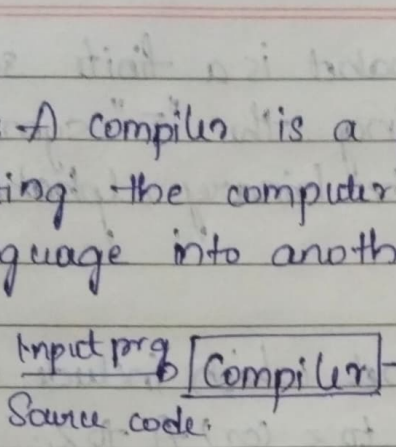
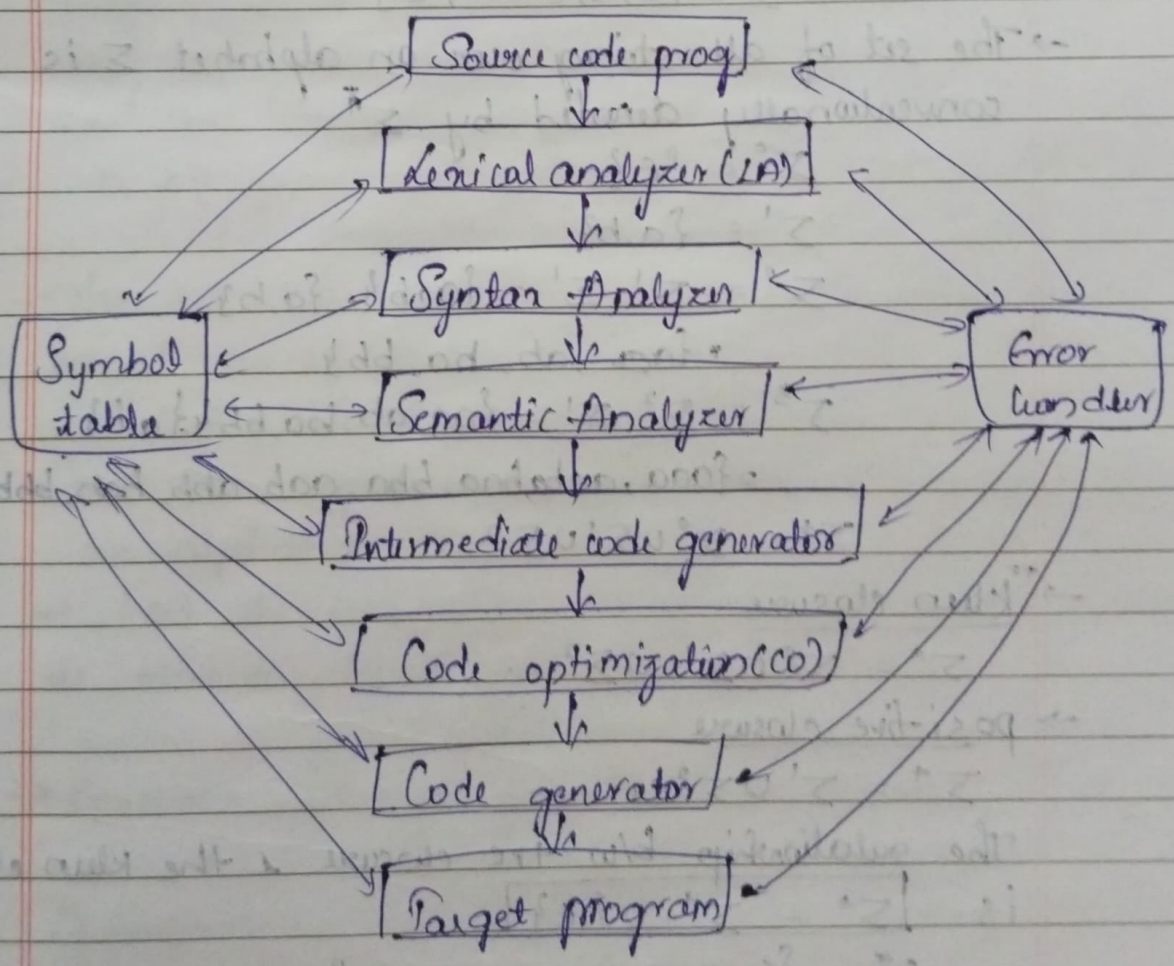
Compiler: A compiler is a program that helps in translating the computer code from one programming language into another programming language



Phases of Compiler



1) Analysis phase - lex, syn, sem
2) Synthesis phase - Int, cod opt, code gen.

<u>Alphabet</u>: An alphabet is a finite set of symbols representing using the "Σ" symbol.

→ Alphabets can be binary alphabet which include 0+1

(or)

→ Character alphabet which include the lower case alphabets a to z (or) A to Z.

i.e $\Sigma = \{0,1\}$

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

→ The set of all string over an alphabet Σ is conventionally denoted by Σ^*

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

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

$\Sigma^2 = \Sigma^1 \cdot \Sigma^1 = \{a, b\} \cdot \{a, b\}$

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

$\Sigma^3 = \Sigma^2 \Sigma^1 = \{aa, ab, ba, bb\} \{a, b\}$

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

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \ldots$

→ **<u>Klen closure</u>

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \ldots$

→ <u>positive closure</u>

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \ldots$

The relationship b/w the closure & the klen closure is $\boxed{\Sigma^* = \Sigma^+ \cup \{\epsilon\}}$

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

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

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

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

$0^* 1^* = \{\epsilon, 1, 0, 01, 001, 011 \ldots\}$

$0^* 1^+ = \{1, 01, 001, 011 \ldots\}$
$\epsilon \cdot 1$

$0^* 1 = \{1, 01, 001, 0001, \ldots\}$

$01^* = \{0, 01, 011, 0111, \ldots\}$

<u>Languages</u>

A set of strings all of which are choosen from some Σ^* where Σ is a particular alphabet called a language.

→ If Σ is an alphabet & $L \subseteq \Sigma^*$ then L is a language over Σ

<u>Ex:</u> The language of all strings consisting of n 0's followed by n 1's for $n \geq 0$

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

→ The set of strings of 0's & 1's with an equal no. of each

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

→ ϕ the empty language is a language over an alphabet

→ $\{\epsilon\}$ the language consisting of empty string is also a lang over any alphabet

$\phi \neq \{\epsilon\}$

→ Construct a language for even number of 0's & 1's

$L = \{00, 11, 00001111, 01010101, 1010, 0101, \ldots\}$

<u>Operations</u>

i) Union of sets ($\cup$)

if A and B are the given sets then the resultant set C on taking union of A to B may be represented as

$C = \{x \mid x \in A \text{ or } x \in B\}$

2) Intersection ($\cap$)

$c = \{x | x \in A \text{ and } x \in B\}$

3) Disjoint sets

Two sets are said to be disjoint when n of the 2 given sets results in a null set.

Ex: $A = \{1,2\}$ $B = \{3,4\}$

$A \cap B = \{\phi\}$

4) Set difference

$A - B$

It is a binary oper^n performed on 2 sets and result in a set of elements that are present in the 1st set and no of present in the 2nd set.

5) Compliment of a set (S')

$S' = U - S$ (U = universal)

This can be defined as set difference of universal set and given set.

6) Symmetric difference ($\oplus$)

$A \oplus B = (A - B) \cup (B - A)$

7) Cartesian product ($\times$)

CP of 2 sets $A + B$ is a set of all possible ordered pairs (a,b) where $a \in A, b \in B$

Strings:

A string is a finite sequence of symbols choosen from some alphabets

Ex: 01101 is a string from the binary alphabet $\Sigma = 0,1$

Empty String: This is a string with 0 occurrence of symbols this is represented by 'e'

Length of a String: the no. of positions for symbols in string

Ex: 01101 length = 5

Concatenation of string: Let X & Y be string then XY denotes the concatenation of X and Y

Sub String: A string x is a substring of another string if if there are strings w & z either / or both of which may be null exist such that

$Y = wxz$

if w is null then x is said to be prefix

if z is null then x is said to be suffix

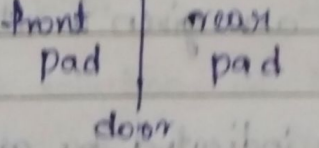
25/08/23

Automata: Automata deals with the definitions & properties of mathematical models used in computation. This models play a major role in areas of computational science

(i) finite automata (FA) used in compilers

(ii) CFG (context free grammar) used in programming & AI

Ex: automatic door in offices



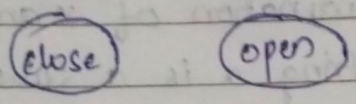
front pad: the sensor detects the person who is about to enter

rear pad: The sensor detects the person behind the door

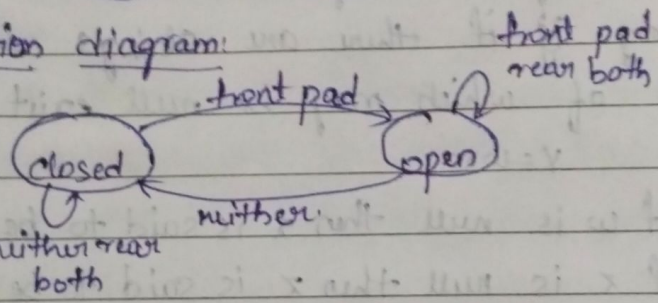
There are 2 states in automatic door - open & close.

four possibilities are:
1) front pad
2) rear pad
3) Both
4) neither

Representations of states in automata: (uses circle always)



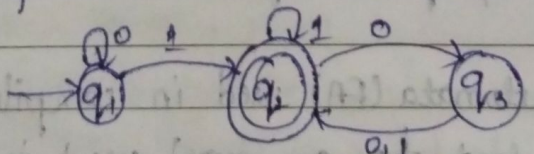
Transition diagram:



Transition table

possibilities / states	front	rear	neither	Both
closed	0	c	c	c
opened	0	0	c	0

Ex:



a finite automata has 3 states
↳ 3 states q_1, q_2, q_3 are given in above transition diagram
↳ initial state q_1 is indicated by arrow
↳ accept (or) final state q_2 is rep with double circles
↳ The arrows going from 1 state to another state are called transitions

GOLKONDA
PAGE NO.
Date / /

→ i/p string 1101, check whether the o/p is accepted or rejected

(i) start with q_1

(ii) read '1', follow transition from $q_1 \to q_2$

(iii) read '1', follow transition from $q_2 \to q_2$

(iv) read '0', follow transition from $q_2 \to q_3$

(v) read '1', follow transition from $q_3 \to q_2$

(vi) accept

Ex: 1, 01, 11, 01010101, 100, 0100, 110000, 0101000000

→ accepted strings

00, 10, 10100 → these are not accepted

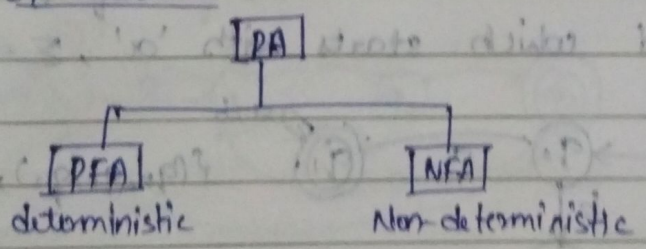
Finite state machine.

The FS sys represents a mathematical model of a sys with certain i/p, the model finally gives certain o/p

→ The i/p when is given to the machine it is processed by various states

→ These states are called intermediate state

Types of automata



A FA is a collection of 5 tuple $(Q, \Sigma, \delta, q_0, F)$

Q - finite set of states, which is non empty

Σ - input alphabet

δ - transition function or mapping func^n using this next state is deterministic.

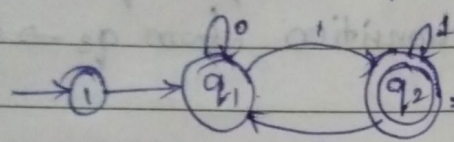
q_0 - initial state i.e $q_0 \in Q$

F - set of final state, $F \subseteq Q$

DFA

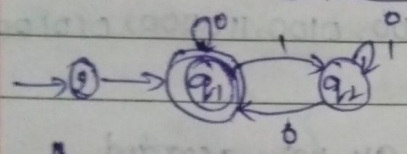
In DFA there is only one part for a specific i/p from current state to the next state. The DFA can be represented by the same 5 tuple described in def^n of finite automata

Ex:



$\Rightarrow (\{q_1, q_2\}, \{0,1\}, \delta, \{q_1\}, \{q_2\})$

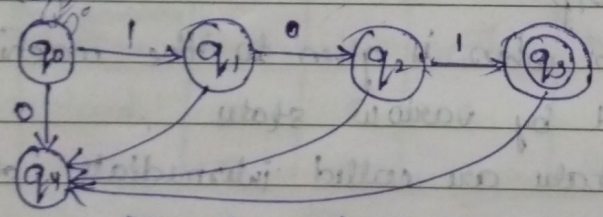
δ is represented as



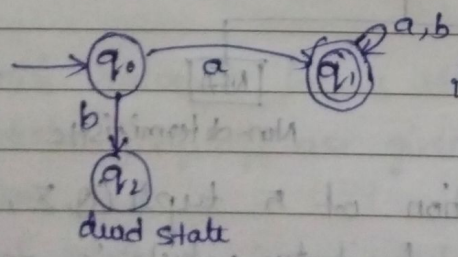
$(\{q_1, q_2\}, \{1,0\}, \delta, \{q_1\}, \{q_1\}) \rightarrow$

state	0	1
q_1	q_1	q_2
q_2	q_1	q_2

define FA which accepts the i/p 101 over the i/p alphabet $\Sigma = (0,1)$

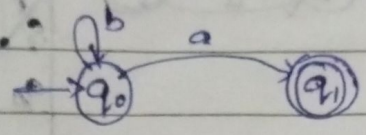


design a DFA in which set of DFA using be accepted which starts with 'a', $\Sigma = (a,b)$

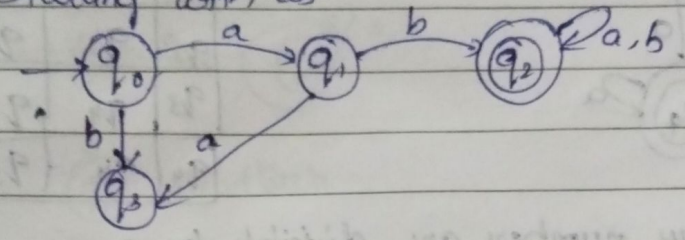


$\{(q_0, q_1, q_2), \{a,b\}, \delta, q_0, q_1\}$

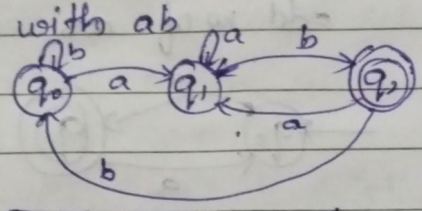
design DFA in which set of all string can be accepted, which ends with 'a'



Starting with ab



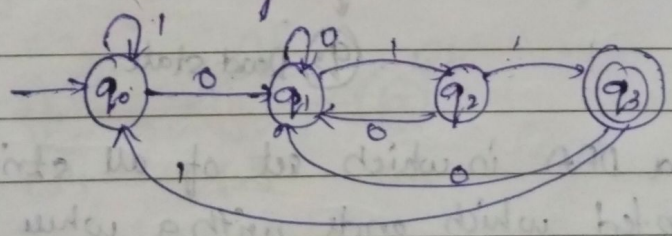
end with ab



self loop must state it has to go one state in DFA

S \ P	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_0

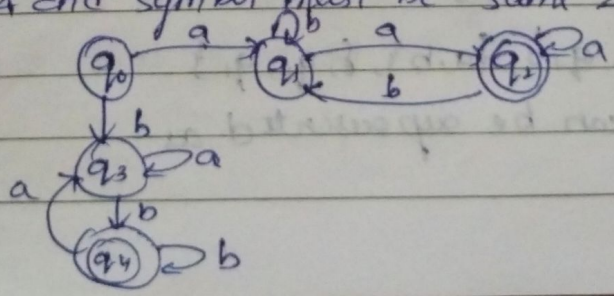
draw a DFA to access string of 0's & 1's ending with the string 011



S \ P	0	1
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_1	q_0

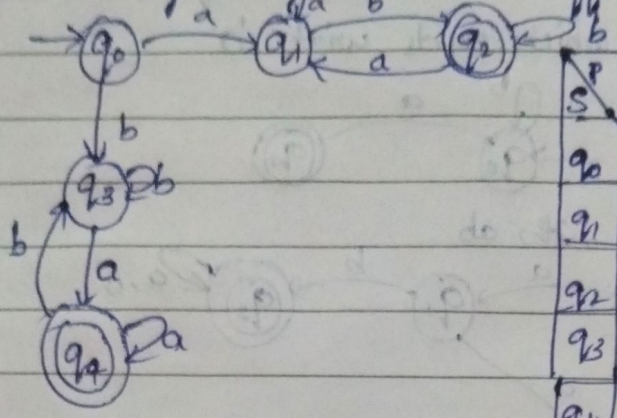
$\{(q_0, q_1, q_2, q_3), (0,1), \delta, q_0, q_3\}$

2) Start & End symbol must be same $\Sigma = a, b$



S \ P	a	b
q_0	q_1	q_3
q_1	q_2	q_1
q_0	q_2	q_1
q_3	q_3	q_4
q_4	q_3	q_4

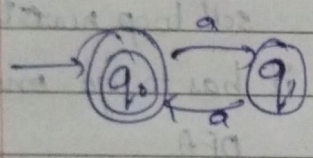
Start and Symbol must be different

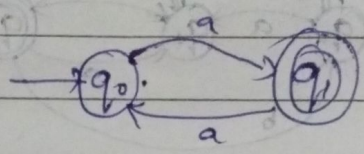


S \ P	a	b
q_0	q_1	q_3
q_1	q_1	q_2
q_2	q_1	q_2
q_3	q_4	q_3
q_4	q_4	q_3

Q) all binary numbers are divisible by 2

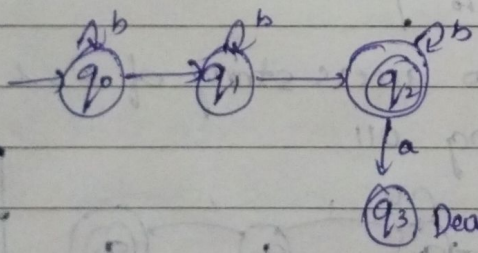
Q) even no. of a's odd no. of a's





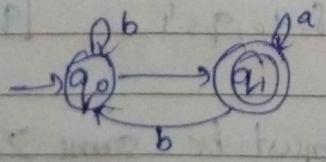
29/8/23

Q) Design a DFA such that no. of a = 2 and there is no restriction over length of b



$\Sigma = (a, b)$

Q) Design a DFA in which set of all string can be accepted which ends with a where $\Sigma = (a, b)$



baaba

$Q = \{q_0, q_1\}$

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

δ q_0 q_1

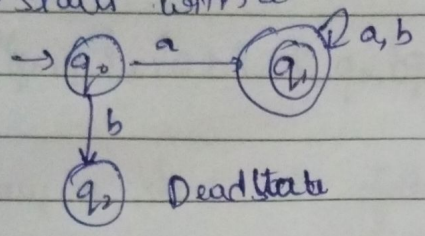
$\{\{q_0, q_1\}, \{a, b\}, \delta, q_0, q_1\}$

δ can be represented as

alphabet

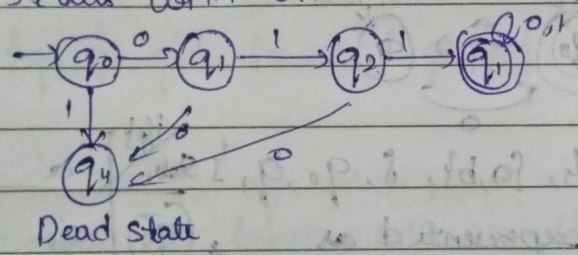
i/p / state	a	b
q_0	q_1	q_0
q_1	q_1	q_0

→ starts with a



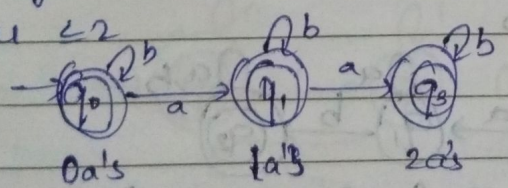
Q) Draw DFA to accept set of strings 0,1 ending with the string 011

→ starts with 011



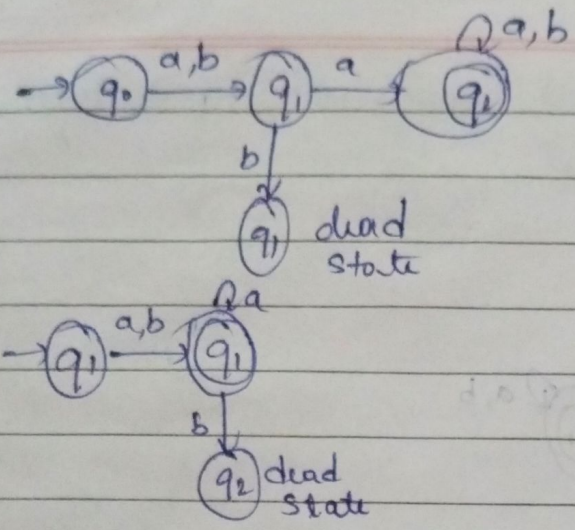
$\{\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, q_0, q_3\}$

Q) Design a DFA in which no. of a's in the string are ≤ 2



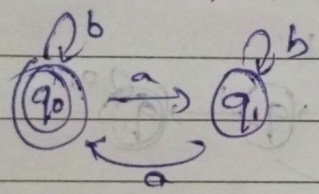
$\{\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_0, q_1, q_3\}\}$

Q) Design a DFA such that 2nd symbol from LHS should be a over $\Sigma = \{a, b\}$



31/8/23

→ Design a DFA in which no of a's are div by 2 and i/p alphabet is a,b

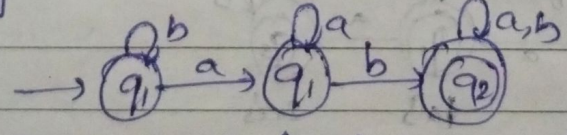


$\{\{q_0, q_1\}, \{a,b\}, \delta, q_0, q_1\}$

δ can be represented as

i/p's / States	a	b
q_0	q_1	q_0
final state q_1	q_0	q_1

→ Design a DFA such that it has to contain the substring as ab

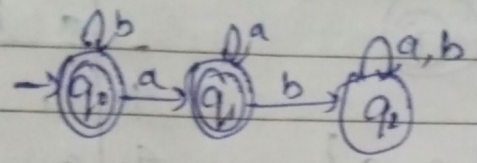


$\{\{q_0, q_1, q_2\}, \{a,b\}, \delta, q_0, q_2\}$

	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_2	q_2

GOLKONDA
PAGE NO.
Date: / /

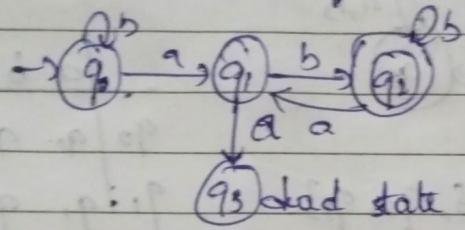
→ Design a DFA such that a should never be followed by b



	a	b
q_0	q_1	q_0
q_1	q_1	•
q_2		

$\{\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_0, q_1\}\}$

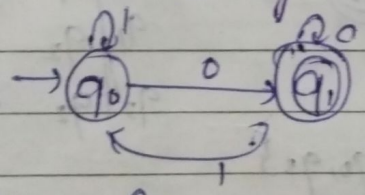
→ Every a should be followed by b

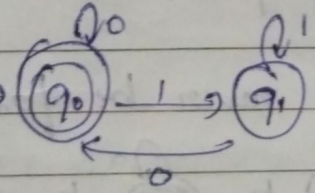


	a	b
q_0	q_1	q_0
q_1	•	q_2
q_2	q_1	q_2

$\{\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, q_2\}$

→ such that all binary no's are div by 2.

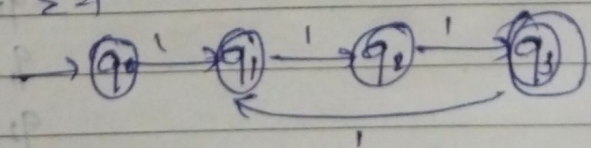


(or) 

(complement)

→ Design a FA (consider DFA) which checks whether the given binary no is div by 3.

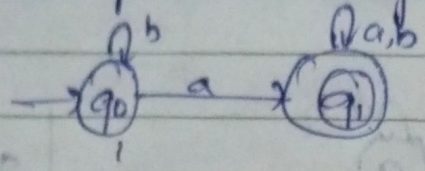
$\Sigma = 1$



	1
q_0	q_1
q_1	q_2
q_2	q_3
q_3	q_1

$\{\{q_0, q_1, q_2, q_3\}, \{1\}, \delta, q_0, q_3\}$

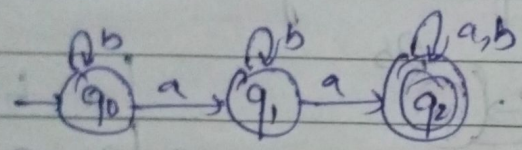
→ That accepts the string with atleast 1 a.



	a	b
q_0	q_1	q_0
q_1	q_1	q_1

$\{\{q_0, q_1\}, \{a, b\}, \delta, q_0, q_1\}$

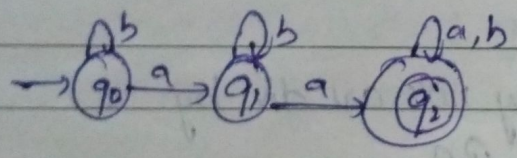
→ Atleast



	a	b
q_0	q_1	q_0
q_1	q_1	q_1

$\{\{q_0, q_1\}, \{a, b\}, \delta, q_0, q_1\}$

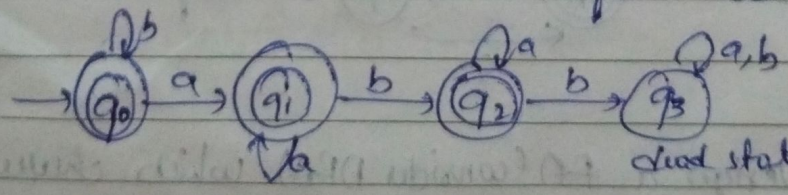
→ Atleast 2 a's



	a	b
q_0	q_1	q_0
q_1	q_2	q_1
q_2	q_2	q_2

$\{\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, q_2\}$

→ such that a should never be followed bb

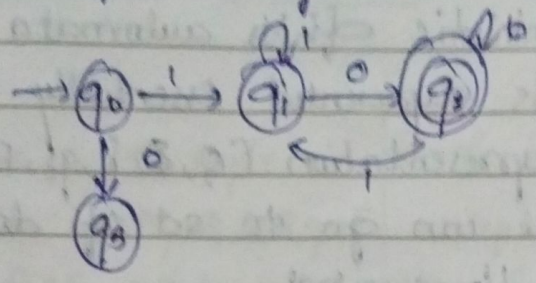


$\{\{q_0, q_1, q_2, q_3\}, \{a, b\}, b, q_0, \{q_0, q_1, q_2\}\}$

	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_2	–

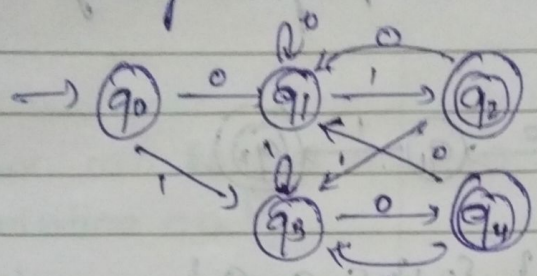
GOLKONDA
PAGE NO.
Date : / /

→ which accepts strings with starts 1 & ends with 0



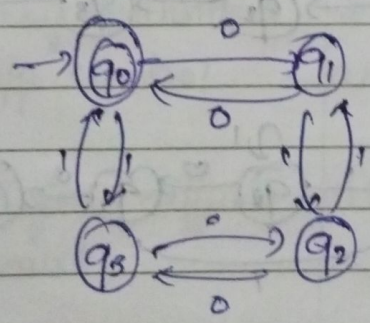
	0	1
q_0	-	q_1
q_1	q_2	q_1
q_2	q_2	q_1

→ accept strings that ends with 01 or 10



	0	1
q_0	q_1	q_3
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_4	q_3
q_4	q_1	q_3

→ which accepts even no of 0's & 1's



This FA will consider 4 diff stages for i/p 0 & 1

1) Even no of 0's & even no of 1's ⟺ q_0 (Final state)
2) Even no of 0's & odd no of 1's ⟺ q_3
3) odd no of 0's & Even no of 1's ⟺ q_1
4) odd no of 0's & odd no of 1's ⟺ q_2

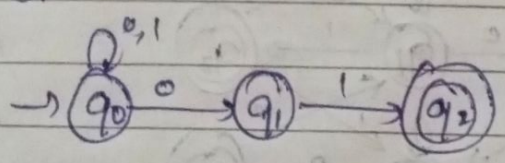
NFA

Non deterministic finite automata.

An NFA is represented essentially like a DFA i.e. 5 tuple representation $(Q, \Sigma, \delta, q_0, F)$

In NFA we can go to set of states over the current I/p symbol.

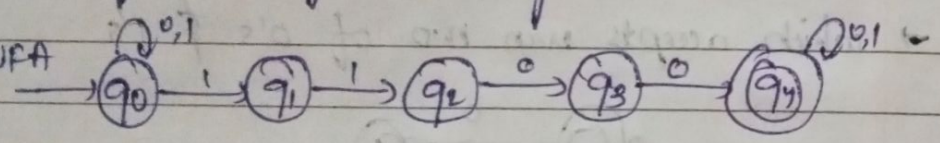
→ Ex: Design NFA accepting all strings ending with 01



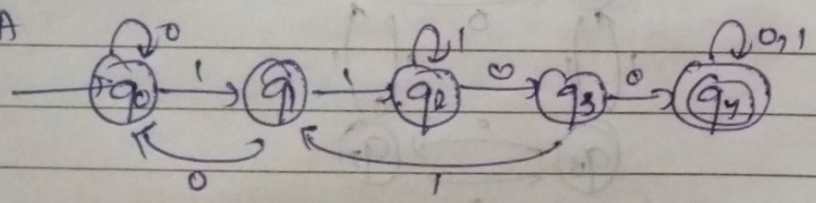
$\{\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, q_2\}$

→ 11 is followed by 00 (string should contain)

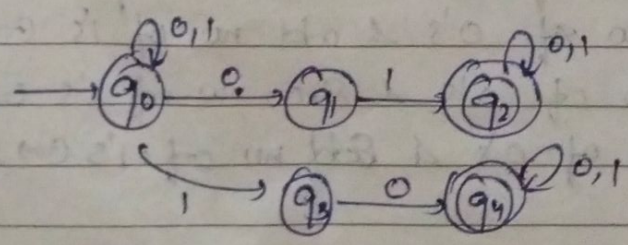
NFA



DFA



→ NFA ⇒ that accepts the string containing either 01 or 10 over $\Sigma = (0,1)$



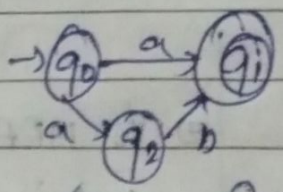
GOLKONDA
PAGE NO.
Date : / /

Difference b/w NFA & DFA

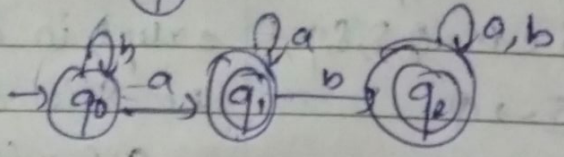
DFA	NFA
1) It is a 5 tuple responses i.e $M = Q, \Sigma, \delta, q_0, F$ $\delta: Q \times \Sigma \to Q$	1) The NFA is same as DFA except in the def^n of δ there δ is defined as follows $\delta: Q \times (\Sigma \cup \epsilon)$ to subset of 2^Q
2) There can be zero or 1 transition on i/p symbol	2) 0 or more transitions
3) No ϵ transition exists, i.e there should not be any transition on an i/p symbol	3) ϵ transitions can exist i.e without any i/p there can be a transition from 1 state to another state
4) It is ~~defined~~ difficult to construct	4) It is easy to construct
5) It takes more space but have speed of computation	5) NFAs are more compact but takes more time.
6) Can recognize a regular language	6) NFA is not well ~~re~~suited for regular lang, as backtracking is needed to explore all choices

→ NFA & DFA to accept the string a (or) ab

NFA



DFA



01/09/23

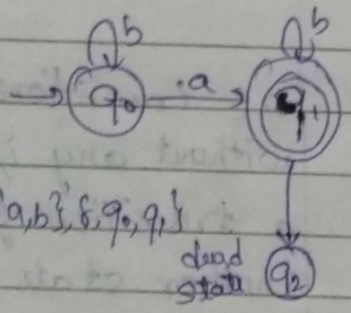
1) Construct DFA for following string over $\Sigma a, b$

a) exactly 1 a

b) atleast 1 a

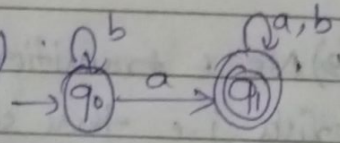
c) atmost 1 a

a)

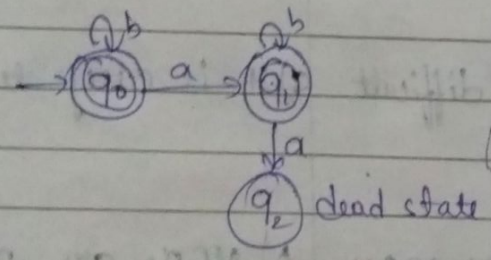


$\{\{q_0, q_1\}, \{a, b\}, \delta, q_0, q_1\}$

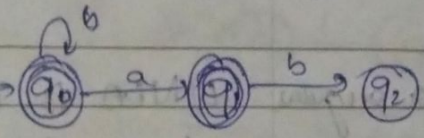
b)



(c)



2) Construct DFA such that it doesn't contain substring



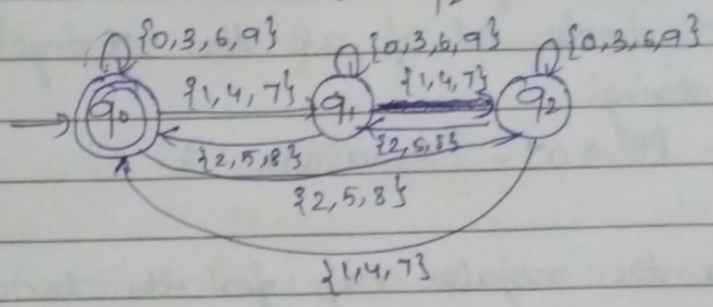
3) Construct DFA for decimal number divisible by 3.

decimal no' = 0 to 9 $\Sigma = 0-9$

rem 0 → {0,3,6,9} → q_0

rem 1 → {1,4,7} → q_1

rem 2 → {2,5,8} → q_2



5-tuple state

$\{\{q_0, q_1, q_2\}, \{0,1,2,\ldots 9\}, \delta, q_0, q_0\}$.

Regular expressions

Let Σ be an alphabet which is used to denote the input set. The regular exp over Σ can be defined as follows

(i) ϕ is a regular expression which denotes the empty set

(ii) ϵ is a RE denotes the set $\{\epsilon\}$ & it is a null string

(iii) for each a in Σ a is a RE and denotes the set {a}

(iv) If r, & s are RE denoting the languages L_1 and L_2 respectively then

a) r+s is equivalent to $L_1 \cup L_2$ i.e union.

b) rs is equivalent to $L_1 L_2$ i.e concatenation

c) $r^* \equiv L_1'$ i.e closure

Note :- (i) r^* is known as clean closure or closure which indicates occurance of r for infinite no of times.

Eg:-

$RE = a^* \Rightarrow \{\epsilon, a, aa, aaa, \ldots\}$

(ii) +ve closure of L can be shown as L^+

The L^+ denotes set of all the strings except the ϵ (or) null string.

Eg:- $RE \rightarrow a^+ = \{a, aa, aaa, \ldots\}$

Ex: (1) Write the regular exp for the language accepting all the combⁿ of a's over $\Sigma = a$.

ans $\Rightarrow a^*$

(2) Design RE for lang accepting all the comb of a's except the null string over $\Sigma = a$

ans $\Rightarrow a^+$

(3) Design RE for lang containing all the string containing any no of a's & b's.

Ans:- $(a+b)^*$

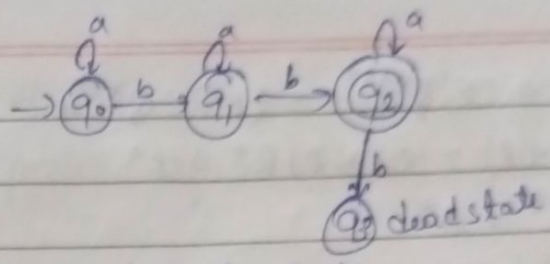
(4) Construct RE for lang containing all strings having any no of a's and b's except the null string

$RE = (a+b)^+$

(5) All the strings which are ending with 00 over the set 0, 1

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

(6) Consisting of exactly 2 b's over the set $\Sigma = a, b$



$(a^*)b(a^*)b(a^*)$

6) Write RE which denotes a lang over the set $\Sigma = 1$ having odd length of strings

$RE = 1(11)^*$

7) Construct RE which denotes a lang L over the set $\Sigma = 0$ having even length of the string

$RE = (00)^*$

8) Write RE to denote the lang L over $\Sigma = a, b$ such that all the strings do not contain the sub string ab.

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

<u>Identity rule</u>

P & Q are equivalent iff P represents the set of strings as q does.

Let P, Q, R are RE then the identity rules are as given bellow

(i) $\epsilon R = R\epsilon = R$

(ii) $\epsilon^* = \epsilon$

(iii) $\phi^* = \epsilon$

(iv) $\phi R = R\phi = \phi$

(v) $\phi + R = R$

6) $R + R = R$

7) $RR^* = R^*R = R^+$

8) $(R^*)^* = R^*$

9) $\epsilon + RR^* = R^*$

10) $(P+Q)R = PR + QR$

11) $(P+Q)^* = (P^*Q^*)^* = (P^*+Q^*)^*$
12) $R^*(\epsilon+R) = (\epsilon+R)R^* = R^*$
13) $(R+\epsilon)^* = R^*$
14) $\epsilon+R^* = R^*$
15) $(PQ)^*P = P(QP)^*$
16) $R^*R+R = R^*R$

06/09/23 NFA to DFA conversion.

→ Let $M = (Q, \Sigma, \delta, q_0, F)$ is a NFA which accepts the language $L(M)$ then should be equivalent DFA denoted by $M' = (Q', \Sigma', \delta', q_0', F')$ such that $L(M) = L(M')$

→ The conversion method will follow the following steps.

S1:- The start state of NFA will be start state of DFA. Hence add q_0 (NFA) to Q' then find the transision from this start state

S2: For each state $[q_0, q_1, q_2, q_3 \ldots q_n]$ in Q' that transition for each i/p symbol Σ can be obtained as

a) $\delta'([q_1, q_2 \ldots], a)$
$= \delta(q_1, a) \cup \delta(q_2, a) \ldots$

b) Add the state $q_1, q_2 \ldots$ to DFA if it isn't already added in Q'.

c) Then find the transision for every i/p symbol from Σ for state $[q_0, q_1, \ldots q_n]$

→ If we get some state $[q_1, q_2 \ldots]$ which isn't in Q' of DFA then add this state to Q'

d) If there is no new state generating the stop the process after finding all the transitions.

S3: For the state $\{q_1, q_2 \ldots\} \in Q$ of DFA. If any state is a final state of NFA then ~~Q, Q~~ $\{q_1, q_2 \ldots\}$ becomes a final state. Thus the set of all the final states belongs to F' of DFA.

1) Convert the given NFA to DFA.

state \ i/p	0	1
q_0	$\{q_0, q_1\}$	q_0
q_1	q_2	q_1
q_2	q_3	q_3
q_3	ϕ	q_2

nfa DFA new state

S1 ⇒ $\delta'([q_0], 0) = \delta'(q_0, 0) = \{q_0, q_1\}$

$\delta'([q_0], 1) = \delta'(q_0, 1) = \{q_0\}$

S2 ⇒ $\delta'([q_0, q_1], 0) = \delta'(q_0, 0) \cup \delta'(q_1, 0)$

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

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

for i/p 1: $\delta'([q_0, q_1], 1) = \delta'(q_0, 1) \cup \delta(q_1, 1)$

$= \{q_0\} \cup \{q_1\}$

$= \{q_0, q_1\}$

$\delta'([q_0, q_1, q_2], 0) = \delta'(q_0, 0) \cup \delta'(q_1, 0), \delta'(q_2, 0)$

$= \{q_0, q_1\} \cup \{q_2\}, \{q_3\}$

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

$\delta'((q_0, q_1, q_2), 1) = \{q_0, q_1, q_3\}$

$\delta'([q_0, q_1, q_2, q_3], 0) = \{q_0, q_1, q_2, q_3\}$

$\delta'([q_0, q_1, q_2], 1] = \{q_0, q_1, q_2, q_3\}$

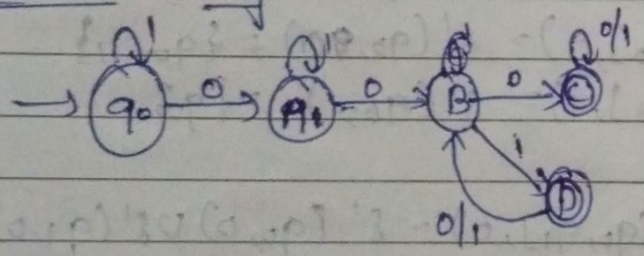
$\delta'([q_0, q_1, q_3]) = \{q_0, q_1, q_2\}$

$\delta'([q_0, q_1, q_3], 1) = \{q_0, q_1, q_2\}$.

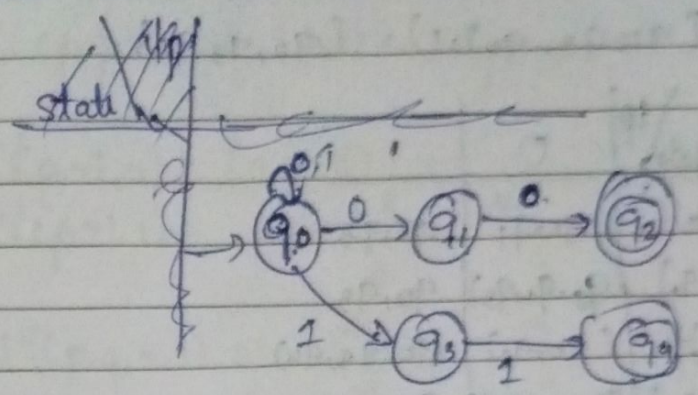
Transition table for DFA

State \ i/p	0	1
q_0	$\{q_0, q_1\}$ A	q_0
$\{q_0, q_1\}$ A	$\{q_0, q_1, q_2\}$ B	$\{q_0, q_1\}$ A
$\{q_0, q_1, q_2\}$ B	$\{q_0, q_1, q_2, q_3\}$ C	$\{q_0, q_1, q_3\}$ D
$\{q_0, q_1, q_2, q_3\}$ C	$\{q_0, q_1, q_2, q_3\}$ C	$\{q_0, q_1, q_2, q_3\}$ C
$\{q_0, q_1, q_3\}$ D	$\{q_0, q_1, q_2\}$ B	$\{q_0, q_1, q_2\}$ B

Transition diag



2) Convert NFA to its DFA.



ip / state	0	1
q_0	(q_0, q_1)	(q_0, q_3)
q_1	q_2	ϕ
q_2	ϕ	ϕ
q_3	ϕ	q_4
q_4	ϕ	ϕ

Sol: $\delta'([q_0], 0) = \{q_0, q_1\}$

$\delta'([q_0], 1) = \{q_0, q_3\}$

$\delta'((q_0, q_1), 0) = \delta'(q_0, 0) \cup \delta'(q_1, 0)$

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

$\delta'((q_0, q_1), 1) = \delta'(q_0, 1) \cup \delta'(q_1, 1)$

$= \{q_0, q_3\}$.

$\delta'((q_0, q_3), 0) = \{q_0, q_1\}$

$\delta'((q_0, q_3), 1) = \{q_0, q_3, q_4\}$

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

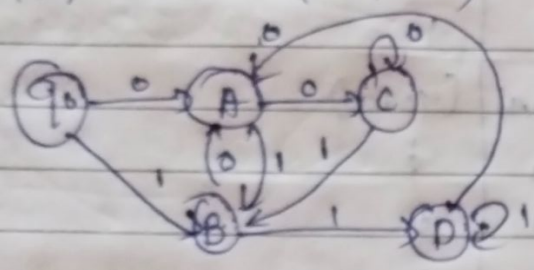
$\delta'((q_0, q_1, q_2), 1) = \{q_0, q_3\}$

$\delta'(\{q_0, q_3, q_4\}, 0) = \{q_0, q_1\}$

$\delta'(\{q_0, q_3, q_4\}, 1) = \{q_0, q_3, q_4\}$

DFA: TD

State \ ip	0	1
q_0	q_0, q_1 (A)	q_0, q_3 (B)
(q_0, q_1) (A)	(q_0, q_1, q_2) (C)	q_0, q_3 (B)
(q_0, q_3) (B)	(q_0, q_1) (A)	(q_0, q_3, q_4) (D)
(q_0, q_1, q_2) (C)	(q_0, q_1, q_2) (C)	(q_0, q_3) (B)
(q_0, q_3, q_4) (D)	(q_0, q_1) (A)	(q_0, q_3, q_4) (D)



oslo/23

Q) Construct DFA from NFA

State \ ip	0	1
p	{p, q}	p
q	r	r
r	s	–
(s)	s	s

$\delta'(p, 0) = \{p, q\}$

$\delta'(p, 1) = \{p\}$

$\delta'((p, q), 0) = \delta'(p, 0) \cup \delta(q, 0) = \{p, q, r\}$

$\delta'((p, q), 1) = \{p, r\}$

$\delta'((p,q,r),0) = \{p,q,r,s\}$
$\delta'((p,q,r),1) = \{p,r\}$

$\delta'((p,r),0) = (p,q,s)$
$\delta'((p,r),1) = (p,r)$

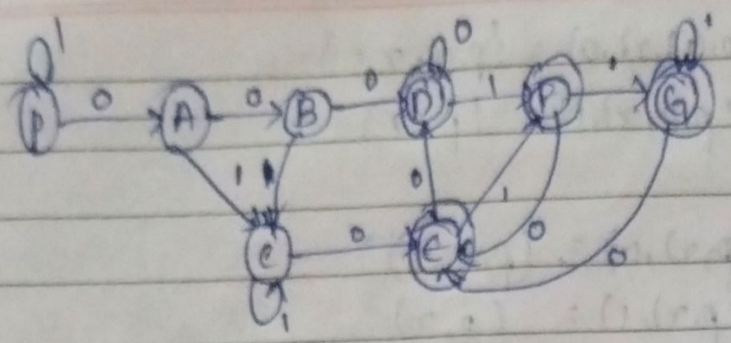
$\delta'((p,q,r,s),0) = (p,q,r,s)$
$\delta'((p,q,r,s),1) = (p,r,s)$

$\delta((p,q,s),0) = (p,q,r,s)$
$\delta((p,q,s),1) = (p,r,s)$

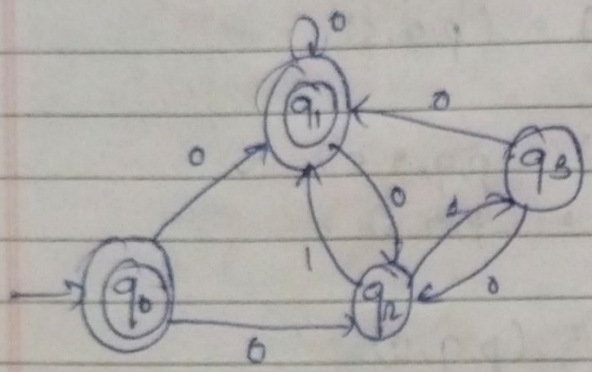
$\delta((p,r,s),0) = (p,q,s)$
$\delta((p,r,s),1) = (p,s)$

$\delta((p,s),0) = (p,q,s)$
$\delta((p,s),1) = (p,s)$

$s \backslash i/p$	0	1
p	(p,q) A	(p)
(p,q) A	(p,q,r) B	(p,r) C
(p,q,r) B	(p,q,r,s) D	(p,r) C
(p,r) C	(p,q,s) E	(p,r) C
* (p,q,r,s) D	(p,q,r,s) D	(p,r,s) F
* (p,q,s) E	(p,q,r,s) D	(p,r,s) F
* (p,r,s) F	(p,q,s) E	(p,s) G
* (p,s) G	(p,q,s) E	(p,s) G



Q2)



δ'	0	1
q_0	(q_1, q_2)	–
q_1	(q_1, q_2)	–
q_2	–	q_1, q_3
q_3	(q_1, q_2)	–

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

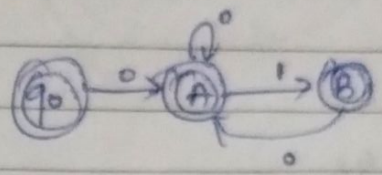
2) $\delta'((q_1, q_2), 0) = (q_1, q_2)$; $\delta'((q_1, q_2), 1) = (q_1, q_3)$

~~$\delta'((q_1), 0) = (q_1, q_2)$~~

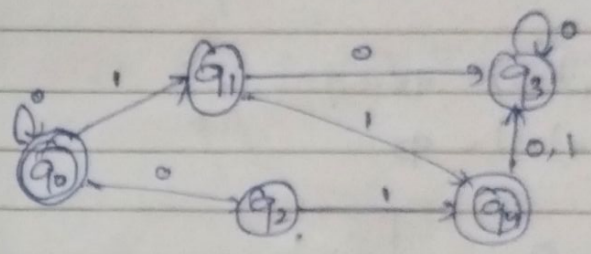
~~$\delta'((q_2, A) = (q_1, q_3)$~~

3) $\delta'((q_1, q_3), 0) = (q_1, q_2)$
$\delta'((q_1, q_3), 1) = \phi$ –

δ'	0	1
$\rightarrow q_0$	(q_1, q_2)	–
$\rightarrow (q_1, q_2)$	(q_1, q_2)	(q_1, q_3)
$\rightarrow (q_1, q_3)$	(q_1, q_2)	–



5)



δ	0	1
q_0	(q_0, q_2)	q_1
q_1	q_3	q_4
q_2	–	q_4
q_3	q_3	–
q_4	q_3	q_3

$\delta'(q_0, 0) = (q_0, q_2)$

$\delta'(q_0, 1) = q_1$

$\delta'((q_0, q_2), 0) = (q_0, q_2)$

$\delta'((q_0, q_2), 1) = (q_1, q_4)$

$\delta'(q_1, 0) = q_3 \quad \delta'(q_1, 1) = q_4$

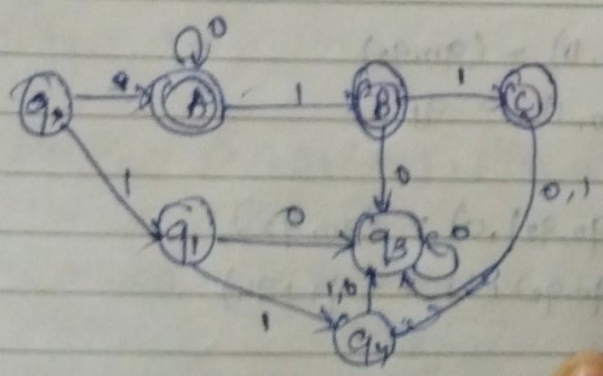
$\delta'((q_1, q_4), 0) = q_3$
$\delta'((q_1, q_4), 1) = (q_3, q_4)$

$\delta'((q_3, 0)) = q_3$
$\delta'((q_3, 1)) = -$

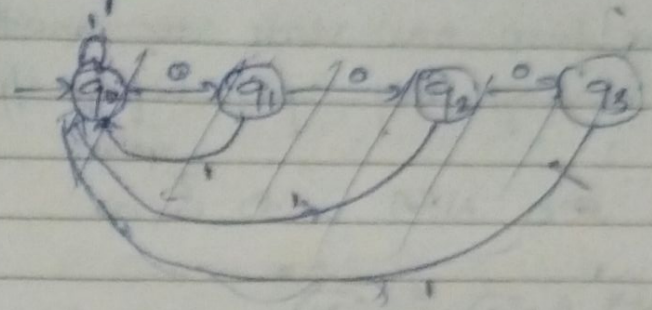
$\delta'((q_4, 0)) = q_3$
$\delta'((q_4), 1) = q_3$

$\delta'((q_3, q_4), 0) = q_3$
$\delta'((q_3, q_4), 1) = q_3$

S	0	1
q_0	(q_0, q_2)	q_1
(q_0, q_2)	(q_0, q_2)	(q_1, q_4)
q_1	q_3	q_4
(q_1, q_4)	q_3	(q_3, q_4)
q_3	q_3	$-$
q_4	q_3	q_3
(q_3, q_4)	q_3	q_3



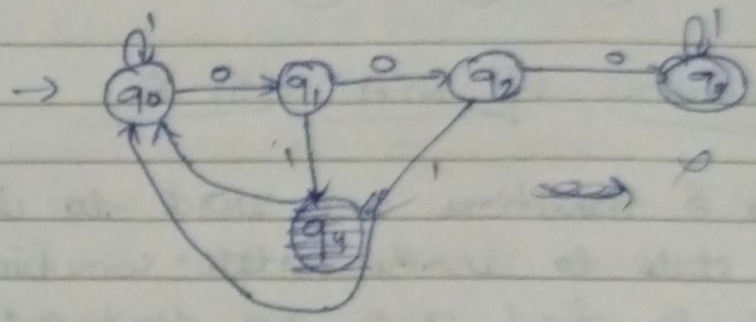
Q) Design DFA with Σ 0,1 accepts the set of all strings with 3 consecutive zeros.



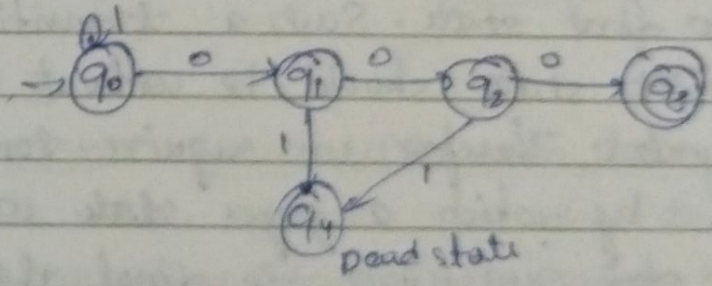
1000

101000

11000111



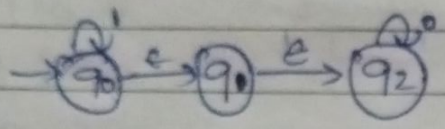
Q) no → 111 only 000



NFA with ε transition.

The ε transition in NFA are given in order to move from one state to another state without having any symbol from i/p set Σ

Consider the NFA with ε as



Significance of NFA with ε

The ε transitions are used to change from one state to another state. Sometime to reach to final state we don't get proper state from start state. In such a case we simply want to reach to certain state which leads to final state. Such a transition to that specific state should be without any input symbol. Therefore we require some ε moves by which a proper state can be obtained for reaching to final state. Thus 'ε moves' plays an important role in NFA.

Def of ε closure:

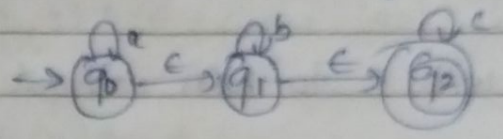
The ε closure of P is a set of all states which are reachable from state P on ε transition such that

→ ε-closure of P = P where P ∈ Q

→ If there exists ε-closure of (P) = {q} and

$\delta(q, \epsilon) = r$ then ϵ-closure$(p) = \{q, r\}$

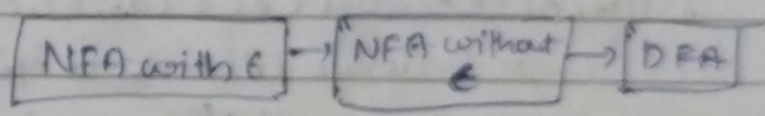
Eg: Find ϵ closure for the following NFA with ϵ



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

Conversion and its equivalence.

The NFA with ϵ can be converted to NFA without ϵ and the NFA without ϵ can be converted to DFA



Conversion from NFA with ϵ to NFA without ϵ

In this method we try to remove all the ϵ transition from the given NFA

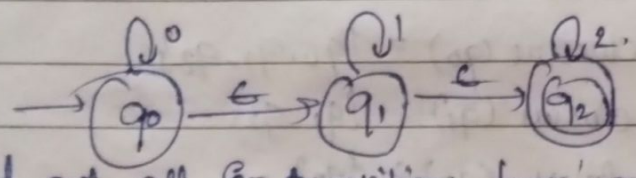
1) Find out all the ϵ transitions from each state from Q.
2) Then obtain the δ' transitions from one state to another state
3) Step 2 is repeated for each input symbol and for each state of given NFA
4) Using the resultant state the transition table for equivalent NFA without ϵ can be built.

Rule for conversion

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

where $\delta'(q, \epsilon) = \epsilon\text{-closure}(q)$

Q) Convert NFA with ϵ to NFA without ϵ.



Find out all ϵ-transitions from each state

S1:- ϵ-closure of $q_0 = \{q_0, q_1, q_2\}$

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

" " " $q_2 = \{q_2\}$

S2:- Find out δ' transitions for each state

$\delta'(q_0, 0) = \epsilon\text{-closure}(\delta(\delta'(q_0, \epsilon), 0))$

$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0), 0))$

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

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

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

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

$= \{q_1, q_2\}$

$\delta'(q_0, 2) = \epsilon\text{-closure}(\delta(q_0, q_1, q_2), 2)$

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

$= q_2$

$\delta'(q_1, 0) = \epsilon\text{-closure}(\delta((q_1, q_2), 0))$

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

$\delta'(q_1, 1) = \epsilon\text{-closure}(q_1) = \{q_1, q_2\}$

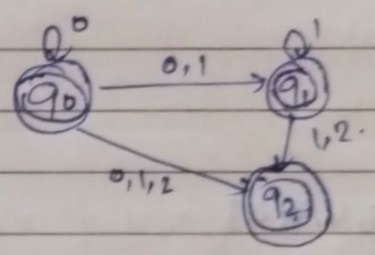
$\delta'(q_1, 2) = \epsilon\text{-closure}(q_2) = \{q_2\}$

$\delta'(q_2, 0) = \phi$

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

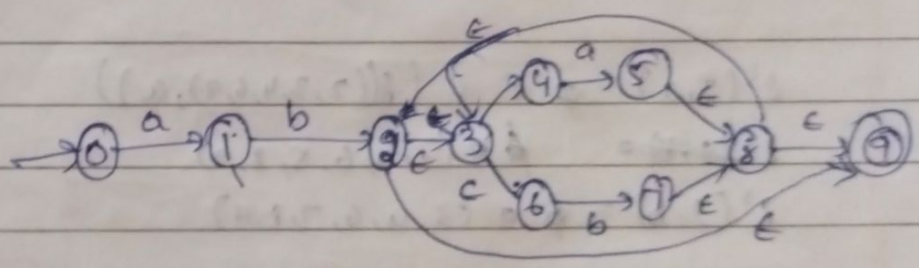
$\delta'(q_2, 2) = q_2$

ϵ \ i	0	1	2
q_0	(q_0, q_1, q_2)	(q_1, q_2)	q_2
q_1	ϕ	(q_1, q_2)	(q_2)
q_2	ϕ	ϕ	q_2



q_0, q_1, q_2 are F.S bcoz ϵ-closure of q_0 & q_1 has q_2 in it.

Q2)



~~δ'~~ ϵ-C of 0 = {0, 1, 2, 3, 4, 5, 8, 9}

1 = {1, 2, 3, 4, 5, 8, 9} ; 2 = {2, 3, 5, 8, 9}

3 = {3, 4, 5, 8, 9}

4 = {4, 5, 8, 9}

5 = {5, 8, 9}

8 = {8, 9}

9 = {9}

PAGE NO.
Date : / /

ϵ-C of 0 = {0}
ϵ-C of 1 = {1}
of 2 = {2, 3, 4, 6, 9}
of 3 ⇒ {3, 4, 6}
4 ⇒ {4}.
5 = {5, 8, 3, 4, 9, 6}
6 = {6}
7 = {3, 4, 6, 7, 8, 9}
8 = {4, 4, 6, 8, 9}
9 = {9}

$\delta'(0,a)$ = E-closure ($\delta(\delta'(0,\epsilon), a)$)
= ϵ-closure ($\delta(\epsilon\text{-}C(0)), a$)
= ϵ-C ($\delta(0,a)$) = ϵ-C(1) = {1}

$\delta'(0,b) = \phi$

$\delta'(1,a) = \phi$
$\delta'(1,b)$ = {2, 3, 4, 6, 9}

$\delta'(2,a) = \phi$ ϵ-C ($\delta((2,3,4,6,9), a)$)
~~$\delta'(2,b)$~~ = ϕ (3, 4, 6, 5, 8, 9)
$\delta'(2,b) = \phi$ (3, 4, 6, 7, 8, 9)

$\delta'(3,a) = \phi$ ϵ-C ($\delta((3,4,6), a)$) = (3, 4, 6, 5, 8, 9)
$\delta'(3,b) = \phi$ ~~ϵ-C~~ = (3, 4, 6, 7, 8, 9)

$\delta'(4,a)$ = {5, 8, 3, 4, 9, 6}

$\delta'(5,a) = \{5, 8, 3, 4, 9, 6\}$

$\delta'(5,b) = \{3, 4, 6, 7, 8, 9\}$.

$\delta'(6,a) = \phi$

$\delta'(6,b) = \{3, 4, 6, 7, 8, 9\}$

$\delta'(7,a) = \{3, 4, 5, 6, 8, 9\}$

$\delta'(7,b) = \{3, 4, 6, 7, 8, 9\}$

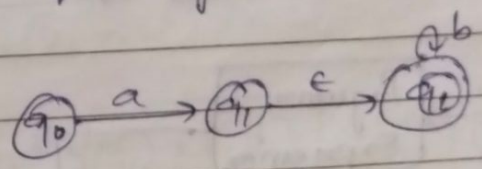
$\delta'(8,a) = \{3, 4, 5, 6, 8, 9\}$.

$\delta'(8,b) = \{3, 4, 6, 7, 8, 9\}$.

$\delta'(9,a) =$

13/09/23.

Q) Convert the following NFA with epsilon to without ϵ.



ϵ-C of $q_0 = \{q_0, q_1, q_2\}$

ϵ-C of $q_1 = \{q_1, q_2\}$

ϵ-C of $q_2 = \{q_2\}$

$\delta'(q_0, a) = \epsilon\text{-}C(\delta(\epsilon\text{-}C(q_0)), a)$

$= \{q_1, q_2\}$

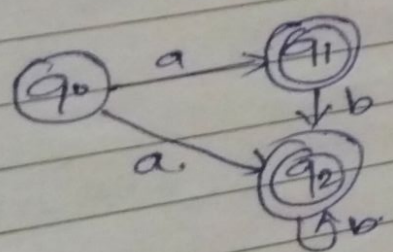
$\delta'(q_0, b) = \phi$

GOLKONDA
PAGE NO.
Date : / /

F.T $\delta'(q_1, a) = \phi$
$\delta'(q_1, b) = q_2$

F.T $\delta'(q_2, a) = \phi$
$\delta'(q_2, b) = \{q_2\}$

δ' \ i	a	b
q_0	$\{q_1, q_2\}$	ϕ
F.T q_1	ϕ	q_2
F.T q_2	ϕ	q_2



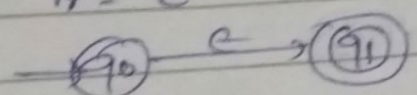
Construct a finite automata for a given regular expression.

Regular Expression → NFA with ε-transition

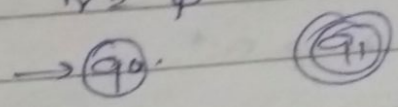
→ If 'r' be a reg exp then there exist a NFA with ε-moves which accepts L(R)

→ Construction of NFA with ε-moves.

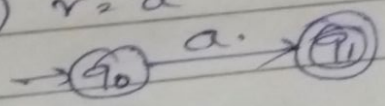
1) $r = \epsilon$



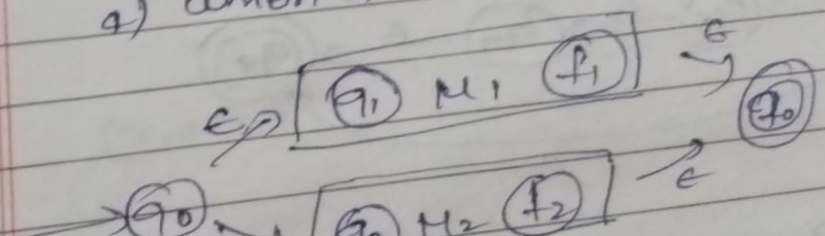
2) $r = \phi$



3) $r = a$

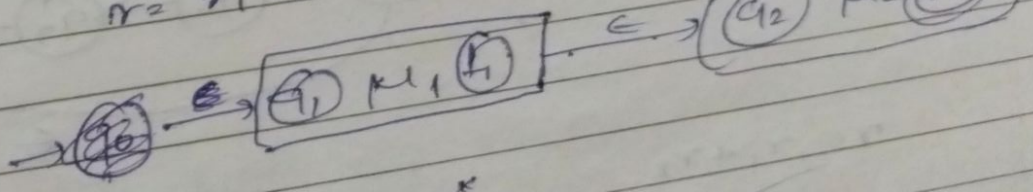


4) union ⇒ $r = r_1 + r_2$

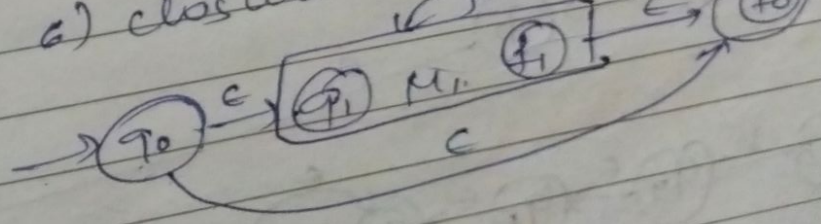


5) concatenation.

$r = r_1 r_2$



6) closure $r = r_1^*$

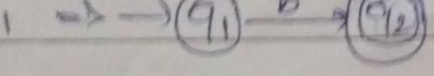


Q) Construct NFA for RE $b + ba^*$

$r = b + ba^*$

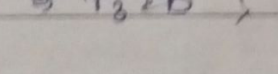
$r_1 = b$; $r_2 = ba^*$

→ $r_1 = 1$ ⇒

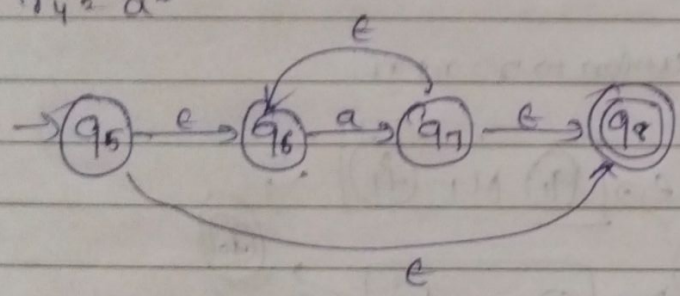


→ $r_2 = ba^*$ ⇒ $r_3 = b$; $r_4 = a^*$

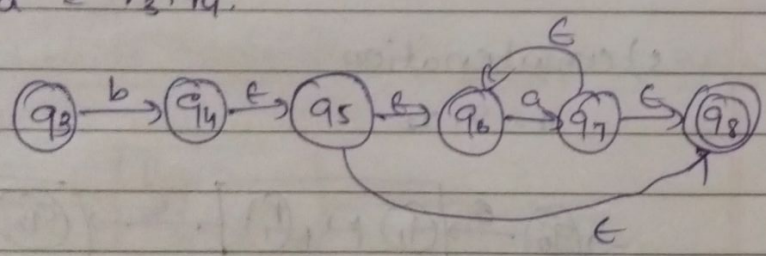
r_3 =



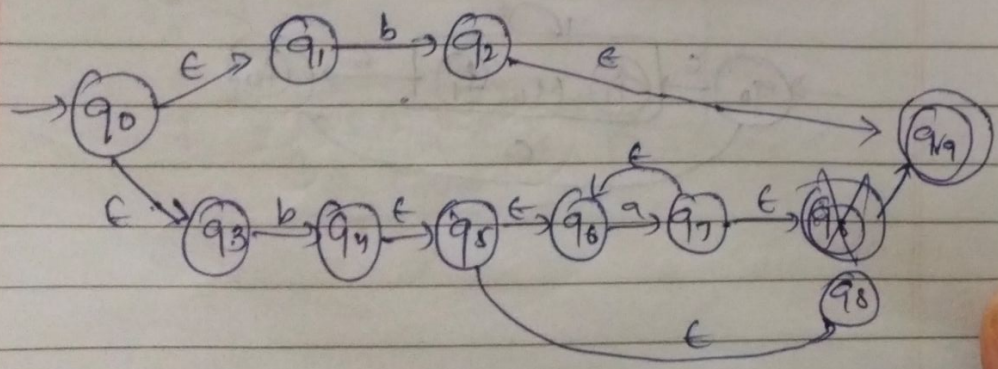
$r_4 = a^*$



→ $r_2 = ba^* = r_3 \cdot r_4$.

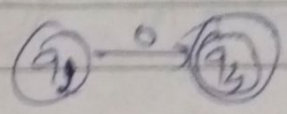


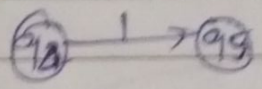
$r = r_1 + r_2$.



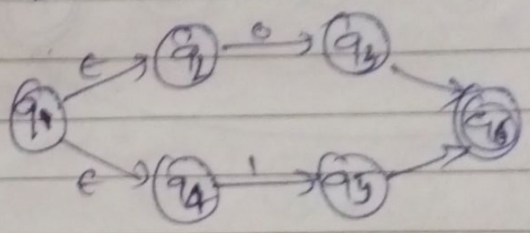
Q) Construct NFA with ε-moves for RE $(0+1)^*$

$r = (0+1)^*$

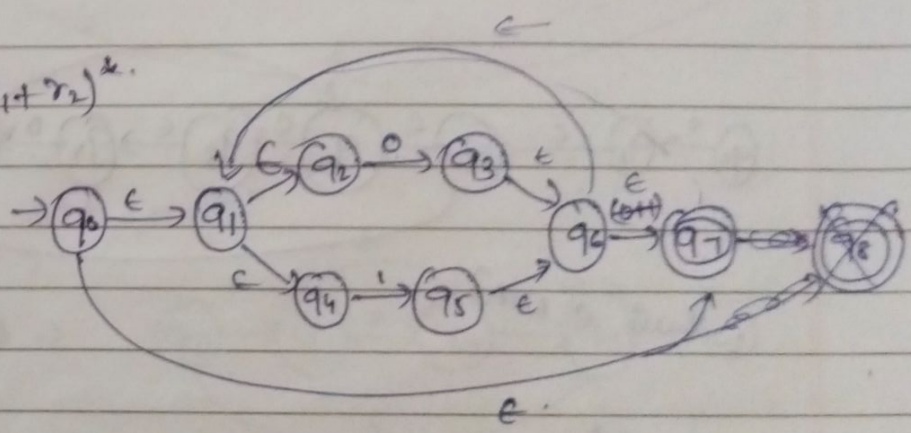
$r_1 = 0 \Rightarrow$ 

$r_2 = 1 \Rightarrow$ 

$r_1 + r_2 \Rightarrow$



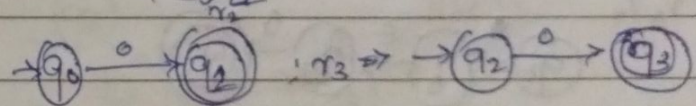
$(r_1 + r_2)^*$

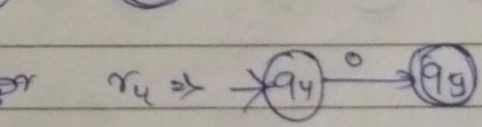


Q) Construct NFA for lang having odd no. of zeros over $\Sigma = 0$

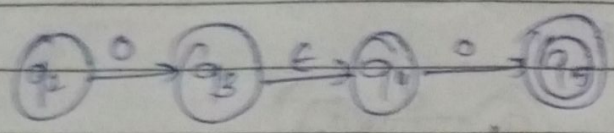
RE = $0(00)^*$

$r_1 \; (r_3 r_4)$

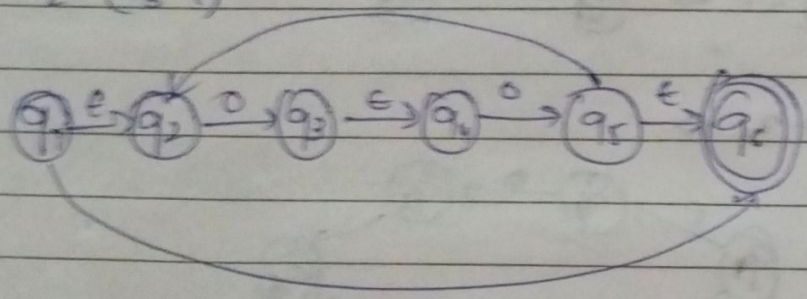
$r_1 \Rightarrow$ 

$r_2 \Rightarrow \epsilon r_3 r_4$ $\quad r_4 \Rightarrow$ 

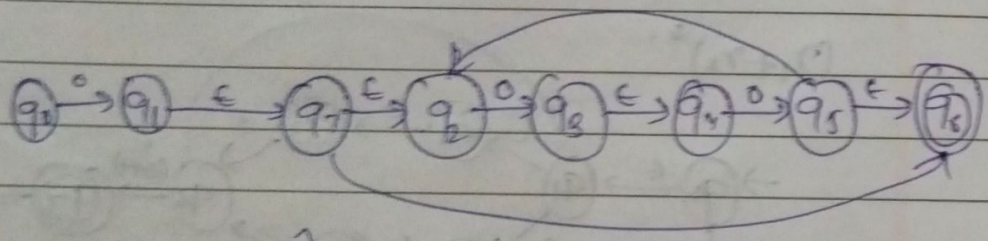
$r_3 r_4 =$



$r_2 = (r_3 r_4)^*$



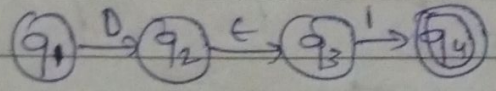
$r = r_1 + r_2$



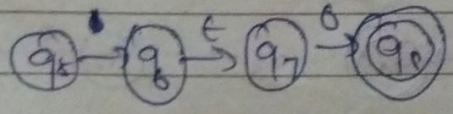
(without ϵ NFA)

5) $\underset{r_1 r_2}{(01} + \underset{r_3 r_4}{10)^*}$

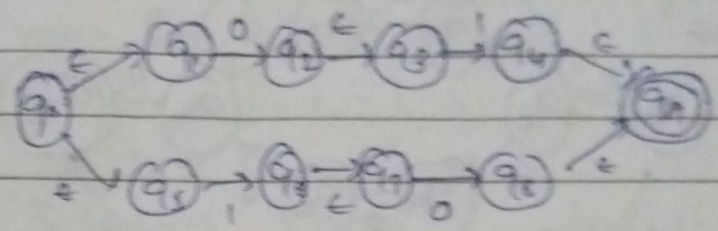
$r_1 r_2$



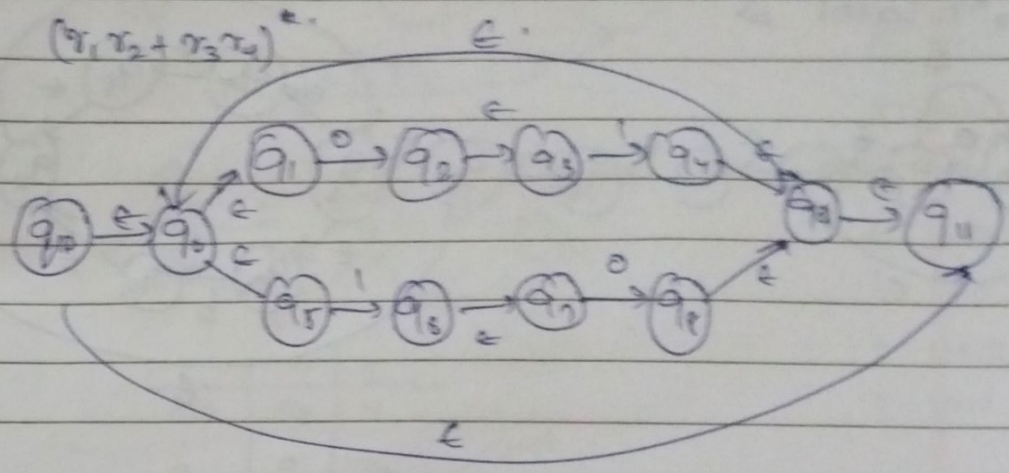
$r_3 r_4$



$r_1 r_2 + r_3 r_4$

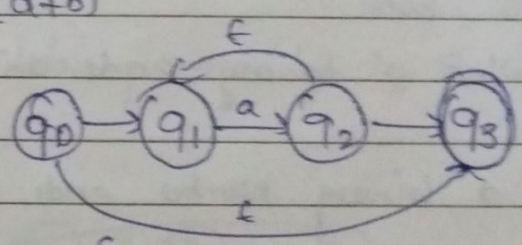


$(r_1 r_2 + r_3 r_4)^*$

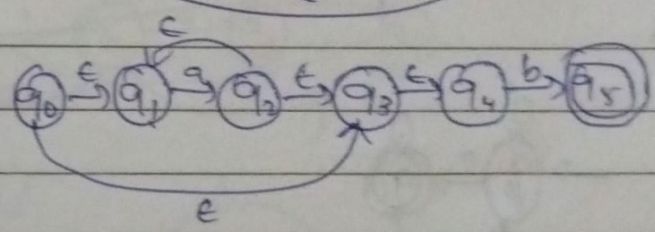


Q) $RE = a^* b (a+b)^*$

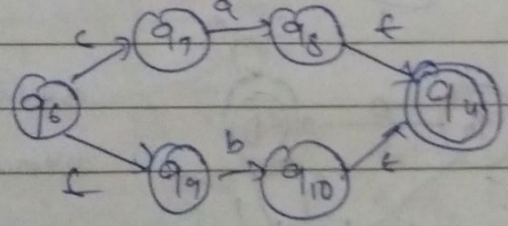
a^* ⇒



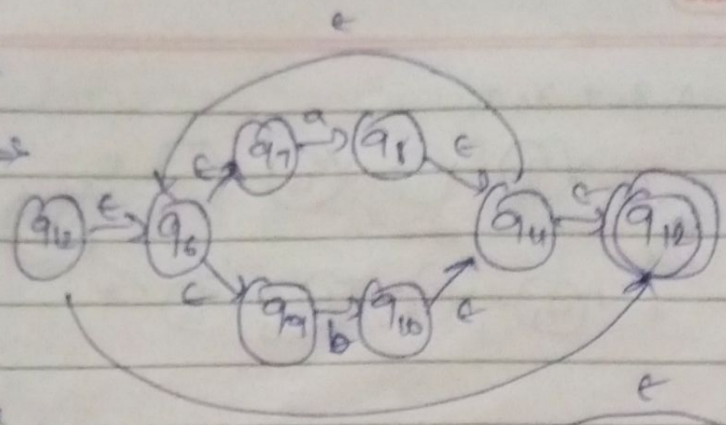
$a^* b$ ⇒



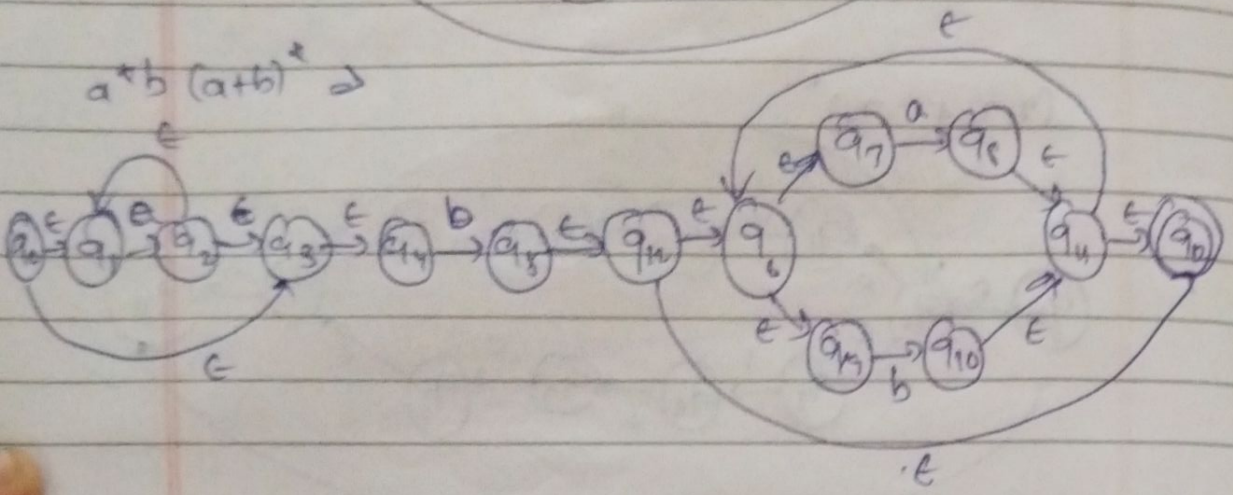
$(a+b)$ ⇒



$(a+b)^*$ ⇒



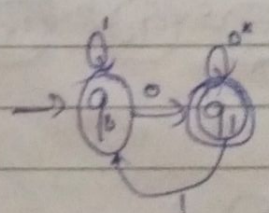
$a^*b(a+b)^*$ ⇒

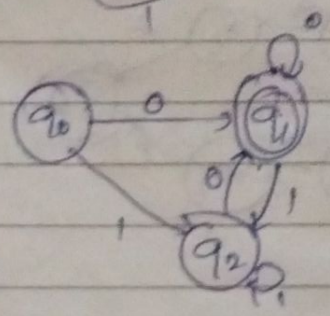


Q) Design DFA to check whether given binary num is even.

Note:- 1) When a binary ends with 0 it is always even.

2) When a binary number ends with 1 its always odd.





odd → q_2 - final state

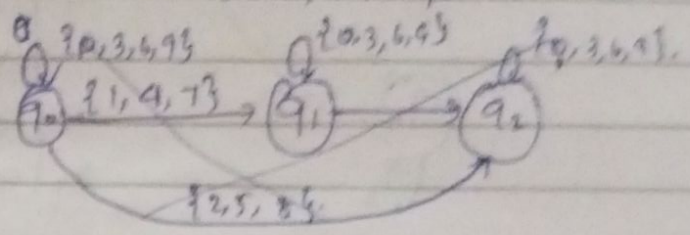
even → q_1 - final state

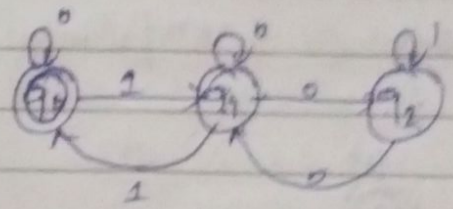
Q) DFA. ST binary number is div by 3 over $\Sigma = 0,1$

rem = 0. = {0, 3, 6, 9} q_0

rem = 1 ⇒ {1, 4, 7} q_1

2 ⇒ {2, 5, 8} q_2



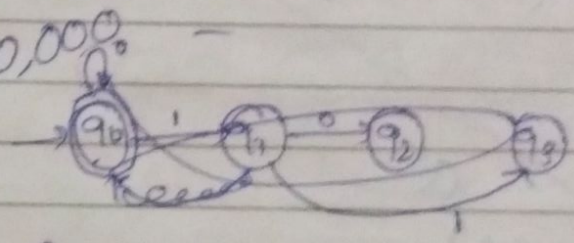


	0	1
q_0	q_0	q_1
q_1	q_2	q_0
q_2	q_1	q_2

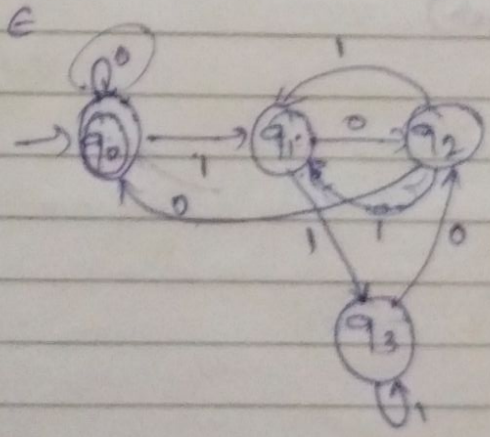
0 - 000 - 0
1 - 001 - 1
2 - 010 - 2
3 - 011 - 0
4 - 100 - 1
5 - 101 - 2
6 - 110 - 0
7 - 111 - 1
8 - 1000 - 2
9 - 1001 - 0

Q) DFA for binary num div by 4.

ε, 0, 00, 000



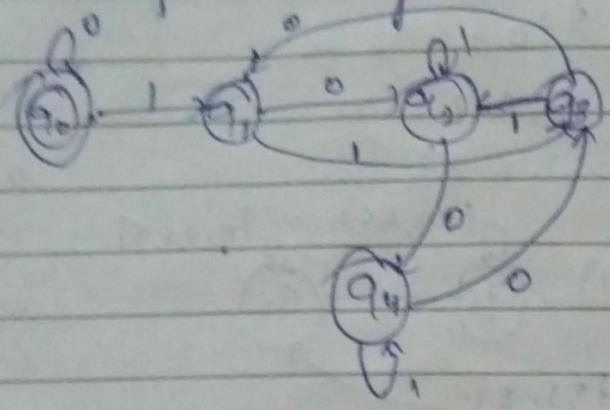
1000



	0	1
q_0	q_0	q_1
q_1	q_2	q_3
q_2	q_0	q_1
q_3	q_2	q_3

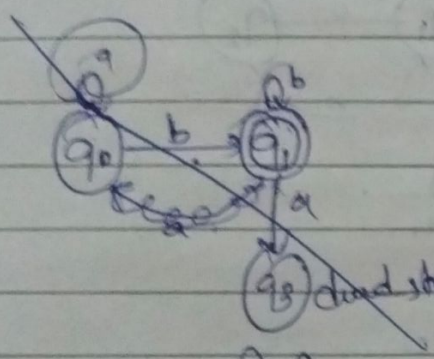
0 000 0
1 001 1
2 010 2
3 011 3
4 100 0
5 101 1
6 110 2
7 111 3
8 - 1000 - 0
9 - 1001 - 1

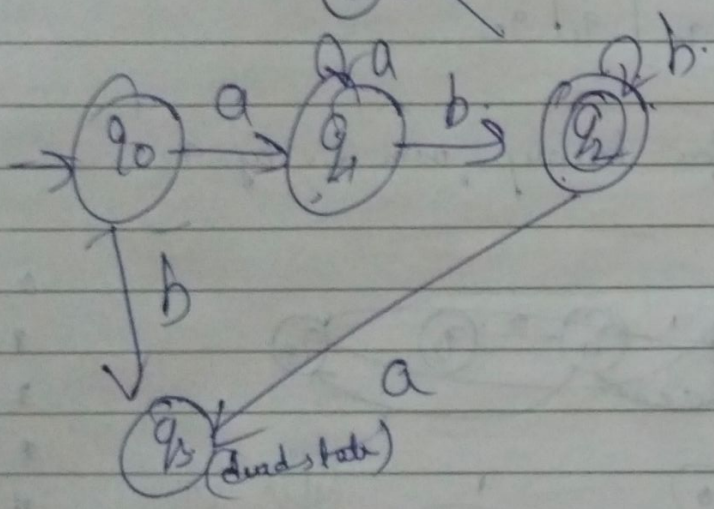
8) DFA for div by 5



	0	1
q_0	q_0	q_1
q_1	q_2	q_3
q_2	q_4	q_0
q_3	q_1	q_2
q_4	q_3	q_4

9) DFA such that $L = \{a^n b^m \mid n, m \geq 1\}$ over $\Sigma = \{a, b\}$

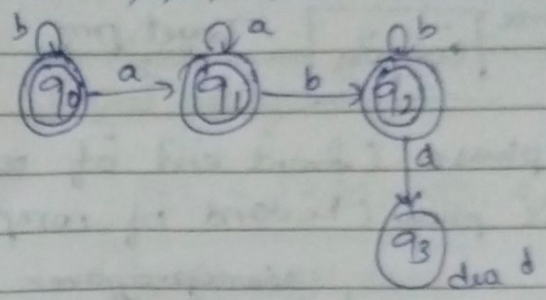


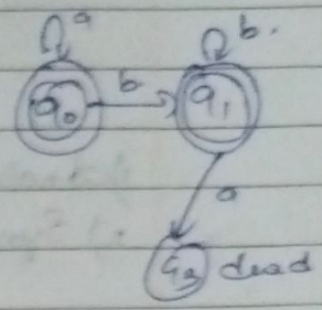


aaaaa

Q) $L = \{a^n b^m \mid n, m \geq 0\}$

$L = \{\epsilon, b, a, ab, abb \ldots, aaab \ldots\}$

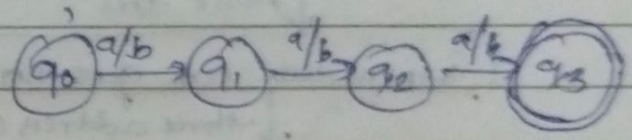




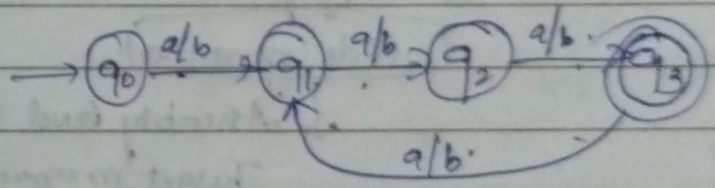
Q) Design a DFA such that i) string with a's & b's and length is exactly 3

(ii) string with a's & b's length is div by 3

exactly 3



length div by 3



Sadh n a/b 3.

20/9/23

Phases of compiler.

Source prog → [compiler] → Target prog

1) Analysis phase (front end of compiler)
2) Synthesis phase (backend of compiler)

source program:-
↓ character stream
[Lexical Analyzer] (also called as scanner)
↓ Tokens
[Syntax analyzer]
↓ parse tree
[Semantic analyzer]
↓ parse tree
[Intermediate code generator]
↓ three address code
[Code optimization]
↓ optimized three add code
[Code generator]
↓ Assembly level language
Target program

Lexical Analyzer:

→ Also called as scanner
→ reads chars from left to right one char at a time.

Ex: position := initial + rate * 60

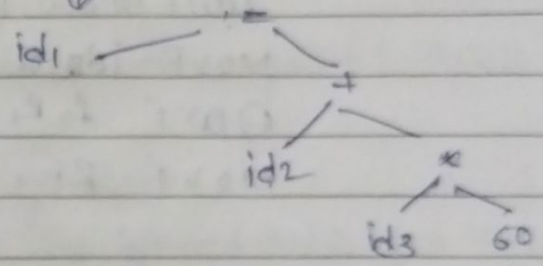
id1 = id2 + id3 * 60

↓

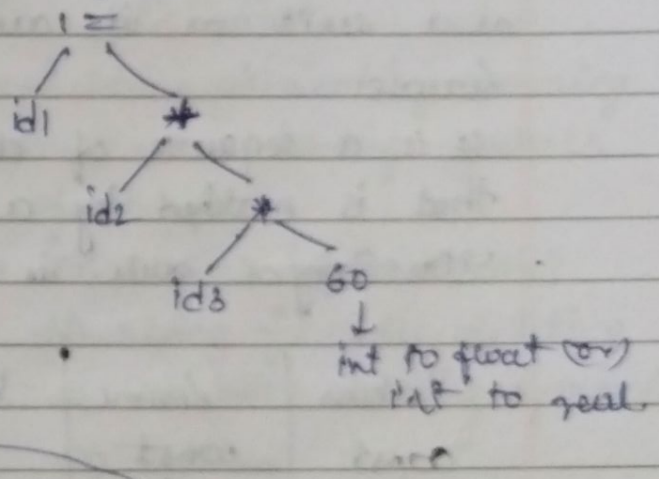
Lexical analyzer

id1, :=, id2, +, id3, *, 60

Syntax analyzer



Semantic analyzer



Intermediate code generator

t1 = int to float (60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3

↓

$\boxed{\text{code optimizer}}$

$t_1 = id3 * 60.0$

$id_1 = id2 + t_1$

(or)

$t_1 = id3 * 60.0$

$t_2 = id2 + t_1$

$id1 = t_2$

↓

$\boxed{\text{code generator}}$

MOVF id3, R_2

MULF #600, R_2 // $R_2 = R_2 * 600$

MOVF id2, R_1 // $R_1 = id2$

ADDF $R_2 R_1$ // $R_1 = R_1 + R_2$

MOVF R_1, id1 // id1 = R_1

Token:

→ It is a sequence of characters that are treated as a unit as it can't be further breakdown.

Lexeme:

It is a sequence of characters in source prg that is matched by a pattern for a token.

↳ Describing a rule is called pattern

Tokens	Lexeme	Informational Descp of pattern
const	const	const
if	if	
relation	<, >, =, <=, >=	<, >=, <=, >
id	pi count pr	Letter followed by letters & Digits
num	3.14	any num const.

21/9/23

Q) Construct phases of the compiler for i=i*70+j+2.

The tokens and associated attribute values for the fort-ran statement e=m*c**2

$$\underset{id_1}{e} = \underset{id_2}{m} * \underset{id_3}{c} ** 2$$

<id1, point to the symbol table for value of e>

<assignment operator>

<m-id 2, point to the symbol table for value m>

<*multiplication operator>

<id3, point to the symbol table for value of c>

<exp-operator>

<2-number>

Term	Defination
1) prefix	→ A string obtained by removing 0 or more trading symbols of string Ex: abc → {e, a, ab, abc}
2) suffix	→ A string form by deleting 0 or more leading symbols of string Ex: abc {abc, bc, c, ∈}
3) substring	→ A string obtained by deleting a prefix and suffix from given string Ex: abc {a, b, c, ab, bc, abc, ∈}

Note
L∪D (L → Letter, D → digit)

1) L∪D is the set of letters and digits
2) LD is the set of string consisting of a letter followed by digit
3) L^4 is the set of all 4 letter string
4) L^* is the set of all strings of letters including ε (single, multi)
5) L(L∪D)* is the set of all strings of letters and digits but begining with a letter.
6) D^+ is the set of all strings of one or more digit doesn't include ε

Recognition of token

Consider the following grammar.

stmt → if exp then stmt |
if exp then stmt else stmt/ε
exp → term relop (relational op) term/term
term → id/num

left hand → terminal.

In the above grammar the tokens are if, then, else, relop, id & num

---

E → E+T/T
T → T*F/F
F → (E) /id
Tokens → +, *, (,), id

~~Tokens → +, *, c~~

if → if

then → then

else → else

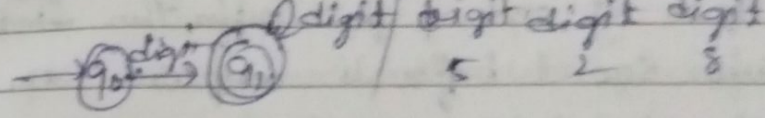
relop → <, >, <=, >=, <>

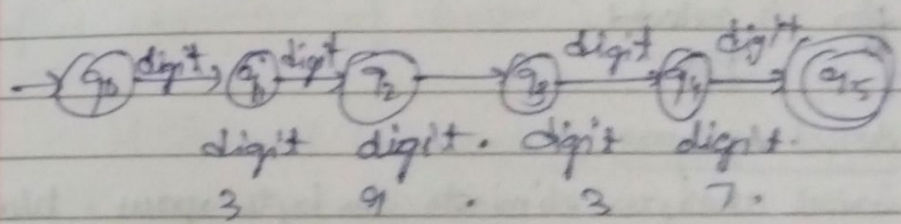
id → letter (letter / digit)*

num → digit +, digit.*

digit → 0-9

Draw the transition diagram 5280 & 39.37





Construct phases of a compiler for a given eq i = i * 70 + f + 2

22/9/23

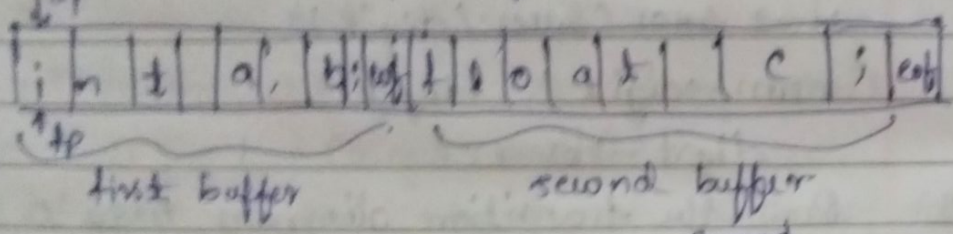
Lexical analysis

int a, b;

float c;

Input Buffering - fp forward pointer -

bp - begin pointer.



→ fp is at end of first buffer then ~~first~~ second is reloaded & fp is here

→ When second buffer is completed the first buffer is reloaded.

Lexical analyzer

→ Lexical analyzer eliminates the byte spaces, blank spaces, tab spaces and new line characters in the source program.

→ It also identifies the keywords, identifiers, constants and punctuation marks

Functions of lexical analyzer

(i) Read input characters from source program

(ii) Dividing the program into valid tokens.

(iii) Remove byte space characters.

(iv) Remove comments.

(v) Helps to identify tokens into the symbol table.

(vi) It generates lexical errors.

Lexical error: A character sequence which is not possible to scan any valid token is a

Lexical error

if(a<b) ⇒ ~~if~~(a<b)

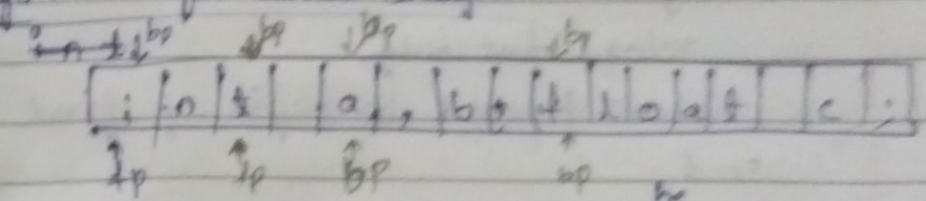
→ lexical error

→ Lexical errors can be of

(i) spelling error (ii) Exceeding length of identifier.

(iii) appearance of illegal character

Algorithm for lexical analyzer



→ begin point & forward point will be pointing to the first character of lexeme

If (forward had end of the first half) then
begin
~~read~~ reload second half,
forward = forward + 1;
else if (forward had end of the second half) then
begin
reload first half;
move forward to begin of the first half;
end;

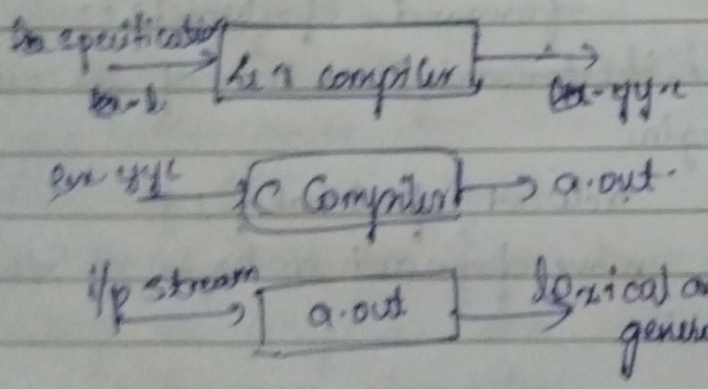
Sentinals

~~if (forward == end) then~~
~~begin~~
~~forward = forward + 1;~~
if(forward == eof) then
begin

if (forward at end of 1st half) then
begin
reload 2nd half;
forward = forward + 1;
else
if (forward at end of second half) then
begin
reload first half;
move forward to begin of first half;
end
else [otherwise]
terminate the lexical analysis

LEX (lexical analyzer generator)

→ lex is a tool or computer program used to generate ~~and~~ the ~~maintain~~ lexical analyzer generator



→ A lex prog consists of 3 parts
i) Declaration
ii) Translation ~~rules~~ rules
iii) Auxiliary procedures

Declaration

→ In Declaration, define the variables in lex prog

Translation ~~Tools~~ Rules

```
%   %
p1 {action1}
p2 {action2}
pn {actionn}
%   %
```

→ where each p_n is a regular expression and each action is a program fragment describing what actions were performs.

Auxiliary procedures

These procedures can be compiled separately and loaded with lexical analyzer

```
DIGITS [0-9]
ID [a-z][a-z][0-9]
%   %
int|float|char|boolean
DIGITS+
DIGIT+ . DIGIT+
ID+
printf ("keyword", yytext);
printf ("Integer", yytext);
    " ("floatpoint", "
    " ("identifier", "
    " ("open parthesis", "
    " ("relop", "
    " ("Closed parenthesis", "
    " (punctuation marks, ")
```

<|>|<=|>=|==|<>
)
,|**);
% %

yylex() function makes a call to the yywrap function when it encounters the end of the i/p otherwise it scans the prog continuously

→ gedit filename

sir filename

gcc lex.yy.c

./a.out

int a,b

ctrl+D

int keyword

a identifier

, punctuation

b identifier

; punctuation

21/09/23 UNIT - II:

Context Free Grammar (CFG)

A grammar $G = (V, T, P, S)$ is said to be CFG where 'V' is a finite set of non terminals, 'T' finite set of terminals, 'P' is a set of production rules that is $A \to \alpha$ where 'A' is a non-terminal symbol and α is a string of terminals and non-terminals in the given grammar. 'S' is a start symbol of the Grammar (non-terminal)

Notational Conventions

→ Terminals are lower case letters such as a, b, c ... z digit (0 ... 9) are terminals, operator symbols like '+', '-', '*', '/', '%', punctuation symbols are terminals (,), {, }, [,]. Bold phase strings are terminals in the given grammar.

→ Non-Terminals symbols are upper case symbols such as (A, B, C ... Z). Start symbol is always non-terminals. Lower case italic names are non terminals.

→ If $A \to \alpha_1, A \to \alpha_2, A \to \alpha_3 .. A \to \alpha_n$ is a set of production may be written as $A \to \alpha_1 | \alpha_2 \dots | \alpha_n$.

Eg: $E \to E+T / T \to E \to E+T$; $E \to T$

$T \to T * F / F \to T \to T*F$; $T \to F$

$F \to (E) / id \to F \to (E)$; $F \to id$.

Derivation

Derivation is the process where a non-terminal is replaced by a non-terminal or terminal in each step by one of its production beginning with the start symbol inorder to obtain the string of terminals.

Eg: $E \to E+E / E*E / (E) / -E / id$.

$-(id+id)$

$E \to -E$

$E \to -(E+E)$

$E \to -(id+id)$

→ Two types of derivation ⇒ 1) LMD 2) RMD

1) Left most derivation

In LMD the left most non-terminal is replaced by a terminal or non terminal beginning with a start symbol

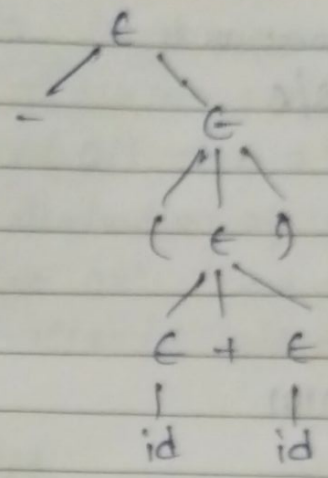
2) Right most derivation

In RMD the right most non terminal is replaced by a terminal / non terminal beginning with a start symbol

~~start symbol~~

LMD	RMD
$E \to E$	$E \to -E$
$E \to (-E)$	$E \to -(E)$
$E \to (E+E)$	$E \to -(E+E)$
$E \to -(id+E)$	$E \to -(E+id)$
$E \to E-(id+id)$	$E \to -(id+id)$

GOLKONDA
PAGE NO.
Date : / /



Ambigous Grammar

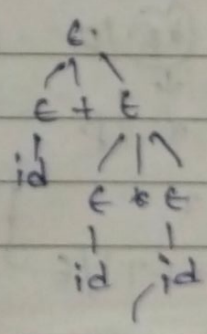
A grammer that produces more than one parse string for some sentence is said to be ambigous

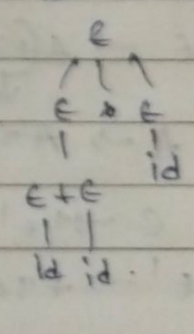
Another way, an ambigous grammar is one that produces more than one leftmost or more than one right most is said to be ambigous grammer

eg: Derive the strid (id+id)* id.

$E \to E+E \mid E*E \mid (E) \mid -E \mid id$.

LMD	RMD
$E \to E+E$	$E \to E*E$
$E \to id+E$	$E \to E+E*E$
$E \to id+E*E$	$E \to id+E*E$
$E \to id+id*E$	$E \to id+id*E$
$E \to id+id*id$	$E \to id+id*id$





Q) Show that the grammar is
$S \to aSbS \mid bSaS \mid \epsilon$

Q) $S \to 0A \mid 1B \mid 0 \mid 1$
$A \to 0S \mid 1B \mid 1$
$B \to 0A \mid 1S \mid 0$
(a) 0101 (b) 1100101

20/01/25

Q) Elimination of left recursion
→ A grammar is left recursive if it has a non-terminal A i.e $A \to A\alpha \mid \beta$ for some string α.
~~for some string~~
→ Top-down parsing methods can't handle left recursive grammar, so a transformation that eliminates left recursion is needed.
→ When there was a production of the form ~~A~~ $A \to A\alpha \mid \beta$ then the grammar could be replaced by non-recursive production.
$A \to \beta A' \; ; \; A' \to \alpha A' \mid \epsilon$

Eg:
Consider the following grammar for arithmetic expression
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

$E \to E+T \mid T$ $\quad$ $T \to T*F \mid F$ $\quad$ $F \to (E) \mid id$
$A \to A\alpha \mid \beta$ $\quad$ $A \to A\alpha \mid \beta$ $\quad$ not in left recursive so remains same

$E \to TE'$ $\quad$ $T \to FT'$
$E' \to +TE' \mid \epsilon$ $\quad$ $T' \to *FT' \mid \epsilon$ $\quad$ } After elimination of left recursion

Q) $S \to AA/0 \quad A \to SS/1$

$S \to SSA/0 \Rightarrow S \to S\alpha/\beta$

$A \to AAS/1 \Rightarrow A \to A\alpha/\beta$

After elimination of left recursion

$S \to 0S' \quad A \to 1A'$

$S \to S1S'/\epsilon \quad A \to A0A'/\epsilon$

Left factoring

→ Left factoring is a grammar transformation that is useful for producing a grammar suitable for parsing techniques.

→ In general, the given grammar is in this form i.e $A \to \alpha\beta_1/\alpha\beta_2$ are 2 productions and the i/p begins with a non empty string derived from α. The left-factor productions can be.

$A \to \alpha A' \quad A' \to \beta_1 / \beta_2$

Ex:- eliminate left factoring for given grammar.

$S \to \underbrace{ie\ tS}_{\alpha\ \beta_1} / \underbrace{i\ E\ tS\ eS}_{\alpha\ \beta_2} / a \quad E \to b.$

$S \to iEtS'/a$

$S' \to \epsilon/eS$

$S \to a \quad E \to b$

} the given grammar is in left factoring.

Note:

If $A \to A\alpha_1/A\alpha_2/\ldots/A\alpha_m/\beta_1/\beta_2\ldots/\beta_n$

$A \to \beta_1 A'/\beta_2 A'/\ldots/\beta_n A'$

$A' \to \alpha_1 A'/\alpha_2 A'\ldots/\alpha_n A'/\epsilon$

Q Eliminate the left recursion for the following grammer

1) $S \to Aa / b$ $\quad A \to Ac / Sd / b$.

$S \to \underbrace{Sda}_{\alpha} / \underbrace{ba}_{\beta_1} / \underbrace{b}_{\beta_2}$

$S \to baS' / bS'$

$S' \to daS' / \epsilon$

Parsing Techniques

- Top-down parsing (TDP)
  - Backtracking
  - Recursive descent parser (RDP)
  - Predictive parser (PP)
  - Non rec predictive parser (NRPP)
- Bottom-up parsing (BUP)
  - LR
    - SLR (simple)
    - CLR (canonical)
    - LALR (look ahead)
  - shift reduce parser.

Algorithm Eliminating Left rec

Input:

Grammer G with no cycles (or) ϵ productions

O/P:

A equivalent grammer with no left recursion

method:

Apply algo to grammer G then the resulting non-left recursive grammer may have ϵ productions.

1) ~~Use~~ Arrange the non-terminals in order.

$A_1, A_2, \ldots A_m$

2) for (each i from 1 to n) {

3) for (each j from 1 to i-1) {

4) replace each production of the form $A_i \to A_j \gamma$ by the production

$A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \delta_k\gamma$ where

$A_j \to \delta_1 \mid \delta_2 \mid \delta_3 \ldots \mid \delta_k$ are all current A_j production.

}

5) Eliminate the immediate LR among the A_i production ~~close~~

}

Q1) $S \to a \mid \wedge \mid (T)$ $T \to \underset{A}{T}\underset{\alpha}{,S} \mid \underset{\beta}{S}$

$T \to ST'$ $T' \to ,ST' / \epsilon$

Q2) $E \to E(T) / T$; $T \to T(F) / F$; $F \to id$

$\to T \to \underset{A}{T}\underset{\alpha}{(id)} / \underset{\beta}{id}$

$T \to (id)T'$

$T' \to idT' / \epsilon$

$\to E \to \underset{A}{E}\underset{\alpha}{(T(id))} / \underset{\beta}{id}$

$E \to idE'$

$E' \to T(id)E' / \epsilon$

(After substitution also no left rec)

Q3) $S \to ABC$; $A \to \underset{A\alpha_1}{Aa} / \underset{A\alpha_2}{Ad} / \underset{\beta_1}{b}$; $B \to \underset{A\alpha_1}{Bb} / \underset{\beta_1}{\epsilon}$; $C \to \underset{A\alpha_1}{Cc} / \underset{\beta_1}{g}$

~~S~~ $A \to bA'$; $A' \to aA' / dA' / \epsilon$

$B \to bB'$; $B' \to bB' / \epsilon$; $C \to gC'$; $C' \to cC' / \epsilon$

Q4) $A \to A\underset{\alpha_1}{Bd} / \underset{A\alpha_2}{Aa} / \underset{\beta}{a}$ $\qquad$ $B \to \underset{A\alpha_1}{Bc} / \underset{\beta_1}{b}$

$A \to \underset{\alpha_1}{Abd} / \underset{\alpha_2}{Aa} / \underset{\beta_1}{a}$ $\qquad$ $B \to bB' \; ; \; B' \to cB'$

$A \to aA'$

$A' \to bdA' / aA'$

Q5) $S \to \underset{\alpha}{aS}\underset{\beta_1}{bS} / \underset{\alpha}{aS}\underset{\beta_2}{aSb} / abb / b$

$S \to aS'S' / abb / b$

$S' \to SbS / aSb$

Top-down parsing

→ The process of constructing the parse tree which starts from the root and goes down to the leaf is top down parsing

→ Top down parsers constructs from the grammar which is free from ambiguity & left recursion.

→ Top down parsers uses left most derivation (LMD) to construct a parse tree.

Classification of Top

with Backtracking
① Brute force technique

without Backtracking
① RDP
② PP
③ Non-recursive

4/9/23

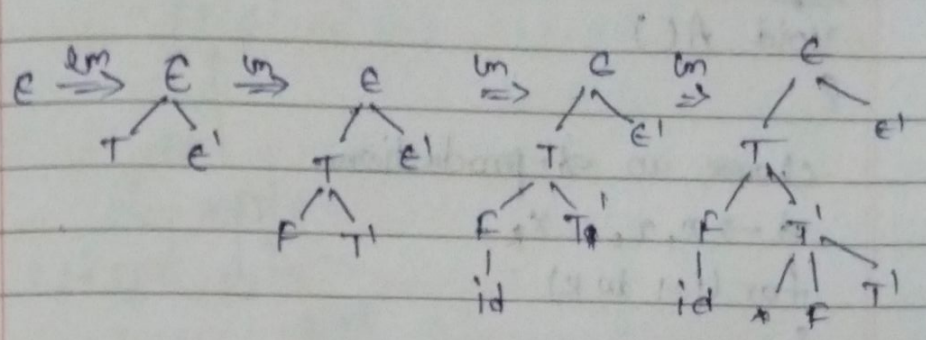
~~Ex. Grammar~~

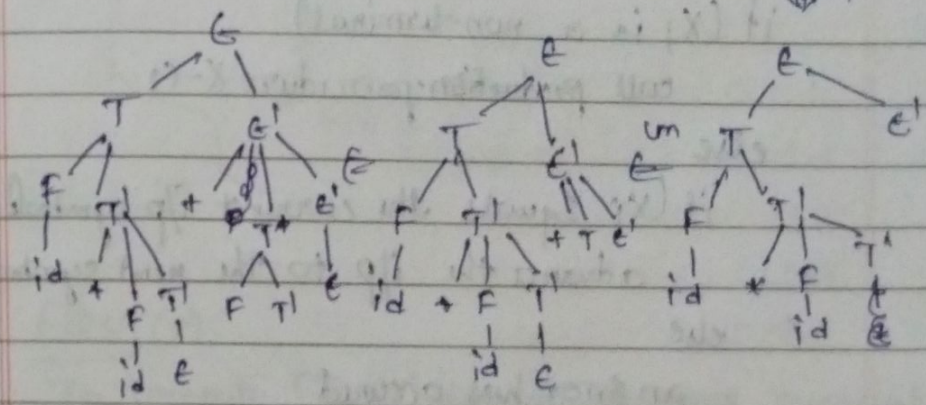
Grammar

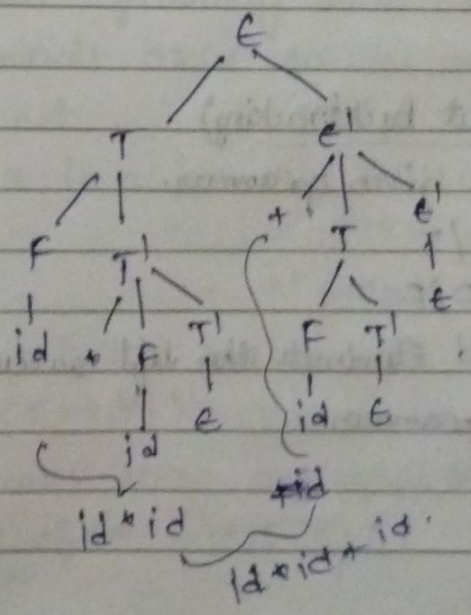
$E \to E+T/T \to E \to TE'$; $E' \to +TE'/\epsilon$

$T \to T*F/F \to T \to FT'$; $T' \to *FT'/\epsilon$

$F \to (E)/id$







Brute force technique (RDP with backtracking)
RDP involves backtracking i.e If a choice of a production rule doesnot work we backtrack to try other alternatives. In general RDP is not ~~so~~ efficient (slower)

Algo:

```
void A()
{
  choose an A-production,
  A -> X1, X2, ... Xk;
  for (i=1 to k)
  {
    if (Xi is a non-terminal)
        call ~~production~~ procedure X-()
    else
        if (Xi Equals the current i/p symbol a)
            advance the i/p to the next symbol
        else
            an Error has occured
  }
}
```

RDB (without Backtracking)

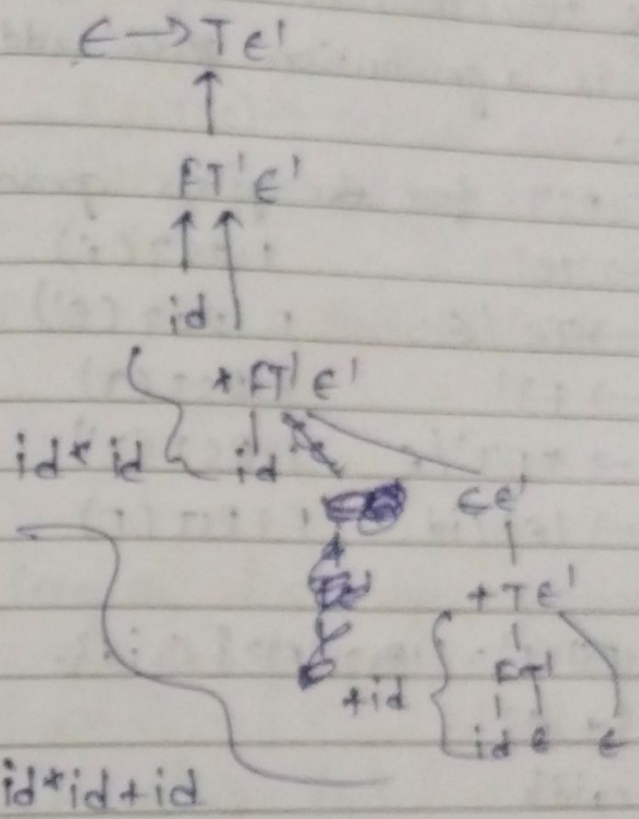
Consider the given grammar.

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$ Eliminate the left recursion from the given grammar

$E \to TE'$; $E' \to +TE' / \epsilon$; $T \to FT'$
$T' \to *FT' / \epsilon$; $F \to (E) / id$
id*id+id



FIRST(x)

To compute FIRST(x) for all grammar symbols x, apply the following rules until no more terminals or ε can be added to any first set

(i) if x is a terminal FIRST(x) = {x}
FIRST(a) = {a}
FIRST(+) = {+}

(ii) if x isn't a terminal i.e.
$Y \to Y_1 Y_2 \ldots Y_k$ is a prodn for some
K greater than or equal to 1, then place a in
FIRST(x)

→ if for some i, a is in FIRST(Y_i) & ε is in FIRST(Y_1), FIRST(Y_2), ... FIRST(Y_{i-1})

→ if ε is in FIRST(Y_j) for all $j = 1, 2, 3, \ldots k$ then add ε to FIRST(X).

→ if $X \to \epsilon$ is a production then add ε to first (X).

Ex: Compute FIRST for the given grammar.

$E \to TE'$	FIRST(E)
$E' \to +TE' / \epsilon$	FIRST(E')
$T' \to FT'$	FIRST(T)
$T' \to *FT' / \epsilon$	FIRST(T')
$F \to (E) / id$	FIRST(F)

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}.

FIRST(E') = {+, ε}

FIRST(T) = {(, id}

FIRST(T') = {*, ε}

FIRST(F) = {(, id}

Q) $S \to ABCDE$

$A \to a / \epsilon$ FIRST(A) = {a, ε}

$B \to b / \epsilon$ " (B) = {b, ε}

" (C) = {c}

b → terminal
B → non-terminal

Q) Compute first for the following grammar
S → ACB / CbB / Ba
A → da / BC
B → g / ε ⇒ FIRST(B) = {g, ε}
C → h / ε ⇒ FIRST(C) = {h, ε}
FIRST(S) → FIRST(A), FIRST(C), FIRST(B)
FIRST(A) → {d, g, h, ε}
→ S → ACB S → CbB S → Ba
F(S) = F(A) F(C) = F(C) = {h, ε, b} F(S) = F(B) = {g, ε, a}
{d, g, h, ε}

Q) Compute first for following
S → i c t S S' / a ; S' → e S / ε ; c → b

FIRST(c) = {b}
FIRST(S') = {b, ε}
FIRST(S) = {i, a}

FOLLOW(A):
To compute FOLLOW(A) for all non-terminals 'A' apply the following rules until nothing can be added to any follow set
1) Place $ in ~~follow~~ FOLLOW(S) where S is the start symbol and $ is the right end marker.
FOLLOW(S) = {$}
2) If there is a production ~~a gives~~ A → αBβ then in FIRST(β) except ε is in FIRST(β) − ε

2) if there is a production $A \to \alpha B \beta$ (or) $A \to \alpha B \beta$, where FIRST (β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B)

FOLLOW(B) = FIRST (β) - ϵ union FOLLOW(A)

Q) Compute follow for

$E \to TE'$; $E' \to +TE'/\epsilon$; $T \to FT'$; $T' \to *FT'/\epsilon$

$F \to (E)/id$

~~FIRST~~ FIRST (E) = {(, id}

FIRST (E') = {+, ϵ}

FIRST (T) = {(, id}.

(T') = {*, ϵ}

(F) = {(, id}

→ FOLLOW(E) = {$,)} ($\because F \to (E)$ / id.)

$B = \epsilon$ $\beta =)$ E is followed by ')'

FOLLOW(E) = FIRST (β) - ϵ = FIRST ()) - ϵ =)

→ FOLLOW(E')

$E \to TE'$ & $E' \to +TE'$

FOLLOW(E') = {FIRST (E') - ϵ} ∪ FOLLOW(E)

= {ϵ - ϵ} ∪ {$,)}

= {$,)}

FOLLOW(E') = {ϵ - ϵ} ∪ {$,)}

= {$,)}

octo/23

$FOLLOW(T) = \{\{+, \epsilon\} - \epsilon\} \cup \{\$,)\}$ $\quad E \to TE'$

$= \{+, \$,)\}$

$E' = +TE'$

$\Rightarrow \{+, \epsilon - \epsilon\} \cup \{\$,)\}$

$= \{+, \$,)\}$

$T \to FT'E$

$FOLLOW(T') =$

$\{FIRST(\epsilon) - \epsilon\} \cup FOLLOW(T)$

$= \{\epsilon - \epsilon\} \cup \{+, \$,)\}$

$= \{+, \$,)\}.$

$T' \to *FT'/\epsilon$

$FOLLOW(T') = \{FIRST(\epsilon) - \epsilon\} \cup FOLLOW(T')$

$= \{\$,), +\}$

$FOLLOW(F)$

$T \to FT$

$FOLLOW(F) = \{FIRST(T') - \epsilon\} \cup FOLLOW(T)$

$= \{\{+, \epsilon\} - \epsilon\} \cup \{\$,), +\}$

$T' \to *FT'/\epsilon$

$FOLLOW(T') = \{FIRST(\epsilon) - \epsilon\} \cup FOLLOW(T')$

$= \{\$,), +\}$

$FOLLOW(F) = T \to FT$

$= \{FIRST(T') - \epsilon\} \cup FOLLOW(T')$

$= \{\$,), +, *\}$

$FOLLOW(F) = \{\$,), +, *\}$

11/10/23

Predictive parsing

Construction of a predictive parsing table algorithm

→ Input grammar G, output parsing table M

Method

For each product^n $A \to \alpha$ of the grammar do the following.

S-1:

for each terminal 'a' in FIRST(A) add $A \to \alpha$ to M[A, a]

S-2:

If ϵ is in FIRST(α) then for each terminal 'b' in FOLLOW(A) add $A \to \alpha$ to M[A, b]

S-3:

If ϵ is in FIRST(α) & \$ is in FOLLOW(A) then add $A \to \alpha$ to M[A, \$]

After performing the three steps there is no production at all in M[A, a] then set M[A, a] to error.

Q) Construct a predictive parsing table for the grammar

$S \to aBa$; $B \to bB/\epsilon$

FIRST(S) = {a} FIRST(B) = {b, ϵ}

FOLLOW(S) = {\$} ~~FOLLOW(B) = {a}~~ $S \to aBa$

FOLLOW(B) = {a, \$} (since $\beta \neq \epsilon$) $B \to bB/\epsilon$

	First(α)	PPE
$S \to aBa$	FIRST(aBa) = {a}	M[S, a]
$B \to bB$	FIRST(bB) = {b}	M[B, b]
rule 2 ⇒ $B \to \epsilon$	FOLLOW(B) = {a}	M[B, a]

I/P symbols

Non-terminals	a	b	$
S	$S \to aBa$		
B	$B \to \epsilon$	$B \to bB$	

Q) PPT for grammar

$E \to TE'$; $T \to FT'$;

$E' \to +TE'/\epsilon$; $T' \to *FT'/\epsilon$

$F \to (E)/id$.

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

FOLLOW(E) = {$,)}

{FIRST( ) − ε} =) − ε ⇒

$F \to (E)/id$

FOLLOW(E') = ~~{$,)}~~

{FIRST(E') − ε} ∪ FOLLOW(E)

= {$,)}

$E \to TE'$

FOLLOW(E') = {ε − ε} ∪ {$,)}

= {$,)}

$E' \to +TE'/\epsilon$

FOLLOW(T) = {E' − ε} ~~$ etc~~

= {{+, ε} − {ε}} ∪ {$,)}

= {$, +,)}

$E \to TE'$

$E' \to +TE'/\epsilon$

{{+, ε} − ε} ∪ {$,)} = {$, +,)}

GOLKONDA
PAGE NO.
Date: / /

$T \to FT'$ ($A \to \alpha\beta$, $\alpha = F$, $\beta = T'$)

$T' \to *FT' \mid \beta$ ($\alpha = *FT'$)

$FOLLOW(T) = \{\epsilon - \epsilon\} \cup \{\epsilon, id\} \{+, \$,)\}$

$= \{id\} \{+, \$,)\}$

$= \{\epsilon - \epsilon\} \cup \{\epsilon, id\} \{+, \$,)\}$

$= \{id\} \{+, \$,)\}$

$T \to FT'$ ($A \to \alpha\beta$, $\alpha = F$, $\beta = T'$)

$T' \to *FT'$ ($A \to \alpha\beta$, $\alpha = *$, $\beta = FT'$)

$FOLLOW(F) = \{\{*, \epsilon\} - \{\epsilon\}\} \cup \{\$, +,)\}$

$= \{*, \$, +,)\}$

$= \{\{*, \epsilon\} - \{\epsilon\}\} \cup \{*, \$, +,)\}$

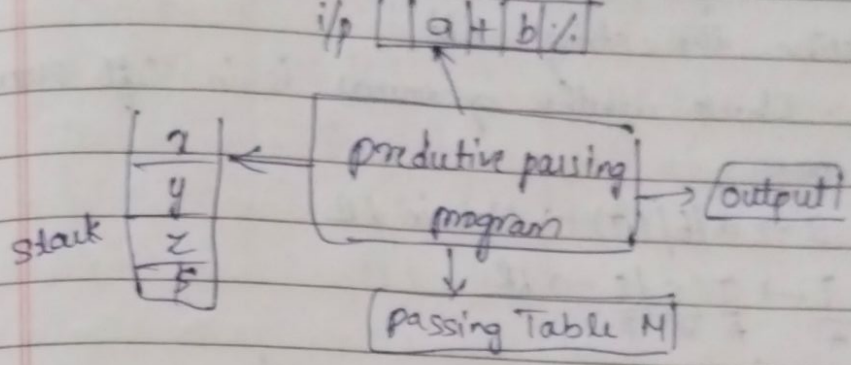
$\{*, \$, +,)\}$

$A \to \alpha$	first(α)	PPϵ
$E \to TE'$	{c, id}	M[E, c], M[E, id]
$E' \to +TE'$	+	M[E', +]
$E' \to \epsilon$	{$,)}	M[E', c], M[E, id]
$T \to FT'$	{c, id}	M[T, c], M[T, id]
$T' \to *FT'$	*	M[T', *]
$T' \to \epsilon$	{+, $,)}	M[T', +], M[T', $], M[T',)]
$F \to (E)$	(	M[F, (]
$F \to id$	id	M[F, id]

i/p symbols

	+	*	(	)	$	id
E			$E \to TE'$			$E \to TE'$
E'	$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$	
T			$T \to FT'$			$T \to FT'$
T'	$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$	
F			$F \to (E)$			$F \to id$.

Model of a non-recursive predictive parsing



Q) Construct non-predictive parser for the following grammar.

$E \to TE'$; $E' \to +TE'/\epsilon$; $T \to FT'$; $T' \to *FT'/\epsilon$

$F \to (E)/id$.

i/p string id+id*id.

stack	i/p	o/p
$ E	id+id*id $	$E \to TE'$
$ E'T	id+id*id$	$T \to FT'$
$ E'T'F	id+id*id$	$F \to id$
$ E'T'~~id~~	~~id~~+id*id$	$T' \to \epsilon$
$ E'	+id*id$	$E' \to +TE'$
$ E'T~~+~~	~~+~~id*id$	$T \to FT'$
$ E'T'F	id*id$	$F \to id$
$ E'T'~~id~~	~~id~~*id$	$T' \to *FT'$
$ E'T'F~~*~~	~~*~~id$	$F \to id$
$ E'T'~~id~~	~~id~~$	$T' \to \epsilon$
$ E'	$	$E' \to \epsilon$
$	$	success

E'
T
$

→ for $ T E' on id = $

so we take E'T

Left recursion :- $A \to A\alpha / \beta$

$A \to \beta A'$

$A' \to \alpha A' / \epsilon$

GOLKONDA
PAGE NO.
Date : / /

Q) Construct NRPP on given grammar :

$S \to a/b/(T)$; $T \to T, S / \epsilon$. and also describe the string (b)

S1:- Check whether grammar is in left recursion or not

$S \to a/b/(T) \to$ Not in LR

$T \to T, S / S \to$ LR

$\hookrightarrow T \to ST'$ & $T' \to ,ST' / \epsilon$.

S2: Compute First & Follow

$S \to a/b/(T)$; $T \to ST'$; $T' \to ,ST' / \epsilon$

FIRST(S) = {a, b, (}

FIRST(T) = {a, b, c}

FIRST(T') = {, , ε}

FOLLOW(S) = ~~{$}~~

$T \to \epsilon ST'$ ($A \to \alpha B \beta$) ⇒ ~~{{a, b, (} − ε}~~ = ~~a~~. {{, $ε} − ε} ∪ Follow(T)

~~{$, a, b, ε}~~ {$, ,} ∪ Follow(T) = {$, , ,)}

~~Follow(T')~~ , $T' \to ,ST' / \epsilon$ ($\alpha B \beta$)

= {{, , ε} − ε} ∪ Follow(T)

= {$, , } ∪ Follow(T) = {$, , ,)}

FOLLOW(T) = $S \to a/b/(T)$ $S \to (T)$

= {), $, ~~$~~}

FOLLOW(T') = ~~{$}~~ $T \to ST'$ ($A \to \alpha B \beta$, ε)

{ε − ε} ∪ Follow(T)

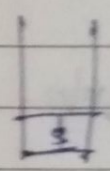
= {)}

	FIRST (α)	PPE
$S \to a$	{a}	M[S, a]
$S \to b$	{b}	M[S, b]
$S \to (T)$	{(}	M[S, (]
$T \to ST'$	{a, b, (}	M[T, a], M[T, b], M[T, (]
$T' \to ,ST'$	{,}	M[T', ,]
$T' \to \epsilon$	{)}	M[T',)]

i/p symbols

	a	b	(	)	,
S	$S \to a$	$S \to b$	$S \to (T)$		
T	$T \to ST'$	$T \to ST'$	$T \to ST'$		
T'				$T' \to \epsilon$	$T' \to ,ST'$

$S \to a/b/(T)$; $T \to ST'$; $T' \to ,ST'/\epsilon$.



stack	i/p	o/p
$ S.	(b)$	$S \to (T)
$)T(	(b)$	$T \to ST'
$)T'S	b)$	$S \to b.
$)T'b	b)$	$T' \to \epsilon
$ Y	Y$	success.

LL(1) Grammar :-

Predictive parser i.e recursive decent parser needs no backtracking, can be constructed for a class of grammars called LL(1). The first "L" indicates scanning the i/p from left for right & second L for producing a left most derivation and 1 for using one i/p symbol of look ahead at each step at make parsing action decisions.

d) Show the above question grammar is LL(1) grammar.
(Multiple entries not LL(1) grammar)
(unique entries not LL(1) grammar)

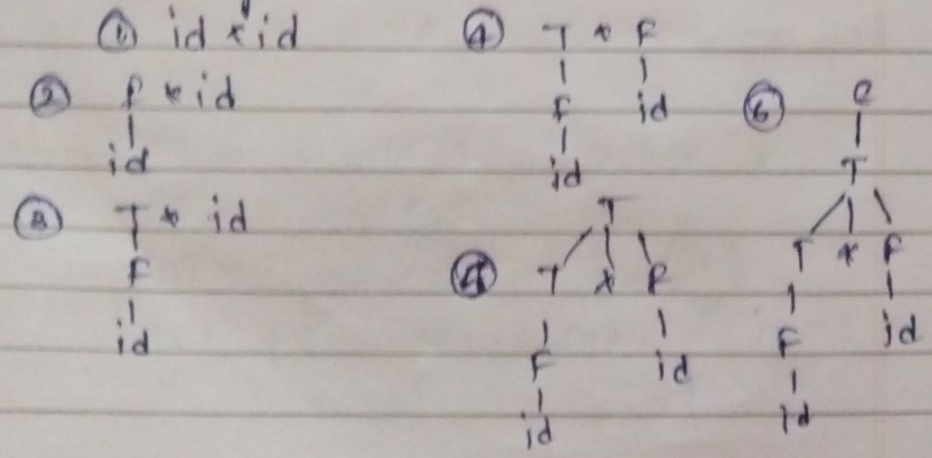
Bottom-up parser.

Constructing a parse tree from leaf node to the root node is called Bottom up parsing

Eg:-

$E \to E+T/T$; $T \to T*F/F$; $F \to (E)/id$

string => "id*id"



Handle pruning

Bottom up parsing during a left to right scan of the input constructs a right most derivation in reverse.

A handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of right most derivation.

Right sentential form	Handle	Reduced production
id * id	id	F → id
F * id	F	T → F
T * id	id	F → id
T * F	T * F	T → T * F
T	T	E → T
E		

RMD in reverse

Bottom up parsing

Shift Reduce Parser

- SLR — simple
- CLR — canonical
- LALR — look ahead

Shift reduce parser

The following actions can be performed in shift reduce parser.

(i) Shift: The next i/p symbol is shifted on the top of the stack

(ii) Reduce: The parser replaces the handle within a stack with a non terminal

GOLKONDA
PAGE NO.
Date : / /

iii) Accept: The parser announces successful completion of parsing

iv) Error: The parser discovers an error and calls an error recovery routine.

~~Gg~~ ~~stack~~ ~~Up~~

Construct shift reduce parser

$E \to E+T / T \quad T \to T*F / T \quad F \to (E) / id$

stack	input	action
$	id*id$	shift
$id	*id$	reduce (F→id)
$F	*id$	reduce (T→F)
$T	*id$	shift
$T*	id$	shift
$T*id	$	reduce (F→id)
$T*F	$	reduce (T→T*F)
$T	$	reduce (E→T)
E	$	accept

9) $M \to R+R / R+C / R \quad ; R \to C$

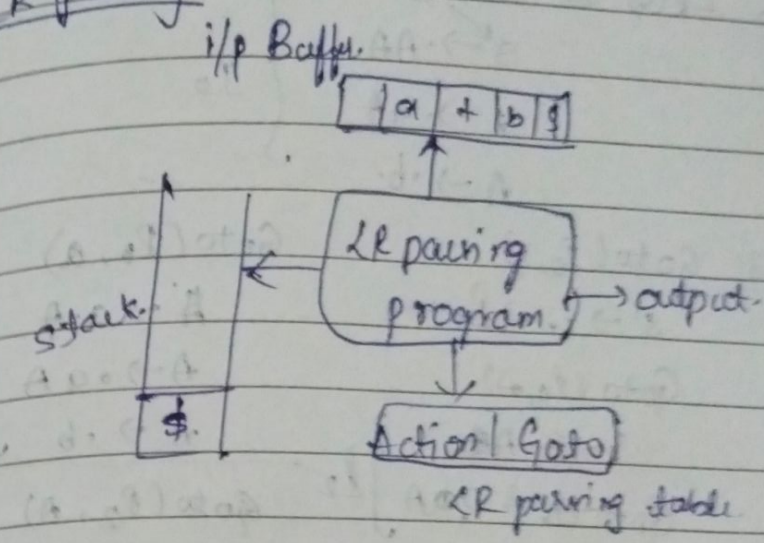
String "C+C"

stack	i/p	action
$	C+C$	shift
$C	+C$	reduce (R→C)
$R	+C$	shift
$R+	C$	shift
$R+C	$	reduce (M→R+C)
$M	$	accept

Conflicts during SRP

(i) shift reduce conflict (ii) reduce-reduce conflict

LR parsing



SLR (Simple LR parser)

algorithm for SLR parsing

S1:- Augmented grammar (G')

If G is a grammar with start symbol S then the augmented grammar G' with new start symbol S'

$S' \to S$

S2:- LR (0) items

An LR(0) item with a dot at some position on the right hand side

$A \to XYZ$

$A \to .XYZ$; $A \to X.YZ$; $A \to XY.Z$; $A \to XYZ.$

S3:- closure set of items (I)

$A \to \alpha . B\beta$

$\beta \to .\gamma$

Q) Construct SLR parsing for following grammer.

$S \to aA$; $a \to aA$; $A \to b$

S1:- Augmented Grammer G' $S' \to S$

S2: $LR(0) \Rightarrow$ $S' \to .S$

$S \to .AA$

$A \to .aA$

$A \to .b$ } I_0

S3: Goto(I_0, S)

$S' \to S.$ } I_1

Goto(I_0, A)

$S \to A.A$

~~Goto (I_0~~ $A \to .aA$

$A \to .b$ } I_2

Goto(I_0, b)

$A \to b.$ } I_4

Goto(I_0, a)

$A \to a.A$

$A \to .aA$

$A \to .b$ } I_3

Goto(I_2, A)

$S \to AA.$ } I_5

Goto(I_2, a)

$A \to aA.$

$A \to .aA$

$A \to .b$ } I_3.

I_2 Goto(I_2, b)

$A \to b.$ } I_4

I_3 Goto (I_3, A)

$A \to aA.$ } I_6

Goto(I_3, a)

$A \to a.A$

$A \to .b$

$A \to .aA$ } I_3

Goto(I_3, b)

$A \to b.$ } I_4

Q)

State	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5			r_1		
I_6	r_2	r_2	r_2		

To perform reduce operation the given production should be in $A \to \alpha$

(1) $S \to AA$

(2) $A \to aA$

(3) $A \to b$

Inorder to perform reduce oper compute FIRST and FOLLOW for the given grammar

FIRST(S) = FIRST(A) = {a, b}

FOLLOW(S) = {$}

FOLLOW(A) = {$, a, b} = FOLLOW(S)

Derive abb

S-S

Stack	input	remarks
0	abb$	S_3
0a3	bb$	S_4
0a3b4	b$	r_3 $A \to b$ 2×1=02
~~0a3A5~~	b$	r_2 $A \to aA$ 2×2=4
0A2	b$	S_4
0A2b4	$	r_3 ($A \to b$) 2×2=2
0A2A5	$	r_1 $S \to AA$
0S1	$	accept

17/10/2023

Q) Construct SLR, CLR & LALR for grammar.

$S \to CC$, $C \to cC$ $C \to d$.

$$\left.\begin{array}{l} S' \to .S \\ S \to .CC \\ C \to .cC \\ C \to .d \end{array}\right\} I_0$$

→ goto (I_0, S)

$S' \to S. \to I_1$

stop since no terminal or non terminal after dot.

goto (I_0, C)

$$\left.\begin{array}{l} S \to C.C \\ C \to .cC \\ C \to .d \end{array}\right\} I_2$$

goto (I_0, c)

$$\left.\begin{array}{l} C \to c.C \\ C \to .cC \\ C \to .d \end{array}\right\} I_3$$

goto (I_0, d)

$C \to d. \} I_4$

→ goto (I_2, C)

$S \to CC. \} I_5$

goto (I_2, c)

$$\left.\begin{array}{l} C \to c.C \\ C \to .cC \\ C \to .d \end{array}\right\} I_3$$

goto (I_2, d)

$C \to d. \} I_4$

→ goto (I_3, C)

$C \to cC. \} I_6$

goto (I_3, c)

$$\left.\begin{array}{l} C \to c.C \\ C \to .cC \\ C \to .d \end{array}\right\} I_3$$

goto (I_3, d)

$C \to d. \} I_4$

→ ~~goto (I_4, d)~~

	c	d	$	S	C
0	S_2	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		
6	r_2	r_2	r_2		

① $S \to CC$ (c<c

② $C \to cC$

③ $C \to d$

Follow(S) = {$}; Follow(C) = {$, c, d}

$A \to \alpha.$

~~⑤ $C \to cC.$ R_4~~ ③ $C \to d.$ R_4

~~① $S \to CC.$ R_5~~ ① $S \to CC.$ R_5

~~② $S \to d.$ R_6~~ ② $C \to cC.$ R_6

i/p

Stack	i/p	Remarks
0	cdd$ (given)	S3
0c3	dd$	S4
0c3d4	d$	r_3 ($C \to d.$) → 2*1=2
0c3C6	d$	r_2 ($C \to cC.$) → 2*2=4
0C2	d$	S4
0C2d4	$	r_3 ($C \to d.$) ⇒ 2
0C2C5	$	r_1 ($S \to CC$) 2*2 → 4

Canonical LR parser

$S \to CC$; $C \to cC$; $C \to d$

$S' \to .S, \$$ → look ahead symbol.

$S \to .CC, \$$ → ($\alpha\beta\beta$ a; S, $; first(C$) = first($) = $)

$C \to .cC, c/d$ } I_0

$C \to .d, c/d$

to get look ahead symbol it should be in form

$A \to \alpha.B\beta, a$

$B \to .\gamma$, first(βa)

→ goto(I_0, S) goto(I_0, C) goto(I_0, c)

$S' \to S., \$$ I_1 $S \to C.C, \$$ $C \to c.C, c/d$

$C \to .cC, \$$ } I_2 $C \to .cC, c/d$ } I_3

$C \to .d, c/d, \$$ $C \to .d, c/d$

goto (I_0, d)
$C \to d\cdot, c/d \cdot I_4$

→ goto (I_2, C)
$S \to CC\cdot, \$ \} I_5$

goto (I_2, c)
$C \to c\cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$
$\} I_6.$

goto (I_2, d)
$C \to d\cdot, \$ \} I_7$

⇒ goto (I_3, C)
$C \to cC\cdot, c/d \} I_8$

goto (I_3, c)
$C \to c\cdot C, c/d$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$
$\} I_3$

goto (I_3, d)
$C \to d\cdot, c/d \} I_4$

→ goto (I_6, C)
$C \to cC\cdot, \$ \} I_9$

goto (I_6, c)
$C \to c\cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$
$\} I_6.$

goto (I_6, d)
$C \to d\cdot, \$ \} I_7$

CLR

⇒

	c	d	$	S	C
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		

(2) $C \to d\cdot, c/d \} I_4$
(1) $S \to cC\cdot, \$ \} I_5$
(3) $C \to d\cdot, \$ \} I_7$
(4) $C \to cC\cdot, c/d \} I_8$
(5) $C \to cC\cdot, \$ \} I_9$

Ans

Stack	I/p Buffer	Remarks
0	dd$	S_4
0~~d4~~	d$	r_3 (~~S~~ $C \to d$)
0C2	d$	S_7
0C2~~d7~~	$	r_3 ($C \to d$)
0C~~2C5~~	$	r_1 ($S \to CC$)
0S1	$	accept.

Ans

	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			Accept.		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89.
47	r_{36}	r_{36}	r_{36}		
5			r_1		
89.	r_2.	r_2	r_2.		

LALR (Look ahead LR parser)

stack	I/p buffer	Remarks
0	dd $	S47
0d34 7 (reducing state)	d $	r_{36} (C→d)
0dC8	d $	S47
0C2d47	$	r_{36} (C→d)
0C7C5	$	r_1 (S→CC)
0S1	$	accept

Unit - III

SDD (Syntax Directed Defination)

→ SDD is a context free grammar along with attributes & rules

→ Attributes are associated with grammar symbols and rules are associated with production.

→ If X is a symbol and a is one of its attributes then we write X.a

→ Two kinds of attributes for non-terminals.

(i) Synthesized attribute

(ii) Inherited Attribute

Synthesized Attribute

A SA at node N is defined only in terms of attribute values at the children of N and at N itself

Inherited attribute :-

An IA at node N is defined in terms of attribute values at parent, siblings and itself

Annotated parse tree

A parse string showing the values at the attributes is called APT

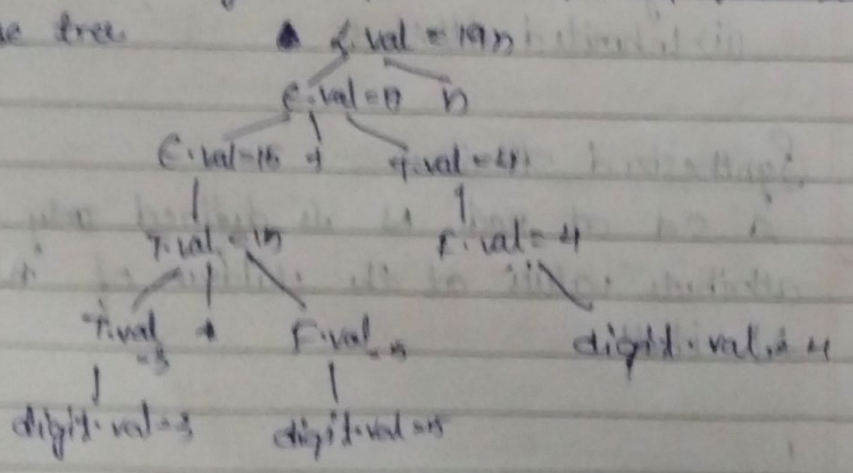
* Q) Construct an APT for the calculator for string [3*5+4n=19n] → end of the string

Eg:

production	Semantic rules.
$E \to E_1 n$	$E.val = E_1.val$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)/digit$	$F.val = (E.val)/digit.val$

As we don't have n in the prod^n we create a new prod^n & store it in a variable.

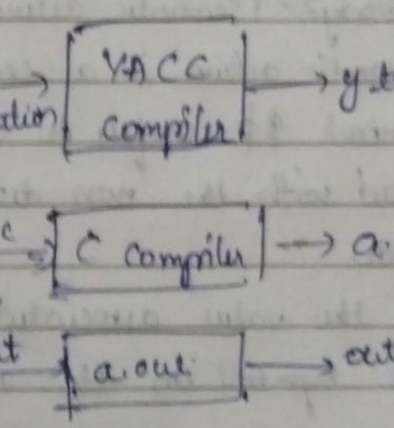
parse tree



Difference between S-attributed & L-attributed

S-attributed	L-attributed
1) Uses only synthesized attribute	1) Uses both synthesized & inherited attributed.
2) Semantic actions are placed at right end of a product. Eg $A \to BC\{\}$	2) Semantic actions are placed at anywhere. Eg: $A \to \{\}BC$, $A \to B\{\}C$, $A \to B\{\}C$.
3) Attributes are evaluated during bottom up parsing	3) Attributes are evaluated by traversing parse tree in depth first search.
4) S-attributed grammar is based on LR grammar	4) It is based on LL grammar
5) These are implemented using LALR	5) implemented using predictive parser.

Topics: YACC (Yet another compiler compiler)



YACC consists of 3 parts.
(i) declaration part
(ii) translation rules
(iii) auxiliary procedure.

Declaration part
There are two optional sections in a declaration part of YACC program
(i) ordinary C declaration
(ii) declaration of a grammar tokens
→ %{#incude <stdio.h> %}
→ %token DIGIT (or) %token identifier.

Translation rules:
In the part of YACC specification a set of production are.
<head> → <body1> | <body2> | ... | <body n>
could be written in YACC as
<head> : <body1> {semantic action 1}
<head> : <body2> {semantic action 2}
→ A YACC semantic action is a sequence of C stmt. the symbol $$ refers to the attribute value associated with the non terminal on the left hand side.
→ $i refers to the value associated with the ith grammar symbol on the right hand side

Eg Q) E → E+T / T
T → T*F / F
F → (E) / id

```
%%
E : E+T   {$$ = $1 + $3}
  | T
T : T*F   {$$ = $1 * $3}
  | F
F : (E)   {$$ = ($2)}
  | digit
%%
```

Auxiliary procedure:-
The 3rd section of YACC specification consists of C routines.

```
%{ #include <ctype.h>
%}
%token DIGIT
%%
expr. expr + term {$$ = $1 + $3}
    | term
term  term * factor {$$ = $1 * $3}
    | factor
factor (expr) {$$ = ($2)}
    | digit
%%

yylex()
{
  int c;
  c = ~~return~~ getchar()
  if (isdigit(c))
  {
    yylval = c - '0',
```

→ ASCII val

```
    return DIGIT;
  }
  return c;
}
```

~~yacc filename~~

yacc compiler

```
yacc filename.y
```

compile through c compiler

```
cc y.tab.b
./a.out
```