02-04-25

Unit-I: Introduction to Automata & Regular Expressions:-

Introduction:-

1) Symbol:- A Symbol is an abstract entity or a user defined entity. Generally the symbols are letters and digits.

2) Alphabet (Σ): An Alphabet is a finite and non-empty Set of Symbols. It is denoted by 'Σ'.

Ex: {0, 1} - binary Alphabet

{a--- z, A--- Z} - English Alphabet

{0, 1, 2, 3, 4, 5, 6, 7} - octal Alphabet

{0, 1, 2, ... 9} - decimal Alphabet

{0--- 9, A--- F} - Hexadecimal Alphabet

* -> {1, 2, 3, ----} => It is infinite; so it is not an Alphabet

* -> { } -> It is finite but empty, so it is not an Alphabet

3) String (w): A string is defined as finite Sequence of Symbols derived from Alphabet. It is denoted by 'w'

Ex:- w = 001100 from Σ = {0, 1}

w = hello from Σ = {a--- z, A-- Z}

4) Length of a String: A length of a string indicates the number of Symbols Present in given string. for a string 'w', the length of string is denoted by "|w|".

Ex:- w = 001100 => |w| = 6

w = hello => |w| = 5

Note:-

=> The empty string or null string is denoted by "ε" or "Λ"

Power of an alphabet:-

The power of an alphabet refers to the set of all strings that can be formed using the symbols of a given alphabet. It is denoted by "Σ^n" where, Σ is alphabet and 'n' is the length of string.

Ex:-

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

$\Sigma^0 = \{\epsilon\}$

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

$\Sigma^2 = \{00, 01, 10, 11\}$

$\Sigma^3 = \{000, 001, 010, 100, 011, 101, 110, 111\}$

$\Sigma^4 = \{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111,$

$1000, 1001, 1010, 1100, 1011, 1101, 1110, 1111\}$.

2) let $\Sigma = \{a, b, c, d\}$

$\Sigma^0 = \{\epsilon\}$

$\Sigma^1 = \{a, b, c, d\}$

$\Sigma^2 = \{ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc,$

$aa, bb, cc, dd\}$

Concatenation:- Concatenation is an operation that combines two strings that is appending new string to the first string.

let w_1, w_2 are two strings, then the concatenated string $w_1 w_2$. also

$$|w_1 w_2| = |w_1| + |w_2|$$

Ex:-

let w_1 = hello, w_2 = World, then concatenated string

Prefix:-

A Prefix of string 'w' is formed by taking any number of symbols starting from the first symbol of the string. This includes the empty string and the string itself.

If $w = vy$ then 'v' is called the Prefix of 'w'.

Ex:-

let w = 1011

Prefix	String
ϵ	→ 1011
1	→ 011
10	→ 11
101	→ 1
1011	→ ϵ

Note:-

for a string "n" there will be "n+1" Prefixes

3/7/25

Proper Prefix:-

A Proper Prefix of any string is any Prefix that is not equal to string.

Ex:- let w = hello

Proper Prefixes are ϵ, h, he, hel, hell

→ for a string 'n' there will be 'n' Proper Prefixes.

Suffix:-

A Suffix of a string is formed by taking any number of symbols from the end of the string.

Ex:-

let w = hello

String	Suffix
hello	→ ϵ
hell	→ o
hel	→ lo

"n+1" Suffixes are generated for string 'n'.

Proper Suffix:

A Proper Suffix is any Suffix except the String itself

Ex:- let w = hello.

then proper suffixes are ∈, o, lo, llo, ello

→ For String "n" there will be "n" proper suffixes generated.

Substring:

A Substring is any Sequence of consecutive Symbols taken from String.

→ A String 'v' is said to be a Substring of 'w' if there exist strings x, y such that w = xvy

→ Substring = String - 1 Prefix - 1 Suffix

Ex:-

let w = hello

the substrings may be , ∈, h, e, l, l, o, he, el, ll, lo, hel, ell, llo, hell, ello, hello.

Reversal (w^r):

The reversal of String is obtained by writing Symbols of a String in reverse order.

Ex:-

let w = hello

then w^r = olleh

Kleene closure (w^*):

The kleen closure of a String 'w' represents Set of all strings that can be formed by concatenating '0' or more occurances of a String

Ex:- let w = ab, then

w^* = {∈, ab, abab, ababab, abababab.....}.

i: $w^0 = (ab)^0 = 1 \Rightarrow ∈$

$w^1 = (ab)^1 = ab$

Positive closure (w^+):

The positive closure of a string 'w' represents set of all strings that can be formed by concatenating one or more occurances of a string.

Ex:- let $w = ab$, then

$w^+ = \{ab, abab, ababab, abababab \cdots\}$

let $w = 0110$

$w^+ = \{0110, 01100110, 011001100110 \cdots\}$

Kleene closure (Σ^*):

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

Ex:-

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

$\Sigma^0 = \{\epsilon\}$

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

$\Sigma^2 = \{00, 01, 10, 11\}$

$\Sigma^3 = \{000, 001, 010, 100, \cdots\}$

Now, $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 100, 011 \cdots\}$

Positive closure (Σ^+):

Formula $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4 \cdots$

Ex:-

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

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

$\Sigma^2 = \{00, 01, 10, 11\}$

$\Sigma^3 = \{000, 001, 010, 100, \cdots\}$

Now, $\Sigma^+ = \{0, 1, 01, 10, 11, 00, 000, 001, 010, 100, \cdots\}$

Language (L):

A language 'L' over alphabet 'Σ' is a set of strings formed from symbols of specific

→ It can be finite or infinite.

→ It is denoted by 'L'.

→ A language over alphabet 'Σ' is a subset of Σ^* [Mathematically $L \subseteq \Sigma^*$]

→ The language 'ϕ' is undefined language and similar to '∞'

→ The language $\{\epsilon\}$ is a language with one empty string

$\emptyset = \{\} \neq \{\epsilon\}$

Ex:-

$L = \{\epsilon, 0, 1, 00, 01, 10, 11, 000\}$

is a finite language over alphabet $\Sigma = \{0, 1\}$

$L = \{\epsilon, 0, 1, 00, 01, 10, 11, \ldots\}$

is an infinite language over alphabet $\Sigma = \{0, 1\}$.

$L = \{\epsilon, 01, 0011, 000111, 00001111, \ldots\}$

is a language of all strings consisting 'n' 0's followed by 'n' 1's.

$L = \{\epsilon, 01, 10, 0011, 1100, 1010, 0101, 000111, 010101, 101010, \ldots\}$

is a language consisting of equal number of 0's and 1's.

4/1/25

Grammar:-

Grammar consist of four tuples.

$G = \{V_N, \Sigma, P, S\}$

(or)

$G = \{N, T, P, S\}$

Where V_N = Set of non terminals

Σ = Set of terminals

P = Production rules

Note:-

→ Non-terminal Symbols are those Symbols that can be replaced multiple times

→ Terminal Symbols can be used only once and cannot be replaced further.

Ex:-

Sentence: Ram goes to college

Sentence → S + P
S → N / Pr
P → V, prep, N

→ Non-terminals (can be again divided further)

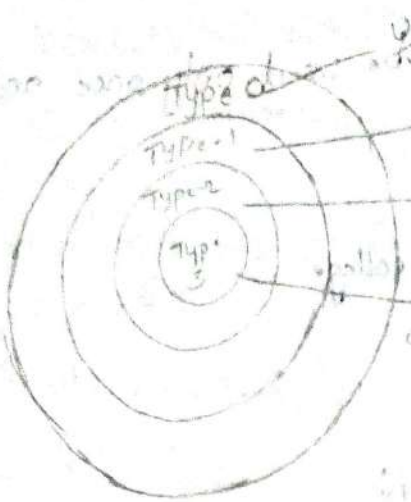
N → Ram / college
V → goes, eat, talk
Prep → for, to, of

→ terminals (Which can't be further divided)

Chomsky Hierarchy:

Chomsky classified grammar into 4 types.

Grammar	language	Machine
Type 0 (Unrestricted grammar)	Recursively Enumerable language	Turing Machine
Type 1 (CSG) Context Sensitive grammar	Context Sensitive language (CSL)	linear bounded Automata
Type 2 (CFG) Context Free grammar	context-free language (CFL)	Pushdown Automata
Type 3 (RG) Regular Grammar	Regular language / Regular expressions	finite Automata



Type 0: Grammar:

Type 0 grammar is called as unrestricted grammar. All formal grammars are type-0 grammars. The language generated by Type 0 grammar is Recursively Enumerable language. These languages are accepted by Turing machine.

→ Type-0 grammar is of form: $\alpha \longrightarrow \beta$

where $\beta \in (V_N \cup \Sigma)^*$

$\alpha \in (V_N \cup \Sigma)^+$ and $\alpha \neq \varepsilon$

Type 1 Grammar:

Type 1 Grammar is called as Context Sensitive grammar. The languages generated by Type 1 Grammar are called as context Sensitive languages. These languages are accepted by linear bounded automata.

→ Type 1 Grammar is of form $\alpha A \beta \rightarrow \alpha B \beta$

where

$\alpha B \beta \in (V_N \cup \Sigma)^*$

$A \in V_N$

$B \neq \varepsilon$

Type 2 Grammar:

Type 2 Grammar is also called as context-free grammar. The languages generated by Type 2 Grammar are called as context free languages. These languages are accepted by pushdown Automata.

→ Type 2 Grammar is of form $A \rightarrow \alpha$

where

$A \in V_N$

$\alpha \in (V_N \cup \Sigma)^*$

Type 3 Grammar:

Type 3 Grammar is also called as Regular Grammar. The languages generated by type 3 grammar are called as Regular languages / Regular Expressions. These languages are accepted by finite Automata.

→ Type 3 Grammar is of form $A \rightarrow \varepsilon$, $A \rightarrow \alpha$, $A \rightarrow \alpha B$

where

$A, B \in V_N$

$\alpha \in \Sigma$

① Generate a language for grammar $S \rightarrow aSb / \varepsilon$

Given that

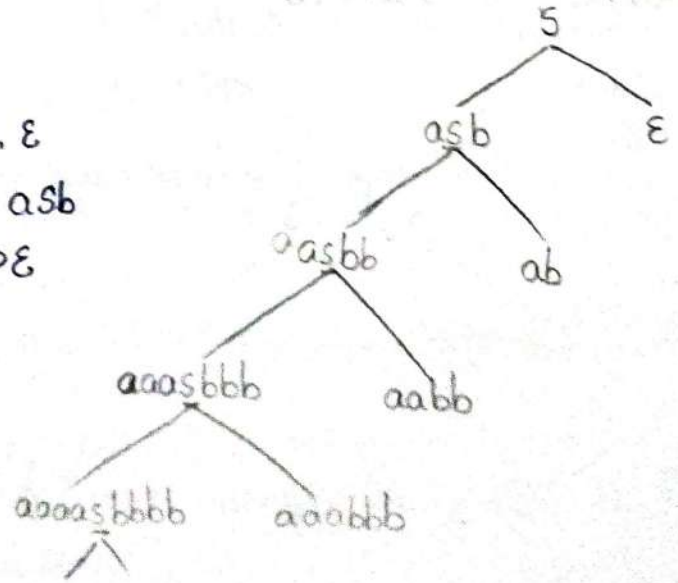
$V_N = S$

$\Sigma = a, b, \varepsilon$

$P = S \rightarrow aSb$

$S \rightarrow \varepsilon$

$S = S$



2) Generate a language from gammar $S \rightarrow asb|ab$

Given that

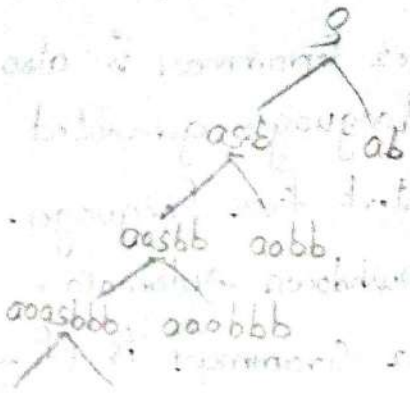
$V_N \Rightarrow S$

$\Sigma = a, b$

$P = S \rightarrow asb$

$S \rightarrow ab$

$S = S$



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

$L = a^n b^n, \quad n > 0$

3) Generate a language from grammar $S \rightarrow a\,s|a$

Given that

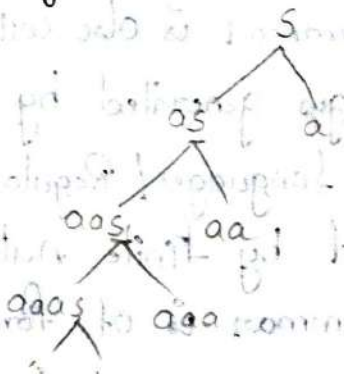
$V_N = S$

$\Sigma = a$

$P: S \rightarrow as$

$S \rightarrow a$

$S : S$



$L = \{a, aa, aaa, aaaa, \cdots\}$

$L = a^n, \quad n > 0 \text{ (or) } n \geq 1$

4) ~~(A → aAla)~~

4) $S \rightarrow asa|c, \quad S \rightarrow aca|\varepsilon$

$V_N = S$

$\Sigma = a, c, \varepsilon$

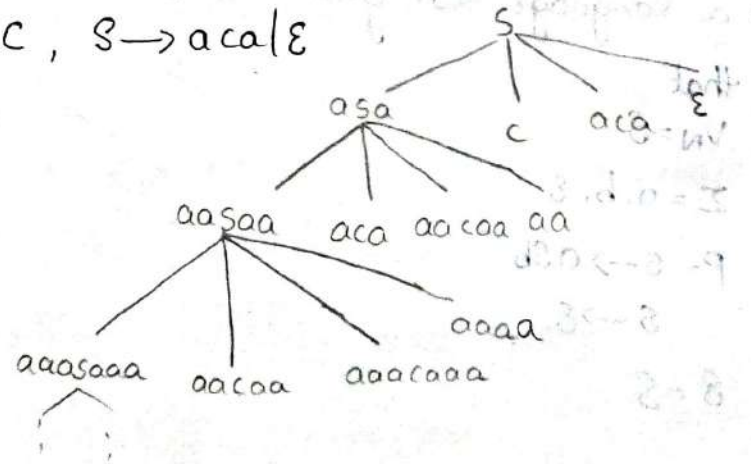
PR: $S \rightarrow asa$

$S \rightarrow c$

$S \rightarrow aca$

$S \rightarrow \varepsilon$

$S : S$



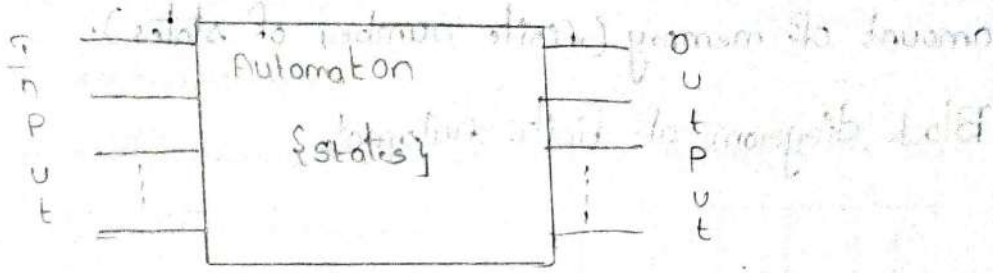
$L = \{\varepsilon, c, aa, aca, aacaa \cdots\}$

9/1/25

Automaton:-

Automaton is a system where materials, energy, and information are transformed and transmitted for performing some operation without direct human involvement.

Any automated machine is a automaton.



Characteristics of Automaton:

1) Input: At every clockpulse a fixed number of inputs are taken simultaneously from input alphabet 'Σ'.

2) output: At each clockpulse, corresponding outputs are generated from finite output alphabet.

3) State: The Automaton can reside in only one state at any discrete time.

4) State transition: The next state is determined by transition function (δ) based on current state and current input.

5) output relation: The output can be determine either by present state or by both present state & present input depending upon type of machine.

finite Automata:

finite automata is a machine that accepts regular expressions.

-> The finite automata accepts language in the format of type 3 grammar

where Q = finite & non-empty set of states.

Σ = finite & non-empty set of input symbols.

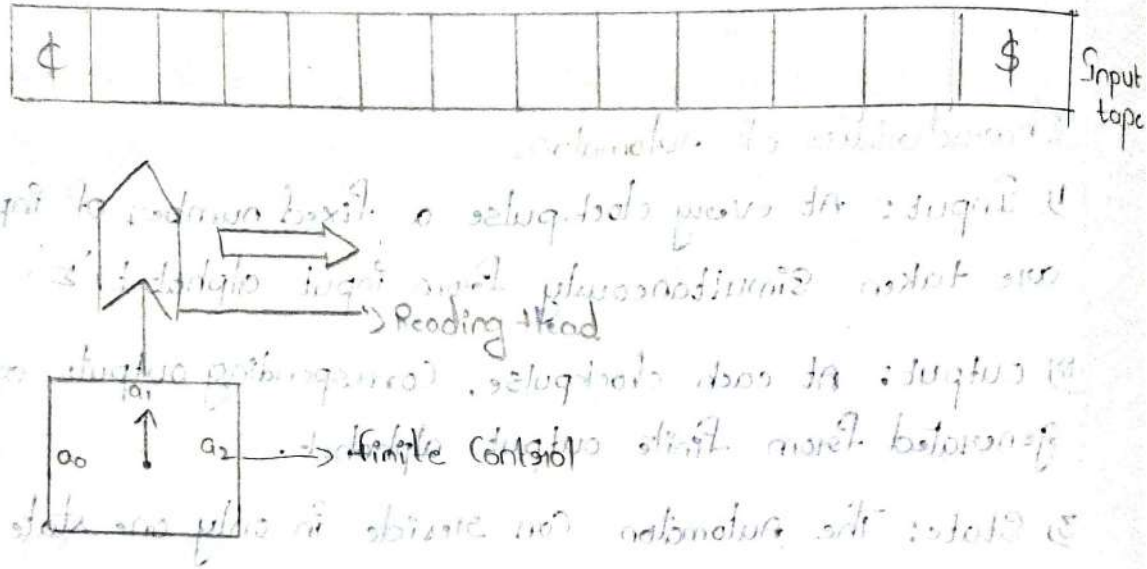
δ = transition function

q_0 = Start state / beginning state

F = finite & non-empty set of final states.

finite Automate is a machine that comes with limited amount of memory (finite number of states).

Block diagram of finite Automate



Input Tape:-

Input tape consists of input symbols. It is divided into different blocks and each block consists a single character from input alphabet. Both left end right end of input tape contain end markers. Between two end markers, input string is placed. The string needs to be processed from left to right.

Reading Head:

The Head Scans each square/block in the input tape and reads input from input tape.

finite Control:

finite Control is Considered as Control unit of

that "the machine is in this state & it is getting this input so it will go to this state"

10/7/25 Representation of finite Automata:-

finite Automata can be represented in two ways

a) Graphical representation

b) Tabular representation

Graphical representation:-

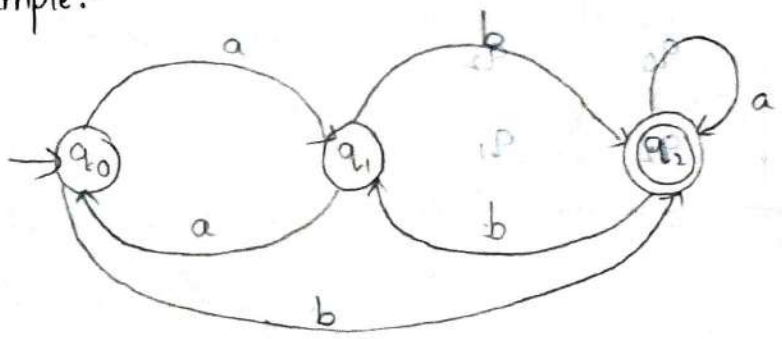
A Graphical representation of finite automata is also called as transition diagram.

In graphical representation, a state is represented as (q)

A beginning state is represented as "→(q)"

The final state is represented as "((q))"

Example:-



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

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

δ: $\delta(q_0, a) \rightarrow q_1$

$\delta(q_0, b) \rightarrow q_2$

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

$\delta(q_1, b) \rightarrow q_2$

$\delta(q_2, a) \rightarrow q_2$

$\delta(q_2, b) \rightarrow q_1$

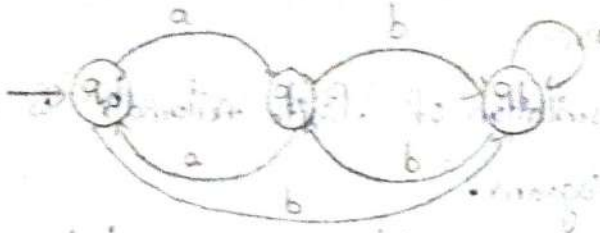
$q_0 = q_0$

$f = q_2$

b) Tabular representation:-

The Tabular representation of finite automata is also called as Transition table.

In tabular format the beginning state is represented as $\rightarrow q_0$ the final state is represented as (q_n)



Present State (pls)	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_0	q_2
(q_2)	q_2	q_1

Note:-

$(ab^*) = \{a, ab, abb, abbb, abbbb \cdots\}$

$(a^* b) = \{b, ab, aab, aaab, aaab \cdots\}$

$(a+b)^* = \{a, b\}^*$

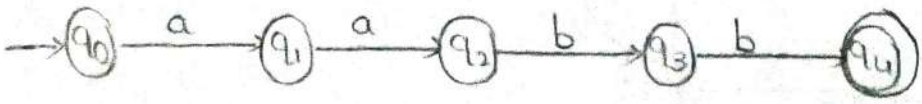
$L = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, baa \cdots\}$.

Q) Create finite automata that accepts the string aabb.

Substring = aabb.

length of substring = 4

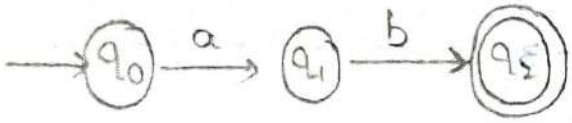
So, we have to take $4+1 = 5$ states.



Create finite automata that accepts the string ab.

Given Substring = ab.

total no: of states = 2+1 = 3.



Create a finite Automata that accepts the language with all a's

Given,

the language need to contain all a's.

So, $L = \{a, aa, aaa, \ldots\} = a^* a^+$

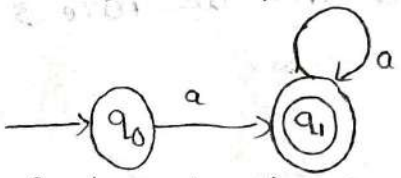
We can write a^+ as aa^*.

{* simply we keep * as self loop.

So, the string is aa^*

Substring = a

Total no: of states = 1+1 = 2 states



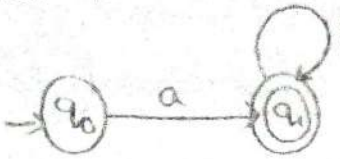
Create finite automata that always starts with Single 'a' and Contains 'n' number b's. $\langle n \geq 0 \rangle$

Given,,

the language always starts with 'a' and ends with 'n' numbe of b's.

So, it can be written as ab^*

So, total no: of states = 1+1 = 2



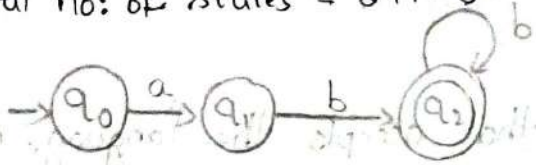
=> for $n>0$ ∴ it won't contain any 'ε' symbols because $n>0$

it can be written as ~~ab*~~ ab^+ => abb^*

So, the string is abb^*

Substring = ab.

Total no: of states = 2+1 = 3



11-7-25 Design a finite automata that accepts string that starts with any number of b's followed by 'a' & again followed by any no: of b's.

Given.

b^+ any no: of b's

a only one a

b^+ any no. of b's

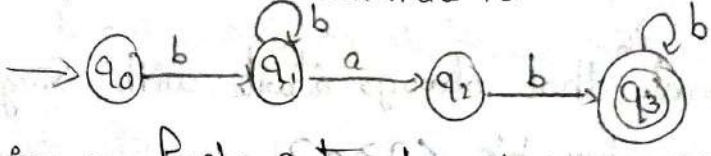
b^+ can be written as bb^*

then, we get $bb^* abb^* \Rightarrow bb^*a$

we cannot write repeated 'ones' then we have 3 states

3+1 = 4 states

Then, the finite automata is



design a finite automata that accepts unary numbers that are multiples of 3.

unary numbers => 1-1

11 - 2

111 - 3

3 - 111

6 - 111 111

9 - 111 111 111

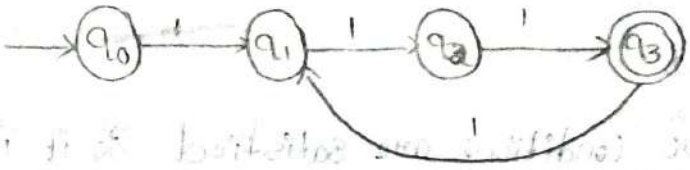
12 - 111 111 111 111

⋮

$\Rightarrow (111)^+$ it can be written as $(111)(111)^*$

In this, the substring length = 3

Then, 3+1 = 4 states



Imp LM

Acceptance of a string by finite Automata :-

for a string to be accepted by finite Automata it should follow the below two conditions -

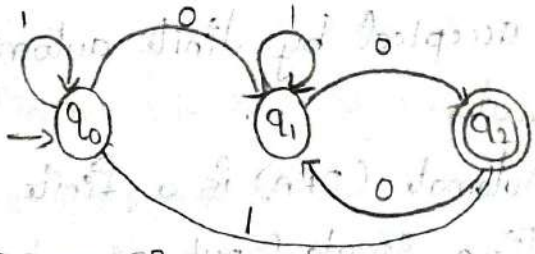
Condition-1 :- The string must be totally traversed (or) traverse.

Condition-2 :- The machine must come to final state

In short, it can be represented as,

$\delta(q_0, w) \rightarrow F$

Ex :-



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

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

$q_0 = q_0$

$F = q_2$

δ : $\delta(q_0, 0) \rightarrow q_1$

$\delta(q_0, 1) \rightarrow q_0$

$\delta(q_1, 0) \rightarrow q_2$

$\delta(q_1, 1) \rightarrow q_1$

$\delta(q_2, 0) \rightarrow q_1$

$\delta(q_2, 1) \rightarrow q_0$

Case-2:-

(a) Let $w = 010010$

$\delta(q_0, w) \rightarrow f$

$\delta(q_0, \underline{0}10010) \rightarrow q_1$

$\delta(q_1, \underline{1}0010) \rightarrow q_1$

$\delta(q_1, \underline{0}010) \rightarrow q_2$

$\delta(q_2, \underline{0}10) \rightarrow q_1$

$\delta(q_1, \underline{1}0) \rightarrow q_1$

$\delta(q_1, 0) \rightarrow q_2$

In this, all the two conditions are satisfied so it is accepted by the finite automate.

(b) $w = 01010$

$\delta(q_0, w) \rightarrow f$

$\delta(q_0, \underline{0}1010) \rightarrow q_1$

$\delta(q_1, \underline{1}010) \rightarrow q_1$

$\delta(q_1, \underline{0}10) \rightarrow q_2$

$\delta(q_2, \underline{1}0) \rightarrow q_0$

$\delta(q_0, 0) \rightarrow q_1$

Hence, the string is not accepted by finite automate.

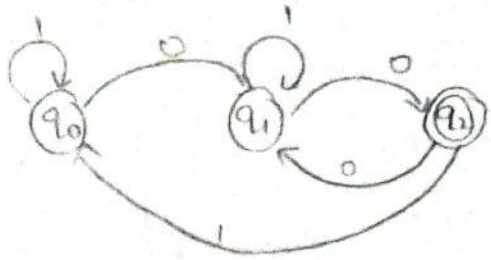
Deterministic finite Automata:-

Deterministic finite Automate (DFA) is a finite Automate where for all cases, for a single input given to a single state the machine goes to single state. that is all the moves of finite automate can be uniquely determined by present state and present input symbol.

A DFA can be represented as

$$M_{DFA} = \{Q, \Sigma, \delta, q_0, f\}$$

Ex:-



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

where $Q \times \Sigma \rightarrow Q$

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

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

$q_0 = q_0$

$F = q_2$

δ: $\delta(q_0, 0) \rightarrow q_1$
$\delta(q_0, 1) \rightarrow q_0$
$\delta(q_1, 0) \rightarrow q_2$
$\delta(q_1, 0) \rightarrow q_1$
$\delta(q_2, 0) \rightarrow q_1$
$\delta(q_1, 1) \rightarrow q_0$

Non deterministic finite Automata (NFA (or) NDFA):

Non deterministic finite Automata is a finite Automata where for some cases, a single input given to single state the machine goes to more than one state. That is, the moves of the machine cannot be uniquely determined by the present state and present input symbol.

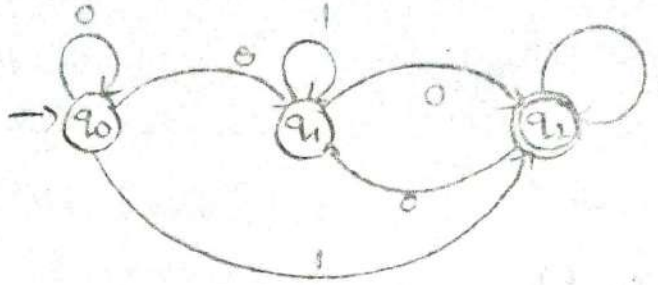
A NFA can be represented as

$M_{NFA} = \{Q, \Sigma, \delta, q_0, F\}$

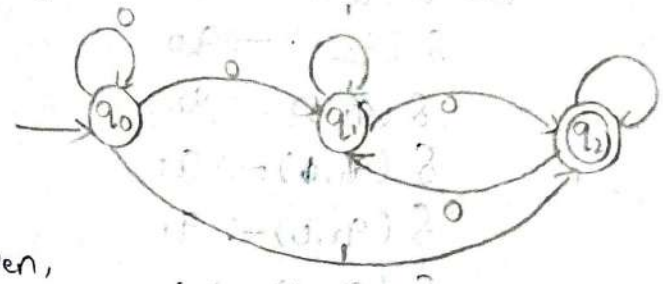
where $Q \times \Sigma \rightarrow 2^Q$

Example:-

PIs	NIs.	
	0	1
$\rightarrow q_0$	$q_0 q_1$	q_2
q_1	q_2	q_1
(q_2)	q_1	q_2

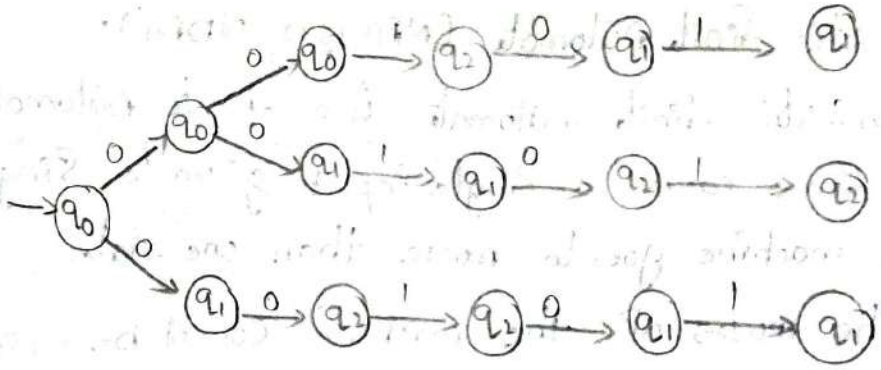


Q) check weather the string w=00101 is accepted by the given finite automate.

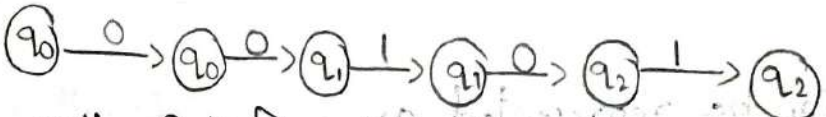


Given,

$\delta(q_0, w) \rightarrow F$



The path is



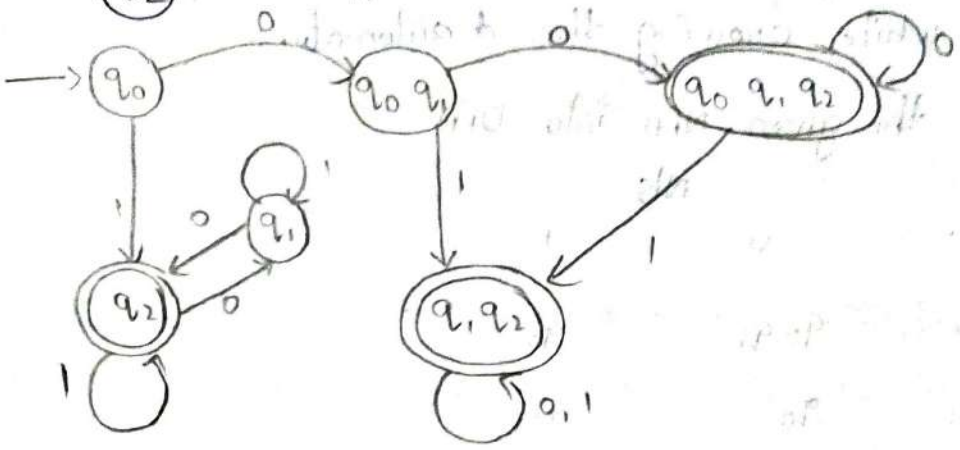
This path satisfies both the conditions of the finite automate. So, this string is the accepted by finite automate.

14/1/25

Q) Construct DFA from given NFA

	N/S (next state)	
P/S	0	1
$\rightarrow q_0$	$q_0 q_1$	q_2
q_1	q_2	q_1
q_2		

	NIS	
PIS	0	1
→ q_0	$\{q_0, q_1\}$	q_2
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_2, q_1\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_2, q_1\}$
$\{q_1, q_2\}$	$\{q_2, q_1\}$	$\{q_1, q_2\}$
q_1	q_2	q_1
(q_2)	q_1	q_2

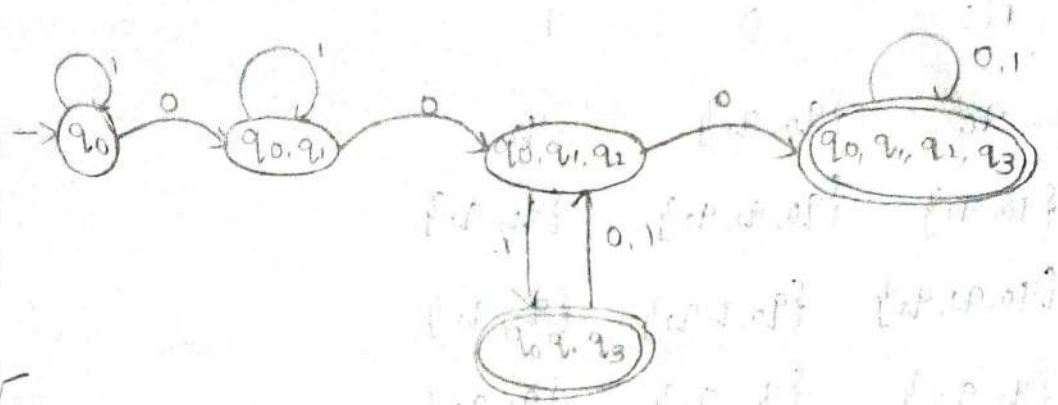


Q) Convert the given NFA to DFA

	NIS	
PIS	0	1
→ q_0	$q_0 q_1$	q_0
q_1	q_2	q_1
q_2	q_3	q_3
(q_3)	-	q_2

(A)

	NIS	
PIS.	0	1
→ q_0	$\{q_0, q_1\}$	q_0
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_3\}$
$\{q_0$	$\{$	$\{$



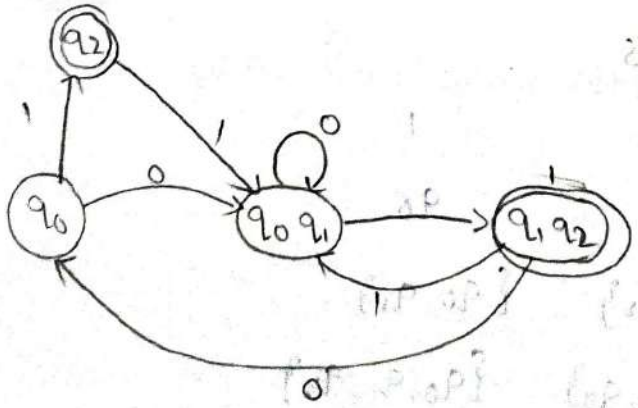
[If we connect q_1, q_2, q_3 states in these automata still they won't get connected to q_0, we can skip q_1, q_2 & q_3 states while drawing the automata.]

Q) Convert the given NFA into DFA.

	NIS	
Pls.	0	1
$\rightarrow q_0$	$q_0 q_1$	q_2
q_1	q_0	q_1
(q_2)	-	$q_0 q_1$

A)

	NIS	
Pls.	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	q_2
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_2, q_1\}$
$\{q_1, q_2\}$	$\{q_0\}$	$\{q_1, q_0\}$
q_2	-	$\{q_0, q_1\}$

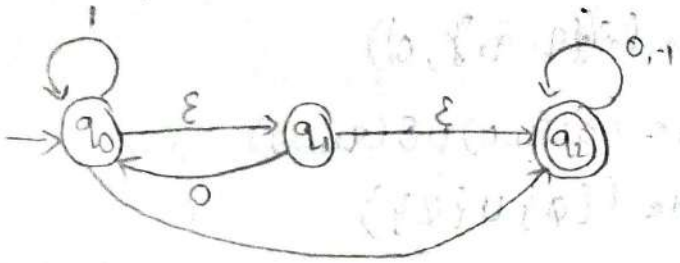


NFA with Empty (ε or λ) or NULL moves:-

If any Finite Automata contains empty moves then the Finite Automata is called Finite Automata with empty moves.

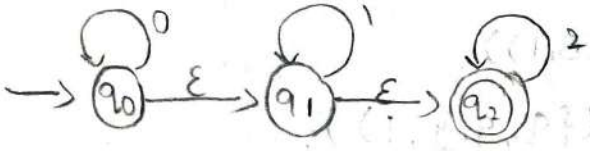
Note:

Any state will take (empty) ε as input and reach itself by default.



17/1/25 Conversion of NFA to DFA using e-closure method:-

Q) Convert the given NFA to DFA ∴ Consider path with 'ε'.



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

e-closure $(q_1) = \{q_1, q_2\} = B$

e-closure $(q_2) = \{q_2\} = C$

$\delta'(A,0) = \varepsilon$-closure $(\delta(A,0))$

$= e\text{-closure}(\delta(\{q_0, q_1, q_2\}, 0))$

$= e\text{-closure}(\delta(q_0,0) \cup \delta(q_1,0) \cup \delta(q_2,0))$

$= e\text{-closure}(q_0)$

$\boxed{\delta'(A,0) = \{q_0, q_1, q_2\} = A}$

$\delta'(A,1) = e\text{-closure}(\delta(A,1))$

$= e\text{-closure}(\delta(\{q_0, q_1, q_2\}, 1))$

$= e\text{-closure}(\delta(q_0,1) \cup \delta(q_1,1) \cup \delta(q_2,1))$

$= e$-closure

$\delta'(A,2)$ = e-closure $(\delta(A,2))$

= e-closure $(\delta(\{q_0,q_1,q_2\},2))$

= e-closure $(\delta(q_0,2) \cup \delta(q_1,2) \cup \delta(q_2,2))$

= e-closure $(\{\phi\} \cup \{\phi\} \cup \{q_2\})$

= e-closure (q_2)

$\delta'(A,2) = q_2 = C$

$\delta'(B,0)$ = e-closure $(\delta(B,0))$

= e-closure $(\delta(\{q_1,q_2\},0))$

= e-closure $(\delta(q_1,0) \cup \delta(q_2,0))$

= e-closure $(\{\phi\} \cup \{\phi\})$

= e-closure (ϕ)

$= \phi$

It is not a valid transition.

$\delta'(B,1)$ = e-closure $(\delta(B,1))$

= e-closure $(\delta(\{q_1,q_2\},1))$

= e-closure $(\delta(q_1,1) \cup \delta(q_2,1))$

= e-closure $(\{q_1\} \cup \{\phi\})$

= e-closure (q_1)

$\delta'(B,1) = q_0\{q_1,q_2\} = B$

$\delta'(B,2)$ = e-closure $(\delta(B,2))$

= e-closure $(\delta\{q_1,q_2\},2))$

= e-closure $(\delta(q_1,2) \cup \delta(q_2,2))$

= e-closure $(\{\phi\} \cup \{q_2\})$

= e-closure (q_2)

$\delta'(B,2) = q_2 = C$

$\delta'(C,0)$ = e-closure $(\delta(C,0))$

= e-closure $(\delta(\{q_2\},0))$

= e-closure $(\delta(q_2,0))$

= e-closure $\{\emptyset\}$

= $\emptyset$

Which is not a valid transition.

$\delta'(C,1)$ = e-closure $(\delta(C,1))$

= e-closure $(\delta\{q_2\},1)$

= e-closure $(\delta(q_2,1))$

= e-closure $(\emptyset)$

= $\emptyset$

Which is not a valid transition.

$\delta'(C,2)$ = e-closure $(\delta(C,2))$.

= e-closure $(\delta\{q_2\},2)$

= e-closure $(\delta(q_2,2))$

= e-closure $\{q_2\}$

$\boxed{\delta'(C,2) = q_2 = C}$

Hence the valid transitions are

$\delta'(A,0) \rightarrow A$

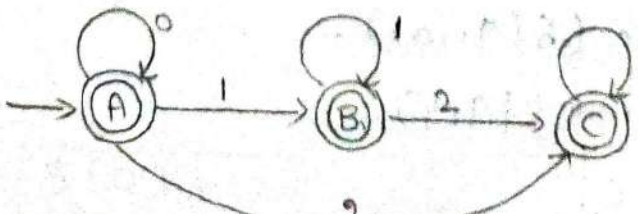
$\delta'(A,1) \rightarrow B$

$\delta'(A,2) \rightarrow C$

$\delta'(B,1) \rightarrow B$

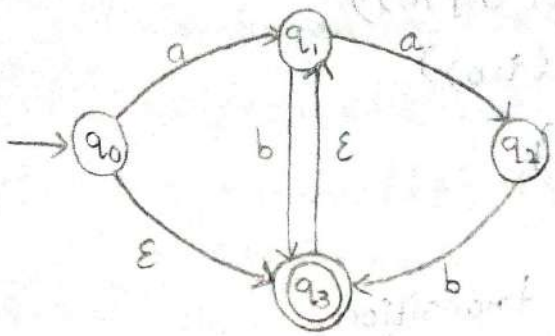
$\delta'(B,2) \rightarrow C$

$\delta'(C,2) \rightarrow C$



P/S	N/S 0	1	2
→(A)	A	B	C
(B)	–	B	C

Q) Convert the given NFA to DFA



e-closure $(q_0) = \{q_0, q_3, q_1\} = A$

e-closure $(q_1) = \{q_1\} = B$

e-closure $(q_2) = \{q_2\} = C$

e-closure $(q_3) = \{q_3, q_1\} = D$

$\delta'(A, a) =$ e-closure $(\delta(A, a))$

$=$ e-closure $(\delta(\{q_0, q_1, q_3\}, a))$

$=$ e-closure $(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_3, a))$

$=$ e-closure $(\{q_1\} \cup \{q_2\} \cup \{\phi\})$

$\boxed{\delta'(A, a) = \text{e-closure}(q_1, q_2) = E}$

$\delta'(A, b) =$ e-closure $(\delta(A, b))$

$=$ e-closure $(\delta(\{q_0, q_1, q_3\}, b))$

$=$ e-closure $(\delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_3, b))$

$=$ e-closure $(\{\phi\} \cup \{q_3\} \cup \{\phi\})$

$=$ e-closure (q_3)

$\boxed{\delta'(A, b) = \{q_3, q_1\} = D}$

$\delta'(B, a) =$ e-closure $(\delta(B, a))$

$=$ e-closure $(\delta(q_1, a))$

$=$ e- closure $(\{q_2\})$

$\boxed{\delta'(B, a) = \{q_2\} = C}$

$\delta'(B,b)$ = e-closure $(\delta(B,b))$

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

= e-closure $(\delta(q_1,b))$

= e-closure$(\{q_3\})$

$\boxed{\delta'(B,b) = \{q_3,q_1\} = D}$

$\delta'(C,a)$ = e-closure $(\delta(C,a))$

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

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

= e-closure $(\{\phi\})$

$= \phi$

Which is not a valid transition

$\delta'(C,b)$ = e-closure $(\delta(C,b))$

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

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

$\delta'(C,b)$ = e-closur $(\{q_3\})$

$\boxed{\delta'(C,b) = \text{closure}(q_3) = F}$

$\delta'(D,a)$ = e-closure $(\delta(D,a))$

= e-closure $(\delta(\{q_3,q_1\},a))$

= e-closure$(\delta(\{q_3,a\} \cup \delta(q_1,a))$

= e-closure $(\{\phi\} \cup \{q_2\})$

$\boxed{\delta'(D,a) = \text{e-closure}(q_2) = C}$

$\delta'(D,b)$ = e-closure $(\delta(D,b))$

= e-closure $(\delta(\{q_3,q_1\},b))$

= e-closure $(\delta(q_3,b) \cup \delta(q_1,b))$

= e-closure $(\{\phi\} \cup \{q_3\})$

$\boxed{\delta'(D,b) = \text{e-closure}(q_3) = C}$

$\delta'(E,a) = e\text{-closure}(\delta(E,a))$

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

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

$= e\text{-closure}(\{q_2\} \cup \{\phi\})$

$\boxed{\delta'(E,a) = e\text{-closure}(\{q_2\}) = C}$

$\delta'(E,b) = e\text{-closure}(\delta(E,b))$

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

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

$= e\text{-closure}(\{q_3\} \cup \{q_3\})$

$\boxed{\delta(E,b) = e\text{-closure}(\{q_3\}) = F}$

$\delta'(F,a) = e\text{-closure}(\delta(F,a))$

$= e\text{-closure}(\delta(\{q_3\},a)$

$= e\text{-closure}(\delta(q_3,a))$

$= \phi$

Which is invalid transition

$\delta'(F,b) = e\text{-closure}(\delta(F,b))$

$= e\text{-closure}(\delta(\{q_3\},b))$

$= e\text{-closure}(\delta(q_3,b))$

$= \phi$

Which is invalid transition

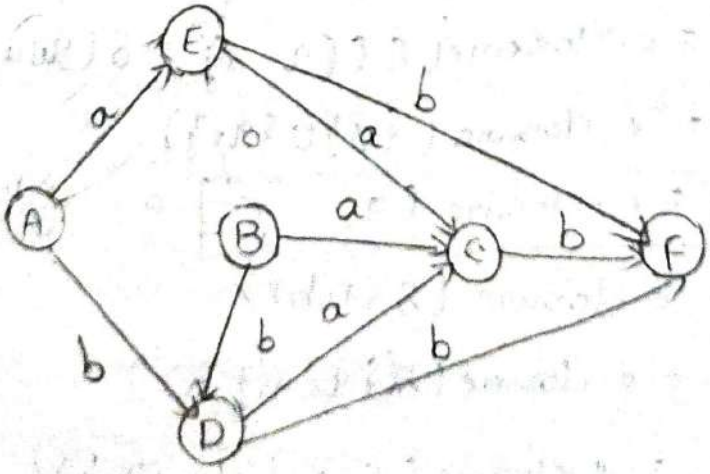
=> The valid Transitions are:-

$\delta'(A,a) = E$ $\quad \delta'(C,b) = F$ $\quad \delta'(E,b) = F$

$\delta'(A,b) = D$ $\quad \delta'(D,a) = C$

$\delta'(B,a) = C$ $\quad \delta'(D,b) = F$

$\delta'(B,b) = D$ $\quad \delta'(E,a) = C$



Minimization of finite Automata

Q) Minimise the given DFA

PS	NS 0	1
→ q_0	q_1	q_2
q_1	q_2	q_3
q_2	q_2	q_4
(q_3)	q_3	q_3
(q_4)	q_4	q_4
(q_5)	q_5	q_4

Equivalence 0 - $S_0 = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

Equivalence 1 - $S_1 = \{ \{\underbrace{q_0, q_1, q_2}_{G_1}\}, \{\underbrace{q_3, q_4, q_5}_{G_2}\} \}$.

Equivalence 2 ⇒ G_1 → initial states; G_2 → final states

	0	G?	1	G?	
q_0	q_1	G_1	q_2	G_1	$G_1 G_1$
q_1	q_2	G_1	q_3	G_2	$G_1 G_2$
q_2	q_2	G_1	q_4	G_2	

	0	G?	1	G?	
q_3	q_3	G_2	q_3	G_2	$G_2 G_2$
q_4	q_4	G_2	q_4	G_2	
q_5	q_5	G_2	q_4	G_2	

Equivalence 2 - $S_2 = \{ \{\underbrace{q_0}_{G_1}\}, \{\underbrace{q_1, q_2}_{G_2}\}, \{\underbrace{q_3, q_4, q_5}_{G_3}\} \}$.

	0	G?	1	G?	
q_1	q_1	G_2	q_3	G_3	$G_2 G_3$
q_2	q_2	G_2	q_4	G_3	

	0	G?	1	G?	
q_3	q_3	G_3	q_3	G_3	$G_3 G_3$
q_4	q_4	G_3	q_4	G_3	
q_5	q_5	G_3	q_4	G_3	

Equivalence 3 - $S_3 = \{ \{q_0\}, \{q_1, q_2\}, \{q_3, q_4, q_5\} \}$.

In this, each and every state is Considered as the single state. So it has got minimised into 3 states

PIS	NIS 0	1
→ q_0	q_1	q_2
q_1q_2	q_2	q_3q_4
$q_3q_4q_5$	$q_3q_4q_5$	q_3q_4

$S_3 = \{\{q_0\}, \{q_1, q_2\}, \{q_3, q_4, q_5\}\}$

→ Since q_1 is not a separate state but it is a subset of $\{q_1, q_2\}$ so, we add q_2 to it

Same for every state.

Based on the equivalence-3, we can write

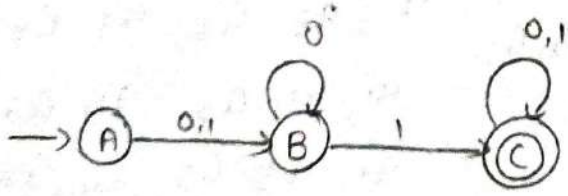
PIS	NIS 0	1
→ q_0	q_1q_2	q_1q_2
q_1q_2	q_1q_2	$q_3q_4q_5$
$q_3q_4q_5$	$q_3q_4q_5$	$q_3q_4q_5$

From $S_3 = \{\{q_0\}, \{q_1, q_2\}, \{q_3, q_4, q_5\}\}$
A B C

Convert the transition table into the above format.

PIS	NIS 0	1
A	B	B
B	B	C
C	C	C

Transition diagram:



Q) Minimise the finite Automata.

PIs	NIs 0	1
→A	f	B
B	C	G_1
©C	C	A
D	G_1	C
E	F	H
F	G_1	C
G_1	E	G_1
H	C	G_1

Equivalence 0 $S_0 = \{A, B, C, D, E, F, G, H\}$

Equivalence 1 $S_1 = \{\{\underbrace{A, B, D, E, F, G, H}_{G_{11}}\}, \underbrace{\{C\}}_{G_{12}}\}$ final state

Equivalence 2 $S_2 \Rightarrow$

PIs	0	G?	1	G?
A	f	G_{11}	B	G_{11}
B	C	G_{12}	G_1	G_{11}
D	G_1	G_{11}	C	G_{12}
E	f	G_{11}	H	G_{11}
F	G_1	G_{11}	C	G_{12}
G	E	G_{11}	G_1	G_{11}
H	C	G_{12}	G_1	G_{11}

$G_1G_1 = \{A, E, G\}$

$G_1G_2 - \{D, F\}$

$G_2G_1 - \{B, H\}$

$S_2 = \{\{\underbrace{A, E, G}_{G_1}\}, \{\underbrace{D, F}_{G_2}\}, \{\underbrace{B, H}_{G_3}\}, \{\underbrace{C}_{G_4}\}\}$

Equivalence 3 $S_3 \Rightarrow$

PIs	0	G?	1	G?
A	f	G_2	B	G_3
E	f	G_2	H	G_3

$\}-G_2G_3$

PIs	0	G?	1	G?
D	G_1	G_1	C	G_4

$\}G_1G_1$

	NIs				
PIs	0	G_1	1	G_2	
B	C	G_4	G	G_1	} $G_4 G_1$
H	C	G_4	G	G_1	

$S_3 = \{ \{\underbrace{A,E}_{G_1}\}, \{\underbrace{G}_{G_2}\}, \{\underbrace{D,F}_{G_3}\}, \{\underbrace{B,H}_{G_4}\}, \{\underbrace{C}_{G_5}\} \}$

Equivalence 4 S_4

	NIs			
PIs	0	G_1	1	G_2
A	F	G_3	B	G_4
E	F	G_3	H	G_4

	NIs			
PIs	0	G_1	1	G_2
D	G	G_1	C	G_5
F	G	G_1	C	G_5

$S_4 = \{ \{A,E\}, \{G\}, \{D,F\}, \{B,H\}, \{C\} \}$

Transition table:-

PIs.	NIs 0	NIs 1
→ AE	F	BH
G	E	G
DF	G	C
BH	C	G
(C)	C	A

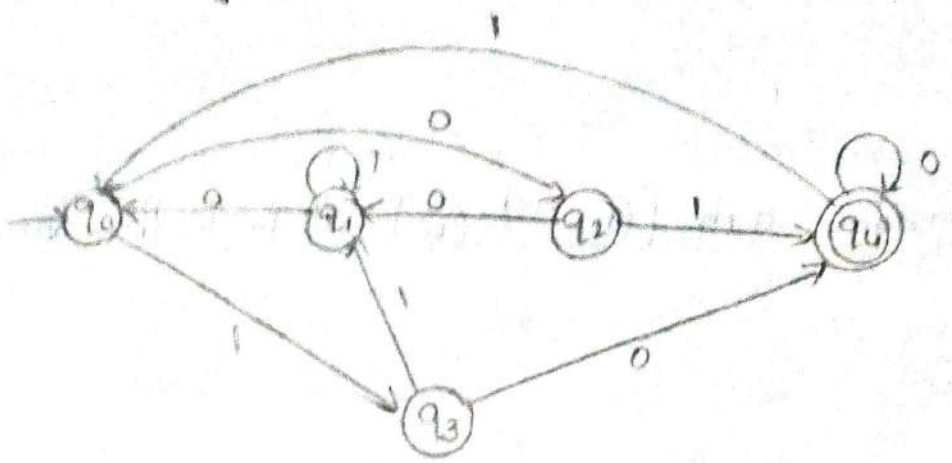
⟹

PIs	NIs 0	NIs 1
→ AE	DF	BH
G	AE	G
DF	G	C
BH	C	G
(C)	C	AE

Assume, $AE = q_0$, $G = q_1$, $DF = q_2$, $BH = q_3$, $C = q_4$

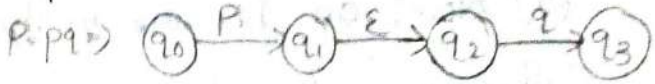
PIs	NIs 0	NIs 1
→ q_0	q_2	q_3
q_1	q_0	q_1
q_2	q_1	q_4

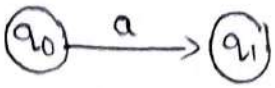
Transition diagram:-

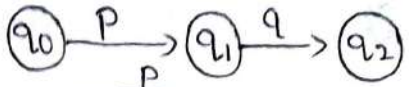


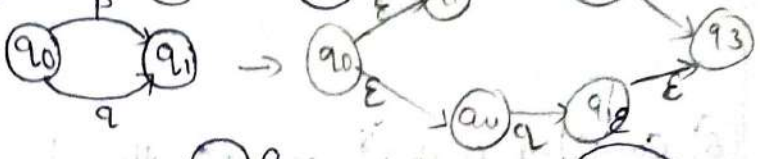
Q) Convert the given regular expression into finite automata.

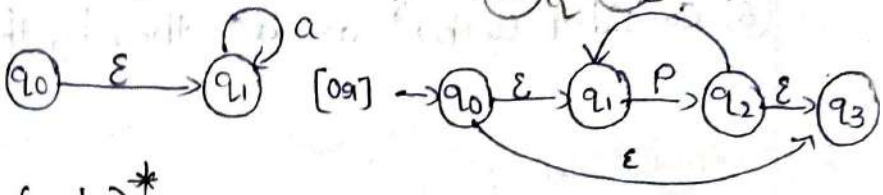
Rules:-



→ RE = a → 

→ RE = pq → 

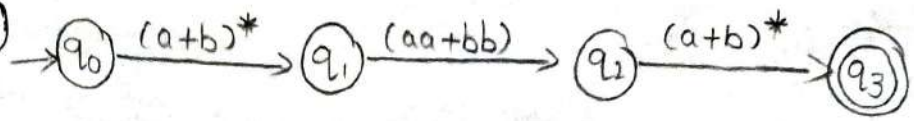
→ RE = p+q → 

→ RE = a* → 

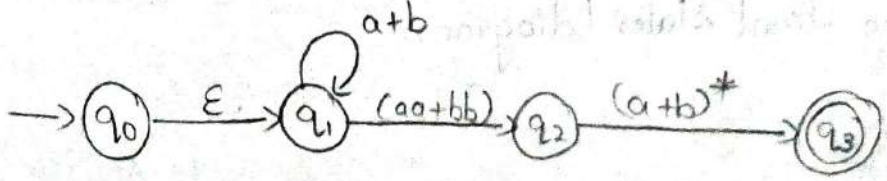
(i) $(a+b)^* \cdot (aa+bb)(a+b)^*$

$\underbrace{(a+b)^*}_{P} \cdot \underbrace{(aa+bb)}_{q} \underbrace{(a+b)^*}_{r}$

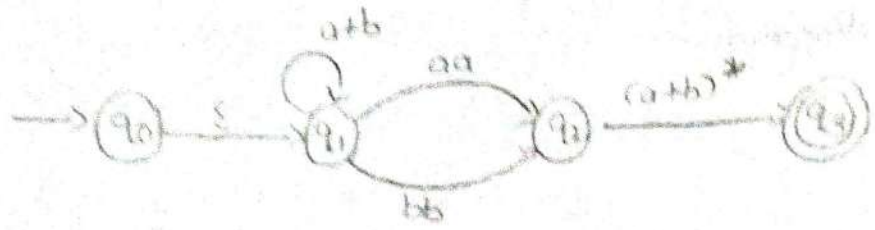
It is in the form of pqr, then

① 

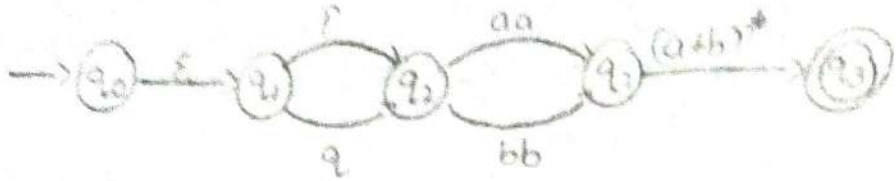
② firstly $(a+b)^*$ consider $\underbrace{(a+b)^*}_{a}$ as a^* then by the rules,



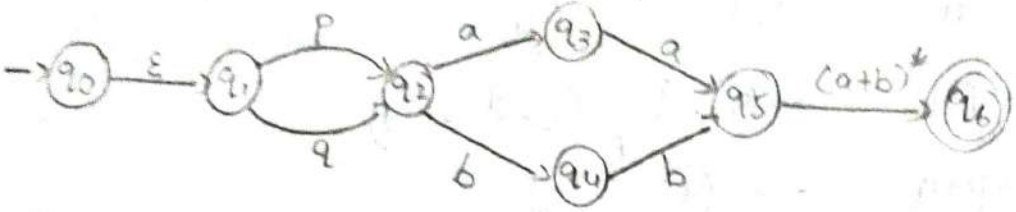
③ Now, (aa+bb) consider



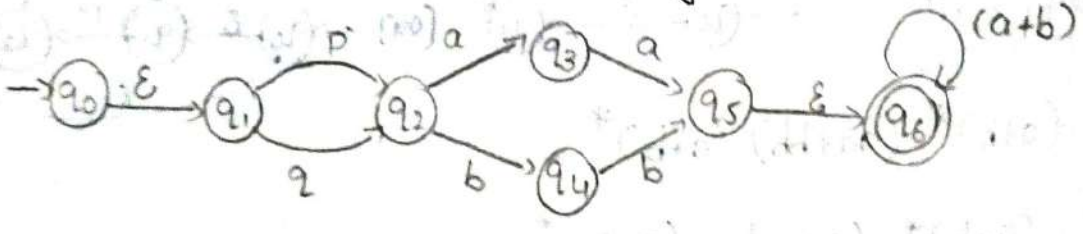
④ Consider, a+b [q_1 a+b q_2]. it is in the form of p_{+q}



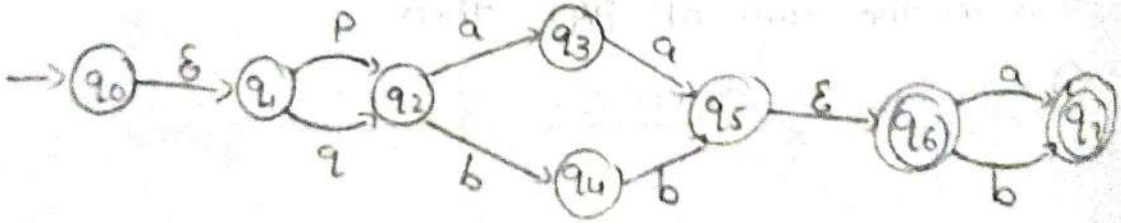
⑤ Now, aa, bb are in the form of pq. So by splitting them, we get. p q



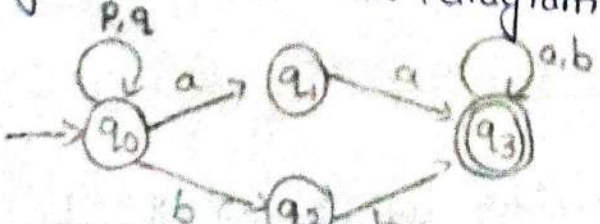
⑥ Consider (a+b)* as a* then by the rules.



⑦ Consider (a+b) as p+q then,



on reducing, remove the ε states and add self loops to get the final states / diagram.



21/1/20 Pumping Lemma:-

Theorem:-

Let $M = \{Q, \Sigma, \delta, q_0, F\}$ be a finite automata with 'n' states. Let L be a regular language accepted by finite Automata 'M'. Let 'w' be a string of language 'L'. and $|w| \geq n$ i.e., $m \geq n$, the length of string is greater than or equal to number of states. Then there exist.

$w = xyz$ Such that $|xy| \leq n$ and $|y| > 0$ and $xy^i z \in L$ for $i \geq 0$

Proof:-

Let 'w' be the string of length 'm'

The number of states in the given finite Automata be 'n'.

$$w = a_1 a_2 a_3 \cdots a_m$$

$$|w| = m$$

$$|w| = m \geq n$$

let q_0 be the start state that accepts the string 'w' and goes to final state.

The Transition function can be represented as

$$\delta(q_0, a_1, a_2, a_3 \cdots, a_i) = q_i \text{ for } i = 1, 2, 3, \cdots m$$

the states $Q_n = \{q_0, q_1 \cdots q_m\} \Rightarrow m+1$ states.

The no: of states = $m+1$ states but,

m should be greater than or equal to n.

m should be greater than or equal to $m+1$ (false).

So Atleast two states should be merged.

Consider two integers j, k such that $0 \le j < k \le n$

Take the pair q_j & q_k [and decompose it into]

[$q_1, q_2, \ldots q_j$,

$q_{j+1} \ldots q_k$, $q_{k+1} \ldots q_n$]

the string can be decomposed as

$a_1 a_2 \ldots a_j$,

$a_{j+1}, \ldots a_k$,

$a_{k+1} \ldots a_m$

Let $x = a_1 a_2 \ldots a_j$

$y = a_{j+1} \ldots a_k$

$z = a_{k+1} \ldots a_m$

Hence the string 'w' can be written as

$w = xyz$

As, $k \le n$, $a_1 \ldots a_k$ i.e., $|xy| \le n$

[When string tak]

when q_0 takes string 'w', first it processes 'x' and reaches the state 'q_j' and 'y' can be applied 'i' number of times and still it will be in the same state 'q_j'. on applying 'z' on 'q_j' we reach the final state 'q_m'. Therefore, xy^iz where $i > 0 \in L$

Q) Prove that $\{a^{i^2} \mid i>0\}$ is not regular.

Sol $L = \{a^{i^2} \mid i>0\}$

let $i=1 = a^{1^2} = a^1 = a$, length $= 1^2$

$i=2 = a^{2^2} = a^4 = aaaa$, length $= 2^2$

$i=3 = a^{3^2} = a^9 = aaaaaaaaa$, length $= 3^2$

$i=n = a^{n^2}$, length $= n^2$

$\therefore |w| = n^2$

According to pumping lemma, if $w = xyz \in L$ then, $xy^iz \in L$, $i>0$, $|xy| \leq n$ & $|y|>0$

Let $i=2$

$$|xy^2z| = |x| + 2|y| + |z|$$

$$= |x| + |y| + |y| + |z| > |x| + |y| + |z|$$

$$< |x| + 2|y| + |z| > n^2 \longrightarrow (1)$$

$$|x| + 2|y| + |z| = \underbrace{|x| + |y| + |z|}_{n^2} + |y| \quad \because \begin{matrix} |xy| \leq n \\ |y| \leq n \end{matrix}$$

$$= n^2 + |y|$$

$$= n^2 + n \longrightarrow (2)$$

From (1) & (2)

$$n^2 < |xy^2z| = n^2 + n$$

$$n^2 < |xy^2z| = n^2 + n < n^2 + n + n + 1$$

$$n^2 < |xy^2z| = n^2 + n < (n+1)^2$$

$$n^2 < |xy^2z| < (n+1)^2$$

n^2 present state

$(n+n^2)$ will be next state

$n^2 + 2n + 1$

$= n^2 + n + n + 1$

Since, it doesn't equal to either n^2 or $(n+1)^2$

23/1/25 Method-2:- **

Given language $L = \{ a^{i^2}, i>0 \}$

Now, the expanded language will be

$a^{1^2} = a$

$a^{2^2} = a^4 = aaaa$

$a^{3^2} = a^9 = aaaaaaaaa$

Now, the generated language is

$L = \{ a, aaaa, aaaaaaaaa, \ldots \}$

According to Pumping lemma, the four conditions to prove that a language is regular or not are

1) $w = xyz$
2) $|xy| \leq n$
3) $|y| > 0$
4) $xy^i z \in L, \ i \geq 0$

Now, we check whether the given language 'L' is regular or not.

Condition-1 :-

$w = xyz$

$w = \underbrace{a}_{x}\underbrace{aaa}_{y}\underbrace{a}_{z}$, where $n = 4$.

Condition-1 is Satisfied.

Condition-2:-

$|xy| \leq n$

$|aaa| \leq 4$

Condition-2 is Satisfied.

Condition-3:-

$|y| > 0$

$|aa| > 0$

$2 > 0$

Condition-3 is Satisfied

Condition-4:-

$xy^i z$, $i \geq 0 \in L$

For $i=2$, xy^2z

$a(aa)^2 a$

$aaaaaaa \notin L$

$xy^2z \notin L$

Condition-4 is not satisfied.

Hence, the given language 'L' is not regular.

Q) Check whether a^* is regular or not.

A) Given language

$L = \{a^*\}$

language generated is.

$L = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$

Now, Select a string which we can represent it in the form of xyz.

According to pumping lemma, the four conditions to prove that a language is regular or not are

1) $w = xyz$

2) $|xy| \leq n$

Now, we check whether the given language 'L' is regular or not.

Condition-1:-

$w = xyz$

$w = \underset{\downarrow x}{a}\ \underset{\downarrow y}{a}\ \underset{\downarrow z}{a}$, $n = 3$

Condition-1 is satisfied.

Condition-2 :-

$|xy| \leq n$

$|aa| \leq n$

$2 \leq 3$

condition-2 is satisfied.

Condition-3 :-

$|y| > 0$

$|a| > 0$

$1 > 0$

Condition-3 is satisfied.

Condition-4 :-

$xy^i z \; ; \; i \geq 0 \in L$

for $i = 2$,

xy^2z

$a(a)^2a$

$aaaa \in L$

for $i = 3$,

xy^3z

$a(a)^3a$

$aaaaa \in L$

Condition-4 is satisfied.

Closure Properties of regular Languages:-

1) Union:- If L_1, L_2 are regular languages then the union of L_1, L_2 also belongs to regular language.

Ex:-

$L_1 = \{a, aaa, aaaaa, \ldots\}$

$L_2 = \{aa, aaaa, \ldots\}$

$L_1 \cup L_2 = \{a, aa, aaa, aaaa, \ldots\}$

Intersection :- If L_1, L_2 are regular languages, then their intersection of L_1, L_2 also belongs to regular languages.

Ex:- $L_1 = a^*$

$L_2 = ab^*$

$L_1 \cap L_2 = a$

Complement:- If L is a regular language then its complement L^c is also a regular language.

Difference :- If L_1, L_2 are two regular languages then their difference $L_1 - L_2$ Also belongs to regular language.

Reversal: If L is a regular language then its reversal L^R is also a regular language.

Kleene closure :- If L is a regular language then the kleen closure of L (L^*) is also a regular language.

Concatenation :- If L_1, L_2 are two regular languages then $L_1 . L_2$ is also a regular language.

Homomorphism:- $M = \{Q, \Sigma, \delta, q_0, F\}$

Ex:- $L = aab$

Inverse Homomorphism:-

Ex:-

$L = 00^*1$

$h^{-1}(0) = a$

$h^{-1}(1) = b$

$\longleftrightarrow$ $h(a) = 0$, $h(b) = 1$

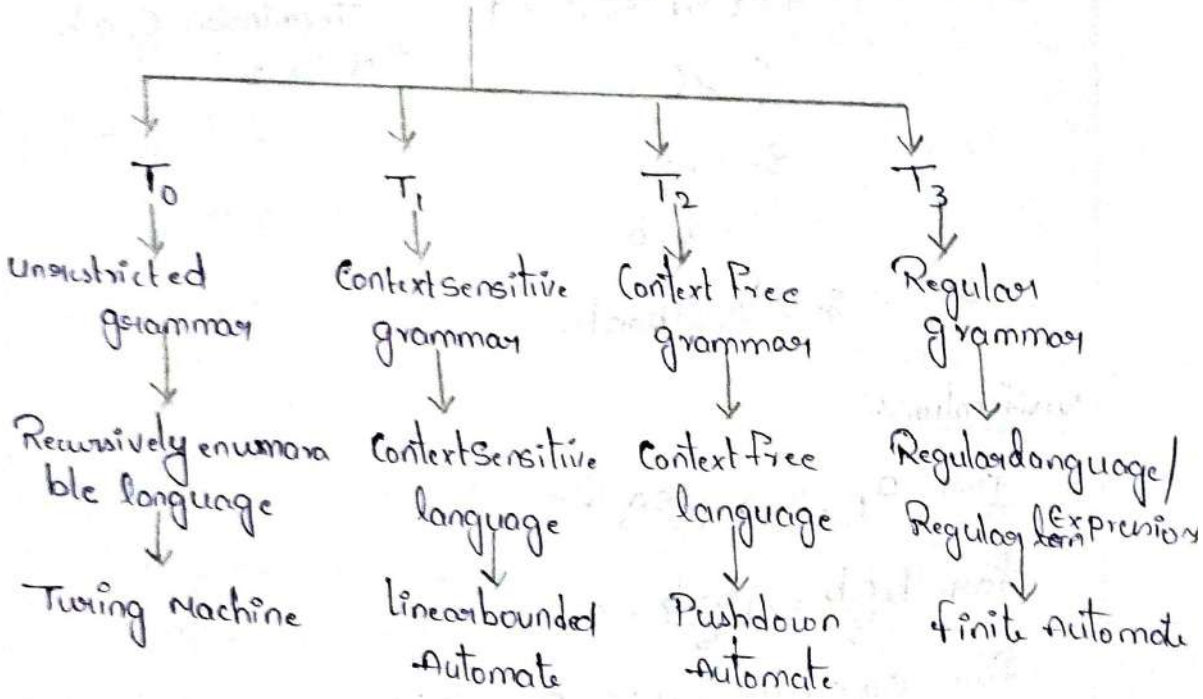
$L = aab$

Unit-2

Context Free Grammars & Push Down Automate :-

- Symbol
  - Alphabet (Σ)
    - String (w)
      - Language (L)
        - Grammar (G)

Grammer (G) = $\{V_N, \Sigma, P, S\}$ (or) $\{NT, T, P, S\}$



Context free grammers :-

→ Context free grammar is also called as Type-2 grammar.

→ They are of form $\alpha \longrightarrow \beta$

Where $\alpha \in V_N$ (Non-Terminals)

$|\alpha| = 1$

$\beta \in (V_N \cup \Sigma)^*$

→ context free grammar generate context free languages.

→ Context free grammar is accepted by push down Automate.

Q) Generate context free grammar for $L = \{wc w^R, w \in \{a,b\}^*\}$ (R = reverse)

A) Given language

$L = \{wc w^R, w \in \{a,b\}^*\}$

$wc w^R$ => for a: aca, $a^R = a$; for ab: abcba, $(ab)^R = ba$

Since, $w \in \{a,b\}^*$ which is a kleene closure, so, we can write it as

$w = \{\varepsilon, a, b, aa, bb, ab, ba, aaa, aab, aba, baa \ldots\}$

(c is terminal (unchangable))

$wcw^R = \{c, aca, bcb, aacaa, bbcbb, abcba, bacab \ldots\}$

As, we know, $G = \{V_N, \Sigma, P, S\}$

Terminals = c, a, b.

N.T = S.

$S \to c$

$S \to asa$ <- aca → asa because S → c. So replace 's' by 'c'.

$S \to bsb$

$S \to asa/bsb/c$.

Verification:-

For $a\underbrace{c}_{S}a \Rightarrow \underbrace{asa}_{S} = S$

For $b\underbrace{c}_{S}b \Rightarrow \underbrace{bsb}_{S} = S$

For $aa\underbrace{c}_{S}aa \Rightarrow a\underbrace{asa}_{S}a = \underbrace{asa}_{S} = S$

For $bb\underbrace{c}_{S}bb \Rightarrow b\underbrace{bsb}_{S}b = \underbrace{bsb}_{S} = S$

For $ab\underbrace{c}_{S}ba \Rightarrow a\underbrace{bsb}_{S}a = \underbrace{aba}_{S} = S$.

The grammar for the given language can be represented as

$G = \{V_N, \Sigma, P, S\}$

Where $V_N = \{S\}$

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

$P = S \to c$

$S \to asa$

$S \to bsb$

8) Generate grammar for expression $(0+1)^*01^*$

Given expression $(0+1)^*01^*$

Here, $(0+1)^*$ & 1^* can be repeated so, assume it as A & B

So, $\underbrace{(0+1)^*}_{A}\,0\,\underbrace{1^*}_{B}$

For A, $(0+1)^* = \{0,1\}^*$

$A = \{\varepsilon, 0, 1, 01, 10, 11, 00, 000, 0001, \dots\}$

$A \to \varepsilon$, $A \to 0A$, $A \to 1A$

For B, $1^* = \{\varepsilon, 1, 11, 111, 1111, \dots\}$

$B \to \varepsilon$, $B \to 1B$.

$A \to \varepsilon$, $A \to 0A$, $A \to 1A$

For 0 $\Rightarrow$ $0 \cdot \varepsilon = \underbrace{0A}_{A}$

For 11 $\Rightarrow$ $11 \cdot \varepsilon = 1\,\underbrace{1A}_{A}$

$= \underbrace{1\,\underbrace{1A}_{A}}_{A}$

For 01 $\Rightarrow$ $01 \cdot \varepsilon = 0\,\underbrace{1A}_{A}$

$= \underbrace{0\,\underbrace{1A}_{A}}_{A}$

So, the production rules are

$S \to AOB$

$B \to \varepsilon$

$B \to 1B$

$A \to \varepsilon$

$A \to 0A$

$A \to 1A$

For B, $B \to \varepsilon$, $B \to 1B$

For 1 $\Rightarrow$ $1 \cdot \varepsilon = \underbrace{1 \cdot B}_{B}$

For 11 $\Rightarrow$ $11 \cdot \varepsilon = 1\,\underbrace{1B}_{B}$

$= \underbrace{1\,\underbrace{1B}_{B}}_{B}$

The grammar generated from the given language is

$G = \{V_N, \Sigma, P, S\}$

$V_N = \{S, A, B\}$

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

P: $S \to AOB$

$A \to \varepsilon \mid 0A \mid 1A$

$B \to \varepsilon \mid 1B$

S : S

9) Generate grammar for language $G = (011+1)^*(01)^*$

The given language L is $(011+1)^*(01)^*$

Now, $\underbrace{(011+1)^*}_{A}\,\underbrace{(01)^*}_{B}$.

$\{011, 1\}^* = \{\varepsilon, 011, 1, 0111, 011011, 1011, 11, \cdots\}$

$A \to \varepsilon, \quad A \to 011A, \quad A \to 1A$

For B, $(01)^*$

$(01)^* = \{\varepsilon, 01, 0101, 010101, 01010101, \cdots\}$

$B \to \varepsilon, \quad B \to 01B.$

So, the Production rules are

$S \to AB$

$A \to \varepsilon$

$A \to 011A$

$A \to 1A$

$B \to \varepsilon$

$B \to 01B.$

The grammar generated from the given language is

$G = \{V_N, \Sigma, P, S\}$

$V_N = \{S, A, B\}$

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

P: $S \to AB$

$A \to \varepsilon \mid 011A \mid 1A$

$B \to \varepsilon \mid 01B.$

S: S.

Q) Generate grammar for language $abba\,(baa)^n\,aab\,(aaba$ $n > 0$

$L = \underbrace{abba}_{A}\ \underbrace{(baa)^n}_{B}\ \underbrace{aab}_{C}\ \underbrace{(aaba)^n}_{D}$

$S \to ABCD$

$A \to abba$

$C \to aab$

$(baa)^n$, $n>0$

$= \{baa, baabaa, baabaabaa, \ldots\}$ $n>0$

$B \rightarrow baa$

$B \rightarrow baaB$

$B = 1^+$

~~$B \rightarrow \varepsilon$~~ ✗

$B \rightarrow 1B$, $n>0$

$(aaba)^n$, $n>0$

$= \{aaba, aabaaaba, aabaaabaaaba \ldots\}$ $B = 1^+$.

baa, baa

$D \rightarrow aaba$

$D \rightarrow aabaD$

The Production rules are

$S \rightarrow ABCD$

$A \rightarrow aaba$

$C \rightarrow aab$

$B \rightarrow baa$

$B \rightarrow baaB$

$D \rightarrow aaba$

$D \rightarrow aabaD$

The grammar generated from the given language is

$G = \{V_N, \Sigma, P, S\}$

$V_N = \{S, A, B, C, D\}$

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

P: $S \rightarrow ABCD$

$A \rightarrow aaba$

$C \rightarrow aab$

$B \rightarrow baa \mid baaB$

$D \rightarrow aaba \mid aabaD$.

S: S

④ Generate grammar for language $L = a^n b^n c^m d^m$, $n, m \geq 1$

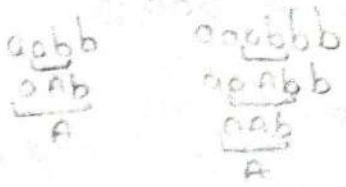
The given language $L = \underbrace{a^n b^n}_{A} \underbrace{c^m d^m}_{B}$, $n, m \geq 1$

For A, $a^n b^n$

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

$A \rightarrow ab$

$A \rightarrow aAb$



For B, $c^m d^m$

$= \{cd, ccdd, cccddd, ccccdddd \ldots\}$

$B \rightarrow cd$

$B \rightarrow cBd.$

The Production rules will be

$S \rightarrow AB$

$A \rightarrow ab$

$A \rightarrow aAb$

$B \rightarrow cd$

$B \rightarrow cBd$

The grammar generated for the language is

$G = \{V_N, \Sigma, P, S\}$

$V_N = \{S, A, B\}$

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

P: $S \rightarrow AB$

$A \rightarrow ab \mid aAb$

$B \rightarrow cd \mid cBd$

S: S.

Derivation & Parse trees:-

The derivation of any grammar is replacing the left hand side of symbol (ie., non terminal) of a Production rule with right hand side symbols. If there are more than

They are
a) Left hand derivation
b) Right derivation

Parse tree:-

The derivation can be represented in the form of tree, where left hand side symbols are is considered as parent node and right hand side symbols are considered as children nodes.

The Parse tree is sometimes called as derivation tree.

Q) Generate the derivation & Parse tree for the string 0100110 using the grammar $S \to 0S|1AA$, $A \to 0|1A|0B$, $B \to 1|0BB$

A) Given String to be generated 0100110.

Left derivation:-

$\Rightarrow S \to 0S$

because the first digit in the string is '0'.

Production rules.

$S \to 0S$
$S \to 1AA$
$A \to 0$
$A \to 1A$
$A \to 0B$
$B \to 1$
$B \to 0BB$.

$\Rightarrow S \to 0S \to 01AA$

because the next digit in the string after '0' is '1'. So, S is replaced by 1AA.

$\Rightarrow S \to 0S \to 01AA \to 010BA$

∴ left most 'A' is replaced by '0B'.

$\Rightarrow S \to 0S \to 01AA \to 0100BBA$

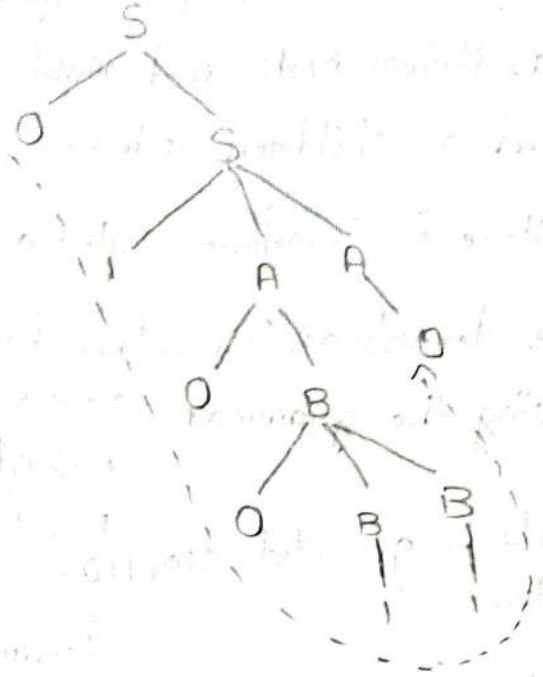
According to the required string & the Present/given Production rules, replace the non-terminals as per

$\Rightarrow S \rightarrow 0S \rightarrow 01AA \rightarrow 010BA \rightarrow 0100BBA \rightarrow 0100\underline{1}BA$

$\Rightarrow \rightarrow 01001\underline{1}A \quad \because B \rightarrow 1$

$\rightarrow 0100110. \quad \because A \rightarrow 0.$

Parse tree:-



Right derivation:-

$S \rightarrow 0\underline{S}$

$\rightarrow 0\underline{1AA} \quad \because S \rightarrow 1AA$

$\rightarrow 01A\underline{0} \quad \because A \rightarrow 1$

$\rightarrow 01\underline{0B}0 \quad \because A$ is replaced with $0B$

$\rightarrow 010\underline{0BB}0 \quad \because B \rightarrow 0BB$

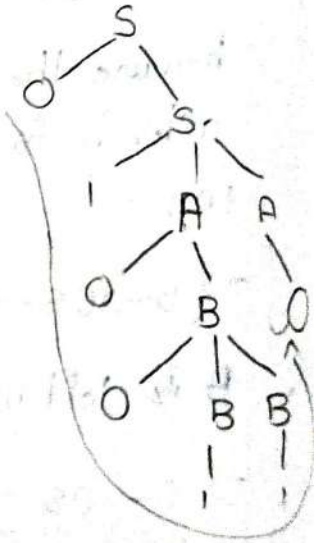
$\rightarrow 0100\underline{1}B0 \quad \because B \rightarrow 1$

$\rightarrow 01001\underline{1}0 \quad \because B \rightarrow 1$

$\rightarrow 0100B\underline{1}0 \quad \because B \rightarrow 1$

$\rightarrow 0100\underline{1}10 \quad \because B \rightarrow 1$

Parse tree :-



$\because$ In this, when two non-terminals occur, then we have

Q) Generate the string abbbb using grammar

$S \to aAB$, $A \to bBb$, $B \to A|\varepsilon$

Sol left most derivation :-

$S \to aAB$.

$\to abBbB$ <: A is replaced with bBb.

$\to ab\varepsilon bB$ <: $B \to \varepsilon$

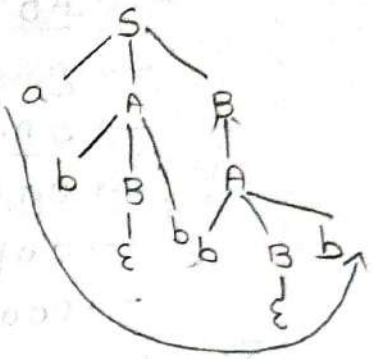
$\to abbA$ <: $B \to A$

$\to abbbBb$ <: $B \to bBb$

$\to abbb\varepsilon b$ <: $B \to \varepsilon$

$\to abbbb$ //.

Parse tree :-



Right most derivation :-

$S \to aAB$

$\to aAA$ <: $B \to A$

$\to aAbBb$ <: $A \to bBb$

$\to aAb\varepsilon b$ <: $B \to \varepsilon$

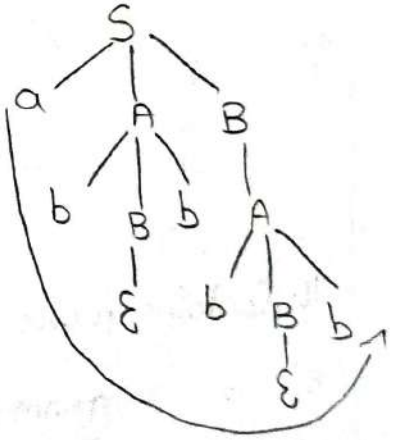
$\to aAbb$

$\to abBbbb$ <: $A \to bBb$

$\to ab\varepsilon bbb$ <: $B \to \varepsilon$

$\to abbbb$ //.

Parse tree :-



Ambiguity :-

A grammar is said to be Ambiguious if more than one parse tree is generated for that grammar.

Ex :-

determine the grammar is ambiguious or not for

Sol Given Production rules.

$S \to aS$ $\quad A \to AS$

$S \to AS$ $\quad A \to a$

$S \to A$

Left derivation:-

$S \to \boxed{AS}$

$\to \boxed{AS}S$

$\to a\boxed{S}S$

$\to a\boxed{AS}$

$\to aa\boxed{A}$

$\to aaa$

Right derivation:-

$S \to aS$

$\to a\boxed{A}$

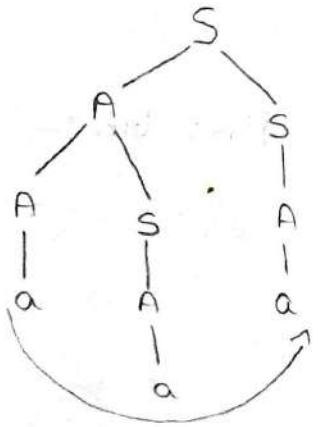
$\to a\boxed{AS}$

$\to aA\boxed{A}$

$\to aA\boxed{a}$

$\to aaa$

Parse tree



Parse tree.

It Contains more than one Parse tree. So, The given grammar is Ambiguious.

(9) check whether the given grammar is Ambiguious or not for generating string id + id * id.

$S \to S+S \mid S*S \mid id$

Sol Given Production functions

$S \to S+S$

$S \to S*S$

$S \to id$

$V_N = \{S\}$

$\Sigma = \{id, +, *\}$

left derivation:-

$S \rightarrow \underline{S+S}$

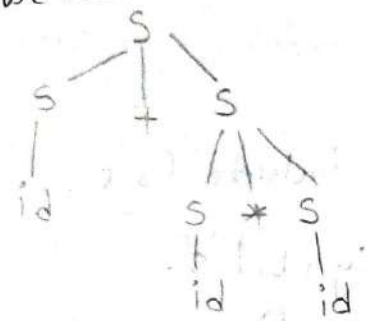
$\rightarrow \underline{id}+S$

$\rightarrow id+\underline{S*S}$

$\rightarrow id+\underline{id}*S$

$\rightarrow \underline{id}+id*id$

Parse tree



right derivation:-

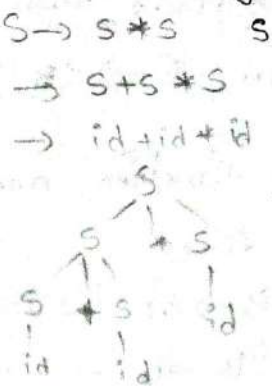
$S \rightarrow \underline{S*S}$

$\rightarrow S+\underline{S*S}$

$\rightarrow S+S*\underline{id}$

$\rightarrow S+\underline{id}*id$

$\rightarrow \underline{id}+id*id$



Parse tree

Simplification of CFG's:-

Simplification of CFG's include

i) Removal of useless Symbols.

→ non-generating

→ non-reachable

ii) Removal of unit productions

iii) Removal of null Productions.

Removal of useless symbols:-

→ Non-generating symbols are the symbols that doesnot Produce any terminal.

→ Non-reachable symbols are the symbols that cannot be reached from start symbol.

8) Remove useless Symbols for the grammar.

$S \to AC,\ S \to BA,\ C \to CB,\ C \to AC,\ A \to a,\ B \to ac|b.$

Sol Given Production functions are

$S \to AC$ $\quad A \to a$

$S \to BA$ $\quad B \to ac$

$C \to CB$ $\quad B \to b.$

$C \to AC$

As the non generating terminals include (S, C).

We can not remove the start symbol 's'.

So, we remove the 'c' Symbol from all the production rules, we get the production rules as.

$S \to BA$

$A \to a$

$B \to b$

A reaches 'S' & B reaches 's'. So, there is no non-reachable symbols.

∴ non-reachable symbols = { }

30/7/25

9) Remove use-less symbols for the grammar.

$S \to aB \mid bx$ $\quad A \to BAd \mid bsx \mid a$ $\quad B \to aSB \mid bBx$

$X \to SBD \mid aBx$ $\quad X \to ad$

Sol Given Production rules

~~$S \to aB$~~

$S \to bx$

~~$A \to BAd$~~

$A \to bsx$

$A \to a$

~~$B \to aSB$~~

~~$B \to bBx$~~

~~$X \to SBD$~~

~~$X \to aBx$~~

$X \to ad$

← Remove the 'B' from the rules because we cannot remove 's' because S is the start symbol.

non-generating symbols: {S, A, B, x}.

Since the Symbols that don't

The production functions become.

$S \rightarrow bx$
~~$A \rightarrow bSx$~~
~~$A \rightarrow a$~~
$x \rightarrow ad$

The non-reachable Symbol = {A}.
after removing the non-rechable Symbols, the final Production functions include $S \rightarrow bx$
$x \rightarrow ad$.

unit productions:-

unit Productions are the Productions that produce Single non-terminal.

Ex:- $S \rightarrow A$

Q) Remove unit Productions from the given CFG.:

$S \rightarrow AB$, $A \rightarrow E$, $B \rightarrow C$, $C \rightarrow D$, $D \rightarrow b$, $E \rightarrow a$

Given production functions

$S \rightarrow AB$ $C \rightarrow D$
$A \rightarrow E$ $D \rightarrow b$
$B \rightarrow C$ $E \rightarrow a$

The unit productions are $A \rightarrow E$
$B \rightarrow C$
$C \rightarrow D$

(i) $A \rightarrow E \Rightarrow A \rightarrow E \rightarrow a \Rightarrow A \rightarrow a$

(ii) $B \rightarrow C \Rightarrow B \rightarrow C \rightarrow D \rightarrow b \Rightarrow B \rightarrow b$

(iii) $C \rightarrow D \Rightarrow C \rightarrow D \rightarrow b \Rightarrow C \rightarrow b$

∴ After removing the unit Productions, the final Production functions are

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

Null Productions:-

Null Productions are the Productions that Produce empty 'ε'.

Q) Remove Null Productions from the given grammar.

$S \rightarrow aA \quad A \rightarrow b|\varepsilon$.

Sol Given Productions/Functions are

$S \rightarrow aA$

$A \rightarrow b$

$A \rightarrow \varepsilon$

To remove the null Production, the ε is produced from 'A'. Substitute the 'ε' in the 'S'

Then, we get

$S \rightarrow a\varepsilon$

$\boxed{S \rightarrow a}$

So, the new rules will be

~~$S \rightarrow aA$~~ $S \rightarrow aA$

~~$S \rightarrow a$~~ $S \rightarrow a$

~~$S \rightarrow \varepsilon$~~ $A \rightarrow b$

$\Rightarrow S \rightarrow aA|a.$

$A \rightarrow b.$

Q) Remove the null productions from the given grammar

$S \rightarrow ABaC \quad A \rightarrow BC \quad B \rightarrow b \quad B \rightarrow \varepsilon \quad C \rightarrow D \quad C \rightarrow \varepsilon \quad D \rightarrow d.$

Sol Given production rules are

$S \rightarrow ABaC$

$A \rightarrow BC$

$B \rightarrow b$

$B \rightarrow \varepsilon$

$C \rightarrow D$

$C \rightarrow \varepsilon$

The ε can be produced by B & C.

S & A produces BAC, So.

$S \rightarrow ABaC$

(i) $B = \varepsilon \Rightarrow S \rightarrow AaC$

(ii) $C = \varepsilon \Rightarrow S \rightarrow ABa$

(iii) $B = \varepsilon$ & $C = \varepsilon \Rightarrow S \rightarrow Aa$

$A \rightarrow BC$

(i) $B = \varepsilon \Rightarrow A \rightarrow C$

(ii) $C = \varepsilon \Rightarrow A \rightarrow B$

(iii) $B = \varepsilon$ & $C = \varepsilon \Rightarrow A \rightarrow \varepsilon$

The production rules are

$S \rightarrow ABaC \mid AaC \mid ABa \mid Aa$

$A \rightarrow BC \mid B \mid C \mid \varepsilon$

$B \rightarrow b$, $C \rightarrow D$, $D \rightarrow d$.

As $A \rightarrow \varepsilon$,

replace $A \rightarrow \varepsilon$ in rules, we get

$S \rightarrow ABaC \Rightarrow S \rightarrow BaC$

$S \rightarrow AaC \Rightarrow S \rightarrow a\varepsilon$

$S \rightarrow ABa \Rightarrow S \rightarrow Ba$

$S \rightarrow Aa \Rightarrow S \rightarrow a$

∴ The final production rules are

$S \rightarrow ABaC \mid AaC \mid ABa \mid Aa \mid BaC \mid aC \mid Ba \mid a$

$A \rightarrow BC \mid B \mid C$

$B \rightarrow b$

$C \rightarrow D$

$D \rightarrow d$.

Chomsky normal form:

A Context free grammar (CFG) is said to be in CNF (Chomsky normal form). if

1) A Non-terminal Produces Set of two-non terminals
2) A Non-terminal will Produce a single terminal.

i.e., $NT \rightarrow NT\ NT$

$NT \rightarrow T$

Process of Converting CFG into CNF:-

1) Remove empty, unit & useless Productions
2) If all the Productions in CFG are in format

$NT \rightarrow NT.NT$, $NT \rightarrow T$

declare the CFG as it is in CNF and stop.

3) Replace mixed Productions

Ex:- $A \rightarrow aB$ ⇓ $A \rightarrow CB$, $C \rightarrow a$

$A \rightarrow Ba$ ⇓ $A \rightarrow BC$, $C \rightarrow a$

$A \rightarrow abc$ ⇓ $A \rightarrow BC$, $B \rightarrow ab$

4) If RHS of any Production rule contains more than three Non-Terminals break that Production.

Ex:- $S \rightarrow \underbrace{AB}_{D}C$ ⇓ $S \rightarrow DC$, $D \rightarrow AB$

Q) Convert the given CFG into CNF.

$S \rightarrow aaaaS$ $S \rightarrow aaaa$

A) The given grammar is

According to chomsky normal form. the grammar should be in the form of

$NT \rightarrow NTNT$

$NT \rightarrow T$

But the given grammar is not in chomsky normal form

To convert the above grammar into CNF, let us take create a new rule

$\boxed{A \rightarrow a}$

The new rule is in the format of $NT \rightarrow T$.

In the given grammar we replace the terminal 'a' with non-terminal 'A'.

Hence the grammar becomes

① $\Rightarrow S \rightarrow AAAAS$

② $\Rightarrow S \rightarrow AAAA$

let us take rule-1 & convert it into CNF.

① $\Rightarrow S \rightarrow AAAAS$.

$S \leftrightarrow$

The above rule is not in CNF, so we create a new rule

$\boxed{B \rightarrow AS}$ $\because NT \rightarrow NT\ NT$ is in CNF.

and hence the grammar rule becomes.

$S \rightarrow AAAB$

Still the Production rule is in CNF.

Now, we consider two Non-Terminals each at a time to create two new rules.

$\boxed{C \rightarrow AA}$ $\because NT \rightarrow NT\ NT$ is in CNF

Hence, the grammar becomes

$S \rightarrow CD$ ∵ NT → NT NT is in CNF.

Now, we Consider rule-2 and Simplify it to CNF.

② ⇒ $S \rightarrow AAAA$

The given rule is not in the format of CNF but, we have a production rule

$C \rightarrow AA$

Hence, the above Production rule becomes

$S \rightarrow CC$ ∵ NT → NT NT is in CNF.

The grammar in CNF is

$S \rightarrow CD$ $\quad C \rightarrow AA$

$S \rightarrow CC$ $\quad D \rightarrow AB$.

$A \rightarrow a$

$B \rightarrow AS$

Q) Convert the CFG into CNF.

$S \rightarrow asa \mid bsb \mid a \mid b$

Sol) Given grammar is

① ⇒ $S \rightarrow asa$

② ⇒ $S \rightarrow bsb$

$S \rightarrow a$

$S \rightarrow b$

To convert the grammar into CNF, Consider two new rules

$A \rightarrow a$ ∵ NT → NT NT is in CNF.

$B \rightarrow b$ ∵ NT → NT NT is in CNF.

Consider the first rule,

① => S → aSa

Replace the 'a' with 'A' as we have the rule A → a

S → ASA

This is not in CNF again consider 'AS' as B.

B → **S → BA** ∵ NT → NT NT is in CNF.

C → AS ∵ NT → NT NT is in CNF.

Now, Consider the rule ②

② => S → bsb.

Replace b as B. based on the rule B → b

S → BSB.

Again consider D → BS ∵ NT → NT NT is in CNF

S → DS ∵ NT → NT NT is in CNF.

The grammar in CNF is:

- S → CA
- C → AS
- S → DB
- D → BS.
- S → a
- S → b
- A → a
- B → b

Greibach Normal form:

The Context free grammar is said to be in GNF. if it follows:

① NT Produces a Terminal & a string of Non-Terminals

② A Non-Terminal should produce a Single Terminal.

i.e., NT → T NT_1 NT_2 NT_3 ... => NT → T (NT)*

Process of Converting a CFG into GNF:

1) Remove all empty, unit Productions and useless Symbols.

2) Convert the CFG into CNF (Chomsky Normal form).

3) Rename all the Non-Terminals as $A_1, A_2, A_3 \ldots$

4) Check whether the given rules are in the format of

$$A_i \longrightarrow A_j A_k \ldots$$

5) If $i<j$, Convert the Production rule into GNF.

6) If $i>j$, we replace A_j with relevant Non-Terminals. Such that the final rule is in form

$$A_i \longrightarrow A_x A_y \ldots A_k \ldots$$

Such that $i<x$

7) If $i=j$ then it is called left recursive which is in format of $A \longrightarrow A\alpha \mid B$.

Which can be converted as $A \rightarrow \beta B \mid \beta$

$B \rightarrow \alpha B \mid \alpha$

8) Convert all the Production rules into GNF.

Q) Convert the grammar into GNF.

$S \rightarrow abSb \qquad S \rightarrow aa$

A) Given Grammar

$S \rightarrow absb \longrightarrow ①$

$S \rightarrow aaa \longrightarrow ②$

Now, convert the above grammar into CNF.

Rules of CNF $\Rightarrow$ NT $\longrightarrow$ NT NT

$A \rightarrow a$ ∴ NT → T is in CNF

$B \rightarrow b$ ∴ NT → T is in CNF

From ① => $S \rightarrow \underbrace{AB}_{C}\underbrace{SB}_{D}$. ∴ a & b are replaced with A & B

$C \rightarrow AB$

$D \rightarrow SB$

$S \rightarrow CD$

From ② => $S \rightarrow AA$ ∴ a is replaced with A.

The final Production rules are

$S \rightarrow CD$ $\quad A \rightarrow a$

$S \rightarrow AA$ $\quad B \rightarrow b$

$D \rightarrow SB$

$C \rightarrow AB$

Convert all non-terminals in the form of $A_1, A_2, A_3, \ldots$

Assume, $A_1 = S$, $A_2 = C$, $A_3 = D$, $A_4 = A$, $A_5 = B$.

Now, the Production rules can be written as.

$A_1 \rightarrow A_2A_3$ $\quad A_4 \rightarrow a$

$A_1 \rightarrow A_4A_4$ $\quad A_5 \rightarrow b$

$A_3 \rightarrow A_1A_5$

$A_2 \rightarrow A_4A_5$

=> check whether the given rules are in the format of

$A_i \rightarrow A_jA_k \ldots\ldots$ ∴ $i < j$.

The rules for greibach normal form includes.

=> $NT \rightarrow TNTNT \ldots$

=> $NT \rightarrow T$.

For $A_1 \rightarrow A_2A_3$ ∴ ex: $1 < 2$. So it can be converted into GNF.

$A_1 \rightarrow A_4A_5A_3$ ∴ $A_2 \rightarrow A_4A_5$

$A_1 \rightarrow aA_5A_3$ ∴ $A_4 \rightarrow a$

For $A_1 \rightarrow A_4 A_4$

$\boxed{A_1 \rightarrow a A_4}$ ∵ $A_4 \rightarrow a$

Which is in GNF.

For $A_3 \rightarrow A_1 A_5$

Since $3 > 1$ which is not in the form of $i<j$, we have to replace A_1.

$A_3 \rightarrow A_4 A_4 A_5$ ∵ $A_1 \rightarrow A_4 A_4$ & $3<4$ which is in the form of $i<j$.

$\boxed{A_3 \rightarrow a A_4 A_5}$ ∵ $A_4 \rightarrow a$

Which is in GNF.

For $A_2 \rightarrow A_4 A_5$ ∵ $2<4$

$\boxed{A_2 \rightarrow a A_5}$ ∵ $A_4 \rightarrow a$

Which is in GNF.

Hence the above grammar is converted into the Greibach Normal form (GNF).

Q) Convert the grammar into GNF.

$S \rightarrow AA|a \quad A \rightarrow SS|b$

A) Given Grammar is.

$S \rightarrow AA \quad A \rightarrow SS$

$S \rightarrow a \quad A \rightarrow b$

Hence there are no empty, unit productions & useless symbols.

And it also obeys the rules of CNF.

Then convert the grammar into the form of $A_1 A_2 A_3 \cdots$

Then consider

Then the Production rules will become

$A_1 \rightarrow A_2 A_2$

$\boxed{A_1 \rightarrow a}$

$A_2 \rightarrow A_1 A_1$

$\boxed{A_2 \rightarrow b}$

Consider $A_2 \rightarrow A_1 A_1$

here $i > j$ Which is not in the form of $i < j$

$A_2 \rightarrow A_1 A_1$

A_1 can be replaced in two ways.

$A_1 \rightarrow A_2 A_2$ & $A_1 \rightarrow a$

$A_2 \rightarrow A_2 A_2 A_1$ & $\boxed{A_2 \rightarrow aA_1}$ ✓ Which is in GNF.

Now, $A_2 \rightarrow A_2 A_2 A_1$

$\because 2 = 2 \Rightarrow i = j$. So, it is in the form of left recursive.

divide this Production rule in the form of

$A \rightarrow A\alpha | \beta$.

and the convert that rule into

$A \rightarrow \beta B | \beta$ & $B \rightarrow \alpha B | \alpha$

$A_2 \rightarrow A_2 \underbrace{A_2 A_1}_{\alpha} | \underbrace{aA_1}_{\beta_1} | \underbrace{b}_{\beta_2}$. $\because$ $A_2 \rightarrow A_2 A_2 A_1$, $A_2 \rightarrow aA_1$, $A_2 \rightarrow b$

For $B_1 \Rightarrow$ $A_2 \rightarrow A_2 \underbrace{A_2 A_1}_{\alpha} | \underbrace{aA_1}_{\beta}$

Convert it into. $A \rightarrow \beta B | \beta$

$A_2 \rightarrow aA_1 B_1 | aA_1$

For $B_2 \Rightarrow$ $A_2 \rightarrow A_2 \underbrace{A_2 A_1}_{\alpha} | \underbrace{b}_{\beta}$

Convert it into $A \rightarrow \beta B | \beta$

Now,

For $A_2 \rightarrow A_2A_1 \mid aA_1 \mid b$

$\alpha \quad \beta \quad B_2$

The rules will become

$A_2 \rightarrow aA_1B_1 \mid aA_1$

$A_2 \rightarrow bB_1 \mid b$

$B_1 \rightarrow A_2A_1B_1 \mid A_2A_1$

For final, $\boxed{A_2 \rightarrow aA_1B_1 \mid aA_1 \mid bB_1 \mid b}$

$B_1 \rightarrow A_2A_1B_1 \mid A_2A_1$

Here, $B_1 \rightarrow A_2A_1B_1 \mid A_2A_1$ are not in GNF.

So, Consider $B_1 \rightarrow A_2A_1B_1$

Replace A_2 with $A_2 \rightarrow aA_1B_1 \mid aA_1 \mid bB_1 \mid b$

Then,

$\boxed{B_1 \rightarrow aA_1B_1A_1B_1 \mid aA_1A_1B_1 \mid bA_1B_1B_1 \mid bA_1B_1}$

Also, $B_1 \rightarrow A_2A_1$

Replacing A_2 we get

$\boxed{B_1 \rightarrow aA_1B_1A_1 \mid aA_1A_1 \mid bB_1A_1 \mid bA_1}$

For rule $A_1 \rightarrow A_2A_2$

$i<j$, So directly Substitute A_2 we get

$\boxed{A_1 \rightarrow aA_1B_1A_2 \mid aA_1A_2 \mid bB_1A_2 \mid bA_2}$

Hence, the grammar is Converted into Greibach Normal form (GNF).

Q) convert the grammar into CNF.

Pushdown Automate :-

A pushdown Automate is defined as set of Seven tuples.

$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, q_f)$

Where Q = Set of States

Σ = Set of Input Symbols.

Γ = Set of stack Symbols

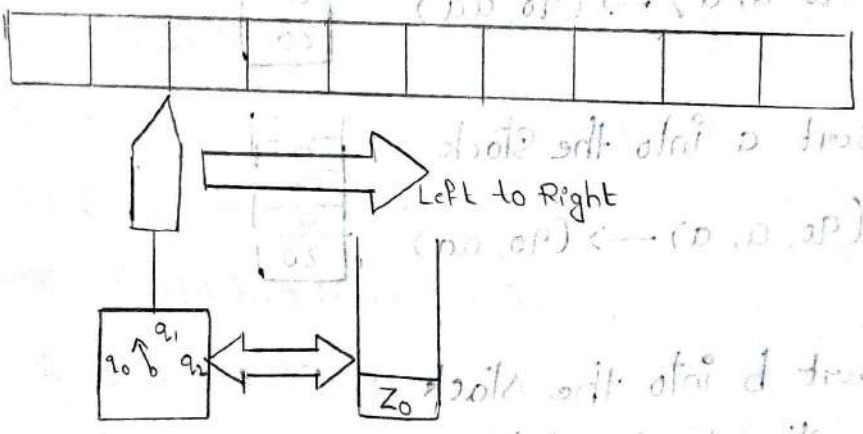
δ = Transition function

q_0 = Start State | initial state | beginning state

Z_0 = The stack Symbol (always stays in bottom)

q_f = final state.

Mechanical Diagram:-



Q) Design a pushdown Automate for language

$L = a^n b^n, n \geq 1$

A) Given language $L = a^n b^n, n \geq 1$

language 'L' starts with '1'.

So, the language will become,

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

here, consider three states q_0 → for push-a (inserting)

q_1 → for pop (when insert 'b' we pop 'a')

q_2 → final state. - ε

Whenever we come across 'b', we pop 'a' from the stack.

The default value of in stack is Z_0.

Consider aaa bbb

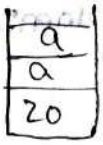
→ insert a into the stack:

$\delta(q_0, a, Z_0) \rightarrow (q_0, aZ_0)$

(the state we are in → the top most state in the stack; the input we are giving)

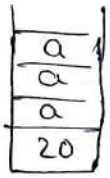
→ insert a into the stack

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



→ insert a into the stack

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



→ insert b into the stack

When 'b' gets inserted the 'a' gets poped from the stack

$\delta(q_0, b, a) \rightarrow (q_1, \varepsilon)$

→ insert 'b' into the stack

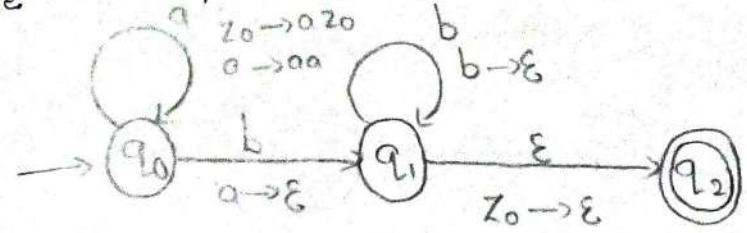
$\delta(q_1, b, a) \rightarrow (q_1, \varepsilon)$

→ insert 'b' into the stack

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

→ insert 'ε' into the stack to pop

The final pushdown automate is



Q) construct a pushdown automate for language

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

A) Given language $L = a^n b^{2n}$ & $n \geq 1$

The language generated is $L = \{abb, aabbbb, aaabbbbbb \dots\}$

Here the three states $q_0 \to aa$ <: when we insert 'a' in a state replace it with 'aa'

$q_1 \to b$ <: pop 'a' when we give input as 'b'

$q_2 \to$ final - ε.

Here we have to replace 'a' with 'aa' so that there won't be any problem while poping 'a' from stack.

The states include:

a is replaced with 'aa'

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

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

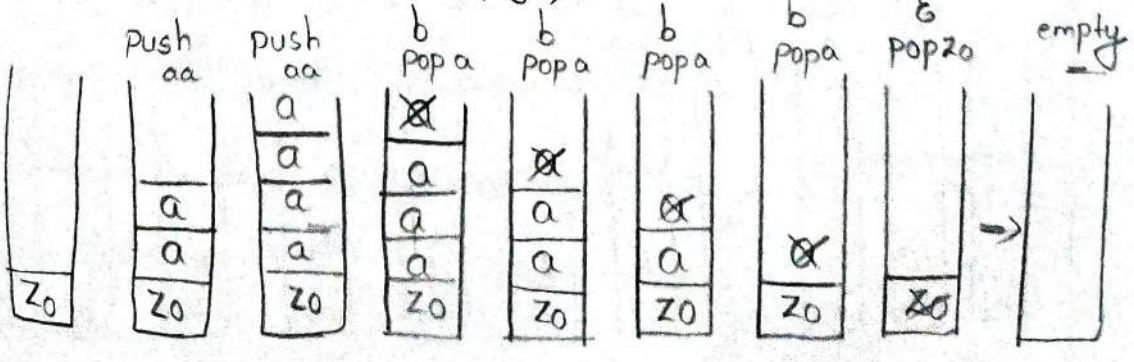
$\delta(q_0, b, a) \to (q_1, \varepsilon)$

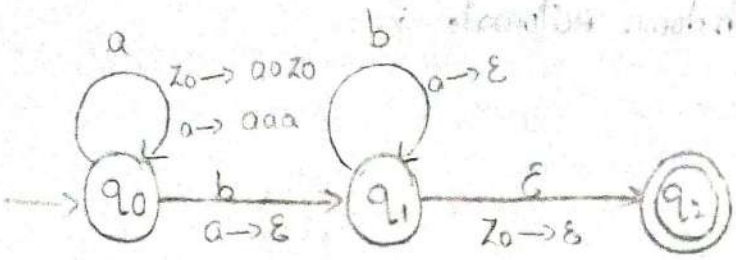
$\delta(q_1, b, a) \to (q_1, \varepsilon)$

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

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

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





Q) design pushdown automate for language

$L = a^m b^m c^n d^n, \quad n, m \geq 1$

Sol) Given $L = a^m b^m c^n d^n$

language $L = \{abcd, abccdd, aabbcd, aabbccdd \ldots\}$

$n, m = 1$ $\quad m=1, n=2$ $\quad m=2, n=1$ $\quad m=2, n=2$

Consider string
aabbccdd

States $\Rightarrow$ $q_0 \to$ push a
$q_1 \to$ b $\to$ pop a
$q_2 \to$ push c
$q_3 \to$ d $\to$ pop c
$q_4 \to$ final

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

$\Rightarrow \delta(q_0, a, a) \to (q_0, aa)$

$\Rightarrow \delta(q_0, b, a) \to (q_1, \varepsilon)$

$\Rightarrow \delta(q_1, b, a) \to (q_1, \varepsilon)$

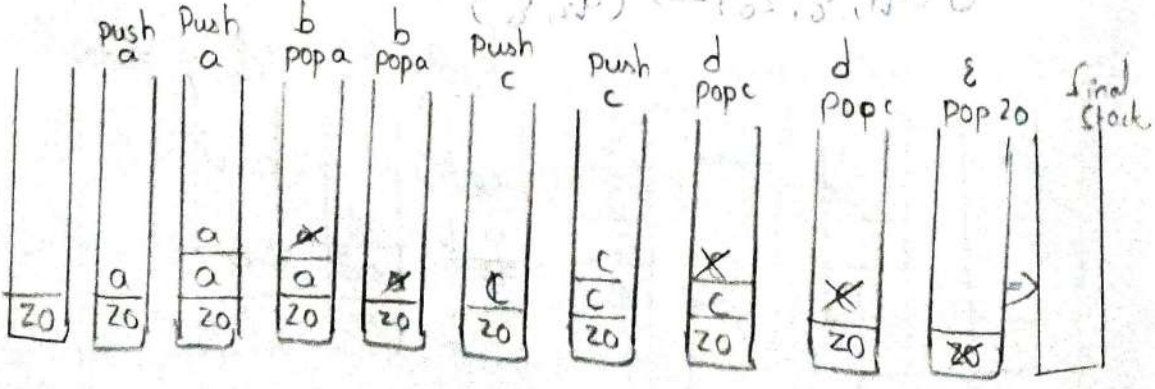
$\Rightarrow \delta(q_1, c, z_0) \to (q_2, cz_0)$

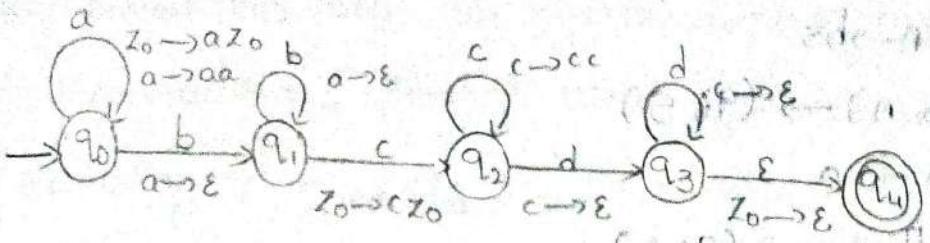
$\Rightarrow \delta(q_2, c, c) \to (q_2, cc)$

$\Rightarrow \delta(q_2, d, c) \to (q_3, \varepsilon)$

$\Rightarrow \delta(q_3, d, c) \to (q_3, \varepsilon)$

$\Rightarrow \delta(q_3, \varepsilon, z_0) \to (q_4, \varepsilon)$





Equivalence of context free grammars & pushdown automata:-

Steps for converting CFG to PDA:-

step 1:- Convert the given grammar into GNF.

step 2:- Consider the only one state q_i

step 3:- If the grammar rule is in the form of $A \to aB$ then the transition function can be written as

$$\delta(q_i, a, A) \to (q_i, B).$$

step 4:- If the grammar rule is in the format of $A \to a$ then the ~~production~~ transition function can be written as $\delta(q_i, a, A) \to (q_i, \varepsilon)$

step-5:- we add the final transition function for rule to the given following transition function

$$\delta(q_i, \varepsilon, Z_0) \to (q_0, \varepsilon)$$

$A \to \varepsilon$

$\delta(q_i, \varepsilon, A) \to (q_i, \varepsilon)$

Q) Convert the given CFG into PDA:

$S \to aAA \quad A \to aS \quad A \to bS \quad A \to a$

A) Given Grammar

$S \to aAA \quad A \to aS \quad A \to bS \quad A \to a$

based on the given grammar

$S \to aAA$ Consider AA as B, & a as a.

Substitute the a & B. in $\delta(q_i, a, A) \to (q_i, B)$.

$A \to aB$

$\delta(q_i, a, S) \to (q_i, AA)$.

For $A \to as$

For $A \to bS$

$(q_i, b, A) \to (q_i, S)$

For $A \to a$

$(q_i, a, A) \to (q_i, \varepsilon)$.

by default $(q_i, \varepsilon, Z_0) \to (q_i, \varepsilon)$

Q) Convert the CNF into PDA

$S \to AB$, $A \to CD$, $B \to b$, $C \to a$, $D \to a$

Given Grammar

$S \to AB$, $A \to CD$, $B \to b$, $C \to a$, $D \to a$

Check whether they are in $NT \to T\ NT\ NT$ or $NT \to T$.

$S \to AB$

$S \to CDB$ ← $A \to CD$

$S \to aDB$ ← $C \to a$

$A \to CD$

$A \to aD$ ← $C \to a$

The grammar is

$S \to aDB$, $A \to aD$, $B \to b$, $C \to a$, $D \to a$.

Then,

$\delta(q_i, a, S) \to (q_i\ DB)$ ← For $S \to aDB$

$\delta(q_i, a, A) \to (q_i, D)$ ← For $A \to aD$

$\delta(q_i, b, B) \to (q_i, \varepsilon)$ ← For $B \to b$

$\delta(q_i, a, C) \to (q_i, \varepsilon)$ ← For $C \to a$

$\delta(q_i, a, D) \to (q_i, \varepsilon)$ ← For $D \to a$

$\delta(q_i, \varepsilon, Z_0) \to (q_i, \varepsilon)$ ← by default.

Q) Convert the Given grammar into PDA.

$S \to 0S1 / A$, $A \to 1A0 | S | \varepsilon$

Sol) Given Grammar.

$S \to 0S1$ $A \to 1A0$

To Convert into CNF, we need to remove empty, Unit productions & ~~start~~ S useless Symbols.

→ Removing 'ε' Productions, we get ∴ A→ε is removed.

S→ OS1 A→1A0

S→A A→S.

here S→A & A→S are two unit productions,

S→A

S→1A0|S ∴ A→1A0, A→S.

S→1A0|0S1 ∴ S→0S1

A→S

A→1A0|0S1 ∴ S→1A0|0S1.

the production rules are S→1A0|0S1

A→1A0|0S1

To Convert it into CNF:

S→1A0|0S1 →① Consider B→0

A→1A0|0S1 →② C→1

① ⇒ S→CAB|BSC B→0

A→CAB|BSC C→1

For S→ CAB (D) S→ BSC (E)

S→DB & D→CA S→EC & E→BS

For A→ CAB (D) A→ BSC (E)

A→DB A→EC

The Grammar is

S→DB B→0

S→EC C→1

D→CA

E→BS

Consider

① $S \to DB$ ② $S \to EC$ ③ $D \to CA$

$S \to CAB$ $S \to BSC$ ∴ $E \to BS$ $D \to 1A$ ∴ $C \to 0$

$S \to 1AB$ $S \to 0SC$ ∴ $B \to 0$

④ $E \to BS$

$E \to 0S$ ∴ $B \to 0$

⑤ $A \to DB$

$A \to CAB$ ∴ $D \to CA$

$A \to 1AB$ ∴ $C \to 1$

⑥ $A \to EC$

$A \to BSC$ ∴ $E \to BS$

$A \to 0SC$ ∴ $B \to 0$

⑦ $B \to 0$ & ⑧ $C \to 1$

Hence, D & E are non-reachable symbols. So we remove those symbols. The grammar is

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

$S \to 0SC$ $A \to 0SC$ $C \to 1$

The rules are

for $S \to 1AB \Rightarrow (q_i, 1, S) \to (q_i, AB)$

for $S \to 0SC \Rightarrow (q_i, 0, S) \to (q_i, SC)$

for $A \to 1AB \Rightarrow (q_i, 1, A) \to (q_i, AB)$

for $A \to 0SC \Rightarrow (q_i, 0, A) \to (q_i, SC)$

for $B \to 0 \Rightarrow (q_i, 0, B) \to (q_i, \varepsilon)$

for $C \to 1 \Rightarrow (q_i, 1, C) \to (q_i, \varepsilon)$

by default $\Rightarrow (q_i, \varepsilon, Z_0) \to (q_f, \varepsilon)$.

Unit- 3 :-

Turing Machines & Introduction to Compilers

Turing Machines:-

A Turing Machine is a machine that accepts recursively Enumerable* (grammar)* languages. Turing machine accepts all languages.

A Turing machine can be defined as set of 7 Tuples.

i.e., $M = \{Q, \Sigma, \Gamma, \delta, q_0, B, F\}$

where Q = Set of states

Σ = Set of Input Symbols

Γ = Set of Input Tape Symbols.

δ = Transition function.

q_0 = Starting state / Initial state

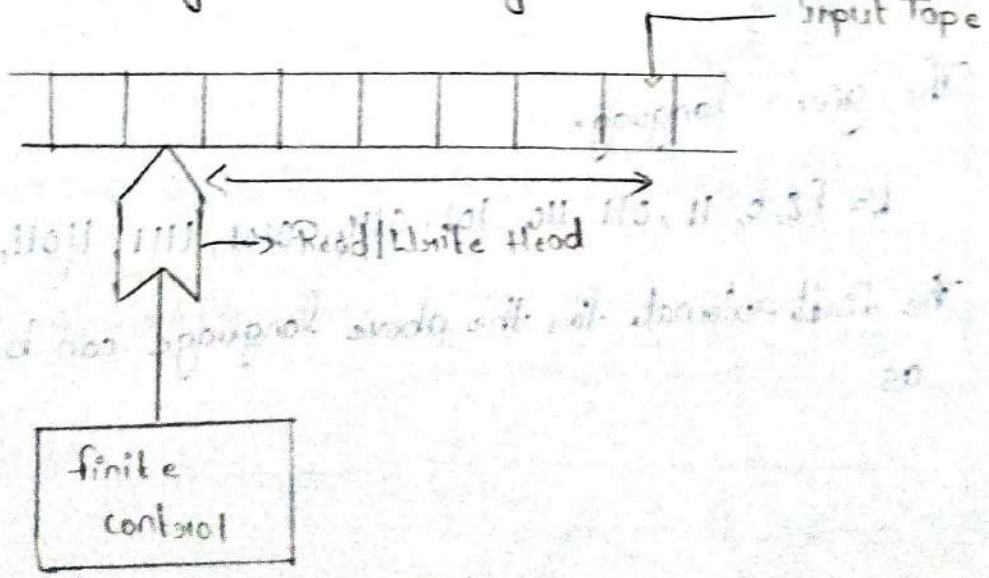
B = Blank symbol belongs to Γ.

F = Set of final states.

The transition function 'δ' is the mapping of

$$Q \times \Sigma \longrightarrow (Q \times \Gamma \times \{L \times R \times H\})$$

Mechanical diagram of Turing Machine:-



Representation of Turing Machine:-

We can represent a Turing Machine in three ways

[2] Transition Diagram

[3] Transition Table.

Instantaneous Description:-

Instantaneous Description (ID) of a turing machine describes

a) The contents of all cells of input tape.

b) The ~~current~~ cell i.e., currently being scanned by read/write head.

c) The current state of the Turing Machine at any instant of time.

Language accepted by turing machine:-

→ A Turing machine accepts all languages.

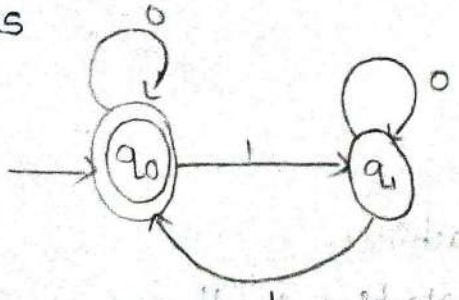
→ It can also perform mathematical operations like Addition, Subtraction, Multiplication, Division, Power function, Remainder etc.

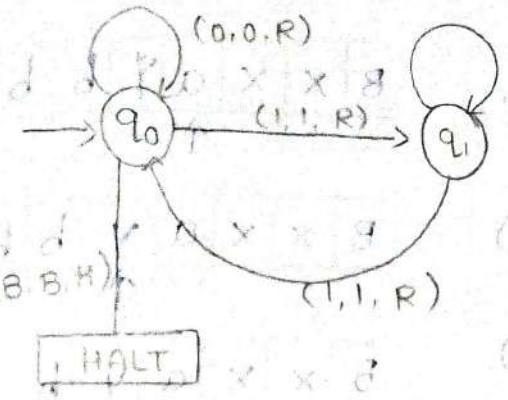
Q) Construct a Turing Machine that accepts all strings with even number of one's over the alphabet $\Sigma = \{0, 1\}$

The given language

$L = \{\varepsilon, 0, 11, 011, 110, 101, 0110, 0101, 1111, 11011, 011011, \ldots\}$

The finite automata for the above language can be constructed as





Q) Design a Turing Machine that accepts the language $L = \{a^n b^n, n \geq 1\}$ Show ID for string "aaabbb"

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

The language generated

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

Consider the string "aaabbb" to construct the Turing machine

B	a	a	a	b	b	b	B

$\delta(q_0, a) \rightarrow \delta(q_1, X, R)$

B	X	a	a	b	b	b	B

$\delta(q_1, a) \rightarrow \delta(q_1, a, R)$

B	X	a	a	b	b	b	B

$\delta(q_1, a) \rightarrow \delta(q_1, a, R)$

B	X	a	a	b	b	b	B

$\delta(q_1, b) \rightarrow \delta(q_2, Y, L)$

B	X	a	a	Y	b	b	B

$\delta(q_2, a) \rightarrow \delta(q_2, a, L)$

B	X	a	a	Y	b	b	B

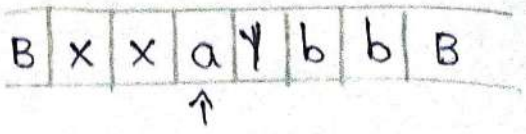
$\delta(q_2, a) \rightarrow \delta(q_2, a, L)$

B	X	a	a	Y	b	b	B

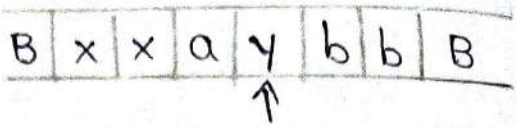
$\delta(q_2, X) \rightarrow \delta(q_0, X, R)$

B	X	a	a	Y	b	b	B

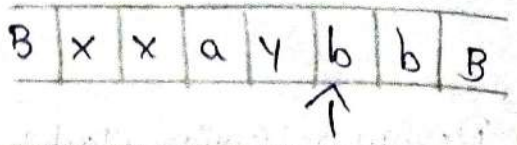
$\delta(q_0, a) \rightarrow \delta(q_1, X, R)$



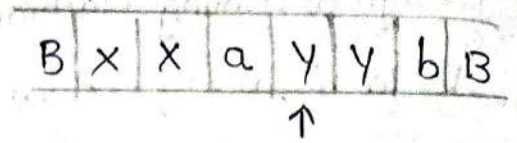
$\delta(q_1, a) \rightarrow \delta(q_1, a, R)$



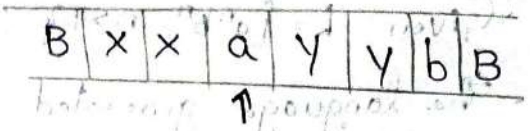
$\delta(q_1, Y) \rightarrow \delta(q_1, Y, R)$



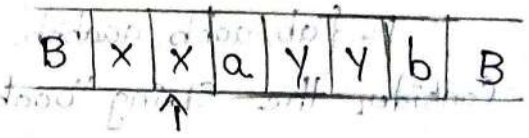
$\delta(q_1, b) \rightarrow \delta(q_2, Y, L)$



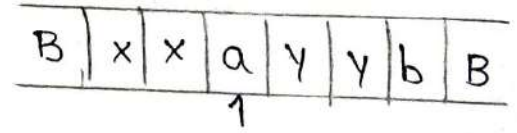
$\delta(q_2, b) \rightarrow \delta(q_2, Y, L)$



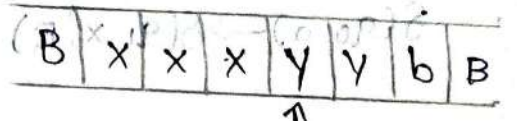
$\delta(q_2, a) \rightarrow \delta(q_2, a, L)$



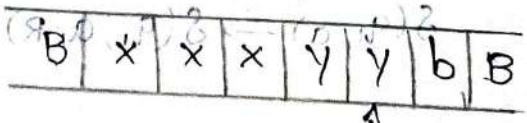
$\delta(q_2, x) \rightarrow \delta(q_0, x, R)$



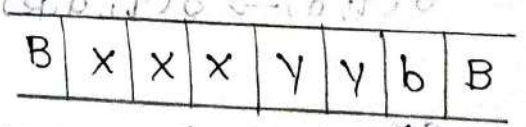
$\delta(q_0 a) \rightarrow \delta(q_1, X, R)$



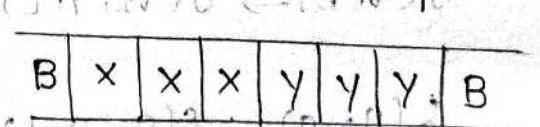
$\delta(q_1, Y) \rightarrow \delta(q_1, Y, R)$



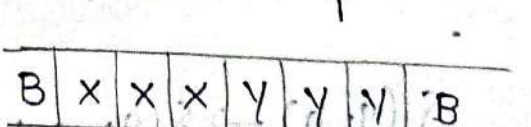
$\delta(q_1, Y) \rightarrow \delta(q_1, Y, R)$



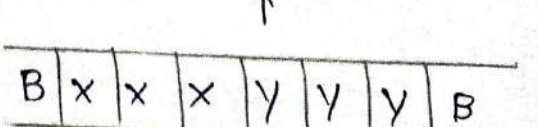
$\delta(q_1, b) \rightarrow \delta(q_2, Y, L)$



$\delta(q_2, Y) \rightarrow \delta(q_2, Y, L)$



$\delta(q_2, Y) \rightarrow \delta(q_2, Y, L)$



$\delta(q_2$

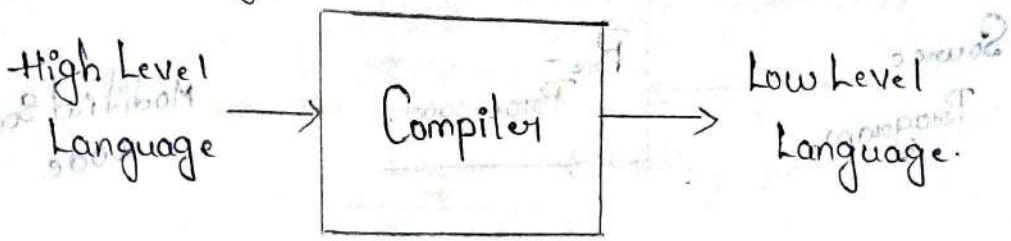
Part - B

Introduction to Compilers:-

Compiler:-

Compiler is a Software that Converts high level language into low-level language.

→ [The input gi]

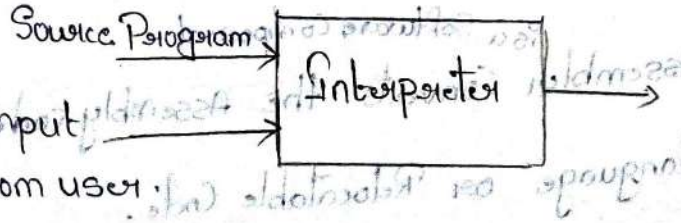


→ The input given to the Compiler is "Source Program".

→ The output generated by the Compiler is "object code" / "Target code.".

Interpreter:-

Interpreter is a Software Component that accepts Source Program and input from the users then converts Source Program into target code line by line and at the end Produces the final output.



Note:-

→ The advantage of Compiler is that it Provides faster execution speed.

→ The advantage of using Interpreter is we it can easily identify errors as the Program is executed line by line

Cousins of Compiler:-

1) Pre Processor:-

The Pre Processor Creates modified source code from the source program. It Performs two operations
i) Add Code related to header files.
ii) replaces macros.


```
#include <stdio.h>
#define { PI  3.14
int main (){
    int r=2, area;
    area= PI* r*r;
    Printf ("%d", area);
    return 0;
}
```

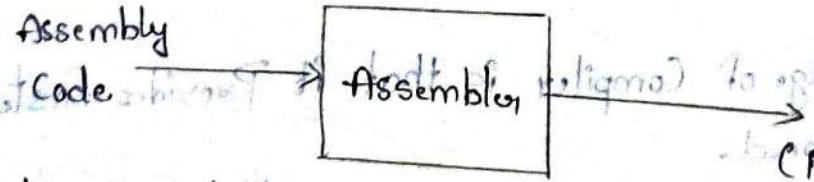
(macros)

stdio.h file code

```
int main(){
    int r= 2, area;
    area = 3.14* r * r;
    Printf ("%d", area);
    return 0;
}
```

2) Assembler:-

The Assembler is a Software Component Converts the Assembly code into the Binary language or Relocatable Code.

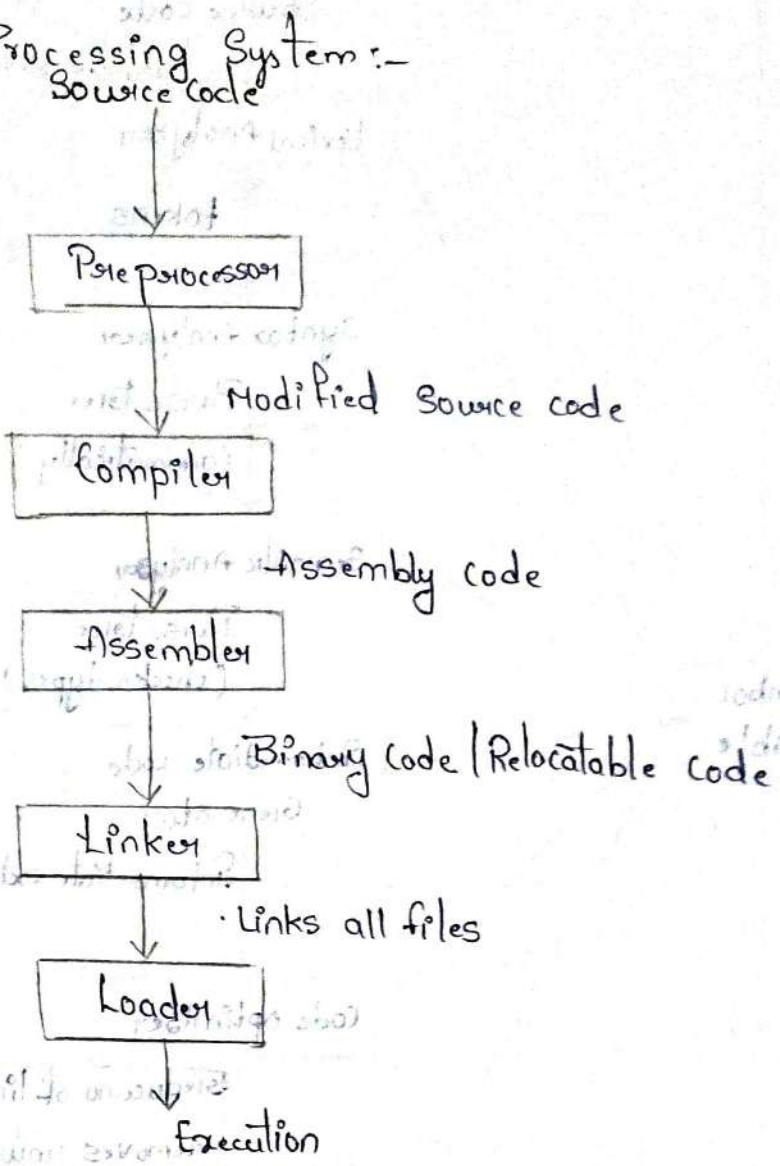


3) Linker & Loader:-

Linker Combines all the different files of Relocatable

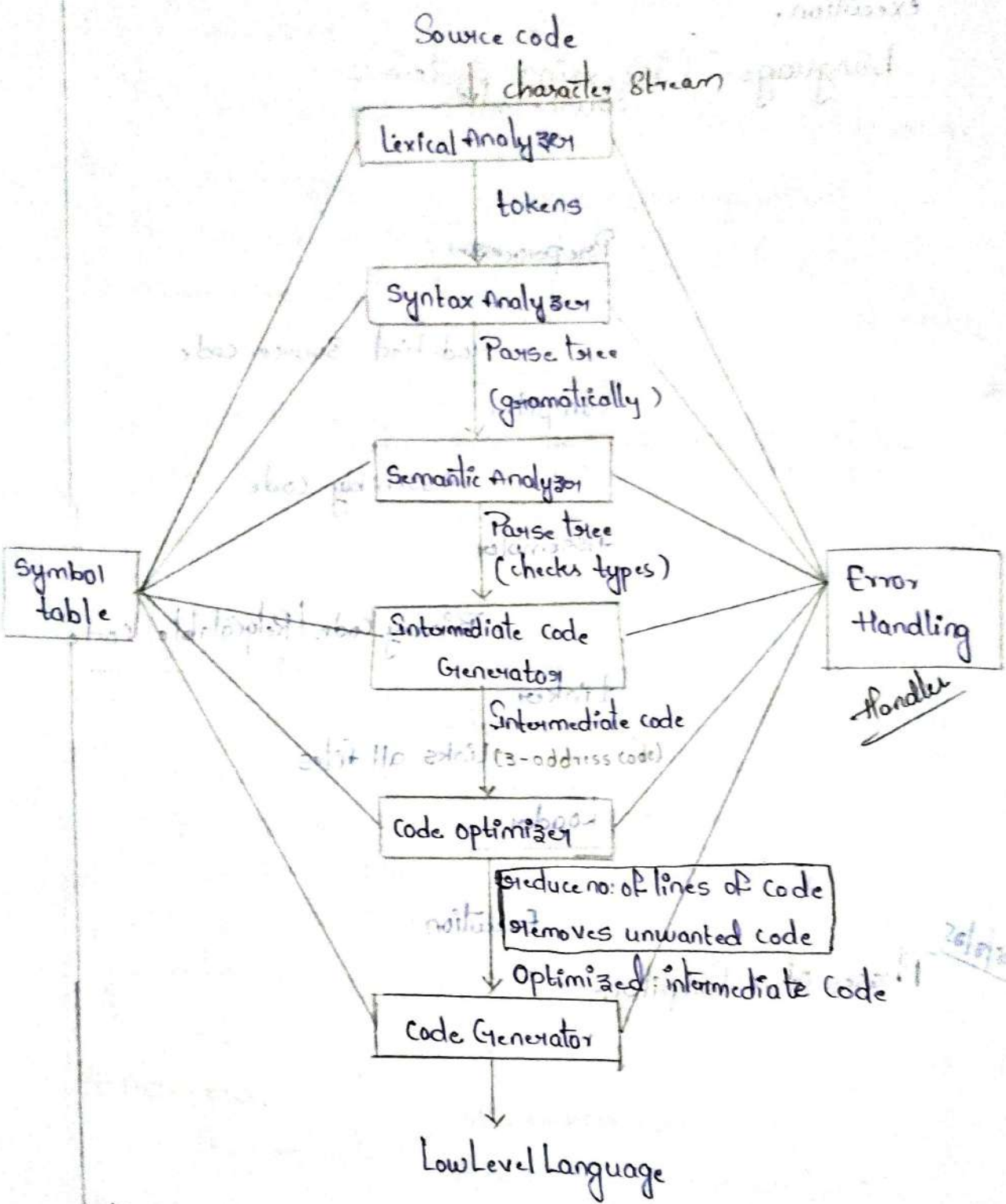
Loader Sends the final Program to cpu for execution.

Language Processing System:-



5/8/25

Phases of a Compiler:-

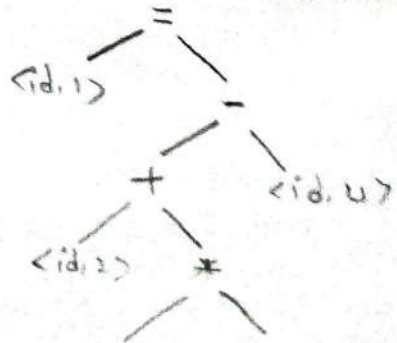


Ex:-

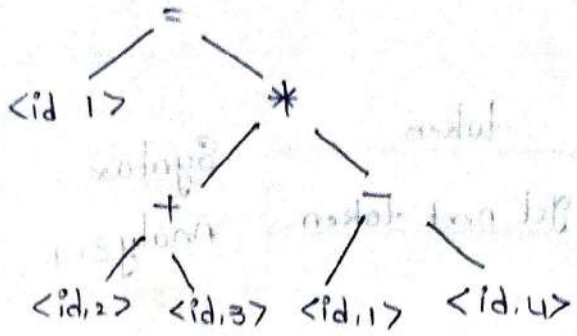
$c = a + b * c - d$

<id,1> <id,2> <id,3> <id,1> <id,4>

Parse-tree:-



G = (a+b) * (c - d)

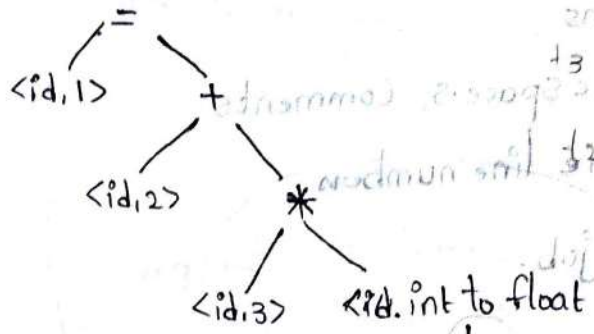


19/25

Example:-

total = number1 + number2 * 5

<id,1> <id,2> <id,3>



t_1 = int to float(5)

t_2 = <id,3> * t_1

t_3 = <id,2> + t_2

<id,1> = t_3

} Intermediate code generator

⇒ t_2 = <id,3> * 5.0

<id,1> = <id,2> + t_2

} Code optimizer ← reduces no of lines of code
removes unwanted code

Low-level language:-

LDF R1 id3 (register) — this is in RAM

MULF R1 R1 # 5.0

LDF R2 id2

ADDF R2 R2 R1

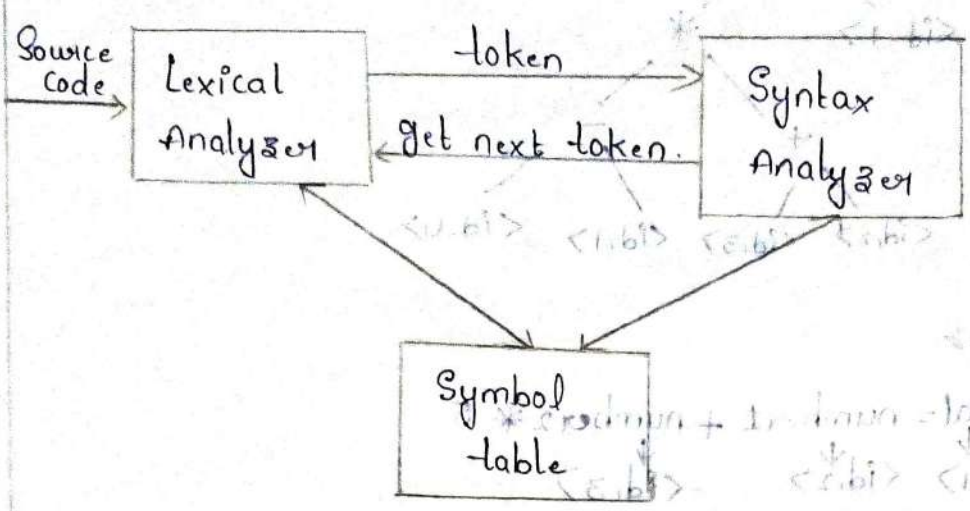
STF

LDF - load float
MULF - Multiply float
ADDF - Addition of float
STF - Store float

id1, id2, id3 → these are present in RAM

Registers Present

Lexical Analyzer:-



1) Generates tokens
2) Removes white spaces, Comments
3) Remembers the line numbers.
4) Pre Processing job.- When the Pre-processor is absent, then the lexical analyzer do the job of the Preprocessor. — adds header code
replaces macros

Token, Lexeme, Pattern:-

Consider, `int add (int a) { return a+10; }`

identifiers → Function names, Variables

Token	Lexeme	Pattern
Keyword	int	int
identifier	add	letter (–, number, letter)*
Special char	(	(
identifier	a	letter (–, number, letter)*
Special char	)	)
Special char	{	{
Keyword	return	return
Operator	+	+

Special char ; ;

Special char } }.

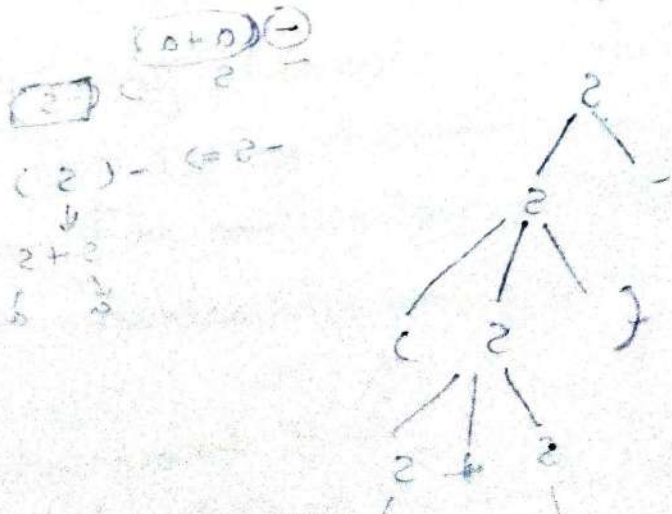
Input buffering :-

Consider $a = n_1 + n_2$;

take buffer of size 6

a	=	n	1	+	n

This will fail to express the whole condition.
So, we use double buffer / two buffers to avoid this problem.



03/09/25

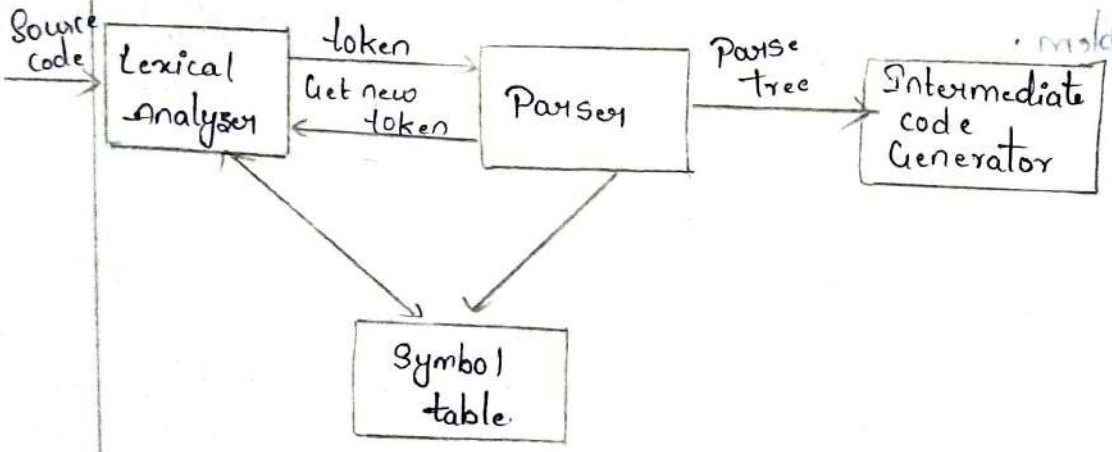
Unit-4:

Parsers & Intermediate Code Generator:-

Parsing:-

Parsing is a technique that takes string of tokens and checks for Grammatical Structure of Source Program using Grammar of Source language.

If it is grammatically Correct Syntax analyzer is going to Produce Parse tree otherwise it will throw through errors.



Construct Parsetree for (grammar), String $-(a+a)$.

G: $S \rightarrow S+S \mid S*S \mid -S \mid (S) \mid a$

To get $-(a+a)$

$-(a+a)$

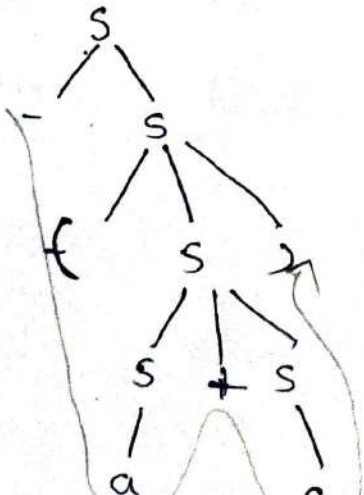
$-S \Rightarrow -S$

$-S \Rightarrow -(S)$

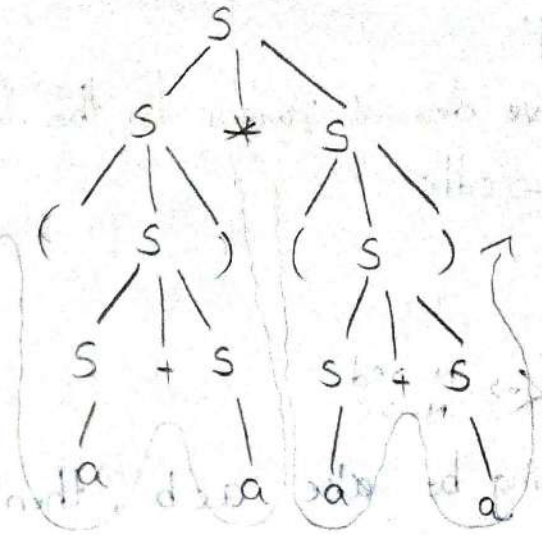
$\Downarrow$

$S+S \Rightarrow -(a+a)$

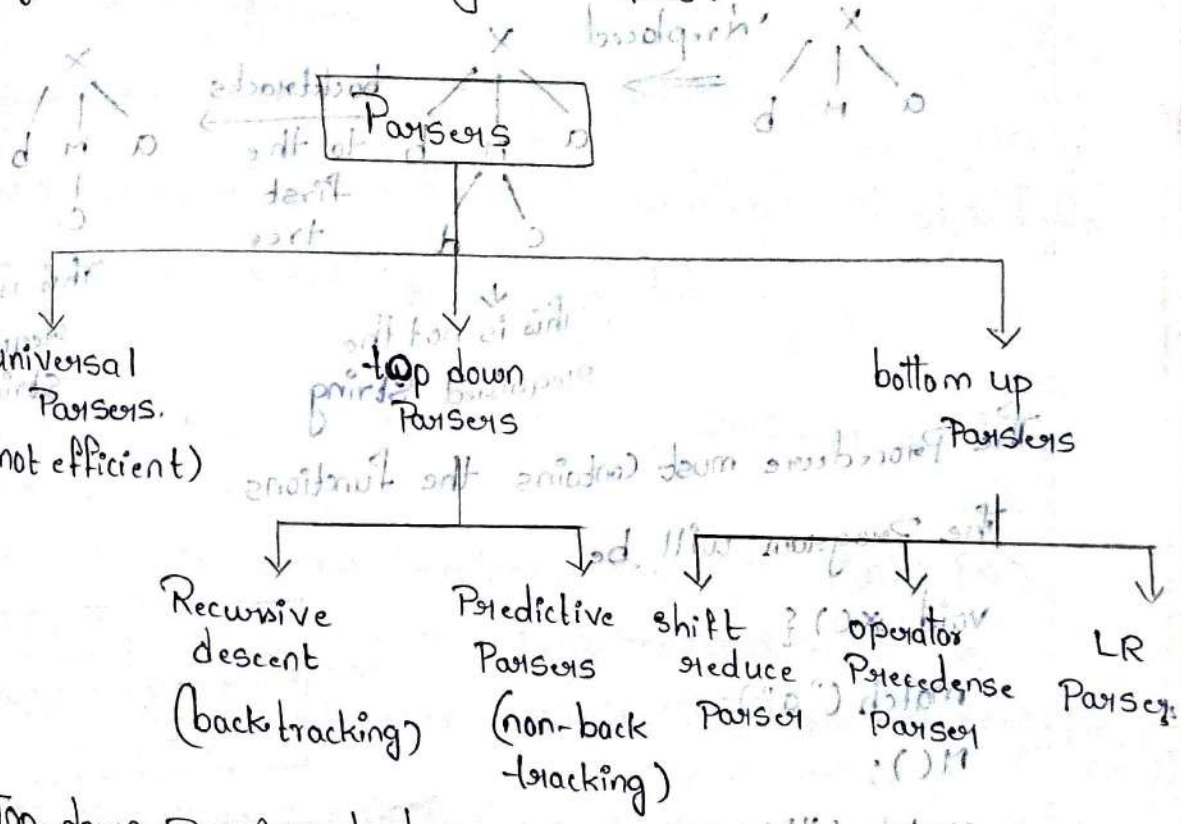
$\downarrow a \quad \downarrow a$



(a+a) * (a+a)



Types of Parsers / Parsing techniques:-



Top-down Parsing technique:-

Top-down parsing technique uses left-most derivation. There are two types of top-down parsing techniques "backtracking parsing & non-backtracking Parsing".

→ Recursive descent Parser:

A Recursive descent parser is a collection of Procedures one for each non-terminal

Scanning until it stops and announces Success if it Scans the entire String.

9) Construct Recursive descent Parser for the following grammar

X → aMb M → cd|c.

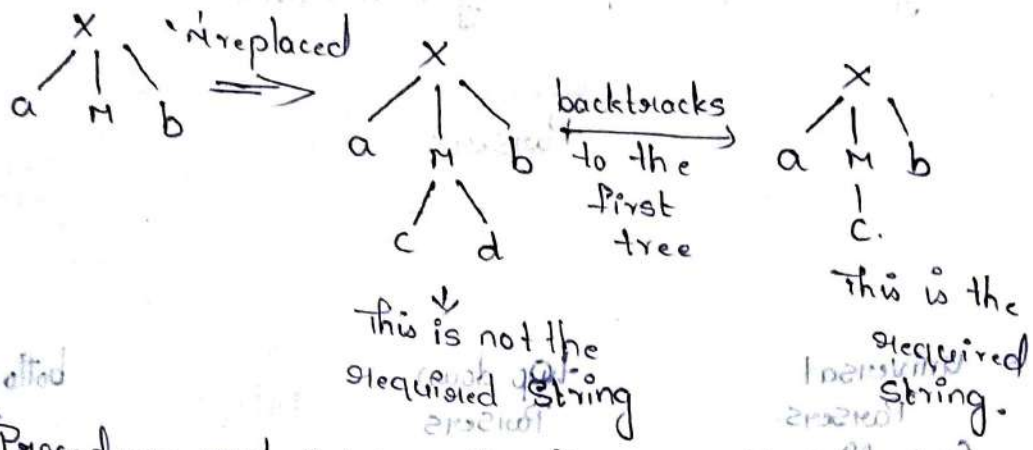
Given Grammar

X → aMb

M → cd|c ⟺ M → cd
M → c

If let the string be abc acb, then.

the Parse tree would be.



The Procedure must contains the functions.

the Program will be

```
void X(){
match("a");
M();
Match("b");
}
Void M(){
  Match("c");
  if(input[pos]=='d'){
    match("d");
  }
}.
```

a	c	b
1	2	3

Main → X() → M()
a b c
backtracks

→ Predictive Parser:-

Recursive Predictive Parser:-

LL1 Grammar:-

→ The first 'L' in LL1 Grammar is scanning from left to right of input.

→ The second 'L' stands for Left most derivation.

→ The third 1 stands for only one symbol is used by parser to make a Parsing decision.

→ LL1 Grammar is always non-left recursive and un-ambi ambiguious.

first follow Procedure / technique:-

for a grammar G, we need two functions first & follow to construct Predictive Parsing table.

FIRST(A):

FIRST(A) is a set of terminals that 'A' can begin with.

→ Rule-1:- If 'A' is a terminal, then FIRST(A) = {A}

→ Rule-2:- If 'A' is a non-terminal and $A \rightarrow X_1 X_2 \cdots X_n$,

→ If FIRST(X_1) dosnot contains empty ($\emptyset$), then FIRST(A) = FIRST(X_1)

- If FIRST(X_1) Contains empty ($\emptyset$) & FIRST(X_2) doesnot contain 'ε' then FIRST(A) = FIRST(X_2)
- In General, FIRST(A) = FIRST(X_k) If FIRST(X_1), FIRST(X_2) ...., FIRST(X_{k-1}) Contains empty (ε) and FIRST(X_k) dosnot Contain 'ε'.

→ Rule-3:- If there exist a Production $A \rightarrow \varepsilon$ then add 'ε' to FIRST(A).

follow(A):-

FOLLOW(A) for a non-terminal 'A' is a set of all terminals that can follow 'A'.

Note:- the FOLLOW doesnot contain 'ε'.

Rule-1:- Set FOLLOW(S) = $, where 'S' is the start Symbol & the Symbol $ indicates end of input.

Rule-2: If there exist a production A→βxγ, where γ is not 'ε', then everything in FIRST(γ), except 'ε' is in Follow(x)

Rule-3:- If A→βx (or) A→βxγ is a production, where FIRST(γ) contains 'ε', then all in FOLLOW(A) is in FOLLOW(x).

A→βx (or) A→βxγ

Q) Construct LL(1) Parse table for the following grammar.

S→aABb, A→c|ε, B→d|ε.

Given Grammar

S→aABb

A→c | B→d

A→ε | B→ε

To Construct LL(1) Parse table, we first need to construct the FIRST FOLLOW table.

	FIRST	FOLLOW
S	a	$
A	c, ε	d, b
B	d, ε	b

first ⇒ S→@ABb

A→c, A→ε } → all the starting terminals

For follow,

Consider 's', the symbol 's' is not present in any right hand production rule, so, we place $ for 's' because s is a start symbol.

→ For 'A', the Production rule aABb produces the 'A'.

a A B b

A is followed by Non-Terminal 'B'. So, replace it with B→ d, B→ ε.

⇒ aBaAdb & aAεb ⇒ aAb

We get non-terminals, so, A followed by d & b.

3 → For 'B', the production rule aABb produces the B'

a A B b

B is followed by non-terminal 'b'.

So the LL(1) Parse table will be:

	a	b	c	d	$
S	S→aABb				
A		A→ε	A→c	A→ε	
B		B→ε		B→d	

This is the LL(1) Parse table for the given grammar.

Construct LL(1) Parse tree

$S \rightarrow AB$, $A \rightarrow a | cB | \varepsilon$, $B \rightarrow d | \varepsilon$.

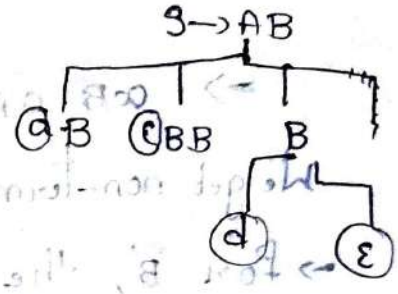
Given Grammar

$S \rightarrow AB$

$A \rightarrow a$ $\quad A \rightarrow cB$, $A \rightarrow \varepsilon$

$B \rightarrow d$ $\quad B \rightarrow \varepsilon$

	FIRST	FOLLOW
for S =>	a, c, d, ε	$
for A =>	a, c, ε	d, $
for B =>	d, ε	d, $



for FOLLOW(B)

The $S \rightarrow AB$ & $A \rightarrow cB$.

From the last rule of FOLLOW(X), we get

if $A \rightarrow BX$ then FOLLOW(X) = FOLLOW(A).

So, for $S \rightarrow AB$, follow(B) = FOLLOW(S).

= $.

for $A \rightarrow CB$, FOLLOW(B) = FOLLOW(A)

follow(A) => $S \rightarrow A\,B$

A is followed by B, then replace B with d & ε.

then follow(A) = d, Not ε because ε donot be in follow.

So, **follow(B) = ε, d, $.**

=> As, we got follow(A) as d, but from $A \rightarrow CB$,

follow(B) = follow(A)

LL(1) Parse table will be,

If the Production rule contains 'ε' we place it in follow.

If the Production rule does not contain 'ε', we place it in FIRST.

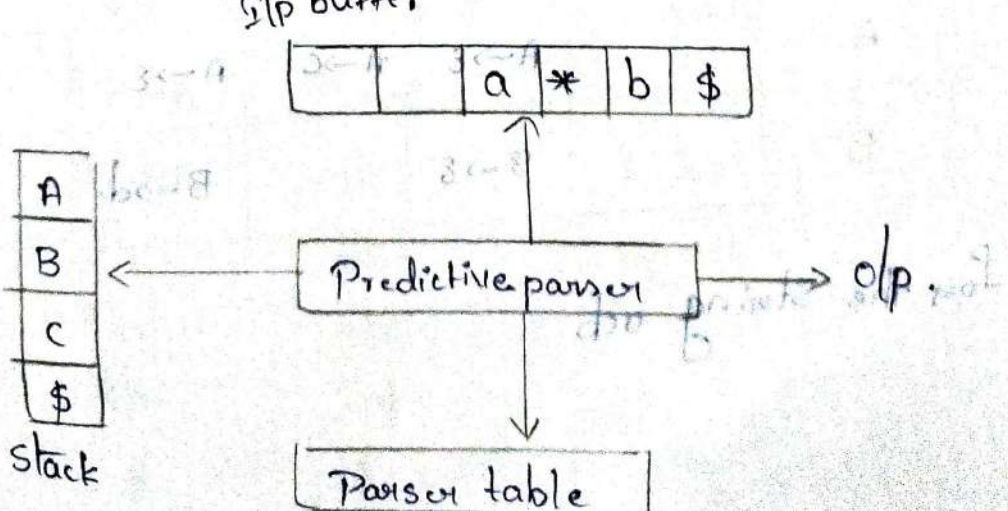
	a	b	c	d	$
S	S→AB		S→AB	S→AB	
A	A→a A→cB		A→a A→cB	A→ε	A→ε
B				B→d B→ε	B→ε

Hence, this contains more than one Production rule So, it is not LL(1) Grammar.

Non-Recursive Predictive Parser:-

The Non-Recursive Predictive Parser explicitly maintains a stack instead of implicit stack that is maintained via during recursive fn calls.

→ The parser imitates left-most derivation.



The - Non-Recursive Predictive Parser contains

1) stack - contains grammar symbols and $ at the bottom of the stack.

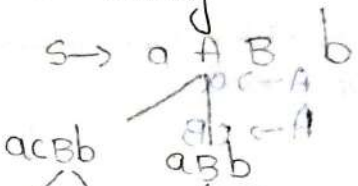
2) Input Buffer - contains string that is to be parsed.

3) Parser table

4) output string

Q) Do Predictive Parsing for the String

Grammar, S → aABb
A → c | ε
B → d | ε



from the given grammar acdb acb adb ab

	FIRST	FOLLOW
S → aABb	a	$
A → c | ε	c, ε	d, b
B → d | ε	d, ε	b.

Parsing table:-

	a	b	c	d	$
S	S → aABb				
A		A → ε	A → c	A → ε	
B		B → ε		B → d.	

For the string acb.

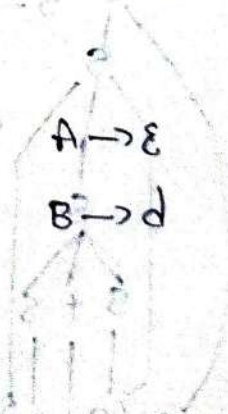
Stack	Ilp	Production Selected.
$S	acb$	S → aABb
$bBAa	acb$	
$bBA	cb$	A → c
$bBc	cb$	
$bB	b$	B → ε
$b	b$	
$	$	

→ For the string ab

Stack	Ilp	Production Selected.
$S	ab$	S → aABb
$bBAa	ab$	
$bBA	b$	A → ε
$bB	b$	B → ε
$b	b$	
$	$	

→ For the string adb

Stack	Ilp	Production Selected
$S	adb$	S → aABb
$bBAa	adb$	
$bBA	db$	A → ε
$bB	db$	B → d
$bd	db$	
$b	b$	
$	$	



Stack	Ip	Production Selected.
$S	acdb$	S→aABb
$bBAa	acdb$	
$bBA	cdb$	A→c
$bBc	cdb$	
$bB	db$	B→d
$bd	db$	
$b	b$	
$	$	

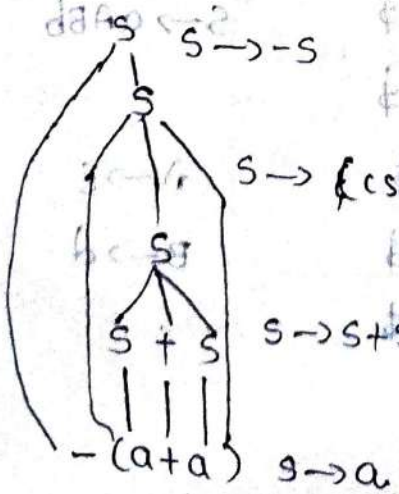
Bottom-up Parsing:-

In Bottom-up Parsing, we construct the parse tree from input string, we consider charactors of input string as leads & move towards the root.

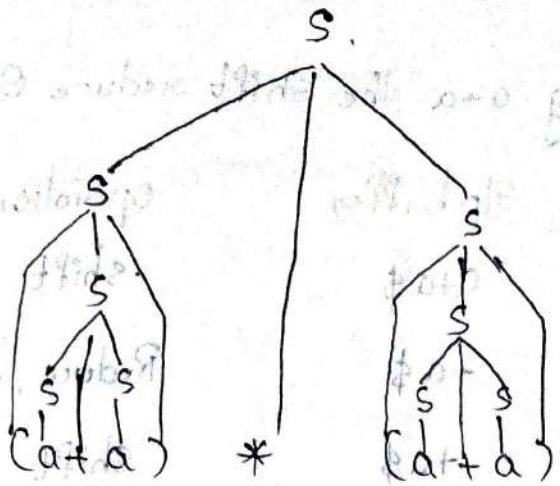
Q) Construct Parse tree for the string -(a+a), using the grammar. G: S→ S+S | S*S | -S | (S) | a

in this, we go from bottom.

Given grammar G: S→ S+S | S*S | -S | (S) | a.



For Grammar (a+a) * (a+a)



3/9/25 Shift Reduce Parser:

Shift reduce parser is a type of bottom up parser, in which stack is used to hold grammar symbols & input buffer is used to store the input string to be parsed.

The shift reduce parser has four operations

1) shift - shift is used to push the input symbol on the top of the stack.

2) Reduce - A reduce action occurs when we have the right end of the handle at the top of the stack to Perform reduction, we locate the left end of the handle with in the stack & choose a non-terminal on the left hand side of the corresponding rule and replace the handle.

3) Accept - Accept occurs when there is successful complition of Parsing.

4) Error - Error occurs when the parser finds an error while Parsing.

Ex:- Consider the following grammar

① $S \rightarrow S+S \mid S*S \mid (S) \mid a$

To Parse the string a+a the shift reduce Operations are

Stack	I/p buffer	operation
$	a+a$	shift
$a	+a$	Reduce, S→a
$S	+a$	shift
$S+	a$	shift
$S+a	$	reduce, S→a
$S+S	$	Reduce, S→S+S
$S	$	Accept.

Hence, the string is parsed.

② $S \rightarrow S+S \mid S*S \mid (S) \mid a \mid -S$.

To Parse the string -(a+a), the shift reduce operations a

Stack	I/p buffer	operation
$	-(a+a)$	shift
$-	(a+a)$	shift
$-(	a+a)$	shift
$-(a	+a)$	reduce, S→a
$-(S	+a)$	shift
$-(S+	a)$	shift
$-(S+a	)$	reduce, S→a
$-(S+S	)$	shift
$-(S+S)	$	reduce, S→S+S
$-(S)	$	

③ S → S+S | S*S | (S) | a | -S.

To parse the string (a+a)-a the shift reduce operations are.

Stack	Ip buffer	Operation
$	(a+a)-a $	shift
$(	a+a)-a $	shift
$(a	+a)-a $	reduce, S→a
$(S	+a)-a $	shift
$(S+	a)-a $	shift
$(S+a	)-a $	reduce, S→a
$(S+S	)-a $	reduce, S→S+S
$(S	)-a $	shift
$(S)	-a $	reduce, S→(S)
$S	-a $	reduce shift
$S-	a $	shift
$S-a	$	reduce, S→a
$S-S	$	Error.

The string cannot be completely Parsed by the given Grammar.

LR Parser:-

LR Parsers are bottom-up Parsers.

→ LR Parsers are non-backtracking shift reduce Parsers.

→ The 'L' in LR Parsers indicates k to left to right scanning of input & 'R' stands for Right most derivation in reverse.

→ LR Parsing is a technique used for Syntactic recognization of Programming languages.

, a set of CFL Production rules.

→ There are three types of LR Parsers

1) Simple LR Parser (SLR Parser)
2) Canonical LR Parser
3) Look ahead LR Parser.

10/9/25

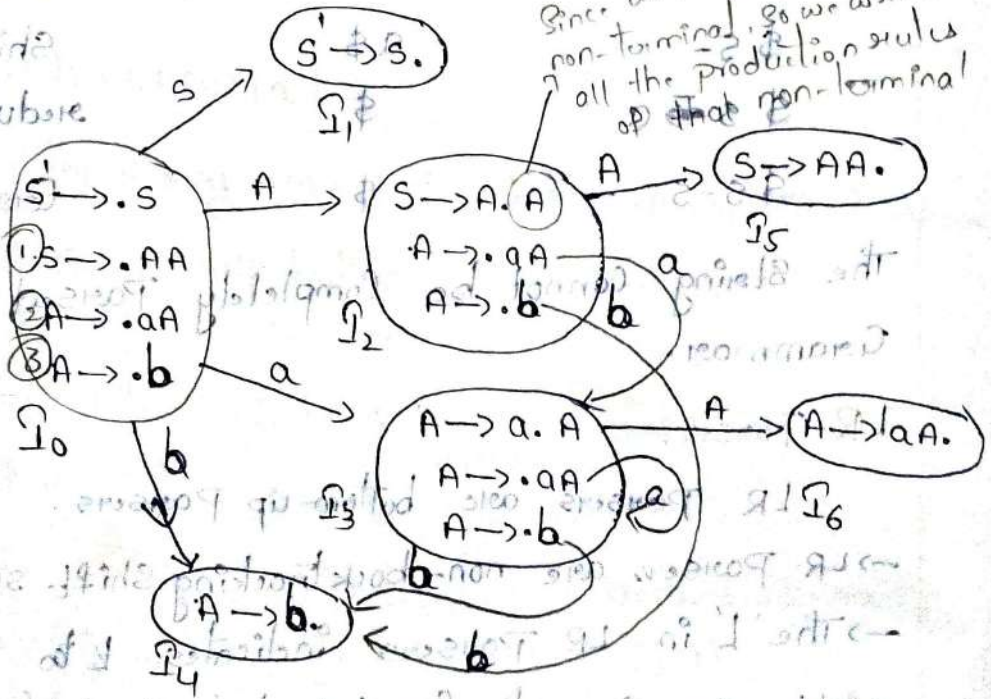
Q) Construct LR(0) Parsing table for the given grammar

S → AA A → aA|a

Given Grammar S → AA , A → aA
A → b

Assume a state called S' → S. &
Remaining are S → AA
A → aA|b

→ It follows DFA



S → shift
R → Reduce.

Non terminals → Goto - mention only state numbers.
Terminals → Action [for shift - S(state)
for Reduce [Production rule number]

Here A → aA. is completely parsed but it is not be accepted because it is not

	Action			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

$S \to AA$

SLR Parser [simple LR Parser]:-

$S \to AA$

$A \to aA$

for the before diagram, the SLR Parser be,

	FIRST	FOLLOW
$S \to AA$	a, b	$
$A \to aAlb$	a, b	a, b

· For follow of S, (first)

$S \to A A$

$A \to aA$ — b

follow will be first of A.

$A \to aA$

$A \to b$

Parsing table,

	Action			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3			
5			R_1		
6	R_2	R_2			

non-terminal

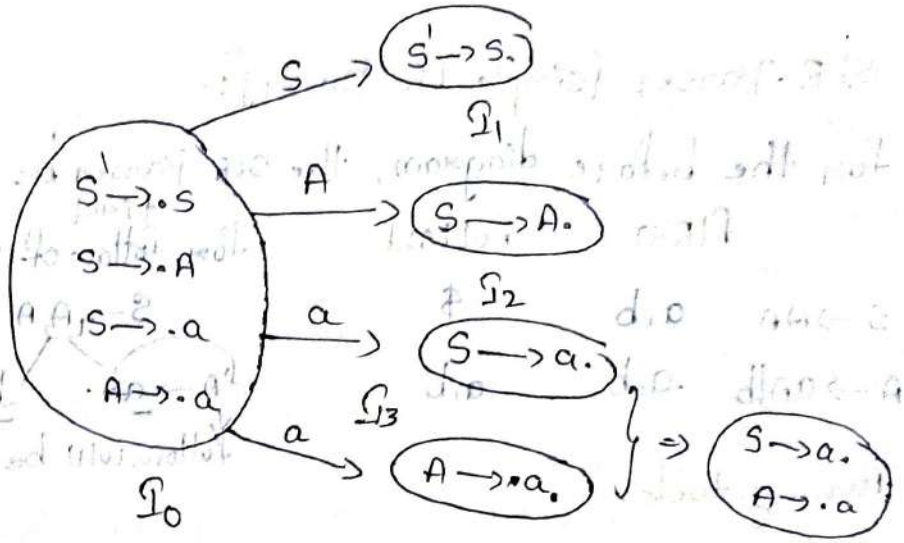
For $S \to AA.$ → This is the rule no: 1,

So, it is represented as R_1 & $S \to AA.$ placed in the follow of S which is $.

Q) Construct the SLR Parser for the given grammar

$S \to Ala$, $A \to a$

	FIRST	follow
$S \to Ala$	a	$
$A \to a$	a	a

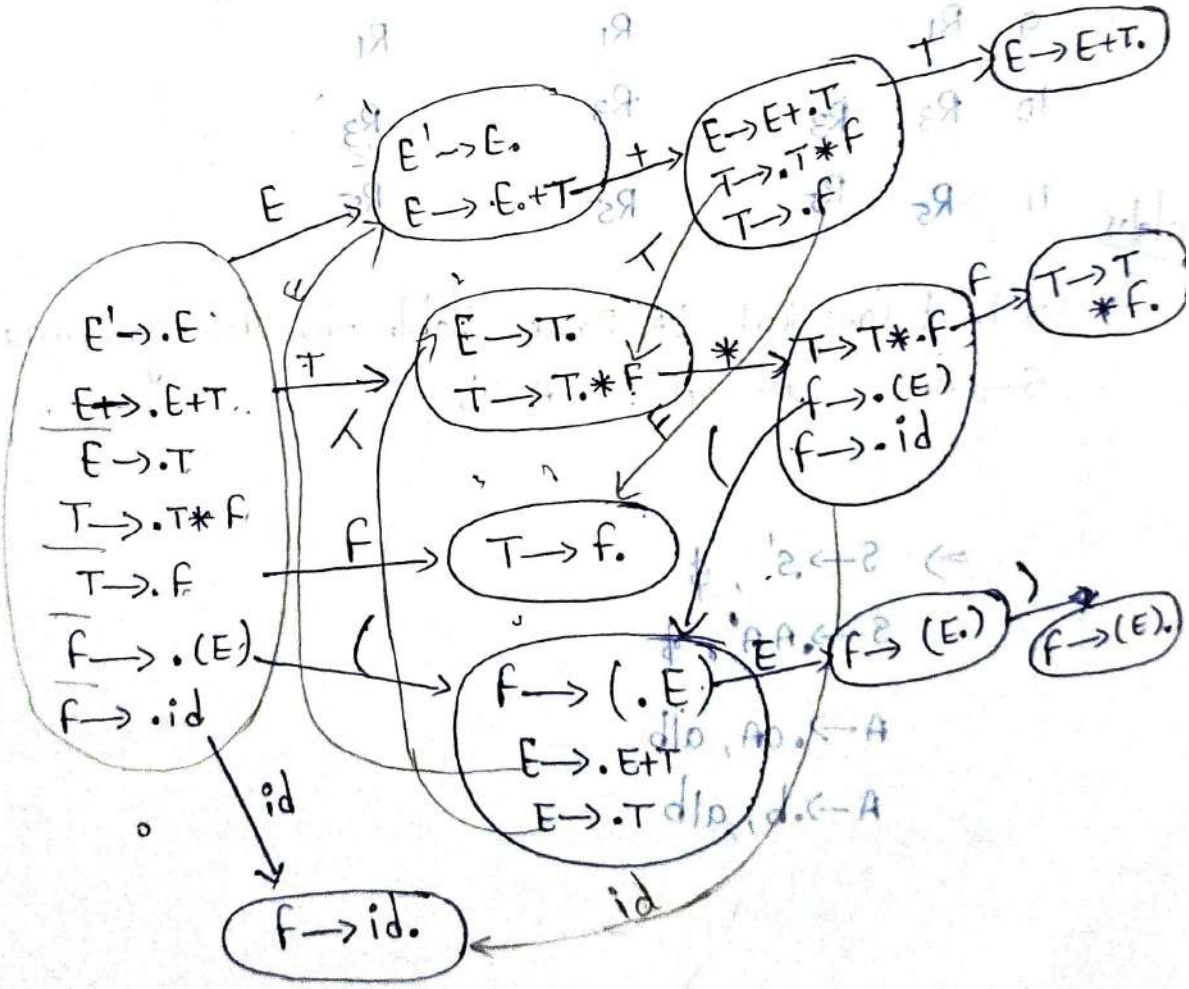


	Action		Goto	
	a	$	S	A
0	S_3		1	2
1		Accept		
2		R_1		
3	R_3	R_2		

11/9/25 Construct SLR Parser for the given grammar

E→E+T, E→T, T→T*F, T→F, F→(E), F→id

	first	follow
E→E+T E→T	(, id	$, +,)
T→T*F T→F	(, id	*, $, +,)
F→(E) F→id	(, id	*, $, +,)

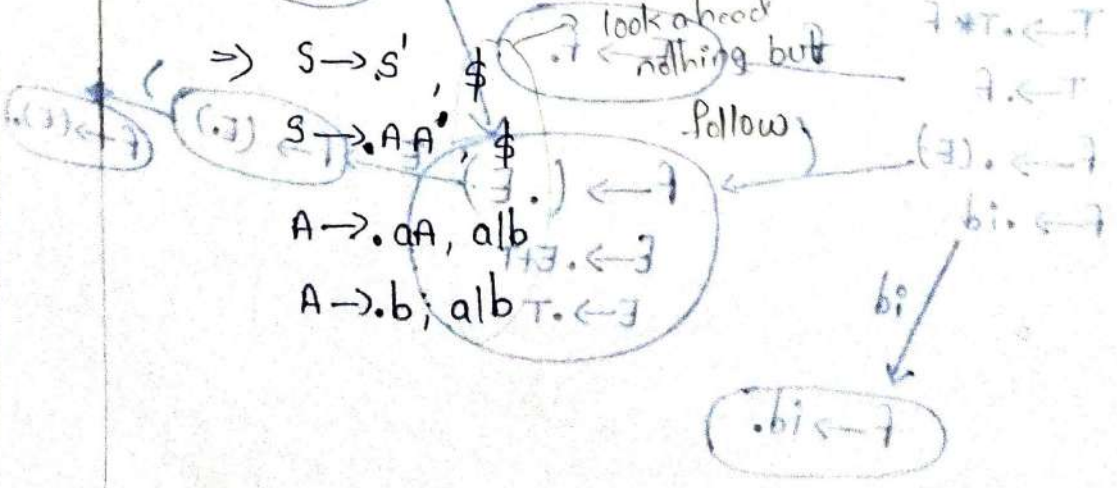


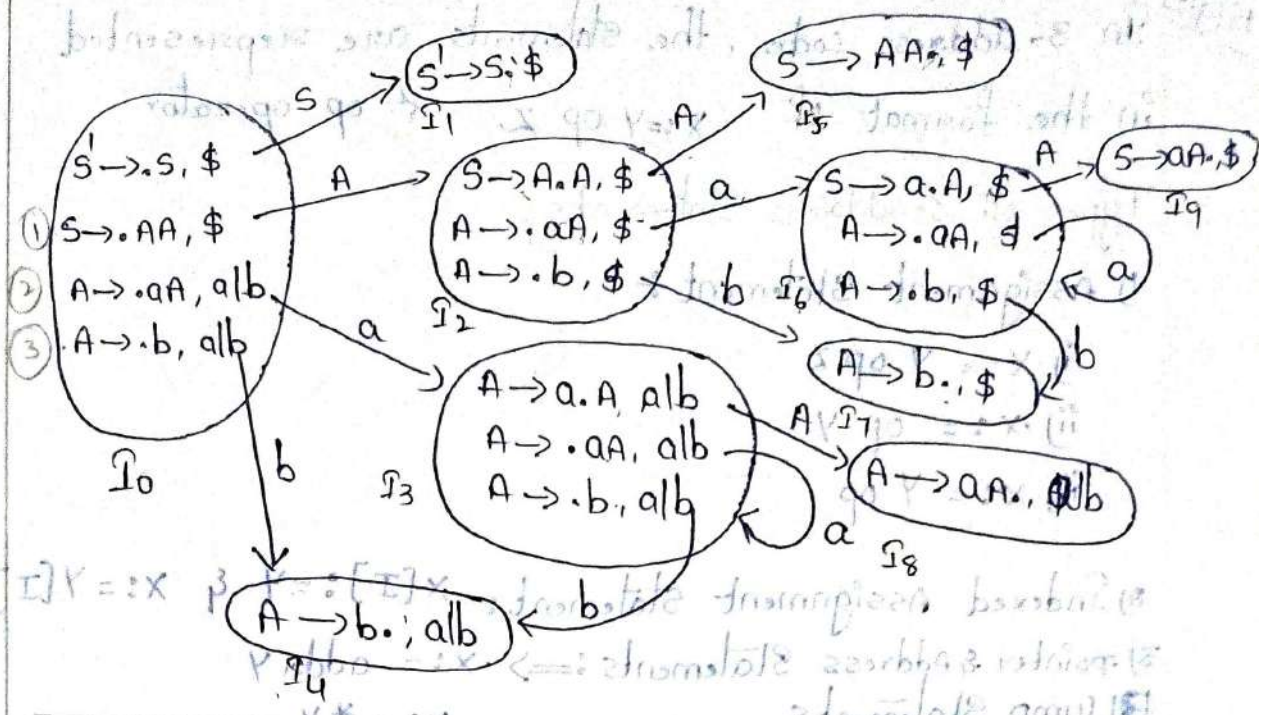
	Action						Goto		
	+	*	(	)	id	$	E	T	f
0			S_4		S_5		1	2	3
1	S_6					Accept			
2	R_2	S_7		R_2		R_2			
3	R_4	R_4		R_4		R_4			
4							8, 1	2	
5	R_6	R_6		R_6		R_6			
6								9, 2	10 3
7			S_4		S_5				10
8				S_{11}					
9	R_1			R_1		R_1			
10	R_3	R_3		R_3		R_3			
11	R_5	R_5		R_5		R_5			

12/9/25

Q) Construct Canonical LR parser table for the grammar

$S \to AA$ $A \to aA$ $A \to b$

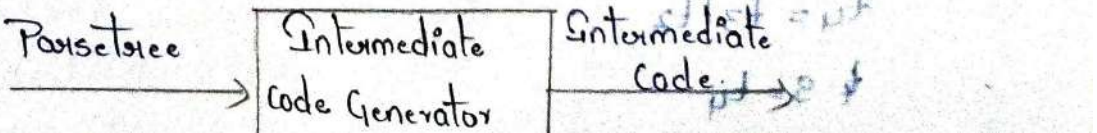




	Action			GoTo	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5			R_1		
6	S_6	S_7			9
7			R_3		
8	R_2	R_2			
9			R_2		

Intermediate Code Generator:- [3-address code]

The Intermediate Code Generator takes Parse tree as input and generates Intermediate code or 3-address code.



15/9/25 In 3-address code, the statements are represented in the format of $x := y \ op \ z$ ← op-operator

Types of 3-address Statements:-

1) Assignment Statement :-
   i) $x := y \ op \ z$
   ii) $x := op \ y$
   iii) $x := y \ op$

2) Indexed Assignment statement: $x[I] := y$ & $x := y[I]$.

3) Pointer & Address Statements :⟹ $x := addr \ y$
   $x := *y$
   $*x := y$.

4) Jump Statements
   - if
   - goto

   if x relop y goto A

relop = relational operator

Representation of 3-address code:-

1) Quadruple
2) Triple
3) Indirect Triple

Q) $S = -z / a * (x+y)$

A) → Intermediate code.

$S = -z / a * (x+y)$

First Solve the one's in the ( )

$t_1 = x + y$

$t_2 = a * t_1$

$t_3 = -z$

$t_4 = t_3 / t_2$

$S = t_4$

operator	operand1	Operand 2	Result:
+	x	y	t_1
*	a	t_1	t_2
-		z	t_3
/	t_3	t_2	t_4
=	t_4		s

Triple :-

In this we assign index values.

$t_1=0$, $t_2=1$, $t_3=2$, $t_4=3$, & $s=4$

operator	operand1	operand2
+	x	y
*	a	(0)
-		z
/	(2)	(1)
=	s	(3)

$s = (t_4)$

$\downarrow (3)$

$s = (3)$

op1 op op2

Indirect Triple:-

the values be any value (its our wish)

101	(0)
102	(1)
103	(2)
104	(3)
105	(4)

index values linked to the values in table

operator	operand1	operand2
+	x	y
*	a	(0)
-		z
/	(2)	(1)
=	s	(3)

2) $s = -(a+b) * (c+d) + (a+b+c)$

$t_1 = a+b$

$t_2 = c+d$

$t_3 = t_1 + c$

$t_4 = -t_2$

$t_5 = t_4 * t_3$

$t_6 = t_5 + t_3$

$s = t_6$

Quadraple.

Operator	Operand1	Operand2	result
+	a	b	t_1
+	c	d	t_2
+	t_1	c	t_3
-		t_1	t_4
*	t_4	t_2	t_5
+	t_5	t_3	t_6
=	t_6		s

Triple:-

$t_1=0$, $t_2=1$, $t_3=2$, $t_4=3$, $t_5=4$, $t_6=5$, $s=6$.

Operator	Operand1	Operand2
+	a (1)	b (c)
+	c (ε)	d
+	(0)	c
-		(0)
*	(3)	(1)
+	(4)	(2)
=	s	(5)

Indirect Triple

101	(0)
102	(1)
103	(2)
104	(3)
105	(4)
106	(5)

Operator	Op1	Op2
+	a	b
+	c	d
+	(0)	c
-		(0)
*	(3)	(1)
+	(4)	(2)
=	s	(5)

Q) $S = -Z * (a+b) / c$

$t_1 = (a+b)$

$t_2 = t_1 / c$

$t_3 = -z$

$t_4 = t_3 * t_2$

$S = t_4$

Quadraple:-

operator	operand1	Operand2	Result
+	a	a	t_1
/	t_1	c	t_2
-		z	t_3
*	t_3	t_2	t_4
=	t_4		S

Triple:-

$t_1=0$, $t_2=1$, $t_3=2$, $t_4=3$, $S=4$

Operator	operand1	operand2
+	a	a
/	(0)	c
-		z
*	(2)	(1)
=	S	(3)

Indirect Triple:-

101	(0)
102	(1)
103	(2)
104	(3)
105	(4)

operator	operand1	operand2
+	a	a
/	(0)	c
-		z
*	(2)	(1)
=	S	(3)

17/9/25 Unit

Unit - V

Code Optimization and Code Generation:-

Basic Block:-

A Block is a set of instructions that start from first line in intermediate code and continues until we don't encounter in looping, branching, halting or jump statements in the basic block, the control will leave from the last statement

Leaders of Basic Block:-

1) The first statement of Intermediate code.
2) Target of Conditional or unconditional jump statement.
3) The statement that is immediately next to Conditional or unconditional jump statements

Ex:-

Given Pseudo Code is:

```
begin
  Product = 0
  j = 1
  do Begin
  Product = product + x[j] * y[j];
  j = j + 1
  end
  while (j <= 20)
end.
```

Intermediate code:-

```
1. Product = 0
2. j = 1
```
Block-1

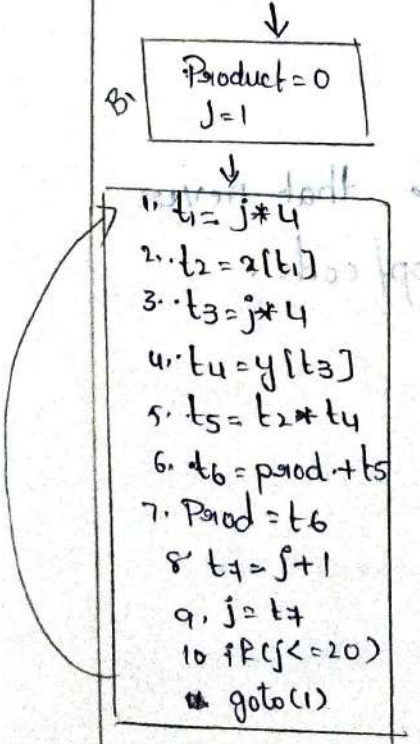
```
3.  t1 = j * 4
4.  t2 = a[t1]
5.  t3 = j * 4
6.  t4 = y[t3]
7.  t5 = t2 * t4
8.  t6 = Product + t5
9.  Product = t6
10. t7 = j + 1
11. j = t7
12. if (j <= 20)
13. goto (3)
```
Block-2

loops, gotos represented in if & goto statements

blocks get separated based on loops, branching halting or jump statements.

flow graph

flow graph describes the flow of execution of blocks, in which they are executed.



Code optimisation:

Code optimizer takes intermediate code as input & then removes unwanted or unnecessary code & also unreachable code.

There are two types of optimisation.

i) Local optimisation
ii) Global optimisation

Constant folding:-

In constant folding, the mathematical operations performed on variables which gives a fixed answer & doesnot change in the entire program or replaced with their values & before compilation

Ex:-

```
x= 3+2
a= x+5
```

after constant folding the code becomes.

```
x=5
a=10
```

loop optimization:-

i) loop invarient expression Evaluation

in this method we take the code that never changes and placed outside of the loop/ code.

Ex:-

```
if (i>min+2){
    sum= sum+a[i]
}
```

⇒ Here min+a is unchangable. So.

```
temp= min+a
if (i>temp){
```

18/9/25 ii) Strength Reduction:-

Replacing expensive operation with cheaper operation

Ex:-
```
for (i=1; i<=n; i++) {
    count = i*5;
    sum = sum + a[i];
}
```
⇒
```
temp = 5
for(i=1; i<=n; i++){
    Count = temp;
    temp = temp+5;
} sum = sum + a[i];
```

i	Cnt
5x1	5
5x2	10
5x3	15
5x4	20

temp	int
8*5	5
5+10	15 10

temp	Cnt
5 + 5	5
5 + 10	10
5 + 15	15
5 + 20	20

iii) Loop fusion:-

Ex:-
```
for (i=0; i<n; i++){
    a[i] = b[i];
}
for (i=0; i<n; i++) {
    c[i] = a[i];
}
```
→ after fusion →
```
for (i=0; i<n; i++){
    a[i] = b[i];
    c[i] = a[i];
}.
```

⇒ This Process is used to Combine the loops.

→ the above example doesn't make any change. when we Combine the loops into one.

Directed Acyclic Graph [DAG]:-

It represents the way how the value is computed by each statement in the basic block

Intermediate Code:-

$t_1 = j * 4$

$t_2 = a[t_1]$

$t_3 = j * 4$

$t_4 = y[t_3]$

$t_5 = t_2 * t_4$

$t_6 = product + t_5$

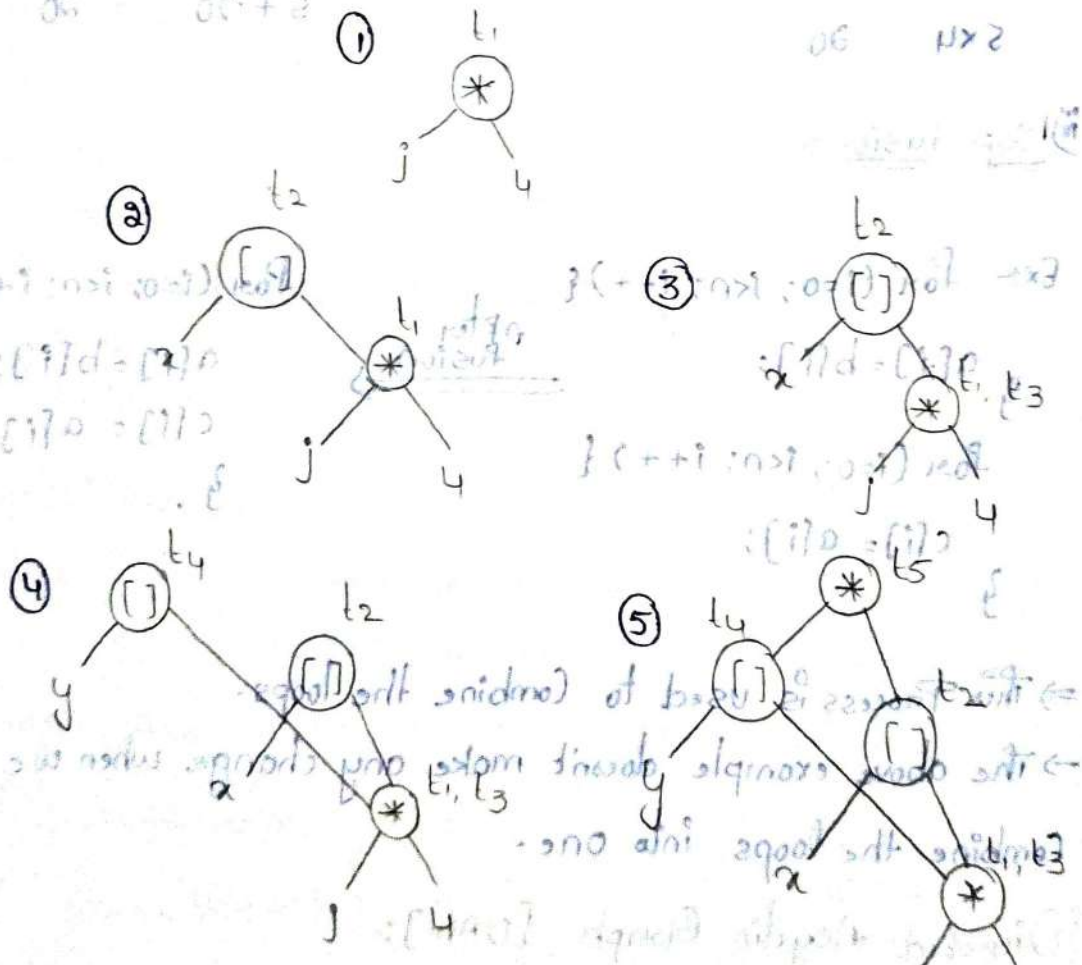
$Product = t_6$

$t_7 = j + 1$

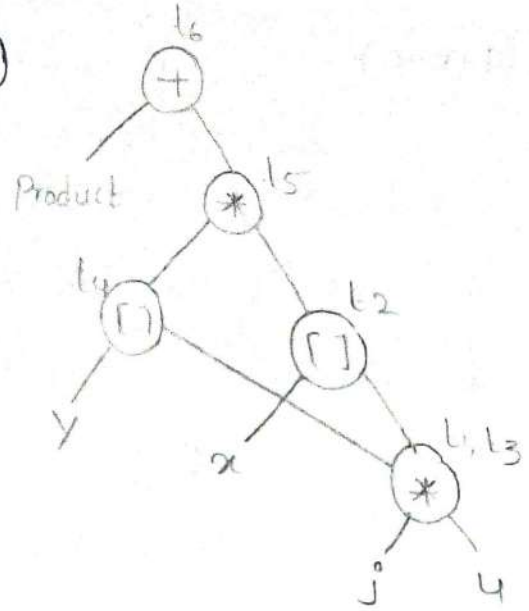
$j = t_7$

if $(j <= 20)$

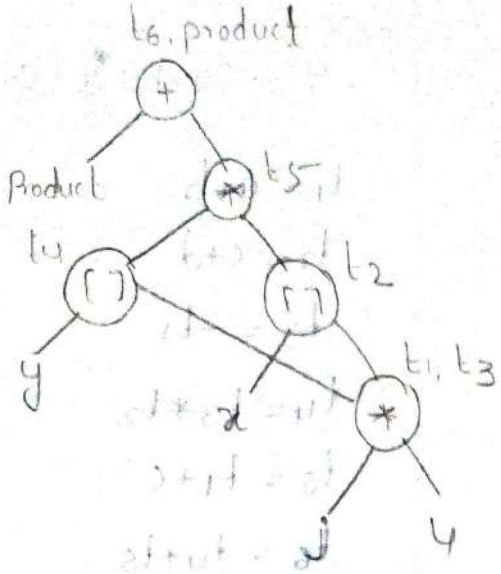
goto (1)



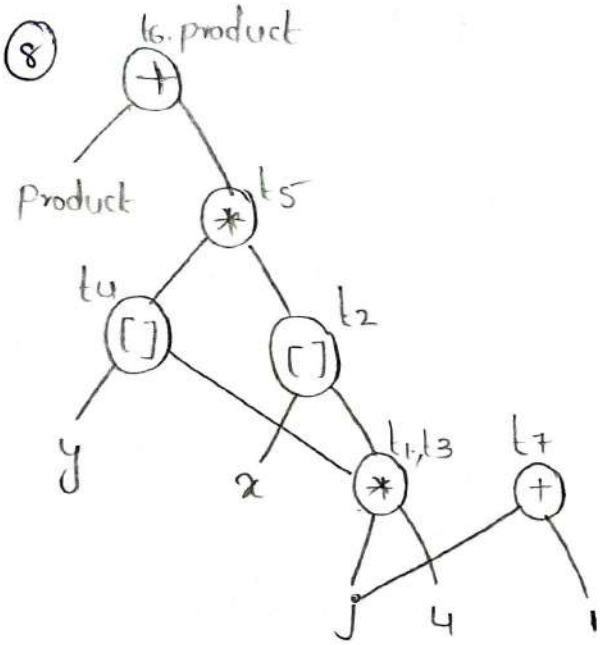
(6)


t6
Product
t5
t4
t2
y
x
t1, t3
j
4

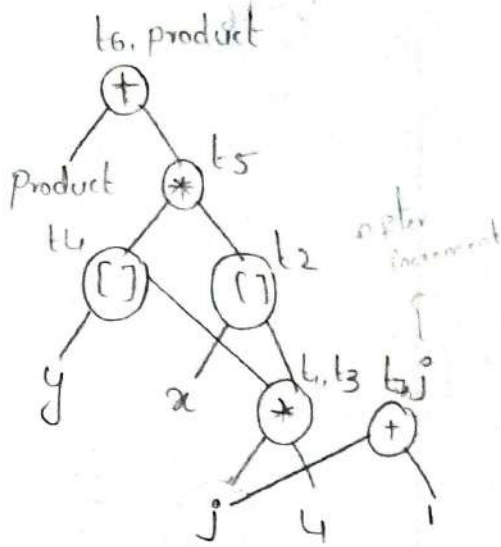
(7)


t6. product
Product
t5
t4
t2
y
x
t1, t3
j
4

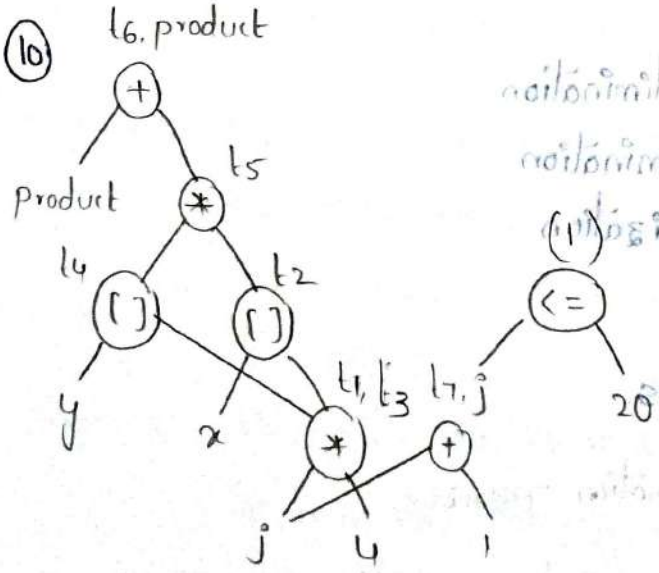
(8)


t6. product
Product
t5
t4
t2
y
x
t1, t3
t7
j
4
1

(9)


t6. product
Product
t5
t4
t2
y
x
t1, t3
t3, j
j
4
1

(10)


t6. product
Product
t5
t4
t2
y
x
t1, t3
t7, j
<=
20
j
4
1

Ex:-

$S: -(a+b) * (c+d) + (a+b+c)$

$t_1 = a+b$

$t_2 = c+d$

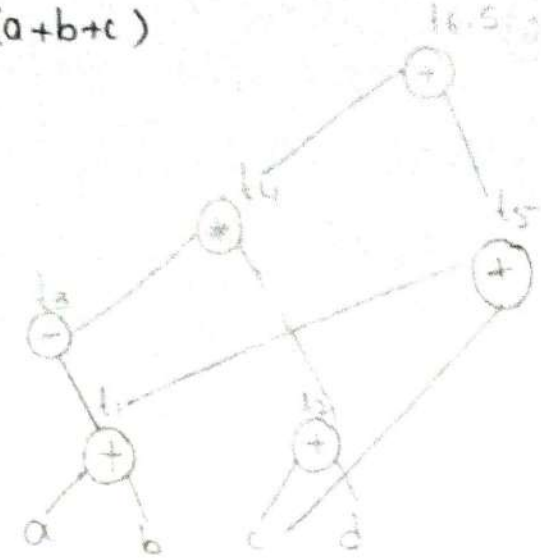
$t_3 = -t_1$

$t_4 = t_3 * t_2$

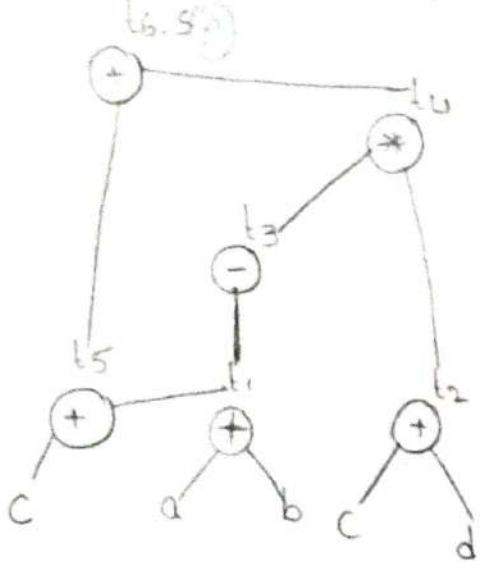
$t_5 = t_1 + c$

$t_6 = t_4 + t_5$

$S = t_6$.



[or]



19/9/25

Peephole optimization:-

1) Redundant instruction elimination
2) Unreachable code elimination
3) flow of control optimization
4) Strength reduction.
5) Use of machine idioms.

Issues with code generation process
(or)
Various factors affecting code generation process:-

1) input
2) structure of target code.

5) Evaluation order.

Simple code Generator:-

→ A simple code Generator Generates target code from intermediate code. The main issue during code generation is utilization of registers because the main issue during registers are limited.

→ The code generation algorithm takes 3-address statements as inputs and assumes that for each operator of Source Program, there exist a equivalent operator in the target Program.

→ The machine code instruction takes the required operands in registers, Performs the operation and stores the result.

→ Register and address descriptors are used to keep track of the contents of registers and addresses.

→ The Simple code generation Algorithm for $x := y \text{ op } z$ is

Step - ① :- call getreg() to obtain the location L where the result of y op z is to be stored. L can be a register or a memory location.

Step - ② : Determine the current location of y. let it be y'. If both the memory and register contains the value of y then Consider y'. If the value is not Present in L then perform the operation.

step-③:- determine the current location of 'x'. let it be 'x'. & generate the instruction op x'

step-④:- If the current value of y, z are in registers, if they have no further use and are not alive at the end of the block then alter the register descriptor. This alteration indicates that y, z will no longer be present in those registers after the execution of x = y op z.

Example:-

① d = (a-b) + (a-c) + (a-c)

Intermediate Code:-

$t_1 = a - b$

$t_2 = a - c$

$t_3 = t_1 + t_2$

$t_4 = t_3 + t_2$

$d = t_4$

Operation	Register	Memory	
MOV	R_0	a	
MOV	R_1	b	
SUB	R_0	R_0 R_1	→ here a-b operation is performed and then the result in a is stored in register R_0
MOV	R_2	a	
MOV	R_3	c	
SUB	R_2	R_2 R_3	
ADD	R_0	R_0 R_2	
ADD	R_0	R_0 R_2	
ST ↓ Store	d	R_0	

(ii) $S = (a+b+c) / (a+b)$

Intermediate Code :-

$t_1 = a+b$.

$t_2 = t_1 + c$

$t_3 = t_2 / t_1$

$t_3 = S$

operation	Register	Memory.
MOV	R_0	a
MOV	R_1	b
ADD	R_0	R_0 R_1
MOV	R_2	c
ADD	R_0	R_0 R_2
MOV	R_3	a
ADD	R_3	$R_3 R_1$
DIV	R_0	R_0 R_3
ST	S	R_0