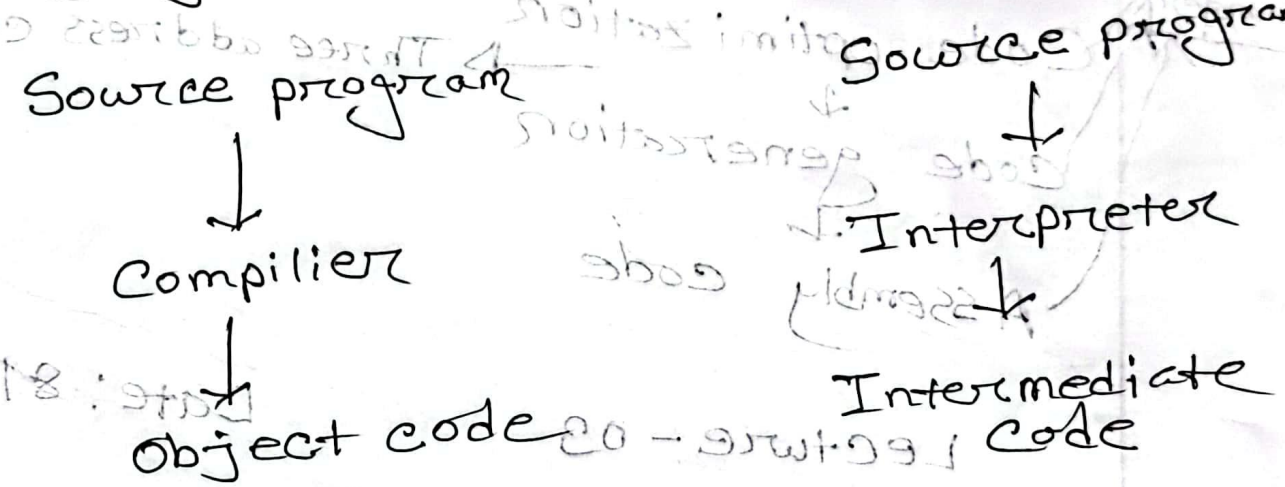
Lecture 01

Date: 3/8/23

Introduction

A translator is a program that takes a program as input written in programming language and produces a program as output in another language.



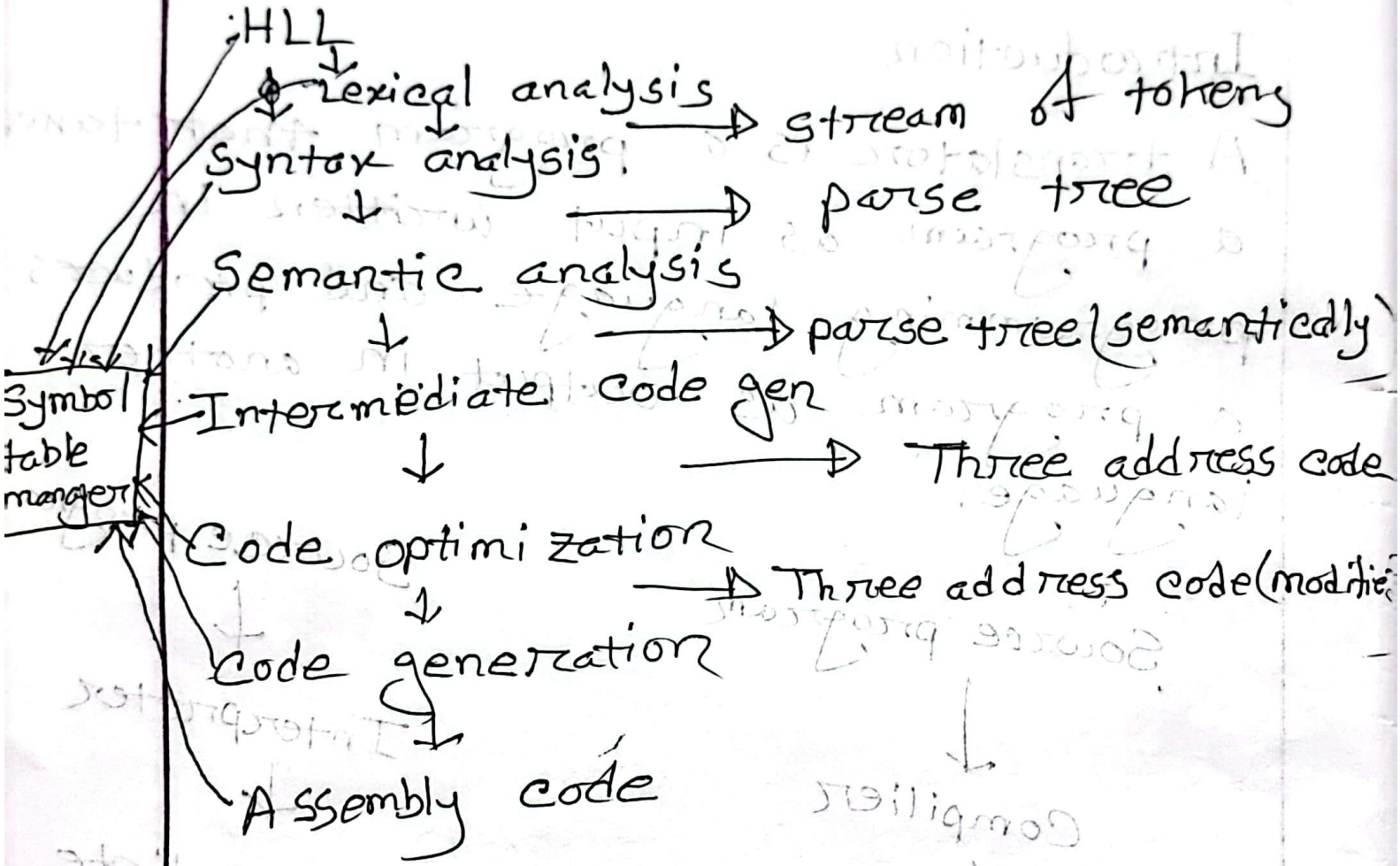
Compilier vs Interpreter

other translators are pre-processor, Assembler

& Execution of a program

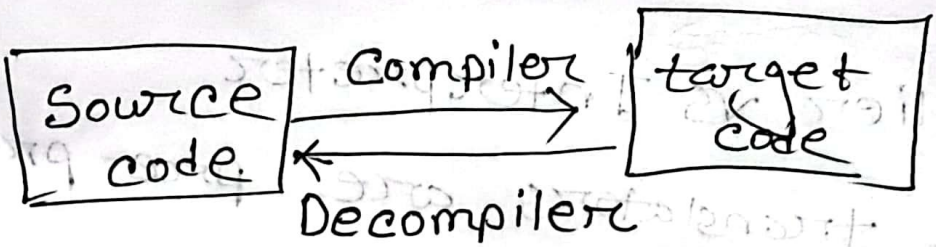
Lecture - 02

Date: 7/8/23



Lecture - 03

Date: 8/8/23



Classification:

i) Native compiler - (same platform & run a

ii) S-to-S / Source to Source) compiler

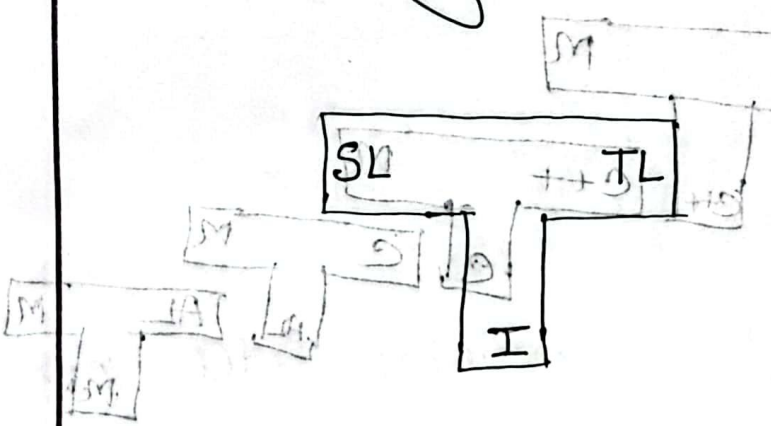
Trans-compiler / Transpiler

iii) Cross-compiler (Different platform এ run করে)

First Compiler

A-0 system: Assembly language কে translate করার compiler.

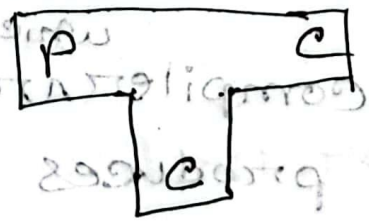
Bootstrapping: Self-Hosting compiler



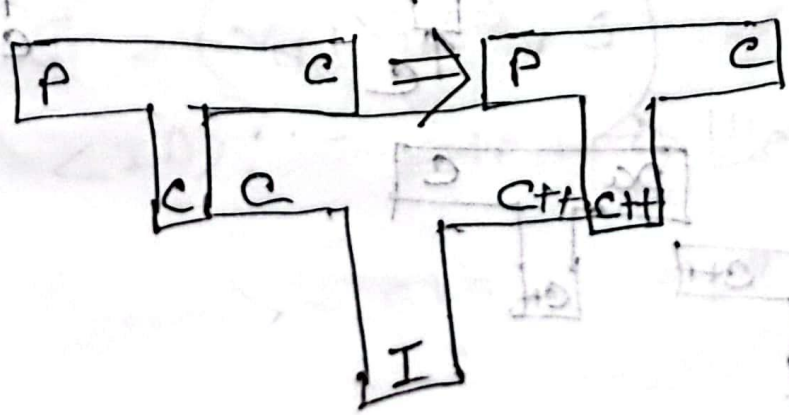
SL → Source language

TL → Target language

I → Implement



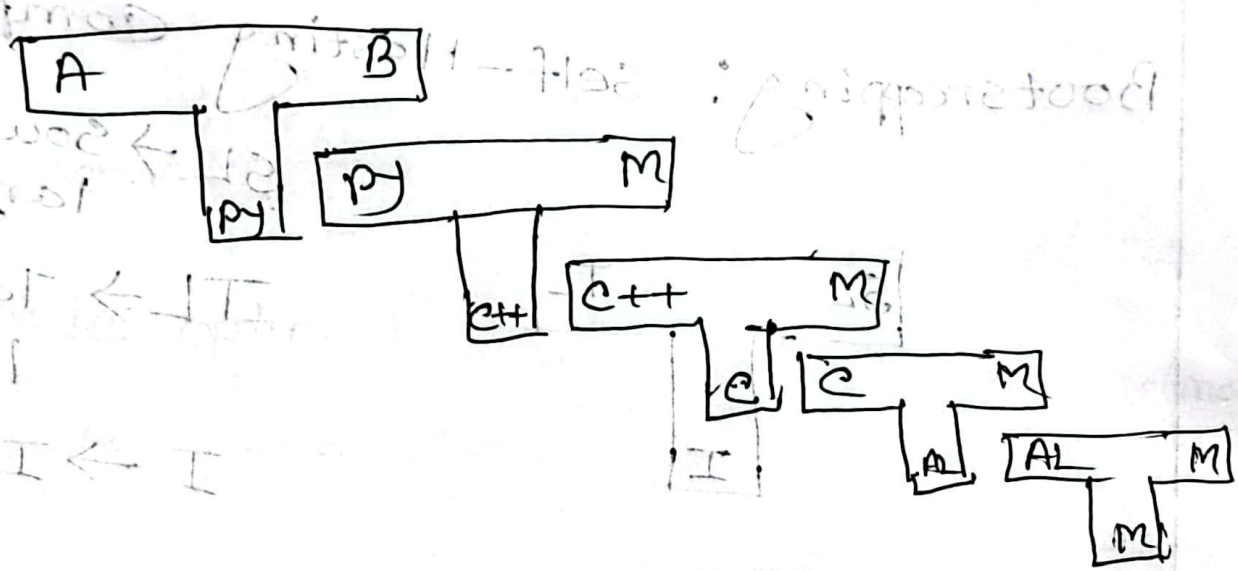
Ex:- P — I/P. Translator written @ in C — O/P C language



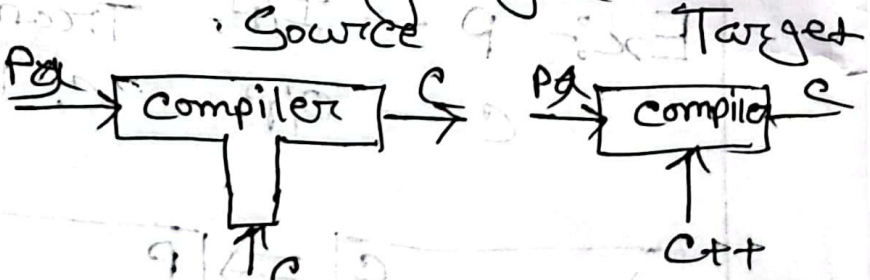
$$P_I^C + C_C^{C++} = P_{C++}^C$$

Tutorial

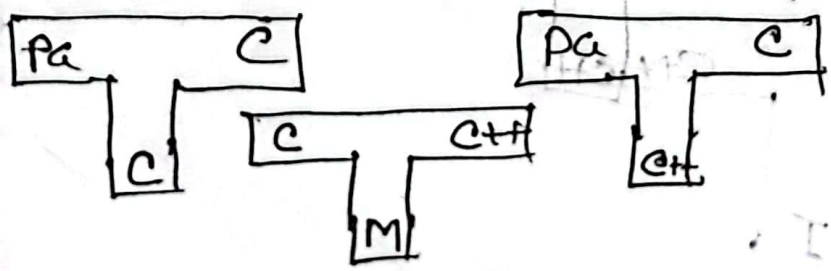
Bootstraping:



Cross Compiler: A compiler runs on one machine but produces object code on another language which



First



Lecture 04 Date: 21/8/23

Lexical Analyzer

Describing by means of how (REs) can be recognised in terms of mathematical models called FSA or FA.

i) RE → ε-NFA (Thompson Construction)
ii) ε-NFA → NFA (ε-closure)
iii) NFA → DFA (subset construction)
iv) DFA minimization (Hopcroft technique)

Lecture 05 Date: 22/8/23

CFG:

BNF: Backus-Nowr-Form (::=, |)

$G = (V_N, V_T, \delta, S)$

<ID> ::= <letter> | <letter> <letter>

<letter> = a|b|...|z|A|B|...|Z

ID → 2 | LL

L → a | B | .. Z | A | B | .. | Z

CFG alternative BNF

CNF → Chomsky Normal Form

R is limited to the following:

i) $A \rightarrow a$

2) $A \rightarrow BC$

3) $S \rightarrow \epsilon$

CYK Algorithm

Conversion of CFG to CNF

Bad rules

1. Start symbol rule

$A \rightarrow uSv : u, v \in \{\Sigma \cup V\}^*$

2. Epsilon Rule

$A \rightarrow \epsilon$

3. unit Rule

$A \rightarrow B$

$B \rightarrow C$

$C \rightarrow A$

4. Mixed rule

5. Long rule

$A \to w, w \in (V \cup \Sigma)^*$

CFG $\to$ CNF

$G = (\{A, B\}, \{a, b\}, R, S)$

$S \to ASA \mid aB$

$BA \to B \mid S$

$B \to b \mid \epsilon$

Lecture 05 Date: 28/8/22

Contex Free Language

A CFG is a way of describing language by recursive rules called production it commits a set of variables (Non-terminals), set of terminals symbols, a start symbol and the production.

$A \rightarrow aA \mid a$

Derivation: Derivation is the approach to defining a CF Language in which the production from head to body of the grammar are used to derive a string consisting of entirely terminals.

?

$E \rightarrow I \mid E+E \mid E*E \mid (E)$

$I \rightarrow Ia \mid Ib \mid a \mid b \mid I0 \mid I1 \mid 0 \mid 1$

"a*(a+boo)*"

$E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow a*(E)$

$\Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(a+E)$

$\Rightarrow a*(a+I) \Rightarrow a*(a+I0) \Rightarrow$

$a*(a+I00) \Rightarrow a*(a+boo)$

left sentential form ↑ ↑ sentence

Parse Tree: pictorial representation of LMD or RMD

$E \rightarrow EAE \mid -E \mid (E) \mid id$ ÷ (id + id)

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

$E \Rightarrow -E$

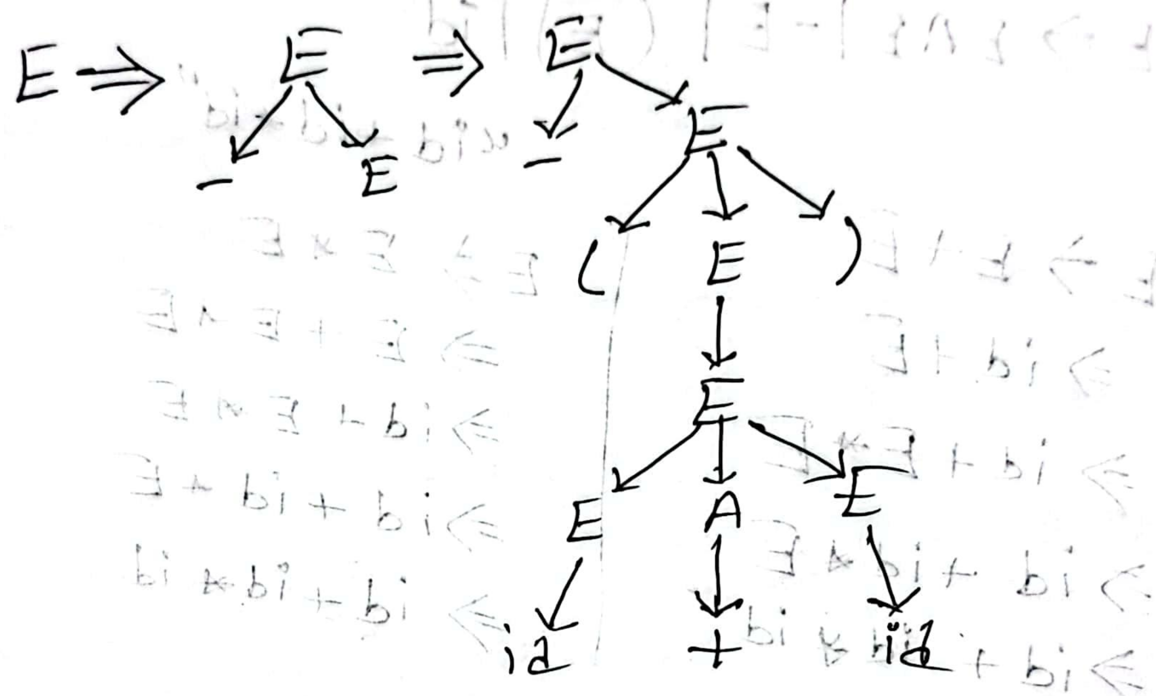
$\Rightarrow -(E)$

$\Rightarrow -(EAE)$

$\Rightarrow -(id\ A\ E)$

$\Rightarrow -(id + E)$

$\Rightarrow -(id+id)$



Lecture 06 Date: 29/8/23

<u>Ambiguity</u>: A grammar that produces more than one parse tree for some sentence is ambiguous

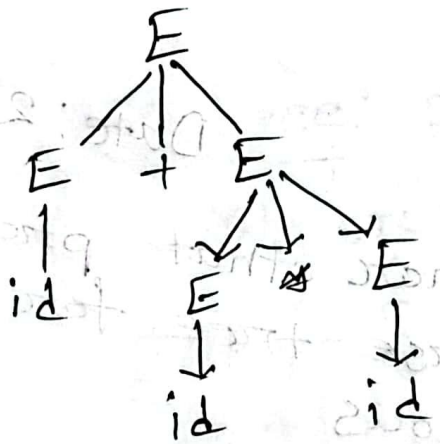
<u>Ambiguous Grammer</u>: For some sentence if a CFG produce more than one RMD or LMD or more than one parse tree, called more than one ambiguous grammer

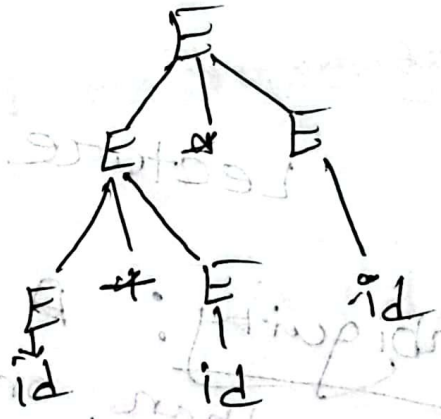
$E \Rightarrow EAE \mid -E \mid (E) \mid id$

"id + id * id"

$E \rightarrow E + E$
$\Rightarrow id + E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

$E \Rightarrow E * E$
$\Rightarrow E + E * E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$





Parsing: In compiler design, parsing is the process of analyzing a text made of a sequence of tokens to determine its syntactic structure w.r.t given grammar.

There are two common forms of parsers –

a) operator precedence

b) Recursive descent

Ways (styles) of Parsing

a) Top-down Parsing → LL parser

b) Bottom-up parsing → LR parser
(shift reduce parsing) → left consume, Right most Derivation

Parsing Algorithms

a) CYK Algorithm

b) Earley's Algorithm

c) LL parsing Algorithm

d) LR parsing Algorithm

Parsers:

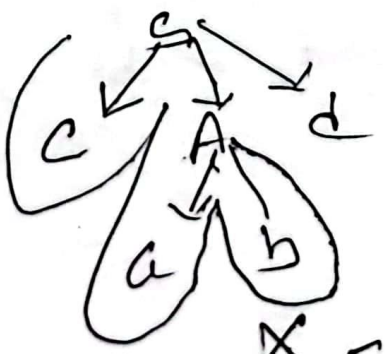
(i) Bottom-up parser

(ii) Top-down parser

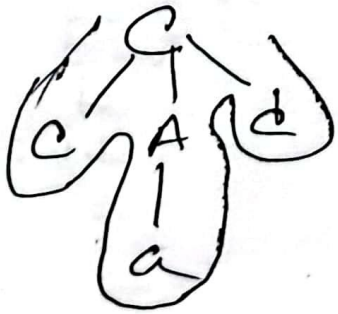
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

cad



X Fail

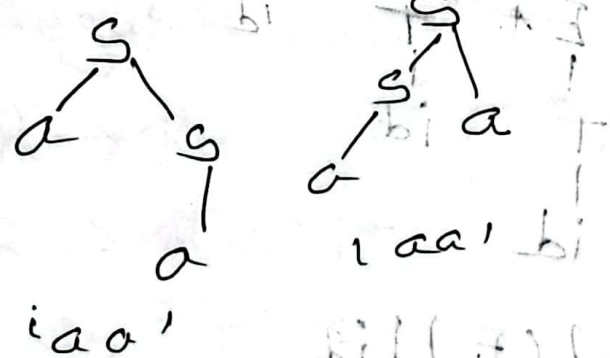


Lecture 07 Date: 4/9/23

1. Operative Associative
2. Operative Precedence

$S \rightarrow aS \mid Sa \mid a$

"aa"



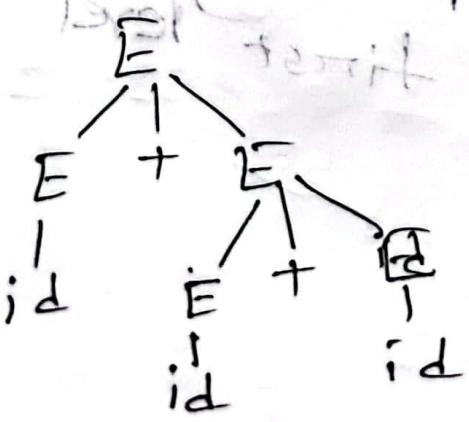
'aa'

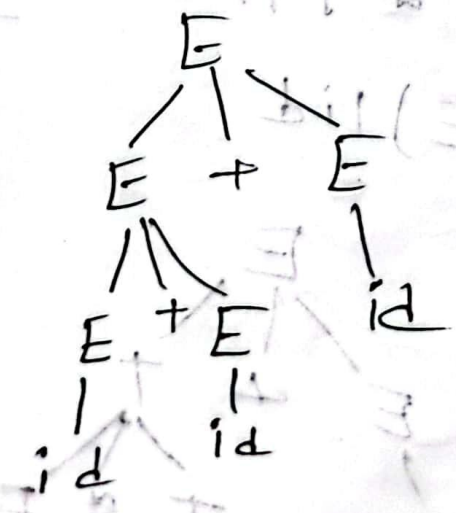
'aa'

Ambiguity for Associative

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

id + id + id





Associativity problem

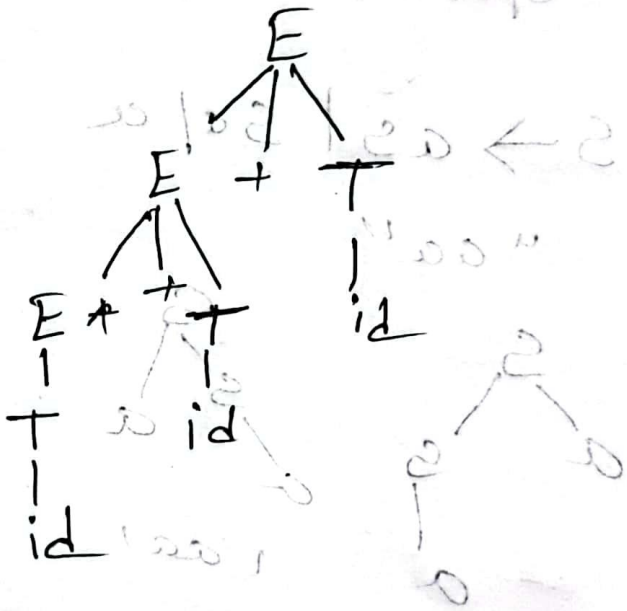
Disambiguation

Associativity Restriction
Left Recursive grammer "id+id+id"

$E \rightarrow E+E \mid id$

$E \rightarrow E+T \mid T$

$T \rightarrow id$



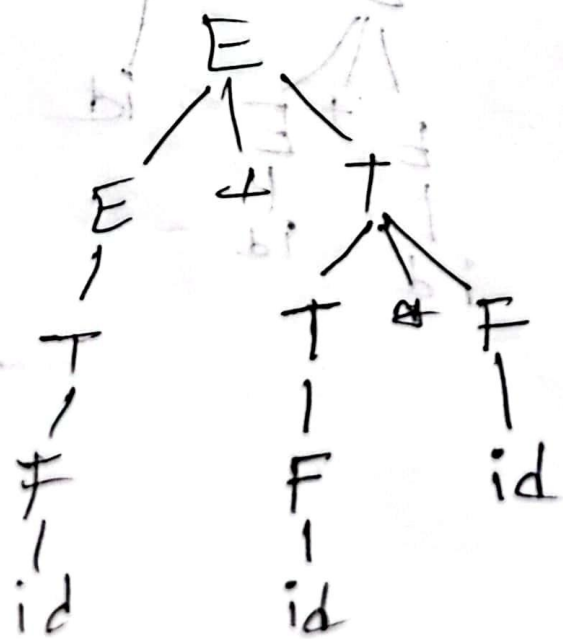
$E \rightarrow E+E \mid E*E \mid (E) \mid id$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

priority high
last level
priority low
first level



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

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow G \uparrow F \mid G$

$G \rightarrow id$

Regular Expression Context Free Grammer

$R \rightarrow R+R \mid RR \mid R^* \mid a \mid b \mid c$

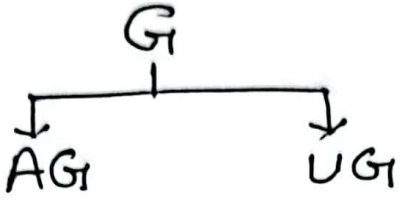
$R \rightarrow R+T \mid T$

$T \rightarrow TF \mid F$

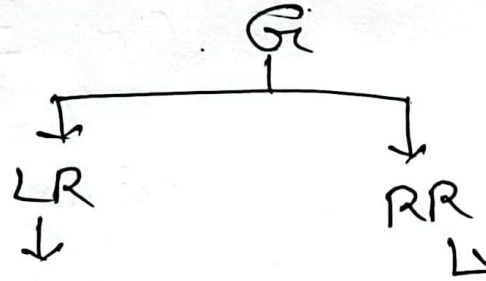
$F \rightarrow F^* \mid G$

$G \rightarrow a \mid b \mid c$

Lecture 08 Date: 6/9/23



$A \to A\alpha \mid \beta$



infinite recursion

↳ কোন সমস্যা না

$A \to A\alpha \mid \beta$

$A \to \beta A'$

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

$S \to S0S1S \mid 00 \mid$

$S \to 0 \mid S'$

$S' \to 0S1SS' \mid \epsilon$

$E \to E+E \mid E*E \mid (E) \mid id$

$E \to E+T \mid T$

$T \to T*F \mid F$

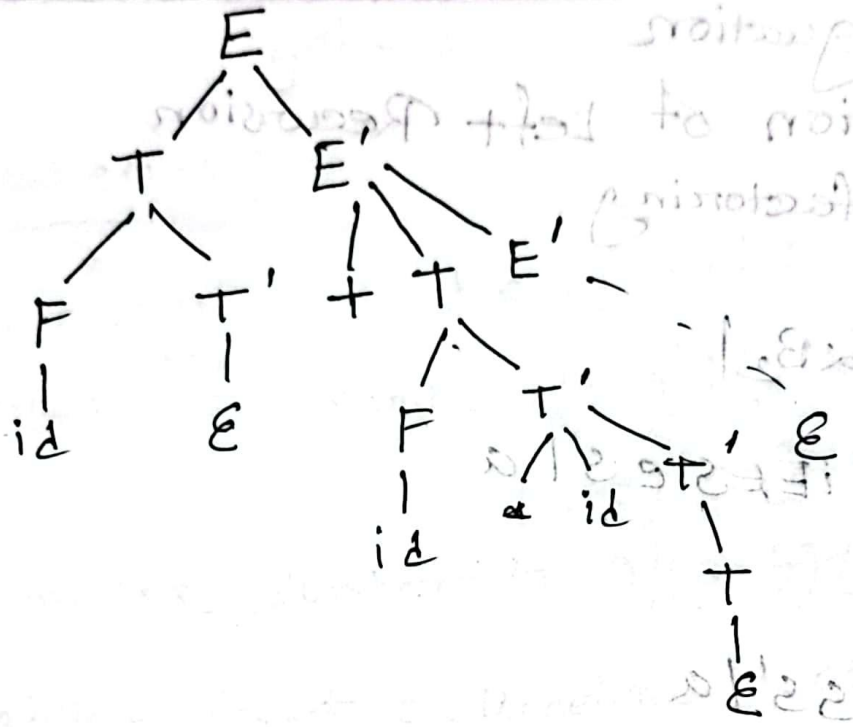
$F \to (E) \mid id$

$E \to TE'$

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

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

$F \to (E) \mid id$



$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid B_1 \mid B_2$

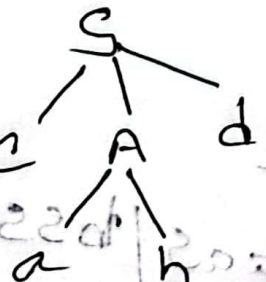
$A \to B_1 A' \mid B_2 A' \mid \ldots$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \varepsilon$

$S \to CAd$

$A \to ab \mid a$

"cad"



যখন মিলবে না তখন backtrack করবে।

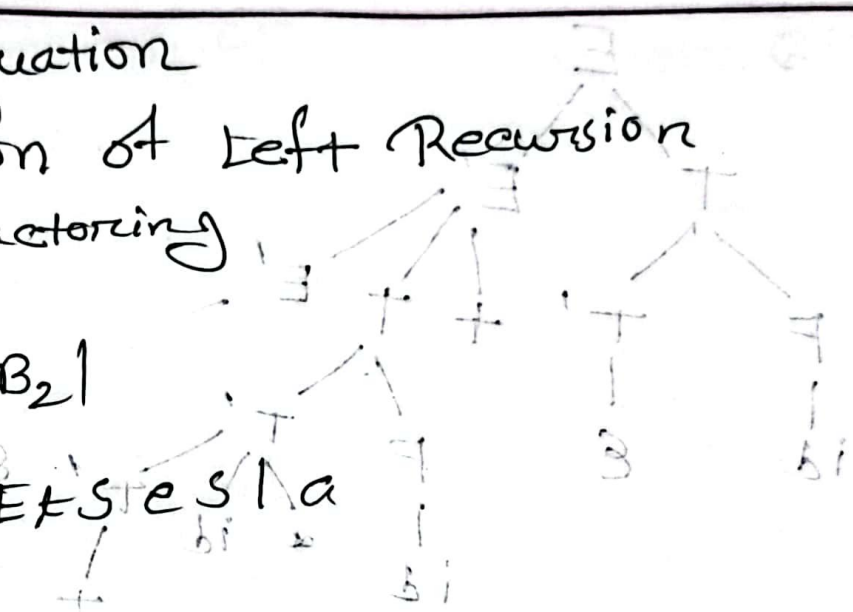
একাধিক option এর কারণে parser deterministic হতে পারে না।

left factoring → elimination of non-determinism

i) Disambiguation

ii) Elimination of Left Recursion

iii) Left factoring



$A \rightarrow \alpha B_1 | \alpha B_2 |$

$S \rightarrow iEtS | iEtSeS | a$

$E \rightarrow b$

$S \rightarrow iEtSS' | a$

$S' \rightarrow \varepsilon | eS$

$E \rightarrow b$

$S \rightarrow cAd$

$A \rightarrow ab | a$

$A \rightarrow aA'$

$A' \rightarrow b | \varepsilon$

$S \rightarrow bSSaaS | bSSaSb | bSb | a$

$S \rightarrow bSS' | a$

$S' \rightarrow SaaS | SaSb | b$

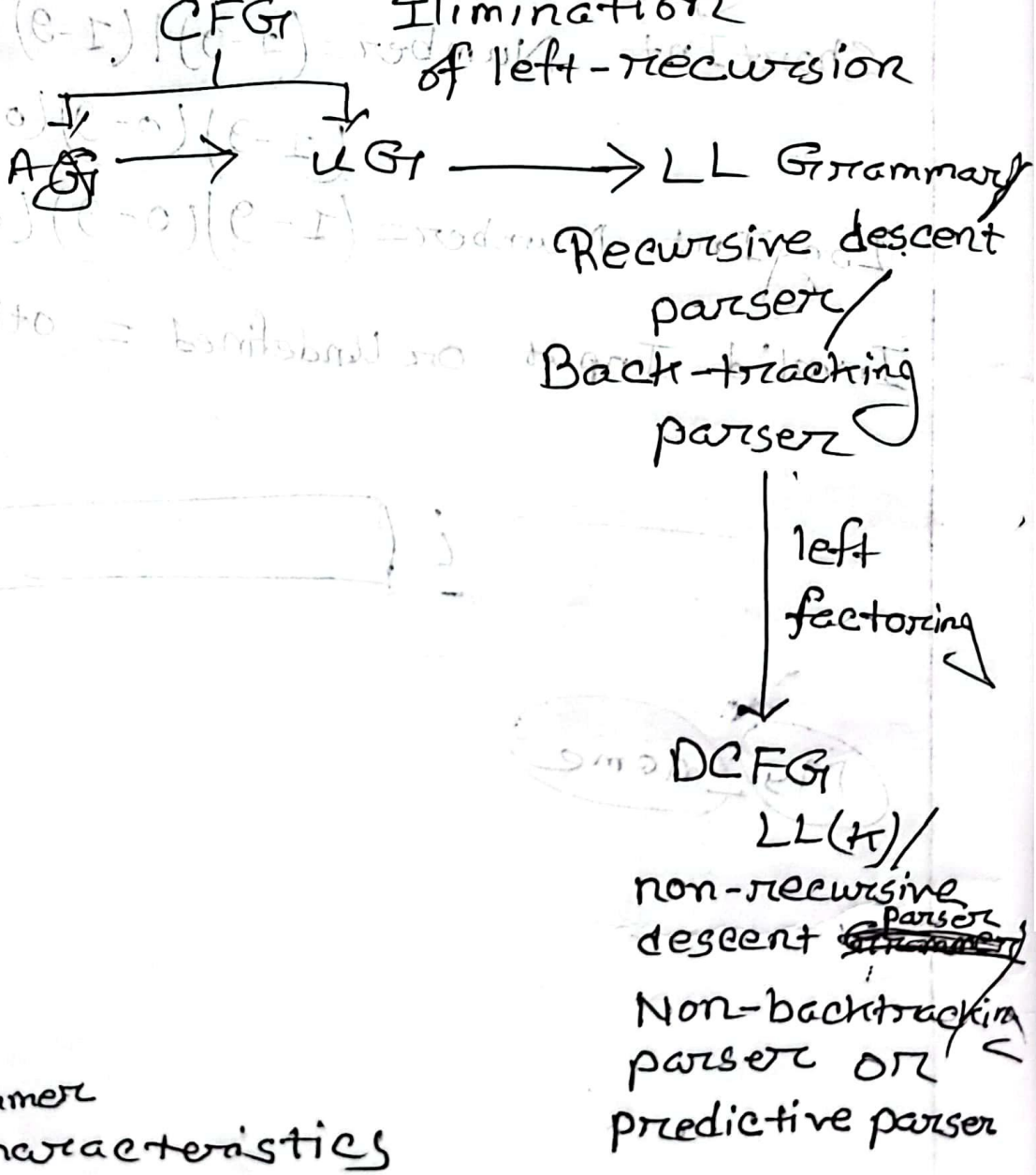
$S \rightarrow bSS' | a$

$S' \rightarrow SaS'' | b$

$S'' \rightarrow aS | Sb$

Lecture 09 Date: 18/9/23

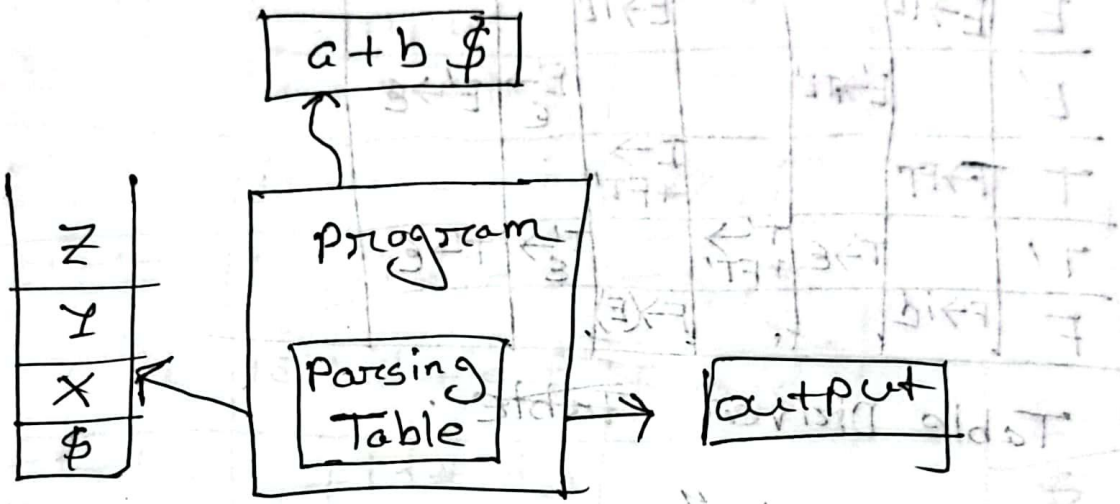
LL(1) Grammer / LL(1) Language / LL(1) parse



LL(1) Grammer Characteristics

- Grammer unambiguous
- Free of Left-recursion
- Deterministic

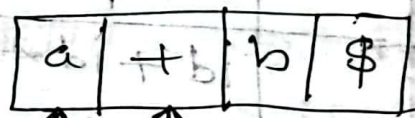
LL(1) Parser / Predictive Parser



X, a

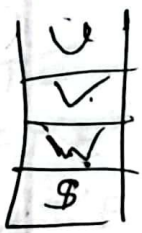
M[X, a]

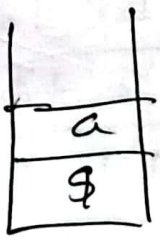
1. if X = a = $ (successfully completed)
2. if X = a ≠ $ (pop)
3. if X is non terminal



M[X, a] = X → UVW (push)

X → non terminal





-এর বাইরে কিছু আসলে error দেখাবে

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

→ LL(1) Grammer

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

Table Driven ~~Table~~:

" id + id * id "

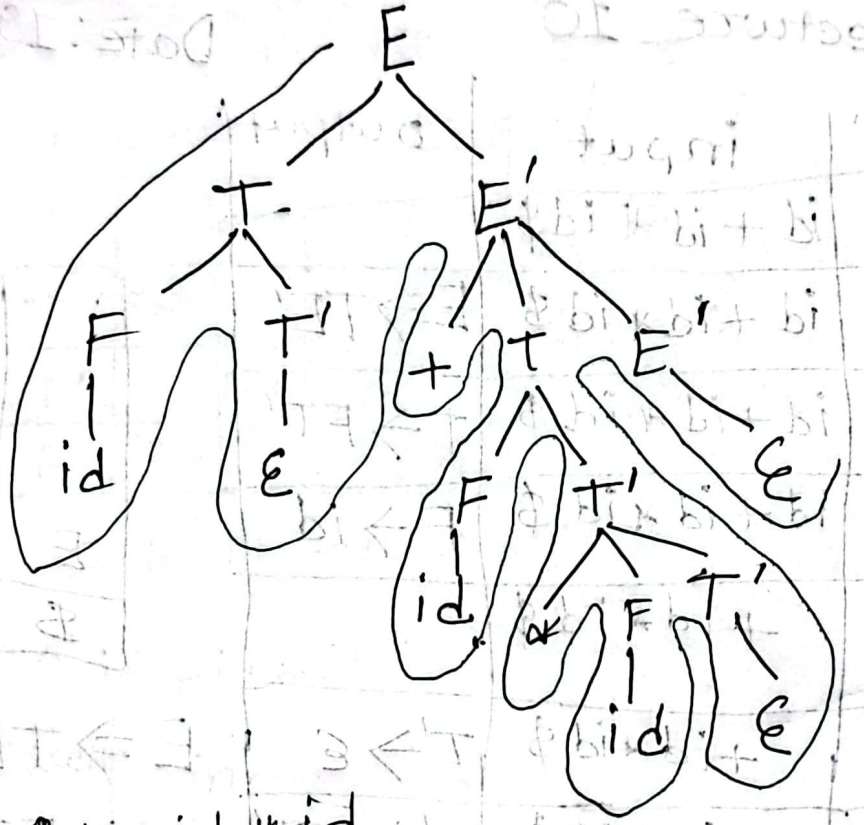
Stack	input	output
$ ~~E~~	id + id * id	~~E → TE'~~
	+ id * id	E → TE'

Lecture 10

Date: 13/3/22.

Stack	input	output
$E	id + id * id $	
$E'T	id + id * id $	E → TE'
$E'T'F	id + id * id $	T → FT'
$E'T'id	id + id * id $	F → id
$E'T'	+ id * id $	
$E'	+ id * id $	T' → ε
$E'T+	+ id * id $	E' → +TE'
$E'T	id * id $	
$E'T'F	id * id $	T → FT'
$E'T'id	id * id $	F → id
$E'T'	* id $	
$E'T'F*	* id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

E ⇒ TE'
⇒ FT'E'
⇒ idT'E'
⇒ idE'
⇒ id+TE'
⇒ id+FT'E'
⇒ id+idT'E'
⇒ id+id*FT'E'
⇒ id+id*idT'E'
⇒ id+id*idE'
⇒ id+id*id



id + id * id

Compiler Lab Date: 19/9/23

Problem-5: Write a program to build a lexical analyzer implementing the following regular expressions. It takes a text as input from a file (e.g. input.txt) and display output in console mode:

Float variable = (a-z|A-Z|0-z)(a-z|A-Z|0-9)*

Float number = ~~(1-9)|(1-9)(0-9)|(1-9)(0-9)(0-9)|~~
~~(1-9)(0-9)(0-9)(0-9)~~

~~D Float~~
~~Long Int~~ Number = (0.(0-9)(0-9)|(1-9)(0-9)*.
(0-9)(0-9)

Double Number = 0.(0-9)(0-9)+|(1-9)(0-9)*.(0-9)(0-9)+

Invalid Input or Undefined = Otherwise

Lecture II

Date: 26/9/23

LL(1)

↳ non-deterministic থাকা যাবে না। Left factoring করতে হবে।

LL(1) parsing table → $ symbol থাকবে

FIRST set = { }

FOLLOW set = { } → $ symbol থাকবে

Rules for First set:

1. First(X) = { t | (t is a terminal and $X \Rightarrow^* t\beta$ or, $t = \varepsilon$ and $X \Rightarrow \varepsilon$) }

2. If X is a terminal then First(X) is just X

2. If there a production $X \to \varepsilon$ then add ε to First(X)

3. If a production $X \to Y_1 Y_2 \cdots Y_k$ then add First $(Y_1 Y_2 \cdots Y_k)$ to First(X)

4. First $(Y_1, Y_2, \cdots, Y_k)$ is either

(a) First (Y_1) (if First (Y_1) does not contain ε

(1) $S \to aABb$
$A \to c \mid \varepsilon$
$B \to d \mid \varepsilon$

$F_o(S) = \{\$\}$
$F_o(A) = \{d, b\}$
$F_o(B) = \{b\}$

First(S) = First(aABb)
$= \{a\}$

First(A) = First(c) ∪ First(ε)
$= \{c\} \cup \{\varepsilon\}$
$= \{c, \varepsilon\}$

First(B) = {d, ε}

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

First(S) =
First(A) = {a, ε}
First(B) = {b, ε}
First(C) = {c}
First(D) = {d, ε}
First(E) = {e, ε}

First(S) = First(ABCDE)
= {a} ∪ {b} ∪ {c}
= {a, b, c}

(3) S → Bb | Cd
B → aB | ε
C → cC | ε

First(B) = {a} ∪ {ε}
= {a, ε}
First(C) = {c, ε}
First(S) = First(Bb) ∪ First(Cd)
= {a, b} ∪ {c, d}
= {a, b, c, d}

Follow(S) = {$}

Fo(B) = {b, $}

Fo(C) = {d}

Parsing Table:

For "I"

	a	b	c	d	$
S	S→aBb				
A		A→ε	A→C	A→ε	
B		B→ε		B→d	

	a	b	c	d	$
S	S→Bb	S→Bb	S→cd	S→cd	
B	B→aB	B→ε			~~B→ε~~
C			C→c	C→ε	

For "J"

	FIRST set	FOLLOW set
$E \to TE'$	$\{+, \epsilon\}$	$\{\$,)\}$
$E' \to +TE' \mid \epsilon$	$\{+, \epsilon\}$	$\{\$,)\}$
$T \to FT'$	$\{id, ($	$\{+, \$,)\}$
$T' \to *FT' \mid \epsilon$	$\{*, \epsilon\}$	$\{+, \$,)\}$
$F \to id \mid (E)$	$\{id, ($	$\{*, +, \$,)\}$

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

Compiler Lab

Date: 26/9/23

Problem 6: Write a program to build a lexical analyzer implementing the following regular expressions. It takes a text as input from a file (eg. input.txt) and display output in console mode.

Character variable = ch_(a-zA-Z0-9)(a-zA-Z0-9)*

Binary variable = bn_(a-zA-Z0-9)(a-zA-Z0-9)*

Binary Number = 0(0|1)(0|1)*

Invalid Input or undefined = otherwise

Lecture 12 Date: 02/10/23

Shift-reduce Parser (Bottom up parser) (LR

It is bottom-up parser because the parser construct parse tree starting from leaf (Bottom) towards the root (top).

$S \to aAcBe$ $w =$ "abbcde"

$A \to Ab \mid b$

$B \to d$

$abbcde \Rightarrow aAbcde$ [A → b]

$\Rightarrow aAcde$ [A → Ab]

$\Rightarrow aAcBe$ [B → d]

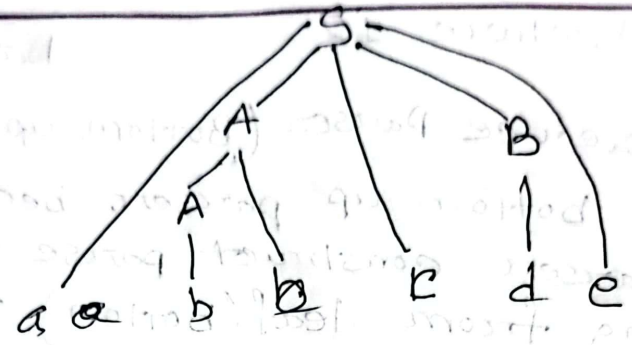
$\Rightarrow S$ [S → aAcBe]

$S \overset{RMD}{\Rightarrow} aAcBe$ [S → aAcBe]

$\overset{RMD}{\Rightarrow} aAcde$ [B → d]

$\overset{RMD}{\Rightarrow} aAbcde$ [A → Ab]

$\overset{RMD}{\Rightarrow}$ "abbcde" [A → b]



Bottom up parsing can be viewed as an attempt to reduce the input ω to the start symbol of a grammer by tracing out right-most derivation (RMD) of string ω in reverse.

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

$\omega =$ "id + id * id"

$E \overset{RMD}{\Rightarrow} E+E \quad [E \rightarrow E+E]$

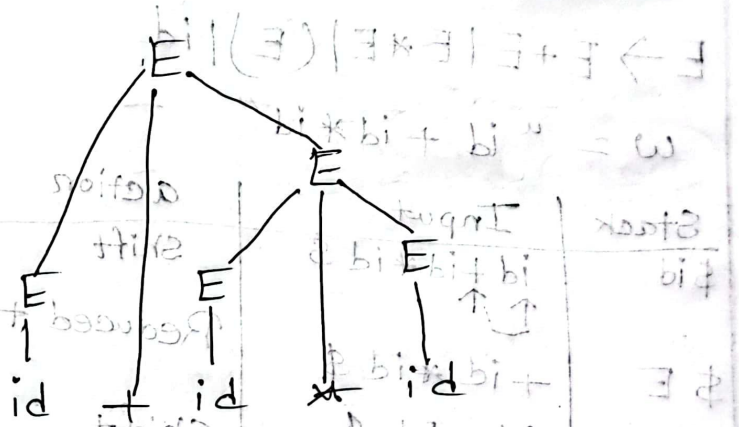
$\Rightarrow E+E*E \quad [E \rightarrow E*E]$

$\Rightarrow E+E*id \quad [E \rightarrow id]$

$\Rightarrow E+id*id \quad [E \rightarrow id]$

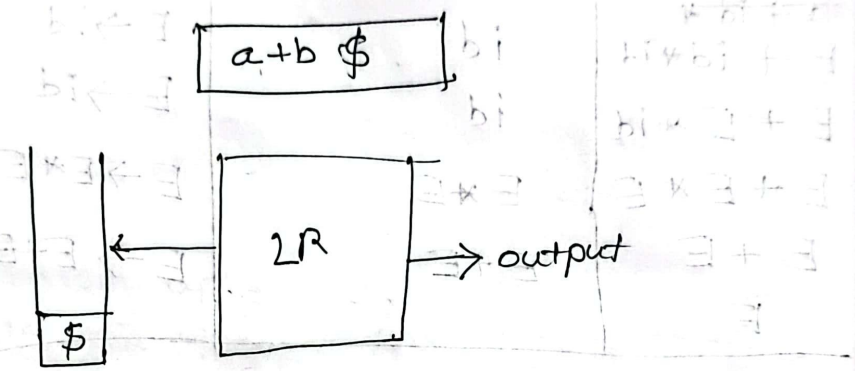
$\Rightarrow id+id*id \quad [E \rightarrow id]$

Right Sentential form	Handle	Production of reduction
id + id * id	id	E → id
~~id + id *~~		
E + id * id	id	E → id
E + E * id	id	E → id
E + E * E	E * E	E → E * E
E + E	E + E	E → E + E
E		



Date: 03/10/12.

LR parsers:



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

$w =$ "id + id * id"

Stack	Input	action
$id	id + id * id $	Shift
$ E	+ id * id $	Reduced to E→id
$ E +	id * id $	Shift
$ E + id	* id $	Shift
$ E + E	* id $	Reduced to E→id
$ E + E *	id $	Shift
$ E + E * id	$	Shift
$ E + E * E	$	Reduced to E→id
$ E + E	$	Reduced to E→E*E
$ E	$	Reduced to E→E+E
		Accept

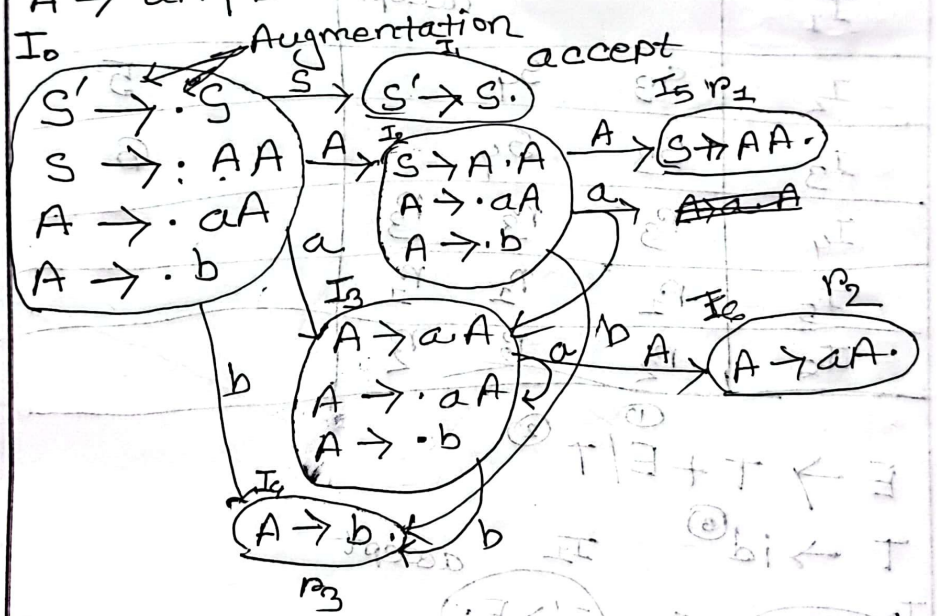
Conflict

① Shift-reduce
② reduce-reduce

0 → Look Ahead
1 → 1 টা input

LR(0) Grammar:

$S \to AA$ ①
$A \to aA$ ② | b ③



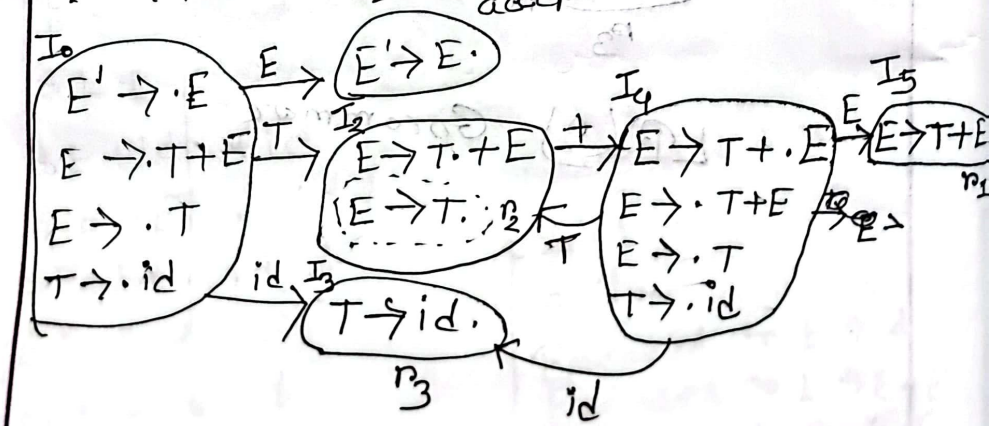
LR(0) Grammar

LR(0) Parse Table

States	Action			Go to	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

$E \rightarrow T+E$ ① / T ②

$T \rightarrow id$ ③



LR(0) parse Table

States	Action			Go to	
	id	+	$	T	E
I_0	S_3			2	1
I_1			accept		
I_2	r_2	S_4/r_2	r_2		
I_3	r_3	r_3	r_3		
I_4	S_3			2	5
I_5	r_1	r_1	r_1		

Date: 9/10/23

- LR(0)
  - SLR(1)
    - CLR(1)
      - LALR(1)

Simple LR (SLR):

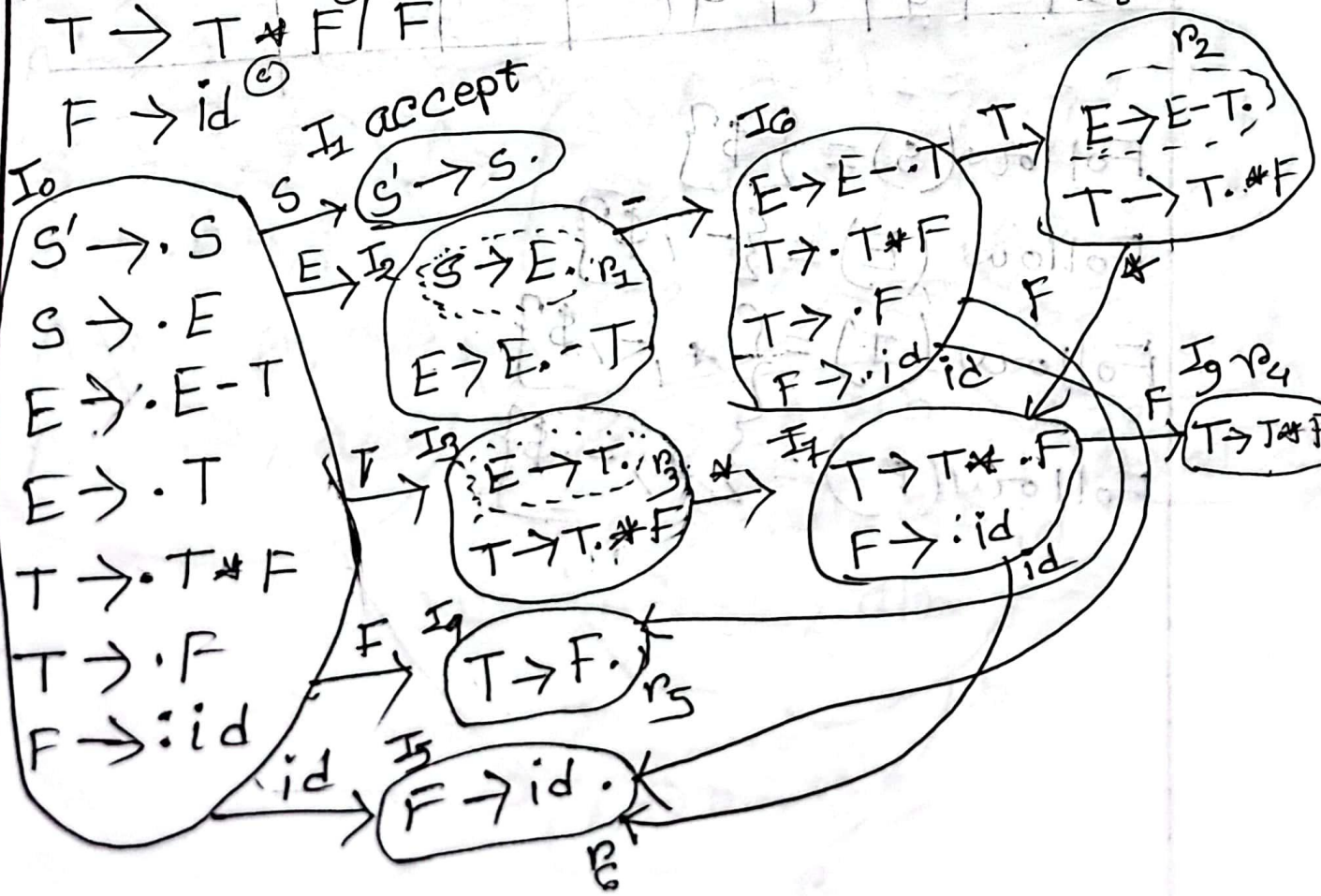
SLR(1) = LR(0) + Follow

$S \to E$ ①

$E \to E - T$ ② / T ③

$T \to T * F$ ④ / F ⑤

$F \to id$ ⑥



States	Action				Go to			
	id	−	*	$	S	E	T	F
I_0	S_5			~~accept~~	1	2	3	4
I_1				accept				
I_2		S_6		r_1				
I_3		r_3	S_7	r_3				
I_4		r_5	r_5	r_5				
I_5		r_6	r_6	r_6				
I_6	S_5						8	4
I_7	S_5							9
I_8		r_2	S_7	r_2				
I_9		r_4	r_4	r_4				

Follow(S) = {$}

Follow(E) = {−, $}

Follow(T) = {*, −, $}

Follow(F) = {*, −, $}

CLR(1) ⇒ Canonical lookahead LR parser.

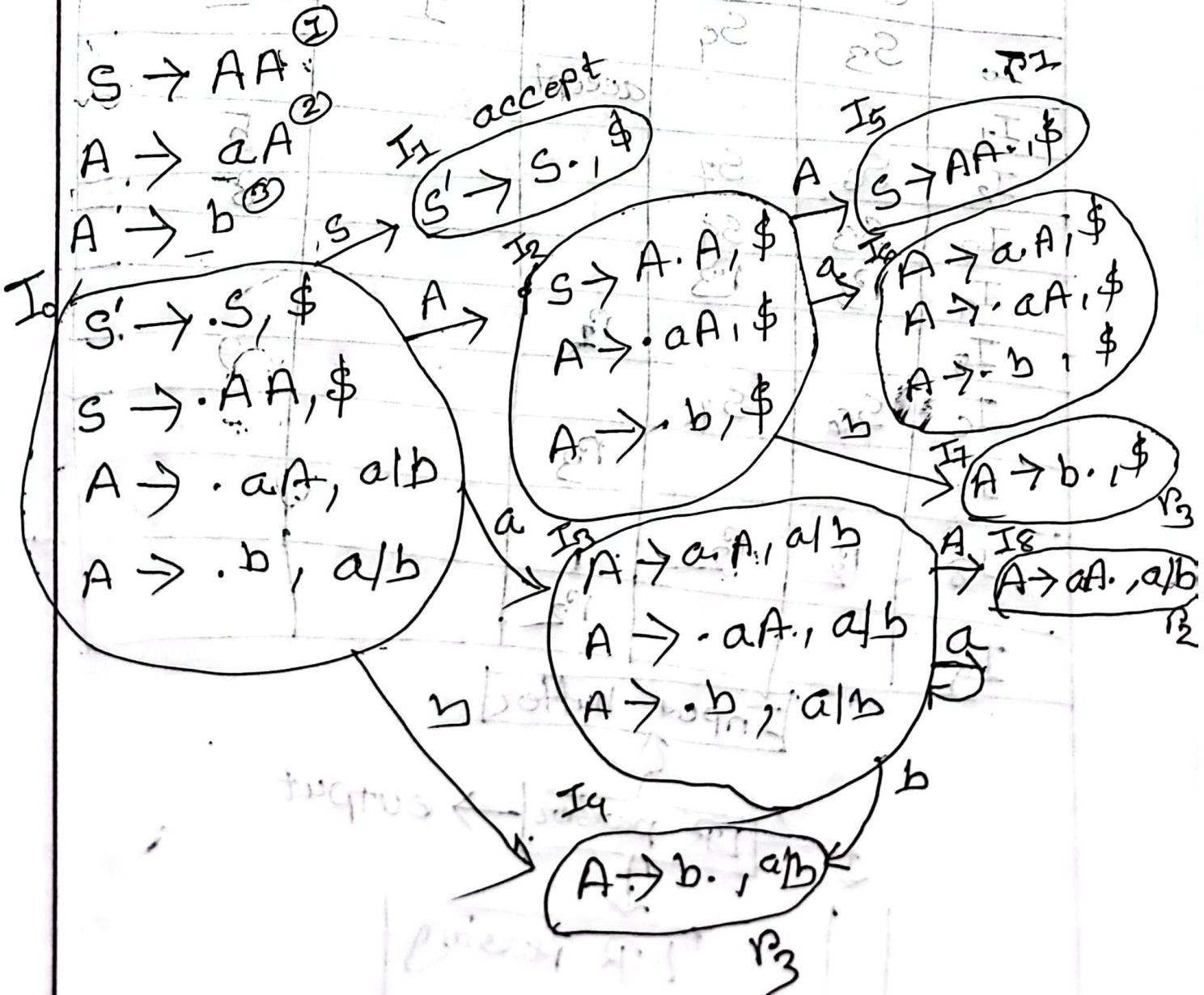
CLR = @ LR(0) + Lookahead

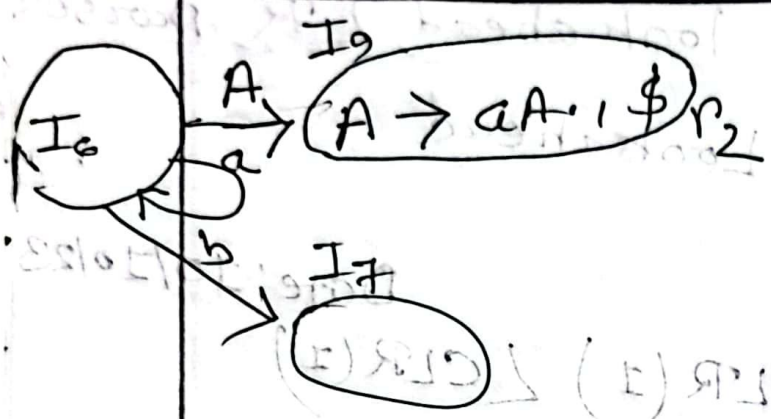
Date: 10/10/23

LR(0) < SLR(1) < LALR(1) < CLR(1)

LR(1) = LR(0) + Lookahead

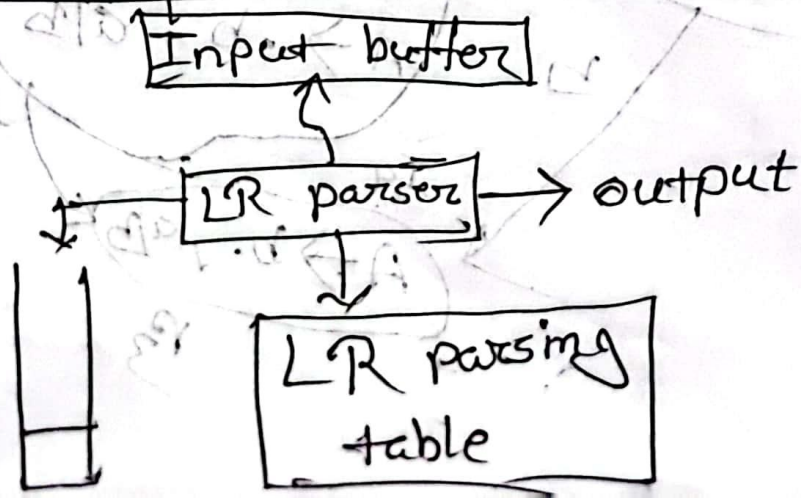
SLR(1) = LR(0) + Follow set





CLR(1) Parsing Table

States	actions			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		



S → AA
A → aA | b

"aabb"

reduced

A → |b|

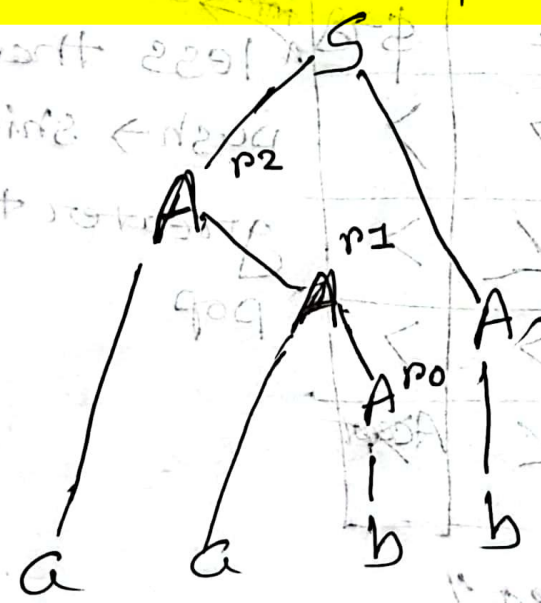
2×2

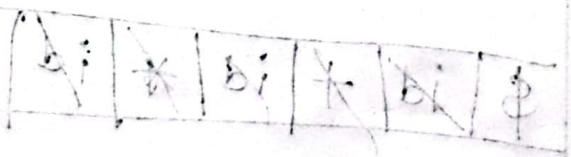
a a b b $

↑ → ↑ → ↑ → ↑ → ↑

0	a	1	a	1	b	4	A	8	A	8	A	2	b	7	A	5	S	1

r4 accept





Date: 16/10/23

Operator Precidence Parser:

E → EAE | id A → + | *	E → E+E | E*E | id

Operator precidence Table

	id	+	*	$ → a
id	Error	>	>	>
+	<	>	<	>
*	<	>	< (crossed out), >	>
$	<	<	<	Accept

↙ stack entity

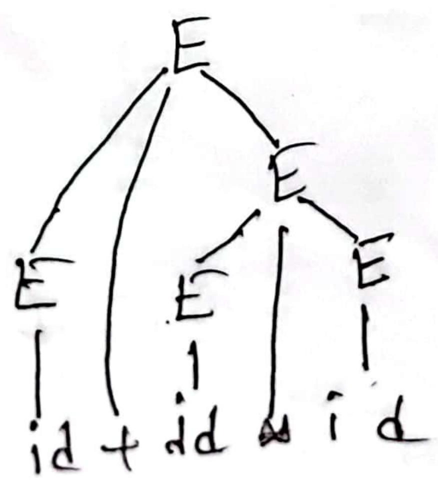
less than হলে push → shift

greater than " pop

$ ← f.

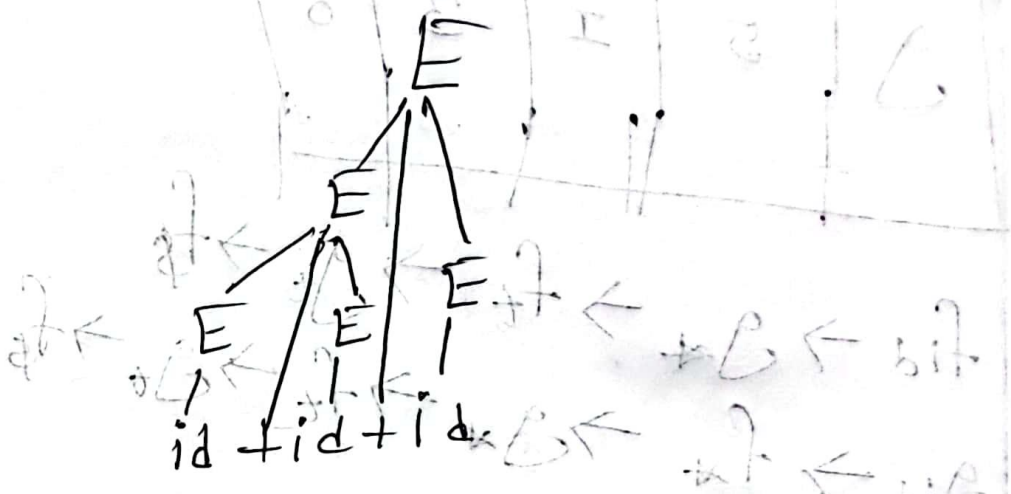
"id + id * id $"

$	~~id~~	~~+~~	~~id~~	~~*~~	~~id~~

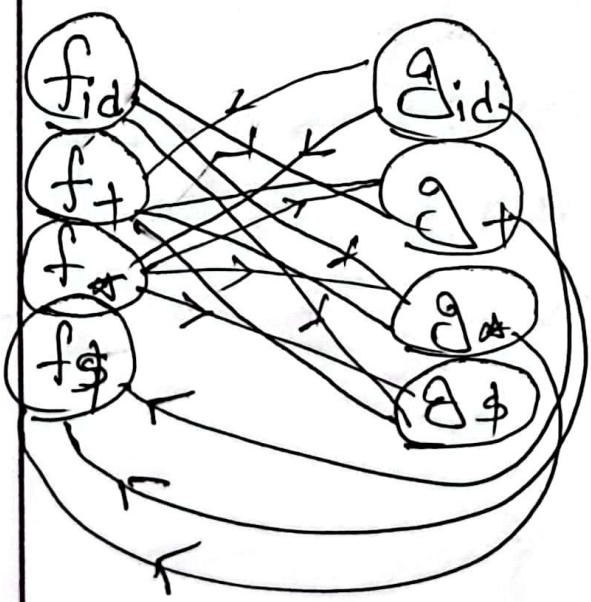


"id+id+id$"

$	id	+	id	+	id		



Entry র সংখ্যা কমানোর জন্য operator Function table কে use করা হয়।



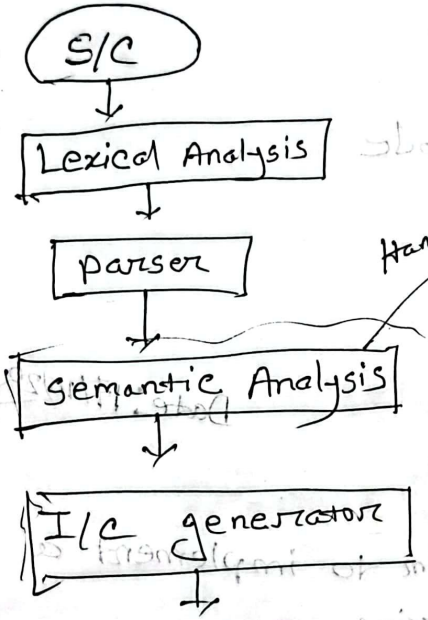
Function Table

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

$f_{id} \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

$g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

Date: 17/10/23



Handler → Syntax directed Translation Schime

SDT

CFG + semantic Rule = SDT

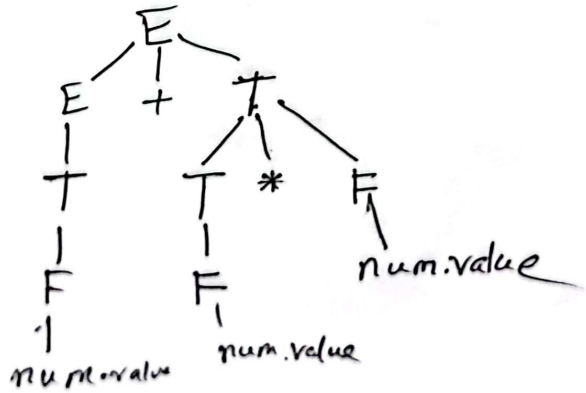
$E \to E+T \quad \{E.val := E.val + T.val\}$

$\quad | T \quad \{E.val := T.val\}$

$T \to T*F \quad \{T.val := T.val * F.val\}$

$\quad | F \quad \{T.val := F.val\}$

$F \to num \quad \{F.val := num.value\}$



Intermediate Code:

a) Postfix notation
b) Syntax tree
c) Tree address code
   i) Quadruple
   ii) Triple

Date: 17/10/22

Compiler Lab

10. Write a program to implement a Shift reducing parsing

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Date: 30/10/23

SDT

a) Postfix notation
b) Syntax Tree
c) Three address code.
   i) Quadruple A := BOPC
   ii) Triple

ডানে fixed use করা হয়।

OP, ARG1, ARG2, Result

a := -b + c * d

$T_1 = -b$

$T_2 = c * d$

$T_3 = T_1 + T_2$

$a = T_3$

	OP	AR1	AR2
0	-	b	
1	*	c	d
2	+	(0)	(1)
3	:=	(2)	

	OP	AR1	AR2	Result
0	-	b		T_1
1	*	c	d	T_2
2	+	T_1	T_2	T_3
3	:=	T_3		a

SDT = CFG + Semantic Action

$E \rightarrow E+T \mid T$ { printf("+"); }

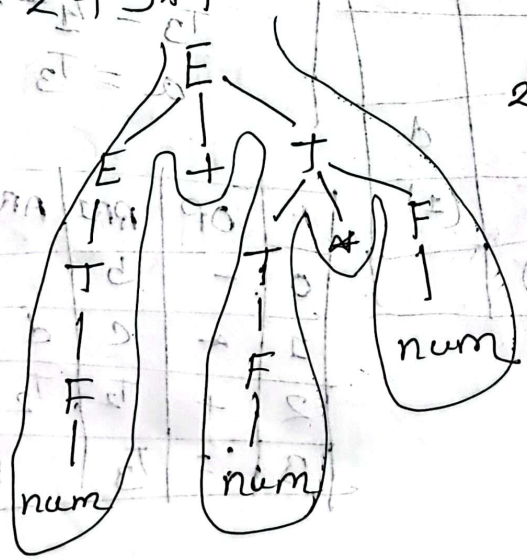
$\mid T$, { }

$T \rightarrow T*F$ { printf("*"); }

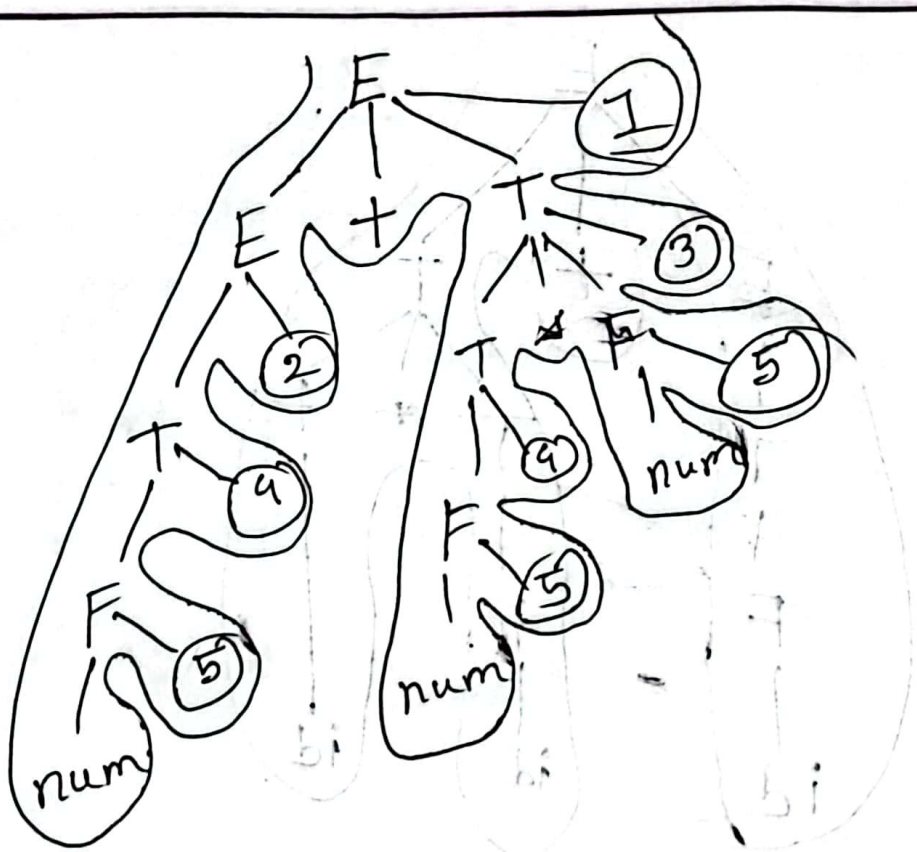
$\mid F$ { }

$F \rightarrow num$ { printf(num.Lval); }

"2+3*4"



			0
	234*+	-	1
		*	2
	(0)	+	3
	(2)	:=	4



234*+

Last class

Date: 7/11/23

SDT for Syntax Tree:

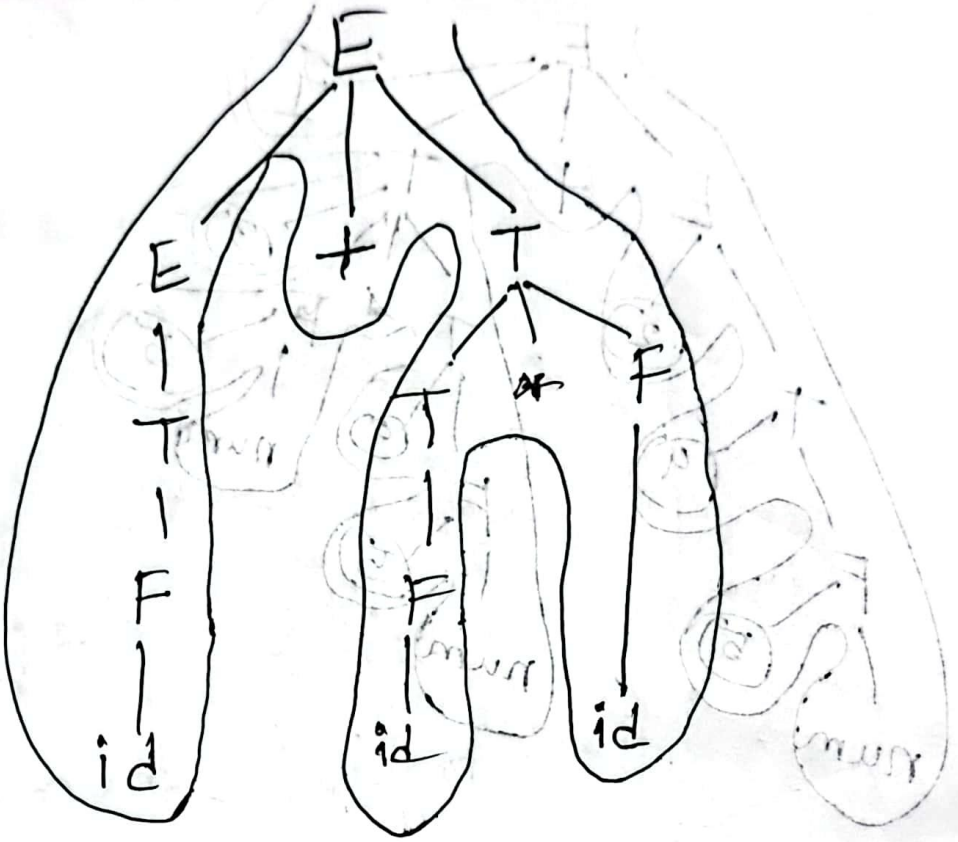
$E \rightarrow E+T$ { E.nptr = mk.nd (E.nptr, ~~value~~ +, T.nptr); }

$\quad | T$ { E.nptr = T.nptr; }

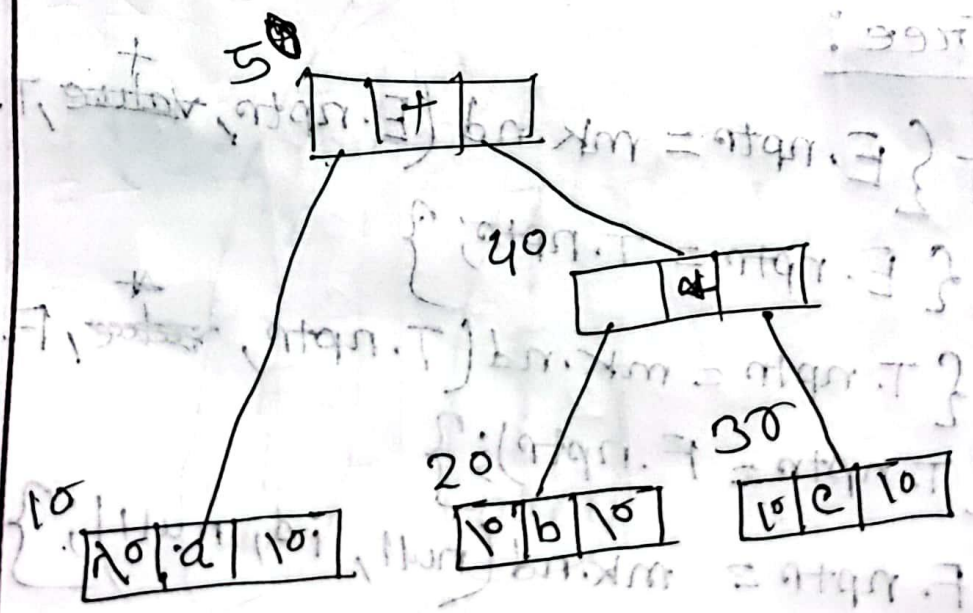
$T \rightarrow T*F$ { T.nptr = mk.nd (T.nptr, ~~value~~ *, F.nptr); }

$\quad | F$ { T.nptr = F.nptr; }

$F \rightarrow id$ { F.nptr = mk.nd (null, id, null); }



"a+b*c" = "(a+(b*c))"



SDT for Type Checking

$E \rightarrow E + T$ {if((E.type == T.type) && ((T.type == int) || (T.type == float))
then
E.type = T.type.
else
Error;
}

| T { E.type = T.type; }

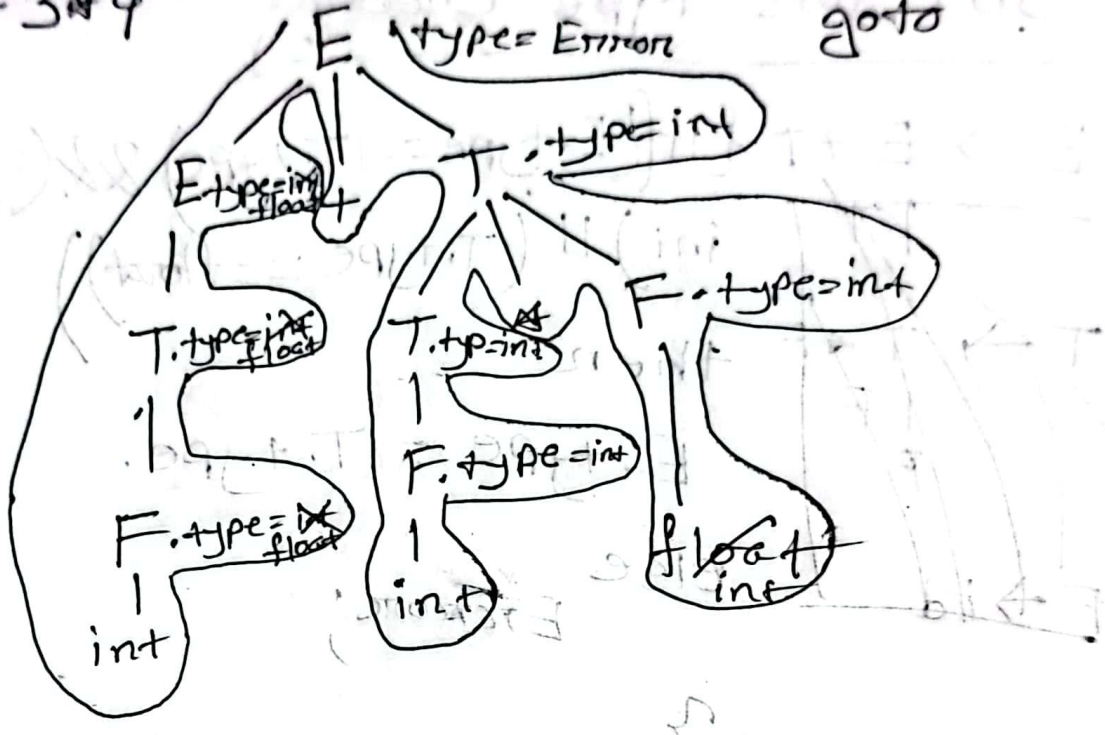
$T \rightarrow T * F$ {if ((T.type == F.type) && ((F.type == int) || (F.type == Float))
then
T.type = F.type;

else
Error;
}

| F { T.type = F.type; }

$F \rightarrow id \mid num$ {F.type = int/float/ char; }

"2+3*4" 2nd reduction error then goto.



$E \rightarrow E * T \quad \{ E.val = E.val * T.val \}$

$\quad | T \quad \{ E.val = T.val; \}$

$T \rightarrow F - T \quad \{ T.val = F.val - T.val \}$

$\quad | F \quad \{ T.val = F.val \}$

$F \rightarrow num \quad \{ F.val = num.lval \}$

"5-3-5*3"

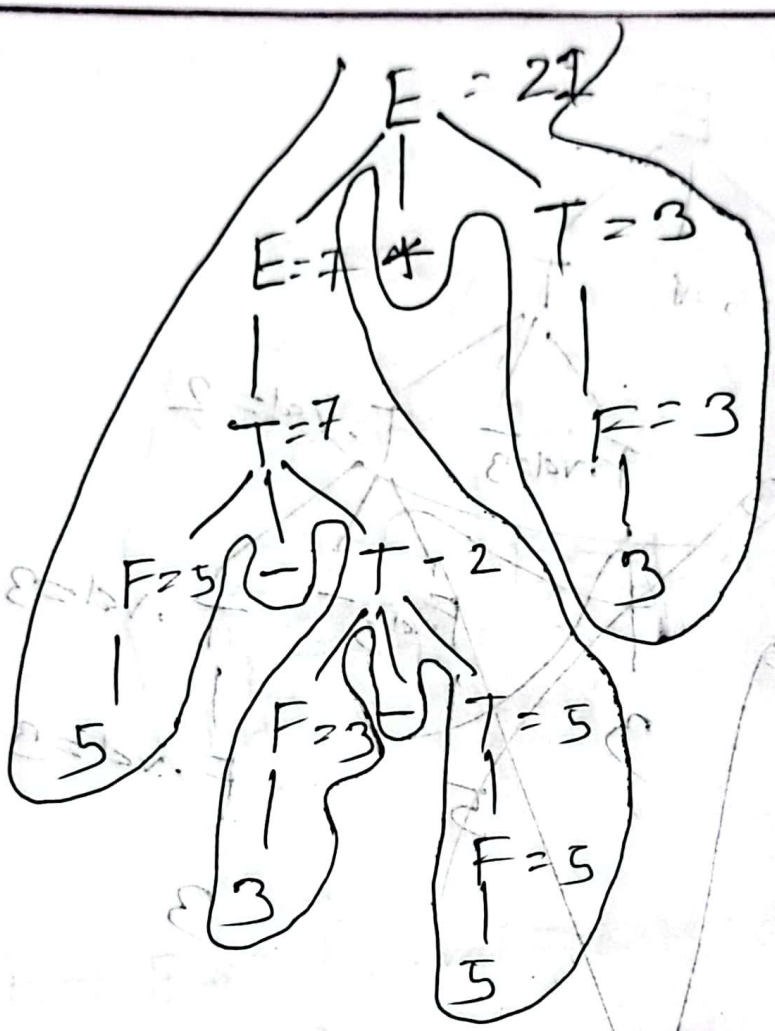
= ((5-(3-5))*3)

= (5+2)*3

= 21

=

2



* Optimization, symbol Table নিজে পড়ে নিতে হবে।

* পরিপাকতন্ত্র পড়ে আসবা পরীক্ষায় আসবে 200%

শ্বসন প্রক্রিয়া + Respiratory system sir গুলা বলে details এ পড়াছে কম

হাসি পাচ্ছে 😂

শুধু Biology Cover করা হয়েছে। এইখান থেকে এক সেট question