SR 22

St. Peter's Engineering College (Autonomous) Dullapally (P), Medchal, Hyderabad – 500100. QUESTION BANK						Dept.	:	CSE(AIML)
						Academic Year 2024-25		
Subject Code	:	AS22-66PC02	Subject	:	AUTOMATA THEORY & COMPILER DESIGN			
Class/Section	:	B. Tech.	Year	:	II	Semester	:	II

BLOOMS LEVEL					
Remember	L1	Understand	L2	Apply	L3
Analyze	L4	Evaluate	L5	Create	L6

*****

Q. No	Question (s)	Marks	BL	CO
	UNIT – IV			
1	Define Compiler.	1M	L1	C224.4
	Define Pre-Processor.	1M	L1	C224.4
	Define an Assembler.	1M	L1	C224.4
	Define First Symbol in CFG.	1M	L1	C224.4
	Define Follow Symbol in CFG.	1M	L1	C224.4
2	Discuss about First & Follow Symbols with examples.	3M	L2	C224.4
	Discuss about Token and Lexemes.	3M	L2	C224.4
	Discuss about Input Buffering.	3M	L2	C224.4
	Discuss about CFG, Derivation of a Grammar.	3M	L2	C224.4
	Discuss about of Parsing and Types of Parsing.	3M	L2	C224.4
3	Explain in detail about Complier Process.	5M	L4	C224.4
	Explain in detail about the role of Lexical Analyzer.	5M	L4	C224.4
	Explain in detail about Recursive Descent Parsing with Backtracking.	5M	L4	C224.4
	Explain in detail about any one Bottom-up Parsing.	5M	L4	C224.4
	Explain in detail about Phases of Compilation.	5M	L4	C224.4
4	**a)** Explain in detail about the role of Syntax Analyzer. **b)** Explain in detail about Predictive Parsing.	10M	L4	C224.4
	a) Explain in detail about LR (0) Parsing. **b)** Explain in detail about Lexical-Analyzer Generator Lex.	10M	L4	C224.4
	a) Explain in detail Scanning & Parsing. **b)** Explain in detail about LL(1) Parsing.	10M	L4	C224.4

Answers

1.

Define Compiler.

A compiler is a software that converts the source code to the object code. In other words, we can say that it converts the high-level language to machine/binary language. Moreover, it is necessary to perform this step to make the program executable. This is because the computer understands only binary language.

Define Pre-Processor.

A preprocessor is a program that processes input data to create output that is used as input for another program, such as a compiler. It is a key component of a compiler that generates input for it.

Define an Assembler.

An assembler is a type of computer program that takes in basic instructions and converts them into a pattern of bits that the computer's processor can use to perform basic operations.

Define First Symbol in CFG.

In a Context-Free Grammar (CFG), the first symbol is the start symbol, which is a special nonterminal symbol. The start symbol is used to derive strings in the language of the CFG.

Define Follow Symbol in CFG.

In a Context-Free Grammar (CFG), the Follow set of a nonterminal AAA (denoted Follow(A)\text{Follow}(A)Follow(A)) is the set of terminals that can appear immediately after AAA in any derivation.

2.

Discuss about First & Follow Symbols with examples.

CFG:

Fiut Sym: The set of Terminal Sym's that begin in string for a given Non-Terminal.

→ Let us Consider a CFG

S → aABC | bA

A → b

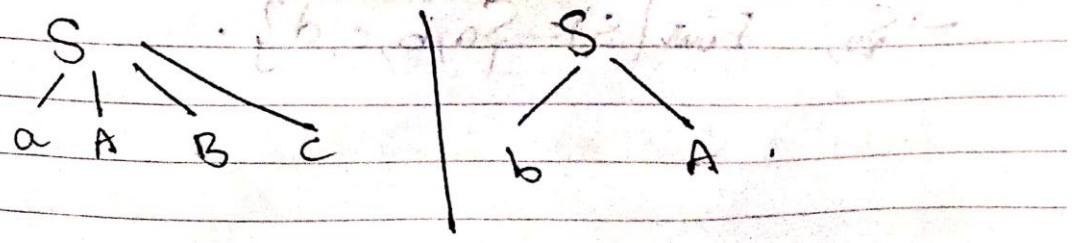
B → c

C → d

→ So, for Fiut(S) i.e Fiut of a Non-Terminal is the fiut Terminals Sym it is encountered in its production. i.e S → aABC or S → bA (S with a or b).

→ Here S is going to elaborate with some terminal So with what terminal it elaborate is called as Fiut(S).

(here S is elaborated with 'a' or 'b').



Page No.
Date

$First(S) = First(a) \cup First(b)$

$a \cup b$

$= \{a, b\}$

$First(A) = \{b\}$

$First(B) = \{c\}$

$First(C) = \{d\}$

eg 2) $S \to Bb \mid Cd$

$B \to aB \mid \epsilon$

$C \to cC \mid \epsilon$

$First(S) = First(B) \cup First(C)$

$\{a, \epsilon\} \cup \{c, \epsilon\}$

$\{a, c, d\}$

→ As we don't have terminals in the production of S, so we need to go with $First(B)$ & $First(C)$, the

~~First~~ $First(B) = \{\epsilon\}$

$First(B) = \{a, \epsilon\}$

$First(C) = \{c, \epsilon\}$

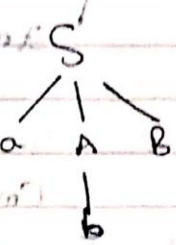
→ So, we go. ~~First(B~~ $First(S) = \{a, c\}$ & have $S \to Bb$, now if we substitute $B = \epsilon$ then $S \to b$ & for $S \to Cd$, $S \to Cd$, if we substitute $C = \epsilon$ then $S \to d$,

→ So, $First(S) = \{a, b, c, d\}$.

Follow Sym's in Compiler

→ Follow Sym's: The terminal Sym's that encounter after Completion of given Non-terminal.

(1.) S → aAB
A → b
B → c



→ The Follow Sym of A is
Follow(A) = First(B)
Follow(A) = {c}

- After Completion of 'A', the terminal Sym's that are going to encounter after the Completion of A that is what follow Sym means.
- That is whatever terminal Sym that follower after 'A'.
- So, in the para True, have the Call(A) i.e 'b', hereafter need to go with the Non-terminal whatever followed after Completion of A.
- So, in S → aAB, after Completion of A, we have B which is a Non-term, So now go with full First(B).
- First(B) - the terminal first terminal that encounter after for 'B' i.e {c}

→ Note: Before finding Follow, Find First Sym's.

Condition 2: Add $ as follow Sym for 1st Non-term, why?

→ For the given Gram, we need to choose '$' as the follow Sym of 'S'.

eg:- PASCAL Code

begin
if ( ) do
begin
Stmt
end
end $

Cmpnd stmt → begin stmt end
Stmt → if
for

Cmpnd stmt
begin stmt end

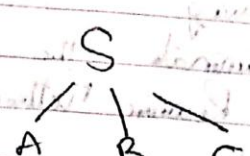
→ So in this eg (Pascal Code), the Code begin with a keyword begin & end with end.
- B/w that we have a Stmt.
- Inorder to identify whether there are Syntactically correct/not we construct a parse Tree.
- Here we have a Stmt in b/w begin & end.
- for Cmpnd stmt → begin stmt end

Cmpnd stmt
begin stmt end

→ How a Compiler Know that the Cmpnd stmt is finished & how it Knows that the prog is Syntactically Correct or not
- Inorder to identify these, Compiler will add spl Sym $ at the end of the prog.
- By this $ Symbol Compiler will come to know that the prog is ended.
→ This is encountered by the first Non-terminal, So Cmpnd stmt add $ as follow Sym for 1st Non-terminal.

Condition3: if $A \to \alpha B$
& Follow(B) = Follow(A).

$S \to ABC$
$A \to a$
$B \to b$
$C \to c/\epsilon$



Follow(C) = Follow(S)

→ Here S' is expanded initially A, B, C.
- After Completion of A, we are going to expand B, after Completion of B, we are going to expand C, after Completion of C there is nothing to expand
- This means S' is Completed.
- After Completion of S', what we are going to identify the terminal i.e. Follow(S).
- Means after Completion of C, Follow(S) is Completed.

∴ Follow(C) = Follow(S).

Condition4:
→ If the prod'n is of the form $A \to \alpha B \beta$
Follow(B) = {First(β) − ∈} ∪ {Follow(A)}.

$S \to ABC$
$A \to a$
$B \to b$
$C \to c/\epsilon$

Follow(B) = First(C)
{c, ∈}

→ So place ∈ in the C, from 3rd rule follow(S).

Fo(B) = {first(C) − ∈} ∪ fo{B}.
{fir(C) − ∈} - Remove 'G' from C

Discuss about Token and Lexemes.

In the context of compilers and lexical analysis, the terms token and lexeme refer to key concepts used in analyzing and processing programming languages.

1. Lexeme:

A lexeme is a sequence of characters in the source code that matches the pattern of a token. It is the actual string or character sequence that the lexical analyzer (lexer) identifies as a meaningful unit.

- Example:
  - In the code snippet int x = 10;, the lexemes could be int, x, =, 10, and ;. These are the raw character sequences that are identified during lexical analysis.

2. Token:

A token is a category or classification that the lexeme belongs to. It represents the syntactic role or function of the lexeme in the programming language. Each token corresponds to a class of strings that are treated as a single unit.

- Example:
  - For the lexeme int, the corresponding token would be KEYWORD.
  - For x, the token would be IDENTIFIER.
  - For =, the token would be ASSIGNMENT_OPERATOR.
  - For 10, the token would be INTEGER_LITERAL.
  - For ;, the token would be SEMICOLON.

Discuss about Input Buffering.

Input buffering is an important technique used in lexical analysis (scanning) for efficient handling of input data during the process of reading source code or text files. It is particularly useful when dealing with large amounts of input that need to be processed quickly, such as in the construction of compilers or interpreters.

Purpose of Input Buffering:

When the lexical analyzer (lexer) reads a source program, it must process the characters one at a time to identify tokens. However, reading characters one by one can be inefficient because:

- It involves frequent I/O operations (which can be slow).
- It might result in excessive overhead when determining the boundaries of lexemes.

Input buffering optimizes this process by loading multiple characters into memory in advance and then reading from this memory rather than from the original input source.

Discuss about CFG, Derivation of a Grammar.

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

- **N** is a set of non-terminal symbols.
- **T** is a set of terminals where **N ∩ T = NULL.**
- **P** is a set of rules, **P: N → (N ∪ T)***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
- **S** is the start symbol.

Example

- The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc.
- The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε
- The grammar ({S, F}, {0, 1}, P, S), P: S → 00S | 11F, F → 00F | ε

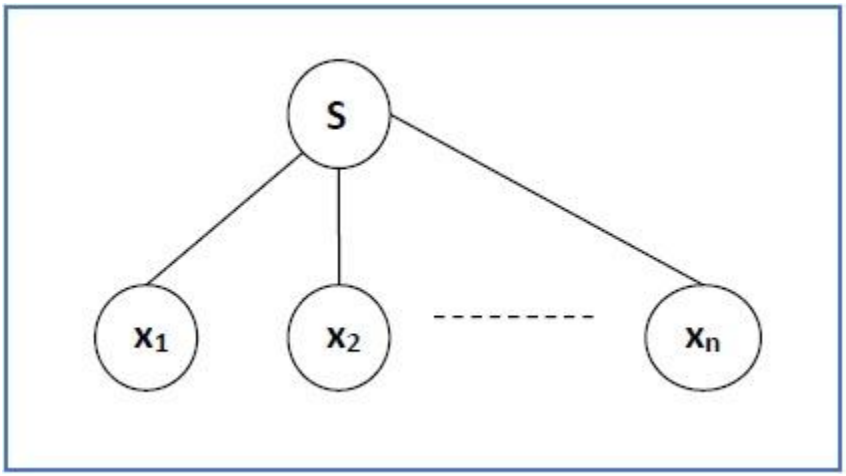
Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

Representation Technique

- **Root vertex** − Must be labeled by the start symbol.
- **Vertex** − Labeled by a non-terminal symbol.
- **Leaves** − Labeled by a terminal symbol or ε.

If $S \rightarrow x_1x_2 \ldots\ldots x_n$ is a production rule in a CFG, then the parse tree / derivation tree will be as follows −



There are two different approaches to draw a derivation tree −

Top-down Approach −

- Starts with the starting symbol **S**
- Goes down to tree leaves using productions

Bottom-up Approach −

- Starts from tree leaves
- Proceeds upward to the root which is the starting symbol **S**

Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

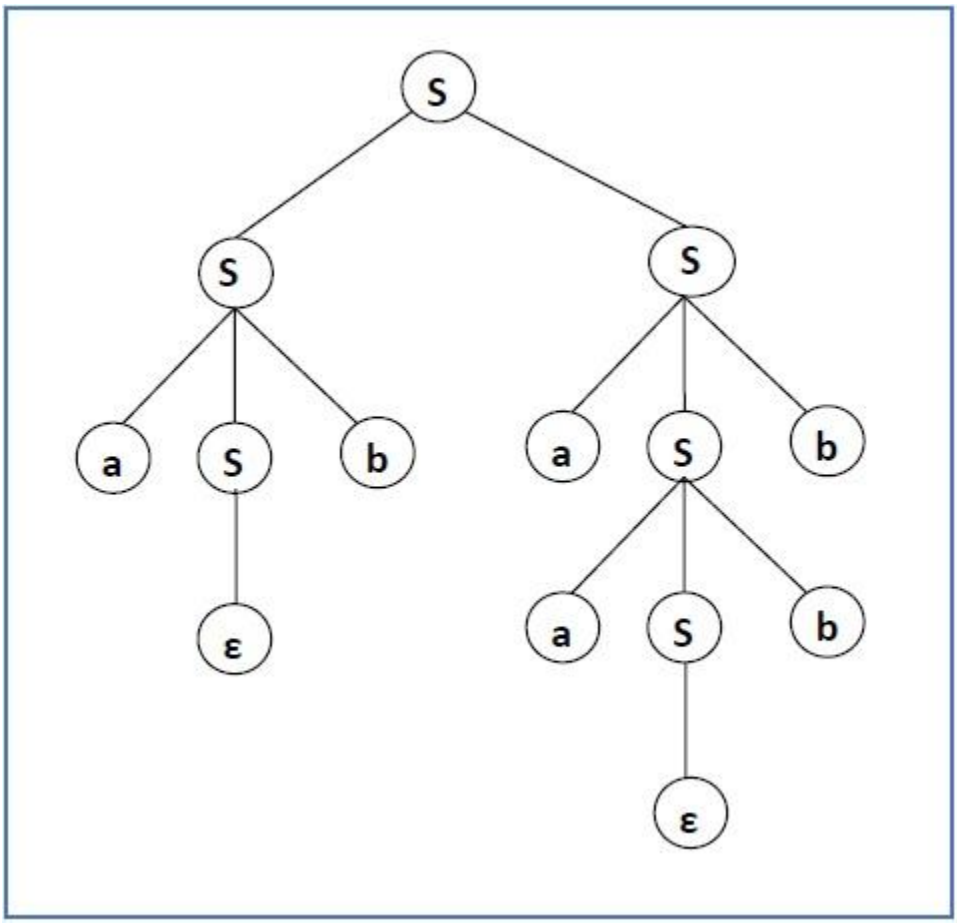
Example

Let a CFG {N,T,P,S} be

N = {S}, T = {a, b}, Starting symbol = S, P = S → SS | aSb | ε

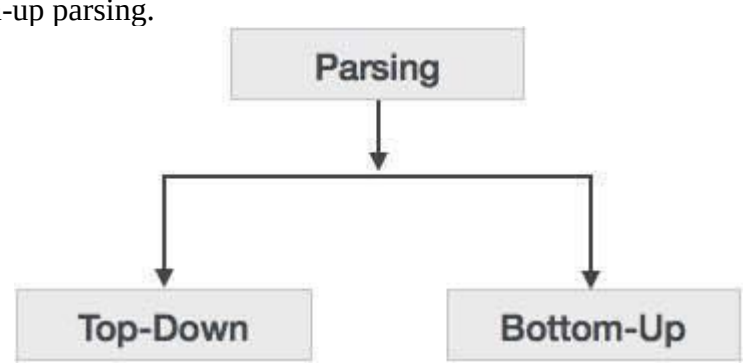
One derivation from the above CFG is "abaabb"

S → SS → aSbS → abS → abaSb → abaaSbb → abaabb



Discuss about of Parsing and Types of Parsing.

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing.



Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

- Recursive descent parsing : It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
- Backtracking : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Example:

Input string : a + b * c

Production rules:

S → E

E → E + T

E → E * T

E → T

T → id

Let us start bottom-up parsing

a + b * c

Read the input and check if any production matches with the input:

a + b * c

T + b * c

E + b * c

E + T * c

E * c

E * T

E

S

3.

Explain in detail about Complier Process.

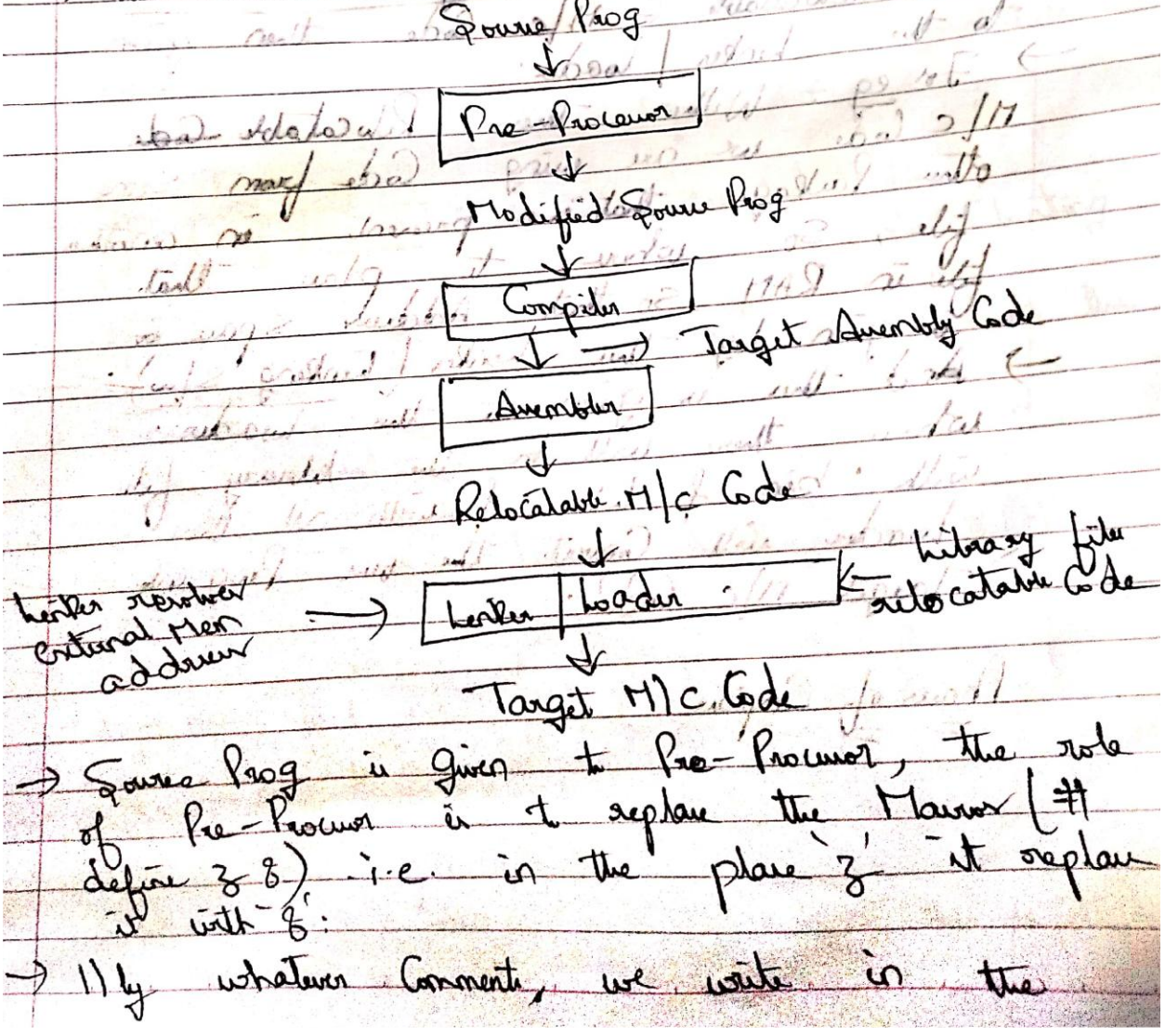
→ When we Compile a prog, the Compiler Convert the HLL into M/c Level lang

Eg:-

```
#include<stdio.h>
#define z 8
main() // Macros
{
  int x,y;
  x=z*y;
}
```

→ Now when we Compile this prog, this prog is Converted into M/c Code.
→ At that time, what kind of process work internally?

→ Now when



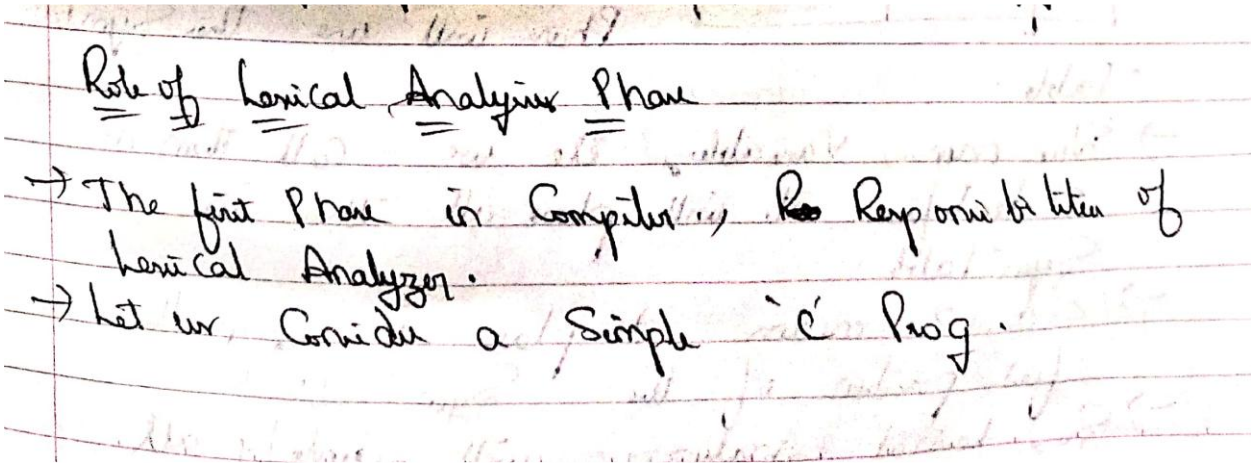
→ Source Prog is given to Pre-Processor, the role of Pre-Processor is to replace the Macros (#define z 8) i.e. in the place 'z' it replace it with '8'.

→ Ily whatever Comments, we write in the

Prog, all those Comments are then Pre-processor will eliminate all those Comments.
→ Then the 'Modified Source Prog' is given to the Compiler as an I/p. Now this Compiler Converts the 'Modified Source Prog into Target Assembly Code.
→ This Target Assembly Code is given as an I/p to a Prog Called Assembler.
→ This Assembler Converts this into Relocatable M/c Code.
→ This Relocatable M/c Code is a Prog Code that can be loaded anywhere in the mem.
→ The Relocatable M/c Code then given to the linker / loader.
→ For eg :- Within this Relocatable M/c Code we are using Code from some other Package that is present in another file, So where to place that file in RAM, So that Address space is given by the linker (linking S/w).
→ And this is given to the loader. where there will be the library file with bin/lib & with all these loader will Convert the our Prog into Target M/c Code.

Phases of Compiler

Explain in detail about the role of Lexical Analyzer.



Role of Lexical Analyzer Phase

→ The first Phase in Compiler is responsible for the working of lexical Analyzer.

→ Let us Consider a Simple 'C' Prog.

```
main()
{
int i=20, j=30, k;
k = i+j;
if(k>=50)
{
printf("hello");
}
}
```

o/p Generated by Lexical Analyzer

```
<main><(><)>
<{>
<int> <id,1><=><num,20>
<;><id,2><=><num,30>
<;><id,3><=><id,1><+><id,2>
<;>
<if>
.
.
.
.
.
```

→ Now this simple prog is given as an I/p to the lexical Analysis phase.
→ Here Lexical Analyzer need to read every character like main, m, a, i ..... (,) & identifiers, values, Symbols etc. into some tokens.
→ After reading 'i', here 'i' is an identifier so it need to maintain a Symbol Table.

Symbol Table

		Type	
1	i	int	---
2	j	int	--
3	k	int	---

→ In Symbol Table, it will store all the identifiers with its type (int/float etc) in the Table
⇒ Such that every To Phase will use this Symbol Table.
→ Fun name, Variables etc we call them as Identifiers it will store all these in Sym Table.
→ <id,1> means it store 'i' in the first position of the Sym Table.
→ So, Lexical Analyzer will identify all

then & place them in the Sym Table.
→ So, we need & that's the reason we write the prog in it with some rules.
→ Responsibilities of Lexical Analyzer

→ Removal of white spaces & Comments.
→ Count num of lines
→ Reading ahead (>= is one sym ahead than >)
>=, > (deciding >= & > are different)
→ Reading Constants
20 - num Constant
20.2 - floating pt num
Imp Note: Variable names should not start with a digit. (C lang).
→ Identify Keywords & Identifiers.
→ " Literals (eg: "hello")
→ " Comparison Operators (<, >, >=, <=).

Explain in detail about Recursive Descent Parsing with Backtracking.

Recursive Descent Parsing (Comes under Top-Down Parsing)

→ This is Top-Down Method, Backtracking
→ We will discuss simple CFG to understand Top-Down this.

$S \to cAd$
$A \to ab \mid a$ } CFG

S & A are Non-terminals.

→ Here we will write fun for each Non-terminal.
→ So, these fun's work

fun for S

```
Void S()
{
  loop to choose all production Choosen production
  S → X1 X2 .... Xk   (S has only one prode)
  for(i=1 to k)
  {
    if(Xi is nonterminal)
      Call Xi()
    else if(Xi = Current I/p Sym)
      advance input
      Sym to next
```

(→ As 'S' has only one production, this loop executes only one; S → cAd.
→ 'A' have 2 production, 'A' loop execute 2 times (A → ab, A → a).)

```
    else
      error occured
```

→ The Given production is the form $S \to X_1 X_2 \cdots X_k$, for $S \to cAd$, $X_1 = c$ i.e X_1, A, θ i.e X_2, d i.e X_3 ....

- The loop for (i=1 to k) execute for 'k' time based on X_i's.
- So, but if if (X_i is nonterminal), here $X_i = c$ which not a Non-terminal, So we need to go with 'else if (X_i = Current input Sym)' where 'c' is the same terminal.

- So we need to go with 'advance input Symbol to next' i.e. with 'A'.
- if this also don't match match then we need to go with the condition which is 'error occured'.

→ Here this we check for the all the production given, if one production is go back fails, go back & check with another (and) next production again if it fails go back & check with next production, as we go back & go with next production, this is called as Backtracking.

→ So for S → cAd
- 'c' is a Terminal, so go back & go with 'A'
- 'A' is a Non-term, call (i+1), so Substitute 'A' with 'a'

S → cad

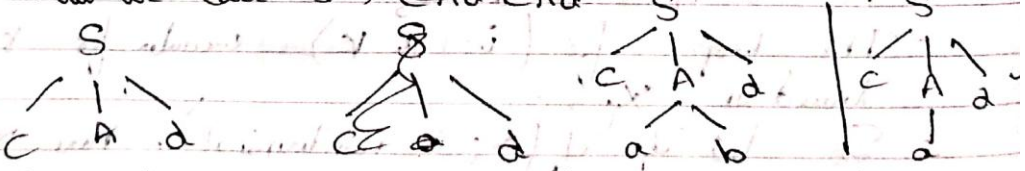
- Next we have we have 'd' will which is terminal.

S → cad

→ This is how the CFG's are evaluated using Backtracking Recursive Descent Parsing using with Backtracking.

→ What is Backtracking?

→ Now let us take the I/p Symbol 'cad'.
- First we call S → cAd cAd



- Now this one will expand as 'A' cad
A → ab or A → a as 'A' have 2 production
- Here 'call the fun' means expanding the Non-terminal.

→ Here for A ⟹ λ', there are 2 productions, go with first one, apply it, if it fails, take second prod, apply it, if it fails, go with next

- As we go back if one fails, its called as Backtracking.

→ So, the production S → cad is not matching, So our guess is rejected.

→ Why we need to write either Recursive Descent Parsing for Expressive Grammar

$$\left.\begin{array}{l} E \to TE' \\ E' \to +TE' \mid \epsilon \\ T \to FT' \\ T' \to *FT' \mid \epsilon \\ F \to (E) \mid id \end{array}\right\} \text{Think in CFG to check entire Lang.}$$

→ So here we need write the fun for every Non-Term.

→ Here here this is not a good practice practice; because everytime we need to go back if so for eg when we have a prod that don't match match with many prod's then how to deal everytime we need to go back, in such Cases lot of men is needed for Compiler to Compile one single prod'n.

→ So, these are the problems with this Parsing.

Explain in detail about any one Bottom-up Parsing.

LR(0) Parsing Problem

(I) For the given gram G, add an Augmented prod'n (now it's Augmented G').

eg:- $A \to aA/b$ (grammar)

$A' \to A$ → Augmented Prod'n.

$A' \to A$ (Augmented Prod'n)
$A \to aA$
$A \to b$

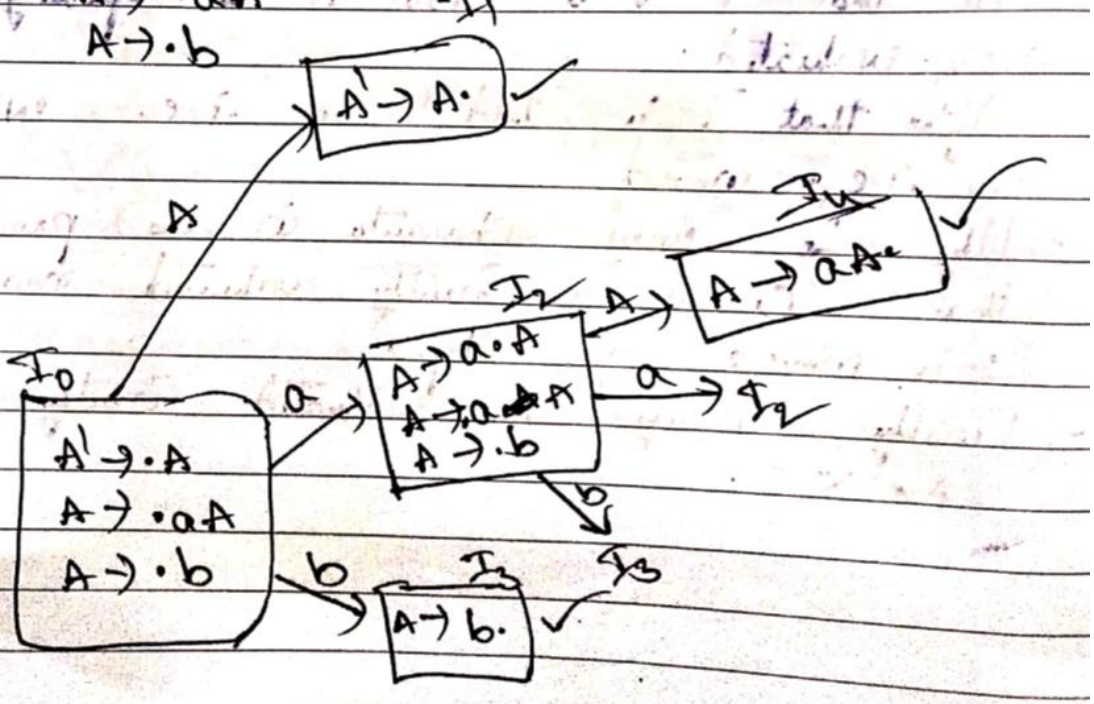
This is the final Augmented gram for the given gram.

(II) Now Start paring using LR(0)

Our augmented grammar add (·) at start to start paring.

Iq's → Sets

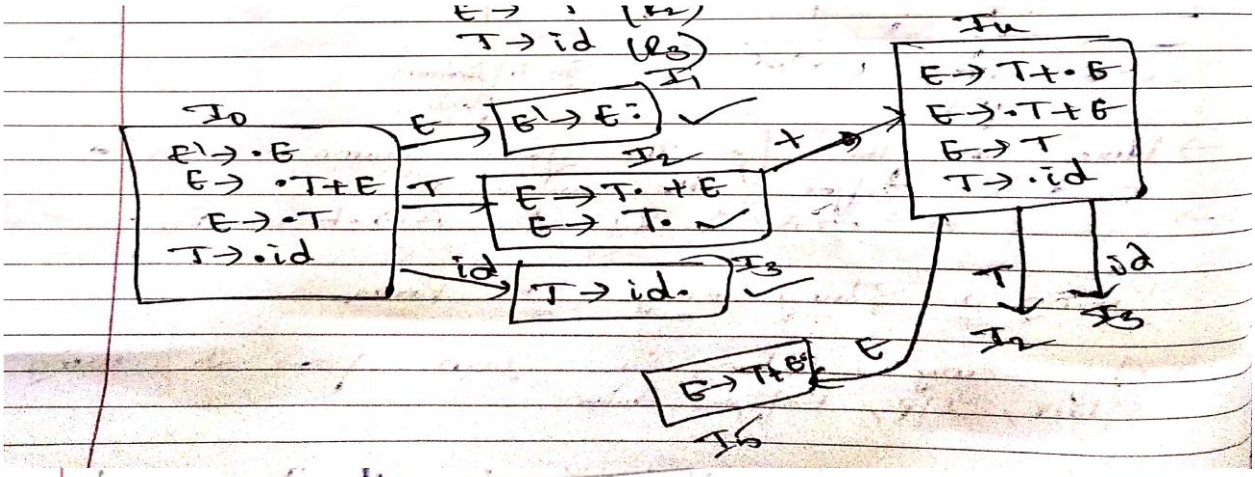
$A' \to \cdot A$
$A \to \cdot aA$
$A \to \cdot b$



	a	b	$	A
I_0	S_2	S_3		1
I_1			accept	
I_2	S_2	S_3		4
I_3	R_2	R_2	R_2	
I_4	R_1	R_1	R_1	

Shift, Action | Go to | Action part

(OR)



LR0 Parsing SR Clash

→ Let the given Grammar is

$E \to T+E \mid T$

$T \to id$

So the Augmented Grammar for the given grammar is

$E' \to E$

$E \to T+E$ (R_1)

$E \to T$ (R_2)

$T \to id$ (R_3)

I4

→ Here in I_2, we have 2 prod'ns, $E \to T.$ & $E \to T. + E$ & $E \to T. + G$ is still Continuing

- So, this is not LR(0) grammar because here there is one 2 choices (one is ended & another is Continuing).
- This is called as LR(0) grammar (Shift & Reduce Conflicts).

SR (Shift Conflict & Reduce Conflict).

	+	id	$	E	T
I_0		S_3		1	2
I_1			accept		
I_2	S_4/R_2	R_2	R_2		
I_3	R_3	R_3	R_3		
I_4		S_3		5	2
I_5	R_1	R_1	R_1		

Action | goto

→ Here in I_2, giving '+' we have a Conflict i.e. Shift & Reduce Conflict is there, by this we can say that the given grammar is not LR(0).

→ To resolve this issue, we need to go with SLR Parser.

SLR(1) | Simple LR

→ All steps are same except when we are adding rules in parse table.
→ We do the same process as previous parsing techniques, Creating Augmented Grammar then using this Augmented Grammar we are creating the Parser.
→ Using this we are going to fill the Parse Table.
→ We write only at follow parts.
→ Here first let us write the rules.

$R_1 \Rightarrow E \to T+E$
$R_2 \Rightarrow E \to T$
$R_3 \Rightarrow T \to id$

} Rules

→ Now let us write the Follow Sym's of Non-Term's E & T.

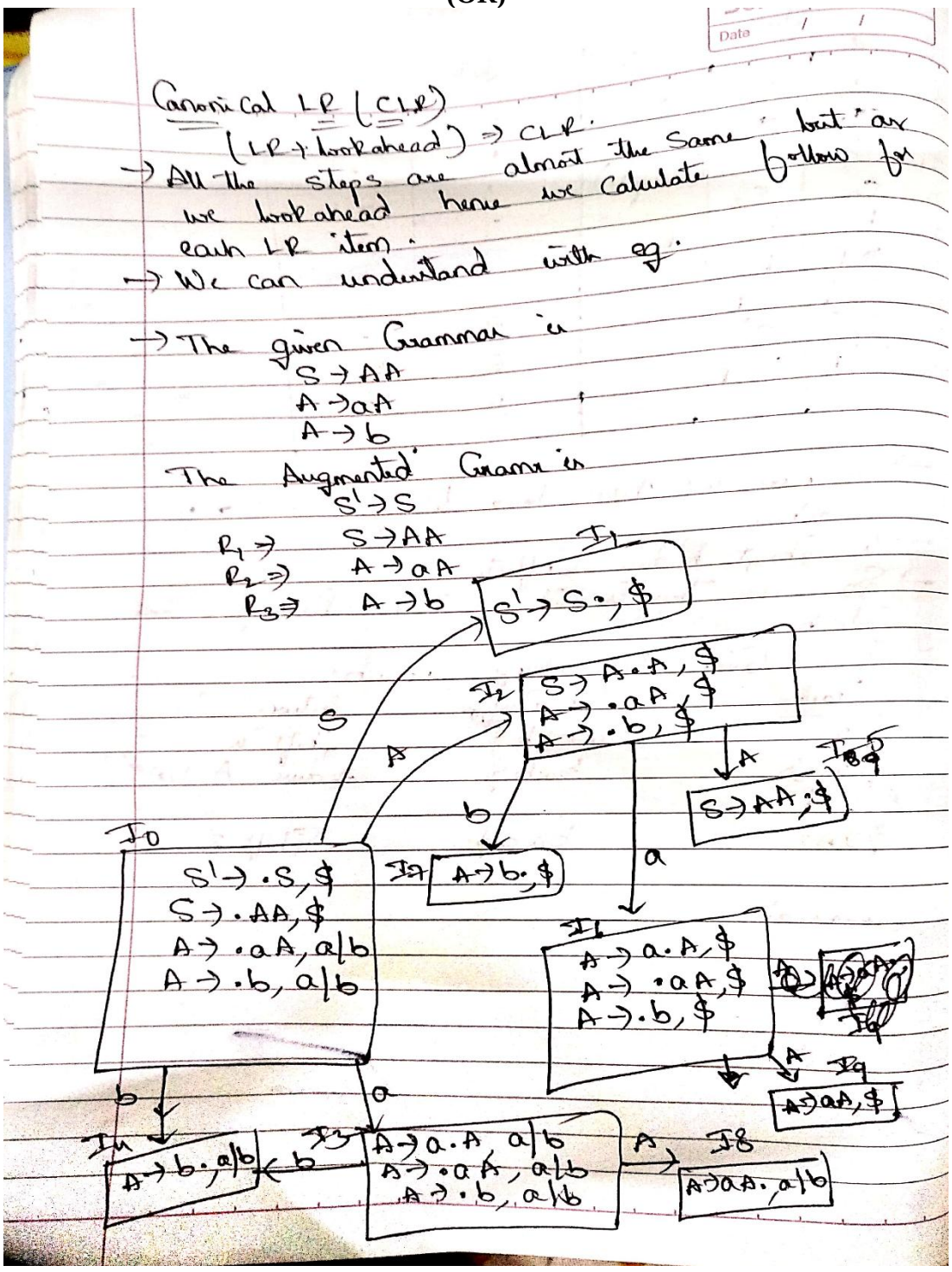
Follow(E) = \$,
Follow(T) = {+, \$}.

Table

	+	id	\$	E	T
I_0		S_3		1	2
I_1			accept		
I_2	S_4		R_2		
I_3	R_3		R_3		
I_4		S_3		5	2
I_5			R_1		

→ Here R_2, $E \to T$ is reduced in I_2. So write R_2 in I_2 at \$.
→ Like R_1, $E \to T+E$ " " I_5. So write R_1 in I_5 at \$.

(OR)


Canonical LR (CLR)
(LR + lookahead) ⇒ CLR.
→ All the steps are almost the same but we lookahead here we calculate both the for each LR item.
→ We can understand with eg.
→ The given Grammar is
S → AA
A → aA
A → b
The Augmented Gramr is
S' → S
R1 → S → AA
R2 → A → aA
R3 → A → b
I0
S' → .S, $
S → .AA, $
A → .aA, a/b
A → .b, a/b
I1
S' → S., $
I2
S → A.A, $
A → .aA, $
A → .b, $
I3
A → a.A, a/b
A → .aA, a/b
A → .b, a/b
I4
A → b., a/b
I5
A → b., $
I6
S → AA., $
I7
A → a.A, $
A → .aA, $
A → .b, $
I8
A → aA., a/b
I9
A → aA., $
S
A
b
a

Parsing Table

St's	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			accept		5
I_2	S_6	S_7			8
I_3	S_3	S_4			
I_4	R_3	R_3			
I_5			R_1		9
I_6	S_6	S_7			
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

→ If we observe clearly, we can notice many similar data like

$I_4 - I_7$, $I_3 - I_6$

$I_8 - I_9$

- If we remove them then we can call this as LALR.

(OR)

LALR(1):

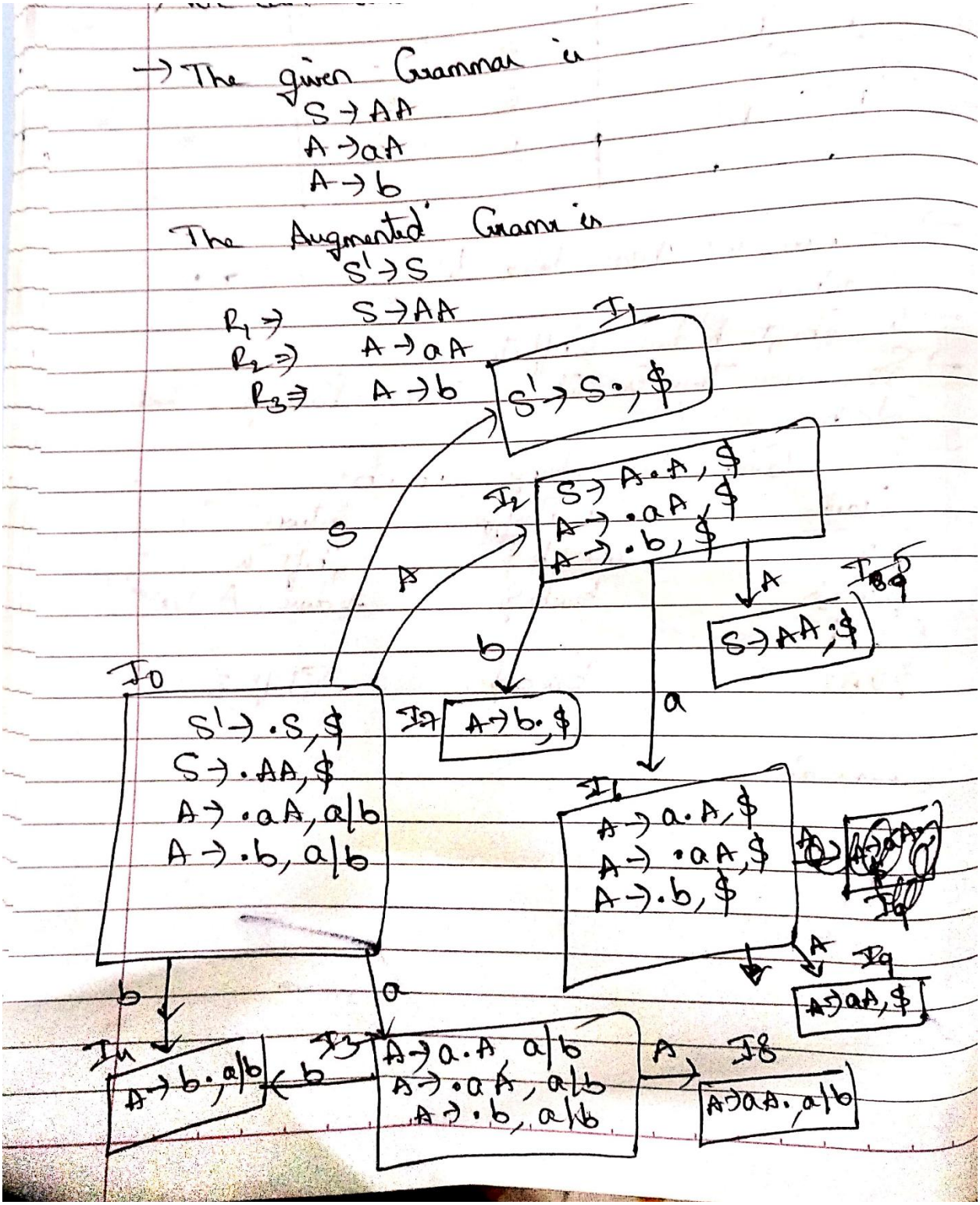
→ The given Grammar is

S → AA
A → aA
A → b

The Augmented Gramr is

S' → S

R1 → S → AA
R2 → A → aA
R3 → A → b



→ Lookahead LR

→ A Compressed / optimized version of CLR is LALR.

→ In the previous eg, or just by observing we can say.

(Solve the same CLR Problem.)

$I_4 - I_7 \Rightarrow I_{47}$

$I_3 - I_6 \Rightarrow I_{36}$

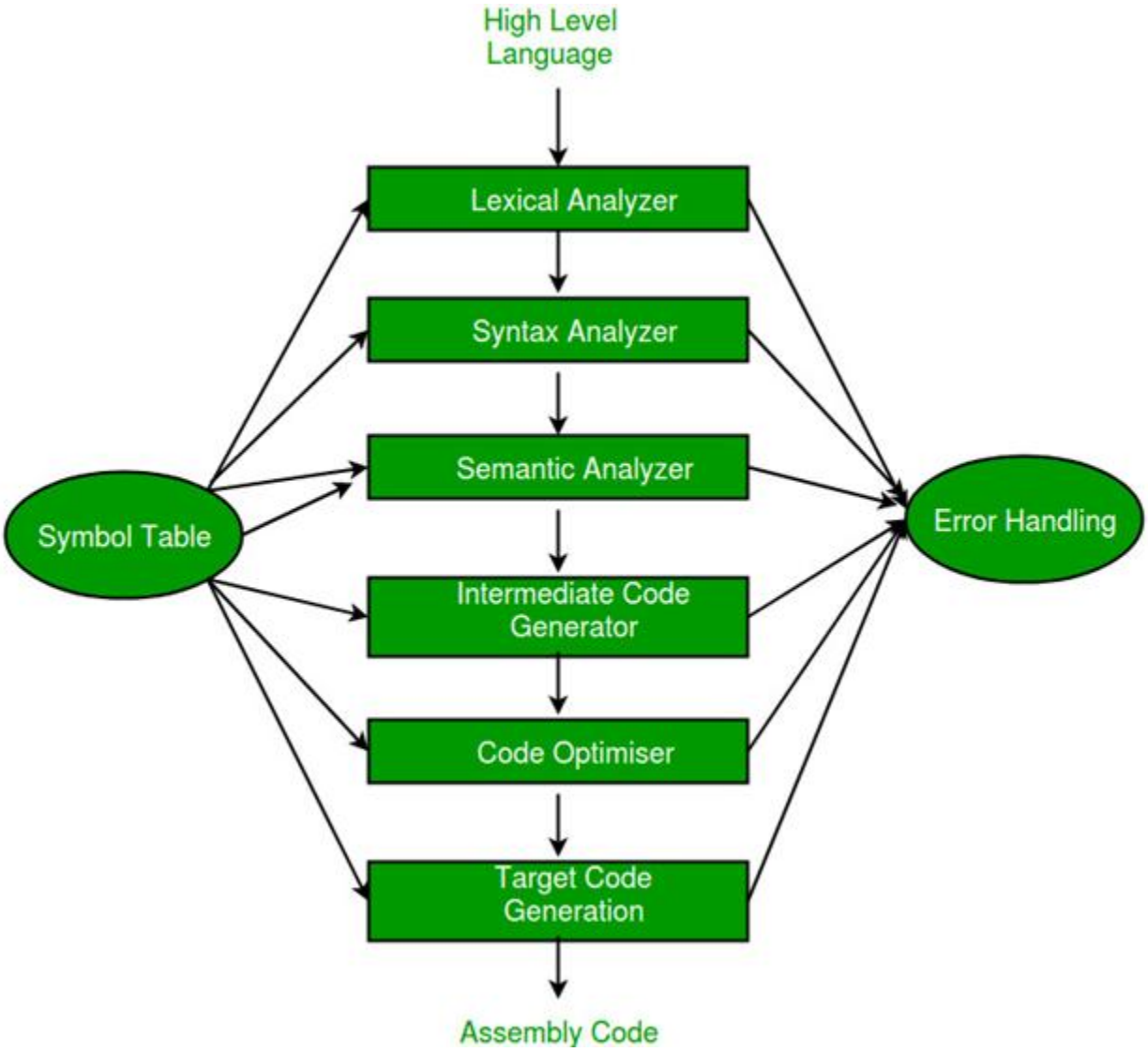
$I_8 - I_9 \Rightarrow I_{89}$

Update as

→ Based on them, let us fill the table.

St's	a	b	$	S	A
I_0	S_{36}	S_{47}		1	2
I_1			accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}		8	9
I_{47}	R_3	R_3	R_3		
I_5			R_1		
I_{89}	R_2	R_2	R_2		

Explain in detail about Phases of Compilation.



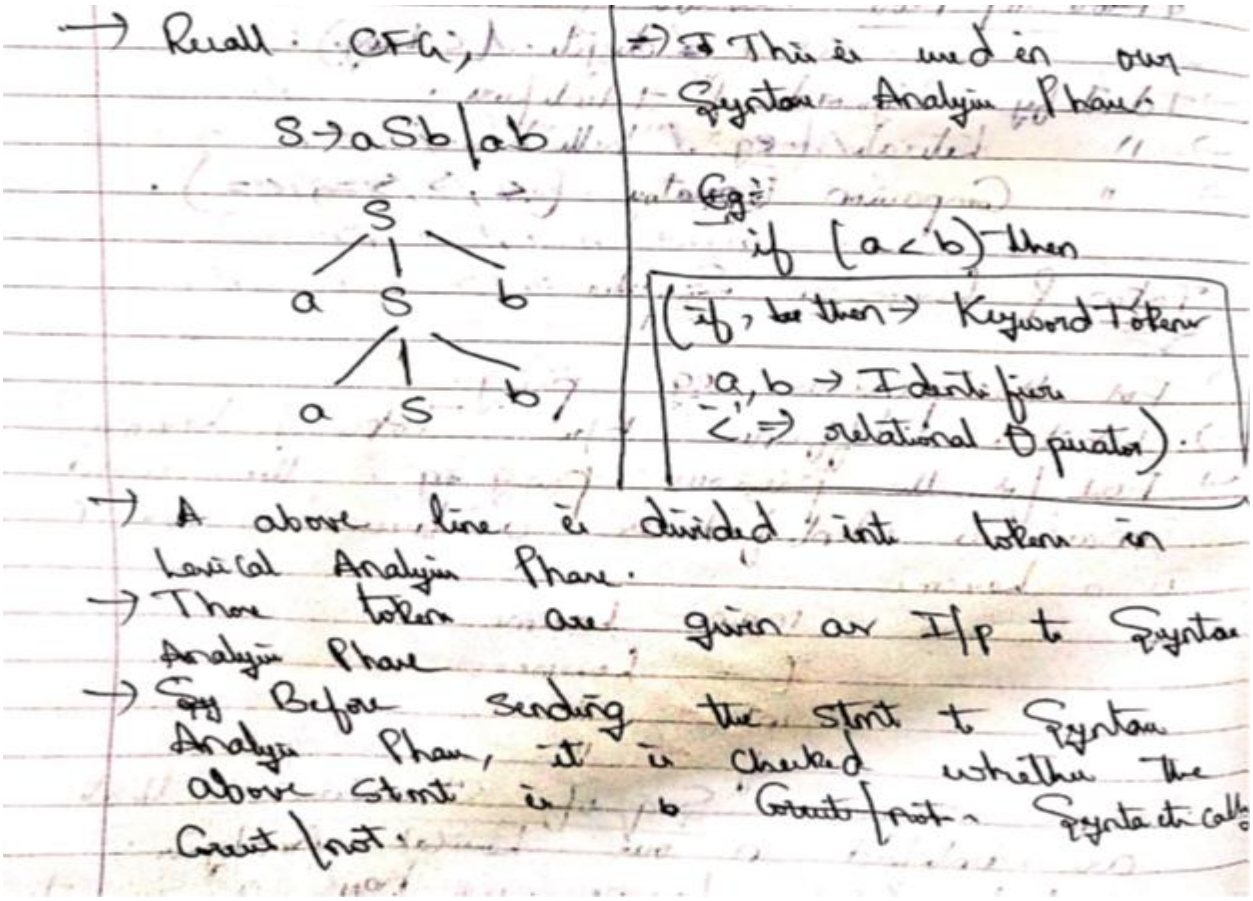
1. Lexical Analysis: The first phase of a compiler is lexical analysis, also known as scanning. This phase reads the source code and breaks it into a stream of tokens, which are the basic units of the programming language. The tokens are then passed on to the next phase for further processing.
2. Syntax Analysis: The second phase of a compiler is syntax analysis, also known as parsing. This phase takes the stream of tokens generated by the lexical analysis phase and checks whether they conform to the grammar of the programming language. The output of this phase is usually an Abstract Syntax Tree (AST).
3. Semantic Analysis: The third phase of a compiler is semantic analysis. This phase checks whether the code is semantically correct, i.e., whether it conforms to the language's type system and other semantic rules. In this stage, the compiler checks the meaning of the source code to ensure that it makes sense. The compiler performs type checking, which ensures that variables are used correctly and that operations are performed on compatible data types. The compiler also checks for other semantic errors, such as undeclared variables and incorrect function calls.

4. Intermediate Code Generation: The fourth phase of a compiler is intermediate code generation. This phase generates an intermediate representation of the source code that can be easily translated into machine code.
5. Optimization: The fifth phase of a compiler is optimization. This phase applies various optimization techniques to the intermediate code to improve the performance of the generated machine code.
6. Code Generation: The final phase of a compiler is code generation. This phase takes the optimized intermediate code and generates the actual machine code that can be executed by the target hardware.

4.

a) Explain in detail about the role of Syntax Analyzer. **b)** Explain in detail about Predictive Parsing.
a) Explain in detail about LR (0) Parsing. **b)** Explain in detail about Lexical-Analyzer Generator Lex.
a) Explain in detail Scanning & Parsing. **b)** Explain in detail about LL(1) Parsing.

a) Explain in detail about the role of Syntax Analyzer.



→ Here (a<b) is an EXPR (Expression), here we can write any expression in the place of (a<b) i.e. (a+b), (a-b) .... etc!

→ If we write in this manner

S → if (EXPR) then

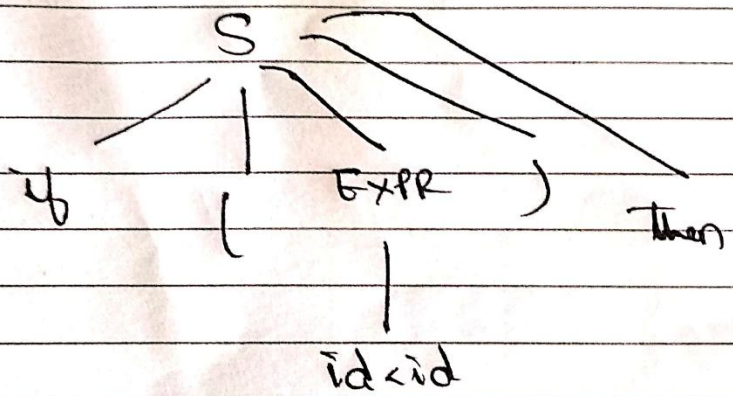
Now this is Syntactically Correct.

S → if (EXPR) then

Here S, EXPR are Non-Terminals & if, then are Terminals

~~G~~

EXPR → id+id | id<id



→ So the line `S → if (EXPR) then` is Syntactically Correct.

b) Explain in detail about Predictive Parsing.

Predictive Parsing

→ Recursive Approach without Backtracking.
→ We use lookahead over I/p Sym (to proceed without Backtracking)

Stmt → expr;
if (expr) stmt
for (optexpr; optexpr; optexpr)
other

→ Here 'stmt' is Non-Terminal & 'expr' is a Terminal for our Convenience purpose.
Stmt - Non-Term
expr - Terminal (for this eg).
if - Term
(- "
) - "
optexpr (optional expr) - Non-Terminal

→ So

optexpr → ∈
expr
for (; expr; expr) other
↑
lookahead

→ So, here 'lookahead' means, lookahead is a Variable

lookahead - Variable

- lookahead is going to pt to first I/p Sym for that we need to look for a loop stmt.
- So here 'for' in for (; ...) is a token given by lexical Analysis phase.
- Like '(', ';', expr, ';' ..... are tokens whatever given by Lexical Analyzer.

→ Like this 'lookahead' looks after one I/p Sym within our I/p.
→ So, this lookahead one I/p Sym ~~is~~ one after the other.
→ We can lookahead one Sym, lookahead 2 I/p Sym's, lookahead 4 Sym's, lookahead 5 Sym's.
→ So, this lookahead one I/p Sym, one after the other & write the logic.
→ Like this, this eliminates the Backtracking.
→ So, now the logic for this is

stmt → expr ;

```
void stmt()
switch (lookahead)
Case expr
    match(expr) match(;)
```

→ Here we write the logic with lookahead Sym.

→ Here we go with a match() fun! What is match() fun?

```
match(Terminal t) // match() applies for Terminal 't'
{
  if(lh=t) // if lookahead = terminal
  lh = nextTerminal // Then go with next Terminal.
  else
  repeat error
}
```

Coding (logic Continuous)

```
Case if
  match(if) match('(') match(expr)
  match(')') stmt(); break;
```

Page No.
Date

Case for
m(for).
Case for
match (for) match (()
optexpr () match (;)
optexpr () match (;)
optexpr () match (;)
optexpr () match (;)

→ So, By using 'lookahead', we are looking ahead every I/p Symbol here using 'lookahead'. This is going to check (the for (optexpr ....) loop & leaves all of expr Cases.

→ Because "The the match() fun logic Suits for for loop
match (Terminal t)
// Terminal t = for
if (lk = t) // if lookahead = to 'for' terminal then lookahead = next Terminal

→ So match (for) match ('(') then lk = next Terminal, So here this lookahead moves to the next I/p Sym '(' , optexpr()

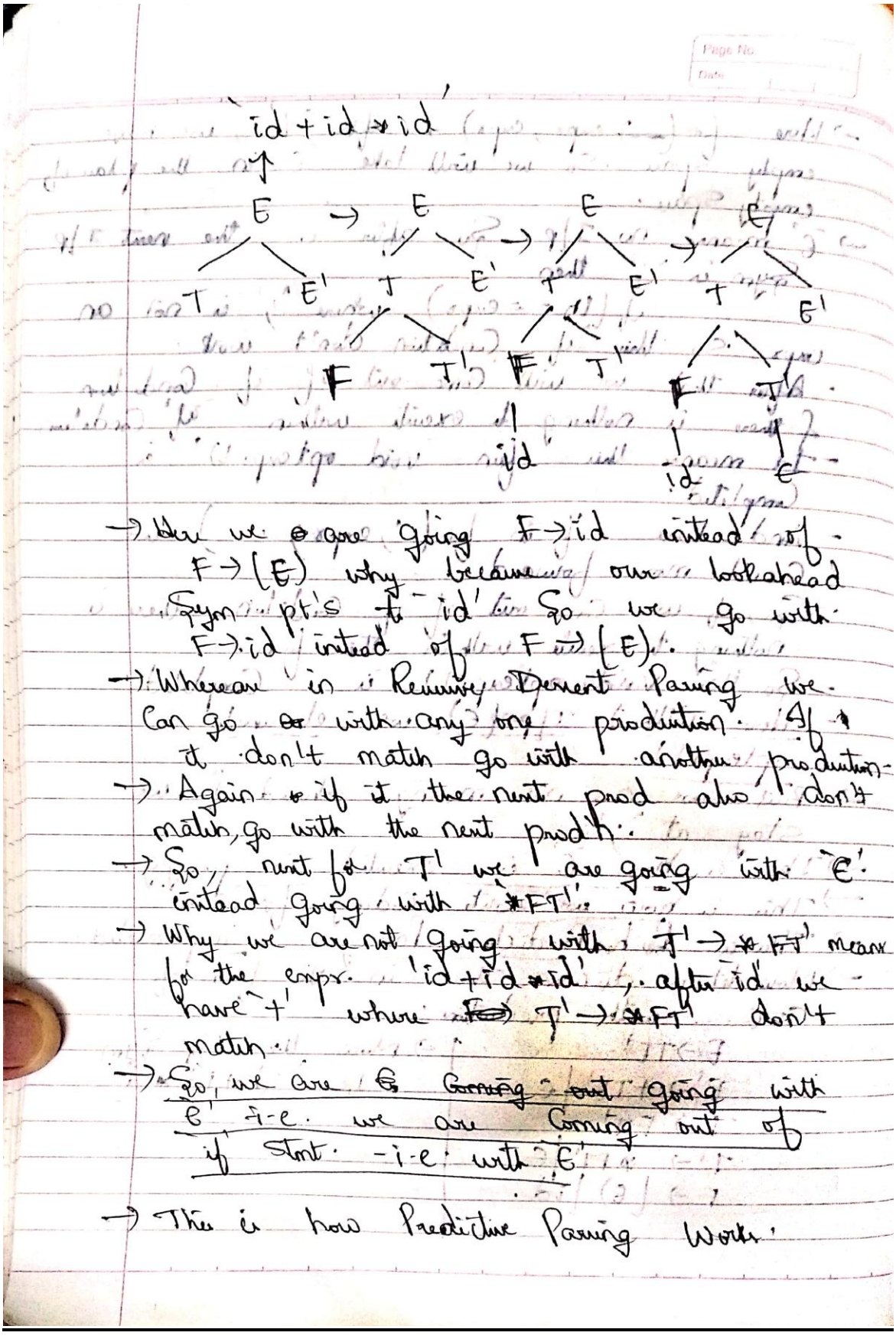
~~void~~ {

→ Here the lookahead pt's to 'for' which terminal then if the lookahead is terminal i.e 'for' then it moves to next I/p Sym '(' f ~~next to the ';' , next to the expr~~
→ match (() then the next I/p Sym is ';' . So which optexpr
void optexpr()
if (lk = expr)
match (expr)

→ Here for (—; expr; expr), after '(', we have empty space. So we will take 'ϵ' in the place of empty space.
→ 'ϵ' means no I/p. So after 'ϵ' the next I/p Sym is ';' then
- if (lh == expr) where ';' is not an expr, so this 'if' Condition don't work.
- After that we will come out of if Condition if there is nothing to execute within 'if' Condition
- It means this 'fun void optexpr1()' is Completed.
- And here 'f' in for(', expr; ...) we cannot move forward.
- So, if we come out of if Condition, there is nothing to execute within this fun.
- So this 'void optexpr1()' is Completed.
- Here without I/p (ϵ) we eliminate optexpr1().
→ But we are not moving ahead if we stay at ';'.
→ This is how 'ϵ' is executed in prog'g.
→ This is how we write Coding for 'ϵ'.
→ This is how Predictive Parsing is implemented.
→ Now let us discuss this in the form of CFG with production.

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

→ Now the I/p Sym be 'id + id * id'



→ Here we are going F→id instead of F→(E) why because our lookahead Sym pt's to 'id' so we go with F→id instead of F→(E).

→ Whereas in Recursive Dement Parsing we can go with any one production. If it don't match go with another production.

→ Again if it the next prod also don't match, go with the next prod'n.

→ So, next for T' we are going with ε' instead going with *FT'.

→ Why we are not going with T'→*FT' means for the expr. 'id+id*id', after id we have '+' where T'→*FT' don't match.

→ So, we are going with ε' i.e. we are coming out of start. i.e. with ε.

→ This is how Predictive Parsing Works.

a) Explain in detail about LR (0) Parsing.

LR(0) Parsing Problem

(I) For the given gram G, add an Augmented prod'n (now it's Augmented G).

eg:- $A \to aA/b$ (grammar)

$A' \to A$ → Augmented Prod'n.

$A' \to A$ (Augmented Prod'n).

$A \to aA$

$A \to b$

This is the final Augmented gram for the given gram.

(II) Now Start parsing using LR(0)

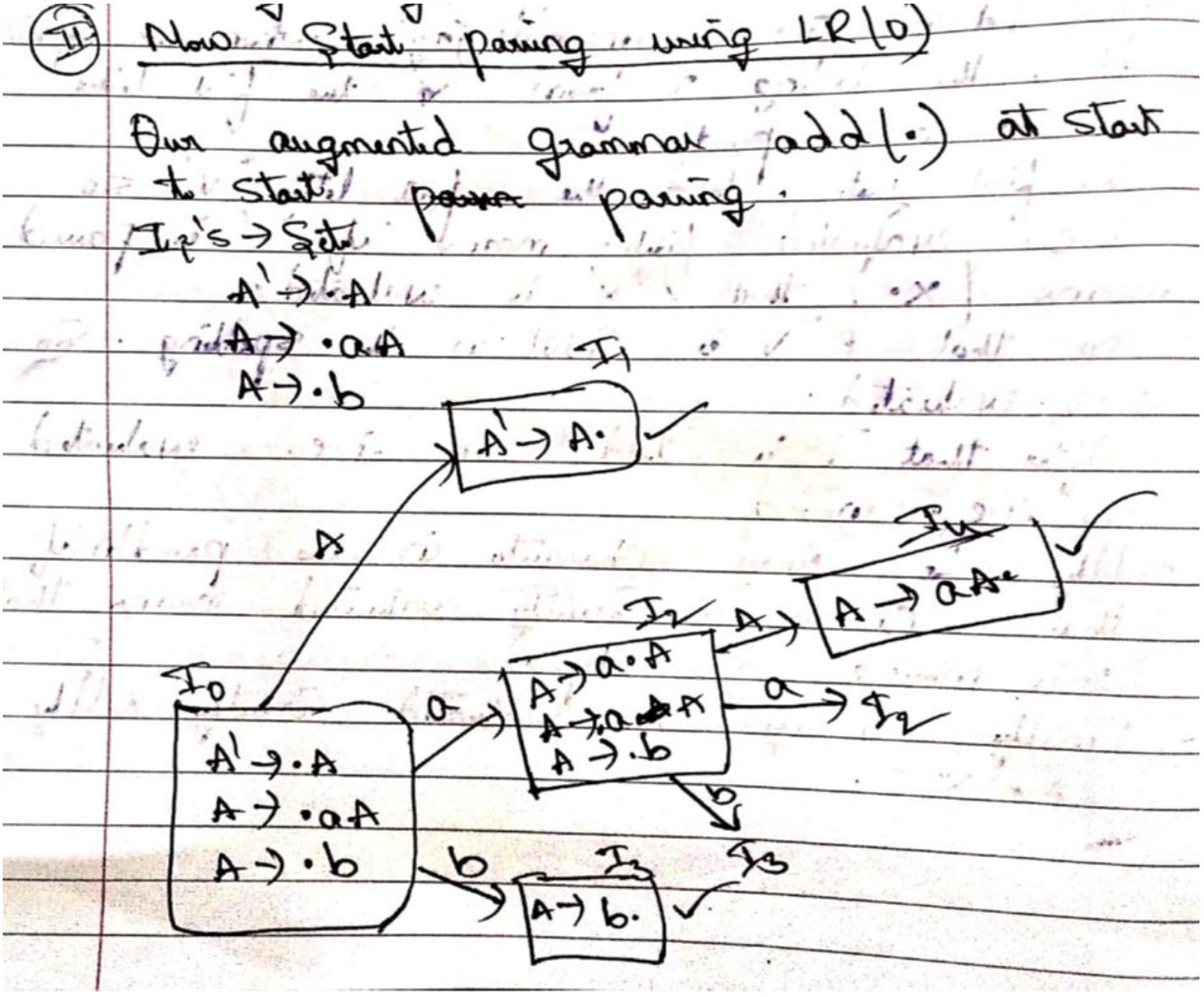
Our augmented grammar add (·) at start to start parsing.

It's → Sets

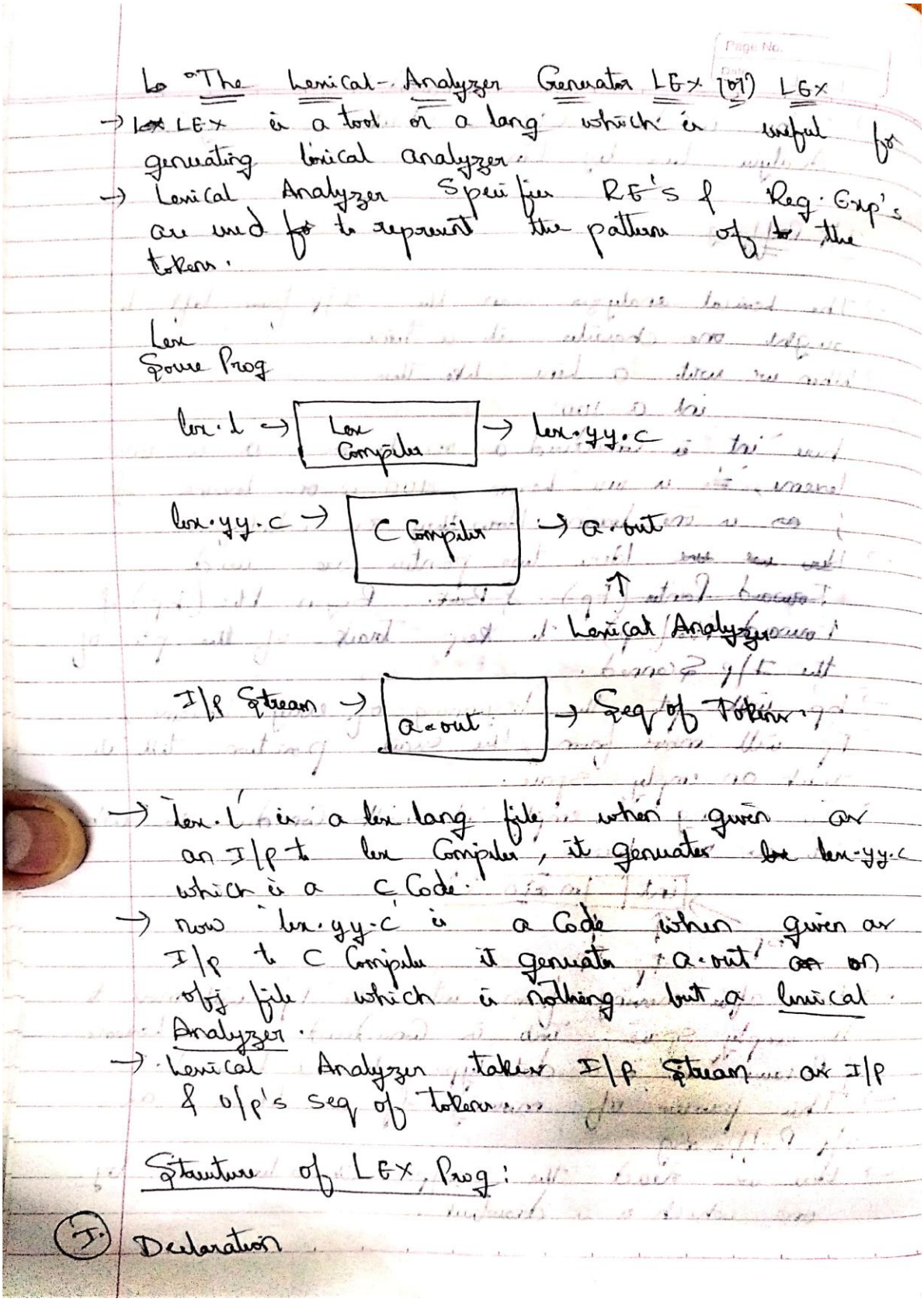
$A' \to \cdot A$

$A \to \cdot aA$

$A \to \cdot b$



b) Explain in detail about Lexical-Analyzer Generator Lex.



→ "The Lexical-Analyzer Generator LEX (or) LEX
→ ~~Lex~~ LEX is a tool or a lang which is useful for generating lexical analyzer.
→ Lexical Analyzer Specifies RE's & Reg. Exp's are used to represent the pattern of the tokens.

→ lex.l is a lex lang file, when given as an I/p to lex Compiler, it generates lex.yy.c which is a C Code.
→ now lex.yy.c is a Code when given as I/p to C Compiler it generates 'a.out' or obj file which is nothing but a lexical Analyzer.
→ Lexical Analyzer takes I/p Stream as I/p & o/p's seq of tokens.

Structure of LEX Prog:

(1) Declaration

(II) Translation rules (III) Auxiliary Fun / Subroutines.
(User defined fun's)

(I) Declaration
→ Used mainly for declaring C variable / Constants.
eg:- %{
int a, b;
float x = 10.0;
%}

Enclosed in this way
%{
:
%}

(II) Translation Rule:
→ Each rule is specified not in the formal
pattern {Action}
Reg. Expr ↑ ↑ C Lang Stmt.

eg:- %% (Start with %%)
pattern1 (Action1)
:
patternn (Actionn)
%%

(III) Auxiliary Fun's :- (~~Optional~~ Optional)
→ Any additional Functions are written here
Structure
declaration
%%
translation rules
%%
• User Subroutines / auxiliary Functions.

Eg:- (Lab Prog)

Prog to Capitalize the given Comments using LEX.

```
%{
#include<stdio.h>
#  "  <ctype.h>
int c;
void display(char *);
%}
letter [a-z]
Com [//]
%%
{Com} {c=1;}
{letter} {if(c==1) display(yytext);}
%%
main()
{
yylex();
}
void display(char *s)
{
int i;
for(i=0; s[i]!='\0'; i++) // '\0' is end of the string.
  printf("%c", toupper(s[i]));
}

int yywrap()
{
return 1;
}
```

→ Declaration

→ defining

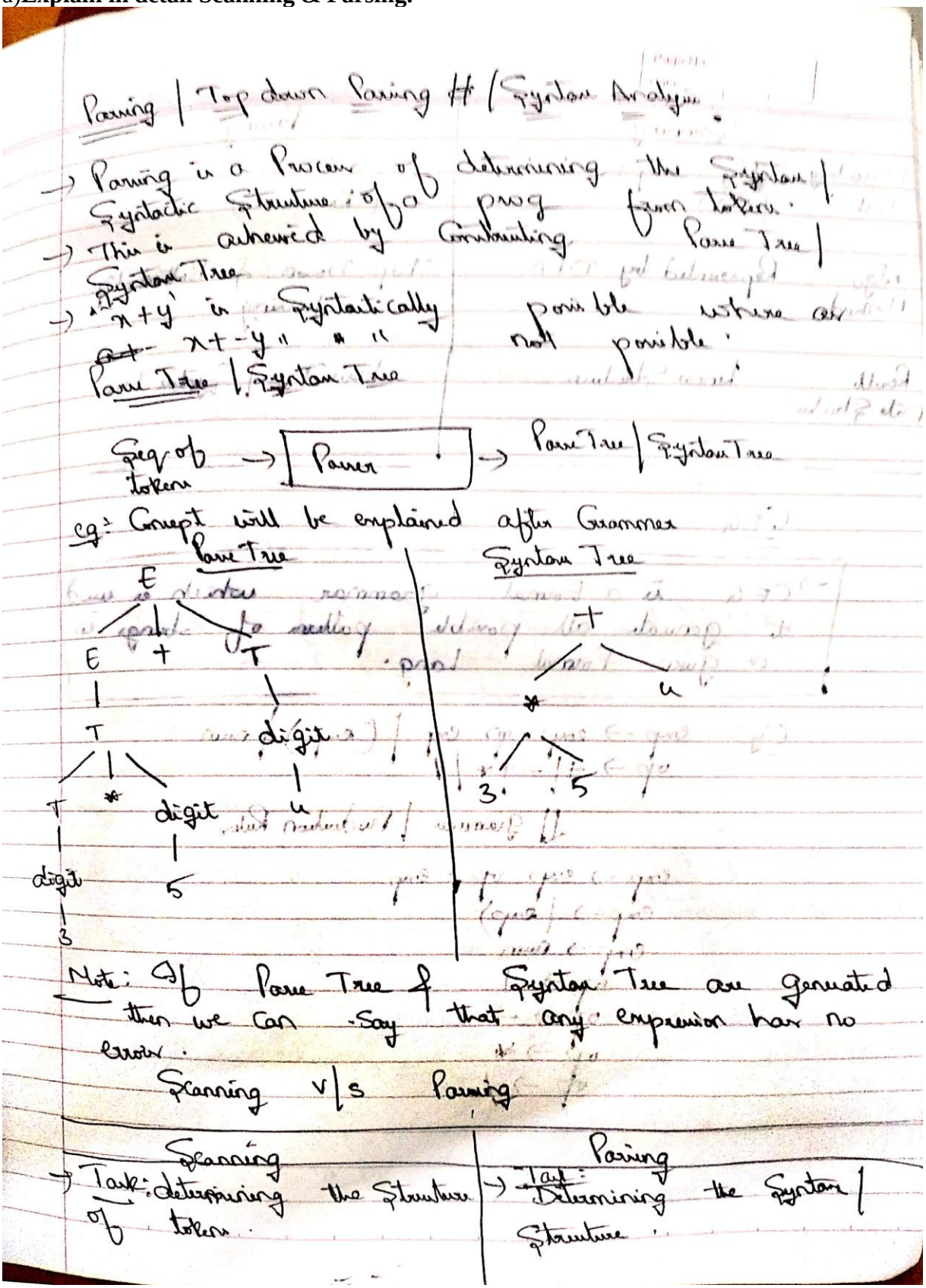
Here the execution starts

→ Whatever we write will be Converted into Uppercase

I/P:-
// Test

O/P:-
TEST.

a)**Explain in detail Scanning & Parsing.**



	Scanning	Parsing
Deriving Tool:	Reg. Expr (FA)	CFG (Parse Tree)
Algo Method	Represented by DFA	Top-Down & Bottom-Up Parsing
Result Data Structure:	Linear Structure	Parse Tree / Syntax Tree

CFG

→ CFG is a Formal grammar which is used to generate all possible patterns of strings in a given Formal Lang.

Eg: exp → exp op exp | (exp) | num

op → + | − | * | /

⇓ Grammar / Production Rules

exp → exp op exp

exp → (exp)

exp → num

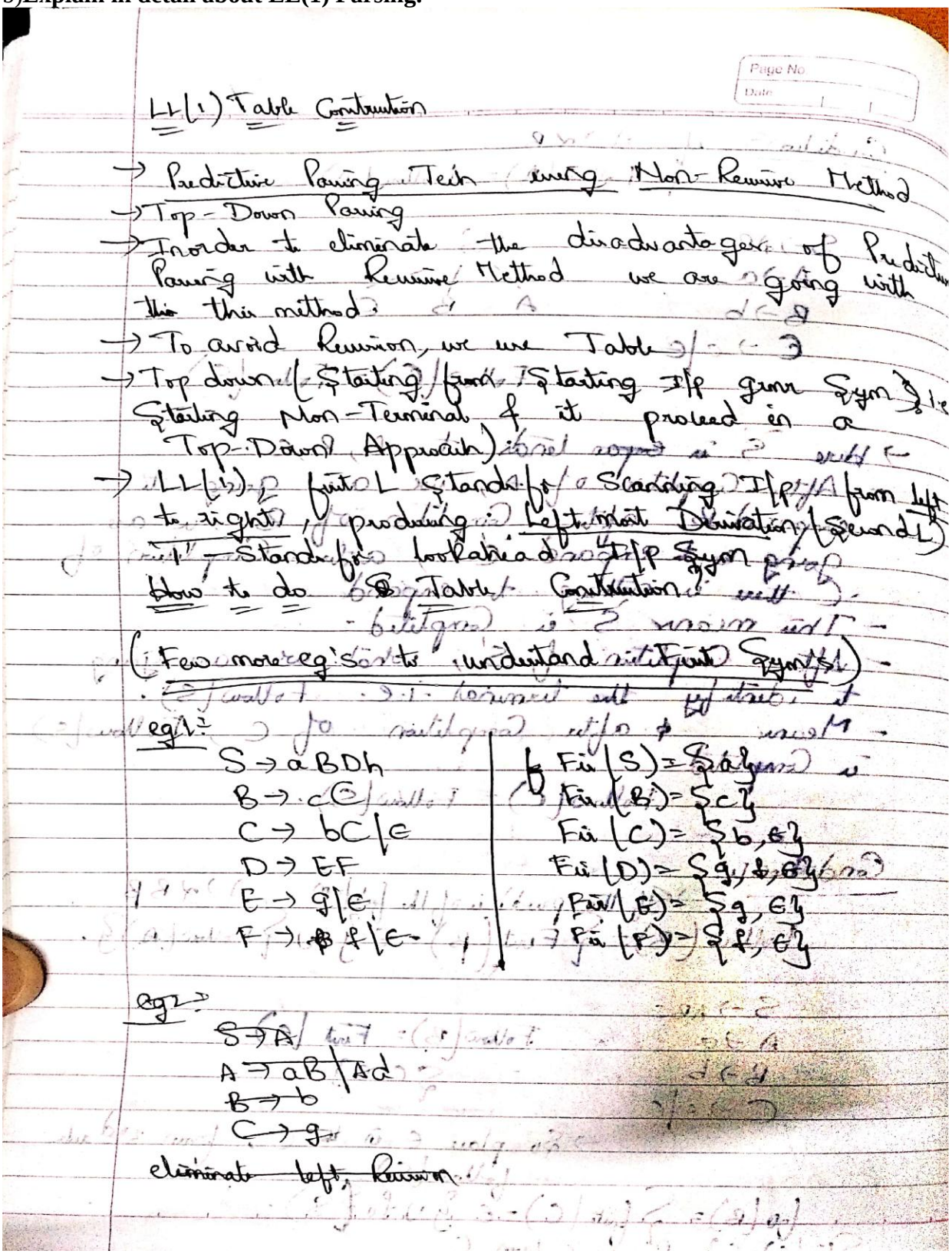
op → +

op → −

op → *

op → /

b)**Explain in detail about LL(1) Parsing.**



LL(1) Table Construction

→ Predictive Parsing Technique is a Non-Recursive Method
→ Top-Down Parsing
→ Inorder to eliminate the disadvantages of Predictive Parsing with Recursive Method we are going with this this method.
→ To avoid Recursion, we use Table
→ Top down (Starting from Starting I/p given Sym) i.e Starting Non-Terminal & it proceed in a Top-Down Approach)
→ LL(1) here L Stands for Scanning I/p from left to right, producing Left most Derivation & second 1 - Stands for lookahead I/p Sym

How to do Table Construction

(Few more eg's to understand with First Sym)

eg1:-

S → aBDh
B → cC
C → bC | ∈
D → EF
E → g | ∈
F → f | ∈

First(S) = {a}
First(B) = {c}
First(C) = {b, ∈}
First(D) = {g, f, ∈}
First(E) = {g, ∈}
First(F) = {f, ∈}

eg2:-

S → A
A → aB | Ad
B → b
C → g

eliminate left Recursion

Eg 2:

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

$F(E) = First(T) = First(F) = \{id, (\}$

$F(E') = \{+, \epsilon\}$

$F(T') = \{*, \epsilon\}$

$First(F) = \{id, (\}$

Eg's on Follow & Sym's

	First	Follow
$S \to ABCDE$	$\{a, b, c\}$	$\{\$\}$
$A \to a \mid \epsilon$	$\{a, \epsilon\}$	$First(B) = \{b, \epsilon\} = \{b, c\} = First(B)$
$B \to b \mid \epsilon$	$\{b, \epsilon\}$	$First(C) = \{c\}$
$C \to c$	$\{c\}$	$d, e, \$$
$D \to d \mid \epsilon$	$\{d, \epsilon\}$	$e \$$
$E \to e \mid \epsilon$	$\{e, \epsilon\}$	$\$$

~~First(A) = First~~

$Follow(A) = First(B) \cup First(C) = \{b, c\}$ ($\because B \to \epsilon$, if we Substitute that in ABCDE), So we get $First(C)$ i.e ACDE $(B \to \epsilon)$

$Follow(C) = First(D)$, and if we Substitute $D \to \epsilon$ in ABCDE we get ABCE if next to C we have E, So ~~Follow(C) = First(D) ∪ First(E)~~

$Fo(C) = First(D) \cup First(E) \cup First(\$)$ because $D \to \epsilon$, $E \to \epsilon$. So ~~First($)~~ Follow($)

$Fr(C) = First(D) \cup First(E) \cup Fo(\$)$.

Follo(D) = Foll Fir(E) ∪ Fo(S), because if we substitute E' → ∈ then it is Foll(S).

eg 2:

	First	Follow
E → TE'	id, (	$,)
E' → +TE' | ∈	+, ∈	$,)
T → FT'	id, (	+, $,)
T' → *FT' | ∈	*, ∈	+, $,)
F → id | (E)	id, (	*, +, $,)

→ Follow(E) = $
– Here there is no E in any prod'n except in last prod'n, (So for (E), after E we have the terminal ")" So ")" as E is first sym, add $.
So Foll(E) = $,)

→ Fo(E') = Foll(E) = $,).
– for E', for E in E → TE', here after (E) there is nothing. So, Foll(E') = Foll(E).
– And for E' → TE', so Foll(E') = Fol(E'), eliminate those conditions.

→ Foll(T) = Fir(E') ∪ Foll(E') = +, $,)
– As E' → +TE', so after T we have E', so First(E') = +, ∈
– If we place E' → ∈ in E' → +TE' then E' → +T, so Follow(T) = Follo(E')

→ Fol(T') = Foll(T) (T → FT')

→ Foll(F) = Fir(T') ($\because$ T → FT')
= {*, ∈}

- In place of F = in A→ T = ∈, place it in T → T → FT' → F, i.e Foll(T)

So: Foll(F) = Fir(T') ∪ Foll(T)
= {* ∈} ∪ {+, $,)}
= {*, +, $,)}

(Continue with LL(1) Table).

LL(1)

→ Standard for Scanning I/p from left to right
|| " Producing Left Most Derivation
1 → 1 Lookahead I/p Sym.

	Fiist	Follow		id	+	*	(	)	$
E→TE'	id(	$)	E	E→TE'			E→TE'		
E'→+TE'/∈	+∈	$)	E'		E'→+TE'			E'→∈	E'→∈
T→FT'	id(	+)$	T	T→FT'			T→FT'		
T'→*FT'/∈	*∈	+)$	T'		T'→∈	T'→*FT'		T'→∈	T'→∈
F→id/(E)	id(	*+)$	F	F→id			F→(E)		

For Each prod'n A→α do the following
(I) For each terminal a in Fiist(α), add A→α to M[A,a]
(II) If ∈ is in Fiist(α), add A→∈ to all the follow Sym's.

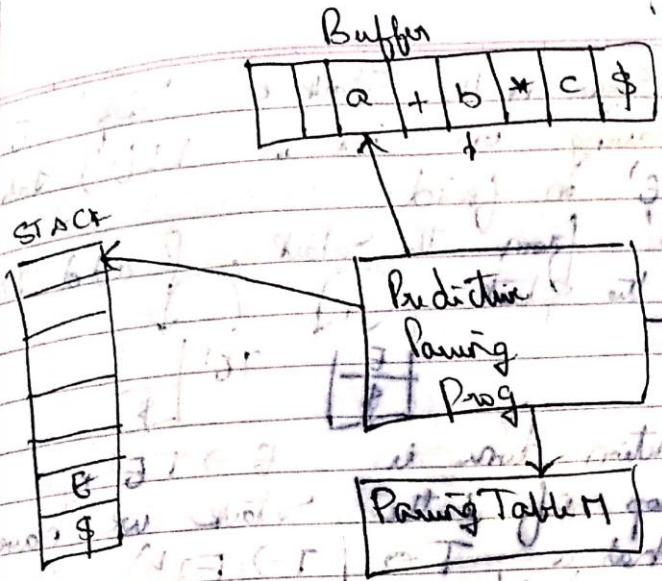
→ Here I, st's that for prod'n $A \to \alpha$, for each terminal, a, in first(α) i.e. for 'id' in $E \to TE'$, add for id & (add $E \to TE'$.

→ II, if ϵ is in first(α) add $A \to \epsilon$ to all the follow Sym's.
i.e if we have ϵ', eg: for $E' \to +TE'$ we have $+E'$, So for $,) add $E' \to \epsilon$. for the follow Sym's i.e. $,).

→ for for $F \to id \mid (E)$, we added $F \to id$ for id & $F \to (E)$ for '('. because we got $F \to id$ for id & $F \to (E)$ for '('. (intra)
→ i.e we derived Fir(F) = id by $F \to id$ & Fir(F) = (E) by $F \to (E)$.

Predictive Parsing: LL(1) Automaton?

→ So, LL(1) parsing accept the I/p Sym's, Strings.
→ So Here for accepting the I/p String, we are using a STACK & we need to write a prog to accept the String. So, what Condition we write & for this prog.
→ And a Parsing Table is being Constructed.
→ And I/p is taken from a Buffer(here & we add $ Sym here because this '$' Sym is added by Compiler (To recognize the ending of a prog).
→ In Stack at the end, we add $ & the First Sym of our Grammar (First Non-Terminal).



Matched	Stack	I/P	Action
	E $	id+id*id $	
	TE' $	id+id*id $	E → TE'
	FT'E' $	id+id*id $	T → FT'
	idT'E' $	id+id*id $	F → id
id	T'E' $	+id*id $	Match id
id	E' $	+id*id $	T' → ∈
id	+TE' $	+id*id $	E' → +TE'
id+	TE' $	id*id $	M → +
id+	FT'E' $	id*id $	T → FT'
id+	idT'E' $	id*id $	F → id
id+id	T'E' $	*id $	M → id
id+id*	*FT'E' $	*id $	T' → *FT'
id+id*	FT'E' $	id $	M → *
id+id*	idT'E' $	id $	F → id

LL(1) Table Should drawn

→ With the LL(1) Table do derived, initially our Stack is 'E$', & the I/P Consists of 'id+id id $'. Here the I/P Symbol a+b*c$,

→ From Lexical Analysis phase to Syntax Analysis phase, our 'a+b*c$' is Converted to 'id+id*id$' (This I/P Given to the Parsing Table)

Last	id+id*id	T'E'$	$	M → id