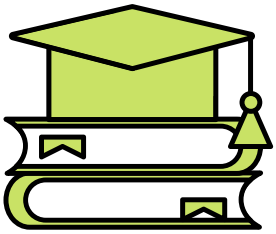


Gatenotes.in

GATE
Computer Science

Ravindrababu Ravula GATE CSE

Hand Written Notes

-: SUBJECT :-

Compiler Design

-7085148007.

INDEX

NAME: Rakesh Nama. STD.: _____ SEC.: _____ ROLL NO.: _____ SUB.: Compiler Design.

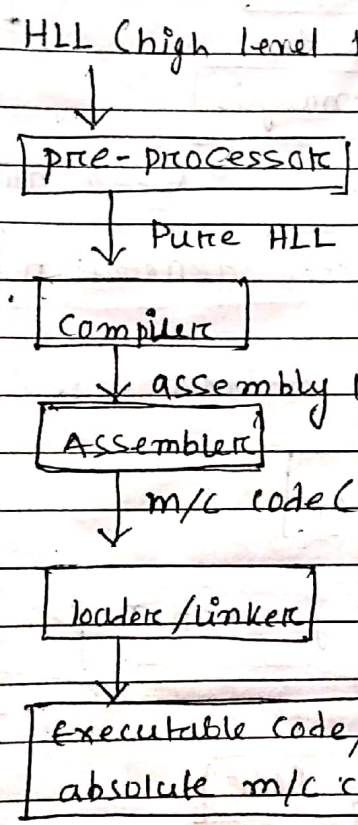
INTRODUCTION

Compiler Design

(www.gatenotes.in)

- **Introduction of and various phases of compiler Design :-**

→ The main aim of compiler design is to convert a pure high level language into low level language.



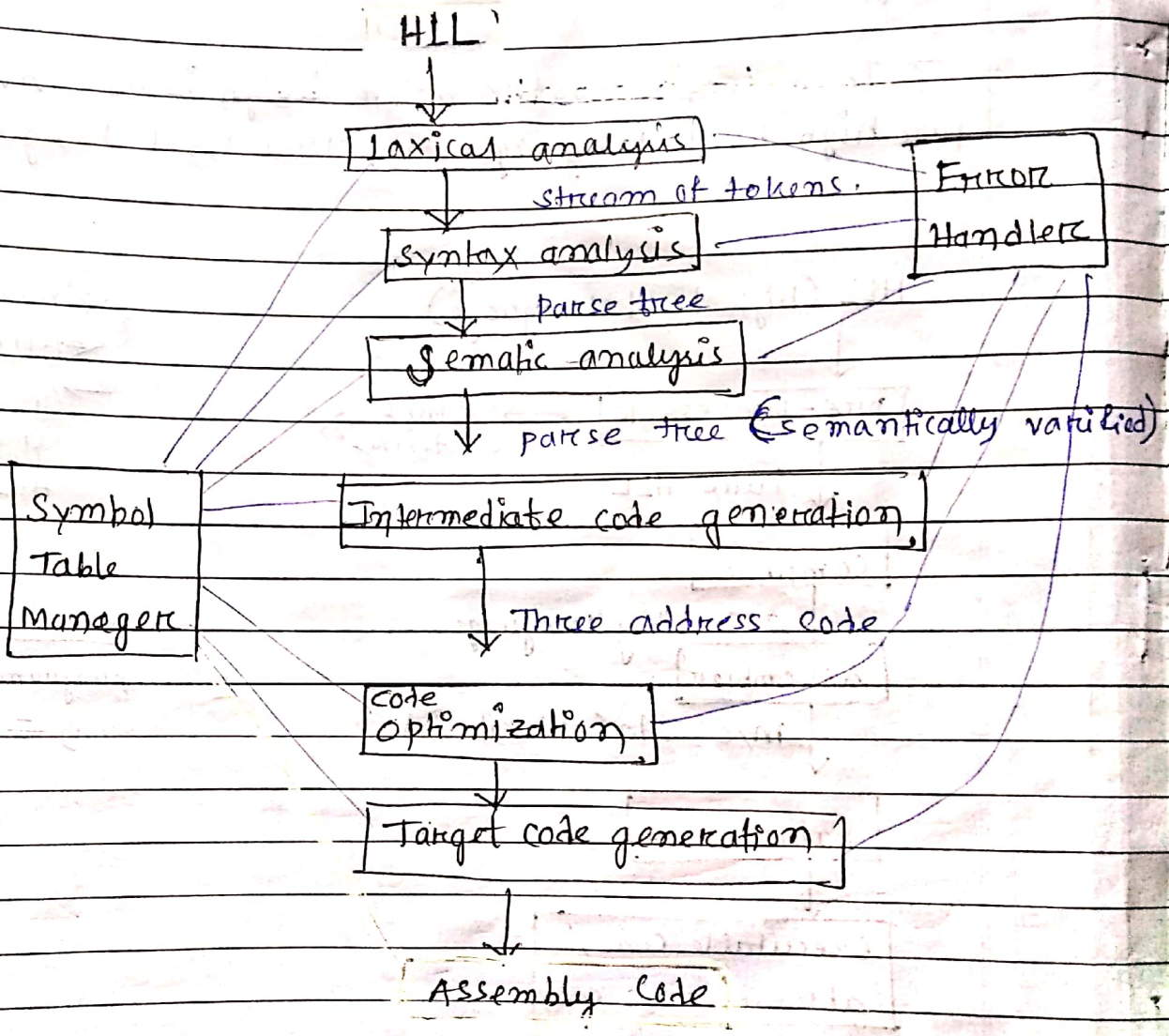
{ <#include> – file inclusion
<#define> – Macro expansion
→ (If any language contain this type of line it called HLL)

→ pure HLL means program not contain any '#' line.

→ preprocessor : is responsible for
1. Macro expansion
2. File inclusion.

→ loader : is responsible for
1. Allocation.
2. Re-allocation.
3. Linking.
4. Loading.

Compiler phases



Lex and yacc : are tools used in unix operating system for compiler design. first compiler is FORTRAN. It took 18 years to build it.

Lexical analysis :

HLL ~~te~~ext are source text is broken into tokens.

ex: If (A>B) → 10 tokens.
a = 10;

Example of all the phases of compiler —

$x = a + b * c$; → source program.

⇓

Lexical analyser

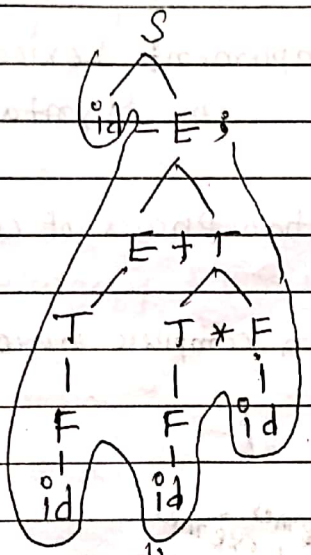
⇓

id = id + id * id → (id = identifier)

⇓

heart of the compiler — Syntax analyser ⇐

S → id = E;	(CFGs)
E → E+T/T	
T → T*F/F	
F → id	



⇓

Semantic analyser (parse tree semantically verified)

⇓

ICG ✓ frontend

⇓

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

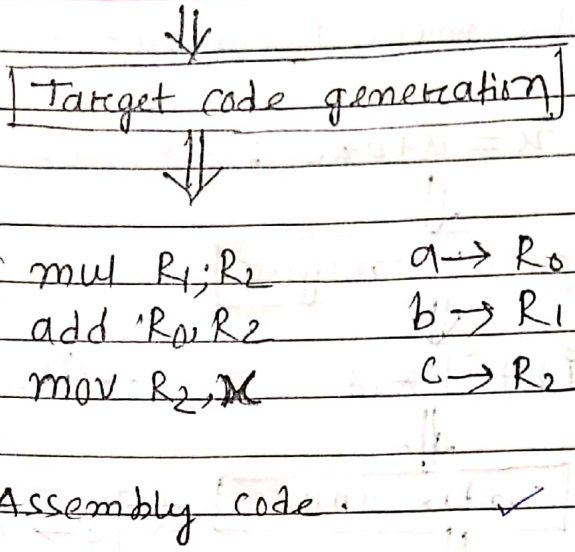
⇓

Code optimization

⇓

$t_1 = b * c$

$x = a + t_1$



→ Lex (tool) used to implement 'Lexical analyser'.
→ yacc (tool) " " " 'Syntax analyser'.

→ Practically we have two phases of compiler — frontend and Backend.
→ To do any project on compiler have tool called 'LANCE'.

Lexical analyser:

Lexemes Lexemes

```
int max(x,y)
int x,y;
/* find max of x and y */ X
{
    return (x>y ? x:y);
}
```

→ Lexical analyser removing the comments and ors white spaces.
→ main function of lexical analyser is converting Lexemes into token.
→ Show the errors. (If getting any error).

Ex: How many token is there in this particular lines ==

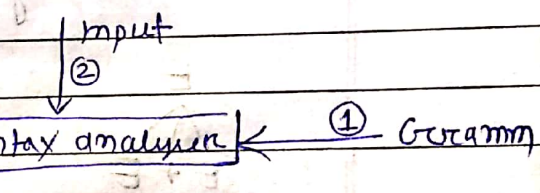
① int/ max/(/x,/y/)/
int/ x,/y/;/
/* find max of x and y */
{/
return/(x>y?/x:y)/;/
}/

→ 25 (tokens)

② printf/(/"%d Hai"/,/&x/)/;/

→ 8 (tokens are there)

→ Question can come from lexical analyser is (How many token is there and what are the responsibility)



• Grammars:

$G_c = (V, T, P, S)$

V → variable
T → Terminals
P → Production
S → start symbol.

Ex: (E) → E+E
/E*E
/id

here,
V = {E}
T = {+, *, id}
start symbol = E.

(id + id * id) generate this string using given rule (LMD)

Left most derivation = (LMD)

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

Right most derivation =

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

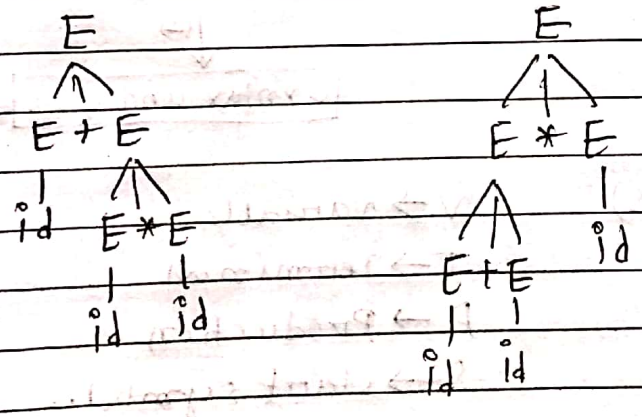
LMD =

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

RMD =

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

- Derived using parse tree -



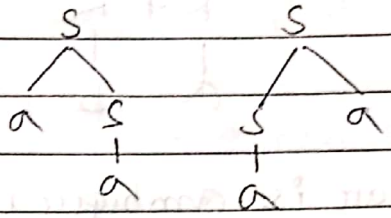
→ To the given grammer and string if we get more than one LMD or RMD and/or parse tree then the grammer is called ambiguous grammar.

→ ambiguity problem is undecidable.
(because no algo to solve the problem)

• Find given grammars are ambiguous or not :

① $S \rightarrow aS/Sa/a$

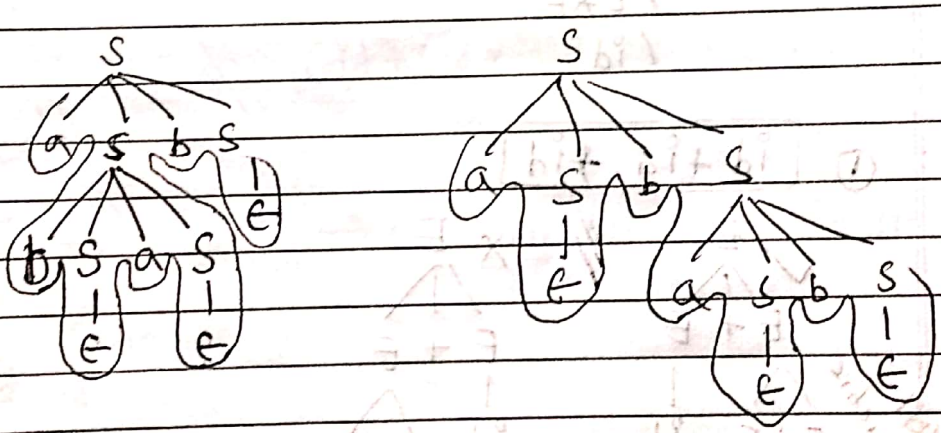
→ $w = aa$



→ This grammar is ambiguous.

② $S \rightarrow aSbS/bSaS/\epsilon$

→ $w = abab$

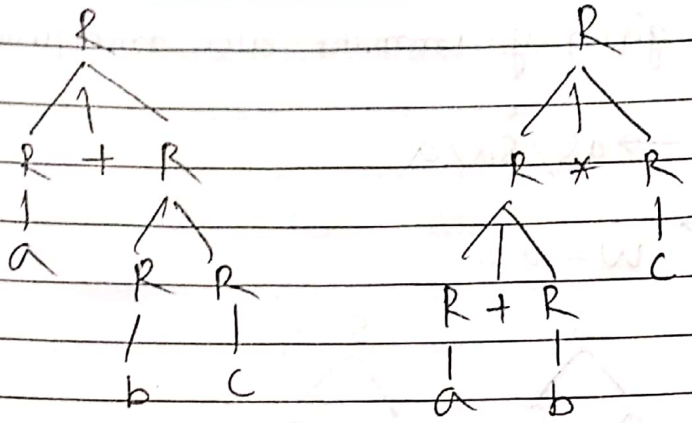


(abab) (abab)

→ To the ~~same string~~ given grammar and string we get more than one parse tree, so that the given grammar is ambiguous.

③ $R \rightarrow R+R/RR/R^*/a/b/c$.

→ $w = a+bc$



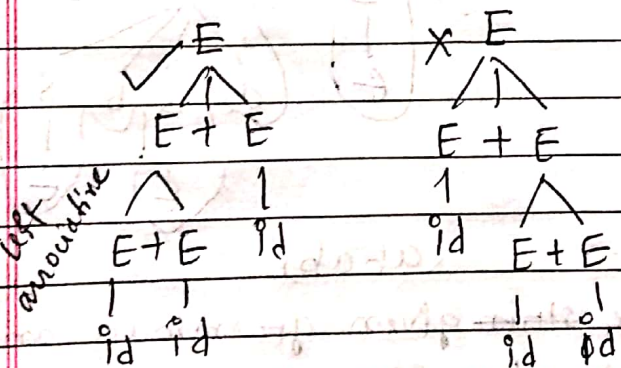
→ this grammar is ambiguous grammar.

• Ambiguous grammar and making them unambiguous:

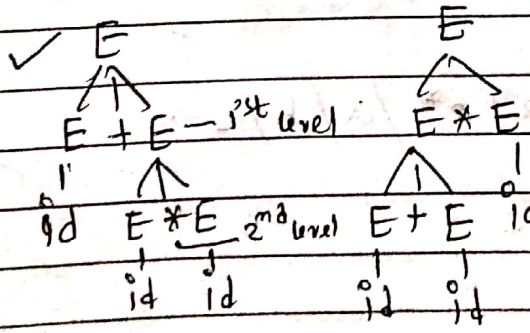
Ex-1

$E \rightarrow E+E$
$/ E*E$
$/ id$

① id + id * id

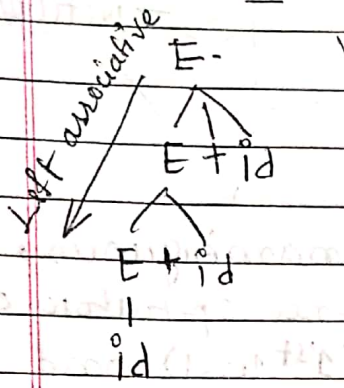


② id + id * id



~~① '+' operators are left associative.~~
~~① 'id' operator is~~

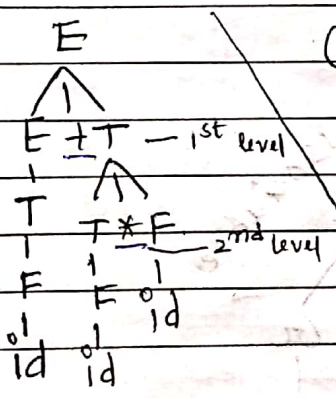
① $E \rightarrow E+id / id$ (This grammar is left recursive)



$((id+id)+id)$

→ Now the grammar is unambiguous.

② $E \rightarrow E+T / T$
$T \rightarrow T*F / F$
$F \rightarrow id$



$(id+(id*id))$

→ Now the grammar is unambiguous.

$(2^{(3^2)}) = 2 \uparrow 3 \uparrow 2$

precedence: $\uparrow > * > +$
3rd level, 2nd level, 1st level

③ $E \rightarrow E+T / T$
$T \rightarrow T*F / F$
$F \rightarrow G \uparrow F / G$
$G \rightarrow id$

here, '+' and '*' is left associative,
'↑' is right associative.

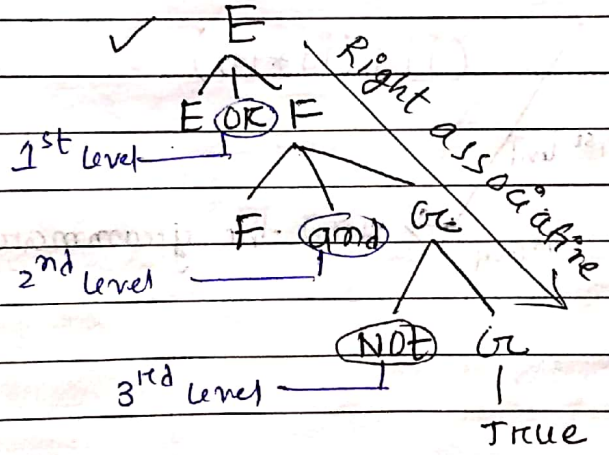
④ boolean expression,

Convert

bExp → bExp OR bExp.	E → E OR F / F
/ bExp and bExp.	F → F and Gr / Gr
/ not bExp	Gr → Not Gr / True / False.
/ True	
/ False.	

→ This grammar is unambiguous, because lowere precedence operator closest to the start symbol (1st level) and higher prece operator are least level.

→ And this grammar follow the Associativity rule.



⑤ R → R+R (convert into Unambiguous grammar)

/ RR	E → E + T / T
/ R*	T → T * F / F
/ a	F → F* / a / b / c
/ b	
/ c	

↳ Now this gramma is unambiguous grammar.

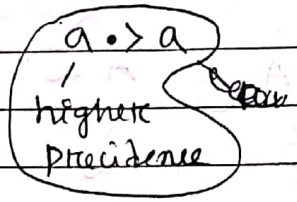
⑥ Given grammar –

A → A $B / B ($, #, @ are operators)
B → B # C / C
C → C @ D / D (here, $, #, @ are left recursive)
D → d

→ $ ·> $

·>

@ ·> #



→ $ <· # <· @ Operators are evaluated.

⑦ Given grammar –

E → E * F → here, '*' is left associative. (left recur)
/ F + E → '+' is Right associative. (Right recur)
/ F

F → F – F → Associativity is undefined.
/ id because, here, '–' define left or Right associative both.

→ And This grammar is ambiguous.

* ≐ + (equal).

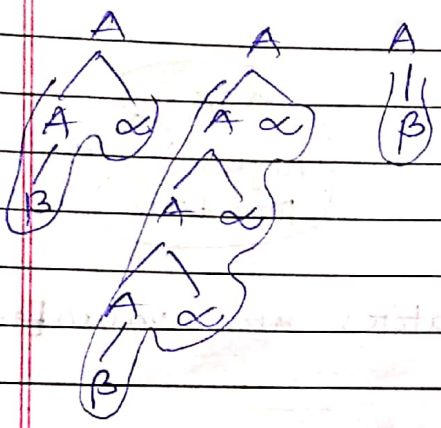
(high precidence) * ·> *

+ <· + (high precidence)

Recursion

Left Recursion

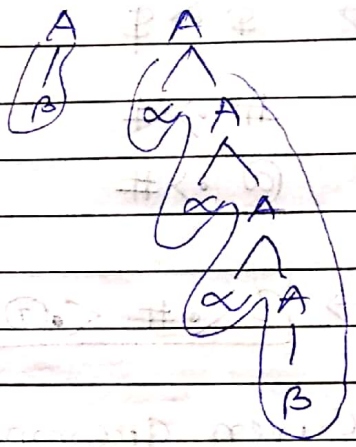
$A \rightarrow A\alpha / \beta$



$L = (\beta\alpha^*)$

Right Recursion

$A \rightarrow \alpha A / \beta$



$L = (\alpha^* \beta)$.

- $\beta\alpha^*$

$A \rightarrow \beta\alpha^*$

**
$$A \rightarrow \beta A' \qquad A' \rightarrow \epsilon / \alpha A' \iff A \rightarrow A\alpha / \beta.$$

→ To eliminate left recursion we can follow this above rule.

Example — eliminate the left recursive,

$$\underset{(A)}{E} \rightarrow \underset{(A)}{E}+\underset{(\alpha)}{T} / \underset{(\beta)}{T}$$

$$\Updownarrow$$

$$E \rightarrow TE' \qquad E' \rightarrow \epsilon / +TE'$$

→ This is equivalent grammar and not left recursive grammar.

Example- Eliminate left recursion from this grammar -

① $S \rightarrow S0S|S/01$
(A) (A) (α) (B)

$\Rightarrow$ $S \rightarrow 01S'$

$S' \rightarrow \epsilon / 0S1SS'$

② $S \rightarrow (L) / x.$
$L \rightarrow L,S / S.$
(A) (A) (α) (B)

$\Rightarrow$ $S \rightarrow (L) / x.$

$L \rightarrow SL'$

$L' \rightarrow \epsilon / ,SL'$

③ $A \rightarrow A\alpha_1 / A\alpha_2 / A\alpha_3 / \ldots$
$/ \beta_1 / \beta_2 / \beta_3 / \ldots$

$\Rightarrow$ $A \rightarrow \beta_1 A' / \beta_2 A' / \beta_3 A' / \ldots$

$A' \rightarrow \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \ldots \epsilon.$

Grammar

Ambiguous ✗

unambiguous ✓

Grammar

Left recursive ✗

Right recursive ✓

Grammar

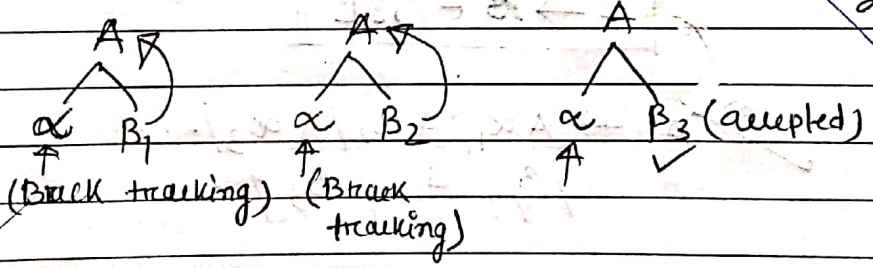
Deterministic ✓

Non-Deterministic ✗

Non-Deterministic :

$A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$ ← Left factoring

$\alpha\beta_3$

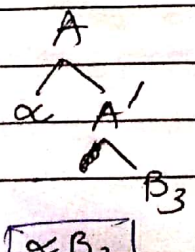


Left factoring procedure or eliminating non-determinism :

$A \to \alpha A'$

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

generate $\alpha\beta_3$



$\alpha\beta_3$

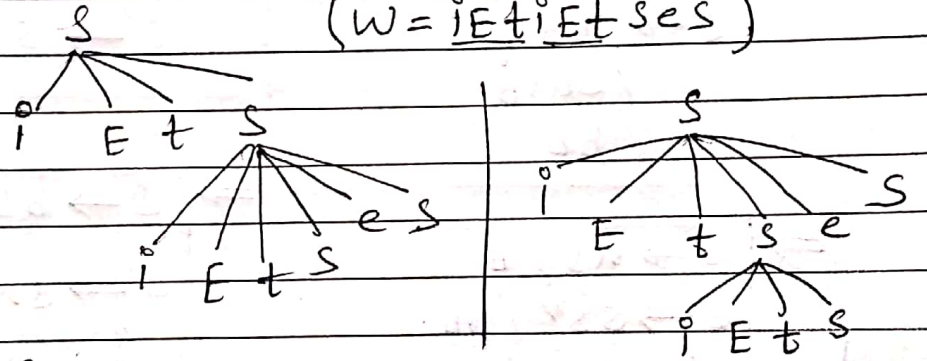
Example –

(i) $S \rightarrow iEtS$
$/\ iEtSeS$ → (Non-Deterministic)
$/\ a$
$E \rightarrow b$

left factoring

(ii) $S \rightarrow iEtSS'/a$
$S' \rightarrow \epsilon/eS$
$E \rightarrow b$ → (Deterministic)

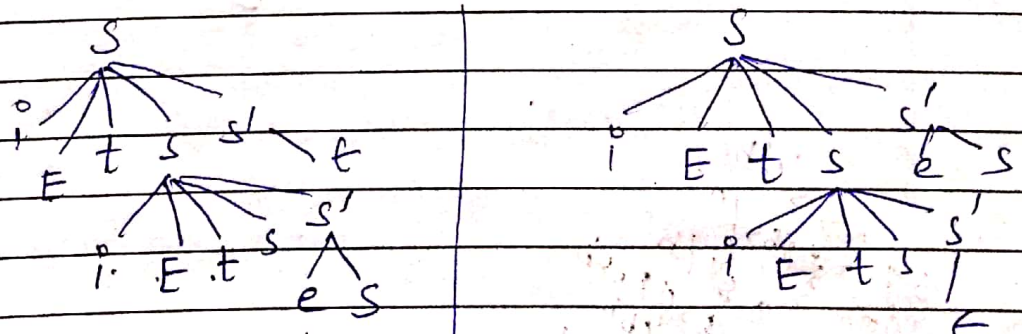
(i) $(W = iEtiEtSeS)$



(ambiguous grammar)

→ for the string we get 2 parse tree from the given grammar.

(ii) $W = iEtiEtSeS$



(also ambiguous grammar)

** → Eliminating non-determinism or left-factoring doesn't eliminate ambiguity.

Example –

(Non-Deterministic)

$S \to @SSbS$
$/ @SaSb$
$/ @bb$
$/ b$

$\Rightarrow$

$S \to aS' / b$
$S' \to SbS / SaSb / bb$

$\Rightarrow$ $S \to aS' / b$
$S' \to SS'' / bb$
$S'' \to SbS / aSb$
(Deterministic)

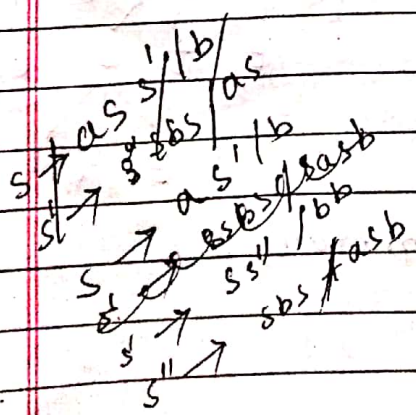
Non-Deterministic

$S \to bSSaaS$
$/ bSSaSb$
$/ bSb$
$/ a$

$\Rightarrow$

$S \to bSS' / a$
$S' \to SaaS / SaSb / b$

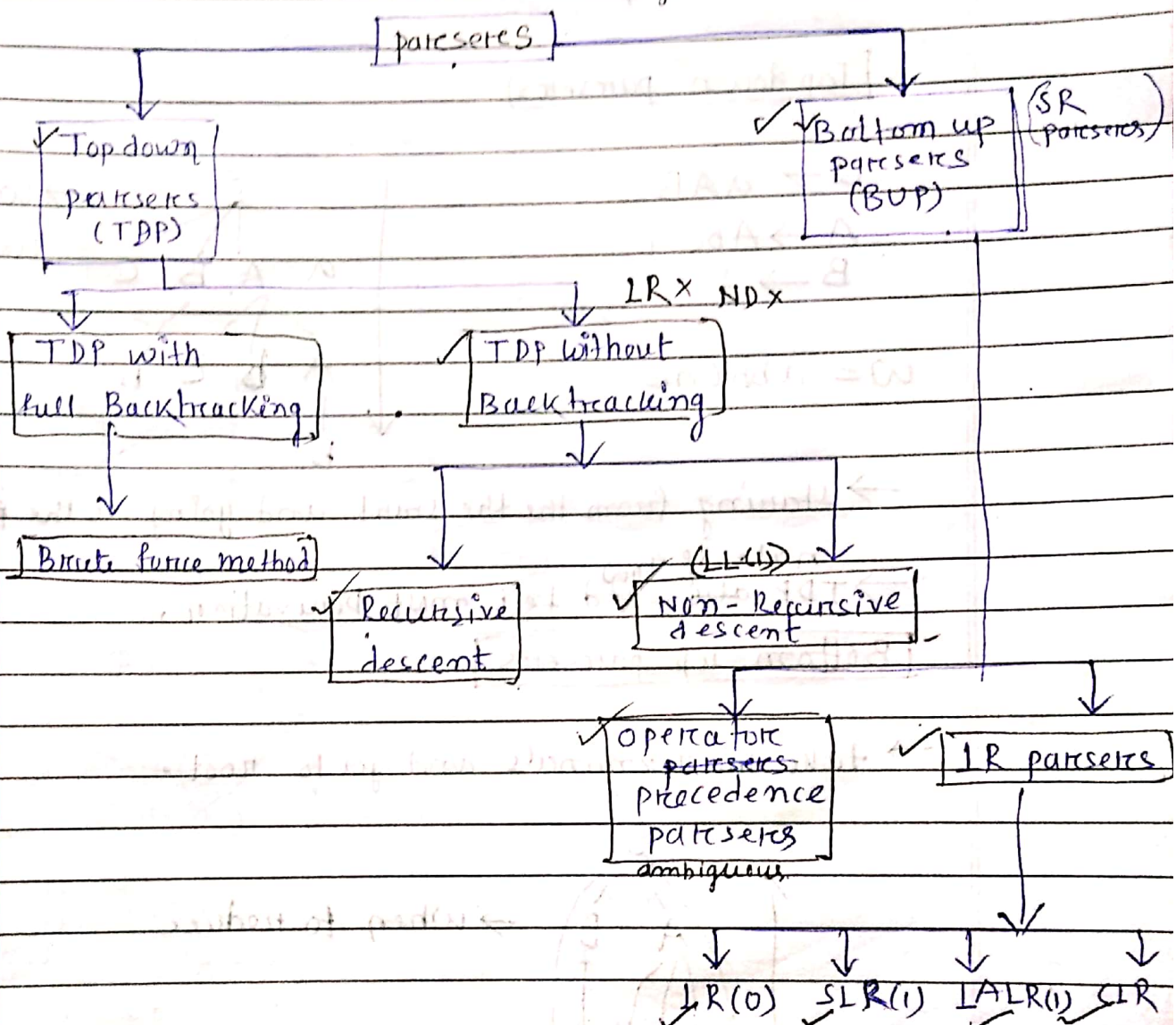
$\Rightarrow$ $S \to bSS' / a$
$S' \to SaS'' / b$
$S'' \to aS / Sb$
(Deterministic)



PARSERS

(www.gatenotes.in)

- Introduction of parsers : (gatenotes.in)



→ parsers is syntax analyser.

→ SR parsers means shift-reduce parsers.

→ TDP without backtracking not accept - left recursive grammars and also Non-Deterministic grammars.

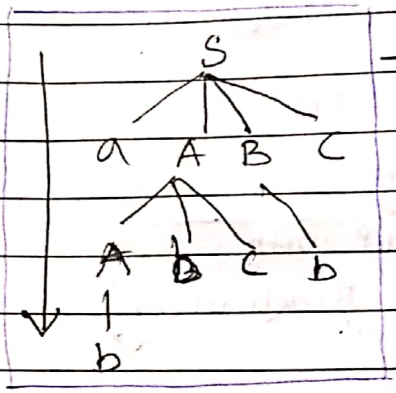
→ parsers not accept ambiguous grammars.
only operator precedence parsers accept ambiguous grammars.

- Basic difference between Top-down parsers and Bottom up parsers:

Top down parsers

$S \rightarrow aABe$
$A \rightarrow Abc/b$
$B \rightarrow d$

$w = abbcde$



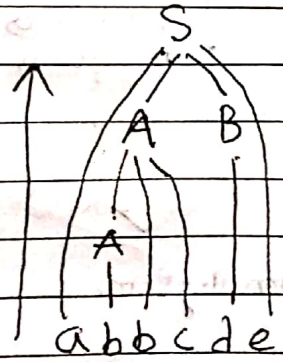
→ what to use.

→ Staring from the the start and going to the ~~bot~~ terminal. follow

→ TDP ~~also used~~ Left most Derivation.

Bottom up parsers

→ take the terminals and go to root.



→ When to reduce

$w = abbcde$.

$S \Rightarrow a\,A\,B\,e$
$\Rightarrow a\,A\,d\,e$
$\Rightarrow a\,A\,b\,c\,d\,e$
$\Rightarrow a\,b\,b\,c\,d\,e$

→ BUp follow Right most Derivation.

• LL(1) parsers :

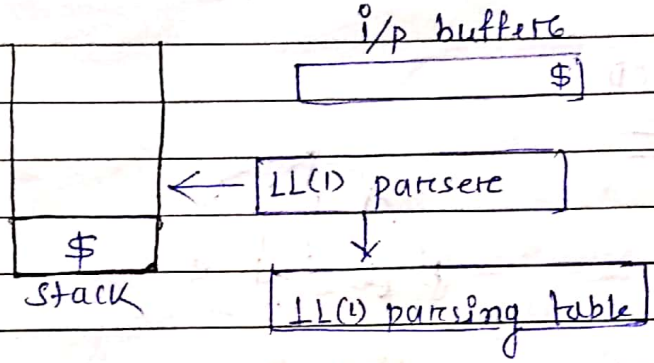
LL(1) → No. of Lookerhead.

Scaning Left to Right

left most derivation

abcd →

• Components of parsers =

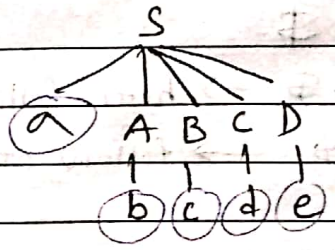


→ '$' is to guess when stop.

• First()
• Follow()

First()

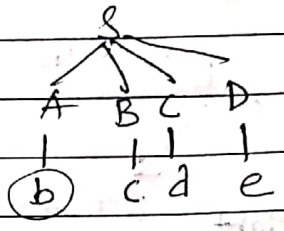
Ex- S → aABCD
A → b
B → c
C → d
D → e



here, first of S is a.
" " A " b
" " B " c
" " C " d
first of D " e

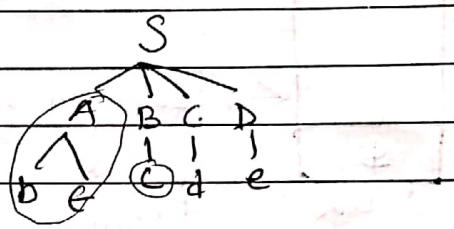
Ex-

$S \rightarrow ABCD$
$A \rightarrow b$
$B \rightarrow c$
$C \rightarrow d$
$D \rightarrow e$



→ here first of S is 'b'.

Ex- $S \rightarrow ABCD$
$A \rightarrow b/\epsilon$
$B \rightarrow c$
$C \rightarrow d$
$D \rightarrow e$



→ here ~~in~~ this case first of S is 'c'.

Follow:

What a terminal ~~that could~~ which can follow a variable in the process of derivation.

$S\$$
$abcd\$$
$ABCD\$$ → here follow of D is '$'
$ABdD\$$ → follow of B is 'd'.

$S \rightarrow ABCD$
$A \rightarrow b/\epsilon$
$B \rightarrow c$
$C \rightarrow d$
$D \rightarrow e$

→ follow of D is follow of 'S'.

→ here in this case follow of S - {$}
follow of A is first of BCD = 'c'.
So, follow of A is {c}.

follow of B is {d}.
" " C is {e}.
follow of D is {$}.

$A \rightarrow BC$

- Example– find first and follow in LL(1):

Ex-1

	first	follow
6 $S \rightarrow \overset{\epsilon}{A}\overset{\epsilon}{B}CDE$	{a, b, c}	{$}
1 $A \rightarrow a/\epsilon$	{a, ε}	{b, c}
2 $B \rightarrow b/\epsilon$	{b, ε}	{c}
3 $C \rightarrow c$	{c}	{d, e, $}
4 $D \rightarrow d/\epsilon$	{d, ε}	{e, $}
5 $E \rightarrow e/\epsilon$	{e, ε}	{$}

→ follow of A is first of BCDF = {b, c}.

Ex-2

	first	follow
3 $S \rightarrow Bb/Cd$	{a, b, c, d}	{$}
1 $B \rightarrow aB/\epsilon$	{a, ε}	{b}
2 $C \rightarrow cC/\epsilon$	{c, ε}	{d}

Ex-3

	first	follow
5 $E \rightarrow TE'$	{id, (}	{$,)}
4 $E' \rightarrow +TE'/\epsilon$	{+, ε}	{$,)}
3 $T \rightarrow FT'$	{id, (}	{+, $,)}
2 $T' \rightarrow *FT'/\epsilon$	{*, ε}	{+, $,)}
1 $F \rightarrow id/(E)$	{id, (}	{*, +, $,)}.

Ex-4:

		first	follow
4	S → ABC / CbB / Ba	{d, g, h, ϵ, b, a}	{$}
3	A → da / BC	{d, g, h, ϵ}	{h, g, $}
2	B → g / ϵ	{g, ϵ}	{$, a, h, g}
1	C → h / ϵ	{h, ϵ}	{b, h, g, $}

Ex-5:

	first	follow
S → aABb	{a}	{$}
A → c / ϵ	{c, ϵ}	{d, b,}
B → d / ϵ	{d, ϵ}	{b}

Ex-6

	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}

A → CAB

here follow of 'A' is = follow of 'A'.

A → ABCD

here follow of B is first of 'C' and 'D'.

Construction of LL(1) parsing table :

Example – 1

	first	follow
$E \to TE'$	{id, (}	{$,)}
$E' \to +TE' / \epsilon$	{+, ε}	{$,)}
$T \to FT'$	{id, (}	{+, $,)}
$T' \to *FT' / \epsilon$	{*, ε}	{+, $,)}
$F \to id / (E)$	{id, (}	{*, +, $,)}

→ LL(1) parser is a top down parser.

→ The main purpus of top down parser is when have two alternative for a variable, find out which production should choose.

Table

↓ variables	terminals → 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	$F \to id$			$F \to (F)$		

→ All the 'ε' production have to place under follow of 'left hand side'.

→ Where should place the production is depends on the ~~first~~ of first of right hand side.

→ To construct parsers tree, this table used.

• Use of parsing to construct a parsers tree –

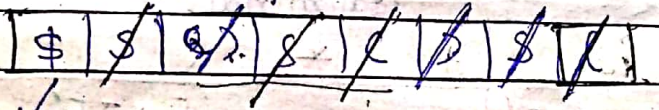
Example – 2 (( ))

	first	follow
S → (S)/ϵ	{(, ϵ}	{$,)}

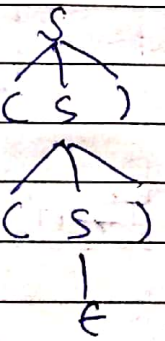
table :

	(	)	$
S	S → (S)	S → ϵ	S → ϵ

generate : (( )) $



bottom of the stack



→ LL(1) parsers algo start with '$' in the bottom of the stack and '$' in the end of the input.

~~when $, $ match then we can say the~~ When we see $ & $ then we can say it is ~~successful~~ successful matching.

Example-3

	first	follow
$S \rightarrow AaAb / BbBa$	$\{\epsilon, a, b\}$	$\{\$\}$
$A \rightarrow \epsilon$	$\{\epsilon\}$	$\{a, b\}$
$B \rightarrow \epsilon$	$\{\epsilon\}$	$\{b, a\}$

table =

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	

→ In every cell of table we got '1' production, so that this grammar is LL(1).

→ Left recursion and Non-Deterministic grammar can not be used for LL(1).

$A \rightarrow \alpha_1 / \alpha_2 / \alpha_3 / \alpha_4 / \ldots \alpha_n$.

A grammar has to be LL(1) when first of $\alpha_1, \alpha_2, \alpha_3, \ldots \alpha_n$ are mutualy exclusive (means no common)

Test that given Grammar is LL(1) or not:

① $S \to aSbS / bSaS / \epsilon$

	first	follow
S	{a, b, ϵ}	{b, a, $}

	a	b	$
S	S → aSbS S → ϵ	S → bSaS S → ϵ	S → ϵ

→ one cell contain more than one production, therefore this grammar is not LL(1).

② $S \to aABb$ (V)
$A \to c / \epsilon$
$B \to d / \epsilon$

	first	follow
S	{a}	{$}
A	{c, ϵ}	{d, b}
B	{d, ϵ}	{b}

	a	b	c	d	$
S	S → aABb				
A		A → ϵ	A → c	A → ϵ	
B		B → ϵ		B → d	

→ each cell contain only '1' production, therefore this given grammar is LL(1).

③
$S \to A / a$
$A \to a$

	first	follow
S	{a}	{$}
A	{a}	{$}

	a	$
S	S → A, S → a	
A	A → a	

→ Cell contain more than '1' production, not LL(1).
→ also it is a ambiguous Grammar.

④ ✓

	First	Follow
$S \to aB / \epsilon$	$\{a, \epsilon\}$	$\{\$\}$
$B \to bC / \epsilon$	$\{b, \epsilon\}$	$\{\$\}$
$C \to cS / \epsilon$	$\{c, \epsilon\}$	$\{\$\}$.

table

	a	b	c	$
S	$S \to aB$			$S \to \epsilon$
B		$B \to bC$		$B \to \epsilon$
C			$C \to cS$	$C \to \epsilon$

→ each cell contain single one production. therefore LL(1).

⑤ ✓

	First	Follow
$S \to AB$	$\{a, b, \epsilon\}$	$\{\$\}$
$A \to a / \epsilon$	$\{a, \epsilon\}$	$\{b, \$\}$
$B \to b / \epsilon$	$\{b, \epsilon\}$	$\{\$\}$.

Parsere table =

	a	b	$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to a$	$A \to \epsilon$	$A \to \epsilon$
B		$B \to b$	$B \to b$

→ each cell contain single one Production, therefore this grammar is LL(1).

⑥ ✗

	First	Follow
$S \to aSA / \epsilon$	$\{a, \epsilon\}$	$\{c, \$\}$
$A \to c / \epsilon$	$\{c, \epsilon\}$	$\{c, \$\}$

	a	c	$
S	$S \to aSA$	$S \to ASA\epsilon$	$S \to \epsilon$
A		$A \to c$, $A \to \epsilon$	$A \to \epsilon$

→ one cell contained more than '1' production, so therefore it is not LL(1).

(7)

	first	follow
S → A	{a, b, c, d}	{$}
A → Bb/Cd	{a, b, c, d}	{$}
B → aB/ϵ	{a, ϵ}	{b}
C → cC/ϵ	{c, ϵ}	{d}

table

	a	b	c	d	$
S	S → A	S → A	S → A	S → A	
A	A → Bb	A → Bb	A → Cd	A → Cd	
B	B → aB	B → ϵ			
C			C → cC	C → ϵ	

→ each cell contain only '1' production so it is LL(1).

(8)

	first	follow
S → aAa/ϵ	{a, ϵ}	{$, a}
A → abS/ϵ	{a, ϵ}	{a}

	a	b	$
S	S → aAa S → ϵ			S → ϵ
A	A → abS A → ϵ			

→ not LL(1).

(9)

S → (iEtS

(9)

		first	follow
X	$S \to iEtSS' / a$	{i, a}	{$, e}
	$S' \to eS / \epsilon$	{e, ε}	{$, e}
	$E \to b$	{b}	{t}

table

	i	t	a	b	e	$
S	S→iEtSS'		S→a			
S'					S'→eS / S'→ε	S'→ε
E				E→b		

→ It not LL(1).

- Recursive descent parser =

Example

$E \to iE'$

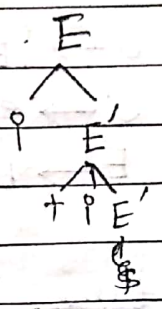
$E' \to +iE' / \epsilon$ → each variable has a function.

```
E()
{
1. if (l == i)
2.   { match('i');
3.     E'();
     }
}
```

```
E'()
{
1. if (l == +)
   {
2.   match('+');
3.   match('i');
4.   E'();
   }
5. else
6.   return;
7. }
```

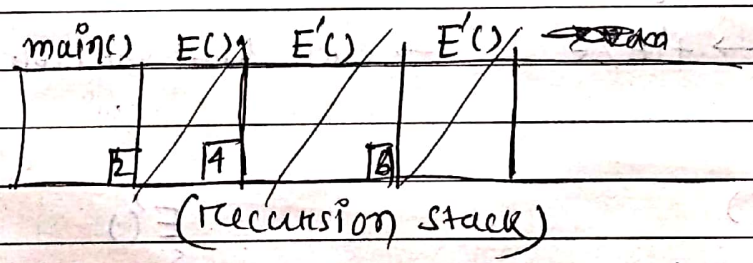
```
match (char t)
{
  if(l==t)
    l = getchar();
  else
    print("error");
}

main()
{
1. E();
2. if (l=='$')
3. print("parsing success");
}
```



l = getchar (incrementing)

according to grammar give i+i$ generated.



(recursion stack)

→ write function for every variable, then call the function using the stack provided by operating system.

- Operator precedence parsers: (Bottom up parsers)

- Operator grammar :-

✓ $E \to E+E/E*E/id$ → no two variable are adjacent.

✗ $E \to EAE/id$, $A \to +/*$ ⇒ $E \to E+E/E*E/id$ ✓

Example: (given grammar are operator precedence grammar or not)

① $S \to SAS/a$
$A \to bSb/b$

⇒ this grammar is not operator precedence grammar, because two variable are not adjacent.

now going to convert this grammar into operator precedence grammar ⇐

$S \to SbSbS/SbS/a$
$A \to bSb/b$

→ operator precendence g can parse ambiguous grammar.

- Operation relation table :
→ using this table we can do parsing ⇐

① example -

$E \rightarrow E+E / E*E / id$.

$id > * > + > \$$

$+ > +$

$* > *$

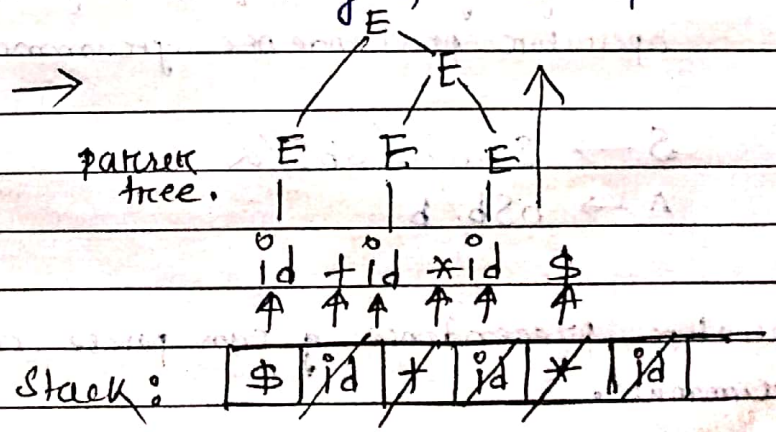
table -

	id	+	*	$
id	—	$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	—

(Operation precedence table)
(Operator relation table)

Example ① parse (id + id * id $) → i/p

using operator precendence table.

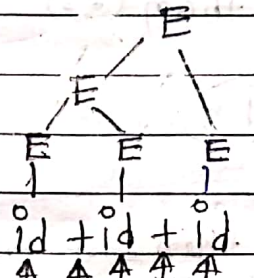


- When top of the stack less than or equal to input then push i/p in stack.
- When top of the stack are greater than to input then pope the top of the stack as terminal.

Example - 2

Construct parsen tree of id + id + id $ using operator precendence table.

→



→ left most '+' is greater than right most '+'.

→

Stack:	$	id	+	id	+	id

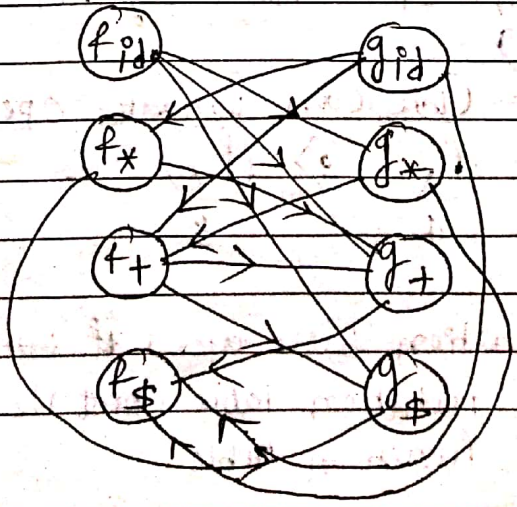
→ We will get one parsen tree whether it is ambiguous.

• Disadvantage of operation relation table =

→ if we have '4' operator then size of the table are $4^2 = 16$.

if we have 'n' operator then size of table are n^2 or $O(n^2)$.

→ To decrease the size of the table we go for operator function table =



f \ g	id	*	+	$
id	—	>	>	>
*	<	>	>	>
+	<	<	>	>
$	<	<	<	—

When greaterthan (>)

f ⟶ g

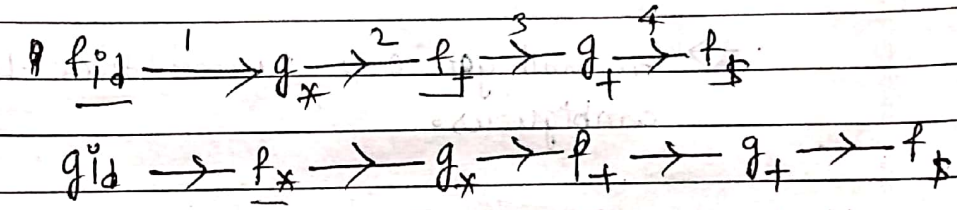
When lessthan (<)

f ⟵ g

any cycle

→ If in this graph then stop we can't make function table.

→ find the longest path from the staring node.



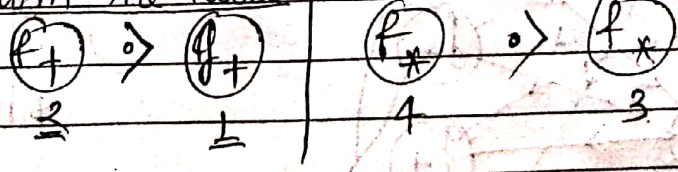
Function table

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

→ This table contain same information (like relation table) but, the ~~size of~~ the size of table less.

with 'n' input operator, size of the table are O(2n)

From the table (we can compare operators)



→ Disadvantage of function table are, if when even blank entries in relation table, there we get non-blank entries in function table.

Example —

- P → (SR)/S → not operator precedence gram. because two variable are adjecent.
- R → bSR/bS
- S → WbS/W
- W → L*W/L
- L → id .

⇒ P → (SR)/S → (I)
R → bSR/bS → (II)
S → WbS/W
W → L*W/L
L → id

P → paragraph.
S → sintence
R → recursive
b → Blank
W → word
L → letter.
id → Identifier.

Convertion into operator precendence grammar.

P → SbSR/SbS/S

P → SbP/SbS/S
S → WbS/W → 'b' is right associative.
W → L*W/L → so Right most 'b' will get higher precedence compare to Left 'b'.
L → id . → '*' also right associative.

Operator relation table —

	id	*	b	$
id	—	⋗	⋗	⋗
*	⋖	⋖	⋗	⋗
b	⋖	⋖	⋖	⋗
$	⋖	⋖	⋖	—

→ If grammar are not ambiguous then we can make table from given grammar.

(a) table =

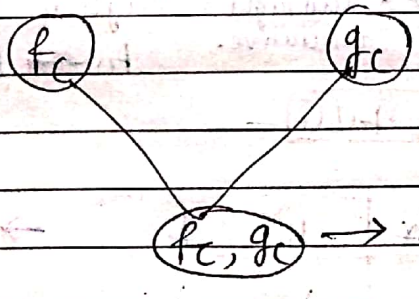
	a	(	)	,	$
a			>	>	>
(	<	<	=	<	
)			>	>	>
,	<	<	>	>	
$	<	<			

(Operator relation table)

	a	(	)	,	$
f	2	0	2	2	0
g	3	3	0	1	0

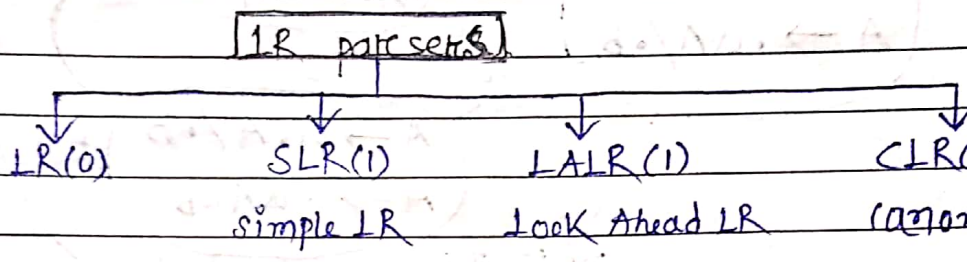
(Operator function table)

f (=) g
0 0

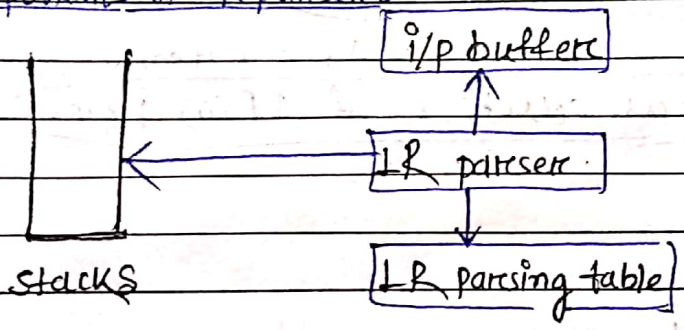


→ both w same value.

• LR parsers: (Bottom up parser)



Components of LR parsers ==



→ LR(0) items is used to construct the LR(0) and SLR(1) parsers.

→ LR(1) items is used to construct the LALR(1) and CLR(1) parsers.

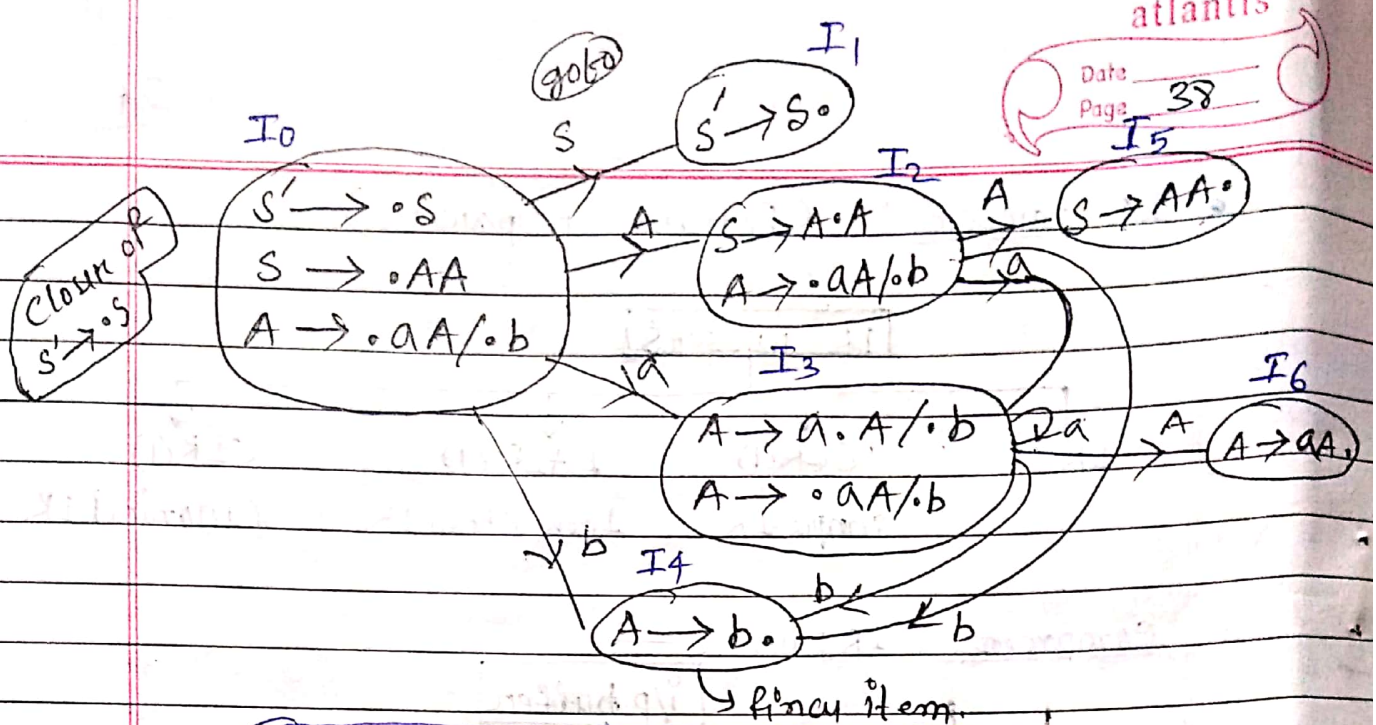
Construct LR(0) parsing table ==

$S' \rightarrow S$
$S \rightarrow AA$ ①
$A \rightarrow \alpha A / b$ ② ③

→ add one more production with given grammar.

→ closer: When a dot is left of a variable we have to add all the production of dot in the left side of

$S' \rightarrow .S$
$S \rightarrow .AA$
$A \rightarrow .\alpha A / .b$



Canonical collection of LR(0) items

Steps ⟹

(I) Take the grammar add a new production.
(II) and start with first ~~first~~ production by dot of the ~~set~~ left of S.
(III) then ~~all~~ applyed colsure.
(IV) then findout transation (goto move) and again apply closure.

→ Some states contain final items.
(any item dote at the ~~ed~~ end is called final item)

• LR(0) parsing table for canonical colection of LR(0) items ⟹

→ next page →