FRANKLIN'S LECTURES

COMPILER DESIGN

Module 2 Part 2

CST302

Q. Define LL (1) grammar.

A grammar whose parsing table has no multiply defined entries is Said to be LL (1). To become a grammar LL (1) following conditions are to be satisfied.

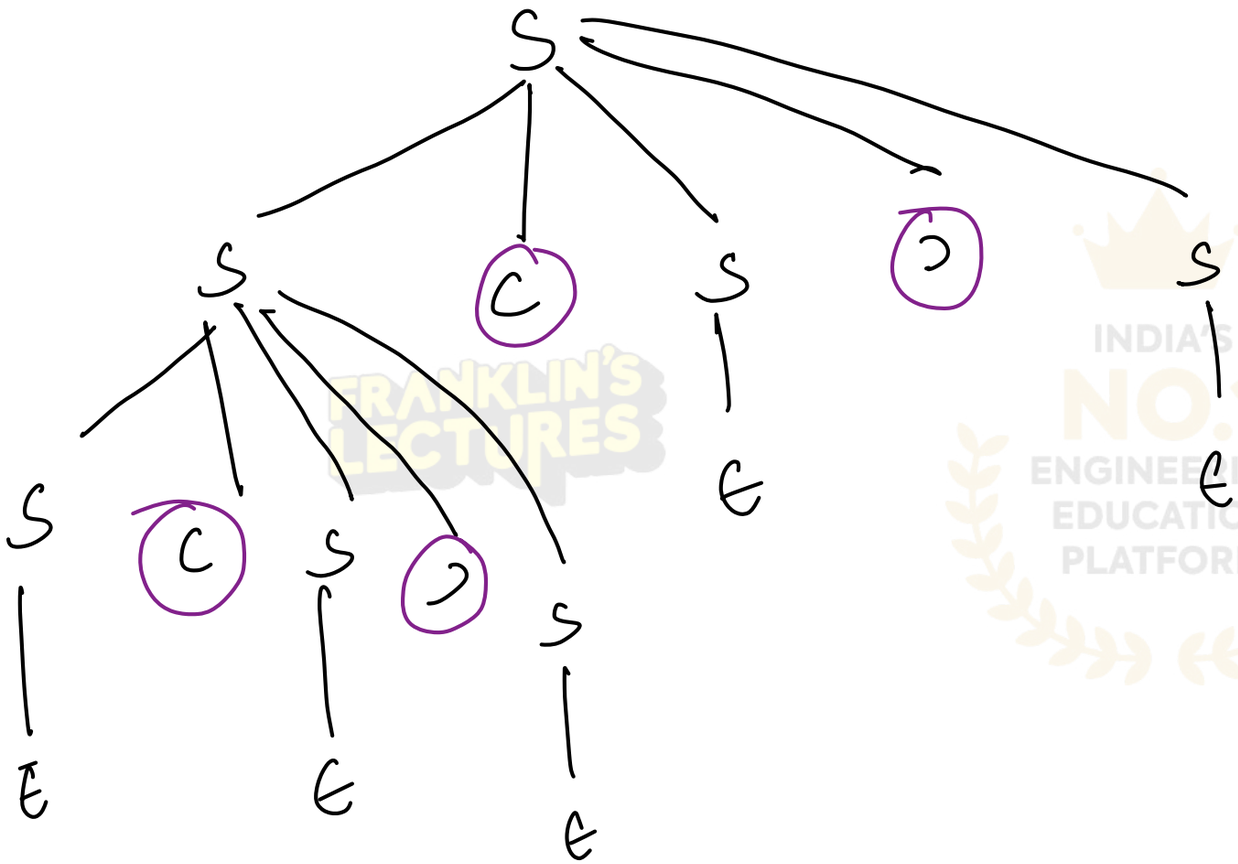
- The grammar should not be ambigous
- The grammar should not be left recursive
- The grammar should be deterministic
- The parsing table should not contain multiply defined entries.

Q. Is the grammar S→S (S)S/∈ ambigous? Justify your answer.

A grammar is said to ambigous if there exist a string ‘w’ in that language which can produce more than one parse tree or more than one leftmost derivation or more than one right most derivations.

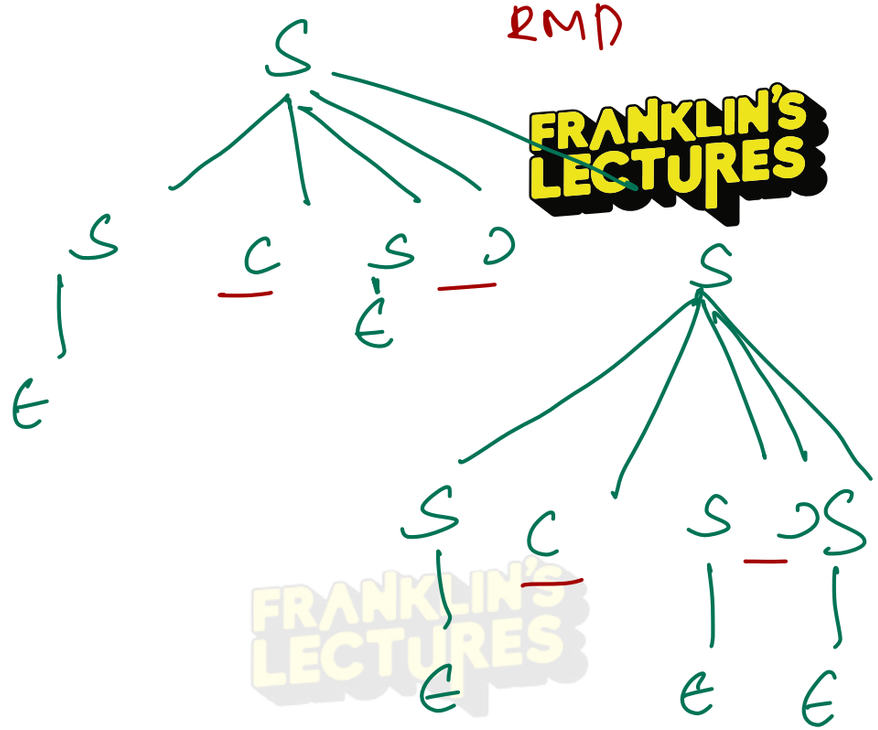
$S \rightarrow S(S)S \mid \epsilon$

LMD



(())(())

RMD

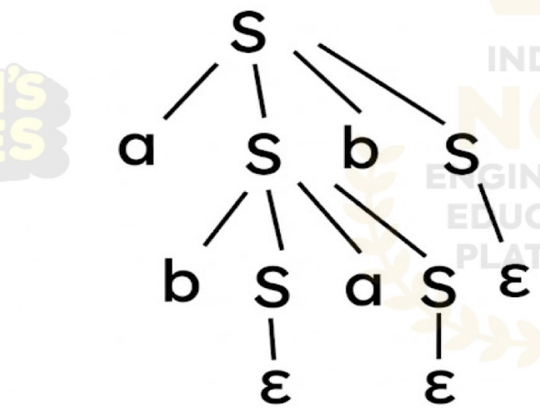


(())(())

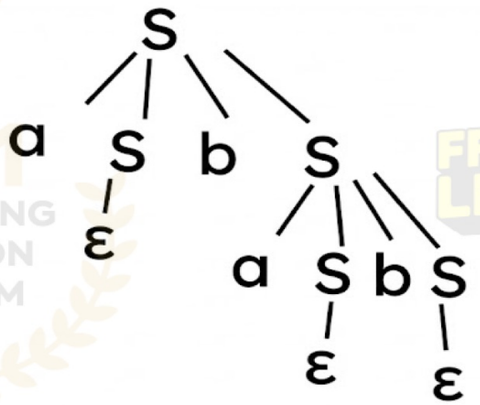
- Here we can generate two different parse trees for the string ()(). so the gives grammar is ambigous

Q. Consider the context free grammar

- S → asbs/bSas/∈. Check whether the grammar is ambigous or not.
  S → aSbs | bSaS | ∈
- **FIGURE**

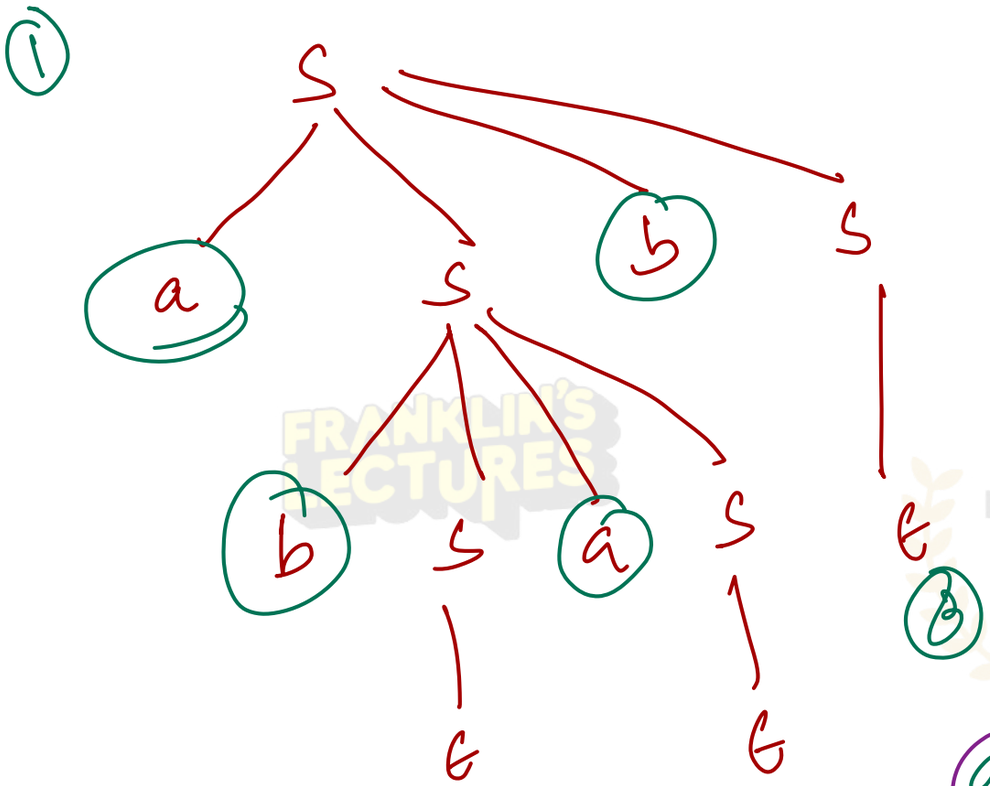


⇒ abab.

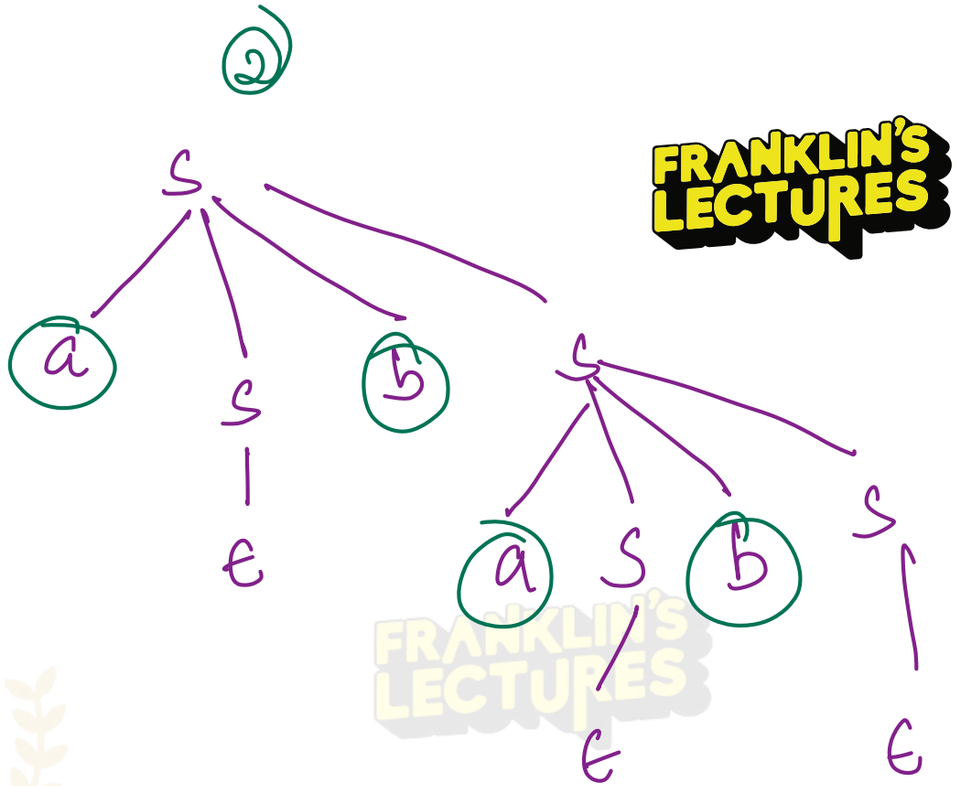


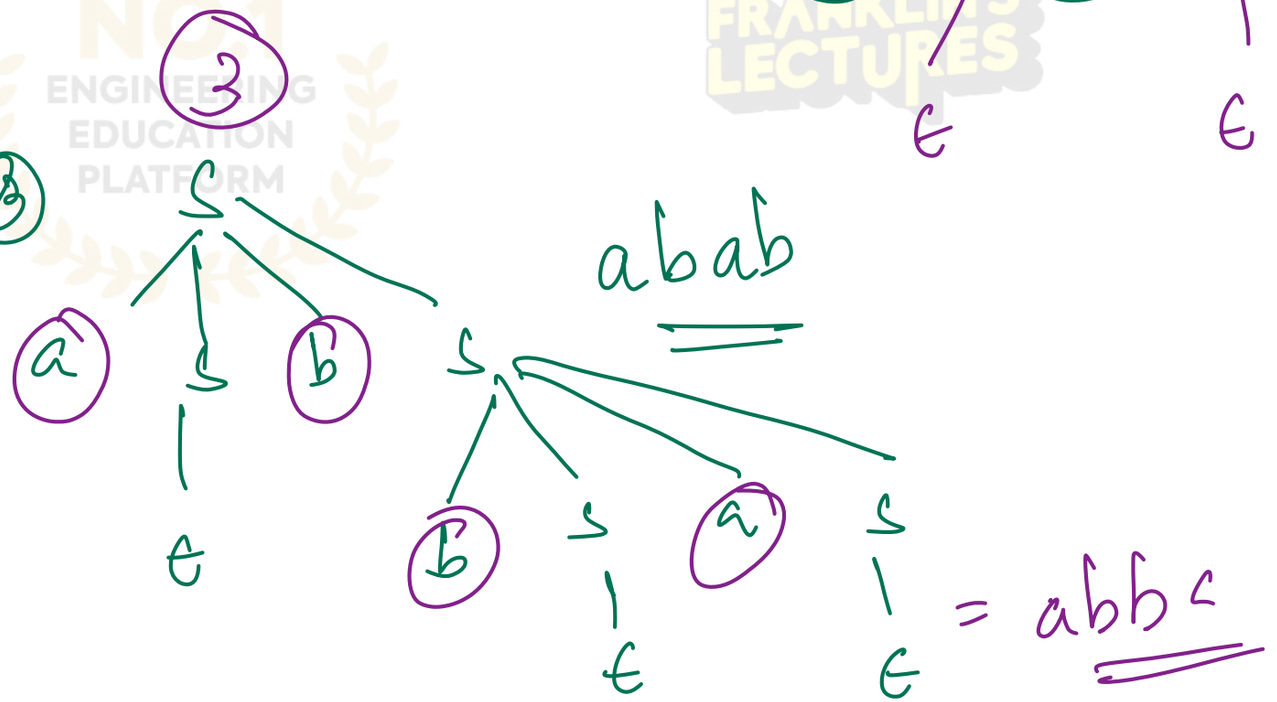
⇒ abab.

$S \rightarrow aSbS \mid bSaS \mid \epsilon$



abab





abab

= abbε

- Here we can generate two different parse trees for the string abab. Hence the given grammar is ambigous.

Q. What is recursive descend parsing? List the problems faced is designing such a parser.

Recursive descend parsing is a top down parsing that finds a leftmost derivation from the input string. It constructs a parse tree for the input String starting from the root and creating the nodes of parse tree in pre order. Recursive descend parsing is the general form of top down parsing that involve back tracking, ie, making repeated scans of the input.

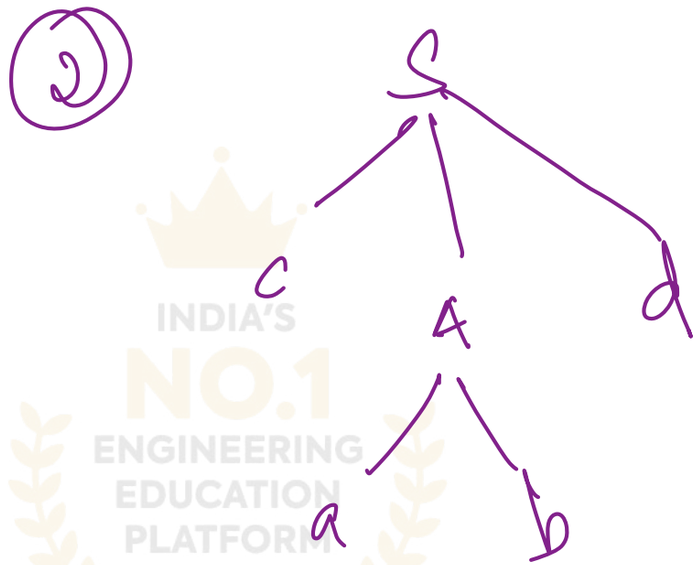
problems in recursive descent parsing

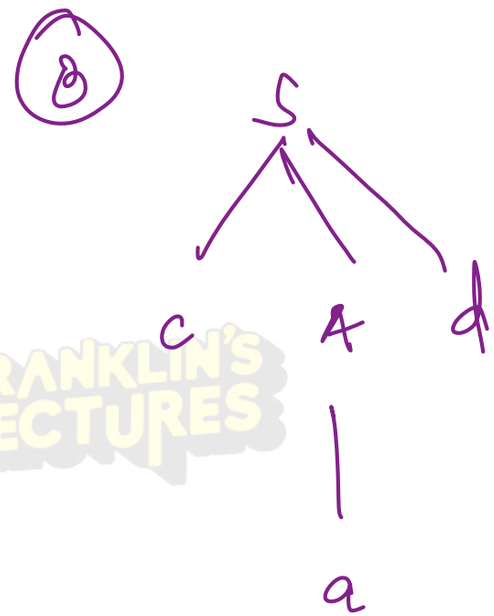
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

$w = cad$

×





Problems :-

1) Left recursion: If a grammar contains left recursion. Recursive descent passer does not work. A grammar is left recursive if I contains production of the forms $A \rightarrow A\alpha/\beta$.
2) Back tracking: It occurs when there are more than one alternative in the productions while parsing the input String.
3) It is very difficult to identify the position of the errors.

Q. Differentiates left most derivation & right most derivation. Show an example for each.

Leftmost derivation :- In this at each Step of derivation we replace left most variable by one of its rules.

eg:- consider the grammar

S → aSbS | a leftmost derivation for the string aaba is

S → aSbS → aabS → aaba

→ aabS

→ aaba

Right most derivation: - In this at each step of derivation we replace right most variable by one of its production rules by right most derivation for the string aaba is

$S \rightarrow aSbS$

$S \rightarrow aSba$

$\rightarrow aaba$.

$S \rightarrow aSbS$

$\rightarrow aSb$

$\rightarrow aaba$

Q. Find out context free of language for the grammar given below.

S → abB

A → aaBb/∈.

B → bbAa.

1)S→abB

abbbAa

abbbba

① $S \rightarrow abB \rightarrow abbbAa \rightarrow abbbba$

② $S \rightarrow abB \rightarrow abbbAa \rightarrow abbbaaBbba$

2) S → abB

→ abbbAa.

→ abbbaaBba

→ abbbaabbAaba

→ abbbaabbaba

3)S→ab<u>B</u>.

abbb<u>A</u>a

aabbaa<u>B</u>ba.

aabbaabb<u>A</u>aba.

abbbaabbaa<u>B</u>baba.

abbbaabbaa<u>B</u>baba.

abbbaabbaabb<u>A</u>ababa

abbbaabbaabbbababa

Language is L(G) = {ab $(bbaa)^n$ bba $(ba)^n$;≥0}

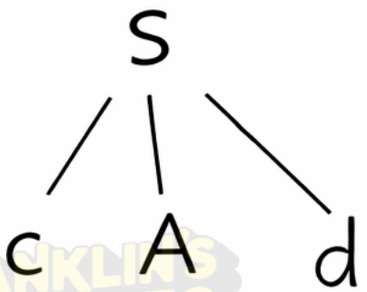
Q. For what type of grammar, recursive descent parser cannot be constructed? Show the steps involved in recursive descent Parsing with backtracking for the string cad with the given grammar:
S → cAd A → ab|a.

Recursive descent parser cannot be constructed for the following type of grammars

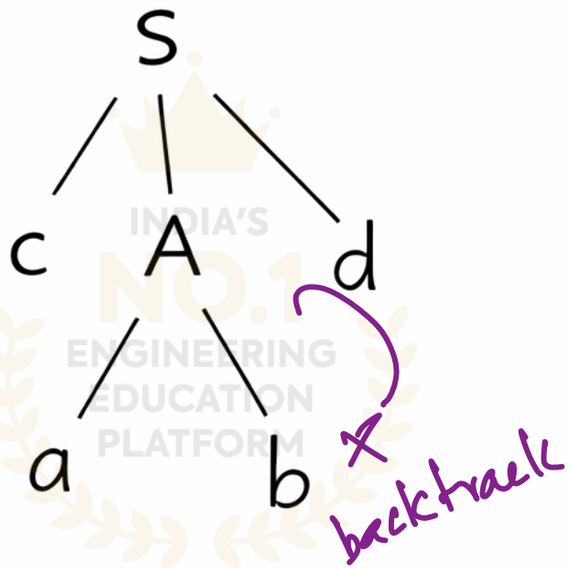
1) Left recursive grammar
2) Ambigous grammar.
3) Non-left factored grammar.

steps involved in recursive descent parsing

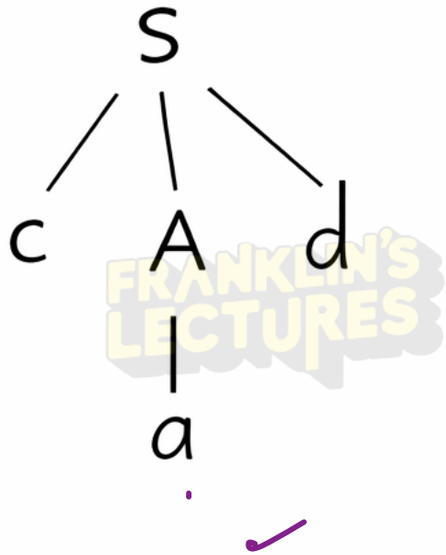
Step 1:



Step 2:



Step 3:



Step 1:

Step 2:

Step 3:

Q. Eliminate the ambiguity from the given grammar.

E→E*E | E-E | E^E | E|E | E+E | (E) | id

The associativity of the grammar is as given below. The operators are listed in the decreasing order of precedence.

(i) ( )

(i) | and + are right associative

(iii)'^' is left associative

(iv) * and - are left associative.

E→ E*P|E-P|P.

P→ P^T|T

T→ F+T|F|T|F

F→(E) |id.

E → E* E | E - E

E → E* P | E - P | P

P → P ^ T | T

T → F + T | F / T | F

F → (E) | id

P

↑ ↓.

E ^ E

E * E

E * P

E * E

P * E

Q. Consider the following grammar.

E → E or T|T

T → T and F|F

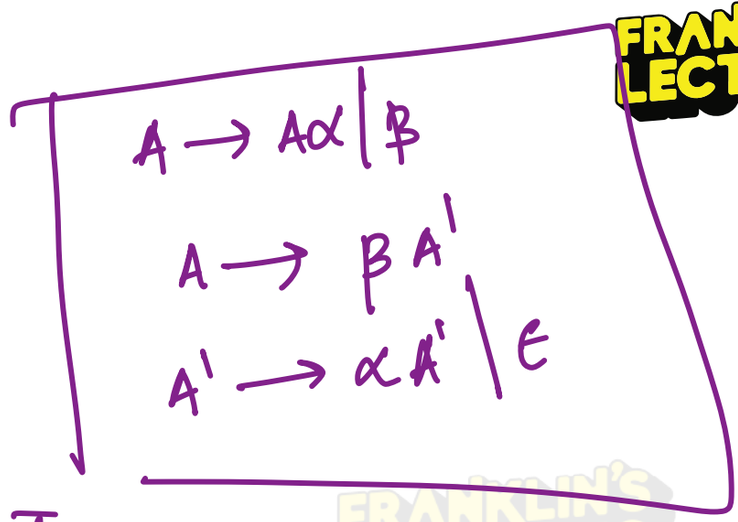
F → not F (E) |true|false.

(i) Remove left recursion from the grammar

(ii) Construct predictive parsing table

(iii) Justify the statement " The grammar is LL(1)".

$A \rightarrow A\alpha_1 | A\alpha_2$



(i)Remove left recursion.

E→E or T|T and

T→T and F|F are left recursive.

Grammar after left recursion elimination is,

E→ TE′

E′ → or TE′|Є

T → FT′

T′ → and FT′|Є

F→ not F|(E) | true|false

$E \rightarrow E\alpha T | T$ (α = or T, β = T)

$E \rightarrow TE'$ $\quad E' \rightarrow \text{or } TE' | \epsilon$

$T \rightarrow T \text{ and } F | F$ (α = and F, β = F)

$T \rightarrow FT'$ $\quad T' \rightarrow \text{and } FT' | \epsilon$

(ii)Compute FIRST and FOLLOW and then construct predictive parsing table

FIRST (E) = FIRST (T) = FIRST (F) = { not, (, true, false}

FIRST (E′) = {or, Є}

FIRST (T′) = {and, Є}

FOLLOW (E) ={$,)}

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

	E	E'	T	T'	F
FIRST	{not, (, true, false}	{or, Є}	{not, (, true, false}	{and, Є}	{not, (, true, false}
FOLLOW	{), $}	{$,)}	{or,), $}	{or,), $}	{and, or,), $}

FOLLOW(T) = FIRST(E′) ={or, $,)}

FOLLOW (T′) = FOLLOW(T) = {or, $,)}

FOLLOW (F) = FIRST (T′) = {and, or,$,)}

Predictive parsing table

	or	and	not	(	)	true	false	$
E			$E \rightarrow T\ E'$	$E \rightarrow T\ E'$		$E \rightarrow T\ E'$	$E \rightarrow T\ E'$	
E′	$E' \rightarrow$ or T E′				$E' \rightarrow \epsilon$			$E' \rightarrow \epsilon$
T			$T \rightarrow F\ T'$	$T \rightarrow F\ T'$		$T \rightarrow F\ T'$	$T \rightarrow F\ T'$	
T′	$T' \rightarrow \epsilon$	$T' \rightarrow$ and F T′			$T' \rightarrow \epsilon$			$T' \rightarrow \epsilon$
F			$F \rightarrow$ not F	$F \rightarrow (\ E\ )$		$F \rightarrow$ true	$F \rightarrow$ false	

$E \rightarrow TE'$

$E' \rightarrow orTE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow andFT' \mid \epsilon$

$F \rightarrow notF \mid (E) \mid true \mid false.$

$F \rightarrow not\ F$

$F \rightarrow (E)$

$F \rightarrow true$

$F \rightarrow false$

iii) In the above predictive parsing table, there is no multiply defined entries & three conditions for LL(1) grammar holds. Hence the grammar is an LL(1) grammar.

Q. Find the FIRST & FOLLOW of the non-terminals in the grammar,

$S \rightarrow aABe$

$A \rightarrow Abc|b$

$B \rightarrow d$

ANS: FIRST (S) = {a}

FIRST(A) = {b}

FIRST (B)= {d}

FOLLOW(S) ={$}

FOLLOW (A) = {b, d}

FOLLOW (B) = {e}

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

FIRST(S) = {a}

FIRST(A) = {b}

FIRST(B) = {d}

FOLLOW(S) = {$}

FOLLOW(A) = {d, b}

FOLLOW(B) = {e}

Q. what is left recursive grammar? Give an example. What are steps in removing left recursion.

A grammar which contains the production of the form $A \rightarrow A\alpha|\beta$ are called left recursive grammar. Left recursion can be eliminated by the following production

$A \rightarrow \beta A'$

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

Eg:- consider the grammar.

$E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E)|id.$

$E \rightarrow TE'$ $\quad E' \rightarrow +TE'$

$T \rightarrow FT'$ $\quad T' \rightarrow *FT'$

This grammar is a left recursive grammar. Grammar after left recursion elimination is,

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)|id.$

Q. Given a grammar:

$S \rightarrow (L) | a$

$L \rightarrow L,S | S$

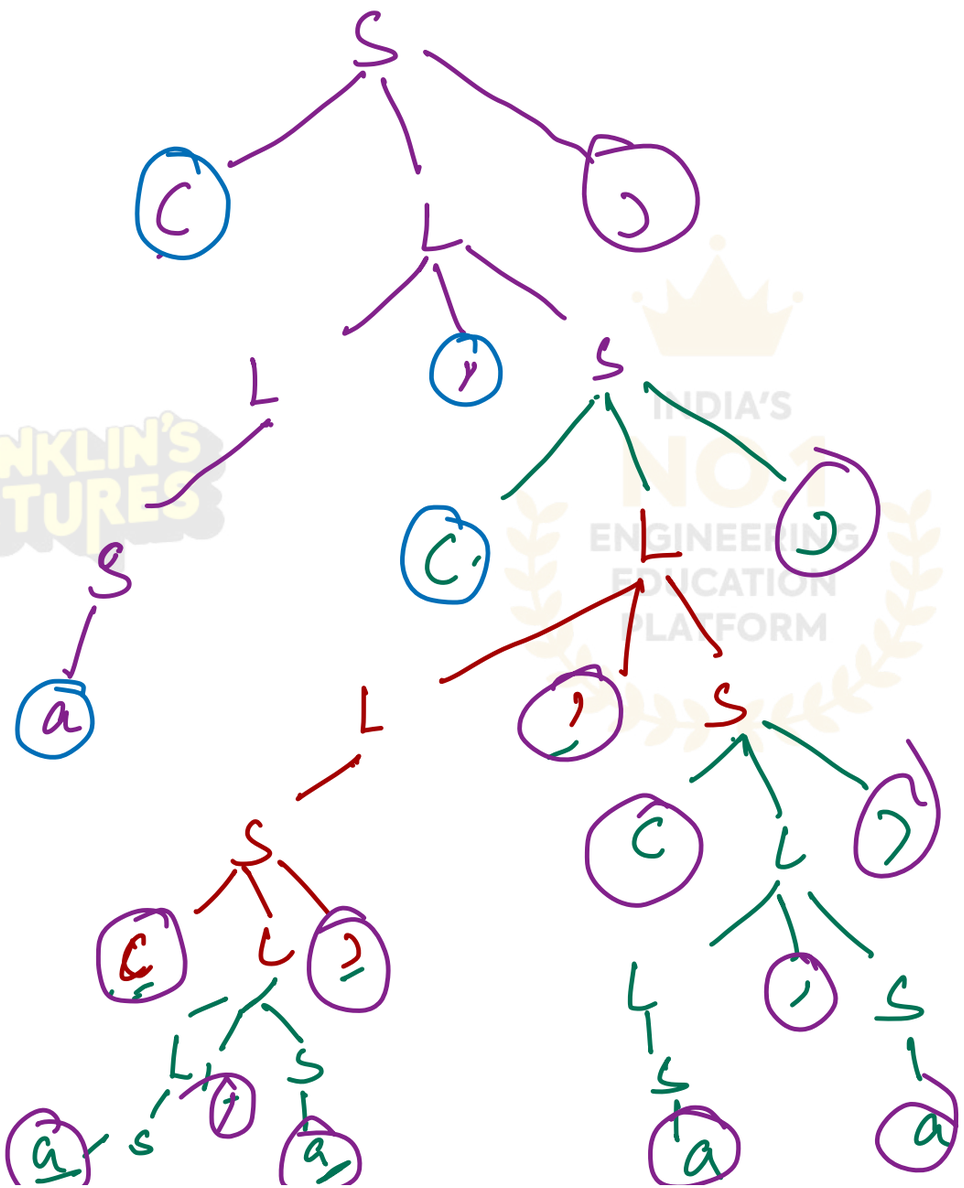
(i) Is the grammar ambigous? Justify.

(ii) Give the parse tree for the string (a, ((a,a), (a,a)))

A grammar is ambigous if there exist more than one parse tree for the string 'w'in that language.

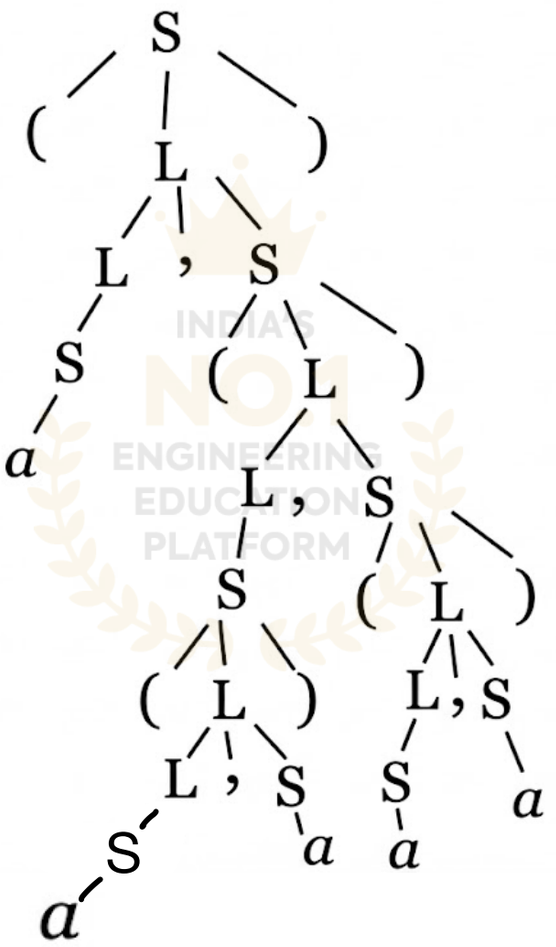
(i) The given grammar is not an ambigous grammar because there is no String 'w' in L which can produce more than one parse tree.

$S \rightarrow (L) \mid a \qquad L \rightarrow L,S \mid S$



$(a,((a,a),(a,a)))$

(ii)



Q. Construct predictive parsing table for the following grammar.

$S \rightarrow (L) | a$

$L \rightarrow L,S | S$

Step1: Remove left recursion

S→ (L)|a.

L→SL′

L→,SL′| Є

Step 2: Do left factoring. Here there is no need of left factoring.

Step3: Find FIRST and FOLLOW.

$L \rightarrow L,S \mid S$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \epsilon$

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

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

$S \rightarrow (L) \mid a$ $L \rightarrow SL'$ $L' \rightarrow ,SL' \mid \epsilon$

FOLLOW(S) = FIRST(L') ∪ {$}
= {, , $} ∪ FOLLOW(L)
= {, , $,)}

FIRST(S) ={ (, a}

FIRST (L) = FIRST(S) = {(,a}.

FIRST (L') = {, , Є}

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

FOLLOW (L) = {)}

FOLLOW (L') = FOLLOW (L) = {)}.

	S	L	L'
FIRST	{(, a}	{(, a}	{, , ϵ}
FOLLOW	{$, , ,)}	{)}	{)}.

- Step 4: construct predictive parsing table

Non terminals	(	)	,	a	$
L′		L′ → ϵ	L′ → , S L′		
L	L → S L′			L → S L′	
S	S → (L)			S → a	

Ip (a, a, a) $ — moves.

STACK	INPUT	OUTPUT
$S	(a, a, a) $	
$)L(	(a, a, a) $	S → (L)
$)L'	a, a, a) $	L → SL'
$)L'a	a, a, a) $	S → a
$)L'	, a, a) $	L' → ,SL'
$)L'S,	, a, a) $	
$)L'a	a, a) $	S → a
$)L'	, a) $	
$)L'S,	, a) $	L' → ,SL'

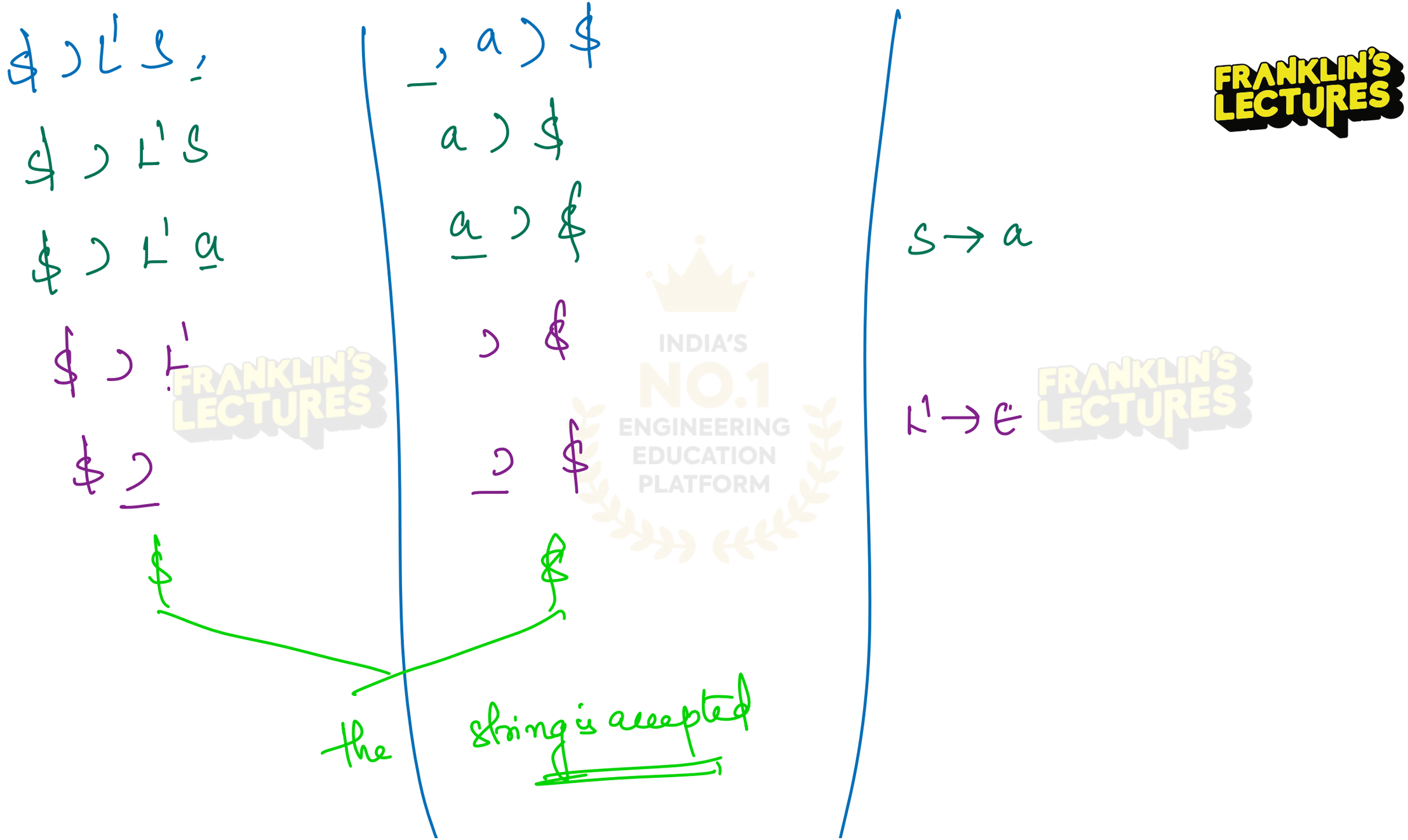
S → (L) | a

L → SL'

L' → ,SL' | ∈

Y1 Y2 Y3

Y3 Y2 Y1


$ ⟩ L' $;
$ ⟩ L' $
$ ⟩ L' a
$ ⟩ L'
$ ⟩
_ , a) $
a) $
a) $
) $
) $
S → a
L' → ∈
the
string is accepted

Q. Compute FIRST and FOLLOW of the following grammar.

$S \rightarrow SS+|SS*|a$

FIRST(S)

FIRST (S) = {a}

FOLLOW (S) ={$,+,*,a}

$S \rightarrow SS+ \mid SS* \mid a$

FIRST(S) = {a}

FOLLOW(S) = {$, +, *, a}

Q. Design a recursive descent parser for the grammar,

$S \rightarrow cAd$, $A \rightarrow ab|a$

construction of recursive descent Parser using procedures is given below:
(with backtracking)

$S \rightarrow cAd$

$A \rightarrow ab \mid b$

```
Procedure S()
{
   if input='c'
   {
     Advance()
     A();
     if input='d'
       {
         Advance();
      return true;}
      else
      return false;
```

$S \rightarrow cAd$

```
}
else
return false;
}
```

```
Procedure A()
{
i save=in-ptr;
if input='a'
{
  Advance();
if input = 'b'
{
```

$A \rightarrow ab$

$A \rightarrow ab$

$A \rightarrow a$

```
Advance();
 return true; }}
in-ptr=isave
if input = 'a'
{
Advance ();
return true;
}
return false;
}
```

A → a

Q. Design recursive descent parser for the following grammar. (without backtracking)

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

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

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T \rightarrow *FT'$

After elimination of left recursion is

- E →TE′
- E′→+TE′|Є
- T→FT′
- T′→*FT′|Є
- F→ (E)|id.

```
Procedure E()
{
 T();
  EPRIME();
}
procedureT()
{
```

E → T E′

Procedure E()
{
T();
EPRIME();
}

E′ → EPRIME()

T → FT′

procedure T
{
F();
TPRIME;
}

```
F();
TPRIME();
}
Procedure EPRIME()
{
if input="+"
{
Advance();
T();
EPRIME();
return true;
}
```

+TE'

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

```
Procedure EPRIME()
{
    if input = "+"
    {
        Advance()
        T();
        EPRIME();
        return true;
    }
else return false.
```

```
return false
}
procedure TPRIME()
{
if input = "*"
{
```

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

```
procedure TPRIME()
{ if input = "*"
  {
    Advance();
    F();
    TPRIME();
    return true;
  }
else return false;
}
```

```
Advance();
F();
TPRIME();
return true;
}
return false
}
Procedure F()
if input = "("
Advance ();
E();
```

$T \rightarrow *FT'$

$T \rightarrow \epsilon$

$F \rightarrow (E)$

$F \rightarrow (E) \mid id$

```
Procedure F() {
  if input = "("
     Advance()
     E();
  if input = ")"
     return true
  else false.
if input = "id"
.
```

```
if input = ")"
return true
else
return false;
}
```

```
else if input = "id"
{
Advance();
return true;
}
return false
}
```

Q. Left factor the following grammar and then obtain LL (1) parsing table (predictive passing table).

$E \rightarrow T+E \mid T$

$T \rightarrow float \mid float\ *T \mid (E)$.

Grammar after left factoring is

$E \rightarrow TE'$

$E' \rightarrow +E \mid \epsilon$

$T \rightarrow float\ T'$

$T' \rightarrow *T \mid \epsilon$

$T' \rightarrow (E)$

$E \rightarrow TE'$

$E' \rightarrow +E \mid \epsilon$

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

$E \rightarrow T+E \mid T$

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

$T \rightarrow float\, T'$

$T' \rightarrow *T \mid \epsilon$

$T \rightarrow (E)$

$E \rightarrow TE'$

$E' \rightarrow +E \mid \epsilon$

step1: find FIRST and FOLLOW.

FIRST (E') = FIRST(T) = {float, (}

FIRST (E′) ={+,∈}

FIRST (T′) = {*,∈}

FOLLOW (E) = {$,)}

FOLLOW (E′) = FOLLOW (E) = {$,)}

FOLLOW (T) = FIRST (E′) U FOLLOW(E)

={+, $,)}

FOLLOW(T′) = FOLLOW (T) = {+, $,)}

	E	E′	T	T′
FIRST	{float, (}	{+, ∈}	{float, (}	{*, ∈}
FOLLOW	{$,)}	{$,)}	{+, $,)}	{+, $,)}

Yp symbols.

Non-terminal	+	*	(	)	float	$
E	—	—	E → T E′	—	E → T E′	—
E′	E′ → + E	—	—	E′ → ∈	—	E′ → ∈
T	—	—	T → (E)	—	T → float T′	—
T′	T′ → ∈	T′ → * T	T′ → (E)	T′ → ∈	—	T′ → ∈

Q. Write non- recursive predictive parsing algorithm.

Algorithm:-

Input: A string 'w' and a parsing table M is for grammar G.

Output: If w is in L(G), a left most derivation of ‘w’, otherwise an error indication.

Method: Initially the parser is in a configuration in which it has $S on the stack with ‘S’ the Start symbol of ‘G’ on

top and w$ in the I/p buffer.
Set ip (i/p pointer) to point to the first Symbol of w$
Repeat
Let ‘X' be the top stack symbol and ‘a’ the Symbol pointed to by ip.
If ,’X’ is a terminal or $ then
If X = a then
Pop ‘X’ from stack and advance ip
else
error()
else / * X is a terminal */
if M [A,a] = X→Y1Y2.. Yk then.
begin

pop X from stack

Push $Y_k, Y_{k-1}, \ldots Y_1$, onto the stack with Y1 on top of the stack.

output the production $X \rightarrow Y_1 Y_2 \ldots . Y_k$.

end

else

error()

until X = $ /* Stack empty*/

OR

Algorithm

Let 'X' be the stack top Symbol & 'a' be the current input symbol. Then,

1) If X = a = $, then parser halts & announces successful completion of parsing.
2) If X= a ≠ $ the parser pops 'X' from the stack & advances i/p pointer
3) Else if 'X' is a non terminal, then consult the parsing table entry. M[X, a]. This entry will be either an 'X' production of the grammar or an error entry.
4) else error

$E \rightarrow EAE \mid (E) \mid -E \mid id \qquad A \rightarrow + \mid \#$

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

FIRST(A) = { +, # }

FOLLOW(E) = { $,), +, # }

FOLLOW(A) = { (, -, id }

THANK YOU