FRANKLIN'S LECTURES

S6 First Series MARATHON

COMPILER DESIGN

Module 1

CST 302

Q. Describe input buffering scheme in lexical analyzer

Input buffering scheme in a lexical analyzer is a technique used to efficiently read input characters from the source program.

- The input is stored in two buffers of equal size

- Each buffer has a sentinel character at the end to detect buffer exhaustion.
- This scheme reduces the number of input/output operations and allows fast character access during lexical analysis.

Q. Define tokens, lexemes and pattern

tokens: A token is the smallest unit recognized by the compiler during lexical analysis. identifier, keyword, constant.

Lexeme :- A lexeme is actual sequence of characters in the source program that matches the patten of a token. count ;+

Pattern :. A pattern is a rule or regular expression that describes the structure of lexemes belonging to a particular token.

Q. Explain bootstrapping with an example.

Bootstrapping is the process of developing a compiler for a programming language using the same language itself.

Initially, a simple or partial compiler written in a low-level language or another existing language, and then it is gradually improved to become a full compiler.

- Makes the compiler portable.
- Helps in easy maintenance and upgrades
- Improves compiler efficiency.

Q. Explain different compiler construction tools

FRANKLIN'S LECTURES

Compiler construction tools are Software tools that helps in the automatic generation of different parts of a Compiler, making Compiler development faster, easier, & less error.

1. lexical Analyzer Generators
2. parser Generators
3. Syntax - Directed Translation Tools
4. Automatic code generation
5. Data Flow Analysis Tools.

Q. Explain in detail the various phases of the compiler with a neat diagram. Illustrate the output of each phase for the input
Sum:=a+b*30
where a and b are float variables.

1. Sum:=a+b*30
   a & b float variables.

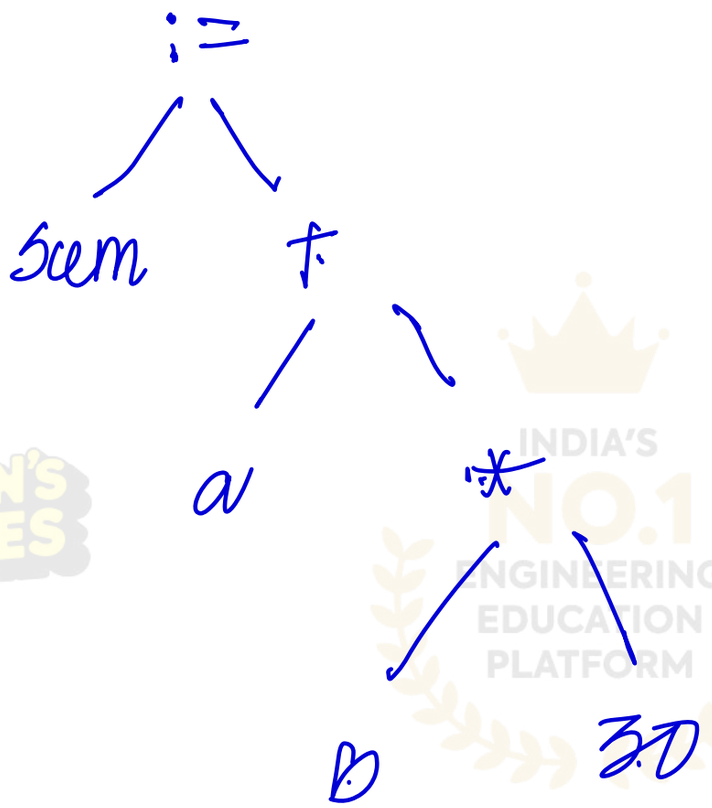
2.

2.

lexeme	Token
sum	id
:=	assign_op
a	id
+	plus_op
b	id
*	mul_op
30	number

3.

sum := a + b * 30



4.

a & b are float variable.

30 → integer.

Integer convert to float.

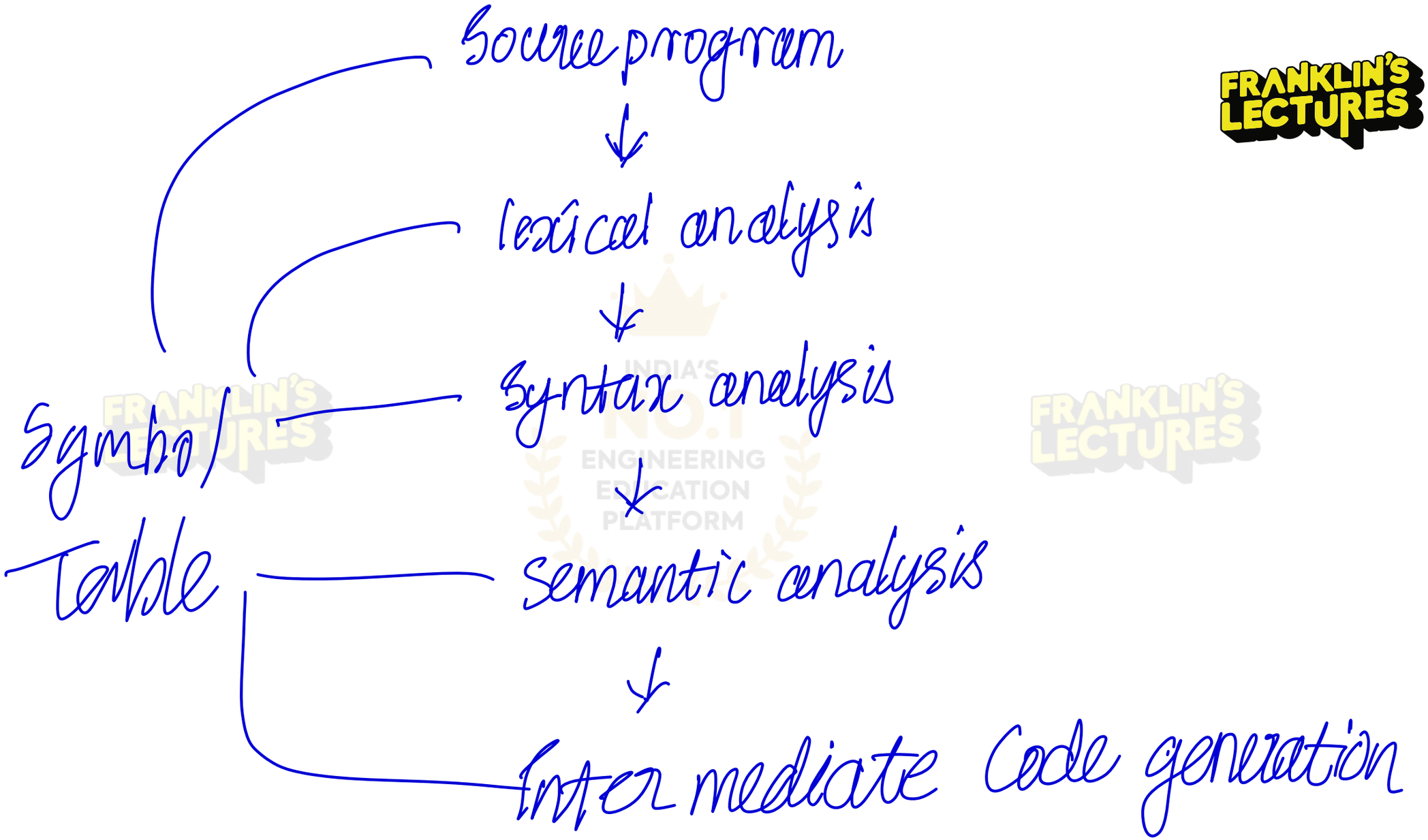
30.0

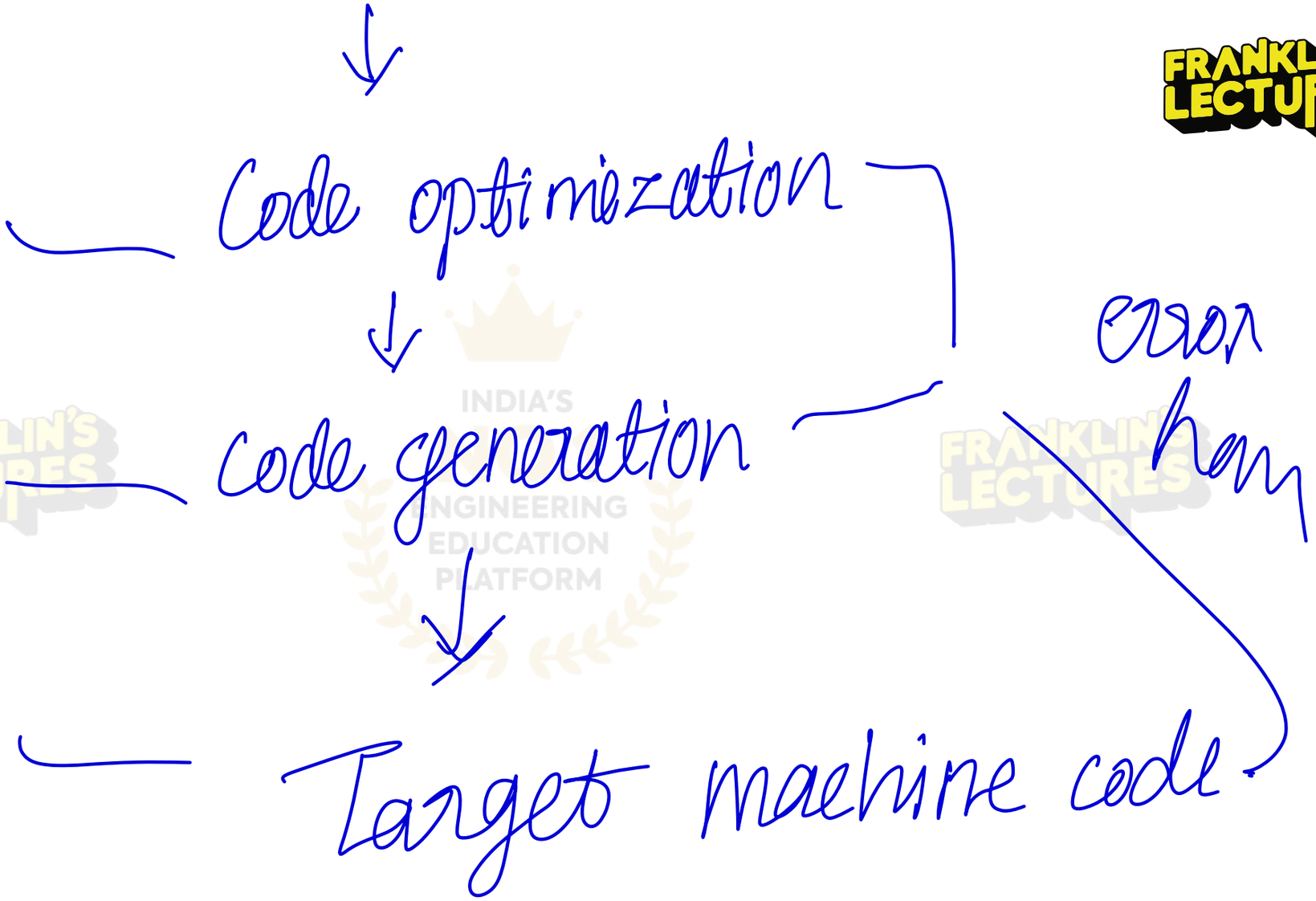
5. Three address code:

t1 = b * 30.0

t2 = a + t1

Sum = t2


Source program
lexical analysis
Syntax analysis
semantic analysis
Intermediate Code generation
Symbol Table


Code optimization
code generation
Target machine code
error
hem

6.

$t1 = b * 30.0$

$Sum = a + t1$

7.

```
LOAD b
MUL 30.0
ADD a
STORE SUM
```

Q. Explain the front and back end model of a compilers

FRANKLIN'S LECTURES

Front end of a Compiler

1. lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate code generation.

Back end of a Compiler

code optimization

code generation.

Source program

↓

Front end

(Analysis + Intermediate Representation)

↓

Intermediate code

↓

Back end (optimization + code generation)

↓

Target Machine code.

FRANKLIN'S LECTURES

S6 First Series MARATHON

COMPILER DESIGN

Module 2

CST 302

Q. Show that the following grammar is ambiguous by giving two parse trees for the string abab. Here e is the empty string.

$$S \longrightarrow aSbS|bSaS|\varepsilon$$

Given $S \rightarrow aSbS|bSaS|\varepsilon$

String abab.

Parse tree 1

$S \rightarrow aSbS$

$\rightarrow a\epsilon bS$

$\rightarrow a\epsilon b\epsilon$

$\rightarrow abaSbS$

$\rightarrow aba\epsilon bS$

$\rightarrow aba\epsilon b\epsilon$

$\rightarrow abab$

Parse tree 2

$S \rightarrow bSaS$

$\rightarrow b\epsilon aS$

$\rightarrow ba\epsilon$

$\rightarrow babSab$

$\Rightarrow abab.$

Q. Describe the recursive procedure of a recursive descent parser.

- The parse starts with the start symbol of the grammar.
- Each procedure tries to match input tokens with the production rules of that non terminal
- if a production contains non terminals. the corresponding procedures are called recursively.

- If the input matches the grammer, parsing succeeds otherwise, it reports an error.

Q. Consider the following grammar

$$E \rightarrow E+T|T$$
$$T \rightarrow T*F|F$$
$$F \rightarrow \sim F|(E)|id$$

i . Remove left recursion from the grammar.
ii . Construct a predictive parsing table.
iii. Justify the statement The grammar is LL(l)

- Given

$$E \rightarrow E+T/T$$

$$T \rightarrow T*F/F$$

$$F \rightarrow \sim F/(E)/id.$$

(i)

$E \to E + T / T$ (α = $+T$, β = T)

$T \to T * F / F$ (α = $*F$, β = F)

Step 1: Remove left recursion from E

$E \to TE'$

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

Step 2: Remove left recursion from T

$T \to FT'$

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

$A \to A\alpha / \beta$

$A \to \beta A'$

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

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

$F \rightarrow \sim f / (E) / id.$

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

FIRST (E') = { +, ε }

FIRST (T) = { (, id }

FIRST (T') = { *, ε }

FIRST (F) = { (, id }

FOLLOW (E) = {), $ }

FOLLOW (E') = {), $ }

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

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

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

Non-terminals	id	(	+	*	)	$
E	$E \to TE'$	$E \to TE'$	—	—	—	—
E'	—		$E' \to +TE'$	—	$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$	—	—	—	—
T'	—	—	—	$T' \to *FT'$	$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$	$F \to (E)$	—	—	—	—

(iii) . The grammer is free from left recursion.

. FIRST sets of alternative productions are disjoint.

• FIRST & FOLLOW set do not conflict

. Each table entry contains only one production.

Q. Design a recursive descent parser for the grammar:s → cAd,A → ab/b

1. Grammer analysis

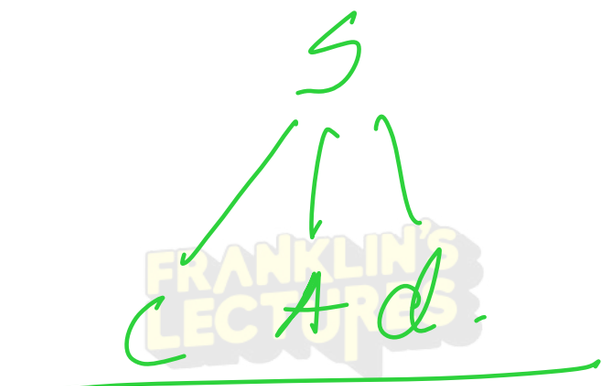
Start Symbol : S

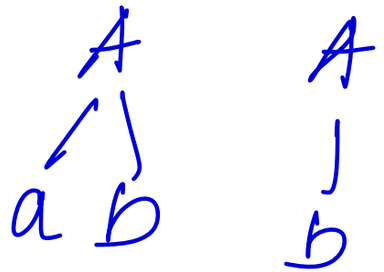
Non-terminal : S, A

terminals : c, a, b, d.

2. Predictive idea

- If input starts with c → use production $S \to cAd$
- For non-terminal A:
  - If next input is a
    use $A \to ab$
  - is b
    use $A \to b$.





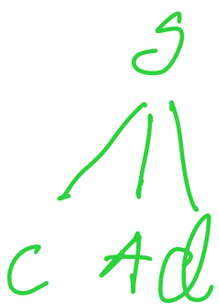
3. Recursive descent parsing procedures

Assume.

- lookahead → current input symbol
- match(x) → checks and consumes terminal x, else error.

Procedure for S

```
Procedure S()
{
  match('c');
  A();
  match('d');
}
```



Procedure for A

```
Procedure A()
{
  if(lookahead == 'a')
  {
    match('a');
    match('b');
  }elseif(lookahead == 'b')
  {
    match('b');
  }
```

else

error();

}.

Q. Write Non-recursive predictive parsing algorithm

FRANKLIN'S LECTURES

Algorithm

input:

- input string w $ (where $ is end marker)
- Grammer G
- predictive parsing table M
- start symbol S.

steps

1. Initialize.

• push $ and start symbol S onto the $tack

• Set input pointer to the first symbol of w$

2. Repeat until stack is empty.

let x = top of stack

let a = current input symbol.

3. if x is a terminal or $.

if x = a, pop x and advance input pointer

else, error.

4. If X is a non terminal

- look up entry $M[X, a]$
- if $M[X, a] = X \rightarrow Y_1\ Y_2 \ldots Y_n$.
  - pop X
  - push $Y_n \ldots Y_2\ Y_1$
- else error

5. If $x = \$$ and $a = \$$

parsing successful

Q. Construct a Recursive descent Parser for handling Arithmetic Express?

1.

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

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

2.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

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

Explanation of Grammer

E → E + T

T → T * F

F → identifiers & parenthesized expressions.

4. Recursive Descent parsing procedure.

assume:

lookahead → Current input Symbol

match → terminal.

error.

Procedure for E

```
Procedure E()
{
  T();
  Eprime();
}
```

Procedure of E'

```
Procedure Eprime()
{
    if (lookahead == '+')
    {
        match('+');
        T();
        Eprime();
    }
    else
        return;
}
```

// ε production.

Procedure – T

```
Procedure T()
{
  F();
  T prime();
}
```

Procedure – T'

```
Procedure T prime()
{
  if(lookahead == '*')
  {
    match('*');
    F();
    T prime();
  }
  else
    return;   // ∈ production.
}
```

Procedure F

```
Procedur F()
{
if(lookahead == 'id')
   match('id');
else if (lookahead == '(')
{
   match('(');
   E();
   match(',');
}
else
   error;
}.
```

FRANKLIN'S LECTURES

Step ahead with the right guidance

- Live + recorded classes
- Expert faculty pool
- Digitally illustrated study materials
- Free crash courses
- 24/7 doubt clearance & counselling support
- Self Evaluation Tests

EMI options available

GATE INTEGRATED
S6 SINDOOR BATCH

SCAN QR

9074 745 741
9633 962 277

Watch Now

Click Here

FRANKLIN'S LECTURES

THANK YOU