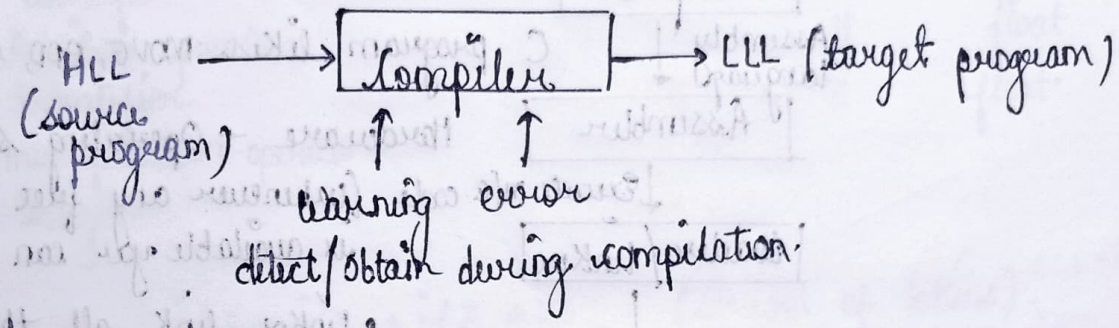
Unit-1
Introduction

Compiler (Translator)

Converts the high level language such as C, C++ etc into low level language (0,1).



Two types of compiler

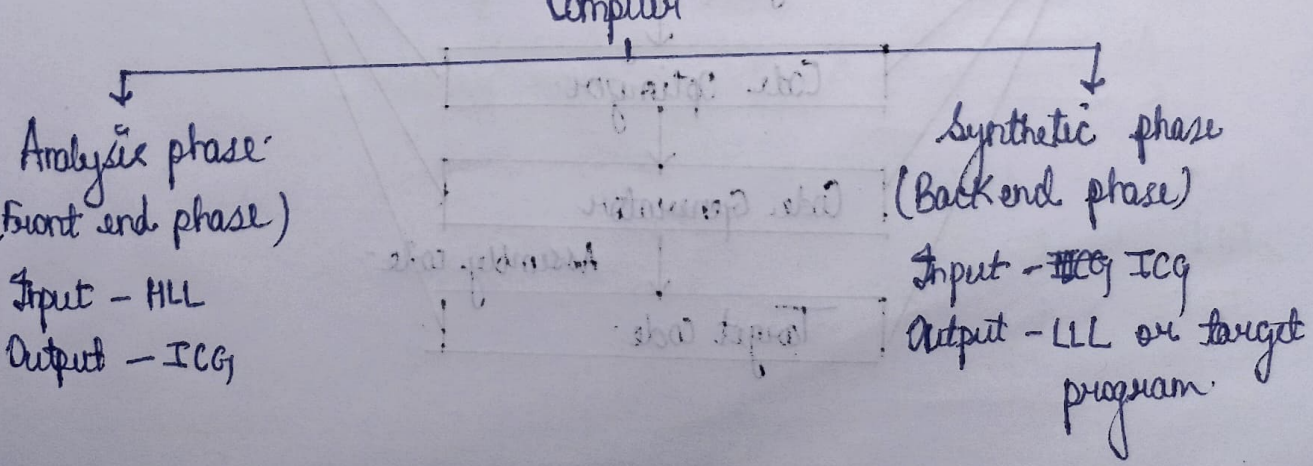
(1.) **Single pass compiler** : Compilation of phases from HLL to LLL occurs simultaneously which means all at a time.

(2.) **Multi Pass Compiler** : In multi pass compiler the compilation of phases is divided into two part front end & back end.

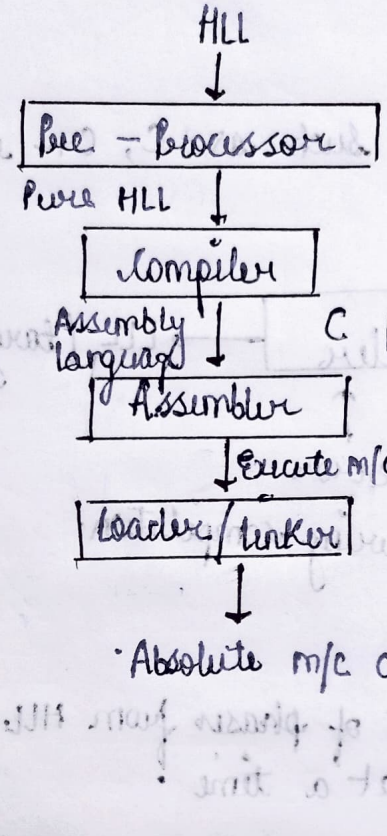
- Input to the front end or first pass is HLL & output is intermediate code generator (ICG).
- Input to the backend or second pass is ICG, & o/p is target program.

NOTE : All the compiler have the same till the ICG.

Compiler can divide into two phases:



* Explain the language processing system.



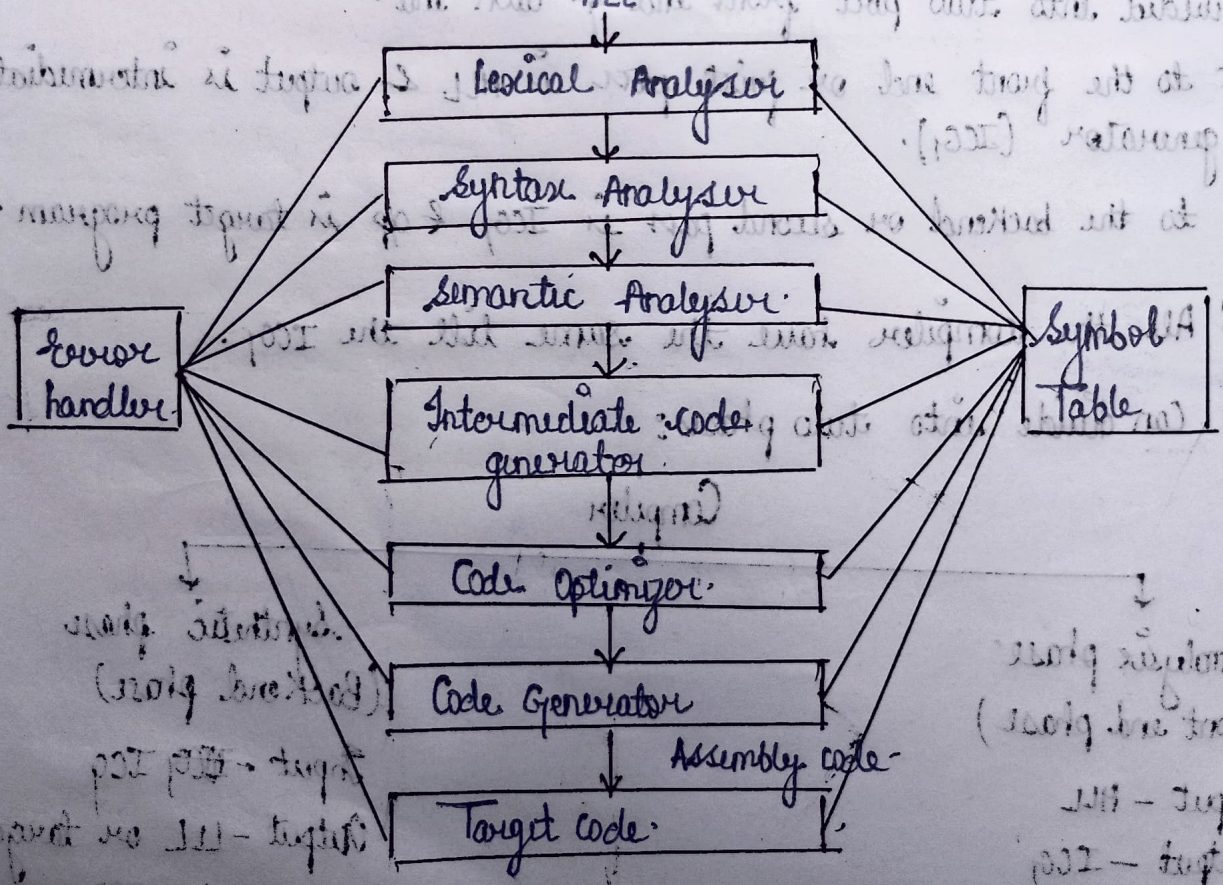
- Exclude the header files
- Include the require or necessary files.

C program like MOVE, ADD, STORE

Hardware + Operating system (whenever any free space / memory is available you can load it)

- Linker link all the different files that are req. for execution
- After linking loader will load all files for execution.

Q) Explain all 6 phases of compiler



Example

float x, y, Z
x = y + Z * 100.

for I) Phase [Lexical Analyser]

x - identifier [id, 1]
= Assignment operator
y identifier.
+ Addition operator
Z Identifier
* multiplication operator
100 Integer.

Symbol table

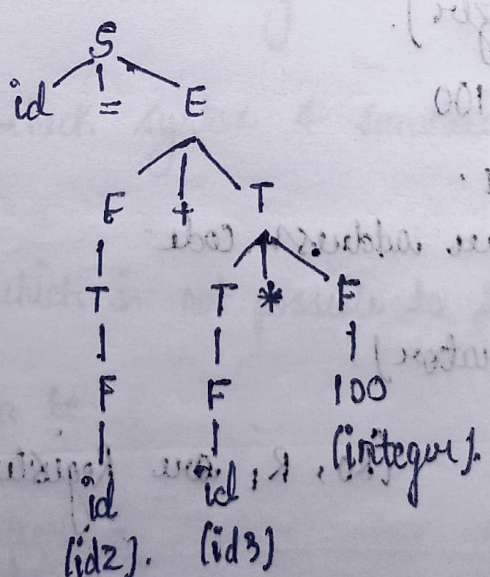
S.N.	VN	Type
1	x	float
2	y	float
3	Z	float

id1 = id2 + id3 * 100. [Stream of tokens].

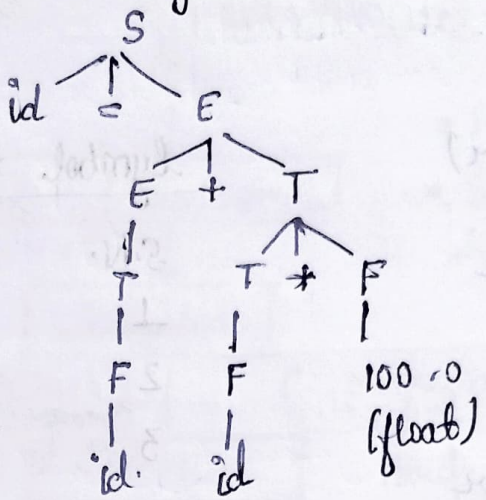
II Phase (Syntax Analyser).

Given context free grammar.

S → id = E
E → E + T / T
T → T * F / F
F → id / int



III Phase (Semantic Analyser)



IV Phase (Intermediate Code Generator)

Three address code.

minimum = 1

maximum = 3.

$x = y + z * 100$

$t_1 = z * 100$

$t_2 = t_1 + y$

$x = t_2$.

V Phase (code Optimizer).

$t_1 = z * 100$

$x = y + t_1$.

Optimizer three address code

VI Phase (code generator)

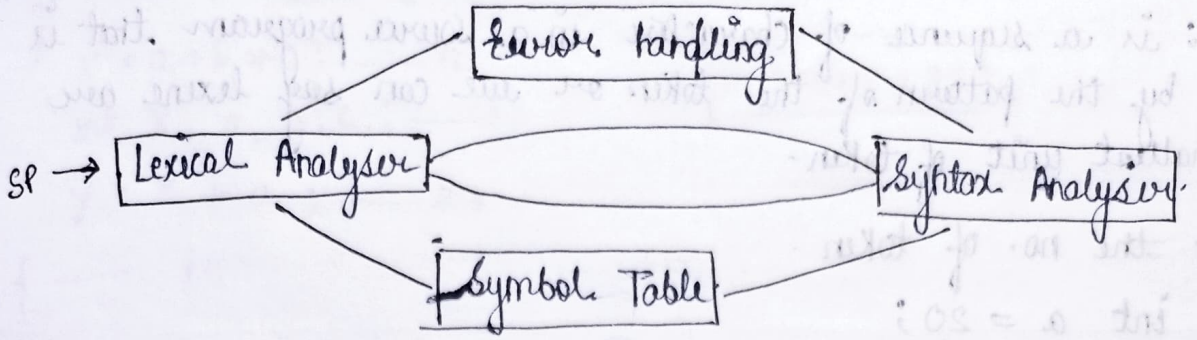
$R_0 \leftarrow z$

$R_1 \leftarrow y$,

(R0, R1 are Registers)

```
MUL R0, 100
ADD R1, R0
STORE x, R1
```

• Smallest unit of token is called lexin.



Lexical Analyser

1. It divides the given program into meaningful words which is known as token.
2. Tokens are normally Identifier, Keywords, Operators, constant & special symbol.
3. It eliminate comment line from the program.
4. It eliminate white space character for ex- blank, tab etc.
5. It helps in giving error message by providing row number & column number.
6. It uses DFA to do tokenization.
7. While doing tokenization lexical analyser always give important to longest matching.
8. Lexical analyser can't check syntax & semantic error.

Lexical error

A character sequence which is not possible to scan any valid token is a lexical error.

Lexical phase error can be
Spelling error.
Exceeding length of identifier.
Appiance of illegal character.

Lexical phase error can be found during the execution of the program.

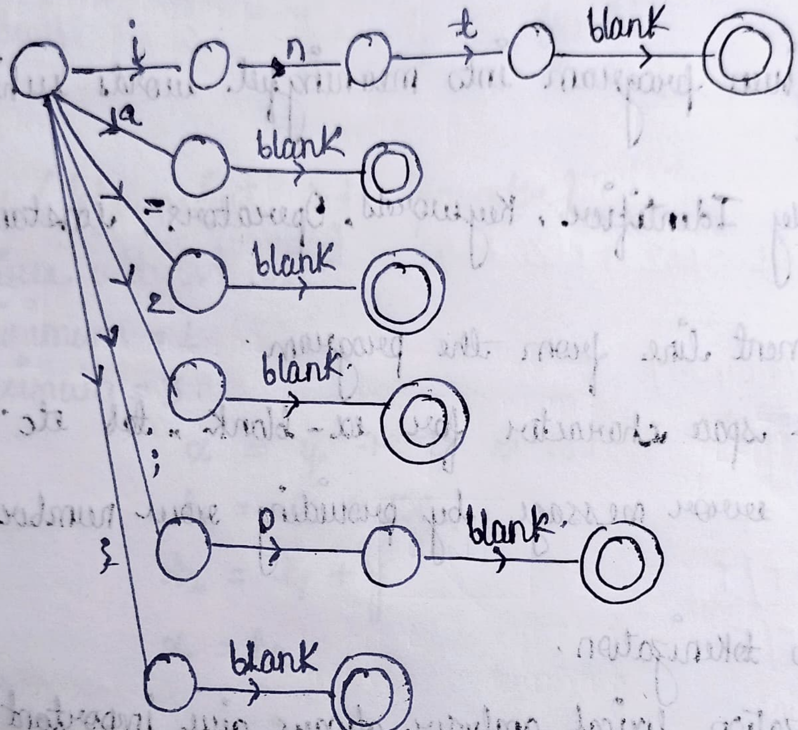
Token: A token is a sequence of character can be treated as a unique in the program of the programming language.

eg- of non token - Comments, Tab, blank, New line.

<u>Lexene</u>: is a sequence of character in a source program that is matched by the pattern of the token or we can say lexene are the smallest unit of token.

eg Count the no. of token.

int a = 20;



13/02/2024

(Q.) Count no. of tokens.

a)

```
int main ()
{
  int i;
  for (i=0; i<5; i++)
  {
    int i= 102;
    printf ("%d the value of i:", i);
    i++;
  }
  return 0;
}
```

#tokens = 42

b) main) (—— 3
—— 4
} x = a + b * c ; —— 12
int x, a, b, c. ; —— 21
y = x + a ; —— 27
{ —— 28

token = 28

c) main ( ) —— 3
—— 4
{ int . x = = 10 , y <= 10 ; —— 13
printf (" %d %d ", x); —— 20
} —— 21

token = 21

d) main ( ) —— 3
—— 4
{ char * c = " string " ; —— 10
float b = 100.74 ; —— 15
char d = 'e' ; —— 20
int f = 200 ; —— 25
int t f = 200 ; —— 31
in /* Comment */ t f = 200 ; —— 37
ch ar d = 'e' ; —— 43
Ch /* Comment */ ar d = 'e' ; —— 49
} —— 50

token = 50

e) int x = 4 ; /* comment */

token = 5

f) x = = = y /* abcd * * */ * abcd */ ;
1 2 3 4 5 6 7 8 9

token = 9

g) int * * p ;
1 2 3 4 5

tokens = 5

h) x = * p ++ ++ + y ;
1 2 3 4 5 6 7 8 9

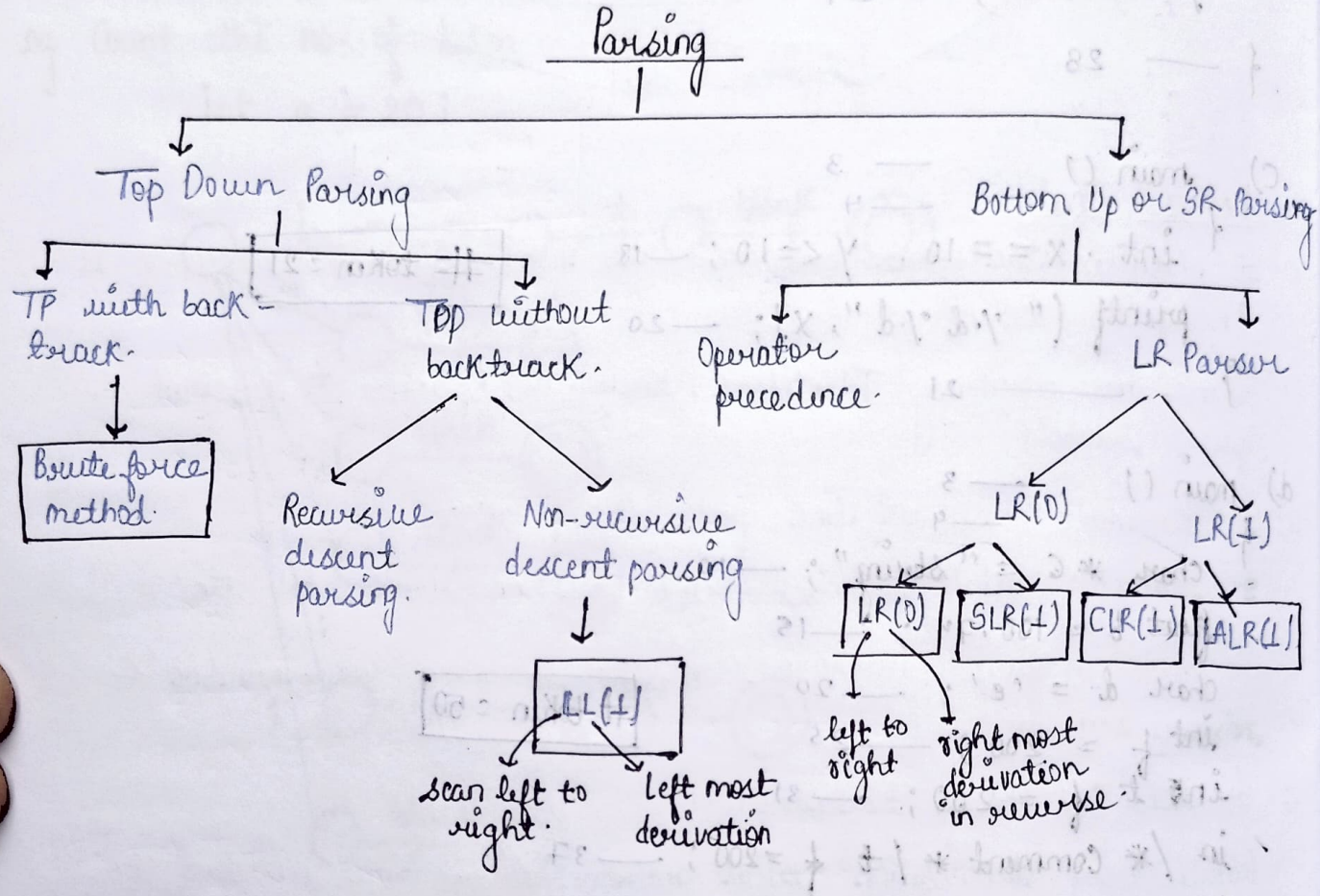
token = 9

i) char ch = " hello ; —— token error " closing quotes are not present.

14/02/2024

Syntax Analysis

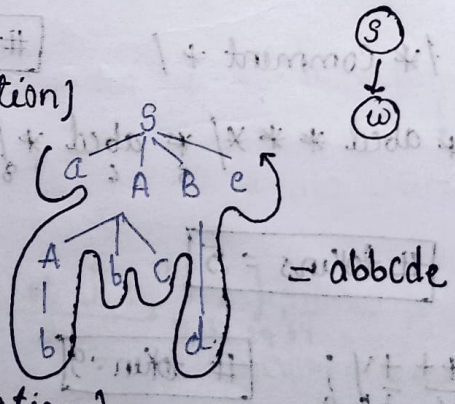
- CFG + Tokens → Parse Tree
- YACC tool used for parsing



(Q.) S → aABe
A → Abc / b
B → d
w = abbcde

Sol" Top down (left most derivation)

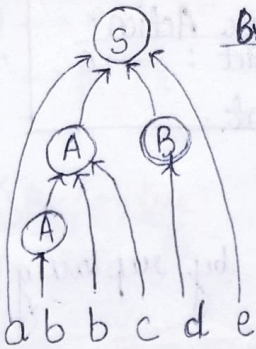
S → aABe
S → a Abc Be
S → a bbc Be
S → a bbc de.



Bottom up (right most derivation.)

S → aABe
S → aA de
S → a Abc de
S → a bbc de.

Bottom - up parsing



15/02/2024

LL Parsing

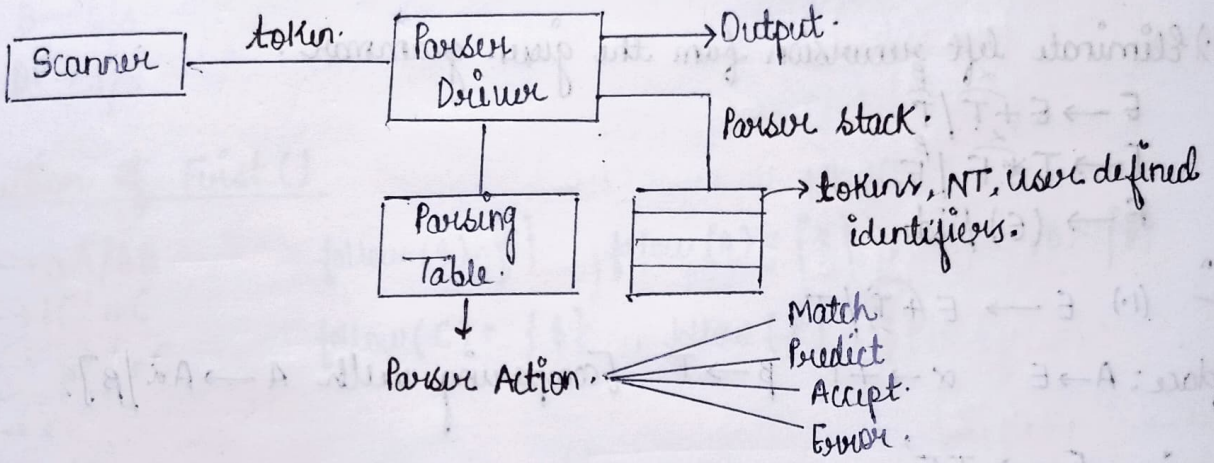


Fig. Architecture of LL parsing

1.) **Parser stack** :- holds grammar symbols.

2.) **Parsing Table** :- Specifies the parsing action & parsing action are as follows: Match, Predict, Accept, Error.

3.) **Parser Driver** :- This func^n interact with parsing table, parsing stack & scanner.

(Q.) $S \rightarrow (S)S/\epsilon$ Show top down parsing for string (()).

Sol^n

Parser stack	Input String	Parser action
$ S	(())$	Predict : S → (S)S
$)S(	(())$	Match : '('
$ S)S	())$	Predict : S → (S)S
$)S(S)S	))$	Match '('
$ S)S)S	))$	Predict : S → ϵ
$)S)S	))$	Match ')'
$ S)S	)$	Predict : S → ϵ
$)S	)$	Match : ')'

Parser Stack	Parser Input	Parser Action:
$S $	$	Predict : $S \to \epsilon$
$	$	Accept.

Elimination of left recursion

Left recursion can be eliminated from the grammar by replacing

$$A \to A\alpha / \beta.$$

$$\boxed{\begin{array}{l} A \to \beta A' \\ A' \to \alpha A' / \epsilon \end{array}}$$

Q.) Eliminate left recursion from the given grammar:

$$E \to E + T / T$$
$$T \to T * F / F$$
$$F \to (E) / id.$$

Sol^n (i.) $E \to E + T / T$.

replace: $A \to E \quad \alpha \to +T \quad \beta \to T$. [comparing with $A \to A\alpha / \beta$].

$\therefore E \to TE'$

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

(ii.) $T \to T * F / F$

$A \quad \alpha \quad \beta$

$\therefore T \to FT'$

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

$\therefore E \to TE'$

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

$T \to FT'$

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

$F \to (E) / id.$

Homework

Q.) $S \to Bb / a$ Find LR & remove it.

$B \to Bc / Sd / e$

$\alpha \quad \beta_1 \quad \beta_2$

Sol^n $\therefore S \to Bb / a$.

$B \to SdB' / eB'$

$B' \to cB' / \epsilon$.

16/02/2024

Left Factoring

Q.) If the grammar is in the form $A \to \alpha\beta_1 / \alpha\beta_2$ this can be eliminated by replacing in the form of

→ $A \longrightarrow \alpha A'$

$A' \longrightarrow \beta_1 / \beta_2$

(2.) $S \longrightarrow \underbrace{ab}_{\alpha} B c / \underbrace{ab}_{\alpha} D d$ (β_1, β_2)

$B \longrightarrow b / \epsilon$

$D \longrightarrow d / \epsilon$

Sol^n

$S \longrightarrow abS'$

$S' \longrightarrow Bc / Dd$

$B \longrightarrow b / \epsilon$

$D \longrightarrow d / \epsilon$

Construction of First ()

(1.) $S \longrightarrow aA / bB$

$A \longrightarrow iC / mC$

$B \longrightarrow pP / eC$

$C \longrightarrow c$

$P \longrightarrow q / f$

follow(S) follow(S) = {$} follow(A) = {$} , follow(B) = {$}

follow(C) = {$} , follow(P) = {$}

Sol^n first(S) = first(aA) ∪ first(bB) , first(A) = {i, m}

= {a} ∪ {b} first(B) = {p, e}

= {a, b} first(C) = {c}

first(P) = {q, f}.

(2.) $S \longrightarrow aABC$

$A \longrightarrow x$

$B \longrightarrow y$

$C \longrightarrow z$

follow(S) = {$}, follow(A) = {y}. , follow(B) = {z}.

Sol^n first(S) = {a}, first(A) = {x}, first(B) = {y}, first(C) = {z}.

follow(C) = {$}.

(3.) $S \longrightarrow ABC$

$A \longrightarrow x$

$B \longrightarrow y$

$C \longrightarrow z$

Sol^n First(A) = {x}, first(B) = {y}, first(C) = {z}.

follow(S) = {$} follow(A) = {y}, follow(B) = {z}, follow(C) = {$}

$first(S) = first(ABC) = first(A) = \{x\}$.

4.) $S \to ABC$
$A \to x/\epsilon$
$B \to y/\epsilon$
$C \to z$

Sol° $first(B) = \{y, \epsilon\}$, $first(C) = \{z\}$

$first(A) = first(x) \cup first(\epsilon)$
$= \{x, \epsilon\}$

or if $C \to z/\epsilon$ then
$\therefore first(S) = \{x, y, z, \epsilon\}$

$first(S) = first(ABC)$.
$= first(A) = \{x, \epsilon\}$
$= \{x, \epsilon\} - \{\epsilon\} \cup First(BC)$
$= \{x\} \cup \{y, \epsilon\} - \{\epsilon\} \cup first(C)$
$= \{x, y\} \cup \{z\}$
$first(S) = \{x, y, z\}$

5.) $S \to ABCD$
$A \to x/\epsilon$
$B \to y/\epsilon$
$C \to z/\epsilon$
$D \to d$.

Sol° $first(S) = first(A) - \{\epsilon\} \cup first(BCD)$
$= \{x, \epsilon\} - \{\epsilon\} \cup first(B) - \{\epsilon\} \cup first(CD)$
$= \{x\} \cup \{y, \epsilon\} - \{\epsilon\} \cup first(C) - \{\epsilon\} \cup first(D)$
$= \{x, y\} \cup \{z, \epsilon\} - \{\epsilon\} \cup \{d\}$
$first(S) = \{x, y, z, d\}$

$first(A) = \{x, \epsilon\}$, $first(yB) = \{y, \epsilon\}$, $first(C) = \{z, \epsilon\}$, $first(D) = \{d\}$

6.) $S \to ABCh$
$A \to x/\epsilon$
$B \to y/\epsilon$
$C \to z/\epsilon$

Sol° $first(S) = first(A) - \{\epsilon\} \cup first(BCh)$.

$= \{x, \epsilon\} - \{\epsilon\} \cup \{y, \epsilon\} + \{\epsilon\} \cup \{z, \epsilon\} - \{\epsilon\} \cup \{h\}$

$= \{x, y, z, h\}$

7) $E \rightarrow 10, * 7 / 5 + T$

$T \rightarrow PS$

$S \rightarrow QP / \epsilon$

$Q \rightarrow +, * / *$

$P \rightarrow a / b / c$

Sol^n first(P) = {a, b, c}.

first(Q) = {+, *}.

first(T) = first(PS)
= {a, b, c}.

first(E) = {10, 5}.

first(S) = first(QP) ∪ first(ϵ)
= {+, *, ϵ}

8.) $S \rightarrow ABC / ad$.

$A \rightarrow eS / Cr / \epsilon$.

$C \rightarrow f / P / \epsilon$.

Sol^n first(C) = {f, p, ϵ}.

first(A) = {e} ∪ first(Cr) ∪ {ϵ}
= {e} ∪ {f, p, ϵ} − {ϵ} ∪ {r} ∪ {ϵ}
= {e, f, p, r, ϵ}

first(S) = {a} ∪ {e, f, p, r, ϵ} − {ϵ}
∪ {bc}

first(S) = {a, b, e, f, p, r}

20/02/2024

Construction of follow() :-

(1.) $S' \rightarrow \perp S \#$

$S \rightarrow qABC$

$A \rightarrow a / bbD$

$B \rightarrow a / \epsilon$

$C \rightarrow b / \epsilon$

$D \rightarrow c / \epsilon$

Sol^n first(S') = {⊥}.

first(S) = {q}

first(A) = {a, b}.

first(B) = {a, ϵ}

first(C) = {b, ϵ}

first(D) = {c, ϵ}

follow(S') = {$}

follow(S) = {#}

follow(A) = first(BC).
= {a, ϵ} − {ϵ} ∪ follow first(C).
= {a} ∪ {b, ϵ} − {ϵ} ∪ follow(S)
= {a, b} ∪ {#}
= {a, b, #}

$S \Rightarrow ABC \rightarrow \epsilon$

follow(B) = first(C).
= {b, ϵ} − {ϵ} ∪ follow(S)
= {b, #}.

follow(C) = follow(S).
= {#}.

follow(D) = follow(A)
= {a, b, #}

(Q.) S → aABb
A → c/ϵ
B → d/ϵ

Sol° first(S) = a first(A) = {c, ϵ} first(B) = {d, ϵ}

follow(S) = {$}
follow(A) = first(Bb)
= {d, ϵ} − {ϵ} ∪ first(b)
= {d, b}.
follow(B) = {b}

(Q.) S → ABC
A → x/ϵ
B → y/ϵ
C → z.

Sol° follow(S) = {$}
follow(A) = first(BC)
= {y, ϵ} − {ϵ} ∪ first(C)
= {y} ∪ {z}.
= {y, z}.

follow(B) = first(C)
= {z}
follow(C) = follow(S).
= {$}.

(Q.) S → ABCh
A → x/ϵ
B → y/ϵ
C → z/ϵ

Sol° follow(S) = {$}
follow(A) = first(BCh)
= {y, ϵ} − {ϵ} ∪ first(Ch)
= {y} ∪ {z, ϵ} − {ϵ} ∪ first(h).
follow(A) = {y, z, h}
follow(B) = first(Ch)
= {z, ϵ} − {ϵ} ∪ first(h)
= {z, h}
follow(C) = first(h)
= {h}

(Q.) E → 10 * 7 / 5 + T
T → PS
S → QP / ϵ
Q → + / *
P → a / b / c

Sol^n follow(E) = {$} E → 5+T
follow(T) = follow(E) = {$}
follow(S) = follow(T) = {$}
follow(Q) = first(P) = {a, b, c}. S → QP
follow(P) = follow(S) ∪ first(S) T → PS
= {$} ∪ {10, 5}
= {5, 10, $}

(Q.) S → ABC / ad
A → eS / C / ϵ
C → f / g / ϵ

Sol^n follow(S) = {$}
follow(A) = first(BC)
= {b}
follow(C) = first(γ) = {γ}

(Q.) S → ABCD
A → x / ϵ
B → y / ϵ
C → z / ϵ
D → d

Sol^n follow(S) = {$}, follow(D) = follow(S) = {$}.
follow(A) = {y, ϵ} − {ϵ} ∪ {z, ϵ} − {ϵ} ∪ {d}
= {y, z, d}
follow(B) = {z, ϵ} − {ϵ} ∪ {d} = {z, d}
follow(C) = {d}

21/02/2024

(Q.) Modify the following CFG so as to make it suitable for top down parsing construct LL(1) parser for the modified grammar. Show moves made by this LL(1) parser on input (w) = id + id * id.

$E \longrightarrow E+T \mid T$

$T \longrightarrow T*F \mid F$

$F \longrightarrow (E) \mid id.$

Sol' **Step 1:** bcoz there is a left recursion in the given grammar so we eliminate the left recursion first.

Left recursion formula.

$A \longrightarrow A\alpha \mid \beta$

$A \longrightarrow \beta A'$

$A' \longrightarrow \alpha A' \mid \epsilon$

$\alpha = +T, \ \beta = T.$

$\alpha = *F, \ \beta = F.$

$\therefore E \longrightarrow TE'$

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

$T \longrightarrow FT'$

$T' \longrightarrow *FT' \mid \epsilon.$

$F \longrightarrow (E) \mid id.$

Step 2: Calculate first() & follow() of this modified grammar.

first(E) = first(TE')
= first(F)
= {(, id}.

first(E') = {+, ε}

first(T) = first(F)
= {(, id}.

first(T') = {*, ε}

first(F) = {(, id}.

follow(E) = {$,)}

follow(E') = follow(E) ∪ follow(E')
= {$,)}

follow(T) = first(E') − {ε} ∪ follow(E') ∪ follow(E)
= {+, $,)}

follow(T') = follow(T) ∪ follow(T')
= {+, $,)}

follow(F) = first(T') − {ε} ∪ follow(T) ∪ {*, ε} − {ε}
= {*, +, $,)}

Step-3

Parsing Table

NT/T	+	*	(	)	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			$F \to (E)$		$F \to id$	

∴ The given grammar is LL(1) grammar because there is no multiple entry in a cell/table.

Stack	Input String	Moves
$E	id + id * id $	$E \to TE'$
$E'T	id + id * id $	$T \to FT'$
$E'T'F	id + id * id $	$F \to id$
$E'T'id	id + id * id $	
$E'T'	+ id * id $	$T' \to \epsilon$
$E'	+ id * id $	$E' \to +TE'$
$E'T+	+ id * id $	
$E'T	id * id $	$T \to FT'$
$E'T'F	id * id $	$F \to id$
$E'T'id	id * id $	
$E'T'	* id $	$T' \to *FT'$
$E'T'F*	* id $	
$E'T'F	id $	$F \to id$
$E'T'id	id $	
$E'T'	$	$T' \to \epsilon$
$E'	$	$E' \to \epsilon$
$	$	Accept

(Q.) $S \to aABb$ Check LL(1) or not. & parse string $w = ab$.

$A \to c / \epsilon$

$B \to d / \epsilon$.

Sol^n

first (S) = a.

first (A) = {c, ϵ}

first (B) = {d, ϵ}.

follow (S) = $

follow (A) = first (Bb)

= {d, ϵ} − {ϵ} ∪ {b}

= {d, b}

follow (B) = {b}.

NT/T	a	b	c	d	$
S	S → aABb				
A		A → ϵ	A → c	A → ϵ	
B		B → ϵ		B → d	

Stack	Input string	Moves
$S	ab $	S → aABb
$bBAa	ab$	
$bBA	b$	A → ϵ
$bB	b$	B → ϵ
$b	b$	
$	$	Accept

26/02/2024

(Q.) $S \to a / \uparrow / (T)$

$T \to T, S / S$

$w = (a, (\uparrow), a)$.

Sol^n ~~first (S) = {a, ↑, (}~~

first Step 1 : Remove left recursion

$A \to A\alpha / \beta$

$A \to \beta A'$

$A' \to \alpha A' / \epsilon$

$T \to ST'$

$T' \to ,ST' / \epsilon$.

$S \rightarrow a / \uparrow / (T)$

Step 2 : Calculate first ( ) & follow( ).

first (S) = {a, ↑, (}

first (T) = first (S) = {a, ↑, (}

first (T') = {, , ϵ}

$T \rightarrow ST'$ $\quad T' \rightarrow ,ST'/\epsilon$

follow (S) = first (T') ∪ first (T')

= {, , ϵ} − {ϵ} ∪ follow (T) ∪ {, , ϵ} − {ϵ} ∪ follow (T')

= {, } ∪ {)} ∪ {, } ∪ follow (T).

= {,) } ∪ {)}

= {, ,), $}.

follow (T) = {)}

follow (T') = {)}

Step 3 : Parsing Table.

NT/T:	a	↑	(	)	$	,
S	$S \rightarrow a$	$S \rightarrow \uparrow$	$S \rightarrow (T)$			
T	$T \rightarrow ST'$	$T \rightarrow ST'$	$T \rightarrow ST'$			
T'				$T \rightarrow \epsilon$		$T \rightarrow ,ST'$

Stack	Input string	Moves.
$S	(a, (↑), a) $	$S \rightarrow (T)$.
$)T(	(a, (↑), a) $	
$)T	a, (↑), a) $	$T \rightarrow ST'$
$)T'S	a, (↑), a) $	$S \rightarrow a$.
$)T'a	a, (↑), a) $	
$T'	, (↑), a) $	$T \rightarrow ,ST'$
$T'S,	, (↑), a) $	

Stack	Input String	Moves
$T'S	(↑),a)$	S → (T)
$T')T(	(↑),a)$	
$T')T	↑),a)$	T → ST'
$T')T'S	↑),a)$	S → ↑
$T')T'↑	↑),a)$	
$T')T'	),a)$	T' → ∈
$T')	),a$	
$T'	,a$	T → ,ST'
$T'S,	,a$	
$T'S	a$	S → a
$T'a	a$	
$T'	$	T' → ∈
$	$	Accept.

(Q.) S → A.
A → aB / aC / aD / Ac
B → bBC / f
C → g / ∈
D → d / ∈

Sol' Step 1 : Remove left factoring & left recursion.

(i) A → aB / aC / aD / Ac

↓

A → Ac / aA'
A' → B / C / D.

∴ S → A.
A → Ac / aA'
A' → B / C / D.
B → bBC / f.
C → g / ∈.
D → d / ∈.

A → aα / aβ / aγ / δ

A → δ / aA'
A' → α / β / γ

(ii) Remove left recursion.

$S \to A$
$A \to aA'A''$
$A'' \to cA''/\epsilon$
$B \to bB/c/f$

X

$S \to A$
$A \to aA'A''$
$A'' \to cA''/\epsilon$
$A' \to B/C/D$
$B \to bBC/f$
$C \to g/\epsilon$
$D \to d/\epsilon$

Step 2 :

first(S) = first(A) = {a}
first(A) = {a}
first(A'') = {c, ϵ}
first(A') = {b, f, g, d, ϵ}
first(B) = {b, f}
first(C) = {g, ϵ}
first(D) = {d, ϵ}

follow(S) = {$}
follow(A) = {$} ∪
follow(A'') = {$} ∪
follow(A') = {c, ϵ} − {ϵ} ∪ follow(A).
= {c, $}.
follow(B) = follow(A') ∪ first(C)
= {c, $} ∪ {g, ϵ} − {ϵ} ∪ follow(B)
= {c, $, g} ∪

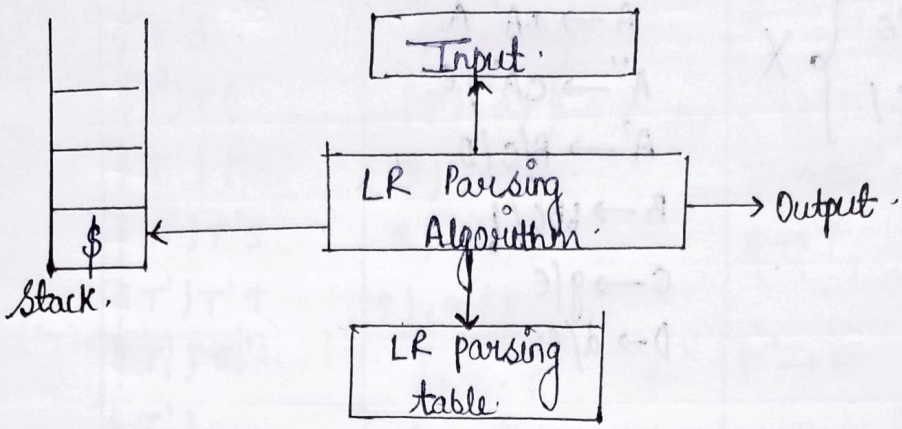
follow(C) = follow(A') ∪ follow(B)
= {c, $} ∪ {c, $, g}
= {c, $, g}

follow(D) = follow(A')
= {c, $}.

NT/T	a	b	c	d	f	g	$
S	S→A						
A	A→aA'A''						
A'	1	A'→B	A'→C A'→D	A'→D.	A'→B	A'→C	A'→C A'→D.
A''			A''→cA''				A''→ϵ.
B		B→bBC			B→f		
C				ϵ→c		c→g, C→ϵ	C→ϵ.
D			D→ϵ	D→d			D→ϵ

∴ Given grammar is not LL(1) gram. bcoz parsing table have more than 1 entries in a cell.

LR Parser Structure



(Q.) $S \to CC$
$C \to cC$
$C \to d$

(i) $w = CdCd$
(ii) $w = dddc$

Sol° (i)

Stack	Input String	Action
$	cdcd$	Shift c
$c	dcd$	Shift d
$cd	cd$	Reduce $C \to d$
$cC	cd$	~~Shift c~~ Reduce $C \to cC$
~~$cCc~~	~~d$~~	Rec
$cC	cd$	Shift c
$cCc	d$	Shift d
$cCcd	$	Reduce $C \to d$
$cCcC	$	Reduce $C \to cC$
$CC	$	Reduce $S \to CC$
$S	$	Accept

(ii)

Stack	Input string	Action
$	dddc$	Shift d
$d	ddc$	Reduce $C \to d$
$C	ddc$	Shift d
$Cd	dc$	Red $C \to d$
$CC	dc$	Red $S \to CC$
$S	dc$	Shift d

Stack	Input string	Action
$ Sd	c $	Reduce C → d.
$ SC	c $	Shift c
$ SCc	$	Not Accepted.

27/02/2024

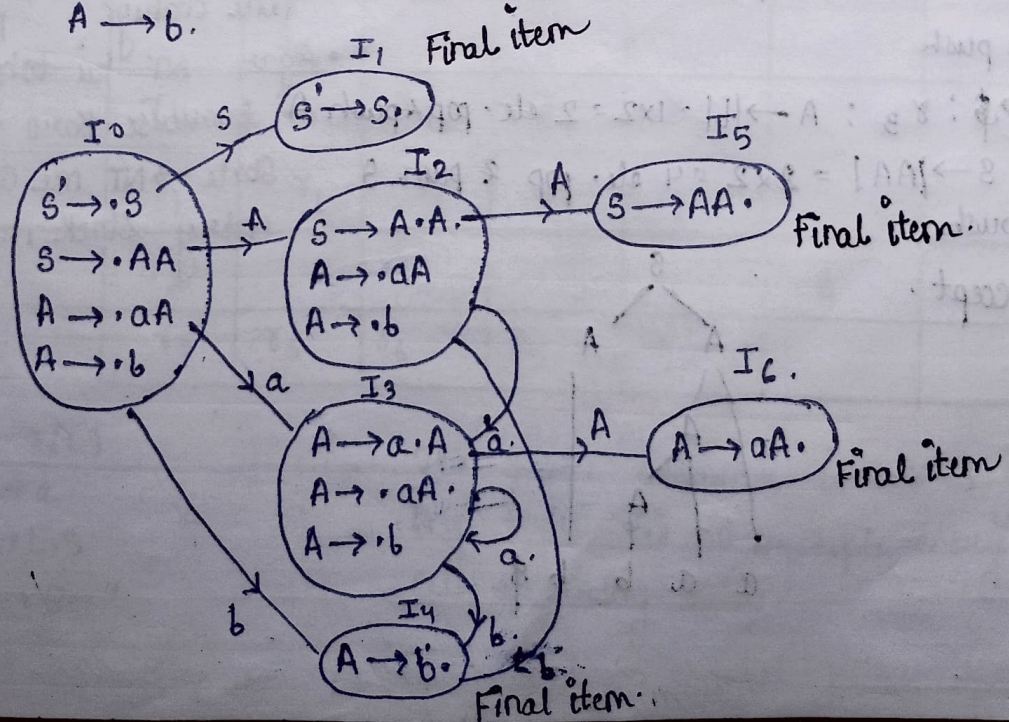
LR(0) Parser

- LR(0) parsing table construction algorithm
  1.) Find augmented grammar
  2.) I_0 = closure (augmented LR(0) items)
  3.) Apply closure and goto function and find all collection of LR(0) items using finite automata / DFA.
  4.) Reduce DFA = LR(0) parsing table

Q.) S → AA — 1
A → aA — 2
A → b — 3
Construct LR(0) parser.
w = aabb

Sol°
1.) Augmented grammar
S' → S
S → AA
A → aA
A → b.

2.)



State	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Since there is no multiple entry in any cell
∴ It is LR(0) grammar.

row
r_1 : $S \to AA$. I_5
r_2 : $A \to aA$. I_6
r_3 : $A \to b$. I_4.

$w = a\ a\ b\ b\ \$$ (yahan last b shift hoke ↑ $ me chala gya.)

Stack

0	a	3	a	3	b	4	A	6	A	6	A	2	b	4	A	5	S	1	Accept

Initial state by default.

$4 \to b : r_3 : A \to |b| = 1 \times 2 = 2$ = 2 element pop (reduce) & push A (by default 2 se multiply hota hai)

no. of variable in RHS of prodⁿ.

$3 \to A : 6$. : push 6.

$6 \to b : r_2 : A \to |aA| = 2 \times 2 = 4$ element pop & push A.

$3 \to A : 6$: push.

$6 \to b : r_2 : A \to |aA| = 4$ pop.

$0 \to A : 2$: push

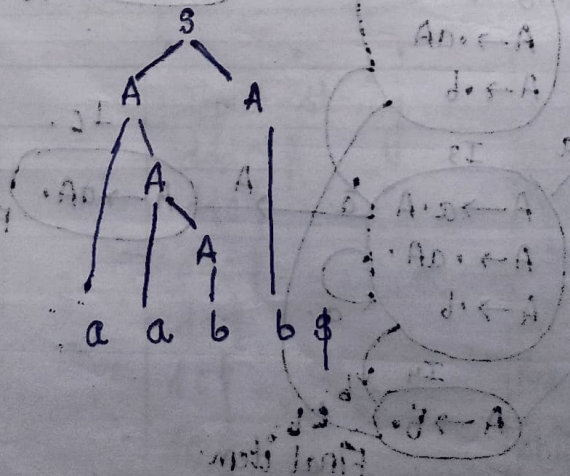
$2 \to b : 4 \to \$: r_3 : A \to |b| = 1 \times 2 = 2$ ele. pop & push A

$5 \to \$: r_1 : S \to |AA| = 2 \times 2 = 4$ ele. pop & push S

$0 \to S : 1$ push

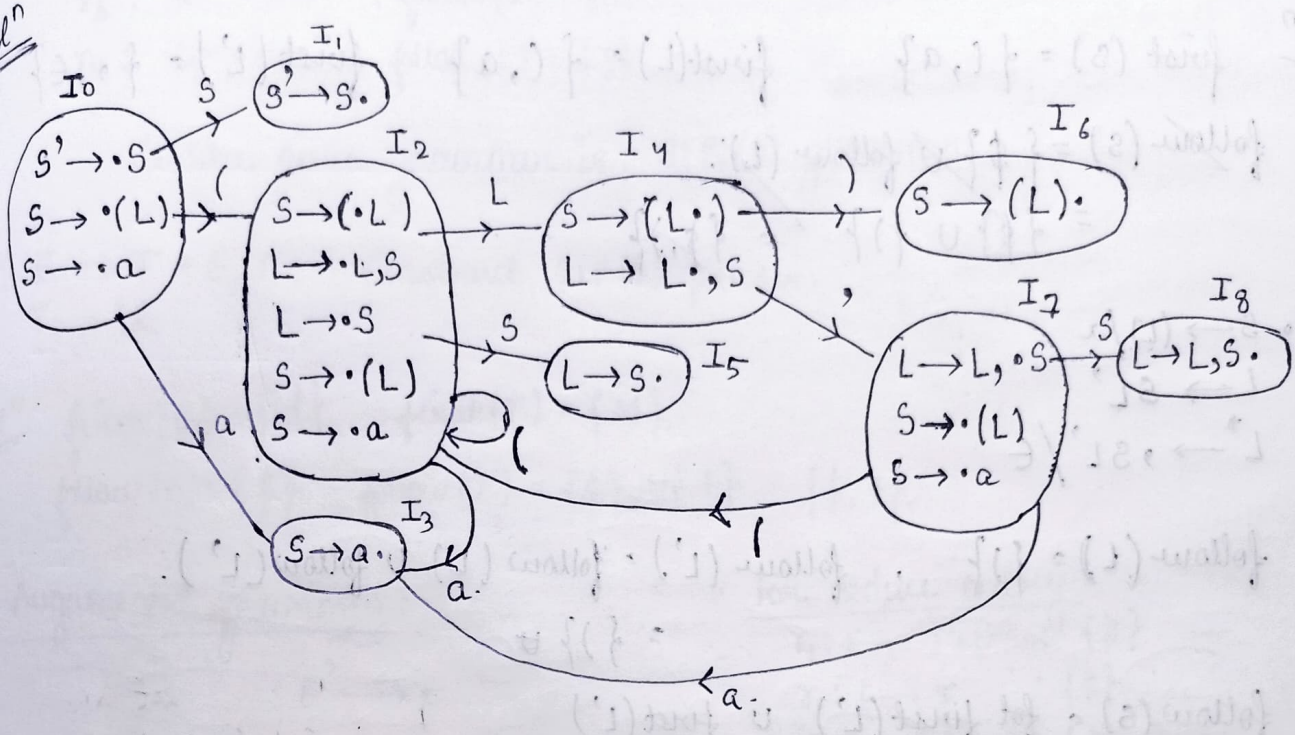
$1 \to \$$: Accept.

- Stack me hmesha state (no.) hona chahiye.
- Agar NT hai toh left traverse karo then State → NT me Goto ki entry stack me fill karo



(Q.) $S \to (L) / a$. Check the following grammar is LR(0) or not.
$L \to L, S / S$

Soln



State	Action					Goto	
	(	)	a	,	$	S	L
0	S_2		S_3			1	
1					Accept		
2	S_2		S_3			5	4
3	r_2	r_2	r_2	r_2	r_2		
4		S_6		S_7			
5	r_4	r_4	r_4	r_4	r_4		
6	r_1	r_1	r_1	r_1	r_1		
7	S_2		S_3			8	
8	r_3	r_3	r_3	r_3	r_3		

1. $S \to (L)$
2. $S \to a$
3. $L \to L,S$
4. $L \to S$

∴ The given grammar is LR(0) grammar bcoz there is no multiple entry in a cells of parsing table.

SLR Parser

(Q.) $S \rightarrow (L) / a$ Construct SLR(1) parser
$L \rightarrow L, S / S$

Sol^n first(S) = {(, a} first(L) = {(, a} first(L') = {, , ∈}

follow(S) = {$} ∪ follow(L).
= {$} ∪ {)} = {$,)}

∴ $S \rightarrow (L) / a$
$L \rightarrow SL'$
$L' \rightarrow , SL' / \epsilon$

follow(L) = {)} , follow(L') = follow(L) ∪ follow(L').
= {)}

follow(S) = first(L') ∪ first(L')
= {, , ∈} − {∈} ∪ follow(L) ∪ {, , ∈} − {∈} ∪ follow(L')
= {, ,)} ∪ {, ,)}.
= {, , $,)}

	Action					Goto	
	a	(	)	,	$	S	L
0	S_3	S_2				1	
1					Accept		
2	S_3	S_2				5	4
3			r_2	r_2	r_2		
4			S_6	S_7			
5			r_4				
6			r_1	r_1	r_1		
7	S_3	S_2				8	
8			r_3				

For reduce entry: only on terminals & present in follow().

$r_1 : S \to (L)\bullet$: reduce if follow(S) = {$, , ,)}

$r_2 : S \to a\bullet$: follow(S) = {$, , ,)}

$r_3 : L \to L,S\bullet$: follow(L) = {)}

$r_4 : L \to S\bullet$: follow(L) = {)}

∴ The given grammar is SLR(0) grammar

(Q) $E \to T+E/T$ Construct SLR(1) parser.
$T \to id$

Sol" first(E) = {id} first(T) = {id}.

follow(E) = {$} follow(T) = {$}. ∪ {+} = {$, +}.

Augmented grammar:

$E' \to E$

$E \to T+E/T$

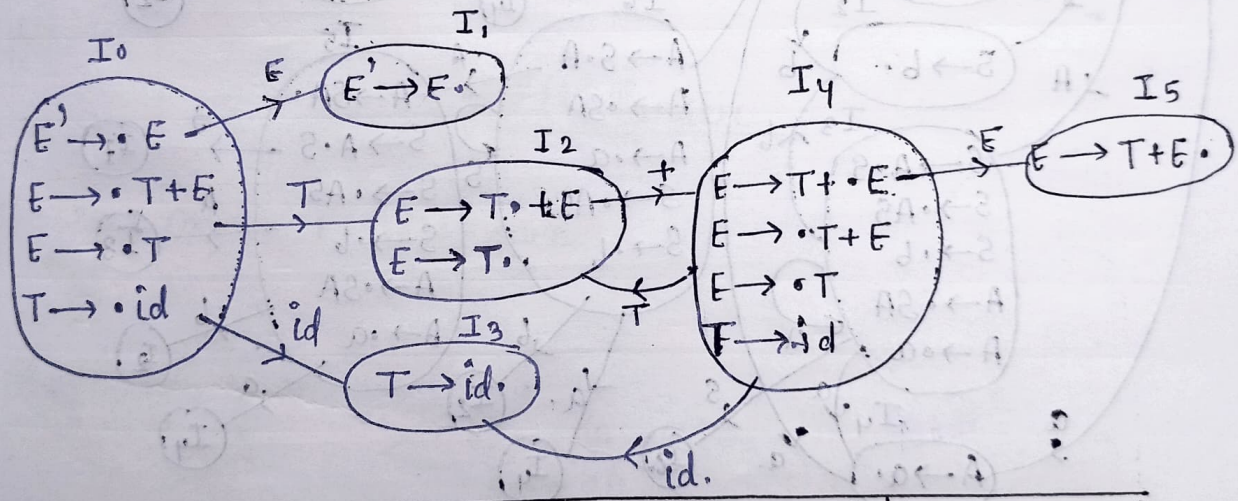
$T \to id$

For reduce entry

$r_1 : E \to T+E$: {$}

$r_2 : E \to T$: {$}

$r_3 : T \to id$: {$, +}



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

∴ The given grammar is SLR(1) parser but not LR(0) grammar.

(Q.) Consider the following grammar:

$S \longrightarrow AS/b$.

$A \longrightarrow SA/a$.

Construct SLR parse table.

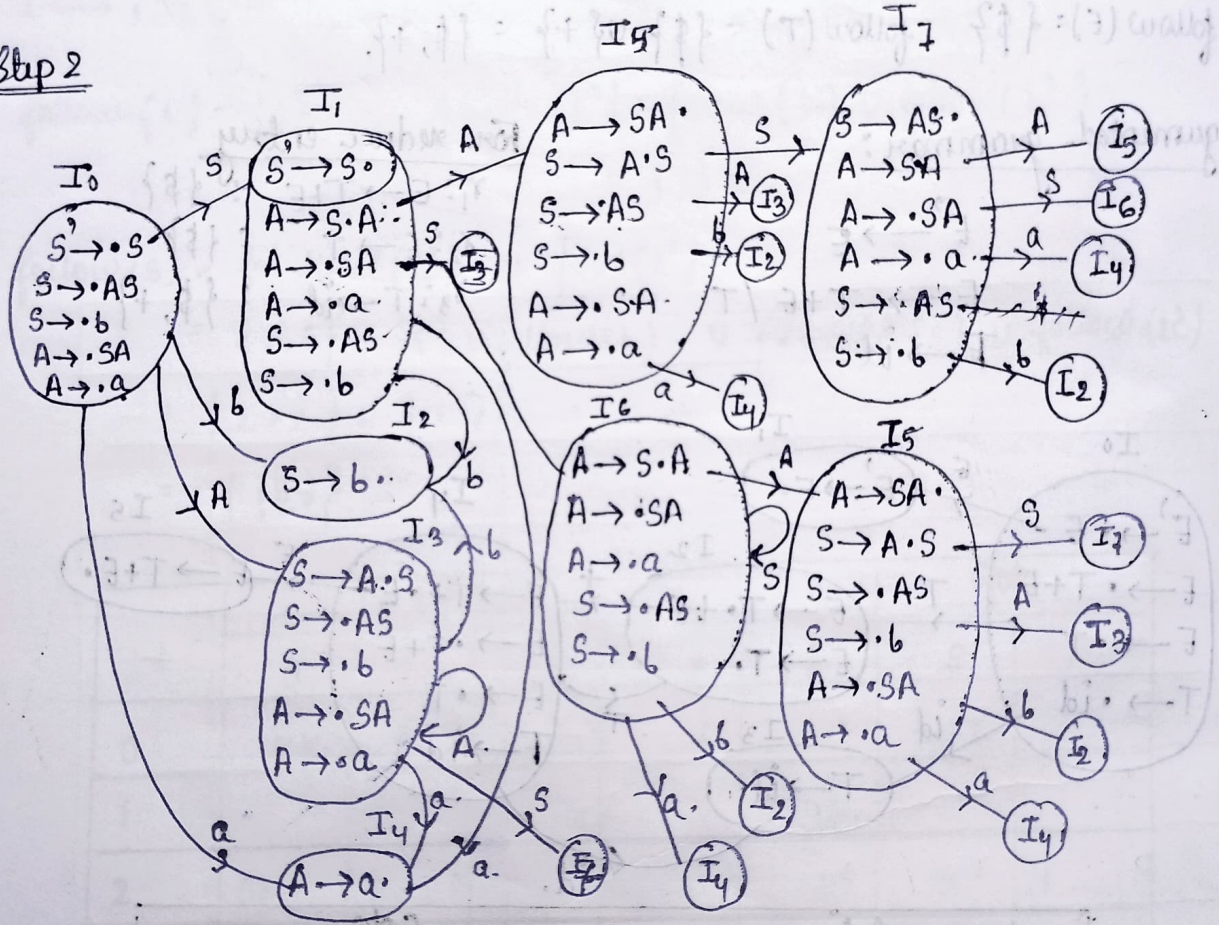
Sol" **Step 1** : Augment the given grammar.

$S' \longrightarrow S$

$S \longrightarrow AS/b$

$A \longrightarrow SA/a$.

Step 2



	Action			Goto	
	a	b	$	A	S
I_0	S_4	S_2		3	1
I_1	S_4	S_2	Accept	5	6
I_2	r_2	r_2	r_2		
I_3	S_4	S_2		3	7
I_4	r_4	r_4			
I_5	S_4/r_3	S_2/r_3		3	7

	Action			Goto	
	a	b	$	A	S
I_6	S_4	S_2		5	6
I_7	S_4/r_1	S_2/r_1	r_1	5	6

First(S) = First(A) ∪ {b}
= {a, b}

First(A) = First(S) ∪ {a}
= {a, b}.

Follow(S) = {$} ∪ First(S) ∪ follow(S)
= {$, a, b}.

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

r_1 : S → AS.
r_2 : S → b.
r_3 : A → SA.
r_4 : A → a.

∴ The given grammar is not SLR grammar because there is Shift-reduce conflict.

01/03/2024

CLR(1) Parser

(Q) S → AA. Construct CLR(1) parser.
A → aA.
A → b.

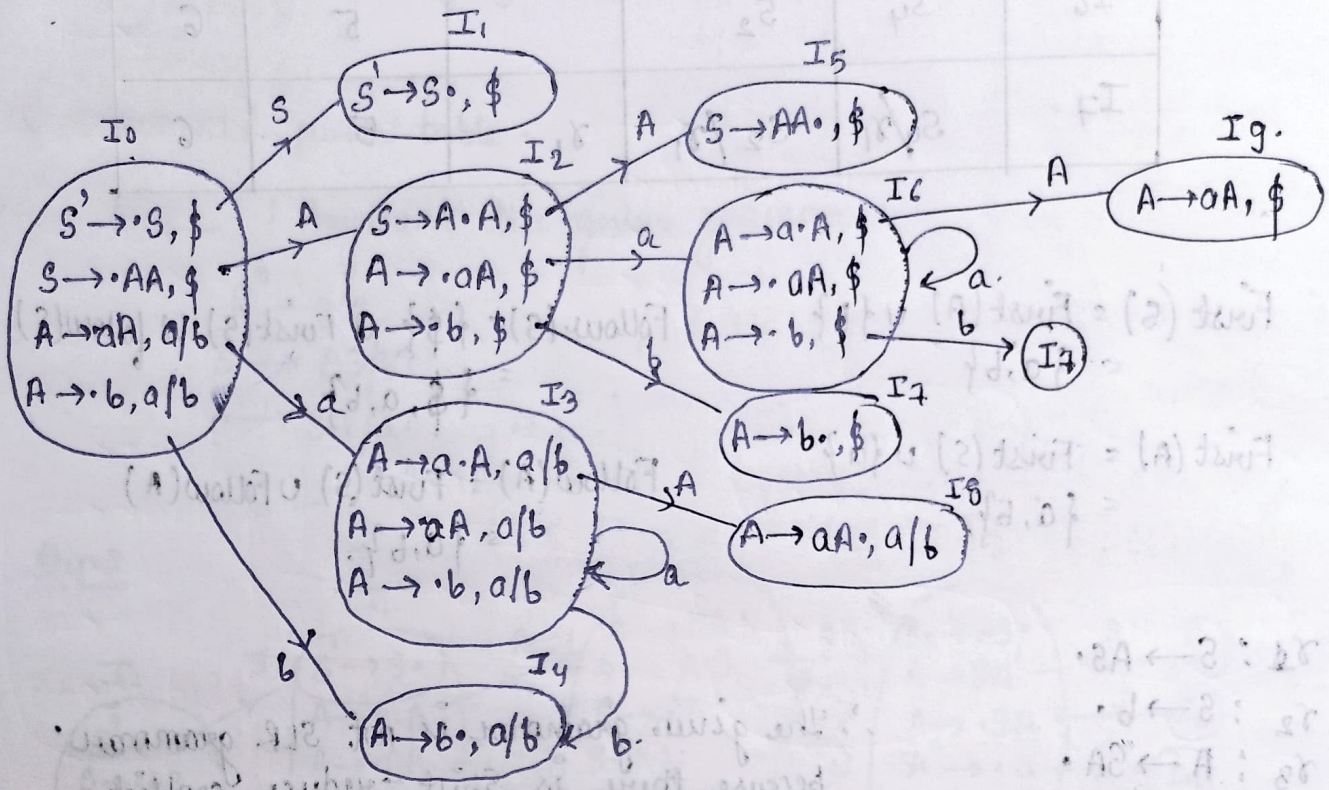
Sol^n Augmented Grammar : S' → S
S → AA
A → aA.
A → b.

LR(1) = LR(0) + LookAhead.

I_0

S' → .S, $
S → .AA, $
A → .aA, a/b
A → .b, a/b.

First(A, $) = {a, b}



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

For reduce entry: ~~only~~ reduce entry always under the look ahead of reduced grammar.

r_1 : $S' \rightarrow AA$ ·

r_2 : $A \rightarrow aA$ ·

r_3 : $A \rightarrow b$ ·

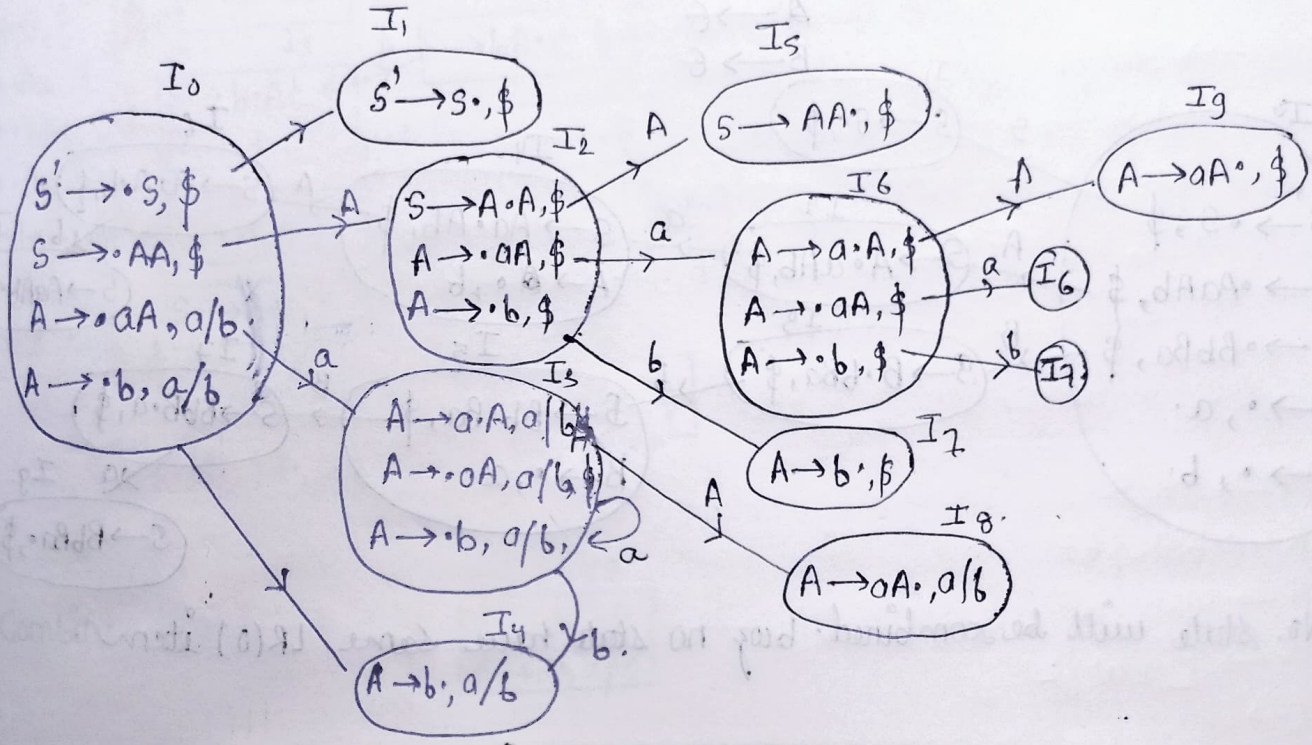
∴ The given grammar is CLR(1) parser, bcoz there no multiple entry in a cell of parse table.

04/03/2024

LALR(1) Parser : minimised form of CLR(1).

Q.) $S \to AA$
$A \to aA$
$A \to b$ Construct LALR(1) parser

Sol^n



Now,

I_3, I_6 → LR(0) item same, diff. lookahead
I_4, I_7 → —"— —"— , —"—
$I_8; I_9$ → —"— —"— , —"—

So we will combine I_{36}, I_{47} & I_{89}

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

(Q.) $S \to AaAb \mid BbBa$ Construct LALR(1).

$A \to \epsilon$

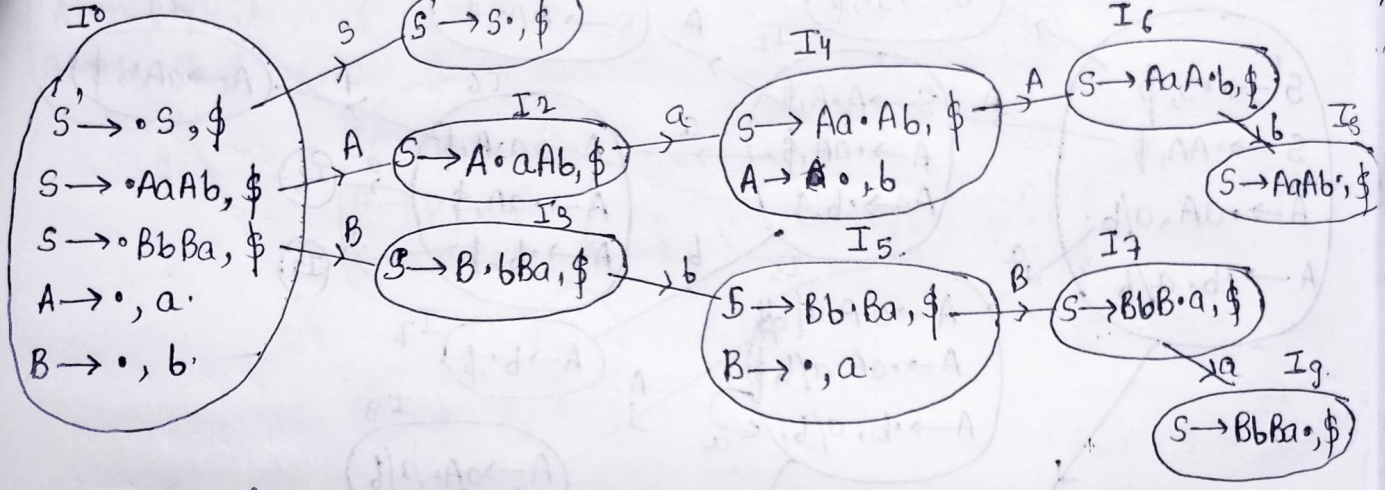
$B \to \epsilon$

Sol Augmented grammar: $S' \to S$

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$



→ No, state will be combined. bcoz no state have same LR(0) item.

	Action			Goto		
	a	b	$	S	A	B
0	r_3	r_4		1	2	3
1			Accept			
2	S_4					
3		S_5				
4		r_3			6	
5	r_4					7
6		S_8				
7	S_9					
8			r_1			
9			r_2			

$S \to AaAb : r_1$

$S \to BbBa : r_2$

$A \to \epsilon : r_3$

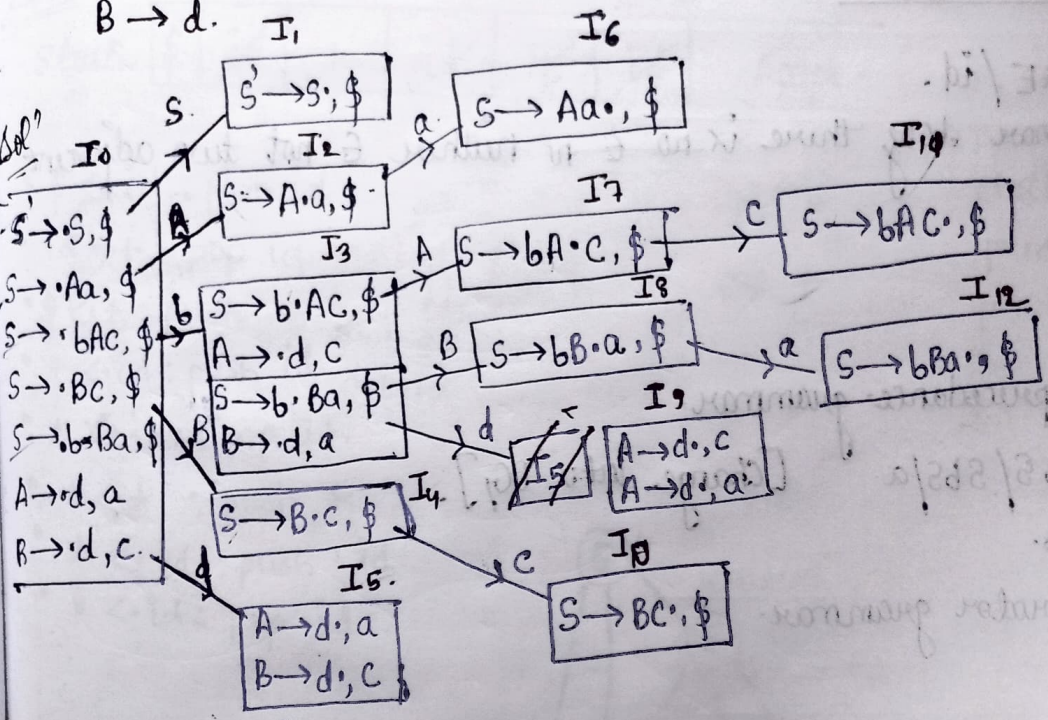
$B \to \epsilon : r_4$

∴ The given grammar is LALR(1) grammar.

(6.) $S \to Aa / bAc / Bc / bBa$. Construct LALR(1).

$A \to d$

$B \to d$.



∴ Combine I_5 & I_9 → I_{59}: $A \to d\cdot, a/c$; $B \to d\cdot, a/c$.

	Action					Goto		
	a	b	c	$	d	S	A	B
0		S_3			S_5	1	2	4
1				Accept				
2	S_6							
3					S_{59}		7	8
4			S_{10}					
59	γ_5/γ_6		γ_5/γ_6					
6				γ_1				
7			S_{11}					
8	S_{12}							
10				γ_3				
11				γ_2				
12				γ_4				

$S \to Aa\cdot : \gamma_1$

$S \to bAc\cdot : \gamma_2$

$S \to Bc\cdot : \gamma_3$

$S \to bBa : \gamma_4$

$A \to d\cdot : \gamma_5$

$B \to d\cdot : \gamma_6$.

∴ The given grammar is not LALR(1) grammar bcoz of reduce conflict.

05/03/2024

Operator Precedence Parser :-

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

→ Operator grammar bcoz there is ~~no ϵ~~ neither ϵ not two adjacent NT's.

2.) $S \rightarrow SAS / a.$
$A \rightarrow bSb / b.$

→ Not operator precedence grammar.

Now, $S \rightarrow SbSbS / SbS / a.$ [Change into OG].
$A \rightarrow bSb / b.$

∴ Now it is operator grammar.

3.) $P \rightarrow \overset{\times}{S}R / R$
$R \rightarrow bS\overset{\times}{R} / bS$
$S \rightarrow WbS / W.$
$W \rightarrow L*W / L$
$L \rightarrow id.$

→ $P \rightarrow SbS\underline{R} / bS\underline{R} / \overset{R}{\cancel{SbS}} / \cancel{bS}.$
$R \rightarrow bS\underline{R} / bS.$

Now, $P \rightarrow SbP / R\cancel{bS}$
$R \rightarrow bP / bS.$
$S \rightarrow WbS / W$
$W \rightarrow L*W / L.$
$L \rightarrow id.$

∴ It is an operator grammar

(Q.) id * id + id $.

Precedence table.

M	id	*	+	$
id	—	·>	·>	·>
*	<·	·>	·>	·>
+	<·	<·	·>	·>
$	<·	<·	<·	—

- ~~identa~~ have greater precedence always.
- $ has least precedence.
- * is left associative :- ·>

(Q.) $E \rightarrow E+E / E*E / id$ $w = id + id * id \$$

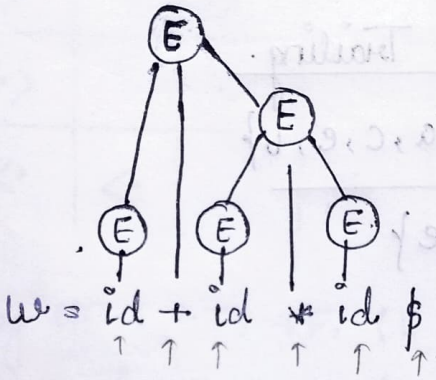
Sol?

Stack	$	id	+	id	*	id	Accept
		E				E	

- $\$ \lessdot id$: push id
- $id \gtrdot +$: pop id
- $\$ \lessdot +$: push +
- $+ \lessdot id$: push id
- $* \lessdot id$: pop 'id'
- $+ \lessdot *$: push '*'
- $* \lessdot id$: push 'id'
- $\$ \lessdot id$: pop id.

- $\$ \lessdot *$: pop *
- $\$ \lessdot +$: pop +
- $\$ - \$$.

stack element & element check karo, if precedence of element greater than SE then push
else if SE > element then pop SE.



07/03/2024

(Q.) $E \rightarrow AB$
$A \rightarrow aA / \epsilon$
$B \rightarrow bB / b$

Sol" 1.) Eliminate ϵ.

$E \rightarrow AB / B$
$A \rightarrow aA / a$.
$B \rightarrow bB / b$.

Now $E \rightarrow AbB / Ab / B$
$A \rightarrow aA / a$.
$B \rightarrow bB / b$.

∴ The grammar is operator precedence grammar.

Leading & Trailing Calculation

(Q.) $E \rightarrow E+T / T$
$T \rightarrow T*F / F$
$F \rightarrow (E) / id$.

Sol^n

	Leading (L to R)	Trailing (R to L)
E	{+, *, (, id}	{id,), *, +}
T	{*, (, id}	{id,), *}
F	{(, id}	{id,)}

- Scan till we find a terminal.
- T → F : so leading of F also included
- If we get NT include its leading & also scan till we reach 1st terminal

(Q.) A → abB / dAc / a
B → aAe / de

Sol^n

	Leading	Trailing
A	{a, d}	{a, c, e, b}
B	{a, d}	{e}

(Q.) A → Bbc / aA
B → dB / ε

Sol^n

	Leading	Trailing
A	{a, b, d, ε}	{a, c}
B	{d, ε}	{e, d}

(Q.) A → BbAd / aB / cd
B → AeBc / BbA / b

Sol^n

	Leading	Trailing
A	{b, a, c, e}	{d, a, c, b}
B	{b, a, c, e}	{b, c, a, d}

(Q.) $E \to E+T/T$ → T follow by NT

$T \to T*F/F$

$F \to (E)/id$

→ NT follow by T

Sol^n

	+	*	(	)	id	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	
)	·>	·>		·>		·>
id	·>	·>		·>		·>
$	<·	<·	<·		<·	

- leading : <· : row-wise
- trailing : ·> : col-wise
- if ( ) : ≐ or a b : ≐

Entry acc to :

1) Terminal followed by NT : +T, *F, (E

2) NT followed by terminal : E+, T*, E)

+T : leading of T : {*, (, id} E+ : trailing of E : (id,), *, +}

F : leading of F : (, id} T : ——"—— T : (id, (, *}

(E : leading of E : (+, *, (, id} E) : ——"—— E : (id,), *, +}

- In $ row start symbol leading & $ col. start symbol trailing