By- Pathik455

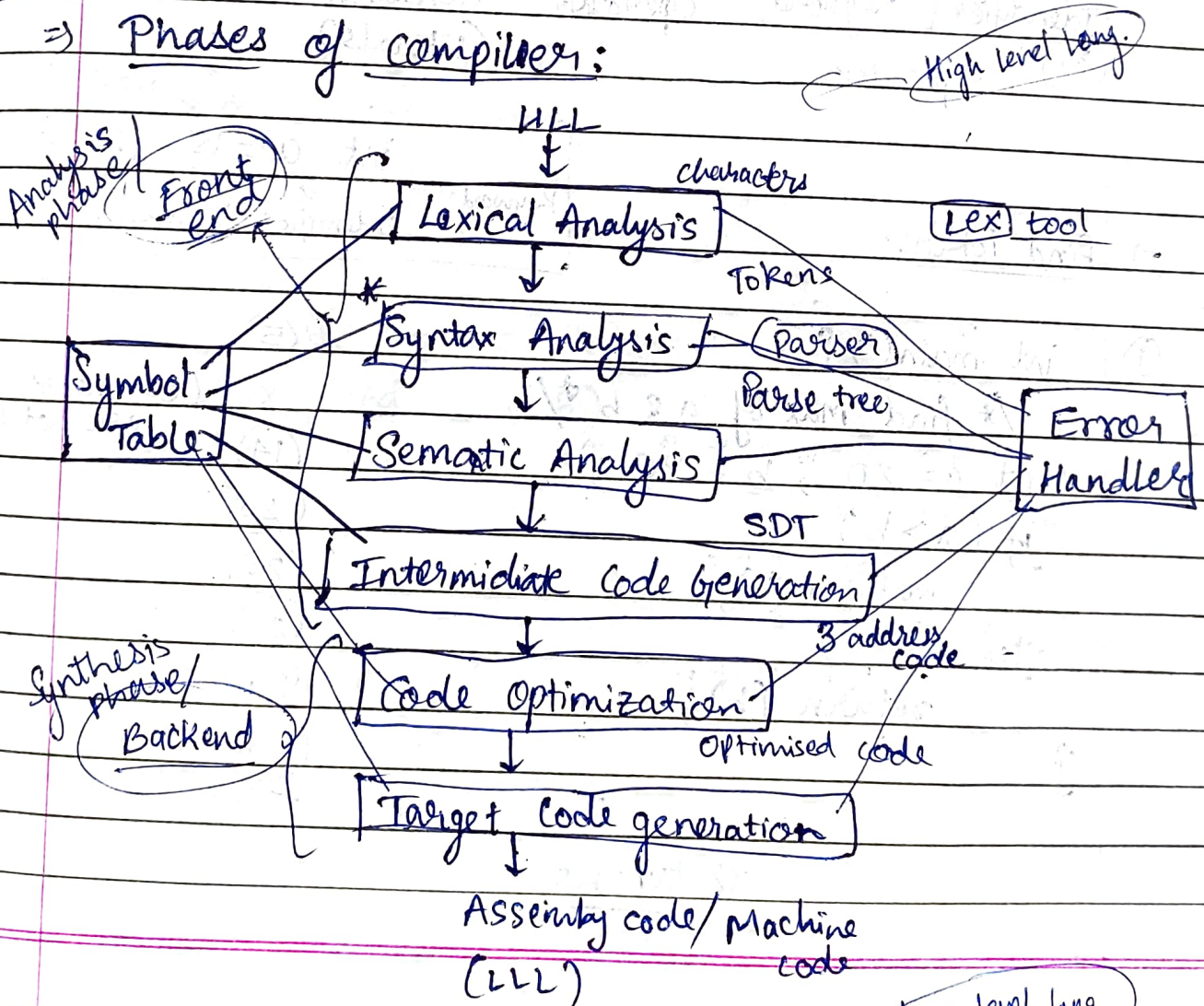
CD

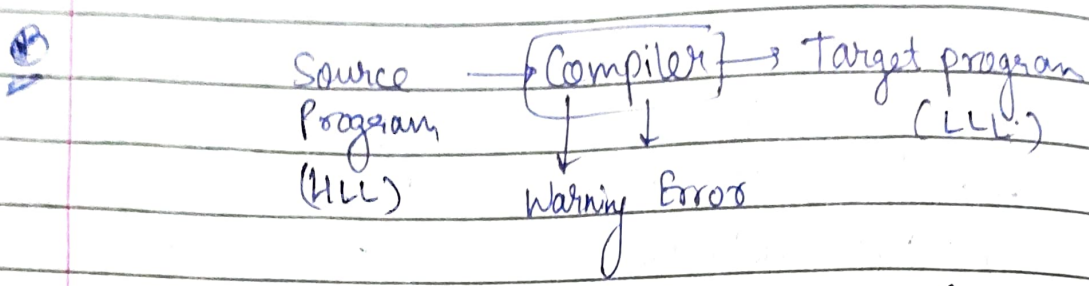
Compiler Design:

1) Lexical Analysis
2) Parser (Syntax Analyzer) ***
3) Semantic Analysis (easy)
4) Intermidiate code generator **
5) Code optimization (easy)

L-2

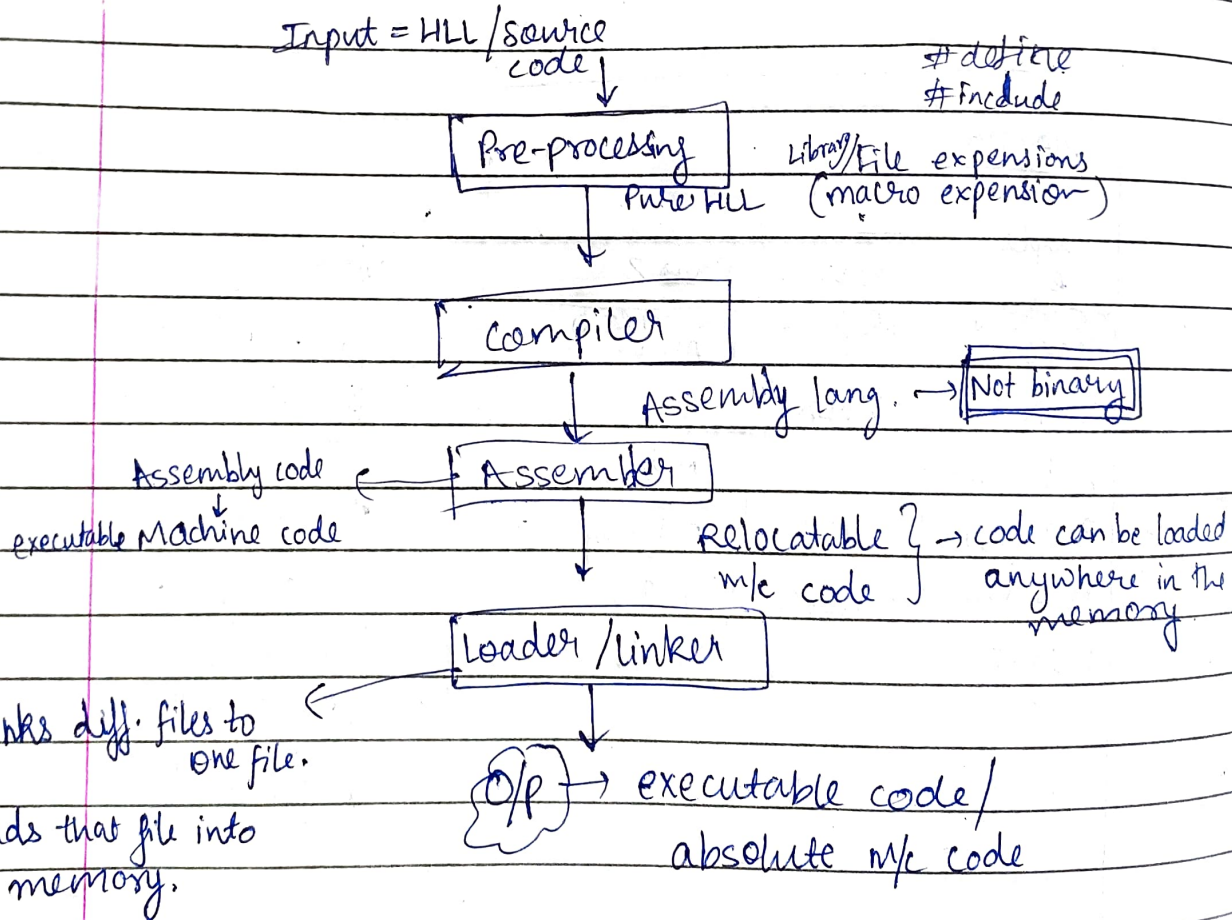
⇒ Phases of compiler:





→ Types of compiler: → Single pass comp. (process takes 1 i...
→ MultiPass comp. " multiple ...

Language Processing System:



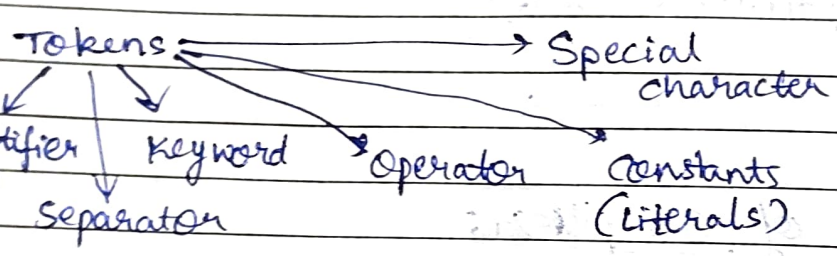
L-3

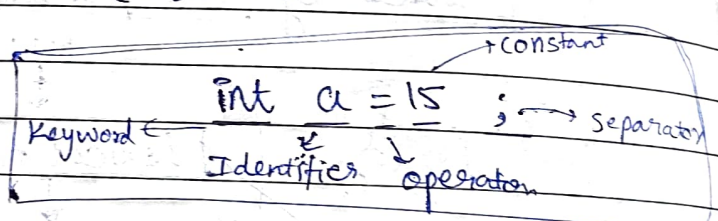
Lexical Analysis - {Lexer, Tokenizer, Scanner}

1) Tokenization

2) Give Error messages → Exceeding length
→ Unmatched String
→ Illegal character

3) Eliminate comments, white spaces (Tab, blank space, new line)





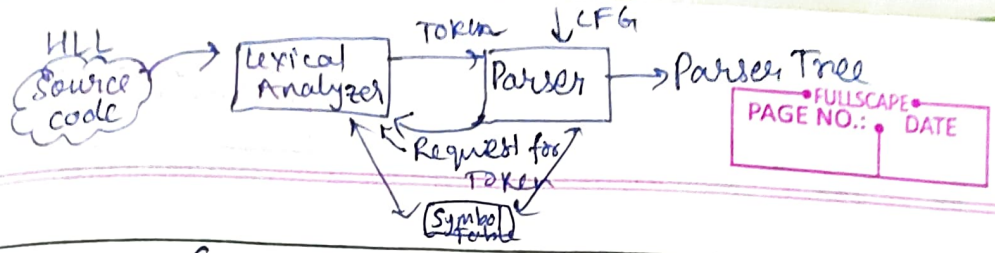
⇒ Find Tokens.

(1)

```
int main() {
  /* find max of a & b */
int a = 20 ; b = 30 ;
if (a > b)
    return (a);
else
    return (b);
}
```

(5)
x (comments are not token)
(14)
(20)
(25)
(26)
(31)
(32)

32 tokens.



FULLSCAPE
PAGE NO.: DATE

(2) printf (" i=%d, &i=%x " , i , &i) ; (10)

1 2 3 4 5 6 7 8 9 10

10 tokens

⇒ Finate Automata (DFA, NFA) is used for tokenisation i.e. (Application of FA) Machine.

(3)
```
int main()
{
    x = y + z ;
    int x, y, z ;
    printf ("sum=%d%d", x);
}
```
4
17
12
5
26 token

(4)
```
main() {
    a = b+++---++=;
    printf ("%d%d", a, b);
}
```
4
15
25 tokens

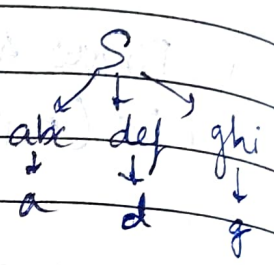
(5)
```
main() {
    int a=10;
    Char b="abc";
    in t c = 30;
    Ch ar d = "xyz";
    in /*comment*/ t m = 40.5;
}
```
4
9
14
20
26
32
33 tokens

L-5

⇒ Finding First() and Follow()

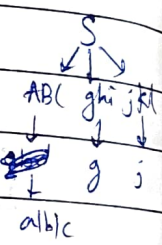
First(A) contains all terminals present in first place of every string derived by A.

(1) S → abc | def | ghi
(2) First(terminal) → terminal
(3) First(ε) → ε



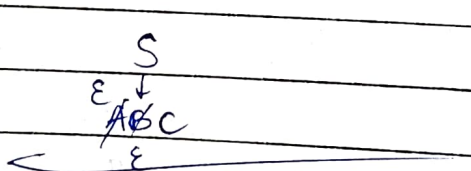
Q.) S → ABC | ghi | jkl
A → a | b | c
B → b
D → d

First(D) = d
First(A) = (a, b, c)
First(S) = (a, b, c, g, j)



Q.) S → ABC
B → a | d | ε
A → a | b | ε
C → e | f | ε

First(C) = (e, f, ε)
First(S) = (a, b, c, d, e, f, ε)



Q.)

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

1.6 Follow()

Follow(A) contains set of all terminals present immediate in right of 'A'.

Rules:

1) Follow of start symbol is $
   Follow(A) = {$}

*Follow never contains {ε}.

2) S → ACD
   C → a | b.
   Follow(A) = First(C) = {a, b}
   Follow(D) = Follow(S) = {$}

3) S → aSbS | bSaS | ε
   Follow(S) = {$, b, a}

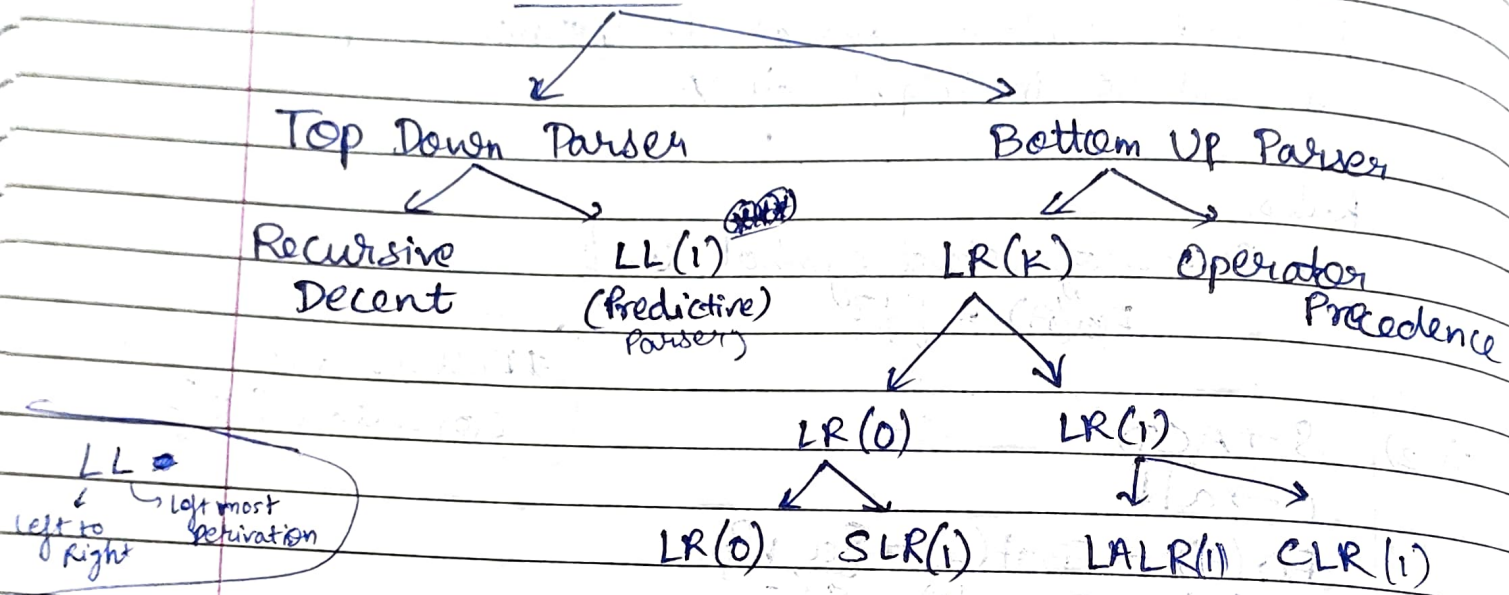
4) S → AaAb | BbBa
   A → ε
   B → ε
   Follow(A) = {a, b}
   Follow(B) = {b, a}

5) S → ABC
   A → DEF
   B → ε
   C → ε
   D → ε
   E → ε
   F → ε
   Follow(A) = First(B)
   ε ⇒ First(C)
   ε ⇒ Follow(S) = {$}

Parsing: It is a process of deriving string from a given Grammar

L-7

Parsers:



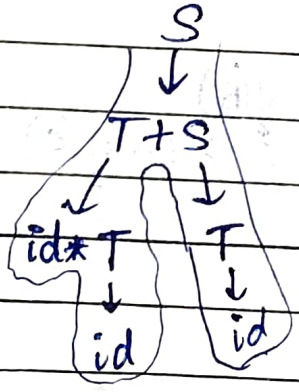
* CFG is used

Q) $S \rightarrow T+S \mid T$

$T \rightarrow id*T \mid id \mid (S)$

$w = id*id+id$ ← Make string.

ans:



id*id+id ✓

L-8 Check LL(1) Grammar: (Parsing Table)

$S \to (L) \mid a$ (1, 2)

$L \to SL'$ (3)

$L' \to \varepsilon \mid ,SL'$ (4, 5)

$S \to aSbS \mid bSaS \mid \varepsilon$

Ans:

First()

$F(S) = \{(, a\}$

$F(L) = \{(, a\}$

$F(L') = \{\varepsilon, ,\}$

Follow()

$Fo(S) = \{\$,), ,\}$

$Fo(L) = \{)\}$

$Fo(L') = \{)\}$

Parse Table:

	(	)	a	,	$
S	1 $S \to (L)$		2 $S \to a$		
L	3 $L \to SL'$		3 $L \to SL'$		
L'		4 $L' \to \varepsilon$		5 $L' \to SL'$	

$S \to (L)$ — first

$S \to a$ — first

$L \to SL'$

$L' \to \varepsilon$ — first = ∴ L' follow

$L' \to ,SL'$ — ↳ ,

Q) $S \to aSbS \mid bSaS \mid \varepsilon$ (1, 2, 3)

	a	b	$
S	1 $S \to aSbS$	2 $S \to bSaS$	3
	3 $S \to \varepsilon$	3 $S \to \varepsilon$	$S \to \varepsilon$

$Fi(S) = \{a, b, \varepsilon\}$

$Fo(S) = \{\$, b, a\}$

$S \to \varepsilon$

↳ follow(S) = {b, a, $}

∴ This is not LL(1) Grammar.

~~Shortcut to find ++(-) or not~~

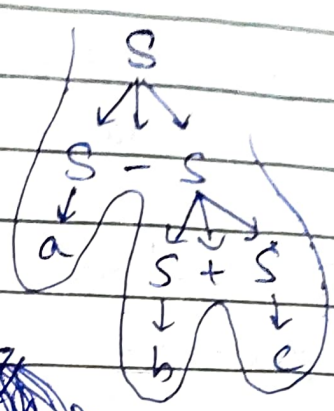
Compiler	Interpreter
1) Takes entire program at once as input	1) It takes single line of code at a time
2) High speed	2) Speed - low
3) Memory required ↑	3) Memory required ↓
4) Errors - displayed together	4) Errors - If one line has error it will not execute further program.
5) Error detection hard	5) Error detection easy
6) compiler are larger in size	6) Smaller in size
7) eg: C, C++ uses compl.	7) Python, perl, Ruby uses interpreter

① Left most Derivative:

eg: $S \to S+S \mid S-S \mid a \mid b \mid c$

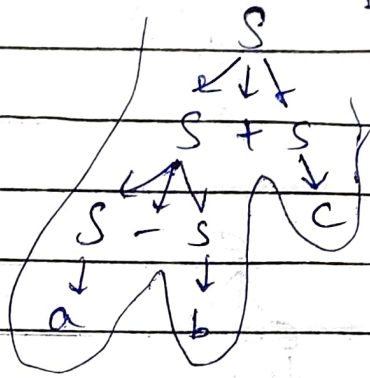
input: $a-b+c$

LMD = $S-S$
$= a-S$
$= a-S+S$
$= a-b+c$



② Right most Derivative:

RMD = $S+S$
$= S+c$
$= S-S+c$
$= S-b+c$
$= a-b+c$



③ Parse Tree:

It is a graphical representation of symbol that can be terminals or non-terminals.

Properties:

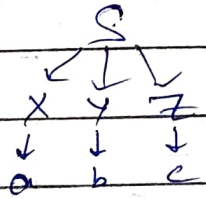
1. Root is always the start symbol
2. All leaf nodes are terminals
3. All interior nodes are Non-Ter.

eg: $S \to XYZ$

$X \to a$

$Y \to b$

$Z \to c/d$



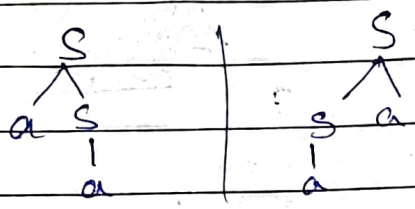
$L = \{abc, abd\}$

Ambiguous Grammar:

→ If we have more than 1 parse tree possible.
∴ Grammar is ambiguous.

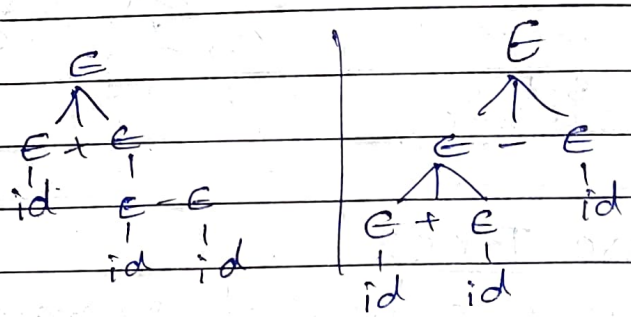
Q.) $S \to aS \mid Sa \mid a$

input - aa



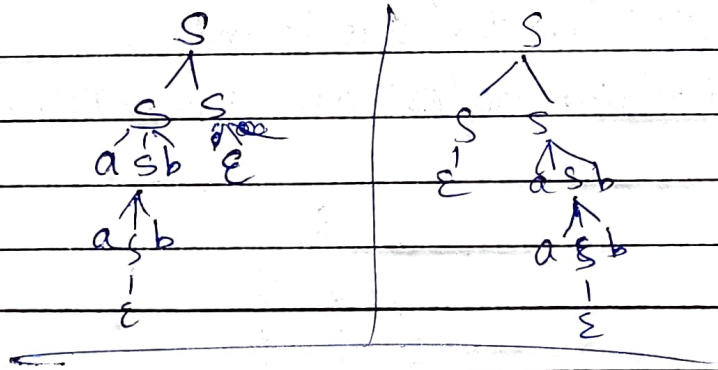
Ambig.

Q) $E \to E+E$
$E \to E-E$
$E \to id$
input = id+id-id

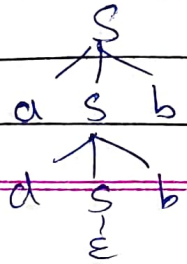


Ambi

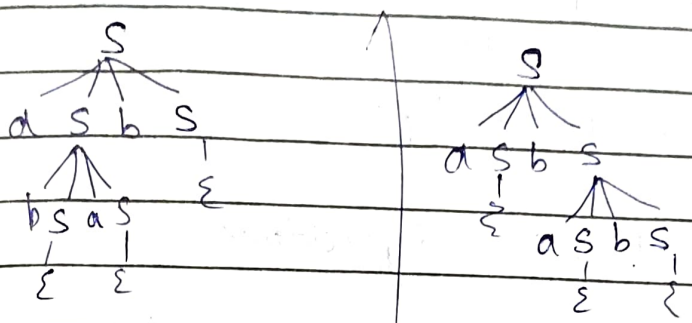
Q) $S \to aSb \mid SS \mid \varepsilon$ "aabb" input



Ambi



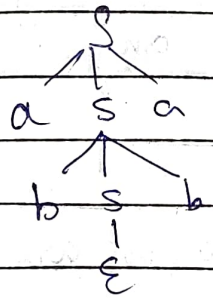
Q) $S \rightarrow aSbS \mid bSaS \mid \varepsilon$ input = abab



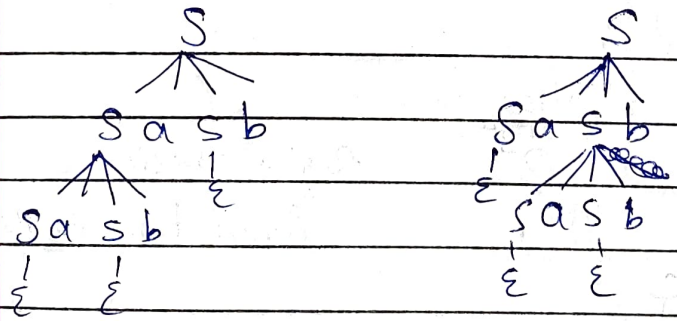
Ambi

Q) $S \rightarrow aSa \mid bSb \mid \varepsilon$

~~Non-Ambi~~ Un-Ambi



Q) $S \rightarrow SaSb \mid \varepsilon$ (input = abab)



Unambi

abab aabb

(1) ⇒ For Associative Problem, make | Left Recursive | ($E \to E+id$)
→ Make Left associative or Right associative

FULLSCAPE PAGE NO.: DATE

(2) Ambigous → Un-ambiguous

(1) → For Precedance Problem:

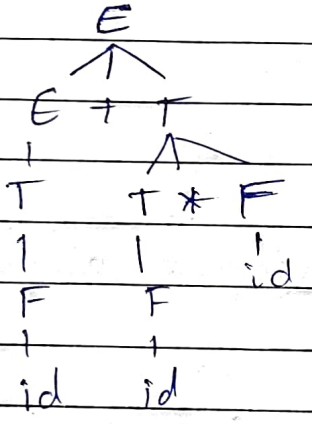
⇒ Introduce several differanant symbols like:

eg: $E \to E+E \mid E*E \mid id$

"+" < "*" : precidance

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

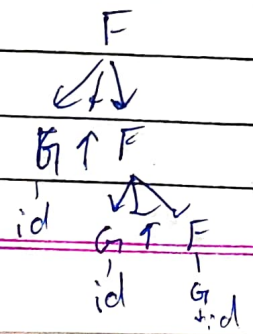
} Once we have reached T, we cannot generate any +



} So, we are taking care of precedence by defining diff. level.

Q.) $(2 \uparrow (3 \uparrow 2))$ 2^{3^2} Right associative (Grow towards Right)

ans: $F \to G \uparrow F \mid G$
$G \to id$



Q) $B \rightarrow B \text{ or } B \mid B \text{ and } B \mid \text{not } B \mid t \mid f$

$B \rightarrow B \text{ or } F \mid F$

$F \rightarrow F \text{ and } G \mid G$

$G \rightarrow \text{not } G \mid t \mid f$

or < and < not precidence

or, and } left associative

Q) $A \rightarrow A \$ B \mid B$

$B \rightarrow B \# C \mid C$

$C \rightarrow C @ D \mid D$

$D \rightarrow d$

Tell associativity & precedance

ans: $\$ < \# < @$ precedance

A, B, C left associative

Q) $E \rightarrow E + E \mid E * E \mid F$

$F \rightarrow id - id$

+ and * have the same precedance

- has high precedance than + & *

How to remove left Recursion?

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

① Direct $B \rightarrow Ba$

② Indirect $A \rightarrow Sm$

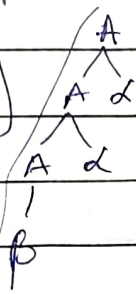
$S \rightarrow Ax$

FULLSCAPE
PAGE NO.: DATE

Recursion

$A \to A\alpha \mid \beta$ — Left Rec.

lang. = $\beta\alpha^*$

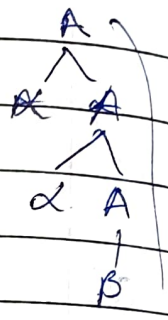



```
A(){
    A();
    α;
}
```

Infinite loop

$A \to \alpha A \mid \beta$ — Right Rec.

lang. = $\alpha^*\beta$




```
A(){
    α;
    A();
}
```

Not going Infinite loop as α checks condition.

∴ L.R. —convert→ R.R.

(*) $A \to A\alpha \mid \beta \Rightarrow$ (**)

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

Q.) $E \to E+T \mid T$

(A: E; α: $+T$; β: T)

ans =>

$E \to TE'$

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

Remove L.R.

Q) $S \to SOS \mid S \mid 01$

(A) (A) (α) (β)

ans $A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$

$S \to 01S'$

$S' \to OSISS' \mid \varepsilon$

Q) $S \to (L) \mid x$

$L \to L, S \mid S$

(A) (A) (α) (β)

ans

$L \to SL'$

$L' \to ,SL' \mid \varepsilon$

$S \to (L) \mid x$

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

Q) $A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots$

$\beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

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

ans:

$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \ldots$

$\mid \varepsilon$

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

① Grammar

Ambiguous (x) Unambiguous (✓)

(2 parse Tree)

→ No parse work for it

G

Determ (✓) Non-Determ (x)

② Grammar

LR (x) RR (✓)

→ Top down parse doesn't Accept it.

→ convert L.R. to R.R.

FULLSCAPE
PAGE NO.: DATE

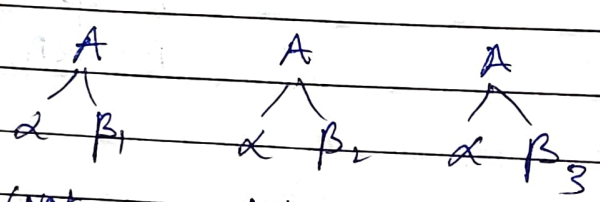
$A \to A\alpha \mid \beta$

$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

Q.) $S \to Aa$
$A \to Sb \mid c$

⇒ $S \to Aa$
$A \to Aab \mid c$ ⇒ $A \to cA'$
$A' \to abA' \mid \varepsilon$

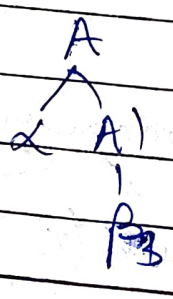
(*) ⇒ $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$ we want $\alpha\beta_3$



(Not acc.) (Backtracking) (Not acc.) (Backtracking) (Accepted ✓)

⇒ Left Factoring (Non-Deterministic ✗ → Determinis ✓)

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$
to
$A \to \alpha A'$
$A' \to \beta_1 \mid \beta_2 \mid \beta_3$



FULLSCAPE PAGE NO.: DATE

Q.) $S \rightarrow bSSaaS$
$\quad | \; bSSasb$
$\quad | \; bSb$
$\quad | \; a$

and! $S \rightarrow bSS' \; | \; a$
$S' \rightarrow SaaS \; | \; SaSb \; | \; b \rightarrow S' \rightarrow SaS'' \; | \; b$
$S'' \rightarrow aS \; | \; Sb$

(*) LL(1) Parser - (Top-Down Parser)
- Left to Right scan.
- Left Most Derivative

⇒ First (): $S \rightarrow AB$ $A \rightarrow a$

First (S) = First (A) = a

$S \rightarrow AB$ $A \rightarrow a \; | \; \varepsilon$ $B \rightarrow b$

First (S) = First (A) = {a, First (B)} = {a, b}.

⇒ Follow ():

- Follow of start symbol is always $
- for $A \rightarrow \alpha B$, follow (B) = follow (A)
- for $S \rightarrow aABC$, $B \rightarrow b$ follow (A) = first (BC) = first (B)

FULLSCAPE
PAGE NO.: DATE

		First()	Follow():	
Q:)	$S \to ABCD$	$\{b\}$	$\{\$\}$	
	$A \to b$	$\{b\}$	$\{c\}$	follow(A) = first(BCD) = first(B)
	$B \to c$	$\{c\}$	$\{d\}$	
	$C \to d$	$\{d\}$	$\{e\}$	
	$D \to e$	$\{e\}$	$\{\$\}$	follow(D) = follow(S)

Q:)

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

Q:)

	First()	Follow()
$S \to Bb \mid Cd$	$\{a, b, e, d\}$	$\{\$\}$
$B \to aB \mid \varepsilon$	$\{a, \varepsilon\}$	$\{b\}$
$C \to eC \mid \varepsilon$	$\{e, \varepsilon\}$	$\{d\}$

Q:)

	First()	Follow()
$E \to TE'$	id, (	$\{\$,)\}$
$E' \to +TE' \mid \varepsilon$	$+, \varepsilon$	$\{\$,)\}$
$T \to FT'$	id, (	$\{+, \$,)\}$
$T' \to *FT' \mid \varepsilon$	$*, \varepsilon$	$\{\$,), +\}$
$F \to id \mid (E)$	id, (	$\{\$,), +, *\}$

	F_i	F_o
(P.) $S \to ACB \mid CbB \mid Ba$	a, b, d, g, h, ε	$
$A \to da \mid BC$	d, g, h, ε	h, g, $
$B \to g \mid \varepsilon$	g, ε	$, a, h, g
$C \to h \mid \varepsilon$	h, ε	g, $, b, a

Q: Construct LL(1) Parsing Table.

⇒ $S \to AaAb \mid BbBa$
$A \to \varepsilon$
$B \to \varepsilon$

First remove
- left recursion.
- left factoring.

	First	Follow
$S \to AaAb \mid BbBa$	a, b	$
$A \to \varepsilon$	ε	a, b
$B \to \varepsilon$	ε	b, a

	a	b	$
S	$S \to AaAb$	$S \to BbBa$	
A	$A \to \varepsilon$	$A \to \varepsilon$	
B	$B \to \varepsilon$	$B \to \varepsilon$	

⇒ The GATE HUB:

Y-10

		First()	Follow()
	$E \to TE'$	$\{id, (\}$	$\{\$,)\}$
	$E' \to \varepsilon \mid +TE'$	$\{\varepsilon, +\}$	$\{\$,)\}$
- Grammar?	$T \to FT'$	$\{id, (\}$	$\{+, \$,)\}$
- First	$T' \to \varepsilon \mid *FT'$	$\{\varepsilon, *\}$	$\{+, \$,)\}$
- Follow	$F \to id \mid (E)$	$\{id, (\}$	$\{*, +, \$,)\}$

eg:

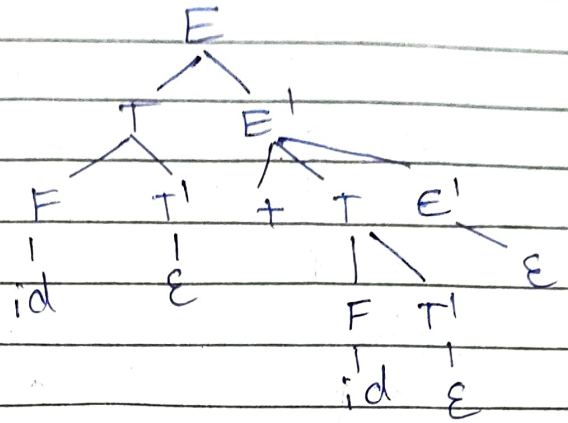
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

Stack	input	Production
$\$E$	id+ id $	$E \to TE'$
$\$E'T$ ← check →	id+ id $	$T \to FT'$
$\$E'T'F$	id+ id $	$F \to id$
$\$E'T'id$	id + id $	Pop();
$\$E'T'$	+ id $	$T' \to \varepsilon$
$\$E'$	+ id $	$E' \to +TE'$
$\$E'T+$	+ id $	Pop()
$\$E'T$	id $	$T \to FT'$
$\$E'T'F$	id $	$F \to id$
$\$E'T'id$	id $	Pop()
$\$E'T'$	$	$T' \to \varepsilon$
$\$E'$	$	$E' \to \varepsilon$
$	$	Accepted

(Reversed stack)

Top

(LL(1) Parsing) or (Predictive parser)
(TOP-DOWN PARSER)



	First( )	Follow( )
S → aBDh	{a}	{$}
B → cC	{c}	{g, f, h}
C → bC | ε	{b, ε}	{g, f, h}
D → EF	{g, f, ε}	{h}
E → g | ε	{g, ε}	{f, h}
F → f | ε	{f, ε}	{h}

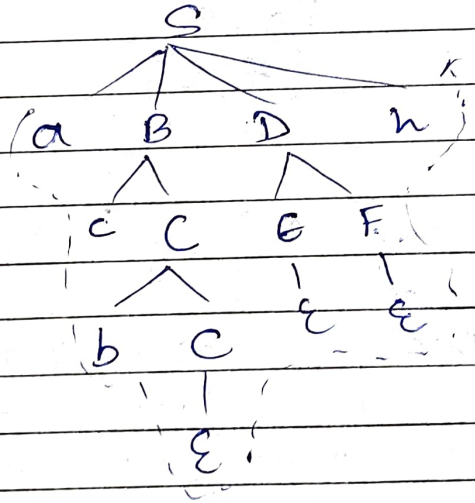
M	a	b	c	f	g	h	$
S	S→aBDh						
B			B→cC		~~ε~~	~~ε~~	
C		C→bC			C→ε	C→ε	C→ε
D				D→EF	D→EF	D→EF	
E				E→ε	E→g	E→ε	
F				F→f		F→ε	

LL(1) parsing Table

Q) and:

Stack	input	production
$ S	a a c b h $	S → a B D h
$ h D B a	a c b h $	POP();
$ h D B	c b h $	B → c C
$ h D C c	c b h $	POP();
$ h D C	b h $	C → b C
$ h D C b	b h $	POP();
$ h D C	h $	C → ε
$ h D	h $	D → E F
$ h F E	h $	E → ε
$ h F	h $	F → ε
$ h	h $	POP();
$	$	Accept

Parse Tree



acbh

By- Pathik455

PAGE NO. DATE

Bottom Up Parser

- LR parser (Unambiguous)
  - LR(0)
  - SLR(1)
  - CLR(1)
  - LALR(1)
- Operator Parser (Ambiguous)

RMD in Reverse
(Right most derivative)

LR(1)

LR(0)

→ Closure (I)

1. Add LR(0) item I to closure (I)
2. If $A \to \alpha \cdot B\beta$ is LR(0) item I and $B \to \gamma$ is in G then Add $B \to \cdot\gamma$ to closure (I) also.
3. Repeat above two steps from every newly added LR(0) item.

→ Goto (I, X)

1. Add LR(0) item I by moving dot after X.
2. Apply closure to the result obtained in step 1.

eg: $S \to AA$
$A \to aA \mid b$

closure $(S \to .AA)$

$\Rightarrow$ $S \to .AA$
$A \to .aA$
$A \to .b$
} closure $\Rightarrow$ same

closure $(S \to A.A)$

$\Rightarrow$ $S \to A.A$
$A \to .aA$
$A \to .b$
} closure $\Rightarrow$ same

Goto $(S \to .AA, A)$

$\Downarrow$

$(S \to A.A) \Rightarrow$ closure

$\Downarrow$

$S \to A.A$
$A \to .aA$
$A \to .b$

Goto $(S \to A.A, A)$

$\Downarrow$

closure $(S \to AA.)$

$\Downarrow$

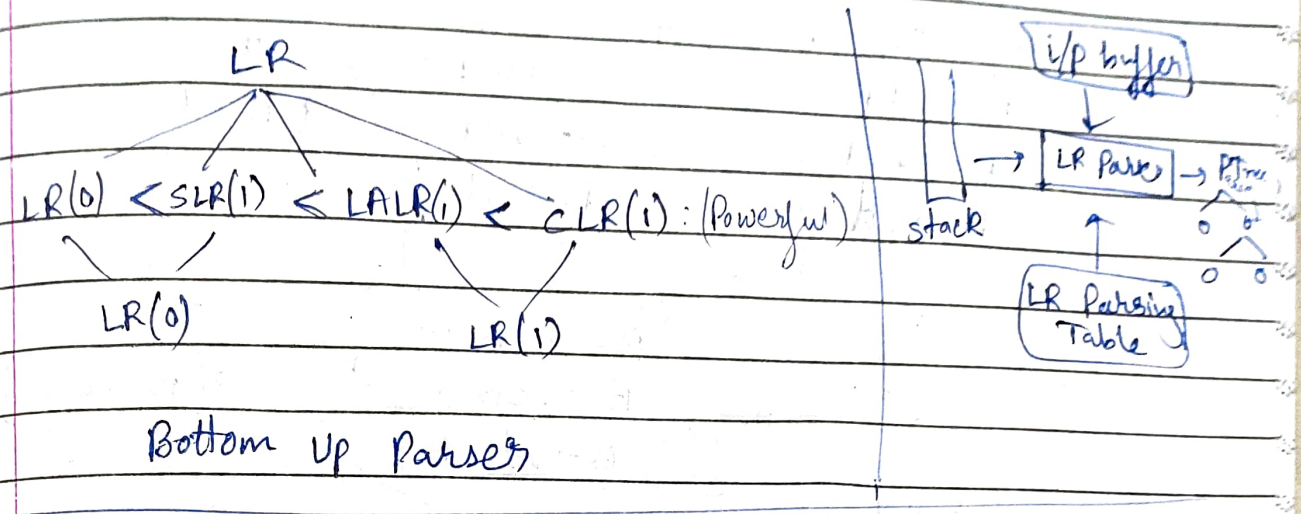
$S \to AA.$

Last Minute Tuisions

By- Pathik455

PAGE NO.: FULLSCAPE DATE

LR Parsers



(1) LR(0):

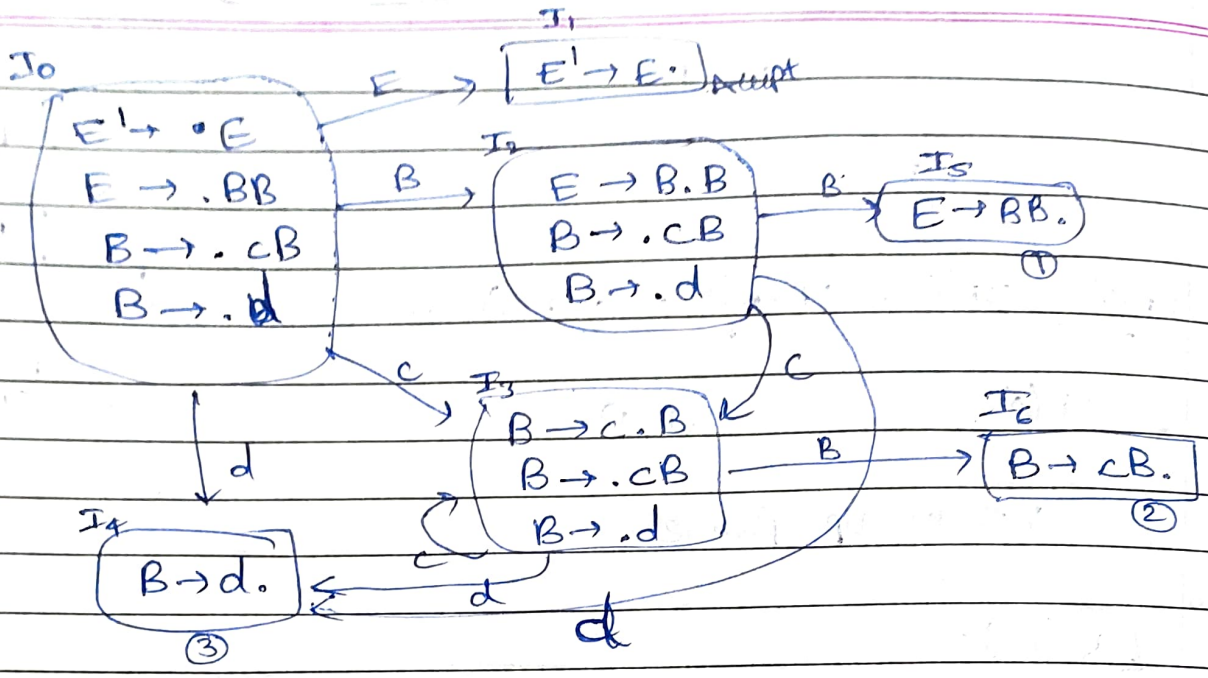
Step- (i) Augment Grammer
(ii) Draw Conical Collections of LR(0) item / DFD
(iii) Number the Productions
(iv) Create the parsing table
(v) Stack implementation
(vi) Draw parsing tree

eg: $E \to BB$
$B \to cB \mid d$ - ccdd $

ans: (i) $E' \to E$
$E \to BB$
$B \to cB \mid d$

(ii) (LR(0) items)
$E' \to .E$
$E \to .BB$
$B \to .cB$
$B \to .d$

$E' \to .E$
$E \to BB$
$B \to .cB \mid d$



(iii) $E' \to E$

① $E \to BB$

② $B \to cB$

③ $B \to d$

(iv) Parsing Table:

state	Action c	Action d	Action $	Goto E	Goto B
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		

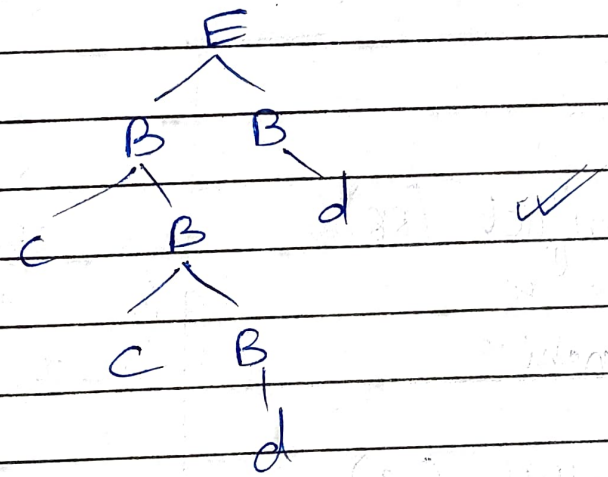
(*) If a column has 'r' and 's' together then it is not LR(0)

(v) stack implementation

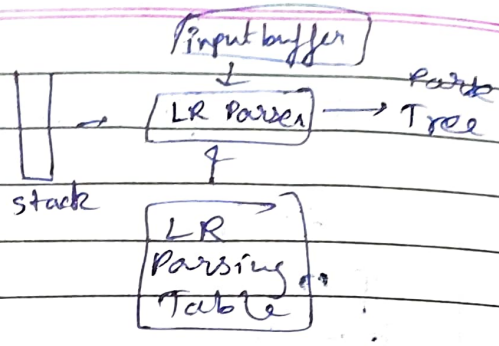
stack	Input	Action
$0	ccdd$	shift c to stack & goto s_3
$0c3	cdd$	shift c and s_3
$0c3c3	dd$	shift d and s_4
$0c3c3d4	d$	reduce r_3, B→d
$0c3c3B6	d$	reduce r_2, B→cB
$0c3B6	d$	reduce r_2, B→cB
$0B2	d$	shift d, s_4
$0B2d4	$	reduce r_3, B→d
$0B2B5	$	reduce r_1, E→BB
$0E1	$	Accept
✓	✓	

Draw Parse tree:

(vi)



(A) SLR(1):



⇒ Steps same as LR(0)

↓

reduce only in follow()

Parsing Table:

State	Action			Goto	
	c	d	$	E	B
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		
I_6	r_2	r_2	r_2		

Conflict:

① Reduce-Reduce conflict (RR)
LR(0) hota hai ✓
SLR(1) me nahi ✗

② Shift-Reduce Conflict (SR)
LR(0) ✓
SLR(1) ✗

$I_4 \Rightarrow B \to d.$
→ follow of B = {c, d, $}
$I_5 \Rightarrow E \to BB.$
follow of E = {$}
$I_6 \Rightarrow r_2 \to B \to cB$

LR(0) or SLR(1) or not ?

Q. $E \to T+E \mid T$

$T \to i$

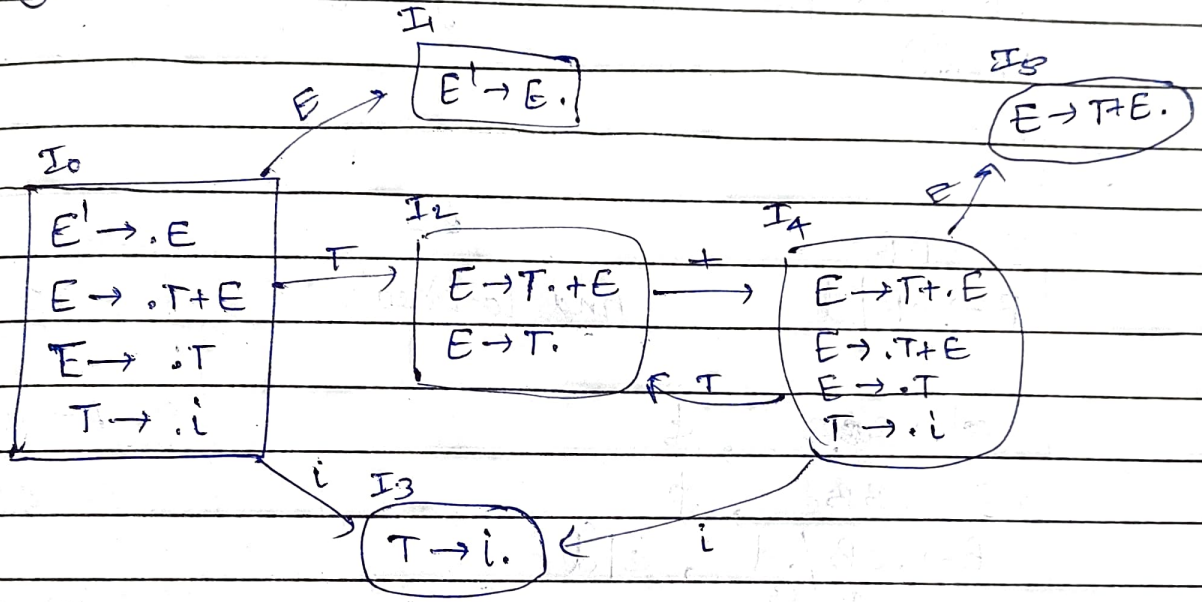
	follow()
E	{$}
T	{+, $}

ans: $E' \to E$

① $E \to T+E$

② $E \to T$

③ $T \to i$



State	Action			Goto	
	+	i	$	E	T
I_0		S_3		1	2
I_1			Accept		
I_2	S_4 / r_2		r_2		
I_3	r_3		r_3		
I_4		S_3		5	2
I_5	r_1		r_1		

SLR me nikal jayega

∴ LR(0) ✗ (SR-conflict) ∴ SLR(1) ✓

CLR(1)

LR(1) item = LR(0) item + look Ahead

LR(0) ⇒ Put reduce in full row
SLR(1) ⇒ Put reduce in follow of P
CLR(1) ⇒ Put reduce in only on Lookahead.
LALR(1)

Q) $E \to BB$
$B \to cB \mid d$ find Lookahead.

ans: $E' \to .E$, $
① $E \to .BB$, $
$B \to .cB$ / $.d$, $c|d$
② ③

I_0
$E' \to .E$, $
$E \to .BB$, $
$B \to .cB \mid .d$, $c|d$

$E \to$ I_1: $E' \to E.$, $

$B \to$ I_2: $E \to B.B$, $
$B \to .cB \mid .d$, $

$B \to$ I_5: $E \to BB.$, $

$c \to$ I_3: $B \to c.B$, $c|d$
$B \to .cB \mid .d$, $c|d$

$B \to$ I_8: $B \to cB.$, $c|d$

$d \to$ I_4: $B \to d.$, $c|d$

$c \to$ I_6: $B \to c.B$, $
$B \to .cB \mid .d$, $

$B \to$ I_9: $B \to cB.$, $

$d \to$ I_7: $B \to d.$, $

$B \to \alpha$

eg: $A \to \alpha.B\beta$, $c|d$ (First)

$B \to .L$, a (First(β))

if β was not there,
$A \to \alpha.B$, $c|d$
$B \to .L$, $c|d$

state	Action			Goto	
	c	d	$	E	B
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		

CLR(1) ✓

convert to LALR(1) →

Find
- Production same & LookAhead diff.

① I_3, I_6
② I_4, I_7
③ I_8, I_9

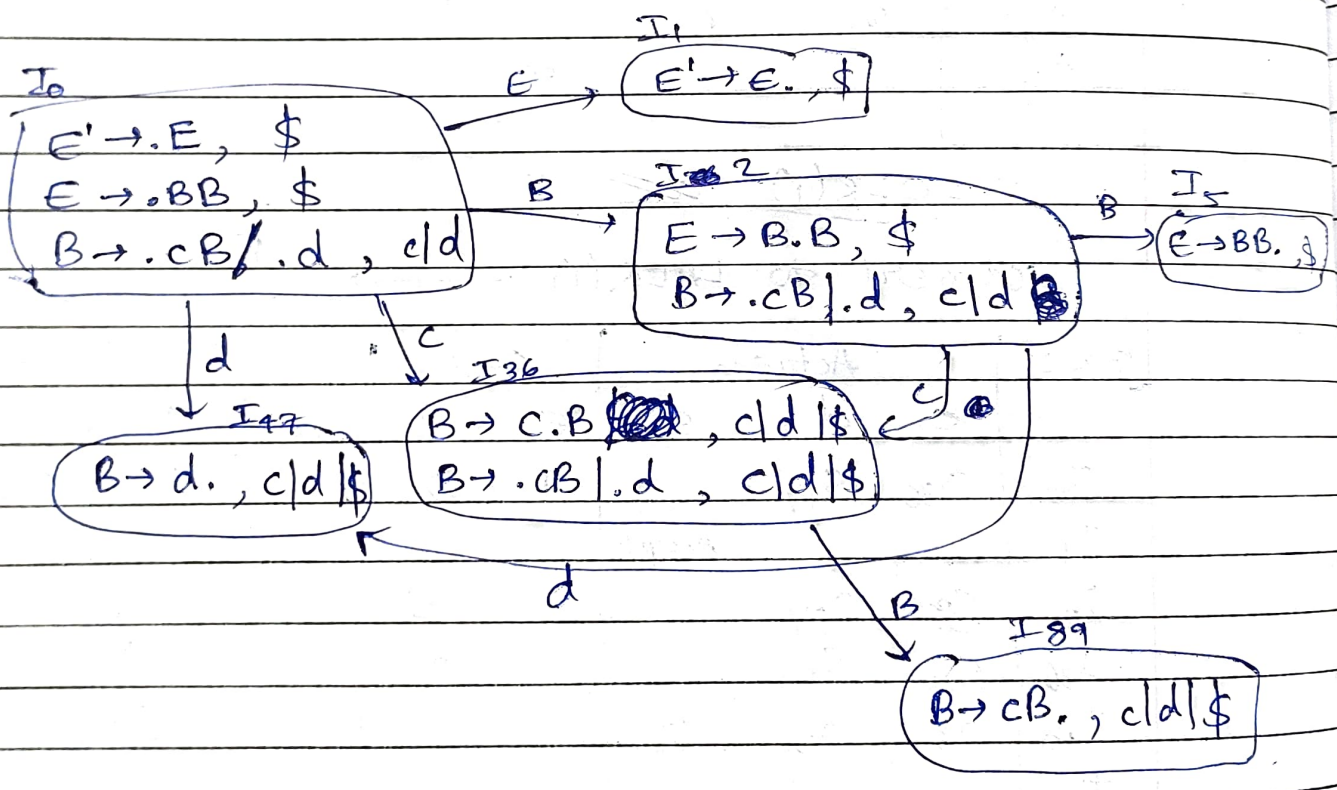
↓ convert into

I_{36}, I_{47}, I_{89}

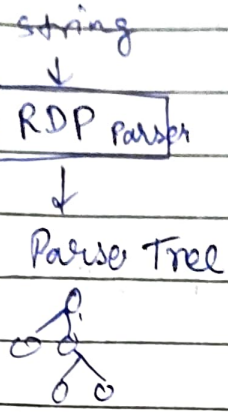
state	Action			Goto	
	c	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
~~I_{36}~~	S_{36}	S_{47}			89
~~I_{47}~~			r_3		
I_{89}	r_2	r_2	r_2		
~~I_{89}~~			r_2		

Rewrite →

state	Action			Goto	
	c	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		



Recursive Descent Parser:



- Top-Down Parsing.
- Backtracking

Example Backtracking

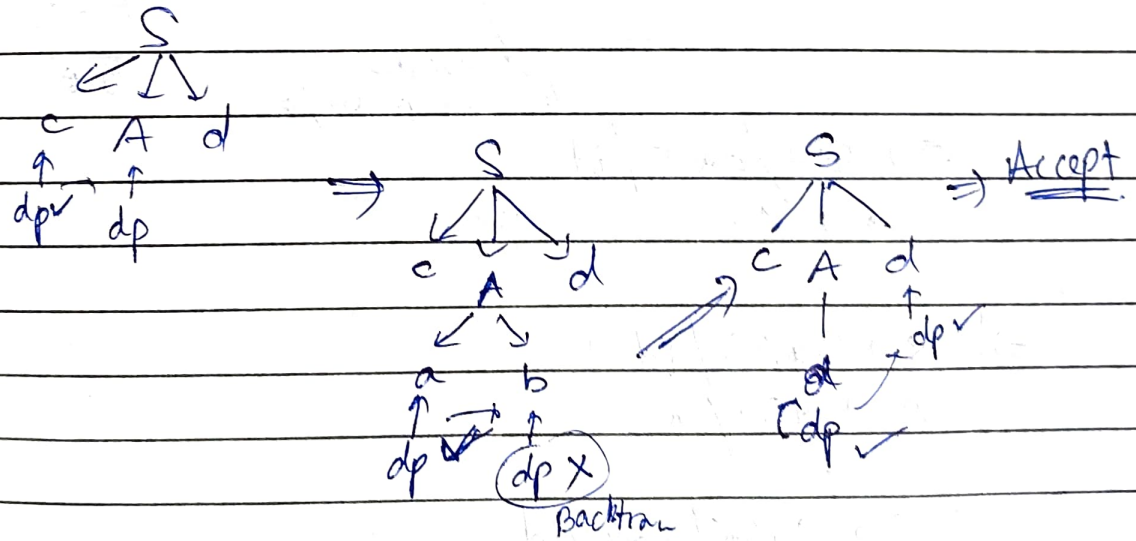
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

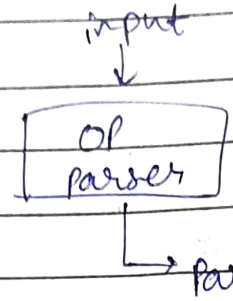
and input string $\omega = cad$ (i/p i/p i/p)

i/p = input

dp = descent pointer.



Operator - Precedence: Parser



- Bottom-Up Parser.

Operator Grammer: ① No Epsilon in right side
② No adjacent variable

Example:

⇒ $E \to E+E / E-E / id$

⇒ $E \to EAE / id$
$A \to + / *$

} Not operator grammer

converting by $A \to +/*$

$E \to E+E / E*E / id$
$A \to + / *$ ✓

⇒ $S \to SaAS / id$
$A \to aSa / a$

∴ $S \to SaSaS / SaS / id$ ✓
$A \to aSa / a$

Example:

AB ✗
A*B ✓
A+B ✓
A/C ✓
A−B ✓
a+ ✓

FULLSCAPE
PAGE NO.: DATE

1. Check OPG or Not
2. Operator Precedence Relation Table
3. Parse the given String
4. Generate Parse Tree.

eg: $T \rightarrow T+T / T*T / id$

Basics

id, a, b, c ⇒ High

$ ⇒ Low

+ > + (left side)

* > *

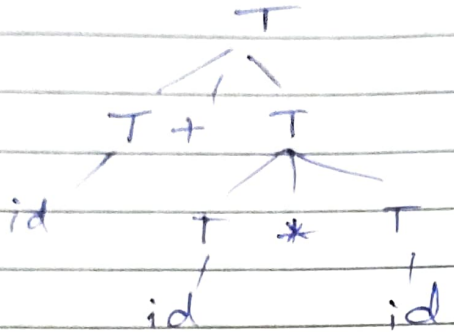
id ≠ id

$ A $

	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	≠	>
$	<	<	<	A

Stack	Relation	Input	Comment
$	<	id+id*id $	shift id
$id	>	+id*id $	reduce T→id
$T	<	+id*id $	shift +
$T+	<	id*id $	shift id
$T+id	>	*id $	reduce T→id
$T+T	<	*id $	shift *
$T+T*	<	id $	shift id
$T+T*id	>	$	reduce T→id
$T+T*T	>	$	reduce T→T*T
$T+T	>	$	reduce T→T+T
$T	A	$	Accept

Generate Parse Tree:



	+	*	a	b	c	d	$
+	>	<	<	<	<	<	>
*	>	>	<	<	<	<	>
a	>	>					>
b	>	>					>
c	>	>					>
d	>	>					>
$	<	<	<	<	<	<	A

Q.) $E \rightarrow E+T \mid T$
$T \rightarrow T*V \mid V$
$V \rightarrow a \mid b \mid c \mid d$

	+	*	id	$	
+	>	<	<	>	
*	>	>	<	>	
id	>	>	-	>	
$	<	<	<	A	

"a+b*c*d"

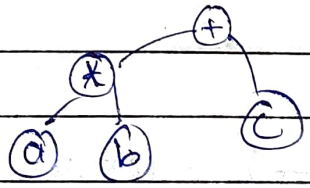
Stack	Rel.	String Input	Comment
$	<	a+b*c*d$	shift a
$a	>	+b*c*d$	reduce V→a
$V	<	+b*c*d$	shift +
$V+	<	b*c*d$	shift b
$V+b	>	*c*d$	reduce V→b
$V+V	<	*c*d$	shift *
$V+V*	<	c*d$	shift c
$V+V*c	>	*d$	reduce V→c
$V+V*V	>	*d$	reduce T→V
$V+V*T	>	*d$	reduce E→T
$V+V*E	>	*d$	??

∴ No Production found for reduction operation.

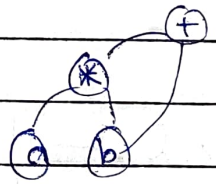
· Since, parsing process failed to complete, the given input cannot be parsed by the given grammer using OPP method :(.

DAG (Direct Acyclic Graph)

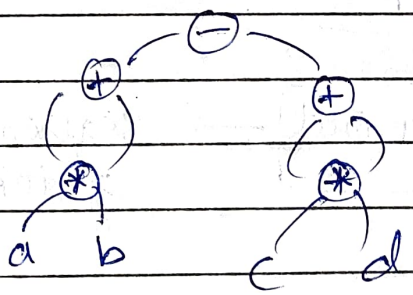
eg: a * b + c



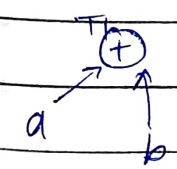
a * b + b



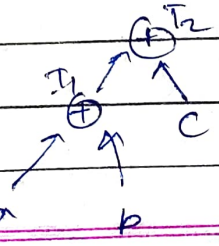
eg: ((a*b) + (a*b)) − ((c*d) + (c*d))



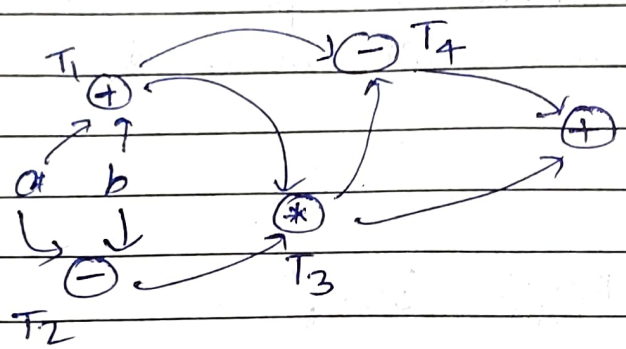
eg: $T_1 = a + b$



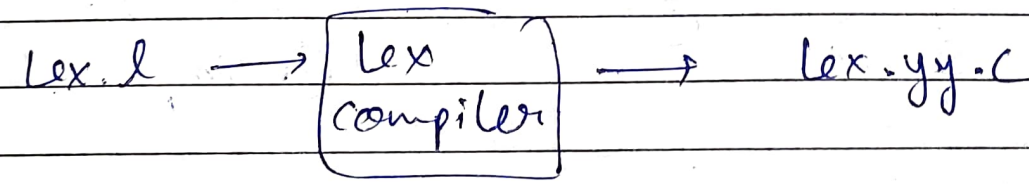
$T_2 = T_1 + c$

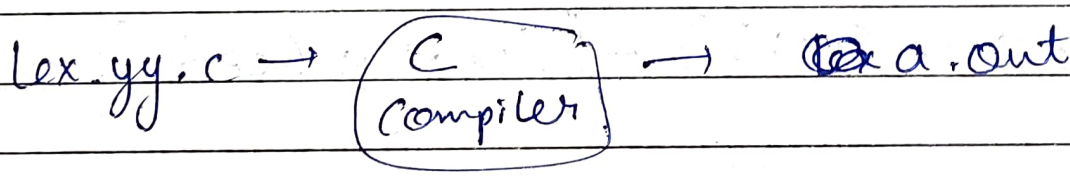


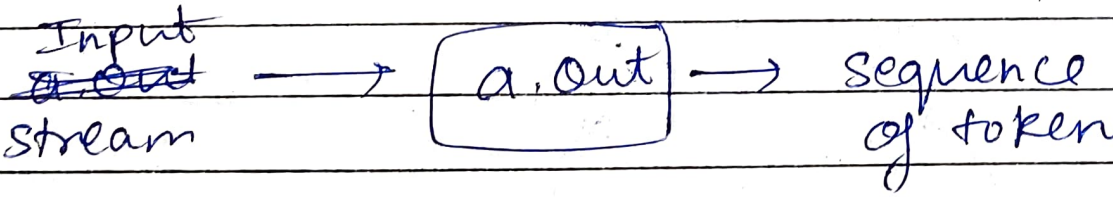
Q) $T_1 = a+b$ $T_2 = a-b$ $T_3 = T_1 * T_2$ $T_4 = T_1 - T_3$ $T_5 = T_4 + T_3$



Lex :







YACC

Generates LALR Parser.

(Yacc)

parser.y → [Yacc compiler] → pqy.tab.c

parser.tab.c → [C compiler] → a.out

Input Stream → [a.out] → (sequence of token) output

Definition

%%

Rules

%%

Supplementary Code