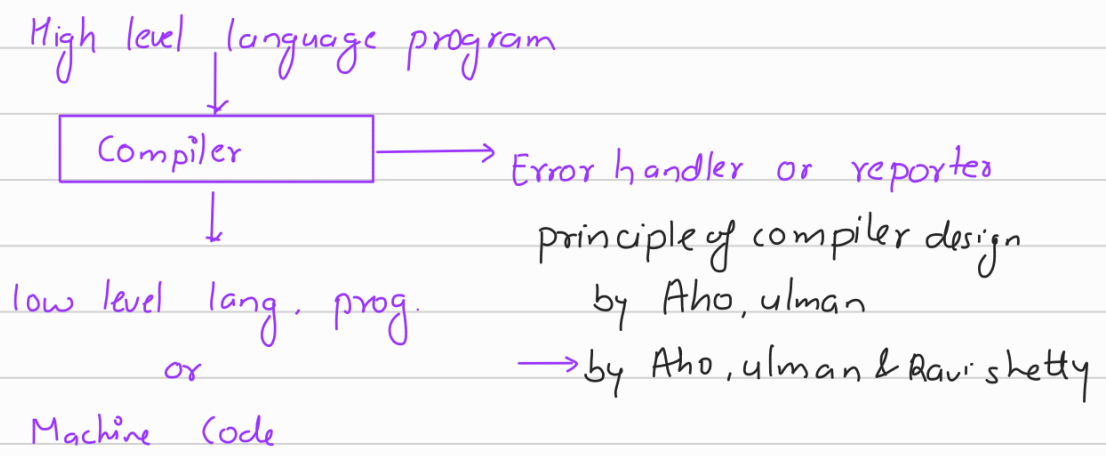
- Compiler is a system program that reads program of one language and translates it in low level prog. or machine code
- High level lang $\xrightarrow{\text{compiler}}$ low level language



principle of compiler design
by Aho, ulman
→ by Aho, ulman & Ravi shetty

- Interpreter is also sys. prog that translate & execute prog line by line

Compiler	Interpreter
- It translate HLL prog to low level or machine code	- It translates & executes prog. line by line
- It requires complete prog. at once for compilation	- No such requirement
- It reports on errors after compila-tion	- It stops execution when an error occurs during execution of any line
- It requires additional memory to store intermediate or assembly code	- It execcutes code line by line so no addition memory is required
- It takes less time to execute the program as compared to interpreter	- It takes more time to execute the prog bcz it always starts with fresh execution from the begining.

- Ex - C, C++, Java (source prog)

Ex - Python, Java (Byte code), Visual basic

In both why?? byte code

Execution of program →

Skeletal program

↓

Pre Processing.

↓

Source program

↓

Compiler

↓

Assembly program

↓

Assembler

↓

Executable code (Relocatable code or objects)

↓

Linker → Relocatable objects, needed obj linked together by OS

↓

Absolute code

↓

Loader

↓

Executed by CPU

done by OS

Phases of compiler →

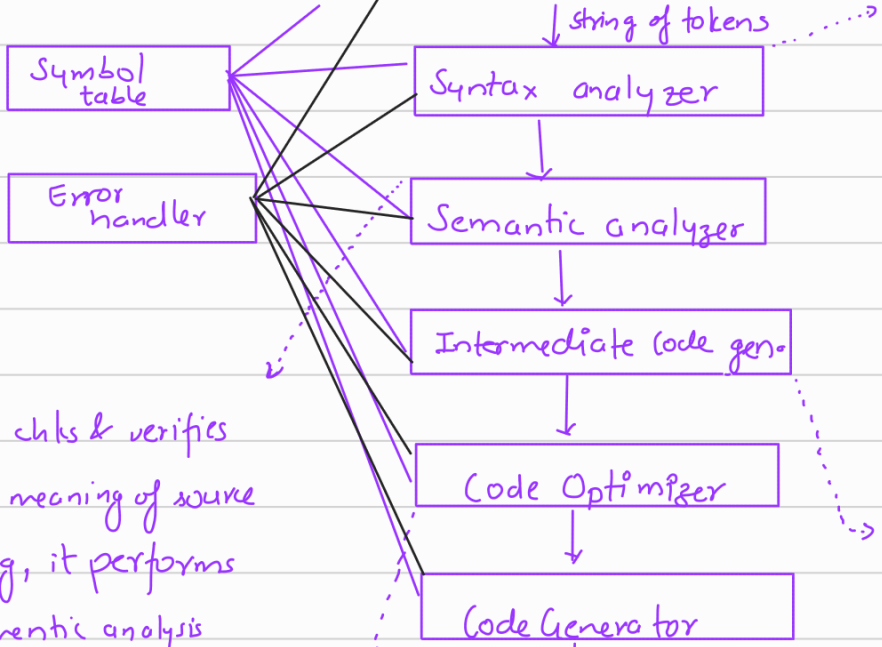
Source program

↓

Lexical Analyzer ..... → It scans the source prog. and generates string of tokens. It is also known as scanner

$z = a + b * c$

Meaning shd remain same ← $id_1 := id_2 + id_3 * id_4$;



Chk if string of token is valid or not as per source code with the help of parsing basically it verify the syntax of source, it is also called parser.
The string of tokens is gererated by grammer of source code.

It chks & verifies the meaning of source prog, it performs semantic analysis of source program
Typechking is one of module of it.

It transforms source code into intermediate code (3-address code)
Every statement of 3 address code will have at most 3 operands & 1 operator.
It is independent from source & target prog. language

It optimizes 3 add. code in order to get efficient target code

It generates target assembly code by using syntax and symentics of target programming language
It transforms every 3 add. code statement into assembly code.

$t_1 := id_3 * id_y;$

↳ id_3, id_y are in secondary memory

for CPU to execute we need at least 1 operand to main memory

```
mov  R1, id3     R1 ← id3
MUL  R1, idy     R1 ← R1 * idy
MOV  t1, R1;     t1 ← R1
```

```
id1 = t1 + id2
MOV  R1, id2
ADD  R1, t1
MOV  id1, R1
```

Syntax tree →

$t_1 := id_3 * id_y$
$t_2 := t_1 + id_2$
$id_1 := t_2$

$t_1 := id_3 * id_y$
$id_1 := id_2 + t_1$

Symbol table →

- It is a data structure that keep track of scope, binding, type, value, token and some other information of all variable names of source program.

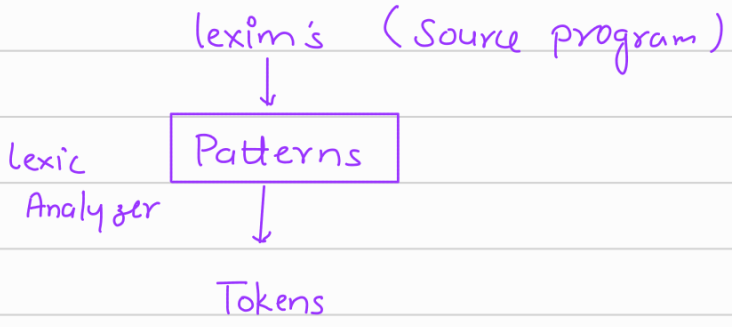
Identifiers	tokens	Scope	binding	type	Value
a	id_1	local	static	int	5
b	id_2	global	dynamic	real	5.5
c	id_3	local	dynamic	real	9.8

Error handler →

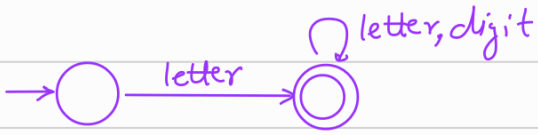
- It reports error that is detected at any phase of the compiler

Lexical Analysis →

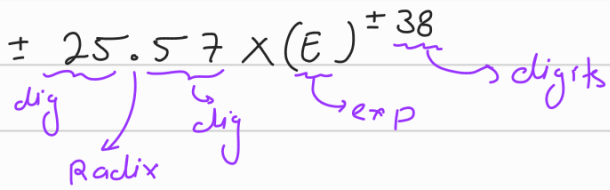
- It scans source prog. and generates strings of tokens
- lexic analyzer has set of rules called pattern which maps lexims to token



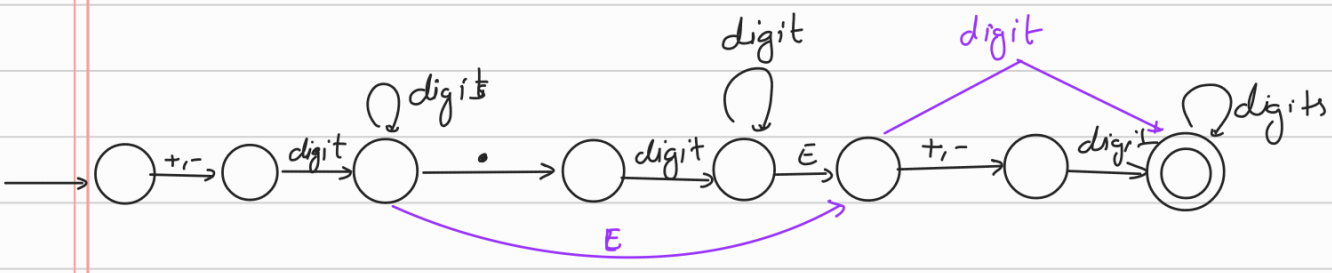
- lexim: Sequence of character with collective meaning in source program is called a lexim Ex: int
- token: Output of lexical analyzer is token
- Patterns are designed using regular expression
- Identifiers: letter (letter + digit)*



- Number : int $(digit)^+$
  real $(digit)^+.(digit)^*$



FA ??



	Leximes	Patterns	Token
1.	Identifier	letter.(letter+digit)*	id
2.	Number		
	int	(digit)+	Number
	real	(digit)+.(digit)*	Number
3.	Operators	Operators	Operators
	+	+	+
4.	Keywords	Keywords	Keywords
	int	int	int
5.	Scope	Scope	Scope

Regular expression →

Operation {+, *, ·}

property {∈ ⇒ Empty string, ∅ = Empty set}

(i) $R + \epsilon = \epsilon + R$

(ii) $R + \emptyset = \emptyset + R = R$

(iii) $R \cdot \emptyset = \emptyset \cdot R = \emptyset$

(iv) $R \cdot \epsilon = \epsilon \cdot R = R$

(v) $R \cdot R^* = R^* \cdot R = R^+$

(vi) $\epsilon + RR^* = R^*$

(vii) $(R_1 + R_2)^* = (R_1^* + R_2^*)^* = (R_1^* + R_2)^* = (R_1^* R_2^*)^*$

(viii) $(R_1^* \cdot R_2)^* \neq (R_1 \cdot R_2^*)^*$

(ix) $R_1 \cdot (R_2 + R_3) = R_1 R_2 + R_2 R_3$

(x) $R_1 R_2 + R_1 R_3 = R_1 (R_2 + R_3)$

(xi) $(R_1^* \cdot R_2)^* \cdot R_1^* = R_1^* (R_2 \cdot R_1^*)^*$

(xii) $\emptyset^* = \epsilon^*$

Lexical analyzer ⟶ Set of pattern

- It is design by finite automata
- Designed by regular expression

Finite automata →

$FA(Q, q_0, \Sigma, \delta, F)$

δ - trasition function

Deterministic FA

$Q \times \Sigma \to Q$

Total function from each state for each I/P there must be one transition

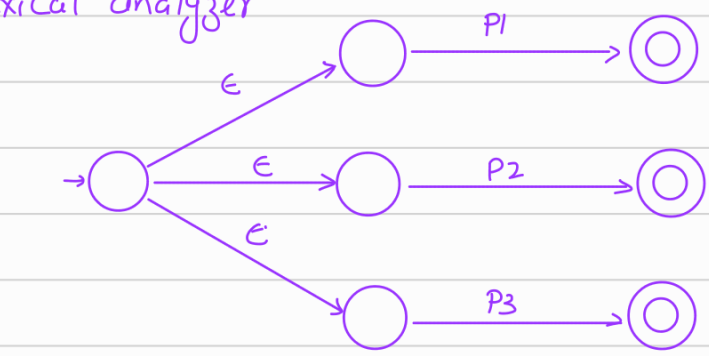
Non-deterministic

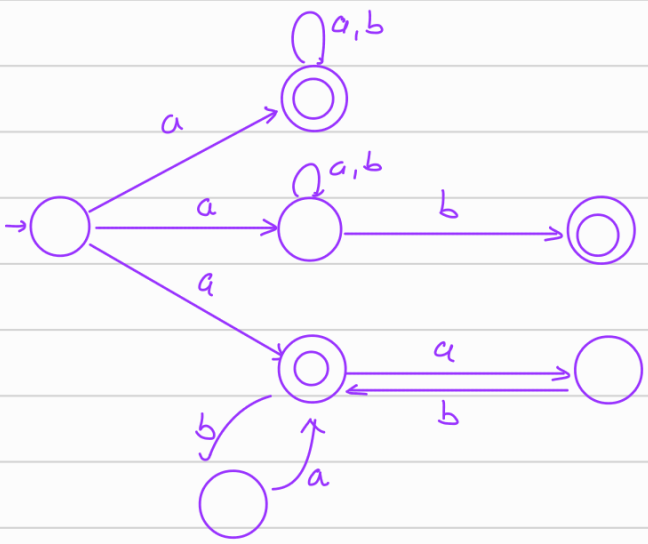
$Q \times (\Sigma \cup \epsilon) \to 2^Q$

Q. Design lexical analyzer that can recognise the following patterns

(i) $a.(a+b)^*$ (ii) $a.(a+b)^*.b$ (iii) $(ba+ab)^*$

lexical analyzer





IIP

States	a	b
→ A	B	C
B	D	E
C	F	I
D	D	G
E	B	H
F	I	C
G	D	G
H	J	G
I	I	F
J	B	H
T	T	T

ϵ-closer($\{q_0\}$) = $\{q_0, q_4\}$ = A

$\delta(A, a) = \delta(\{q_0, q_4\}, a) = \{q_1, q_2, q_8\}$

$\delta(A, b) = \{q_5\}$

$\delta(B, a) = \{q_1, q_2\}$

$\delta(B, b) = \{q_1, q_2, q_3, q_4\}$

$\delta(C, a) = \{q_4\}$

$\delta(C, b) = \epsilon$

$\delta(D, a) = \{q_1, q_2\}$

$\delta(D, b) = \{q_1, q_2, q_3\}$

$\delta(E, a) = \{q_1, q_2, q_8\}$

$\delta(E, b) = \{q_1, q_2, q_3, q_5\}$

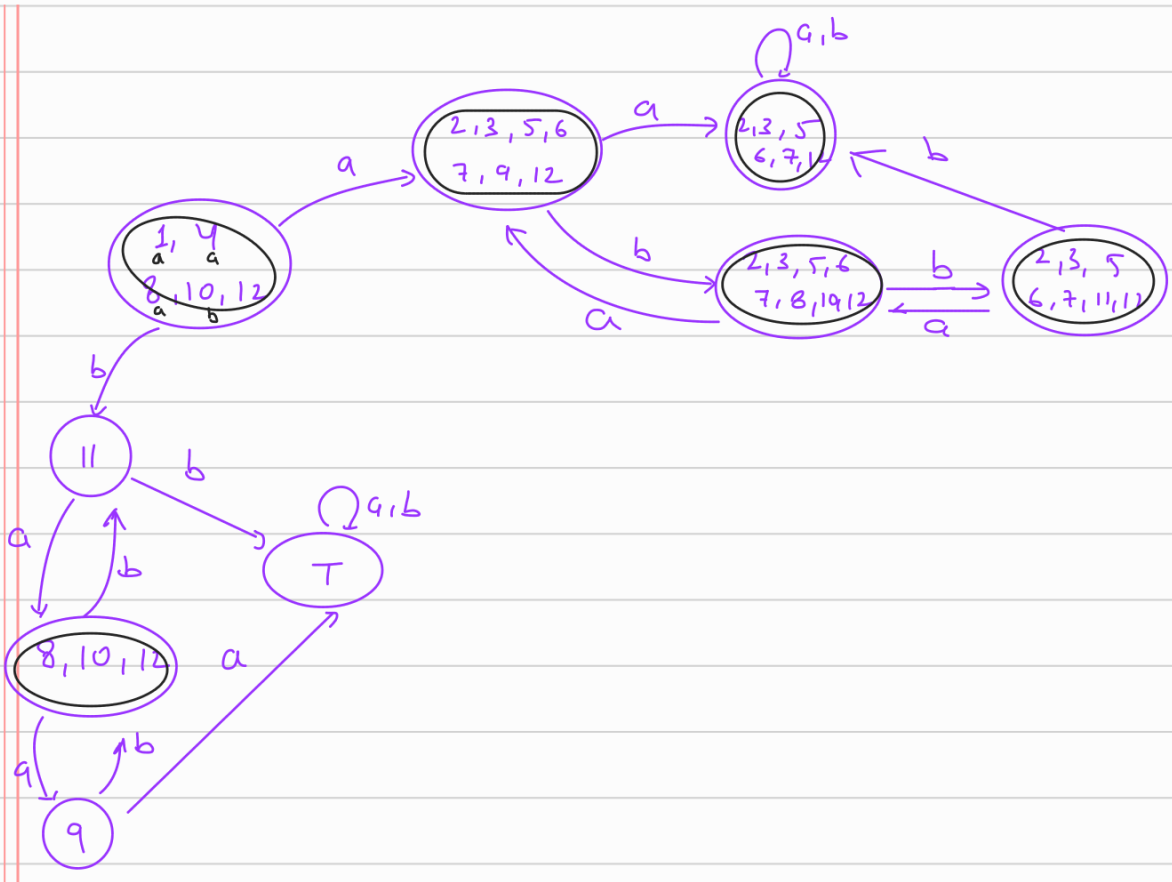
⋮

Using follow pos →

$(a \cdot (a+b)^* + a(a+b)^* b + (ab+ba)^*)\#$

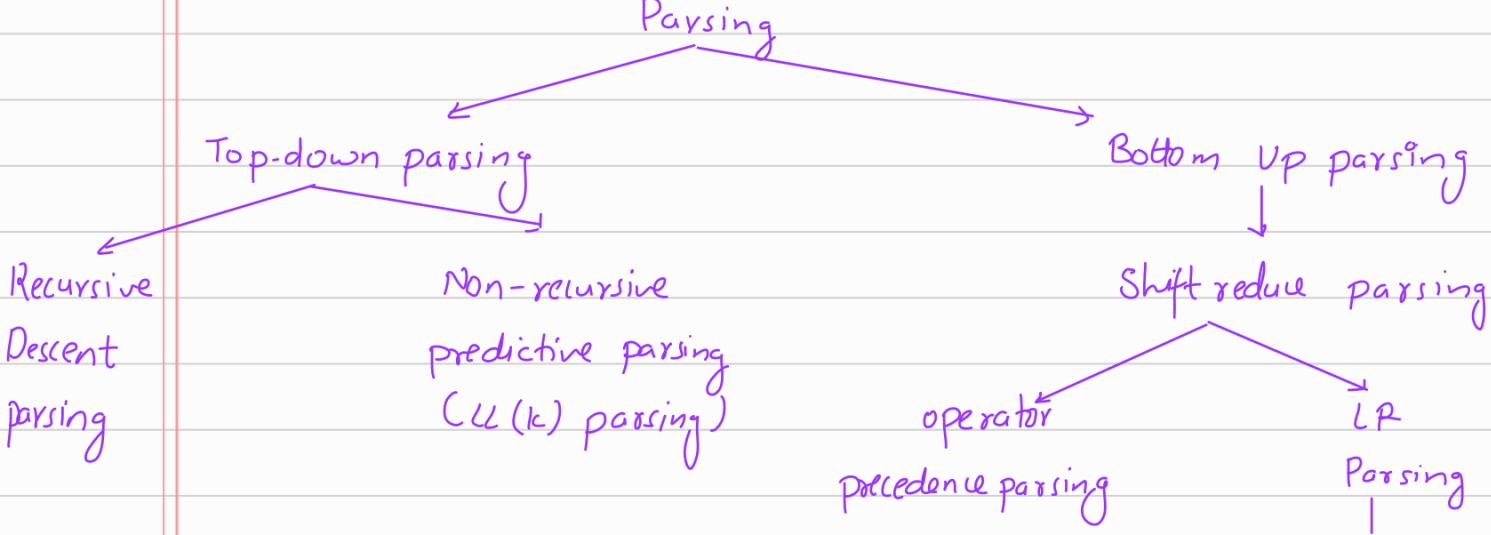
1 2 3 4 5 6 7 8 9 10 11 12

Pos	Follow pos	
1	2, 3, 12	initial state
2	2, 3, 12	1, 4, 8, 10, 12
3	2, 3, 12	
4	5, 6, 7	
5	5, 6, 7	
6	5, 6, 7	
7	12	
8	9	
9	8, 10, 12	
10	11	
11	8, 10, 12	
12	-	



Syntax analysis →

- It checks syntax of source program
- It is also known as parsing
- Parsing is to check wheather given string of tokens generated by gramm-r of source programming language
- Source Program
  ↓
  Lex Analyzer
  ↓ string of tokens
  Syntax Analyzer — parser



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

Grammer of language →

$G(V, T, S, P)$

V: set of finite variables
T: set of finite terminals
S: starting variable
P: set of finite production

$A \to z \quad \in V$
$\alpha, \beta \cdots \in (V \cup T)^*$
$a, b, c \cdots \in T$

- Regular language $\forall P$

Left linear	Right linear
$A \to Ba$	$A \to aB$
$A \to a$	$A \to a$

where $A, B \in V$, $a \in T^*$

- Context free grammer $\forall P : A \to \alpha$

$A \in V$, $\alpha \in (V \cup T)^*$

Ex - $S \to aSbS / bSaS / \epsilon$ for $n_a(\omega) = n_b(\omega)$, $\omega \in \{a, b\}^*$

Derivation (Parsing) →

- Set of step to generate set of string from the grammer is called derivation

Derivation

Left Most Derivation
- left most variable is replaced by its production in each step of derivation

Right Most Derivation
- rightmost variable is replaced by its production in each step of derivation

Left sentential form →

- Each step to generate left most derivation is called left sentential form similar for right sentential form.

Ex $S \rightarrow aSbS / bSaS / \epsilon$ "ababba"

LMD →

Step	Sentential form	Production
1.	S	$S \rightarrow aSbS$
2.	aSbS	$S \rightarrow \epsilon$
3.	abS	$S \rightarrow aSbS$
4.	abaSbS	$S \rightarrow \epsilon$
5.	ababS	$S \rightarrow bSaS$
6.	ababbSaS	$S \rightarrow \epsilon$
7.	ababbaS	$S \rightarrow \epsilon$
8.	ababba	

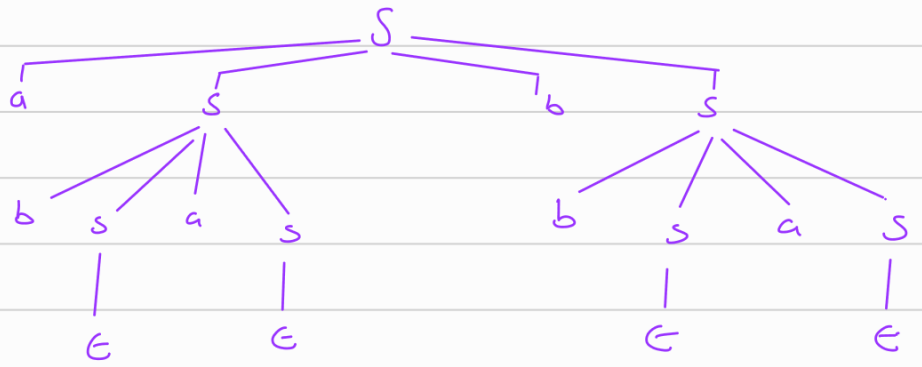
RMD →

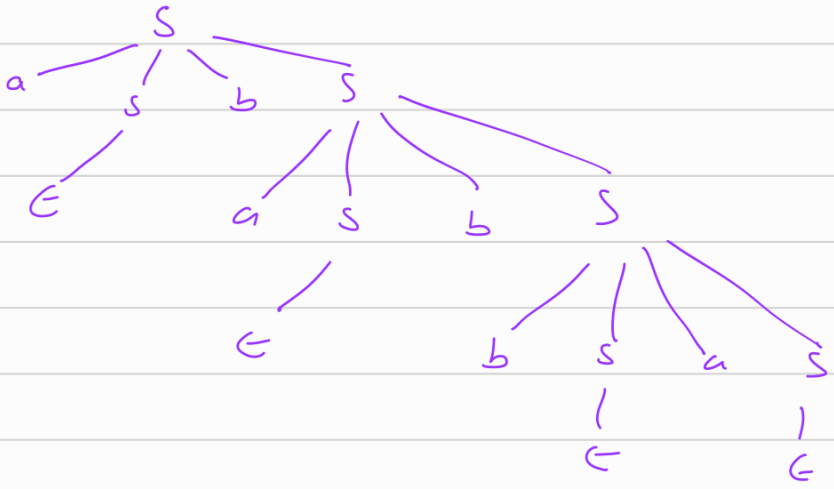
Step	Sentential form	Production
1.	S	$S \rightarrow aSbS$
2.	aSbS	$S \rightarrow aSbS$
3.	aSbaSbS	$S \rightarrow bSaS$
4.	aSbaSbbSaS	$S \rightarrow \epsilon, S \rightarrow \epsilon, S \rightarrow \epsilon, S \rightarrow \epsilon$
5.	ababba	

Parse tree → (Derivation tree)

- It is a graphical way to generate derivation in the form of a tree
- The root of the tree is S (starting variable)
- All the non leaf nodes are represented by variables while all leaf nodes are represented by terminal

Ex $S \rightarrow aSbS / bSaS / \epsilon$ "ababba"





∵ More than 1 LMD, RMD or parsetree ∴ Ambiguous grammer

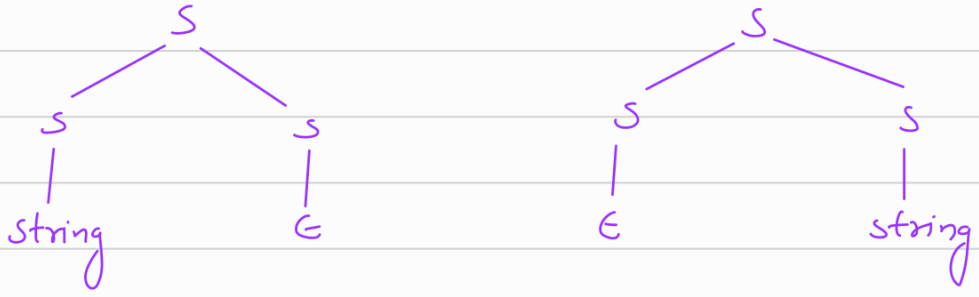
Ambiguous grammer→

- When ∃ more than one LMD, RMD, parse tree for any string of the grammer then the grammer is called ambiguous grammer.

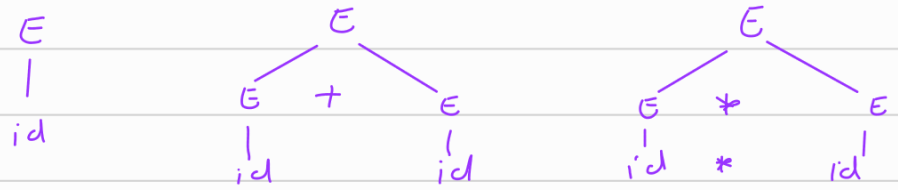
Inherently ambiguous grammer→

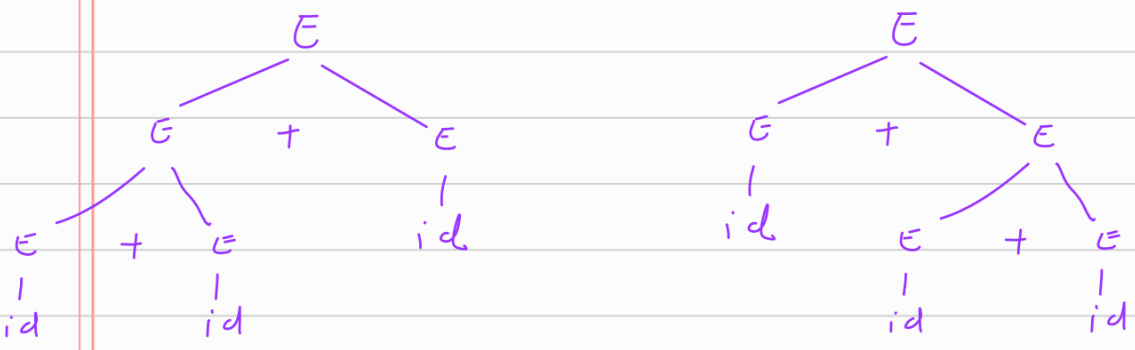
- When ∃ more than one LMD, RMD, parse tree for every string of the grammer then the grammer is called ambiguous grammer.

Ex- $S \rightarrow aSbS / bSaS / \epsilon / SS$



Ex: $E \longrightarrow E+E/E*E/id$ identify it as ambiguous or inherently ambiguous?





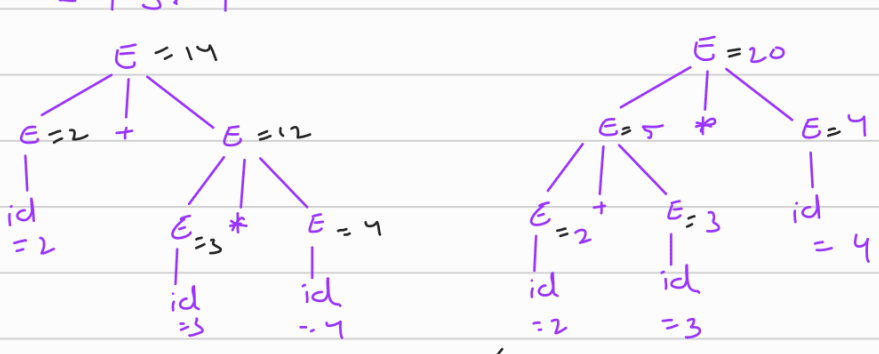
∴ Ambiguous grammer

$E \rightarrow E+E/E-E/E*E/E/E/E\hat{}E/(E)/id$

This is also ambiguous

- Ambiguous grammer can not be converted to unambiguous grammer
- But for some languages, we can have both ambiguous & unambiguous grammer

2 + 3 * 4



More than 1 result ∴ Ambiguous

- As we move deeper operator presedence must increase as subtree will be solved first.

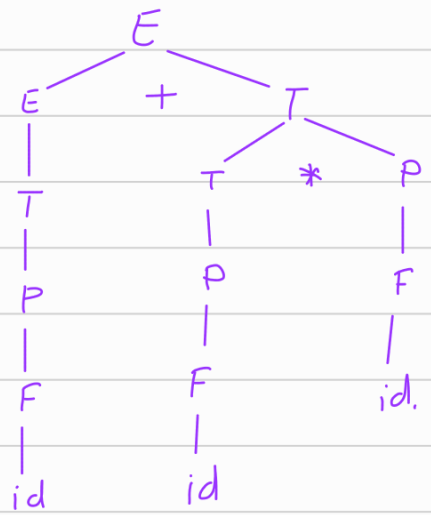
Unambiguous grammer for the Expression →

Precedence ↑ ↓

	Operator	Associativity
(1)	+, -	Left associativity
(2)	*, /	Left associativity
(3)	^	Right -"-

$E \longrightarrow E + T / E - T / T$ —
$T \longrightarrow T * P / T / P / P$ — } Alc to associativity
$P \longrightarrow F \wedge P / F$ —
$F \longrightarrow (E) / id.$ —

Checking if it is now ambiguous or not chk for id + id * id



only one ∴ not ambiguous

Ex- $E \rightarrow E * T / T$ (left ass.)
$T \rightarrow F + T / F$ (right ass)
$F \rightarrow id$

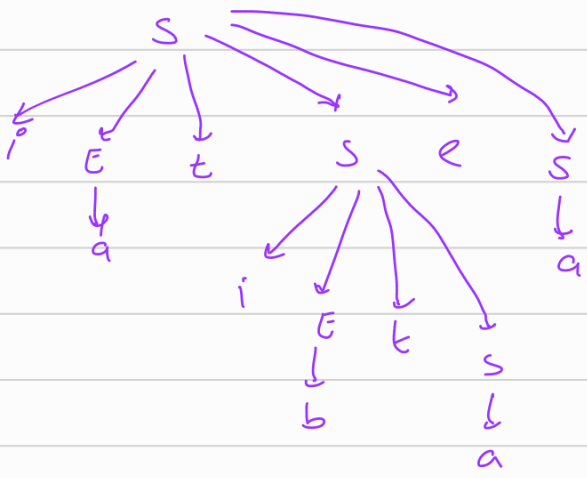
prec. + > prec *

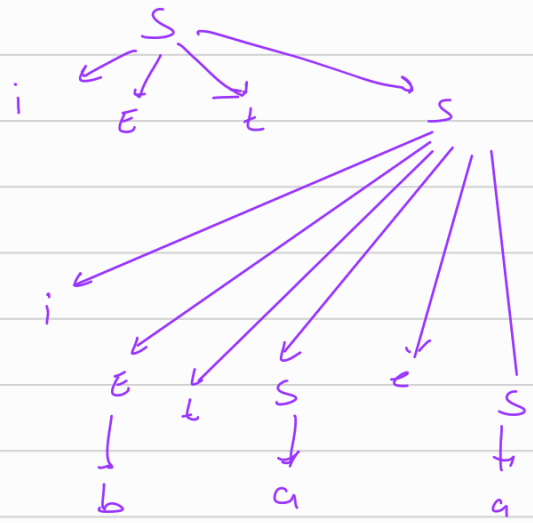
Dangling if else structure →

S → iEtS / iEtSeS / a

E → b

Ambiguous or not





∴ Ambiguous

parsing →

1. Top-down parser → (left Most Derivative)

(a) Recursive descent parser →

To design this parser we write recursive subroutine code for all variables of the grammer.

(b) Non Recursive predictive parser →

Also called LL(k) parser

- left to right scanning of I/P
- leftmost derivation
- k symbol of lookahead

Before designing parser 1st simplify grammer.

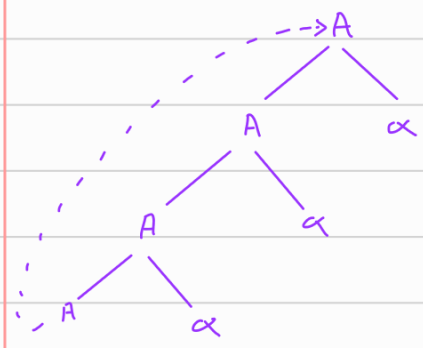
(1) Remove left recursion

(2) Left factoring

(1) Left recursion →

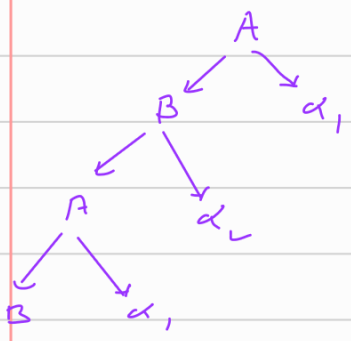
$A \rightarrow A\alpha / \beta$ (Compiler ko nahi bola ki β le to vo nahi lega)

→ direct left recursion



$A \rightarrow B\alpha_1 / \beta_1$ ⟶ indirect left recursion

$B \rightarrow A\alpha_2 / \beta_2$



Remove direct left recursion →

$A \rightarrow A\alpha / \beta \quad \Rightarrow \quad L = \{ \beta\alpha^n \mid n \geq 0 \}$

- We convert left recursion to right recursion

$A \rightarrow \beta A' \quad \Rightarrow L = \{ \beta\alpha^n \mid n \geq 0 \}$

$A' \rightarrow \alpha A' / \epsilon$

Ex. $A \rightarrow A\alpha_1 / A\alpha_2 \cdots / A\alpha_n / \beta_1 / \beta_2 \cdots \beta_m$

↓ convert to right rec.

$A \rightarrow \beta_1 A' / \beta_2 A' \cdots / \beta_m A'$

$A' \rightarrow \alpha_1 A' / \alpha_2 A' \cdots / \alpha_n A' / \epsilon$

Remove indirect left recursion →

(a) Convert indirect left recursion to direct left recursion
(b) Now remove direct left recursion.

Ex $S \to AS/a$

$A \to \boxed{S}A/b$ (LR)

↓

$A \to (AS/a)A/b$

$A \to AS.A/aA/b$

∴ new gmr

$S \to AS/a$

$A \to ASA/aA/b$ (α = SA; β_1 = aA; β_2 = b)

??

$A \to aAA'/bA'$

$A' \to SAA'/\epsilon$

Ex $E \to E+T/T$ (α = +T; β = T)

$T \to T*F/F$ (α = *F; β = F)

$F \to (E)/id$

α = recursive ; β = bino rec.

$E \to TE'$

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

$T \to FT'$

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

$F \to (E)/id$

Ex $S \to Aa/d$

$A \to Bb/a$

$B \to \boxed{S}aB/Bb/b$

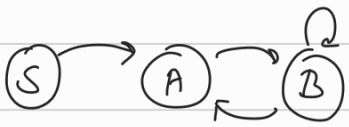
↳ recursive point

↓ convert to direct recursion

$S \to Aa/d$

$A \to Bb/a$

$B \to \boxed{A}aaB / daB / Bb / b$

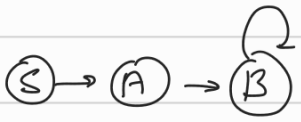


↓

$S \to Aa/d$

$A \to Bb/a$

$B \to B\underbrace{baaB}_{\alpha_1} / \underbrace{aaaB}_{\beta_1} / \underbrace{daB}_{\beta_2} / \underbrace{Bb}_{\alpha_2} / \underbrace{b}_{\beta_3}$



$B \to aaaBB' / daBB' / bB'$

$B' \to baaBB' / bB' / \epsilon$

→ We can process right recursion using parser ??
Yes, because LL(k) scan from left to right in non rec. predictive parser.

Left factoring →

- We have to find the left most part common in the set of productions of a variable.

$A \to \alpha\beta_1 / \alpha\beta_2 / \text{-------} / \alpha\beta_n$

↓

$A \to \alpha A'$

$A' \to \beta_1 / \beta_2 / \beta_3 \text{ ------} / \beta_n$

Ex $S \to iEtSeS / iEtS / a$

$E \to b$

↓ left factoring

$S \to iEtSS' / a$

$S' \to eS / \epsilon$

$E \to b$

Ex $S \to aAb / aAd / aB / b$

$A \to bA / bB / b$

$B \to bB / aA / bBd / bSa / a$

$S \to aS' / b$

~~$S' \to Ab / Ad / B$~~

$S' \to AS'' / B$

$S'' \to b / d$

$A \to bA'$

$A' \to A / B / \epsilon$

$B \to bB' / aB''$

$B' \to B / Bd / Sa / BB''' / Sa$

$B'' \to A / \epsilon$

$B''' \to d / \epsilon$

Non recursive predictive parser →

- First(X) →
- It is a set of all terminals from where 'X' can begin

Ex (i) $E \to TE'$

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

$T \to FT'$

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

$F \to (E) / id$

Variables	First	Follow
E	{ (, id }	{ $,) }
E'	{ +, ∈ }	{ $,) }
T	{ (, id }	{ +, $,) }
T'	{ *, ∈ }	{ +, $,) }
F	{ (, id }	{ *, + $,) }

First →

$E \to TE'$

$T \to FT'$

$\therefore E \to FT'E'$

↳ $(E)T'E'$

↳ $id\ T'E'$

Follow

E → E ke right dekho in right hand of production

$T \to TE'$ all prod. where E' can begin

↳ $T + TE'$

$TE = T$ (follow of E)

E' → ke peeche kuch nahi ∴ parent ka lo

- Include 'ϵ' in first but not in follow

Follow (x) →

- It is a set of all terminals that follow variable "X".
- Add "$" in starting of starting variable
- "ϵ" can't be included in the follow of any variable

Ex- S → ABC
A → aA/ϵ
B → bB/Ad/ϵ
C → AB/SC/d
→ A = ϵ in S
→ A = ϵ, B = ϵ in C

Variables	First	Follow
S	{a, b, d, ϵ}	{$, a, b, d}
A	{a, ϵ}	{$, a, b, d}
B	{b, a, d, ϵ}	{$, a, b, d}
C	{a, b d, ϵ}	{$, a, b, d}

for Follow S C
∴ first of C

for follow A
A d, A B ∴ first of B, if B = ϵ A C
∴ first of C

follow of C _
∴ parent as nothing

Ex S → aA/BC
A → bAd/ϵ
B → aS/Cb/ϵ
C → Ae/bB/Sd/ϵ

Variable	first	follow
S	{a, b, e, d, ϵ}	{$, d, a, b, e (follow B)}
A	{ϵ, b}	{d, e, $, a, b (follow S)}
B	{a, ϵ, b, d, e}	{a, b, e, d, $}
C	{a, b, e, d, ϵ}	{a, b, e, d, $ (follow S)}

First (x) →

S1→ If X is a terminal then include it in first (x)
S2→ If X is deriving ϵ, then first (x) includes 'ϵ'
S3→ If X ⟶ $Y_1 Y_2 \cdots Y_n$ is a production then include first of Y_1 in the first of X except ϵ, if ϵ ∈ First (Y_2) then include first (Y_2) in first (x) ....

S4→ If $\forall i \in [1, N] \in \in first(Y_i)$ then $first(X) = first(Y_1) \cup first(Y_2) \cup \dots \cup first(Y_n) \cup \{\in\}$

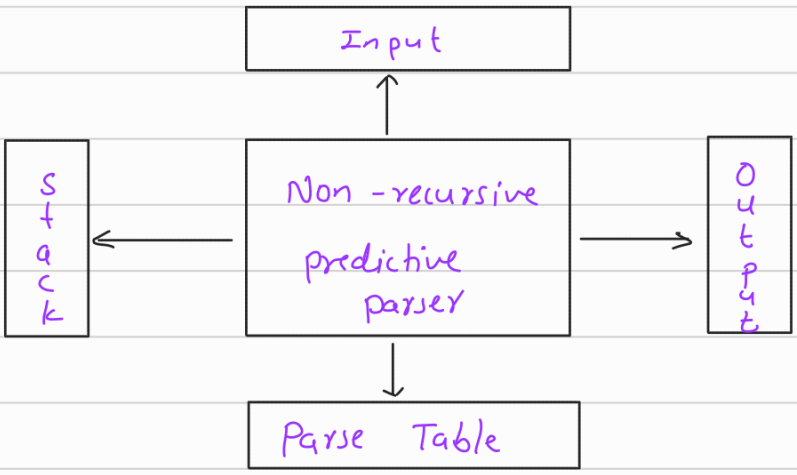
Follow(X) →

S1→ Add "$" in the follow of starting variable

S2→ If $A \to \alpha B \beta$ is a production then include first of "β" in the follow of B (except '$\in$')

S3→ If $A \to \alpha B \beta$ where '$\in$' $\in first(\beta)$ then include follow of "A" in the follow of B

LL(k) parser →



Parse table for LL(1) parser →

$E \to TE'$ $\quad T \to FT'$ $\quad F \to (E) / id$

$E' \to +TE' / \in$ $\quad T' \to *FT' / \in$

Variable ↓	Terminals					
	+	*	id	(	)	$
E	-	-	$E \to TE'$	$E \to TE'$	-	-
E'	$E' \to +TE'$	-	-	-	$E' \to \in$	$E' \to \in$
T	-	-	$T \to FT'$	$T \to FT'$	-	-
T'	$T' \to \in$	$T' \to *FT'$	-	-	$T' \to \in$	$T' \to \in$
F	-	-	$F \to id$	$F \to (E)$	-	-

- All black entry represent error
- Agar $\in$ hai to follow of parent wala me likho warna first me

Algorithm to construct parse table →

1. Create a table M where all the row headers are represented by variables and terminals represented in column
2. For each production $A \to \alpha$ do
   (i) Compute first of α and for each $a \in first(\alpha)$ do
   $M[A, a] = \{ A \to \alpha \}$
   (ii) If α derives 'ϵ' then compute follow of (A) and for each $b \in follow(A)$ do
   $M[A, b] = \{ A \to \alpha \}$
3. Return M

LL(1) parsing → M [TOS, ip]

$ → To say empty

To rep. end of I/P string

Stack	Input	Output
$ E (TOS)	id + id * id $ (ip)	E → TE'
$ E'T (TOS) → (in reverse)	id + id * id $ (ip)	T → FT'
$ E'T'F (TOS)	id + id * id $ (ip)	F → id
$ E'T' ~~id~~ (TOS)	~~id~~ + id * id $ (ip)	id matched
$ E'T'	+ id * id $	T' → ε
$ E'	+ id * id $	E' → +TE'
$ E'T +	+ id * id $	'+' matched
$ E'T	id * id $	T → FT'
$ E'T'F	id * id $	F → id
$ E'T'id	id * id $	'id' matched

$ E' T' ↑	* id $ ↑	T' → * F T'
$ E' T' F * ↑	* id $ ↑	'*' matched
$ E' T' F ↑	id $ ↑	F → id
$ E' T' id ↑	id $ ↑	'id' matched
$ E' T' ↑	$ ↑	T' → ∈
$ E' ↑	$ ↑	E' → ∈
$ ↑	$ ↑	Accepted

Ex- id id

Stack	Input	Output
$ E	id id $	E → T E'
$ E' T	id id $	T → F T'
$ E' T' F	id id $	F → id
$ E' T' id	id id $	id match
$ E' T'	id $	Error

Parsing algorithm → LL(1)

Input : Parse table M, input string str

1. Push $ & starting variable in 'S', where 'S' is a starting variable
2. Place $ at the end of str : str = str + $
3. let 'a' is top of stack & 'b' is ip ie input pointer that point to str and initially points to 1st letter of str.
4. Repeat following step until a = b = $
   (a) If a = b, pop stack & advance input pointer "ip"
   (b) else if let A → α is a production at location M[a, b]

Then pop the stack & push α in reverse order,
& Output is $A \to \alpha$

(c) else error

LL(1) Grammer → 1 Char considering at time of parsing each time (top 1 in stack)

When each location of parse table is having at most 1 entry / production then such grammer is called LL(1) grammer

Eg:
$E \longrightarrow TE'$
$E' \longrightarrow +TE'/\epsilon$
$T \longrightarrow FT'$
$T' \longrightarrow *FT'/\epsilon$
$F \longrightarrow (E)/id$

} This is LL(1) (Lk table above

Eg:
$S \longrightarrow iEtSS'/a$
$S' \longrightarrow eS/\epsilon$
$E \longrightarrow b$

Find given grammer is LL(1) or not!

Variables	First	Follow
S	{i, a}	{$, e}
S'	{e, ∈}	{$, e}
E	{b}	{t}

Parse table →

Variable	Terminal					
	i	t	a	e	b	$
S	$S \to iEtSS'$		$S \to a$			
S'				$S' \to \epsilon$ $S' \to eS$		$S' \to \epsilon$
E					$E \to b$	

∵ 2 present in same location ∴ Not LL(1) grammer

LL(1) Test →

(i) When each variable of grammer has a single production then such grammer is always LL(1) grammer

(ii) when $A \to \alpha / \beta$

(i) $First(\alpha) \cap First(\beta) = \phi$

(ii) If $\beta \xrightarrow{*} \epsilon$ then

$First(\alpha) \cap Follow(A) = \phi$

If both (i) & (ii) are true for the grammar then grammer is LL(1) grammer

(iii) $S \to AaRb / BbAa$

$A \to \epsilon$

$B \to \epsilon$

$\because F(\alpha) = a$, $F(\beta) = b$

$\because F(\alpha) \cap F(\beta) = \phi \therefore LL(1)$

(iv) $S \to AaBb / BaAb$

$A \to \epsilon$

$B \to \epsilon$

$\because F(\alpha) = a$, $F(\beta) = a$

$\because F(\alpha) \cap F(\beta) = a \neq \phi$

$\therefore$ Not LL(1)

(v) $S \to AaAb / BbBa$

$A \to \epsilon$

$B \to \epsilon$

LL(1) ✓

(vi) $S \to Ab / BC$

$A \to aA / \epsilon$

$B \to bB / \epsilon$

$C \to dSb / \epsilon$

$F(\alpha) = a, b$

$F(\beta) = b, d$

$F(\alpha) \cap F(\beta) = b \neq \phi \therefore$ Not LL(1)

(vii) $S \to Ab / BC$

$A \to aA / \epsilon$

$B \to bB / \epsilon$

$C \to dSb / \epsilon$

LL(1) ✗

$B \xrightarrow{*} \epsilon$

$Follow(S) = b$

$First(\alpha) = b$

Cond^n 2 voilate

(viii) If a grammer is ambiguous it can never be LL(k) grammer

∵ $S \to iEt\,ss'/a$
$S' \to eS/\epsilon$
$\epsilon \to b$
} is Ambiguous ∴ not LL(k)

(ix) If grammer is unambiguous then it may be LL(k) grammer

Note: If grammer is LL(1) it will be LL(k) but vice aversa need not to be always true.

Eg $S \to aA/bB$
$A \to b$
$B \to d$

LL(2) → Combination of all 2 terminal in parse table

is Not LL(1) but LL(2) ∵ One prod. placed at ab & other at bd

Bottom Up parser → (Also called Shift reduce parser)

- Operator precedence parser
  - Direct
    - * For ambiguous grammer
    - * Using precedence & associativity of operator.
  - Indirect
    - * For unambiguous gramming
    - * Using leading & trailing
- LR parser

Operator precedence parsing →

- We require operator precedence grammer for operator precedence parsing.

- **Operator grammer -**
- When right hand part of every production is not having two consecutive variables then such grammer is called operator grammer

		voilate operator grammer
Ex-	$A \rightarrow aAbB$	✗
	$A \rightarrow aABb$	✓
	$A \rightarrow Aa\,bB$	✗
	$A \rightarrow \epsilon$	✗
	$A \rightarrow AaB$	✗
	$A \rightarrow BC$	✓

Operator precedence grammer →

- ϵ-free operator grammer is called operator precedence grammer

Operator grammer = Operator precedence grammer + ϵ

- Why ϵ free?

Operator precedence parsing →

- **Direct methode -**
- It is used for the ambiguous grammer, we use associativity & precedence of operator.
- There are 3 type of precedence we have to use $\cdot>$, $<\cdot$, $\doteq$

$T1 <\cdot T2 \longrightarrow$ Terminal T1 has lower precedence then T2

$T1 \cdot> T2 \longrightarrow$ Terminal T2 has more precedence then T1

$T1 \doteq T2 \longrightarrow$ Both T1 & T2 have same precedence.

Expression grammer →

- Example of expression grammer

$E \longrightarrow E+E / E*E / (E) / id$

$E \longrightarrow E+E / E-E / E*E / E/E / E\hat{}E / (E) / id$

Parse (Precedence) table →

$E \longrightarrow E+E / E-E / E*E / E/E / E\hat{}E / (E) / id$

T2 (R)

T1 (L)	id	+	−	*	/	^	(	)	$
id	Error	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	Error Missing operator	$\dot{>}$	$\dot{>}$
+	$\dot{<}$	$\dot{>}$ L to R	$\dot{>}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{>}$	$\dot{>}$
−	$\dot{<}$	$\dot{>}$	$\dot{>}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{>}$	$\dot{>}$
*	$\dot{<}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{<}$	$\dot{<}$	$\dot{>}$	$\dot{>}$
/	$\dot{<}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{<}$	$\dot{<}$	$\dot{>}$	$\dot{>}$
^	$\dot{<}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{<}$	$\dot{<}$	$\dot{>}$	$\dot{>}$
(	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\doteq$	Error
)	Error	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	Error	$\dot{>}$	$\dot{>}$
$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	Error	ε-free NO-case

Precedence	Associativity
+, −	L → R
*, /	L → R
^	R → L

Precedence ↑

Ex $\$ id + id * id \$$ id → need to be identified first

$\$ \dot{<} id \dot{>} + \dot{<} id \dot{>} * \dot{<} id \dot{>} \$$

Shd be valid

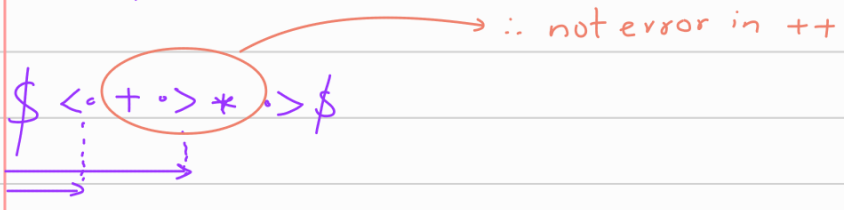
$E \to id$

$\$ E + \dot{<} id \dot{>} * \dot{<} id \dot{>} \$$

$ E + E · * <· id ·> $

$ E + E * E $ All replaced by variable then we remove all variable from expression to identify precedence

$ + + $



Operator precedence parsing →

- Input string 'str' and precedence table or parse table

S1 → Add $ at the begin and ending of string 'str'

Eg: $ id + id * id $

S2 → Fill the precedence b/w every pair of consequtive terminal in the string String $ str $

Eg. $ <· id ·> + <· id ·> * <· id ·> $

S3 → Start from leftmost '$' and move toward the rightmost '$'.
- find first greater precedence while moving in right dir^n & then move left dir^n & find first less precedence "<·"
- the substring b/w^n "<· ·>" should be a valid handle
- If so replace it with left hand variable in production
  i.e for A → handle , handle replaced by A

Handle → it is a valid substring present at right side of production set.

→ Production on right hand part of production.

$E \longrightarrow E+E / E*E / (E) / id$

Handle

S4 → Repeat step 3 until we have string or expression with variables and operators

S5 → Remove all the variable from the resulting expression
Now we have to find precedence b/w operator to parse the expression

S6 → Now again fill the precedence b/w consequtive pair of terminals of expression of step 5

$\$ <\cdot + <\cdot * \cdot> \$$

S7 → Start from left most $ & move toward the right most $ and find ·> then move left to find <·

$\$ <\cdot + <\cdot * \cdot> \$$

S8 → The operator b/w <· & ·> will be parsed first for that substitute all the variable in the expression in same order

$\$ E+E*E \$$

- Now check $E*E$ is valid handle or not if yes then replace it with left hand variable else error

$\because$ $E \longrightarrow E*E$ $\therefore$ replace

exp: $\$ E+E \$$

Sq → Repeat step 5 to 8 until we get $\$S\$$ where S is starting variable, It indicates end of parsing.

$\$+\$$

$\$ <\cdot + \cdot> \$$

$\$ E+E \$$

$\$ E \$ \longrightarrow$ end of parsing.

(b) Indirect methode →

- It is used for unambiguous grammer

(i) leading →

$$\text{leading}(A) = \{ a \mid A \xrightarrow{*} \alpha a \beta \text{ where } \alpha \text{ is a single variable or '}\epsilon\text{'} \}.$$

or

$$\text{leading}(A) = \{ a \mid A \longrightarrow a\beta \text{ or } A \longrightarrow B a \beta \text{ or } A \longrightarrow B \text{ and } \forall\, a \in \text{leading}(B) \}$$

(ii) Trailing →

$$\text{Trailing}(A) = \{ a \mid A \xrightarrow{*} \alpha a \beta \text{ where } \beta \text{ is a single variable or '}\epsilon\text{'} \}$$

or

$$\text{Trailing}(A) = \{ a \mid A \longrightarrow \beta a \text{ or } A \longrightarrow \beta a B \text{ or } A \longrightarrow B \text{ and } \forall\, a \in \text{trailing}(B) \}$$

Ex→ $E \longrightarrow \underset{\alpha\ a\ \beta}{E+T} / \underset{\alpha\ a\ \beta}{E-T} / T$

$T \longrightarrow \underset{\alpha\ a\ \beta}{T*P} / \underset{\alpha\ a\ \beta}{T/P} / P$

$P \longrightarrow F \wedge P / F$

$F \longrightarrow \underset{\alpha\ a\ \beta}{(E)} / \underset{\alpha\ a\ \beta}{id}$

Scan from L→R in production & place 1st variable in leading for trailing just move from R→L

Variables	Leading	Trailing
E	{+, −, *, /, ^, (, id}	{+, −, *, /, ^,), id}
T	{*, /, ^, (, id}	{*, /, ^,), id}
P	{^, (, id}	{^,), id}
F	{(, id}	{), id}

Parse table →

	id	+	−	*	/	^	(	)	$
id		⋗	⋗	⋗	⋗	⋗		⋗	⋗
+	⋖	⋗	⋗	⋖	⋖	⋖	⋖	⋗	⋗
−	⋖	⋗	⋗	⋖	⋖	⋖	⋖	⋗	⋗
*	⋖	⋗	⋗	⋗	⋗	⋖	⋖	⋗	⋗
/	⋖	⋗	⋗	⋗	⋗	⋖	⋖	⋗	⋗
^	⋖	⋗	⋗	⋗	⋗	⋖	⋖	⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	
)		⋗	⋗	⋗	⋗	⋗		⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖		

- Move production by product

$E \rightarrow E + T$

+ is trail of E

E +
L R

∴ Select + from Right of table right = trail

∴ we need to fill trailing of E in parse table → Always fill greater precedence

for leading remember L L L

leading <· left to right

$E \to E \underset{L}{+} \underset{R}{T}$ leading(T) place less then

- Repeat same for each production

* $A \to \alpha a B b \beta$

wher B may be 'ε' then a = b

∴ $F \to (E)$

Algorithm to fill precedence table →

Input: Grammer

S1→ Compute leading & trailing of all variables

S2→ For each production $A \to \delta$ do

(a) if $A \to \alpha a B \beta$ then

∀ b ∈ leading (B) do

fill $a <\cdot b$ in precedence table

(b) if $A \to \alpha B a \beta$ then

∀ b ∈ trailing (B) do

fill $b \cdot> a$ in precedence table

(c) if $A \to \alpha a B b \beta$ Where B is a single variable or ε

fill $a \doteq b$ in precedence table

S3→ Return precedence table

Precedence Graph →

- Input: precedence table

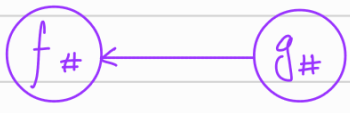
1. Assign function name $f_\#$ to all the terminals '#' present at row index of precedence table & assign $g_\#$ function name to all the terminals '#' present at coloumn index.

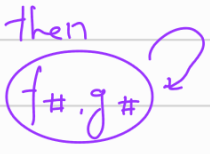
2. For each terminal $f_\#$ & $g_\#$ create a seprate node and create single node for all $f_{\#_1}$ & $g_{\#_2}$ which have $\#_1 \doteq \#_2$ in precedence table

Eg. f_{id}, g_{id}, f_+, g_+, & so on for them we create single seprate node while f_c & $g_)$ will be in a single node

3. Create edges b/w^n nodes of $f_\#$ & $g_\#$ as follows

(a) if $f_\# \gtrdot g_\#$ then
create a edge from 

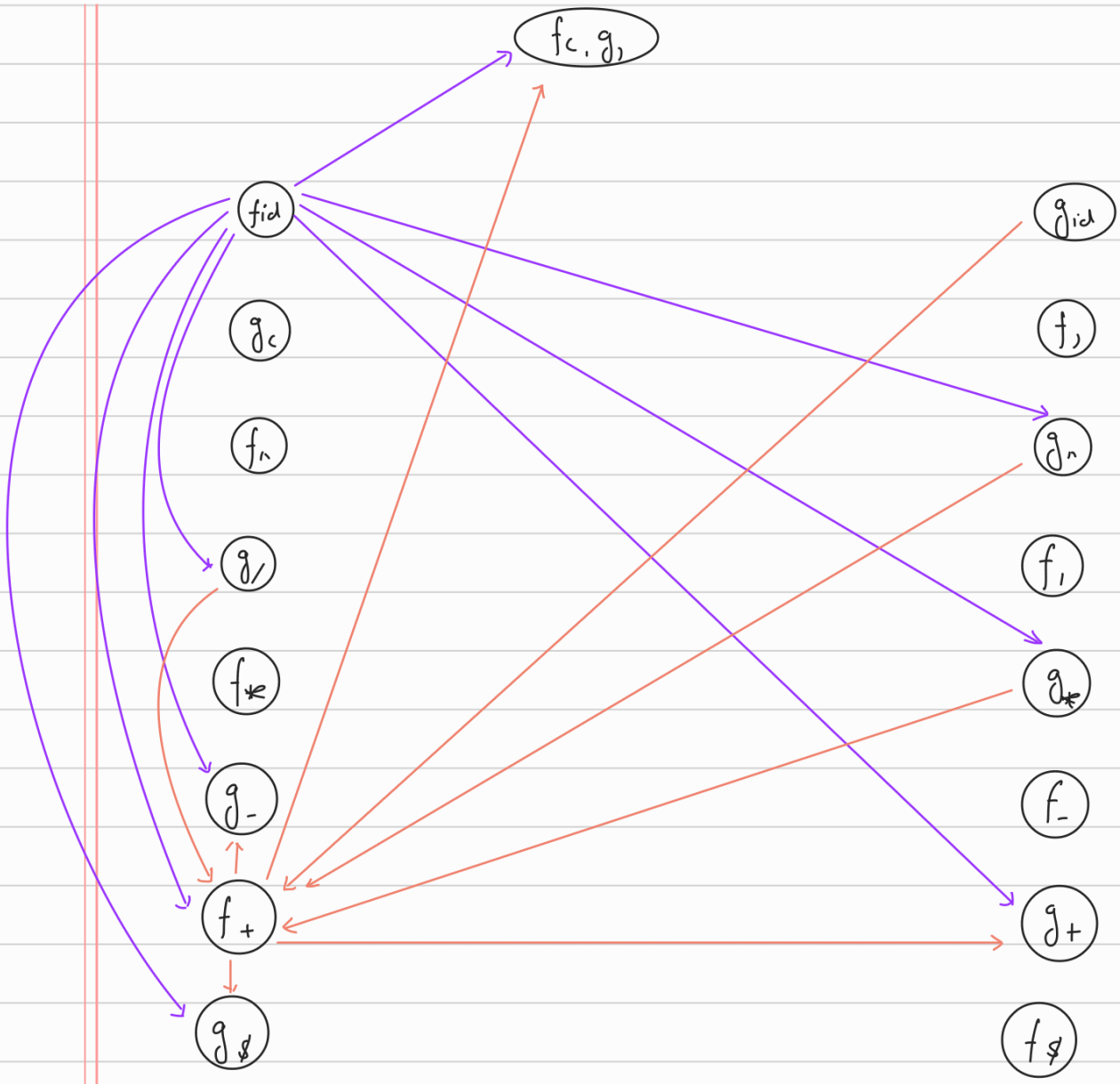
(b) if $f_\# \lessdot g_\#$ then
create a edge from 

(c) if $f_\# \doteq g_\#$ then
self loop 

4. Return precedence graph

f \ g	id	+	−	*	/	^	(	)	$
id		⋗	⋗	⋗	⋗	⋗		⋗	⋗
+	⋖	⋗	⋗	⋖	⋖	⋖	⋖	⋗	⋗

To construct graph we move from higher precedence to lower precedence & assign the node in alternate way. ⇒ f g g f



Ex $E \to E+T/T$

$T \to T*F/F$

$F \to (E)/id$

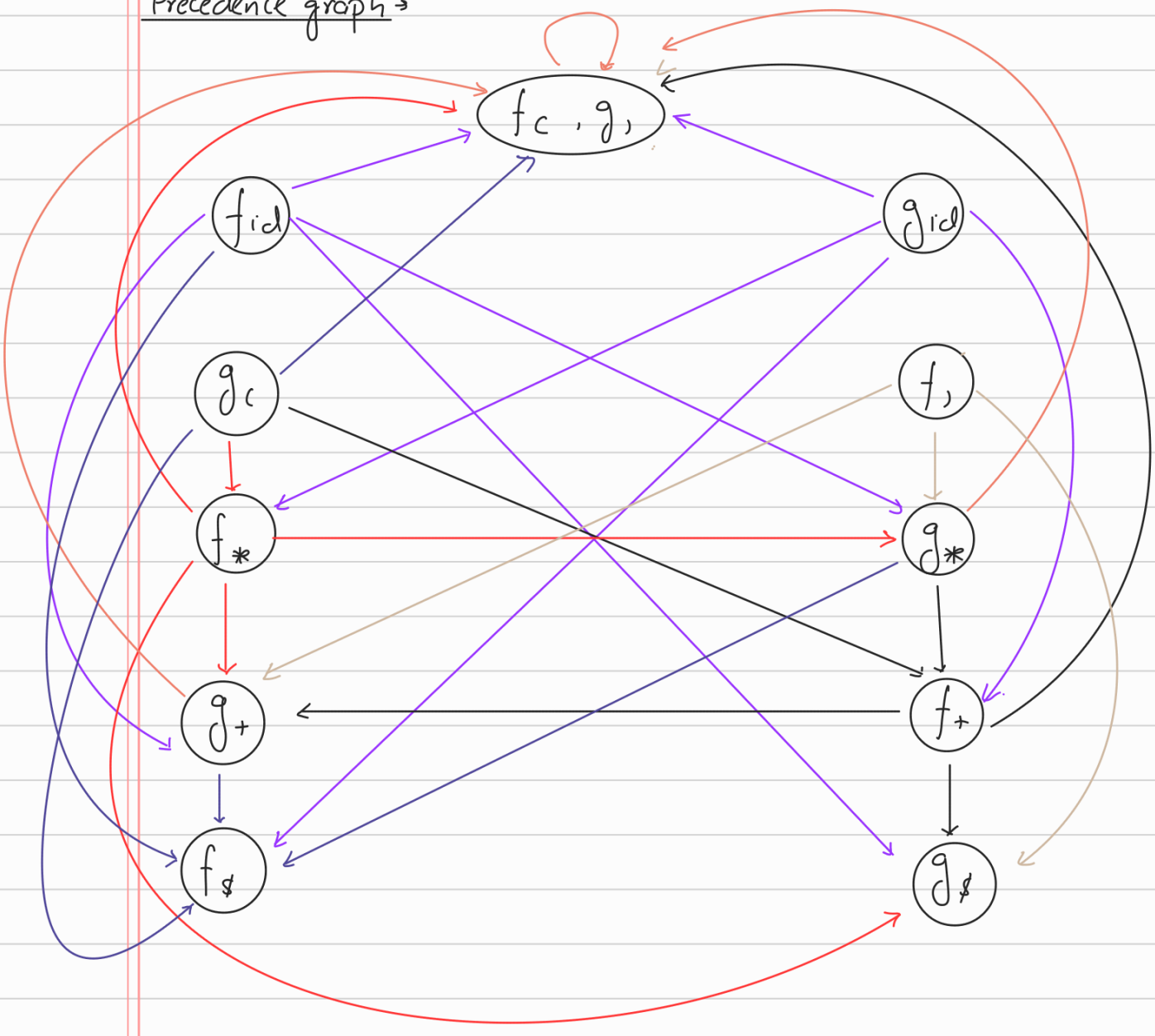
Variable	Leading	Trailing
E	+, *, (, id	*, +,), id
T	*, (, id	*,), id
F	(, id	), id

Prec. table →

	g_{id}	g_+	g_*	$g_($	$g_)$	$g_\$$
f_{id}		$\cdot>$	$\cdot>$		$\cdot>$	$\cdot>$
f_+	$<\cdot$	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
f_*	$<\cdot$	$\cdot>$	$\cdot>$	$<\cdot$	$\cdot>$	$\cdot>$
$f_($	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\doteq$	
$f_)$		$\cdot>$	$\cdot>$		$\cdot>$	$\cdot>$
$f_\$$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$		

← E

Precedence graph →



Precedence function →

- longest path from node 'n' represents precedence function of n.

Function	id	(	)	*	+	$
f	4	0	4	4	2	0 → for sinks
g	5	5	0	3	1	0

longest path length chk from upper levels par node ke barabar se

- If ∄ no cycle in graph that mean the grammer is operator precedence grammer.

$T_1 \vee T_2$ $T_1 = T_2$

Ex → $S \rightarrow (L) / a$

$L \rightarrow L, S / S$

Variable	leading	Trailing
S	{ (a }	{) a }
L	{ , (a }	{ ,) a }

	g_a	$g_($	$g_)$	$g_,$	$g_\$$
f_a			$\cdot>$	$\cdot>$	$\cdot>$
$f_($	$<\cdot$	$<\cdot$	$\doteq$	$<\cdot$	
$f_)$			$\cdot>$	$\cdot>$	$\cdot>$
$f_,$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	
$f_\$$	$<\cdot$	$<\cdot$			

$\$ S \$$

- Operator precedence parsing algorithm →

Input: Precedence table and input string

S1→ Push $ on the stack and add $ at the end of the string.

S2→ a := TOS and b := value of current input pointer that points I/p string, initially 1st letter of string

S3→ Repeat following steps until a = b = $ do

Shift Action (a) If a <· b or a ≐ b then push b on the stack ∴ a := b & b := advance i/p pointer value.

Reduce Act. (b) If a ·> b then repeat pop from stack until current TOS is having ·> than recently popped symbol

(c) else error

Ex $ id + id * id $

Stack	Input	Output
$	id + id * id	Shift id

For understanding

$ <· id	+ id + id * id	Reduce
$	+ id * id $	Shift +
$ <· +	id * id $	Shift id
$ <· + <· id	* id $	Reduce
$ <· +	* id $	Shift *
$ <· + <· *	id $	Shift id
$ <· + <· * <· id	$	Reduce
$ <· + <· *	$	Reduce
$ <· +	$	Reduce
$	$	Accepted

LR Parser →

- LR(0) Parser
- SLR(1) Simple LR parser
- LALR(1) Lookahead LR parser
- CLR(1) Canonical LR parser

LR(0) Item set

LR(1) Item set

LR(K) Parsing →

- Left to right scanning of I/P
- Rightmost derivation in reverse order
- K - Symbol of look-ahead

LR(0) Itemset → Not stop when we reach starting variable

⇓

Soln ⟶ Augumented production

Augumented production

$S' \rightarrow S$ Stops when reach S' & we can reach S' by only 1 times

Augumented Grammer

$S \rightarrow AA$
$A \rightarrow aA/b$

$\Rightarrow$

$S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA/b$

follow $(S') = \emptyset$ ⇓ stop

LR(0) Itemset → (Place '.' before right hand part of production)

{'.' is very imp}

First Item

$I_0 : S' \rightarrow .S$

$I_n : \cdots$
$A \rightarrow \alpha . B\beta$
$B \rightarrow .\delta_1$
$B \rightarrow .\delta_2$
$\vdots$
$B \rightarrow .\delta_n$

⇒ . before variable then generate all production all of that variable we have to add in that state

I_0: $S' \rightarrow .S$ (Goto I_1)
$S \rightarrow .AA$ (Goto I_2)
$A \rightarrow .aA$ (Goto I_3)
$A \rightarrow .b$ (Goto I_4)

New state → $I_1 : S' \rightarrow S.$ No scope for growing ∴ close state

I_2: $S \rightarrow A.A$ (Goto I_5) It can grow further
$A \rightarrow .aA$ (Goto I_3)
$A \rightarrow .b$ (Goto I_4)

I_3: $A \rightarrow a.A$ (Goto I_6)
$A \rightarrow .aA$ (Goto I_3)
$A \rightarrow .b$ (Goto I_4)

$I_n : \cdots$
$A_1 \rightarrow \alpha_1 . B\beta_1$
$A_2 \rightarrow \alpha_2 . B\beta_2$
$\vdots$
$A_n \rightarrow \alpha_n . B\beta_n$

. B
jitni bhi .B wale honge vo chale jaengi
$A_n \rightarrow \alpha_i B . \beta_i$

$I_4 : \cdots$
$A_1 \rightarrow \alpha_1 B . \beta_1$
$A_2 \rightarrow \alpha_2 B . \beta_2$
& move '.' over variable
(one by one production we check)

Always start from begining

I_4 : $A \longrightarrow b.$

I_5 : $S \longrightarrow AA.$

I_6 : $A \longrightarrow a.A$

$A \longrightarrow .aA$

$A \longrightarrow .b$

Same as I_3 $\Rightarrow$

$I6$:

$A \longrightarrow aA.$

LR(0) parse table →

(i) Compute LR(0) itemset

Ex Grammer

$S \rightarrow AA$

$A \rightarrow aA/b$

Augumental prod →

0. $S' \rightarrow S$
1. $S \rightarrow AA$
2. $A \rightarrow aA$
3. $A \rightarrow b$

Augumented production

States	Action a	Action b	Action $	Go to S	Go to A
I_0	S_3	S_4		1	2
I_1	Acc.	Acc.	Acc.		
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		

- There are two types of action in LR parser

(i) Shift action →

When '.' is shifted over a terminal in any state then we have to place shift (S_n) entry for that terminal where n is a state where '.' is shifted.

(ii) Reduce action →

When '.' is reached at end of the production in any state then we place reduce (R_n) entry where 'n' is a production number which is going to reduce

When '.' is at the end of agumented production then we have to place accepted entry

For LR(0) parser we have to place reduce R_n entries corresponding to all the terminals.

- At most 1 entry in action part of LR(0) parse table ∴ grammer is LR(0) grammer

SLR(1) parse table →

(i) Compute LR(0) item set

(ii) The shift and goto entries of SLR(1) parse table is similar to LR(0) parse table but reduce entries differ

For reduce entries of SLR(1) parse table.
let $A \to \alpha.$ is a production in any state then we have to place R_n entries corresponding to follow(A) only where n is a no. of production

A → Follow

α. → Follow

States	Action a	Action b	Action $	Go to S	Go to A
I_0	S_3	S_4		1	2
I_1			Acc.		
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		

Ex- Grammer →

$E' \to E$ $\quad\quad$ $T \to F$

$E \to E+T$ $\quad\quad$ $F \to (E)$

$E \to T$ $\quad\quad$ $F \to id$

$T \to T*F$

LR(0) parse table →

States	Action +	*	(	)	id	$	Goto E	T	F
I_0			S_4		S_5		1	2	3
I_1	S_6 / Acc.	Acc.	Acc.	Acc.	Acc.	Acc.			
I_2	R_2	S_7 / R_2	R_2	R_2	R_2	R_2			
I_3	R_4	R_4	R_4	R_4	R_4	R_4			
I_4			S_4		S_5		8	2	3
I_5	R_6	R_6	R_6	R_6	R_6	R_6			
I_6			S_4		S_5			9	3
I_7			S_4		S_5				10
I_8	S_6			S_{11}					
I_9	R_1	R_1	S_7 / R_1	R_1	R_1	R_1			
I_{10}	R_3	R_3	R_3	R_3	R_3	R_3			
I_{11}	R_5	R_5	R_5	R_5	R_5	R_5			

- There are two type of conflict in LR parser
  (i) Shift reduce (SR) conflict
  (ii) Reduce Reduce (RR) conflict

SLR(1) Parse table →

States	Action +	*	(	)	id	$	Goto E	T	F
I_0			S_4		S_5		1	2	3
I_1	S_6					Acc.			
I_2	R_2	S_7		R_2		R_2			
I_3	R_4	R_4		R_4		R_4			
I_4			S_4		S_5		8	2	3

I_5	R_6	R_6		R_6		R_6		
I_6			S_4		S_5		9	3
I_7			S_4		S_5			10
I_8	S_6			S_{11}				
I_9	R_1	S_7		R_1		R_1		
I_{10}	R_3	R_3		R_3		R_3		
I_{11}	R_5	R_5		R_5		R_5		

LR(1) item set →

Ex $S \to AA$
$A \to aA / b$

$\Rightarrow$ Aug.

(0) $S' \to S$
(1) $S \to AA$
(2) $A \to aA$
(3) $A \to b$

The computation of LR(1) itemset is similar to LR(0) itemset but in LR(1) we have to add one symbol of look ahead along with the production of itemset.

I_0 : $S' \to .S, \$$ $G(I_1)$
$S \to .AA$, first $(\epsilon.\$) = \$$ $G(I_2)$ (Parent)
$A \to .aA$, first $(A\$) = a/b$ $G(I_3)$
$A \to .b$, first $(A\$) = a/b$ $G(I_4)$

look ahead for newly generated production from parent one
S' is highest ∴ $ is only look ahead
Kharch pani of Aug. prod. is $

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

I_2 :
$S \to A.A, \$$ $G(I_5)$
$A \to .aA, \$$
$A \to .b, \$$

I_3 :
$A \to a.A, \ a/b$
$A \to .aA, \ a/b$
$A \to .b, \ a/b$

I_n : - - - -
$A \to \alpha.B\beta, a$
All prod. with .B we have to copy here in this state only
$B_1 \to .\delta_1$, first (βa)
$B_2 \to .\delta_2$, first (βa)
$B_n \to .\delta_n$, first (βa)

LALR(1)

I_4 :

$A \longrightarrow b.\ ,\ a/b$

I_5 :

$S \longrightarrow AA.\ ,\ \$$

I_6 :

$A \longrightarrow a.A\ ,\ \$$

$A \longrightarrow .aA\ ,\ \$$

$A \longrightarrow .b\ ,\ \$$

I_7 :

$A \longrightarrow b.\ ,\ \$$

I_8 :

$A \longrightarrow aA.\ ,\ a/b$

I_9 :

$A \longrightarrow aA.\ ,\ \$$

LALR(1) itemset →

I_0 : Same as LR(1)

I_1 : ——"——

I_2 : ——"——

I_{36} : $A \longrightarrow a.A\ ,\ a/b/\$$

$A \longrightarrow .aA\ ,\ a/b/\$$

$A \longrightarrow .b\ ,\ a/b/\$$

I_{47} : $A \longrightarrow b.\ ,\ a/b/\$$

I_5 : Same as LR(1)

I_{89} : $A \longrightarrow aA.\ a/b/\$$

(0) $S' \longrightarrow S$

(1) $S \longrightarrow AA$

(2) $A \longrightarrow aA$

(3) $A \longrightarrow b$

parse table same as LR(1)

reduce differs

parse table →

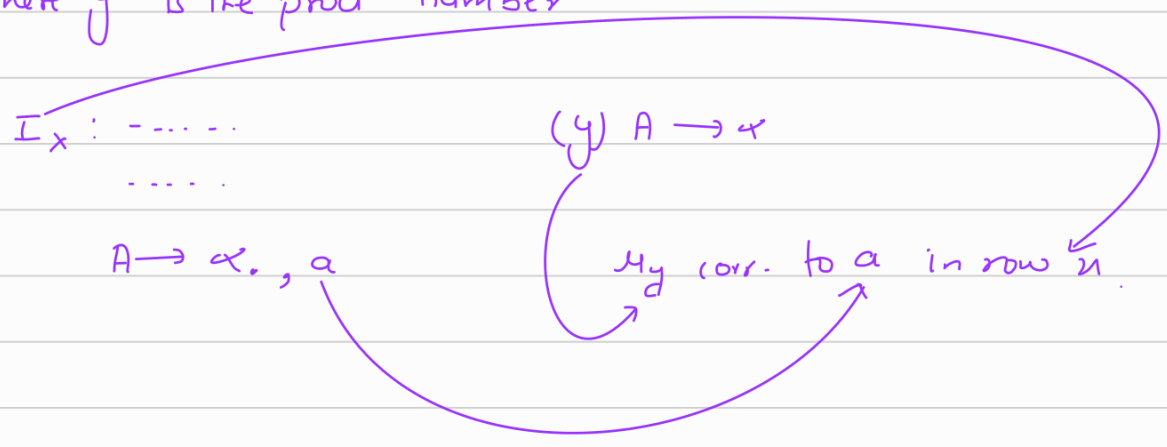
State	Action a	Action b	Action $	Goto S	Goto A
I_0	S_{36}	S_{47}		1	2
I_1			Acc.		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		

CLR(1) parse table →

↳ Canonical

- The shift & goto entries of CLR(1) parser are similar to SLR(1) parser But the reduce entries differs

let in any state I_x production $\{A \longrightarrow \alpha, a\}$ is going to reduce then we have to place reduce entry (r_y) corresponding to look ahead 'a' only where 'y' is the prod^n number



Maintain same ahead

	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			(At look ahead) Acc.		
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		

NO double entry in action ∴ grammer is CLR(1)

LR(0) item set →

Ex $S \longrightarrow L = R / R$
$L \longrightarrow * R / id$
$R \longrightarrow L$

⟹ Augumented

(0) $S' \longrightarrow S$
(1) $S \longrightarrow L = R$
(2) $S \longrightarrow R$
(3) $L \longrightarrow * R$
(4) $L \longrightarrow id$
(5) $R \longrightarrow L$

SLR(1) X
LALR(1) ✓
CLR(1) ✓

I_0:
$S' \longrightarrow . S$ G (I_1)
$S \longrightarrow . L = R$ G (I_2)
$S \longrightarrow . R$ G (I_3)
$L \longrightarrow . * R$ G (I_4)
$L \longrightarrow . id$ G (I_5)
$R \longrightarrow . L$ G (I_2)

dont open if all prod. match

I_1: $S' \longrightarrow S.$

I_2:
$S \longrightarrow L . = R$
$R \longrightarrow L.$

I_3: $S \longrightarrow R.$

I_4:
$L \longrightarrow * . R$
$R \longrightarrow . L$
$L \longrightarrow . * R$
$L \longrightarrow id$

I_5: $L \longrightarrow id.$

I_8: $R \longrightarrow L.$

I_6: $S \longrightarrow L=.R$

$R \longrightarrow .L$

$L \longrightarrow .*R$

$L \longrightarrow .id$

I_9: $S \longrightarrow L=R.$

I_7: $L \longrightarrow *R.$

LR(0) Parse table →

States	Action				Goto		
	=	*	id	$	S	L	R
I_0		S_4	S_5		1	2	3
I_1	Acc / S_6	Acc	Acc	Acc			
I_2	R_5	R_5	R_5	R_5			
I_3	R_2	R_2	R_2	R_2			
I_4		S_4	S_5			8	7
I_5	R_4	R_4	R_4	R_4			
I_6		S_4	S_5			8	9
I_7	R_3	R_3	R_3	R_3			
I_8	R_5	R_5	R_5	R_5			
I_9	R_1	R_1	R_1	R_1			

SLR(1) parse table →

States	Action				Goto		
	=	*	id	$	S	L	R
I_0		S_4	S_5		1	2	3
I_1				Acc			
I_2	S_6 / R_5			R_5			
I_3				R_2			
I_4		S_4	S_5			8	7
I_5	R_4			R_4			
I_6		S_4	S_5			8	9

I_7	r_3	r_3	
I_8	r_5	r_5	
I_9		r_1	

LR(1) itemset →

I_0: $S' \to .S, \$$
$S \to .L=R, \$$
$S \to .R, \$$
$L \to .*R, =/\$$
$L \to .id, =/\$$
$R \to .L, \$$
$L \to .*R, \$$
$L \to .id, \$$

(within state)

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

I_2: $S \to L.=R, \$$
$R \to L., \$$

I_3: $S \to R., \$$

I_4: $L \to *.R, =/\$$
$R \to .L, =/\$$
$L \to *.R, =/\$$
$L \to .id, =/\$$

I_5: $L \to id, =/\$$

I_6: $S \to L=.R, \$$
$R \to .L, \$$
$L \to .*R, \$$
$L \to .id, \$$

I_7: $S \to *R., =/\$$

I_8: $R \to L., =/\$$

I_9: $S \to L=R., \$$

I_{10}: $R \to L., \$$

I_{11}: $L \to *.R, \$$
$R \to .L, \$$
$L \to .*R, \$$
$L \to .id, \$$

I_{12}: $L \to id, \$$

I_{13}: $L \to *R., \$$

CLR(1) parse table →

Items	Action				Goto		
	=	*	id	$	S	L	R
I_0		S_4	S_5		1	2	3
I_1				Acc			
I_2	S_6			r_5			
I_3				r_2			
I_4		S_4	S_5			8	9
I_5	r_4			r_4			
I_6		S_{11}	S_{12}			10	9
I_7	r_3			r_3			
I_8	r_5			r_5			
I_9				r_1			
I_{10}				r_5			
I_{11}		S_{11}	S_{12}			10	13
I_{12}				r_4			
I_{13}				r_3			

- LALR(1) Item set →

1. Find LR(1) item set
2. Find common core in LR(1) item set

- Common core → Same set of production of two or more states with diff. look ahead

(a) Merge common core of LR(1) itemset as a single state in LALR(1) item set along with look ahead.

I_x : $A_1 \to \alpha_1 . B\beta_1 , a_1$
$A_2 \to \alpha_2 . B_2 \beta_2 , a_2$
⋮
$A_n \to \alpha_n . B_n \beta_n , a_n$

I_y : $A_1 \to \alpha_1 . B\beta_1 , b_1$
$A_2 \to \alpha_2 . B_2 \beta_2 , b_2$
⋮
$A_n \to \alpha_n . B_n \beta_n , b_n$

exact same except look ahead

⇓

I_{ny}: $A_1 \longrightarrow \alpha_1 . B_1 \beta_1 , a_1/b_1$

$A_2 \longrightarrow \alpha_2 . B_2 \beta_2 , a_2/b_2$

$\vdots$

$A_n \longrightarrow \alpha_n . B_n \beta_n , a_n/b_n$

(b) States which are not having any common core in multiple states then we have to use these state in LALR(1) item set same as LR(1) item set

let number of states in SLR(1), LALR(1) & CLR(1) parser is n_1, n_2, n_3 respectively then $\boxed{n_1 = n_2 <= n_3}$ Imp

ⓑ 'S → S

Ex $S \xrightarrow{1} aA\,d \;/\; a\xrightarrow{2}Bd / b\xrightarrow{3}Ae / b\xrightarrow{4}Be$ — LR(0) itemset

$A \xrightarrow{5} d$ — LR(1) itemset

$B \xrightarrow{6} d$ — LALR(1) item set

Ambiguous

LR(0) itemset →

I_0: $S' \longrightarrow .S$ (I_1)

$S \longrightarrow .aAd$ (I_2)

$S \longrightarrow .aBd$ (I_2)

$S \longrightarrow .bAe$ (I_3)

$S \longrightarrow .bBe$ (I_4)

I_1: $S' \longrightarrow S.$

I_2: $S \to a.Ad$ (I_4)

$S \to a.Bd$ (I_5)

$A \to .d$ (I_6)

$B \to .d$ (I_6)

I_3: $S \to b.Ae$ (I_7)

$S \to b.Be$ (I_8)

$A \to .d$ (I_6)

$B \to .d$ (I_6)

I_4: $S \to aA.d$ (I_9)

I_5: $S \to aB.d$ (I_{10})

I_6: $A \to d.$

$B \to d.$

I_7: $S \to bA.e$ (I_{11})

I_8: $S \to bB.e$ (I_{12})

I_9: $S \to aAd.$

I_{10}: $S \to aBd.$

I_{11}: $S \to bAe.$

I_{12}: $S \to bBe$

LR(0):

	Action					Goto		
I	a	b	d	e	f	S	A	B
I_0	S_2	S_3				1		
I_1	Acc	Acc.	Acc	Acc	Acc			
I_2			S6				4	5
I_3			S6				7	8
I_4			S9					
I_5			S10					
I_6	r_5 r_6	r_5 r_6	r_5 r_6	r_5 r_6	r_5 r_6			
I_7				S_{11}				
I_8				S_{12}				
I_9	r_1	r_1	r_1	r_1	r_1			
I_{10}	r_2	r_2	r_2	r_2	r_2			
I_{11}	r_3	r_3	r_3	r_3	r_3			
I_{12}	r_4	r_4	r_4	r_4	r_4			

SLR(1):

	Action					Goto		
I	a	b	d	e	f	S	A	B
I_0	S_2	S_3				1		
I_1					Acc			
I_2			S6				4	5
I_3			S6				7	8
I_4			S9					
I_5			S10					
I_6			r_5 r_6	r_5 r_6				
I_7				S_{11}				
I_8				S_{12}				
I_9					r_1			

I_{10}	r_2
I_{11}	r_3
I_{12}	r_4

LR(1) item set →

I_0: $S' \longrightarrow .S, \$$ (I_1)
$S \longrightarrow .aAd, \$$ (I_2)
$S \longrightarrow .aBd, \$$ (I_2)
$S \longrightarrow .bAe, \$$ (I_3)
$S \longrightarrow .bBe, \$$ (I_3)

I_1: $S' \longrightarrow S., \$$

I_2: $S \rightarrow a.Ad, \$$ (I_4)
$S \rightarrow a.Bd, \$$ (I_5)
$A \rightarrow .d, d$ (I_6)
$B \rightarrow .d, d$ (I_6)

I_3: $S \rightarrow b.Ae, \$$ (I_7)
$S \rightarrow b.Be, \$$ (I_8)
$A \rightarrow .d, e$ (I_9)
$B \rightarrow .d, e$ (I_9)

I_4: $S \rightarrow aA.d, \$$ (I_{10})

I_5: $S \rightarrow aB.d, \$$ (I_{11})

I_6: $A \rightarrow d., d$
$B \rightarrow d., d$

I_7: $S \rightarrow bA.e \$$ (I_{12})

I_8: $S \rightarrow bB.e, \$$ (I_{13})

I_9: $A \rightarrow d., e$
$B \rightarrow d., e$

I_{10}: $S \rightarrow aAd., \$$

I_{11}: $S \rightarrow aBd., \$$

I_{12} $S \longrightarrow bAe., \$$

I_{13} $S \longrightarrow bBe., \$$

CLR(1):

I	Action a	b	d	e	f	Goto S	A	B
I_0	S_2	S_3				1		
I_1					Acc			
I_2			S_6				4	5
I_3				S_9			7	8
I_4			S_{10}					
I_5			S_{11}					
I_6			r_5 r_6					
I_7				S_{12}				
I_8				S_{13} r_5				
I_9				r_6				
I_{10}					r_1			
I_{11}					r_2			
I_{12}					r_3			
I_{13}					r_4			

LALR(1) itemset →

I_0: $S' \longrightarrow .S, \$ \ (I_1)$
$S \longrightarrow .aAd, \$ \ (I_2)$
$S \rightarrow .aBd, \$ \ (I_2)$
$S \rightarrow .bAe, \$ \ (I_3)$
$S \rightarrow .bBe, \$ \ (I_3)$

I_1: $S' \rightarrow S., \$$

I_2: $S \rightarrow a.Ad, \$ \ (I_4)$
$S \rightarrow a.Bd, \$ \ (I_5)$
$A \rightarrow .d, d \ (I_{69})$
$B \rightarrow .d, d \ (I_{69})$

I_3: $S \rightarrow b.Ae, \$ \ (I_7)$
$S \rightarrow b.Be, \$ \ (I_8)$
$A \rightarrow .d, e \ (I_{69})$
$B \rightarrow .d, e \ (I_{69})$

I_4: $S \rightarrow aA.d, \$ \ (I_{10})$

I_5: $S \rightarrow aB.d, \$ \ (I_{11})$

I_{69}: $A \rightarrow d., d/e$
$B \rightarrow d., d/e$

I_7 : $S \to bA.e, \$$ (I_{12}) I_{11} : $S \to aBd., \$$

I_8 : $S \to bB.e, \$$ (I_{13}) I_{12} : $S \to bAe., \$$

I_{10} : $S \to aAd., \$$ I_{13} : $S \to bBe., \$$

LALR(1):

I	Action a	b	d	e	f	Goto S	A	B
I_0	S_2	S_3				1		
I_1					Acc			
I_2			S_9				4	5
I_3			S_9				7	8
I_4			S_{10}					
I_5			S_{11}					
I_{69}			r_5 r_6	r_5 r_6				
I_7				S_{12}				
I_8				S_{13}				
I_{10}					r_1			
I_{11}					r_2			
I_{12}					r_3			
I_{13}					r_4			

- Given Grammer is ambiguous as it has multiple way to create add, bde

Ex $S \to AaAb / BbBa$ LL(1) parse table

$A \to \epsilon$ SLR(1) parse table

$B \to \epsilon$ LALR(1) parse table

CLR(1) parse table

LL(1):

Var.	First	Follow
S	{a, b, ε}	{$}
A	{ε}	{a, b}
B	{ε}	{a, b}

Parse table →

Var.	$	a	b
S	S → AaAb S → BbBa	S → AaAb	S → BbBa
A		A → ∈	A → ∈
B		B → ∈	B → ∈

SLR(1):

S → AaAb / BbBa

A → ∈

B → ∈

LR(0) itemset →

I_0:

(0) S' → . S (G1)

(1) S → . AaAb (G2)

(2) S → . BbBa (G3)

(3) A → .

(4) B → .

I_1:

S' → S.

I_2:

S → A.aAb (G4)

I_3:

S → B.bBa (G5)

I_4:

S → Aa.Ab (G6)

A → .

I_5:

S → Bb.Ba (G7)

B → .

I_6:

S → AaA.b (G8)

I_7:

S → BbB.a (G9)

I_8:

S → AaAb.

I_9:

S → BbBa.

SLR(1) parse table →

Item	Action			Goto		
	a	b	$	S	A	B
I_0				1	2	3
I_1			Acc.			
I_2	S_4					
I_3	S_5					
I_4				6		
I_5				7		
I_6		S_8				
I_7	S_9					
I_8			r_1			
I_9			r_2			

LR(1) itemset →

I_0:

$S' \longrightarrow .S, \$$

$S \longrightarrow .AaAb, \$$

$S \longrightarrow .BbBa, \$$

$A \longrightarrow ., a$

$B \longrightarrow ., b$

I_1:

$S' \longrightarrow S., \$$

I_2:

$S \longrightarrow A.aAb, a$

I_3: $S \longrightarrow B.bBa, b$

I_4:

$S \rightarrow Aa.Ab, a$

$A \rightarrow ., b$

I_5:

$S \rightarrow Bb.Ba, b$

$B \rightarrow ., a$

I_6:

$S \rightarrow AaA.b, b$

I_7:

$S \rightarrow BbB.a, a$

I_8:

$S \longrightarrow AaAb., b$

I_9:

$S \rightarrow BbBa., a$

CLR(1) parse table →

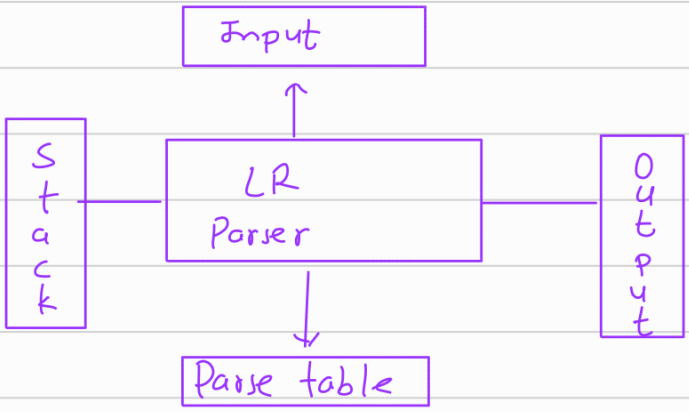
Item	Action			Goto		
	a	b	$	S	A	B
I_0						
I_1						
I_2						
I_3						
I_4						
I_5						
I_6						
I_7						
I_8						
I_9						

LALR(1) itemset →

LALR(1) parse table →

Item	Action			Goto		
	a	b	$	S	A	B
I_0						
I_1						
I_2						
I_3						
I_4						
I_5						
I_6						
I_7						
I_8						
I_9						

LR parsing →



Ex.

$E' \rightarrow E$ (0)

$E \rightarrow E * T$ (1)

$E \rightarrow T$ (2)

$T \rightarrow T * F$ (3)

$T \rightarrow F$ (4)

$F \rightarrow (E)$ (5)

$F \rightarrow id$ (6)

SLR(1) parse table →

States	Action						Goto		
	+	*	(	)	id	$	E	T	F
I_0			S_4		S_5		1	2	3
I_1	S_6					Acc.			
I_2	r_2	S_7		r_2		r_2			
I_3	r_4	r_4		r_4		r_4			
I_4			S_4		S_5		8	2	3
I_5	r_6	r_6		r_6		r_6			
I_6			S_4		S_5			9	3
I_7			S_4		S_5				10
I_8	S_6			S_{11}					
I_9	r_1	S_7		r_1		r_1			
I_{10}	r_3	r_3		r_3		r_3			
I_{11}	r_5	r_5		r_5		r_5			

After reduce refer to goto of top 2 symbol of stack

Stack	Input	Output
0	id + id * id $	Shift id 5
0 id 5	+ id * id $	reduce F → id
0 F 3	+ id * id $	reduce T → F
0 T 2	+ id * id $	reduce E → T
0 E 1	+ id * id $	shift + 6
0 E 1 + 6	id * id $	shift id 5
0 E 1 + 6 id 5	* id $	reduce F → id
0 E 1 + 6 F 3	* id $	reduce T → F
0 E 1 + 6 T 9	* id $	Shift * 7
0 E 1 + 6 T 9 * 7	id $	Shift id 5
0 E 1 + 6 T 9 * 7 id 5	$	reduce F → id
0 E 1 + 6 T 9 * 7 F 10	$	reduce T → T * F
0 E 1 + 6 T 9	$	reduce E → E + T
0 E 1	$	Accepted

$A \to \alpha$ remove $2|\alpha|$ element frm stk

Ex id * id + id

Stack	Input	Output
0	id * id + id $	Shift id 5
0 id 5	* id + id $	reduce F → id
0 F 3	* id + id $	reduce T → F
0 T 2	* id + id $	shift * 7
0 T 2 * 7	id + id $	Shift id 5
0 T 2 * 7 id 5	+ id $	reduce F → id.
0 T 2 * 7 F 10	+ id $	reduce T → T * F