Page No.
Date :

Unit : 02

* Role of Syntax Analyzer

Also called as Parser

It is crucial component of the compiler design process in programming languages. Its primary role is to check the source code for grammatical structure according to the rules of the programming language (syntax rule)

Role of Parser :-

① Syntax Checking : The main task of the syntax analyzer is to read the sequence of tokens generated by the lexical analyzer and ensures that the input string (code) follows grammar rules of programming language. If the code violates these rules, the parser generates an error.

② Parse Tree / Derivation tree Generation :

The syntax analyzer generates a parse tree, also known as derivation tree, which represent the hierarchical syntactical structure of the source code. Each node in the tree represent a construct in grammar of the language.

③ Error Handling and Reporting :
When the parser detects a syntax error, it reports the error to the programmer and attempts to recover from the error to continue parsing the rest of code. This is essential for helping developers identify where the code deviates from expected syntax.

④ Semantic Analysis Preparation :
The syntax analyzer passes the parsed structure to semantic analyzer. It helps in checking the meaning and logic of the code, but this phase cannot start until the syntax is correct.

⑤ Intermediate code Generation :
In some compilers, the syntax analyzer also pays a role in generating intermediate code, which is abstract representation of source code that can be further processed in later stage of compilation.

Key Outputs of a Syntax Analyzer :
- Parse tree : Visual representation of the structure of the code

- Error Message : Indication of syntax errors with location information
- Abstract syntax tree : A simplified, more abstract form of parse tree used in subsequent phases

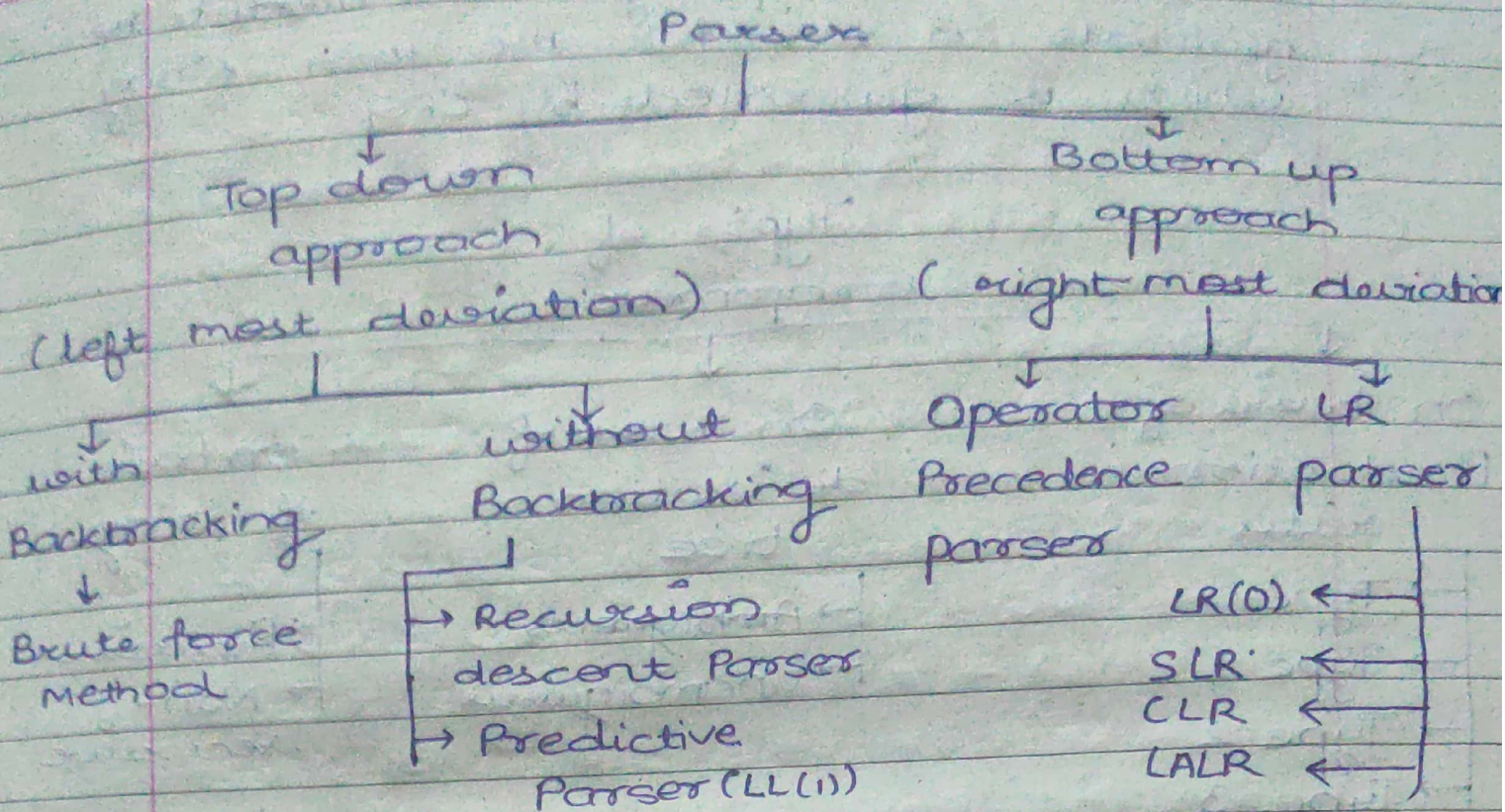
BookNotes :

→ Syntax Analysis or Parsing is second phase after lexical analysis. It checks the syntactical structure of given input and it does so, by building a data structure called Parse tree or Syntax tree.

→ The Parse tree is constructed using predefined grammar of the language and input string.

→ If the given input string can be produced with help of syntax tree then that i/p string is said to be correct syntax, if not then error is reported by syntax analyzer.

→ The main goal of syntax analysis is to create parse tree or abstract syntax tree of source code, which is the hierarchical representation of source code that reflects grammatical structure of the program.



* Grammar

It is finite set of formal rules for generating syntactically correct sentences or meaningful sentence
We will here use, CFG (context Free Grammar)

CFG : (V, T, P, S)

V → finite set of non terminals
T → finite set of terminals
P → set of production rules
S → start symbol

Production rules : $\underset{V}{\alpha} \longrightarrow \underset{V \cup T}{\beta}$

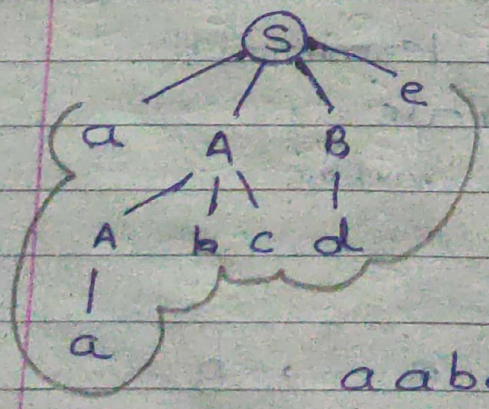
→ Parser is the program or module to generate a tree for the given string which is generated by CFG

Types of Grammar

- On basis of Production rules
  - → Type 0
  - → Type 1
  - → Type 2
  - → Type 3
- On basis of derivation tree
  - → Ambiguous
  - → Unambiguous
- On basis of string
  - → Recursive
  - → Non-recursive

→ Example to understand the difference between top down and bottom up approach :

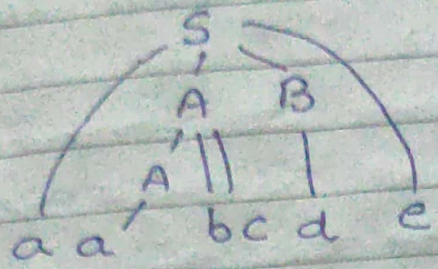
TOP-DOWN ⇒



S → a ABe
B → a Abc B e
S → a a b c B e
S → a a b c b e

a a b c d e

BOTTOM UP :



S → AABe
S → aAde
S → aAbcde
S → aabcde

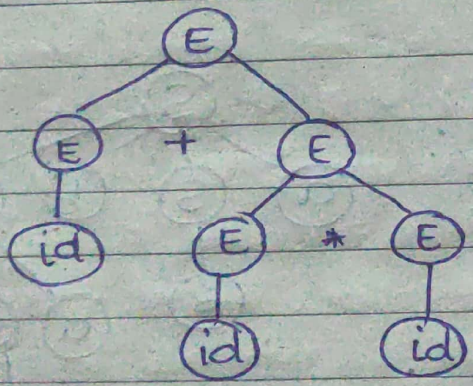
Ambiguous Grammar

Production rule of Grammar : E → E+E | E* E | id

input string : id + id * id

Left-most

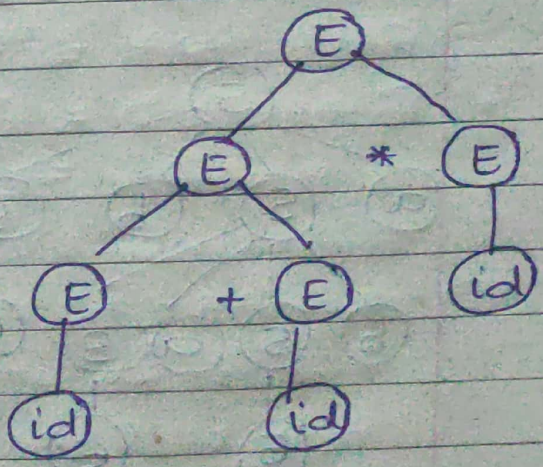
E → E + E
E → id + E * E
E → id + id * id



2 + 3 * 4
= 2 + 12
= 14

Right-most

E → E+E
E → E* E + E
E → id * id + id



2+3* 4
= 5* 4
= 20

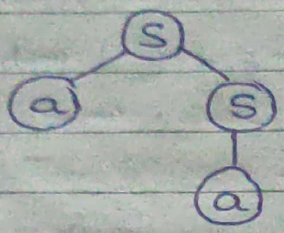
Ans are diff
So, Unambiguous

Page No.
Date :

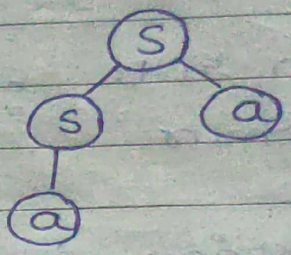
Que) Check whether given grammar is ambiguous or not.

① $S \to aS \mid Sa \mid a$ $\quad w = aa$

$S \to aS$
$S \to aa$



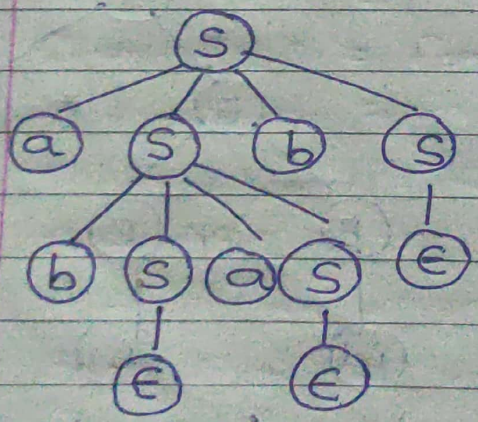
$S \to Sa$
$S \to aa$



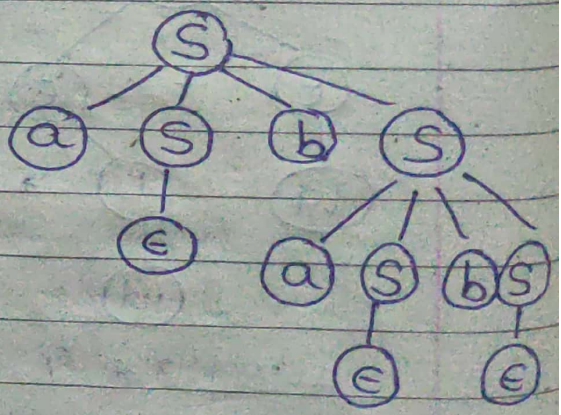
∴ Ambiguous Grammar

② $S \to aSbS \mid bSaS \mid \epsilon$ $\quad w = abab$

$S \to aSbS$
$S \to abSaSbS$
$S \to abab$



$S \to aSbS$
$S \to aSbaSbS$
$S \to abab$

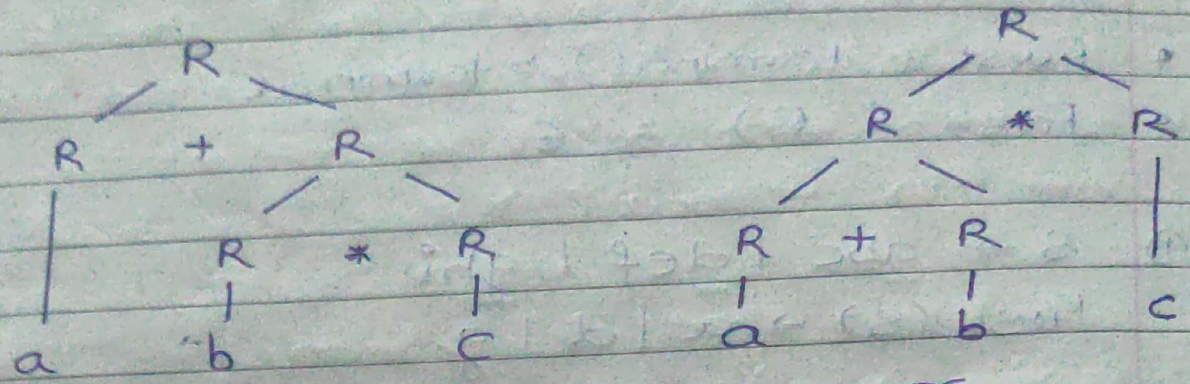


Ambiguous Grammar

③ $R \to R+R \mid RR \mid R^* \mid a \mid b \mid c$

$R \to R+R$
$R \to R+RR$
$R \to a+bc$

$R \to RR$
$R \to R+RR$
$R \to a+bc$



Ambiguous Grammar

④ $S \to AB$
$A \to aA \mid b$
$B \to bB \mid a$

$w = abbaa$

$S \to AB$
$S \to aAB$
$S \to aAbB$
$S \to abba$

$S \to AB$
$S \to aAbB$
} No other possible

∴ Unambiguous Grammar

First and follow

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

- First (terminal) = terminal
- First (ϵ) = ϵ

① S → abc | def | ghi
First (S) → a | d | g

② S → ABC | ghi | jkl
A → a | b | c
B → b
D → d
First (D) = d
First (B) = b
First (A) = a, b, c
First (S) = First (A) ∪ g ∪ j
= a, b, c, g, j

③ S → ABC
A → a | b | ϵ
B → c | d | ϵ
C → e | f | ϵ
First (C) = e, f, ϵ
First (B) = c, d, ϵ
First (A) = a, b, ϵ

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

[Here First (A) mei ϵ hai, so we do not write ϵ in First (S) rather we put it in S so we get S → BC. Now we see First (B) and then we see First (C) and S → ϵϵϵ so we add ϵ]

④ E → TE'
E' → * TE' | ϵ
T → FT'
T' → ϵ | +FT'
F → id | (E

First (F) = id, (
First (T') = ϵ, +
First (T) = First (F) = id, (
First (E') = *, ϵ
First (E) = First (T) = id, (

⑤ S → AaAb | BbBa
A → ϵ
B → ϵ
First (B) = ϵ
First (A) = ϵ
First (S) = First (A) U First (B)
= a, b

- If α is the string of grammar symbol the first (α) is a set of terminals that begins the string derived from α.
- If $\alpha \to \epsilon$ and $\alpha^* \to \epsilon$ (α derives ϵ in more than one step) the 'ϵ is also in first (α)
- ϵ can ~~start~~ only be present in first() not in follow ( ).

Follow ( )

It contains set of all terminal present in immediate right of 'A'

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

- S → ACD
C → a/b
Follow (A) = First (C) = {a, b}
Follow (D) = {$}
If A ke right mei terminal like Ab the follow mei directly 'b' likh lege
But in case of non terminal we write First of it. For eg AB
the follow (A) = First (B)

Yaha D ke aage kuch nhi hai matlb 'ϵ' hai so follow (D) = Follow(S)

- S → aSbS | bSaS | ϵ

Follow (S) = $, b, a

Har ek mei check krna jaha jaha S likha hai

Here S is ~~starting~~ symbol so it will contain $

Also, Follow never contains ϵ

- S → AaAb | BbBa

A → ϵ

B → ϵ

Follow (A) = a, b

Follow (B) = b, a

- S → ABC

A → DEF

B → ϵ

C → ϵ

D → ϵ

E → ϵ

F → ϵ

Follow (A) = First (B)

↓

ϵ hai first of B but hum ϵ vaise ~~ko~~ likhte nhi hai

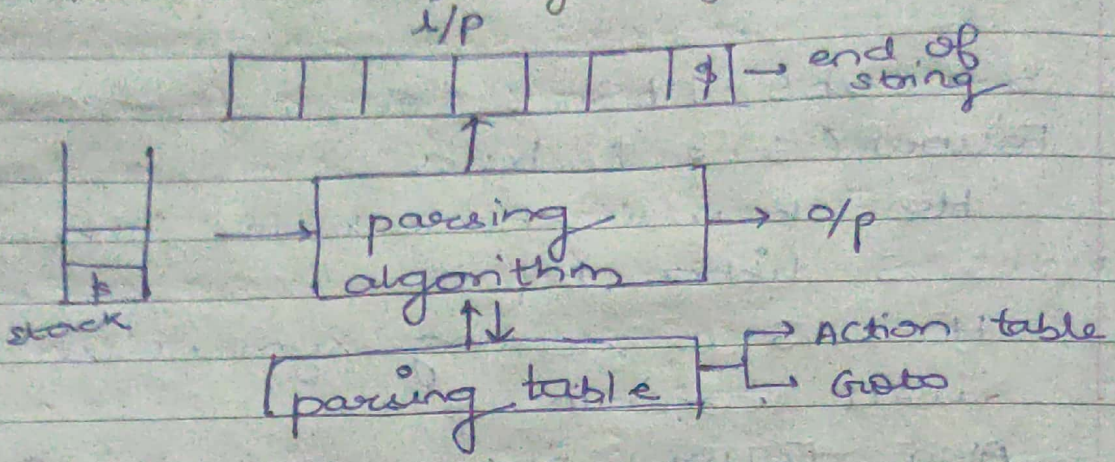
So put ϵ instead of B in S → ABC

S → A ϵ C ϵ

So, Follow (A) = First (C) = Follow(S)

= {$}

Top down Parsing algorithm



We know that top down parsing technique parses the input, and start constructing a parse tree from root node gradually moving down leaf node

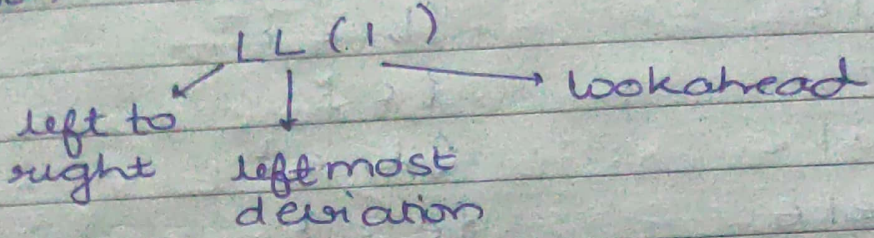
Recursive Descent Parsing :

It is top-down parsing technique that constructs the parse tree from the top and input is read from left to right.

A form of recursive descent parsing that does not require any back tracking is known as predictive parsing

LL(1) Parsing

A LL parser accepts LL grammar. The LL grammar is subset of CFG but with some restrictions to get simplified version, in order to achieve easy implementation



A grammar G is LL(1) if $A \to \alpha | \beta$ are two distinct productions of G :

- for no terminal, both α and β derives string beginning with a
- at most one of α and β can derive empty string
- If $\beta \to t$, then a does not derive any string beginning with terminal in follow(A)

→ In parsing table, each block should have 1 action, if there is more than 1 actio it leads to ambiguity and such table is not LL(1) parsing table

		first()	follow()
Que)	S → ABCDE	{a, b, c}	{a, b, c, d, e, $}
	A → a | ϵ	{a, ϵ}	{b, c}
	B → b | ϵ	{b, ϵ}	{c}
	C → c	{c}	{d, e, $}
	D → d | ϵ	{d, ϵ}	{e, $}
	E → e | ϵ	{e, ϵ}	{$}

Que)		first()	follow()
	S → BbICd	{b, d, a, c}	{$}
	B → aB | ϵ	{a, ϵ}	{b}
	C → cC | ϵ	{c, ϵ}	{d}

Que) Design LL(1) parsing table for grammar given below and show the moves made by predictive parser on input string
id + id * id

	first()	follow()
E → TE'	(, id	$,)
E' → +TE' | ϵ	+, ϵ	$,)
T → FT'	(, id	$,), +
T' → *FT' | ϵ	*, ϵ	+, $,)
F → (E) | id	(, id	+, $,), *

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

stack	input	o/p
$	id + id * id $	
$E	id + id * id $	
$E'T	id + id * id $	$E \to TE'$
$E'T'F	id + id * id $	$T \to FT'$
$E'T'id	id + id * id $	$F \to id$
$E'T'	+ id * id $	
$E'	+ id * id $	$T \to \epsilon$
$E'T+	+ id * id $	$E' \to +TE'$
$E'T	id * id $	
$E'T'F	id * id $	$T \to FT'$
$E'T'id	id * id $	$F \to id$
$E'T'	* id $	
$E'T'F*	* id $	$T' \to *FT'$
$E'T'F	id $	ϵ
$E'T'id	id $	$F \to id$
$E'T'	$	

Page No.
Date :

Que)

	first()	follow()
$S \to iEtSS' \mid a$	$\{i, a\}$	$\{\$, e\}$
$S' \to eS \mid \epsilon$	$\{e, \epsilon\}$	$\{\$, e\}$
$E \to b$	$\{b\}$	$\{t\}$

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

As more than 1 rule, This is Not a LL(1) grammar

Que)

$S \to aSbS \mid bSaS \mid \epsilon$

first(S) a, b, ϵ

follow(S) = {a, b}

The given grammar is not LL(1)

Note :

① $first(\alpha_1) \cap first(\alpha_2) \cap \ldots first(\alpha_n) = \phi$

② $first(\alpha) \cap follow(A) = \phi$

To construct the top down parser the CFG should not have :

1. Left recursion
2. Non-determinism

In compiler design, the top down approach refers to method of syntax analysis (or parsing) where the parser starts from highest level of parse tree, which represent start from the highest level of the parse tree, which represents the start symbol of the grammar, and works its way down to the leaves, which represent the terminals. The process invdves breaking down a high-level rule into its component until the entire input string is matched.

Top down Parsing : The process begins at start symbol and attempts to match the input string by recursively applying production rules to expand non-terminals.

Page No.
Date :

Recursive Descent Parsing

→ This is common method used in top-down parsing
→ It consist of set of recursive procedures where each procedure corresponds to non-terminal in the grammar
→ The parser tries to match the input string by calling these procedures according to rules of grammar

Predictive Parsing

→ A specific type of recursive descent parsing
→ It is non-backtracking and requires a lookahead symbol (usually one token)
→ The parser predicts which rules to apply based on next input symbol
→ LL(1) parser are common example where 'L' refers to left to right scanning of the input and '1' refers to lookahead token

Adv :
① Simplicity
② Readability

Disadv :
① Limited Power
② Inefficiency

Ambiguous Grammar :

A CFG is called ambiguous if there exits more than one derivation tree or parse tree

Eg. $S \rightarrow S+S \mid S*S \mid S \mid a$

Unambiguous Grammar :

A CFG is called unambiguous if there exist one and only one derivation tree or ~~post~~ parse tree

Eg $X \rightarrow AB$

$A \rightarrow Aa \mid a$

$B \rightarrow b$

* Bottom Up Parsing

It is another approach to syntax analysis, where the parser starts from input symbol and works its way up to construct the parse tree, eventually arriving at start symbol of the grammar. This approach is opposite of top-down parsing

→ The parser begin with the input token and tries to combine them into higher level syntactically structure, ultimately producing start symbol of the grammar

→ It effectively reduce the input string to start symbol by applying production rules in reverse

Shift Reduce Parsing

→ The most common form of bottom up parsing

→ The parser operates using two main function :

- Shift : The next input symbol is shifted onto a stack
- Reduce : The symbol on stack that match right hand side of a production rule are reduced to the non terminal on left hand side

LR Parsing :

→ A powerful type of bottom up parsing that handles a wide range of grammar

→ LR(K) parser where 'L' stands for left to right scanning, 'R' stands for constructing the rightmost derivation in reverse and k indicates the number of lookahead token used to make parsing decision

→ The most common types are SLR, LALR, CLR

→ LALR parser are widely used in practice as they offer a good balance between

power and complexity

Adv :
① Powerful
② Efficiency

Disadv :
① Complexity
② Stack Management

Page No.
Date :

youtube LL(1)

We will check if it is LL(1) grammar or not. Or we can can whether the LL(1) parser will accept this grammar or not

Eg

	first	follow
$S \to (L) \mid a$	(, a	\$, , ,)
$L \to SL'$	(, a	)
$L' \to \epsilon \mid , SL'$	ϵ , ,	)

Passing table

	(	)	a	,	\$
S	$S \to (L)$		$S \to a$		
L	$L \to SL'$		$L \to SL'$		
L'	.	$L' \to \epsilon$		$L' \to , SL'$	$L' \to \epsilon$

Short trick:

$S \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \cdots$

If there is no Null production or ϵ production

First = F

$F(\alpha_1) \cap F(\alpha_2) \cap F(\alpha_3) \cap \cdots = \phi$

Then LL(1)

if $\neq \phi$ then not LL(1)

If $S \to \alpha_1 \mid \alpha_2 \mid \epsilon$

$First(\alpha_1) \cap First(\alpha_2) \cap Follow(S) = \phi$

① $S \to aSa \mid bS \mid c$

$a \cap b \cap c = \phi$ (It is LL(1))

② $S \to i(tSS_1 \mid a$ in $a, \uparrow = \phi$

$S_1 \to es \mid \epsilon$ $e \cap e, \$ \neq 0$

$C \to b$

It is not LL(1) grammar

→ Top down parser

→ LL(1) parser → lookahead symbol

↙ left to right ↘ leftmost derivation

Eg. $S \to AA$

$A \to aA$

$A \to b$

① LL(1) Parsing algo

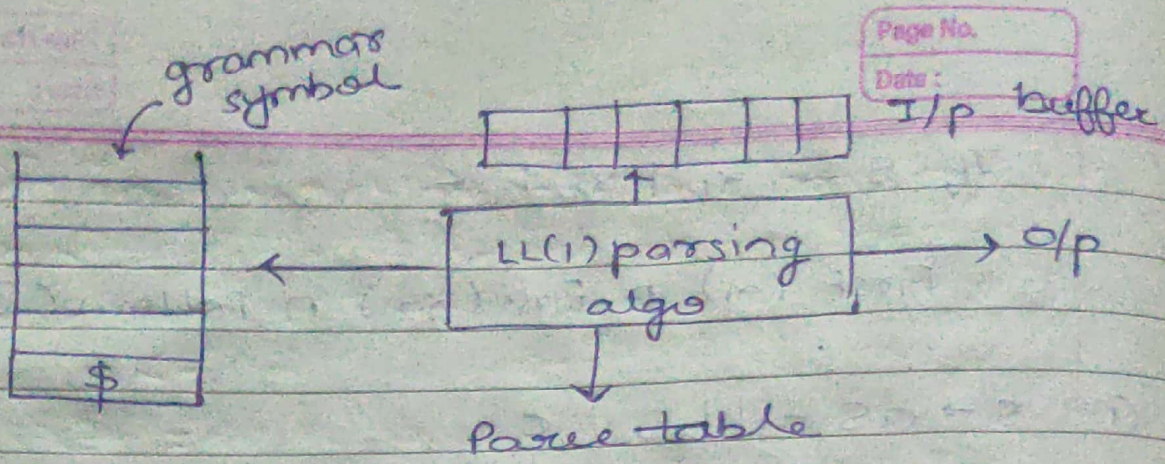
② Stack

③ Parse table

↓

Ye decide karta hai ki kaunsi move choose karni hai aur uske help se LL(1) parsing algo takes decision

	a	b	$
S	$S \to AA$	$S \to AA$	
A	$A \to aA$	$A \to b$	



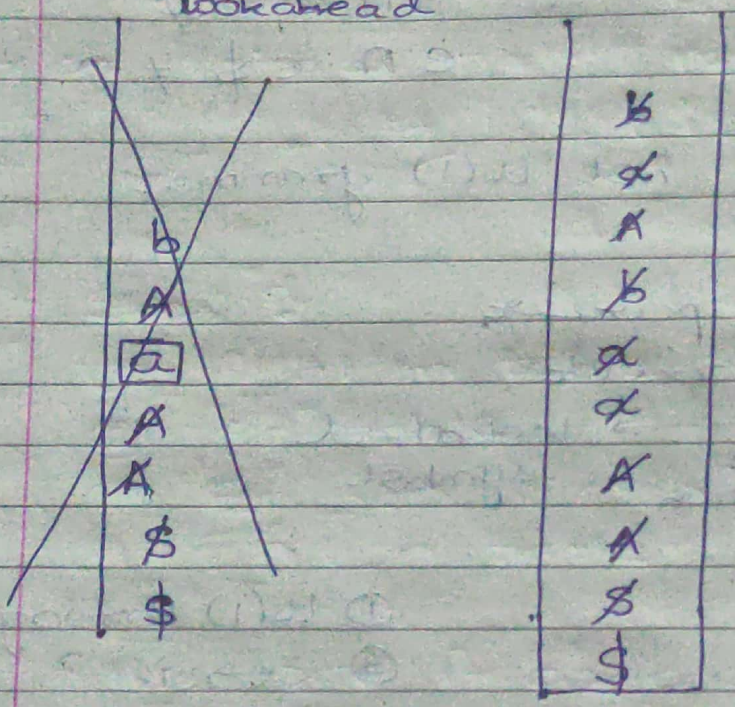
@	b	a	b	$

lookahead

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$



a	b	a	b	$

Operator Precedence Parsing

For operator precedence parsing, it is necessary that the given grammar should be operator grammar.

There are 2 cond^n for operator grammar:

① Grammar ke RHS mei koi 'ε' nhi hona chahiye

② Grammar ke RHS me non-terminals adjacent nhi hone chahiye ya sath hone nhi chahiye

$E \to AB$
$A \to a$
$B \to b \mid \varepsilon$
✗

$E \to EOE$
$E \to id$
$D \to + \mid * \mid /$
✗

$E \to E+E$
$E * E$
$E \mid E \mid id$
✓

Operator Grammar have the property that no production right side is empty or has two adjacent non-terminals.

Eg. $E \to EAE \mid id$
$A \to + \mid *$

The above grammar is not an operator ~~pre~~ grammar but we can convert the grammar into operator grammar like:

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

In operator precedence parsing we can define 3 disjoint precedence relation :

① <.
② =
③ .> between certain pair of terminal

For eg $T \rightarrow T+T \mid T*T \mid id$

string : id + id * id

Steps :

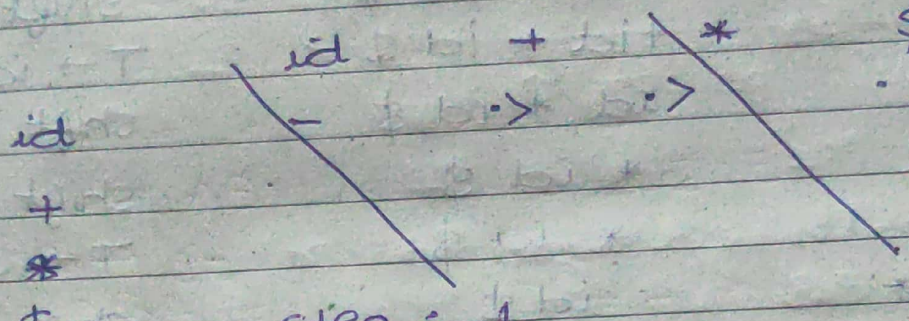
① Relation table
② OP table
③ Function Graph
④ Function table

Rules for Precedences :

① id has higher presedence than any other symbol
② $ has lowest precedence
③ if two operator has equal precedence then we check the associativity of that particular operator.

Relation	Meaning
$a <\cdot b$	a yields precedence to b or vice versa
$a =\cdot b$	a has the same precedence as b
$a \cdot> b$	a takes precedence over b

Precedence table



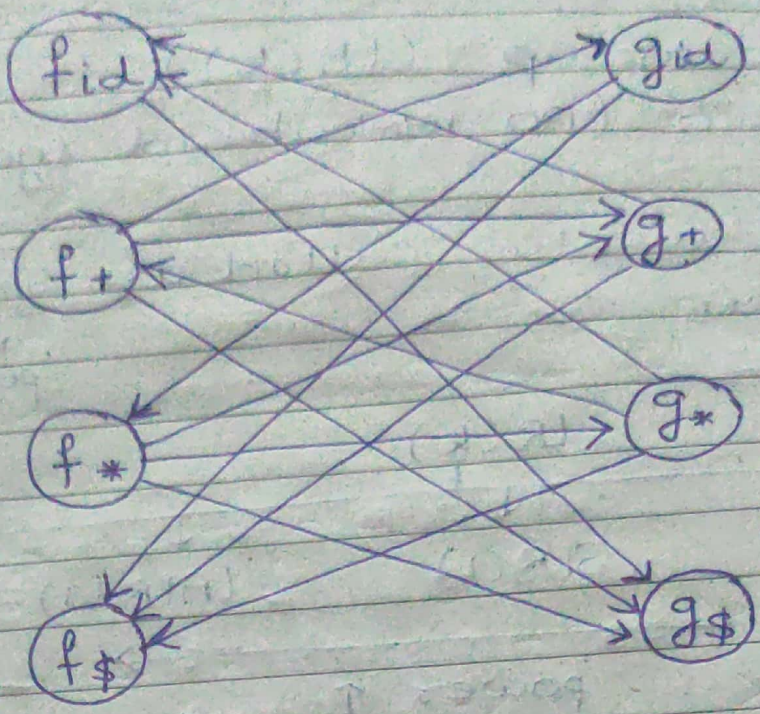
	id	+	*	$
id	-	<·	<·	·>
+	·>	·>	<·	·>
*	·>	·>	·>	·>
$	<·	<·	<·	Accept

Step 2 : If LHS < RHS
then shift
else
reduce

Operational table :

	Stack	input	Comment
	$	id + id * id $	
$ < id	$ id	+ id * id $	Shift
id > +	$ T	+ id * id $	T → id
$ < +	$ T +	id * id $	shift
+ < id	$ T + id	* id $	shift
id > *	$ T + T	* id $	T → id
+ < *	$ T + T *	id $	shift
* < id	$ T + T * id	$	shift
id > $	$ T + T * T	$	T → id
* > $	$ T + T	$	T → T * T
+ > $	$ T	$	T → T + T
	$ T	$	Accept

Step 3 : Function graph



Step 4 :
He we need to find the maximum longest path

For f+ : f+ → g+ → f$
f+ → g$
f+ → g+ → fid → g$

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

→ Time Complexity : O(2n)
fid < gid
4 5

So, function table ki error handling capacity is less

* LR(k)

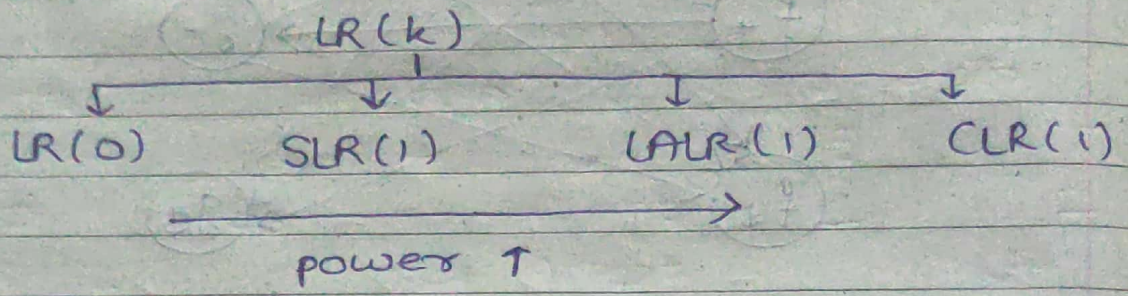
Bottom up (LR(k))

If k = 0 (no lookahead symbol)

LR(k) is also called as Shift reduce Parser

Shift → push

reduce → pop



→ Augmented grammar.

S → AB

A → a

B → b

AG :- S' → S

S → AB (1)

A → a (2)

B → b (3)

In general,

we add S' → S (start symbol) and number the production rules

Each LR parsing table contains two columns :

① Action → set of terminals → Action can be shift or reduce

② Goto → set of non terminals

LR(0) Parsing table

→ For this we use LR(0) canonical items

→ For making canonical item we add augmented grammar

$E \rightarrow T+E \mid T$

$T \rightarrow id$

AG : $E' \rightarrow .E$

→ '.' batata hai abhi tak hmne kya dekha

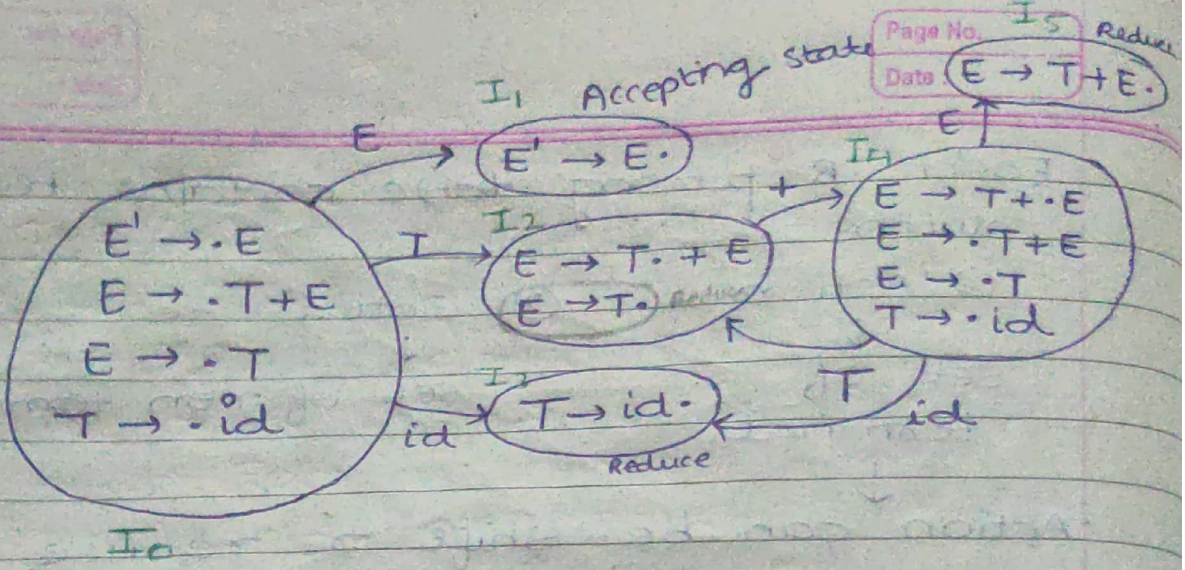
→ If '.' is on the right side it is LR(0) item

→ If $S \rightarrow .AB$, abhi tak kuch nhi dekha

→ If $S \rightarrow A.B$, seen A but not B

→ If $S \rightarrow AB.$, seen A and B

Now, E ke same '.' hai so E ke saare production mei same '.' lagana h and Ily T ke bhi production mei '.' aayega



state	Action			Goto	
	id	+	$	E	T
0	S_3			1	2
1			Accept		
2	r_2	S_4/r_2	r_2		
3	r_3	r_3	r_3		
4	S_3			5	2
5	r_1	r_1	r_1		

In this table, we can see there is shift and reduce at one cell, so it is not LR(0)

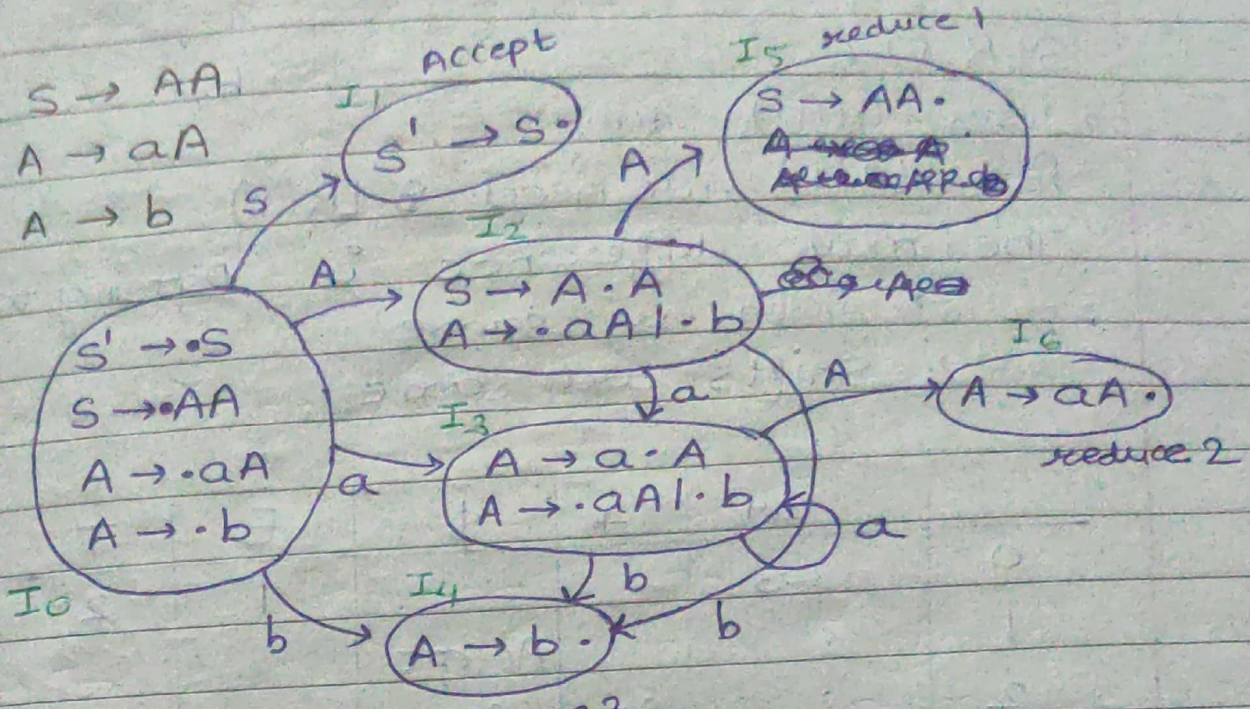
Conflicts are :

① Shift Reduce

② Reduce Reduce

Page No.
Date

Q.] $S \to AA$
$A \to aA$
$A \to b$



States	Action			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

∴ LR(0) Grammar

SLR (1)

→ Uses LR(0) items

For SLR(1), we need & LR(1)

The main difference is Parsing table:

→ Goto is same

→ Shifts are same

	id	+	$	E	T
0	S_3			1	2
1			Accept		
2		S_4	r_2		
3		r_3	r_3		
4	S_3			5	2
5			r_1		

→ In LR(0) hmne reduced tino terminals mei likha tha

Now, For eg. I_2 mei $E \to T.$ reduce hai so hum abhi E ka follow() nikalege aur vaha likhege

follow (E) = $

follow (T) = $, +

→ Yaha Shift reduce conflict nhi hai

→ Jo LR(0) hai vo SLR(1) rahega but jo SLR(1) hai vo LR(0) reh bhi sakta hai or nhi rahega

Eg 2

states	Action			Goto	
	a	b	$	S	A
0	S_3	S_4			2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5			r_1		
6	r_3	r_3	r_3		

follow (A) = $, a, b
follow (S) = $
follow (A) = $, a, b

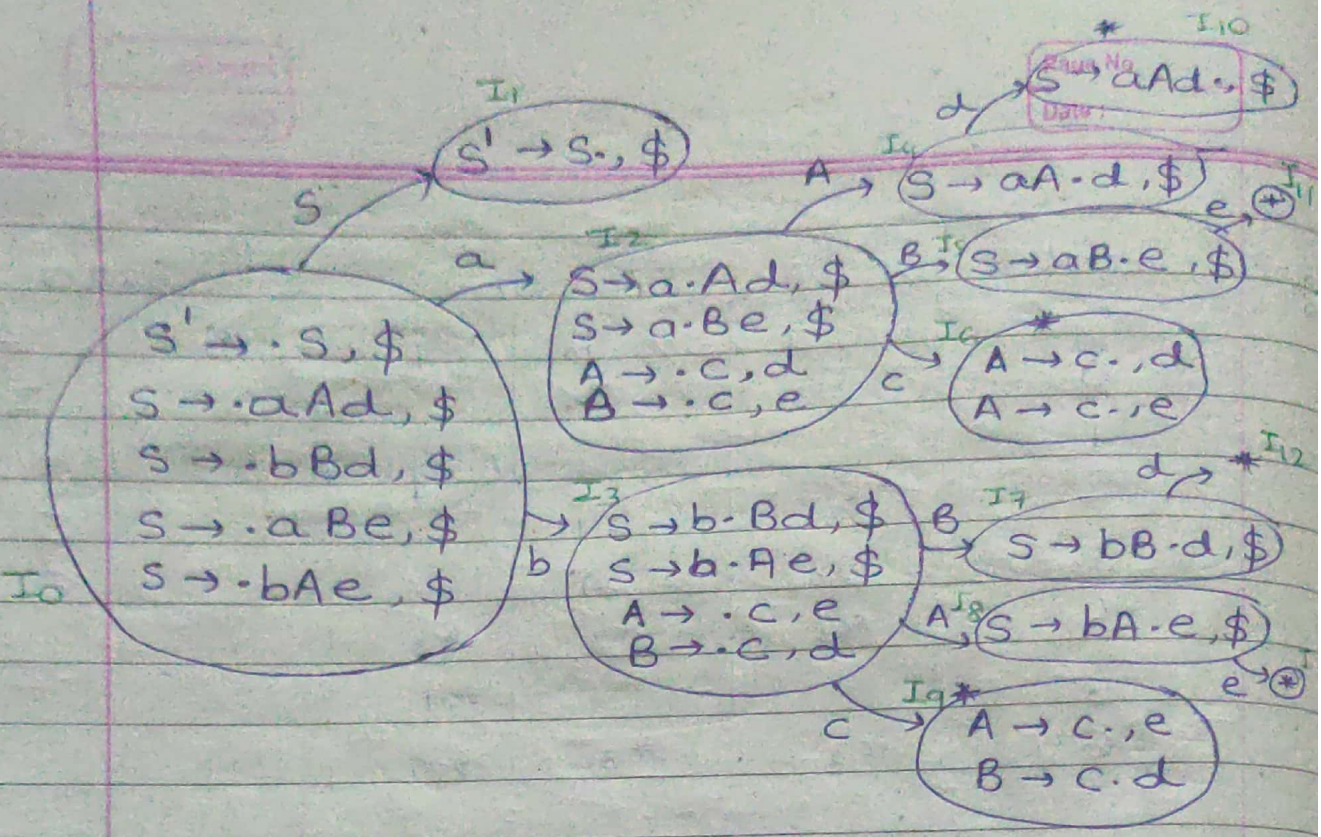
CLR Parsing table

→ Uses LR(1) cannonical items

Eg: S → aAd | bBd | aBe | bAe
A → c
B → c

$ ka first add kiya sabme

AG: S' → ·S, $
S → ·aAd, $
S → ·bBd, $
S → ·aBe, $
S → ·bAe, $



I_2 mei jab lookahead add krege tab first A and B ke aage jo hai unka nikalege $F(d, \$) = d, \$$

$F(e, \$) = e, \$$

Parsing table :

Action Goto

Yaha reduce wha likhege jaha lookahead hai

LALR < CLR

LALR Parsing table

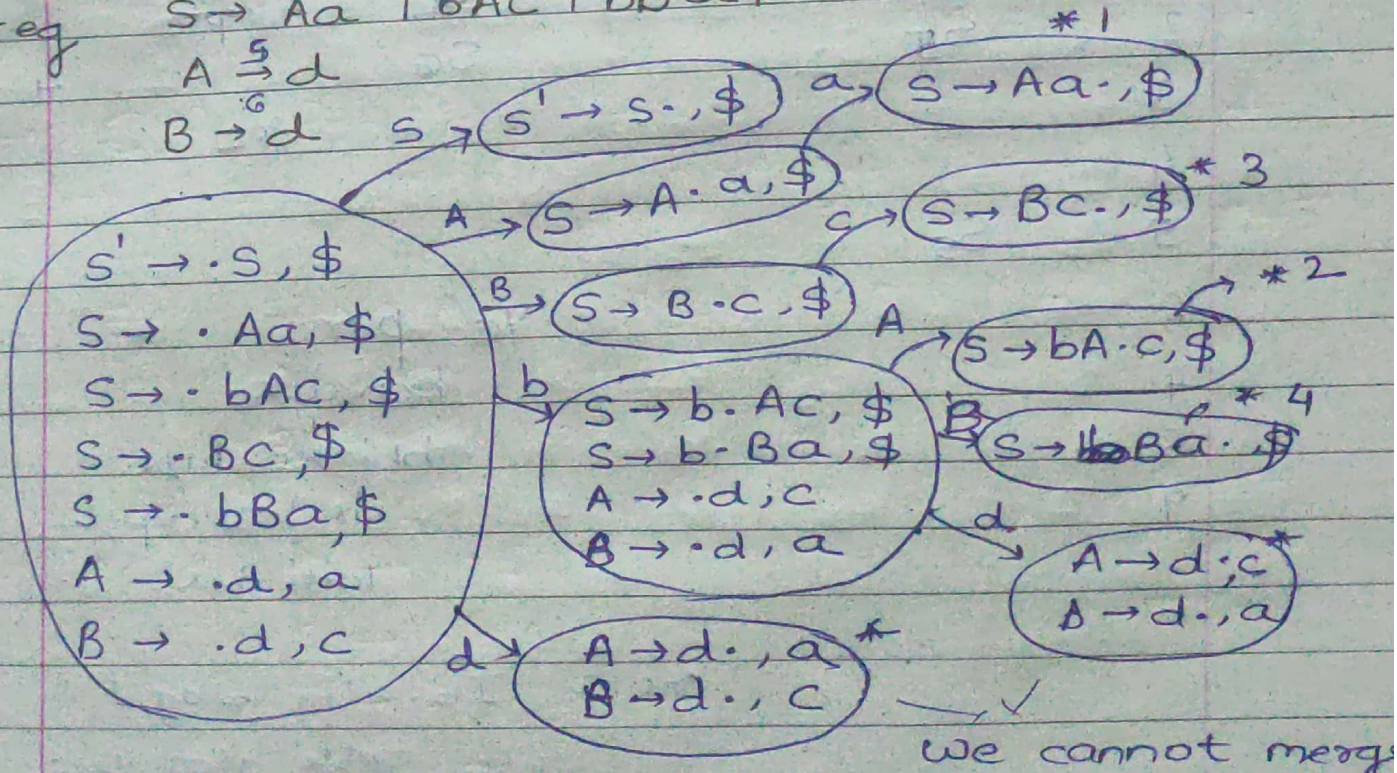
CLR mei no. of states sabse jada hoti hai so we merge some states

For eg I6 and I9 are same but the difference is of lookahead.

For eg $S \to {}^{1}Aa \mid b{}^{2}Ac \mid bB{}^{4}a \mid B{}^{3}c$

$A \overset{5}{\to} d$

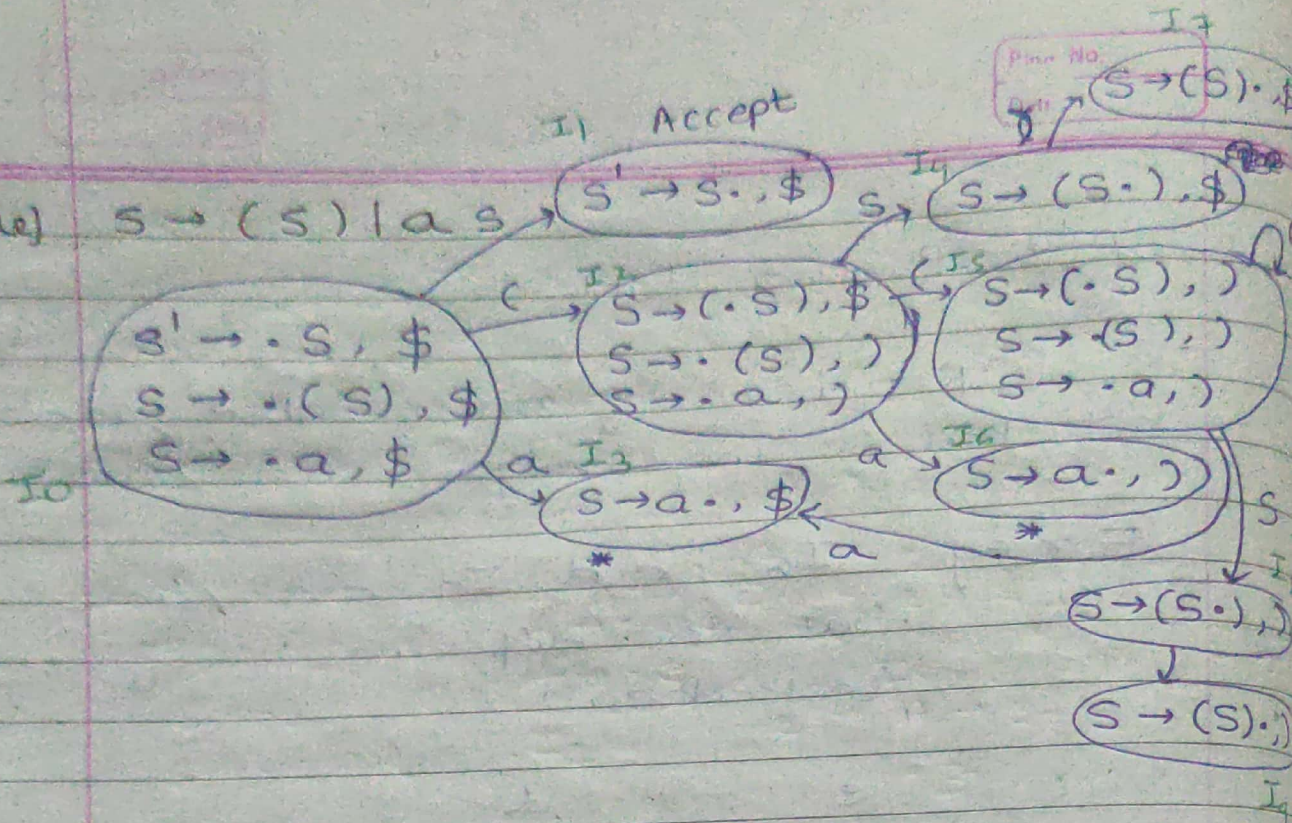
$B \overset{6}{\to} d$



we cannot merge these bcz lookahead diff

i.e multiple entries

Que) $S \to (S) \mid a\,S$



	(	)	a	$	S
0	S_2		S_3		1
1	~~S_2~~		~~S_3~~	Accept	
2	S_5		S_6		4
3				r_2	
4					
5	S_5				
6					
7					
8					
9					

Ye pura banaya abhi ka hije sir ka copy krti hu.

mera Io ... Ii alag hai isliee table alag aaya

CLR ⇒

	(	)	a	$	S
0	S_2		S_3		1
1				Accept	
2	S_4		S_6		5
3				r_2	
4	S_4		S_6		8
5		S_7			
6		r_2			
7				r_1	
8		S_9			
9		r_1			

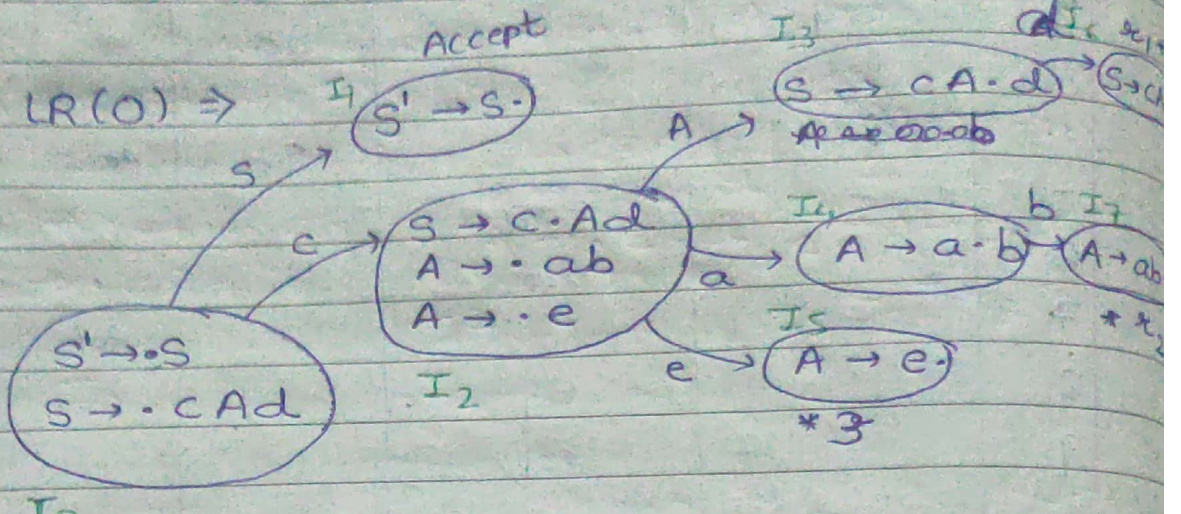
LALR ⇒

	(	)	a	$	S
0	S_2		S_3		1
1				Accept	
24	S_4		S_6		58
36	r_2			r_2	
58		S_{79}			
79		r_1		r_1	

Page No.
Date :

Ques $S \to cAd$ (1)
$A \to ab$ (2) $| e$ (3)

LR(0) ⇒



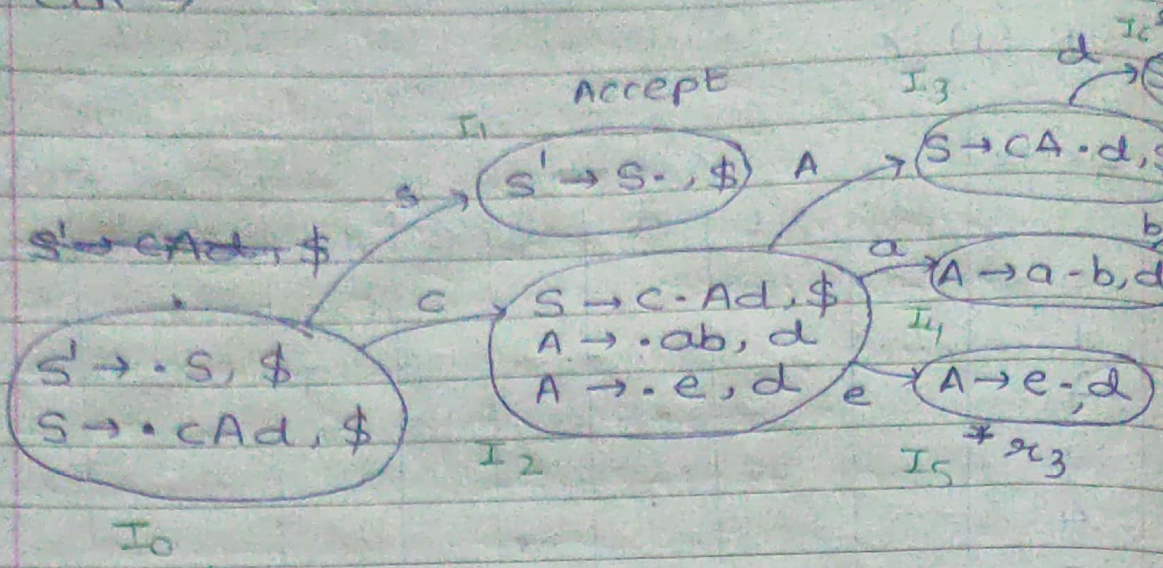
state	Action						Goto	
	c	d	a	b	e	$	S	A
0	S_2						1	
1						Accept		
2			S_4		S_5			3
3		S_6						
4				S_7				
5	r_3	r_3	r_3	r_3	r_3	r_3		
6	r_1	r_1	r_1	r_1	r_1	r_1		
7	r_2	r_2	r_2	r_2	r_2	r_2		

SLR (1) →

State	Action						Goto	
	c	d	a	b	e	$	S	A
0	S_2						1	
1						Accept		
2			S_4		S_5			3
3		S_6						
4				S_7				
5			r_3			r_1		
6								
7			r_2					

follow (A) = d
follow (S) → $
follow (A) = d

CLR ⇒



State	Action						Goto	
	c	d	a	b	e	$	S	A
1						Accept		
2			S_4		S_5			3
3		S_6						
4				S_7				
5		r_3						
6						r_1		
7		r_2						
0	S_2						1	

LALR ⇒ No reduction possible

YACC

Unit : 3

* Syntax Directed Defn (SDT)

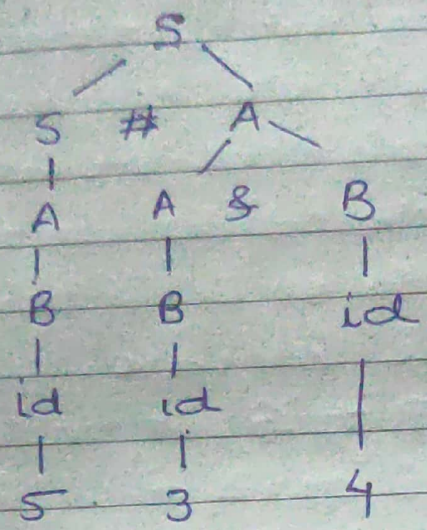
→ Entered 3rd phase (Semantic)
→ Syntax Directed Translation
→ Where we check logic / meaning
→ Input is parse tree
→ Output is annotated parse tree
→ Annotated means adding extra text / comments

Appl^{n} :

① Executing Arithmetic Expression
② Conversion from infix to postfix
③ —— // —— infix to prefix
④ — // —— binary to decimal
⑤ — // — decimal to binary
⑥ Counting number of reduction
⑦ Creating syntax tree
⑧ Generating intermediate code
⑨ Type checking
⑩ Storing type info into symbol table

$S \rightarrow S \# A \mid A \quad \{S.val = S.val * A.val;\}$
$A \rightarrow A \& B \mid B \quad \{A.val = A.val + B.val;\}$
$B \rightarrow id \quad \{B.val = id.val\}$

5 # 3 & 4



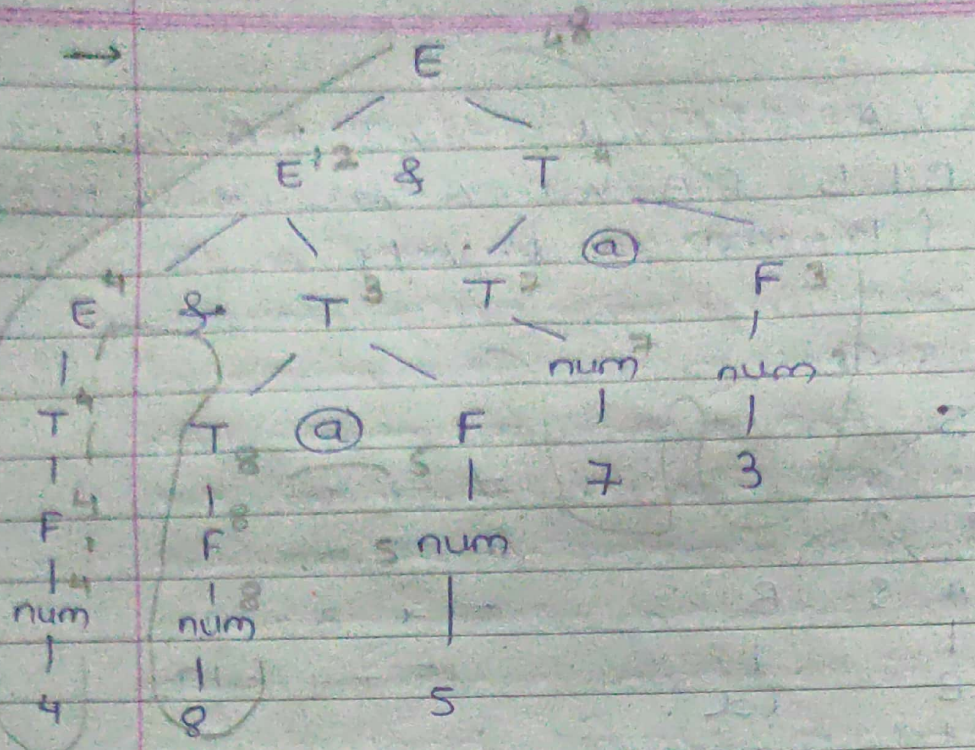
→ *
& → +

5 * 3 + 4
15 + 4 = 19

Que] $E \rightarrow E \& T \quad \{E.val = E.val * T.val\}$
$\mid T \quad \{E.val = T.val\}$
$T \rightarrow T @ F \quad \{T.val = T.val - F.val\}$
$\mid F \quad \{T.val = F.val\}$
$F \rightarrow num \quad \{F.val = num\}$

I/p : 4 & 8 @ 5 & 7 @ 3

→



8 @ 5 = 8 - 5 = 3

4 & 3 = 4 * 3 = 12

7 @ 3 = 7 - 3 = 4

12 & 4 = 12 * 4 = 48

Theory :

A syntax directed def^n (SDD) is a formalism used to define syntax and semantics of a programming language in a structured way. It associates grammar rules with semantic actions or attributes that can be evaluated to perform various task during compilation

Types :

① L-Attributed Definitions :

→ All attributes are either synthesized or inherited in manner that allows them to be evaluated in single left to right traversal of parse tree

→ Used in bottom-up parsing

② S-Attributed Definition :

→ Only synthesised attributes are used

→ The def^n are particularly straight-forward to implement in bottom up parser

Semantic analysis

It is the phase in compiler design that occurs after syntax analysis and involves checking the source code of semantic consistency according to the language rules. It ensures that the program's logic is correct by enforcing rules related to datatype, scop object lifetimes and other attributes that go beyond simple syntax.
The semantic analyzer typically operate on the parse tree generated by the syntax analysis phase, enhancing it with additional information necessary for later stages of compilation. such as intermediate code generation

Importance

① Type checking : It makes sure that operation like addition or multiplicatio are done between compatible data types, such as adding two number instead of trying to add a number to a word.

② Scope checking : It ensures that the variables and functions are used in correct place, meaning they are declared before they are used and within right part of program

③ Consistency checking : It checks that things like functions are used correctly such as making sure that when a function is called, it gets the right number and type of argument

④ Error detection : It finds mistakes that are more about logic than structure, like using a variable that was never declared

⑤ Preparing for code generation :
It gathers all the necessary information so that next step can be done correctly, such as knowing what type of data each variable holds

TSCEP

TSCEP

Semantic Analyzer :

It uses syntax tree and symbol to check whether the given program is semantically consistent with language definition. It gather type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during ICG.

Errors recognized by Semantic analyzer

1. Type mismatch
2. Undeclared variable
3. Reserved identifier misuse

Functions of Semantic analysis :

1. Type checking ⇒ Ensures that datatypes are used in a way consistent with their definitions
2. Label checking ⇒ A program should contain 'label references'
3. Flow control check : keep a check that control structures are used in a proper manner (eg. no break statement outside a loop)

Eg.
float x = 10.1;
float y = x * 30;

In above eg integer 30 will be typecasted to 30.0 before multiplication by the semantic analyzer

Static Semantics :

It is named so because of the fact that these are checked at compile time. The static semantics and meaning of the program during executions are indirectly related.

Dynamic Semantic Analysis :

It defines the meaning of different units of program like expressions and statements. These are checked at runtime unlike static semantics

Unit : 1

Compiler :- A compiler is a program that translates code written in high level programming language into machine code that a computers processor can execute. This process allows the written code to be understood and run by the computer

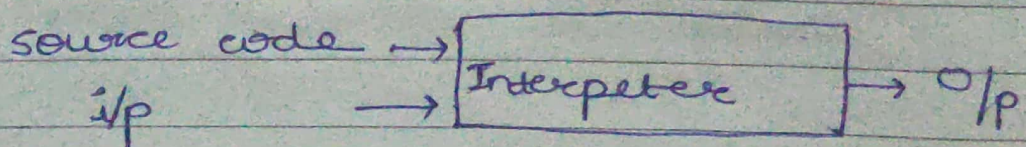
Type of language processor :
① Compiler
② Interpreter

→ Error report krne ka kaam is of compiler
→ Intepreter gives error statement by statement
→ The main job of language processor is to report the errors

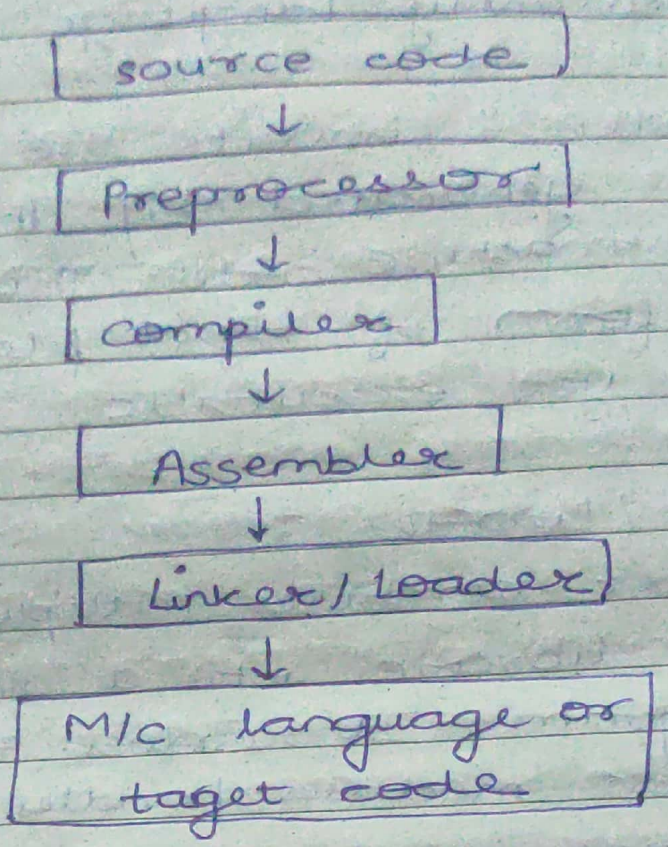
Steps of Compiler ⇒

src code → compiler → target code
i/p → compiler → o/p

Steps of Interpeter ⇒



* Analysis of source Program



- Source code : The orignal program written by developer in high level language

- Preprocessor : Handles directives like #include, #define, etc. It processes these instructions and produces an expanded source code with all macros expanded and header files included.

- Compiler : Translates preprocessed source code into an intermediate form, often called object code. This stage

syntax and semantics, optimizes code, and converts high level instructions into assembly language.

- Assembler : Converts the assembly code generated by the compiler into machine code. This code is in the form of object files, which are not yet executable.

- Linker / Loader :
  → Linker combines multiple objects files and libraries into a single executable program. It resolves references between modules, handles external libraries and produces an executable file.
  → Loader loads the executable into memory for execution by the CPU.

- Machine code / Target code :
  The final output that is executed by the processor. It's a sequence of binary instruction that the computer's hardware can directly interpret and run.

* Structure of
  Phases

Lexical

Syntax

Seman

Interme
gene

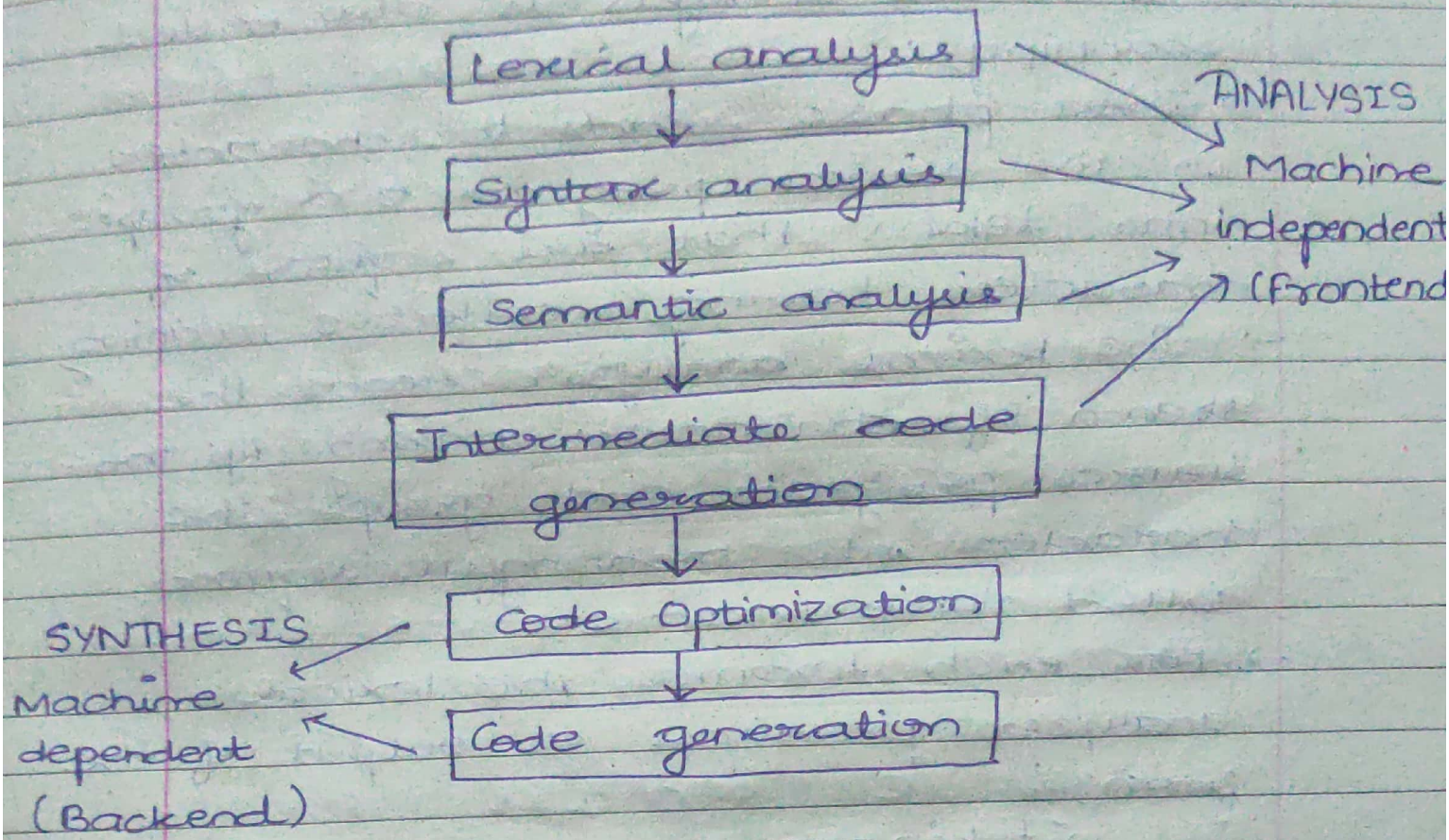
SYNTHESIS
Machine
dependent
(Backend)

Code

Code

There are mai
① Analysis
② Synthetic

* Structure of Compiler / Phases of Compiler



There are mainly 2 phases :

① Analysis

② Synthetic

Lexical Analysis :

→ The lexical analysis is also called scanning.

→ These phase reads the character in the source program and grouped into tokens that are sequence of characters having collective meaning.

→ The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequence called lexemes.

→ For each lexeme, the lexical analyzer produces as output of the form :

<token-name, attribute-value>

that it passes on to the suequent phase, syntax analysis

→ In the token, the first component token-name is an abstract symbol that is used during syntax analysis and the second component attribute-value points to an entry in symbol table for this token

for eg. suppose a source program contains assignment statement

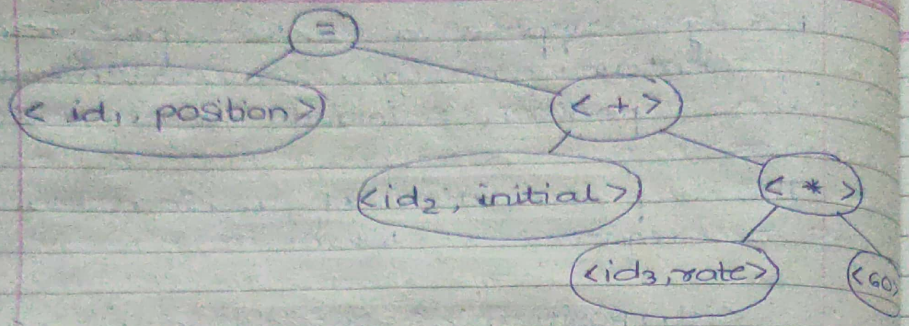
position = initial + rate * 60

The representation of lexical sequence as sequence of tokens as :

<id1, position> <=> <id2, initial> <+> <id3, rate> <*> <60>

Syntax Analysis :

→ The second phase of the compiler is syntax analysis or parsing.
→ The parser uses the first component of the token produced by lexical analyzer to create a tree like intermediate representation that depic grammatical structure of token stream
→ Typical representation is a syntax tree in which each interior node represents an operation and children of node represents the arguments of operation
→ The syntax tree for above stream is :



→ The tree has an interior node labeled with (id3, rate) as its left child and integer 60 as its right child

→ (id3, rate) represents the identifier rate

Semantic Analysis

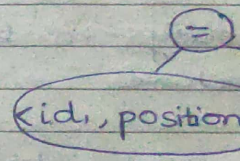
→ The semantic analyzer uses syntax tree and the information in the symbol table to check the source program for semantic consistency with language definitions

→ It also gathers type information and saves it in either syntax tree or the symbol table, for subsequent use during intermediate code generation

→ An important pa
is type checking
checks that ea
matching opera

→ For eg, many p
definition requir
be an integer
convert the int
point number

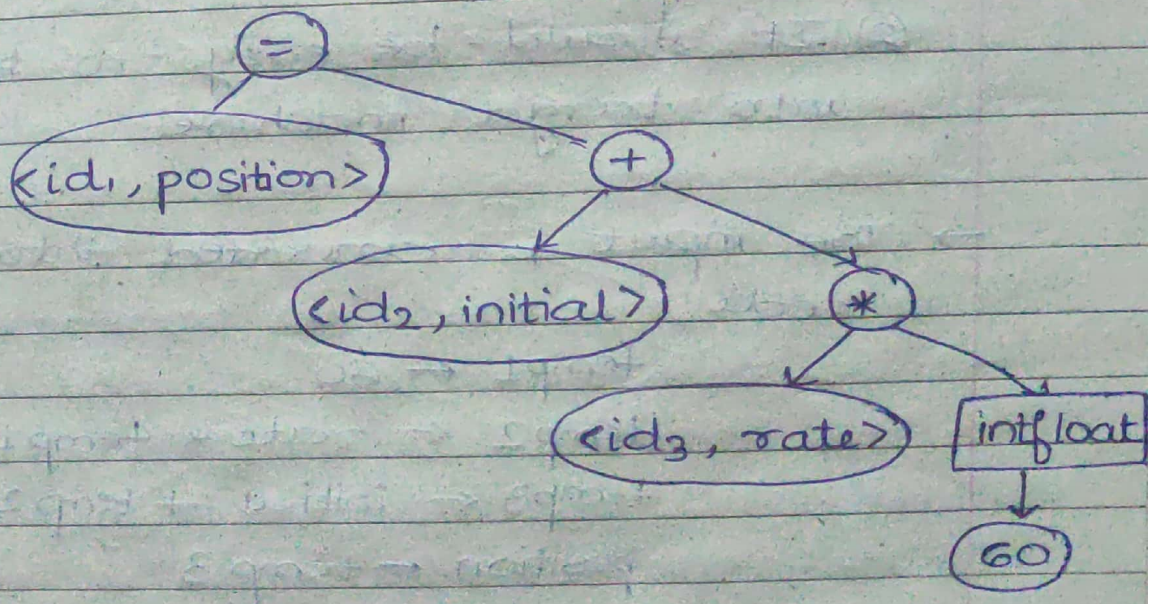
→ The output of
an extra node
which explicitly
argument into



→ An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.

→ For eg, many programming language definition require an array index to be an integer, the compiler may convert the integer into floating point number.

→ The output of semantic analyzer has an extra node for operator intfloat, which explicitly converts its integer argument into floating point number.



Intermediate code generation

→ In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms

→ Syntax tree are form of intermediate representation; they are commonly used during syntax and semantic analysis

→ The intermediate representation should have two important properties :

① It should be simple and easy to produce

② It should be easy to translate into target machine

→ The input is converted into 3 address code

```
temp1 ← 60
temp2 ← rate * temp1
temp3 ← initial + temp2
position ← temp3
```

Code Optimization

→ This phase attempts to improve the intermediate code so that better target code will result.

→ The objective of performing optimization are :
① faster execution
② shorter code
③ target code that consumes less powe

→ In our eg, optimized code is :

```
temp1 = rate * 60
position = initial + temp1
```

Code Generator

→ It takes an input as intermediate representation of source program and maps it into target language

→ If the target language is machine code, registers or memory location are selected for each of the variable used by program

→ Then the intermediate instruction are translated into sequence of machine instr that perform same tasks

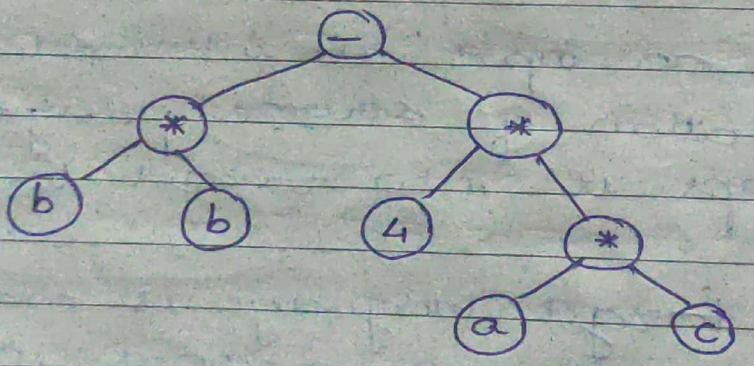
→ Code Generated is : MOV temp1, 60 in this way

```
LDF   R2, rate
MULF  R2, #60.0
LDF   R1, initial
ADDF  R1, R2
STF   position, R1
```

Que) b * b - 4 * a * c

lexical analysis :

<id1, b> < * > <id1, b> <-> <4> <*>
<id2, a> < * > <id3, c>



* Types of Compiler

① One Pass Compiler
② Two Pass Compiler
③ Multi Pass Compiler

• ONE - PASS COMPILER :
(Narrow compiler)

→ It reads the code only once and then translate it.
→ It processes the source code in one pass, meaning it goes through the entire source code only once to generate target code.
→ It is used where simple and small programs are present and where speed is crucial.
→ One pass compiler is fast since all compiler code is loaded in memory at once.
→ It can process the source text without overhead of OS having to shut down one process and start another.
→ All phases are executed at a time in main memory.
→ It is faster but it requires more memory.
→ Eg PASCAL

- TWO PASS COMPILER :

  Pass 1 : First four phases
  Pass 2 : Last two phases

- MULTI PASS COMPILER :
  (wide compiler)

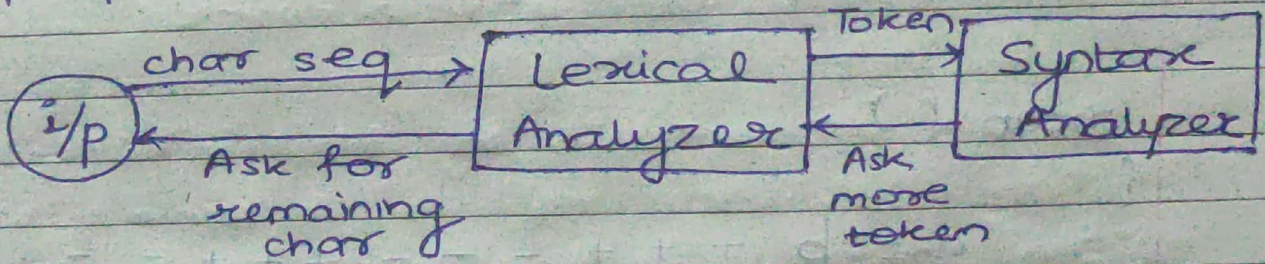
→ It can process the source code of a program multiple times.
→ In first pass, the compiler can read the source code, scan it, extract the tokens and save the result in an output file.
→ In second pass, the compiler can read the output file produced by first pass, build the syntatic tree and implement syntatical analysis. The output of this phase is a file that include syntactical tree
→ In the third pass, the compiler can read the output file produced by second pass and check that tree follows the rule of language or not. The output of semantic analysis phase is annotated tree syntax. This pass continues until target o/p is produced
→ Requires less memory

Lexical Analyzer

- The first job of lexical analysis is tokenization. There are different type of tokens :
  1. identifier — a, b, x, etc
  2. keywords — if, while, else, etc
  3. seprator — (,), {, etc
  4. special symbol — @, #, $, etc
  5. Operators — <, >, =, ==, etc
  6. Contants — 20, 30, etc

- The second job is is to show error message

- The third job is to remove white space, comments, macro, newline, preprocessor directives, etc



Input buffering [from classnotes and google pdf]

Compiler Construction tools (classnotes)

Lex and YACC → from chatgpt (watsapp)

* Specification of tokens

① unit minimum (int a , int b)
{
if (a < b)
return a ;
else
return b ;
}

② main ( )
{
a = b + + + - - - + + + = = ;
printf ("%d %d" , a , b) ;
}

main ( )
{
a = b + + + - - - + + + = = ;
printf ("%d %d" , a , b) ;
}

③ main) (
{
    int a = 10;
    char b = "abc";
    int c = 30;
    in/ * comment */ t m = 20;
{