Automata Theory And Compiler Design:-

Automata:-

It is a "abstract machine (or) a model".

Automata Theory:-

It is a study of "abstract machine" (or) a "computing device".

History of Automata Theory:-

→ In 1930's, "Allen Turing" studied Abstract machine.

→ He studied this theory to know "what a computing device could do and what a computing device couldnot do".

→ In 1940's and 1950's, "Finite Automata" was studied by some scientists.

→ We use this study in "software" and haidware", "computing device".

→ In late 1950's, "chomsky" studied, Grammer, these Grammers are used the "development of software".

→ In 1960's, "S. cook", he extended study of "Allen Turing" and he was able to seperate Problems that can be solved by computer and that

can't be solved by computer.

What was the reason to study Automata Theory:

→ It was used to design and develop the some softwares.

→ We use this in lexical analyser software, digital circuits software
  ↳ Breaks the Programs into logical units
    ↳ key words
      Identifiers
      Punchuations
to verify some systems,
to web scan systems
we use the automata Theory.

Introduction To Finite Automata Theory:-

Finite Automata:-

The Automata that exists in finite no. of states.

→ The Purpose of the "state" is to "remember certain portion of system history".

Advantage:-

→ We can develop the device/system in fixed no. of states/resources.

How to represent Finite Automata:-

→ To represent finite Automata, we use

(circle) ○ → To represent states

(start) ⟶ ⇒ To represent start state;

It is label to the start state. It will be there in arrow.

(double circle) ◎ → To represent the final states.
↓ Accepting states.

(arc) ↷ → To represent transistion. [movement].

It is labelled with the some input.

How To model a software:-

→ The finite automaton for the part of lexical analyser that recognises the string "then".

→ It requires 5 states. It refers to the Prefixes of the "then". [t, th, the, then, ○ (one empty state)].

Structural Representation:-

Grammars

→ used in the development of the processing the data with a recursive structure.

Regular Expressions

→ used for the structure or organise the data.

Eg:- UNIX style

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

[A-Z].[A-Z]-

[A-Z] → any capital letter

[a-z] → any small letter

* → any no.of occurances

[ ] → a blank space

Eg:- Delhi IN

The given regular expression represent a capitalized word followed by blank space followed by 2 capital letters

IN is a single name followed by two letters.

For multiword city Names

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

West Bengal IN [a-z]*

[A-Z] [A-Z]

Automata & Complexity:-

It discusses limitations of computations.

→ what a computer can do at all?

This study is called decidability and problems that can be solved with sometimes are called decidable.

→ what a computer can do efficiently?

This study is called intractability and the problems that can be solved with no more time called "tractable (or) undecidable".

Central Concepts of Automata Theory:-

Alphabet: It is a finite non empty set of symbols.

It is denoted by "Σ".

Eg:- $\Sigma = \{0,1\}$ -- Binary Alphabet.

$\Sigma = \{a, \ldots, z\}$ .. Small letter Alphabet.

ASCII : character [A-65-ASCII-values]

String:- It is a finite sequence of input symbols chosen from an alphabet sigma.

Eg:- 0111 is a string. choose from a binary alphabet. $\Sigma = \{0,1\}$

It is represented by ω.

chosen from an alphabet Σ.

1) Empty string:- It is a string that contains zero occurances of input symbols.

→ It is represented as ϵ (Epsila).

→ It can be a string that can be taken from any alphabet.

2) Length of the string:-

→ It can be no. of positions of symbols in the string. → It is represented as "|w|"

Eg:- $w = 1010111$

The length of the string $|w| = 7$

3) Powers of the string:-

→ If Σ is an alphabet then Σ^k is a set of strings of length k.

Eg:- $\Sigma = \{a, b\}$

$\Sigma^0 = \phi$, $\Sigma^1 = \{a, b\}$, $\Sigma^2 = \{aa, ab, ba, bb\}$

$\Sigma^3 = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$

→ The set of all strings over some alphabet Σ is denoted by "Σ^*".

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cdots$$

→ The set of all non empty strings is denoted by "Σ^+".

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cdots\cdots$

The relation blw the both strings is "$\Sigma^* = \Sigma^+ \cup \epsilon$".

4) Concatenation of strings:-

If x and y are two strings then the concatenation of x,y is represented by a "xy" where the string "x followed by y".

Eg:- $x = 10001$, $y = 00110$. } $\Sigma = \{0,1\}$

$xy = 1000100110$

→ $\epsilon x = x\epsilon = x$.

↳ empty string [identity of concatenation].

Language:- It is a set of strings choosen from the Σ^*.

Where Σ is some alphabet.

Eg:- $\Sigma = \{0,1\}$.

L = set of strings with equal no.of 0s and equal no.of 1's.

$L = \{01, 10, 0011, 1100, 1001, 0110, 0101, 1010 \ldots\}$

→ It is represented by "L"

→ If Σ is an alphabet and L is subset of Σ^*, then L is a language.

→ If L is a language over Σ then it is also a language over
superset of Σ.

Eg:- Set of binary integers whose value is prime

L = {10, 11, 101, 111, ...}

→ Σ^* is also a language over the alphabet Σ.

→ ϕ is empty language $\phi \neq \{\epsilon\}$

↳ empty ↳ language with empty string ε

↳ empty language

Problem:-

It is a question of deciding whether a string belongs to some particular language or not.

Eg:- L = {0, 100, 10, 000, 100, 010, 110, ...}

determine 000 is part of Given string or not (Yes)

→ It determine the membership of the string in a language.

→ If Σ is an alphabet and L is a language the question determineing whether the string 'w' is there in language L or not is a "problem."

→ There are two types of finite Automata:

1) Deterministic Finite Automata:-

→ Deterministic refers that for each input symbol there is only one state to which finite automata can transition from its current state.

→ DFA is represented by "5 tuple" notation. It consits of : $A = \{Q, \Sigma, \delta, q_0, F\}$

→ It consists of i) A set of states. It is represented by "Q".

ii) A set of input symbols, represented by "Σ".

iii) A transition function $\delta(q_0, 1)$ which takes two arguments "state and "input symbol" and Give one output.

iv) starting state, represented as "q_0"

v) set of final states are (or) Accepting states, represented as "F".

Notations of DFA:-

→ There are two types of Notations:

1) Transition Diagram:-

It is a Graph that contains

i) For each state, there is a node

(ii) for each input symbol and for each state transition function δ is represented by an arc labelled with an input symbol.

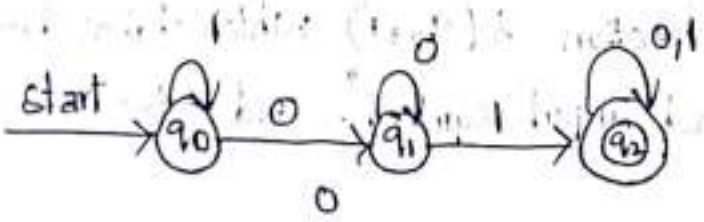
(iii) There is an arrow into the starting state labelled as start.

(iv) The final state or states must be in double circle ◎ and other states must be in single circle ○.

Question:- Construct DFA which accepts all strings of "0's and 1's" that have the sequence "01"

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

$L = \{01, 001, 0101, 011, 101, \ldots\}$



2) Transition Table:-

→ Transition Table is a tabular representation of function δ.

→ Rows corresponds to states.

→ Columns corresponds to input symbols.

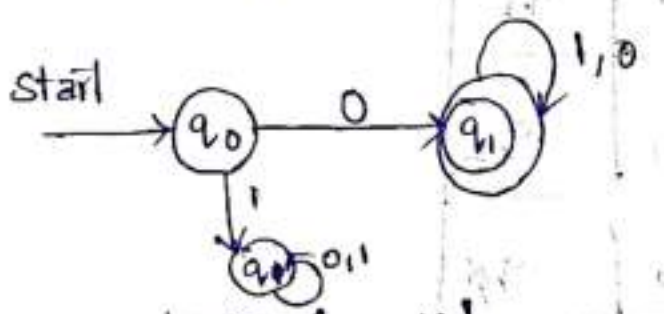
→ Entry corresponds to state which results from $\delta(q, a)$.

	0	1
→ q_0	q_1	q_0
q_1	q_1	q_2
* q_2	q_2	q_2

→ construct DFA with accepts all strings over alphabet $\Sigma = \{0,1\}$ starts with "0".

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

$L = \{0, 01, 00, 000, 001, 010, 011, \ldots\}$



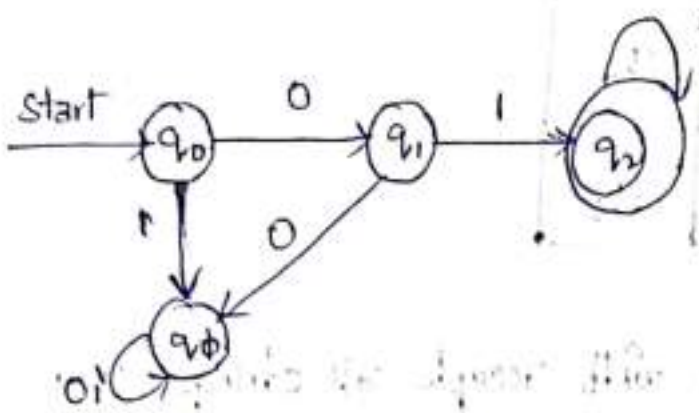
	0	1
→ q_0	q_0	q_1
* q_1	q_1	q_1

	0	1
→ q_0	q_1	q_ϕ
q_ϕ	q_ϕ	q_ϕ
* q_1	q_1	q_1

→ Construct DFA with accepts all string over alphabet $\Sigma = \{0,1\}$ starts with "01".

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

$L = \{01, 101, 1001, 0101, 011, 0100, 0101, \ldots\}$

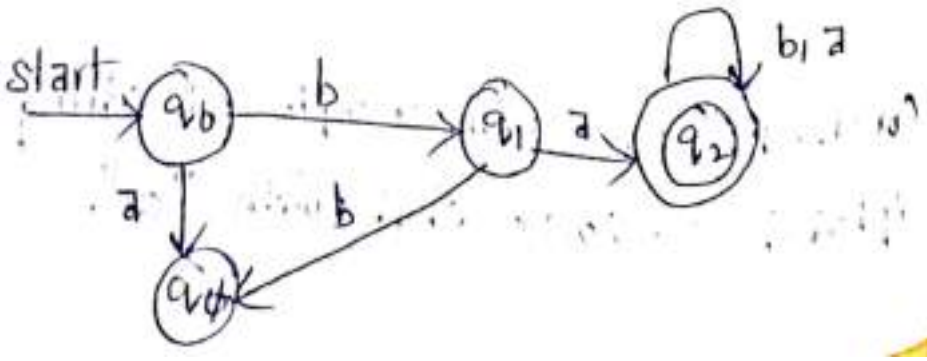


	0	1
$\rightarrow q_0$	q_1	q_ϕ
q_1	q_ϕ	q_2
$*q_2$	q_2	q_2
q_ϕ	q_ϕ	q_ϕ

→ Construct DFA with accepts all string over alphabet $\Sigma = \{a, b\}$ starts with "ba".

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

$L = \{ba, bab, babb, baba, \ldots\}$

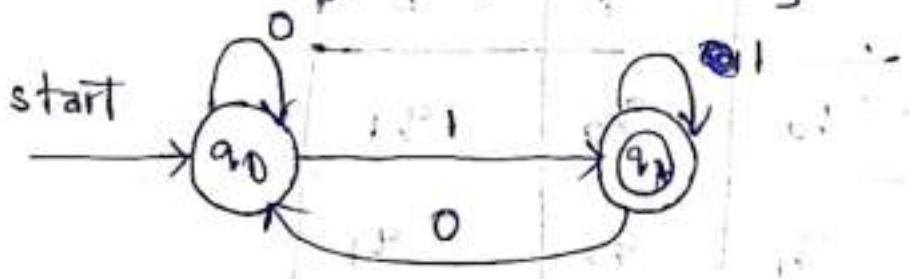


	a	b
$\rightarrow q_0$	q_ϕ	q_1
q_1	q_2	q_ϕ
$*q_2$	q_2	q_2
q_ϕ	q_ϕ	q_ϕ

→ Construct DFA for all strings over alphabet $\Sigma = \{0,1\}$ which ends with "1".

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

$L = \{01, 11, 001, 011, 101, 111 \ldots\}$

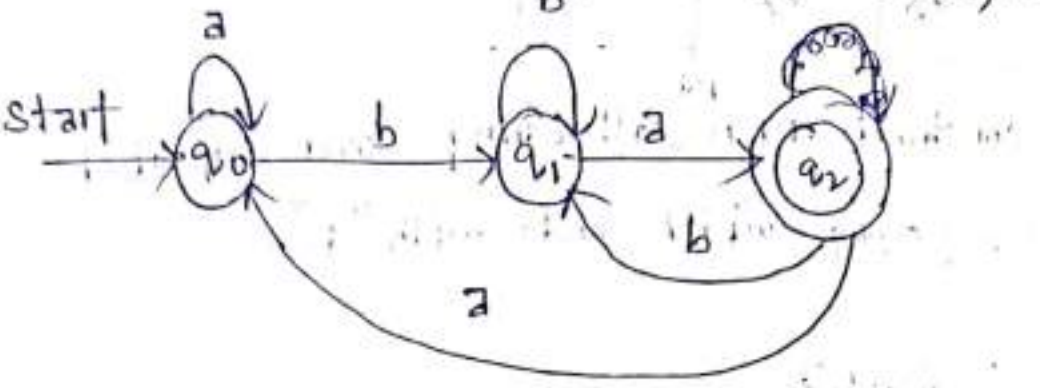


	0	1
$\rightarrow q_0$	q_0	q_1
$*q_1$	q_0	q_1

→ Construct DFA for all strings over the alphabet $\Sigma = \{a, b\}$ which ends with "ba"

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

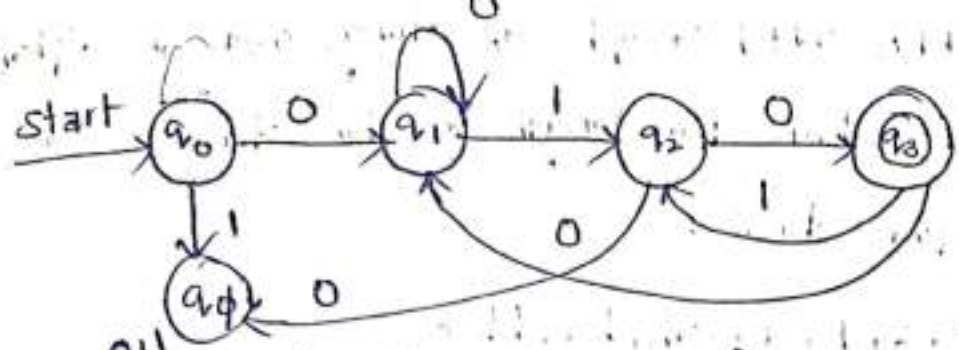
$L = \{ba, aba, bba, aaba, abba, baba, bbba, \ldots\}$



	a	b
→ q_0	q_0	q_1
q_1	q_2	q_1
* q_2	q_0	q_1

→ Construct DFA for all strings over the alphabet $\Sigma = \{0, 1\}$ which ends with 010

$L = \{010, 0010, 1010, \ldots\}$



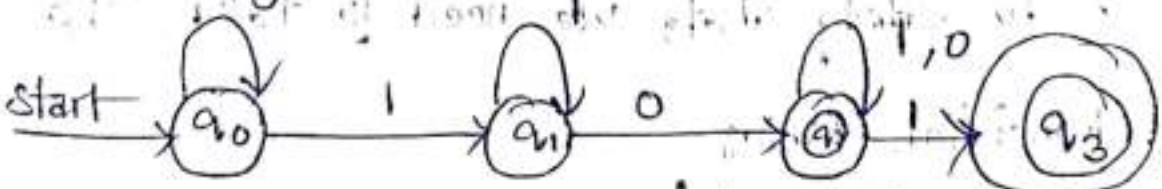
011

	0	1
→ q_0	q_1	q_ϕ
⊖ q_1	q_1	q_2
q_2	q_3	q_ϕ
* q_3	q_1	q_2
q_ϕ	q_ϕ	q_ϕ

→ construct DFA for all strings over the alphabet "011" which contains 10ϕ

010, 101 1101

L = { 10, 0101, 1010, 1011, 0010, ...



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

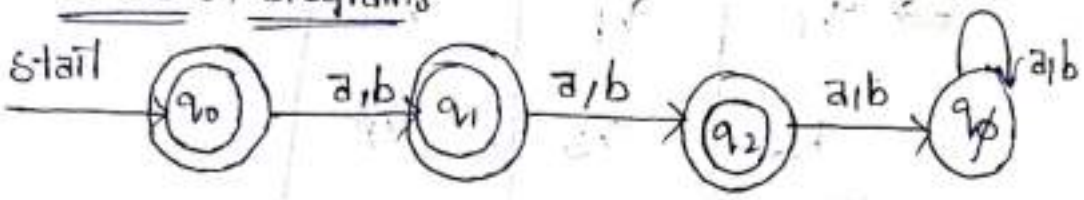
→ Construct DFA which accepts all strings over an alphabet $\Sigma = \{a, b\}$ where the length of the string is less than or equals to "2":

$L = \{\epsilon, a, b, aa, ab, ba, bb\}$

→ It depends on the length of strings not min "LOS".

→ The length of the string = 2 => 3 states

Transistion Diagram:



→ This DFA satisfies all the strings of the length of the string is "≤ 2".

→ There is no transistion of q_2, if we take self transistion then we get more than 2 states.

→ But there is no numbers matching with the more than 2 in the language.

→ To satisfy that, we need to take "q_ϕ".

Transistion Table:

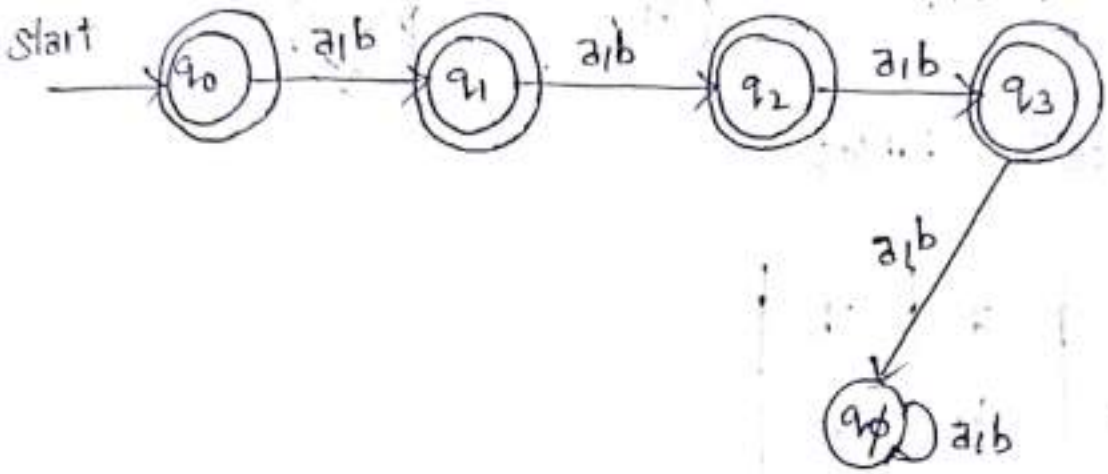
	a	b
→* q_0	q_1	q_2
* q_1	q_2	q_2
* q_2	q_ϕ	q_ϕ
q_ϕ	q_ϕ	q_ϕ

→ construct DFA which accepts all strings over an alphabet $\Sigma = \{a, b\}$ where the length of the string is less than or equals to "3".

$L = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\}$

→ The length of the string is 3 ⇒ 4 states.

Transition Diagram:-



Transition Table:-

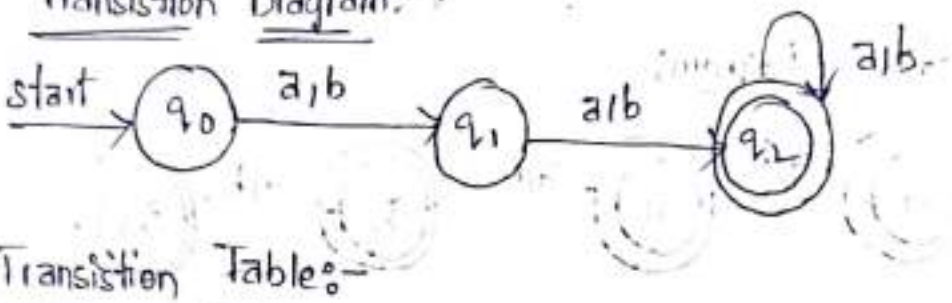
	a	b
→ * q_0	q_1	q_1
* q_1	q_2	q_2
* q_2	q_3	q_3
* q_3	q_ϕ	q_ϕ
q_ϕ	q_ϕ	q_ϕ

Ex:- → Construct DFA which accepts all the strings over the alphabet $\Sigma = \{a, b\}$ where the length of the string is "greater than or equal to 2"

$L = \{aa, ab, ba, bb, aaa, aab, aba, abb, \ldots\}$

→ The length of the string is 2 = 3 states

Transition Diagram:-



Transition Table:-

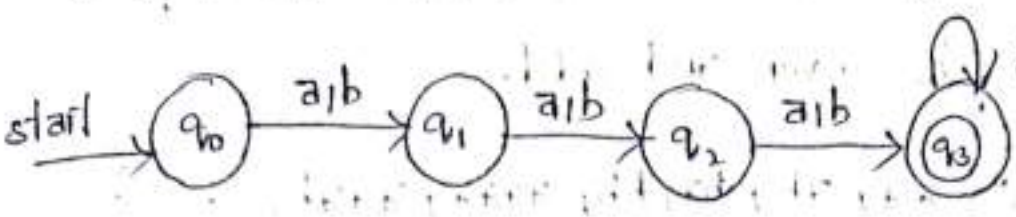
	a	b
→ q_0	q_1	q_1
q_1	q_2	q_2
* q_2	q_2	q_2

→ Construct DFA which accepts all the strings over the alphabet $\Sigma = \{a, b\}$ where the length of the string is "Greater than" or equal to "3"

$L = \{aaa, aab, aba, abb, aaaa, aaab, \ldots\}$

The length of the string is 3 = 4 states.

Transition Diagram:-



Transition Table:-

	a	b
→q_0	q_1	q_1
q_1	q_2	q_2
q_2	q_3	q_3
q_3	q_3	q_4

→ construct DFA which accepts all the strings over the alphabet $\Sigma = \{a, b\}$ where the "length" of the string is "2".

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

→ The length of the string is 2 = 3 states

Transition Diagram:-

Transition Table:-

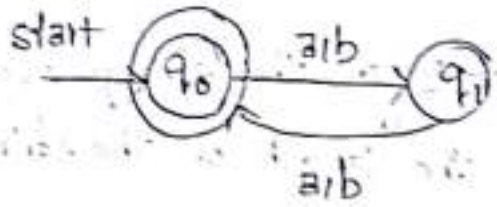
	a	b
→q_0	q_1	q_1
q_1	q_2	q_2
*q_2	q_ϕ	q_ϕ
q_ϕ	q_ϕ	q_ϕ

→ Construct the DFA which accepts all the strings over an alphabet $\Sigma = \{a, b\}$ where the length of the string is even and odd.

$L = \{\epsilon, aa, ab, ba, bb, aaaa, aaab, \dots\}$

→ length of the string is = even and odd ⇒ Takes only 2 states.

Transistion Diagram:- For even



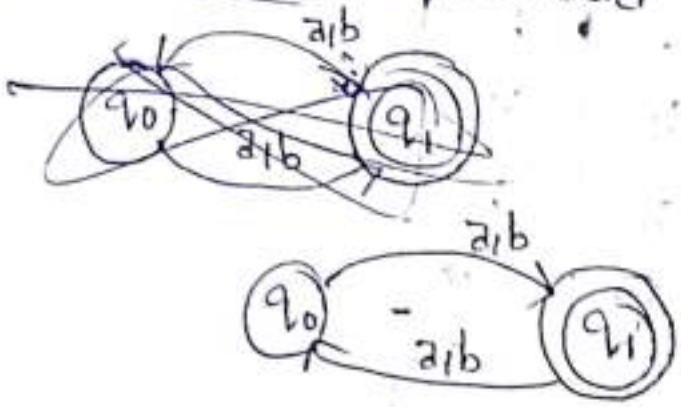
Even numbers [q_0 is final state]

Odd numbers [q_1 is final state]

Transistion Table:-

	a	b
→* q_0	q_1	q_1
q_1	q_0	q_0

Transistion Diagram:, For odd



Transistion Table:-

	a	b
→q_0	q_1	q_1
*q_1	q_0	q_0

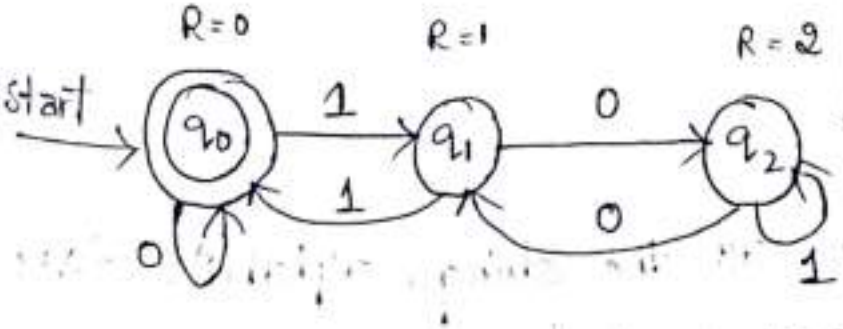
→ Construct DFA which accepts all the strings over an alphabet $\Sigma = \{0,1\}$ where the binary integers are divisible by 3.

↳ The lengths of the

$L = \{0, 11, 110, 1001, 1100, 1111, \ldots\}$

→ The length of string is depends on the "remainder of the number".

→ The length of the string is "3".



Transistion Table:-

	0	1
→* q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_2

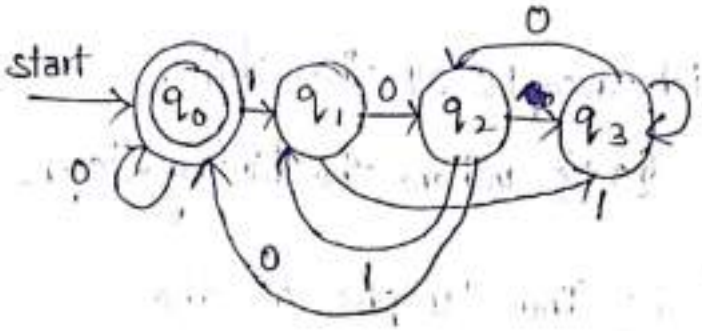
Decimal number	Binary equivalent	Rem ain der
0	0	0
1	1	1
2	10	2
3	11	0
4	100	1
5	101	2

→ Construct DFA which accepts all the strings over alphabet $\Sigma = \{0,1\}$ where binary integers are divisible by "4".

$$L = \{0, 100, 1000, 1100, 1110, \ldots\}$$

(0, 4, 8, 12, 16)

remainder $R = 0,1,2,3$



Transition Table:

	0	1
→* q_0	q_0	q_1
q_1	q_2	q_3
q_2	q_0	q_1
q_3	q_2	q_3

- Decimal binary remainder

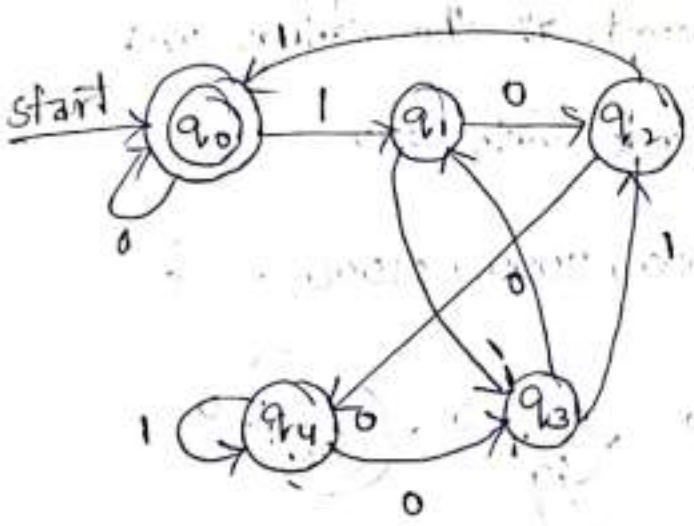
0	0	0
1	1	1
2	10	2
3	11	3
4	100	0
5	101	1
6	110	2
7	111	3

→ Construct DFA all the strings alphabet $\Sigma = \{0,1\}$ where are divisible by "5".

$$L = \{0, 101, 1010, 1111, \ldots\}$$

(0, 5, 10, 15)

$$R = 0,1,2,3,4$$



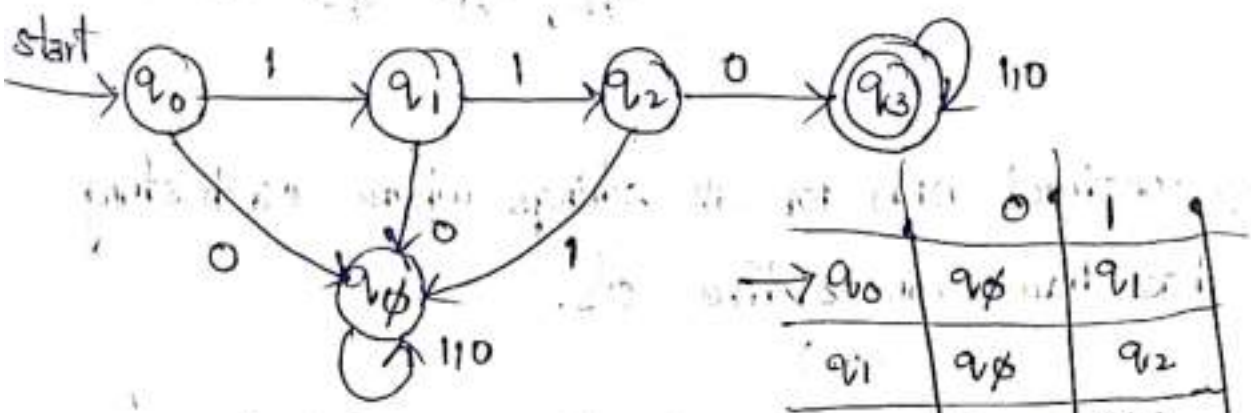
Decimal	binary	remainder
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	0
6	110	1
7	111	2
8	1000	3
9	1001	4

Transistion Table:-

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

→ construct DFA which accepts all the strings over an alphabet $\Sigma = \{0,1\}$ starts with 110.

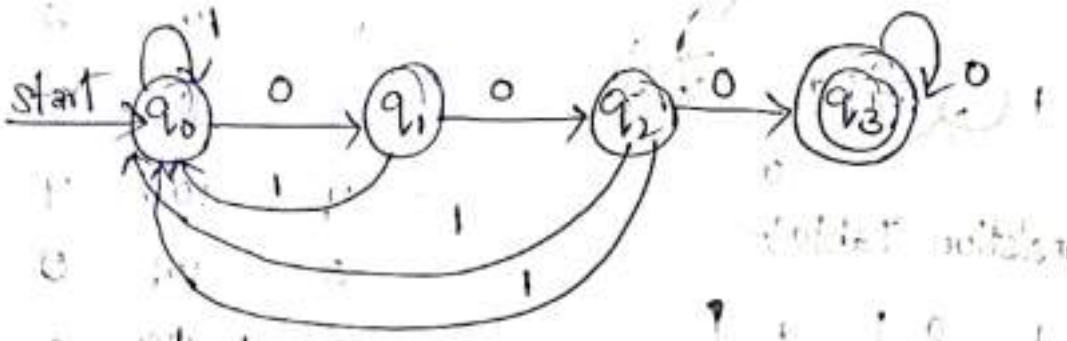
$L = \{ 110, 1101, 1100, 11000, 11001, 11010, \cdots \}$



	0	1
→ q_0	q_ϕ	q_1
q_1	q_ϕ	q_2
q_2	q_3	q_ϕ
* q_3	q_3	q_3
q_ϕ	q_ϕ	q_ϕ

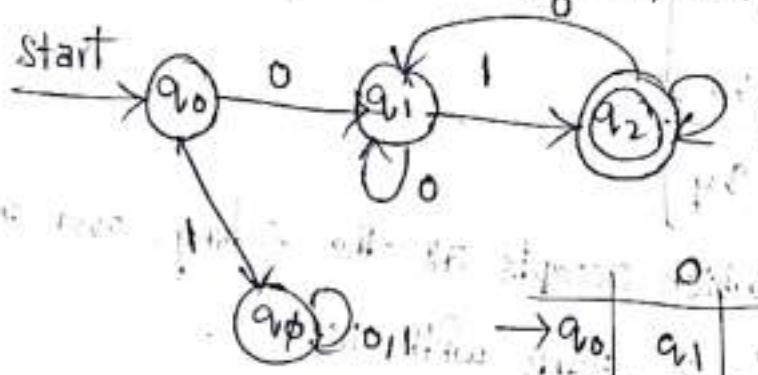
→ construct DFA which accepts all the strings over an alphabet $\Sigma = \{0,1\}$ ends with 000

$L = \{000, 1000, 0000, 11000, 01000, \ldots\}$



→ construct DFA $\Sigma = \{0,1\}$ starts with 0 and ends with 1.

$L = \{01, 001, 011, 0001, 0101, 0111, 0001, \ldots\}$



	0	1
→q_0	q_1	q_ϕ
q_1	q_1	q_2
q_2	q_1	q_2

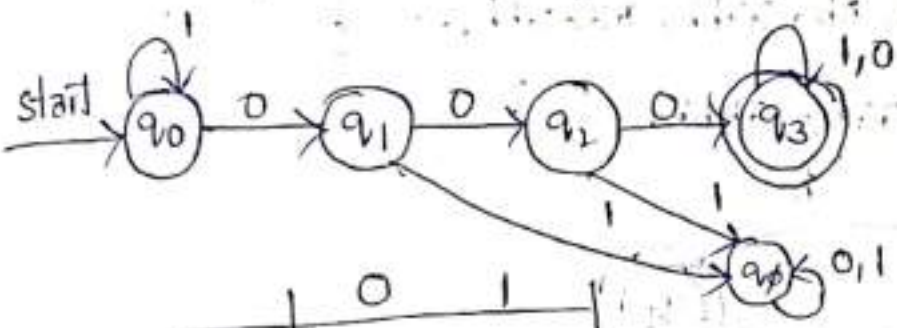
→ Construct DFA for all strings where each string has three consecutive 0's.

(or)

Construct DFA for all strings which have/ contains "000".

$L = \{ 000, 1000, 0001, 0010, 00010, 10001, \ldots \}$

Length of the string = 3 then n+1 states = 4

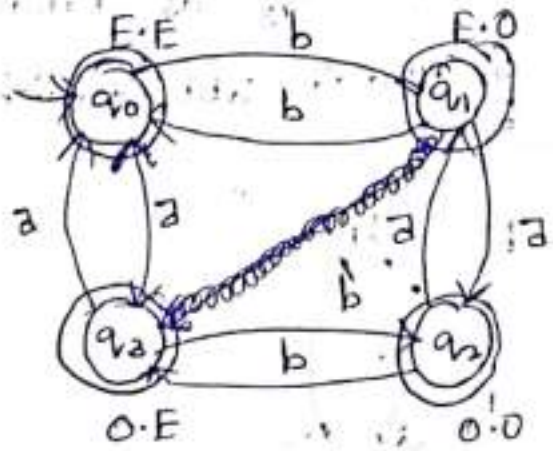


	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_2	q_ϕ
q_2	q_3	q_ϕ
$*q_3$	q_3	q_3

→ Construct DFA's for:

1) Even number of a's and even number of b's.
2) Even number of b's and odd number of a's.
3) Even number of a's and even number of b's.
4) odd number of a's and odd number of b's.

1) Even number of a's and odd number of b's.



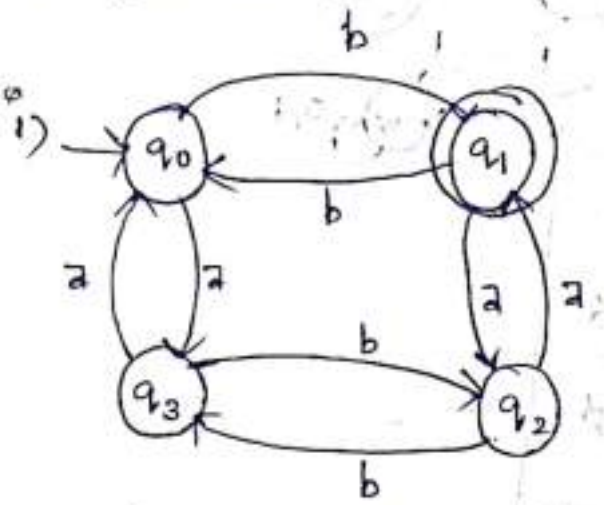
i) $L = \{b, bbb, abb, aabbb, \ldots\}$

ii) $L = \{a, aaa, abb, aaabb, \ldots\}$

iii) $L = \{\epsilon, aa, bb, aabb, aaaabb, \ldots\}$

iv) $L = \{ab, aaab, abbb, \ldots\}$

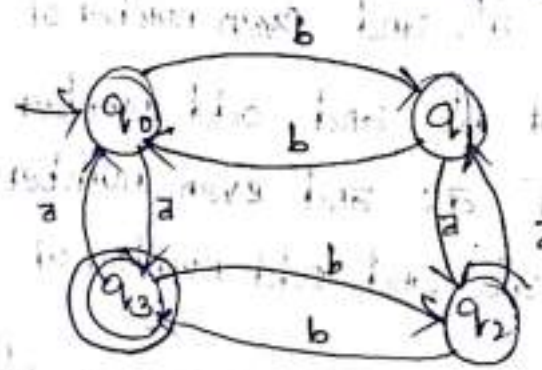
i)



here we take "q_1" as final state.

$L = \{b, bbb, aab, aabbb, \ldots\}$

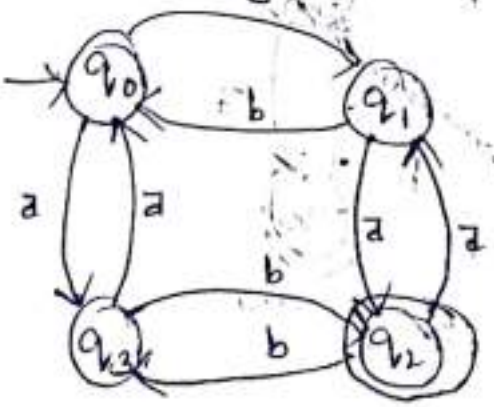
ii)



here we take "q_3" as final state

$L = \{a, aaa, abb, aaabb, \ldots\}$

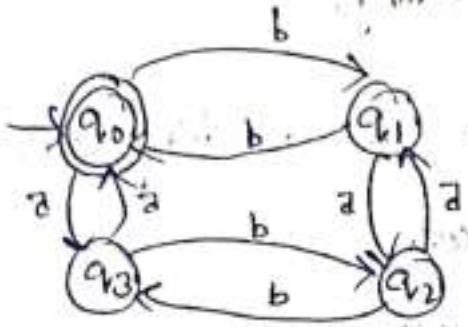
iii)



here we take "q_2" as final state

$L = \{\epsilon, aa, bb, aabb, aaaabb, \ldots\}$

iv)



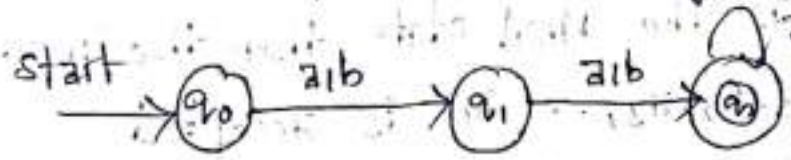
Here we take "q_0" as final state

$L = \{ab, aaab, abbb, \ldots\}$

→ Design a DFA that $L(M) = \{x / x$ is a string of 0s and 1s and $|x| \geq 2\}$

$L = \{aa, ab, ba, bb, aaa, aab, aba, abb \ldots\}$

→ The length of the string is 2 = 3 states



	a	b
→ q_0	q_1	q_1
q_1	q_2	q_2
q_2	q_2	q_2

How DFA process a string:

Let $a_1, a_2, a_3, \ldots, a_n$ is a string (or) some input sequences.

→ Start DFA from initial state (q_0)

→ Consult translation from $\delta(q_0, q_1)$ to find state to which transistion takes place from q_0 on input symbol q_1.

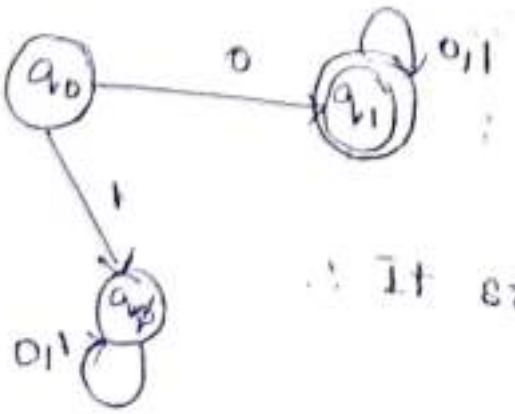
→ for the next input symbol evaluate $\delta(q_1, q_2)$ to find next state (q_2).

→ continue this process to find $q_3, q_4, \ldots q_n$

→ If q_n is the final state then the input sequence $a_1, a_2, a_3, \ldots a_n$ is accepted otherwise it is rejected.

Example:-

0010010



∴ It satisfies the data.

Extended Transistion Function:-

An extended transistion function describes, what happens when we start from any state and follow some input sequence.

→ Extended Transistion function is denoted by $\hat{\delta}$.

→ Extended Transistion function we give state and input sequence.

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

Where $w = xa$.

Language of DFA:-

The language of DFA is denoted by $L(A)$.

→ It is defined as

$$L(A) = \{ w / \hat{\delta}(q_0, w) \text{ is in } F \}.$$

→ If L is a language of DFA ($L(A)$) for some DFA "A" then L is called a "regular language".

Non-Deterministic finite Automata:- (NFA)

→ In which the finite automata can exists in any number of states on input from a state.

→ NFA is also a "5 tuple" notation. It consists of set of $(Q, \Sigma, \delta, q_0, F)$.

→ Q: Set of states

Σ: Set of input symbols

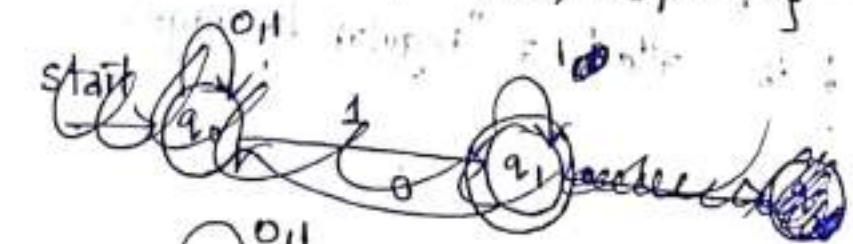
δ: A transition function $\delta(q_0, 1)$, which takes two arguments "state and input symbol" and give one output.

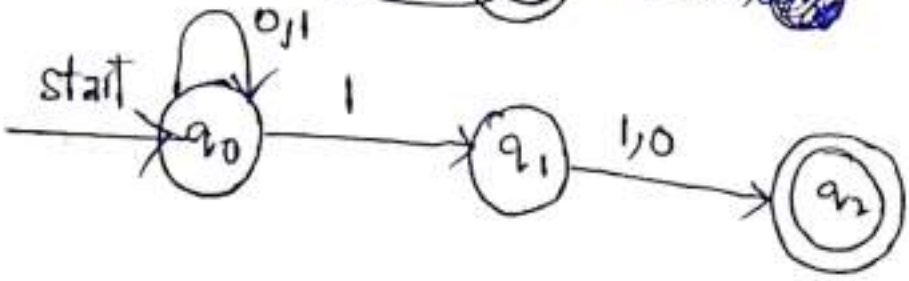
q_0: starting state.

F: set of final states (or) Accepting states.

→ Construct NFA which accepts all strings whose second last symbol is "1".

$L = \{10, 11, 010, 011, 111, 110, \ldots\}$





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

Extended Transition Function:-

The Language of NFA:-

It is also denoted by $L(A)$ and it is defined as $L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \phi\}$

Extended Transition function:-

$$\hat{\delta}(q_0, w) = \{r_1, r_2, \ldots r_n\}$$

Where, $r_1, r_2, \ldots, r_n$ are final states.

Finite Automata with epsilon Transitions:-

→ It is represented as "NFA-ϵ" (or) "ϵ-NFA".

→ It is a "5 tuple" notation. where

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

Q = set of states

$\Sigma = \Sigma \cup \{\epsilon\}$

δ = A transition function. $\delta(q_0, 1)$, which takes two arguments "state and input symbol" and give one output.

q_0 = starting state

F = set of Final states/Accepting states.

Usec of Epsilan:-

1) Each epsilan is along the path is "Invisible", i.e, it contributes nothing to string along the path."

→ Construct epsilan NFA which accepts decimal numbers consisting of i) an optional "+ or −"

ii) A string of digits

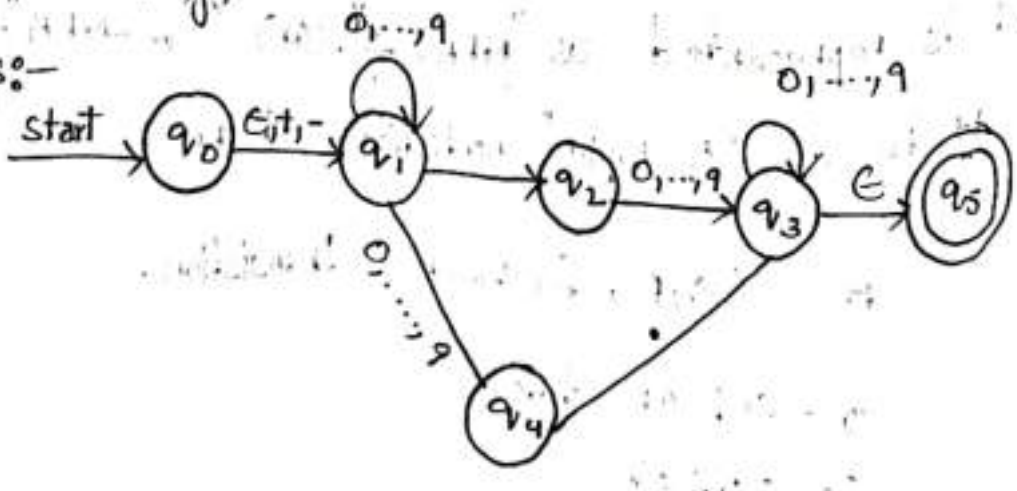
iii) A decimal point

iv) Another string of digits.

v) Either these string of digits can be empty but

vi) Atleast one of string of digits must be non empty.

Ans:-



Extended Transition Function:-

$$\hat{\delta}(q_0, w) = \bigcup_{j=1}^{m} (\epsilon\text{-closure}(r_j))$$

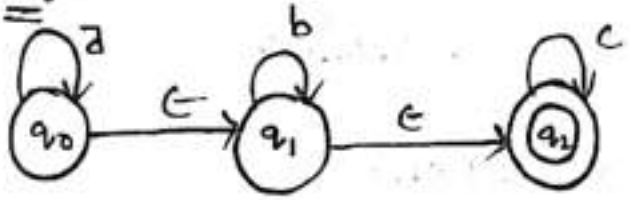
$$= \epsilon\text{-closure}(r_1) \cup \epsilon(\text{closure}(r_2) \cup \ldots$$

$$\cup \epsilon\text{-closure}(r_m)$$

Epsilan Closure:-

ϵ-closure of a state (q) consists of every state that can be reached from q along a path with "ϵ".

Example:-



ϵ

ϵ – closure (q_0) :- There is an "invisible transition to the "itself. " $\{q_0, q_1, q_2\}$ "

ϵ-closure (q_1) : " $\{q_1, q_2\}$ "

ϵ-closure (q_2) : " $\{q_2\}$ "

Eliminating ϵ-Transistions:-

Conversion of "NFA-ϵ" (or) "ϵ NFA" to "NFA without Transistions of ϵ":-

→ Find ϵ-closures for all states of the given ϵ-NFA.

→ Find Transistions(δ) for each state / input symbol.

→ ~~Construct~~ $\hat{\delta}(q, a) = \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q), a))$

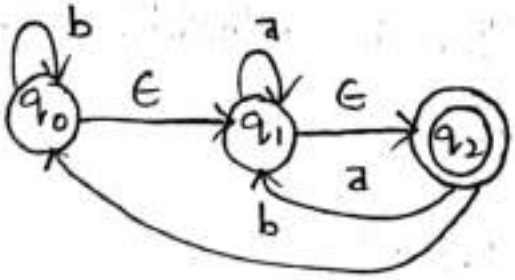
→ construct NFA without epsilan Transistions based on the computations of Transistion function.

→ Decide Final state as (F') are

i) $F' = F \cup \{q_0\}$ if ϵ-closure (q_0) is having F as a member.

ii) $F' = F$, otherwise

→



Step 1:- Find ϵ-closures for all states

$\epsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$

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

$\epsilon\text{-closure}(q_2) = \{q_2\}$

Step 2:- find the Transistions for each state on input symbol.

$\Rightarrow \hat{\delta}(q_1, a) = \epsilon\left(\text{closure}\left(\delta\left(\epsilon\text{-closure}(q_0), a\right)\right)\right)$

$= \epsilon\left(\text{closure}\left(\delta((q_0, q_1, q_2), a)\right)\right)$

$= \epsilon\left(\text{closure}\left(\delta((q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a)\right)\right)$

$= \epsilon\left(\text{closure}(\phi \cup q_1 \cup q_1)\right)$

$= \epsilon\text{-}(\text{closure}(q_1))$

$= \{q_1, q_2\}$

Step 3:-

$\Rightarrow \hat{\delta}(q_1, b) = \epsilon\left(\text{closure}\left(\delta\left(\epsilon\text{-closure}(q_0), b\right)\right)\right)$

$= \epsilon\left(\text{closure}\left(\delta((q_0, q_1, q_2), b)\right)\right)$

$= \epsilon\left(\text{closure}\left(\delta((q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b)\right)\right)$

$= \epsilon\left(\text{closure}(q_0 \cup \phi \cup q_0)\right)$

$= \epsilon\left(\text{closure}(q_0)\right)$

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

$\Rightarrow \hat{\delta}(q_1, a) = \epsilon\left(\text{closure}\left(\delta\left(\epsilon\text{-closure}(q_1), a\right)\right)\right)$

$= \epsilon\left(\text{closure}\left(\delta((q_1, q_2), a)\right)\right)$

$= \epsilon\left(\text{closure}\left(\delta(q_1, a) \cup \delta(q_2, a)\right)\right)$

$= \epsilon\ (\text{closure}(q_1 \cup q_1))$

$= \epsilon\ (\text{closure}(q_1))$

$= \{q_1, q_2\}$

$\Rightarrow \hat{\delta}(q_1, b) = \epsilon\ (\text{closure}(\delta(\epsilon\text{-closure}(q_1), b))$

$= \epsilon\ (\text{closure}(\delta(q_1, q_2), b))$

$= \epsilon\ (\text{closure}(\delta(q_1, b) \cup \delta(q_2, b)))$

$= \epsilon\ (\text{closure}(\phi \cup q_0))$

$= \epsilon\ (\text{closure}(q_0))$

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

$\Rightarrow \hat{\delta}(q_2, a) = \epsilon\ (\text{closure}(\delta(\epsilon\text{-closure}(q_2), a))$

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

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

$= \epsilon\ (\text{closure}(q_1))$

$= \{q_1, q_2\}$

$\Rightarrow \hat{\delta}(q_2, b) = \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_2), b))$

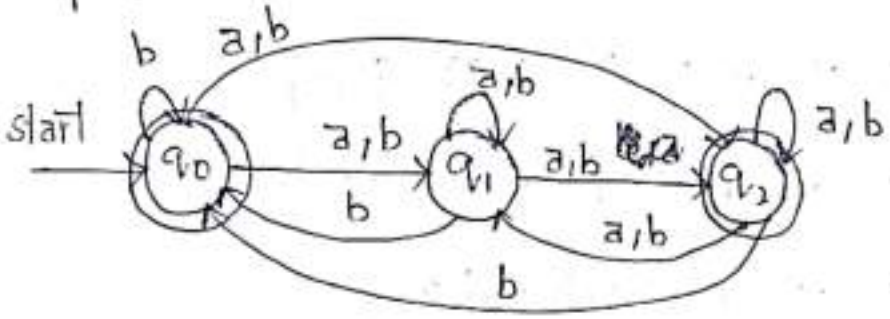
$= \epsilon(\text{closure}(\delta(q_2, b))$

$= \epsilon(\text{closure}(\delta(q_2, b))$

$= \epsilon(\text{closure}(q_0))$

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

Step 3:-



Step 4:-

$F' = F \cup \{q_0\}$ if ϵ-closure(q_0) is having F as a member.

Otherwise $F' = F$.

$F = \{q_2\}$

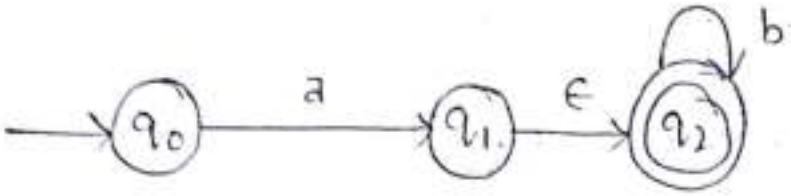
ϵ-closure$(q_0) = \{q_0, q_1, q_2\}$

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

$= q_2 \cup \{q_0\}$

$= \{q_2, q_0\}$.

→ convert following ε-NFA - ε to the NFA without ε.



Ans:-

Step 1:- Find ε closures for all states

$$\epsilon\text{-closure}(q_0) = \{\phi\} \{q_0\}$$

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

$$\epsilon\text{-closure}(q_2) = \{q_2\}$$

Step 2:- Find the Transitions for each state on input symbol.

$$\Rightarrow \hat{\delta}(q_0, a) = \epsilon\text{-(closure}(\delta(\epsilon\text{-closure}(q_0), a))$$

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

$$= \epsilon\text{-(closure}(q_1))$$

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

$$\Rightarrow \hat{\delta}(q_0, b) = \epsilon\text{(closure}(\delta(\epsilon\text{-closure}(q_0), b)$$

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

$$= \epsilon\text{-closure}(\phi)$$

$$= \phi$$

$\Rightarrow \hat{\delta}(q_1, a) = \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_1), a))$

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

$= \phi$

$\Rightarrow \hat{\delta}(q_1, a) = \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_1), b))$

$= \epsilon\text{-}(\text{closure}(\delta(\epsilon\text{-closure}(q_1, b))))$

$= \epsilon\text{-}(\text{closure}(q_2))$

$= q_2 \notin q_2$

$\Rightarrow \hat{\delta}(q_2, a) = \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_2), a))$

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

$= \phi \notin \phi$

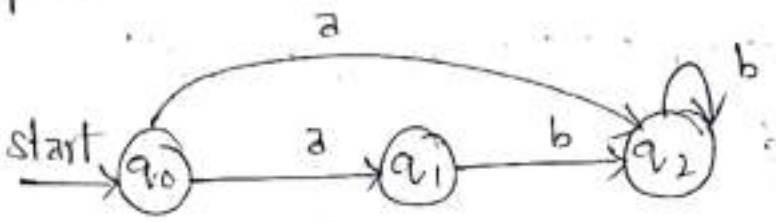
$\Rightarrow \hat{\delta}(q_2, b) = \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_2), b))$

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

$= \epsilon(\text{closure}(q_2))$

$= q_2 \in q_2$

Step 3:-



Step 4:-

$F' = F \cup \{q_0\}$ if ϵ-closure (q_0) is having F as a member

Otherwise $F' = F$

$F = \{q_2\}$ ϵ-closure $(q_0) = q_0$

Since F is not a member of ϵ-closure (q_0) so, $F' = F$

$F' = \{q_2\}$.

Conversion of NFA to DFA:-

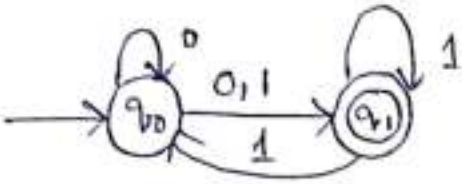
Step 1: start with initial state and find its transitions.

Step 2: If there are any new states find transitions for each of them.

Step3:- Continue step 2 until there are no new transitions.

Step4:- Mark all the states which has the final state of NFA as final state of DFA.

1) convert the following NFA into DFA / Find the equivalent DFA for the given NFA.



Step 1:- Find the transitions which starts with initial state.

	0	1
→ q_0	$[q_0, q_1]$	q_1
$[q_0, q_1]$	$[q_0, q_1]$	q_1
q_1	ϕ	$[q_0, q_1]$

Step 2:- Find the transitions for the new states occured

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

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

$$\delta([q_0, q_1], 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$$

$= [q_0 q_1] \cup \phi$

$= \{q_0, q_1\}$

$\delta([q_0 q_1], 1) = \delta(q_0, 1) = \delta(q_0, 1) \cup \delta(q_1, 1)$

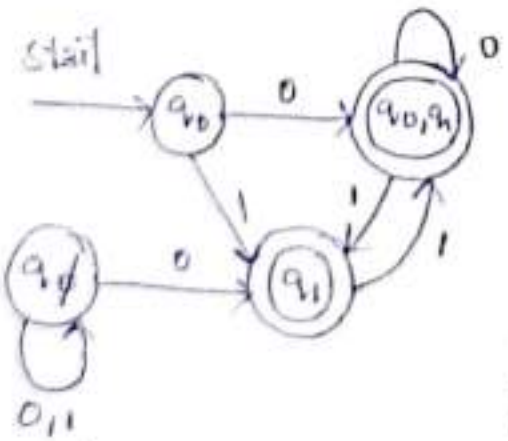
$= q_1 \cup q_1$

$= q_1$

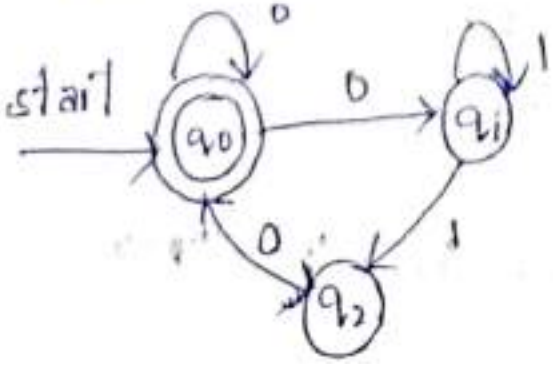
$\delta(q_1, 0) = \phi$

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

Step 3:- Continue the previous step if there are no new states, after that we need draw the transition diagram from the table.



2) Determine the equivalent DFA to the given NFA.



Step 1:-

→ q_0 ~~[q_0,q_1]~~

	0	1
→ q_0	[q_0 q_1 q_2]	ϕ
[q_0,q_1,q_2]	[q_0,q_1]	[q_1,q_2]
[q_1,q_2]	q_0	[q_1,q_2]
q_ϕ	q_ϕ	q_ϕ

Step 2:-

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

$\delta(q_0,1) = \phi$

$\delta([q_0,q_1],0) = \delta(q_0,0) \cup \delta(q_1,0)$

$= q_0 \cup \phi$ ~~q_1~~ $[q_0 q_1] \cup \phi$

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

$\delta([q_0,q_1],1) = \delta(q_0,1) \cup \delta(q_1,1)$

$= \phi \cup [q_1,q_2]$

$= [q_1,q_2]$

$\delta(q_1, q_2) =$

$\delta(q_1, 0) = \phi$

$\delta(q_1, 1) = q_1, q_2$

$\delta(q_2, 0) = q_0$

$\delta(q_2, 1) = \phi$

$\delta((q_1, q_2), 0) = \delta(q_1, 0) \cup \delta(q_2, 0)$

$= \phi \cup q_0$

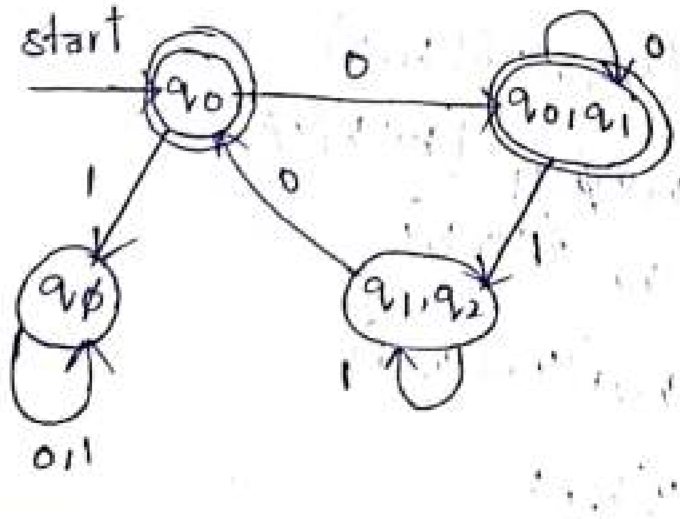
$= q_0$

$\delta((q_1, q_2), 1) = \delta(q_1, 1) \cup \delta(q_2, 1)$

$= (q_1, q_2) \cup \phi$

$= q_1, q_2$

step 3:-



Conversion of ε-NFA to DFA:-

Step1: Take epsilon-closure for the starting state of NFA as the starting state of DFA

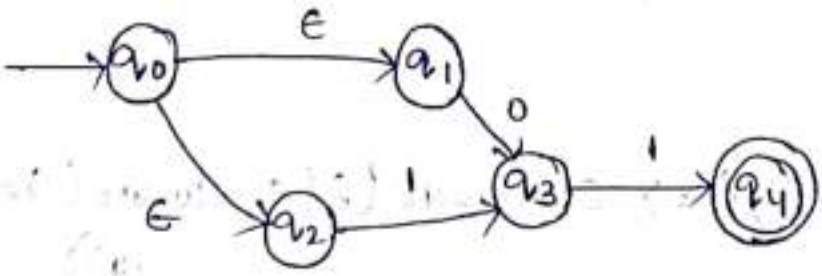
Step2: Find the states for each input from the current state.

Step3: If we find a new state, take it as a current state repeat step2.

Step4: & repeat step 2 and step 3 until there is no new state.

Step5: Mark the states of DFA as final states which contains the final state of NFA.

1) Convert the given ε-NFA into DFA



Step1: finding the ε-closures for the starting state of NFA as the starting state of DFA

ϵ - closure $(q_0) = \{q_0, q_1, q_2\}$

	0	1
$\rightarrow[q_0, q_1, q_2]$	$\{q_3\}$	$\{q_3\}$
q_3	ϕ	q_4
q_4	ϕ	ϕ
q_ϕ	ϕ	ϕ

$\hat{\delta}([q_0, q_1, q_2], 0) \Rightarrow \epsilon$ - closure $(\delta(\epsilon\text{-closure}(q_0, q_1, q_2), 0))$

$\Rightarrow \epsilon$ - closure $(\delta(\epsilon$ - closure $q_0 q_1 q_2), 0))$

$\Rightarrow \epsilon$ - closure $(\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0))$

$\Rightarrow \epsilon$ - closure $(\phi \cup q_3 \cup \phi)$

$\Rightarrow \epsilon$ - closure (q_3)

$\Rightarrow q_3$

$\hat{\delta}([q_0, q_1, q_2], 1) \Rightarrow \epsilon$ - closure $(\delta(\epsilon$ - closure $(q_0, q_1, q_2, 1))$

$\Rightarrow \epsilon$ - closure $(\delta(q_0 q_1 q_2), 1))$

$\Rightarrow \epsilon$ - closure $(\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1))$

$\Rightarrow \epsilon$ - closure $(\phi \cup \phi \cup q_3)$

$\Rightarrow \epsilon$ - closure (q_3)

$\Rightarrow q_3$

$\hat{\delta}([q_3],0) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_3),0))$

$\Rightarrow \epsilon\text{-closure}(\delta(q_3),0))$

$\Rightarrow \epsilon\text{-closure}\delta(q_3,0)$

$\Rightarrow \phi$

$\hat{\delta}([q_3],1) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_3),1))$

$\Rightarrow \epsilon\text{-closure}(\delta(q_3),1))$

$\Rightarrow \epsilon\text{-closure}\delta(q_3,1)$

$\Rightarrow q_4$

$\hat{\delta}([q_4],0) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_4),0))$

$\Rightarrow \epsilon\text{-closure}(\delta(q_4),0))$

$\Rightarrow \epsilon\text{-closure}(\delta(q_4,0))$

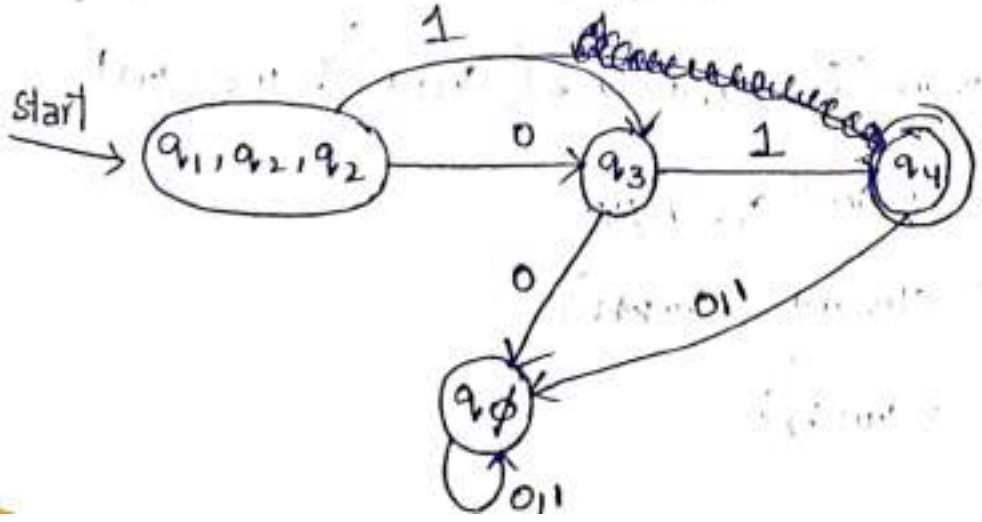
$\Rightarrow \phi$

$\hat{\delta}[q_4,1] \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_4),1)$

$\Rightarrow \epsilon\text{-closure}(\delta(q_4),1))$

$\Rightarrow \epsilon\text{-closure}(\delta(q_4,1))$

$\Rightarrow \phi$



→ Determine the equivalent DFA for the given ∈-NFA.

i) ∈ - closure $(q_0) = \{q_0, q_1\}$

ii) ∈ - closure $(q_1) = \{q_1, q_2\}$

iii) ∈ - closure $(q_2) = \{q_2\}$

	a	b	c
→ $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
$\{q_1, q_2\}$	ϕ	$\{q_1, q_2\}$	

$\hat{\delta}([q_0 q_1], a) \Rightarrow$ ∈-closure (δ (∈-closure $(q_0 q_1)$, a))

$\Rightarrow$ ∈- closure ($\delta(q_0 q_1), a)$)

$\Rightarrow$ ∈- closure $(\delta[q_0, a) \cup \delta(q_1, a)\} \cup \delta(q_2, a))$

$\Rightarrow$ ∈- closure $(q_0 \cup \phi \cup \phi)$

$\Rightarrow$ ∈- closure $(q_0 q_1)$

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

$\hat{\delta}([q_0 q_1], b) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0 q_1), b))$

$\Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b))$

$\Rightarrow \epsilon\text{-closure}(\phi \cup q_1 \cup \phi)$

$\Rightarrow \epsilon\text{-closure}(q_1)$

$\Rightarrow \{q_1, q_2\}$

$\hat{\delta}([q_0 q_1 q_2], c) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, q_1, q_2), c))$

$\Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, c) \cup \delta(q_1, c) \cup \delta(q_2, c))$

$\Rightarrow \epsilon\text{-closure}(\phi \cup \phi \cup q_2)$

$\Rightarrow \{q_2\}$

$\hat{\delta}([q_1 q_2], a) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1, q_2), a))$

$\Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1, a) \cup \delta(q_2, a))$

$\Rightarrow \epsilon\text{-closure}(\delta \phi \cup \phi)$

$\Rightarrow \phi$

$\hat{\delta}[(q_1, q_2), b] \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1, q_2), b))$

$\Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1, b) \cup \delta(q_2, b))$

$\Rightarrow \epsilon\text{-closure}(q_1 \cup \phi)$

$\Rightarrow \{q_1, q_2\}$

NFA - Applications: Text Search.

→ It is used for web search (www). and "finding information from text".

Finding strings in Text:-

1) inverted indices — maintain some set of words & occurances
   • It requires more memory to traveller to search indices
2) Finite Automata
   ↓
   It is NFA
   →
   1) Take a transistion on an intial state upon each input symbol.
   2) To accept the string $a_1, a_2, \ldots a_k$ Take transistion from q_0 to q_1 on a_1
   1) Take Transistion from q_1 to q_2 on a_2 and so on ... Then q_k becomes the final state

→ Thats why it is used.

→ This is not suitable for where documents require rapid change.

= UNIT - II =

Regular Expression: It is a simple expression to describe a language that the DFA accept.

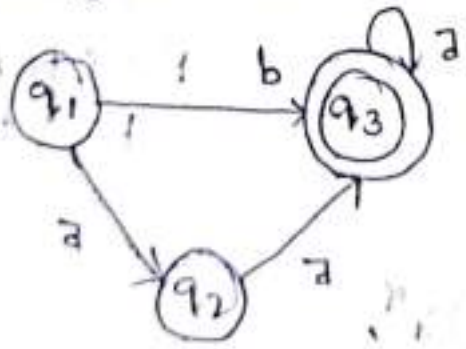
Finite Automata:-

Step1: Take equation for each state based on incoming edges.

Step2: Add ε to eqn of initial state.

Step3: Simplify the eqns. using Arden's method and find regular expression for the final state.

1) convert given DFA into regular expression.



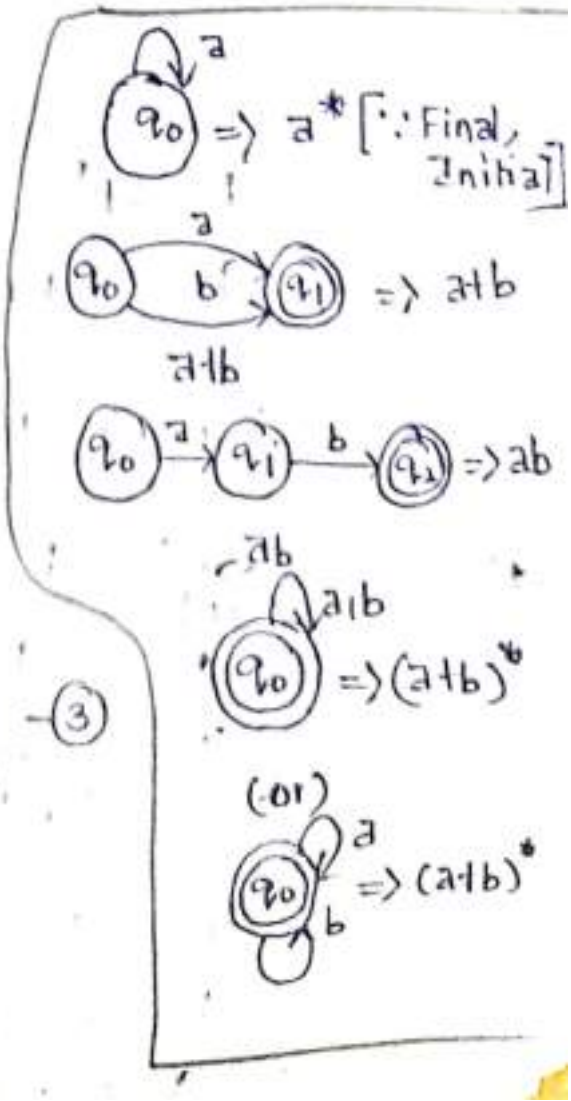
Step1: Incoming edges

1) $q_1 = \varepsilon$ — (1)

2) $q_2 = q_1 a$ — (2)

3) $q_3 = q_1 b + q_2 a + q_3 a$ — (3)

Step 2:- $q_1 = \varepsilon$ — (1)



Step 3:- $R = Q + RP \Rightarrow R = QP^*$

Subsitute eq① in eq②

$q_2 = q_1 a$

$= \varepsilon a$

$q_2 = a$ — ④

Subsitute eq④ & ③ in eq.③

$q_3 = \varepsilon b + q_2 a + q_3 a$

$= \varepsilon b + aa + q_3 a$

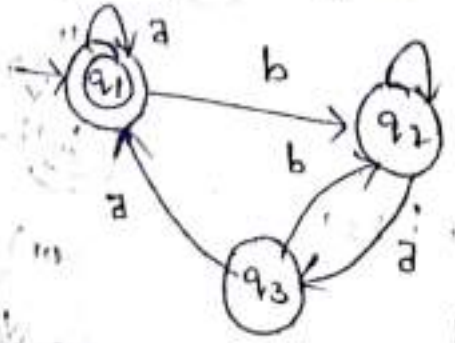
$\frac{q_3}{R} = \frac{b + aa}{Q} + \frac{q_3}{R} \cdot \frac{a}{P}$ [This is in the form of $R = Q + RP$ $R = Q + RP$]

$R = QP^*$

$q_3 = (b + aa)a^*$ //

∴ This is regular expression.

②



1) $q_1 = q_1 a + q_3 a$ — ①

2) $q_2 = q_2 b + q_1 b + q_3 b$ — ②

3) $q_3 = q_2 a$ — ③

step2: $q_1 = q_1 a + \varepsilon$

step3: $R = Q + RP$

Substitute eq ③ in eq ②

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

$q_2 = q_2 b + q_1 b + q_2 ab$

$q_2 = q_1 b + q_2 b + q_2 ab$

$q_2 = q_1 b + q_2 (b + ab)$

$[\because R = Q + RP]$

$\boxed{q_2 = (q_1 b)(b + ab)^*}$ ⇓ $\therefore R = QP^*$

Substitute ④ in ③

$q_3 = q_2 a$

$q_3 = (q_1 b)(b + ab)^* a$ — ⑤

Substitute eq ⑤ in eq ①

$q_1 = q_1 a + q_3 a + \varepsilon$

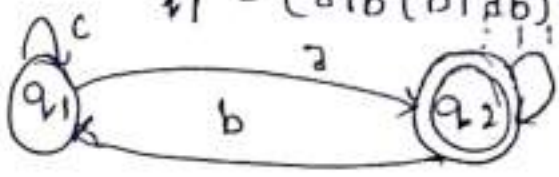
$q_1 = q_1 a + (q_1 b)(b + ab) a + \varepsilon$

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

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

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

⑥



Sol:- Step 1: $q_1 = q_2 b + q_1 c$ — (1)

$q_2 = q_1 a + q_2 d$ — (2)

Step 2: $q_1 = q_2 b + q_1 c + \varepsilon$ — (3)

Step 3: $q_1 = (q_2 b + \varepsilon) + q_1 c$

$q_1 = (q_2 b + \varepsilon) c^*$ $\Rightarrow RP = R = QP^*$

subsitute eq (3) in eq (2)

$q_2 = q_2 d + q_1 a$

$q_2 = q_2 d + (q_2 b + \varepsilon) c^* a$

$q_2 = q_2 d + q_2 b c^* a + c^* a$

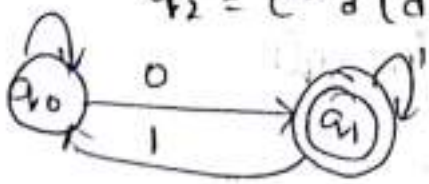
$q_2 = q_2 (d + b c^* a) + c^* a$

$\therefore [R = RP + Q]$

$\therefore$ Now write this in $R = QP^*$ form

$q_2 = c^* a (d + b c^* a)^*$

(1)



Step 1: $q_0 = q_0 1, 1 0 + q_1, 1$ — (1)

$q_1 = q_1 1 + q_0 0$ — (1)

Step 2: $q_0 = q_0 0 + q_1 1 + \varepsilon$

$R = RP + Q$

$q_0 = (q_1 1 + \epsilon) 0^*$ → ③

Substitute eq ③ in eq ②

$q_1 = q_1 1 + (q_1 1 + \epsilon) 0^*$

$q_1 = q_1 1 + q_1 1 0^* 0 + 0^*$

$q_1 = q_1 (1 + 1 0^* 0) + 0^* 0$

$= q_1 = (0^* 0)(1 + 10^* 0)^*$

DFA to regular expression by state elemination method:-

Step 1:- Initial state should not have incoming edges, if exists create a new state make it as initial state.

Step 2:- Finite automata should have only one final state, if there are multiple final states then create a new state and make it as final.

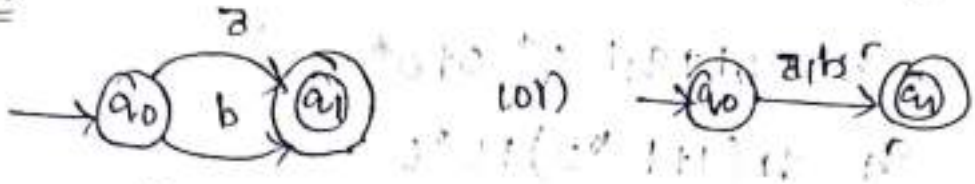
Step 3:- Final state should not have any outgoing edges if exists create a new state and make it as final state.

Step 4:- Eliminate every state except initial and final states.

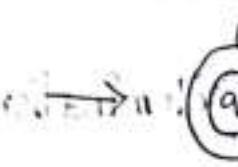
Step 5: Finally FA should have only intial state and final states.

Step 6: The label from initial state to final state becoming the regular expression:

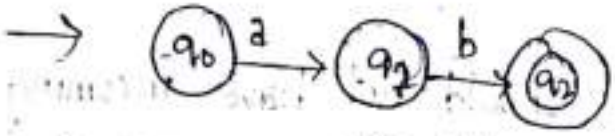
Q:-

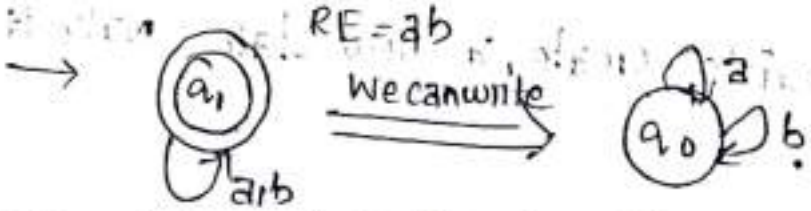


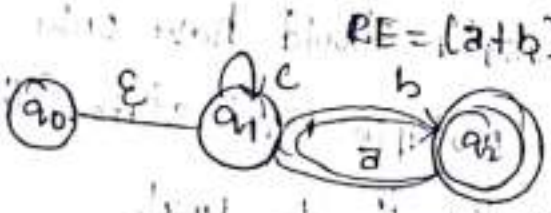
↳ R.E = a+b



$\Rightarrow RE = a^*$







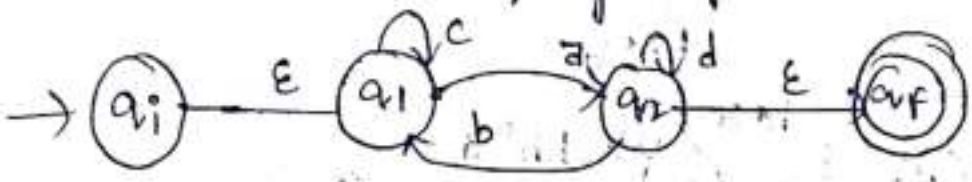
$\varepsilon \cdot (c^*(b+ab)) = c^*b + c^*ab$

(2)

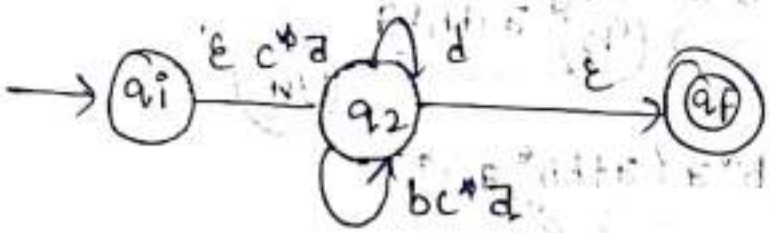
Since, there is an incoming edge for q_1 take a new state as initial state:-

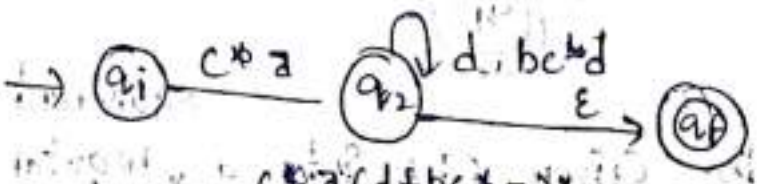
step3:-

since there are outgoing edges from

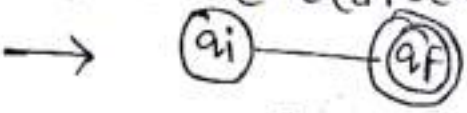


eliminate q_1 then DFA



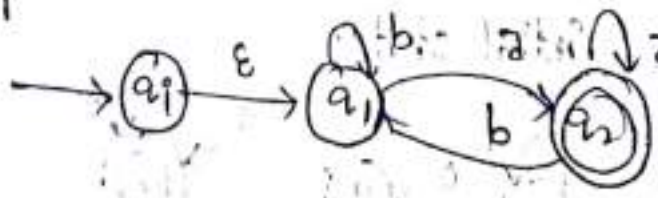


$c^*a(d+bc^*a)^*$

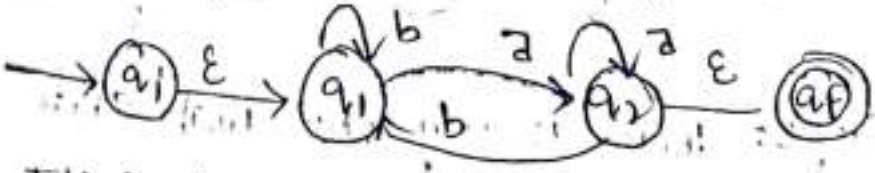


=> Write the regular expression:

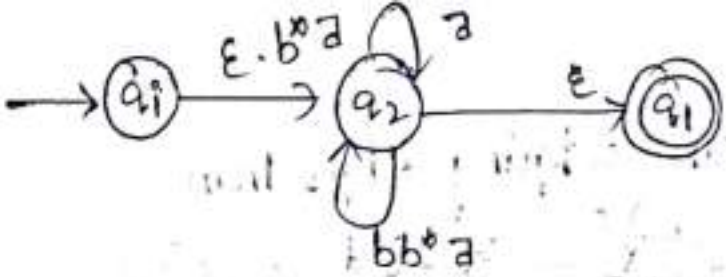
step1:

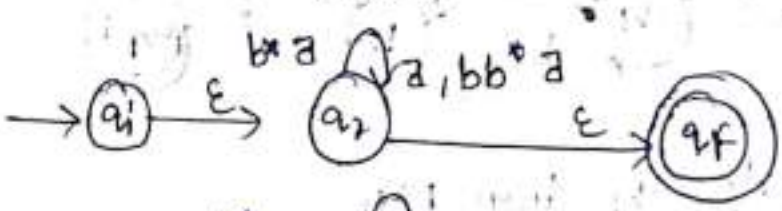


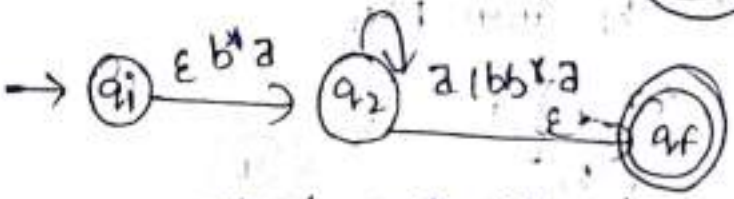
step2:

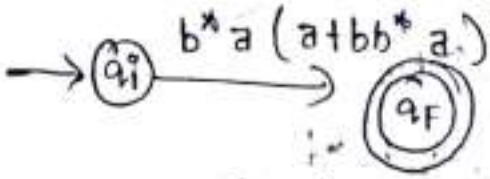


step3: Eliminate the 'q1' state.

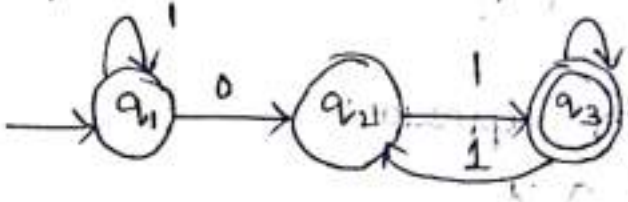




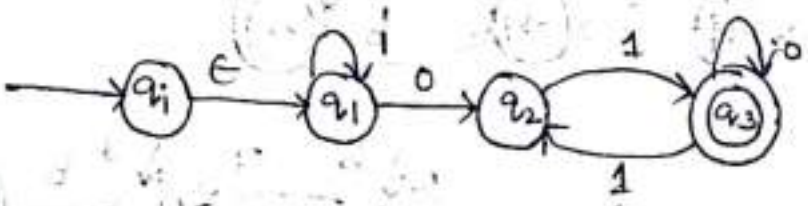




⇒ Convert the Given DFA into a equivalent regular expression.



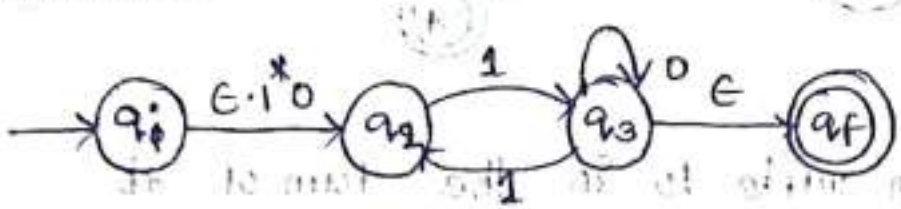
Step1: Since there is an incoming edge to intial, $\$$ create a new state (q_i) and make it as intial state.



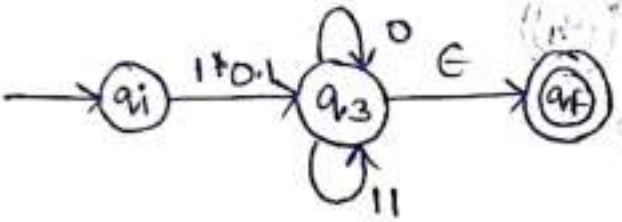
Step2: There is only one final state.

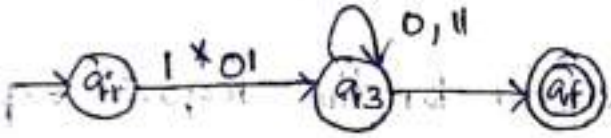
Step3: Since there are outgoing edges to final state, create a new state and make it as final state.

Step4: Eliminate q_1 then the resultant DFA becomes

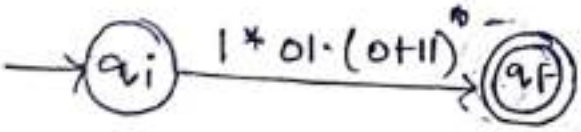


Eliminate q_2 then the resultant DFA becomes





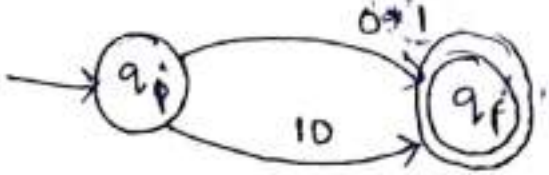
Eliminate q_3 then the resultant DFA becomes



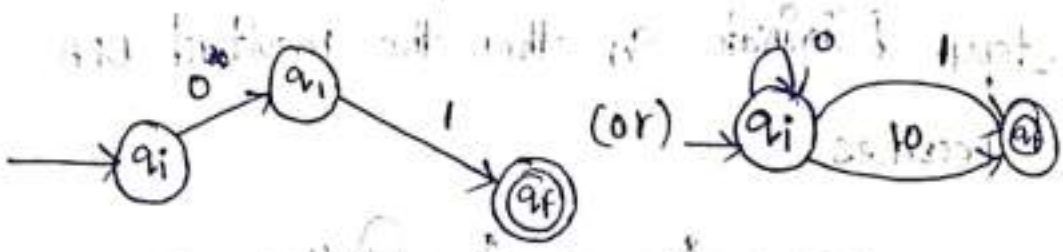
Regular Expression (RE) = $1^*01(0+11)^*$

⇒ convert given regular expression into its equivalent finite Automato.

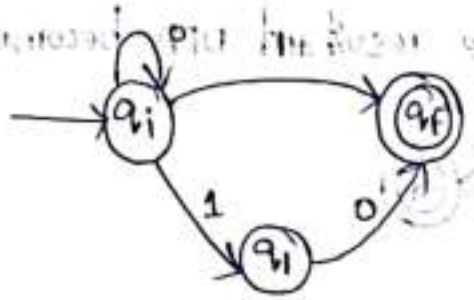
$$\underset{a}{0^*1} + \underset{b}{10}$$



a is $\underbrace{0^*}_{a}\underbrace{1}_{b}$ and it is in the form of ab

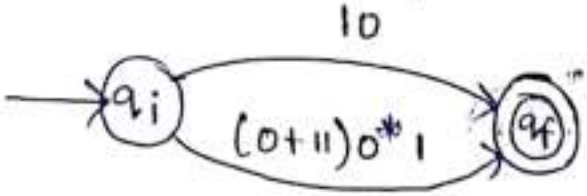


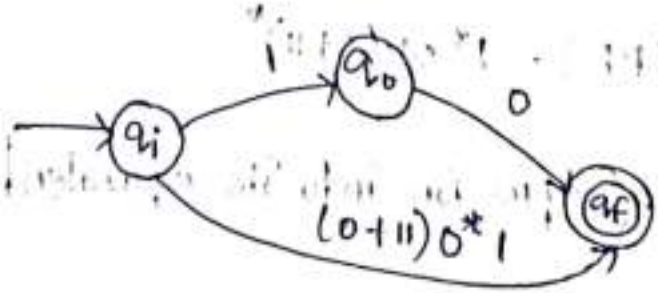
We can write 10 in the form of ab

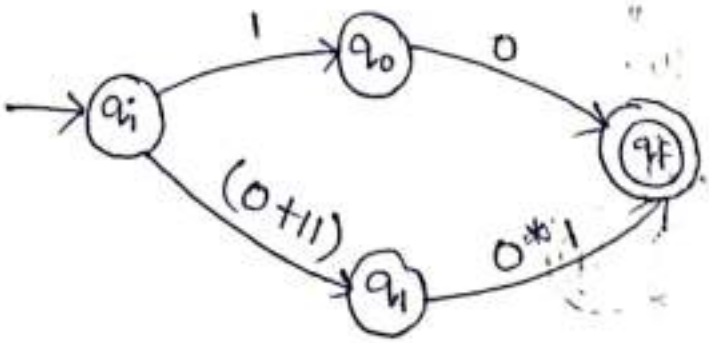


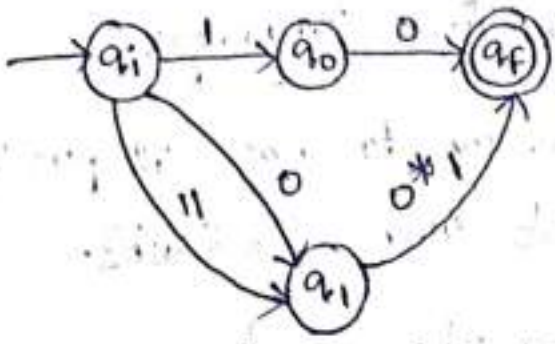
⇒ Find Finite Automata for the regular expression

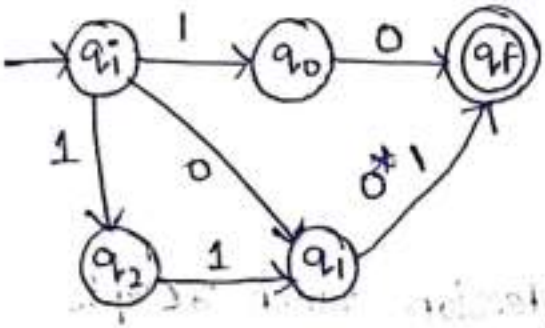
$$\underbrace{10}_{a} + \underbrace{(0+11)0^*1}_{b}$$

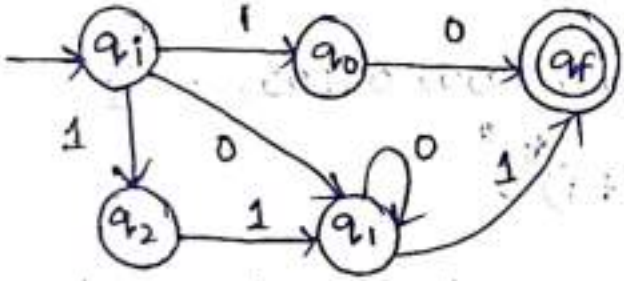






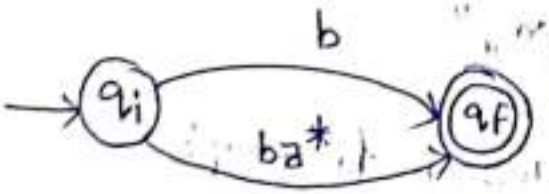


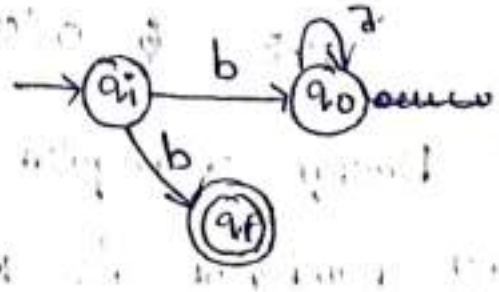


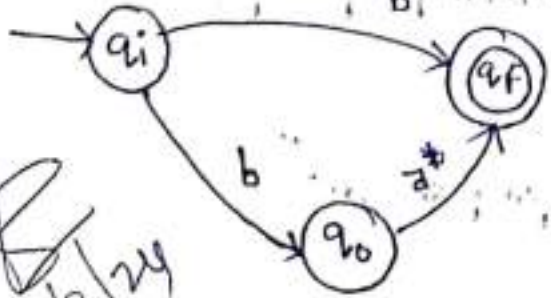


⇒ Find finite Automato for the regular expression

$$\frac{b}{a} + \frac{ba^*}{b}$$







b/3/24

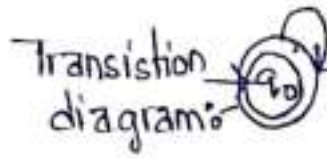
Regular Expression for the Given languages:-

i) Design the regular expression to access all possible combinations of a's and b's over $\Sigma = \{a, b\}$.

Sol:- $L = \{\epsilon, a, b, ab, aab, baa, aba, \ldots\}$

"$R.E = (a+b)^*$"

Transition diagram:



ii) Design a regular expression which accepts all strings which ends with "00".

Sol:- $L = \{00, 100, 1100, 0100, 01100, \ldots\}$

"$R.E = (0+1)^*00$"

iii) Design a regular expression for to accepts all strings which starts with "0" and ends with "1".

Sol:- $L = \{0101, 01, 011, 01011, \ldots\}$

"$R.E = 0(0+1)^*1$"

iv) Design a regular expression to accepts any number of a's following by any no. of b's and any no. of c's.

Sol:- "$R.E = (a^* b^* c^*)$"

⇒ v) atleast one b's, one a's, one c's

" R.E = $(aa^* bb^* cc^*)[or](a^+ b^+ c^+)$ "

vi) begins with "00 (or) 11" and ends with "00(or)11"

sol:- " R.E = $(00+11)(0+1)^*(00+11)$ "

vii) third character from the right end is "a"

sol:- " R.E = $(a+b)^* a (a+b)(a+b)$ "

viii) design R.E for atleast 2 b's.

sol:- " RE = $(a+b)^* b (a+b)^* b (a+b)^*$ "

ix) " exactly two "b's.

R.E = $a^* b\ a^* b\ a^*$

x) Even length strings over $\Sigma = (0)$

" R.E = $(00)^*$ "

$L = \{\epsilon, 00, 0000, 000000, \ldots\}$

xi) odd length strings over $\Sigma = (1)$

" R.E = $1(11)^*$

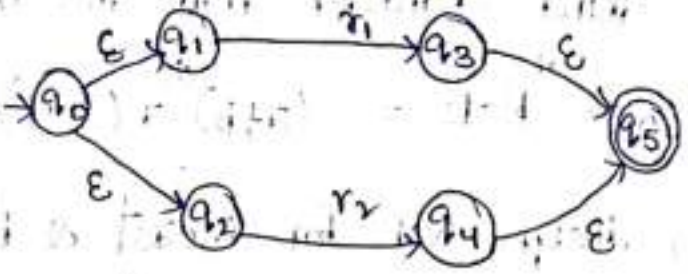
$L = \{\epsilon, 111, 1, 11111, 1111111, \ldots\}$

Regular Expression to Finite Automata using Subset method:

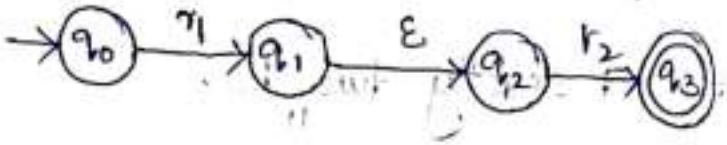
Step1: Convert the given expression into NFA with ε

Step2: Convert NFA with ε into DFA.

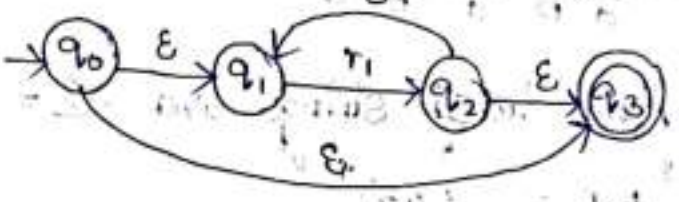
⇒ $r_1 + r_2$ →



⇒ $r_1 r_2$ →



⇒ r_1^* →



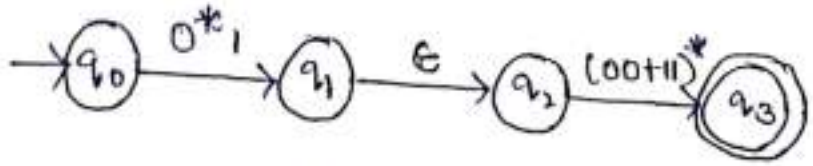
Q:- Convert the Given expression into the finite Automata using "subset" method.

$$\underbrace{0^*1}_{r_1}\underbrace{(00+11)^*}_{r_2}$$

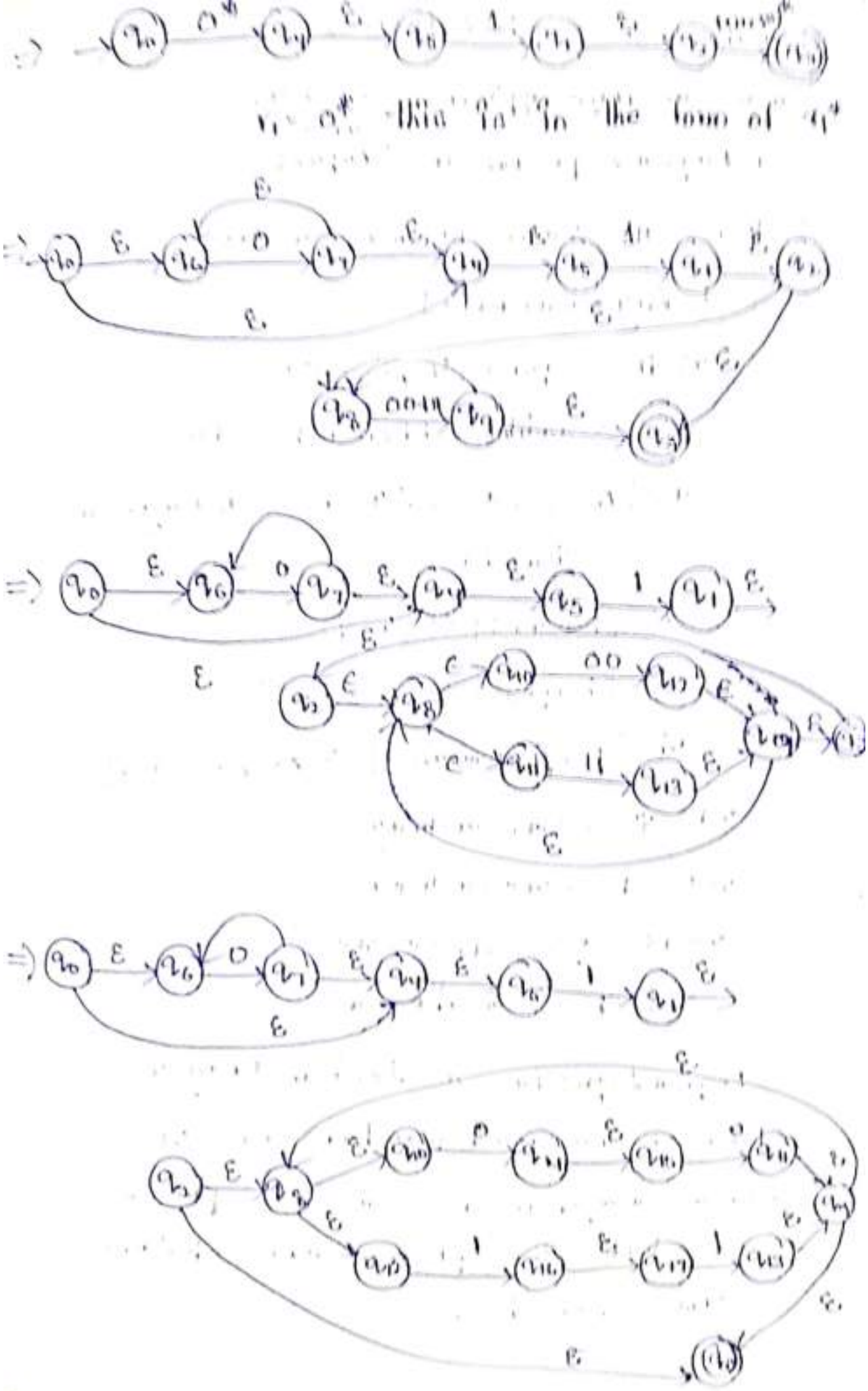
A:- This is in the form of "$r_1 r_2$"

where $r_1 = 0^*1$, $r_2 = (00+11)^*$

⇒



$r_1 = 0^*1$ Again this is in the form of $r_1 r_2 \Rightarrow r_1 = 0^*$, $r_2 = 1$



Applications of Regular Expressions:-

=> There are three applications of R.E

1) Regular Expressions in Unix:-

→ There are various notations in unix

i) Any character [.]

ii) The sequence [1,2,.....,10]

=> addition = 1+2+3+4+.....+10

iii) [A-z], [a-z], [0-9] → A range of character

iv) [:digit:] → [0-9]

v) [:alpha:] → [A-z, a-z]

vi) [:alphanumeric:] → [A-z, a-z, 0-9]

vii) * → zero (or) many

viii) + → one (or) many

ix) ? → zero (or) one

x) | → concatenation

2) Regular Expressions in Lexical Analyser:-

→ One of the most important phase of lexical analyser is to scan the program and divide the program into individual unix called tokens.

→ semantic analyser provides some tasks to the token.

→ First phase of the compiler it pro divides the program into tokens.

→ In tokens, there are identifier and keyword

→ In unix, there are lex and flex they decide what how the lexical analyser do in unix operating system.

8) Regular Expressions in finding patterns in the string:

→ To find the patterns in the string we use RE.

123A Main st:

[0-9]+[A-Z]?[A-Z][a-z]*([A-Z](a-z)*)*

(Street|st\.|Road|Rd\.):

Algebraic laws of Regular Expression:

Associative laws for regular Expression RegEx

$A+(B+C) = (A+B)+C$ and $A(B.C) = (A.B)C$

Commutative laws for Regular Expression RegEx

$A+B = B+A$ however $A \cdot B \neq B \cdot A$

Identity for Regular Expression: RegEx

$$\phi + A = A + \phi = A$$

$$\epsilon \cdot A = A \cdot \epsilon = A$$

Distributivity for Regular Expression

$A(B+C) = AB + AC$ [left]

$(B+C)A = BA + CA$ [right]

→ ϕ is an anclilator

Idempotent $A + A = A$

closure laws for RegularExpression: RegEx

$(A^*)^* = A^*$, $\phi^* = \epsilon$, $\epsilon^* = \epsilon$, $A+ = AA^*$, and $= A^*A$

$A^* = A + + \epsilon$.

Demorgon's Law:

$(L+B)^* = (L^*B^*)^*$.

Test for the Regular Expression whether $E = F$

1) convert the E and F into concreate R.E C & D
2) Test $L(C) = L(D)$ th. if Language of C is equals to the Language of D then we can say $E = F$ is TRUE otherwise FALSE.

pumping lemma:- → To prove the language is regular expression or not.

→ repetion [Pumping] lemma [word] → It is entry point, using that we can ~~create~~ derive other words.

→ Repetition of that word, by that word can derive the other words.

→ There are two types of pumping lemma:

1) Regular Expression for pumping lemma
2) context free languages

Theorem:-

let "L" be the Regular language, Then there exists a constant "c" such that for each and every string "w" in "L".

$|w| \geq c$

We can break the w in "v" three "v" strings; 'x, y, z'

- $|y| > 0$
- $|xy| \leq c$
- For all $k \geq 0$, the string xy^kz is also in L.

procedure to prove a language is not regular using pumping lemmas:

1) consider L is a regular language

2) select constant 'c' such that $|w| \geq c$, for every string (w) in L.

3) Take a string (w) from L and divide w into x, y and z.

such that

$|y| > 0$

$|xy| \leq c$

for $k \geq 0$, $xy^k z$ is in L.

4) If not, say L is a inregular.

problem:-

prove $L = \{ a^m b^n \mid \gcd(m,n) = 1 \}$ is not a regular language.

Ans:- $L = \{aabbb, aaabbbb, aaaabbbbb, \ldots\}$

1. let L is a regular language

2. Constant $c = 5$, where $|w| \geq c$

3. Let $w = \underset{x}{aa}\underset{y}{abbbb}\underset{z}{bb}$

divide w into x, y, z such that $|y| > 0$ & $|xy| \leq c$

then $x = aa$, $y = ab$, $z = bbb$

for every $i \geq 0$, compute $xy^i z$

let $i = 0$.

$xy^0 z \Rightarrow xy^0 z \Rightarrow x \cdot y^0 \cdot z = xz = aabbb$, this is in L

let $i = 1$.

$\Rightarrow xy^1 z \Rightarrow x \cdot y^1 \cdot z = xyz = aaabbbb$ is in L

let $i = 2$.

$\Rightarrow xy^1 z \Rightarrow x \cdot y^2 \cdot z = xyyz = aaabaabbbb$ is not in L

∴ The given language L is irregular not reg.

Q: prove $L = \{0^n \mid n$ is a perfect square$\}$ is not a regular language.

A: $L = \{0, 0000, 000000000, 0000000000000000\}$

1) let L is a regular language.

2) constant $c \geq 1$, where $|w| \geq c$

3) select $w = 0000$.

divide w in xyz such that

$|y| > 0$ & $|xy| \leq c$ then

$x = \epsilon$, $y = 0$, $z = 000$

for every $i \geq 0$, compute $xy^i z$

let $i = 0$,

$\Rightarrow xy^i z \Rightarrow xz = \epsilon \cdot 000 \Rightarrow 000$ is not in L

let $k=1$,

$\Rightarrow xy^kz \Rightarrow xyz \Rightarrow \epsilon \cdot 0 \cdot 000 \Rightarrow 0000$ is in L.

let $k=2$,

$\Rightarrow xy^kz = xyyz \Rightarrow \epsilon \cdot 0 \cdot 0 \cdot 000 \Rightarrow 00000$ is not in L

∴ The given language L is not regular.

Context-free Grammers:- (CFG) set of rules.

→ Grammer is a Quadruple Notations

$G = (V, T, P, S)$

V → set of non-terminals (or) variables, (uppercase letters)

T → set of terminals (lowercase letters)

P → set of productions $(\alpha \to \beta)$ - these are constructed to terminals & Non-Terminals $(\alpha = aA, \beta = bBa)$

$\Rightarrow \alpha, \beta \in (V \cup T)^*$

S → start symbol

Types of Grammers:- 4 types of Grammers

1) Type 0:- It is a unrestricted Grammer.

→ The language Generated by this Type Grammer is "Recursively Enumerable language (REL)".

→ To represent this we use "Turing Machine (TM)".

Produ
$\Rightarrow \alpha \rightarrow \beta$

where α has atleast one non-terminal.

Context free Grammer (CFG):-

A context free grammer (G) is a 4 tuple notation. where $G = (V, T, P, S)$

Where
- V → set of non Terminals or Variables → represented in Upper case letters.
- T → set of Terminals → represented in lowercase letters.
- S → start Symbol
- P → set of productions of the form $(\alpha \rightarrow \beta)$ where $\alpha \in V$, $|\alpha| = 1$ & (β can be a terminal or terminal followed by non-Terminal or non-terminal followed by a terminal)

Derivations:-
→ productions of the Grammer are used to infer the strings of the language.
→ There are two approaches
1) Recursive Inference
2) Derivations

1) Recursive Inference Approach:-

→ This is a bottom up approach (body to head).

→ We start with known symbols (terminals).

Grammer is:

1. E → I
2. E → E+E
3. E → E*E
4. E → (E)
5. I → b
6. I → a
7. I → Ib
8. I → Ia
9. I → Ia
10. I → Ib

→ The string to be derived is a*(a+boo)

a*(a+boo)	E*(E+Ioo)	E*E
I*(a+boo)	E*(E+Io)	E
E*(a+boo)	E*(E+I)	
E*(I+boo)	E*(E+E)	
E*(E+boo)	E*(E\	

S.NO	string Infered	for language of	Productions used	steps used
1)	a	I	6	-
2)	a	E	1	1)
3)	b	I	5	-
4)	bo	I	7	3)
5)	boo	I	7	4)

6) boo E 1 5)

7) a+boo E 2 2),6)

8) (a+boo) E 4 7)

9) a*(a+boo) E 3 2),8)

2) Derivation Approach:-

→ This is a topdown approach.

→ Let "G" be a "CFG",

$$\alpha A \beta \in (V \cup T)^*$$

Where, A is a non terminal and $A \to \gamma$ then

$$\alpha A \beta \overset{G}{\Rightarrow} \alpha \gamma \beta .$$

⇒ a * (a+boo)

Grammer is:

1. E → I
2. E → E+E
3. E → E*E
4. E → (E)
5. I → b
6. I → a
7. I → Io
8. I → I1
9. I → Ia
10. I → Ib

E → E*E → I*E → a*E → a*(E) → a*(E+E)

→ a*(I+E) → a*(a+E) → a*(a+I) → a*(a+Io

→ a*(a+Io) → a*(a+Ioo) → "a*(a+boo)".

Left Most Derivation And Right Most Derivation:

1) LeftMost Derivation:

In which At each step of derivation

If we replace left most variable with its production body.

→ We represent leftmost derivation by $\overset{*}{\underset{lm}{\Rightarrow}}$

2) Rightmost Derivation:

In which, at each step of derivation, we replace rightmost variable with its production body.

→ we represent leftmost derivation by $\overset{*}{\underset{rm}{\Rightarrow}}$

Examples:-

→ E
→ E*E
→ E*(E)
→ E*(E+E)
→ E*(E+I)
→ E*(E+Io)
→ E*(E+Ioo)
→ E*(E+boo)

→ E*(I+boo)
→ E*(a+boo)
→ I*(a+boo)
→ a*(a+boo)

→ Derive "abababa" from the given Grammer and leftmost derivation approach.

$S \rightarrow SbS \mid a$

case i) → S

→ SbS {S → SbS}
→ SbSbS {S → SbS}
→ SbSbSbS {S → Sbs}
→ abSbSbS {S → a}
→ ababSbS {S → a}
→ abababS {S → a}
→ "abababa" {S → a}

case ii) → S

→ SbS {S → SbS}
→ abS {S → a}
→ abSbS {S → SbS}
→ abSbSbS {S → SbS}
→ ababSbS {S → a}
→ abababS {S → a}
→ abababa {S → a}

Language of Context free Grammer:

Let "G" be a context free Grammer, then its language is denoted by $L(G)$ and it is defined as $L(G) = \{w \text{ in } T^* / S \overset{*}{\underset{G}{\Rightarrow}} w\}$.

→ The language of CFG is known as "context free language".

Sentential Form:-

Let "G" be a "CFG", $\alpha \in (V \cup T)^*$, S is a start Symbol, If $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ then α is in "sentential Form".

→ There are two types:

1) Left most sentential form:

If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$ then your α is left most sentential form.

2) Rightmost Sentential form:

If $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$ then your α is rightmost sentential form.

Parse tree / derivation tree / syntax tree:-

A parse tree is a tree which is constructed with the following conditions:

1) root node must be a start symbol.

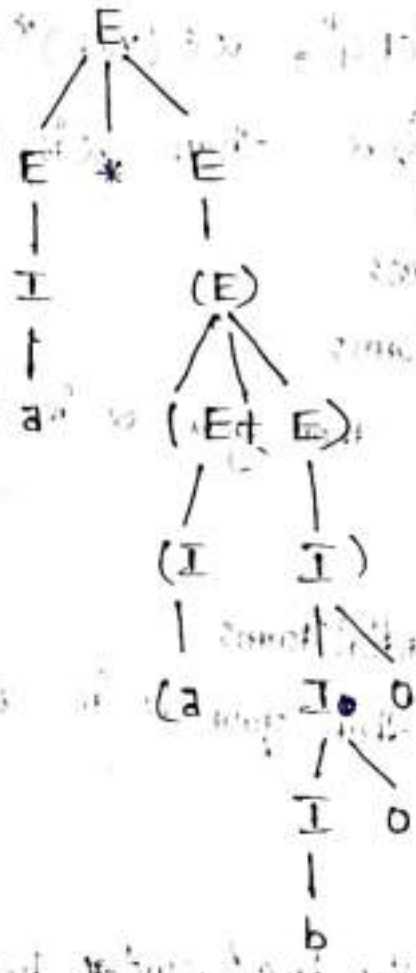
2) Intermideate / Interal node must be a variable (including starting stymbol).

3) leafnode must be a terminal (or) ε

→ construct parse tree for the string

a*(a+boo) using the CFG E→E+E/E*E/(E)/I. I→Ib/Ia/Io/I1/b/a.

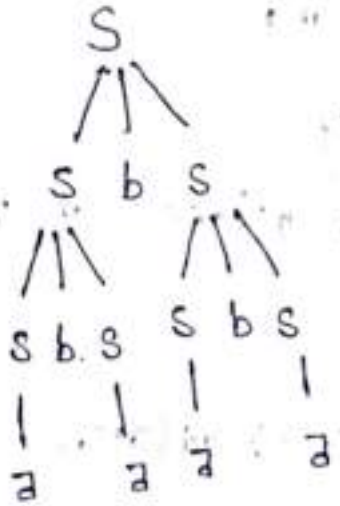
Ans:-



→ construct parse tree for the string

abababa using the Grammer

S → sbs / a.

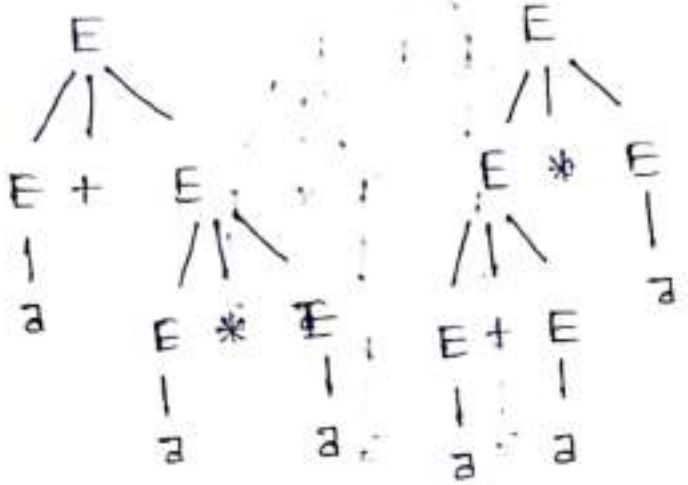


Ambiguos Grammer:

A "CFG" "G" is said to be ambiguos for atleast one string in its language if there exist two distinct parse trees.

Example: Grammer is $E \to E+E$, $E \to E*E$, $E \to a$

construct "a+a*a" parse tree.



∴ The Given Grammer is ambiguos Grammer.

Removing Ambiguity:

Removing ambiguty is done

i) by adding precedence and

ii) by adding associativity.

→ To implement these take an unequivalent an unambiguous grammer.

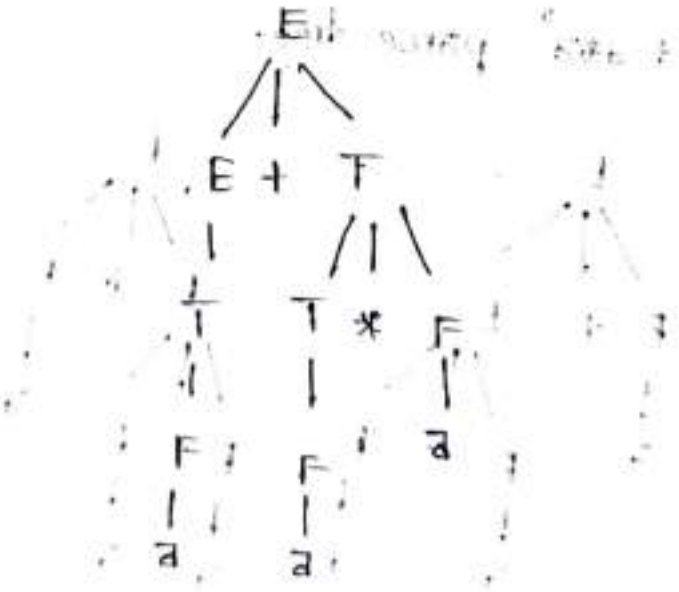
⇒ a+a+a ⇒ (a+a)+a ⇒ a+(a+a)

⇒ E→E+E · → E→E*E → F→a

→ E→E+T → T→T*F

→ E→T → T→F

a+a*a



Leftmost derivations in ambiguity:

→ There are more derivations in leftmost and Rightmost ambiguity.

→ There are only one derivation in the Un ambiguty Grammer.

Inherent Ambiguos:-

A danguage is said to be ambiguty if all its Grammel is ambiguos.

— Unit-III —

push down Automata (PDA):-

→ If we add stack to the finite Automata then it will become Pushdown Automata.

→ If we want to store the infinite information, we cannot store in finite automata then we use the push down Automata.

→ PDA = FA + stack

→ This is used to represent the context free Grammer.

→ In PDA, we use three components

1) input tape / input string
2) finite control (Push, Pop)
3) stack

Formal Defination of PDA:-

→ It is represented by "P", it is a 7 tuple

Notation $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Where, P = PDA

Q = set of states

$\Sigma \rightarrow$ set of input symbols

$T \rightarrow$ set of stack elements

$\delta \rightarrow$ It is a transition Function

$$\delta(q_0, a, x) = (q_n, \alpha)$$

$q \rightarrow$ current state

$a \rightarrow$ input to be processed

$x \rightarrow$ top of the stack

$q_n \rightarrow$ new state

$\alpha \rightarrow$ Symbol to be replaced in place of x

$q_0 \rightarrow$ Initial state

$z_0 \rightarrow$ top of stack

$F \rightarrow$ set of accepting states/Final states

Q:- Construct PDA for the given language

$$L = \{a^n b^n / n \geq 1\}$$

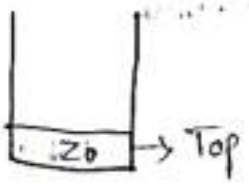
Sol:- $L = \{a^n b^n / n \geq 1\}$

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

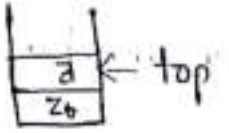
Let us assume "aaabbb" is our input tape/string

q_0 is the initial state.

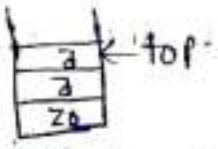
Let z_0 be at the top of the stack.



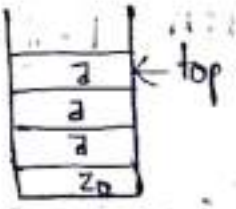
$\delta(q_0, a, z_0) = (q_0, az_0)$



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

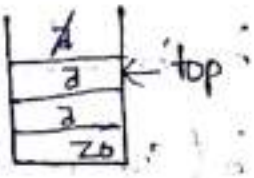


$\delta(q_0, a, a) = (q_0, aaa)$

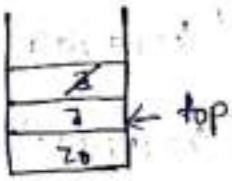


$\delta(q_0, b, a) = (q_1, \epsilon)$

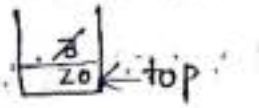
$\delta(q_1, b, a) = (q_1, \epsilon)$



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



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



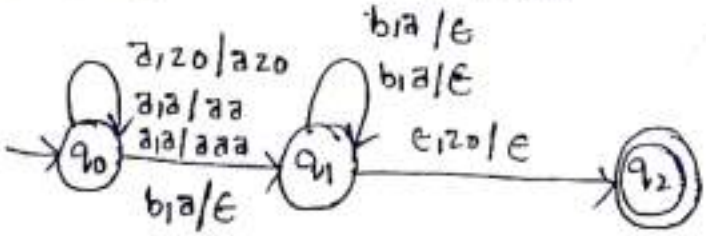
Stack empty.

$PDA = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

$Q \rightarrow \{q_0, q_1, q_2\}$ $\quad \Gamma \rightarrow \{a, z_0\}$ $\quad F \rightarrow \{q_2\}$

$\Sigma \rightarrow \{a, b\}$ $\quad \delta$

$q_0 \rightarrow q_0$ $\quad z_0 \rightarrow \{z_0\}$



Instantaneous Description of PDAs-

→ To check the acceptance of string

find the acceptance of a string aabbb by the PDA $P = \{q_0, q_1, q_2\}$

$$P = [\{q_0, q_1, q_2\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \{q_2\}]$$

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

$\delta(q_0, a, az_0) = (q_0, aaz_0)$

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

$\delta(q_0, b, aaa) = (q_1, \varepsilon)$

$\delta(q_0, b, aa) = (q_1, \varepsilon)$

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

$\delta(q_1, \varepsilon, z_0) = (q_2, z_0)$

$\delta(q_0, aabb, z_0) \vdash (q_0, abb, az_0)$

$\vdash (q_0, bb, aaz_0)$

$\vdash (q_1, b, az_0)$

$\vdash (q_1, \varepsilon, z_0)$

$\vdash (q_2, z_0)$

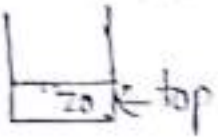
∴ the Given string aabb is accepted by $t(q_2, z_0)$ by the PDA

construct PDA to accept the language $L = \{a^n b^{2n} / n \geq 1\}$ and find acceptance of string "aaabbbbbb".

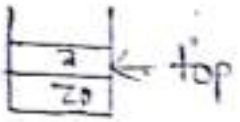
Ans:- push for "a" and pop for every two "b"s.

$L = \{abb, aabbbb, aaabbbbbb, \ldots\}$

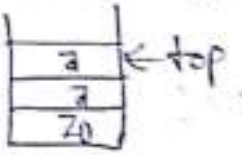
Let aabbbbϵ is the input tape, q_0 is the initial state and z_0 is at the top of stack



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



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



$\delta(q_0, b, a) = (q_1, aaz_0)$

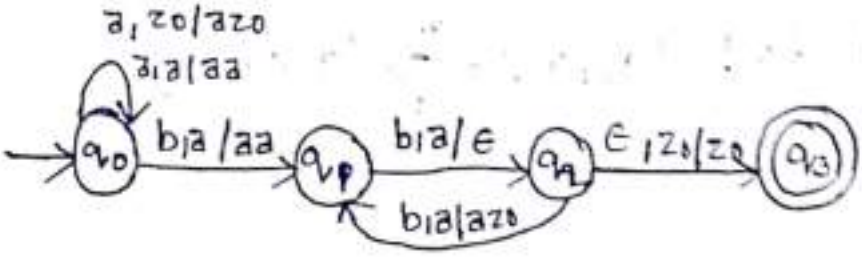
$\delta(q_1, b, a) = (q_2, \epsilon)$

$\delta(q_1, b, a) = (q_1, az_0)$

$\delta(q_1, b, a) = (q_2, \epsilon)$

$\delta(q_2, \epsilon, z_0) = (q_3, z_0)$

$P = [\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \{q_3\}]$



$\delta(q_0, aaabbbbbb, z_0) \vdash (q_0, aabbbbbb, az_0)$

$\vdash \delta(q_0, abbbbbb, aa) \vdash (q_0, bbbbbb, aaa)$

$\vdash (q_1, bbbbb, aaa) \vdash (q_2, bbbb, aa) \vdash (q_1, bbb, a)$

$\vdash (q_2, bb, a) \vdash (q_1, b, a) \vdash (q_2, \epsilon, z_0) \vdash (q_3, z_0)$

∴ The given string aaabbbbbb is accepted $\vdash (q_3, z_0)$ by the given PDA.

→ Construct PDA for the given language $L = \{a^n b^m c^n / m, n > 0\}$

A:- push for 'a's, Pop for 'c's and 'b's for no change. $m=2, n=3$, $m=1, n=2$

$L = \{abc, abbc, aaabbccc, aabcc, \ldots\}$

languages of PDA:-

1) Acceptance by Final state

$$L = \{w \mid \delta(q_0, w, z_0) \overset{*}{\underset{P}{\Rightarrow}} (q, \epsilon, \alpha)\}$$

continuation of $L = \{a^m b^n c^n / m, n > 0\}$

let aabcc $\in$ as input tape, q_0 is the initial state and z_0 at the top of the stack.

→ $\delta(q_0, a, z_0) = (q_0, az_0)$

→ $\delta(q_0, a, a) = (q_0, aa)$

→ $\delta(q_0, b, a) = (q_1, aa)$

→ $\delta(q_1, c, a) = (q_e, a)$

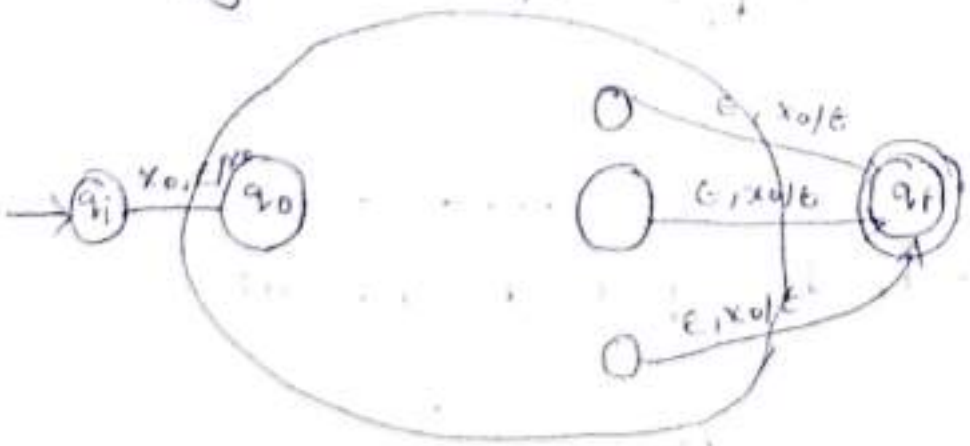
→ $\delta(q_2, c, a) = (q_2, z_0)$

→ $\delta(q_{r2}, \epsilon, z_0) = (q_3, z_0)$

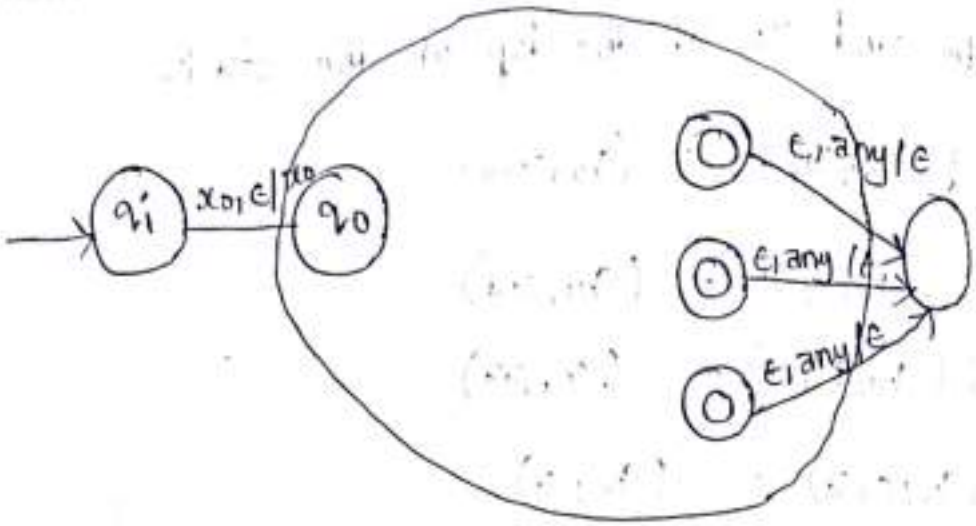
3) Acceptance by Empty stack:

$$N(P) = \{ w / \delta(q_0, w, z_0) \overset{*}{\underset{P}{\Rightarrow}} (q, \epsilon, \epsilon) \}$$

4) From empty stack to final state:



4) From final state to empty stack



→ Equivalence of PDA and CFG's

Conversion of CFG to PDA:-

A For non terminal: $A \to \alpha$

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

For terminal:

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

d

Example:-

Convert the given CFG into PDA

$S \to AB$

$A \to 0S/0$

$B \to 1S/1$

$S \to AB$ $B \to 1S$ $A \to 0$

$A \to 0S$ $B \to 1$

Consider $S \to AB$

$$\delta(q_1, \epsilon, S) = (q_1, AB)$$

$A \to 0S$

$$\delta(q_1, \epsilon, A) = (q_1, 0S)$$

$A \to 0$

$$\delta(q_1, \epsilon, A) = (q_1, 0)$$

$B \to 1S$

$$\delta(q_1, \epsilon, B) = (q_1, 1S)$$

$B \to 1$

$$\delta(q_1, \epsilon, B) = (q_1, 1)$$

$$\Rightarrow \delta(q_1, 0, 0) = (q_1, \epsilon)$$

$$\delta(q_1, 1, 1) = (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, S) = \{(q_1, AB)\}$$

$$\delta(q_1, \epsilon, 0) = \{(q_1, 0S), (q_1, 0)\}$$

$$\delta(q_1, \epsilon, 1) = \{(q_1, 1S), (q_1, 1)\}$$

$$\delta(0, 0, 0) = \{(q_1, \epsilon)\}$$

$$\delta(1, 1, 1) = \{(q_1, \epsilon)\}$$

Construct PDA

convert the given CFG to PDA:-

$E \to E+E / E*E / I / (E)$

$I \to I_0 / I_1 / I_a / I_b / a / b$

→ $E \to E+E$

for Non-Terminal $A \to \alpha$

$\delta(q_1, \epsilon, A) = (q_1, \alpha)$

→ $E \to E+E$

$\delta(q_1, \epsilon, E) = (q_1, E+E)$

→ $E \to E*E$

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

→ $E \to I$

$\delta(q_1, \epsilon, E) = (q_1, I)$

→ $E \to (E)$

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

→ $I \to I_0$

$\delta(q_1, \epsilon, I) = (q_1, I_0)$

→ $I \to I_1$

$\delta(q_1, \epsilon, I) = (q, I_1)$

→ $I \to I_a$

$\delta(q_1, \epsilon, I) = (q_1, I_a)$

→ $I \to I_b$

$\delta(q, \epsilon, I) = (q, Ib)$

→ $I \to a$

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

→ $I \to b$

$\delta(q, \epsilon, I) = (q, b)$

For Terminal

$T = \{a, b, +, *, 0, 1\}$

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

→ $\delta(q, a, a) = (q, \epsilon)$

→ $\delta(q, b, b) = (q, \epsilon)$

→ $\delta(q, 0, 0) = (q, \epsilon)$

→ $\delta(q, 1, 1) = (q, \epsilon)$

→ $\delta(q, +, +) = (q, \epsilon)$

→ $\delta(q, *, *) = (q, \epsilon)$

$\Rightarrow \delta(q, E, E) = \{(q, E+E), (q, E*E), (q, I), (q, (E)\}$

$\Rightarrow \delta(q, \epsilon, I) = \{(q, I0), (q, I1), (q, Ia), (q, Ib), (q, a), (q, b)\}$

$\therefore PDA = [\{q\}, \{a, b, 0, 1, +, *\}, \Gamma, \delta, q, z_0, \{q\}]$

Conversion of PDA to CFG:-

convert the given PDA to CFG

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

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

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

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

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

based on Transistion functions

"if there are no stack symbols - 1

$1 \to 2$

$2 \to 4$.

$PDA = [\{q_0, q_1\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \{q_1\}]$

$Q = \{q_0, q_1\}$

$\therefore Q \Gamma Q$

Ans:- Let us assume, S be the start symbol

Find variables

$V = \{S, [q_0 a q_0], [q_0 a q_1], [q_0 z_0 q_0], [q_0 z_0 q_1], [q_1 a q_0], [q_1 a q_1], [q_1 z_0 q_0], [q_1 z_0 q_1]\}$

$T = \{a, b\}$

Find productions:

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

$[q_0 z_0 q_0] \to a [q_0 a q_0] [q_0 z_0 q_0]$

$[q_0 z_0 q_0] \to a [q_0 a q_1] [q_1 z_0 q_1]$

$[q_0 z_0 q_1] \to a [q_0 a q_0] [q_0 z_0 q_1]$

$[q_0 z_0 q_1] \to a [q_0 a q_1] [q_1 z_0 q_1]$

→ $\delta(q_0, a, a) = (q_0, aa)$

$[q_0 a q_0] \rightarrow a [q_0 a q_0] \; [q_0 a q_0]$

$[q_0 a q_0] \rightarrow a [q_0 a q_1] \; [q_1 a q_1]$

$[q_0 a q_1] \rightarrow a [q_0 a q_0] \; [q_0 a q_1]$

$[q_0 a q_1] \rightarrow a [q_0 a q_1] \; [q_0 a q_0]$

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

$[q_0 a q_0] \rightarrow b [q_1 a q_0]$

$[q_0 a q_1] \rightarrow b [q_1 a q_1]$

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

$[q_1 a q_1] \rightarrow b$

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

$[q_1 z_0 q_1] \rightarrow \epsilon$

$S \rightarrow [q_0 z_0 q_0]$

$S \rightarrow [q_0 z_0 q_1]$

CFG

Construct PDA for the Given PDA

$\delta(q_1, 1, z_0) = (q_1, 1 z_0)$

$\delta(q_1, 1, x) = (q_1, xx)$

$\delta(q_1, 0, x) = (p, x)$

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

$\delta(p, 1, x) = (p, \epsilon)$

$\delta(p, 0, z_0) = (q_1, z_0)$

$P = \{\{p, q\}, \{0, 1\}, \{x, z_0\}, \delta, q_1, z_0, \{p\}\}$

Sol:- Let S be the start Symbol

Let CFG $\Rightarrow G = (V, T, P, S)$

Find Variables

$V = \{S, [PxP], [PxQ], [QxP], [QxQ],$

$[PzoP], [PzoQ], [QzoP], [QzoQ]\}$

$T = \{0, 1\}$

P → Find productions: $S \to qz_0q$ $S \to qz_0p$

$\to \delta(q, 1, z_0) = (q_i, xz_0)$

$[q\ z_0\ q] \to 1[q\ x\ q][q\ z_0\ q]$

$[q\ z_0\ q] \to 1[q\ x\ p][p\ z_0\ q]$

$[q\ z_0\ p] \to 1[q\ x\ q][q\ z_0\ p]$

$[q\ z_0\ p] \to 1[q\ x\ p][p\ z_0\ p]$

$\to \delta(q, 1, x) = (q, xx)$

$[q\ x\ q] \to 1[q\ x\ q][q\ x\ q]$

$[q\ x\ q] \to 1[q\ x\ p][p\ x\ q]$

$[q\ x\ p] \to 1[q\ x\ q][q\ x\ p]$

$[q\ x\ p] \to 1[q\ x\ p][p\ x\ p]$

→ $\delta(q,0,x) = (p,x)$

$[q \times q] \to 0\,[p \times q]$

$[q \times p] \to 0\,[p \times q]$

→ $\delta(q,\epsilon,x) = (q,\epsilon)$

$[q \times q] \to \epsilon$

→ $\delta(p,1,x) = (p,\epsilon)$

$[p \times p] = 1$

→ $\delta(p,0,z_0) = (q,z_0)$

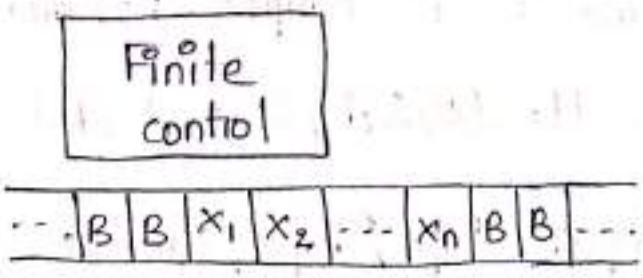
$[p\ z_0\ q] = z_0\, 0_0\, [q\ z_0\ q_0]$

$[p z_0 p] \to 0\,[q z_0 p]$

Turing Machine:

Turing machine consists of a finite control, which can be any finite set of states.

→ There is a tape divided into cells, each cell can hold finite number of symbols.



$X_1, X_2, \ldots, X_n$ → Input Symbols

B → Blank symbol

→ choose input string from the input alphabet and place in tape.

→ Extend the length of the tape to the left and right and place blank symbol over the extended cells.

→ The Blank is a tape but not input symbol.

→ There is a read write head (or) tape head; Initially the tape head is at leftmost cell that holds the input.

→ A move of the turing machine is a function of the state of the finite control and tape symbol, scanned.

→ In one move a turing machine will

1) change state: The next state optionally may be same as the current state.

2) Write a tape symbol to the cell scanned, this tape symbol replaces the symbol in that cell.

3) move the tape head to left or right but don't allow the head to remain stationary.

Defination:-

Turing machine is a "7 tuple", notation, And the symbol is $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$.

$\Gamma \rightarrow$ input tape / Tape symbols

$B \rightarrow$ Blank symbol

$Q \rightarrow$ set of states

$\Sigma \rightarrow$ set of input symbols

$q_0 \rightarrow$ Initial / starting state

$F \rightarrow$ Final state set

$\delta \rightarrow$ Transistion Function

where, the transistion Function maps to

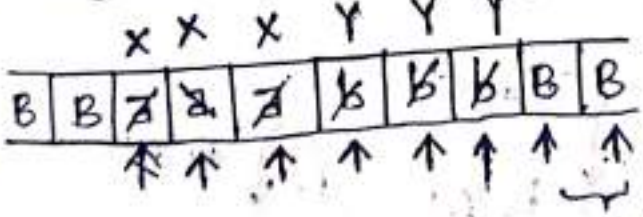
$Q \times \Sigma \Rightarrow "Q \times \Gamma \times \{L, R\}"$.

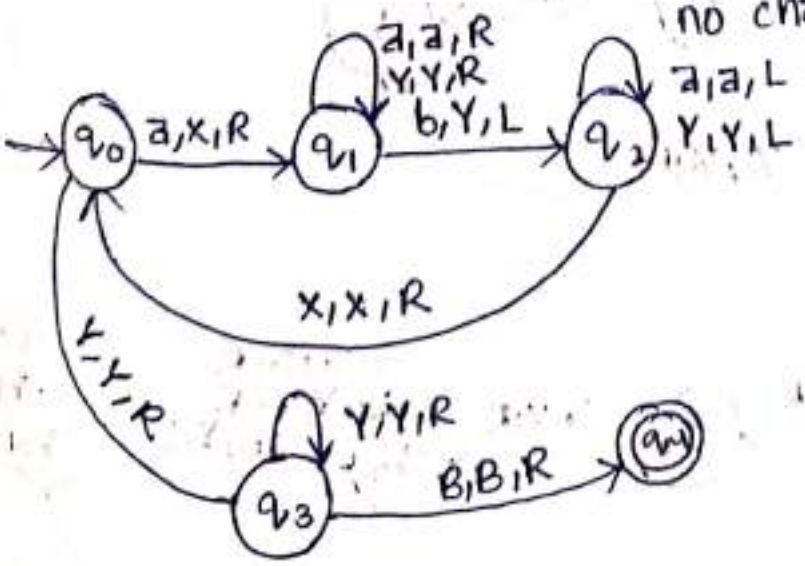
Transistion diagram in Turing machine:

Q Design Turing machine for the language $L = \{a^n b^n / n > 0\}$.

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

let "aaabbb" be the input tape, q_0 be the initial state.





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

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

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

$\Gamma = \{a, b, X, Y, B\}$

$F = \{q_4\}$

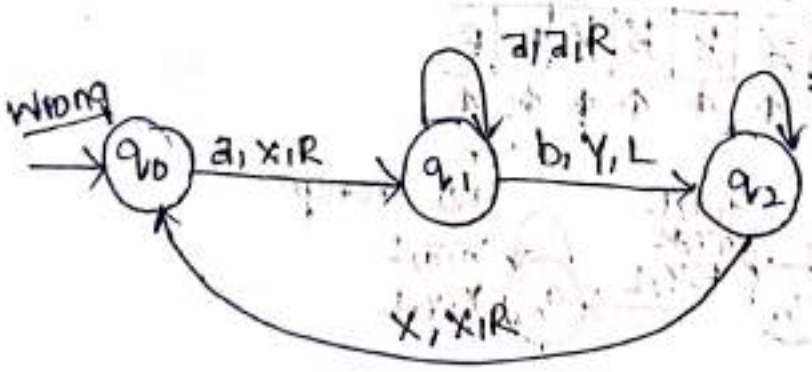
② Design Turing machine for the language

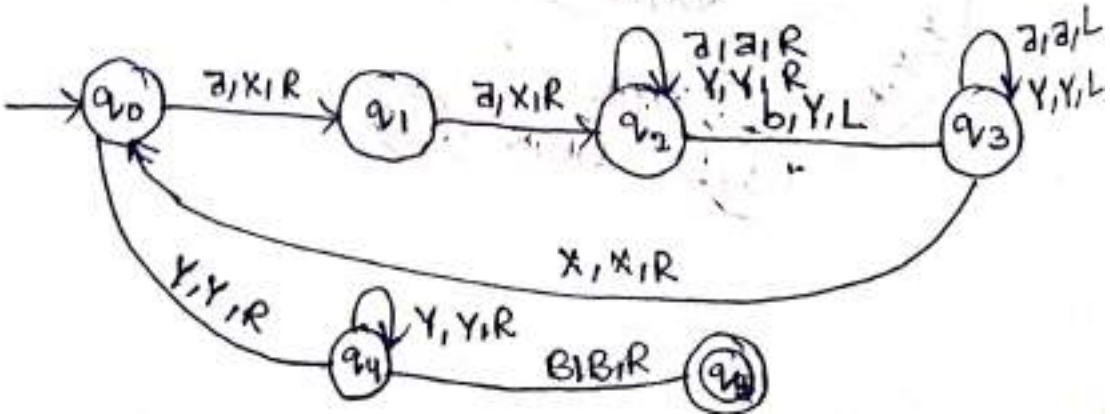
$L = \{a^n b^n / n \geq 1\}$.

A:- $L = \{ab, aabb, aaabbb, \ldots\}$

Let "aaaabb" be the input tape and q_0 be the initial state. [∵ For 2 a's we need to change to X].

		X	X	X	X	Y	Y		
B	B	a	a	a	a	b	b	B	B
		↑	↑	↑	↑				





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

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

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

$\Gamma = \{a, b, X, Y, B\}$

$F = \{q_5\}$

Transistion Table:-

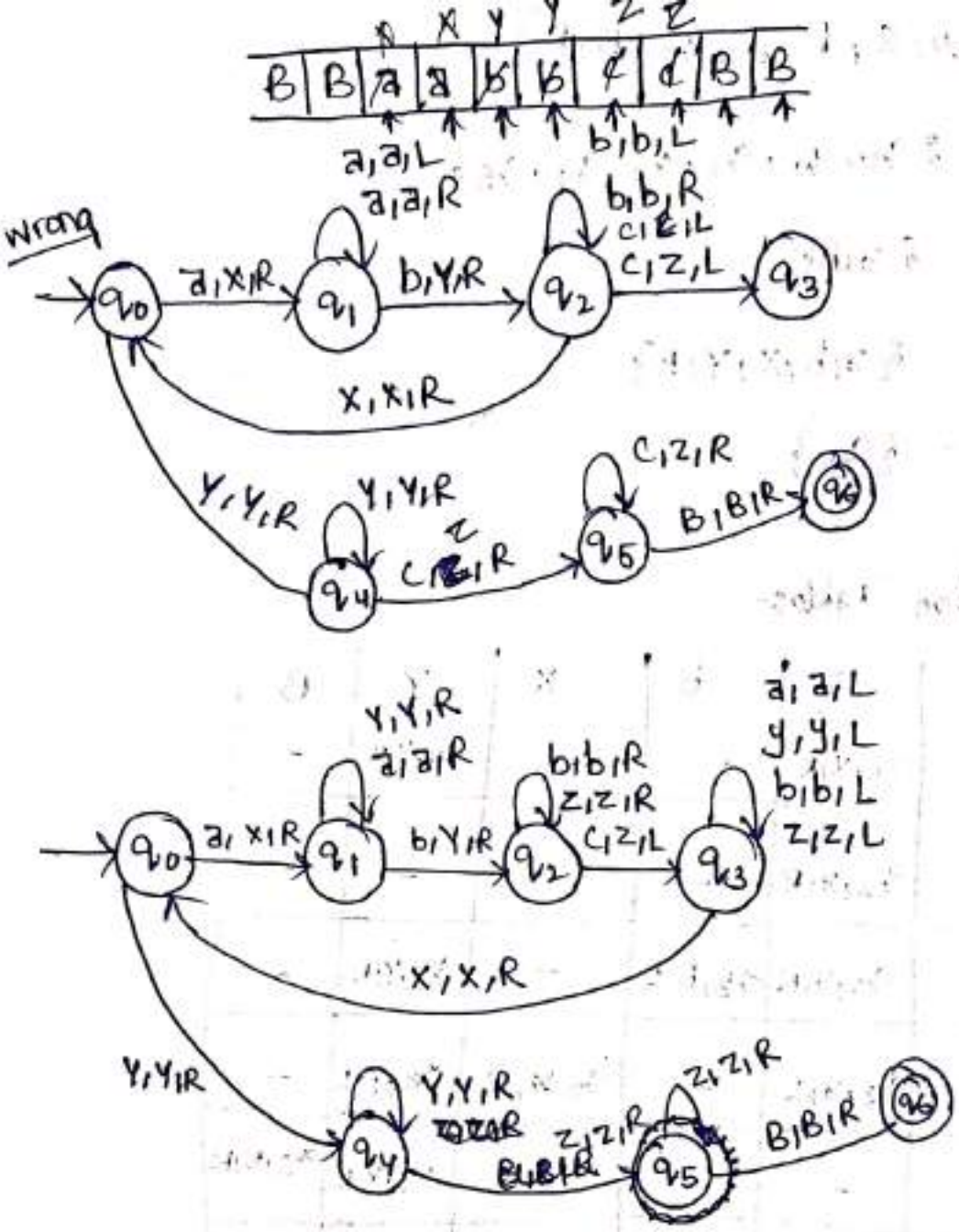
	a	b	X	Y	B
q_0	q_1, X, R	-	-	q_4, Y, R	-
q_1	q_2, X, R	-	-	-	-
q_2	q_2, a, R	q_3, b, L	-	q_2, Y, R	-
q_3	q_3, a, L	-	q_0, X, R	q_3, Y, L	-
q_4	-	-	-	q_4, Y, R	q_5, B, R
q_5	-	-	-	-	-

(3) Design Turing machine for the given language

$L = \{a^n b^n c^n / n > 0\}$.

A:- $L = \{abc, aabbcc, aaabbbccc, \ldots\}$

let "aabbcc" be the Input tape and q_0 be the initial state.



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

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

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

$\Gamma = \{a, b, c, x, y, B\}$

$F = \{q_6\}$

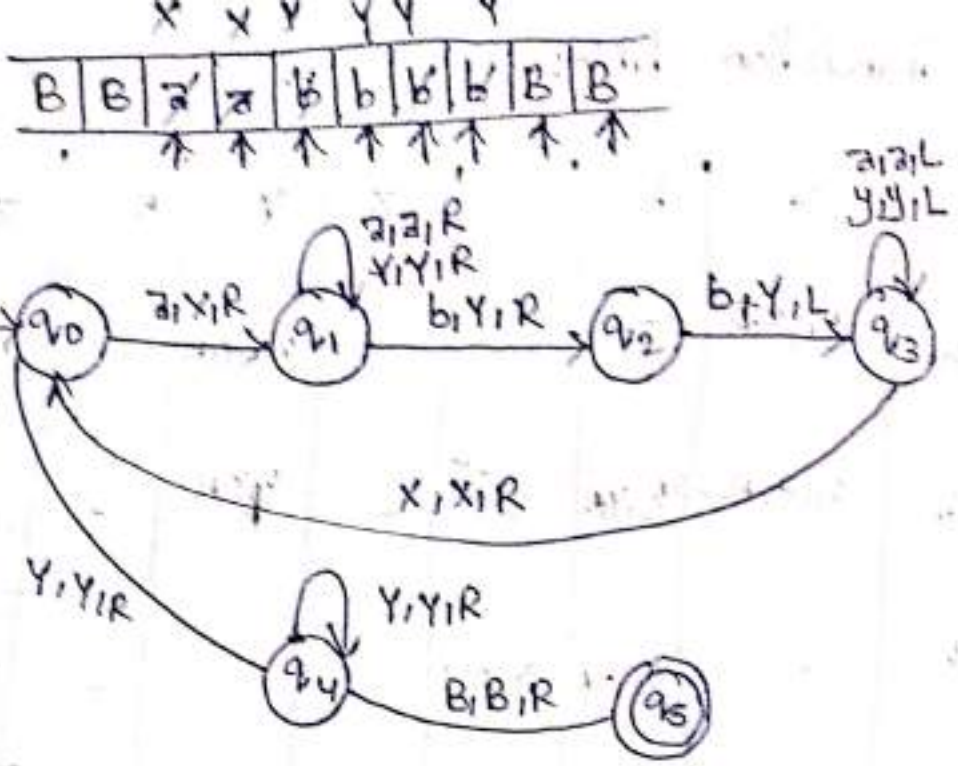
Transition table:

	a	b	c	X	Y	B	Z
q_0	q_1, X, R	-	-	-	q_4, Y, R	-	
q_1	q_1, a, R	q_3, Y, R	-	-	$q_1 Y, R$	-	-
q_2	-	q_2, b, R	q_2, Z, L	-	-	-	q_2, Z, R
q_3	q_3, a, L	q_3, b, L	-	q_0, X, R	q_2, Y, L	-	q_3, Z, L
q_4	-	-	-	-	q_4, Y, R	-	q_5, Z, R
q_5	-	-	-	-	-	q_6, B, R	q_5, Z, R
q_6	-	-	-	-	-	-	-

(4) Design turing machine for the language $L = \{a^n b^{2n} / n \geq 1\}$

A:- $L = \{abb, aabbbb, aaabbbbbb, \cdots\}$

let "aabbbb" be the input tape and q_0 be the initial state [∵ for 2 b's we need to change to Y]



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

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

$\Gamma = \{a, b, X, Y, B\}$

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

$F = \{q_5\}$

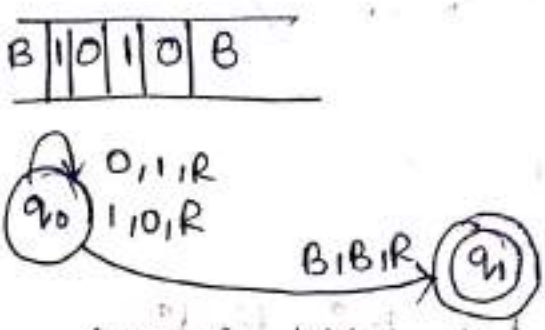
Transition Table:

	a	b	X	Y	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, Y, L	–	–	–
q_3	q_3, a, L	–	q_0, X, R	q_3, Y, L	–
q_4	–	–	–	q_4, Y, R	q_5, B, R
q_5	–	–	–	–	–

(5) Construct Turing machine for

i) Ones complement ii) Two's complement.

A:- i) One's complement



⇒ Two's complement

1 0 1 0 1

0 1 0 1 0

1

0 1 0 1 1

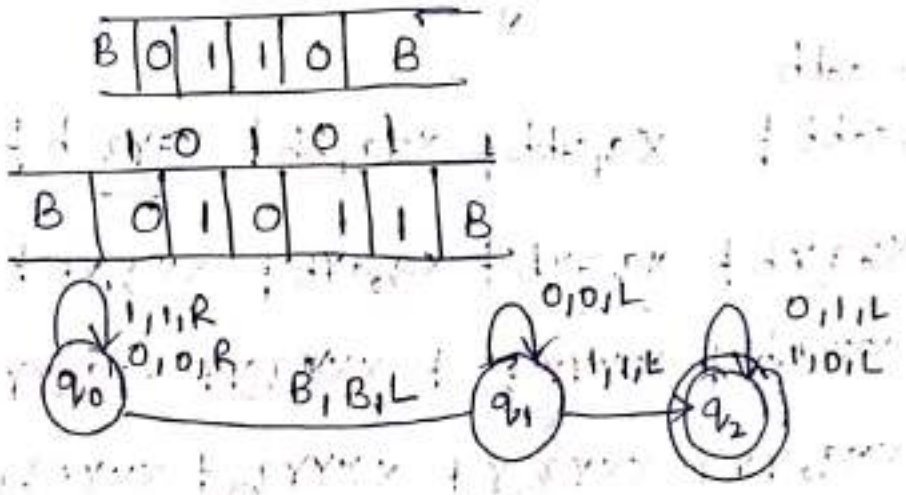
ii) Two's complement

1 0 1 0

0 1 0 1

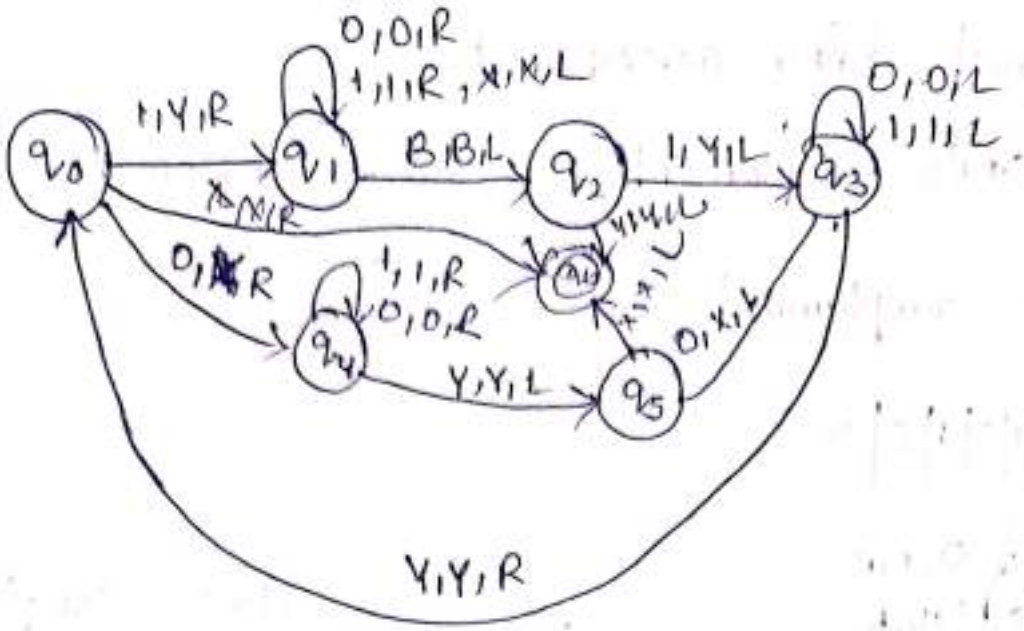
1 1

0 1 1 0



(6) Construct Turing machine for the palindrome of the 0's and 1's.

010, 000, 0101, 01, 111



Instantaneous Description For Turing Machine:-

Let "aaabbb" be the input string/tape.

_____

_____

Acceptance For the Turing machine:

$w = aabb$

$q_0aabb \vdash Xq_1abb \vdash Xaq_1bb \vdash Xaq_2Yb \vdash$

$Xaq_2Yb \vdash Xq_2aYb \vdash Xq_0aYb \vdash XXq_1Yb \vdash$

$XXYq_1b \vdash XXYYq_2 \vdash XXYq_2Y \vdash XXq_2YY \vdash$

$XXq_0YY \vdash XXYq_3Y \vdash XXYYq_3 \vdash XXYYq_4$

∴ The Acceptance of Turing machine is verified.

Language of Turing machine:

Language of Turing machine M is denoted by L(M) and it is defined as

$$L(M) = \{ w \text{ in } \Sigma^* / q_0 w \vdash^* \alpha p \beta \}$$

$\Rightarrow \alpha, \beta$ are any strings of tape symbols

$\Rightarrow$ P can be any state.

Undecidability :-

Recursive language: A language, if their exists a turing machine, That accepts all strings in L and rejects all strings not in L.

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

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

Let "abab" be the input tape symbol, if we find the Acceptance for this, this is not present in the language then it is a "recursive language".

Recursively Enaumerable language:- If the A language, if their easits a turing machine that accepts all strings in L that may or may not rejects all strings in the L.

→ These are said to be partially decidable.

(→ The)

Decidability:-

The recursive languages are said to be decidable.

Undecidability:

A language which is not decidable is said to be undecidable.

→ There is no turing machine for the undecidable languages.

→ The languages which are not recursively enumerable.

Language that is not recursively Ennumerable:-

1) In order to prove that not a recursively enumerable we need to

i) Ennumerating binary strings:- In order to prove this we need to write "binary strings"

{ε, 0, 1, 00, 01, 10, 11, 000, 001, ...}

Here "ε" is the "0th" string

"0" is the "1st" string and so on.

Constructing codes for turing machine: Transistion

→ Assign positive integers to the symbols of in turing machine.

→ For input symbols, Assign the / Take the $x_1(0)$, $x_2(1)$ and $x_3(B)$.

→ For directions, Take the $D_1(L)$, $D_2(R)$.

Diagnolized Language: It is represented by "L_d".

It is a set of strings "w_i" and "w_i" is not in "Turing machine (or) $L(M_i)$" for some "M_i".

Table of Acceptance: We have write "0 and 1" in the "acceptance table. Rows" we take Turing machine and for columns we take strings. The strings must be accepted by the turing machine.

→ If turing machine accepts take "1", if not accepts we takes "0".

→ $L(M_i)$ will have the diagonal of w_i and M_i.

Proof of the language that is not recursively enumerable (Diagnolized Language):-

let L_d be $L(M_i)$ for some M_i.

1) If w_i is in L_d, then w_i is in $L(M_i)$.
But, by the definition L_d, w_i is not in the $L(M_i)$ L_d.

2) If w_i is not there in L_d, then it is not there in $L(M_i)$.

But by the definition of L_d, w_i is in the L_d.

∴ There is no turing machine for the L_d.

So that we can have turing machine for L_d, which means L_d is not recursively enumerable language. This proof is called as "proof by contradiction". This means: Firstly we assume that there will be the L_d later that we assume there will not have L_d.

Undecidable problem that is not recursively Enumerable:

Complements of recursive and recursively enumerable language:

Theorem 1: If L is recursive then, $\bar{L}$ is also recursive.

proof:- Let "L" be a recursive language.

So, "L" halts always for Turing machine "M".

→ Let "M'" be the Turing machine of "$\bar{L}$".

To get changes: To get "M'" do some changes:

→ Make all accepting states as non-accepting states.

→ Take a new state as a accepting state and write translations from each non-accepting state of "M" on each tape symbol to the accepting state.

The Turing machine "M'" accepts strings which are not accepted by "M" and rejects all strings which are accepted by "M".

That means, the Turing machine "M'" always halts.

∴ "$\bar{L}$" is recursive.

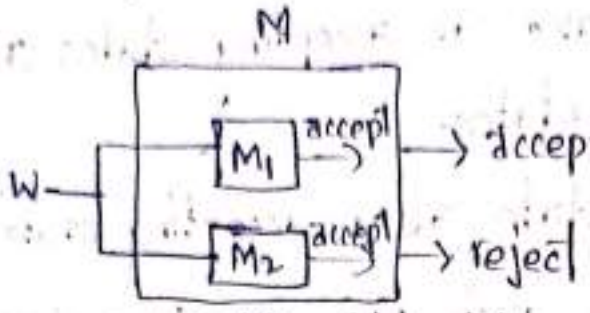
Theorem:- If "L" and "$\bar{L}$" are recursively enumerable then "L" is recursive.

Proof:- Take a Turing Machine "M".

(Let "L" be recursive)

Let "M_1" be the Turing machine for "L" and "M_2" be the Turing machine for "$\bar{L}$".

we can have a Turing machie 'M' by taking M_1 & M_2 as tapes.



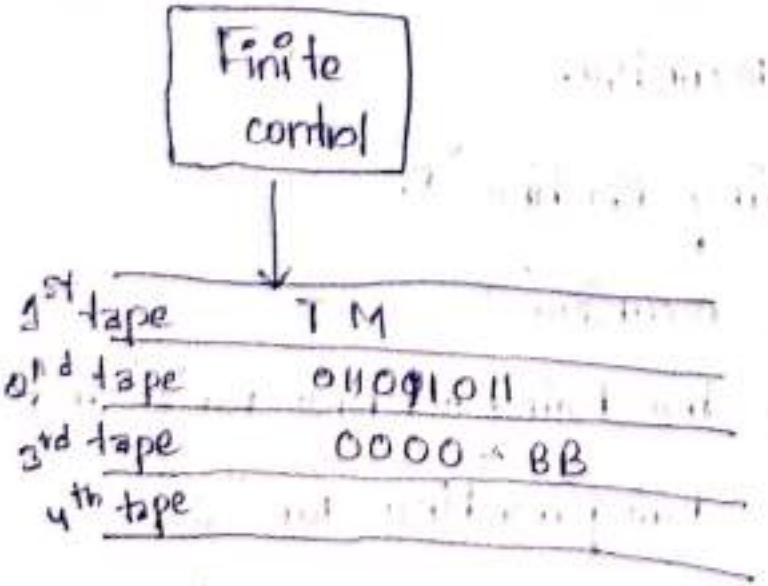
"M_1" acts same as "M"

Since "M" always halts

∴ L is recursive

Universal language:-

It is a language of strings representing turing machine and an input that is accepted by the turing machine.



→ 1st tape holds TM, w

→ 2nd tape holds simulated code of TM

→ 3rd tape holds the state q_i in oform.

→ 4th tape is used to do scratch.

→ It is represented by the "d".

(1) Examine input to ensure that the code of Turing machine is legitimate.

(2) place input in 2nd tape in encoded form i.e, place 10 for each '0', 100 for each 1 and 1000 for each B.

(3) place start state (0) in tape 3, and also place tape head at 1st symbol of simulated in 2nd tape.

(4) for each move of TM of the form

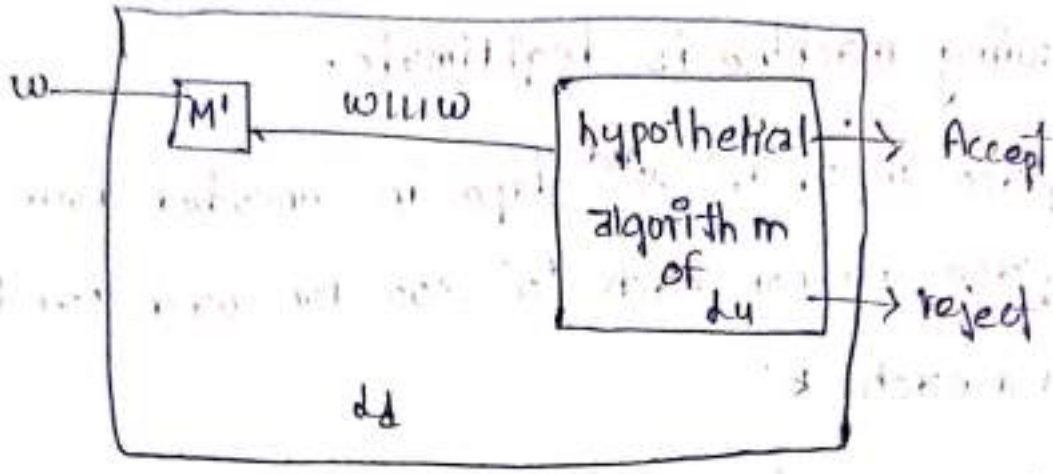
$$\delta(q_i, x_j) = (q_k, x_l, Dm)$$

a) replace the contents 0^j with 0^l

b) change the content of tape 3 by places 0^k and make all previous 0's 10's.

Theorem: If L_u is RE, then L_u is not recursive

Proof: Let $\bar{L}_u$ is Recursively Ennumerable, then there exists a Turing Machine M.



If there exist TM for $\bar{L}_u$ then TM can also possible for L_d.

As per the defination of L_d, there exist no TM for L_d.

So that there exist no TM fo $\bar{L}_u$ i.e. L_u is non recu recursively ennumerable.

So that L_u is not recursive.

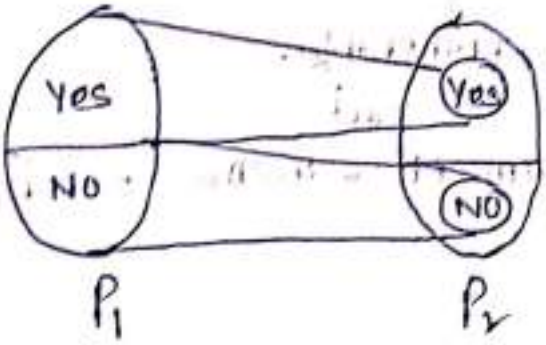
Undecidability about Turing machine:

Turing Reducability: It is a technique to convert one language into another language, when L_1 is recursive in L_2

→ In general an instance of, If there is an algorithm to convert P_1 to P_2 then we can say that an arbitrary instance of P_1 is reduced to an instance of P_2 with the same answer.

→ If P_1 is (recursively enumerable) is not recursive then P_2 is not recursive.

→ If P_2 is not recursive then P_1 can't be recursive.



Reduction turns the positive.

Reduction from P_1 to P_2 is TM which take an instance of P_1 and accepts instance of P_2.

In General, The reduction from P_1 to P_2 is program which states instance of P_2 as Output.

Theorems:- If there is reduction from P_1 to P_2

a) If P_1 is undecidable, then P_2 is also undecidable

Proof:-

Let P_1 is undecidable and then exists an algorithm to solve P_2

So, P_2 is decidable, but P_1 is reduced to P_2

Since, there is reduction from P_1 to P_2 and P_1 is undecidable.

So that P_2 is also undecidable.

b) If P_1 is non-recursively enumerable then P_2 is also non-recursively enumerable.

Proof:- Let P_1 be form RE and there exists an algorithm for P_2

Since there is a reduction from P_1 to P_2 and P_1 is noth RE P_2 is non RE.

Turing Machine for empty language:

L_e → empty language

L_{ne} → non-empty language

$L_e = \{M / L(M) \neq 0\}$

$L_{ne} = \{M / L(M) \neq 0\}$

→ L_e and L_{ne} are complement to each other.

→ L_{ne} is recursively enumerable, and L_{ne} is not recursive.

→ L_e is not recursively enumerable.

Rice Theorem and Property of RE Languages :-

Property of RE Languages:- It is a set of RE languages. There are two properties

① Trivial Property of RE Language: A property said to be Trivial, when it is has no strings or when it is empty or a set of all RE languages.

② Non-Trivial Property of RE Languages: A property said to be nonTrivial, when it is a non-empty nor a set of all RE languages.

Rice Theorem:-

For Every non-Trivial property of a recursively enumerable language is undecidable.

Implementation of Turing machine Specifications:-

Applications of Rice Theorem:-

① The Rice theorem is used to say the given problems are undecidable or not.

→ There are four problems

① whether the language accepted by TM is empty.

② whether the language accepted by TM is finite.

③ whether the language accepted by TM is regular.

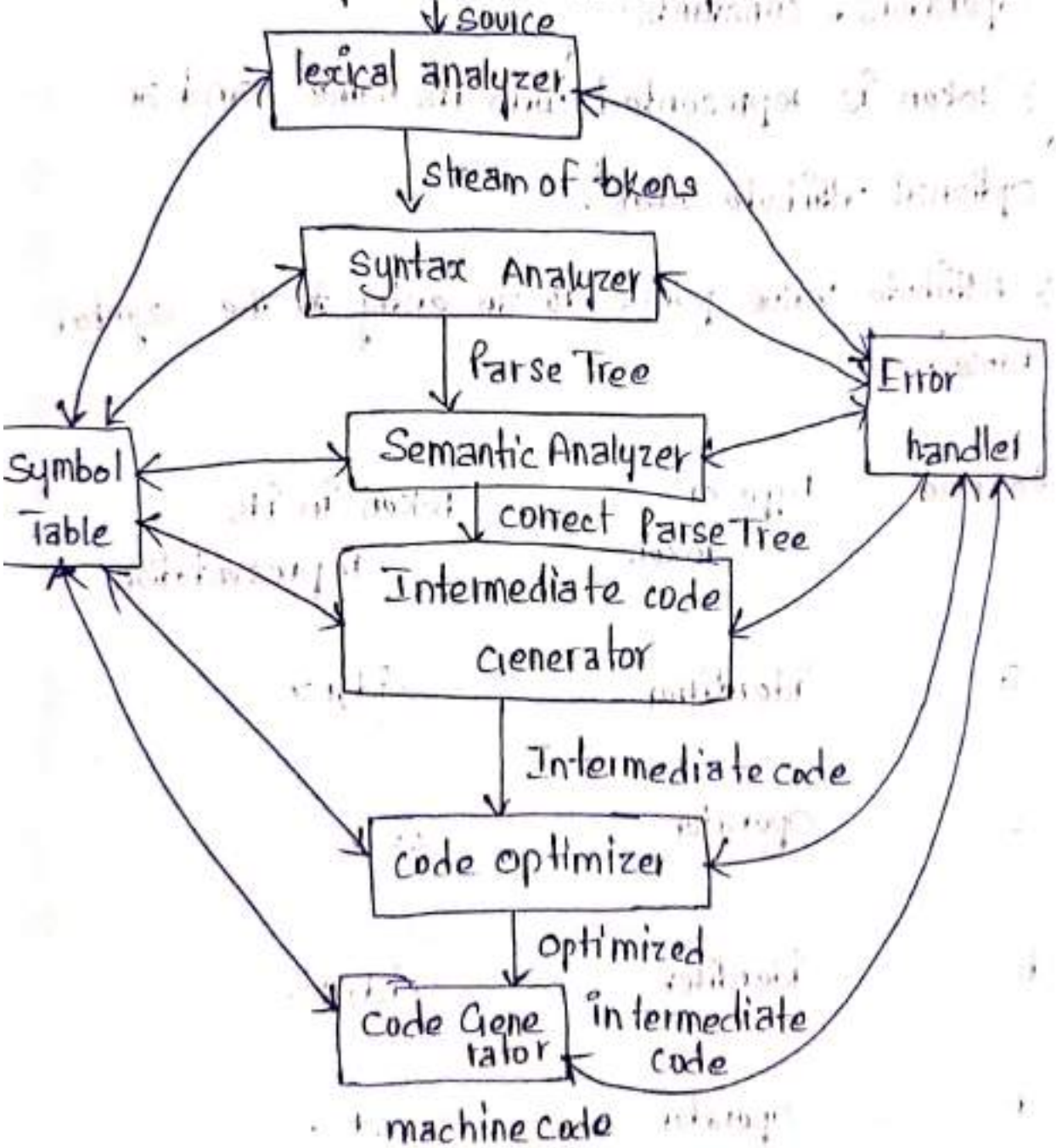
④ whether the language accepted by TM is context free Grammer or not.

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 meaniqful 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.

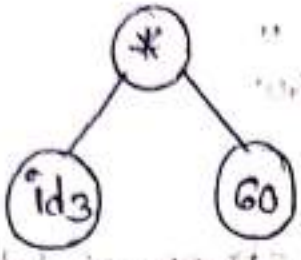
Stream of tokens ⇒ "id1 = id2 + id3 * 60"

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

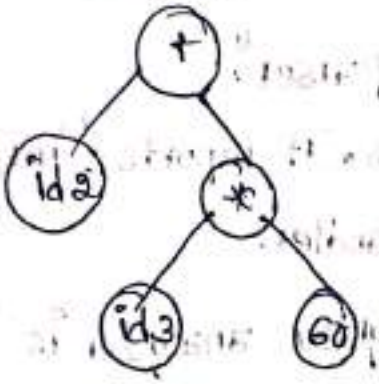
→ In parse tree take operators as "internal nodes" and "root" node.

→ It take identifiers, constants as "leafnodes".

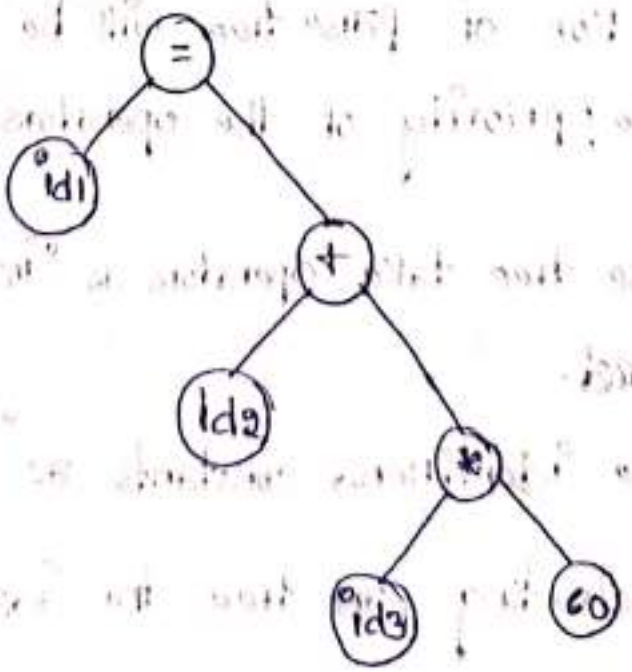
Step1: constructing paise tree for id3 * 60



step2: Next we will construct the parse tree for the id2.



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



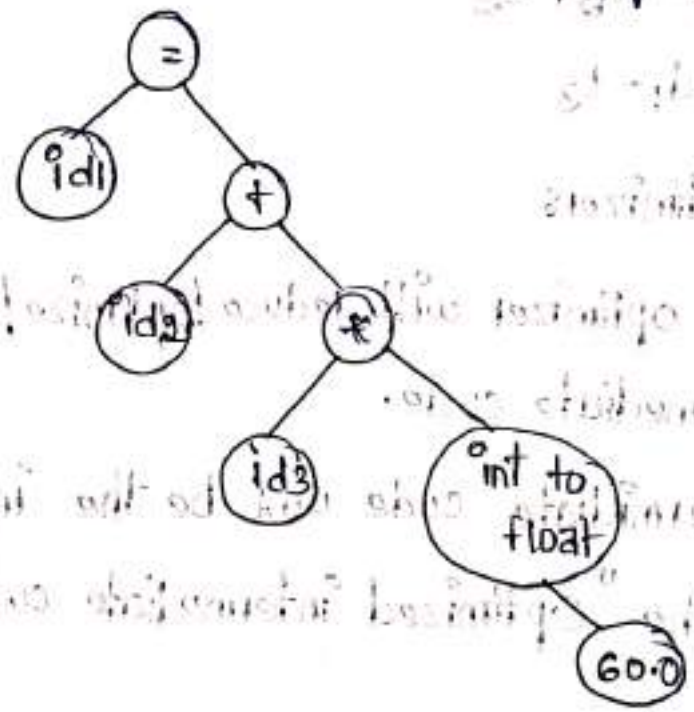
→ It is constructed from ~~top~~ down to top.

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: $t_1 = id_3 * 60.0$

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"

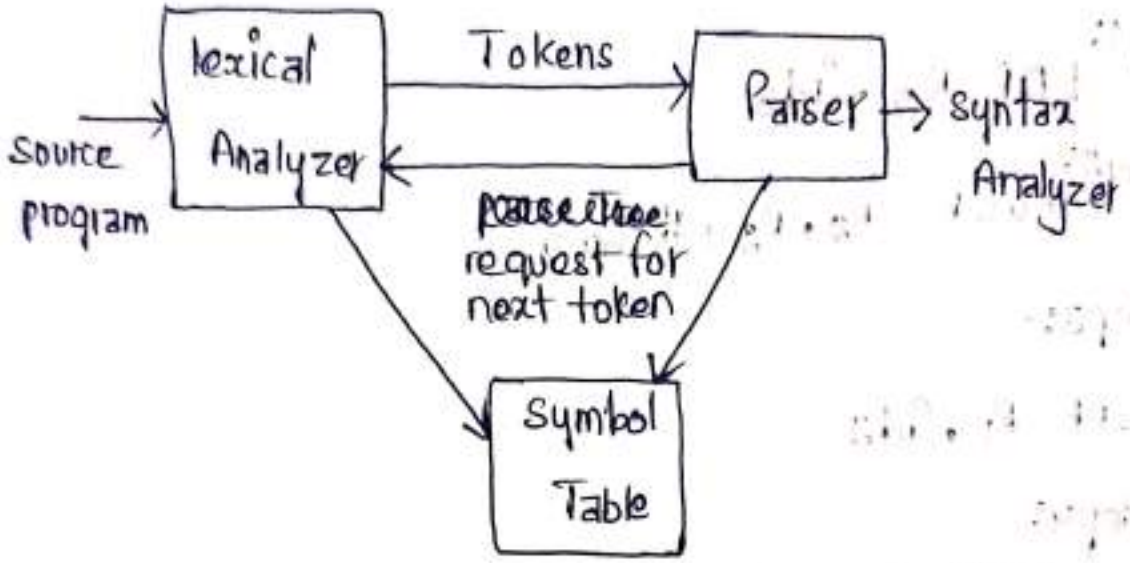
The role of lexical Analyzer:-

→ 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 Complier 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 compilier.

Tokens, Patterns, lexemes:-

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

→ Token is a pair of token name, and its 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 beginning 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 whether the.

(1) 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]

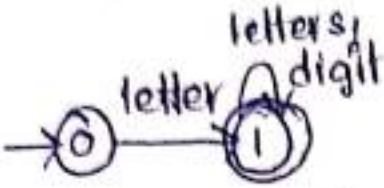
→ regular expression for id → letter (. letter | digit)*

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

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

Transistion diagram for Identifiers:

Ex:- abc12

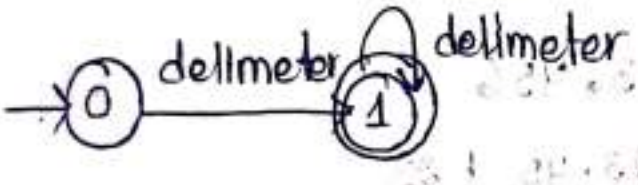


Transistion diagram for delemeter:-

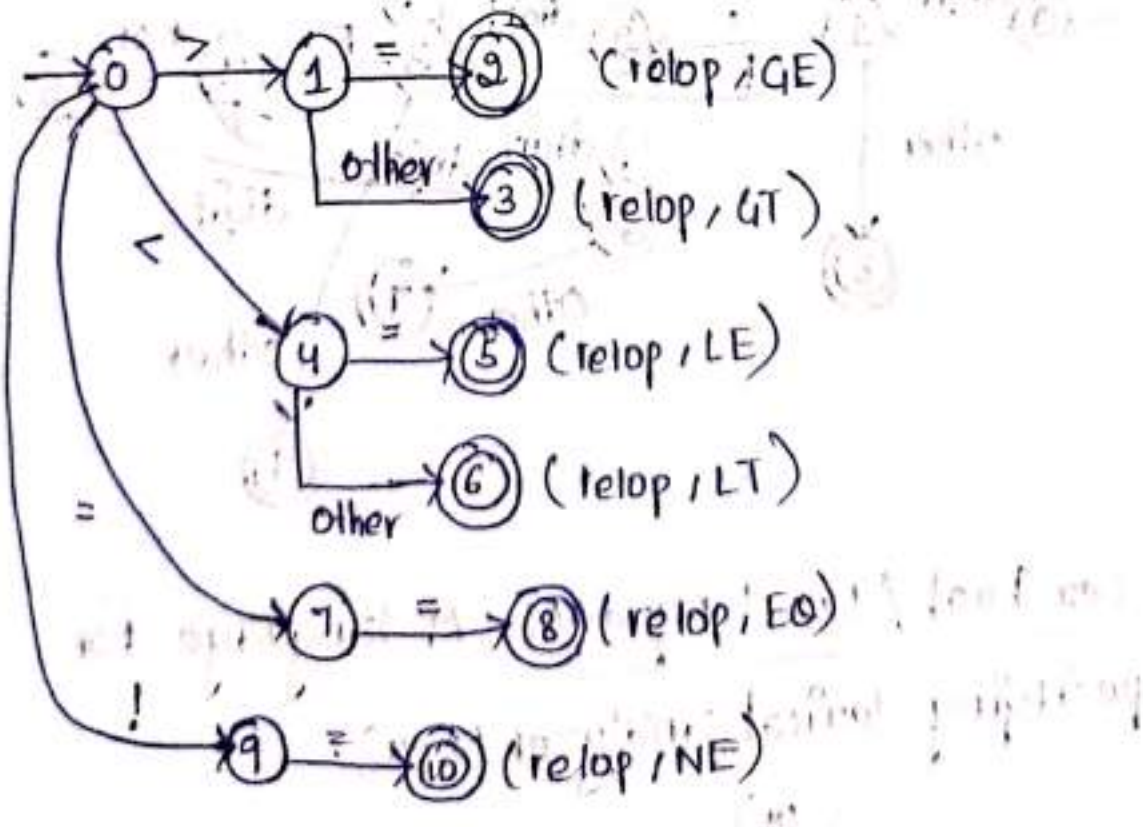
↳ It may be a blank space, newline character, tap space.

Regular expression:-

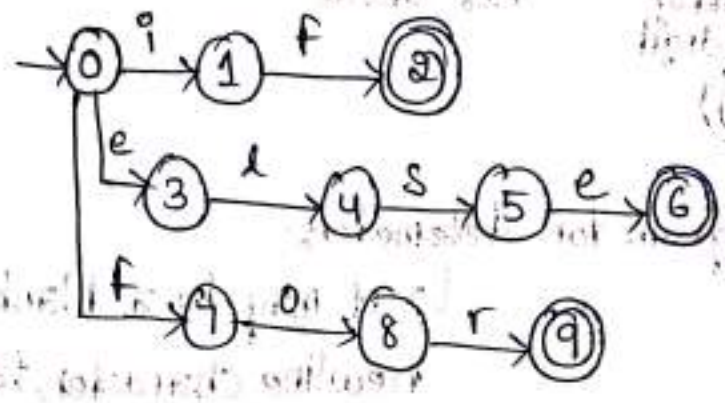
ws → delimeter (delimeter)*



Transistion diagram for Relational Operators:



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

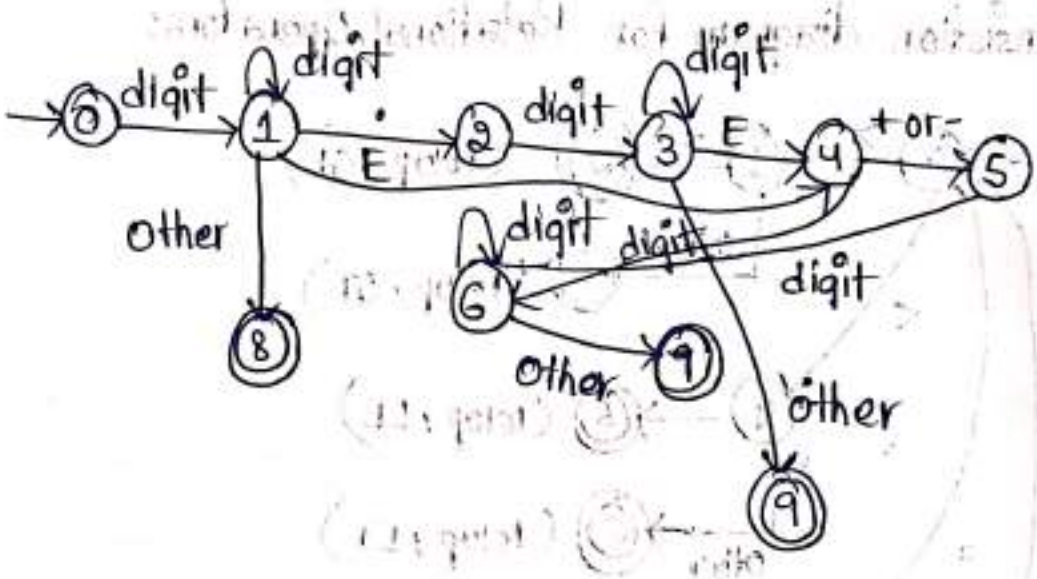


Regular expression for digits:

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).

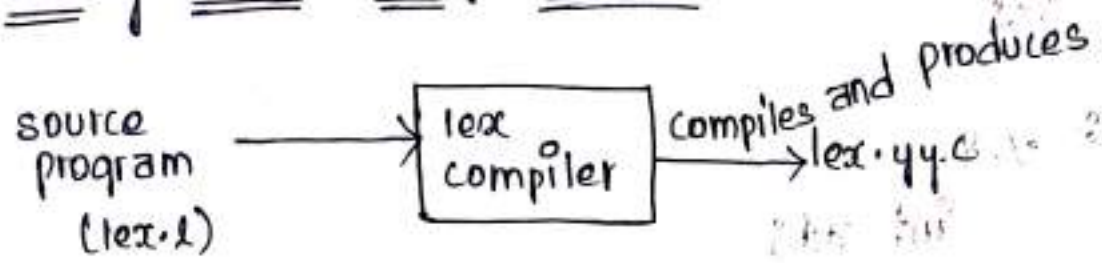
(or)

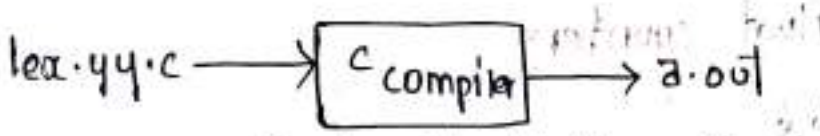
The lexical analyzer generater (Lex).

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

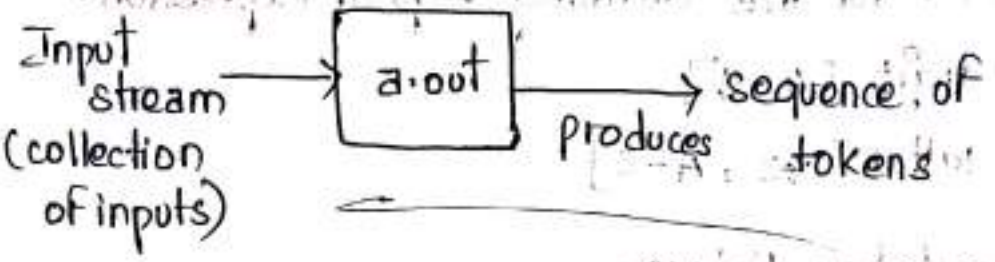
→ It specifies the RE, RE are used to represent the patterns for the tokens.

Creating lexical Analyzer with lex:-





Produce object / binary file



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 ovred]

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 Auxilary 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 less than 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:

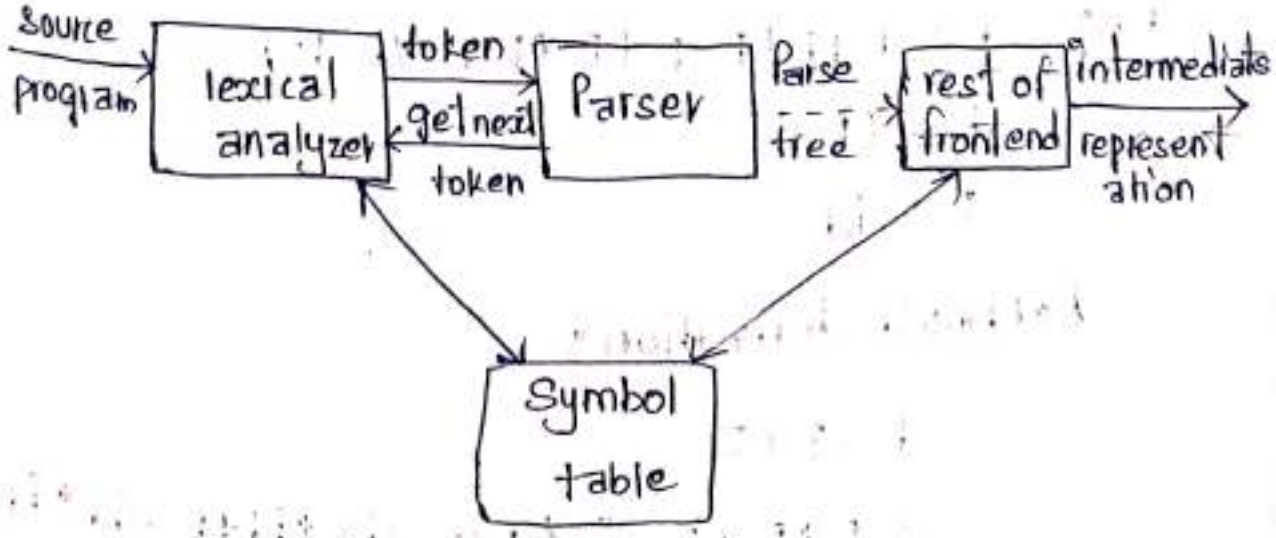
→ Parser of a grammer that takes as input string of stream of tokens from the lexical analyzer and produces output as parse tree.

→ 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-termial
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".

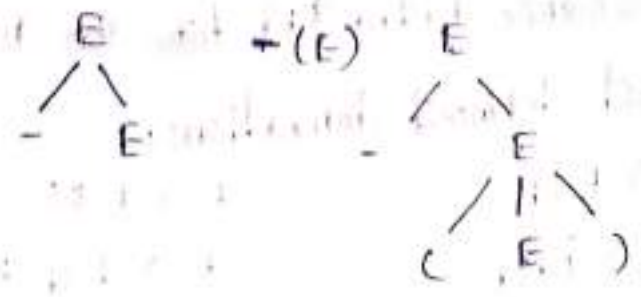
Parse Tree:

→ Inner nodes of a parse tree are non-terminal symbols

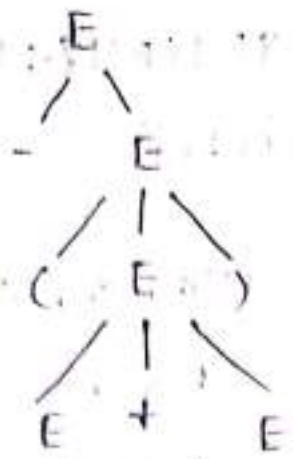
→ The leaves of a parse tree are terminal symbols.

→ A parse tree can be seen as a graphical representation of a derivation.

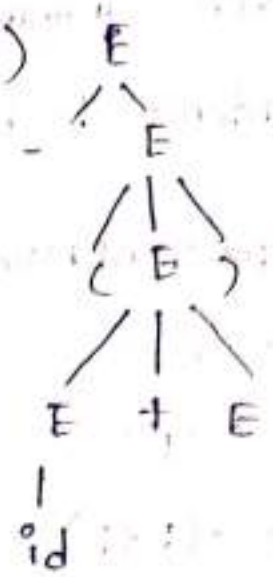
Ex:- E → -E | E → -(E) | E → -(E+E) | E → -(id+E) | E → -(id+id)



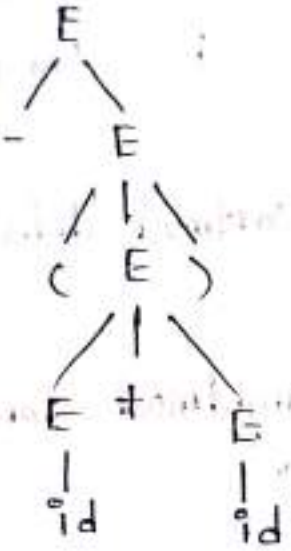
-(E+E)



-(id+E)



-(id+id)



Ambiguity:

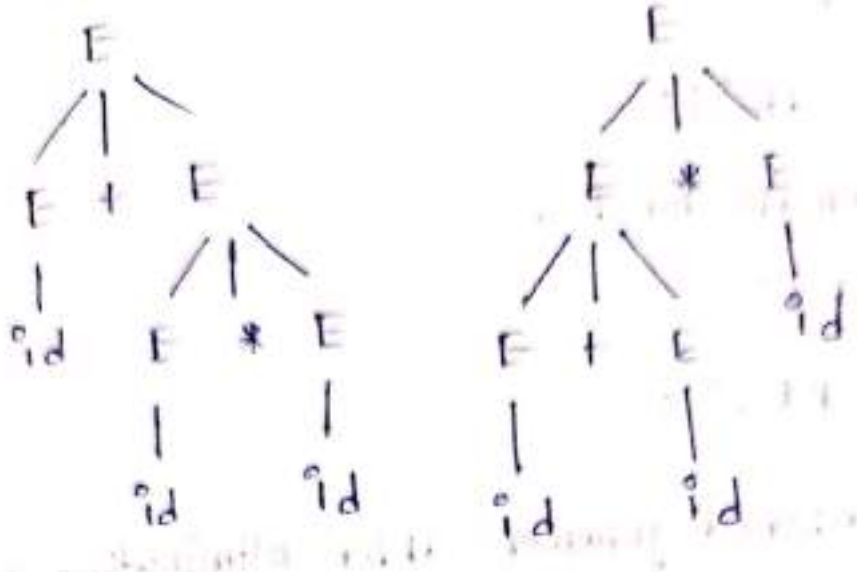
A grammer that produces more than one parse for some sentence is called "ambiguos grammer."

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

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

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

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

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 $X \to Y_1 Y_2 \ldots Y_k$ 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, $Y_1, \ldots Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$ 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'→e			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 ambigvos 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	$(A \rightarrow b)$	$S \rightarrow aABe$
aAbcde	$(A \rightarrow Abc)$	$\rightarrow aAde$
aAde	$(B \rightarrow d)$	$\rightarrow aAbcde$
aABe	$(S \rightarrow aABe)$	$\rightarrow abbcde$

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
$id_1	$+ id_2 * id_3 \$$	reduce by $E \to id$
$E	$+ id_2 * id_3 \$$	shift
$E+	$id_2 * id_3 \$$	shift
$E+id_2	$* id_3 \$$	reduce by $E \to id$
$E+E	$* id_3 \$$	shift
$E+E*	$id_3 \$$	shift
$E+E*id_3	$\$$	reduce by $E \to id$
$E+E*E	$\$$	reduce by $E \to E*E$
$E+E	$\$$	reduce by $E \to E+E$
$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.

Examples:-

Consider the Grammer:

$E \to E+E \mid E*E \mid id$ and input id+id*id.

stack	Input	Action	Stack	Input	Action
$E+E	*id$	Reduce by $E \to E+E$	$E+E	*id$	shift
$E	*id$	Shift	$E+E*	id$	shift
$E*	id$	shift	$E+E*id	$	Reduce by $E \to id$
$E*id	$	Reduce by $E \to id$	$E+E*E	$	Reduce by $E \to E*E$
$E*E	$	Reduce by $E \to E*E$	$E+E	$	Reduce by $E \to E*E$
$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	$				

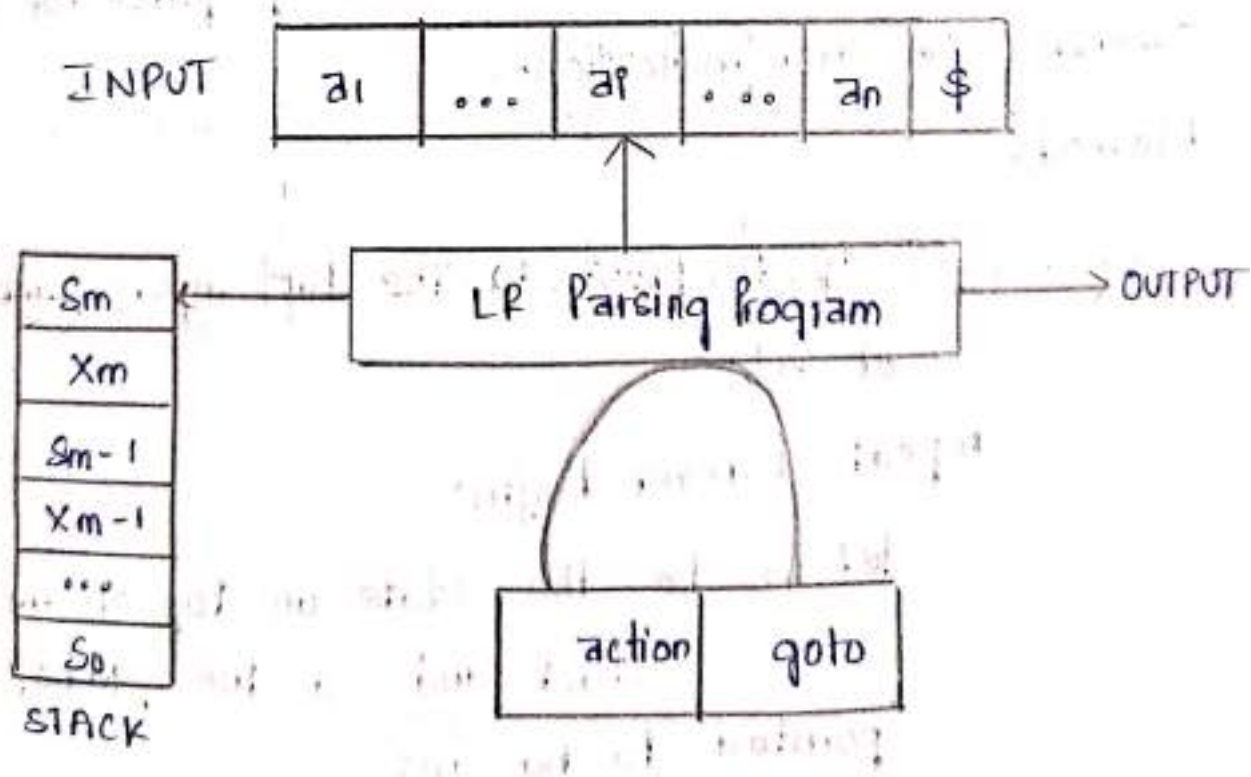
LR Parsers:-

→ 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 $s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$.

→ 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 stacks.
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 3: 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_9: $E \to T.$

$T \to T.*F$

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) = S5; shift
0 id 5	+id*id$	GOTO(I_5, +) = r6; reduce by F→id
0 F 3	+id*id$	GOTO(I_0, F) = 3 GOTO(I_2, +) = r4; reduce by T→F
0 T 2	+id*id$	GOTO(I_0, T) = 1 GOTO(I_1, +) = S6; shift
0 E 1	+id*id$	GOTO(I_6, id) = S5; shift
0 E 1 + 6	id*id$	GOTO(I_5, *) = r6; reduce by F→id
0 E 1 + 6 id 5	*id$	GOTO(I_6, F) = 3 GOTO(I_3, *) = r4; reduce by T→F
0 E 1 + 6 F 3	*id$	GOTO(I_6, T) = 9 GOTO(I_9, *) = S7; shift
0 E 1 + 6 T 9	*id$	GOTO(I_7, id) = S5; shift
0 E 1 + 6 T 9 * 7	id$	GOTO(I_5, $) = r6; reduce by F→id
0 E 1 + 6 T 9 * 7 id 5	$	GOTO(I_7, F) = 10 GOTO(I_{10}, $) = r3; reduce by T→T*F
0 E 1 + 6 T 9 * 7 F 10	$	GOTO(I_7, F) = 10 GOTO(I_9, $) = r3; reduce by T→T*F
0 E 1 + 6 T 9	$	GOTO(I_6, T) = 9 GOTO(I_9, $) = r1; reduce by E→E+T
0 E 1	$	GOTO(I_0, E) = 1 GOTO(I_1, $) = accept

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.

Inheited Attribute:

An inheited 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
$L \to En$	$L.Val = E.Val$
$E \to E_1 + T$	$E.Val = E_1.Val + T.Val$
$E \to T$	$E.Val = T.Val$
$T \to T_1 * F$	$T.Val = T.Val \times F.Val$
$T \to F$	$T.Val = F.Val$
$F \to (E)$	$F.Val = E.Val$
$F \to digit$	$F.Val = digit.lexval$

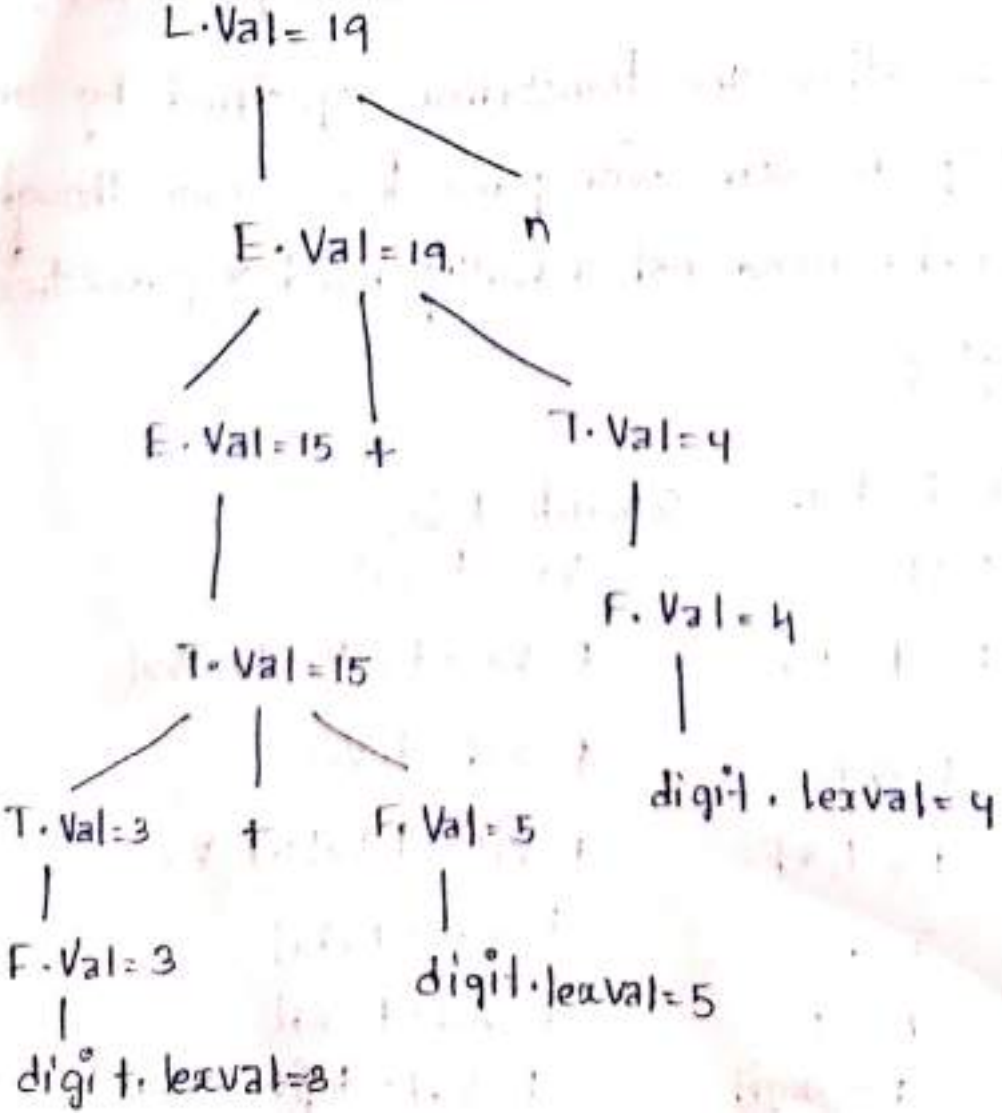
The SDD based for arithemetic expressions with operators "+" and "*". It evalutes expressions terminated by an endmarker n.

→ In the SDD, each of the non-terminals has a single synthesized attributed, 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



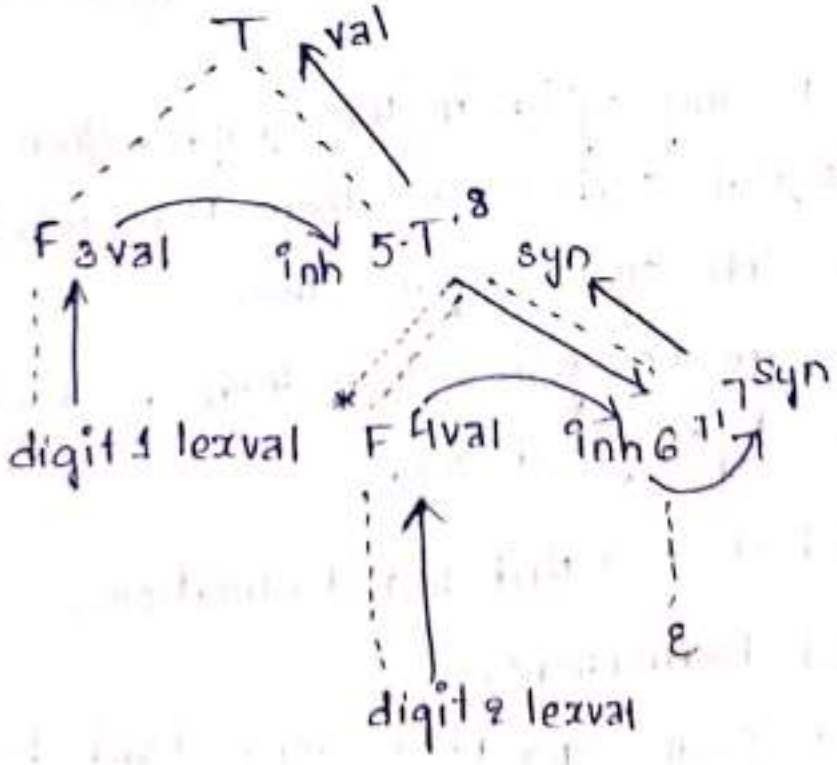
Evaluation Orders for SDD's:

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 tree.

→ 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 seqences 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 tree.

→ 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 $A \to X_1 X_2 \cdots X_n$ 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:

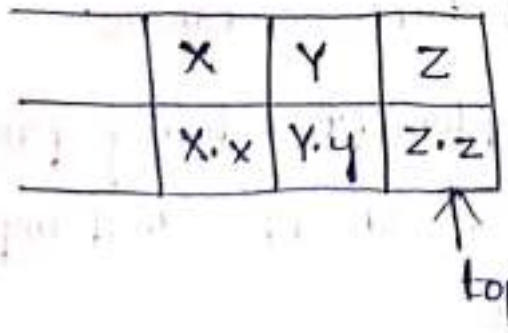
→ Simplest SDD implementation occurs when the grammar is passed bottom-up and the SDD is S-attributed.

→ 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

$A \rightarrow A\alpha | \beta$

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.

$A \rightarrow \beta R$

$R \rightarrow \alpha R | \varepsilon$

Example:- Consider the following E-productions from an SDT for translating infix expression into postfix notation:

$E \rightarrow E_1 + T$ { Print ('+'); }

$E \rightarrow T$

→ 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:

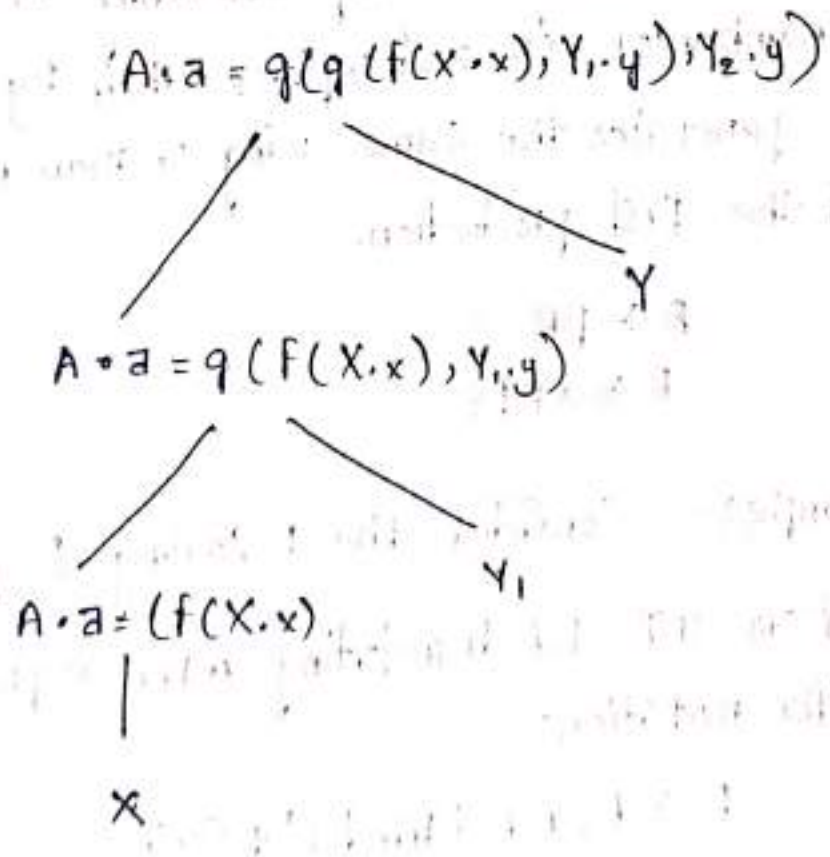
$E \rightarrow TR$

$R \rightarrow +T$ {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 themselves 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 LR 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 SDDs:-

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.

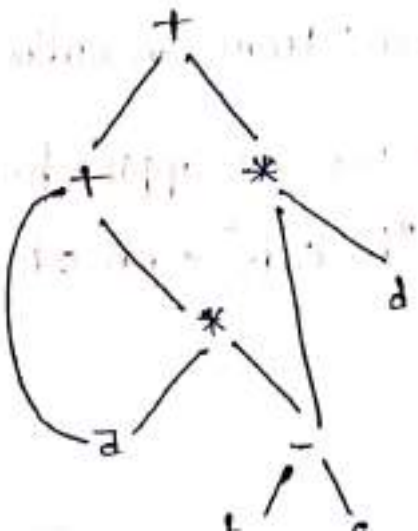
→ The difference is that a node N in a DAG has more than one parent if N represents a common subexpression.

→ In a syntax tree, the tree for the common subexpression would be replicted 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>.

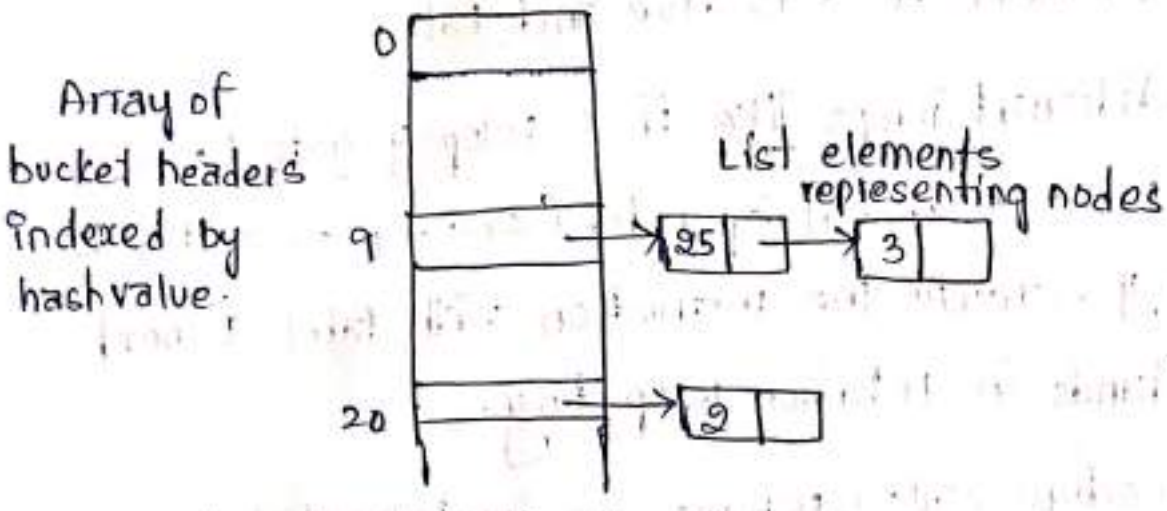
Method:

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 array holds expressions from an entire program.

To avoid this, we use an approach, i.e; hash table, nodes are kept 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

Quadruples:

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 follows:

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.

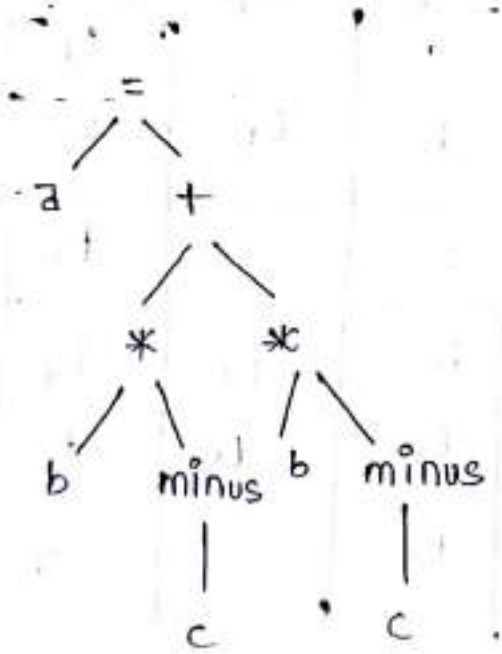
x op y by its position.

→ 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

$p_1 = a + b$
$q_1 = p_1 - c$
$p_2 = q_1 * d$
$p_3 = e - p_2$
$q_2 = p_3 + q_1$

(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 readAnay() { /* 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);
}
```

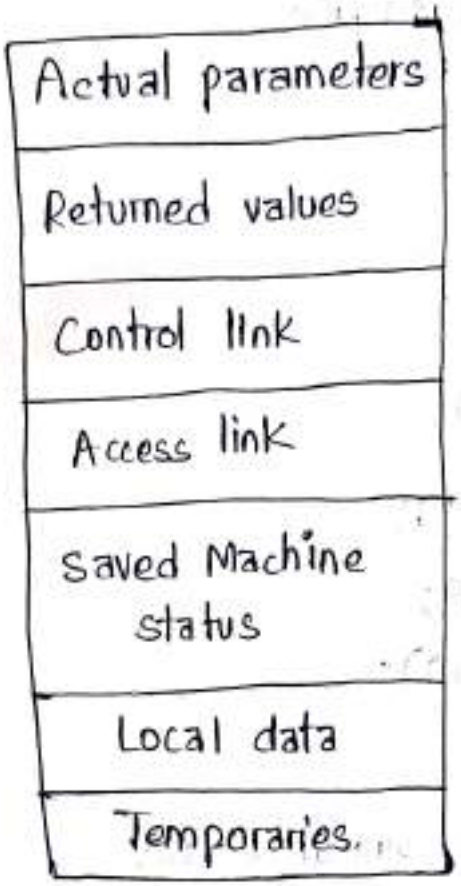
There are three common cases:

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 ... 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 record:

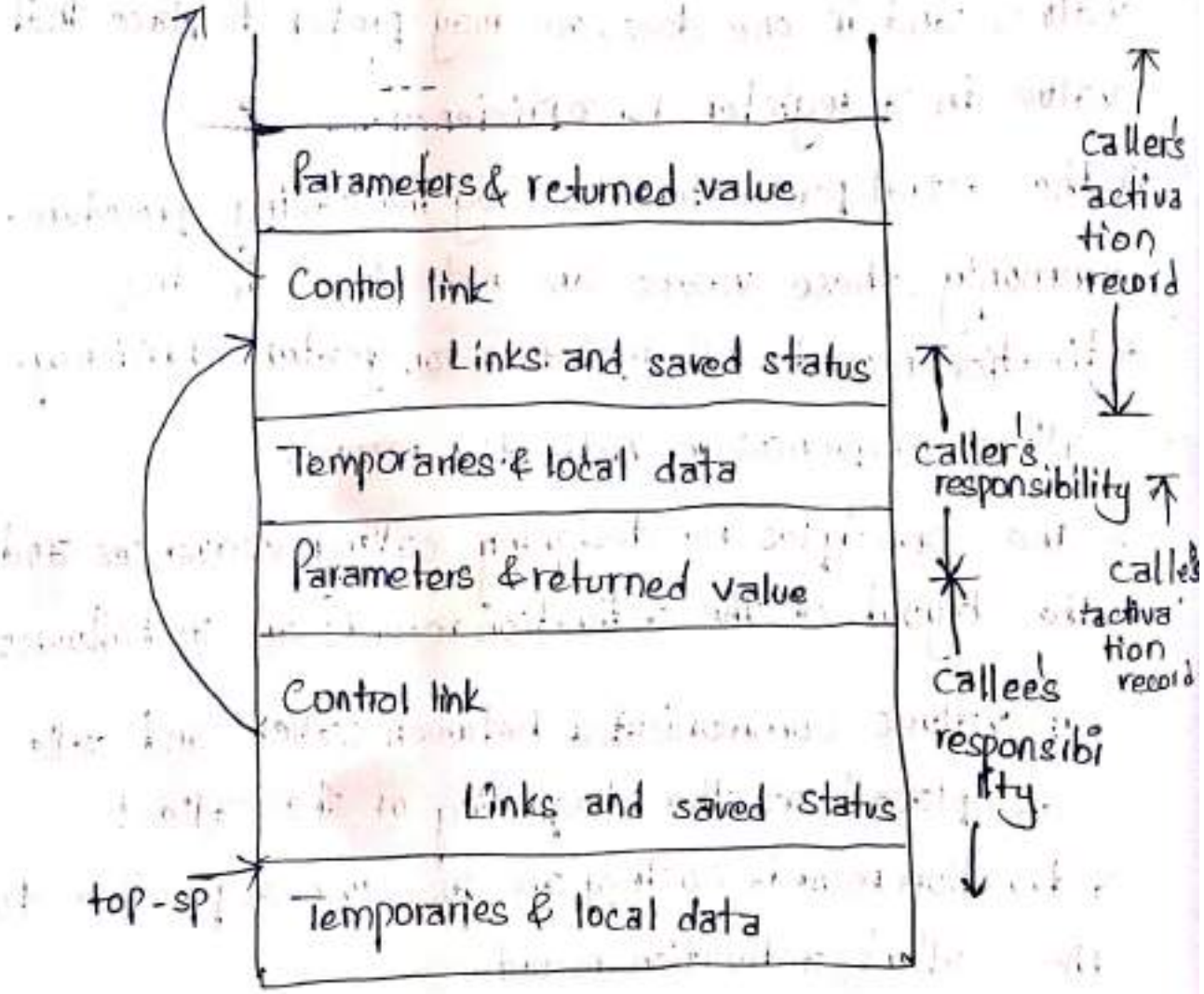
→ The principles for designing calling sequences and the layout of the activation records are as follows:

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) =
    val a = array (0,10); let
        fun readArray (inputFile) = a ... a ... ;
        fun exchange (i,j) = ... a ... ;
        fun quicksort (m,n) =
            val 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:-

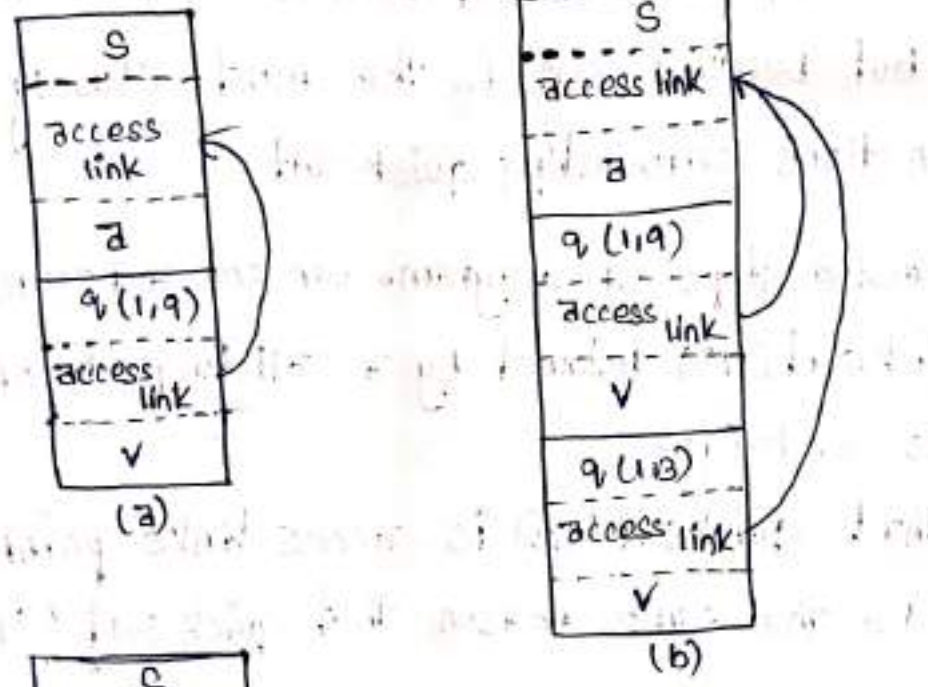
→ A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the access links to each activation record.

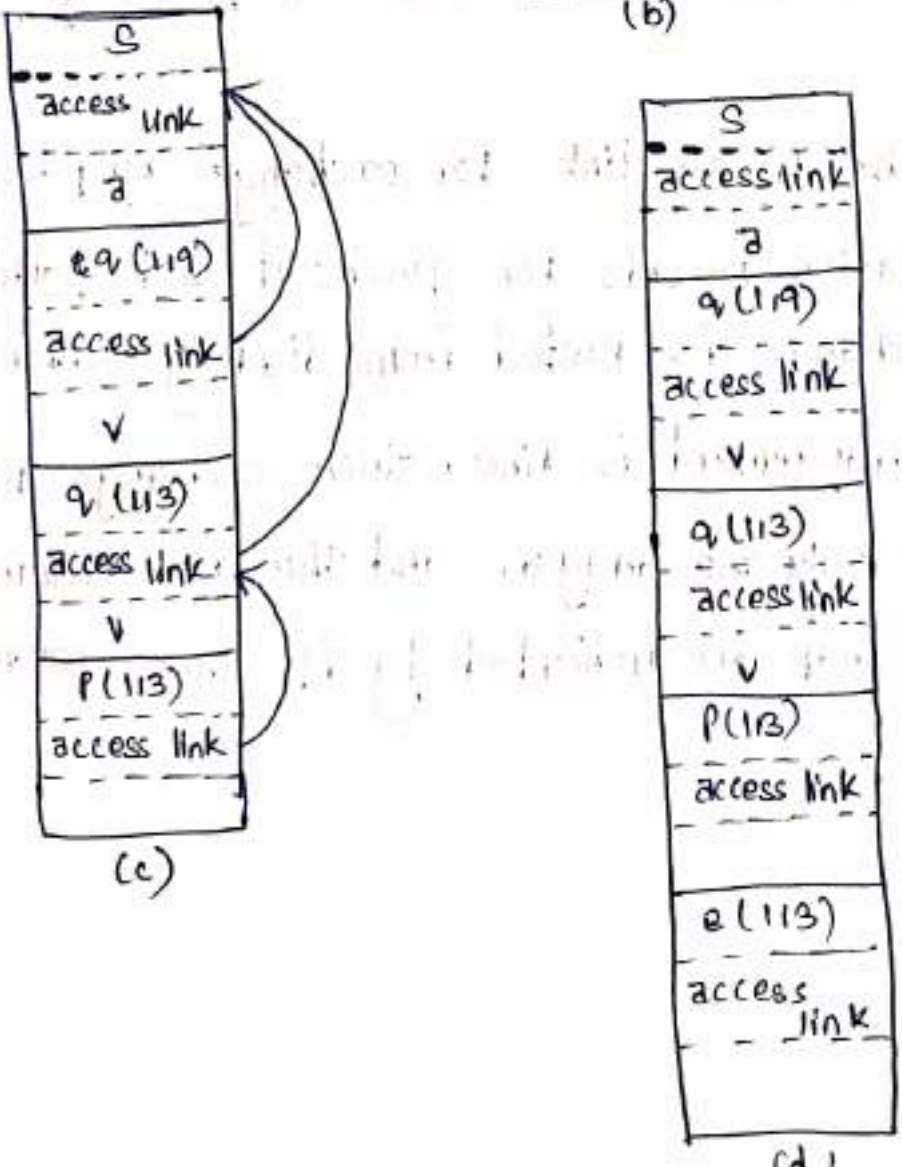
→ If the procedure p is nested immediately within procedure q in the source code, then the access link in any activation of p points to the most recent activation of q.

→ 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) is 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.

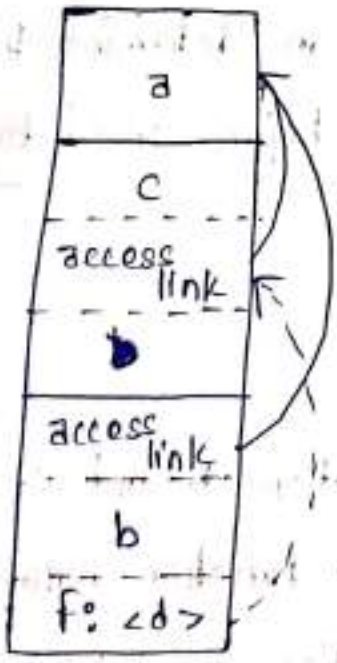
→ Hence when procedures are used as parameters, the caller needs to pass, along with the name of the procedure-parameter the proper access link for that parameter.

→ 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.

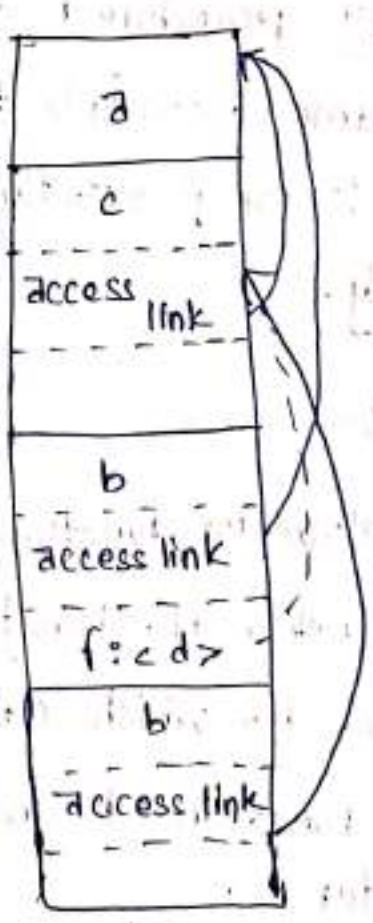
Example:-

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.

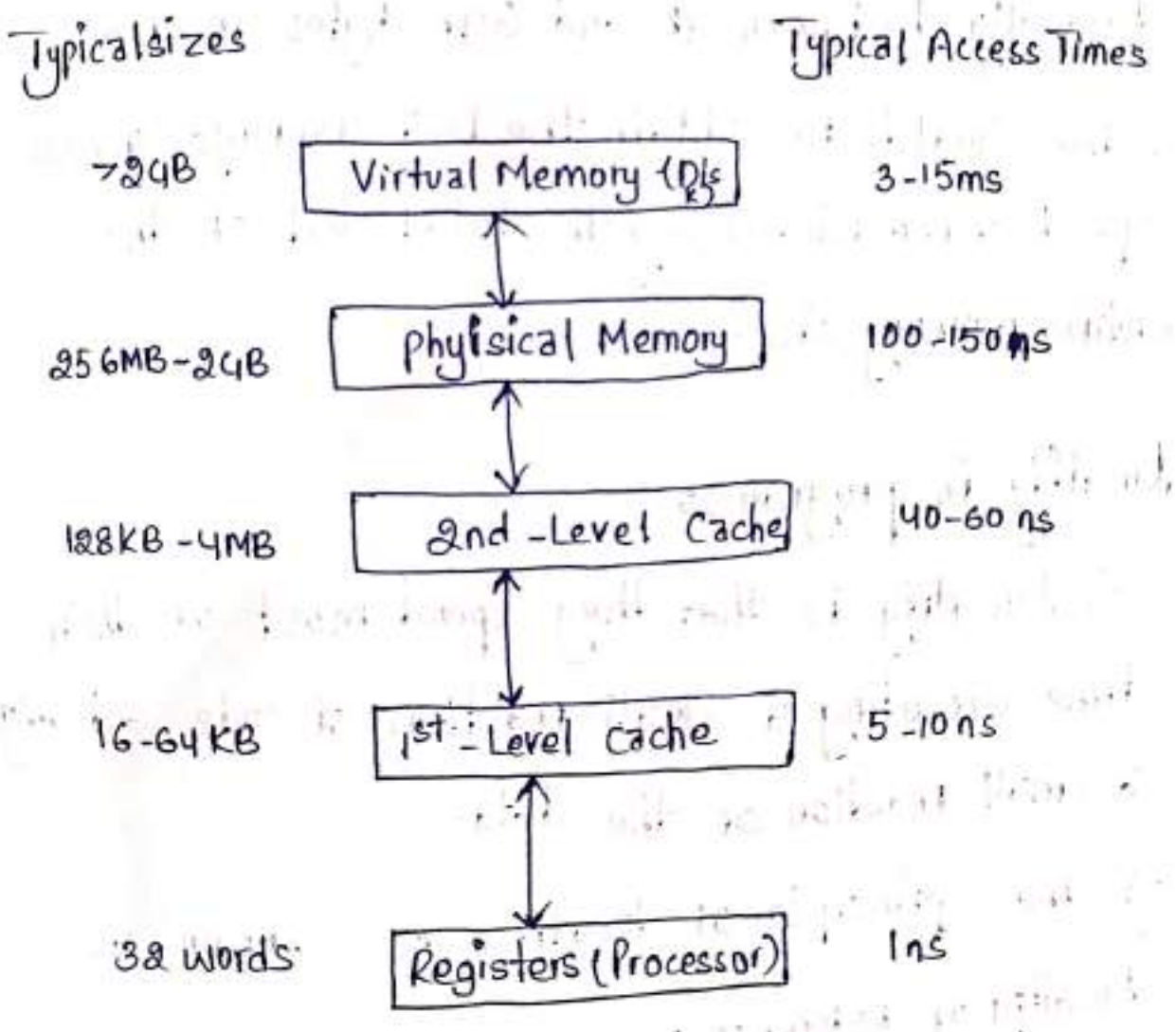
Note:-

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 contiguous 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 contiguous area of the heap.

→ free chunks of memory as holes. Holes of various size are scattered throughout memory.

→ when a process arives, 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.