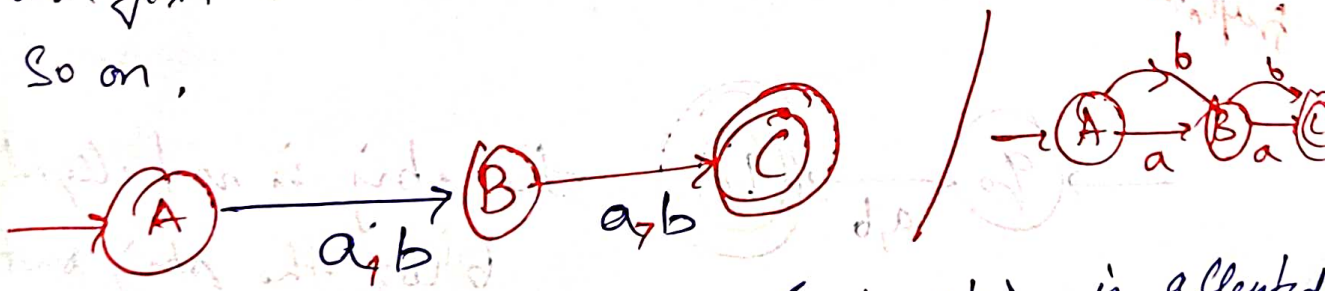
$\Sigma = \{a, b\}$

$\Sigma^2 = \{a, b\}\{a, b\}$

$= \{aa, ab, ba, bb\}$

(aa, ab, ba, bb — strings)

For designing automata we need to scan one by one. First it will check aa then it ab ... - next so on.

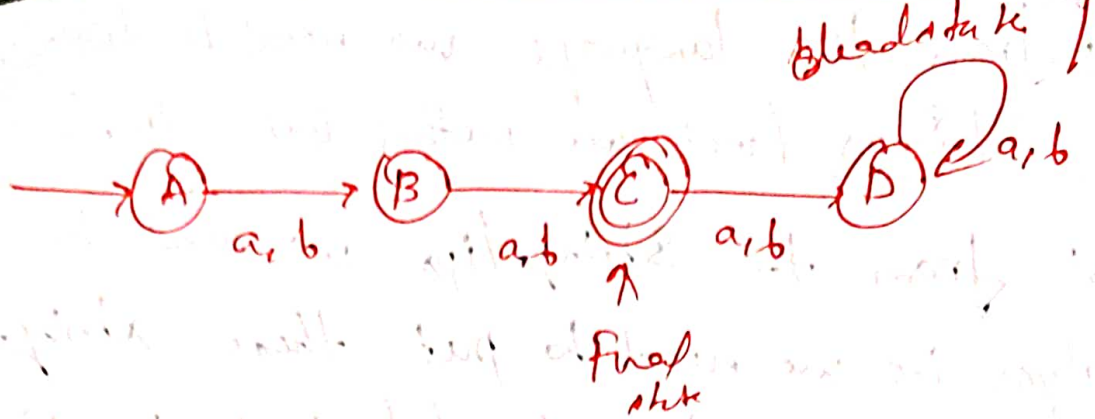


Here we need to check each string is accepted

① language

② M/c to accept all the strings

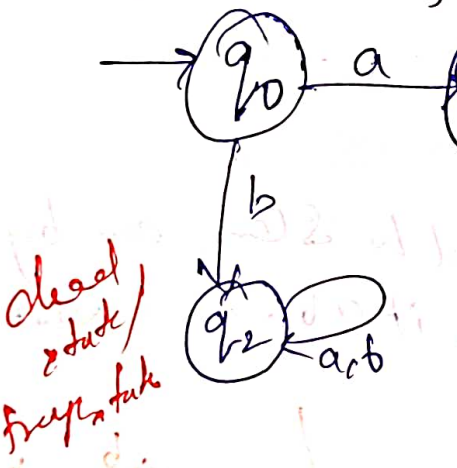
③ DFA (restriction)



* Let's Consider. Construct a DFA over $\Sigma = \{a, b\}$ where the String Starting with a.

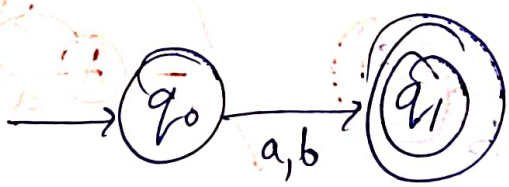
A) $L = \{a, aa, aaa, \ldots, ab, ba\}$

DFA = $\{Q, \Sigma, \delta, q_0, F\}$

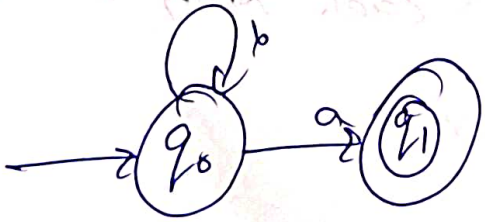


$\Sigma = \{a, b\}$

we have denoted for a but we doesn't denoted for b.



the string is not accepted b'coz the string should not start with b.



It will also not accepted b'coz it will start with b.

How (q1) a,b we kept q_1 as self loop a,b bcoz after a, aa, aaa, abc. anything it may be by keeping a self loop it will accept the string starting with a.

Same for (q2) a,b bcoz it shd not (unfortunately) come back to q_0 or it shd not move to q_1 i.e final state that's why we are keeping self loop.

$\delta: Q \times \Sigma \rightarrow Q$

	(a,b)		
q_0	q_0, a	q_2, a	total 6 states.
q_1	q_0, b	q_2, b	
q_2	q_1, a		
	q_1, b		

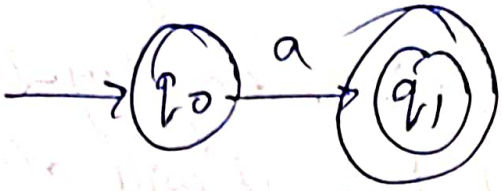
* Construct a DFA which accept a language of all strings.
  - Containing 'a'
  - Ending with 'a'.

(The string shd contain a.

i.e {a, aa, aaa, ba, ab, abab ... }

infinite lang.

first we need to design minimum string shd be accepted.



the first string is allepted

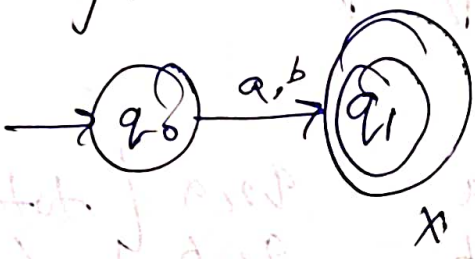
① fill in the blanks a, aa, aaa, ...

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

a is completed.

③ possibility for b.

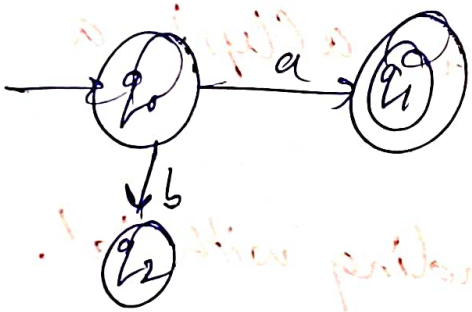
①



Only b starting but our question is wrong.

②

if we use b as in deadstate



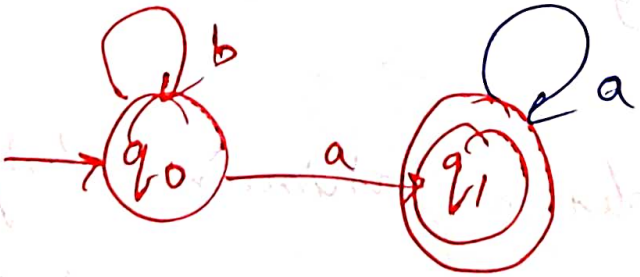
this is also wrong bcoz it shd allept string containing a

ba, baa, ab.

it not allept b. then it is wrong.

③

③ we shd not have trapstate here.

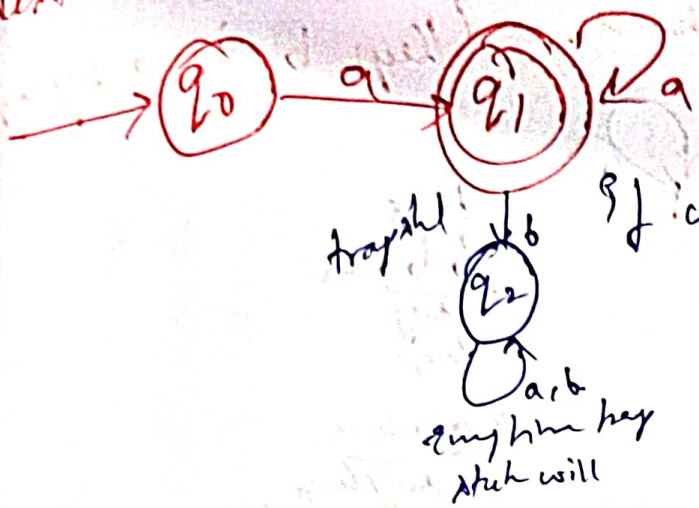


ba
baa

Here first we have created for string one a. & if we have more no. of a after first only a

we can keep a self loop.

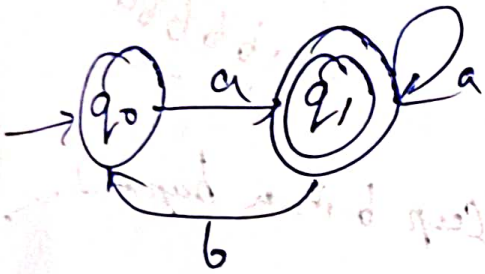
Next we need to consider q_1 on b.



If we keep it here state means it will not accept the string.

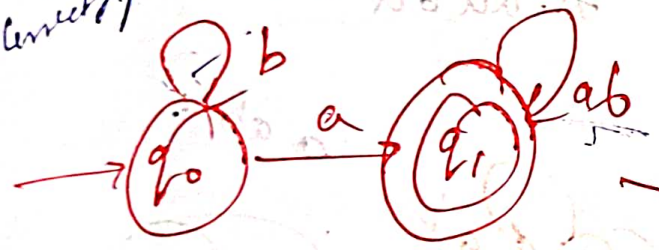
baab

aaba ✓



aab (it also not accepted bcoz b should be in final state. it is going to q_0.

the correct figure is



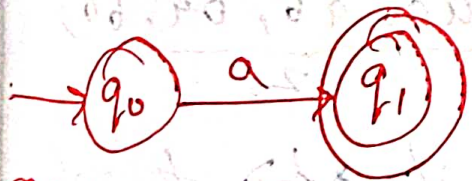
→ Contain 'a'

② End with 'a',

① language / lang is a collection of string.

$L = \{ a, aa, aaa, ba, baba, aba, abaaba, ab, abab \times \}$

minimum string with a.

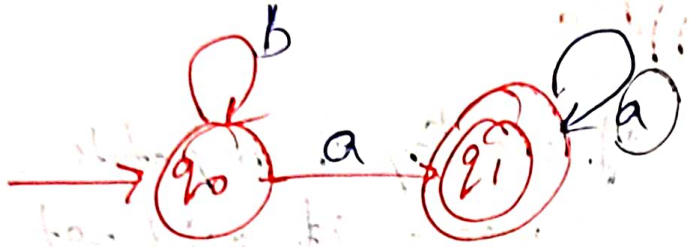


a ✓ is accepted.

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

④ fill in the blanks

ba (we can get b or more than b we can get b^n we need only this should end with a. (keep b in a self loop)

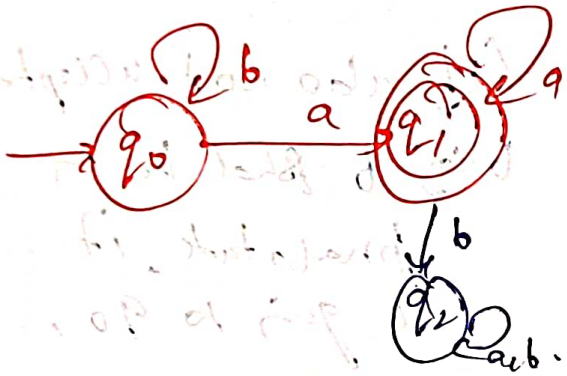


b a
b b b a
self loop

a a a a a
self loop

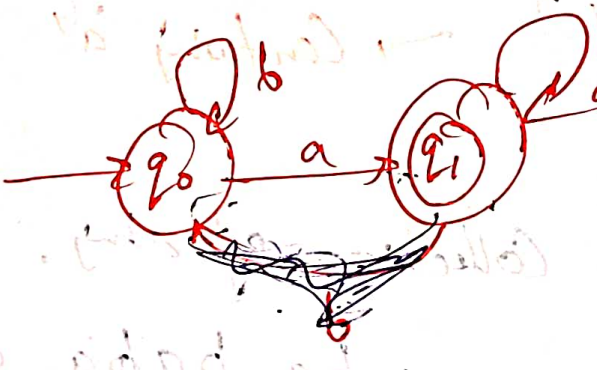
aabbbba

if string aba



if we keep b in a separate state
then aaba it will not allepha

finally

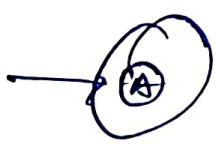


aaba

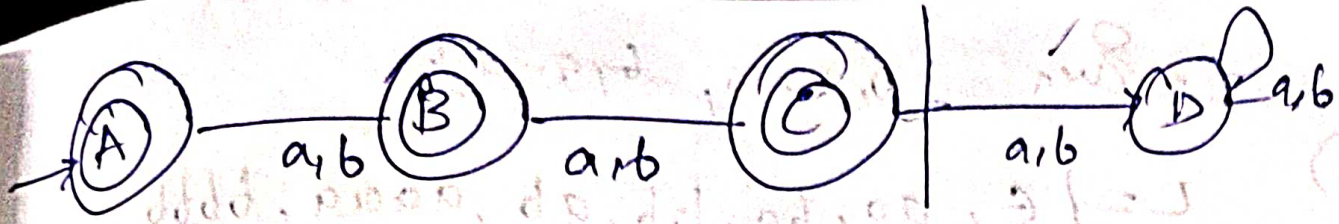


Construct DFA $\Sigma = \{a, b\}$ where length is atmost 2.

$L = \{\overset{0}{\in}, 1, 2\} = \{\underset{0}{\in}, \underset{}{a, b}, \underbrace{aa, ab, ba, bb}_{2}\}$



$L = 0$. there is no i/p string. that's why initial state is a final state.

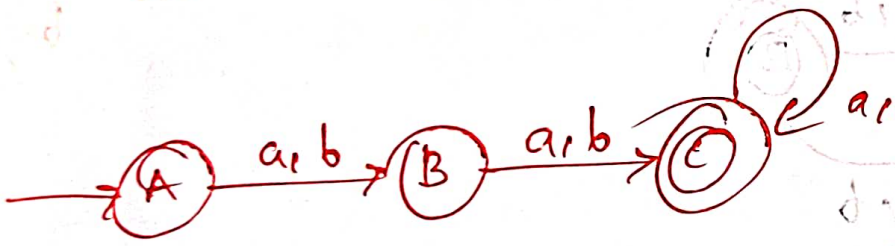


Till here the string is accepted. we need to satisfy the DFA restriction

Each and every state will accept i/p symbols (validate the DFA) it shd accept a, b only once. Here Σ is not accepting then we need a deadstate, D that shd accept {a, b}.

- $\Sigma = \{a, b\}$, where length is atleast 2 (i.e min of 2 after that it's no.

Sol: $L = \{\underbrace{aa, ab, ba, bb}_{L=2}, \underbrace{aaa, bbb, aba, bab}_{L=3}, \ldots\}$



aa — order
aaa — after self loop
aabab — self loops

Assignment

Hw

i. when the length is even — length {0, 2, 4, 6, 8 ...}
ii. when the length is odd {1, 3, 5, 7 ...}

$\Sigma = \{a, b\}$.

* Two Consecutive Zeros.

L = Even $\{0, 2, 4, 6, 8, \ldots\}$

A) $L = \{\epsilon, aa, ba, bb, ab, aaaa, bbbb, \ldots\}$ infinite lang.



back transition for a.

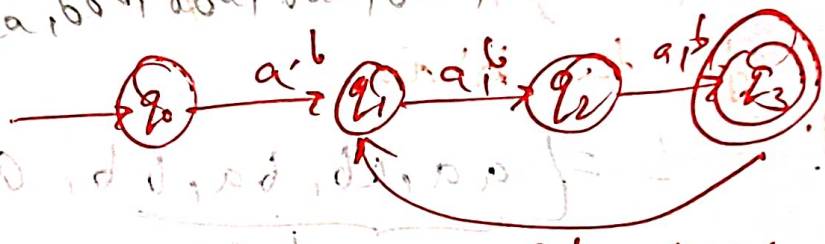
aaaaaa also accepted.

L = odd

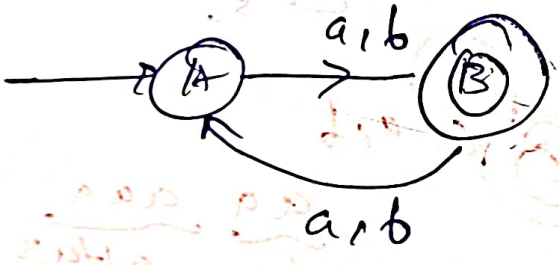
L = {

* multiples of 3 $\Sigma = \{a, b\}$

aaa, bbb, abb, bab, bba, aab, ...



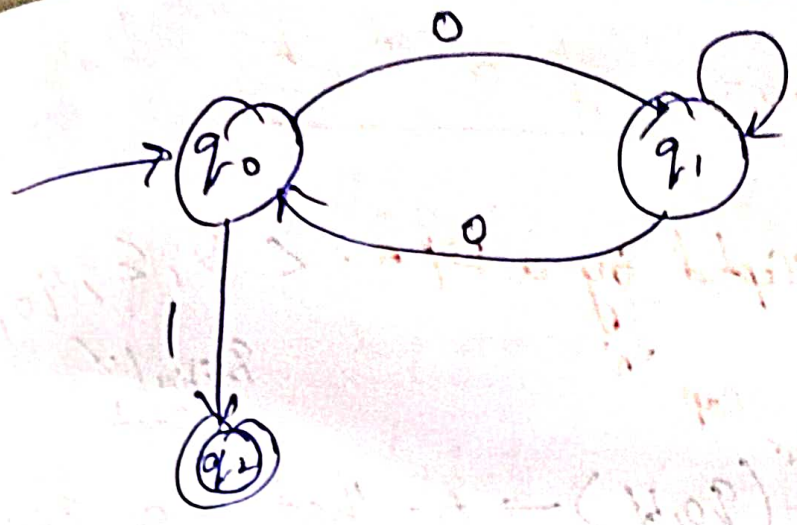
back transition to optimize the statement use.



* Construct a DFA which accept a language of all strings not starting with 'a' or not ending with '1'. L = {3, 6, 9, 12, 15 ...} If.

* Multiples of 3 over $\Sigma = \{a, b\}$.

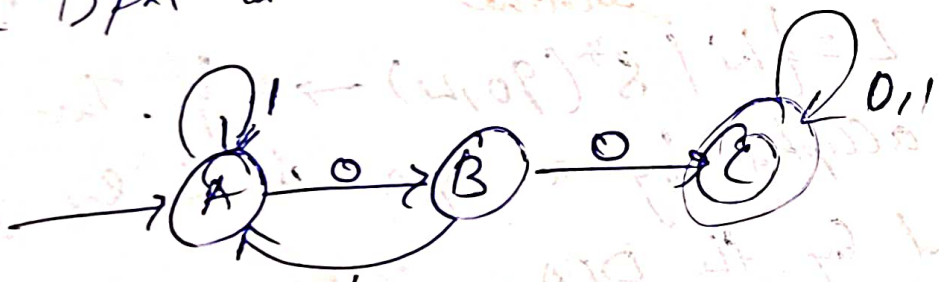
* Design a DFA with $\Sigma = \{0, 1\}$ accepts the strings with an even number of 0's followed by single 1.



* Design DFA to accept Strings over $\Sigma = \{0,1\}$ with two Consecutive 0's.

$L = \{00, 100, 001, 1001, \ldots\}$ Infinite language.

The DFA is



10100

The above DFA will be $M = \{Q, \Sigma, \delta, q_0, F\}$

where $Q = \{A, B, C\}$

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

δ = Transition function

$q_0 = A$

$F = C$.

* The language of a DFA

A string 'w' is accepted by a DFA. $\langle Q, \Sigma, q_0, \delta, A \rangle$

Cap 'A' — Initial state — final state

if and only if $\delta^*(q_0, w) \rightarrow A$. That is a string is accepted by a DFA if and only if the DFA starting at the initial state ends in an accepting state/final state after reading the string.

A Language 'L' is accepted by a DFA $\langle Q, \Sigma, q_0, \delta, A \rangle$ if and only if $L = \{ w \mid \delta^*(q_0, w) \rightarrow A \}$ such that. That is, the language accepted by a DFA is the set of strings accepted by the DFA.

- ϕ is an Empty language [ϕ - fire]
- $\{\epsilon\}$ Epsilon is a language for any Σ.
- But $\phi \neq \{\epsilon\}$

ϕ → this is an Empty string

$\{\epsilon\}$ → language (There is a string). A string with as Empty or 0 value. (null input)

{Empty string} (Empty transition)

ϵ (No transition)

ϕ

Operation of DFA

- Union $L_1 = \{aba\} \cup L_2 = \{bcd\}$
  $= \{abcd\}$
- Intersection $L_1 = \{aba\} \cap L_2 = \{bcd\}$
  $= \{b\}$
- Complement $L = \{\text{Even no's}\}$
  $\bar{L} = \{\text{odd no's}\}$ likewise.
- Relative Complement.

  Exclude B part we will Concentrate on A part.

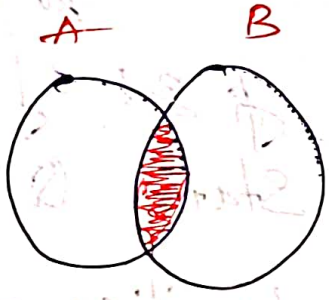


Venn-diagram

main

- Symmetric Difference

  Exclude the intersection part and summarize the Union part.



- Concatenation of language.

  $L_1 \circ L_2$ $L_1 = \{hello\}$ $L_2 = \{world\}$

  $L_1 \circ L_2 = \{hello\ world\}$.
- Reversal of a language

  $L_1 = abc$ $L_1^R = cba$.

NFA (Non-deterministic finite Automata).

(One-Many) transition.

A Non-deterministic finite Automata (M) is reverse of deterministic algorithm. It is a collection of five-tuple namely.

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

where.

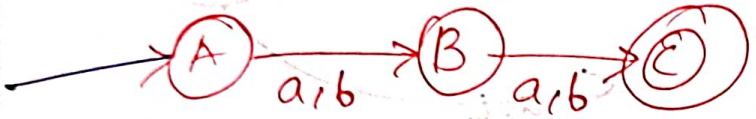
- Q = A finite set of States i.e., $\{q_0, q_1, q_2, \ldots\}$
- Σ = A finite set of input symbols including the input symbol ε.
- q_0 = An initial State or Start State, $q_0 \in Q$.
- F = The finite set of final States or accepting States. ($F \subseteq Q$)
- δ = A transition function defines the rules for change of the state. It takes two arguments a State and the input symbols and returns a subset of o/p States (more than one o/p State).

i.e $Q \times \Sigma \rightarrow 2^Q$

CONSTRUCTION OF A NFA (Examples)

Q1. Construct a NFA over $\Sigma = \{a, b\}$, whose the length is Equal to 2.

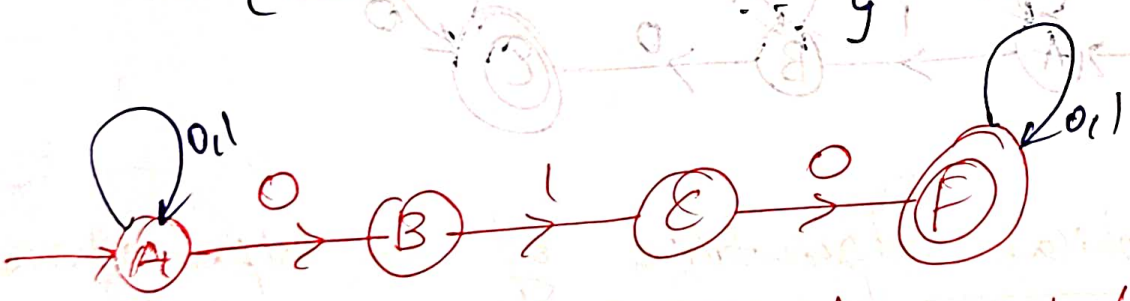
Sol The language will be : $L = \{aa, ab, ba, bb\}$



Q. Design a NFA that accepts Strings Containing 010 as a Substring.
(Existing string).

Sol The language will be

$L = \{010, 00\underline{010}11, 11111\underline{010}0, 10100\underline{1000}, \ldots\}$



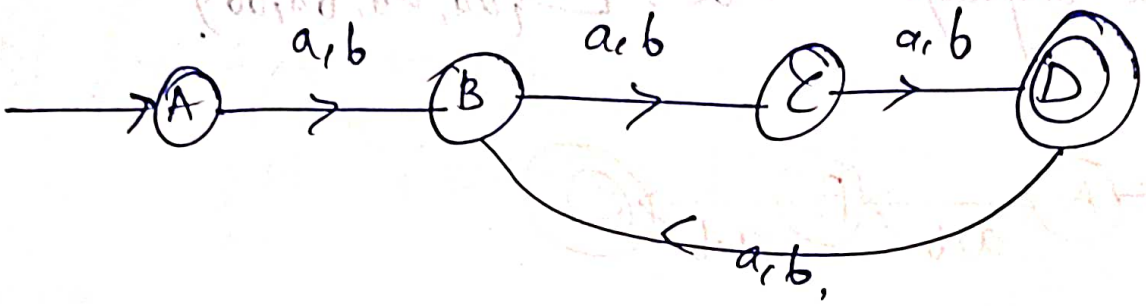
We can have more no. of no. of 0's and 1's b/4 010 or after 010 we are keeping self loop over them.

Draw a transition table for above NFA

δ	0	1
→ A	{A, B}	A
B	ϕ	C
C	F	ϕ
* F	F	F

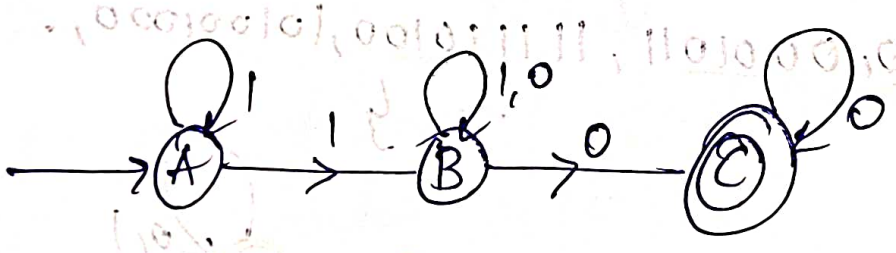
Eg 3 Multiples of 3 over $\Sigma = \{a, b\}$

Sol: $L = \{length(3), length(6), length(9) \dots\}$



Eg 4 Design a FA with $\Sigma = \{0, 1\}$ accepts those string which starts with 1 and ends with 0.

Sol: $L = \{10, 1110, 11010, \dots\}$



Graphical Representation of an NFA or NFSA

An NDFA is represented by digraphs called State diagrams.

→ The vertices represent the states.

→ The arcs labeled with an input alphabet = transitions.

- The initial State = Empty Single incoming arc.
- The final State = double circles.

1) Let a non-deterministic finite automata be →

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

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

$q_0 = \{a\}$

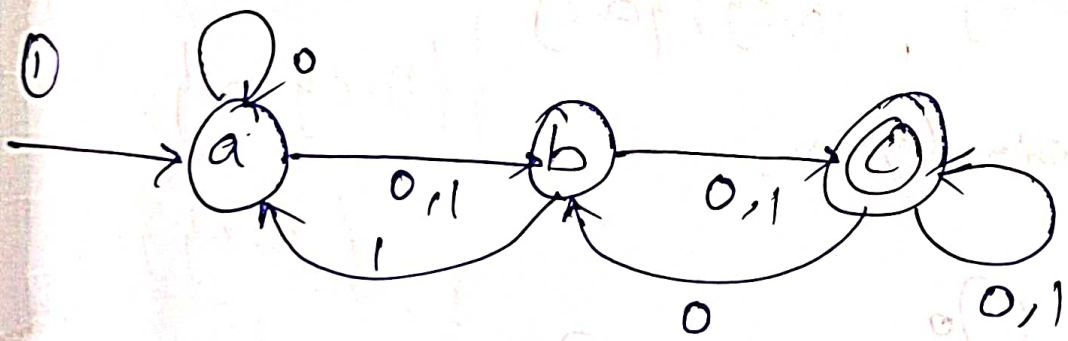
$F = \{c\}$

The transition function δ as shown below.

(2)

present state	0	1
→ a	a, b	b
b	c	a, c
c	b, c	c

(1)



NDFA graphical representation.

First they have given transition diagram of a NFA. Construct a transition table after that converting NFA table to DFA table at last we drew a diagram for DFA.

NFA – DFA (Conversion)

Transition table for DFA from NFA T.T.

Construct an Equivalent DFA from the following transition table.

δ	ε	0	1
→ A	A	A	B
B	B	{B,C}	B
*C	C	C	{B,C}

T.T for DFA

δ	ε	0	1
→ A	A	A	B
B	B	{B,C}	B
*{B,C}	{B,C}	{B,C}	{B,C}

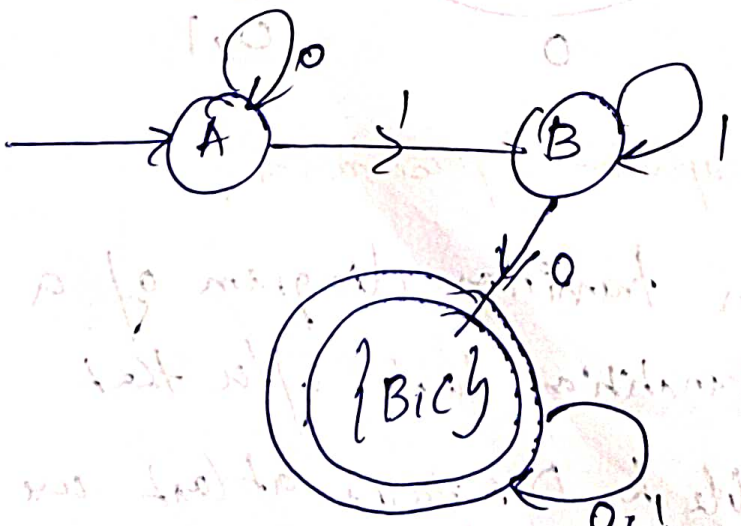
(See any new state is coming/appearing Yes 'B' is a new state.)

{{B,C},0} = {B,C} {C}

{B,C}

C is Common

Draw transition diagram



finally done.

NFA – DFA

Q. Find the Equivalent DFA for following model.



①

δ	ε	0	1
→ q0		q0	q1
q1		{q1,q2}	q1
*q2		q2	{q2,q1}

②

T.T. for NFA

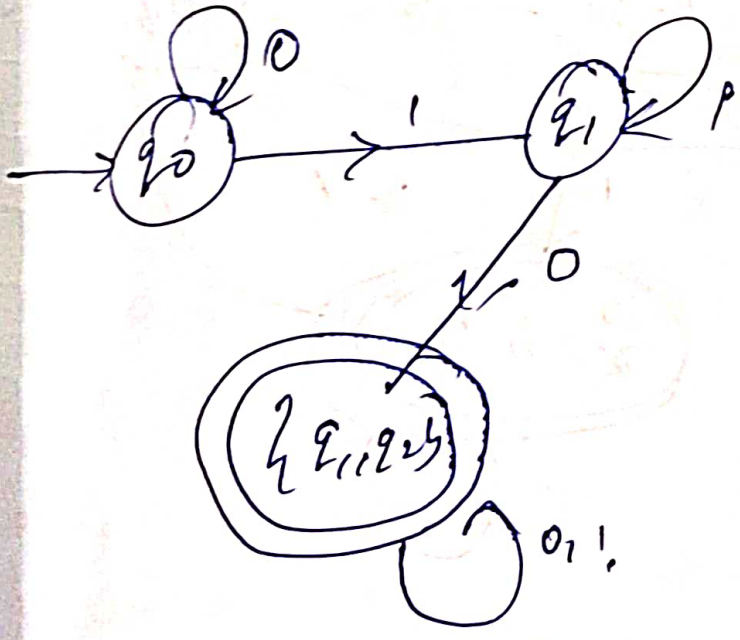
Convert NFA to DFA ③

T.T T.T

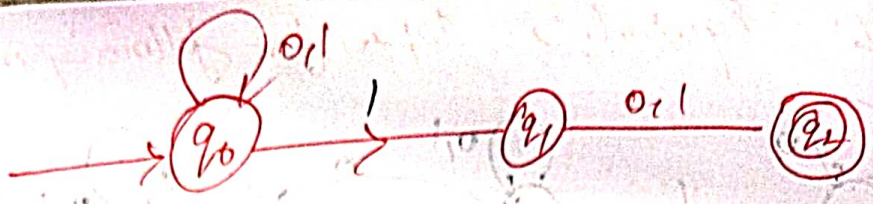
δ	ε	0	1
→ q0		q0	q1
*q1		{q1,q2}	q1
*{q1,q2}		{q1,q2}	{q1,q2}

T.T for DFA

Next draw a transition diagram.



③



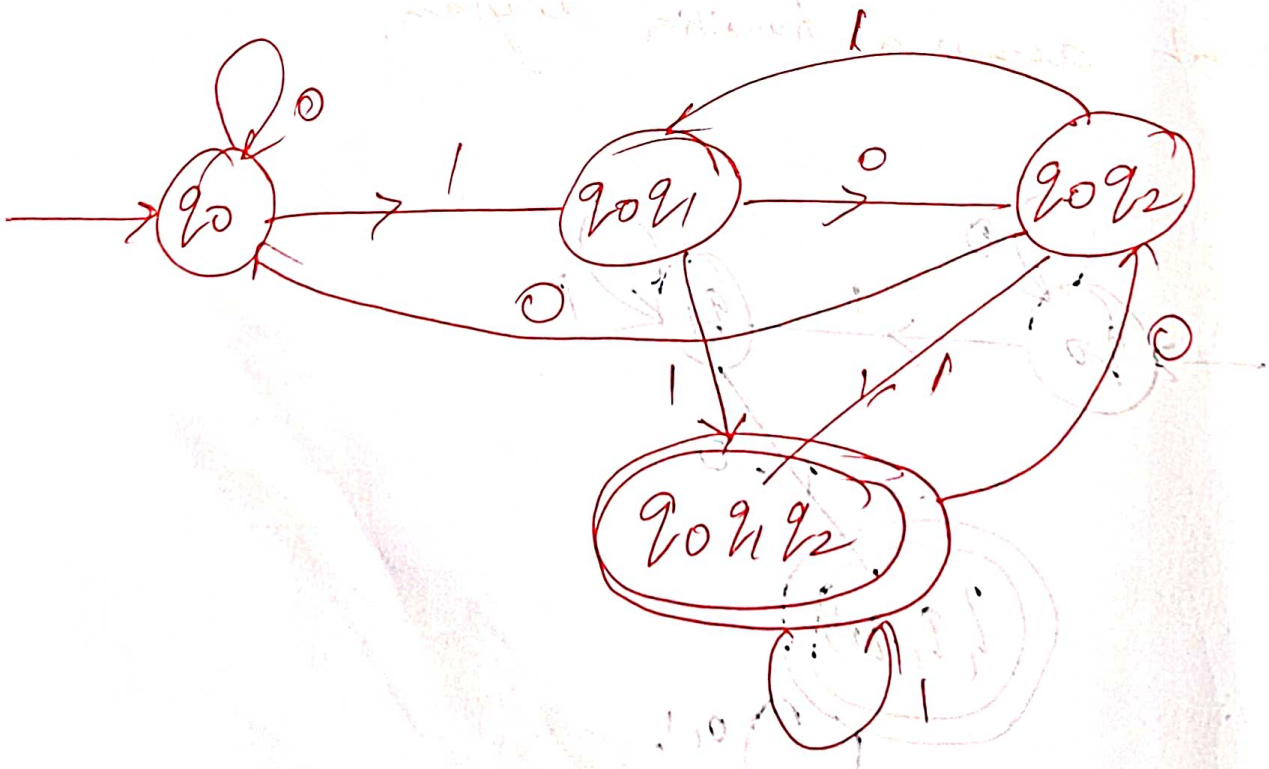
Construct table for NFA.

	0	1
→ q_0	q_0	$q_0 q_1$
q_1	q_2	q_2
q_2	–	–

⇒

Convert NFA T.T → DFA T.T.

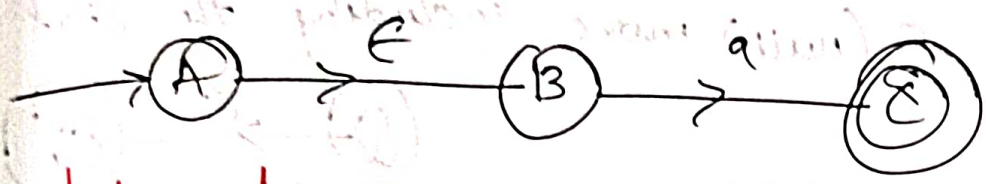
	0	1
→ q_0	q_0	$q_0 q_1$
$q_0 q_1$	$q_0 q_2$	$q_0 q_1 q_2$
$q_0 q_2$	q_0	$q_0 q_1$
$q_0 q_1 q_2$	$q_0 q_2$	$q_0 q_1 q_2$



DFA T. Diagram.

NFA With ε-Transition

- Every DFA is NFA (To satisfy the lang)
- But every NFA is not DFA.



Key features for NFA with ε-transition.

- In this type of machines changing of state is possible without reading the strings.
- Such transitions are labeled with ε (ε-Transition).

Note: 'ε' does not belong to any alphabet.

Formal description of ε-NFA ⟶

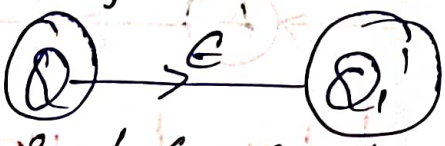
A ε-NFA is a 5 tuple quintuple (contains) $A = (Q, \Sigma, \delta, q_0, F)$

- Q, Σ, q_0, F
- $\delta: Q \times \Sigma\varepsilon \to Q$ is the transition fn.
- 'ε' is never a member of Σ.
- $\Sigma.\varepsilon$ is defined to be $(\Sigma \cup \varepsilon)$

See 'ε'- Epsilon is an Empty transition. Suppose Consider a transition diagram with 'ε' transition, So here have to optimize the transition we are able to get all the ε-transition together. (for that U need to Undistand ε-Closure),

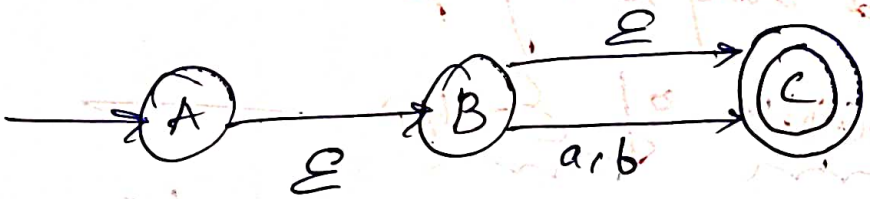
ε-closure

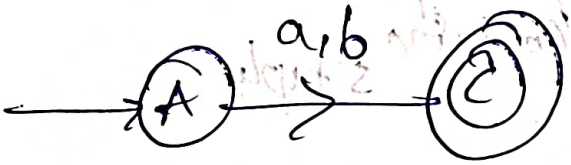
ε-closure for a given state 'Q' means a set of states which can be reached from the state 'Q' with only ε(null) move including the state 'Q' itself.



i.e. (Here we are going to consider from Q state how it is going to next state Q1' with including ε Epsilon (Empty/null moves) including the Q state)

Ex: A NFA accepting a language having length Exactly 1 over Σ={a,b}.



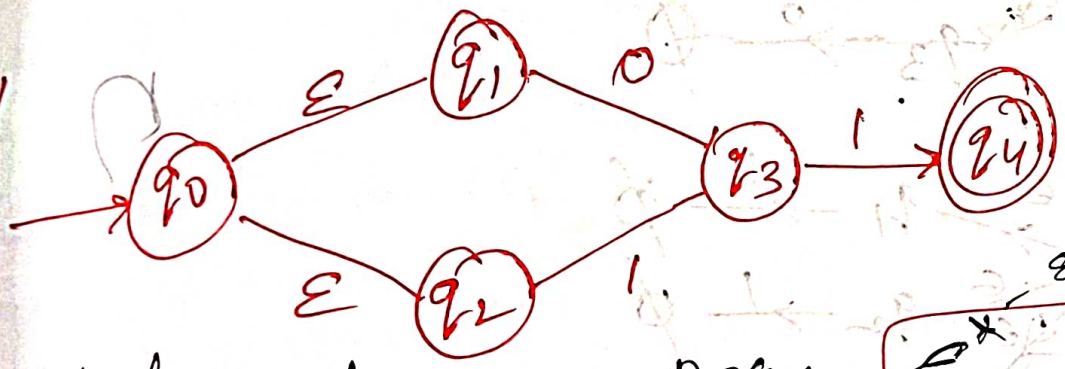


Steps in the design of a ε-NFA to DFA.

- Step1 :- Take 'ε'-closure for the beginning state of NFA.
- Step2 :- Find the states that can be traversed from the present for each i/p symbol (Union of transition value and their closures for each states of NFA

Present in Current State of DFA.

- **Step3** :- If any new State is found take it as Current State and repeat step 2.
- **Step4** :- Do repeat Step 2 and Step 3 Until no new state present in DFA transition table.
- **Step5** :- Mark the States of DFA wch Contains final State of NFA as final States of DFA.



To find ε-closure :- process

ε-closure

ϵ^* I/p ϵ^*

Note :- ε on Empty transition i.e ε on q_0 will not happen the transition diagram we shd take initial that as q_0 on ε. i.e.

q_0 —ε→ q_0 —0→ ∅.

q_0 —ε→ q_1 —0→ q_3 —ϵ^*→ q_3.

q_0 —ε→ q_2 —0→ ∅

q_0 —ε→ q_0 —1→ ∅

q_0 —ε→ q_1 —1→ ∅

q_0 —ε→ q_2 —1→ q_3 —ϵ^*→ q_3

	0	1
→ q_0	q_3	q_3
q_1	q_3	ϕ
q_2	ϕ	q_3
q_3	ϕ	q_4
q_4	ϕ	ϕ

$q_1 \xrightarrow{\epsilon} q_1 \xrightarrow{0} q_3 \xrightarrow{\epsilon} q_3$

$q_1 \xrightarrow{\epsilon} q_2 \xrightarrow{0} \phi$

$q_1 \xrightarrow{\epsilon} q_3 \xrightarrow{0} \phi$

$q_1 \xrightarrow{\epsilon} q_1 \xrightarrow{1} \phi$

$q_1 \xrightarrow{\epsilon} q_2 \xrightarrow{1} \phi$

$q_1 \xrightarrow{\epsilon} q_3 \xrightarrow{1} \phi$

$q_2 \xrightarrow{\epsilon} q_2 \xrightarrow{0} \phi$

$q_2 \xrightarrow{\epsilon} q_3 \xrightarrow{0} \phi$

$q_2 \xrightarrow{\epsilon} q_2 \xrightarrow{1} q_3 \xrightarrow{\epsilon} k$

q_3

$q_3 \xrightarrow{\epsilon} q_3 \xrightarrow{0} \phi$

$q_3 \xrightarrow{\epsilon} q_4 \xrightarrow{0} \phi$

$q_3 \xrightarrow{\epsilon} q_3 \xrightarrow{1} q_4$

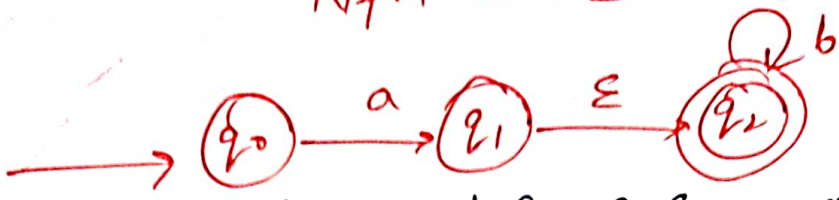
$q_3 \xrightarrow{\epsilon} q_4 \xrightarrow{1} \phi$

$q_4 \xrightarrow{\epsilon} q_4 \xrightarrow{0} \phi$

$q_4 \xrightarrow{\epsilon} q_4 \xrightarrow{1} \phi$

Deterministic finite Automata DFA	Non-deterministic finite Automata NFA / NDFA
• DFA stands for deterministic finite Automata.	• NFA stands for Non-deterministic finite Automata.
• for Each Sigma value (0,1....n) there must be provided path from Each Stat.	• for Each Sigma value (0,1....n), there is no need to Specify the path from Each state.
• DFA provides a path for Each i/p, so all digital Computers are deterministic	• There is more than one possible transition from one state on the same input Symbol. It means multiple choices are available Corresponding to a Single i/p.
• There is only one possible transition from one state on the one i/p Symbol. It mean multiple choices Corresponding to an input are not available.	• NFA does not provide a path for Each i/p, so NFA is not used in digital Computers.
• DFA Cannot use Empty String (Epsilon) transition.	• NFA Can use an Empty String (Epsilon) transition.
• DFA is more difficult to Construct & understand	• NFA is Easier to Construct & understand.
• All DFA are NFA	• All NFA are not DFA.
• DFA Requires more Space B'Coz it has normally more states.	• NFA Requires less Space than DFA. B'Coz it usually has fewer states.
• Deadstate are more Common in DFA	• Dead state are less Common in NFA.
• DFA-NFA Conversion, time & Space Complexity remains	• NFA-DFA Conversion, time & Space Complexity Increased to

NFA with ε to NFA without ε.



first obtain ε-closures of q0, q1 & q2 as follows.

1. ε-Closure(q0) = {q0}
2. " (q1) = {q1, q2}
3. " (q2) = {q2}

Now the δ' transition on each i/p symbol is obtained as

$\delta'(q_0, a) = \varepsilon\text{-closure}(\delta(\delta'(q_0, \varepsilon), a))$ — $\delta'(q_0, \varepsilon)$ - ε-closure (q_0).

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

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

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

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

$= \emptyset$

q1

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

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

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

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

$= \varepsilon\text{-closure}(\emptyset \cup \emptyset) = \emptyset$

$\delta'(q_1, b) = \varepsilon\text{-closure}(\delta(\delta'(q_1, \varepsilon), a))$

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

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

q2

$\delta'(q_2, a) = \varepsilon\text{-clos}(\delta(\delta'(q_2, \varepsilon), a))$

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

$= \varepsilon\text{-clos}(\delta(q_2, a)) = \varepsilon\text{-clos}(\emptyset) = \emptyset$

$\delta'(q_2, b) = \varepsilon\text{-clos}(q_2) = q_2$

$\delta'(q_0, a) = \{q_0, q_1\}$

$\delta'(q_0, b) = \emptyset$

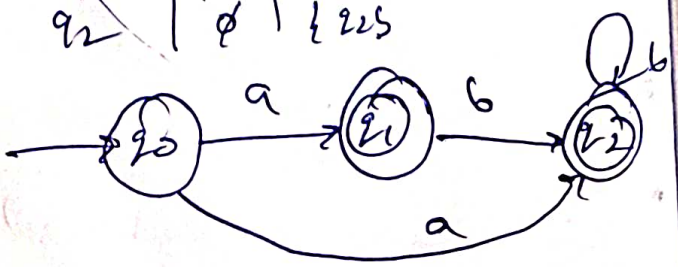
$\delta'(q_1, a) = \emptyset$

$\delta'(q_1, b) = \{q_2\}$

$\delta'(q_2, a) = \emptyset$

$\delta'(q_2, b) = \{q_2\}$

	a	b
→ q_0	$\{q_1, q_2\}$	$\emptyset$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\{q_2\}$



UNIT-2

Regular Expression :- A Regular Expression is a concise notation for denoting Regular Sets. Regular Expression describes the language accepted by a finite Automata.

Let Σ be a finite alphabet. Then the Regular Expressions over Σ denoting the Regular Sets are recursively defined as follows. (properties)

① ϵ is RE then the corresponding to the language $L = \{\epsilon\}$ i.e., string is ϵ Regular Set. Regular set of ϵ

② ϕ is RE then the corresponding to the $L = \{\}$ i.e., empty

③ x is RE " " " " " $L = \{x\}$

If 'x' is a RE over language $L(x)$
'y' is a RE over language $L(y)$

⇒ i) $x+y$ is RE Corresponding/denoting to $L(x) \cup L(y)$, where $L(x \cup y) = L(x) \cup L(y)$. (union)

⇒ ii) xy is RE Corresponding/denoting to $L(x) \cdot L(y)$, where $L(xy) = L(x) \cdot L(y)$. (Concatenation)

⇒ iii) R^* is RE Corresponding to $L(R^*) = (L(R))^*$ (star (closure) or kleen closure)

Regular Set :- Let Σ be a finite alphabet, then a class containing set of strings over Σ is called regular set. A regular set is recursively defined as follows.

(a) the Empty set $\{\}$ is a regular set over Σ.

(b) the string of length 0 is a regular set over Σ i.e $\{\epsilon\}$.

(c) Each i/p symbol say 'a' in Σ is a regular set over Σ.

(d) If P and Q are two regular sets over Σ.
- $P \cup Q$ is a Regular set over Σ,
- PQ " " " " " "
- P^* " " " " " "

Examples –

1. Let $\Sigma = \{0\}$, then the set of string $\{0, 00, 000 \ldots\}$ is a regular set.
2. Let $\Sigma = \{0, 1\}$, then the set of strings $\{01, 10\}$ is a regular set.
3. Let $\Sigma = \{a, b\}$, then the set of strings starting with a and ending with b is a regular set.

Regular language :- Regular Language is a formal language generated (or accepted) from regular sets over input alphabet Σ.

Example If $\Sigma = \{a, b\}$ then all set of strings that a obey the condition (i.e Even no. of a's) is a regular

language.
i.e the regular language (L) = {aa, aab, abab...}

Properties of RL —

(1) A RL is accepted by NFA, DFA (Exp. Land dead-any) Turing Machine]
(2) A RL is described by Regular Expressions.
(3) A RL generates a regular grammar.

Application of Regular Expressions :—

Regular Expressions have a wide variety of applications ranging from Compilers to Search Engines, thus are widely used in lexical analysis and Software Engineering applications.

(1) Regular Expressions are used in web Search Engines to Search the keywords through a database of filenames on FTP Servers (File transfer protocol).
(2) They are most useful in Smaller Search Engines like find/replace tool in text Editor.
(3) RE plays a major role in Sftw Engineering for Ensuring reusability, for testing, there a relation is Established b/w finite Automata & regular language.
(4) the most important area where regular Expressions

are used in lexical analysis. It is a process of dividing i/p program into sequence of tokens.

Algebraic laws (Identity laws) of Regular Expressions :-

Algebraic laws of regular Expressions provide a set of rules and properties that describe how regular Expressions can be mathematically simplified. These laws are fundamental fields of theoretical computer science and formal language theory. Think $\phi = 0$ In R.E $\epsilon = 1$ (indicates) $\Sigma = R.E$

① Identity laws ——

- Union identity: $R \cup \phi = R$
- Concatenation identity: $R . \epsilon = \epsilon . R = R$

② Domination Laws ——

- Union domination: $R \cup \Sigma^* = \Sigma^*$
- Concatenation domination: $R . \phi = \phi . R = \phi$

③ Idempotent Laws ——

- Union idempotent — $R \cup R = R$
- ~~Concatenation idempotent — Not applicable in RE~~ (by using Kleene Closure it is applicable $L(R)^* = (L(R))^*$.

④ Commutative Laws ——

Union Commutativity — $R \cup S = S \cup R$

⑤ Associative Laws ——

- Union Associativity — $(R \cup S) \cup T = R \cup (S \cup T)$
- Concatenation Associativity — $(R . S) . T = R . (S . T)$

6. Distributive Laws:

Concatenation Distributes Over Union,

- Left distribution $R.(S \cup T) = (R.S) \cup (R.T)$
- Right distribution $(R \cup S).T = (R.T) \cup (S.T)$

7. Annihilation Laws:

Union Annihilation $R \cup R^* = R^*$

Concatenation Annihilation $\emptyset \cdot R = \emptyset$ (Annihilation)

8. Kleene Star Laws:

- Kleene Star of Empty Set: $\emptyset^* = \epsilon$

The Kleene star of Empty Set is Empty String.

- Kleene Star of Empty String $\epsilon^* = \epsilon$

The Kleene star of the Empty String is Empty String

- Double Kleene Star $(R^*)^* = R^*$

The Kleene Star applied twice is the same as applying it once

- Concatenation with Kleene star

$R.R^* = R^*.R \cup \epsilon = R^+$ (Concatenation a regular Expression with its Kleene Star is Equivalent to its positive Closure

9. Absorption Laws:

- Union Absorption: $R \cup (R.S) = R$
- Concatenation Absorption: $R.(S \cup \epsilon) = R \cup (R.S)$

Conversion of finite Automata to Regular Expression.

B4 that learn about how to write a RE by a question.

Ex

① RE denoting RL with strings having any no. of a's and any no. of b's.

$$RE = (a+b)^*$$

② RE denoting the language with string starts with 'a' and end with 'b'

$$RE = a(a+b)^*b$$

③ All the strings of 0's and 1's

$$L = \{ \epsilon, 0, 1, 00, 01, 10, 100, 110 \cdots \}$$

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

④ 0's & 1's ends with 00

$$L = \{ 00, 100, 000, 1100, 10100, \cdots \}$$

$$RE = (0+1)^*00$$

⑤ Begin with '0' and Ends with '1'

$$L = \{ 01, 001, 011, 00011, 01001, \cdots \}$$

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

Conversion FA to RE

By using Arden's Theorem we are going to Convert.

FA to RE (two ways)

- State Elimination Alg (by simply Eliminate the states one by one and where only initial & final state will be there and transition we will be getting that is the final Expression.
- Arden's Theorem Alg (AT use some part of Simplification of a Exp by using Certain Exp of its own form).

THEOREM: ARDEN'S THEOREM States that —

If P and Q are two regular Expression over Σ, and if P does not Contain Σ, then the following Equation in R given by: $R = Q + RP$ has an unique Equal Solution i.e., $R = QP^*$.

That means, whenever we get any Equation in the form of $R = Q + RP$, then we can directly replaced by $R = QP^*$. So, here first

we will also prove that it is the Unique Solution of this Equation.

Eg — $q1 = q1 \cdot 0 + e$;

$q2 = q1 \cdot 1 + q2 \cdot 0$;

$q3 = [q2 \cdot 1 + q3 \cdot 0 + q3 \cdot 1$

Now,

$$\frac{q1}{R} = \frac{q1 \cdot 0}{R\,P} + \frac{\in}{Q} \quad - (1)$$

$R = Q + RP$ $\therefore q1 = \in + q1 \cdot 0$

$R = Qp^2$ $q1 = \in \cdot 0^*$ {By Arden's theorem}

$q1 = 0^*$ $(\in \cdot R = R \cdot \in = R)$

$\therefore q2 = 0$

Substituting in — (2)

$$\frac{q2}{R} = \frac{0^* \cdot 1}{Q} + \frac{q2 \cdot 0}{R\,P}$$

$Qp^2 = 0^* \cdot 10^2$

$q2 = 0^* \cdot 10^2$

this is how we are going to Simplify by using Arden's theorem.

Proof – Arden's Theorem

Let's start by taking following Equation as Equation

$$R = Q + RP \quad \text{---(1)}$$

Now Replacing R by $R = Qp^*$ we get.

$$R = Q + Qp^*p$$

Taking Q as Common,

$$R = Q(\varepsilon + p^*p)$$

$(1+p^*p)$ → (is replaced by ε as by identity II)

(As we know by identity law, $\varepsilon + RR^* = \varepsilon + R^*R = R^*$)

$$R = Qp^*$$

Thus $R = Qp^*$ is the Solution of the Equation $R = Q + RP$.

Conversion of FA to RE by Arden's Theorem

Steps:

1. Transition diagram should not have Epsilon transitions.
2. Must have only one Single Initial State.
3. Vertices are Suppose $q_1, q_2, q_3 \ldots q_j \ldots q_n$, where q_i is the final State

④ Wij denote the regular Expression representing set of labels of Edges from q_i to q_j.

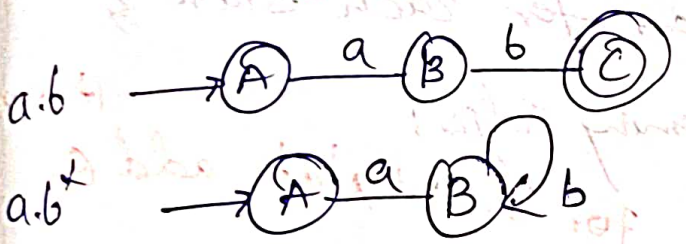
⑤ Write the Equation for each State and check their incoming Edges.

⑥ Take the final State and Simplify the Equation for the Same final State in the form of Arden's theorem Expression.

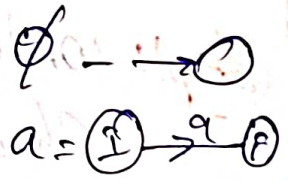
By this we will see the Conversion Example of RE to FA.

① Design the Equivalent FA – RE

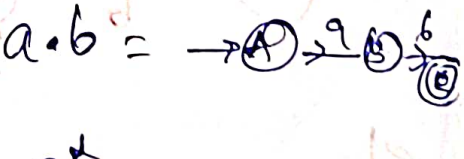
ab^*.



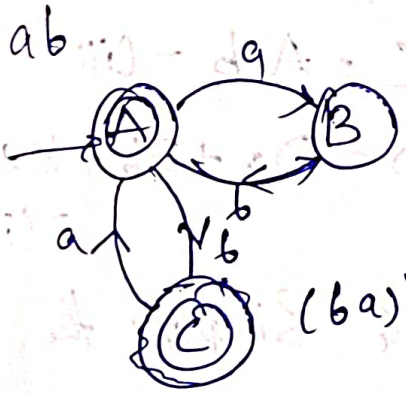
B and Symbol



It should go in a same direction



② $(ab+ba)^*$

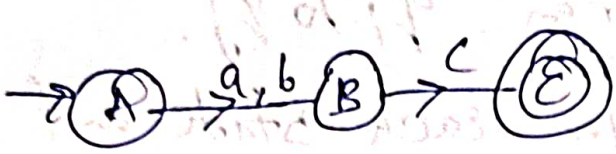


$(ab)^*$ + $(ba)^*$ = $(ab+ba)^*$

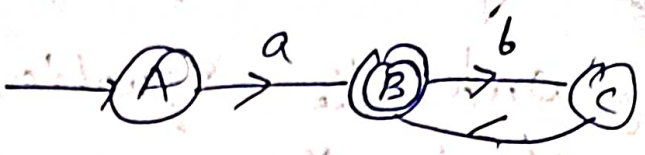
a^* = v = 0, 1, 2, 3, ∞



③ $(a+b)c$



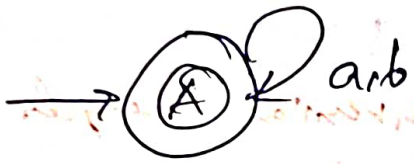
④ $a(bc)^* = a.(bc)^*$



⑤ $(a+b)^*$ ~~(aa+bb+(ab+ba)(ab+ba))~~

~~Kleene~~ Kleene closure theorem.

Step 1: $(a+b)^*$



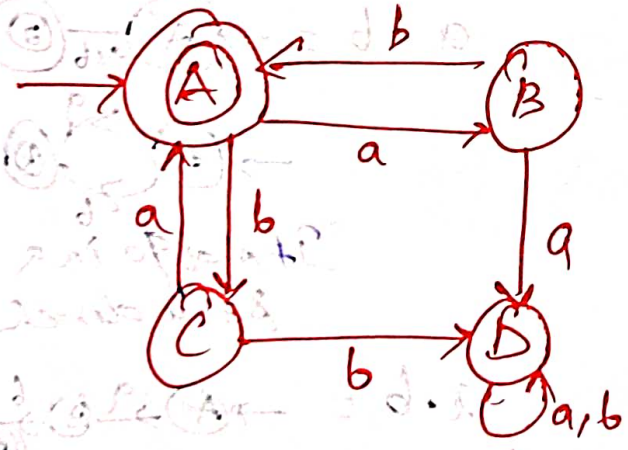
FA – RE

---

① Derive the RE from finite Automata.

Step 1: Write the Equation for Each State &
Check their incoming Edges.

(Always) For Initial state add ε



$A = \varepsilon + Bb + Ca$ —(i)

$B = Aa$ —(ii)

$C = Ab$ —(iii)

$D = Da + Db + Ba + Cb$ —(iv)

Step 2: Take the final State & Simplify the Equation for the same final State in the form of Arden's theorem

Here we need to focus on final state.

i.e $A = \in + Bb + Ca$

by substituting (ii) & (iii).

$A = \in + Aab + Aba$

$A = \in + A(ab+ba)$

$R \quad Q \quad R \quad P$

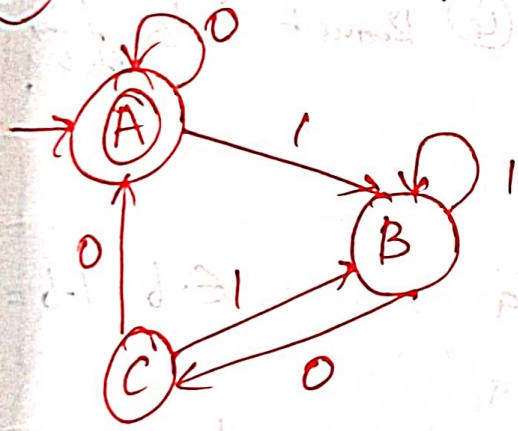
$R = Qp^*$

$A = \in \cdot (ab+ba)^*$

i.e $R = Q + Rp$

$R = Qp^*$

(Q2) Derive RE from finite Automata,



Step1:

$A = \in + A0 + C0 \quad -(1)$

$B = B1 + A1 + C1 \quad -(2)$

$C = B0 \quad -(3)$

Now we need to Concentrate (target) on final state i.e A.

See by substituting C in A and again B in A it's going on increasing B value (think logic), Therefore first we will Simplify B.

$B \Rightarrow B1 + A1 + C1$ sub (3) in (2)

$= B1 + A1 + B01$

$= A1 + B1 + B01$

$R \quad Q \quad P$

$R = B = A1 + B(1+01)$

i.e $B = A1(1+01)^*$

$A = \epsilon + A0 + C0$

$= \epsilon + A0 + B00 \rightarrow$ (3) in (1)

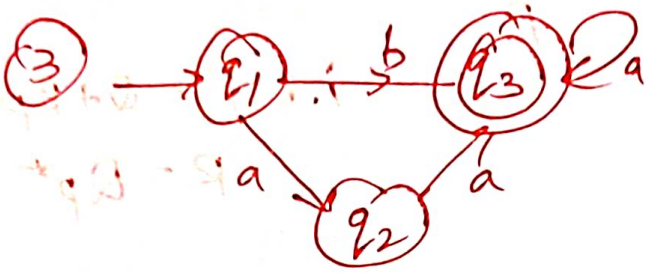
$A = \epsilon + A0 + A1(1+01)^*00$

$A = \epsilon \cdot (0 + 1(1+01)^*00)^*$

↓

Final Exp

$\epsilon = 1$



① Identify all incoming Edges for each state

② for initial state add ϵ

③ Calculate all the Equations

④ Result is the final state.

① $q_1 = \epsilon$

$q_2 = q_1 a$ — ②

$q_3 = q_1 b + q_2 a + q_3 a$

put q_1 & q_2 values in q_3

$\Rightarrow q_3 = \epsilon b + q_1 a a + q_3 a$

$= \epsilon b + q_1 aa + q_3 a$

$= \epsilon b + \epsilon aa + q_3 a$

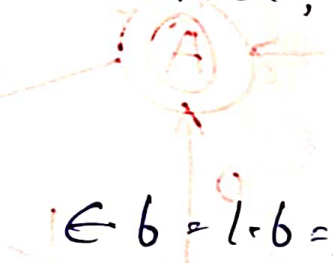
$q_3 = \epsilon(b + aa) + q_3 a$

$q_3 = \underline{b + aa} + q_3 a$

$R \quad Q \quad + R P$ → by using Arden's theorem

$R = Qp^*$ $\quad \boxed{q_3 = (b + aa)a^*}$

Result final state is the solution.

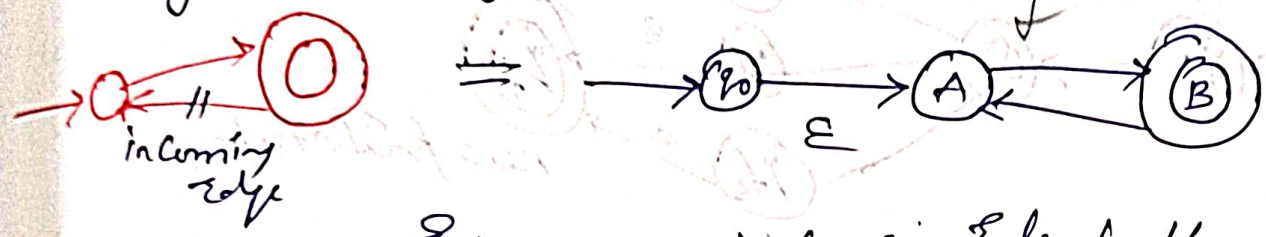


$\epsilon b = 1 \cdot b = b$

Pumping Lemma for Regular Languages

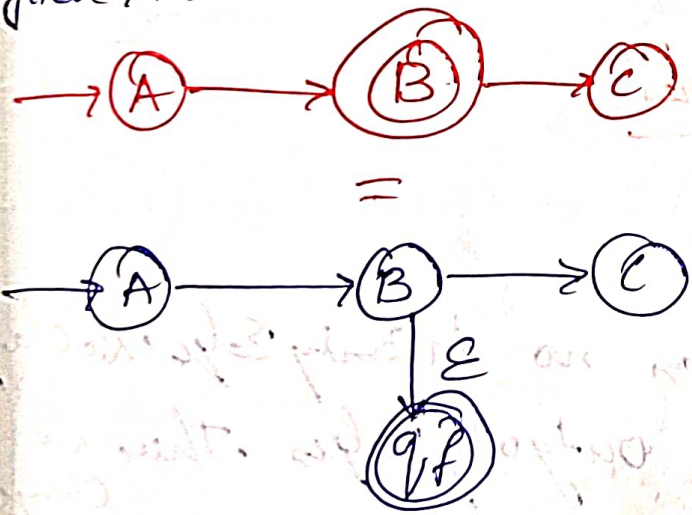
Converting FA/DFA → RE by State Elimination method: (we need to eliminate states at last we have initial state & final state)

Rule-1: INITIAL State should not have any Incoming Edge:



If there exists any Incoming Edge to the initial state then create a new initial state (q_0) having no Incoming Edge to it.

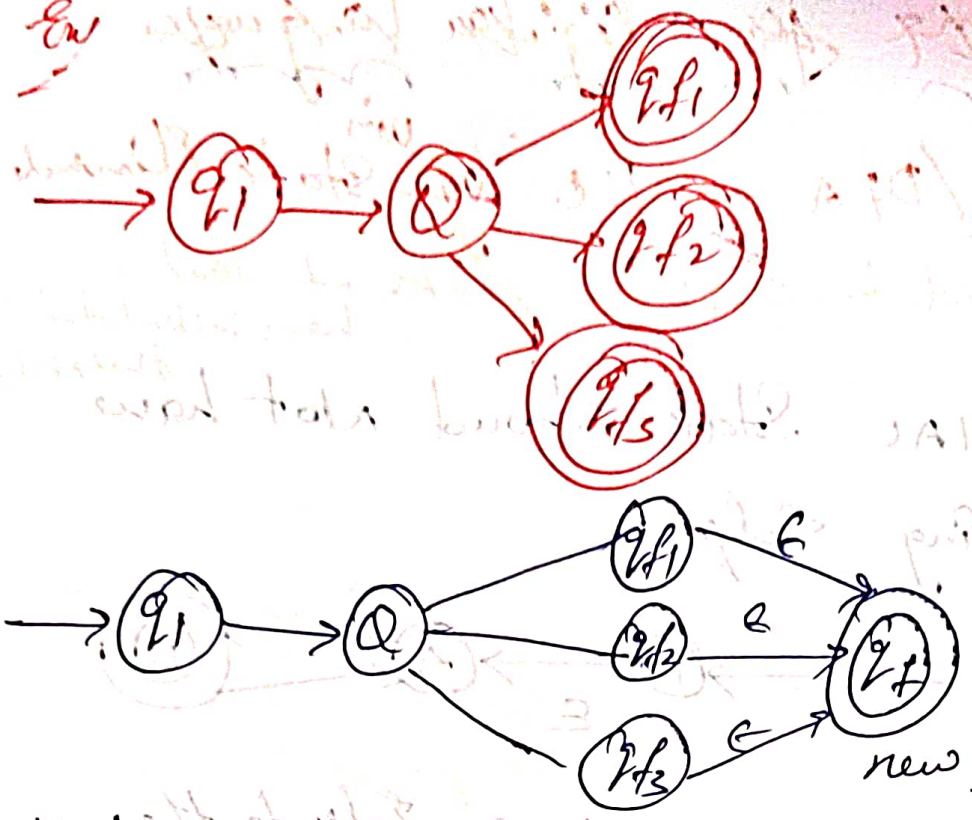
Rule-2: Final State should not have any (a) outgoing Edges. (There must exist only one final state), in FA/DFA.



(b) (or)

If there exists multiple final states in the DFA, then convert all the final states into non-final states and create a new single final state

Ex



Rule

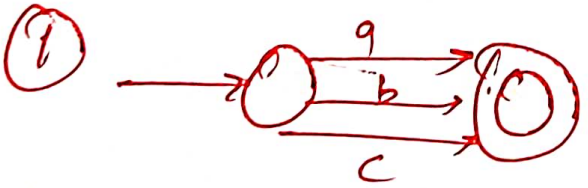
Step 3 → Other than the initial state & final state delete / Eliminate other states one by one. (These states may be Eliminated in any order).

In the End.

→ only an initial state going to the final state will be left.

→

Examples

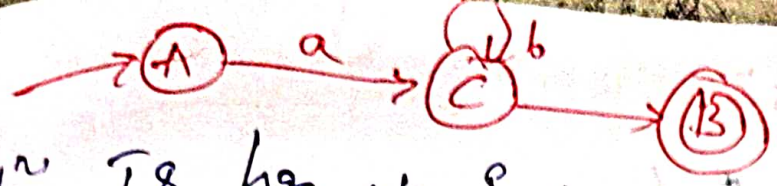


Sol'n. Initial State has no incoming Edges / No change

Final state has no outgoing Edges. Thus No change

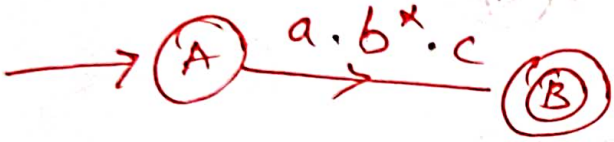
the final R.E = (a+b+c).

(2)

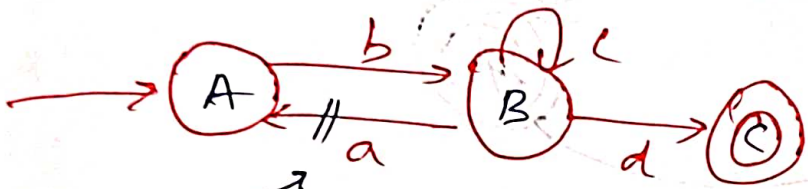


Soln: If has no Incoming Edge! No change.
If " " Outgoing " " "

(2) Eliminate Other states i.e Eliminate 'C'.
(See here C Containing Self loop b. i.e b^* an same other state Exept It & If then we need to write Exp as ab^*c



(3)

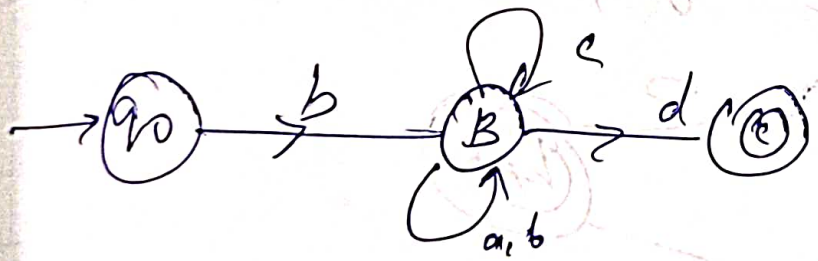


Rule1: By using 1 Rule,



Rules satisfied

Rules: Eliminate state (you can Eliminate in any order Either A or B)



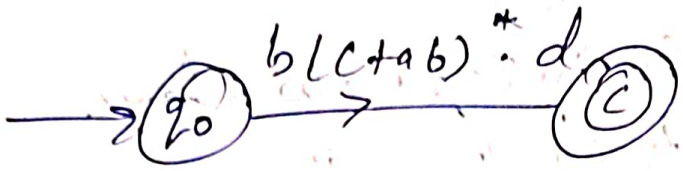
Eliminate 'A'

If we have a loop we will make it into single one.



(Here if both are Coming to same state and going to another state we use Union operator. (+).

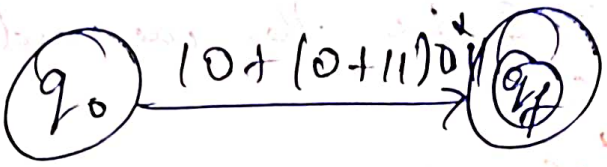
Eliminate "B"



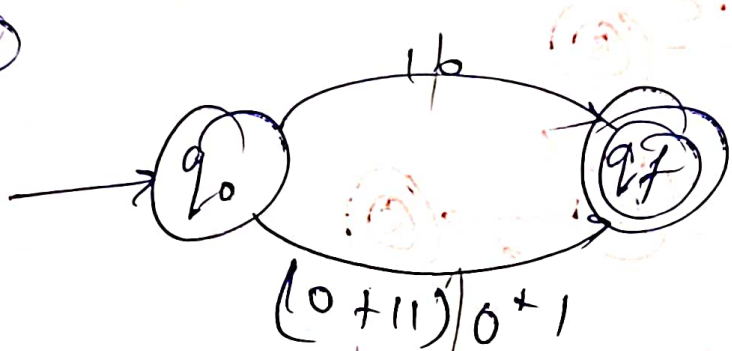
$\Rightarrow b(c+ab)^*.d$

Design a FA for RE $10+(0+11)0^*1$

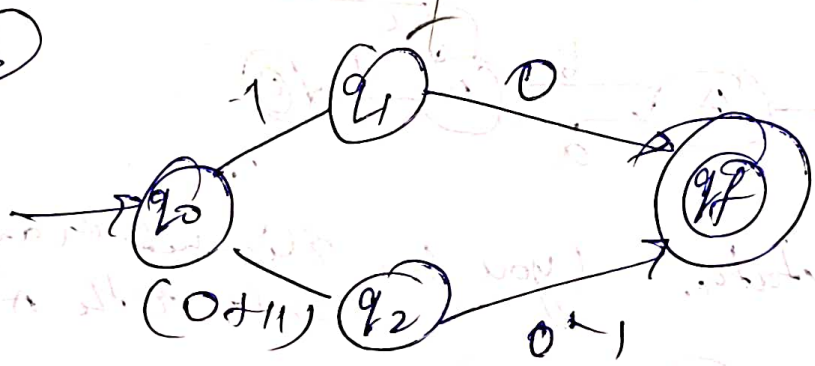
①



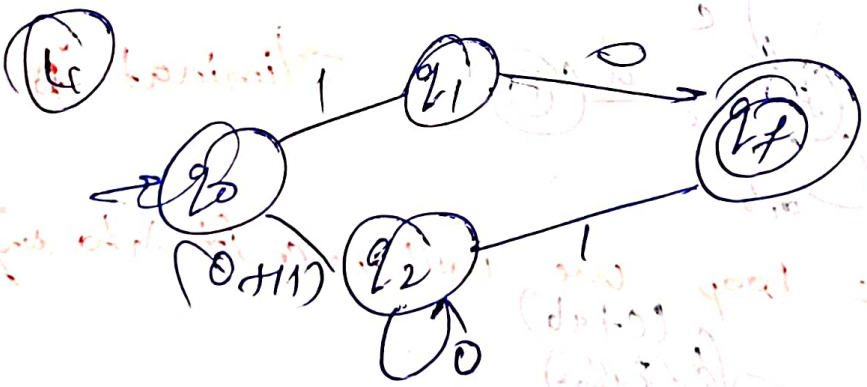
②



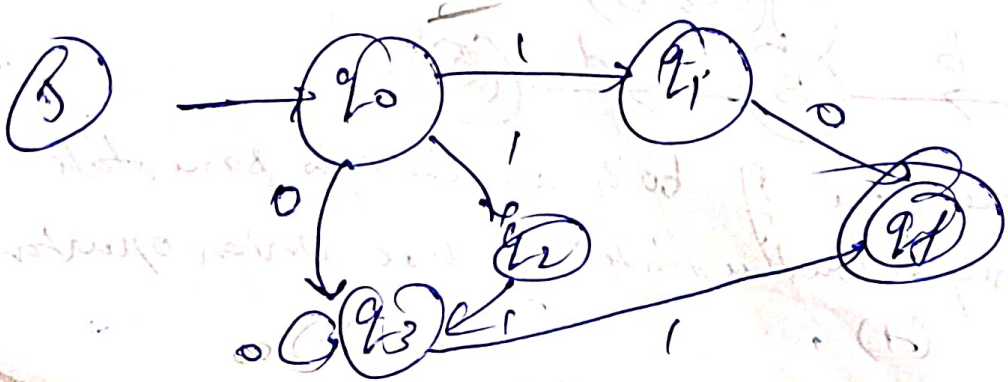
③



④



⑤



b^*

Pumping Lemma for Regular Languages

Regular Language – The language accepted by finite automata is known as Regular language.

Note:
1. $\{a^n / n >= 0\}$ if power = 1 = always Regular.
2. 2 different powers = always Regular.
   $\{a^n b^m / n, m >= 1\}$
3. $\{a^n b^n / n >= 1\}$ → Infinite → = Not Regular / Same power.
   $\{a^n b^n / n <= 203\}$ → Finite L – Regular.

Pumping Lemma – Algorithm:

pumping Lemma is used to prove that a language is not Regular. It cannot be used to prove that language is Regular.

There are two pumping lemmas, that are defined for

- Regular languages
- Context-free languages.

Pumping Lemma for Regular languages

Theorem – If 'A' is a Regular language, then 'A' has a pumping length 'p' such that any string 's' where $|s| \geq p$ may be divided into three parts $s = xyz$. Such that the following

Condition must be true.

1) $xy^iz \in A$ for every $i \geq 0$
2) $|y| \geq 0$
3) $|xy| \leq p$

Pumping lemma is used as proof of the irregularity of a language. It means that if a language is regular, it always satisfies the pumping lemma. If atleast one string is made from pumping, not in language 'A', then 'A' is not regular.

Example

Pumping lemma Example/proof ②

Q. Using pumping lemma prove the following language is not Regular $L=\{a^n b^n / n \geq 0\}$

① First it follow some steps

(1) Assume that language A is a Regular.
(2) It has to have a pumping length (Say p)
(3) All strings longer than p can be pumped $|s| \geq p$
(4) Now, find a string 's' in A such that $|s| \geq p$
(5) Divide s into xyz strings

(6) Show that $xy^iz \notin A$ for some i.
(7) Then consider how s can be divided into xyz
(8) Show that none of the above things satisfies all three pumping conditions simultaneously.
(9) s cannot be pumped.

Example

Solution

Assume that $|A|$ is regular and has a pumping length = p.
Let a string $S = a^p b^p$.
Now divide the S into the parts, xyz.
To divide the S, let's take the value of p=7.
(assume)
∴ S = aaaaaaa bbbbbbb (by putting p=7 in $S = a^p b^p$).

Case 1:- y consists of a string having the letter only 'a'.

$\underbrace{aa}_{x}\ \underbrace{aaaa}_{y}\ \underbrace{abbbbbbb}_{z}$

Case 2:- y consists of a string having the letter only 'b'

$\underbrace{aaaaaaab}_{x}\ \underbrace{bbbb}_{y}\ \underbrace{b}_{z}$

Case 3:- y consists of a string with the both 'a' & 'b'.

$\underbrace{aaaaaa}_{x}\ \underbrace{aabb}_{y}\ \underbrace{bbbbb}_{z}$

* above Cases, we need to show $xy^iz \notin A$ for some i; Let the value of $i=2$.

$xy^iz \Rightarrow xy^2z$

In Case 1:

$xy^2z = aa\ aaaa\ aaaa\ abbbbbbb$

No. of 'a' = 11 , No. of 'b' = 7.

Since the No. of 'a' ! = No. of 'b'. but the original lang. has an equal number of 'a' & 'b'.

∴ this string will not lie in our language

In Case 2:

$xy^2z = aaaaaaaabb\ bbbb\ bbb\ b$

No. of 'a' = 7, No. of 'b' = 11.

Since the No. of 'a' ! = No. of 'b'. but the original language has an equal No. of 'a' & 'b'.

∴ this string will not be in our language.

In Case 3

$xy^2z = \underline{aaaaa}\ \underline{aabb}\ \underline{aabb}\ bbbb$

No. of 'a' = 8 . No. of 'b' = 9.

$a^5 a^2 b^2 a^2 b^2 b^5$

$= a^9 b^9$ but former is diff. $a^7 b^2 a^2 b^7$

Since the no. of 'a', but the original language has an equal number of
$!=$ no. of 'b'. 'a' and 'b', & also this string did not follow the $a^n b^n$ pattern.

∴ this string will not be in our language.

we can see at $i=2$ all the above three strings do not lie in the language.

$A=\{a^n b^n / n \geq 0\}$

∴ the language $A=\{a^n b^n / n \geq 0\}$ is not Regular.

- State the pumping lemma — with Theorem & proof

- Applications of pumping lemma —

① It is used for proving that certain languages are not regular.

② It is useful to determine whether the language accepted by finite automata is finite or infinite.

The general steps applied in pumping lemma are as follows,

1. Generate an input string(s) from the given language L.
2. Divide the length of input string(s)

as x, y, z respectively and apply the conditions that are imposed by p.l.

3. Verify whether they exist any contradiction to pumping Lemma or any value.

Context - free Grammar :—

A Context free grammar (CFG) describes the programming language constructs. It consists of non-terminals, terminals start symbols and productions. It is denoted by $G = (V_n, V_t, S, P)$

1. Non-terminals (V_n) :— Non-terminals are the finite set of variables that represents a language. i.e. a set of strings. these are generally represented by capital letters. A, B, C...

2. Terminals (V_t) :— Terminals are the finite set of symbols from which the strings for the language are formed. these are generally represented by small letters a, b, c. $\alpha, \beta, (,)$ etc.

3. Start Symbol (S) :-

Start Symbol is the initial non-terminal Symbol in the grammar. It represents the language being defined by the grammar. It is generally denoted by S.

4. production (P)

Productions are the set of rules that describes the recursive definition of a lang. All productions are of the form.

$$\alpha \longrightarrow \beta$$

where:

α : A non-terminal i.e., $\alpha \in V_n$

β : A Combination of terminals and non-terminals i.e.

$\beta \in (V_t \cup V_n)$.

* Consider the following simple grammar.

$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

} $\rightarrow P$

$V_n = \{S, A, B\}$
$V_t = \{a, b\}$
$S = S$.

Let's use this grammar to derive a string 'ab'.

* Start with the start Symbol S, using:

Apply the PR $S \rightarrow AB$ $S \rightarrow AB$

" " " $S \rightarrow aB$ $A \rightarrow a$

" " " $S \rightarrow ab$ $B \rightarrow b$.

This is how the PR are applied in a specific order to generate a desired string according to grammar rules.

Derivation :- Derivation is a sequence of production rules. It is used to get the i/p string through these production rules. During parsing, we have to take two decisions, these are as follows,

- we have to decide the non-terminal which is to be replaced.
- we have to decide the production rule by which the non-terminal will be replaced.

Here we have two options to decide which nonterminal to be placed with production rule,

(1) **Leftmost Derivation :** In the leftmost derivation, the input is scanned and replaced with the production rule from left to right. So in leftmost derivation, we read the input string from left to right.

Example : production rules :

1. $E \Rightarrow E+E$
2. $E \Rightarrow E-E$
3. $E \Rightarrow a \mid b$

Input / to obtain string is.

1. a − b + a.

The L-M-D is

(1) $E \Rightarrow E+E$

(2) $E \Rightarrow E-E+E$ (Using $E \to E$)

(3) $E = a-E+E$ (Using $E \to a$)

(4) $E = a-b+E$ (, $E \to b$)

So $E = a-b+a$.

(2) Rightmost Derivation: In rightmost derivation, the input is scanned and replaced with the production rule from right to left. So in rightmost derivation, we read the input string from right to left.

Example: production rules:

1. $E = E+E$
2. $E = E-E$
3. $E = a|b$

input

1. $a-b+a$

The R-M-D is

(1) $E = E-E$

$E = E-E+E$

$E = E-E+a$

$E = E-b+a$

$E = a-b+a$.

When we use the L-M-D or R-M-D, we may get the same string.

Derive the string "abb" for leftmost derivation and rightmost derivation using a CFG given by.

1. $S \rightarrow AB \mid \varepsilon$
2. $A \rightarrow aB$
3. $B \rightarrow Sb$

Sol:

LMD

$S \rightarrow AB$
$\rightarrow aBB$ (Using $A \rightarrow aB$)
$\rightarrow aSbB$ (Using $B \rightarrow Sb$)
$\rightarrow a\varepsilon bB$ (Using $S \rightarrow \varepsilon$)
$\rightarrow abB$ (Using $B \rightarrow Sb$)
$\rightarrow abSb$ (Using $S \rightarrow \varepsilon$)
$\rightarrow ab\varepsilon b$ ($\varepsilon \rightarrow 1$)
$\rightarrow abb$

RMD

$S \rightarrow AB$
$\rightarrow ASb$
$\rightarrow A\varepsilon b$
$\rightarrow aB\varepsilon b$
$\rightarrow asbb$
$\rightarrow a\varepsilon bb$
$\rightarrow abb$.

Ex: Derive the string "aabbabba" for LMD & RMD using a CFG.

1. $S \rightarrow aB \mid bA$
2. $A \rightarrow a \mid as \mid bAA$
3. $B \rightarrow b \mid bs \mid aBB$.

Sol: LMD

1. S
2. aB $S \to aB$
3. aaBB $B \to aBB$
4. aabB $B \to b$
5. aabbS $B \to bS$
6. aabbaB $S \to aB$
7. aabbabS $B \to bS$
8. aabbabbA $S \to bA$
9. aabbabba $A \to a$

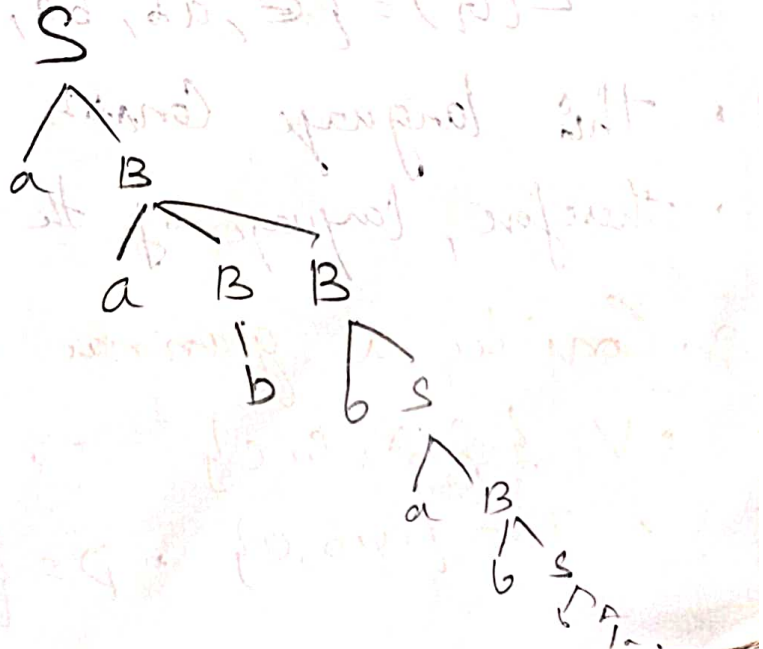
RMD

1. S
2. aB $S \to aB$
3. aaBB $B \to aBB$
4. aaBbS $B \to bS$
5. aaBbbA $S \to bA$
6. aaBbba $A \to a$
7. aabSbba $B \to bS$
8. aabbAbba $S \to bA$
9. aabbabba $A \to a$

Q) Derive the string "00101" for LMD & RMD using a CFG given by. (Assignment)

(1) $S \to A1B$
(2) $A \to 0A \mid \varepsilon$
(3) $B \to 0B \mid 1B \mid \varepsilon$

Q) (Derivation Tree)



Language of a Grammar

Language of grammar is the set of all strings that can be generated from that grammar.

- If the language consists of finite number of strings, then it is called as a F.L.
- If the language consists of infinite number of strings, then it is called as an I.L.

Ex: 1 Consider a grammar $G = (V, T, P, S)$ where

- $V = \{S\}$ · $T = \{a, b\}$ · $S = \{S\}$
- $P = \{S \to aSbS,\ S \to bSaS,\ S \to \in\}$

This grammar generates the strings having equal no. of a's and b's. So Language of the grammar is:

$$L(G) = \{\in, ab, ba, aabb, bbaa, abab, baba, \ldots\}$$

- This language consists of infinite no. of strings.
- Therefore, Lanuage of the grammar is infinite

2. Consider a grammar $G = (V, T, P, S)$ where,

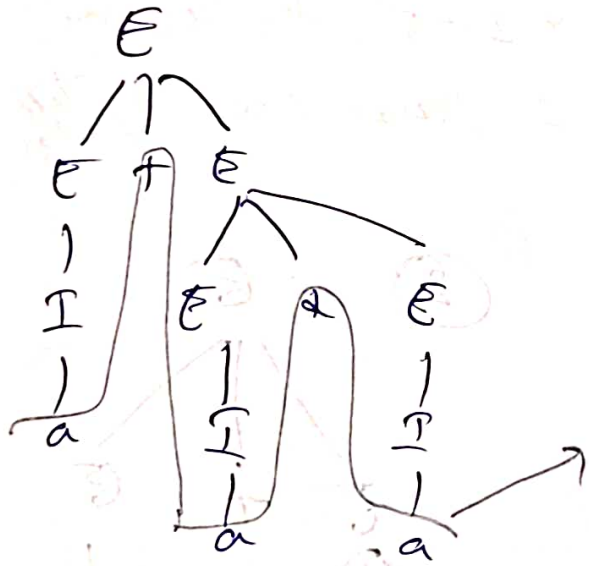
- $V = \{S, A, B, C\}$ · $S = \{S\}$
- $T = \{a, b, c\}$ · $P = \{S \to ABC,$
  $A \to a$
  $B \to b,\ C \to c\}$

this grammar generates only one string 'abc'
So. Language of this grammar is — $L(G) = \{abc\}$
. this language consists of finite no. of strings.
. therefore, language of the grammar is finite.

Parse Tree — Tree representation of a derivation called "parse tree".

Ex $E \to E+E$
$E \to E*E \mid I$
$I \to a$

It will traverse the tree it will get the string.



yield of the tree by getting a string.
a+a*a.

Ambiguity of a Grammar —

A grammar is said to be ambiguous. If there exists more than one leftmost derivation or more than one rightmost derivation, or more than one parse tree for the given input string.

If the grammar is not ambiguous, then it is called unambiguous.

Eg: Let us Consider a grammar G with 5 production rule.

(1) $E \rightarrow I$

(2) $E \rightarrow E+E$ ("3*2+5")

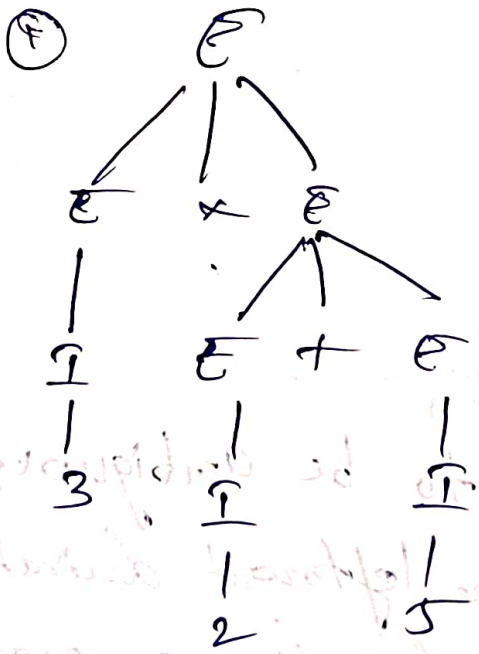
(3) $E \rightarrow E*E$

(4) $E \rightarrow (E)$

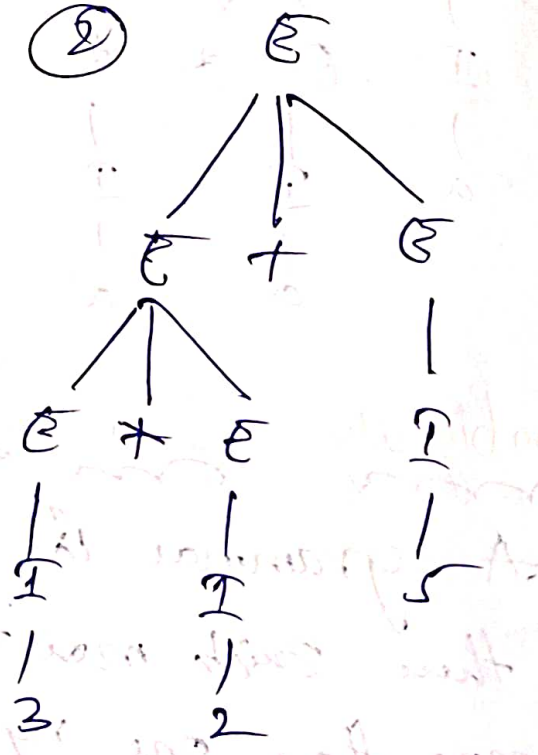
(5) $I \rightarrow \varepsilon | 0 | 1 | 2 | \ldots | 9$

Sol: For the string "3*2+5", the above grammar can generate 2 parse trees by leftmost derivation.

(1)



(2)



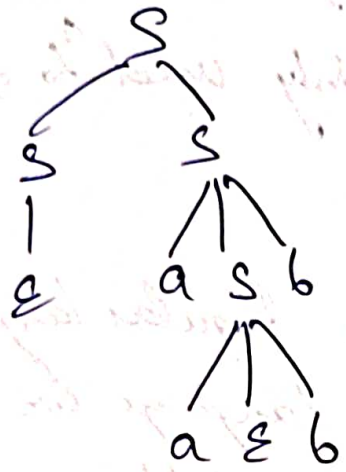
Here, the above grammar G is ambiguous for above 2 parse trees for a single string.

② $S \rightarrow aSb \mid SS$

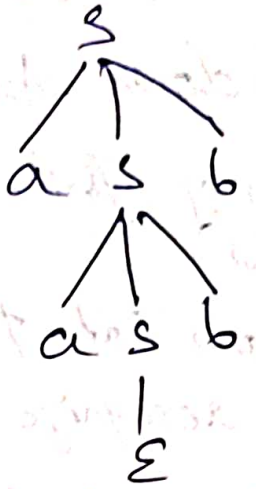
$S \rightarrow \varepsilon$

"aabb"

①



②



Ambiguous

③ $A \rightarrow AA$

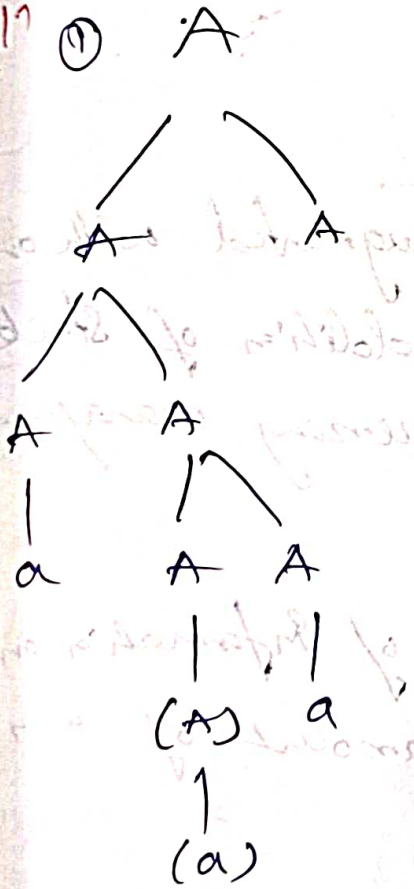
$A \rightarrow (A)$

$A \rightarrow a$

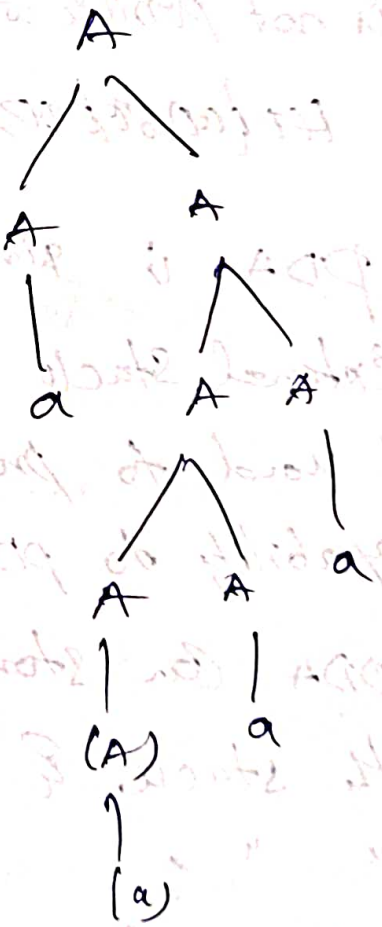
"a(a)aa"

Sol"

①



②



27/9/24

Unit - 3

Push Down Automata - used to recognize context free lang

→ PDA is simply an NFA argumented with an external stack memory.

→ The addition of stack is used to provide a LIFO memory management capabality to PDA.

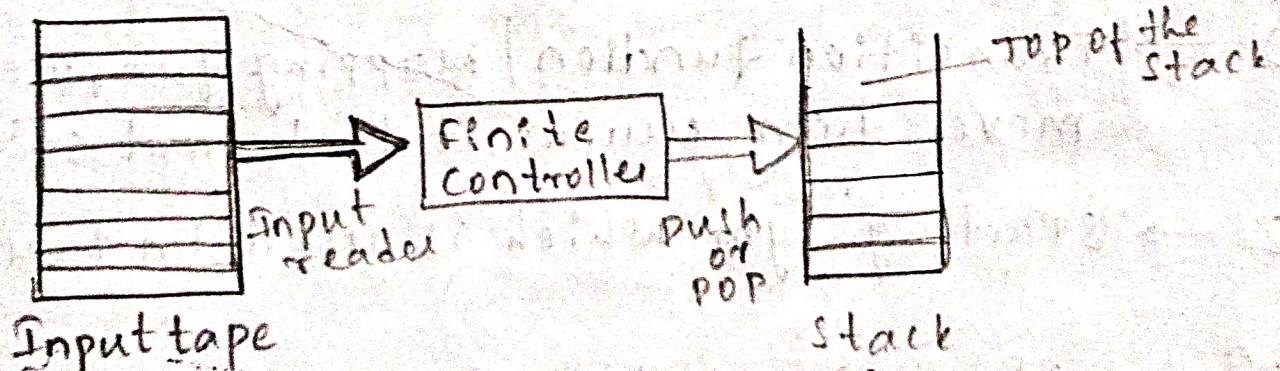
→ PDA can store an unbounded amount of info in the stack & it can access amount of info in the stack

⇒ A PDA can push the element on the top of the stack & pop of an element from the top of the stack

⇒ To read an element into the stack the top elements must be poped off.

⇒ PDA is more poweful than FA

⇒ Any lang which can be acceptable by FA can also be acceptable by PDA.



PDA components:

Input tape - divided in many cells or symbols.

→ The I/p head is read only & move from left to right, one symbol at a time.

Finite control - It has some pointer which points a current symbol which is to be read.

Stack - It is a data structure, in which we can push & remove the items from (end only). It has an infinite size.

In PDA, it is used to store the items temporarily.

Formal Defⁿ of PDA:

The PDA can defined as a collection of 7 tuples.

$$\{Q, \Sigma, \sqrt{\ }, \delta, q_0, Z, f, S\}$$

$\sqrt{\ }$ ↓ tau

$Q \rightarrow$ Finite set of states

$\Sigma \rightarrow$ Set of input symbols

$q_0 \rightarrow$ Initial state

$f \rightarrow$ final state

$\delta \rightarrow$ Transition function / Mapping fun, used to move from current state to next state

$Z \rightarrow$ starting sym which is used in tau

$\sqrt{\ } \rightarrow$ stack sym which can be pushed & poped from stack.

NDPA	DPDA
→ Non deterministic pushdown automata	→ Deterministic pushdown automata
→ More powerful	→ Less powerful
→ It is not possible to convert every NDPA to a corresponding DPDA	→ can possible → Is a subset
→ The lang accepted by NPDA is not a subset of lang accepted by DPDA	
→ Lang accepted by NPDA is called N CFL	→ is called DCFL (Deterministic context free Language) which is subset of NCFL

Example

1) Construct PDA for the Lang $L = \{a^n b^n / n \geq 1\}$ by an empty stack

$M = \{Q, \Sigma, \Gamma, q_0, Z, f, \delta\}$

Procedure:

Step ①: Initially push all a's on to the stack

Step ②: whenever 'b' occurs change the state & pop 'a' from the stack. (Here we have to change the state only for 1st pop operation.

for remaining pop opⁿ we no need to change the state. At last stop the stack)

Step 3: Repeat step 2 until stack is empty (i.e entire input string is processed.

$$L = \{ ab, aabb, aaabbb, \ldots \}$$

Ip symbol, Top of the stack

a	a	a	b	b	b	∈
①	↑	②	③	④	⑤	⑥

$\delta(q_0, a, z_0) = (q_0, az_0)$ — a, Z0 — q_0

$\delta(q_0, a, a) = (q_0, aaz_0)$ — a, a, Z0

$\delta(q_0, a, a) = (q_0, aaaz_0)$ — a, a, a, Z0

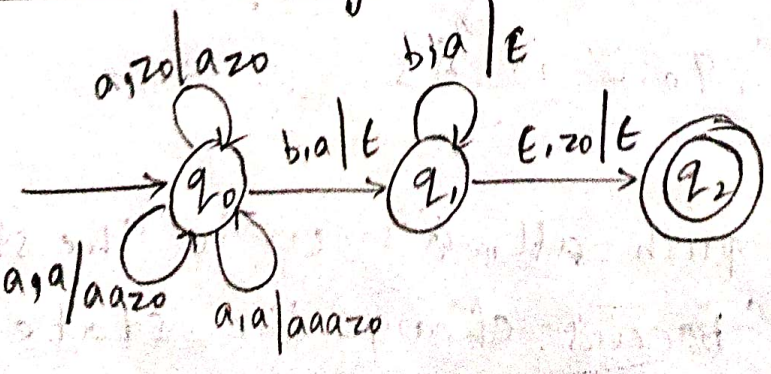
$\delta(q_0, b, a) = \delta(q_1, \epsilon)$ — ~~a~~, a, a, Z0

$\delta(q_1, b, a) = \delta(q_1, \epsilon)$ — ~~a~~, a, Z0

$\delta(q_1, b, a) = \delta(q_1, \epsilon)$ — ~~a~~, Z0

$\delta(q_1, \epsilon, z_0) = \delta(q_2, \epsilon)$ — ~~Z0~~ ⟹ ∪

Transition Diagram



$\delta(q_0, aabb, z_0) = \vdash(q_0, abb, az_0)$
$\vdash(q_0, bb, aaz_0)$
$\vdash(q_1, b, az_0)$
$\vdash(q_1, \epsilon, z_0)$
$\vdash(q_2, \epsilon)$

2) Construct a PDA for the $L = \{a^n b^{2n} / n \geq 1\}$ by an empty stack.

$M = \{Q, \Sigma, \sqrt{\ }, q_0, z, f, \delta\}$

$L = \{abb, aabbbb, ----\}$

Logic

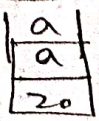
1) a → 2 a's
2) b → pop
3) Repeat (2)

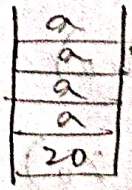
(or)

(1) a → a
(2) b → 'a'
b → skip
b → 'a'
b → skip

* | a | a | b | b | b | b | ε |

$\delta(q_0, a, z_0) = (q_0, aaz_0)$



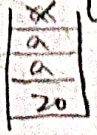


$\delta(q_0, a, a) = (q_0, aa \cdot aaz_0)$

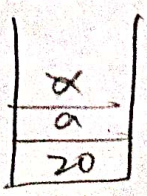
$\delta(q_0, b, a) = (q_1, \epsilon)$



$\delta(q_1, b, a) = (q_1, \epsilon)$



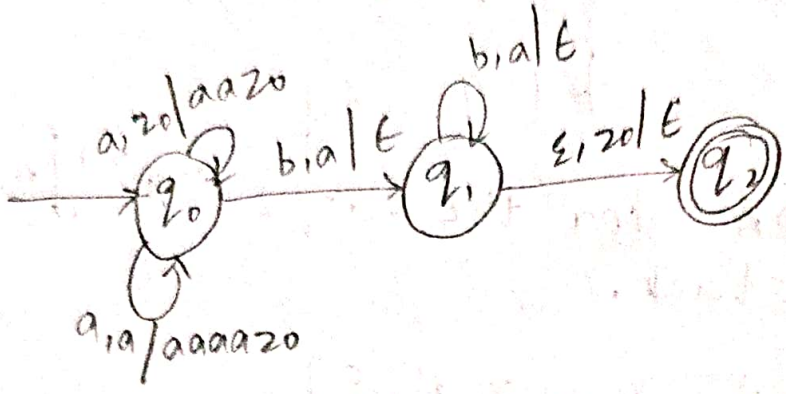
$\delta(q_1, b, a) = (q_1, \epsilon)$



$\delta(q_1, b, a) = \delta(q_1, \epsilon)$ | ~~a~~ / z_0 |

$\delta(q_1, \epsilon, z_0) = (q_2, \epsilon)$ | z_0 |

Transition Diagram



* | a | a | b | b | b | b | ϵ |

$\delta(q_0, a, z_0) = (q_0, a z_0)$ | a / z_0 |

$\delta(q_0, a, a) = (q_0, a a z_0)$ | a / a / z_0 |

$\delta(q_0, b, a) = (q_1, \epsilon z_0)$ | ~~a~~ / a / z_0 |

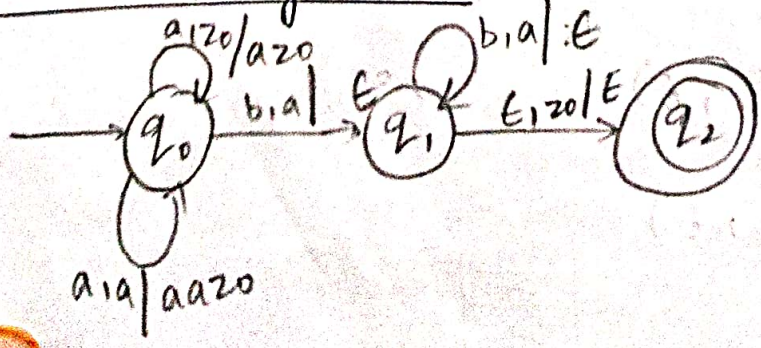
$\delta(q_1, b, a) = skip$

$\delta(q_1, b, a) = (q_1, \epsilon z_0)$ | ~~a~~ / z_0 |

$\delta(q_1, b, z_0) = skip$

$\delta(q_1, \epsilon, z_0) = (q_2, \epsilon)$ | ~~z_0~~ |

Transition Diagram



(5) $L = \{ wcw^R \mid w \in (a+b)^* \}$ odd palindrome

$L = \{ abacaba, bababcabab, \ldots \}$

* $\delta(q_0, a, z_0) = (q_0, az_0)$

  $\delta(q_0, b, z_0) = (q_0, bz_0)$

a,b
z0

* $\delta(q_0, a, a) = (q_0, aaz_0)$

  $\delta(q_0, a, b) = (q_0, abz_0)$

  $\delta(q_0, b, a) = (q_0, baz_0)$

  $\delta(q_0, b, b) = (q_0, bbz_0)$

a
a
z0

b
a
z0

a
b
z0

b
b
z0

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

  $\delta(q_0, c, b) = (q_1, b)$

a,b
a,b
z0

* $\delta(q_1, a, a) = (q_1, \epsilon)$

  $\delta(q_1, b, b) = (q_1, \epsilon)$

X
a
z0

X
a
z0

X
b
z0

X
b
z0

* $\delta(q_1, \epsilon, z_0) = (q_2, \epsilon)$

~~$\delta(q_1, \epsilon, z_0)$~~

X
z0

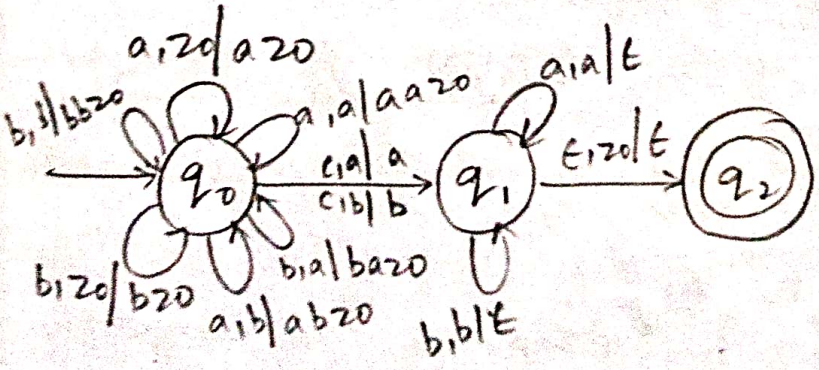
X
z0

X
z0

X
z0

~~z0~~

Transition diagram



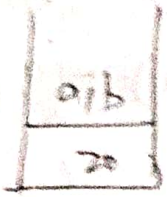
(b) $L = \{w \in \Sigma^* \mid w \in (a+b)^*\}$ even palindrome

$L = \{aba, aba, baba, baba - \ldots\}$

Transition fun:

$\delta(q_0, a, z_0) = \delta(q_0, a z_0)$

$\delta(q_0, b, z_0) = \delta(q_0, b z_0)$

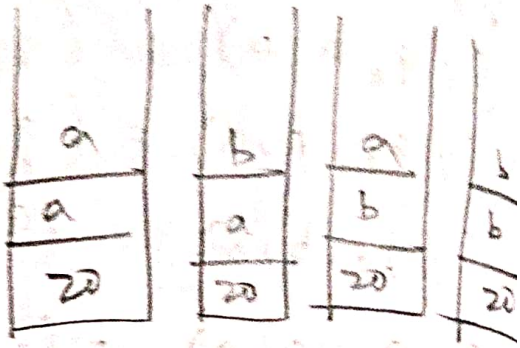


$\delta(q_0, a, a) = \delta(q_0, a a)$

$\delta(q_0, a, b) = \delta(q_0, a b z_0)$

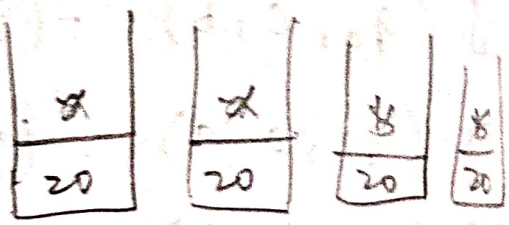
$\delta(q_0, b, a) = \delta(q_0, b a z_0)$

$\delta(q_0, b, b) = \delta(q_0, b b z_0)$



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

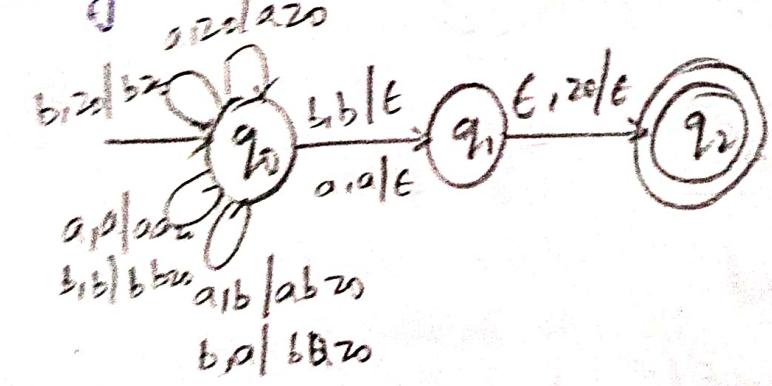
$\delta(q_0, b, b) = \delta(q_1, \epsilon)$



$\delta(q_0, \epsilon, z_0) = \delta(q_2, \epsilon)$



Diagram:



(5) $L = \{a^2 b^n \mid n \geq 1\}$

$L = \{aab, aaaabb - - - \}$

a	a	a	a	b	b	ϵ

$\delta(q_0, a, z_0) = \{q_0, a z_0)$ — push a

$\delta(q_0, a, a) = (q_0, a z_0)$ — skip

$\delta(q_0, a, a) = (q_0, a a z_0)$ — push a

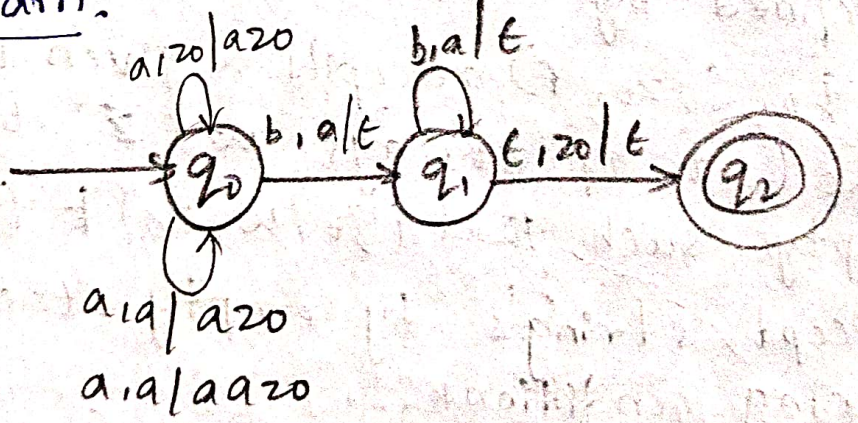
$\delta(q_0, a, a) = (q_0, a a z_0)$ — skip

$\delta(q_0, b, a) = (q_1, \epsilon)$ — pop a

$\delta(q_1, b, a) = (q_1, \epsilon)$ — pop a

$\delta(q_1, \epsilon, z_0) = (q_2, \epsilon)$ — pop z_0

Diagram:



Language of PDA:

Acceptance of language by PDA

⇒ A PDA accepts a language in 2 ways:

- Acceptance by final state
- Acceptance by empty stack.

* Acceptance by Final State:

A PDA is defined by
$P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ accepts a given lan
by generating the string $w \in \Sigma^*$ by entering
the final state such that
$(q_0, w, z_0) \vdash (q_0, \varepsilon, \alpha)$ where $q_f \in F$ and $\alpha \in$
i.e A PDA can accept strings through final
state based on the conditions.

a) If string is completely traversed (i.e, it comes to an end.)

b) If the PDA has visited its final state
If a strings of PDA are accepted by final state then the final state, should be of type
$\delta(q_i, \varepsilon, z_0) = (q_f, z_0)$.

* Acceptance by Empty stack:

A PDA is defined by
$P = (Q, \Sigma, \Gamma, q_0, \delta, z_0, F)$ accepts a given Lang b
generating the set of strings $w \in \Sigma^*$ by maki
its stack empty such that $(q_0, w, z_0) \vdash (q_i, \varepsilon, \varepsilon$
A PDA can accept strings by empty stack base
on the following conditions:

a) If string is completely traversed (i.e it comes to an end)

b) If the PDA has its stack empty
thus, if a strings of PDA are accepted by empt
stack then the final state should be of typ
$\delta(q_i, \varepsilon, z_0) \vdash (q_f, \varepsilon)$

Conversion of CFG to PDA :

Ex: $S \rightarrow AB$
$A \rightarrow 0S \mid 0$
$B \rightarrow 1S \mid 1$

Rules:

LHS RHS

For Non-terminal $A \rightarrow \alpha$

$\delta(q, \epsilon, A) = (q, \alpha)$

For Terminal a (or) $+$ (or) 1

$\delta(q, a, a) = (q, \epsilon)$

For Non Terminals:

$\Rightarrow S \rightarrow AB$

$\delta(q, \epsilon, S) = (q, AB)$

$\Rightarrow A \rightarrow 0S \mid 0$

$\delta(q, \epsilon, A) = (q, 0S)$

$\delta(q, \epsilon, A) = (q, 0)$

$\Rightarrow B \rightarrow 1S \mid 1$

$\delta(q, \epsilon, B) = (q, 1S)$

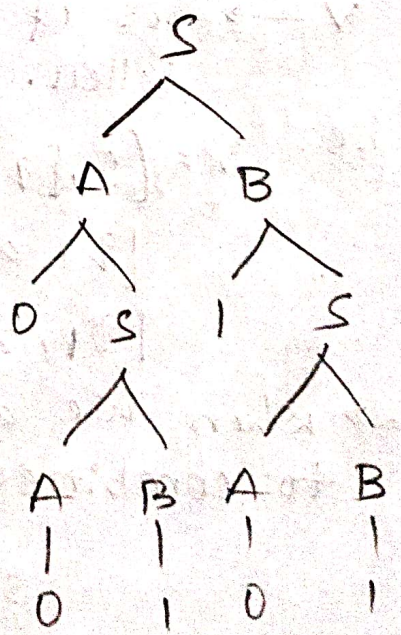
$\delta(q, \epsilon, B) = (q, 1)$

For Terminals:

$\Rightarrow \delta(q, 0, 0) = (q, \epsilon)$

$\Rightarrow \delta(q, 1, 1) = (q, \epsilon)$

Parse Tree



Conversion of PDA to CFG:

$M = (Q, \Sigma, \sqrt{}, q_0, z, f, \delta)$

Transition fun^n:

$\delta(q_0, a, z_0) = (q_0, a z_0)$

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

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

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

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

$\delta(q_1, \epsilon, z_0) = (q_1, \epsilon)$

$CFG = \{V, T, P, S\}$

$M = (Q, \Sigma, q_0, f, z, \sqrt{}, \delta)$

$= (\{q_0, q_1\}, \{a, b\}, q_0, f, z_0, \{a, z_0\})$

$V \longrightarrow$ set of variables / Non-teiminals / capital letters.

i.e., $V = (S, [q_0, a, q_0], [q_0, a, q_1], [q_1, a, q_0], [q_1, a, q_1], [q_0, z_0, q_0], [q_0, z_0, q_1], [q_1, z_0, q_0], [q_1, z_0, q_1])$

→ when we are declaring variables we nee to combine 'Q' with '$\sqrt{}$'.

$\{q_0, q_1\}$ $\{a, z_0\}$

T → Terminals / small letters / operators

i.e T = {a, b}

S → starting symbol

S → [q0 z0 q0]

S → [q0 z0 q1]

↓ Initial Stack ↓ Initial state of stack symbol ↓ Q's

P → Production

⇒ Total 'δ' transition fun. are given, for all we need to write productions.

$$\delta(q_0, a, z_0) = (q_0, a z_0)$$

The stack contains 2 stack symbols we need to write 2 × 2 = 4 productions.

① $\delta[q_0\ z_0\ q_0] = a\,[q_0\ a\ q_0]\,[q_0\ z_0\ q_0]$

② $\delta[q_0\ z_0\ q_0] = a\,[q_0\ a\ q_1]\,[q_1\ z_0\ q_0]$

③ $\delta[q_0\ z_0\ q_1] = a\,[q_0\ a\ q_0]\,[q_0\ z_0\ q_1]$

④ $\delta[q_0\ z_0\ q_1] = a\,[q_0\ a\ q_1]\,[q_1\ z_0\ q_1]$

$\delta(q_0, a, a) = (q_0, aa)$ / $(2 \times 2 = 4)$

① $\delta[q_0\ a\ q_0] = a[q_0\ a\ q_0][q_0\ a\ q_0]$

② $\delta[q_0\ a\ q_0] = a[q_0\ a\ q_1][q_1\ a\ q_0]$

③ $\delta[q_0\ a\ q_1] = a[q_0\ a\ q_0][q_0\ a\ q_1]$

④ $\delta[q_0\ a\ q_1] = a[q_0\ a\ q_1][q_1\ a\ q_1]$

$\delta(q_0, b, a) = (q_1, a)$ / $(1 \times 2 = 2)$

① $\delta[q_0\ a\ q_0] = b[q_1\ a\ q_0]$

② $\delta[q_0\ a\ q_1] = b[q_1\ a\ q_1]$

$\delta[q_1, b, a) = (q_1, a)$ / $(1 \times 2 = 2)$

① $\delta[q_1\ a\ q_0] = b[q_1\ a\ q_0]$

② $\delta[q_1\ a\ q_1] = b[q_1\ a\ q_1]$

$\delta(q_1, a, a) = (q_1, \epsilon)$ / $(1 \times 1 = 1)$ for ϵ

① $\delta[q_1\ a\ q_1] = a$

$\delta(q_1, \epsilon, z_0) = (q_1, \epsilon)$ / $(1 \times 1 = 1)$

① $\delta[q_1, z_0, q_1] = \epsilon$

② $\{L = a^n b^m c^{(n+m)} / n, m \geq 1\}$ by a final state

Turing Machine :

⇒ To overcome the limitations of "PDA and "FA" we use turing machine.

⇒ PDA can be used to remember a current state of I/p. where as the turing machine has an external memory which remembers Arbitary long sequence of inputs.

⇒ It has unlimited memory capability.

⇒ This model has a facility by which the i/p at left (or) right on the tape can be read easily.

⇒ This machine can produce a certain o/p based on its i/p.

Formal description of Turing Machines

$M = \{Q, \Sigma, q_0, \sqrt{}, \delta, f, B\}$

where,

$Q \rightarrow$ finite nonempty set of i/p states

$\Sigma \rightarrow$ Finite non empty set of i/p symbols.

$q_0 \rightarrow$ Initial state

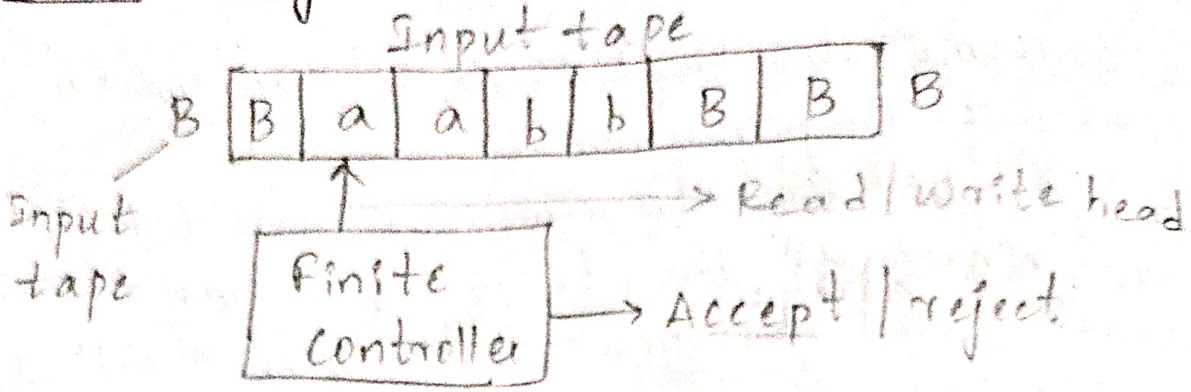
$\sqrt{} \rightarrow$ tape symbol

$f \rightarrow$ final state

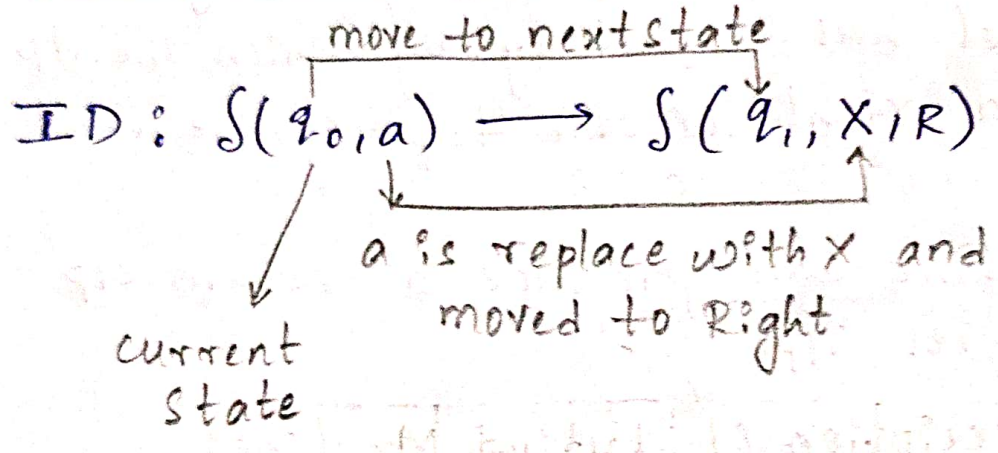
$B \rightarrow$ Blank symbol, used as an end marker for i/p

$\delta \rightarrow$ Transition function / Mapping fun.

Block Diagram:



Instantaneous description of Turing Machine:



Operations:

⇒ Read a symbol pointed by Read head

⇒ write a sym pointed by the tape head

⇒ Make a tape head either left or Right.

NOTE:

⇒ FA recognises (used for) for regular lang.

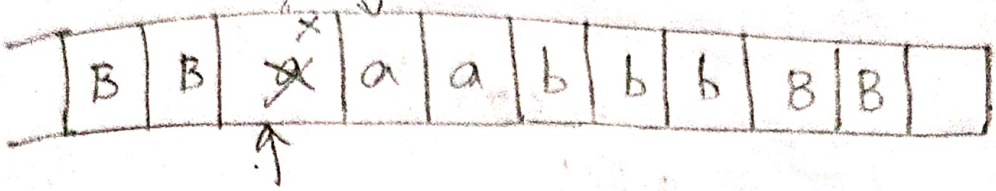
⇒ PDA is used for context free Languages.

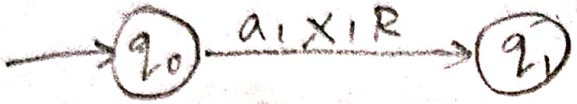
⇒ Turing Machine is used for Recursively enumerable languages.

$L = \{ a^n b^n \mid \text{where } n \geq 1 \}$

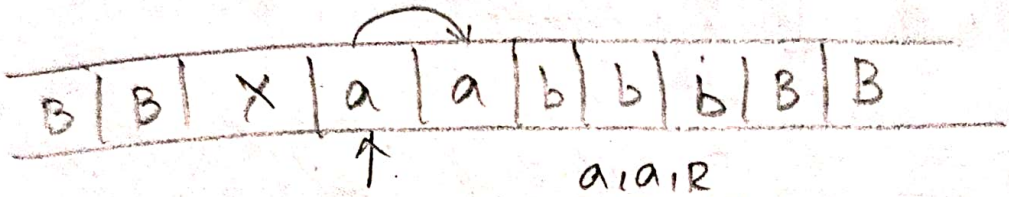
string = ab, aabb, aaabbb - - - -

1)

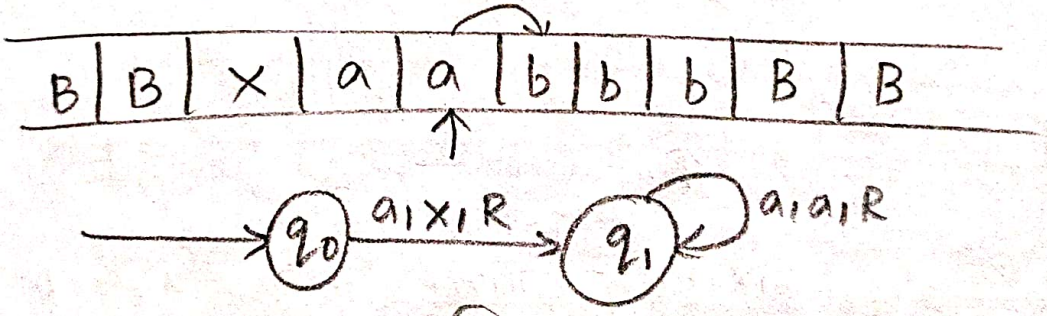




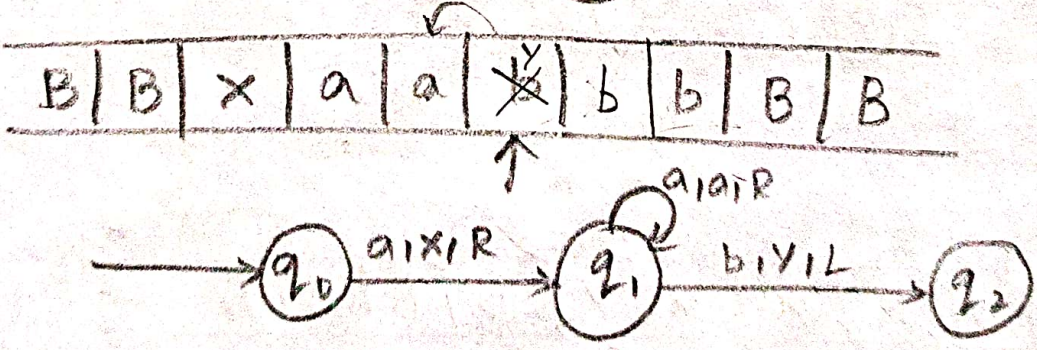
2)



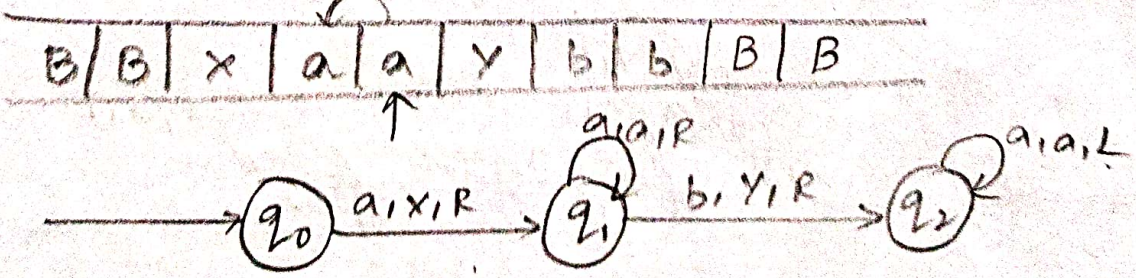
3)



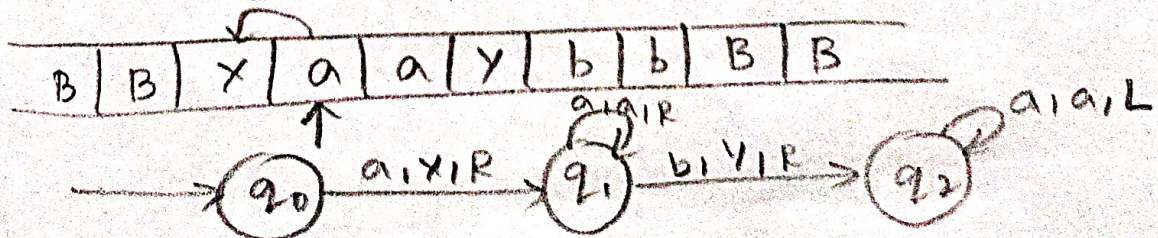
4)



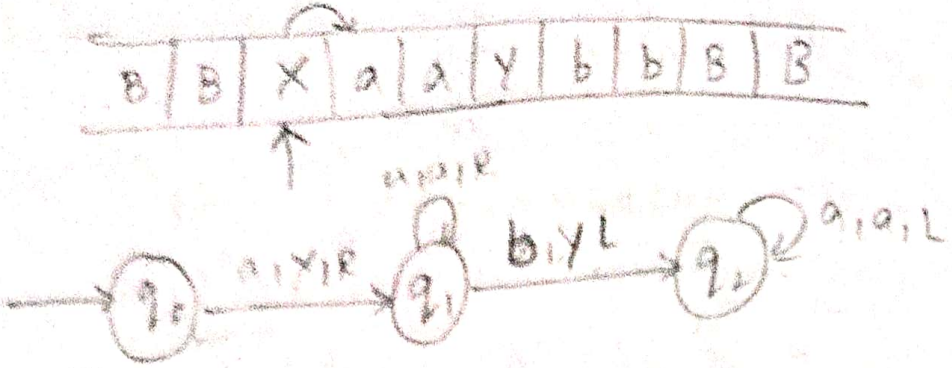
5)



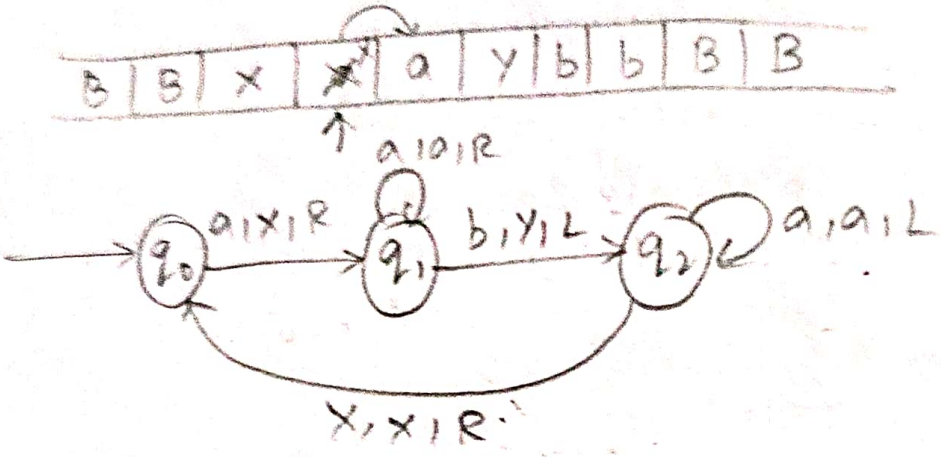
6)



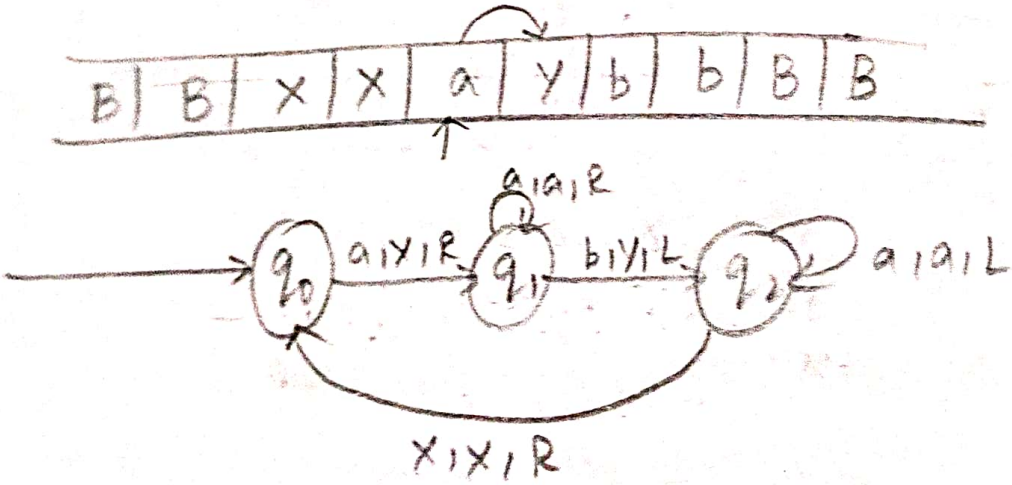
(7)


B B X a a Y b b B B
a,a,R
a,X,R
b,Y,L
a,a,L
q0
q1
q2

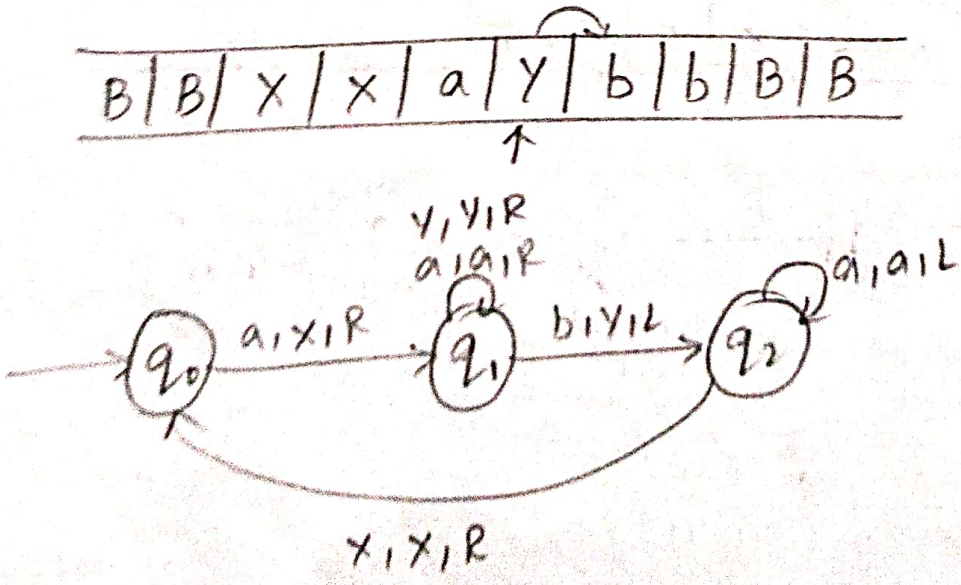
(8)

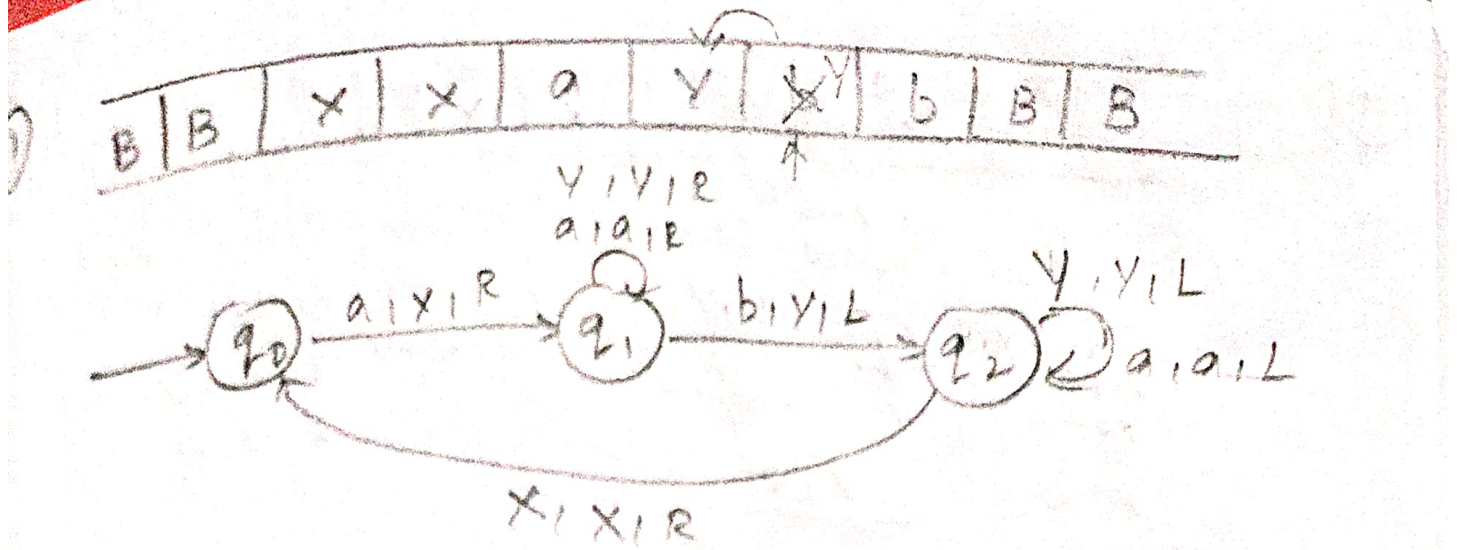

B B X X a Y b b B B
a,a,R
a,X,R
b,Y,L
a,a,L
X,X,R
q0
q1
q2

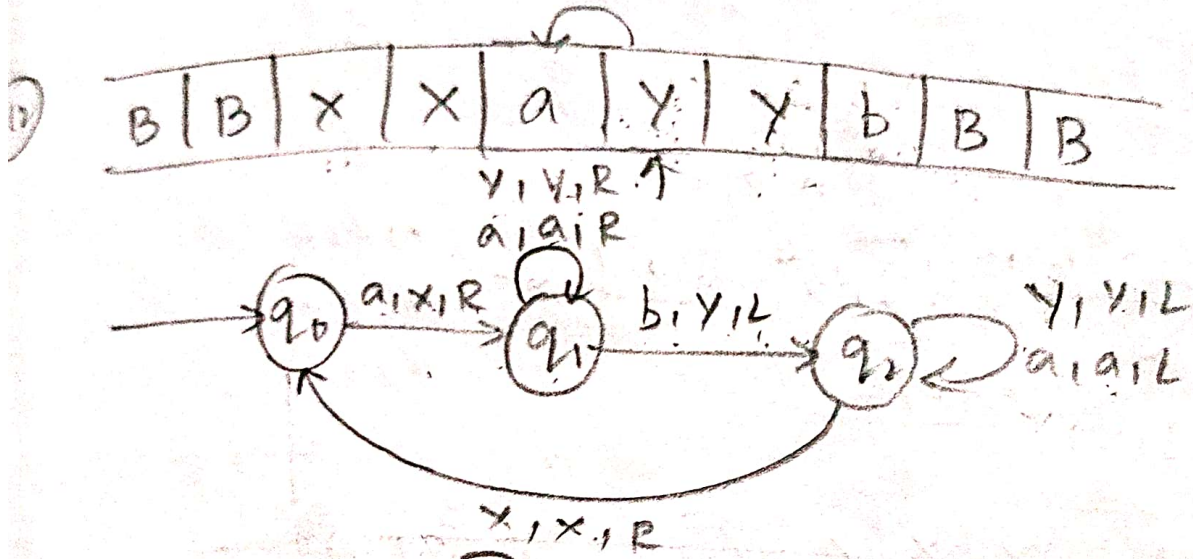
(9)

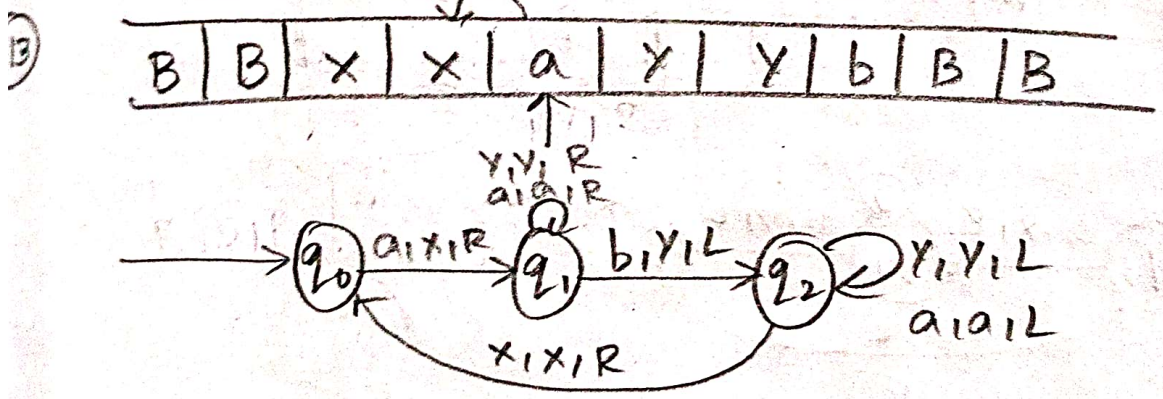

B B X X a Y b b B B
a,a,R
a,X,R
b,Y,L
a,a,L
X,X,R
q0
q1
q2

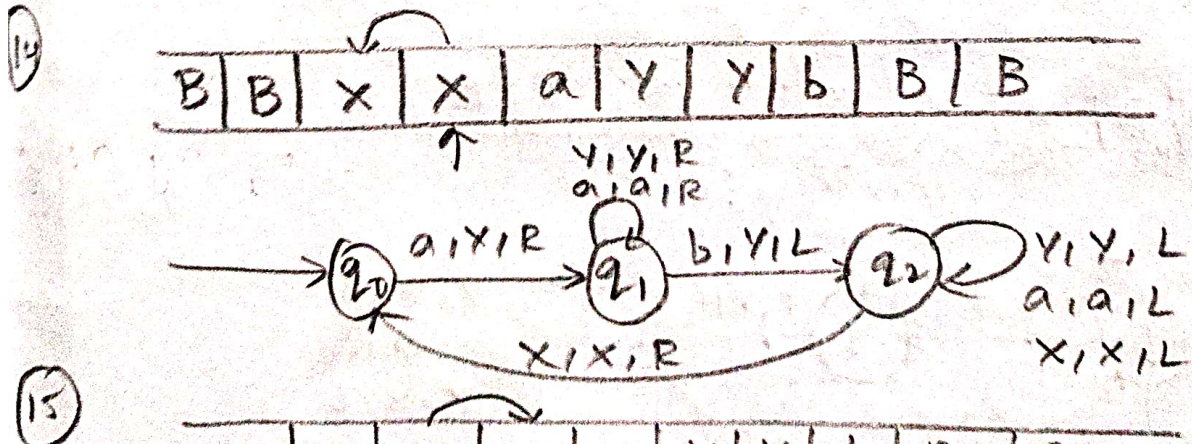
(10)

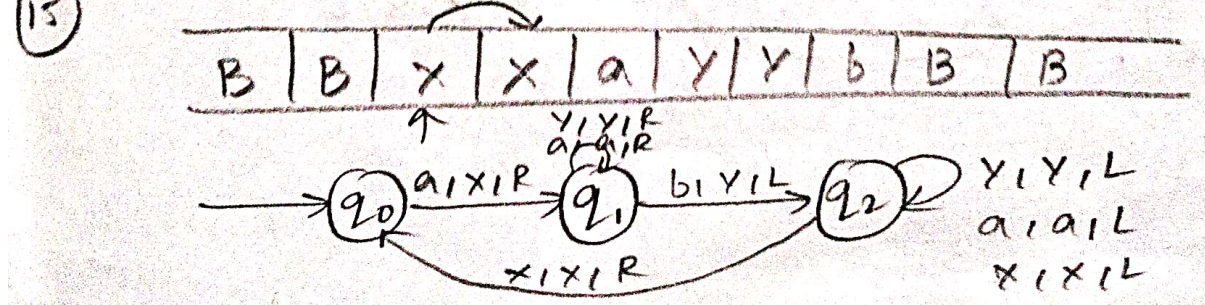

B B X X a Y b b B B
Y,Y,R
a,a,R
a,X,R
b,Y,L
a,a,L
X,X,R
q0
q1
q2


B B X X a Y Y b B B
Y,Y,R
a,a,R
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,R
q0
q1
q2

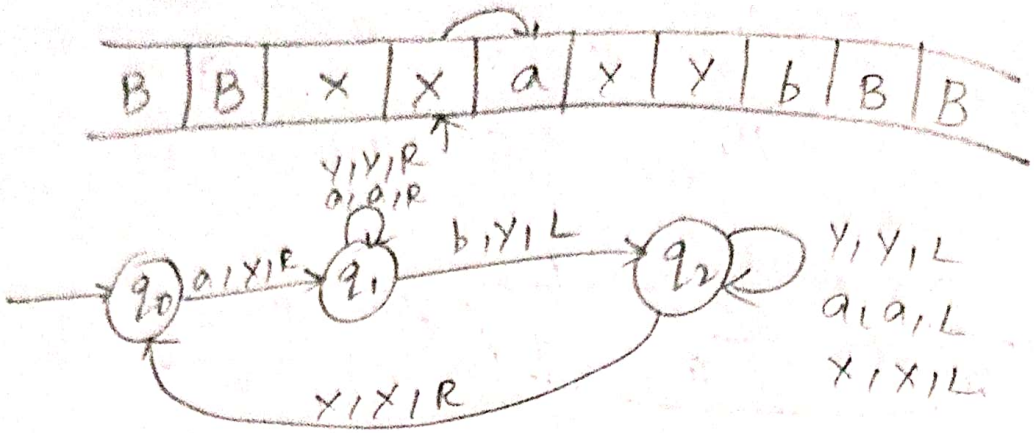

B B X X a Y Y b B B
Y,Y,R
a,a,R
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,R
q0
q1
q2


B B X X a Y Y b B B
Y,Y,R
a,a,R
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,R
q0
q1
q2

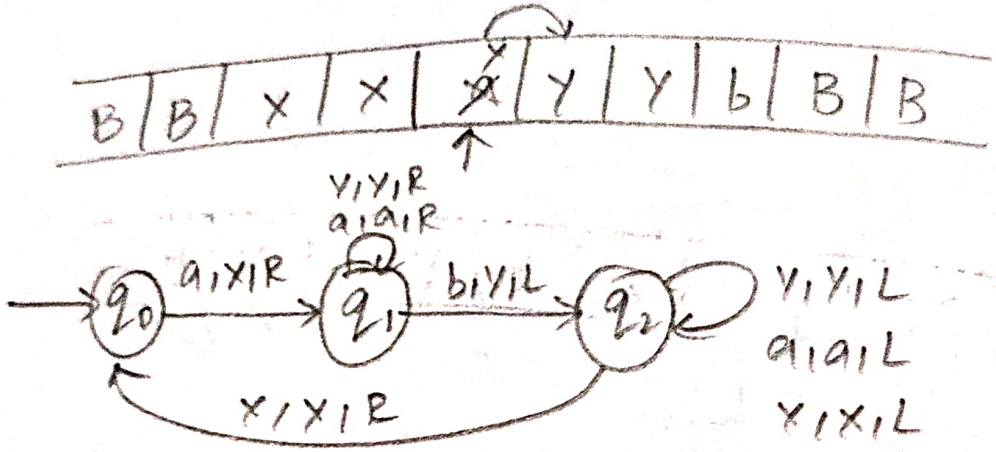

B B X X a Y Y b B B
Y,Y,R
a,a,R
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,R
q0
q1
q2


B B X X a Y Y b B B
Y,Y,R
a,a,R
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,L
X,X,R
q0
q1
q2

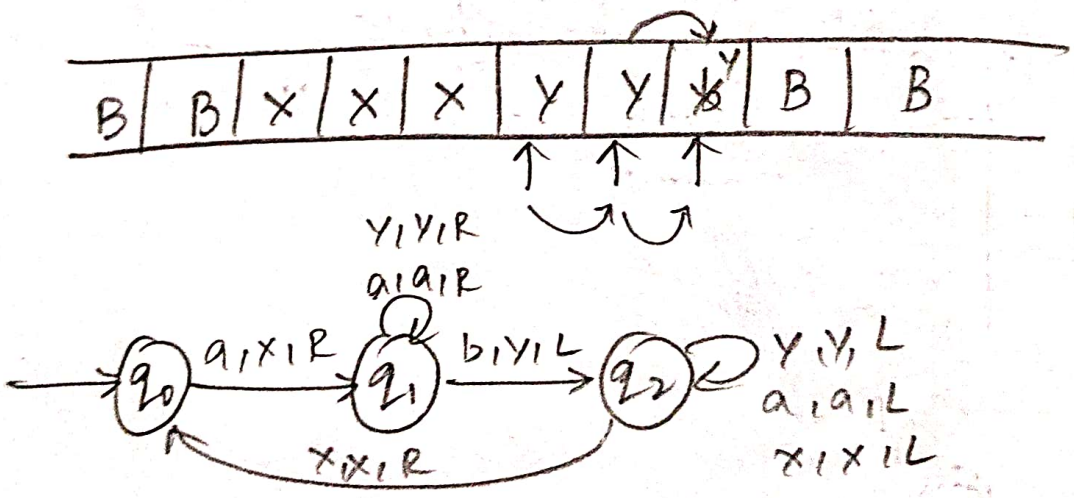
(16)


B B x x a y y b B B
Y,Y,R
a,a,R
q0
q1
q2
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,L
X,X,R

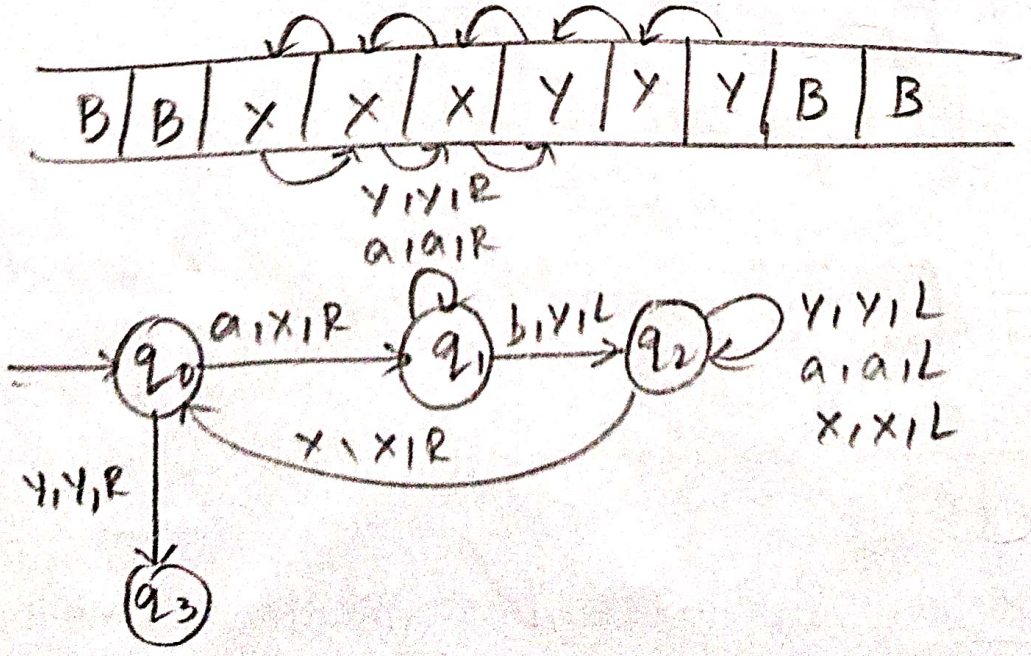
(17)

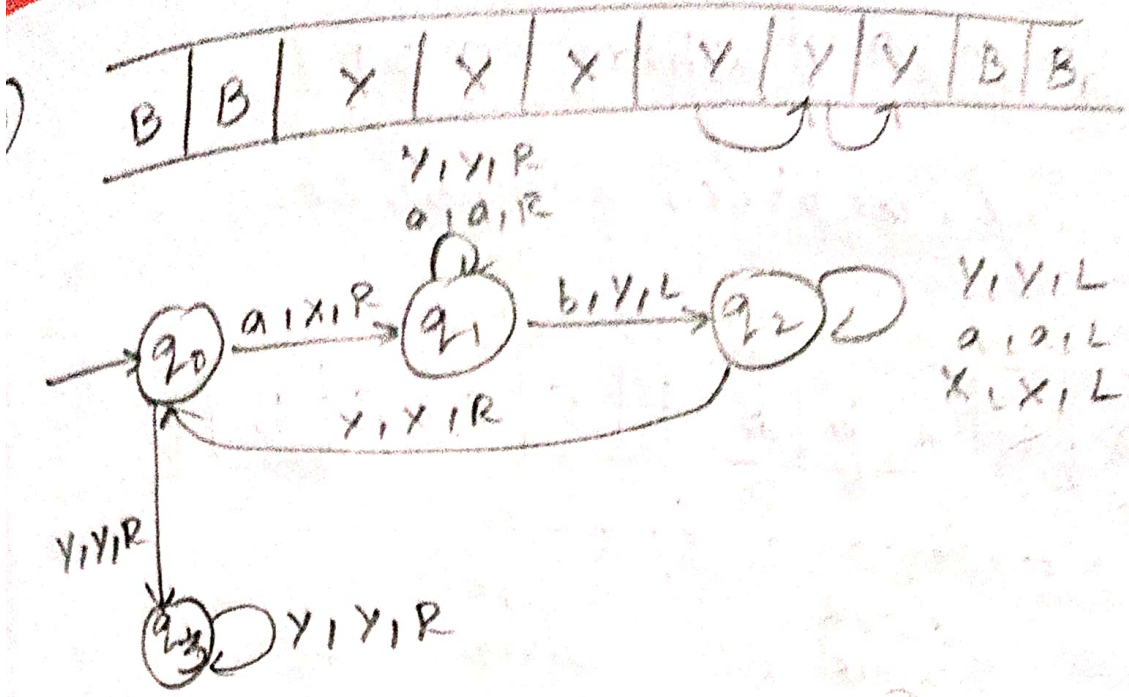

B B x x x y y b B B
Y,Y,R
a,a,R
q0
q1
q2
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,L
X,X,R

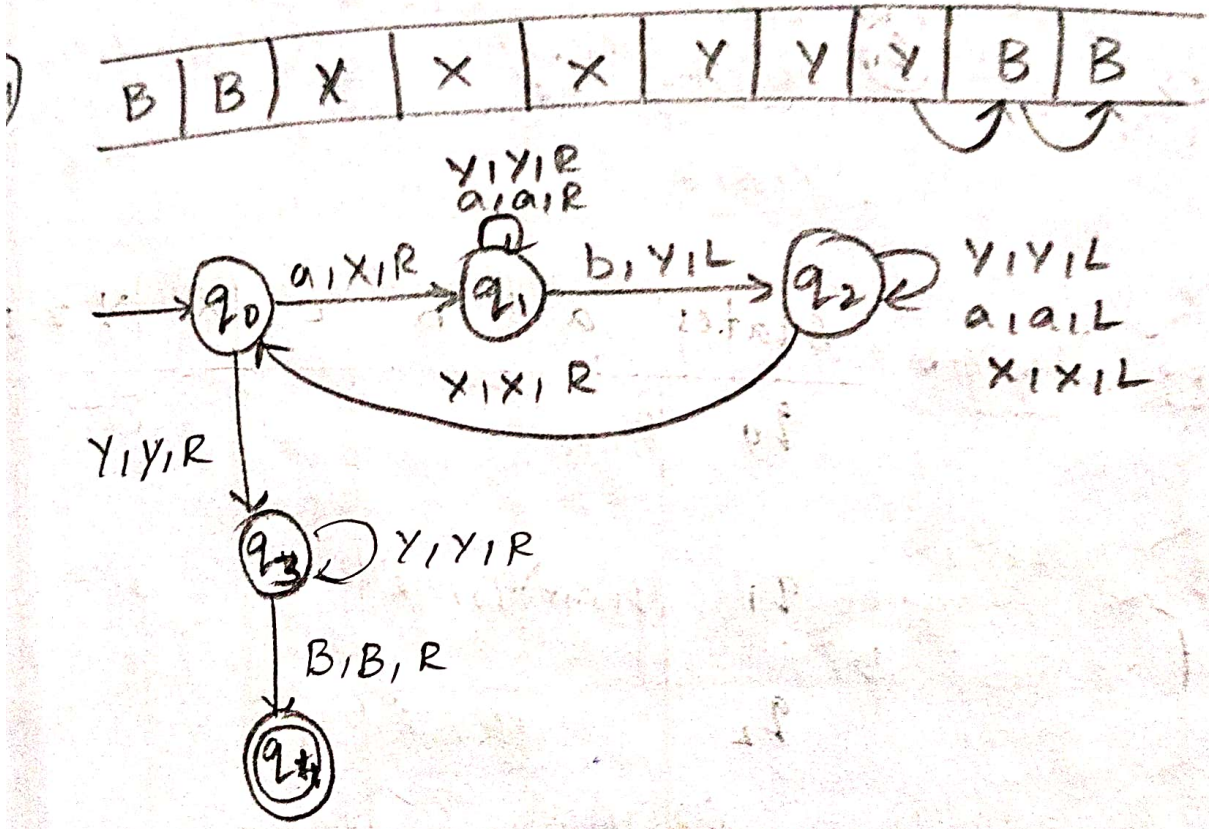
(18)


B B x x x Y Y b B B
Y,Y,R
a,a,R
q0
q1
q2
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,L
X,X,R

(19)

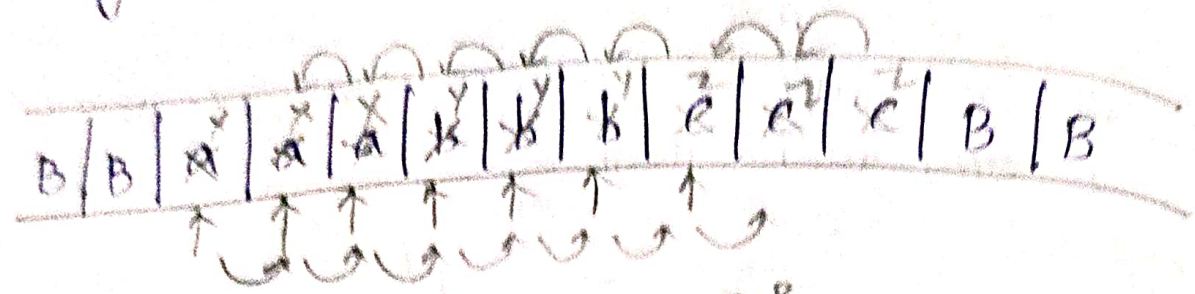

B B x x x Y Y Y B B
Y,Y,R
a,a,R
q0
q1
q2
q3
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,L
X,X,R
Y,Y,R

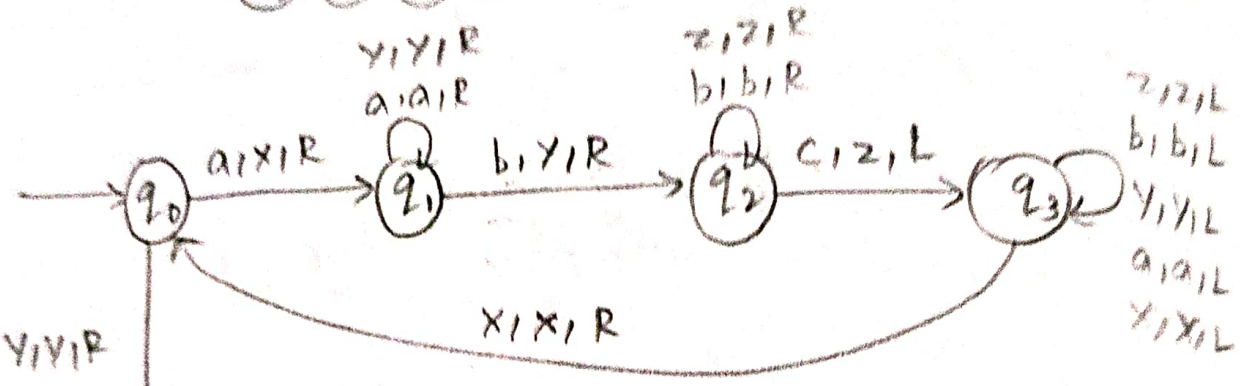

B B X X X Y Y Y B B
Y,Y,R
a,a,R
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,L
Y,Y,R
Y,Y,R
Y,Y,R

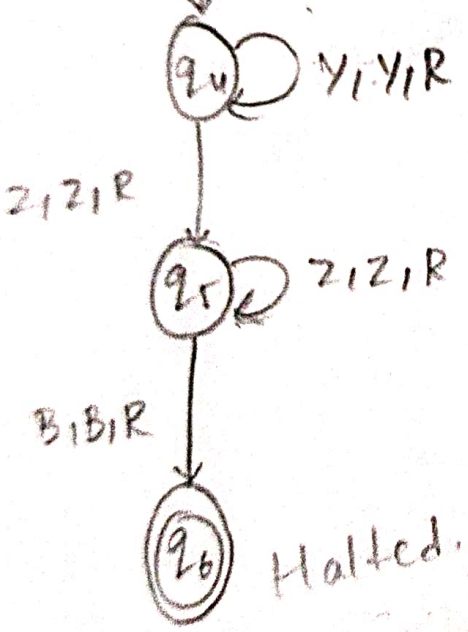

B B X X X Y Y Y B B
Y,Y,R
a,a,R
a,X,R
b,Y,L
Y,Y,L
a,a,L
X,X,L
X,X,R
Y,Y,R
Y,Y,R
B,B,R

② $L = \{a^n b^n c^n \mid \text{where } n \geq 1\}$

String: abc, aabbcc, aaabbbccc ....







States	a	b	c	x	Y	Z	B
q_0	(q_1, X, R)				(q_4, Y, R)		
q_1	(q_1, a, R)	(q_2, Y, R)			(q_1, Y, R)		
q_2		(q_2, b, R)	(q_3, Z, L)			(q_2, Z, R)	
q_3	(q_3, a, L)	(q_3, b, L)		(q_0, X, R)	(q_3, Y, L)		
q_4					(q_4, Y, R)	(q_5, Z, R)	
q_5							(q_6, B, R)

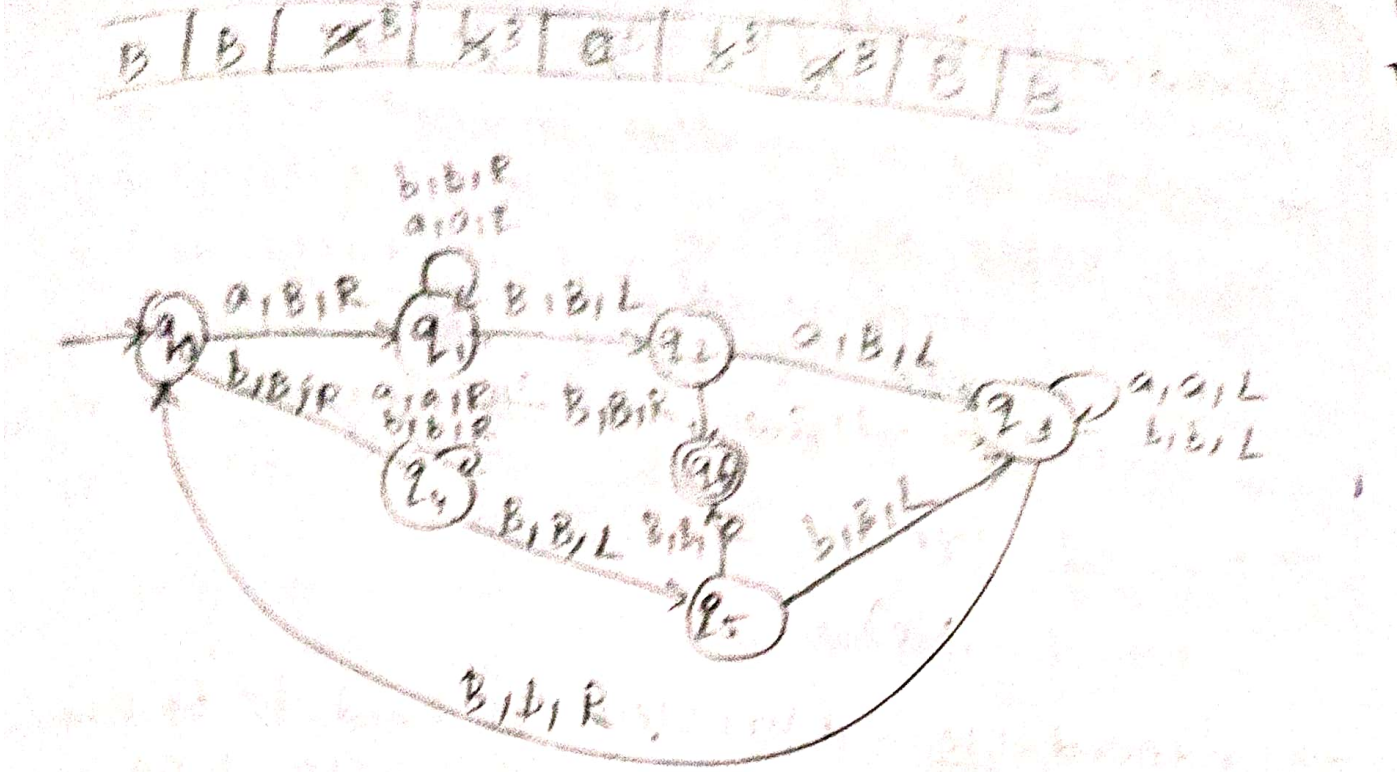
③ Odd palindrome

w c w^R

a

b

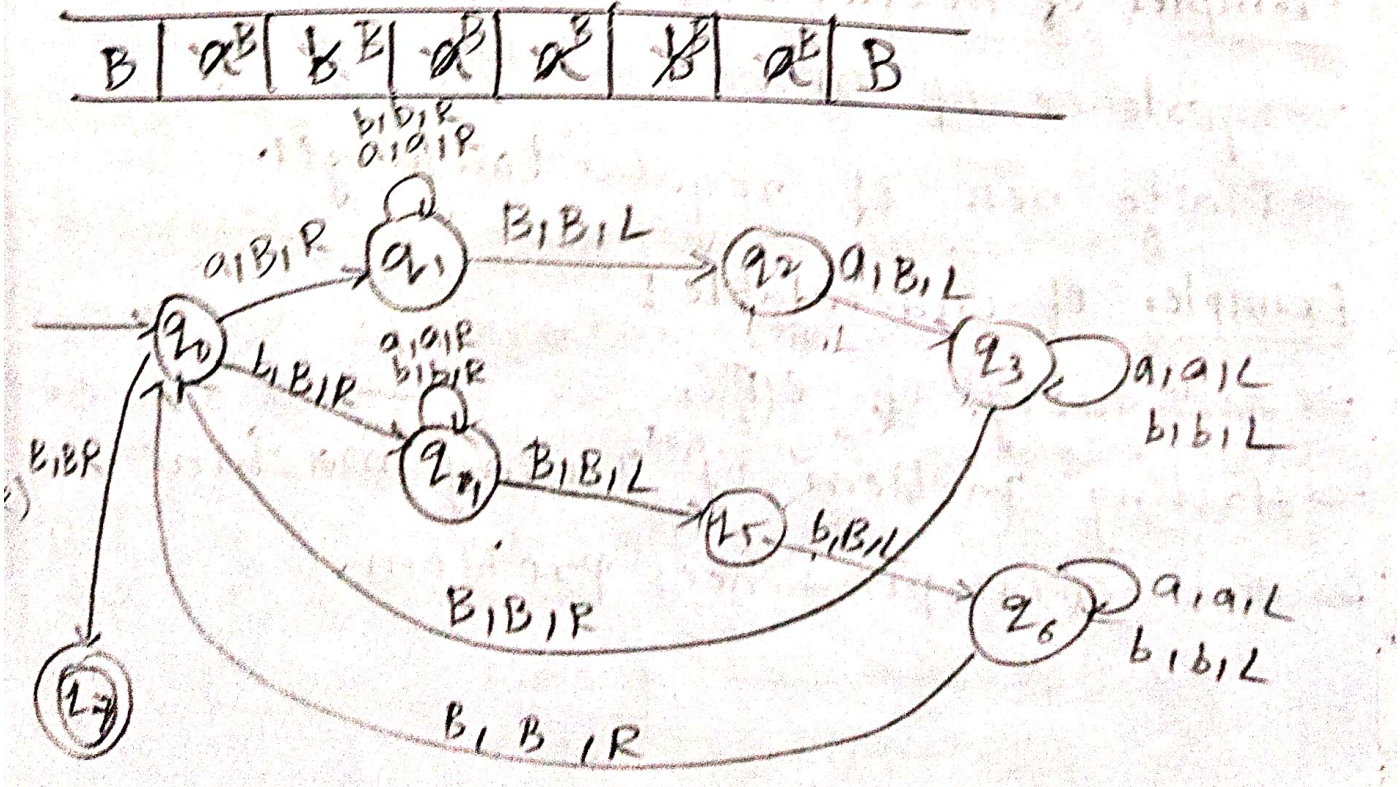
ababa



)Even Palindrome

w w^R

abaaba



Undecidability:

⇒ Decidable - a problem is said to be decidable or solvable if & only if there exists an algorithm (Turing Machine alg) that takes as input and an instance of the problem and determines whether the answer to that instance is Yes or No.

Ex: 2 + 3 = 5 (Yes)

2 + 4 = 7 (No)

⇒ Undecidable - a problem is said to be undecidable or unsolvable if and only if there is no algorithm that takes as input an instance of the problem and determines whether the answer to that instance is Yes (or) No.

Ex: $x + 5 = 7$ (it may (or) may no right)

Examples of Decidable:

⇒ Equalence of DFA

⇒ finite-ness of regular Languages.

Examples of Undecidable:

⇒ Ambiguity of CFG

⇒ Halting problem of Turing machine.

⇒ Post correspondence problem.

Undecidability Problems on Turing Machines:

The given problem is divided that can be solved by a turing machine into 2 classes.

- those that have an alg (i.e Turing machine that halts whether or not it accepts its i/p)
- Those that are only solved by turing machines that may run forever on i/ps they donot accepts

Note:

Here we will concentrate on techniques for showing problems to be "undecidable". i.e. to have no algorithm whether or not they are accepted by turing Machine that fails to halt on some inputs.

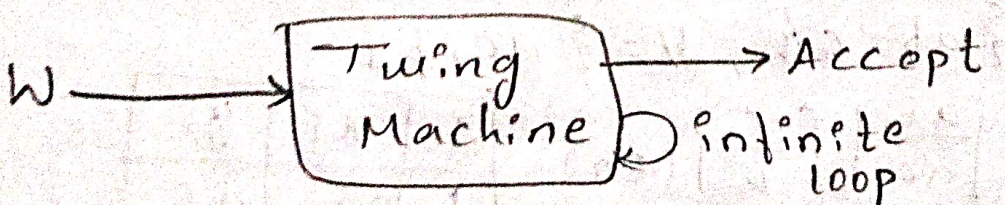
Pumping lemma - to show that the given ~~language~~ Grammar is not regular.

Turing Machine - to show the given is undecidable.

Turing Machine Languages:

→ Recursively enumerable languages:

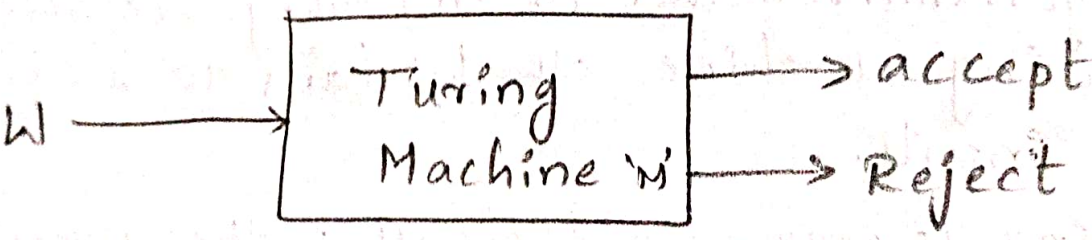
A language 'L' over an alphabet Σ is said to be recursively enumerable, if there exists a turing machine that accepts it i.e ∀ strings not in 'L', the machine goes into a infinite loop.



If the i/p string 'w' is the correct i/p string the turing machine is accepted else it will be in infinite loop.

⇒ Recursive Languages

A lang 'L' over an alphabet Σ is said to be recursive, if there exists a turing machine that recognizes it. so for all strings of 'L' the machine halts in a "final state" and for all strings not in 'L', the machine halts in "non final state".



Post Correspondence problem (PCP)

⇒ The PCP consists of 2 list of strings that are of equal length over the i/p alphabet.

⇒ The list are

$A = \{w_1, w_2, \ldots w_n\}$ & $A = \{1, 3, 5\}$

$B = \{x_1, x_2, \ldots x_n\}$. $B = \{2, 4, 6\}$

→ PCP contains set of lists – Dominos & elements (or) dice

⇒ Dominos are denoted as pair of elements

Ex: $X = \{ab, c\}$, $Y = \{ac, b\}$

$= \left[\frac{ab}{ac}\right]\left[\frac{c}{b}\right]$ (or)

i	x_i	y_i
1	ab	ac
2	c	b

To proceed further
We can use each domino for finite no. of times to find the soln.
PCP is non deteuministic problem. (Problems having multiple soln.)
The strings of dominos should match i.e the top & bottom of strings are same then it should be stop.

Ex: Prove that PCP has the soln with the 2 lists $A = \{1, 10111, 10\}$

$B = \{111, 10, 0\}$

The Dominos are $\left[\frac{1}{111}\right], \left[\frac{10111}{10}\right], \left[\frac{10}{0}\right]$

consider more nos of matching set.

$= \left[\frac{1 0 1 1 1}{1 0}\right]\left[\frac{1}{1 1 1}\right]\left[\frac{1}{1 1 1}\right]\left[\frac{1 0}{0}\right]$

Top string = 10111 1 1 10

Bottom string = 10 1 1 1 1 1 1 0

∴ Both strings are same

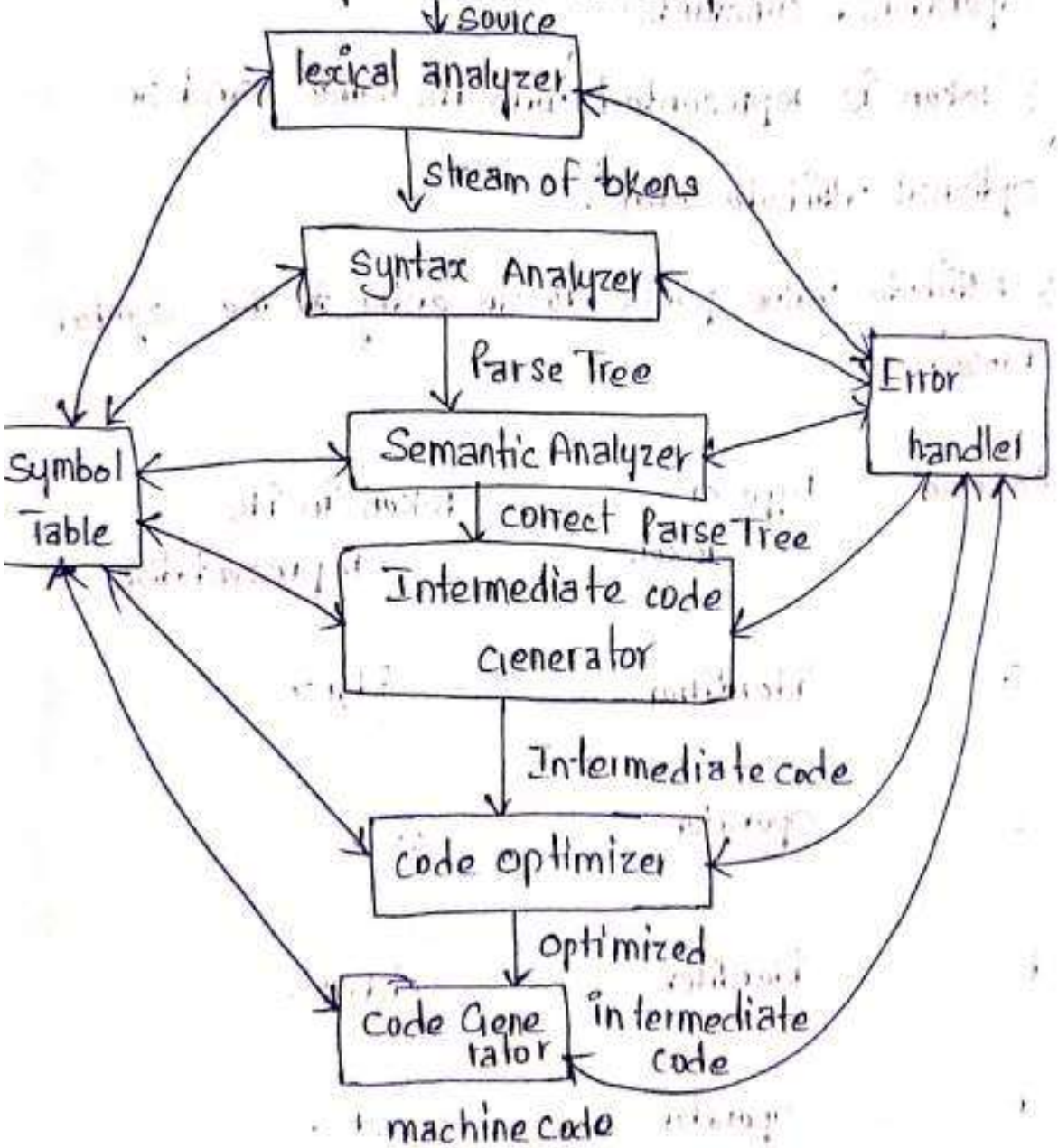
Hence, PCP has soln.

Compiler Design : UNIT : IV

Compiler: It is a system software to convert high level language to the machine code.

The structure of Compiler:-

→ There are 6 phases in the compiler.



(1) Lexical Analyzer: It reads the source program character by character and it groups the characters into a meanigful sequences called "lexlems". These are also called as "tokens".

Example: keywords, variable names (Identifiers), Operators, constants.

→ Token is represented with its name, and an optional attribute value.

→ Attribute value points to an entry to the symbol table.

lexeme	Type of token	Token in its representation
a	identifier	<id,1>
=	Operator	<=>
b	identifier	<id,2>
+	Operator	<+>
c	identifier	<id,3>
*	operator	<*>
60	~~Consonant~~ Constant	60

→ After converting it into tokens, it is represented as a = b+c*60;

"<id1> = id2+id3*60"

② Syntax Analyzer: It takes stream of tokens as the input and it gives parse tree as the output.

→ It checks the syntax of the program.

→ It is also called as "parser".

→ If there are any errors, It reports / informs to the same to the error handler.

→ The main task of the syntax analyzer is "Parse Tree" construction.

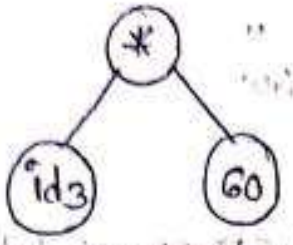
Stream of tokens ⇒ "id1 = id2+id3*60;"

→ construction of parse tree will be based on the "presedence / priority of the operators".

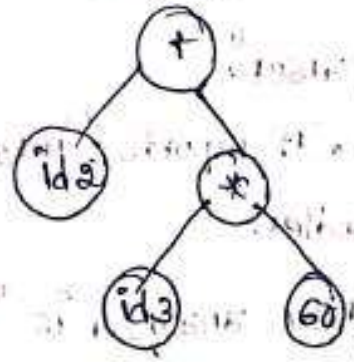
→ In parse tree take operators as "internal nodes" and "root" node.

→ It take "identifiers, constants as "leafnodes".

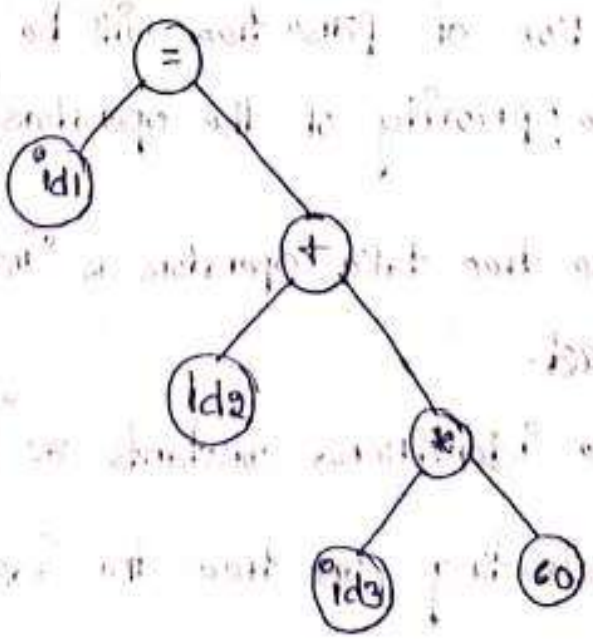
Step1: constructing parse tree for "id3*60



Step2: Next we will construct the parse tree for the id2.



Step3:- Next we will construct the parse tree for the id1.



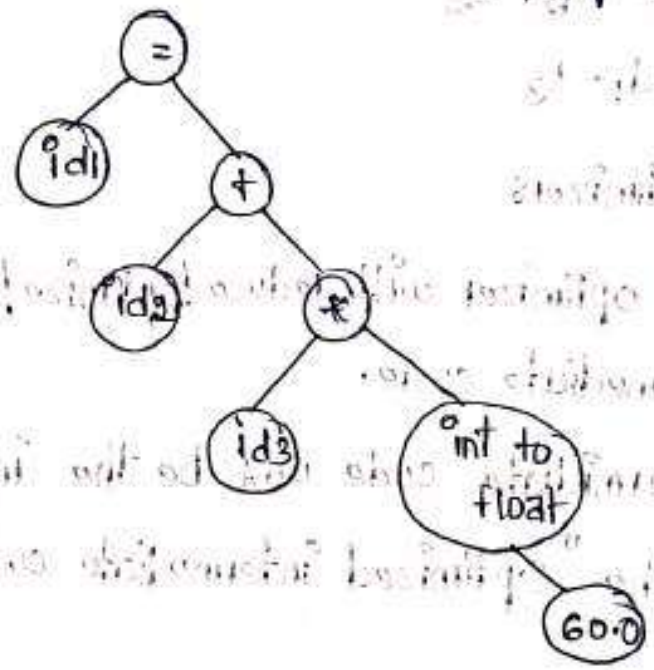
→ It is constructed from ~~top~~ down to top.

(3) Semantic Analyzer: It takes the parse tree as the input and generates the correct parse tree.

→ semantic means "meaningful".

→ It uses the type checker to (make) check the type of identifiers and to make automatic conversion if there is any need.

→ Let us assume a, b & c are floating point variables



(4) Intermediate Code Generator: It takes the correct parse tree as input and generates the "intermediate code".

→ To represent this, there are three address code.

① Instruction has at most 3 operands.

② Assignment instruction has almost one operator on the right hand side.

③ Intermediate code generator uses temperarly variables for storing intermediate results.

step1: t_1 = (int to real) 60, t_1: Temperory variable

step 2: $t_2 = id_3 * t_1$

step3: $t_3 = id_2 + t_2$

step4: $id_1 = t_3$

⑤ Code Optimizer:

→ The code optimizer will reduce/optimize/ minimize the intermediate code.

→ The intermidiate code will be the input and Generates the "optimized intermediate code".

step1: $t_1 = id3 * 60.0$

step2: $id_3 = id_2 * t_1$

⑥ Code Generator: It takes the optimized intermediate code as the input and Generates the machine code".

→ The instructions for this is LDF, STF, ADDF, MULF because the example is in the "Floating point representation.

→ For LDF and STF they need 2 operands and ADDF, MULF needs 3 operands.

step1:-
LDF R2, id3

step2:-
MULF R2, R2, #60.0

step3:-
LDF R1, id2

step4:-
ADDF R1, R, R2

step5:-
STF id3, Ri

∴ the machine code for the a = b+c*60 is, "STF id3 ; R1;"

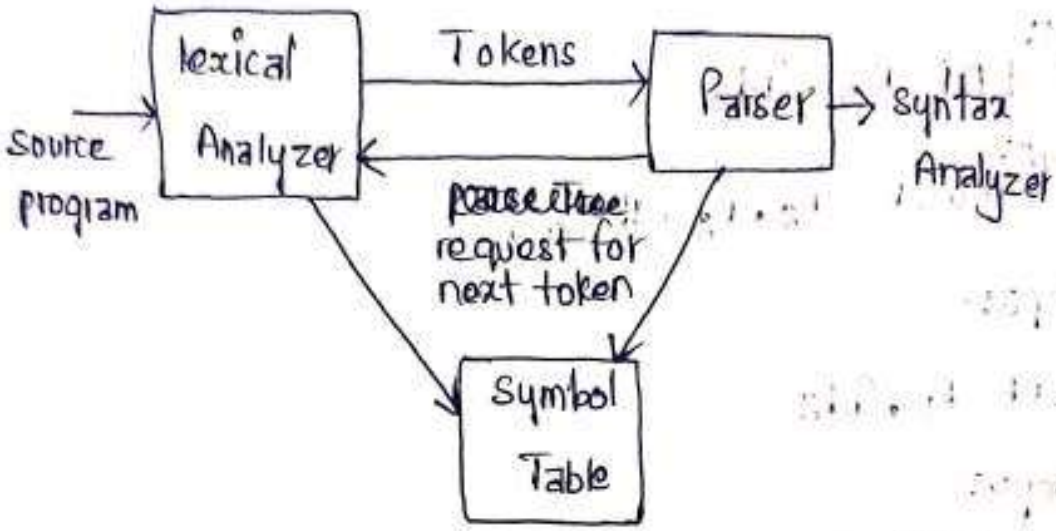
The role of lexical Analyzer:-

→ It generates the stream of tokens.

→ It eliminates the comments (For user understands). from the source program. It also eliminates the white space characters, such as blank space, tab space and new line characters-

→ It counts the no. of lines and it generates Symbol tables.

→ It produces error messages with line numbers and column numbers.



lexical Analysis versus parsing:- Benefits:-

→ The task of designed compiler is simplified. It is not easy to construct the syntax analyzer with comments.

→ Efficiency of the (lexical analyzer) compiler is improved. Buffering techniques will be used to improve the lexical analyzer.

→ Portability of compiler is enhanced. By restricting the abnormalities we can enhance the compiler.

Tokens, Patterns, lexemes:-

→ lexical analyzer meaning ful sequence or group is called lexeme.

→ Token is a pair of token name and it's attribute value or optional attribute value

→ Token names are there for keywords, constants, Identifiers, Operators.

Pattern:- Description of the form of lexeme. that we can make the

lexeme:- Group of characters, meaningful sequence of characters in the source program.

Token	Informal description	sample lexeme
if	characters i, f	if
else	characters e, l, s, e	else
comparision operations	>, <, >=, <=, !=	>, <, >=, <=, !=
id	letter followed by letters and digits	P1, A1
number	any numeric constants	0.635, 10.00
literals	any thing surounded by " "	"lasya"

Attributes for the tokens:

$e = mc^2$

one * represents the single exponential

two ** represents the two exponential

e => <id,1>

= => <=> (or) <assigned op>

m => <id,2>

* => <*> (or) <mulop>

c => <id,3>

<**> or <exp_op>

2 => <2> number, Integer value 2.

Lexical Errors:

The errors that are recognized by the lexical analyzer is

1. Spelling Errors [Ex: di not present so error]
2. unmatched strings. [if we forget the completion of the double quoate].
3. appreaience of the illegal characters. [if there are unwanted characters in the program].

① Delete one character from the matching pair.

② Inserting a missing character to the matching input.

③ Replace a character by another character.

④ Transpose to adjacent characters. (swap).

These are the error recovering options.

⑤ Delete sufficient element from the remaining input until lexical analyzer recognizes a will form tokens.

Input Buffering:- We use two buffers.

depends on the block size the buffer size will be produced.

→ The normal size of buffer is 4096 bytes.

→ using "One read" comand we can read the "n" number of characters.

→ EOF (End of file) is used to end the character where we need to stop the character at particular stage.

→ In order to operate these buffers there are two pointers

① lexeme Beginning:
② Forward pointer: } Both are at the begining of the buffer.

Then it will move forward until the end of the file, But lexeme begin will move directly to the second buffer.

Sentinels: Some special characters, For reading the each character there are two tests

① whether the Buffer is at the end read the

② What character read by the lexical Analyzer.

① when we add the special character to the Buffer at the end (sentinel) we can know that the buffer is at the end.

Recognization of Tokens:

We need to write the regular expressions for the patterns. These patterns are drawn/designed as the Transiction diagrams. These are recognized as tokens.

→ recognize the keywords, constants, identifiers.

→ regular expression for digits - "0-9".

→ regular expression for number = digit (. digits) (E(+-) digits)?

→ regular expression for white space is w (blank | Tab | newline).

→ regular expression for : letter [A - z]

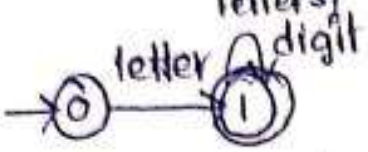
→ regular expression for id → letter (. letter | digit)*

→ regular expression for "if, else, then is same. "if, else, then.

→ regular expression for relop → <|>|<=|>=|=|<>.

Transistion diagram for Identifiers:

Ex:- abc12

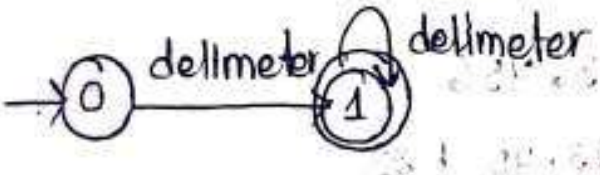


Transistion diagram for delemeter:-

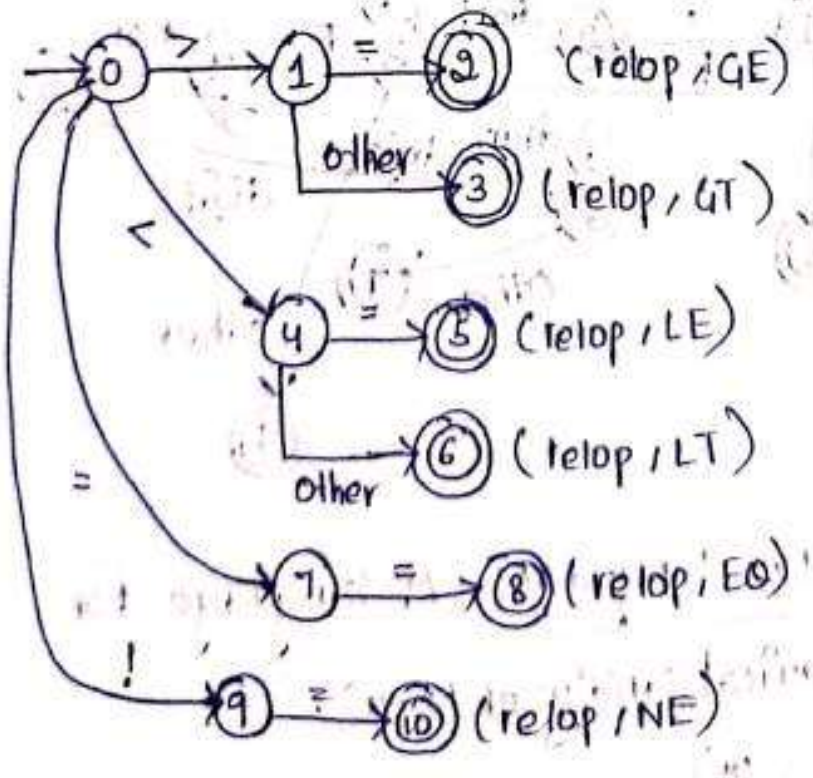
↳ It may be a blank space, newline character, tap space.

Regular expression:-

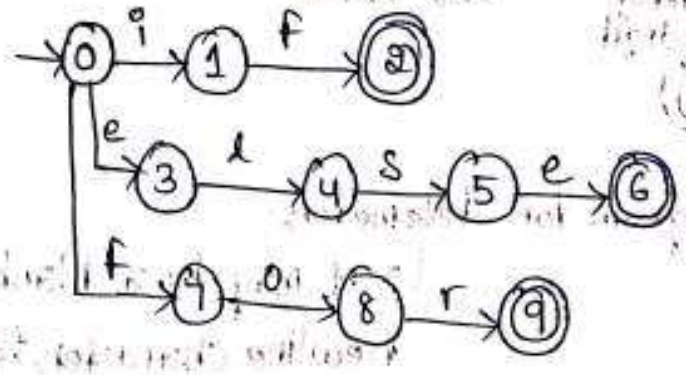
ws → delimeter (delimeter)*



Transistion diagram for Relational Operators:



Transition Diagram for the "if, else, for":-

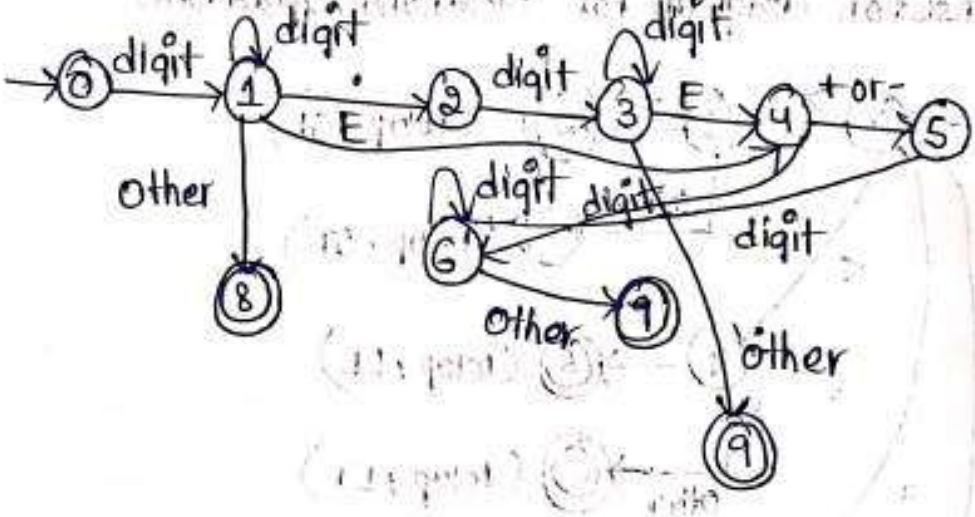


Regular expression for digits:

either digits are (int no, Float no)

int digits → 123, 456

Float digits → 123.45 E23



Lex Tool / Lex language:- A language for specifying lexical analyzer (Lex).

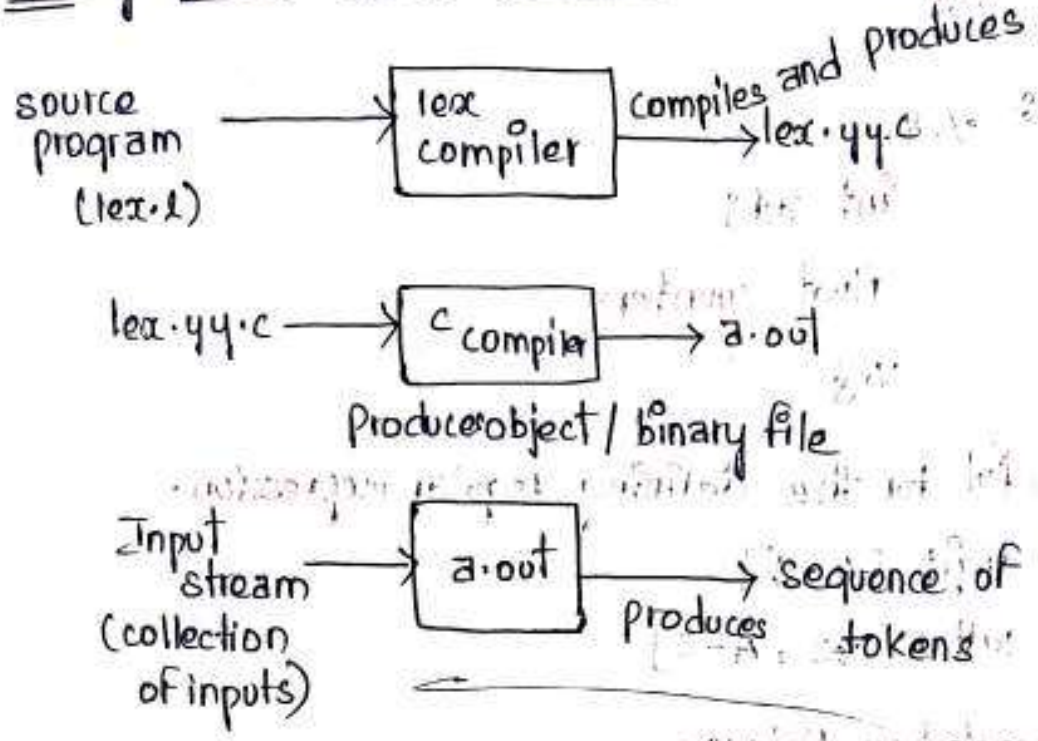
(or)

The lexical analyzer generater (Lex).

→ It is a tool or language which is used to generate the lexical analyzer.

→ It specifies the RE, RE are used to represent the patterns for the tokens.

Creating lexical Analyzer with lex:-



The Structure of lex program: It contains mainly three sections:

1. Declarations
2. Translation Rules
3. Auxilary functions.

Declarations

%% [Translation rules are started]

Translation Rules

%% [Translation rules are overed]

Auxilary functions

① Declarations:-

→ Used for to declare the c variables or constants.

```
%{
    variables, constants
%}
```

Ex:-
```
%{
    int a,b;
    float counter;
%}
```

→ useful for the defining regular expression.

digit [0-9]

letter [a-z, A-Z]

② Translation Rules:-

We can specify a rule in the form of pattern followed by an action.

pattern {Action}

↳ regular expression ↳ c language statement

If There are three rules:

```
%%
    pattern1 {Action1}
    pattern2 {Action2}
    pattern3 {Action3}
%%
```

→ These are used in order to specify the patterns.

③ Auxiliary Functions:

→ All the functions which are used they are defined in this section.

Lex program to recognize identifiers, keywords, relop, and numbers:

lex.l

```
%{
  /* lex program for recognizing tokens */
%}
letter [a-z, A-Z]
digits [0-9]
id {letter} ({letter}|{digit})*
numbers {digit}+ (.{digit}+)? (E[+-]?{digit}+)?
%%
{id}  {printf("%s is an identifier", yytext);}
if    {printf("%s is a keyword", yytext);}
else  {printf("%s is a keyword", yytext);}
"<"   {printf("%s is lessthan operator", yytext);}
```

[: same for the <=, >, >=, == and != operators].

```
{numbers}    { printf("%s is a number", yytext);}

%%
```

Input:-

ram is 123 If else!

Output:

It generates the printf statements as the output.

Syntax Analysis:

Role of a Parser:

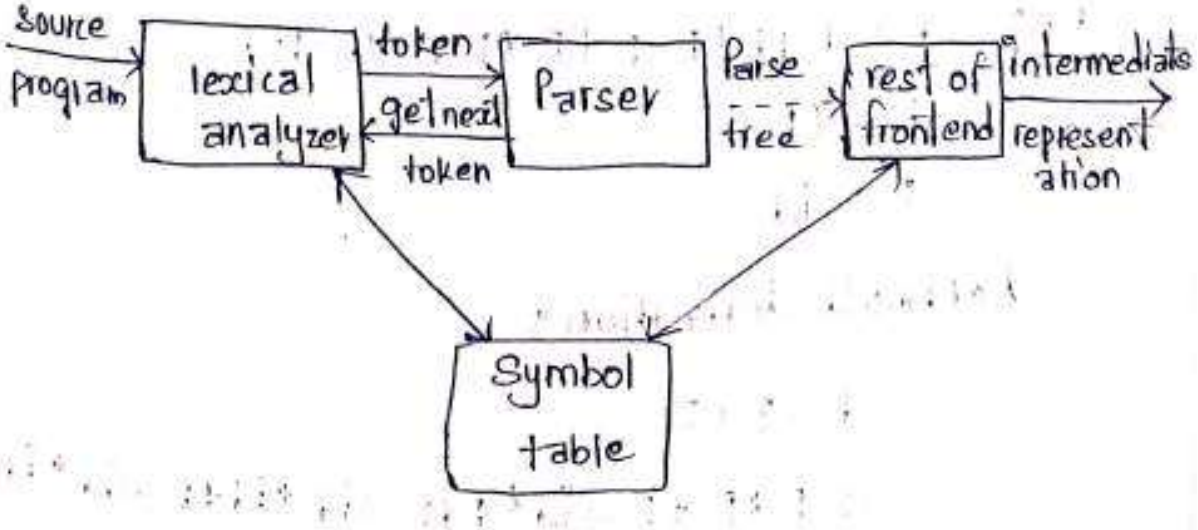
→ Parser of a grammer that takes as input string of stream of tokens from the lexical analyzer and produces output as parse tree.

→ The goal of a parser is to determine the syntatic validity of a source string is valid, a tree is built for use by the subsequent phase of the computer.

→ The tree reflects the sequence of derivations or reduction used during the parser.

→ There are two ways of identifying an elementary parse tree

1. By deriving a string from a non-terminal
2. By reducing a string of symbol to a non-terminal.



Context - Free Grammers:

→ Inherently recursive structures of a programming language are defined by a context - free Grammer.

→ In a context free Grammer, we have 4 Tuples G (V, T, P, S).

where, V → Finite set of terminals

T → Finite set of non-terminals

P → Finite set of productions.

S → start symbol.

Derivations:

1. At each derivation step, we can choose any of the non-terminal in (each derivation ste) the sentential form of G for the replacement.

2. If we always choose the left-most non-terminal in each derivation step, the derivation is called "left-most derivation".

Ex: E → E+E / E-E | E*E | E/E | -E.
E → (E)
E → id

Leftmost derivation:

E → E+E

→ E*E+E → id* E+E → id *id+E → id * id+id

Given Grammer: E→E+E | E*E | (E) | -E | id

sentence to be derived: -(id+id)

LEFTMOST DERIVATION

E → -E
E → -(E)
E → -(E+E)
E → -(id+E)
E → -(id+id)

RIGHTMOST DERIVATION

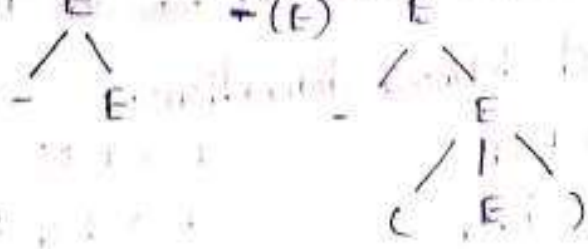
E → -E
E → -(E)
E → -(E+E)
E → -(E+id)
E → -(id+id)

- String that appear in leftmost derivation are called "left sentinal forms".
- String that appear in rightmost derivations are called "right sentinal forms".

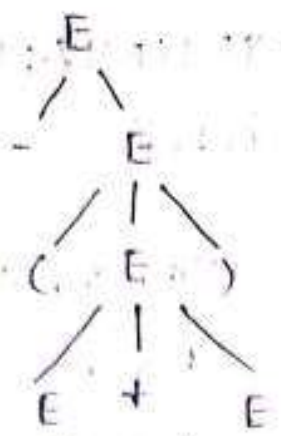
Parse Tree:

→ Inner nodes of a parse tree are non-terminal symbols
→ The leaves of a parse tree are terminal symbols.
→ A parse tree can be seen as a graphical representation of a derivation.

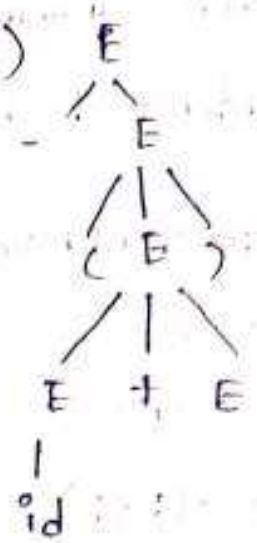
Ex:- E → -E | E → -(E) | E



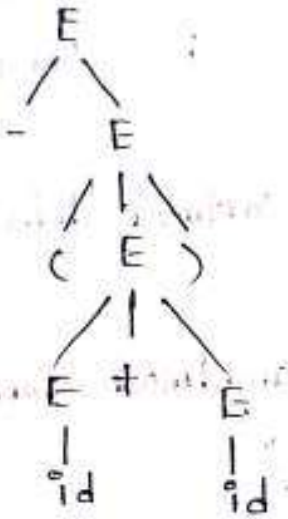
-(E+E)



-(id+E)



-(id+id)



Ambiguity:

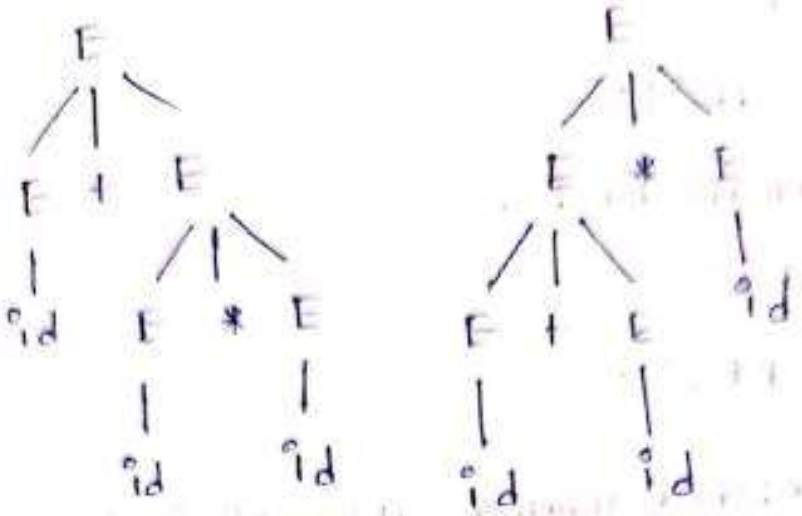
A grammer that produces more than one parse for some sentence is called "ambiguos grammer."

Ex:- Given Grammer: $L \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$

The sentence id+id*id has the following two distinct leftmost derivations:

$E \rightarrow E+E$	$E \rightarrow E*E$
$E \rightarrow id+E$	$E \rightarrow E+E*E$
$E \rightarrow id+E*E$	$E \rightarrow id+E*E$
$E \rightarrow id+id*E$	$E \rightarrow id+id*E$
$E \rightarrow id+id*id$	$E \rightarrow id+id*id$

The two corresponding parse trees are:



Eliminating left Rewrsion:

A grammer is said to be left recursive if it has a non terminal A such that there is a derivation $A \Rightarrow A\alpha$ for some string α. Topdown parsing methods cannot handle the left-recursive grammers.

⇒ If there is a production $A \to A\alpha \mid \beta$ it can be replaced with a sequence of two productions

$$A \to \beta A'$$

$$A' \to \alpha A' \mid \varepsilon$$

Example:- Consider the following grammer for arithemetic expressions:

$$E \to E+T \mid T$$

$$T \to T*F \mid F$$

$$F \to (E) \mid id$$

First eliminate the left rewrsion for E as

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

Then eliminate for T as

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

Thus the obtained grammer after eliminating left rewrsion is:

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

Algorithm to eliminate left rewrsion:

1. Arrange the non-terminals in some order $A_1, A_2, \ldots A_n$.

2. for i := 1 to n do begin

    For j := 1 to i-1 do begin

        replace each production of the form

        $A_i \rightarrow A_j \gamma$

        by the productions $A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \ldots \delta_k \gamma$.

where $A_j \to \delta_1 | \delta_2 | \ldots | \delta_k$ are the current A_j-productions;

end

eliminate the immediate left recursion among the A_j-productions

end.

Left Factoring:

Left Factoring is a grammer transformation that is useful for producing a grammer suitable for predictive parsing.

→ If there is any production $A \to \alpha\beta_1 | \alpha\beta_2$, it can be un rewritten as

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2$.

Consider the grammer, G: $S \to iEtS | iEtSeS | a$

$E \to b$

left factored, this grammer beomes:

$S \to iEtSS' | a$

$S' \to eS | \varepsilon$

$E \to b$

Top down parsing:

It can be viewed as an attempt to find a left most derivation of an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

→ There are two types of top-down parsing:

1. Recursive ~~Decent~~ Descent Parsing:-

→ Recursive descent parsing is one of the top-down parsing techniques that use a set of recursive procedures to scan its input.

→ This parsing method may involve "backtracking", that makes repeated scans of the input.

Example:

A left-recursive grammer can cause a recursive-descent parser to go into an infinite loop.

→ Hence, elimination of left-recursion must done before parsing.

Consider the Grammer for arithemetic expressions:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id.$

After eliminating the left-recursion the grammer becomes:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Now we can write the procedure for grammer as follows:

Rewrisive Procedure:

```
Procedure E()
begin
   T();
   EPRIME();
End
   Procedure EPRIME()
     begin
       If input_symbol = '+' then
           ADVANCE();
           T();
           EPRIME();
        end
```

```
Procedure T()
    begin
        F();
        TPRIME();
    End
procedure TPRIME()
        begin
            If input_symbol = '*' then
             ADVANCE();
             else if input_symbol = '(' then
             ADVANCE();
             E();
             else if input_symbol = ')' then
             ADVANCE();
          end
         else ERROR();
```

stack Implementation:

Procedure	Input String
E()	id + id * id
T()	id + id * id
F()	id + id * id
ADVANCE()	id + id * id
TPRIME()	id + id * id
EPRIME()	id + id * id
ADVANCE()	id + id * id
T()	id + id * id
F()	id + id * id
ADVANCE()	id + id * id
TPRIME()	id + id * id
ADVANCE()	id + id * id
F()	id + id * id
ADVANCE()	id + id * id
TPRIME()	id + id * id

2. Predictive Parsing:-

→ predictive parsing is a special case of rewrsive descent parsing where no backtracting is required.

→ The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Predictive Parsing program:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If X = a = $, the parser halts and announces successful completion of parsing.
2. If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal, the program consults entry M[X, a] of the parsing table M. This entry will either be an X-production of the grammer or an error entry.

If M[X, a] = [X → UVW], the parser replaces X on top of the stack by UVW.

If M[X, a] = error, the parser calls an error recovary routine.

Predictive parsing table Construction:

→ The construction of a predictive parser is aided by two functions associated with a grammer G:

1. FIRST 2. FOLLOW.

Rules for first():

1. If X is terminal, then FIRST(X) is {X}.

2. If $X \to \varepsilon$ is a production, then add ε to FIRST(X)

3. If X is non-terminal and $X \to a\alpha$ is a production then add a to FIRST(X).

4. If X is non-terminal and $X \to Y_1 Y_2 \dots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is all of FIRST(Y_1), ..., FIRST(Y_{i-1}); that is, $Y_1, \dots Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST(Y_j) for all $j = 1, 2, \dots, k$ then add ε to FIRST(X).

Rules for follow():

1. If S is a start symbol, then FOLLOW(S) contains $.

2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ε is placed in follow(B).

3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm:-

Input: Grammer G

Output: Parsing table M

Method:

1. For each production $A \to \alpha$ of the grammer, do steps 2 and 3.

2. For each terminal a in FIRST(α), add $A \to \alpha$ to M[A, a].

3. If ε is in FIRST(α), add $A \to \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and \$ is in FOLLOW(A), add(A), add $A \to \alpha$ to M[A, \$].

4. Make each undefined entry of M be error.

Example:-

Consider the following Grammer:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

After eliminating left-recursion the grammer is

$E \to TE'$

$E' \to +TE' \mid \varepsilon$

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id$

First ():

FIRST (E) = { (, id }

FIRST (E') = { +, ε }

FIRST (T) = { (, id }

FIRST (T') = { *, ε }

FIRST (F) = { (, id }

Follow ():

FOLLOW (E) = { $,) }

FOLLOW (E') = { $,) }

FOLLOW (T) = { +, $,) }

FOLLOW (T') = { +, $,) }

FOLLOW (F) = { +, *, $,) }

Predictive parsing table:

Non terminal	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

LL(1) Grammer:

The parsing table entries are single entries. So each location has not more than one entry.

→ This of grammer is called LL(1) grammer.

Consider this following grammar:

$S \to iEtS \mid iEtSeS \mid a$

$E \to b$

After eliminating left Factoring, we have

$S \to iEtSS' \mid a$

$S' \to eS \mid \varepsilon$

$E \to b$

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals:

FIRST(S) = {i, a}

FIRST(S') = {e, ε}

FIRST(E) = {b}

FOLLOW(S) = {$, e}

FOLLOW(S') = {$, e}

FOLLOW(E) = {t}

parsing table:-

Non-Termial	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S'→eS S'→ε			S'→ε
E		E→b				

Since, there are more than one production, the grammer is not LL(1) grammar.

Actions Performed in predictive parsing:

1. Shift
2. Reduce
3. Accept
4. Error

Implementation of Predictive parser:

1. Elimination of left rewirsion, left factoring and ambiguos grammer.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table
4. parse the given input string using stack and parsing table.

Bottom-up Parsing:-

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called "bottom-up Parsing".

→ A General type of bottom-up parser is a "shift-reduce parser".

Shift-Reduce Parsing:-

→ It is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginining at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammer:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

The sentence to be recognized is "abbcde".

REDUCTION (LEFTMOST)		RIGHTMOST DERIVATION
abbcde	(A → b)	S → aABe
aAbcde	(A → Abc)	→ aAde
aAde	(B → d)	→ aAbcde
aABe	(S → aABe)	→ abbcde

The reductions trace out the right-most derivation in reverse.

Actions in shift-reduce parser:

- shift - The next input symbol is shifted onto the top of the stack.
- reduce - The parser replaces the handle within a stack with a non-terminal.
- accept - The parser announces successful completion of parsing.

• error - The parser discover that a syntax error has occured and calls an error recovary routine.

Stack Implomentation of shift-reduce parsing:

Stack	Input	Action
$	$id_1 + id_2 * id_3 $$	shift
$id1	$+ id_2 * id_3 $$	reduce by E→id
$E	$+ id_2 * id_3 $$	shift
$E+	$id_2 * id_3 $$	shift
$E+id2	$* id_3 $$	reduce by E→id
$E+E	$* id3 $$	shift
$E+E*	$id3 $$	shift
$E+E*id3	$$	reduce by E→id
$E+E*E	$$	reduce by E→E*E
$E+E	$$	reduce by E→E+E
$E	$$	accept

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift-reducing parser.

1. shift - reduce Conflict:

The parser cannot decide whether to shift or to reduce.

Example:-

Consider the Grammer:

E → E+E | E*E | id and input id+id*id

stack	Input	Action	stack	Input	Action
$E+E	*id$	Reduce by E→E+E	$E+E	*id$	shift
$E	*id$	shift	$E+E*	id$	shift
$E*	id$	shift	$E+E*id	$	Reduce by E→id
$E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$E			$E		

2. Reduce - reduce Conflit:

The parser cannot decide which of several reduction to make.

Consider the Grammer:

$M \rightarrow R+R \mid R+c \mid R$

$R \rightarrow c$

and input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c$	shift	$	c+c$	shift
$c	+c$	Reduce by R→c	$c	+c$	Reduce by R→c
$R	+c$	shift	$R	+c$	shift
$R+	c$	shift	$R+	c$	shift
$R+c	$	Reduce by R→c	$R+c	$	Reduce by M→R+c
$R+R	$	Reduce by M→R+R	$M	$	
$M	$				

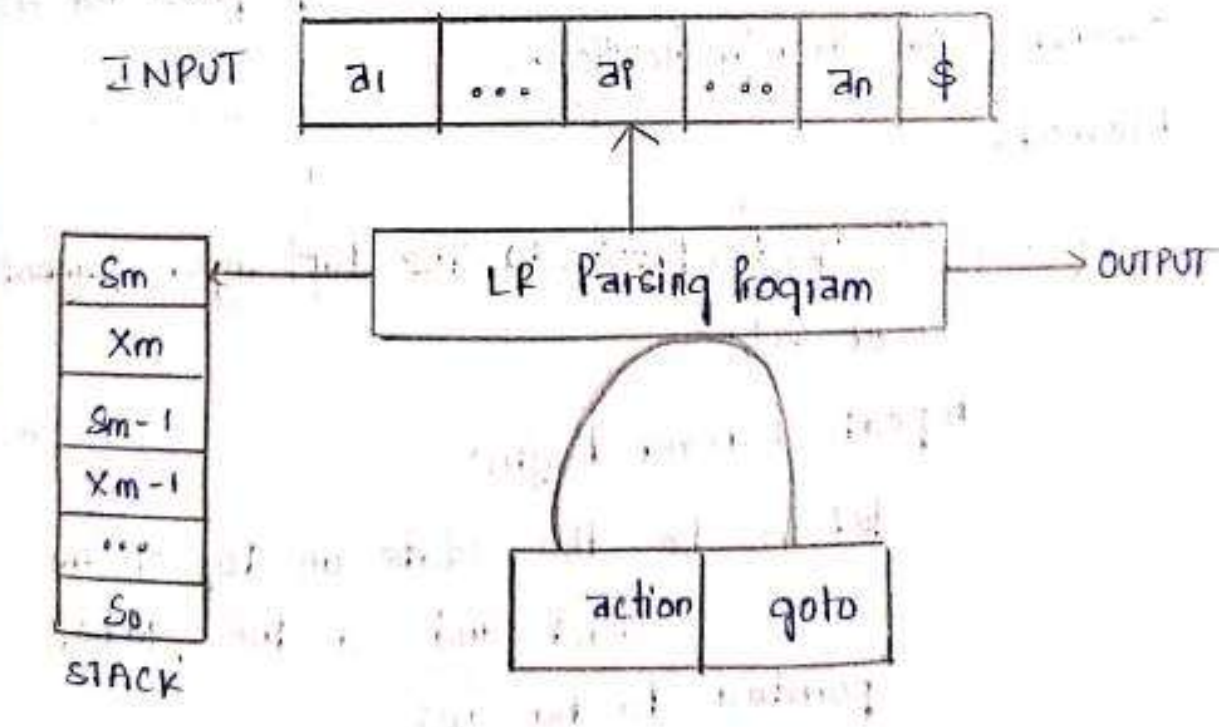
LR Parsers:-

→ An efficient bottom-up syntax analysis technique that can be used to paise a large class of CFG is called LR(k) paising.

→ The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and 'k' for the number of input symbols. When 'k' is comitted, it is assumed to be 1.

The LR parsing algorithm:

The schematic form of LR parser:



→ The driver program is the same for all LR parser.

→ The parsing program reads characters from an input buffer one at a time.

→ The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2 \ldots X_ms_m$.

→ The parsing table consists of two parts: action and goto functions.

LR parsing algorithm:

Input: An input string w and an LR parsing table with functions action and goto for grammer G.

Output: If w is in L(G), a bottom-up parse for w; Otherwise, an error indication.

Method:

```
set ip to point to the first input symbol
    of w$;
repeat forever begin
    let s be the state on top of the
        stack and a the symbol
    pointed to by ip;
if action[s,a] = shift s' then begin
```

```
    push a then s' on top of the stack;
    advance ip to the next input symbol
  end
  else if action[s,a] = reduce A→β then begin
      pop 2*|β| symbols off the stack;
      let s' be the state now on top of stack;
      Output the production A→β
    end
    else if action[s,a] = accept then
        return
      else error()
    end
```

Constructing SLR(1) Parsing table:

To perform SLR parsing, take grammer as input and do the following:

1. Find LR(0) items.
2. Completing the closure
3. Compute goto(I, X) where,
   I is set of items
   X is grammar symbol.

Algorithm for Construction of SLR parsing table:

Input: An augmented Grammer G'.

Output: The SLR parsing table functions actions and goto for G'.

Method:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$ the collection of sets of LR(0) items for G'.

2. state i is constructed from I_i, The parsing functions for state i are determined as follows:

   a) If $[A \to \alpha \cdot a\beta]$ is in I_i. The and goto$(I_i, a) = I_j$, then set action$[i,a]$ to "shift j". Here a must be terminal.

   b) If $[A \to \alpha \cdot]$ is in I_i, then set action$[i,a]$ to "reduce $A \to \alpha$" for all a in FOLLOW(A).

   c) If $[S' \to S \cdot]$ is in I_j, then set action$[i,\$]$ to "accept".

3. The goto transitions for state i are connected for all non-terminals A using the rule:

   If goto$(I_i, A) = I_j$, then goto$[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error".

5. The initial state of the parser is the one constructed from the set of items containing [s'→.s].

Example for SLR parsing:-

Construct SLR parsing for the following grammer:

E→E+T|T

T→T*F|F

F→(E)|id

The Given Grammar is:

E→E+T - (1)

E→T - (2)

T→T*F - (3)

T→F - (4)

F→(E) - (5)

F→id - (6)

Step1: Convert given grammer into augmented grammer.

E'→E

E→E+T

E→T

T→T*F

T→F

F→(E)

F→id

Step 2: Find LR(0) items

I_0: $E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

GOTO(I_0, E)

I_1: $E' \to E.$

$E \to E.+T$

$T \to T.*F$

GOTO(I_0, F)

I_3: $T \to F.$

GOTO(I_0, ()

I_4: $F \to (.E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$F \to .(E)$

$F \to .id$

$T \to .F$

GOTO(I_0, id)

I_5: $F \to id.$

GOTO(I_1, +)

I_6: $E \to E+.T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

GOTO(I_2, *)

I_7: $T \to T*.F$

$F \to .(E)$

$F \to .id$

GOTO(I_4, E)

I_8: $F \to (E.)$

$E \to E.+T$

GOTO(I_4, T)

I_2: $E \to T.$

$T \to T.*F$

GOTO(I_4, F)

I_3: $T \to F.$

GOTO(I_6, T)

I_9: $E \to E+T.$

$T \to T.*F$

GOTO(I_6, F)

I_3: $T \to F.$

GOTO(I_6, ()

I_4: $F \to (.E)$

GOTO(I_6, id)

I_5: $F \to id.$

GOTO (I_7, ()

I_4: F → (.E)

E → .E + T

E → .T

T → .T * F

T → .F

F → .(E)

F → .id

GOTO (I_7, id)

I_5: F → id.

GOTO (I_8,))

I_{11}: F → (E).

GOTO (I_8, +)

I_6: E → E + .T

T → .T * F

T → .F

F → .(E)

F → .id

GOTO (I_9, *)

I_7: T → T * .F

F → .(E)

F → .id

GOTO (I_4, ()

I_4: F → (.E)

E → .E + T

E → .T

T → .T * F

T → .F

F → .(E)

F → id

FOLLOW (E) = {$,), +}

FOLLOW (T) = {$, +,), *}

FOLLOW (F) = {*, +,), $}

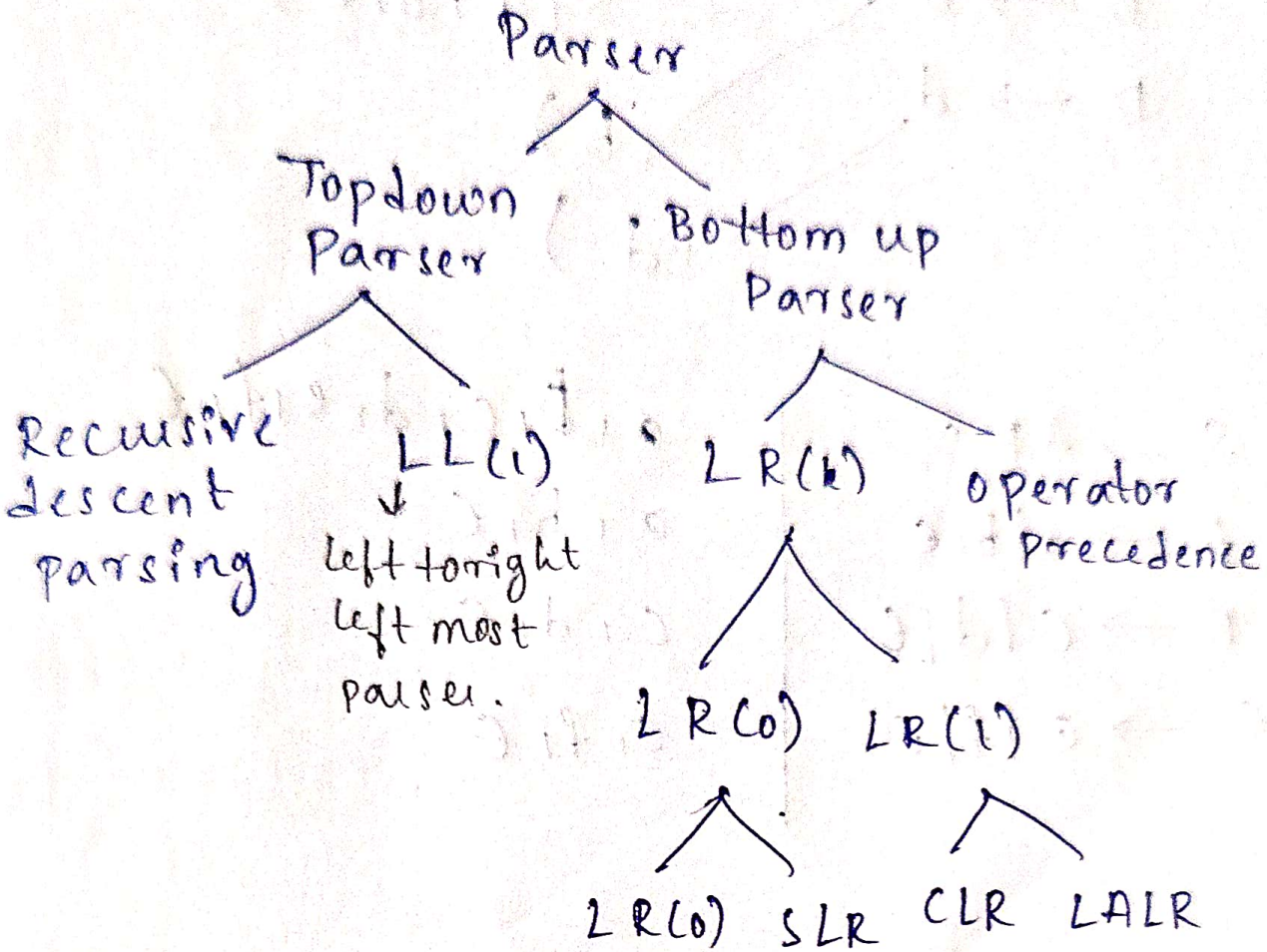
SLR parsing table:

	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
I_0	S_5			S_4			1	2	3
I_1		S_6				ACC			
I_2		r_2	S_7		r_2	r_2			
I_3		r_4	r_4		r_4	r_4			
I_4	S_5			S_4			8	2	3
I_5		r_6	r_6		r_6	r_6			
I_6	S_5			S_4				9	3
I_7	S_5			S_4					10
I_8		S_6			S_{11}				
I_9		r_1	S_7		r_1	r_1			
I_{10}		r_3	r_3		r_3	r_3			
I_{11}		r_5	r_5		r_5	r_5			

Stack Implementation:

STACK	INPUT	ACTION
0	id+id*id$	GOTO(I_0, id) = S_5; shift
0 id 5	+id*id$	GOTO(I_5, +) = r_6; reduce by F→id
0 F 3	+id*id$	GOTO(I_0, F) = 3 GOTO(I_2, +) = r_4; reduce by T→F
0 T 2	+id*id$	GOTO(I_0, T) = 1 GOTO(I_1, +) = S_6; shift
0 E 1	+id*id$	GOTO(I_6, id) = S_5; shift
0 E 1 + 6	id*id$	GOTO(I_5, *) = r_6; reduce by F→id
0 E 1 + 6 id 5	*id$	GOTO(I_6, F) = 3 GOTO(I_3, *) = r_4; reduce by T→F
0 E 1 + 6 F 3	*id$	GOTO(I_6, T) = 9 GOTO(I_9, *) = S_7; shift
0 E 1 + 6 T 9	*id$	GOTO(I_7, id) = S_5; shift
0 E 1 + 6 T 9 * 7	id$	GOTO(I_5, $) = r_6; reduce by F→id
0 E 1 + 6 T 9 * 7 id 5	$	GOTO(I_7, F) = 10 GOTO(I_{10}, $) = r_3; reduce by T→T*F
0 E 1 + 6 T 9 * 7 F 10	$	GOTO(I_7, F) = 10 GOTO(I_9, $) = r_3; reduce by T→T*F
0 E 1 + 6 T 9	$	GOTO(I_6, T) = 9 GOTO(I_9, $) = r_1; reduce by E→E+T
0 E 1	$	GOTO(I_0, E) = 1 GOTO(I_1, $) = accept

Parser Techniques



⟹ Parsing is a process of deriving strings from a given grammar.

⟹ This parsing comes under syntax analysis phase.

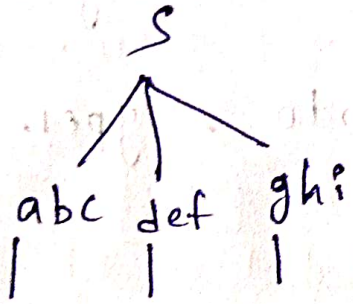
How to find first & follow?

Eg: ① S → abc | def | ghi

② First (terminal) = terminal

③ first (ε) = ε

first() & follow()



terminals
(a, B, g)

Eg: S → ABcd | ghi | jKL

A → a | b | c

B → b

D → d

(a, b, c, g, j)

(a, b, c)

(b)

(d)

Eg:

S → ABC	a, b, c, d, e, f, ε
A → a | b | ε	a, b, ε
B → c | d | ε	c, d, ε
C → e | f | ε	e, f, ε

$E \rightarrow TE'$	id, C
$E' \rightarrow *TE' / \epsilon$	*, ϵ
$T \rightarrow FT'$	id, C
$T' \rightarrow \epsilon / +FT'$	ϵ, +
$F \rightarrow id / (E$	id, C

find follow()

- follow (A) contains set of all terminals present immediate in right of 'A'.

Rules:

- we focus on RHS
- follow of start symbol is '$'
- If the production has given

$S \rightarrow ACD$

$C \rightarrow a/b$

fo(A) = First (C) = {a, b}

fo(D) = $S \rightarrow ACD$ = $

g: $S \rightarrow aSbS / bSaS / \epsilon$

fo(S) = b, a, $

g: $S \rightarrow AaAb / BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

fo(A) = $

fo(B) = $

LL(1) / Non-recursive predictive Parser / Dynamic Parser (without backtracking)

↑ Passing technique or left derivation

LL(1) Parsing

↓

Represent the scanning of the input will be done from the left to right.

(1) represents the number of "look ahead"

look ahead means how many symbols are you going to see when you want to make a decision

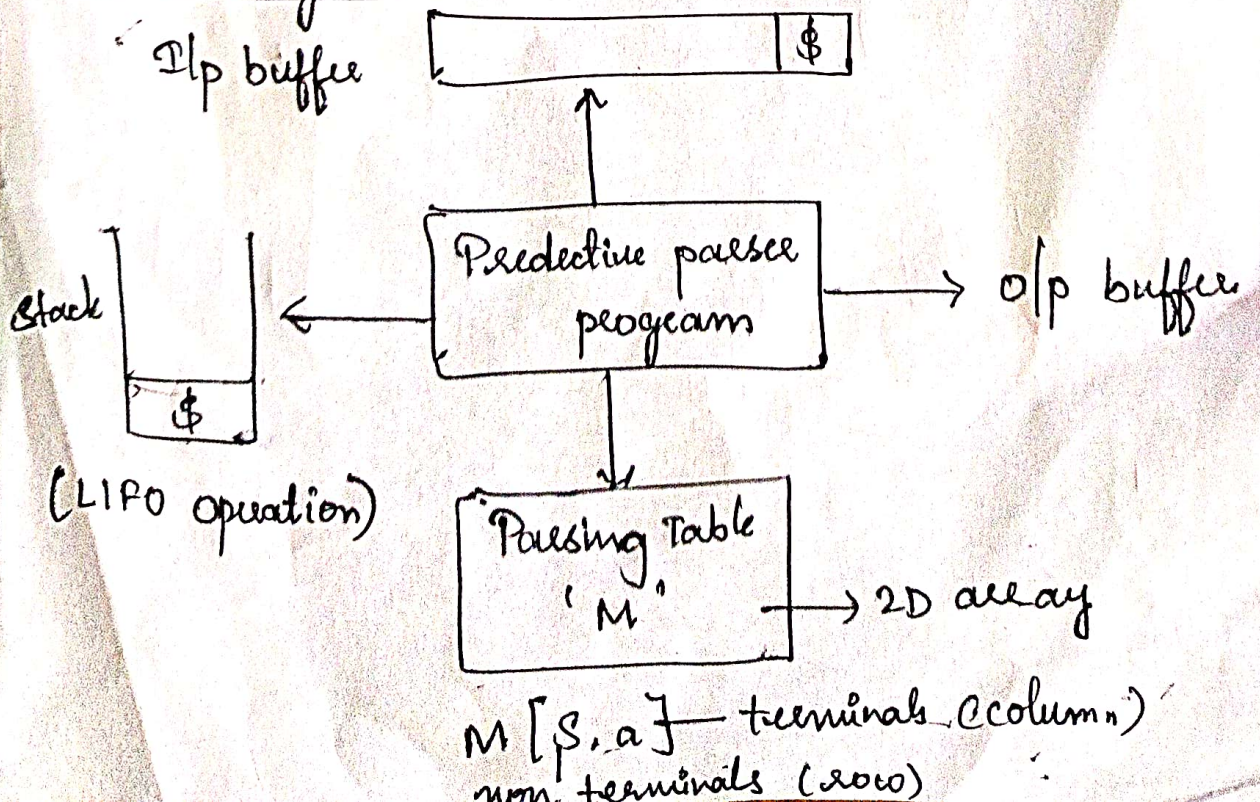
Conditions:

→ The grammar is free from left recursion.

→ The grammar should not be ambigous

→ The grammar has to be left factored in so that the grammar is deterministic grammar.

Block diagram:



Ex: Grammar Passing Table

	First	Follow
1 2 S → (L) | a	{ (, a }	{ $,) }
L → S L' 3	{ (, a }	{) }
L' → ∈ 4 | , S L' 5	{ ∈, , }	{) }

Parse table

	(	)	a	,	$
S	1		2		
L	3		3		
L'		4		5	

when we get ∈ we should see follow of L'

S → (L)

S → a

L → S L' → S ka first

L' → ∈

L' → , S L'

LALR:

→ LALR Parser is a lookahead LR Parser. (Left to Right, Right most derivative)
→ It is the most powerful parser which can handle large classes of grammar
→ The size of CLR parsing table is quite large as compared to other parsing table.
→ LALR reduces the size of the table
→ LALR works similar to CLR
→ The only difference is, it combines the similar states of CLR parsing table into 1 single state.
→ The general syntax becomes [A → α.B, a]
where A → α.B is production
a is a terminal

→ LALR < CLR, SLR

Parsers: Top down, bottom up

Example:

$S' \to S$
$S \to CC$
$C \to aC$
$C \to b$

Production 1 is
$S' \to .S,$
for this product look a head symbol is $
$S' \to .S, \$$ → look a head symbol

① $S' \to .S, \$$ → After S we don't have any variable so $ is final
$S \to .CC, \$$ → C we have as variable so $C \to .a/b$
$C \to .aC, a/b$ $C \to .aC, \$ \to C \to a/b$
$C \to .b, a/b$ non-terminal is present so we should.
goto (I0, S)

When we use goto then . increases 1 position

I_1: $S' \to S., \$$
I_2 goto (I0, C)
$C \to C.C, \$$
$C \to .aC, \$$

I_3 goto(I_0, a)

C → a . C, a/b

C → . aC, a/b

C → . b, a/b

I_4 goto(I_0, b)

C → b . , a/b

I_5 goto(I_0, C)

S → C . , $

goto(I_0, a)

I_6 C → a . C, $

C → . aC, $

C → . b, $

goto(I_0, b)

I_7 C → b . , $

goto(I_3, a)

C → aC . , a/b

goto(I_3, C)

I_8 C → aC . , a/b

goto(I_3, a)

C → a . C, a/b

C → . aC, a/b

C → . b, a/b

} I_3

goto(I_3, b)

C → b . , a/b } I_4

goto(I_6, C)

I_9 C → aC . , $

goto(I_6, a)

C → a . C, $ } I_6

goto ~~goto~~ goto(I_6, b)

C → b . , $ } I_7

goto(

I_3 & I_6 are same, so we have to combined them to produce a new state it has same production but different look ahead symbols.

We combine them to reduce the size of the table.

I_3 & I_6 → I_{36}

C → a . C, a/b/$

C → . aC, a/b/$

C → . b, a/b/$

I_4 & I_7 → I_{47}

C → b . , a/b/$

I_8 & I_9 → I_{89}

C → aC . , a/b/$

	terminals			non terminals	
	a	b	$	S	C
0	S36	S47		1	2
1			Accepted		
2	S36	S47			5
36	S36	S47			89
47			r3		
5			r1		
89	r2	r2	r2		

CLR Parsing

⇒ Stands for canonical LR parser.
⇒ More powerful parser.
⇒ It makes use of "lookahead symbols."
⇒ General syntax:

$$[A \longrightarrow \alpha B, a]$$

where

$A \longrightarrow \alpha B$ ⟹ is the production &
a ⟹ is the terminal or right end marker $

LR(1) items = LR(0) items + look ahead.

↓

Large set of items.

LALR
CLR > LR(1) (uses)

Parsing Techniques

CLR — Canonical Left to right
Right most derivative.

In SLR we Cal — LR(0) items
" CLR " " — LR(1)
= LR(0) + look a head.

Steps

⇒ Argumented grammer
⇒ Cal LR(1)
⇒ conith parse table.

Ex $S \longrightarrow CC$

$C \longrightarrow cC \mid d$

Step 1

Argumented grammar.

$S' \longrightarrow S \quad 0$

$S \longrightarrow CC \quad 1$

$C \longrightarrow cC \quad 2$

$C \longrightarrow d \quad 3$

Last

Reduce productions

'.' should present at end of production

'r'

Step 2 cal LR(1) items.

I_0 $S' \longrightarrow .S, \$$

$S \longrightarrow .CC, \$$

$C \longrightarrow .cC, c/d$

$C \longrightarrow .d, c/d$

goto(I_0, S)

I_1 $S' \longrightarrow S., \$$

after '.' there is no nonterminal.

goto(I_0, C)

I_2 $S \longrightarrow C.C, \$$

$C \longrightarrow .cC, \$$

$C \longrightarrow .d, \$$

goto(I_0, c)

I_3 $C \longrightarrow c.C, c/d$

$C \longrightarrow .cC, c/d$

$C \longrightarrow .d, c/d$

goto(I_0, d)

I_4 $C \longrightarrow d., c/d$

Look ahead

$S' \longrightarrow .S, \$$

$.S\epsilon, \$$

first($\epsilon, \$$)

($\$$)

$S \longrightarrow .CC, \$$

first($C, \$$)

= (C)

= {c, d}

$S \longrightarrow C.C, \$$

first($C\epsilon, \$$) = $\$$

$C \longrightarrow c.C, c/d$

goto (I_0, C)

$S \rightarrow C\,C\,.\,,\,\$$

goto (I_2, c)

$C \rightarrow c.C\,,\,\$$

$C \rightarrow .cC\,,\,\$$

$C \rightarrow .d\,,\,\$$

goto (I_2, d)

$C \rightarrow d.\,,\,\$$

goto (I_3, C)

$C \rightarrow cC.\,,\,c/d$

goto (I_3, c)

$C \rightarrow c.C\,,\,c/d$ (Present in I_3)

$C \rightarrow .cC\,,\,c/d$

$C \rightarrow .d\,,\,c/d$

goto (I_3, d)

I_4 $C \rightarrow d.\,,\,c/d$ (Present in I_4)

first (C, $) = $

goto (I_6, C)

I_9 $C \rightarrow cC.,\$$

goto (I_6, c)

I_6 $C \rightarrow c.C,\$$ already completed

$C \rightarrow .cC,\$$

$C \rightarrow .d,\$$

goto (I_6, d)

I_7 $C \rightarrow d.,\$$

perform shift operation for terminals.

Construction of CLR Pause table

	Action (Terminals)			goto (non-terminals)	
State	c	d	$	S	C
0	S_3	S_4	Accepted	1	2
1					
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

① Recursive Decent parser:

⇒ We have to write recursive procedure for each and every non terminal that is available in Grammar.

Procedure

→ Non terminal
  └ Call the fun.

→ terminal
  └ (Compare i/p with terminal then increment)

↔ If a non terminal produces 3 productions then we have to write those 3 in corresponding non terminals.

→ No need to define main fun../declare any variables.

Ex

$E \rightarrow iE'$

$E' \rightarrow +iE' / \epsilon$ (i+i$)

```
E()
{
 if (input == 'i')
        input++;
 EPRIME();
```

```
EPRIME()
{
 if (input == '+')
    { input ++;
    }
      if (input == 'i')
      {
         input ++;
         EPRIME();
      else
         return;
}
```

Ex

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E) / id$

(id+id$)

```
E()
{
  T();
   EPRIME();
}
EPRIME()
{
  if (input == '+')
   {     input++;
         T();
         EPRIME();
   }else
         return;
}
```

```
T()
{
   F();
  TPRIME();
}
TPRIME()
{
  if (input == '*')
   {     input++
         F();
         TPRIME();
   }
   else
         return;
}
F()
{
   if (input == '(')
   {     input++;
         E();
      if (input == ')')
         input++;
   }
   else if (input == 'id')
         input++;
}
```

(2) SLR(1) :

$S \longrightarrow cAd$

$A \longrightarrow able$

String - ced $

Step (1) Augmented grammar & numbering

$S' \longrightarrow S$ (0)

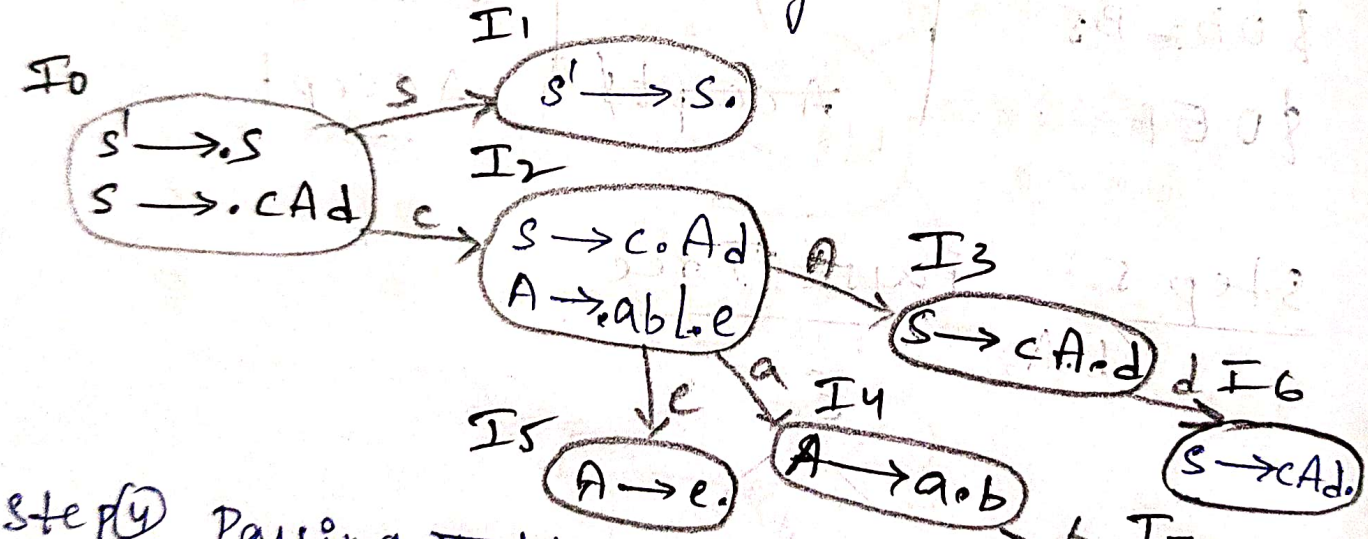
$S \longrightarrow cAd$ (1)

$A \longrightarrow ab$ (2) / e (3)

Step (2) first() & follow()

$S' \longrightarrow S$	{c}	$
$S \longrightarrow cAd$	{c}	$
$A \longrightarrow able$	{a, e}	d

Step (3) construct DFA diagram



Step (4) Parsing Table

States	Action						Goto	
	a	b	c	d	e	$	S	A
I_0			S_2				1	
I_1						A		
I_2	S_4				S_5			3
I_3				S_6				
I_4		S_7						
I_5				r_3				
I_6						r_1		
I_7				r_2				

a Pgm ... high level ... language ... PL

reduction

I_5 : $A \rightarrow e.$

[d, 5] : r_3 — no

↓

follow

I_6 : $S \rightarrow cAd.$

[$, 6] : r_1

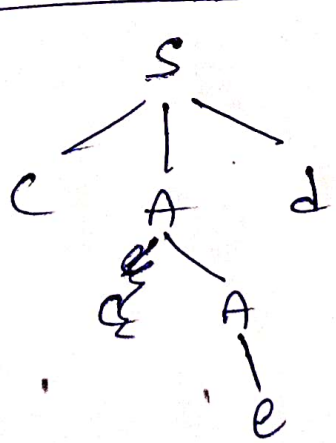
I_7 : $A \rightarrow ab.$

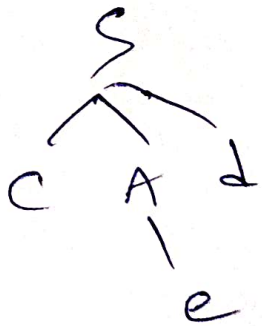
[d, 7] : r_2

Step ④ stack Implementation

state	Input	Action
0	ced $	shift ($c \rightarrow s_2$)
0 c_2	ed $	shift ($e \rightarrow s_5$)
0 $c_2 e_5$	d $	reduce ($A \rightarrow e_3$)
0 $c_2 A_3$	d $	shift ($d \rightarrow s_6$)
0 $c_2 A_3 d_6$	$	reduce ($S \rightarrow cAd$)
0 S_1	$	Accept.

step ⑥ Parse tree





Bottom Up Parser:

$E \rightarrow BB$

$B \rightarrow cB \mid d$

String → ccdd$

Ex ① LR(0):

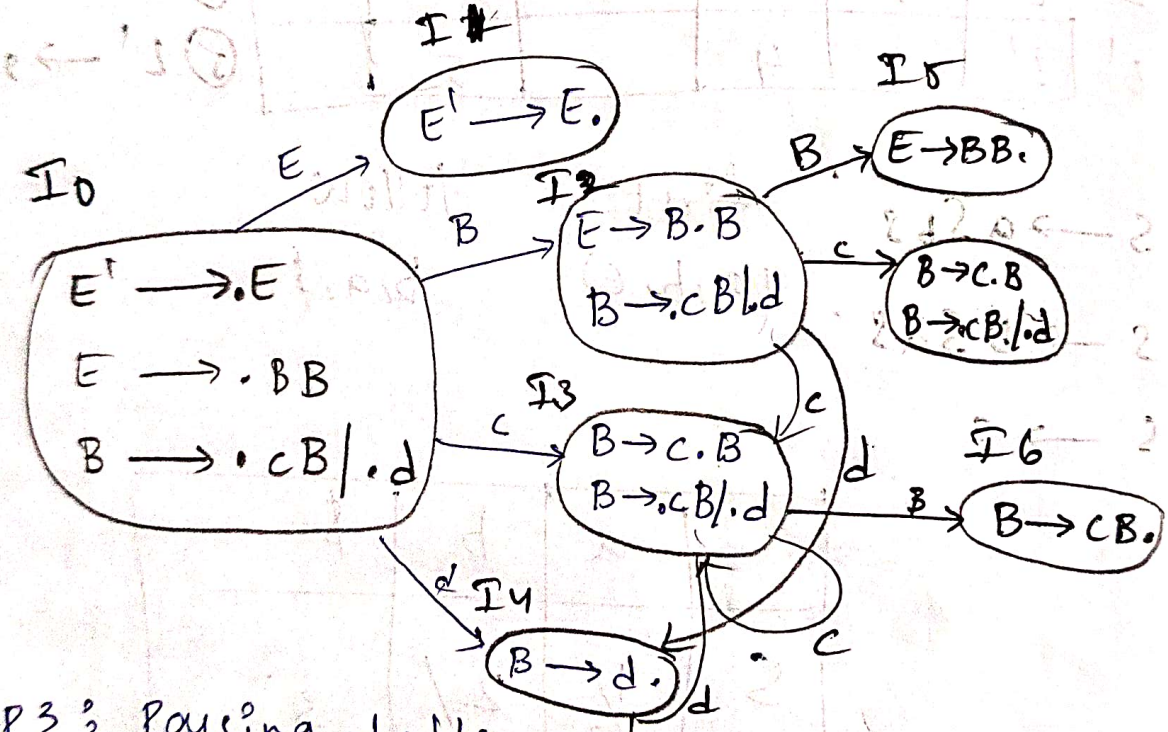
Step 1: Augment the grammer + number it

$E' \rightarrow E$ — (0)

$E \rightarrow BB$ — (1)

$B \rightarrow cB \mid d$ — (2) (3)

Step 2: Draw DFA diagram



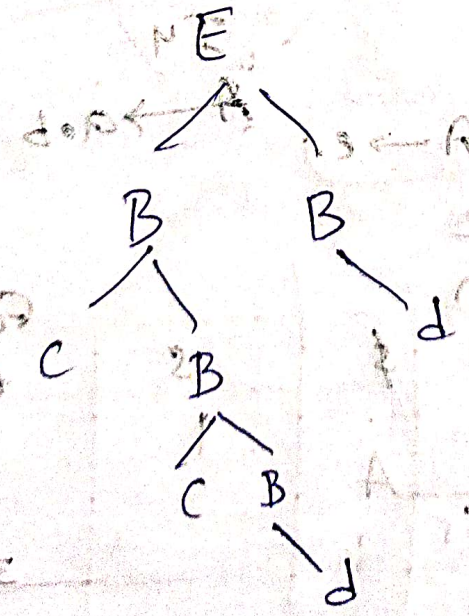
Step 3: Parsing table.

State	Action			Goto	
	c	d	$	B	E
I_0	S_3	S_4		2	1
I_1			A (accept)		
I_2	S_3	S_4		5	
I_3	S_3	S_4		6	
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

step 4 : Stack Implementation

Stack	Input	Action
$0	ccdd$	shift C → s_3
$0C3	cdd$	shift c → s_3
$0C3C3	dd$	shift d → s_4
$0C3C3d4	d$	reduce B → d
$0C3C3B6	d$	reduce B → cB
$0C3B6	d$	reduce B → cB
$0B2	d$	shift d → s_4
$0B2d4	$	reduce B → d
$0B2B5	$	reduce E → BB
$0E1	~~Accept$~~	Accept

step 5 : Parse Tree



Unit - V

Syntax-directed definations:

Synthesized Attributes:

Synthesized attribute for a nonterminal A at a parse-tree node N is defined by a sementic rule associated with the production N.

Inherited Attribute:

An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N.

Evaluate an SDD at the nodes of a parse tree:

To visualize the translation specified by an SDD, it helps to work with parse trees, even though a translator need not actually build a parse tree.

Example:

Production	Semantic Rules
$L \to En$	$L.Val = E.Val$
$E \to E_1 + T$	$E.Val = E_1.Val + T.Val$
$E \to T$	$E.Val = T.Val$
$T \to T_1 * F$	$T.Val = T.Val \times F.Val$
$T \to F$	$T.Val = F.Val$
$F \to (E)$	$F.Val = E.Val$
$F \to digit$	$F.Val = digit.lexval$

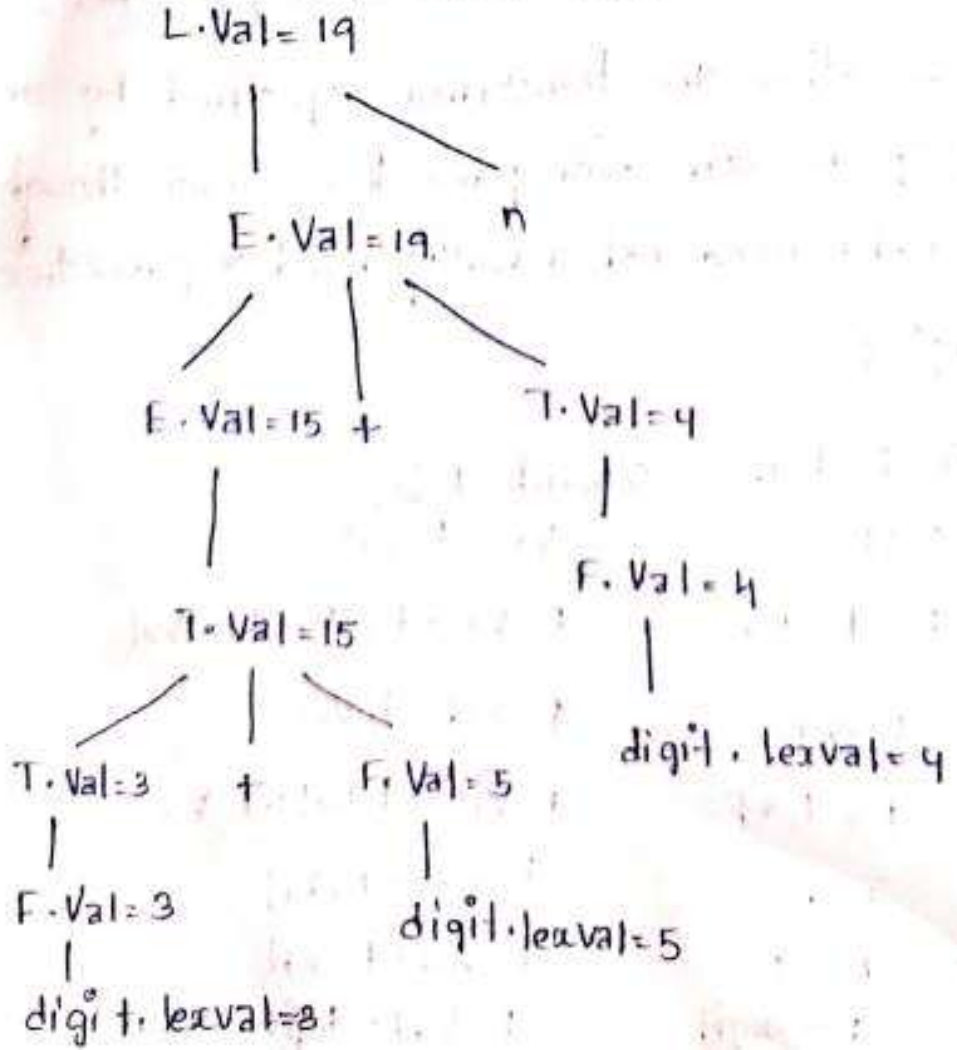
The SDD based for arithemetic expressions with operators "+" and "*". It evaluates expressions terminated by an endmarker n.

→ In the SDD, each of the non-terminals has a single synthesized attribute, called "val".

→ Suppose that the terminal digit has a synthesized attribute lexval, which is an integer value returned by the lexical analyzer.

Example:

3*5+4n



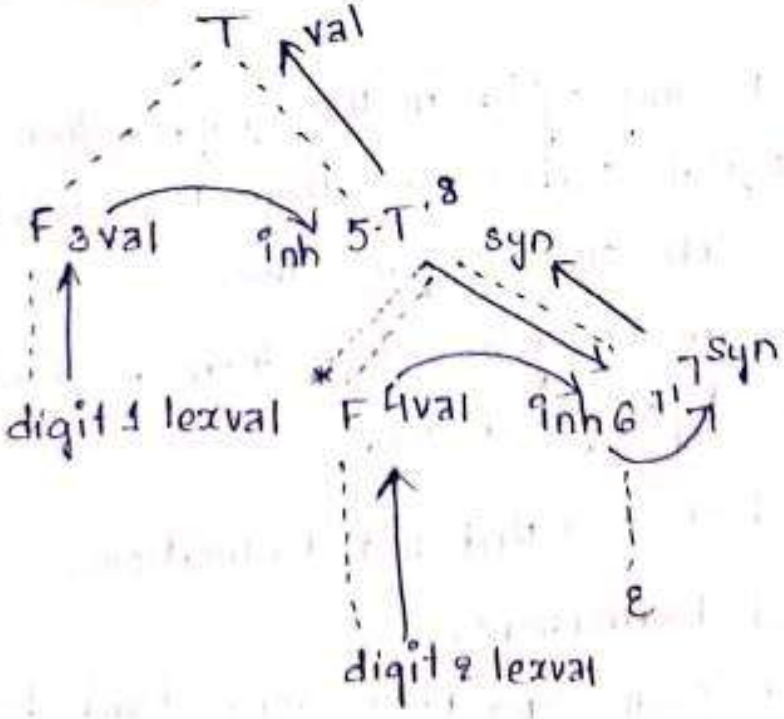
Evaluation Orders for SDD's:

Dependency Graphs:

A dependency graph depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.

Example:

The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree:



Odering the Evaluation of Attributes:

→ The dependency graph characteize the possible orders in which we can evaluate the attributes at the various nodes of a paise tiee.

→ If the dependency graph has an edge from node M to N, then the attribute corresponding to M must be evaluated before the attribute of N.

→ Thus, the allowable order of evaluation are those sequences of nodes $N_1, N_2, \ldots, N_k$ such that if there is an edge of the dependency graph for N_i to N_j then $i<j$.

→ Such an Odering embeds a directed graph into a linear order, and is called a "topological sort" of the graph.

→ If there is any cycle in the graph, then there are no topological sorts, i.e; there is no way to evaluate the SDD on this paise tiee.

→ If there are no cycles, then there is always at least one topological sort.

S - Attributed & L - Attributed Definations:-

S - Attributed Definations:

For a given SDD, it is very haid to tell whether there exists any paise trees whose

dependency graphs have cycles.

→ Translations can be implemented by using classes of SDD's that guranlee an evalution order, because they didnot permit dependency graphs with cycles.

→ There are two classes which are implementing efficiently in connection with top-down or bottom-up parsing.

L-Attributed Definations:-

The second class of SDD's is called L-attributed definations.

→ The class is behind that between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left.

→ Each attribute must be either:

1) synthesized con

2) Inherited, but the rules are as follows:

Suppose, there is a production $A \to X_1 X_2, \ldots X_n$ there is an inherited attribute X_i a computed by a rule associated with his production.

a) Inherited attributes associated with the head A

b) Either inherited or synthesized attributes associated with the occurances of symbols $x_1, x_2 \ldots x_{i-1}$ located to the left of x.

Semantic Rules with Controlled side Effects:

→ A balance between attributes grammars and translation schemes.

→ Attribute Grammers have no side effects and allow any evaluation order consistent with the dependency graph.

→ Translation schemes impose left-to-right evaluation and allow semantic actions to contain any program fragment.

Side Effects in SDD's are one of following ways:

i) permit incidental side effects that do not constrain attribute evaluation.

ii) Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order. The constraints can be thought of as implicit edges added to the dependency graph.

Syntax-Directed Translation Schemes:

Postfix Translation schemes:

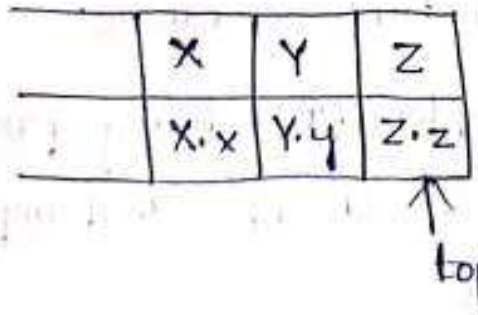
→ Simplest SDD implementation occurs when the grammar is passed bottom-up and the SDD is S-attributed.

→ We can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production SDT's with all actions at the right ends of the production bodies are called "postfix SDT's".

Parser-Stack Implementation of Postfix SDT's:

→ The attributes can be put on a stack in a place were they can be found during the reduction.

→ Best plan is to place the attributes along with the grammar symbol in records on the stack itself.



State/grammar Symbol
Synthesized attribute(s)

→ We can allow for more attributes, either by making the records large enough or by putting pointers to records on the stack.

→ If attributes are all Synthesized, and actions occurs at the ends of the productions, then we can compute the attributes for the head.

→ If we reduce by a production as A→XYZ then we have all attributes of X,Y,Z at known positions on the stack.

SDT's with Actions Inside productions:

→ An action may be placed at any position within the body of a production.

→ It is performed immediately after all symbols to its left are processed.

→ If the parse is bottom-up, then we perform action a as soon as this occurance of appears on the top of the parsing stack.

→ If the parse top-down, then we perform a just before we attempt to expand this occurances.

→ SDT's can be implemented during parsing, include postfix SDT's and a class of SDT's that implements L-attribute definations.

Eliminate Left Recursion from SDT's:-

→ When transforming the grammar, treat the actions as if they were terminal symbols.

→ The principle is based on the idea that the grammar transformation perserves the order of the terminals in the generated string.

→ The actions are executed in the same order in any left-to-right parse, top-down or bottom-up.

→ For eliminating left recursion is to take two productio

$A \to A\alpha \mid \beta$

that Generates strings consisting of a β and any number of α's and replace them by productions that generates the same using a new nonterminal R of the first production.

$A \to \beta R$

$R \to \alpha R \mid \varepsilon$

Example:- Consider the following E-productions from an SDT for translating infix expression into postfix notation:

$E \to E_1 + T$ { Print ('+'); }

$E \to T$

→ If we apply the standard transformation to E, the remainder of the left-recursive production is

α = +T {Print ('+');} and β, the body of the production is T. If we introducing R for the remainder of E, we get of productions:

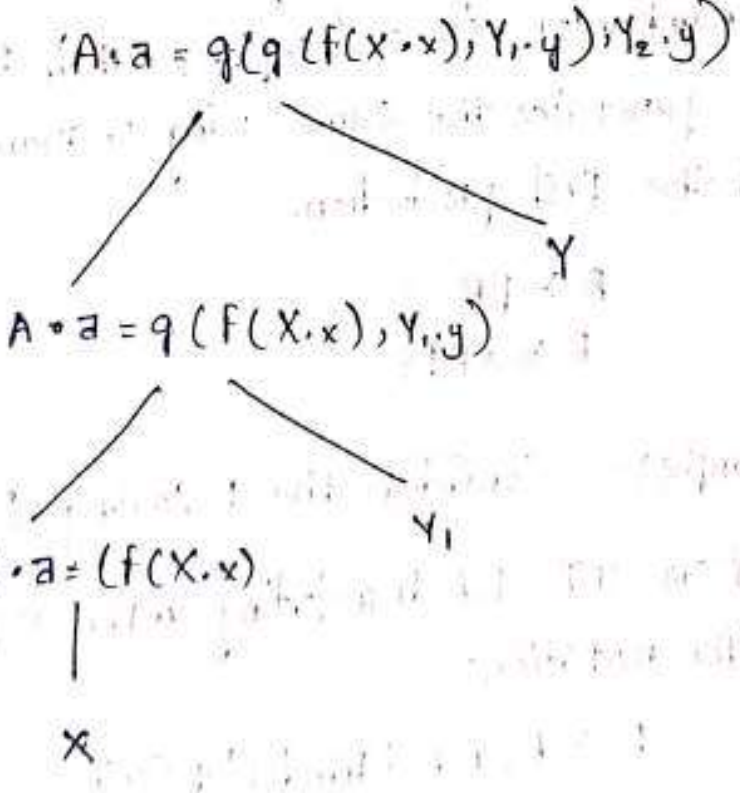
$E \rightarrow TR$

$R \rightarrow +T$ {Print ('+');} R

$R \rightarrow \varepsilon$.

$A \rightarrow XR$

$R \rightarrow YR \mid \varepsilon$



Implementing L-Attributed SDD's:

Translation during Recursive-Descent Parsing:

→ A recursive-descent parser has a function A for each nonterminal A. We can extend the parser into a translator as follows:

- The arguments of function A are the inherited attributes of nonterminal A.
- The return-value of function A is the collection of synthesized attributes of nonterminal A.

→ In the body of function A, we need to both parse and handle attributes:

1) Decide upon the production used to expand A.
2) Check that each terminal appears on the input when it is required.
3) Preserve, in local variables, the values of all attributes needed to compute inherited attributes for non terminals in the body or synthesized attributes for the head non-terminal.
4) Call functions corresponding to nonterminals in the body of the selected production, providing them with the proper arguments.

Example:-

Let us consider the SDD and SDT of example for while-statements. A pseudocode of the relevant parts of the function S appears in the below:

```
string S(lable next){
    string Scode, Ccode;
    lable L1, L2;
    if (current input == token while) {
        advance input;
        check '(' is next on the input,
                and advance;
        L1 = new();
        L2 = new();
        C.code = C(next, L2);
        check ')' is next on the input,
                and advance;
        S.code = S(L1)
        return("label ||L1|| C.code|| "label"
                ||L2|| Scode);
    }
    else
}
```

On-The-Fly Code Generation:-

The elements we need to make this technique work are:

1) For one or more nonterminals, a main attribute.

2) The main attributes are synthesized.

3) The rules that evaluate the main attribute(s) ensure that -

a) The main attribute is the concatentation of main attributes of nonterminals appearing in the body of the production, perhaps, with other elements that are not main attributes, such as the string label or the values of labels L_1 and L_2.

b) The main attributes of nonterminals appear in the rule in the same order as the nonterminals themselve appear in the production body.

→ The main attribute can be constructed by emitting the (nonterminals) non-main-attribute elements of the concatenation, we can rely on the recursive calls to the translation.

Example:-

We can modify the function of diagram to emit elements of the main translation S.code instead of saving them for concatenation into a return value of S.code. The revised function S appears in the following:

```
void S(label next){
    label L1, L2;
    if (current input == token while) {
        advance input;
        check '(' is next on the input, and
                advance;
        L1 = new();
        L2 = new();
        Print ("label", L1);
        C (next L2);
        check ')' is next on the input,
                and advance;
        print ("label", L2);
        S(L1);
        }
    else /* other statement types */
}
```

L-attributed SDD's and LL Parsing:

→ Suppose an L-attributed SDD is based on LL-grammar & that we have converted it to an SDT with actions embedded in the productions.

→ We can then perform the translation during LL parsing by extending the parser stack to hold actions & certain data items needed for attribute evaluation.

→ The data items are copies of attributes.

The two principles to manage attributes on the stack:

1) The inherited attributes of nonterminal A are placed in the stack record that represents the nonterminal.

2) The synthesized attributes for a nonterminal A are placed in a seperate synthesize-record that is immediately below the record for A on the stack.

→ We had seperated data using for different purposes into different records.

→ Action-records contain pointers to code to be executed.

The principle that makes all this copying of attributes work is:

- All copying takes place among the records that are created during one expansion of one nonterminal.

Bottom-up parsing of L-attributed SDD's:-

It has three parts:

1) start with the SDT construction, which places embeded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized attributes.

2) Introduce into the grammar a marker nonterminal in place of each embeded action. Each such space gets a marker that is on production for marker M. i.e; $M \to \varepsilon$.

3) Modify the action a if marker nonterminal M replaces it in some production $A \to \alpha \{a\} \beta$, and associates with $M \to \varepsilon$ an action a' that

a) copies, as inherited attributes [in the same way] of M, any attributes of A or symbols of α that action a needs.

b) Computes, attributes in the same way as a, but makes those attributes be synthesized attributes of M.

Variants of Syntax Trees:

Directed Acyclic Graphs for Expressions:

→ Syntax tree for an expression is like that of a DAG has leaves corresponding to atomic operands & interior nodes corresponding to operators.

→ Thus, DAG not only represents expressions more sufficiently, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

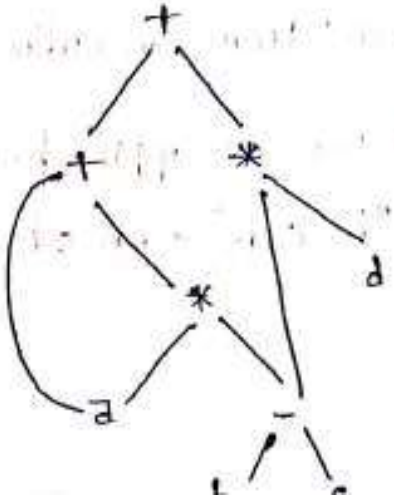
→ The difference is that a node N in a DAG has more than one parent if N represents a common subexpression.

→ In a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.

Example:-

The below shows the DAG for the expression:

a + a * (b - c) + (b - c) * d.



Value-Number Method for constructing DAGs:-

Algorithm:-

The value-number method for constructing the nodes of a DAG.

Input:

Label op, node 1, and node r

Output:

The value number of a node in the array with signature <op, 1, r>.

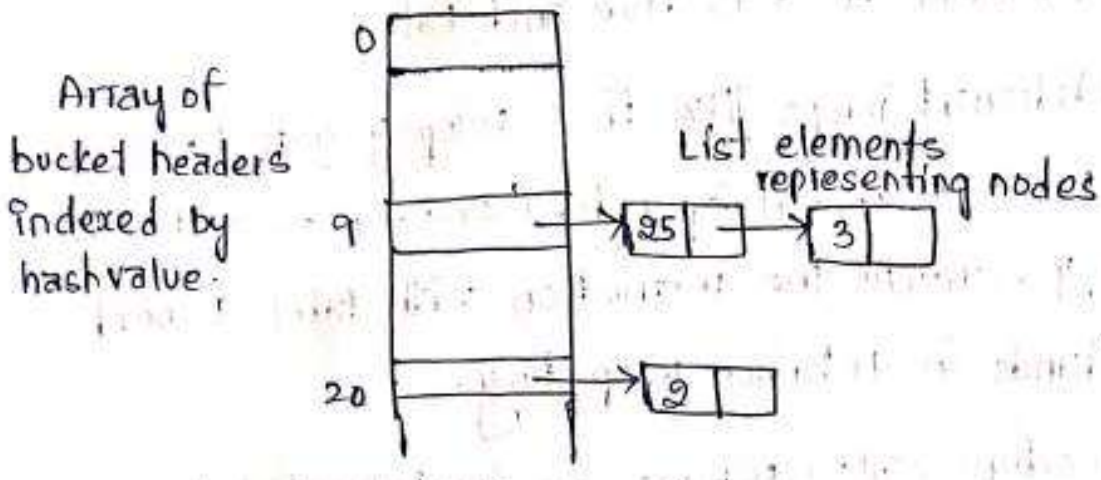
Method:

Search the array for a node M with label op, left child l and right child r. If there is such a node, return the value number of M. If not, create in the anay a new node N with label op, left child l, right child r, and return its value number.

With this algorithm yields the desired output, searching the entire array every time we are asked to locate one node is expensive, especailly if the anay holds expressions from an entire program.

To avoid this, we use an approach, i.e; hash table, nodes are kepl into "buckets" each of have a few nodes.

To construct a hash table, for the nodes of a DAG, we need a hash function h that computes the index of the bucket for a signature $\langle op, 1, r \rangle$ in a way that distributes the signatures across buckets



Three Address Code:-

The list of common three-address instruction form are as follows:

1. Assignment instructions of the form $x = y \ op \ z$, where op is a binary arithemetic or logical operation, x, y, z are addresses.

2. Assignment of the form $x = op \ y$, where op is an unary operation. Unary operation includes unary minus, logical nagation, conversion operators.

3. Copy instructions of the form $x = y$, where x is assigned the value of y.

4. An Unconditional jump goto L. The three address instruction with label L is the next to be executed.

5. Conditional jumps of the form if x goto L and if False x goto L. These instructions execute the instruction with Label L, next if x is true and false.

6. Conditional jumps like if x relop y goto L, where relop is a relational operator (<, ==, >=, etc) to x and y, execute the instruction with label L next if x stands in relation relop to y.

7. procedure calls & returns are implemented using the instructions: param x for parameters; call p, n and y = call p, n for procedure & function calls, returns y, where y represents a returned value is optional. Their typical use as the sequence of three-address instructions:

```
Param x1
Param x2
.....
Param xn
call P, n.
```

8. Indexed copy Instructions of the form x = y[i] and x[i] = y.

x = y[i] sets x to the value in the location i units memory beyond location y.

x[i] = y sets the contents of the location i units beyond x to the value of y.

9. Address & pointer assignments of the form:

x = &y => sets the r-value of x to be the location (l-value) of y.

x = *y => x is a pointer name or a temperary whose r-value is a location.

*x = y => r-value of the object pointed to by x to the r-value of y.

Example:-

Consider the statement:

do i = i + 1; while (a[i] < v);

There are two possible translations for the statement.

In (a): uses a symbolic label L, attached to the first instruction.

(b): shows position number for the instructions, starting at position 100.

```
L: t1 = i + 1
   i = t1
   t2 = i * 8
   if t3 < v goto L
```

(a) symbolic labels

```
100: t1 = i + 1
101: i = t1
102: t2 = i * 8
103: t3 = a[t2]
104: if t3 < v goto 100
```

(b) Position Numbers

Quardruples:

These instructions can be implemented as objects or as records with fields for the operator and operands.

→ A quadruples (or just "quad") has four fields, op, arg_1, arg_2, result.

where op is an internal code for the operator.

Example:

Three address code for the assignment

```
a = b * -c + b * -c;
```

The three-address code and quadruples are as follow:

t_1 = minus c

t_2 = b * t_1

t_3 = minus c

t_4 = b * t_3

t_5 = t_2 + t_4

a = t_5

(a) Three address code

	op	arg1	arg2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

Triples:

→ It has only three fields i.e; op, arg1 and arg2.

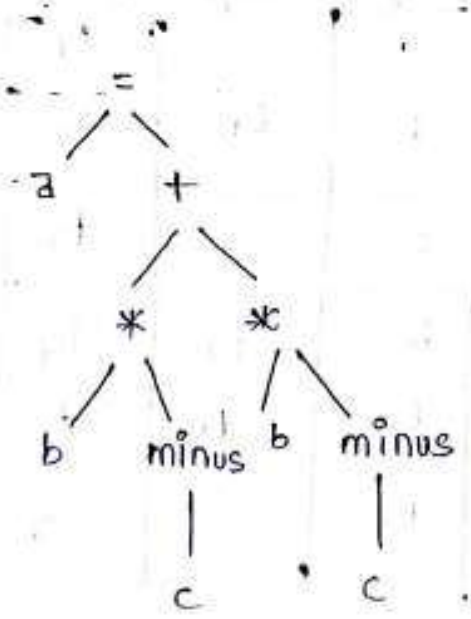
→ Using triples, we refer to the result of an operation x op y by its position.

→ A triple representation refer to position (or) parenthesized numbers represent pointers into the triple structure itself.

→ Triples are equivalent to signatures. Hence, the DAG & triple representations of expressions are equivalent.

→ Since syntax-tree variants and three-address code represents control flow quite differently.

Example: The syntax tree and triples representation of a = b * -c + b * c;



(a) Syntax tree

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	(a)	(4)

Static Single - Assignment:

→ SSA is an intermediate representation that facilitates certain code optimizations.

→ Two distinctive aspects distinguish SSA from three-address code.

→ The first is that all assignments in SSA are to variables with distinct names; hence the term SSA.

→ The below diagram shows the representation of TAC and in SSA.

Note: The subscripts distinguish each difenation of variables p and q in the SSA representation.

$p = a + b$
$q = p - c$
$p = q * d$
$p = e - p$
$q = p + q$

(a) TAC

$p_1 = a + b$
$q_1 = p_1 - c$
$p_2 = q_1 * d$
$p_3 = e - p_2$
$p_4 q_2 = p_3 + q_1$

(b) SSA

Intermediate Program in TAC and SSA.

Stack Allocation of Space:

Activation Trees:

stack Allocation would not be feasible if procedure calls or activation of procedures did not nest in time. The following example illustrates nesting of procedure calls.

Example:

The below shows the sketh of a program that reads nine integers into an anay a and sorts them using the recursive quicksort algorithm:

```
int a[11];
void read Anay() { /* Reads 9 integers into
                      a[1], ...., a[9] */
  int i;
  ....
}
int partition (int m, int n){
  Returns 'p';
  ....
}
void quicksort(int m , int n)
{
   int i;
   int (n>m)
   {
```

```
    i = partition (m,n);
    quicksort (m, i-1);
    quicksort (i+1, n);
    }
}
main ()
{
read Array ();
a[0] = -9999;
a[10] = 9999;
quicksort (1,9);
}
```

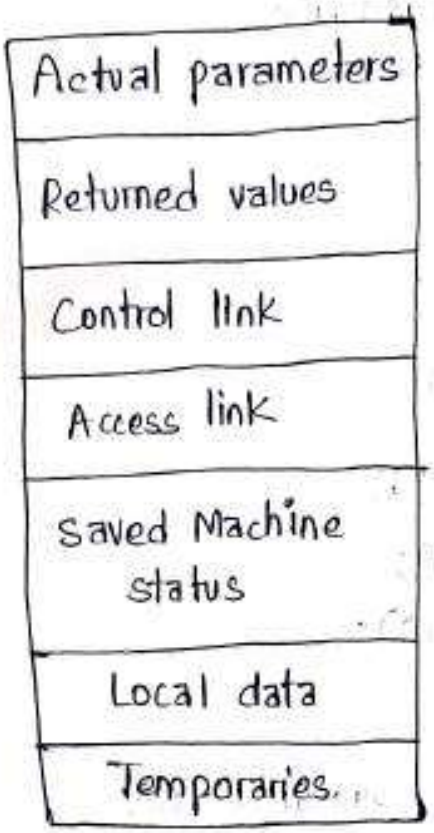
There are three common cases:

1) The activation of q terminates normally. Then, in any language, control resumes just after the point of P at which the call to q was made.

2) The activation of q, or some procedure q called, either directly or indirectly aborts. In that case, P ends simultaneously with q.

3) The activation of q terminates because of an exception that q cannot handle. Procedure P may handle the exception in which the activation of P continues, although not nessarcly from the point at which the call to q was made.

Activation Records:



1) Temperary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

2) Local data belonging to the procedure whose activation record this is.

3) A saved machine status, with information about the state of the machine before the call to procedure. This information includes the return addres and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.

4) An "access link" may be needed to locate data needed by the called procedure but found else where.

5) A control link, pointing to the activation record of the caller.

6) Space for the return value of the called function, if any. Again, not all called procedures return a value, and, if one does, we may prefer to place that value in a register for efficiency.

7) The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but registers for greater efficiency.

Calling Sequences in Activation records:

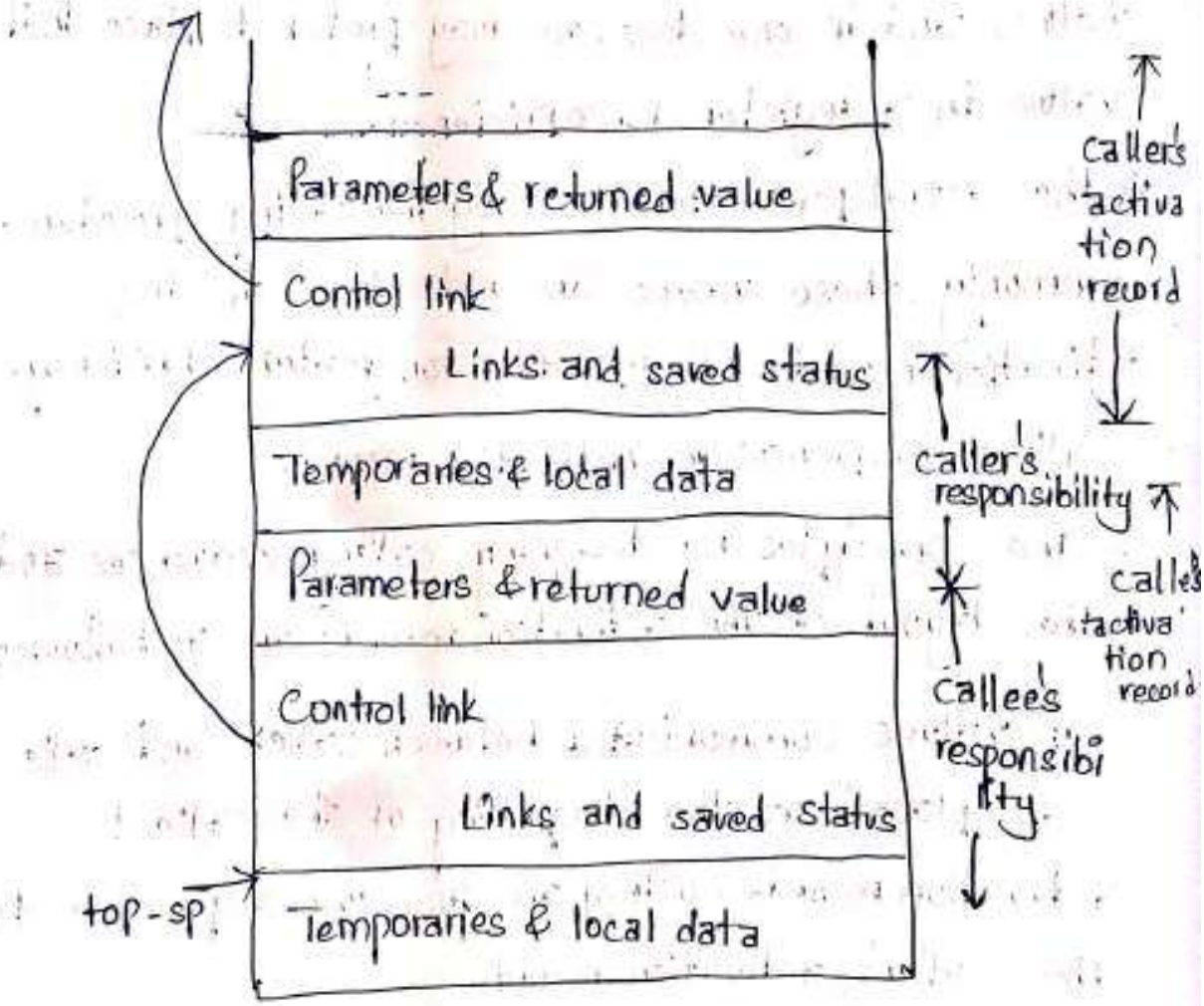
→ The principles for designing calling sequences and the layout of the activation records are as follows:

1) Values communicated between caller and calle are placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.

2) Fixed-length items are placed in the middle. If exactly the same components of the machine status are saved for each call, then the same code can do the saving & restoring for each.

3) Items whose size may not be known early enough are placed at the end of the activation record.

4) We must locate the top-of-stack pointer. A common approach is to have it point to the end of the fixed-length fields in the activation record.



Access to Nonlocal Data on the stack:

Nesting depth for procedures:

Let us give nesting depth 1 to procedures that are not nested within any other procedure.

Example:

All C functions are at nesting depth 1. If a procedure p is defined immediately within a procedure at nesting depth i, then give p the nesting depth i+1.

Example:

The below contains a sketch in ML of quicksort example

```
fun sort (inputFile, outputFile) =
    valled a = anay (0,10); let
      fun readAnay (inputFile) = a ... a... ;
      fun exchange (i,j) = ... a ... ;
      fun quicksort (m,n) =
        valled v = ... ;
        fun partition (y,z) =
          ... a ... v ... exchange ..
        in
          ... a ... v ... partition ... quicksort
        end
    in
```

```
...a...readArray ...quicksort...
end
```

→ In the code, sort are several functions:

readArray, exchange each access the array a. Function partition is defined and it is accessed the array a and the pivot value v, and also called the function exchange. Since partition is defined because to accesses both the array a and the pivot value v, and also calls the function exchange.

Access Links:-

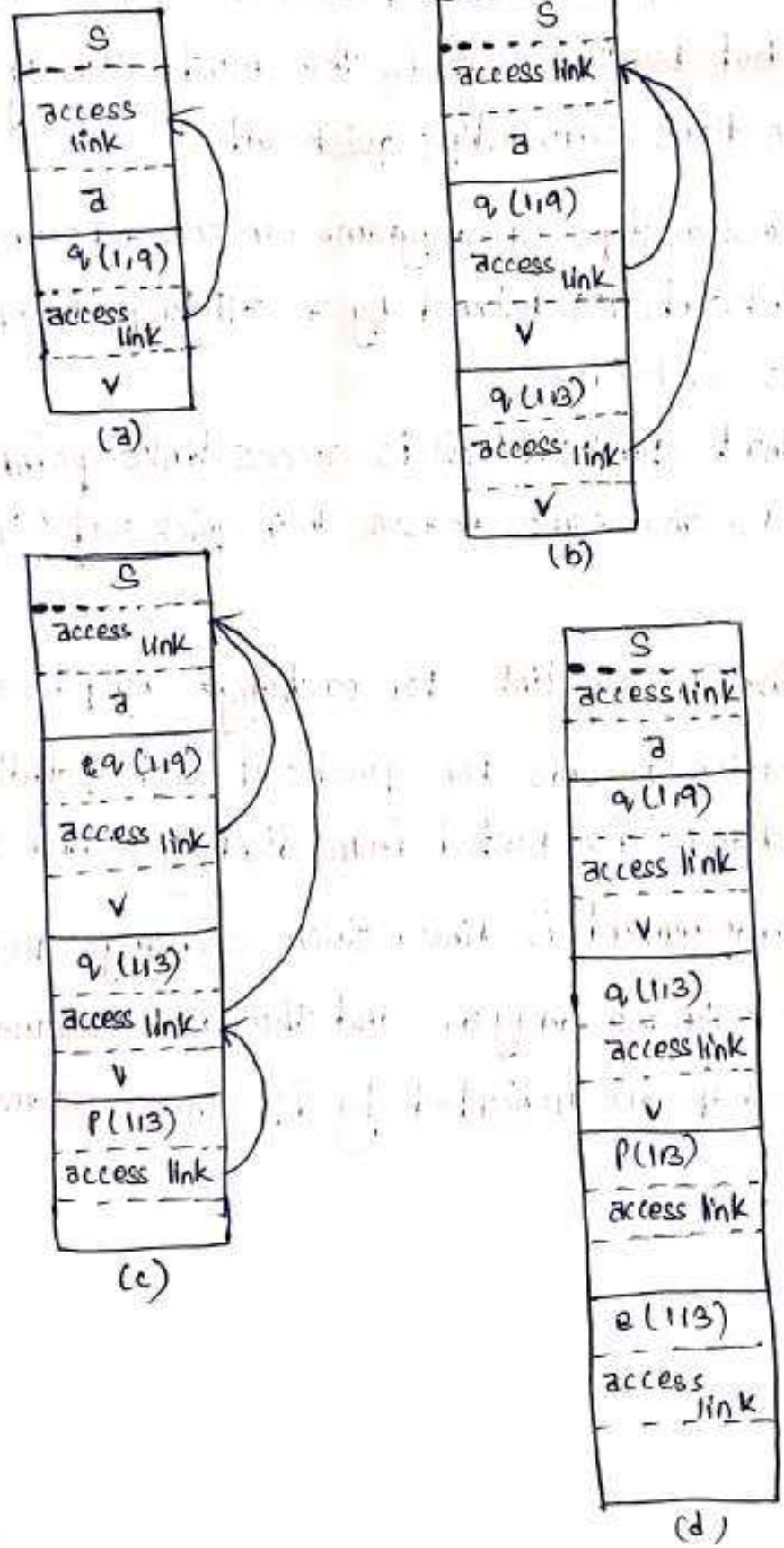
→ A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the access links to each activation record.

→ If the procedure p is nested immediately within procedure q in the source code, then the access link in any activation of p points to the most recent activation of q.

→ Nesting depth of q must be exactly one less than the nesting depth of q.

Example:

The below diagram shows a sequence of stacks that might result from execution of the function sort.



In (a): We see the situation after sort has called readArray to load input to the array a and then called quicksort(1,9) to sort the array.

→ The access link from quicksort(1,9) points to the activation record for sort, not because sort called quicksort, but because sort is the most closely nested functions surrounding quicksort.

→ In successive steps of diagram, we see a recursive call to quicksort(1,3) followed by a call to partition, which calls exchange.

→ notice that quicksort(1,3) is access links points to sort, for the same reason that quicksort(1,9)'s does.

In (d): The access link for exchange by passes the activation records for quicksort and partition, since exchange is nested immediately within sort.

→ That arrangement is fine, since exchange needs to access only the array a, and the two elements is must swap are indicated by its own parameters i and j.

Access links for procedure Parameters:-

When a procedure p is passed to another procedure q as a parameter, and q then calls its parameter, it is possible that q does not know the context in which p appears in the program.

→ It is impossible for q to know this so how to set the access link for p.

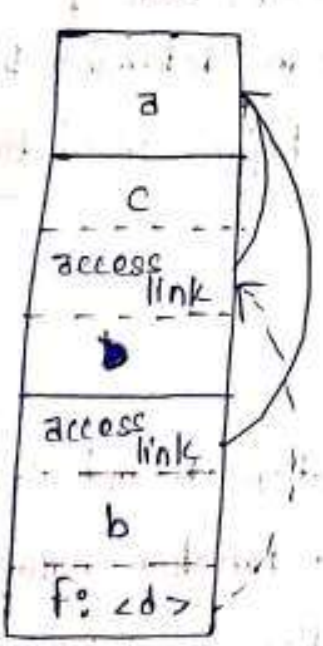
→ Hence when procedures are used as parameters, the caller needs to pass, along with the name of the procedure-parameter the proper access link for that parameter.

→ The caller always knows the link since if p is passed by procedure r as an parameter, then p must be a name accessible to r, and r can determine the access link for p exactly if p were beign called by r directly.

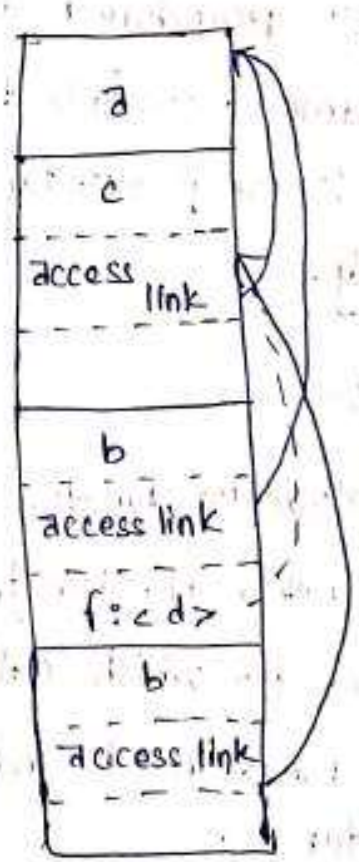
Example:-

The below ML function a has functions b and c nested within it. Function b has a function valued parameter f, which calls it. function c defines with a function d and c then calls b with actual parameter c.

```
fun a(x) =
  let
    fun b(f) =
      ...f...;
    fun c(y) =
      let
        fun d(z) = ...
        in ... b(d)...
        end;
      in... c(1)...
      end;
```



(a)



(b)

Heap Management:-

Memory Manager:

→ The subsystem that allocates and deallocates space within the heap.

→ It keeps track of all the free space in heap storage at all times.

→ Serves as an interface between application programs and the operating systems.

→ It has two functions:

1. Allocation:

→ It produces a chunk of contiguos heap memory of the requested size.

→ If no chunk of the needed size is available, it increases the heap storage by getting consewtive bytes of virtual memory from the OS.

2. Deallocation:

→ It returns deallocated space to the pool of free space (so it can reuse the space)

→ Typically, does not return memory to the OS. even if heap usage drops.

The properties of memory managers are:

1. Space Efficiency:

A memory manager should minimize the total heap space needed by a program. space efficiency is achieved by minimizing "fragmentation".

2. Program Efficiency:

A memory manager should make use of the memory subsystem to allow the programs to run faster. The time taken to execute an instruction can vary widely depending on where objects are placed in memory.

3. Low Overhead:

Because memory allocation and deallocation are operations in many programs. These operations are important to be as efficient as possible. The over head is minimized because the fraction of execution time spent performing allocation and deallocation.

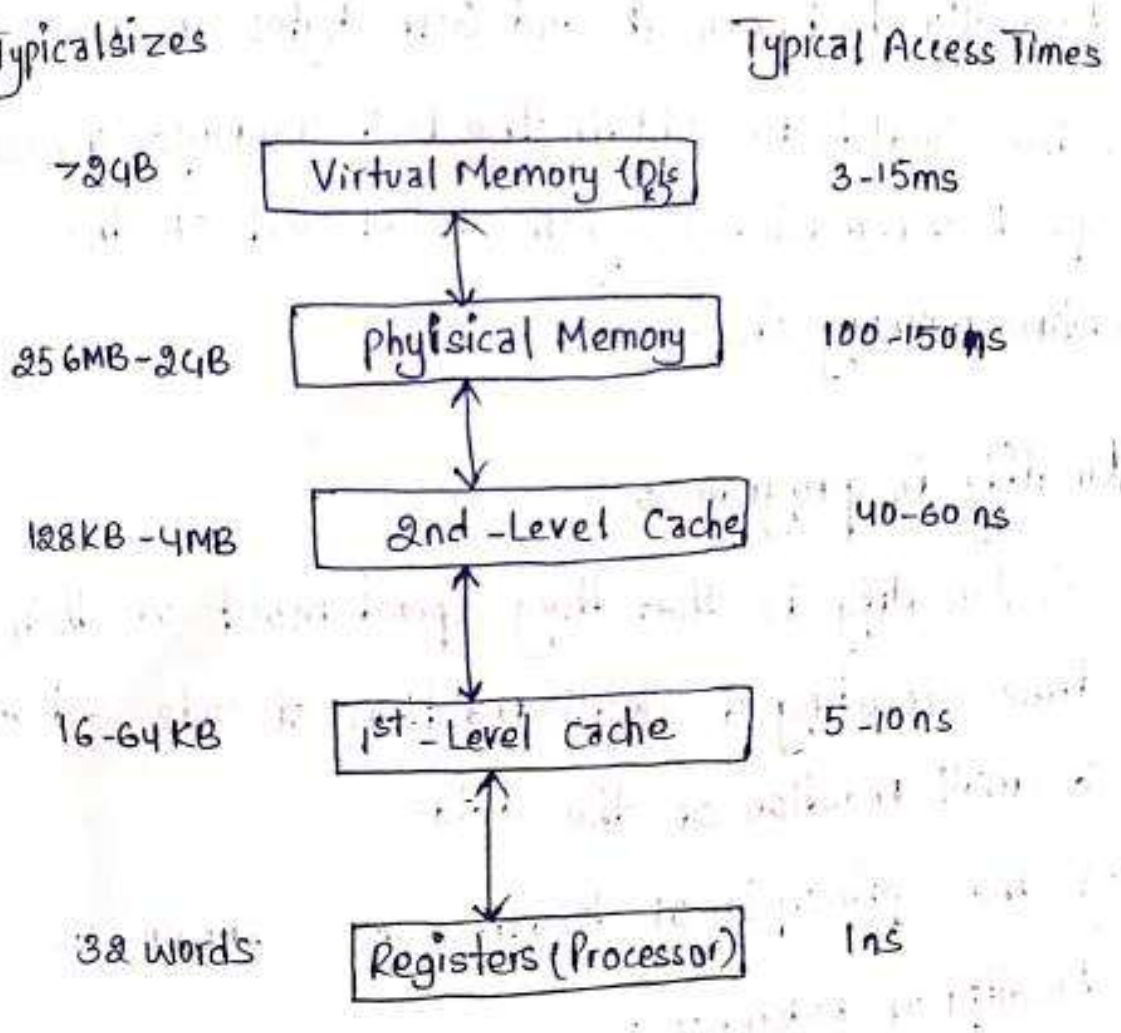
Note:-

Cost of allocation is dominated by small requests; the overhead of managing large objects is less important because it amortized over a large amount of computation.

Memory hierarchy of a Program:

→ All modern computers arrange their storage as a memory hierarchy.

→ The memory hierarchy follows, which consists of a series of storage elements, with the smaller faster ones, "closer" to the processor, and the larger slower ones further away.



→ Data transferred as blocks of contiguous storage.

→ To amortize the cost of access, larger blocks are used with the slower level of the hierarchy.

→ Between main memory and 'cache', data is transferred in blocks called as cache lines, which are typically from 32 to 256 bytes long.

→ Between virtual memory (disk) and main memory, data is transferred in blocks known as pages, typically between 4K and 64K bytes in size.

→ The Goal is to obtain the best average access speed while minimizing the total cost of the entire memory size.

Locality in programs:

→ Locality is that they spend most of their time executing a small fraction of code and only a small fraction of the data.

→ The principle of locality is known as "Locality of Reference".

→ The phenomenon of the same value or related storage locations being frequently accessed.

→ "90/10 Rule" comes from the observation as "A program spends 90% of its time in 10% of its code".

→ There are two types of access locality:

1. Temporal locality:

The memory locations the program accesses are likely to be accessed again within a short period of time.

Example:

Instructions in the body of the inner loops.

2. Spatial locality:

Memory locations close to the location accessed are likely also to be accessed within a short period of time.

Example:

Traversing the elements in a 1D array.

In this, programs spend 90% of their time executing 10% of the code because:

1) program often contain many instructions that are never executed.

2) only a small fraction of the code that could be invoked is actually executed in a typical run of the program.

3) The program spends most of its time executing innermost loops and tight recursive cycles in a program.

Reducing Fragmentation:

→ At the beginning execution, the heap is one contiguos unit of free space.

Main memory is divided into two parts:

1) Resident operating system, usually held in low memory with interrupt vector.

2) User processor then hold in high memory.

→ As the program allocates and deallocates memory this space is broken into free and used chunks of memory and free chunks need not reside in a contiguos area of the heap.

→ free chunks of memory as holes. Holes of various size are scattered throughout memory.

→ when a process arrives, it is allocated memory from a hole large enough to acuomodate it.

→ we reduce fragmentation by controlling how the memory manager places new objects in the heap.

→ It has been found that a good strategy for minimizing fragmentation for real-life programs is to allocate the requested memory in the smallest available hole that is large enough.