FRANKLIN'S LECTURES

COMPILER DESIGN

Module 2 Part 1

CST302

SYLLABUS

Module - 2 (Introduction to Syntax Analysis)

Role of the Syntax Analyser – Syntax error handling. Review of Context Free Grammars -Derivation and Parse Trees, Eliminating Ambiguity. Basic parsing approaches - Eliminating left recursion, left factoring. Top-Down Parsing - Recursive Descent parsing, Predictive Parsing,LL (1) Grammars.

PRACTICAL APPLICATIONS

- Syntax Used in programming language Analysis
- Parsers, SQL query parsing in databases, and browser engines.

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM
FRANKLIN'S LECTURES

Phases of compiler

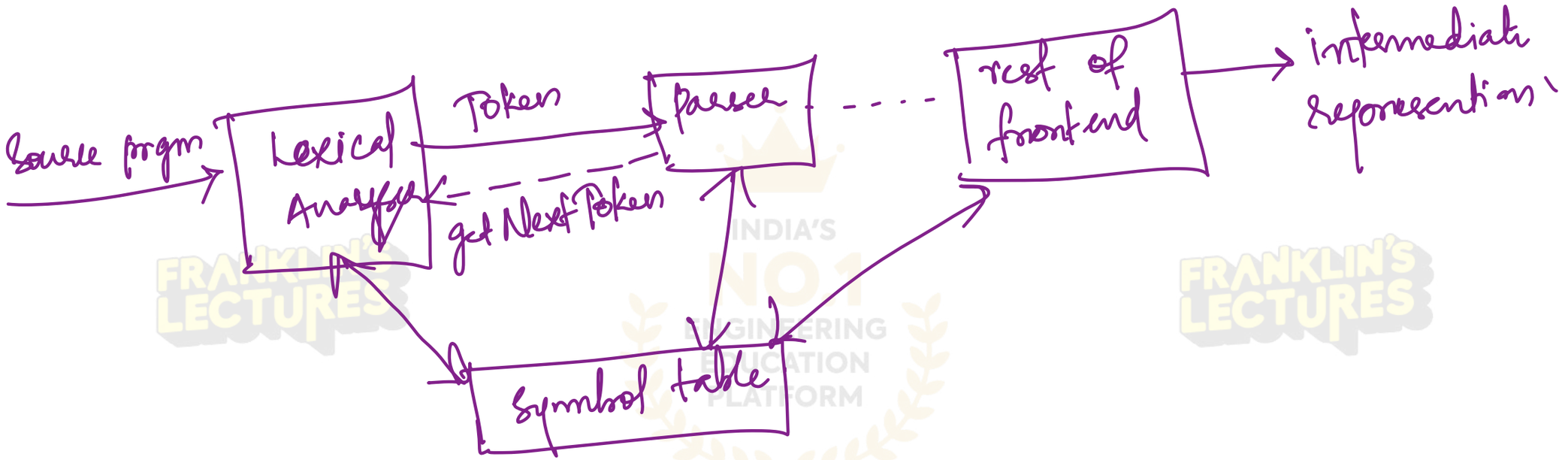
1. Lexical Analysis ✓
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate code generation
5. code optimization
6. Code generation

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

SYNTAX ANALYSIS

- Syntax of programming language constructs can be described by CFG or BNF.

- **The role of parser**
- The parser obtains a string of tokens from the lexical analyser as shown in the figure and verifies that the String can be generated by the grammar for the source language. A parser should report any syntax error.


Source prgm
Lexical Analyser
Token
Parser
get Next Token
rest of frontend
intermediate representation
Symbol table

- **Syntax error handling**

A good compiler should assist the programmer in identifying and locating errors. Errors in a program can be

- Lexical, such as mis-spelling an identifier, keyword or operator.
- Syntactic, such as an arithmetic expression with unbalanced paranthesis.
- Semantic, such as an operator applied to an incompatible operands.
- Logical, such as an infinite recursive calls.

Often many of the error detection and recovery in a compiler is done during the Syntax analysis phase.

The error handler in a parser has the following goals.

1) It should report presence of errors clearly and accurately
2) It should recover from each error quickly enough to be able to detect subsequent errors.
3) It should not subsequently slow down the processing of correct program.

FRANKLIN'S LECTURES
INDIA'S NO.1 EDUCATION PLATFORM

Error recovery strategies. [Essay 7,5,6]

There are many different general strategies that a parser can employee to recover from a Syntactic error.

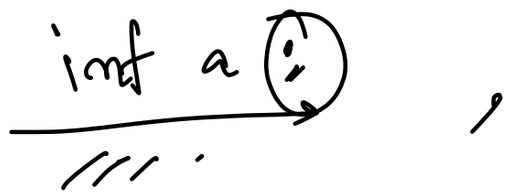
Some error recovery strategies are,

1) Panic mode recovery.
2) Phrase level recovery.
3) Error production.
4) global correction.

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

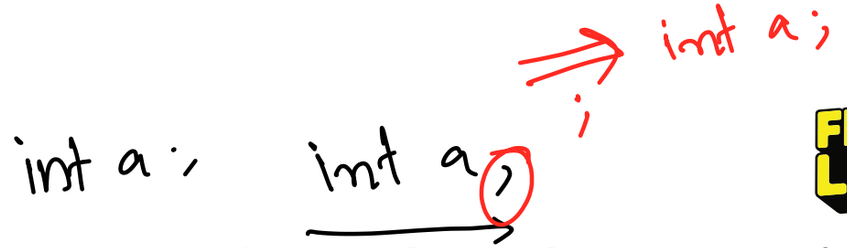
1. Panic mode recovery → quick.

This is the simplest method to implement and can be used by most of the parsing method. On discovering an error the parser discards input symbols one at a time until one of a designated set of synchronizing tokens are found. The synchronizing tokens are usually delimiters such as semicolon, end, } etc. The Panic mode error correction often skips a considerable amount of inputs without checking for additional errors. It has the advantages of simplicity and it is guaranteed not to go into an infinate loop.



, end. } {

2. Phrase level recovery :



On discovering an error, parser may perform local correction on remaining input. A typical local error correction would be to replace a comma by a semicolon, delete an extraneous Semi colon or insert a missing semi colon.

3. Error production :

If an error production is used by the parser we can generate appropriate error diagnostics, to indicate the erranoius construct that have been recognised is the input. (line no, which type error etc are provided).

4) Global correction :

aabaab

A compiler would make a few changes as possible for processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least cost correction. Given an incorrect input string 'x' and grammar 'G', these algorithm will find a parse tree for a related string 'y' such that the number of insertions, deletions and changes of tokens required to transform 'x' into 'y' is as small as Possible. These methods are too costly to implement in terms of time and space.

x → aabaab

y → abab

- no. of insertion - 0
- no of deletion - 2
- changes of token

(3)

- **Context Free grammar** 'G' $G = (V, T, P, S)$

- It is also called Type 2 grammar.
- A grammar G= (V, T, P, S) is called a context Free grammar if every productions in the grammar are is the form of $A \longrightarrow \beta$, here A is a variable and β is a strings of terminals and variables.

ie $A \in V$ and $\beta \in (VUT)^*$.

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

- V ⟶ Variable.
- T ⟶ Terminals.
- P ⟶ Productions.
- S ⟶ start symbol.

$A \longrightarrow \beta$

A ↓ variable, β ↓ variables | terminals.

eg: $A \rightarrow aabB \mid aB$ - - - -

$B \rightarrow id$.

Notational conventions:-

Terminal :-

- Lowercase letters early in the alphabets Such as a, b, c are terminals.
- Operator Such as +, -, *, etc are terminals.
- Punctuations symbols ;, {, },' etc are terminals.
- Digits 0, 1...9 are terminals
- Bold phase strings such as if,id,and, not, or etc are terminals. (true, false etc)

Non terminals / variable.

- Upper case letters early is the alphabet A, B,C.
- The letter S → Start symbol.
- Lower case italic names such as expr, stmt etc. expr, stmt
- Upper case letters such as X,Y,Z represents grammar symbol.ie, non-terminals or terminals
- Lower case letters late is the alphabet u, v, z represent strings of terminals.
- Lower case greek letters α, β … for example represents strings of grammar symbol.

Derivations

It Is an approach used to define the language of a grammar. Two types of derivations are,

1) Right most derivation

2) Left most derivation

Left most derivations - In this at each step of derivation we replace the left most variable by one of its rules.

Eg:-Consider the grammar. aabbaa

S ⟶ aAS|a.

A ⟶ SbA |ba.

$S \rightarrow aAs \mid a$

$A \rightarrow SbA \mid ba$

$\Rightarrow aabbaa$

$S \rightarrow aAS$ abas X

$\rightarrow aSbAS$

$\rightarrow aabAS$

$\rightarrow aabbaS$

$\rightarrow aabbaa$

Left-most derivation

Right most derivation.

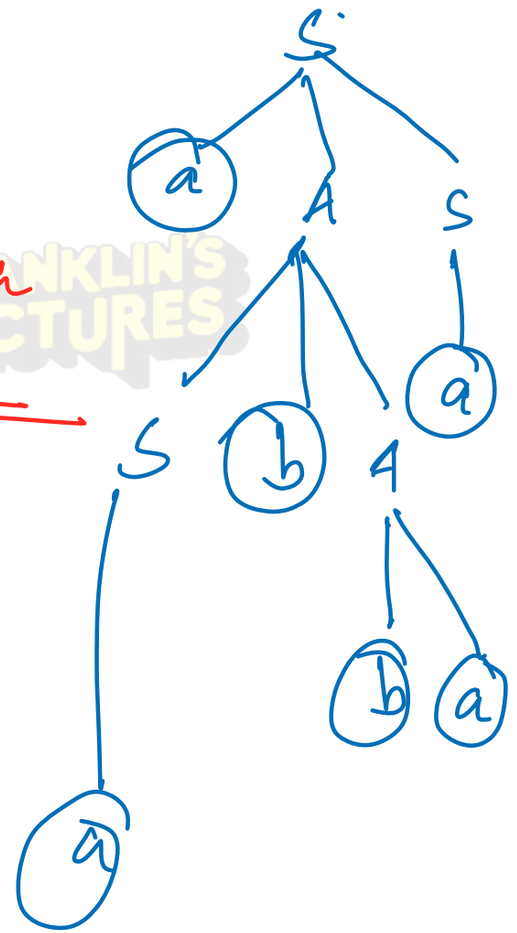
$S \rightarrow aAS$

$\rightarrow aAa$

$\rightarrow aSbAa$

$\rightarrow aSbbaa$

$\rightarrow aabbaa$



FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

The leftmost derivation for the string aabbaa is

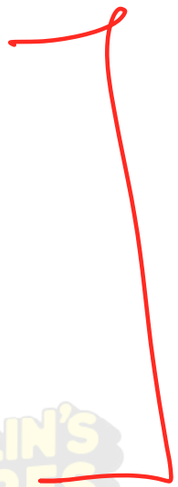
$S \rightarrow a\,\underline{A}S$

$\rightarrow aS\underline{b}AS$

$\rightarrow aaba\underline{A}s$

$\rightarrow aabba\underline{S}$

$\rightarrow aabbaa$



leftmost derivation

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

2) Right most derivations: - In this at each step of derivation way we replace the right most variable by one of its rules.
eg:- consider the grammar gives above, the right most derivation for the string aabbaa is,

S → aA<u>S</u>

→ a<u>A</u>a

→ aSb<u>A</u>a

→ a<u>S</u>bbaa

→ aabbaa.

right most derivation

Derivation Tree (parse Tree)

A derivation in a CFG can be represented by using trees called derivation tree or parse tree. ie, graphical representation of derivation is called parse tree that gives the choice of replacement. Derivation on a parse tree for a CFG, G= (V,T, P,S) is a tree which satisfies the following conditions.

1. Every vertex or node has a label which is a variable or a terminal symbol or $\in$.
2. The root node has the label S
3. The bottom most nodes called leaf nodes are labelled from TU{ $\in$}

4. If an interior node is labelled by A∈V and its children are labelled as $x_1, x_2,...x_n$ from left to right, then $A \rightarrow x_1, x_2, ...x_n \in P$

Eg - Consider the grammar,

$S \rightarrow aAS|a|SS$

$A \rightarrow SbA|ba$ derivation tree for the string, aabaa is,

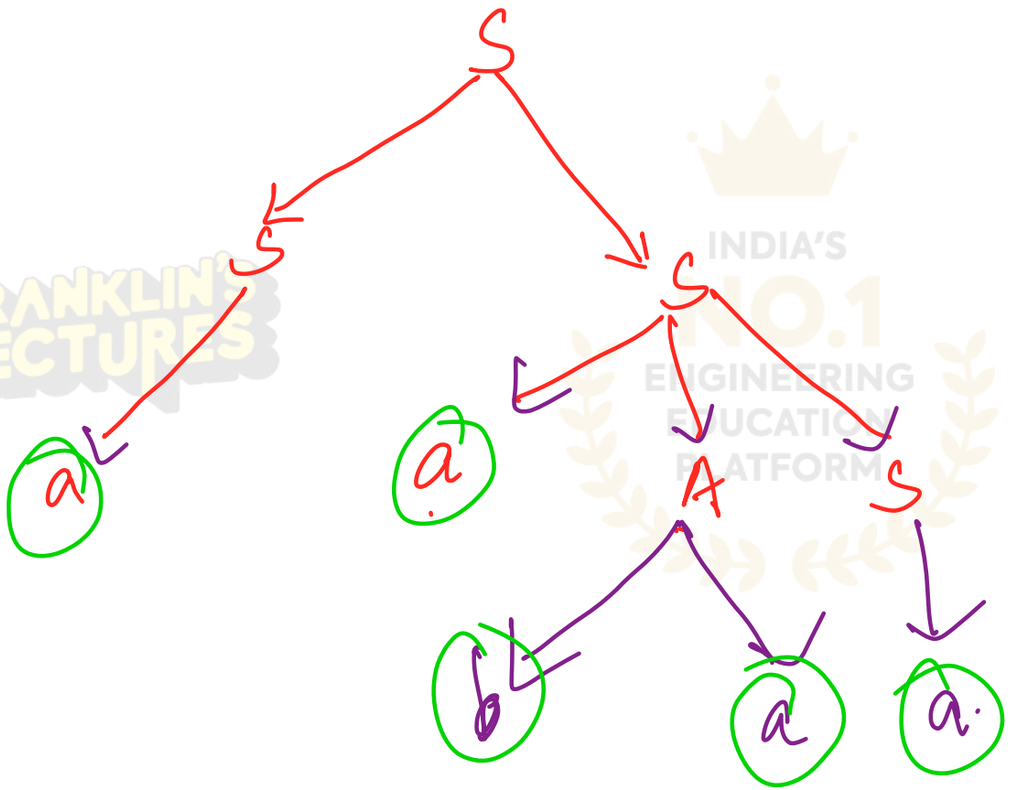
FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

FIGURE

$S \rightarrow aAS \mid a \mid SS$

$A \rightarrow SbA \mid ba$

aabaa



$G = V, T, P, S$

terminals

variable

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

start symbol

production

Elimination of left recursion

A grammar which contains the production of the form $A \rightarrow A\alpha/\beta$
Top down parser can't parse left recursive grammar. So we have to eliminate left recursion. Left recursion can be eliminated by the following productions,

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \in$

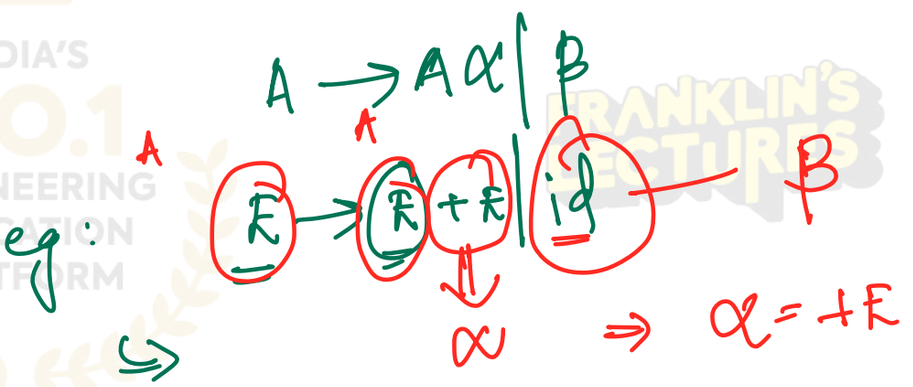
$A \rightarrow A\alpha | \beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \in$

eg:



$\Rightarrow$ $E \rightarrow id\,E'$

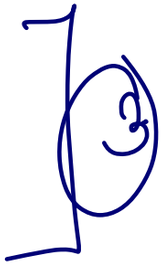
$E' \rightarrow +E\,E' \mid \in$

Q:- Eliminate left recursion from the grammar,

$E \rightarrow E+T|T$

$T \rightarrow T*F|F$

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



First two productions are left recursive after eliminations of left recursion, the grammar becomes,

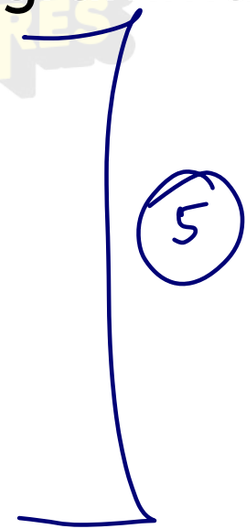
$E \rightarrow TE'$

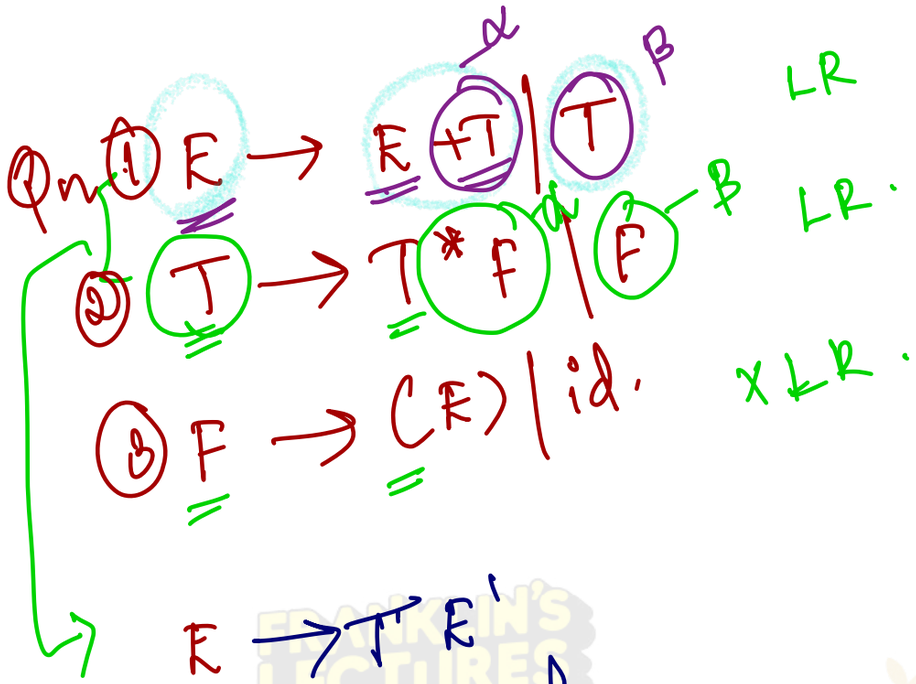
$E' \rightarrow +\ TE'|\epsilon$

$T \rightarrow FT'$

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

$F \rightarrow (E)|id$





$A \to A\alpha \mid \beta \Rightarrow$ left recursive.

$\Downarrow$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

Left factoring

Left factoring is a grammar transformation that is useful for predictive parsers. If there are two alternatives for a production, we do left factoring,

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$, can be left factored as follows.

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta 1 | \beta 2$$

$S \rightarrow aabE | aabba$ (α = aab, β_1 = E, α = aab, β_2 = ba)

$S \rightarrow aabS'$

$S' \rightarrow E | a$

eg :- $S \rightarrow iEtSeS / iEtS$ can be left factored as

$S \rightarrow iEtSS'$

$S' \rightarrow eS | \epsilon$

$S \rightarrow iEtSS'$

$S' \rightarrow eS | \epsilon$

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$

$\Downarrow$

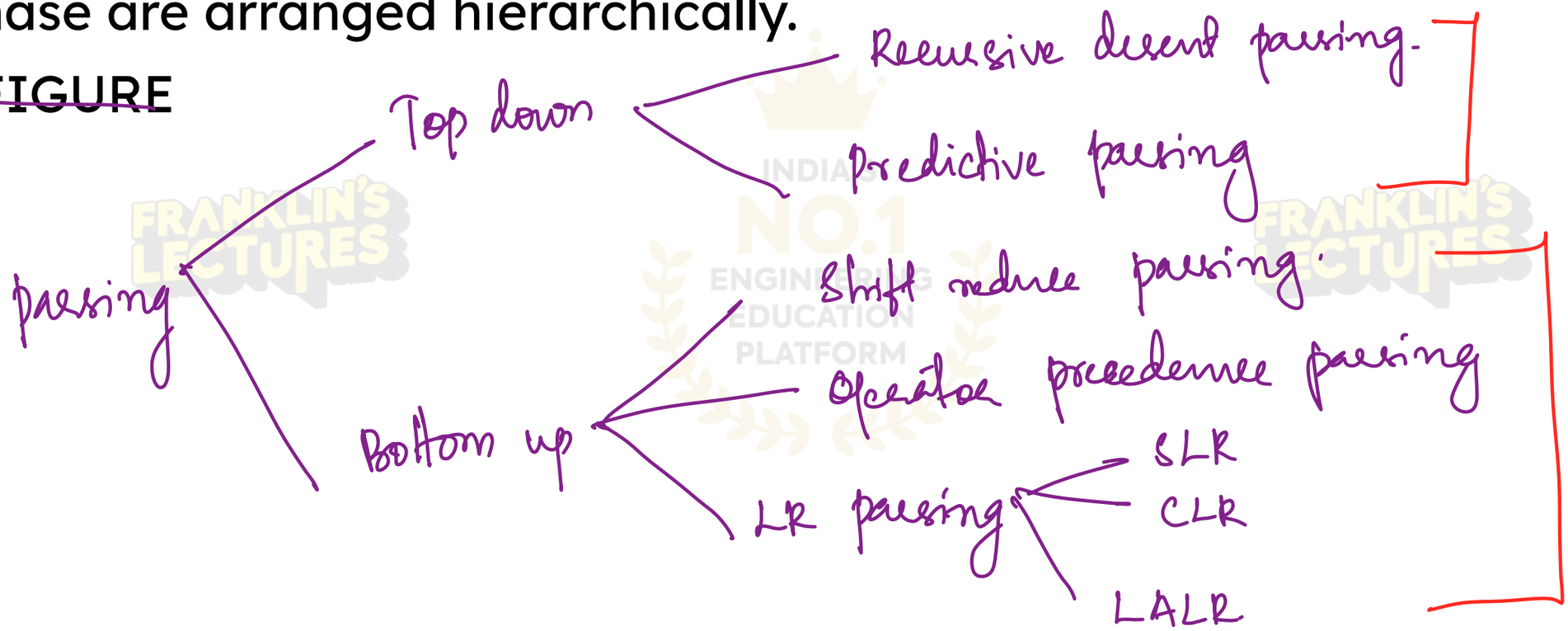
$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

Parsing

Second phase of compiler. Here tokens obtained during lexical analysis Phase are arranged hierarchically.

- ~~FIGURE~~



Top down parsing

In top down parsing, the parser build parse trees from the top to the bottom. ie construction of parse trees for the input string from the root and creating the nodes of the parse tree in Preorder.
Top down parsing can be classified into two,

1) Recursive descent parsing.
2) Predictive parsing.

Recursive descent parsing

In top down parsing the parser find a left most 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 preorder.

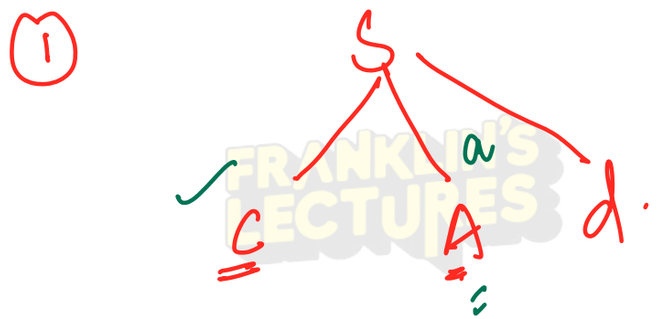
- Recursive descend parsing is the general forms of top down parsing that involve back tracking. ie, making repeated scans of the input.
- eg:- given the grammar S → cAd, A → ab|a
- And input string w = cad.
- To construct a parse tree for the above example in top down, initially create a tree consisting of single node labelled S. An input pointer points to 'c, the first symbol of ‘w’, that use the production for S.

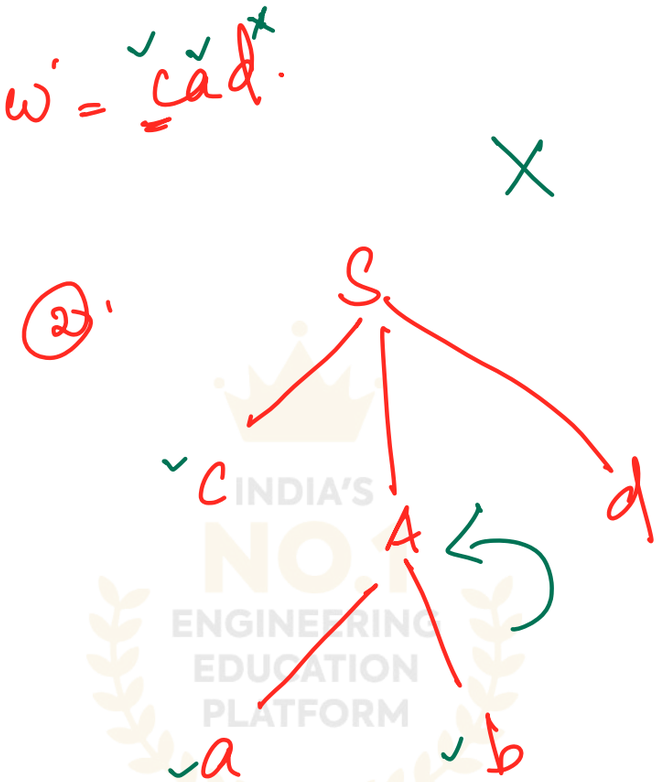
- The leftmost leaf labelled 'c' matches first symbol of ‘w’. So now advance the input pointer to ‘a’, the second symbol of ‘w’ & consider the next leaf labelled ‘A’. Now expand 'A' using first alternative for ‘A’. Now have a match for second input symbol. So advance input pointer to 'd', the third input Symbol and compare 'd' to next leaf Labelled 'b'. 'b' does not match 'd'. So report a failure & go back to 'A' to see whether there is another alternative for ‘A’ that might, produce a match a In going back to ‘A’ reset the input pointer to position 2. ie, try for second alternatives for A to obtain the tree. The leaf a matches the second symbol of 'w' and the leaf 'd' matches the third Symbol. So we get a parse tree for w.

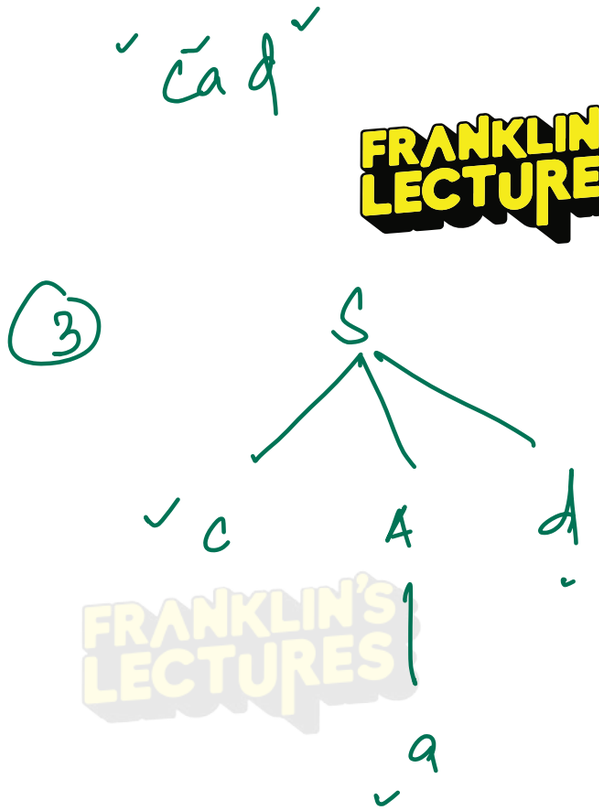
Figure

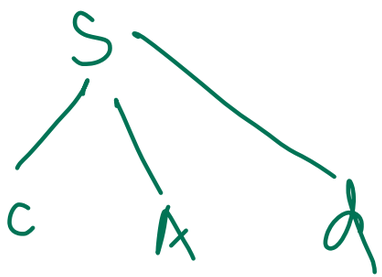
$S \rightarrow cAd$

$A \rightarrow ab \mid a$









Problems in recursive descent parsing.

1. Left recursion: - In left recursion the grammar has productions of the form $A \longrightarrow A\alpha|\beta$ $\Rightarrow$ $A \rightarrow \beta A'$ $A' \rightarrow \alpha A' | \epsilon$
2. Backtracking :- It occurs when these is more than one alternative in the production while parsing the input string.

eg-

$S \longrightarrow cAd$

$A \longrightarrow ab|a$

Two alternative for A.

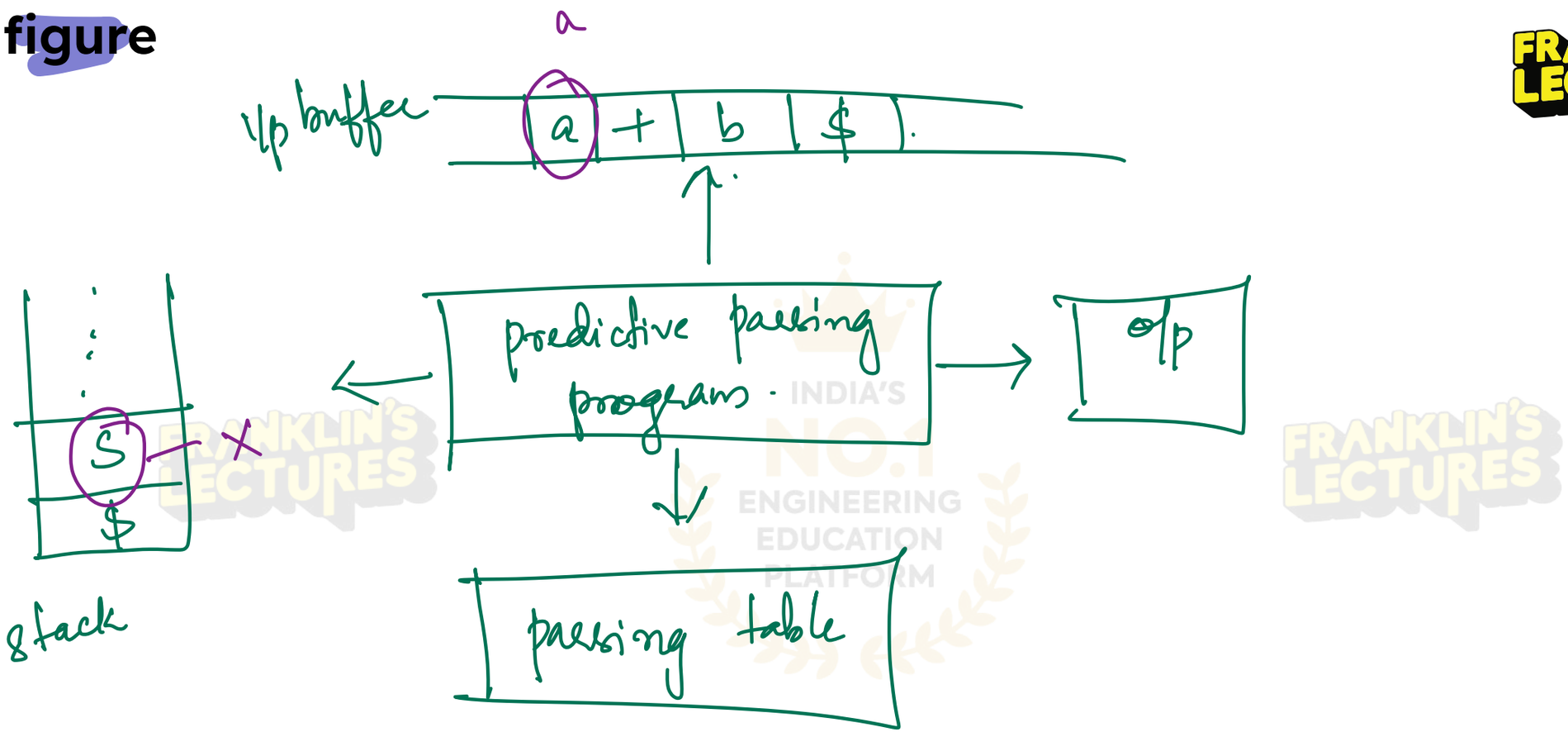
3. It is very difficult to identify the position of the errors.

Predictive parsing

For parsing an input string using Predictive parsing technique the grammar should not be left recursive and left factoring is to be done.

A predictive parser has an input buffer, a stack, a parsing table & an output Stream. The input buffer contains the string to be parsed followed by $, a Symbol used as right end marker to indicate end of input string. The stack contains the sequence of grammar symbols with $ on the bottom to indicate bottom of the stack .Initially stack contains the start symbol of the grammar on top of the $'. The parsing table is a two dimensional array M [A, a] where 'A' is a non terminal and 'a' is a terminal or $ symbol.

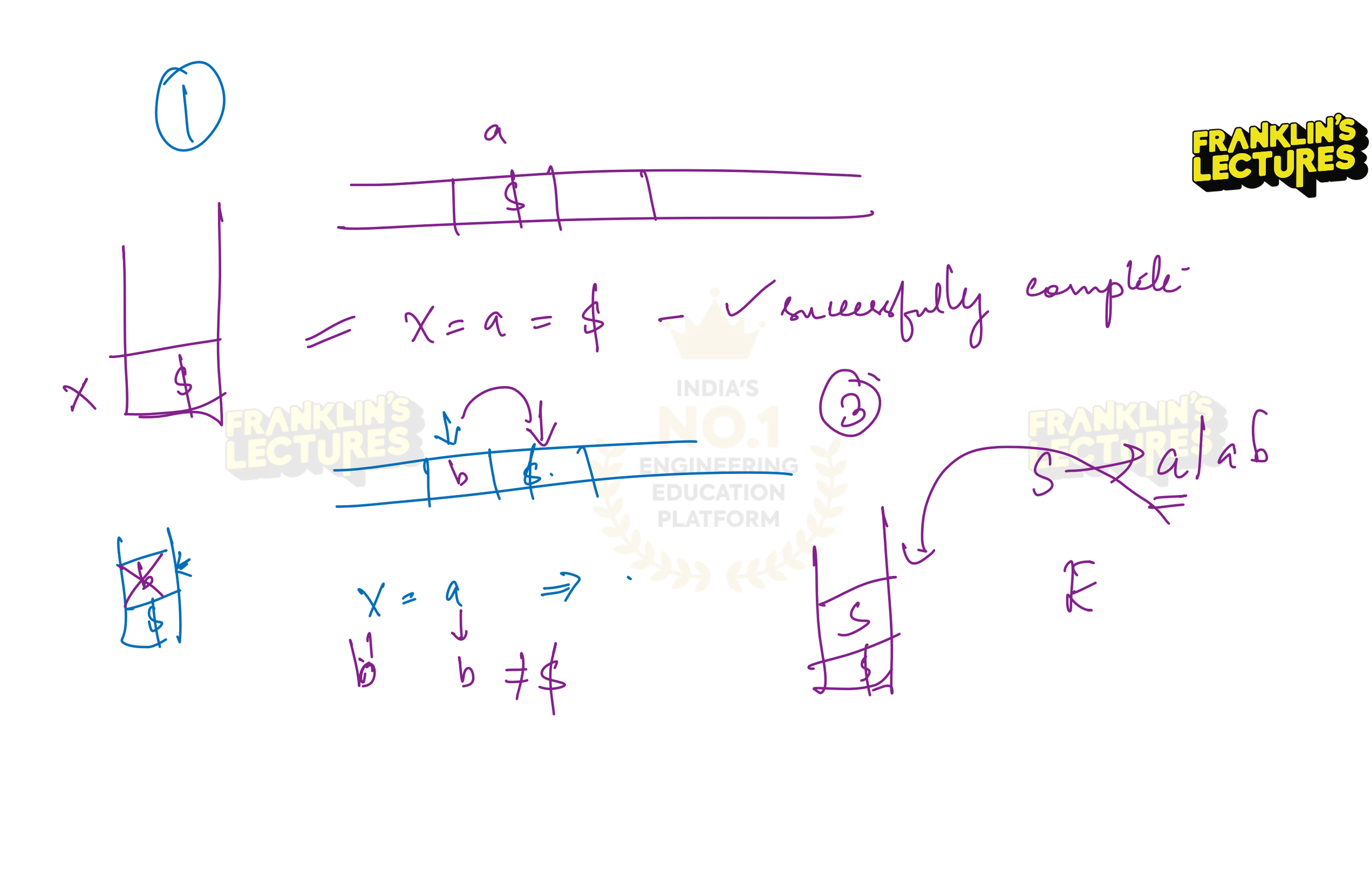
- figure


a
i/p buffer
a
+
b
$
predictive parsing program
o/p
S
X
$
stack
passing table

The parser is controlled by a Program that behaves as follows:

The program consider ‘X′, the symbol on top of the stack and 'a' the current input symbol. These two symbols determine the actions of parser. There are three Possibilities.

1. If X = a = $, the parser halts and announces successful completion of parsing.
2. If X=a≠$, the parser pops ‘X’ out of the Stack and advances input pointer to the next input symbol.
3. If ‘X′ is a non terminal the program consults parsing table entry M[X, a]. This entry will be either an X production of the grammar or an error entry.


1
a
$
X
$
= X = a = $ - ✓ successfully complete
b
$
X = a ⇒
b
b ≠ $
3
S → a | a b
S
$
E

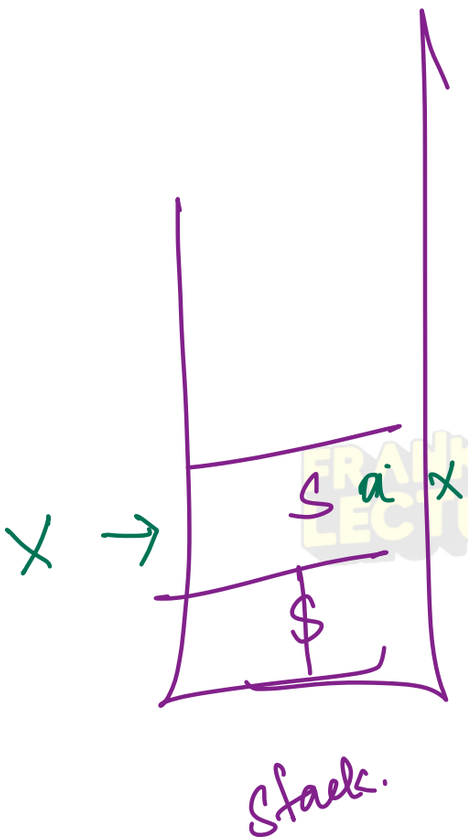
Predictive parsing Algorithm

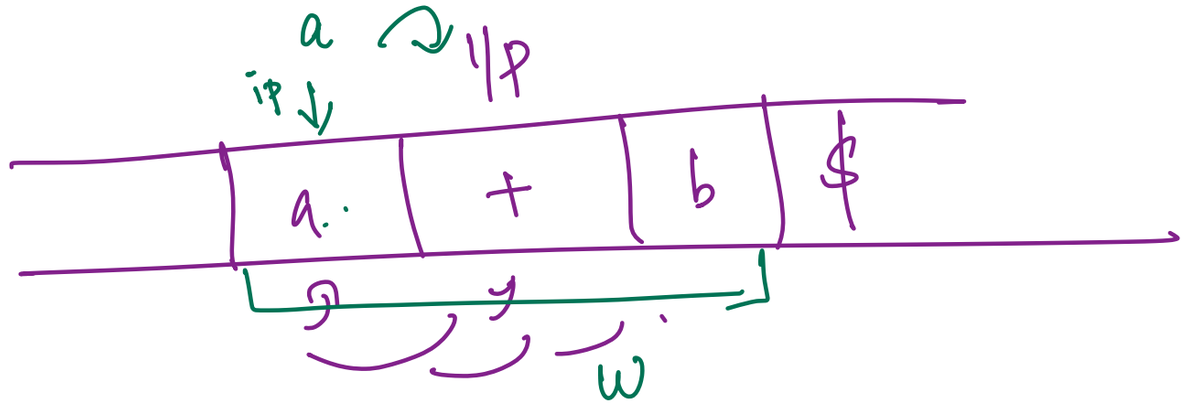
Input: A string ‘w’ and a parsing table M for grammar G

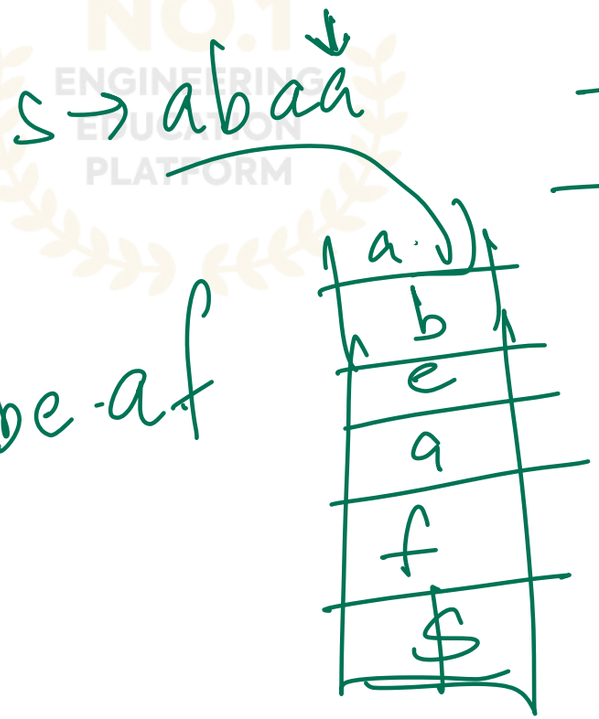
Output: If 'w' is in L(G), a leftmost derivation of 'w', otherwise an error indication.

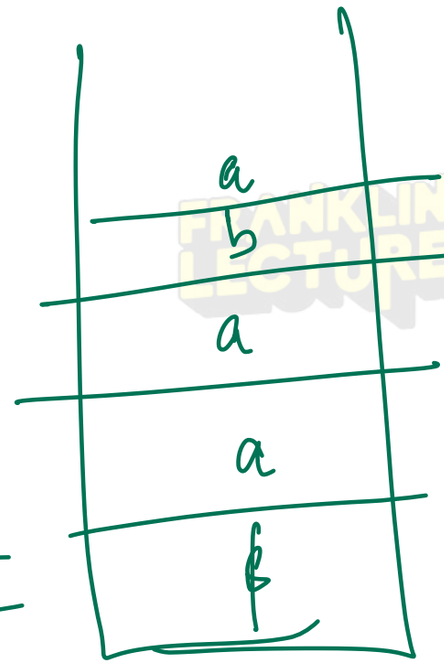
Method: Initially the parser is in a configuration in which it has $S on the stack with 'S’ the start symbol of ‘G’ on top and w$ in the input buffer.

set ip (i/p pointer) to point to the first Symbol of w$.


X
$
a
x
stack.


a
i/p
ip
a..
+
b
$
w


S → abaa
S → abe-af
a
b
e
a
f
$


a
b
a
a
f

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 non terminal */

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

If M[A, a] = $X \longrightarrow Y_1 Y_2 \ldots Y_k$ then

begin

pop 'X' from stack

Push y_κ $Y_{K-1} \ldots Y_1$ onto the stack with Y_1 on the top.

Output the production $X \longrightarrow Y_1 Y_2 \ldots Y_k$.

end.

else

error()

until X= $ /* stack empty */

FIRST() and FOLLOW()

- First () and FOLLOW() are two functions associated with the grammar that help us to fill the entries of M table (predictive parsing table)
- FIRST():- Is a function which gives a set a terminals that begins the Strings derived from the production rule.
- FOLLOW():- Is a function which gives set of terminals that can appear immediately to the right of a given symbol.

$S \rightarrow aab$ FIRST(S) = {a}

$S \rightarrow asb \mid a$ FOLLOW(S) = {b}

Algorithm to compute FIRST(X) for a grammar Symbol X

1) IF X is a terminal, then FIRST(X) = {x}
2) If $X \rightarrow \epsilon$ is a production then add 'ϵ' to the set of FIRST (X).
3) If 'X' is a non-terminal and,

$X \rightarrow Y_1, Y_2, \ldots Y_k$ is a production then FIRST (Y_1) is added to FIRST(X) and If FIRST (Y_1) contains ϵ then add FIRST(Y_2) to FIRST (X) and if FIRST (Y_2) contains ϵ then add FIRST (Y_3) to FIRST(X) and so on. Finally add ϵ to FIRST(X) if FIRST (Y_i) contains ϵ

$S \rightarrow ABC \mid \epsilon$

$A \rightarrow a \mid \epsilon$

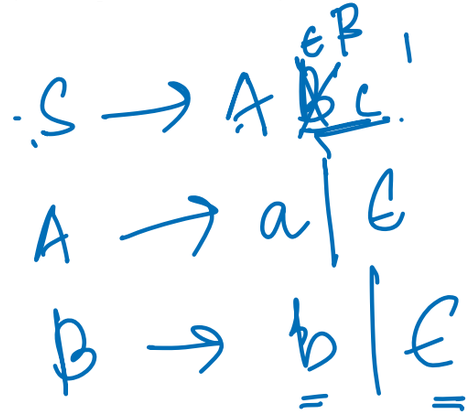
$B \rightarrow b \mid \epsilon$

$C \rightarrow c \mid \epsilon$

FIRST(S) = {a, b, c, ϵ}

Algorithm to Find FOLLOW (A)

1) Place $ in FOLLOW(S) where 'S' is the start symbol and $ is the input right end marker.
2) If there is a production A → αBβ, then everything in FIRST(β) except for ∈ is placed in FOLLOW(B).
3) If there is a production A → αB or A → αBβ where FIRST(β) contains ∈ (ie,β →*∈) then everything in FOLLOW (A) is in FOLLOW(B).



FOLLOW(A) = { b, $, c }

Q. Find FIRST() and FOLLOW(Y) of the Grammar given below.

Start Symbol

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \in.$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid \in$

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

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

FIRST()

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

FIRST(E') = {+, ϵ}

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

FIRST(T') = {*, ϵ}

FIRST(F) = {(, id}

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

FOLLOW()

FOLLOW(E) = {$,)}

FOLLOW(E') = {$,)}

FOLLOW(T) = {+, FOLLOW(E')} = {+, $,)}

FOLLOW(T') = FOLLOW(T) = {+, $,)}

FOLLOW(F) = {* ∪ FOLLOW(T')} = {*, +, $,)}

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

Q. Find FIRST() and FOLLOW()

S → A

A → aB|Ad

B → bBC|f

C → g.

FRANKLIN'S LECTURES
NO.1 ENGINEERING EDUCATION PLATFORM

$S \rightarrow A$

$A \rightarrow aB \mid Ad$

$B \rightarrow bBC \mid f$

$C \rightarrow g$

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

FIRST(A) = {a}

FIRST(B) = {b, f}

FIRST(C) = {g}

FOLLOW(S) = {$}

FOLLOW(A) = {$, d} = FOLLOW(S) ∪ d

FOLLOW(B) = FOLLOW(A) ∪ FIRST(C)

= {$, d, g}

FOLLOW(C) = FOLLOW(B)

= {$, d, g}

FRANKLIN'S LECTURES
INDIA'S NO1 ENGINEERING EDUCATION PLATFORM

CONSTRUCTION OF PREDICTIVE PARSING TABLE

The following Algorithm can be used to construct a predictive parsing table for a grammar ‘G’.

Algorithm: Construction of predictive Parsing table.

Input: Grammar “G”.

Output: parsing table M.

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

Method:

1) For each production A → α of the form do steps 2 and 3.
2) For each terminal in FIRST(α) add A → α to M[A, a]
3) If ‘Є’ is in FIRST(α) add A → α to M[A, b] for each terminal b in FOLLOW (A). If ‘Є’ is in FIRST(α) and ‘$′ is an in FOLLOW(A) add A → α to M [A,$]

FRANKLIN'S LECTURES
NO.1 ENGINEERING EDUCATION PLATFORM

Q. Construct predictive parsing table for the following grammar...

$E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E)|id$

Step1: Eliminate left recursion if the given grammar is left recursive:

Given grammar is left recursive.
So after left recursion elimination grammar becomes,

$E \rightarrow TE'$

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

Step 1 : Eliminate left recursion.

Step 2 : Eliminate left factoring -

Step 3 : Find FIRST() & FOLLOW().

Step 4 : Table construction.

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

FRANKLIN'S LECTURES

INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

FRANKLIN'S LECTURES

FRANKLIN'S LECTURES

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

$\Rightarrow \quad A \rightarrow \alpha A'$

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

Step2: Left factoring :

In this grammar there is no need of left factoring

Step3: Find FIRST() and FOLLOW() of the all grammar symbols.

FIRST (E) = {(, id}

FIRST (E′) = {+, ∈}.

FIRST (T) = {(, id}.

FIRST (T′)= {*, ∈}.

FIRST (F) = {(,id}

$E \to TE'$ — $E' \to +TE' \mid \in$ $T \to FT'$ $T' \to *FT' \mid \in$ $F \to (E) \mid id$.

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

FIRST(E') = {+, ∈}

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

FIRST(T') = {*, ∈}

FIRST(F) = { (, id }

FOLLOW(E) = { $,) }

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

FOLLOW(T) = FIRST(E') = {+} ∪ FOLLOW(E')
= { +, $,) }

FOLLOW(T') = FOLLOW(T)
= { +, $,) }

FOLLOW(F) = FIRST(T')
= {*} ∪ FOLLOW(T')
= { *, +, $,) }

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

FOLLOW (E) = {$,)}

FOLLOW (E′) = { $,)}

FOLLOW (T) = {+, $,)}.

FOLLOW (T′) = {+, $,)}.

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

Step 4: Construction of predictive Parsing table.

	E	E'	T	T'	F
FIRST	c, id	+, ε	c, id	*, ε	c, id
FOLLOW	$,)	$,)	+, $,)	+, $,)	*, +, $,)

Non terminals	Input Symbol					
	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	$F \to id$			$F \to (E)$		

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

$$\frac{E' \to +TE'}{E' \to \epsilon}$$

$F \to (E)$ = $F \to id$

Moves made by predictive parser on input id+id*id$

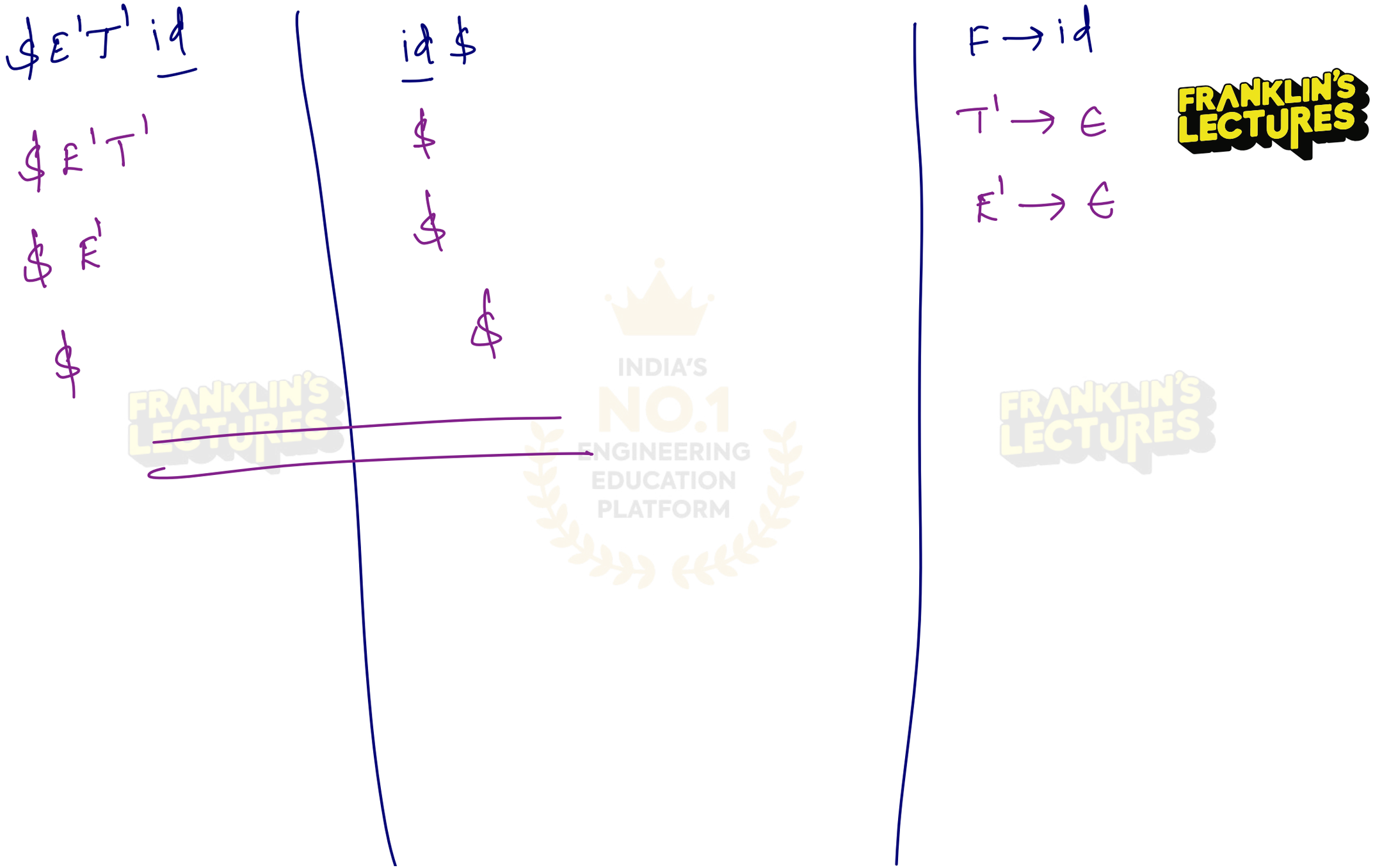
TABLE

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM
FRANKLIN'S LECTURES
FRANKLIN'S LECTURES

STACK	INPUT	OUTPUT
$ E	id + id * id $	
$ E' T	id + id * id $	E → T E'
$ E' T' F	id + id * id $	T → F T'
$ E' T' id	id + id * id $	
$ E' T'	+ id * id $	T' → ∈
$ E' T +	+ id * id $	
$ E' T	id * id $	E' → + T E'
$ E' T' F	id * id $	T → F T'
$ E' T' id	id * id $	F → id
$ E' T'	* id $	T' → * F T'
$ E' T' F *	* id $	

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

FRANKLIN'S LECTURES
INDIA'S NO 1 ENGINEERING EDUCATION PLATFORM



FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM
FRANKLIN'S LECTURES
FRANKLIN'S LECTURES

LL(1) Grammar :

A grammar whose parsing table has no multiply defined entries is said to be LL(1). The first L in LL(1) stands for scanning input from left to right, the second L for left most derivation and 1 for using 1 input symbol.

To become a grammar LL(1):

1) The grammar should not be an ambigous grammar.

2) The grammar should not be left recursive.

3) The grammar should be deterministic

4) The parsing table should not contain multiply defined entries.

Q. How we can check whether a grammar is LL(1) or not without constructing a predictive parsing table?

The grammar is said to be LL(1) grammar if the following two conditions are satisfied:

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING EDUCATION PLATFORM

Conditions:

1. For every productions of the form

$A \rightarrow \alpha_1|\alpha_2|.....\alpha_n$

$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \Phi \nexists i,j \geq n$

where $i \neq j$

2) For every non-terminal A if FIRST(A) contains $\in$ then,

$FIRST(A) \cap FOLLOW(A) = \Phi$

FRANKLIN'S LECTURES
INDIA'S NO.1 ENGINEERING LEARNING PLATFORM

Q. Check whether the grammar

S → iEtSA | a

A → eS | ∈.

E → b is LL(1) or not without constructing Predictive parsing pable.

FOLLOW($) = $.

$S \to iEtSA \mid a$ $\Rightarrow$ FIRST(iEtSA) ∩ FIRST(a)

$= \{i\} \cap \{a\} = \phi$ ✓

$A \to eS \mid \epsilon$ = FIRST(eS) ∩ FIRST(ϵ)

$= \{e\} \cap \{\epsilon\} = \phi$

Condition 1

a) S → iEtSA|a: FIRST (iEtSA) ∩ FIRST (a)

:{i}∩{a} = Φ

b) A → eS|∈: FIRST (eS) ∩ FIRST(∈)

:{e} ∩ {∈}= Φ

FOLLOW(S) = {$} ∪ FIRST(A) = {$, e}

FOLLOW(A) = {$, e}

Condition 2

According to condition 2 FIRST(A) has ∈ so FIRST(A) ∩ FOLLOW(A) is empty .
check whether the condition 2 is satisfied or not Satisfied
So FIRST(A) ∩ FOLLOW(A) =Φ
but {e,∈ } ∩ {$,e} = {e}. Thus we can say that condition 2 is not Satisfied
Since one of the condition fails
so the given grammar is not LL(1).

FRANKLIN'S LECTURES

THANK YOU