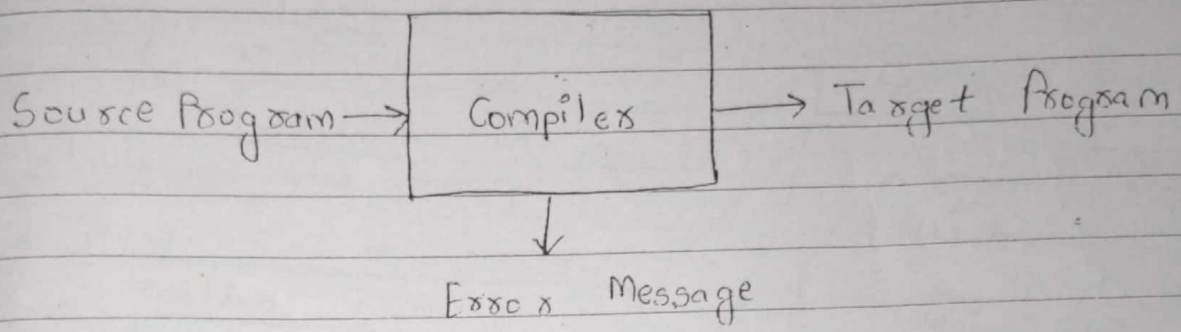
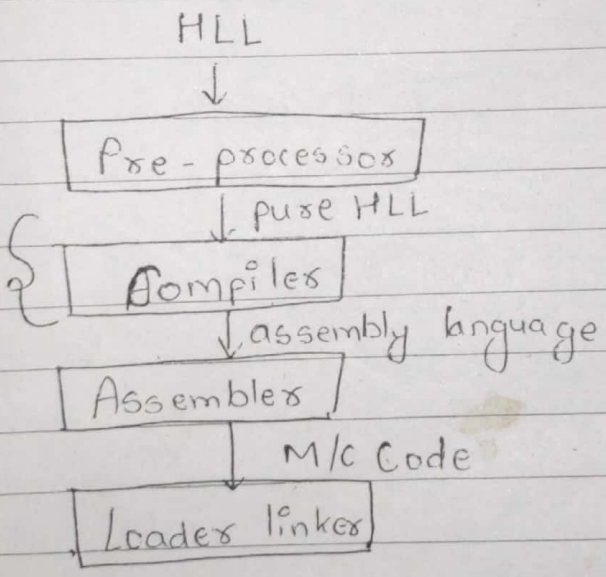
Q 1) what is compiler?

↳ A compiler is a program that takes a program written in a source language and translates it into an equivalent program in a target language.

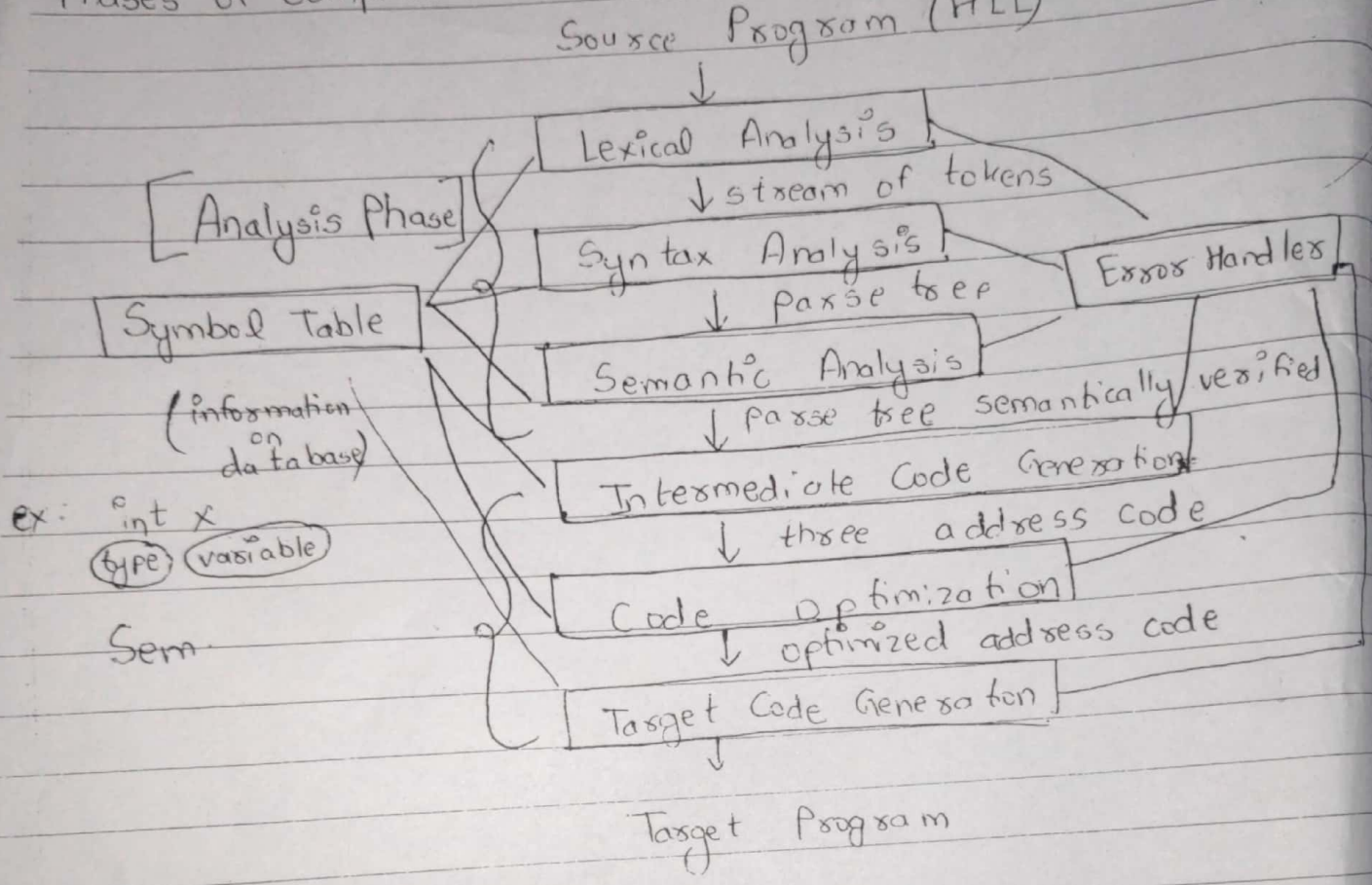


#



V.V.V. Imp

Phases of Compiler:



ex: int x

(type) (variable)

Sem

Useless Grammar

Context-free Grammar: Defⁿ: CFG consisting of a finite set of grammar rules is a quadruple (V, T, P, S)

V T P S

variable terminal ↓ Production Start

where,

ex: $S \rightarrow 0S0 \mid 1S1 \mid \Sigma$

$\rightarrow 01S10$

$\rightarrow 0110$

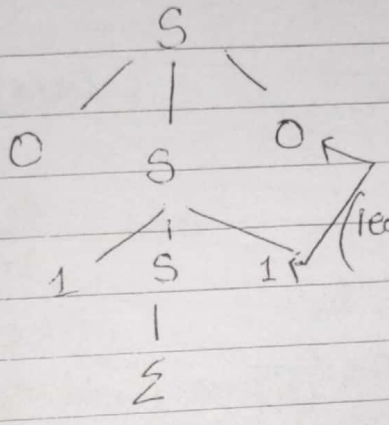
V = variable

T = set of terminals

P = set of rules i.e. left side of production

S = start symbol

Parse Tree:



leaf nodes are variables not here

Example 1:

x = a+b*c;

LA

id = id + id * id;

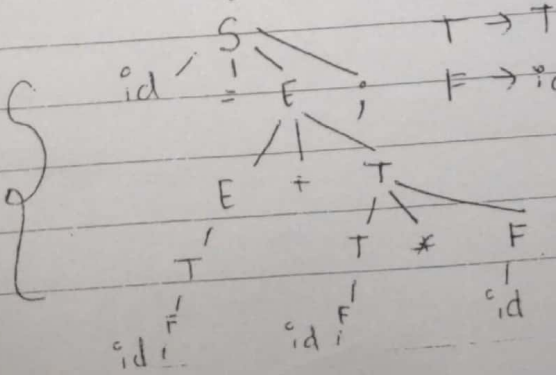
SA ⇐ S → id = E;

E → E+T | T

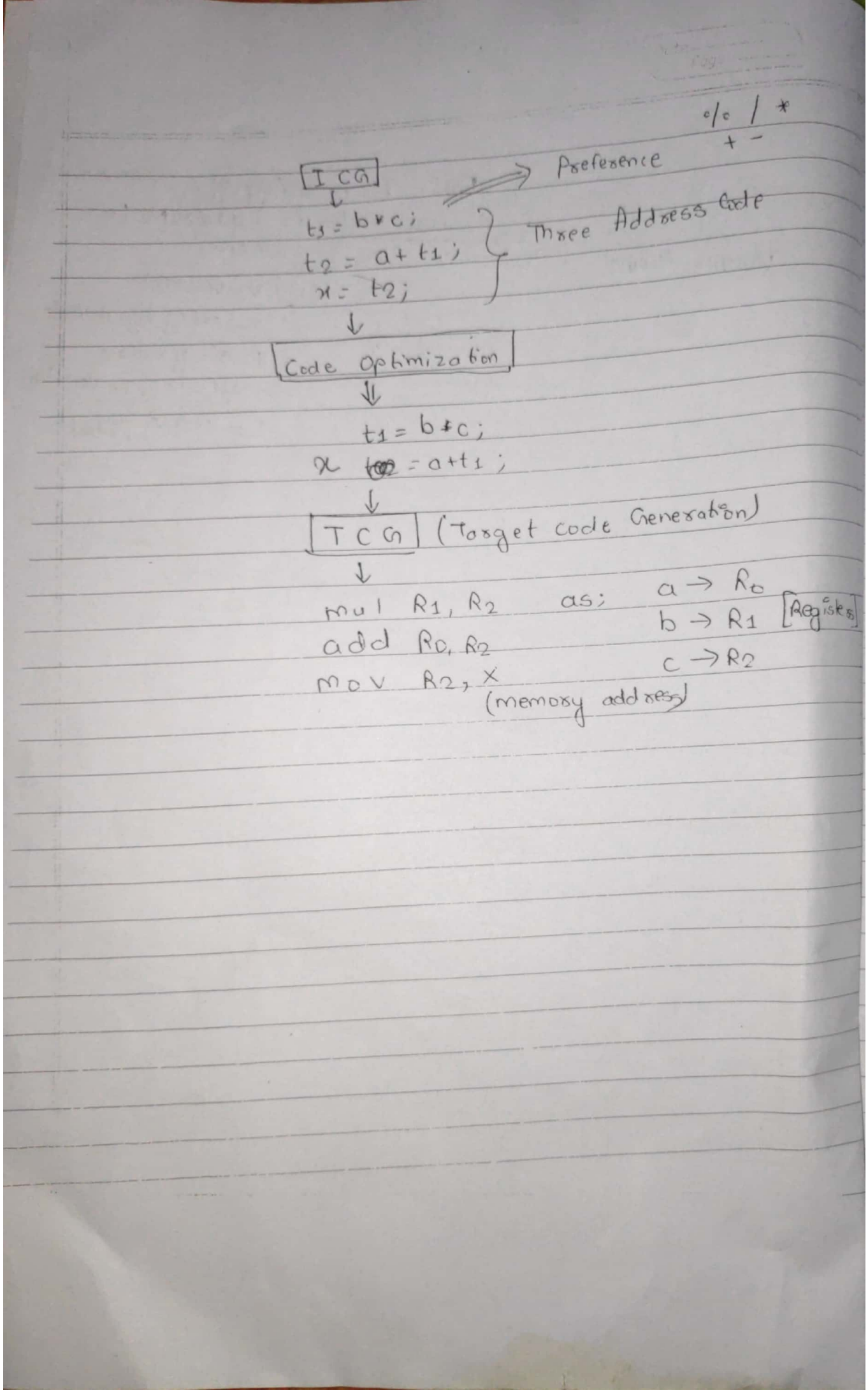
T → T * F | F

F → id

Context-Free Grammar



parse-tree ⇒ SA ⇒ parse tree semantically verified



type checking error

Symbol Table :

A symbol table stores information about keywords and tokens found during Lexical Analysis.

The symbol table is consulted on almost all phases of compiler.

For ex : insert ("dist", id);

// insert a symbol table entry associating the string "dist" with token type - id.

lookup ("dist");

// an occurence of the string "dist" can be looked up in the symbol table. If found, a reference to its token type is returned. Else, lookup returns zero.

Error Handling :

1. Error may encounter at any phase of compiler.
2. Objective of Error Handling is to go as far as possible in the compilation whenever an error is encountered.

For ex : ⊙ Handling missing symbols during Lexical Analysis by inserting symbols.
(preference for symbolic error)

⊙ Automatic type conversion during Semantic Analysis

(space — isn't used as a token)

CHAPTER - 2

Lexical Analysis

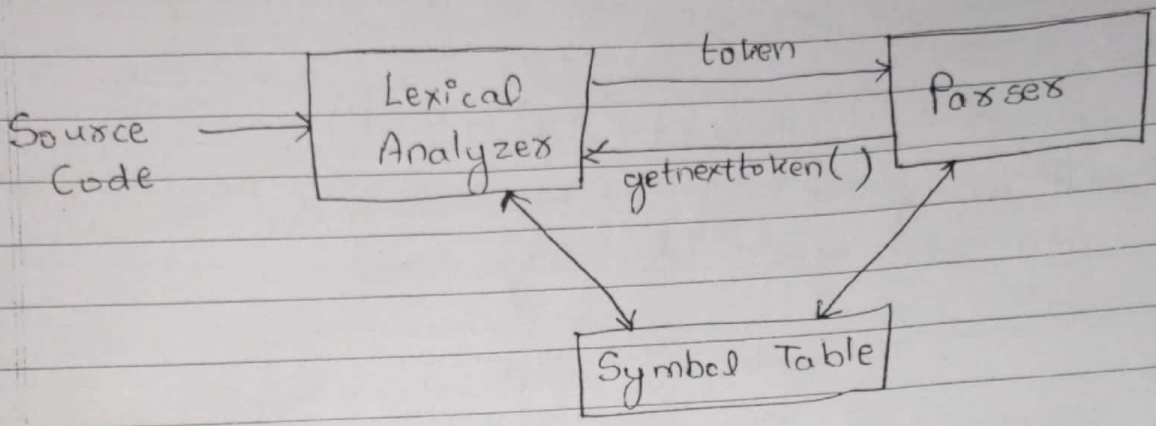
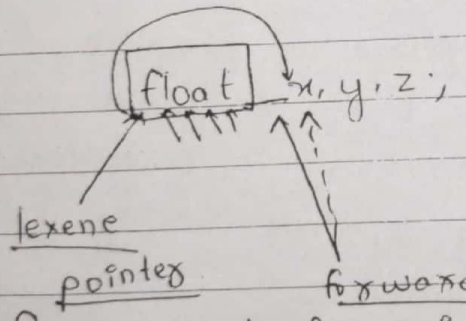


Fig: Interaction of Lexical Analyzer with parser

Look ahead and Buffering



[After space; float is a token.

[each start of a token until the next]

Specification of Tokens:

1) Alphabets : A finite set of symbols (a, b, c ... z)

2) String : A finite set of alphabets.

$\Sigma^* = \{a, \ldots z\}$

$L = \{bba, abb, can, dog \ldots\}$

3) Kleen Closure:

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

$\hookrightarrow \Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb \ldots\}$

4) Language. Collection of string
(selected strings from Kleen closure)

$L \subseteq \Sigma^*$

5) Regubr Expression:-

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

① Starting with zero; ending with 1:

$0(1+0)^*1$

or;

$0\,\varepsilon^*\,1$

(Regular Language are accepted by Finite Automata and are Expressed by Regular Expression.

⊙ Binary number multiple of 2:

[2, 4, 6, 8, 10 ...]

110,

{10, 100, 1000 ... }

i.e. Σ^*0 → Must be 0 at the end and ahead of 0 any amount of number can be.

(Regular Expressions)

Q. R.E that gives binary strings having at most two 1's.

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

~~$0^*(1+0)\ 0^*(1+0)\ 0^*$~~

$L = \{ 0, 1, 11, 01, 10, 010, 001, 100, \ldots$
01001000, 01001, 010100, ... $\}$

$$L \rightarrow (0^* + 0^*10^* + 0^*10^*10^*)$$

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

Languge

Q. R.E for string containing single 1.

$$0^*10^*$$

L = collection of strings

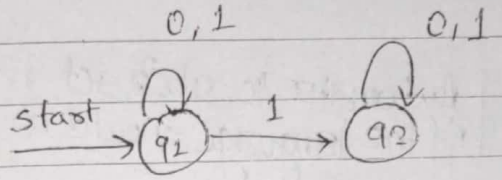
= {1, 01, 10, 010, 0001, 0001000000, ... }

Q. R.E string containing at least single 1.

$L = \{1, 001, 010, 0110, 0111100, \ldots\}$

$0^* 1 0^* + 0^* 1^* 0^*$ ✗

↳ $\boxed{\Sigma^* 1 \Sigma^*}$ ✓



Q. Contains the string 001 as substring:

$\Sigma^* 001 \Sigma^*$

Q. String with exactly 3 ones:

$0^* \underline{1} 0^* \underline{1} 0^* \underline{1} 0^*$

Q. String that have substring either 001 or 100

$\Sigma^* 001 \Sigma^* + \Sigma^* 100 \Sigma^*$

↳ $\Sigma^* (001 + 100) \Sigma^*$

Q. R.E for the string ending with 11.

$\Sigma^* 11$

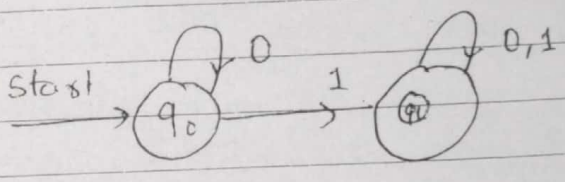
Q. R.E for identifying (denoting) C identifiers:

(Alphabet + _) (Alphabet + digit + _)*

(Alphabet + _) → a-z, A-Z

Finite Automata [Automata is abstract model that tells if the language should be accepted or not]

1) D.F.A (Deterministic finite Automata)

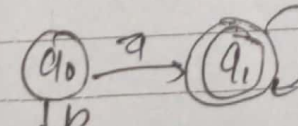


δ	0	1
→ q_0	q_0	q_1
* q_1	q_1	q_1

DFA: on applying certain string one state goes to any other particular state.

Q → set of finite states
Σ → set of alphabets (0,1)
δ → transition
q_0 → start
F → set of final states

2) start with a for Σ(a,b)



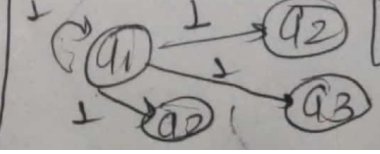
→ this is dead state so not we don't start with b (Trapping)

$0 \times \Sigma \Rightarrow 0$

δ	a	b
q_0	q_1	q_2 (Dead state)
q_1	q_1	q_1
q_2	q_2	q_2

NFA: on applying string on state it may go to one or multiple state.

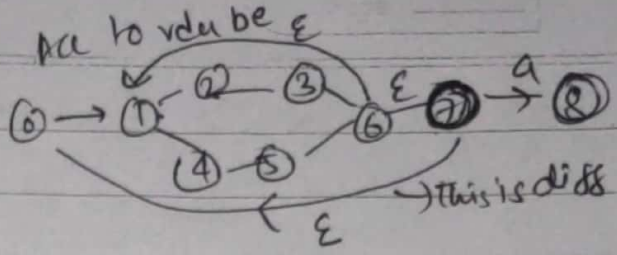
Suppose,



[multiple choices]

Non-Deterministic Automata

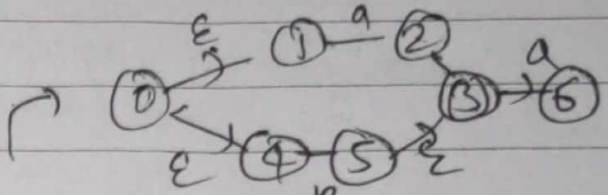
(a+b)*a



Reducibility

1) NFA to DFA
2) RE to NFA
3) RE to DFA

Acc to geeks for geeks.
(a+b)*a is



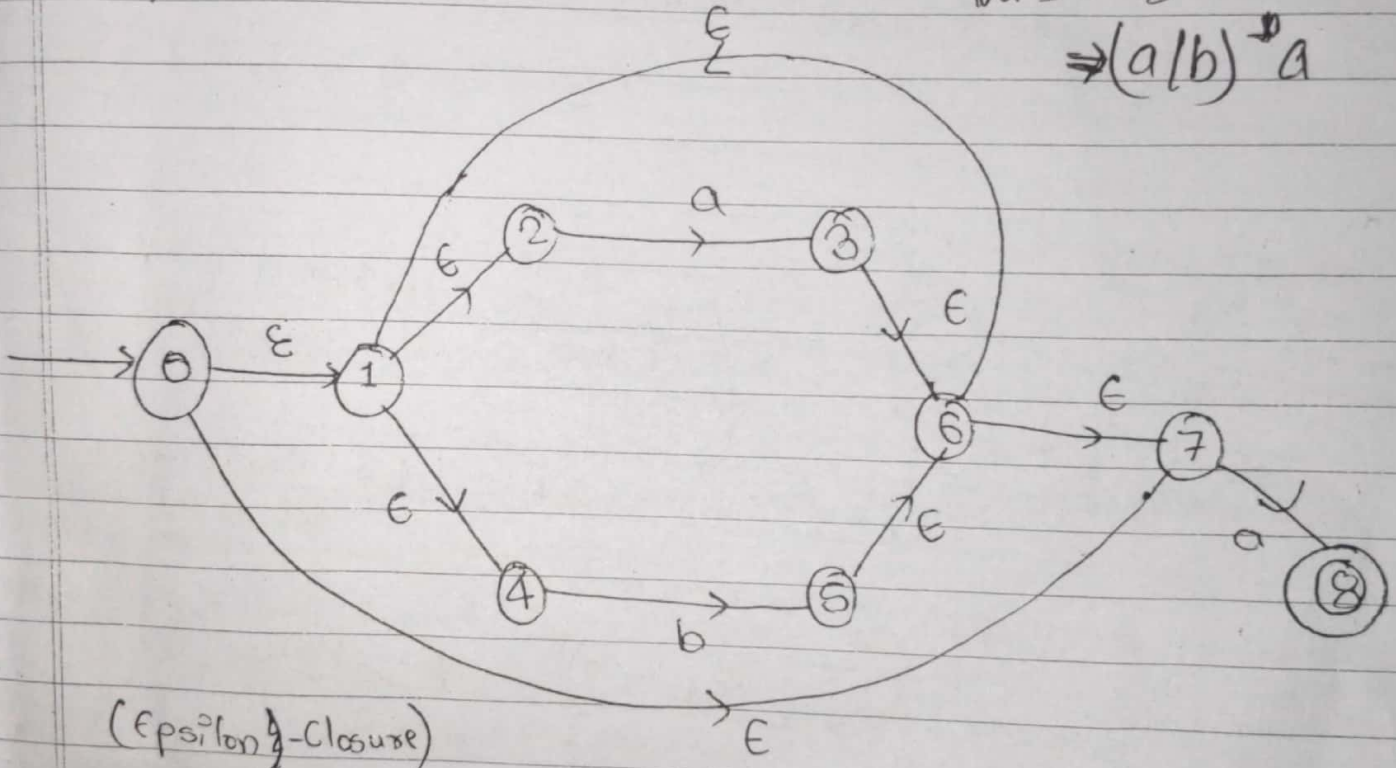
but different by sir

1) N.F.A to D.F.A (subset construction)

⇒

This is reg expression for above figure

⇒ (a/b)*a



(Epsilon)-Closure)

ε-closure of state 1 = {1, 2, 4}

ε-closure of state 0 = {0, 1, 2, 4, 7}

ε-closure of state 2 = {2}

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

→ 1) N.F.A to D.F.A (subset construction)

(a लाई Liye-मा कहाँ जाँछ)

⇒ From above fig:

$S_0 = \epsilon\text{-closure}(\{0\}) = \{0, 1, 2, 4, 7\}$ (2 —a→ 3, 7 —a→ 8)

Mark S_0

$\epsilon\text{-closure}(move(S_0, a)) = \epsilon\text{-closure}(3, 8)$

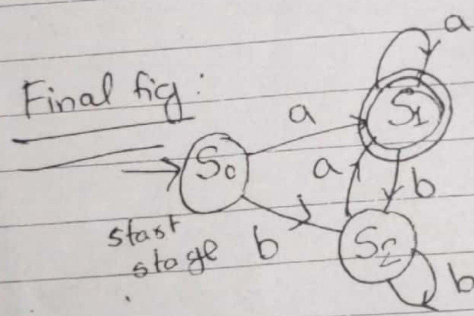
$= \{3, 6, 7, 1, 2, 4, 8\}$ —a→ ?

$= S_1$

$\epsilon\text{-closure}(move(S_0, b)) = \epsilon\text{-closure}(5)$

$= \{5, 6, 7, 1, 2, 4\} = S_2$

Final fig:



(Brief version of above figure)

If bba then; we can get to the final state

Mark S1

$\epsilon\text{-closure}(move(S1, a)) = \epsilon\text{-closure}(8, 3)$

$= S1$

$\epsilon\text{-closure}(move(S1, b)) = \epsilon\text{-closure}(5)$

$= S_2$

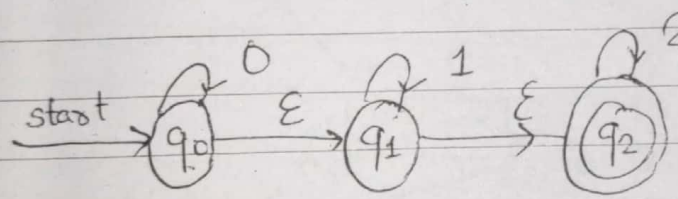
Mark S2

ε-closure (move (S_2, a)) = ε-closure (8, 3)
= S_1

ε-closure (move (S_2, b)) = ε-closure (5)
= S_2

↳ Those which have ((8)) are final states: S_1
= {3, 6, 7, 1, 2, 4, 8}

Start state → (0) S_0; S_0 #



Σ = {0, 1, 2}

S_0 = ϵ-closure ({q_0}) = {q_0, q_1, q_2}

Mark S_0:

ε-closure (move (S_0, 1)) = $^{\epsilon}${q_1} = S_2 = {q_1, q_2}
ε-closure (move (S_0, 0)) = $^{\epsilon}${q_0} = S_0 = {q_0, q_1, q_2}
ε-closure (move (S_0, 2)) = $^{\epsilon}${q_2} = S_3 = {q_2}

Mark S_1:

ε-closure (move (S_1, 0)) = $^{\epsilon}${q_0} = S_0 = {q_0, q_1, q_2}
ε-closure (move (S_1, 1)) = {} / ϕ
ε-closure (move (S_1, 2)) = {} / ϕ

Mark S_0 :

ε-closure (move $(S_2, 0)) = \phi$

ε-closure (move $(S_2, 1)) = \varepsilon \{q_1\} = S_2 = \{q_1, q_2\}$

ε-closure (move $(S_2, 2)) = \phi$

Mark S_3:

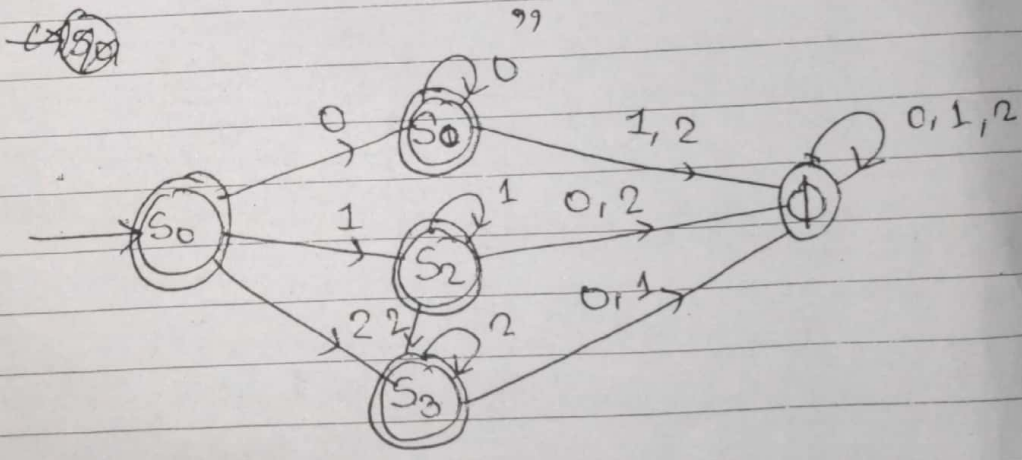
ε-closure (move $(S_3, 0)) = \phi$

ε-closure (move $(S_3, 1)) = \phi$

ε-closure (move $(S_3, 2)) = \varepsilon \{q_2\} = S_3 = \{q_2\}$

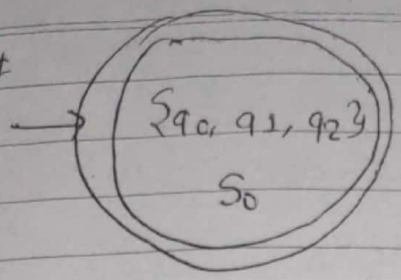
Mark ϕ:

ε closure (move (ϕ, 0, 1, 2)) = ϕ

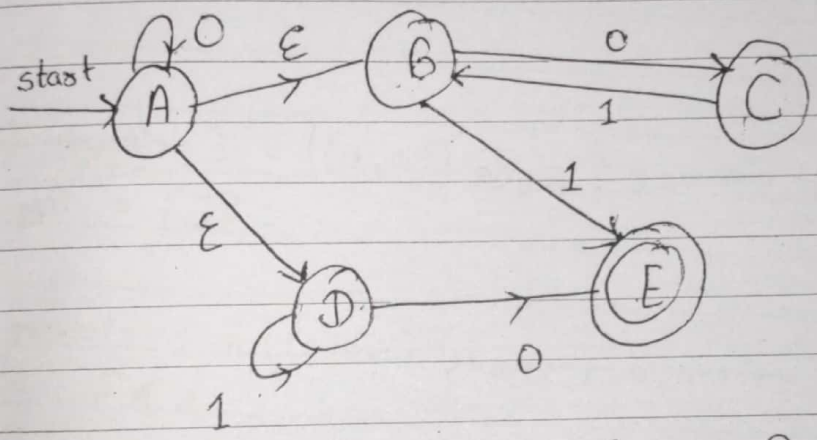


Final state consists (q2) : S_3 and S_0

Or #



#



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

S_0 = ε-closure ({A}) = {A, B, D}

Mark S_0

ε-closure (move (S_0, 0)) = ε-closure (A, C, E)

= {A, B, D, C, E}

= S_1

ε-closure (move (S_0, 1)) = ε-closure (E, D)

= {E, D}

= S_2

Mark S_1 :

ϵ-closure (move $(S_1, 0)$) = ϵ-closure (A, C, E)

$= \{A, B, C, D, E\}$

$= S_1$

ϵ-closure (move $(S_1, 1)$) = ϵ-closure (B, E, D)

$= \{B, E, D\}$

$= S_3$

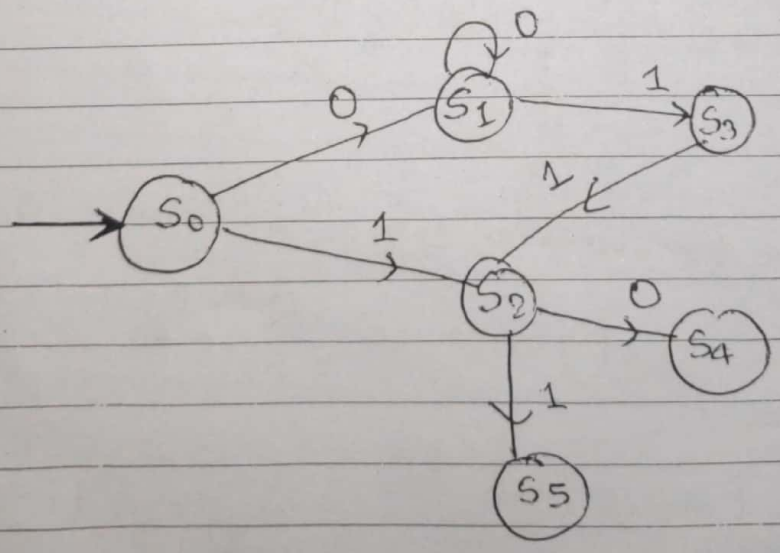
Mark S_2 :

ϵ-closure (move $(S_2, 0)$) = ϵ-closure (E)

$= \{E\} = S_4$

ϵ-closure (move $(S_2, 1)$) = ϵ-closure (D)

$= \{D\} = S_5$



Mark S_3:

ϵ - closure (move(S_3, 0)) = ϵ closure (C, E)

= {C, E}

= S_6

ϵ - Closure (move (S_3, 1)) = ϵ -closure (F, D)

= {F, D}

= S_2

Mark S_4:

Mark S_5:

$S_0 = \{A, B, D\}$

	0	1
→ {A, B, D}	{A, B, C, D, E} (including yourself)	{E, D}
*{A, B, C, D, E}	{A, B, C, D, E}	{B, D, E}
*{D, E}		
*{B, D, E}		
*{E}		
{D}		
*{C, E}		
{B}		
{C}		
ϕ		

There cannot be more than 5 final-states when the states are {A, B, C, D, E}

NFA लाई DFA मा लाँदा Trap (ϕ) state chahinxa

Yedi NFA ~~dha~~ nai DFA xa

that means DFA → to DFA; no trap state needed

2) RE to NFA (Thomson's Construction)

Input ⇒ RE, γ over alphabet A

Output ⇒ ϵ-NFA accepting $L(\gamma)$

Eg: $\gamma = a(a+b)^*b$ (starting with a, ending with b)

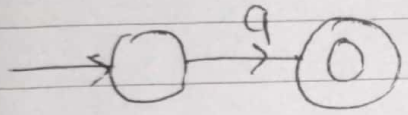
↓

[Regular expression]

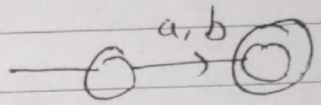
$L(\gamma) = \{ab, aab, abbb, aabbb\}$

[Language of regular expression → collection of strings generated by Regular Language.

$\gamma = a$

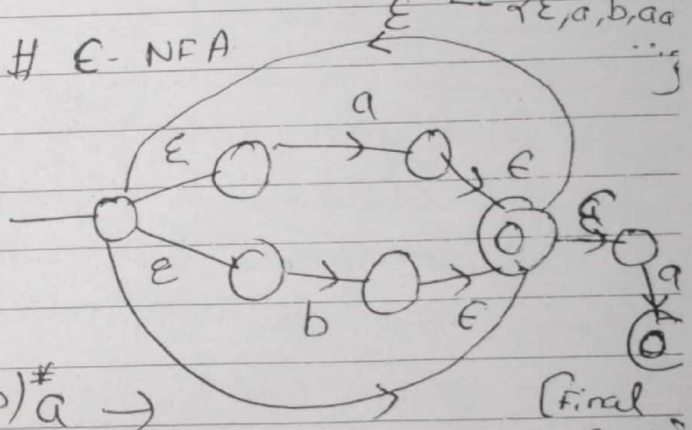


$\gamma = a+b$



∈-NFA

$L = \{\varepsilon, a, b, aa \ldots\}$



$(a+b)^*a$ → Concatenation

[Final State]

∈ will accept every string given by:

⊙ R.E to D.F.A

↳ 1) Convert the given expression to augmented expression by concatinating with hash.
i.e. $(r)\#$

Ex: $r = (a+b)^*abb$ or $(a|b)^+abb$

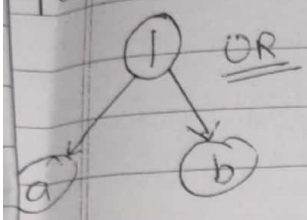
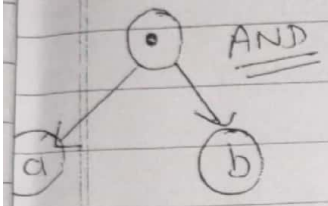
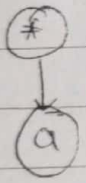
↳ $r = (a+b)^*abb\#$

2) Construct the syntax-tree of this augmented R.E. In this tree, all the operators will be inner-node and all the alphabets, symbols including # will be leaves. (leaf nodes)

3) Number each leaves.

4) Traverse tree to construct nullable, firstpos, lastpos and followpos.

~~Num~~ Rule to evaluate nullable firstpos and lastpos.

Node - n	Nullable (n)	firstpos (n)	lastpos(n)
leaf labelled ε	True	$\{\}$	$\{\}$
non null leaf position i	False	$\{i\}$	$\{i\}$
 OR	nullable (a) or nullable (b)	firstpos(a) $\cup$ firstpos (b)	last pos (a) $\cup$ lastpos (b
 AND	nullable(a) and nullable(b)	if nullable(a) is true then; firstpos (a) $\cup$ first pos (b) else firstpos(a)	If nullable (b) is true then; last pos (a) $\cup$ last pos(else lastpos (b)
	True	first pos(a)	lastpos(a)

Q. Convert the R.E $(a+b)^*abb$ into D.F.A directly.

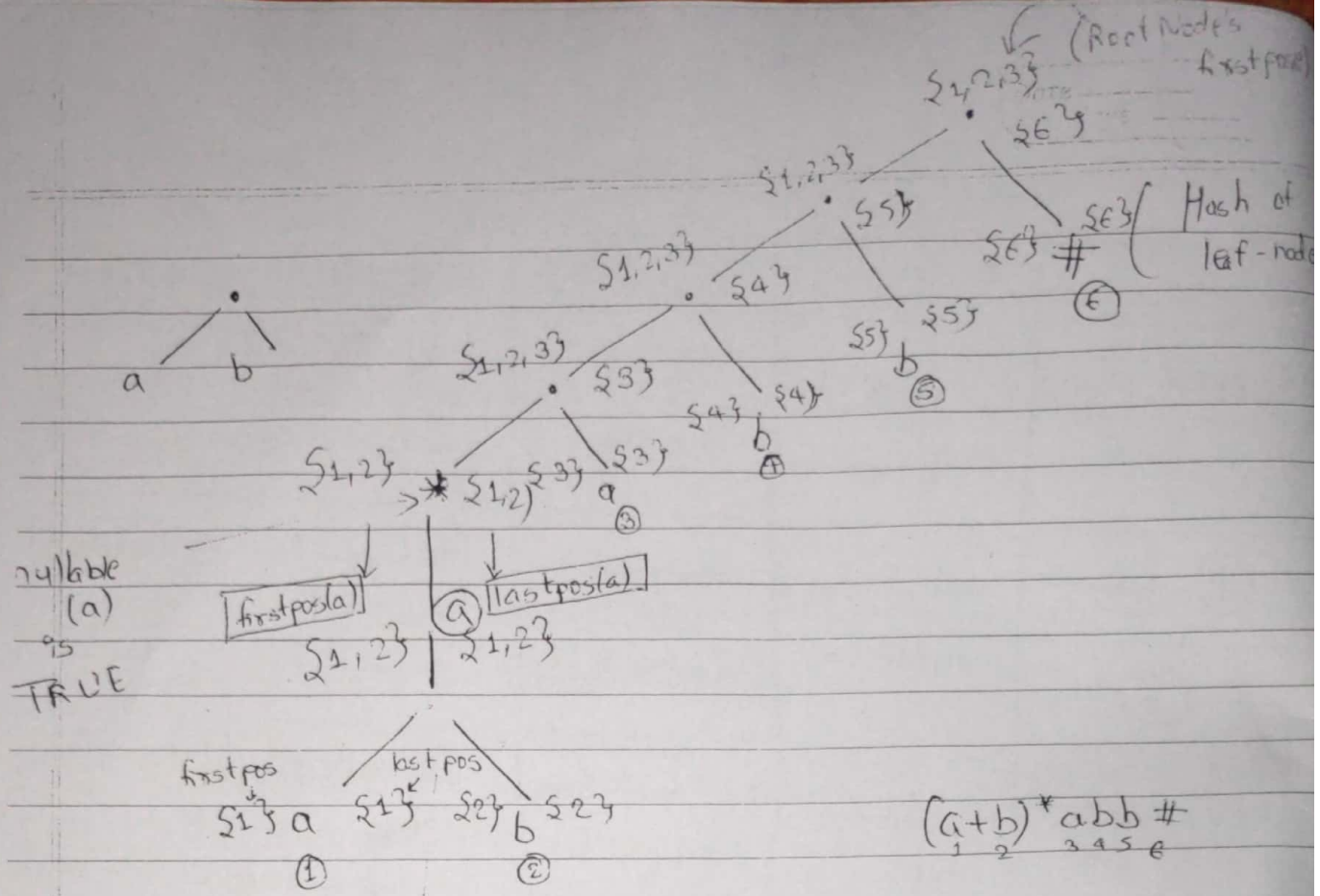
Soln;

1. Augmenting the regular expression; we get.

$$(a+b)^*.abb.\#$$

2.

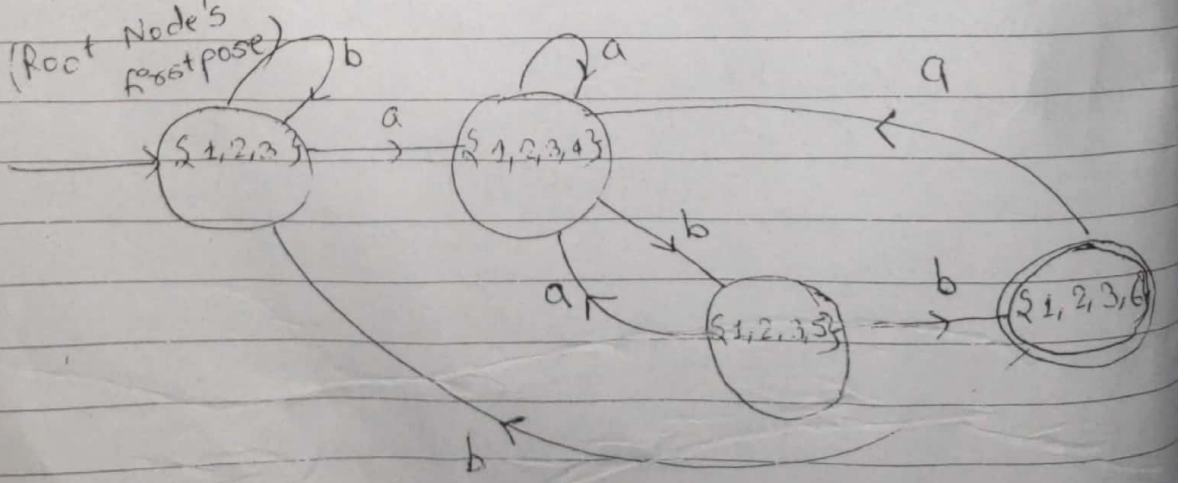
(P.T.O)



$(a+b)^{*}abb\#$
1 2 3 4 5 6

Only star and concat(.) nodes will be considered!!!

Node		followpos
a	1	{1, 2, 3}
b	2	{1, 2, 3}
a	3	{4}
b	4	{5}
b	5	{6}
#	6	—



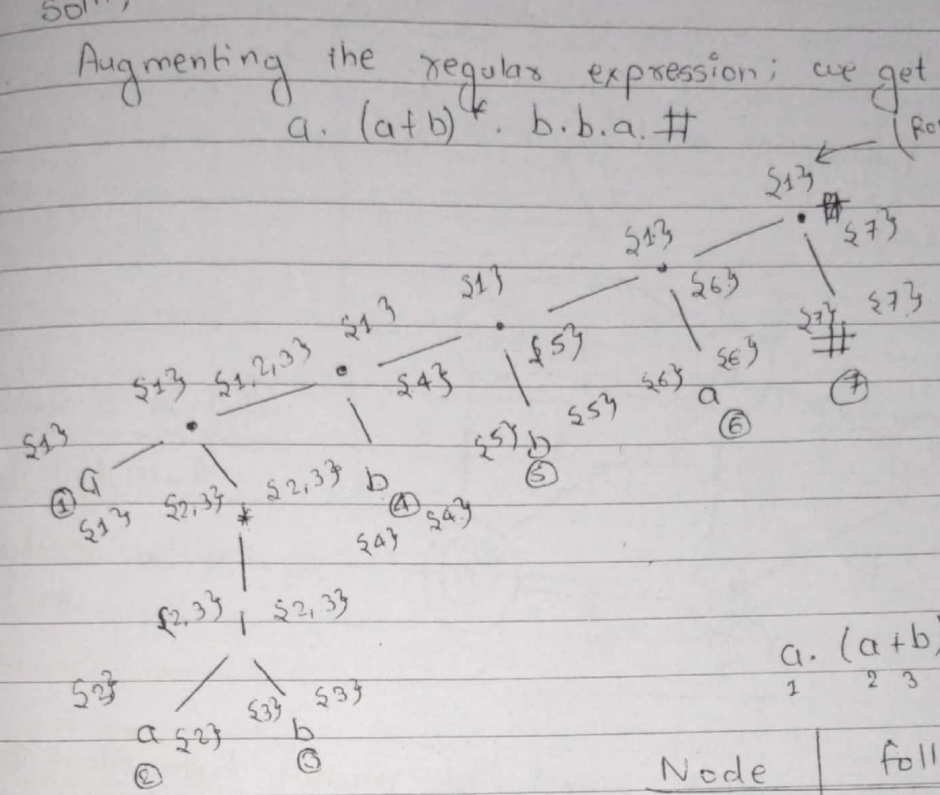
2. Convert the R.E $a.(a+b)^*.b.b.a$ into D.F.A directly.

Soln;

1. Augmenting the regular expression; we get.

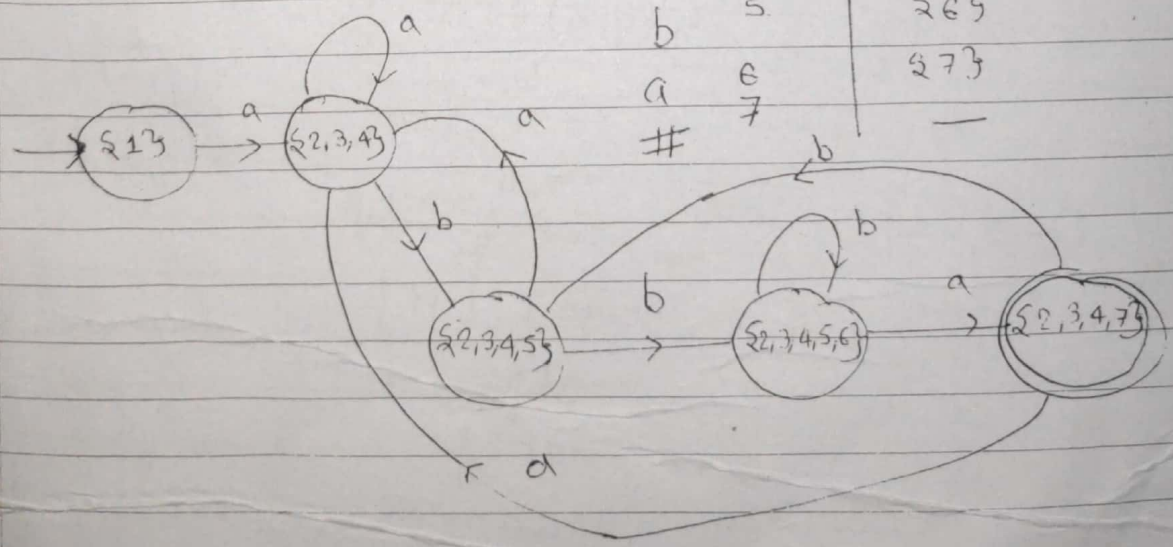
$a.(a+b)^*.b.b.a.\#$

2.



$a.(a+b)^*.b.b.a\ \#$
1 2 3 4 5 6 7

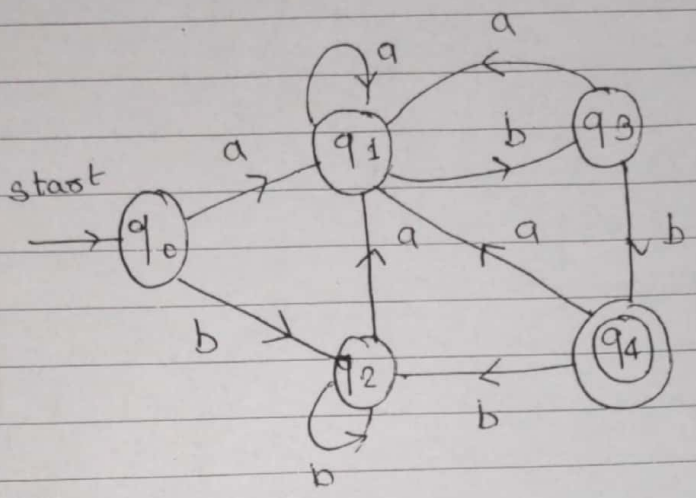
Node		followpos
a	1	{2, 3, 4}
a	2	{2, 3, 4}
b	3	{2, 3, 4}
b	4	{5}
b	5	{6}
a	6	{7}
#	7	—



⑥ State Minimization in D.F.A

↳ reducing the number of states from D.F.A.

[Language must be same]



q_4 is reachable from q_0

So; it is Reachable state.

Those states which cannot be reached from start state is called Unreachable State.

$\delta(p, a) \in F$

$\delta(q, a) \in F$

[p & q are equivalent even in:

$\delta(p, a) \notin F$

$\delta(q, a) \notin F$]

0- equivalent states:

Non-Final & Final

$[q_0, q_1, q_2, q_3]$ $[q_4]$

States Grouping

δ	a	b
q0	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	q4
q4	q1	q2

Does q0 and q1 belong in same group?

Equivalent for 'a'

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

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

q0 and q1 on 'b'

go to {q2 and q3}

which is on same group

i.e. [q0, q1, q2, q3]

and not on Final Group.

[q4]

Equivalent for 'b'

1 - equivalent

[q0, q1, q2] [q3] [q4]

can be compared with anyone of these three

q3 and q2 on b

goes to

q4 and q2

Not on same states

So; not-equivalent at all.

0-eq: $[q_0, q_1, q_2, q_3]$ $[q_4]$

1-eq: $[q_0, q_1, q_2]$ $[q_3]$ $[q_4]$

2-equivalence:

Compare q_0 and q_1:

q_0 on b $\longrightarrow q_2$

q_1 on b $\longrightarrow q_3$ (different state

[Not 2-equivalence]

↳ 1-eq. $[q_0, q_1, q_2]$

2-eq: $[q_0, q_2]$ $[q_1]$ $[q_3]$ $[q_4]$

δ	a	b
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2

Equivalent

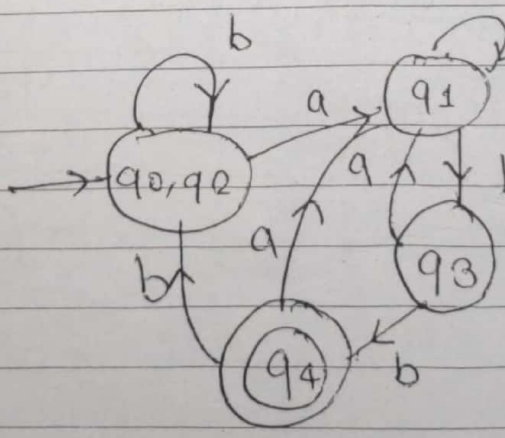
3-eq: $[q_0, q_2]$ $[q_1]$ $[q_3]$ $[q_4]$

as;

$\delta(q_0, a) \notin F$

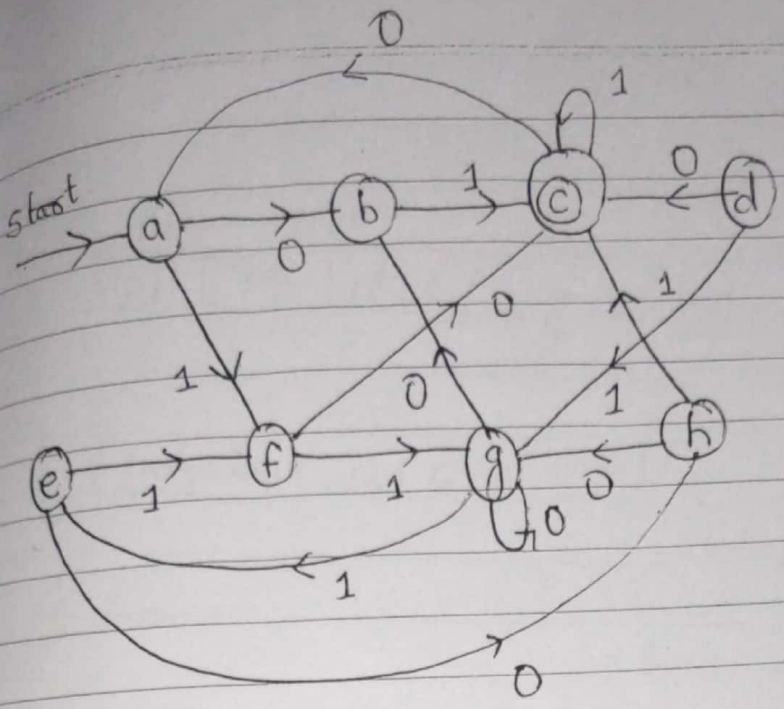
$\delta(q_2, a) \notin F$] So equiv

STOP!



This is called DFA State Minimization

Q)



Since (d) is an Unreachable state

#0-equivalence : [a, b, e, f, g, h] [c]

Non-Final and Final States

δ	0	1
a	b	f
b		c
e	h	f
f	c	g
g	b, g	e
h	g	c
c	a	c

a and b are not equivalent as:

$\delta(a, 1) \rightarrow f$

$\delta(b, 1) \rightarrow c$

} different states

a and e are equivalent

e and g are equivalent

b and h are Equivalent.

1-equivalence : [a, e, g] [b, h] [f] [c]

Comparing b and h

↓

No state

{ Not equivalent }

Comparing a and e

$a \xrightarrow{0} b$

$e \xrightarrow{0} h$

] Same state

Comparing e and g

$e \xrightarrow{1} f$

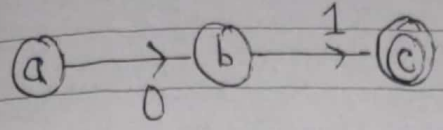
$g \xrightarrow{1} e$

] Different states

2 - equivalence : [a, e] [g] [b] [h] [f] [c]

3 - equivalence : [a] [e] [g] [b] [h] [f] [c]

(As a and e gives different states)



b and h are Equivalent.

1-equivalence : [a, e, g] [b, h] [f] [c]

Comparing a and e

a $\xrightarrow{0}$ b
e $\xrightarrow{0}$ h
] Same state

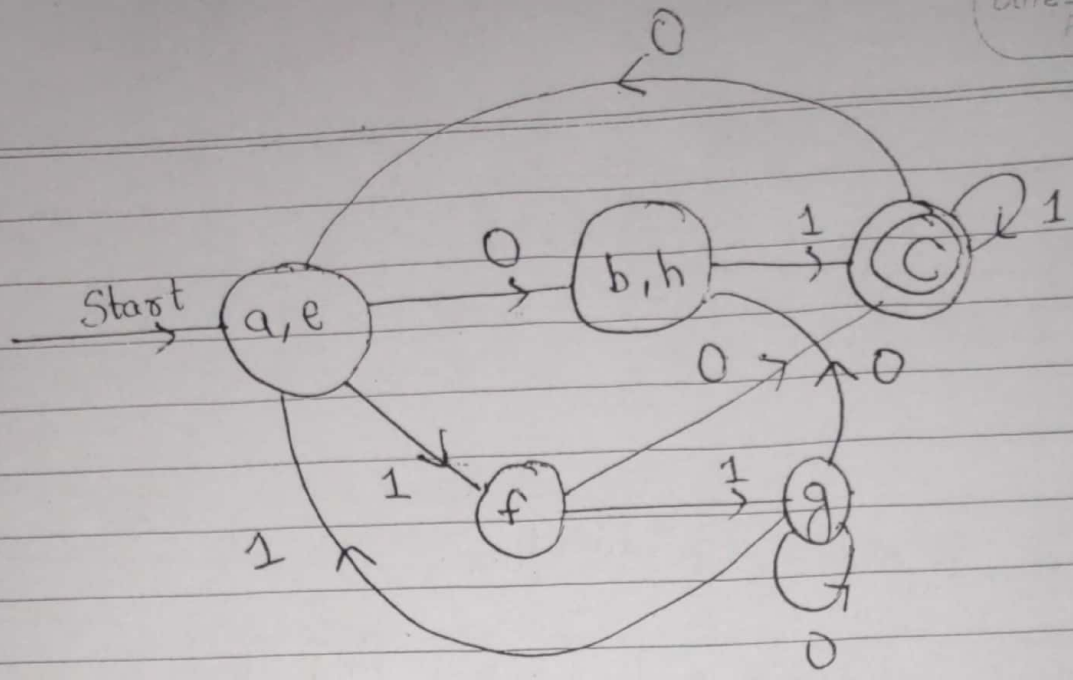
Comparing e and g

e $\xrightarrow{1}$ f
g $\xrightarrow{1}$ e
] Different states

2-equivalence : [a, e] [g] [b, h] [f] [c]

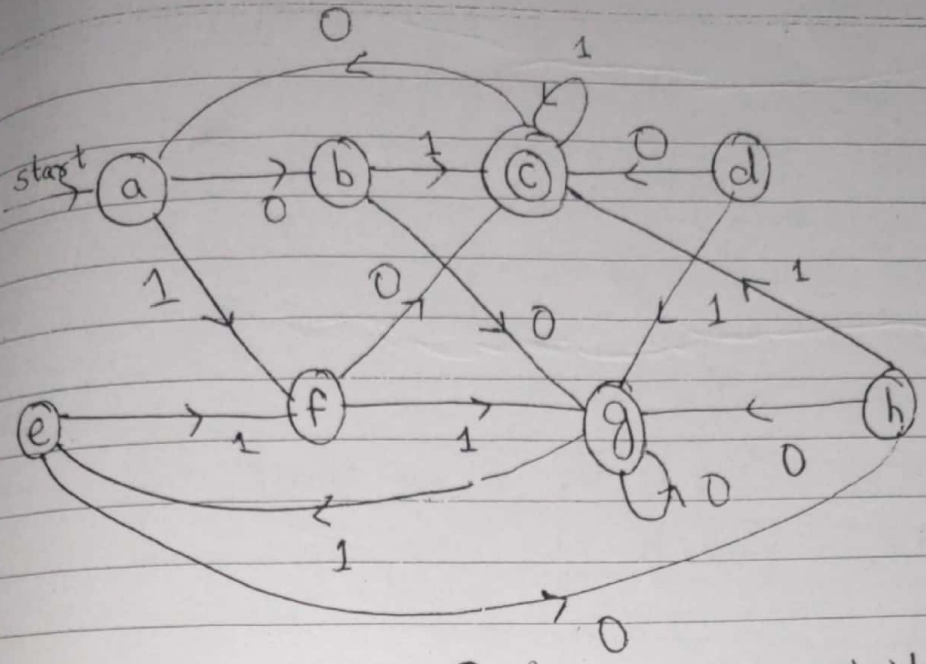
3-equivalence : [a, e] [g] [b, h] [f] [c]

P. T. O



Minimization D.F.A

Q)



Since (d) is an Unreachable State

0-equivalence : [a, b, e, f, g, h] [c]

Non-Final & Final States

δ	0	1
a	b	f
b	g	c
e	h	f
f	c	g
g	g	e
h	g	c
c	a	c

a and b are not equivalent as:

$\delta(a, 1) \rightarrow f$
$\delta(b, 1) \rightarrow c$
] different states

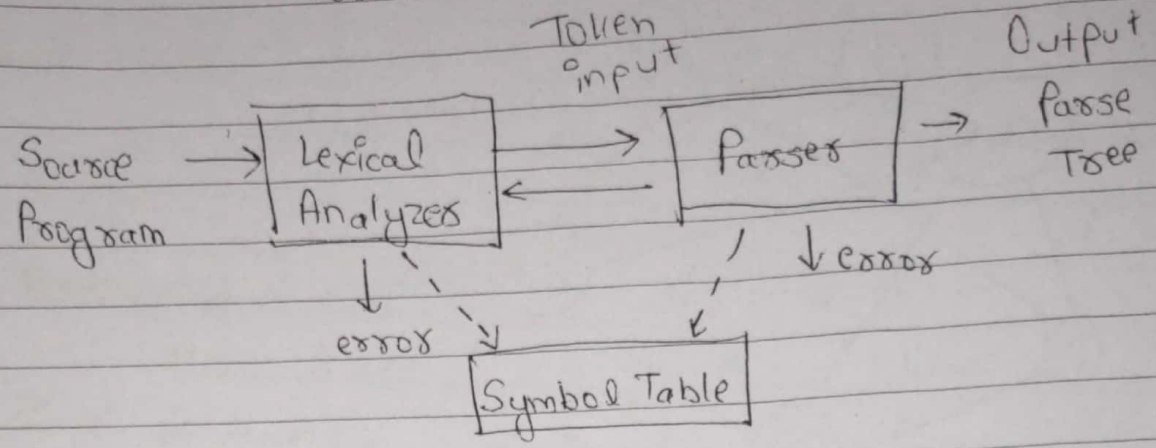
a and e are equivalent

e and g are equivalent

h and h are equivalent

Date
Page

Parser



Syntax analyzer is also called the parser. Its job is to analyze the source program based on the definition of its syntax.

It works in lock-step with lexical Analyzer and is responsible for creating the parse-tree.

↳ It implements a Context-Free Grammar.

⇓

- Gives a precise syntactic specification of a programming language.

Top-Down Parser

↳ the parse tree is created top-to-bottom, starting from the root.

Bottom-Up Parser

↳ the parse is created bottom to top; starting from the leaves.

[Top-leftmost will be default start-symbol]

CFG (N, T, P, S) — N: non-terminal; T: terminal; P: production rule : $\alpha \to \beta$

$\alpha \in N^* N N^*$

$\beta \in (N \cup T)^*$

$S \in N$ is a designated start symbol.

[α and β can be both terminals and non-terminals]

CFG: Derivations

$E \to E+E$

$\to id+E$

$\to id+id$

$L(G)$ → Language of Gramma

↳ A sentence of $L(G)$ is a string of terminal symbols of G.

If S is the start symbol of G then w is a sentence of $L(G)$ iff $S \Rightarrow w$ where w is a string of terminals of G.

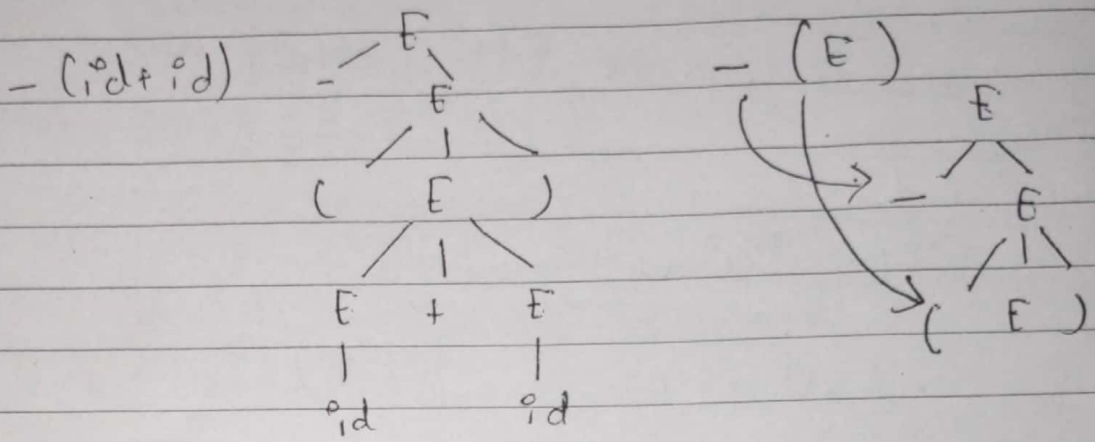
#rm → Right Most Derivation;

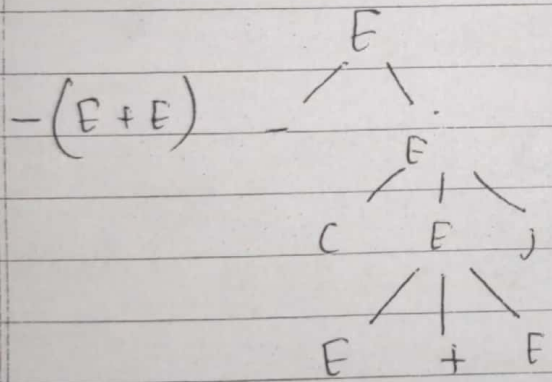
we replace rightmost variable first.

($\overset{*}{\Rightarrow}$ (hidden step)) derives in 0 or more than 1 step.

A Parse-tree is a graphical representation of a CFG derivation.

Inner nodes are non-terminal symbols.





Ambiguity:

↳ A grammar produces more than one parse-tree for a sentence is called an ambiguous grammar.

[Compiler Design]

Class Work

1) ans:
Parsing is a technique where a token string is taken as input and with the help of existing grammar, converted to Parsing Tree / Parse Tree. [Syntax Analyzer]

Types of Parsing:

- Top-Down Parser:
  ↳ This type generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. starts from start symbol and ends on terminals; using leftmost derivation.

  i) Recursive descent parser / Backtracking / Brute force Parser: Generates parse tree using Brute force.

  ii) Non-recursive descent parser: Predictive parser without backtracking / dynamic parser using parse table to generate parse tree.

- Bottom-up Parser:
  ↳ It generates parse tree for the given input string with the help of grammar productions by compressing the non-terminals. It starts from non-terminals and ends on start symbol; uses reverse of the rightmost derivation.

i) LR parser: Generates parse-tree for the given string by using unambiguous grammar.

ii) Operator precedence parser: Generates parse tree from given grammar and string where 2 consecutive non-terminals and epsilon never appear on the right-hand side of any production.

Alt:- A parser that reads and understands operator precedence grammar. It is bottom-up parser that interprets operator grammar. Ambiguous grammar are not allowed in any ~~grammar~~ except operator precedence parser.
parser

2) ans:

CFG consisting of a finite set of grammar rules is a quadruple (N, T, P, S).

Types of derivations in C.F.G:

i) Leftmost derivation: It is obtained by applying production to the leftmost variable in each successive step.

$S \rightarrow AaSS$

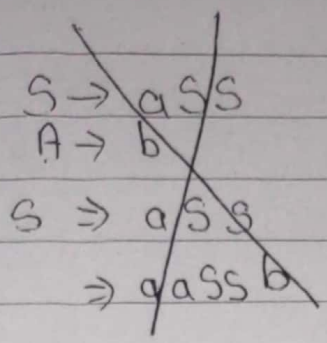
$A \rightarrow a$

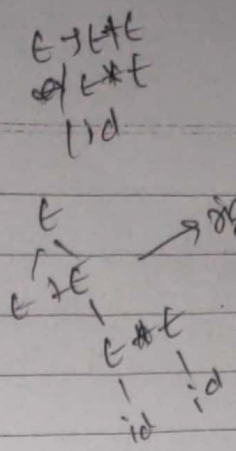
$S \Rightarrow AaSS$

$\Rightarrow aaSS$

$\Rightarrow aaAaSSS$ and so on.

ii) Rightmost derivation: It is obtained by applying production to the rightmost variable in each step





The rightmost derivation for the string 'a+a*a' may be-

$$X \rightarrow X^*X \rightarrow X^*a \Rightarrow X+X^*a \rightarrow X+a^*a \rightarrow a+a^*a$$

added later

iii) Mixed derivation : It is is obtained by applying productn to the leftmost and rightmost variable simultaneously as per the requirement in each successive step.

$S \Rightarrow aSS$
$\Rightarrow aSb$
$\Rightarrow aaSSb$
$\Rightarrow aabSb$
$\Rightarrow aabaSSb$
$\Rightarrow aabaSbb$
$\Rightarrow aababbb$

3) ans:
A CFG is said to be ambiguous if there exists more

than one derivation tree for the given input string i.e, more than one Left Most Derivation Tree (LMDT) or Right Most Derivation Tree (RMDT).

Defn ↳ $G = (V, T, P, S)$ is a CFG is said to be ambiguous if and only if there exists a string in T^* that has more than one parse tree.

$$A \rightarrow \alpha$$

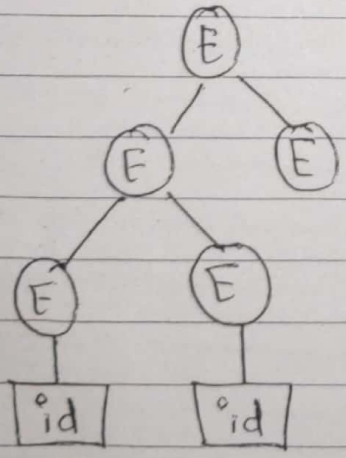
where; $\alpha \in (V \cup T)^* S$

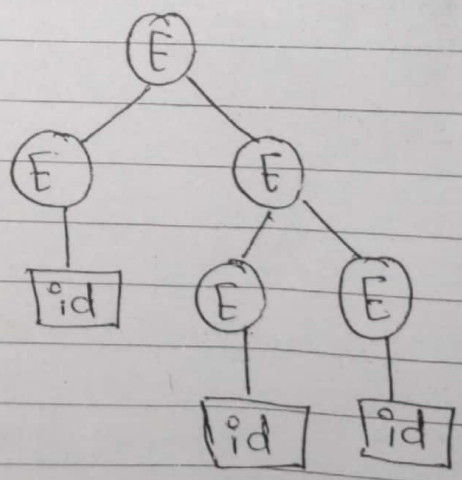
(Start Symbol)

For Example:

$$E \rightarrow E + E \mid id$$

We can create 2 parse tree from this grammar to obtain $id = id + id$





4) ans:

A Grammar $G(V, T, P, S)$ is left recursive if it has a production in the form:

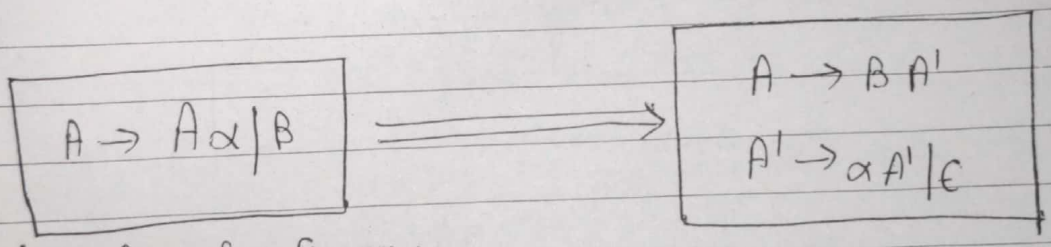
$$A \rightarrow A\alpha | \beta$$

↳ Above grammar is left recursive because the left of production is occuring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

$$A \rightarrow \beta A'$$
$$A \rightarrow \alpha A' | \epsilon$$

Elimination of Left Recursion

Left Recursion can be eliminated by introducing new non-terminal A such that.



Left Recursive Grammar

Removal of Left Recursion

This type of Recursion is also called Immediate Left Recursion.

Example: Removal of left-recursion from the grammar:

$E \rightarrow E(T) \mid T$

$T \rightarrow T(F) \mid F$

$F \rightarrow id$

Solution:

Eliminating immediate left-recursion among all $A\alpha$ productions we obtain:

$E \rightarrow TE'$

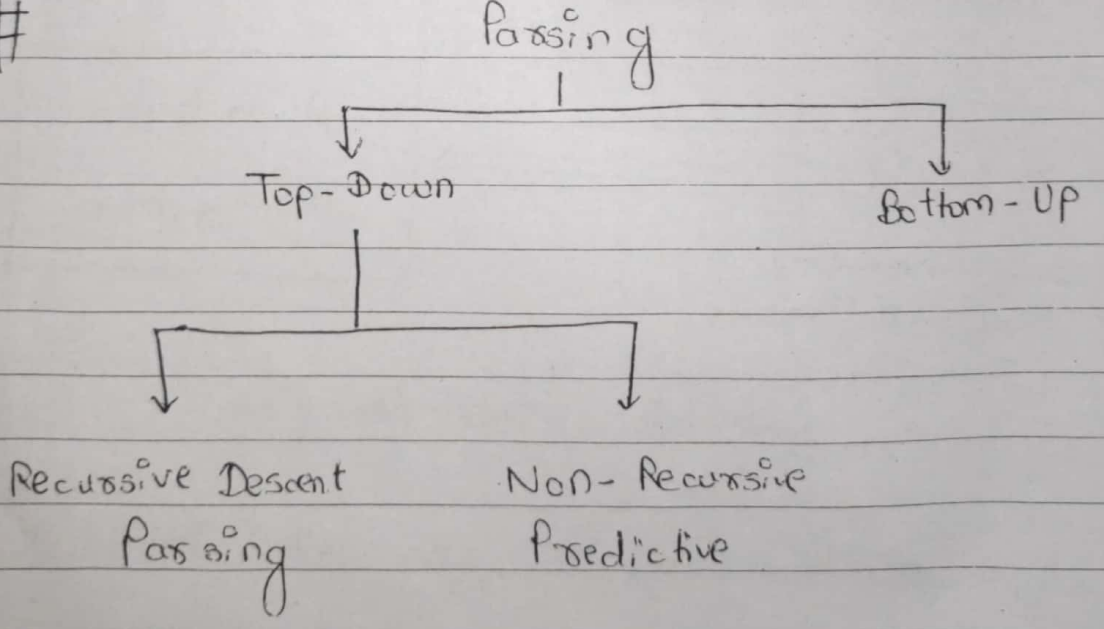
$E' \rightarrow (T)E' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow (F)T' \mid \epsilon$

$F \rightarrow id$

#



1) Recursive Descent Parsing:

Rules

1) Use 2 pointers iptr for pointing the input symbol to be read and optr for pointing the symbol of output string i.e initially start symbol : S.

2) If the symbol pointed by optr is non-terminal ; use the first production rule for expansion.

3) While the symbol pointed by iptr and optr is same, increment both pointers.

4) The loop at the above step terminates when
   a) A non-terminal at the output or, (case A)
   b) The end of the input or, (case B)
   c) On matching terminal symbols pointed by iptr and optr is same or, (case C)

5) If (A) is true ; expand the non-terminal using the first rule and go to step 2).

6) If (B) is true ; terminate with success.

7) If (C) is true ; decrement both the pointers to the place of last non-terminal expansion, and use the next production rule for the non-terminal.

8) If there is no more production and (B) is not ode; report error.

Example: Consider the Grammar

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Input string is "cad"

Production Rules:

Rule 1		
Rule 2	Rule 3	

[Algorithm Rules]

Input	Output	Rules Fired ↓
(iptr) cad	(optr) S	[Rule 1, Try $S \rightarrow cAd$]
(iptr) cad	(optr) cAd	[Rule 3, Match c]
c (iptr) ad	c (optr) Ad	[Rule 5, Try $A \rightarrow ab$]
c (iptr) ad	c (optr) ab d	[Rule 3, Match a]
ca (iptr) d	c a (optr) bd	[Rule 7, dead end, backtrack]
c (iptr) ad	c (optr) Ad	[Rule 5, Try $A \rightarrow a$]
c (iptr) ad	c (optr) a d	[Rule 3, Match a]
ca (iptr) d	ca (optr) d	[Rule 3, Match d]
cad (iptr)	cad (optr)	[Rule 6, Terminate with success.]

ement → c (iptr) ad

nters

backtrack

the non-terminal at the output

However;

[caa is not produced by this Grammar]

Left Recursion:

example: $A \rightarrow A\alpha$;

(same)

Eg: $E \rightarrow E+E \mid E*E \mid id$ (terminal)

Non-immediate left recursion:

$S \rightarrow Aab \mid c$ → $S \rightarrow Scab$

$A \rightarrow Sc \mid c$

Top-Down Parsing Method cannot handle Left-recursive Grammar as ↳

A()
{
A()
α
}
∞

① Removing Left Recursion:

$A \rightarrow A\alpha \mid \beta$

⇓

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

Then; we have:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \dots \mid \beta_m A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \dots \alpha_n A' \mid \varepsilon$

(where β doesnot start with Non-terminal symbol)

[General Form:]

$E \rightarrow E+E$ | id

$E \rightarrow E*E$

$E \rightarrow id$

$E \rightarrow idE'$

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

$E \rightarrow idE'$

$E' \rightarrow *EE' \mid \varepsilon$

$E \rightarrow id E'$

$E' \rightarrow +E E' / \varepsilon$

$E \rightarrow id E''$

$E'' \rightarrow *E E'' / \varepsilon$

$E \rightarrow id$

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

2) Non- Recursive Predictive Parsing (Table Driven Pars

Consider the Grammar:

$S \rightarrow aBa$

$B \rightarrow bB / \varepsilon$

Parsing Table is:

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

Table :- LL(1) Parsing Table

Given String $w = abba$

Stack	Input	Output
$ S	abba$ (sentence to be parsed)	S → aBa (1 2 3)
$ a B a	abba$	B → bB
$ a B	bba$	
$ a B b	bba$	
$ a B	ba$	B → bB (1 2)
$ a B b	ba$ (Pop b's)	
$ a B	a$	B → ε
$ a	a$	
$	$	Accept, and successful completion

Note: base of the Stack

'S' on small 'a' [From the table]

Defn / Theory:

Input [ ] [$]

[P. T. O]

OR //

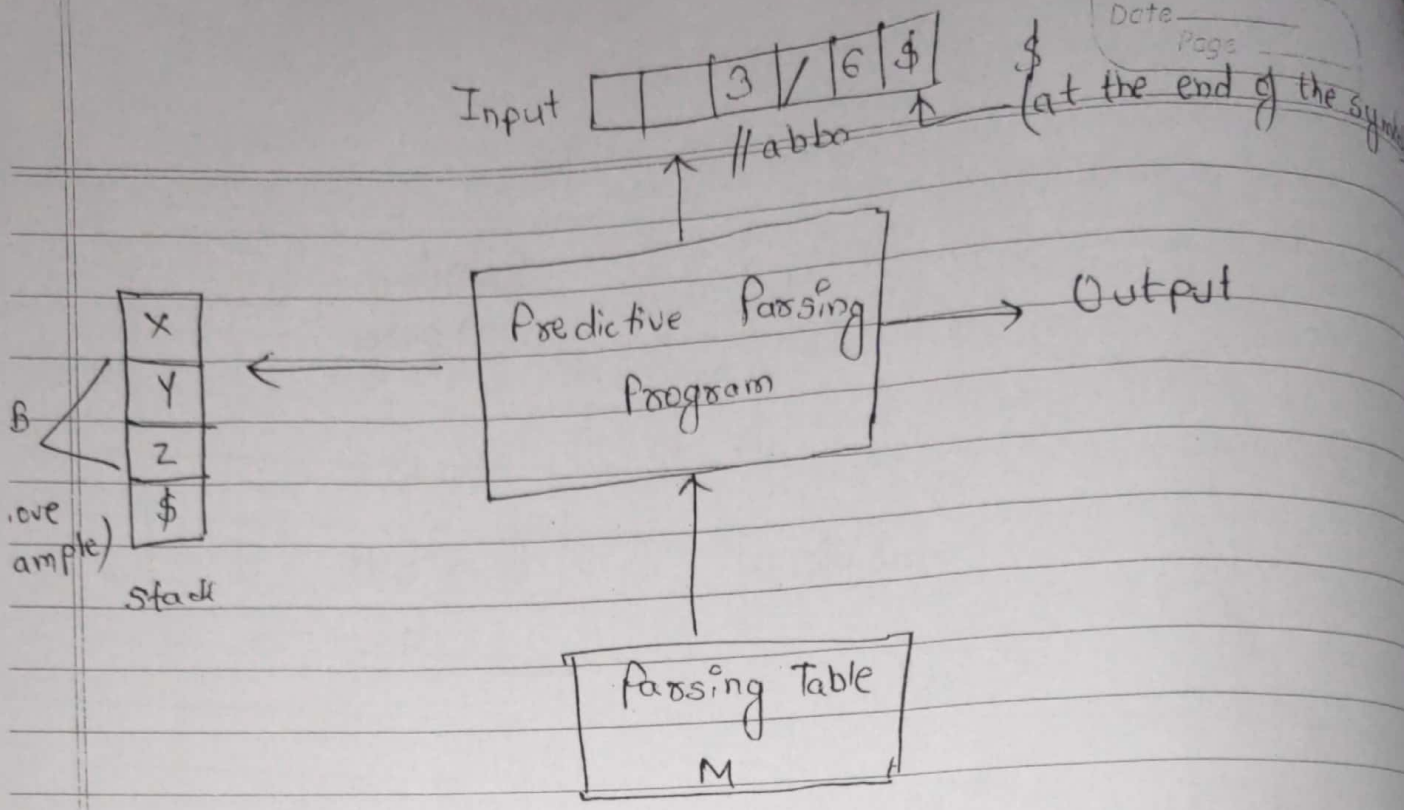


Fig: Non-recursive Predictive Parsing

Example

$R \rightarrow id\,S \mid (R)S$ (id S) or ((R)S)

$S \rightarrow +RS \mid .RS \mid *S \mid \varepsilon$ $\{+, ., *, \varepsilon\}$

$FIRST(R) = \{id, (\}$

$FIRST(S) = \{+, ., *, \varepsilon\}$

(start state ... $ 'dollor symbol' huxa gale

↳ A follows B

S → ABCDE
A → a | ε
B → b | ε
C → c
D → d | ε
E → e | ε

FIRST → {a, b, c}

A	{a, ε}
B	{b, ε}
C	{c}
D	{d, ε}
E	{e, ε}

FOLLOW

S	{$}
A	{b, c}
B	{c}
C	{d, e, $}
D	{e, $}
E	{$}

[never consists ε]

↳ FIRST(α) = { the set of terminals that begin all strings

Follow of B

S → Bb | Cd
B → aB | ε
C → cC | ε

[No capital 'S' ; other places]

S	{a, b, c, d}
B	{a, ε}
C	{c, ε}

FOLLOW

S	{$}
B	{b}
C	{d}

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → id | (E)

E	{id, (}
E'	{+, ε}
T	{id, (}
T'	{*, ε}
F	{id, (}

FOLLOW

E	{$,)}
E'	{$,)}
T	{+, $,)}
T'	{+, $,)}
F	{*, +, $,)}

Scanned with CamScanner

	FIRST	FOLLOW
$S \to A\ C\ B \mid CbB \mid Ba$	$\{d, g, h, \varepsilon, b, a\}$	$\{\$\}$
$A \to da \mid BC$	$\{d, g, h, \varepsilon\}$	$\{h, g, \$\}$
$B \to g \mid \varepsilon$	$\{g, \varepsilon\}$	$\{\$, a, h, g\}$
$C \to h \mid \varepsilon$	$\{h, \varepsilon\}$	$\{g, \$, a, b\}$

(follow of A)

↳ FOLLOW(A) = { the set of terminals that can immediately follow non-terminal A. }

$B \to \alpha A \beta$ → If $\beta = \varepsilon$ ∅

otherwise; then, Follow of A = follow of B

everything in First(β) except ε is placed in Follow(A)

	FIRST	FOLLOW
$S \to A\ B$	$\{b, a, c\}$	$\{\$\}$
$A \to Ca \mid \varepsilon$	$\{b, \varepsilon\}$	$\{c, b, \$, a\}$
$B \to BaAC \mid c$	$\{c\}$	$\{\$, a\}$
$C \to b \mid \varepsilon$	$\{b, \varepsilon\}$	$\{\$, a\}$

After removing Left Recursion:

So; the new Grammar is

~~S → AB~~

follow of A is first of B

		FIRST	FOLLOW
P_1	$S \to AB$	{b, a, c}	{$}
P_2 P_3	$A \to Ca \mid \varepsilon$	{a, ε} (for ε follow)	{c, b, a, $}
P_4	$B \to cB'$	{c}	{$}
P_5 P_6	$B' \to aAB' \mid \varepsilon$	{a, ε}	{$}
P_7 P_8	$C \to b \mid \varepsilon$	{b, ε}	{a, $}

follow

Now; Parsing Table (consists of variables and terminals)

	a	b	c	$
S	S→AB (1)	S→AB (1)	S→AB (1)	
A	(3) (2)	(2) (3)	(3)	(3)
B			(4)	
B'	(5)	(7)		(6) (8)
C	(8)			

एक होगा

Double Entry; meaning the Grammar is not L.L.1 #

came from Ca

	a
A	A → α

M[A, a]

S → aSbS / bSaS / ε

{a}	{$, b, a}
{b}	
{ε}	

Passing Table:

Variables / N.T ↓	Terminals → a	b	$
S	(1) S → aSbS, S → ε	(2) S → bSaS, S → ε	S → ε

Not L.L.(1)

S → aABb
A → c / ε
B → d / ε

	FIRST	FOLLOW
S	{a}	{$}
A	{c, ε}	{d, b}
B	{d, ε}	{b}

Passing Table:

	a	b	c	d	$
S	S → aABb				
A			A → ε	A → c	A → ε
B			B → ε	B → d	

It is L.L.(1) grammar

	FIRST	FOLLOW
↳ #S → aB | ε	{a, ε}	{$}
B → b C | ε	{b, ε}	{$}
C → c S | ε	{c, ε}	{$}

Passing Table:

L.L.(1)

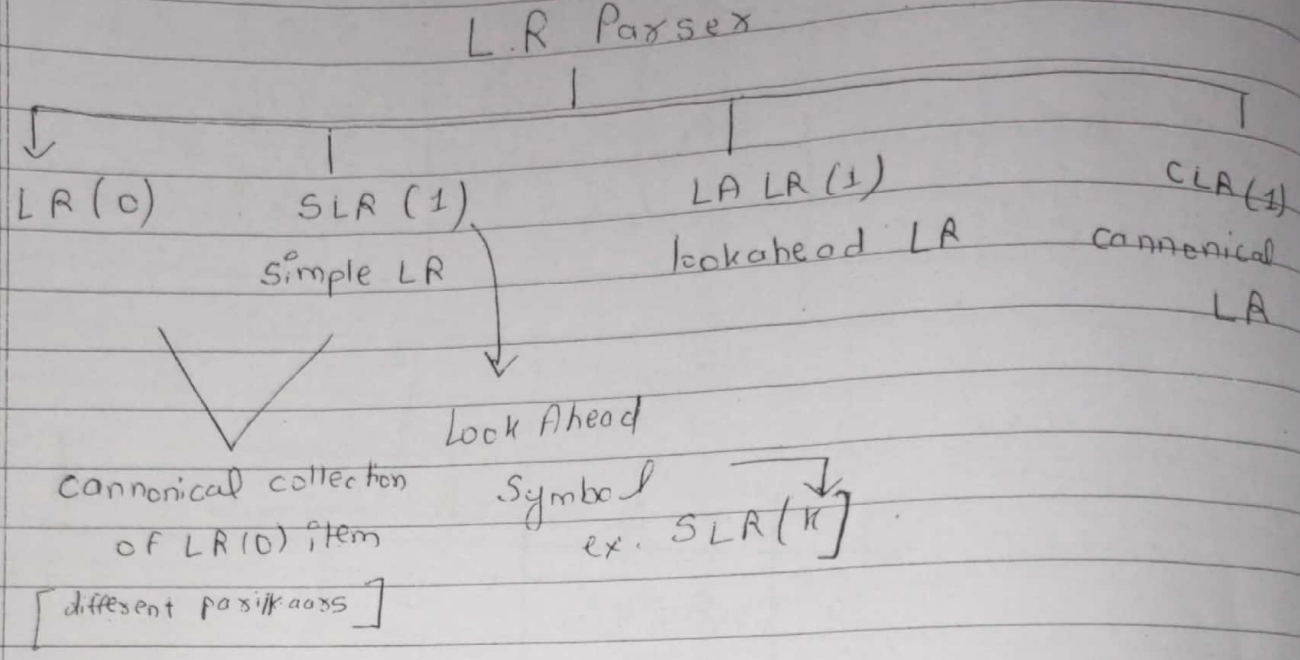
	a	b	c	$
S	S → aB			S → ε
B		B → b C		B → ε
C			C → c S	C → ε

	FIRST	FOLLOW
↳ #S → aSA | ε	{a, ε}	{$, c}
A → c | ε	{c, ε}	{$}

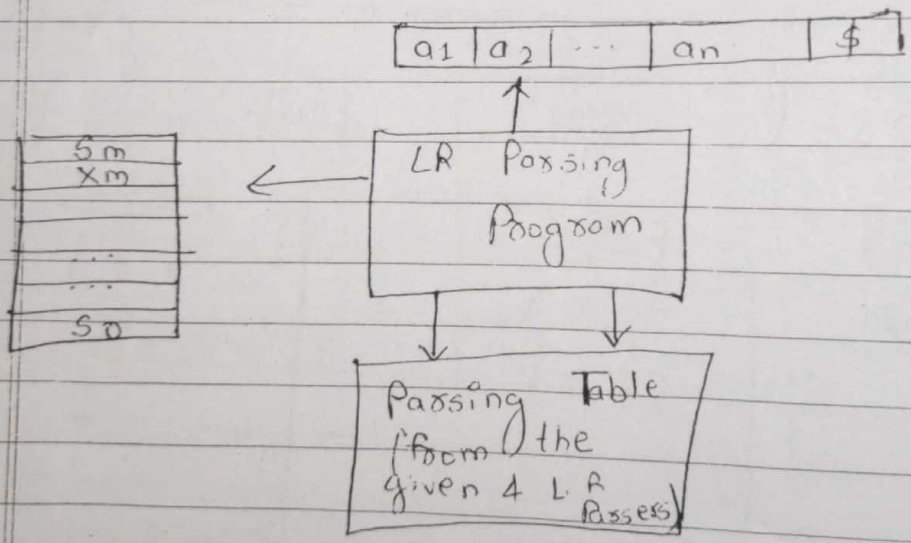
Passing Table:

	a	c	$
S	S → aSA	S → ε	S → ε
A		A → c A → ε	A → ε

[Bottom-up Parsing]



Structure of L.R Parsers



↳ For the construction of LR(0) and SLR(1); we have to construct the cannonical collection of L.R(0) item.

From start string to

derive (Terminal) reduce

or

[vice - versa]

$S \to .abAA$

$S \to a.bAA$

$S \to ab.AA$

$S \to abA.A$

$S \to abAA.$ → final item

item

non-final item

[If there is dot (.) on the Rightmost side]

#Reduction : abAA can be replaced by S

as $S \to abAA$

Leftmost side — replaced by S

Consider a grammar :

$S \to AA$

$A \to aA \mid b$

Augmented the given grammar :

$S' \to S$

$S \to AA$

$A \to aA \mid b$

DFA

closure (Closure)

& goto

$S' \to .S$

$S \to .AA$

$A \to .aA$

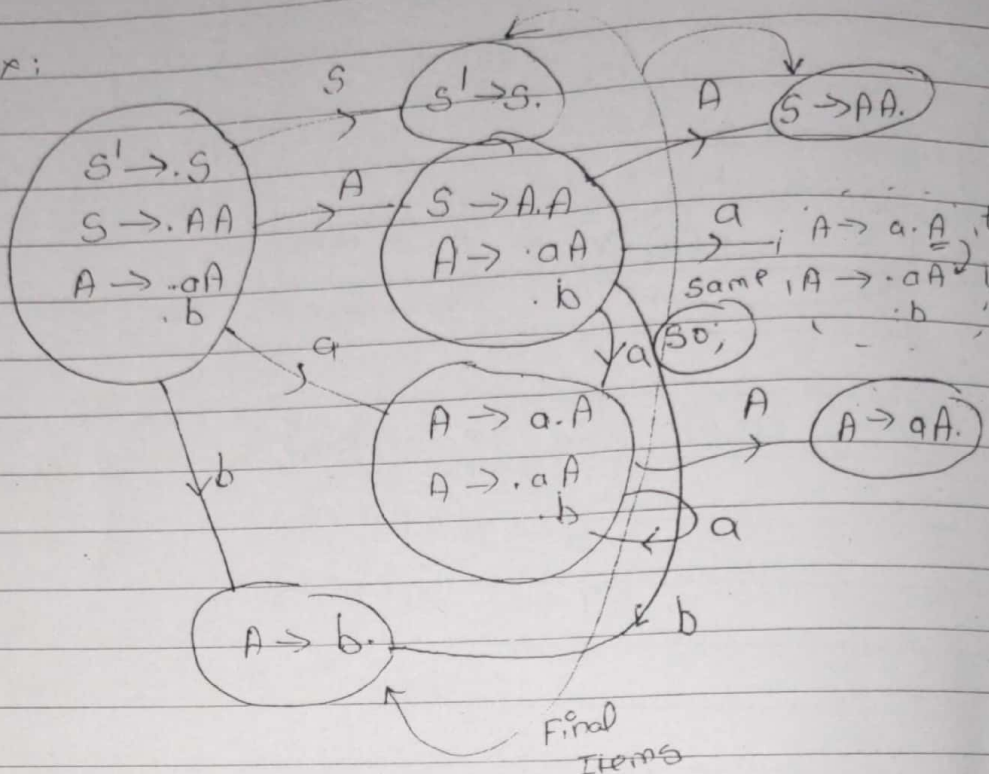
$.b$

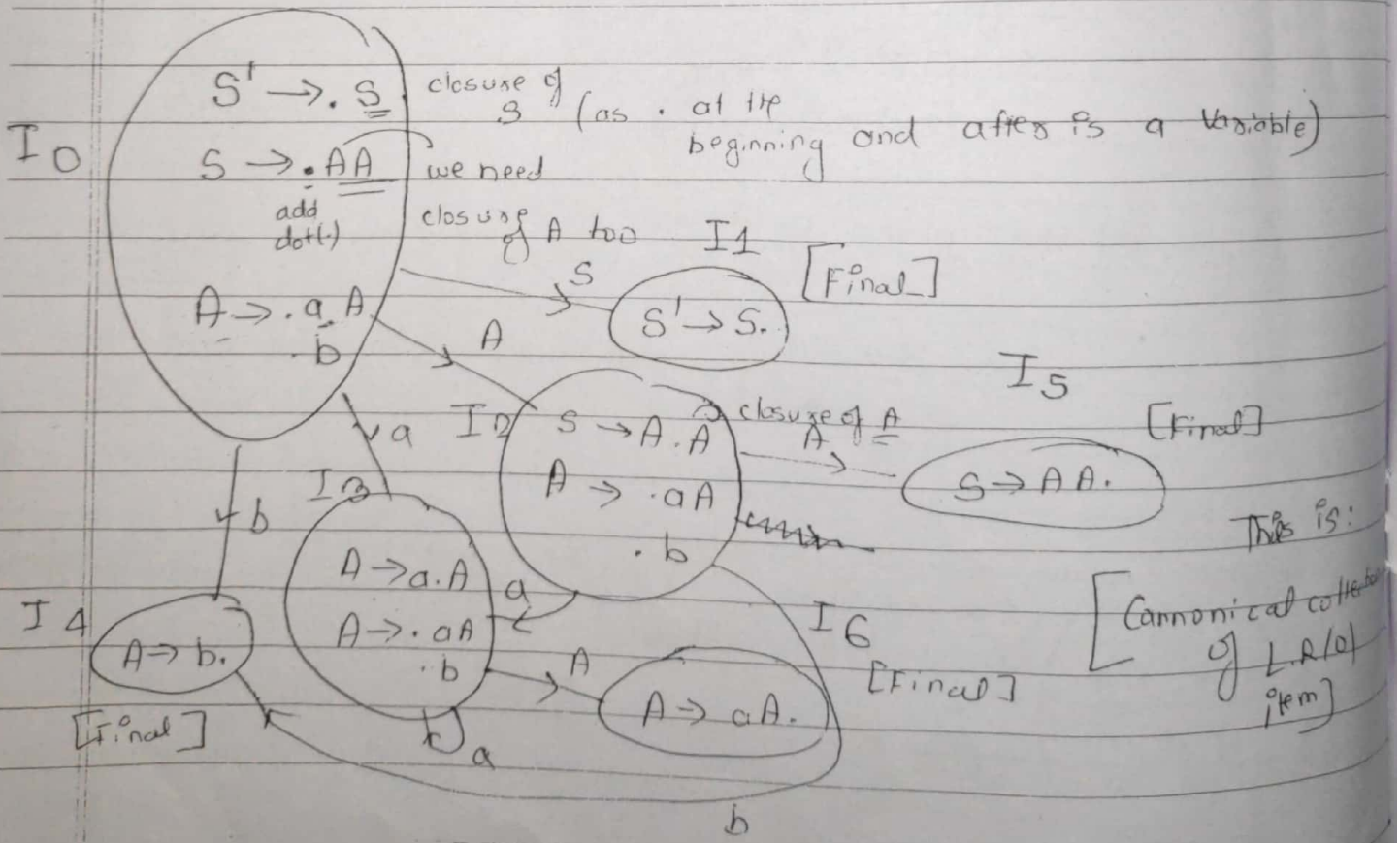
State 0

(Closure of S : Variable after dot (.) nothing but (S) with dot at the beginning of production of S)

goto (go from one state to beginning)

For ex:





↳ Parsing Table consists of 2 parts: → Action Part
→ Goto Part

Construction of Parsing Table → consists of terminal symbols [Ter. a, b, $]

Goto → consists of Variables [Non-Terminal]

[no need to write $]

	Action			Goto	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

[naming conventions]

Action Part (Final) → terminals:

we use any number

Accept at $S' \to S.$

[start state]

reduction

Note $S \to aaB$

Final item at last:

$S \to aaB.$ ready for reduction

we have;

$S' \to S$

$S \to AA$ (r1)

$A \to aA$ (r2) | b = (r3)

reduction numbers i.e (r_x)

we have;

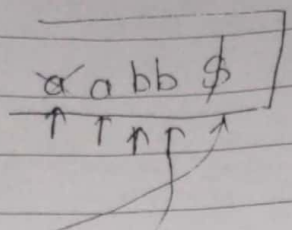
$S \to AA$ gives aabb

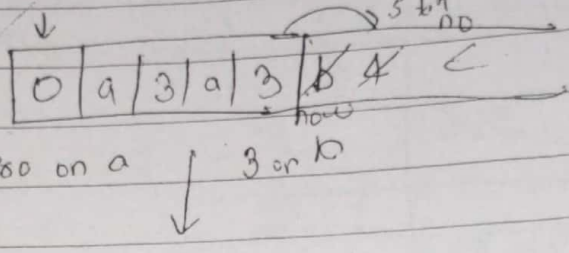
$\to aAA$

$\to aaAA$

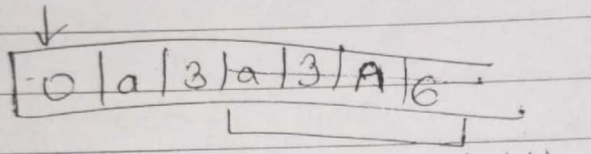
$\to aabA$

$\to aabb$





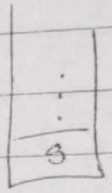
we have a1 a2 $

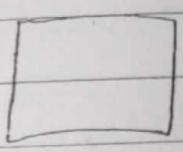


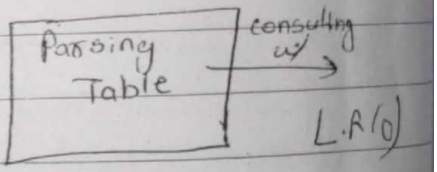
replace 4 of these

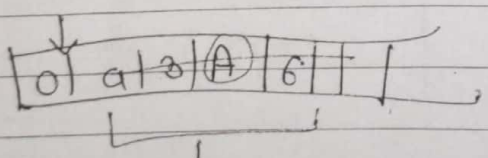
A → aA (2)

by A #



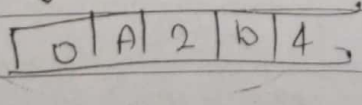


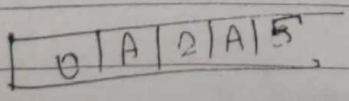


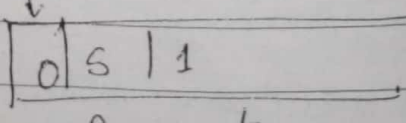


b on 6 ↦ 2

[Store 4 on $]

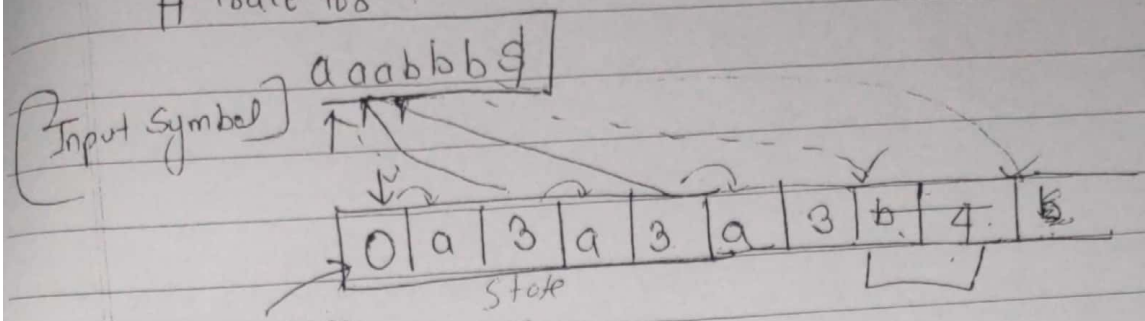




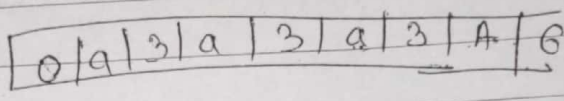


Accept

Trace for :



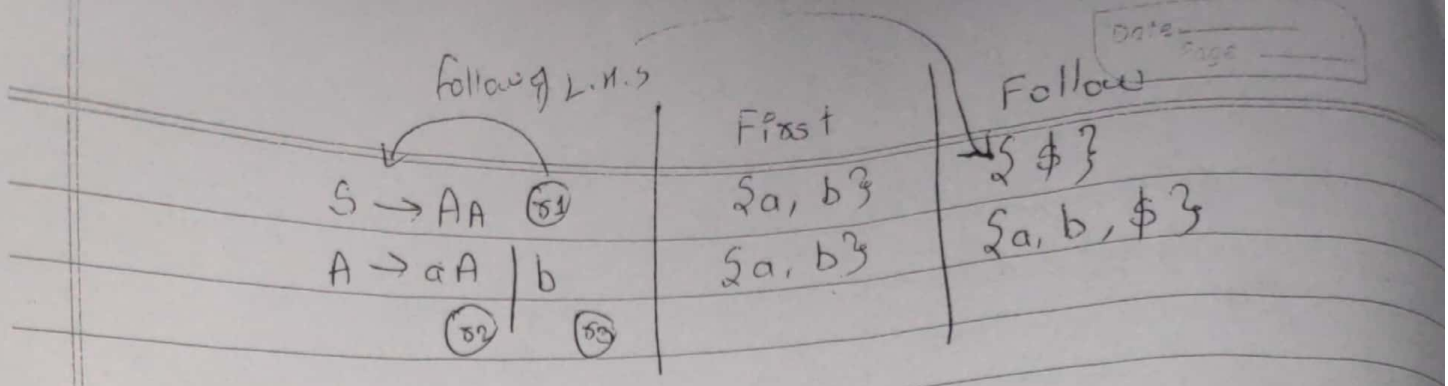
Now;



SLR (1) Parsing table

	Action			Goto	
	a	b	$	S	A
0	S_3	S_3		1	2
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	γ_3	γ_3	γ_3		
5			γ_1		
6	γ_2	γ_2	γ_2		

(Hijo entire row γ_1 लेख्यारयौ।)



Note: In SLR(1) Parsing Table; reduction (r) is only done for Follow of Left Hand Side (L.H.S).

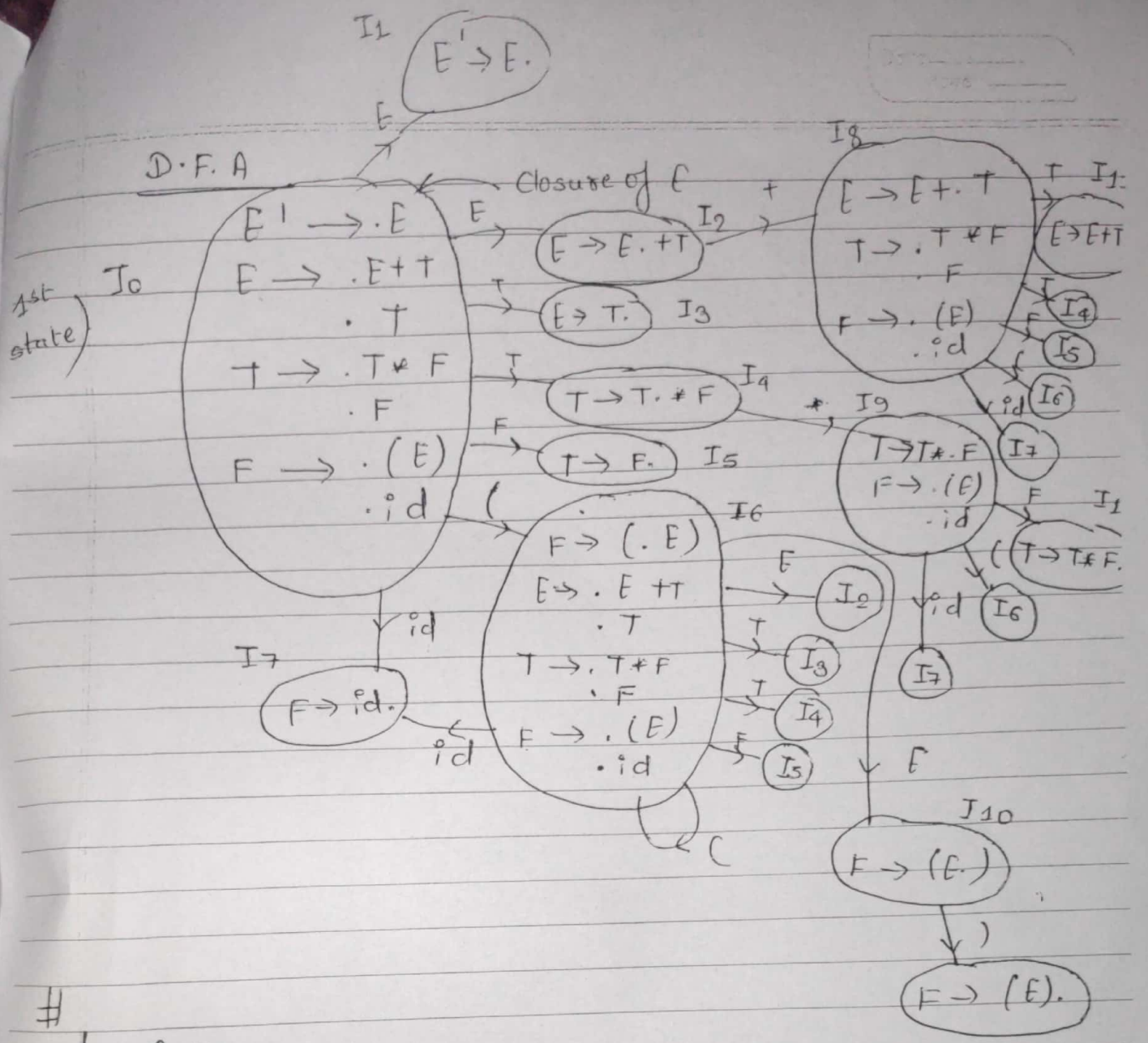
↳ Everything is same for L.R(0) and SLR(1) Parsing Table; except for (rx) / reduction
↳ follow of L.H.S //

Q. Construct L.R(0) and S.L.R(1) Parsing Table for the Grammar given below:

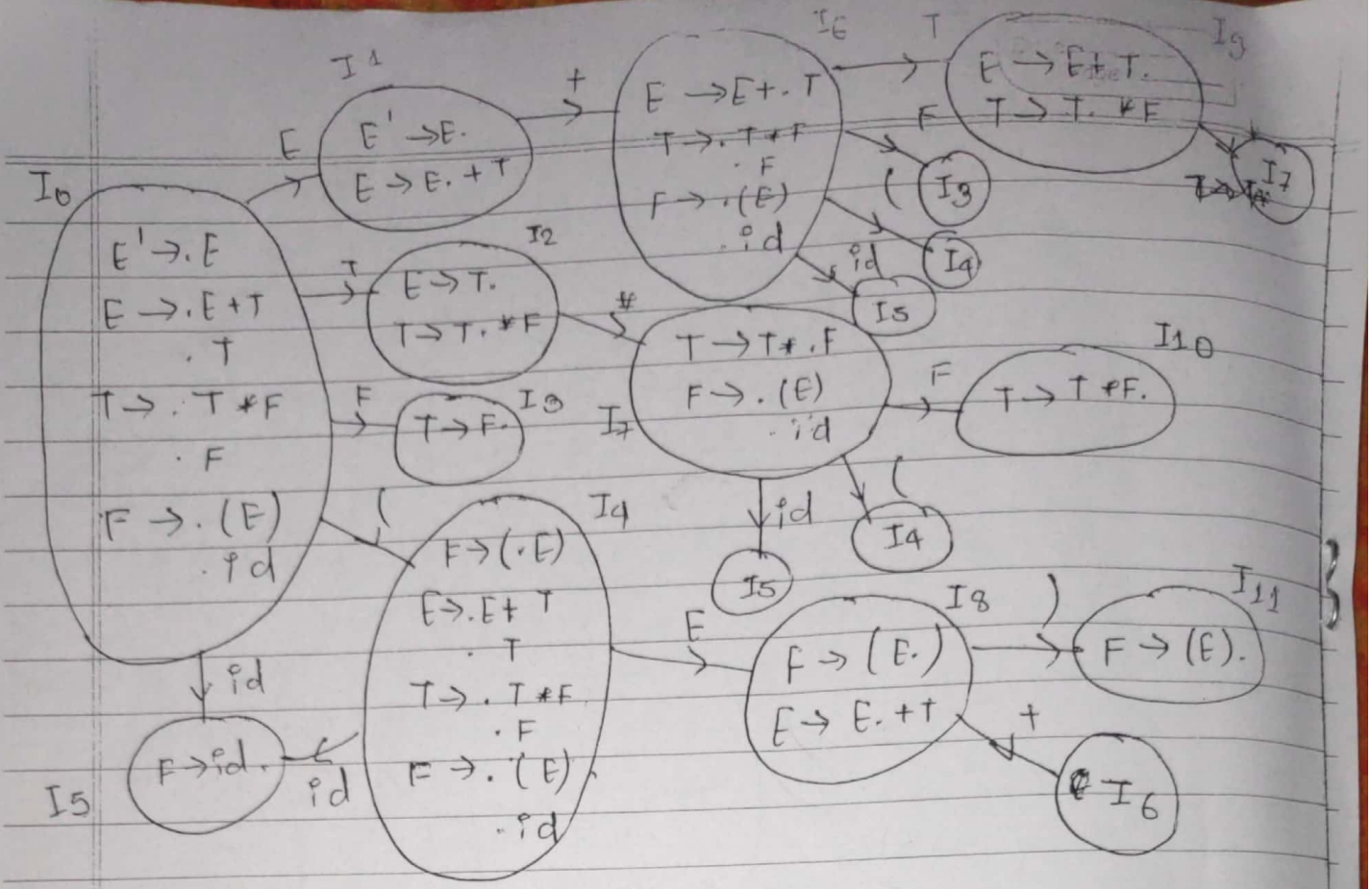
$$E \longrightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

Soln; Augmenting the given grammar; we have:

$$E' \rightarrow E$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$



#	Action						GOTO			
	id	+	*	(	)	$	E	T	F	(don't write E' here)
0	S7			S6						
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										
13										



State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6							
2			S7						
3									
4							8	2	3
5									
6									
7									
8									
9									
10									
11									

Final Item