Introduction to Bottom Up parsers

outcome :

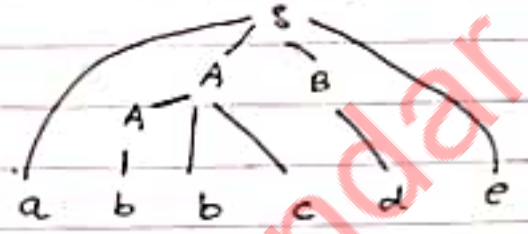
- ✓ Understanding the Bottom-Up approach of parsing
- ✓ Revisit the classification of parsers.

Generation of parse Tree - Bottom Up Approach

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$



Decision : when to reduce

$S \rightarrow aA\underline{Be}$ $S \rightarrow a\underline{A}de$ $S \rightarrow a\underline{A}bcde$ $S \rightarrow abbcde$

(rightmost derivation in reverse)

classification of parsers :

Points to Note :

- ✓ Top down parsers with backtracking can handle non-determinism - but those without backtracking cannot.
- ✓ only operator precedence parsers can handle ambiguous grammar.
- ✓ power of LR(0) < SLR(1) < LALR(1) < CLR(1)

Summary :

- ✓ Understanding Bottom Up parsing.
- ✓ classification of parsers

Bottom Up : → operator precedence
→ LR : LR(0), SLR(1), LALR(1), CLR(1)

Operator Precedence parsers

outcome :

- ✓ understanding the operator Precedence Grammar.
- ✓ How to convert a CFG into operator Grammar.
- ✓ Generation of operator relation Table
- ✓ operator Precedence Parsing.

Operator Precedence Parser :

1. It is mainly used for mathematical expressions.
2. It process operator grammar.
3. It can handle ambiguous Grammars.
4. It uses operator Relation Table.

Why can it handle ambiguity ?
If a grammar is ambiguous, for the same string, we can generate many different parse trees → the reason behind that is the production rules of the ambiguous grammar are defined without considering the associativity and precedence of the operators involved in those production rules → that is why all other parsers need unambiguous grammar. on the other hand, due to the use of the operator relation table, operator precedence parser can smoothly handle ambiguous grammar - wherein the associativity and precedence of the operators are clearly defined.
↳ Handles ✓

operator Grammar :

used to define operators

Restrictions :

① No adjacent non-terminals.
② No epsilon (ε) productions.

$E \rightarrow EAE \mid id$
$A \rightarrow + \mid *$
$\Rightarrow$
$E \rightarrow E+E \mid E*E \mid id$

conversion to operator Grammar :

$S \rightarrow SAS \mid a$
$A \rightarrow bSb \mid b$
$\Rightarrow$
$S \rightarrow SbSbS \mid SbS \mid a$
$A \rightarrow bSb \mid b \rightarrow$ unreachable

↳ $S \rightarrow SbSbS \mid SbS \mid a$

construction of operator Relation Table

$E \rightarrow E+E \mid E*E \mid id$

Precedence	Associativity
*, /, %	Left to Right
+, -	Left to Right

Note :
id → Highest Precedence
$ → Lowest Precedence

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

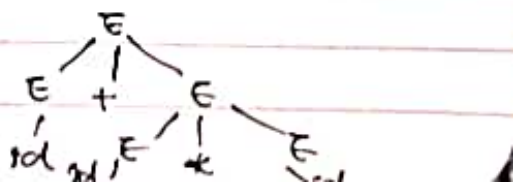
summary done

Rules for parsing :

① If TOS < LookAhead, PUSH (shift) LookAhead into the stack
② If TOS > LookAhead, POP (Reduce)

eg: parse id + id * id $

E + (E * E)



Improved operator Precedence Parser

Outcome :

- ✓ Disadvantage of using operator Relation Table.
- ✓ construction of operator Function Table.

For 'n' terminals, size of operator relation table = $O(n^2)$ $[(n+1)(n+1)]$

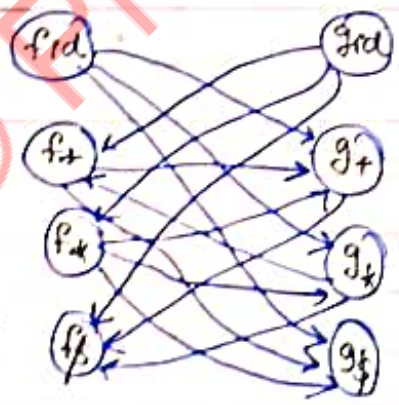
↓ can be reduced

construction of operator Function Table

rows are defined by function (f)
columns are defined by function (g)

↓

A directed graph is drawn, where all the terminals of f will be treated as one set of nodes, while the terminals of g will be treated as another set of nodes.



From this directed graph, operator function table will be constructed.

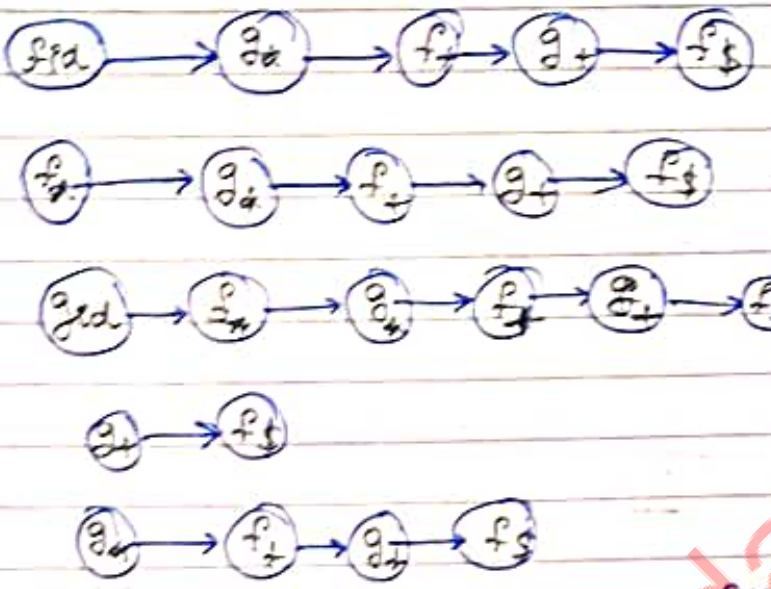
Restriction : we cannot draw it if the graph contains any cycle.

Prog. operator relation table Rule :

Direction of edge :

greater precedence → lower precedence

procedure : Find out the longest path starting at each individual node and note them down.



operator function table

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

Rule for filling: No. of edges in longest path starting from every node.

Note: $f\$$ and $g\$$ will never contain any outgoing edges, as it($) has the least precedence among all other symbols. So $f\$ = 0$, $g\$ = 0$ in cells.

significance of cell data: More the value, more the precedence.

Table size = $\boxed{o(2n)}$ only

Now $f* = 4$, $g* = 3$

$E \rightarrow E + E \mid E * E \mid id$

(f ↑, g ↑)

left > right ⇒ left associativity

(rows) (columns)

Summary:

1. Disadvantages of operator Relation Table.
2. Construction of operator Function Table.

Operator Precedence Parsing - Solved Problem

outcome : construction of operator function table for a given CFG.

Q. construct the operator function table for operator Precedence Parsing from the following CFG,

P → SR | S
R → bSR | bS
S → Wbs | W
W → L*W | L
L → id

Grammar description

sentence : word followed by a blank or a sentence or a word

word : a letter concatenated with a word or a letter.

letter : identifier

Paragraph : sentence followed by Recursive_# sentences of a sentence.

Recursive_# sentences : a blank followed by a sentence & Recursive_# sentences (OR) a blank followed by a sentence

converting into operator grammar :

P → SbSR | SbS | S

P → SbP | SbS | S
P → useless

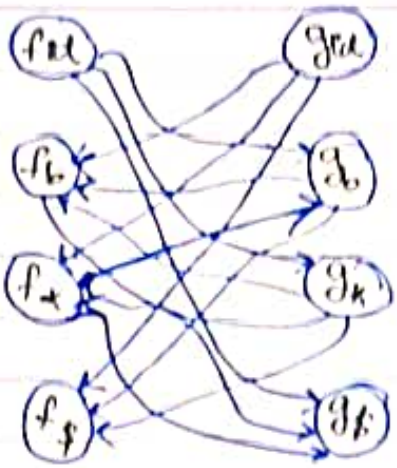
W → L*W
P → SbP
S → WbS
(b has right associativity)

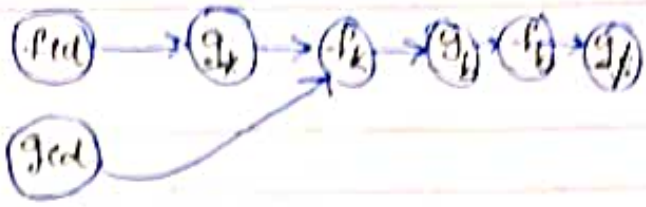
precedence increases

P → SbP | SbS | S
S → WbS | W
W → L*W | L
L → id

(* > b evaluate *)

	id	b	*	$
id	-	>	>	>
b	<	<	<	>
*	<	>	<	>
$	<	<	<	-





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

Disadvantage of operator function table :

In the operator relation table, we never compared the ids, but here, we are comparing the ids. so although operator function table reduces the size of the table, which could eventually enable the parser to take lesser time for parsing, the error detection capacity will be lesser than that of the operator relation table.

Summary : construction of operator function table.

Introduction to LR parsers

outcome :

- ✓ Understanding the organization of LR parsers
- ✓ What are LR(0) items ?
- ✓ CLOSURE and GOTO procedures.
- ✓ Derivation of canonical collection of LR(0) items.

LR parsers : (L) → scan from left to right
(R) → rightmost derivation in reverse

* i/p buffer
* Stack ← LR parser
* LR parsing table

LR- parsers

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

- SLR(1): use LR(0) items for the construction of the parsing table
- LALR(1) < CLR(1): use LR(1) items for the construction of the parsing table

LR(0) Parser :

1. conversion to augmented Grammar.
2. use CLOSURE and GOTO properties.

$S \to AA$
$A \to aA \mid b$
$\Rightarrow$
$S' \to S$
$S \to AA$
$A \to aA \mid b$

$S' \to S$: new start symbol S' introduced (augmented grammar)

LR(0) Items :

$S \to \cdot AA$ — The compiler hasn't read anything yet.

$S \to A \cdot A$ — The compiler has read till here.

$S \to AA \cdot$ — The compiler has read the entire RHS.

CLOSURE :

Whenever there is a '.' to the left of a non-terminal, include all its productions in the set, adding a '.' preceding the rules.

↓

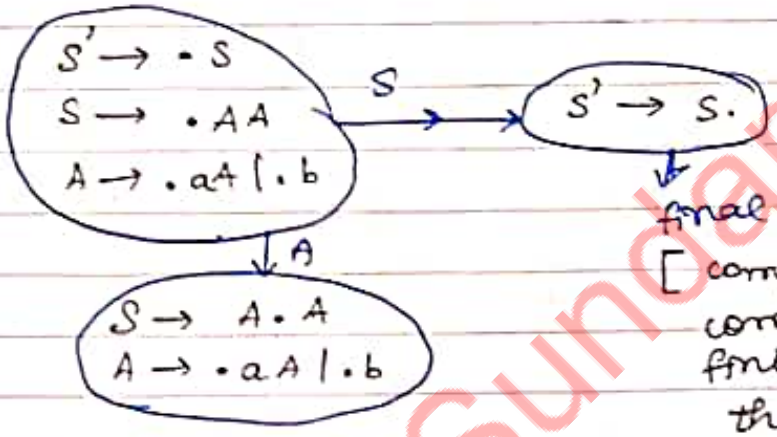
closure is of an LR(0) item

$S' \to \cdot S$
$S \to \cdot AA$
$A \to \cdot aA \mid \cdot b$

→ (closure of $S' \to \cdot S$)

GOTO:

on a canonical collection of LR(0) items, we apply GOTO for every non-terminal by moving the '.' to the right of the same, thus the new collection is obtained.



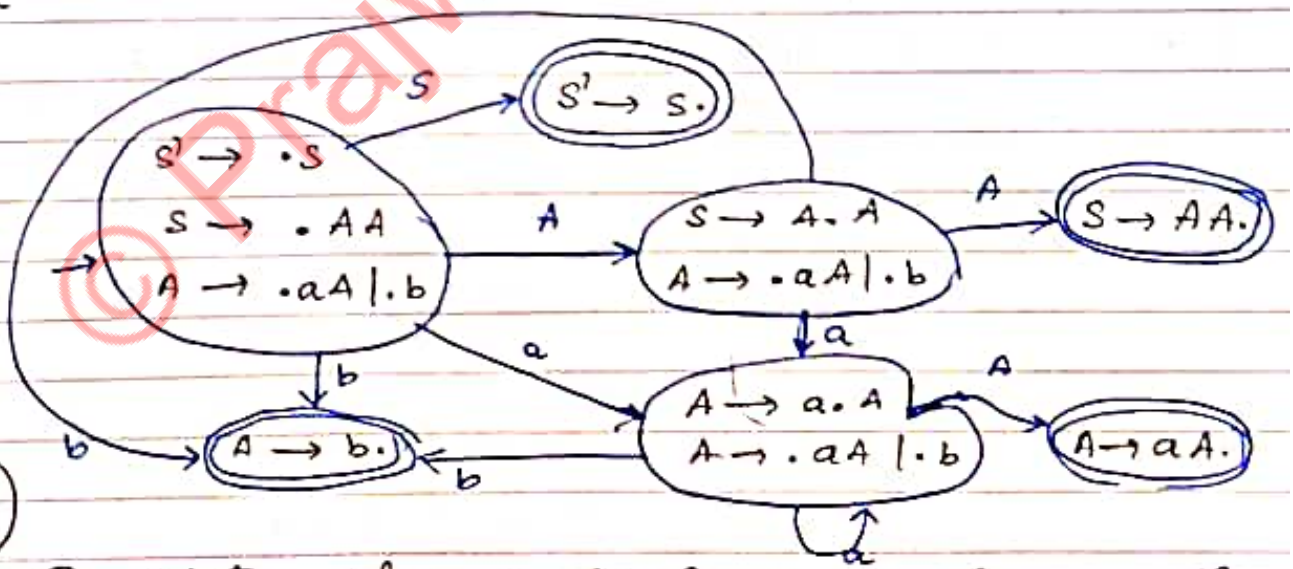
final item [compiler has completely finished reading the RHS].

canonical collection of LR(0) items:

Take CLOSURE & GOTO.

they are nothing but finite state machines.

using this, we shall create the passing tables.

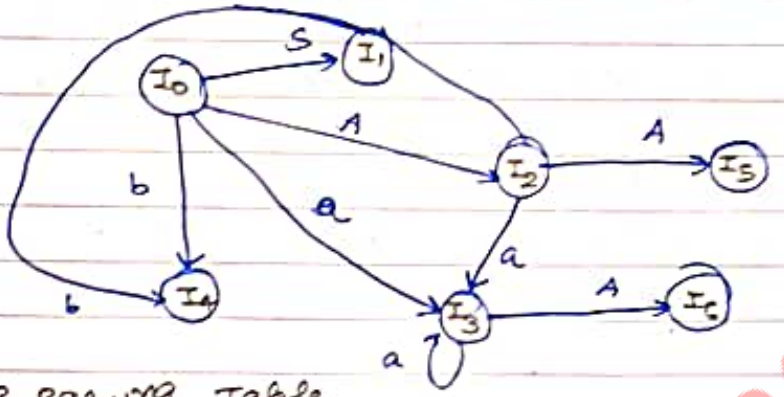


(summary: all done)

- ✓ states where . is placed at the end of every production rule are called final items.
- ✓ Moves involving non-terminals are called GOTO moves, while moving terminal symbols are known as the SHIFT moves.
- ✓ Final states guarantee of reduction. When the compiler reaches any final state, now the compiler can reduce the RHS → LHS

LR(0) Parsers – Part 1

Outcome : construction of LR(0) parsing table.



[remaining states from the previous canonical collection]

LR Parsing Table

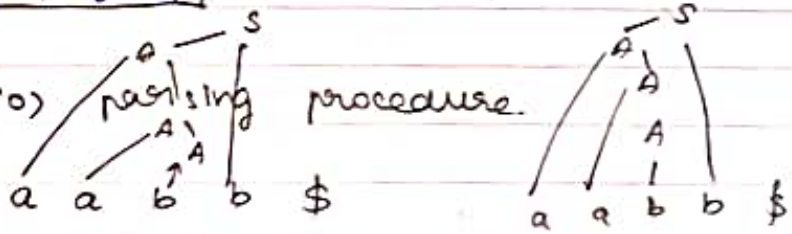
States	Action			GO TO	
	a	b	$	A	S
0	s_3	s_4		2	1
1			Accept		
2	s_3	s_4		5	
3	s_3	s_4		6	
4	r_{iii}	r_{iii}	~~r_{iii}~~		
5	r_i	r_i	~~r_i~~		
6	r_{ii}	r_{ii}	~~r_{ii}~~		

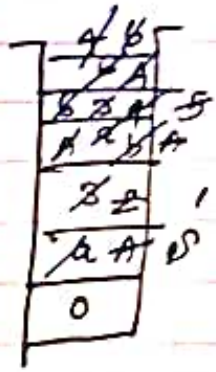
(i) $S \rightarrow A\ A$
(ii) $A \rightarrow a\ A$
(iii) $A \rightarrow b$
rules numbered

Summary : construction of LR(0) parsing table.

LR(0) Parsers – Part 2

Outcome : LR(0) parsing procedure.





Sn : ① push Input
② push the state n

Reduce : eg 4 b $A \rightarrow b$

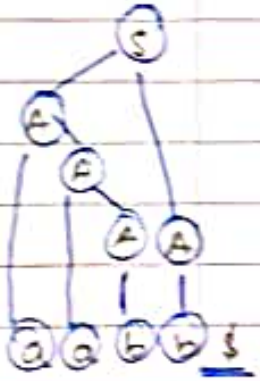
$|RHS| = 1$, $1 \times 2 = 2$ → pop twice

summary : LR(0) parsing procedure.

Tabular Representation : PPL

Stack	Input	Action
0	a a b b $	Shift 3
0 a 3	a b b $	Shift 3
0 a 3 a 3	b b $	Shift 4
0 a 3 a 3 b 4	b $	Reduce 3
0 a 3 a 3 A 6	b $	Reduce 2
0 a 3 A 6	b $	Reduce 2
0 A 2	b $	Shift 4
0 A 2 b 4	$	Reduce 3
0 A 2 A 5	$	Reduce 1
0 S 1	$	Accept

Parse Tree :

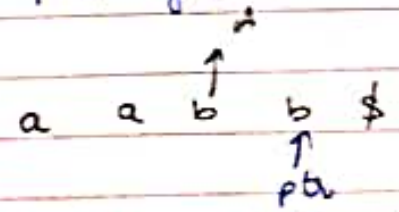


SLR(1) Parsers

Outcome :

- ✓ Problem with LR(0) parsing procedure.
- ✓ Difference between LR(0) & SLR(1) Parsers.
- ✓ Construction of SLR(1) parsing table.

LR(0) parsing :



We aren't reducing the b pointed by the pointer in the input buffer → rather we are reducing the previous one. In the parsing procedure, we got lucky because the other b also gets reduced to A. If the (b) ptr pointing variable doesn't exist in the input buffer, the final reduction to S wouldn't have been possible

This is why it is called LR(0) parser → with 0 lookahead symbols. We were pointing to something in the input buffer, but reducing the previous symbol from the buffer → [which is also the reason why for the states containing the final items, we were using the same reduction rules for all the terminal symbols.]

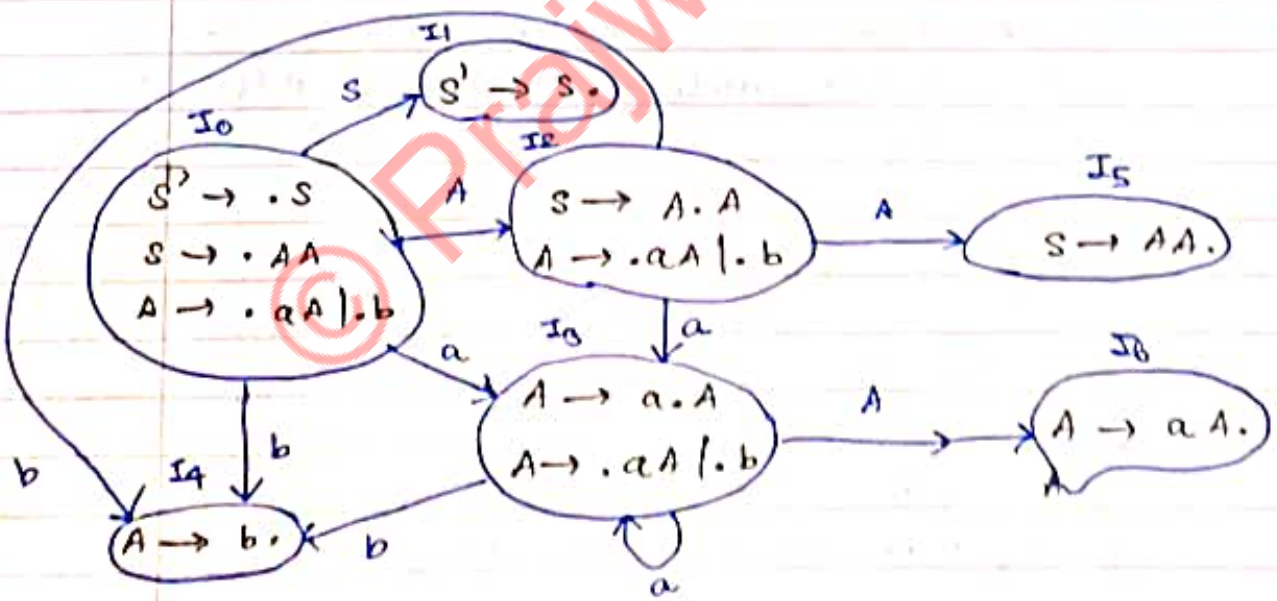
This is the problem associated with the LR(0) parsing procedure.

SLR(1) Parsing Table

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

→

$S' \to S$
$S \to AA$
$A \to aA \mid b$



	FIRST	FOLLOW
$S \to AA$	{a, b}	{$}
$A \to aA \mid b$	{a, b}	{$, a, b}
$A \to b$		

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

States	Action			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r_{iii}	r_{iii}	r_{iii}		
5			r_i		
6	r_{ii}	r_{ii}	r_{ii}		

FOLLOW(A) = { a, b, $ }
↳ reduce iii in row 4, columns

similarly row 6 r_{ii}

FOLLOW(S) = { $ }
↳ reduce (i) in row 5, column

comparing SLR(1) and LR(0) parsing tables:
content-wise we have more space in the SLR(1) parsing table → Now we already know that during the parsing procedure, at any point, if we end up in any blank spaces on the table, that will signify error, and as spaces are more in case of SLR(1) parsing table, it means that SLR(1) parsing is more powerful than LR(0) parsing. the power has been achieved in SLR(1) parsing because in here for reduction, we are considering the look-ahead.

summary:
- ✓ LR(0) problem
- ✓ LR(0) vs SLR(1) diff
- ✓ SLR(1) parsing table

Conflicts in LR(0) and SLR(1) parsers

outcome :

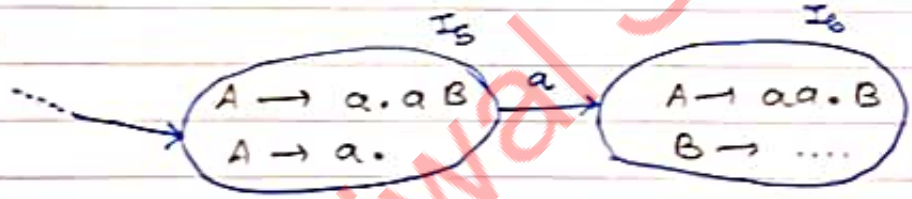
- ✓ S-R and R-R conflicts in LR(0) parsing table.
- ✓ S-R and R-R conflicts in SLR(1) parsing table

LR(0) parsing table :

In the previous example, we were lucky that we didn't encounter any conflicts, that is, no entry in the LR(0) parsing table has multiple entries. But in reality, this may happen.

iii) $A \to aaB$ $\quad A \to aaB \mid a$

iv) $A \to a$



Here, in LR(0) parsing table.

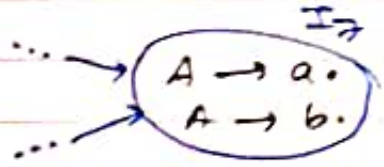
	a
5	s_6 / r_{iv} → (both appear)

↓

shift-Reduce conflict

iii) $A \to a$

iv) $A \to b$ $\quad A \to a \mid b$



Reduce-Reduce conflict

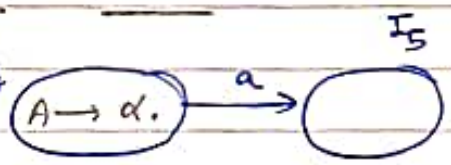
	a	b	$
7	r_{iii} / r_{iv}	r_{iii} / r_{iv}	r_{iii} / r_{iv}

If a grammar shows either S-R or R-R conflict, that grammar will not be an LR(0) grammar.

__/__/__

SLR(1) parser :

SR conflict

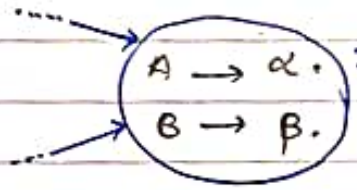


and if $FOLLOW(A) = \{\dots, a, \dots\}$

a

4 | S_5 / r_x | ⇒ SR conflict

RR conflict



$FOLLOW(A) \cap FOLLOW(B) \neq \phi$

$= \{\dots, x, \dots\}$

x

6 | r_y / r_z | → RR conflict

Also, a grammar that shows either S-R or R-R conflict will not be an SLR(1) grammar.

summary

- ✓ S-R and R-R conflicts in LR(0) PT.
- ✓ S-R and R-R conflicts in SLR(1) PT.

Determining the type of grammar - set 1

outcome : understanding how to determine the type of a given grammar.

Q1. Determine whether the following grammar is LL(1) or LR(0) or SLR(1) :

FIRST		FOLLOW
{d, a}	$S \to dA \mid aB$	{$}
{b, c}	$A \to bA \mid c$	{$}
{b, c}	$B \to bB \mid c$	{$}

	a	b	c	d	$
S	S→aB			S→dA	
A		A→bA	A→c		
B		B→bB	B→c		

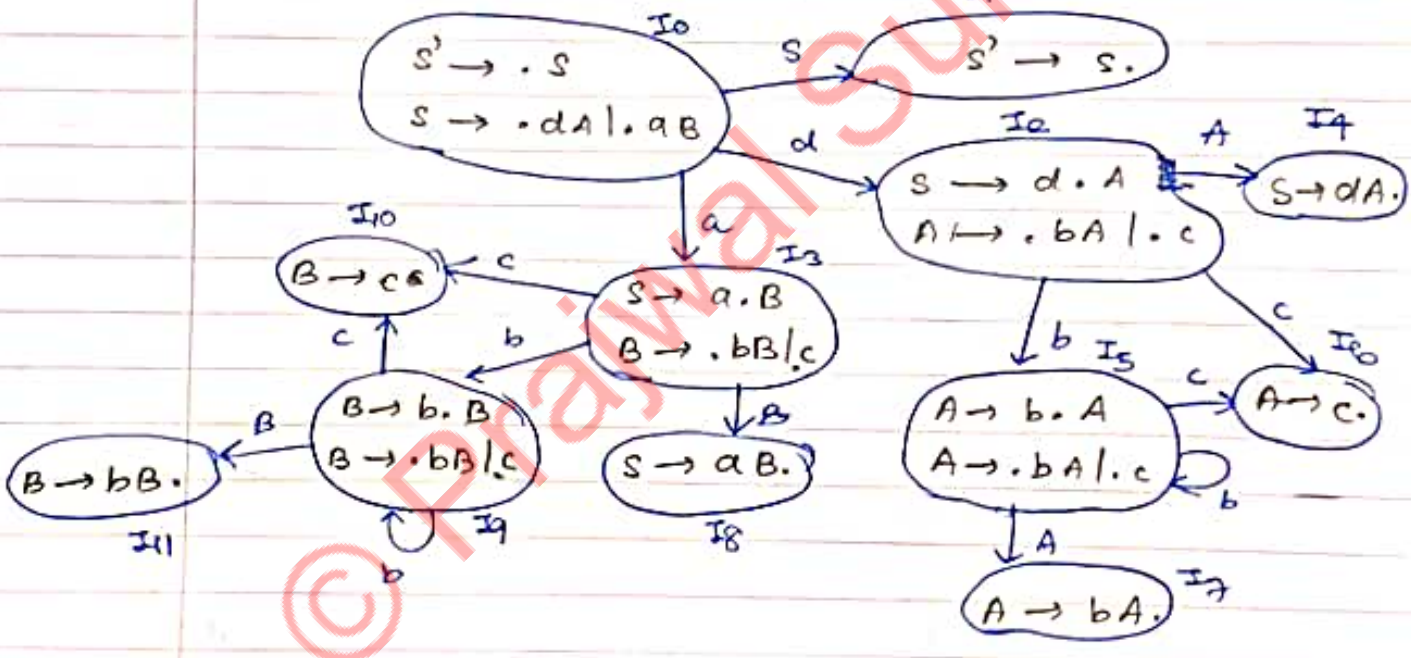
✓ LL(1) grammar

Augmented grammar:

$S' \rightarrow S$
$S \rightarrow dA \mid aB$
$A \rightarrow bA \mid c$
$B \rightarrow bB \mid c$

After drawing CC:
For all final items:
✓ 1 final item.
✓ no shift move.

canonical collection:



As final items have
✓ 1 final item — RR ✗
✓ no shift move — SR ✗
] safe! LR(0) ✓ grammar

SLR(1) is a reduced version of LR(0)

LR(0) ✓ ⟶ definitely SLR(1) ✓

✓ LR(0) ✓ LL(1)
✓ SLR(1)

Summary: How to determine grammar type

Determining the type of grammar - set 2

outcome :

✓ 2 solved problems on how to determine the type of a given grammar.

Q1 : Determine whether the given grammar is LL(1) or LR(0) or SLR(1) :

FIRST		FOLLOW
{a}	S → A | a	{$}
{a}	A → a	{$}

	a	$
S	S → A S → a	
A	A → a	

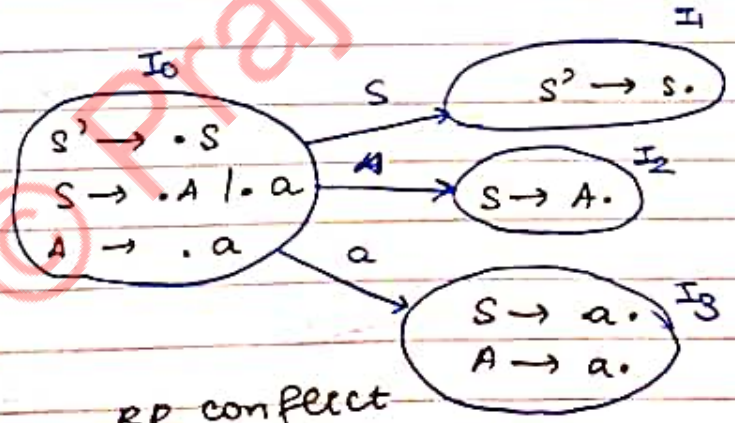
✗ not an LL(1) grammar

Augmented grammar :

S' → S
S → A | a
A → a

Ambiguous grammar → to derive (a)

S → A | a (or) S → a



RR conflict

I_3 has 2 final states → LR(0) ✗

NOW FOLLOW(S) ∩ FOLLOW(A) = {$} ≠ ϕ NOT NULL

RR conflict exists in SLR(1) table also ✗

LL(1)	LR(0)	SLR(1)
✗	✗	✗

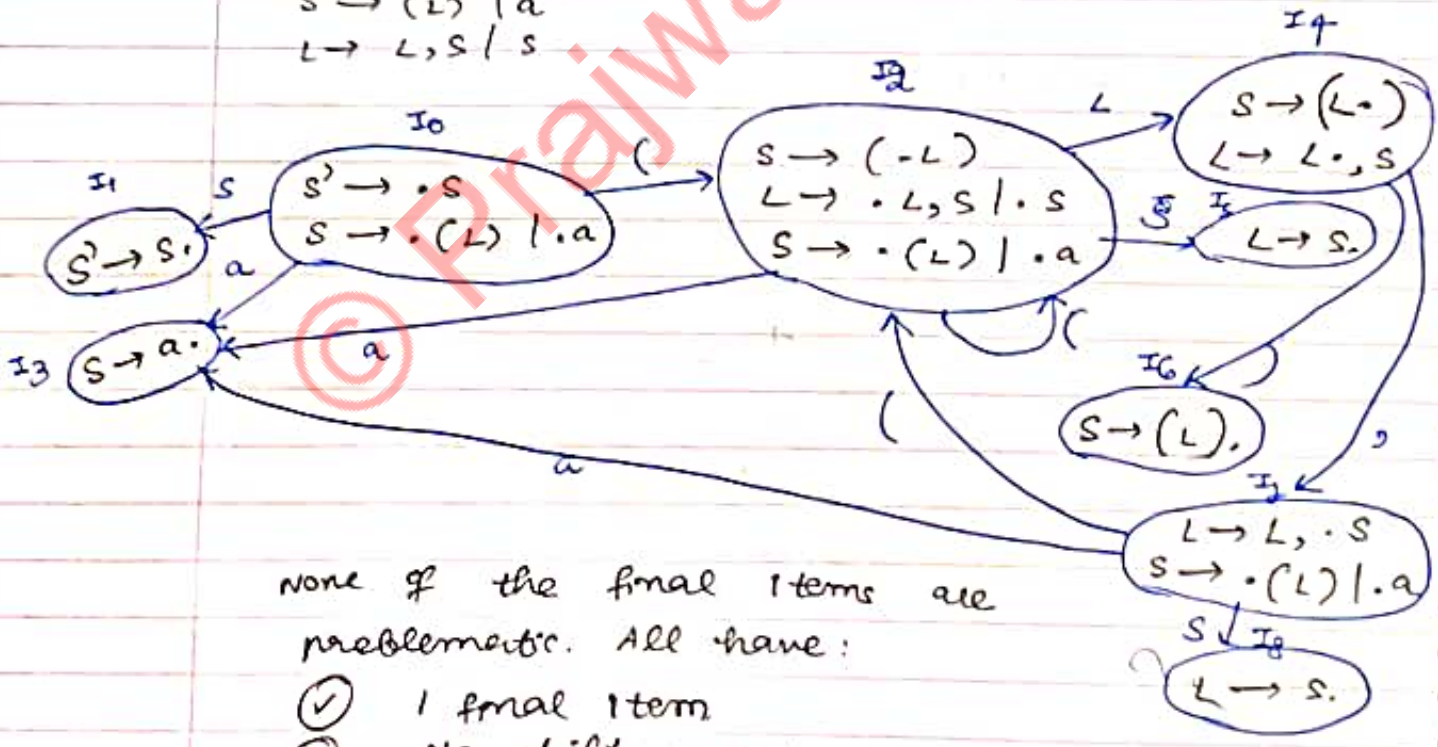
Q2: Determine the type of grammar:

FIRST		FOLLOW
{ (, a }	S → (L) | a	{ $,), , }
{ (, a }	L → L, S | S	{), , }

	(	)	,	a	$
S	S→(L)			S→a	
L	L→S			L→S	

There are no multiple entries in any cell BUT STILL, It is not LL(1) ⊗ grammar due to the presence of left recursion. L → L, S

S' → S
S → (L) | a
L → L, S | S



None of the final items are problematic. All have:

✓ 1 final item
✓ No shift move

LL(1)	SLR(0)	SLR(1)
⊗	✓	✓

Summary: 2 solved problems.

Determining the type of grammar - set 3

outcome : solved problem on how to determine the type of a given grammar.

Q. Determine the grammar type :

{i} $E \rightarrow E + T \mid T$ { $, + }

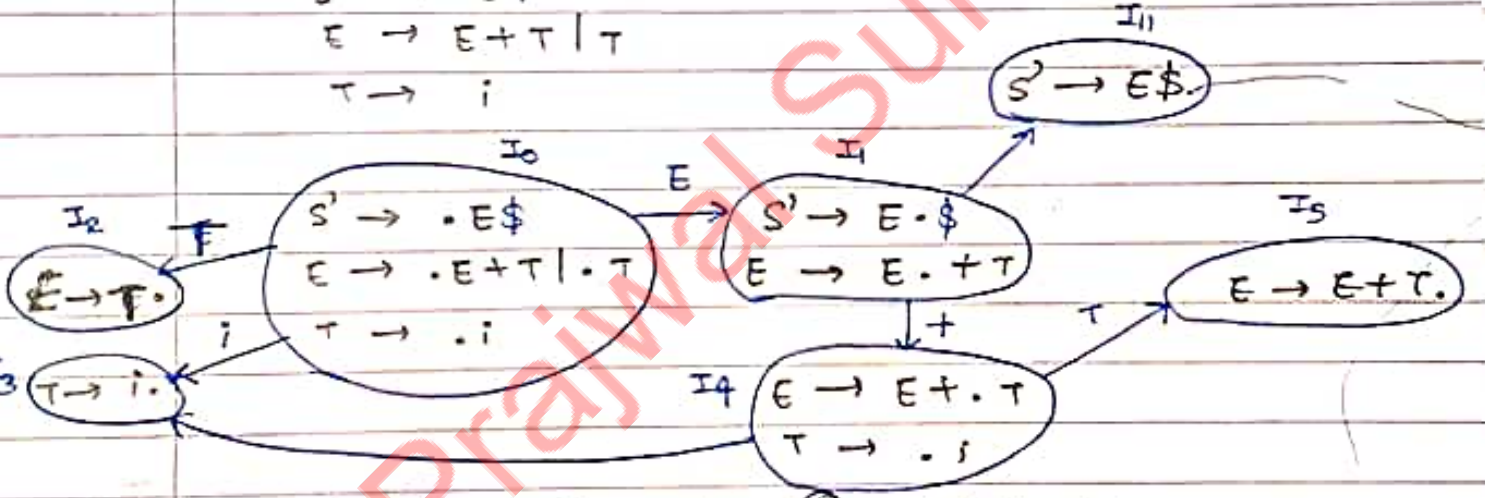
{i} $T \rightarrow i$ { $, + }

$E \rightarrow E + T \mid T$ left recursion → ⊗ not LL(1)

$S' \rightarrow E\$$

$E \rightarrow E+T \mid T$

$T \rightarrow i$



All final items : ✓ 1 final item

✓ no shift move

LL(1)	LR(0)	SLR(1)
✗	✓	✓

Q: {i} $E \rightarrow T + E \mid T$ { $ }

{i} $T \rightarrow i$ { +, $ }

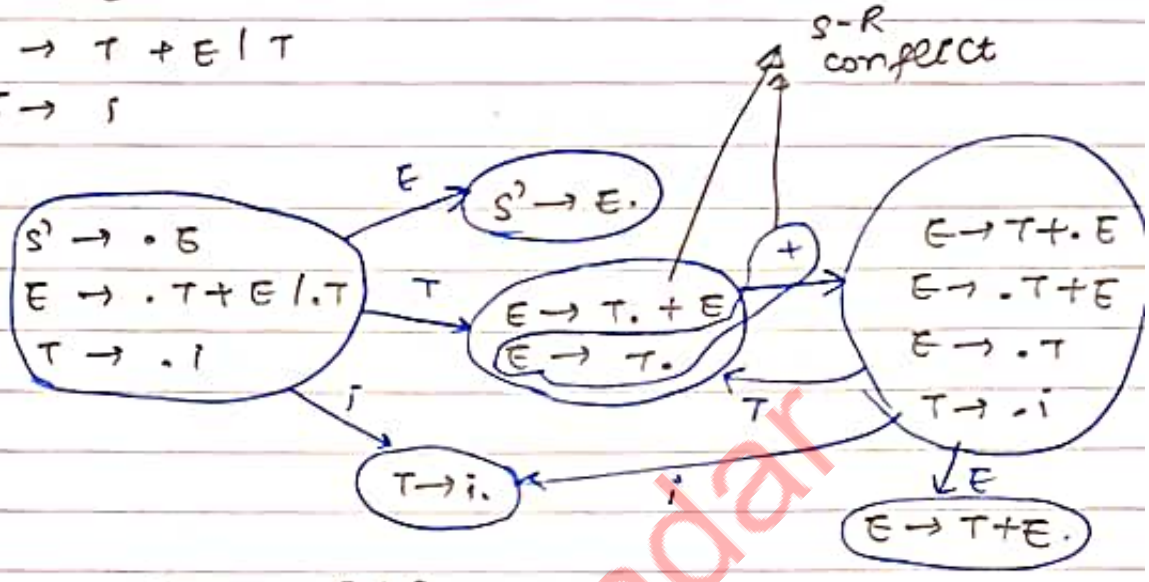
	+	i	$
E		E→T+E E→T	
T		T→i	

⊗ not LL(1) grammar

$S' \to E$

$E \to T + E \mid T$

$T \to i$



FOLLOW(E) = { $ }

↳ not involved in shift (+)

no SR conflict in SLR(1) parser

LL(1)	LR(0)	SLR(1)
⊗	⊗	✓

Summary : solved problem on how to determine type of a given grammar.

Determining the type of grammar – set 4

outcome : ✓ 2 solved problems

Q1: Determine if LL(1), LR(0), SLR(1) :

$E \to E + T \mid T$

$T \to T F \mid F$

$F \to F* \mid a \mid b$

$E \to E + T$

$T \to T F$

$F \to F *$

left recursive

LL(1) ⊗

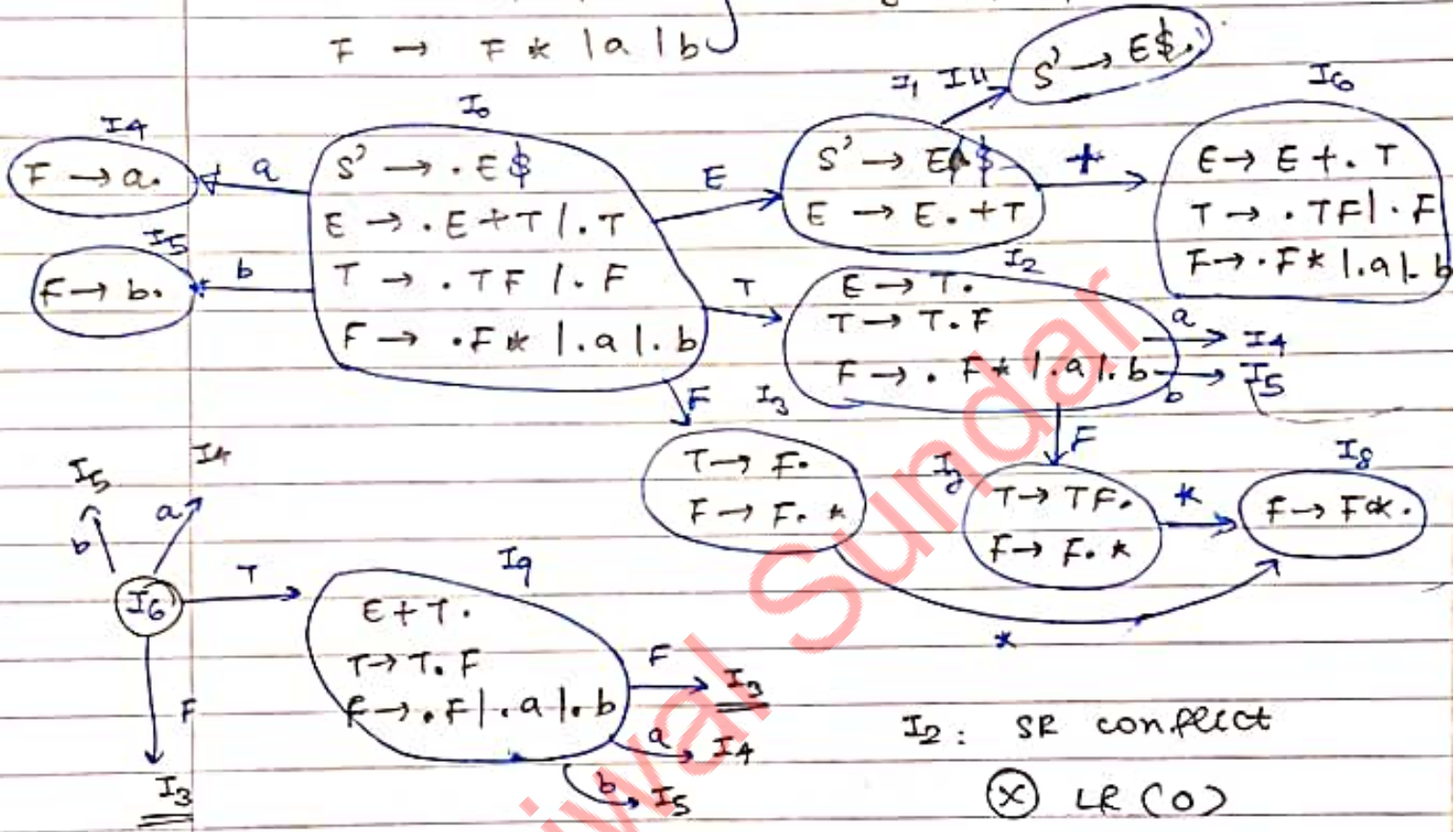
↓

[top down parser]

FIRST	FOLLOW
{a,b}	{+, $}
{a,b}	{+, $, a, b}
{a,b}	{*, +, $, a, b}

$S' \to E\$$
$E \to E+T \mid T$
$T \to TF \mid F$
$F \to F* \mid a \mid b$

augmented grammar



I_2: SR conflict

⊗ LR(0)

FOLLOW(E) = {+, $} ≠ {a, b} ✓

I_3: SR FOLLOW(T) ∉ * ✓

I_4, I_5, I_8: single final items ✓

I_7: FOLLOW(T) ∉ * ✓

I_9: FOLLOW(E) = {~~a~~ ~~b~~ +, $}
(not these; no problem)

SR conflict ⊗ no conflict

LL(1) ⊗ LR(0) ⊗ SLR(1) ✓

∴ Grammar is only SLR(1).

Q2. Determine type of grammar :

FIRST		FOLLOW
{a, b}	S → AaAb | BbBa	{$}
{ε}	A → ε	{a, b}
{ε}	B → ε	{b, a}

	a	b	$
S	S → AaAb	S → BbBa	
A	A → ε	A → ε	
B	B → ε	B → ε	

LL(1) ✓ grammar

note : For productions A → ε, the LR(0) items A → · ε ≅ A → ε · ≅ A → ·

S' → S
S → AaAb | BbBa
A → ε
B → ε

S' → · S
S → · AaAb | · BbBa
A → ·
B → ·

→ reduce-reduce conflicts

LR(0) ✗

FOLLOW(A) ∩ FOLLOW(B) ≠ ϕ

↓

in SLR(1) parsing table also, there will be R-R conflicts ✗

LL(1) ✓ LR(0) ✗ SLR(1) ✗

LR(0) ✓ → SLR(1) ✓

keypoint to note,

If LR(0) ✗ → again verify SLR(1)

summary :
2 solved Q.

Determining the type of grammar - Pt. 5

Outcome : Solved problems on how to determine the type of a given grammar.

Q1. Determine the type of the following grammar :

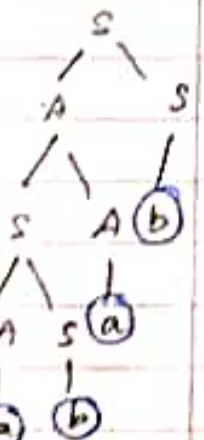
$S \to AS \mid b$
$A \to SA \mid a$

	FIRST	FOLLOW
S	{a, b}	{$, a, b}
A	{a, b}	{a, b}

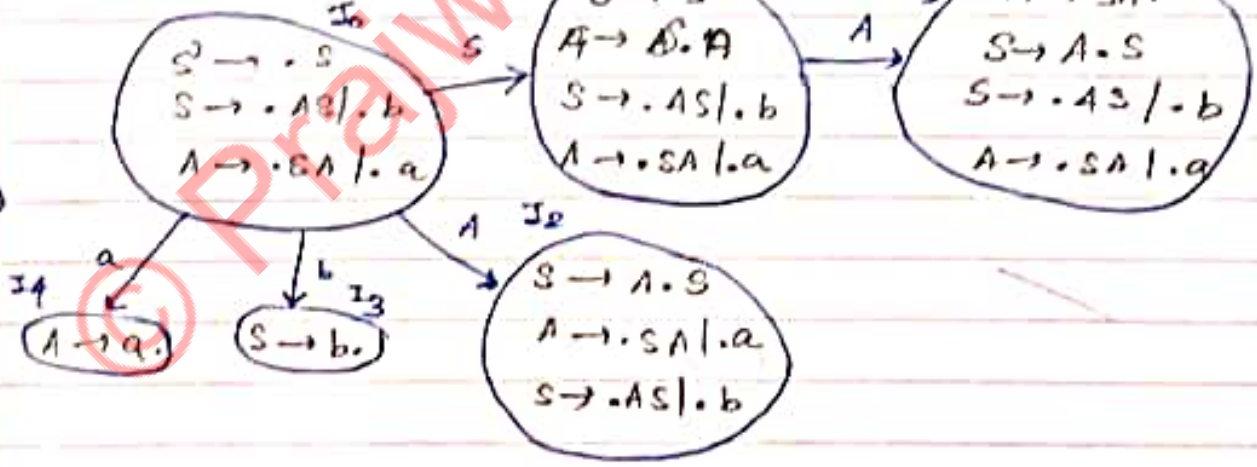
	a	b	$
S	$S \to AS$	$S \to AS$, $S \to b$	
A	$A \to a$	$A \to SA$	

⊗ not LL(1) grammar

Ambiguous grammar :



$S' \to S$
$S \to AS \mid b$
$A \to SA \mid a$



Final item containing shift moves detected ! LR(0) ⊗

FOLLOW(A) : {a, b} ∈ {a, b} so not SLR(1) ⊗ also

LL(1)	LR(0)	SLR(1)
⊗	⊗	⊗

∴ as grammar is ambiguous. ∴

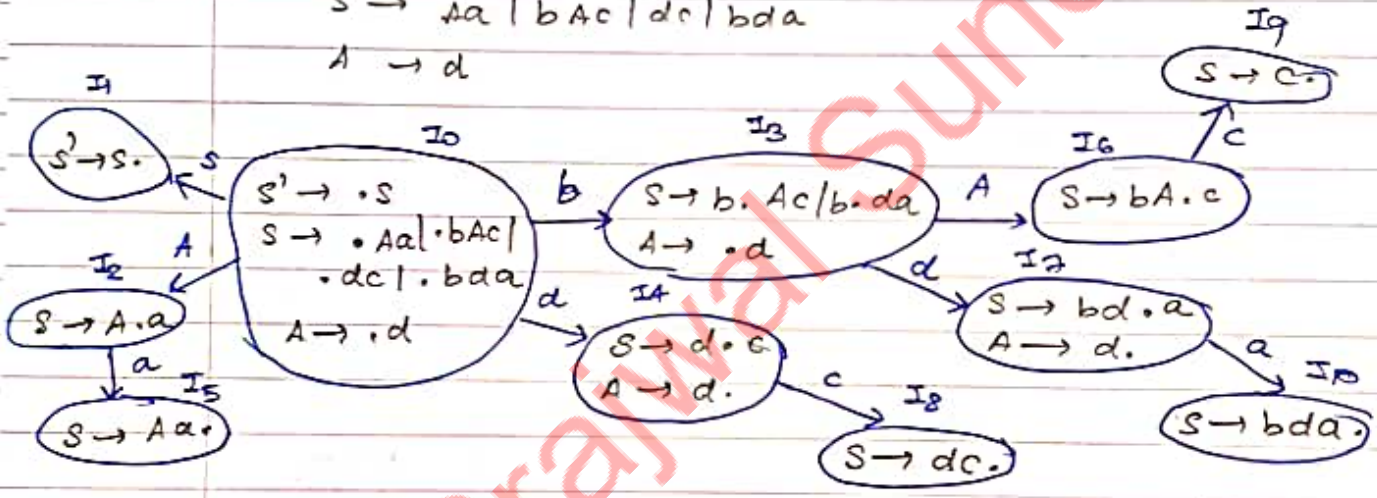
Q2 : Determine the type of grammar :

FIRST		FOLLOW
{a, b}	S → Aa | bAc | dc | bda	{$}
{d}	A → d	{a, c}

	a	b	c	d	$
S		S→bAc S→bda		S→Aa S→dc	
A				A→d	

⊗ not an LL(1) grammar

S' → S
S → Aa | bAc | dc | bda
A → d



I4 : A → d. —c→ SR conflicts ⊗ not LR(0) grammar ⊗

I7 : A → d. —a→

c ∈ FOLLOW(A) ⊗ a ∈ FOLLOW(A) ⊗

↳ not SLR(1) grammar also ⊗

LL(1)	LR(0)	SLR(1)
⊗	⊗	⊗

But grammar isn't ambiguous

key : ambiguous → LL(1) ⊗ LR(0) ⊗ SLR(1) ⊗

[not reverse]

Summary : ✓ solved ✓ done

canonical collection of LR(1) items

outcome :

- ✓ Understanding LR(1) items.
- ✓ Derivation of the canonical collection of LR(1) items from a given grammar.

LR parsers :

organization wise, all LR parsers are the same. However, the only difference lies in how we are constructing the LR parsing table.

↓

LR(1) items = LR(0) items + 1 look-ahead symbol

(i) .

(ii) $S \to aA$

(iii) :

LR(0) item

$S \to \cdot aA$

$S \to a \cdot A$

$S \to aA \cdot$

LR(1) item

$S \to aA \cdot$, a / b

↳ final item will be associated with the look-ahead as well

In

$S \to aA \cdot$, a / b

⋮

states	action			
	a	b	c	$
r_1	r_{11}	r_{11}		

In LR(1) items, the reduced moves only will be placed under the look-ahead symbols which are already mentioned in the state itself.

↳ How to figure out the look-ahead symbols?

↓

example :→

canonical collection of LR(1) items

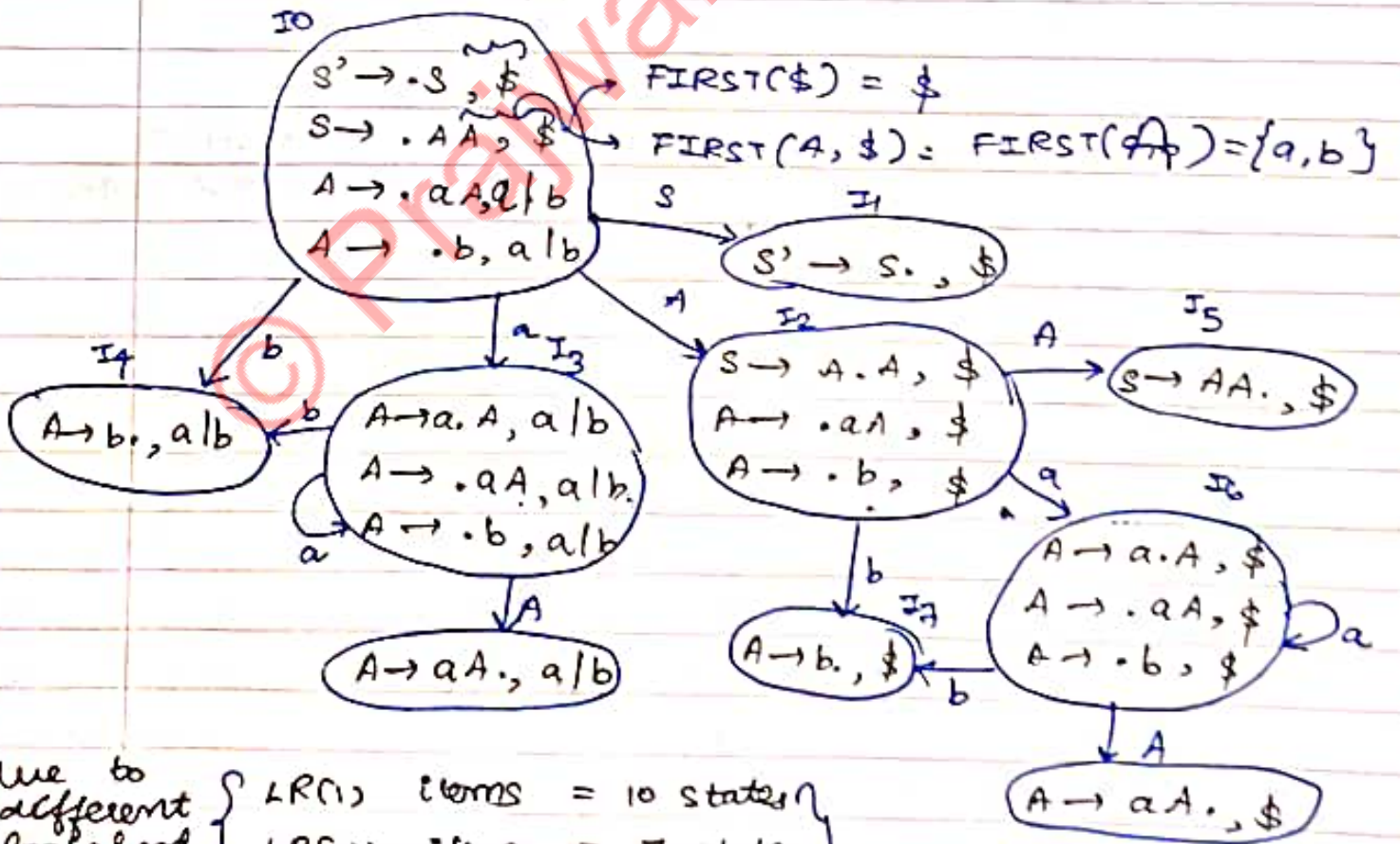
$S \to AA$
$A \to aA \mid b$

→ augmented grammar →

$S' \to S$
$S \to AA$
$A \to aA \mid b$

Rules :

1. For start symbol's production, the look ahead is '$'.
2. In the closure set, the look aheads of the LR(1) items are determined based on the rest of the items' FIRST set.
3. Look ahead will never change during transitions. However, they may change within state while determining the closure set.



due to different lookahead symbol { LR(1) items = 10 states ; LR(0) items = 7 states }

Summary : ✓ LR(1) items ✓ canonical collection of LR(1) items

CLR(1) Parser

outcome : construction of CLR(1) parsing table.

$S' \rightarrow S$ (i) $S \rightarrow AA$ (ii) $A \rightarrow aA$ (iii) $A \rightarrow b$

$S' \rightarrow S$ not numbered as it wasn't a part of the original grammar.

CLR(1) parsing table :

(looking at the canonical collection of LR(1) items)

States	Action			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_6	S_7		5	
3	S_3	S_4		8	
4	r_{iii}	r_{iii}			
5			r_i		
6	S_6	S_7		9	
7			r_{iii}		
8	r_{ii}	r_{ii}			
9			r_{ii}		

Filling in the same way as was followed for LR(0) and SLR(1) parsers.

↓

shift & Goto → no difference.

Reduce → only fill in those columns as specified by look-ahead

LR(0) parsing table : 0 to 6 states : r in all

SLR(1) parsing table : 0 to 6 states : r in FOLLOW

CLR(1) parsing table : 0 to 9 states : r in look-ahead

Summary : construction of CLR(1) parsing table

LALR(1) parser

outcome : conversion from CLR(1) to LALR(1) parsing table.

canonical collection of LR(1) items :

If we merge the states which are similar wrt to the LR(0) items, but different because of the look aheads, the no. of states in the resulting collection will be equal to that of the canonical collection of LR(0) items.

↳ The parsing table for that is known a LALR(1) or LOOK-Ahead LR(1) parsing table.

No. of states in CLR(1) > LR(0) = SLR(1) = LALR(1)

Taking canonical collection of LR(1) items

old state	new state	
I_3, I_6	I_{36}	} (merging of states)
I_4, I_7	I_{47}	
I_8, I_9	I_{89}	

LR(1) table →

states	Action a	Action b	Action $	GOTO A	GOTO S
0	S_{36}	S_{47}		2	1
1			✓		
2	S_{36}	S_{47}		5	
36	S_{36}	S_{47}		89	
47	r_{iii}	r_{iii}	r_{iii}		
5			r_1		
36	S_{36}	S_{47}		89	
47			r_{iii}		
89	r_{ii}	r_{ii}			
89			r_{ii}		

remove redundant states →

states	Action a	Action b	Action $	GOTO A	GOTO S
0	S_{36}	S_{47}		2	1
1			✓		
2	S_{36}	S_{47}		5	
36	S_{36}	S_{47}		89	
47	r_{iii}	r_{iii}	r_{iii}		
5			r_1		
89	r_{ii}	r_{ii}	r_{ii}		

To merge reduced states, just perform union

Summary : CLR(1) → LALR(1) pt ✓

__/__/__

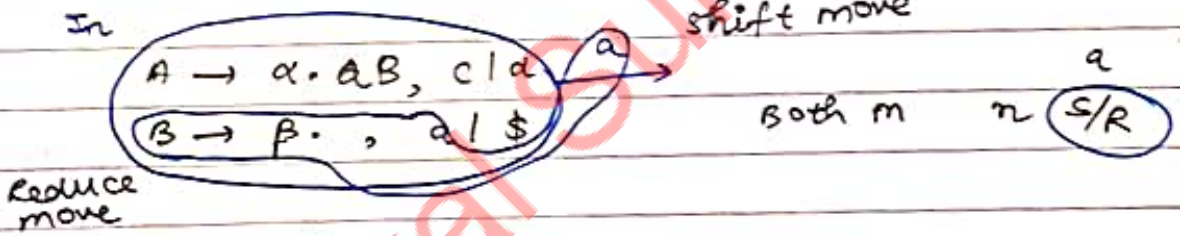
Conflicts in CLR(1) and LALR(1) Parsers

outcome :

- ✓ S-R and R-R conflicts in CLR(1) and LALR(1) parsing tables.
- ✓ Difference between CLR(1) and LALR(1) parsers wrt R-R conflict.

Conflicts wrt LR(1) Items

Shift-Reduce conflict :



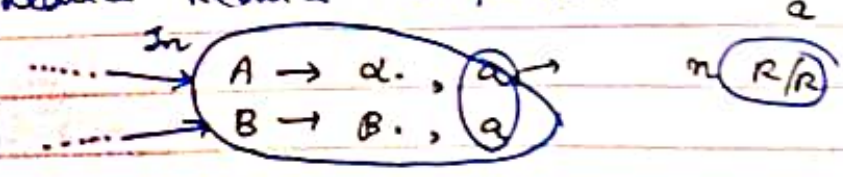
we generally use :

- ✓ uppercase letters for non-terminals
- ✓ lowercase letters for terminals.

Then why notations like α, β, γ ?

↳ when we are concerned about the presence of the element rather than the type of it, that is, we don't want to know whether the element is a terminal or non-terminal, rather we want the element to be in its place → In those cases, we use symbols like $\alpha, \beta, \gamma \ldots$

Reduce-Reduce conflict :



Any grammar with such situations is neither CLR(1) nor LALR(1).

Reduce - Reduce conflict

I_x { $A \to \alpha .,\ a$; $B \to \beta .,\ b$ } I_y { $A \to \alpha .,\ b$; $B \to \beta .,\ a$ } } no conflict for CLR(1)

↓ merge

I_{xy} { $A \to \alpha .,\ a/b$; $B \to \beta .,\ a/b$ } } But conflict for LALR(1) !

Keypoint:

No S-R conflict in CLR(1) $\Rightarrow$ No S-R conflict in LALR(1)

No R-R conflict in CLR(1) $\not\Rightarrow$ No R-R conflict in LALR(1)

Summary:

✓ S-R, R-R conflicts ✓ CLR(1) & LALR(1) (just RR)

Determining the type of grammar - set 6

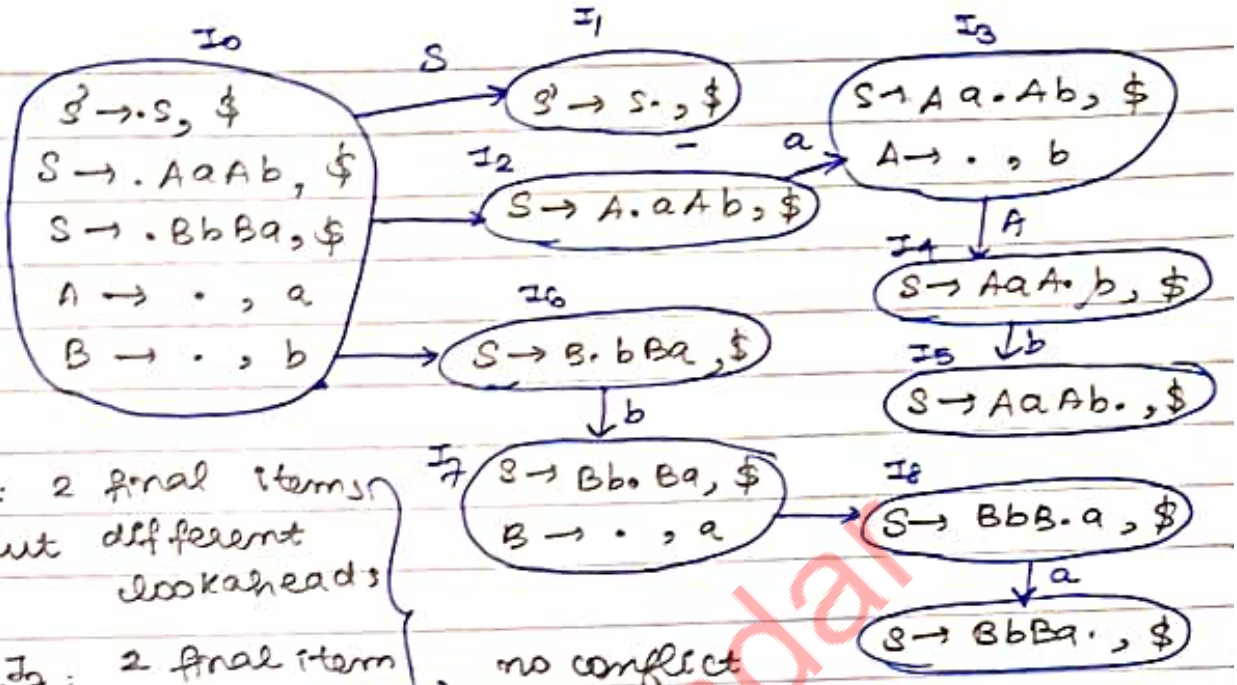
Outcome: A solved problems on how to determine the type of a given grammar.

Q1: Determine whether the following grammar is CLR(1) or LALR(1):

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

Augmented grammar →

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



I0: 2 final items but different lookaheads;

I3, I7: 2 final item with only GOTO moves, no shift moves

→ no conflict

CLR(1) ✓

Also, there are no states which we can merge with LR(0) items → so, CLR(1) and LALR(1) parsing tables will be the same for this grammar →

CLR(1)	LALR(1)
✓	✓

Q2: Determine whether grammar is CLR(1) & LALR(1):

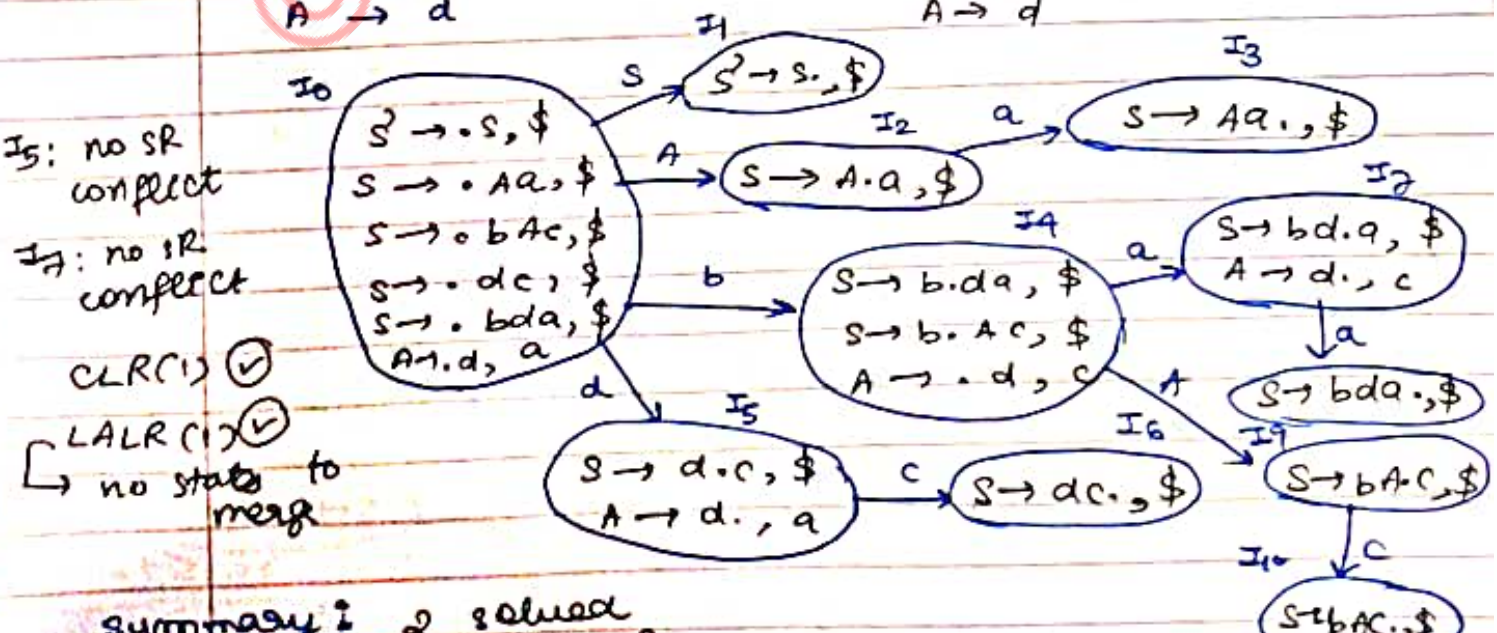
$S \to Aa \mid bAc \mid dc \mid bda$

$A \to d$

$S' \to S$

$S \to Aa \mid bAc \mid dc \mid bda$

$A \to d$



I5: no SR conflict

I7: no SR conflict

CLR(1) ✓

LALR(1) ✓

↳ no states to merge

summary: 2 solved problems

Determining the type of grammar - set 7

outcome : solved problem on how to determine the type of a given grammar.

Q1 Determine whether the following grammar is LL(1) / LR(0) / SLR(1) / CLR(1) / LALR(1) :

FIRST		FOLLOW
{d, b, $}	S → Aa | bAc | Bc | bBa	{$}
{d}	A → d	{a, c}
{d}	B → d	{c, a}

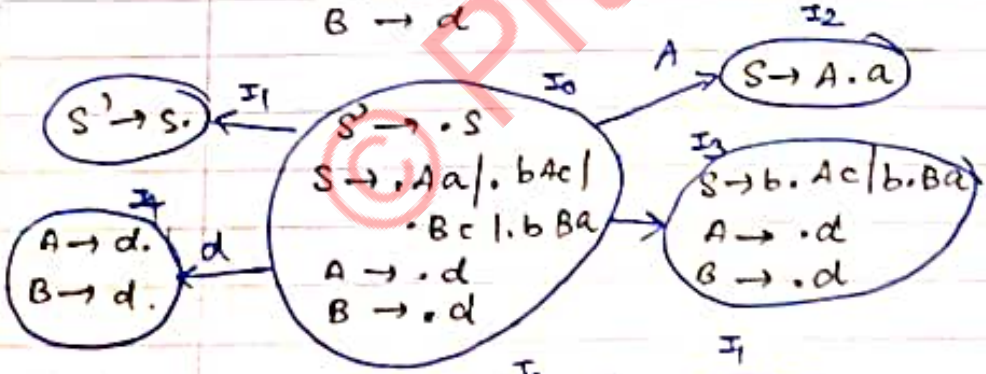
	a	b	c	d	$
S	,	S → bAc S → bBa		S → Aa S → Bc	
A				A → d	
B				B → d	

⊗ not an LL(1) grammar

S' → S
S → Aa | bAc | Bc | bBa
A → d
B → d

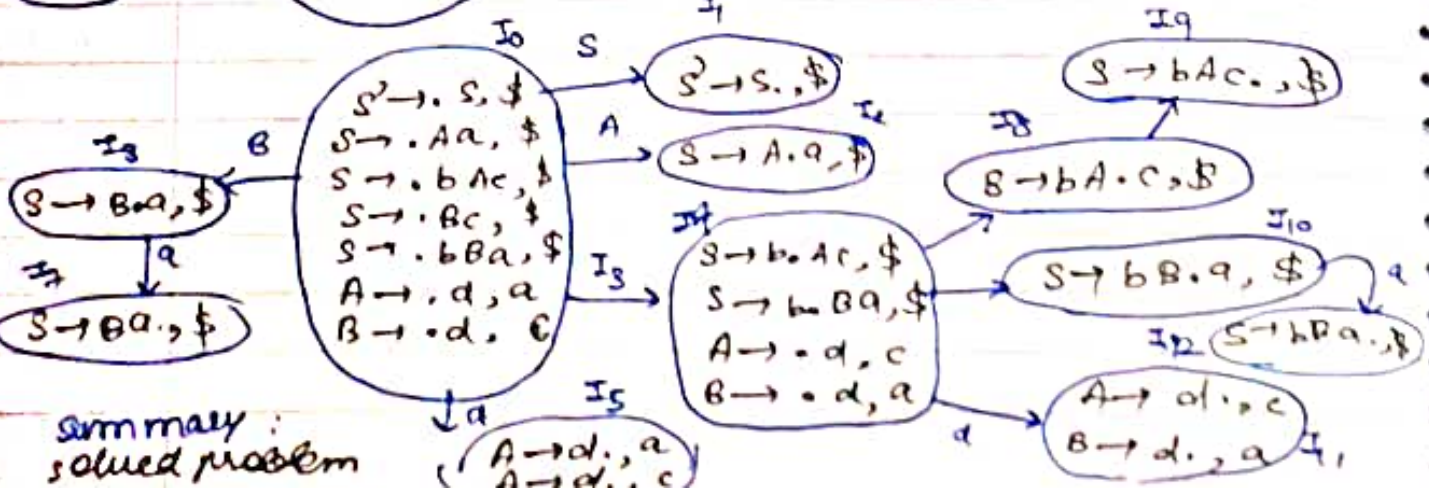
CLR(1) ✓

I5 I11 RR conflict
LALR(1) ✗



IA : RR conflict in LR(0) ⊗

FOLLOW(A) ∩ FOLLOW(B) = {a, c} ≠ ϕ
RR in SLR(1) also ⊗



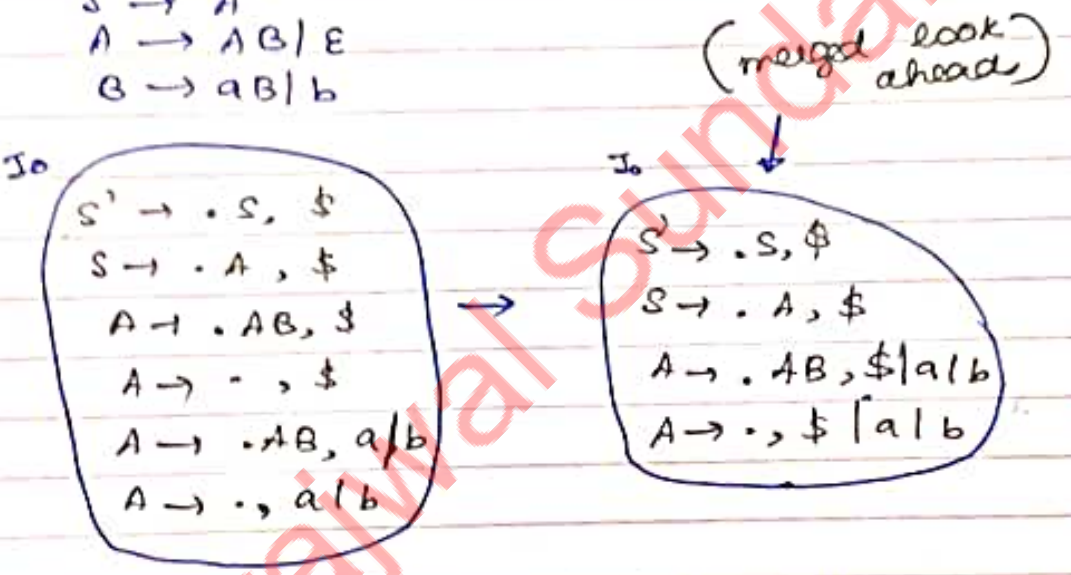
summary : solved problem

Interesting Closure Representation of LR(1) Items

outcome: canonical collection of LR(1) items with merged look-aheads

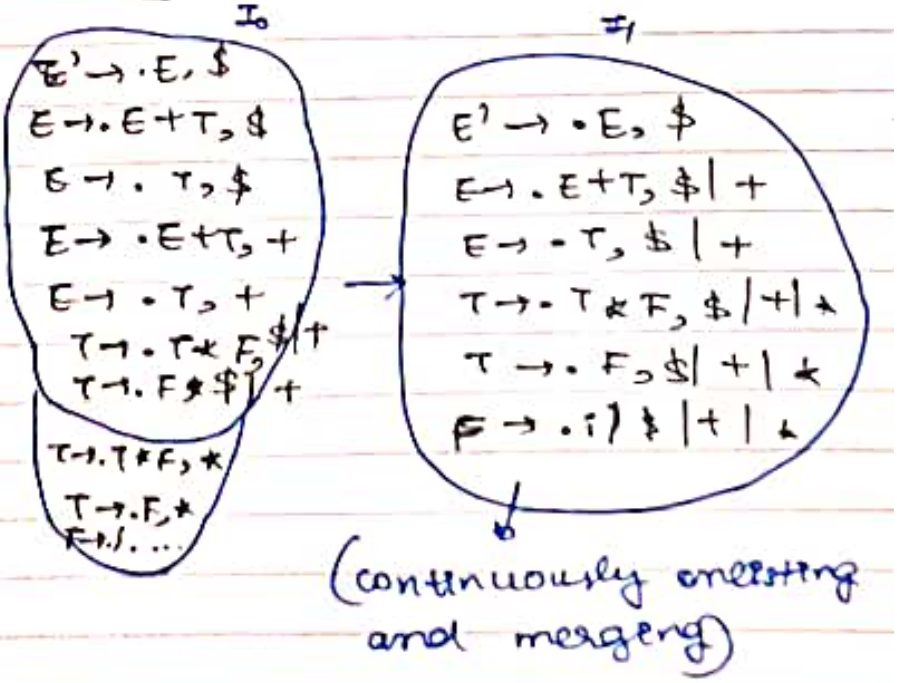
Closure Representation of LR(1) items:

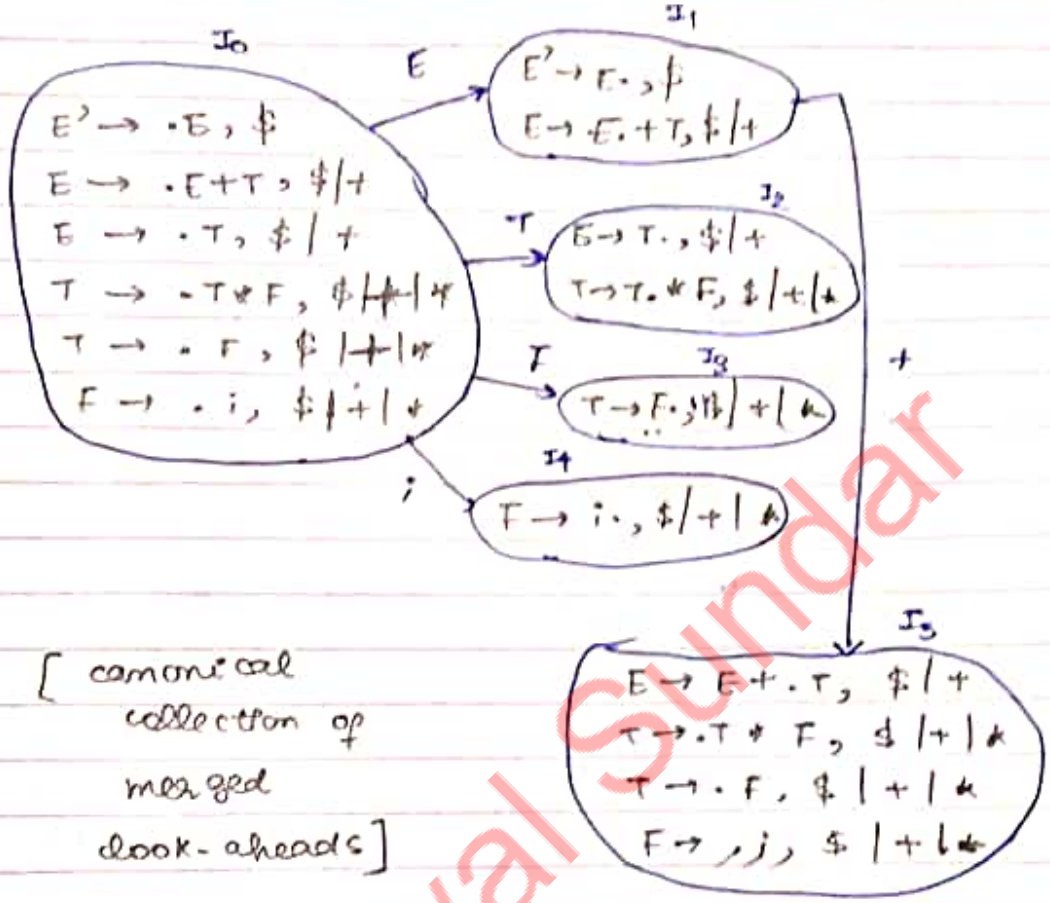
$S' \rightarrow S$
$S \rightarrow A$
$A \rightarrow AB \mid \varepsilon$
$B \rightarrow aB \mid b$



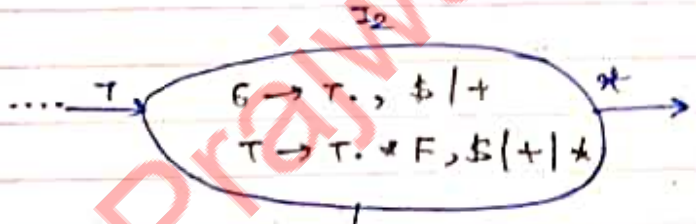
canonical collection of LR(1) items with merged look-aheads:

$E' \rightarrow G$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow i$





[canonical collection of merged look-aheads]



This might look like an S-R conflict, but it is NOT.

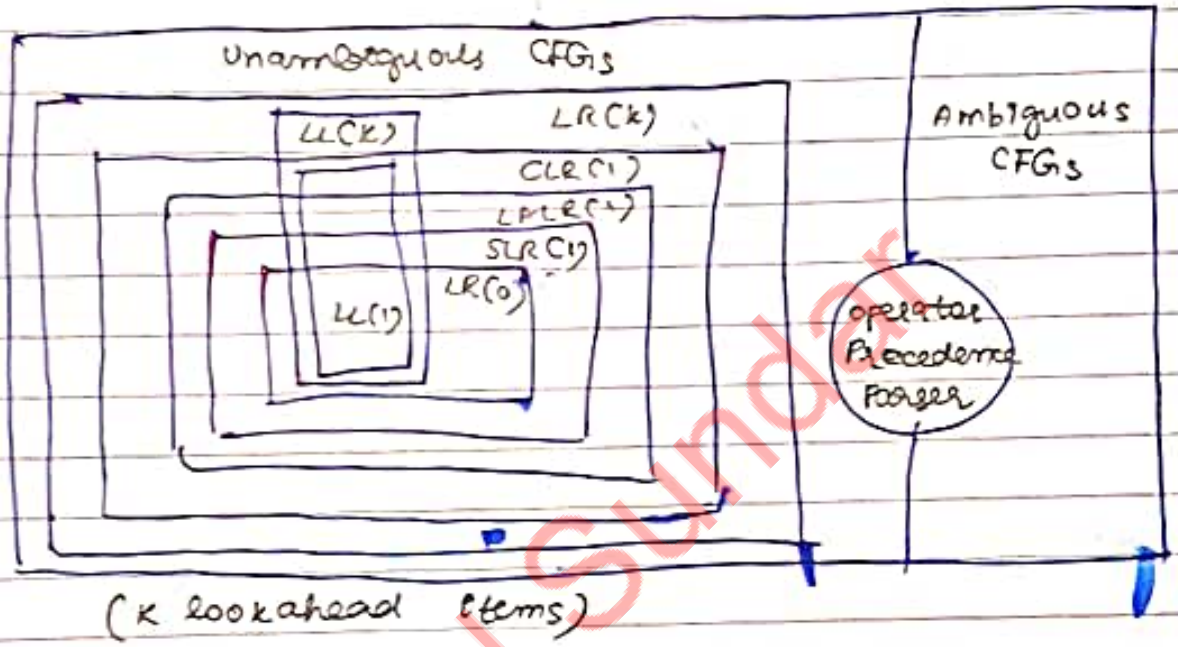
↓

- E → T will be a reduction rule underneath $ and +. Whereas the shift move will be entered underneath *.
- not a conflict

Summary: canonical collection of LR(1) items with merged look-aheads.

CFGs - The Big Picture

outcome : Relationship of all context Free Grammars.



summary : Relationship of all CFGs.

CFGs - solved Problems (set-1)

outcome : 4 solved problems on CFGs

[LL(1) < SLR(1) < LALR(1) < CLR(1)]

Q1: which of the following statements is true? (GATE 1998)

wrt power

X (A) SLR > LALR

X (B) LALR > LR (canonical)

✓ (C) CLR > LALR

X (D) SLR = CLR = LALR

Q2: which of the following statements is false? (GATE 2001)

✓ F (a) Unambiguous grammar has same leftmost & rightmost derivation.

T (b) An LL(1) parser is a top-down parser.

T (c) LALR is more powerful than SLR.

T (d) An ambiguous grammar can never be LR(k) ∀ any k.

Q3: (GATE 2003) considering the SLR(1) and LALR(1) parsing tables of a CFG, find out the false statement.

(A) GOTO of both tables may be different.
(B) shift entries are identical in both tables.
(C) Reduce entries in tables may be different.
(D) Error entries in tables may be different.

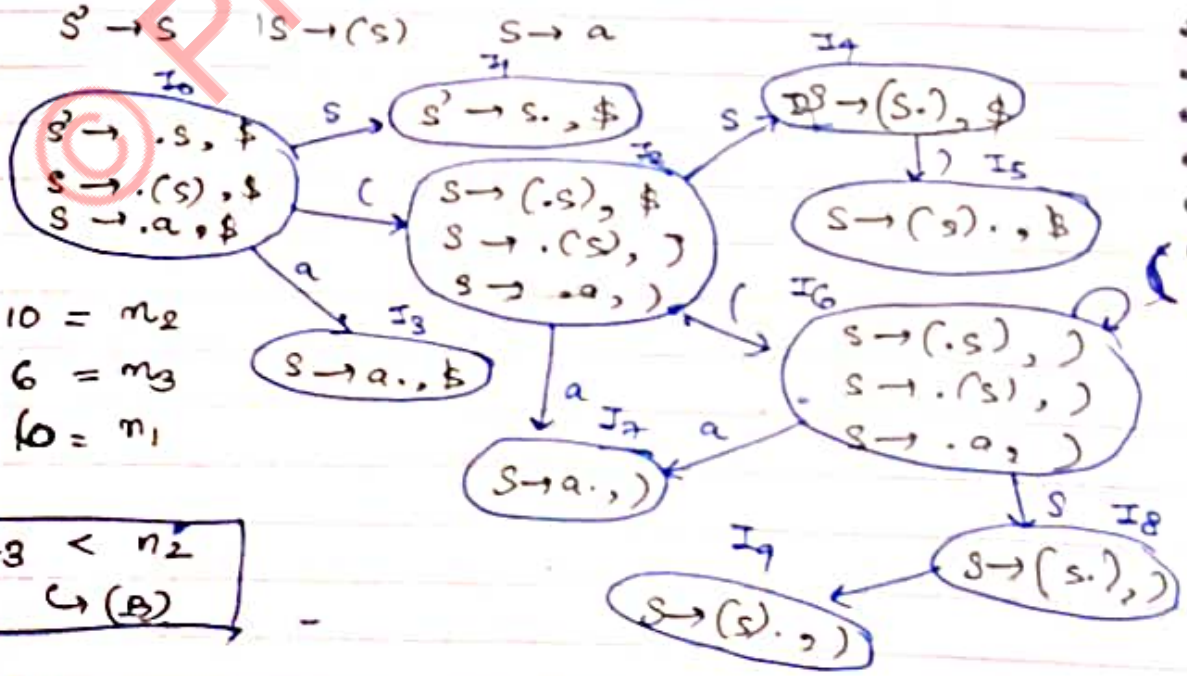
* GOTO → always same
* SHIFT → always same
* REDUCE → may / may not be same
* ERROR → may / may not be same

Q4: consider the grammar: $S \to (S) \mid a$

Let the number of states in SLR(1), LR(1) and LALR(1) parsers for the grammar be n_1, n_2 and n_3 respectively. The following relationship holds good:

(A) $n_1 < n_2 < n_3$ (B) $n_1 = n_3 < n_2$
(C) $n_1 = n_2 < n_3$ (D) $n_1 \geq n_2 \geq n_3$

Wkt CLR(0): $I_3 \equiv I_8$, $I_4 \equiv I_8$, $I_2 \equiv I_6$, $I_5 \equiv I_9$



CLR(1) → 10 = n_2
LALR(1) → 6 = n_3
SLR(1) → 6 = n_1

$n_1 = n_3 < n_2$
↳ (B)

summary: 4 solved problems on CFGs

CFGs - solved problems - set 2

outcome : 4 solved problems on CFGs.

Q1: (GATE 2003) consider the grammar shown below:

$S \rightarrow CC$
$C \rightarrow cC \mid d$

(A) LL(1) (B) SLR(1), not LL(1)
(C) LALR(1), not SLR(1) (D) LR(1), not LALR(1)

FIRST		FOLLOW
{c,d}	$S \rightarrow CC$	{$}
{c,d}	$C \rightarrow cC \mid d$	{c,d,$}

	c	d	$
S	S→CC	S→CC	
C	C→cC	C→d	

✓ LL(1) grammar

Q2: (GATE 2006) consider the following grammar

$S \rightarrow S * E \mid E$
$E \rightarrow F + E \mid F$
$F \rightarrow id$

consider the following LR(0) items corresponding to the grammar :

(i) $S \rightarrow S * \cdot E$
(ii) $E \rightarrow F \cdot + E$
(iii) $E \rightarrow F + \cdot E$

which two items will appear in the same state of the canonical sets of items for the grammar ?

(A) (i) and (ii)
(B) (ii) and (iii)
(C) (i) and (iii)
(D) None of the above

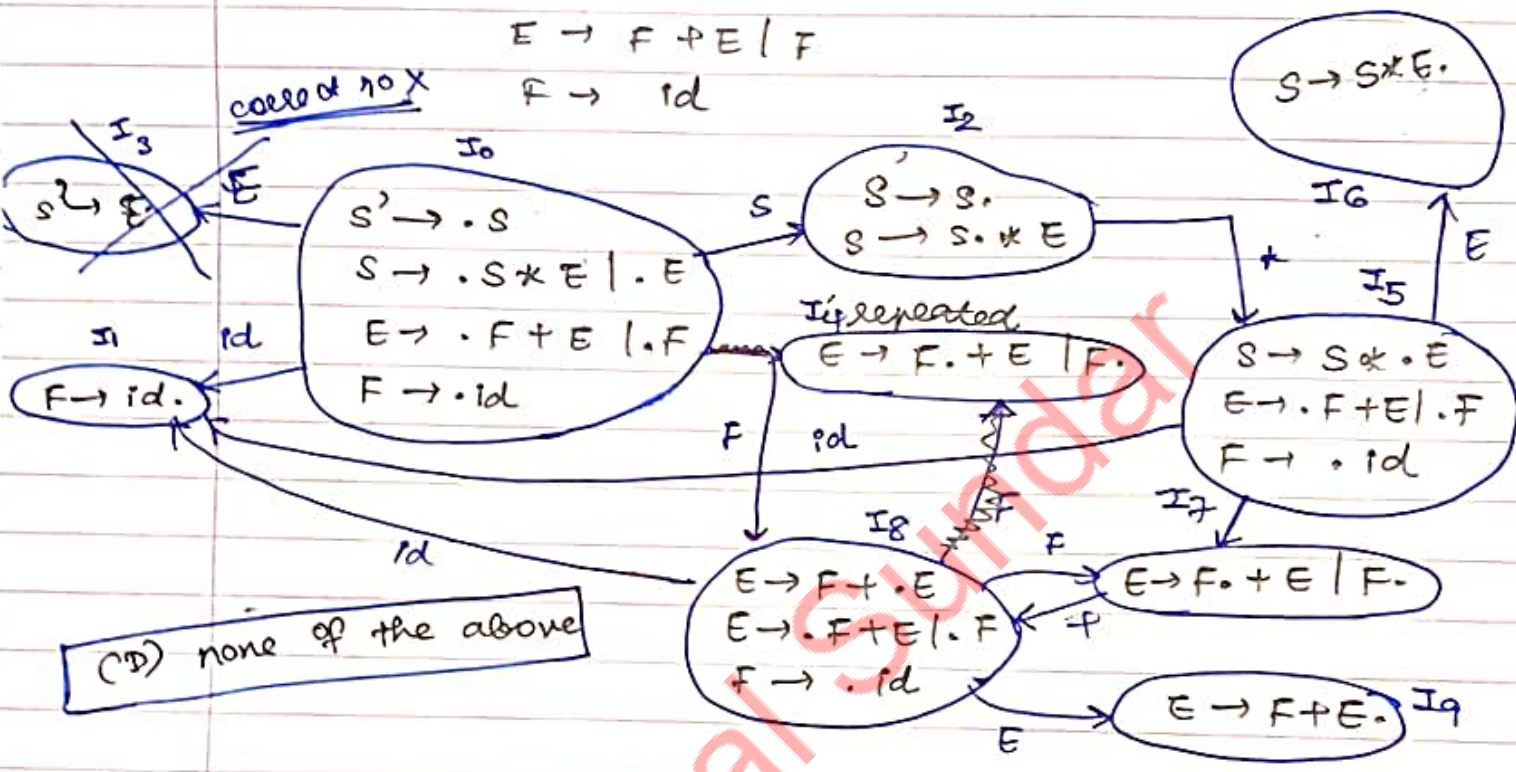
Augmented grammar :

$S' \rightarrow S$

$S \rightarrow S * E \mid E$

$E \rightarrow F + E \mid F$

$F \rightarrow id$



Q3: consider the grammar

$E \rightarrow E + n \mid E * n \mid n$

For a sentence $n + n * n$, the handles in the right-sentential form of reduction are :

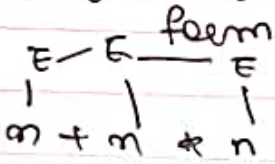
(A) n, $E + n$, $E + n * n$

(B) n, $E + n$, $E + E * n$

(C) n, $n + n$, $n + n * n$

(D) n, $E + n$, $E * n$

right-sentential form



Q4: consider the grammar given below :

$S \rightarrow SS \mid a \mid \varepsilon$

the no. of inadequate states in the DFA with LR(0) items are :

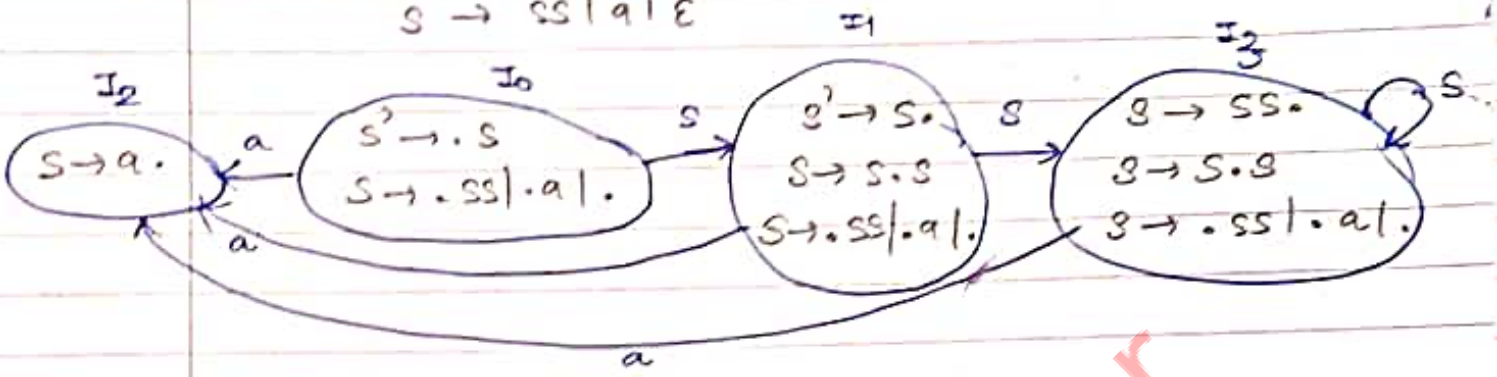
(A) 1 (B) 2 (C) 3 (D) 4

LR(0) items →

Augmented grammar :

$S' \to S$

$S \to SS \mid a \mid \varepsilon$



$S \to \cdot \xrightarrow{a}$ in I_0, I_1, I_3 ③ S-R conflicts

also $S \to SS\cdot,\ S \to \cdot$ in I_3 ① R-R conflict

No. of inadequate states = ~~(3)~~ 4

Usefulness of Ambiguous Grammars

outcome : Advantages of using ambiguous Grammars.

Ambiguous Grammar vs Unambiguous Grammar :

(X) $E \to E+E \mid E*E \mid id$

↳ (associativity, precedence not specified)

(Y)
$G \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

} (unambiguous version of the above)

For X: ✓ productions are less. } flexible
✓ simple to understand }

For Y: ✓ associativity and precedence is fixed → not flexible

To derive id

X : E → id

(1)

Y : E → T → F → id

(3)

(steps of reduction)

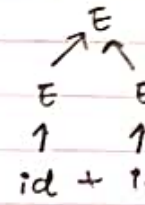
However ↓

X : causes numerous conflicts while parsing

↓

These can be resolved by taking meaningful decisions.

id + id + id



E → E+E·
E → E· +E

S/R conflict

↓

choose R due to left associativity

id + id * id

E → E+E·
E → E· * E

S/R conflict

choose S due to higher multiplication precedence

Y → Yet
A → Another
C → compiler
C → compiler

⇒ LALR(1) parser for all grammar [ambiguous & unambiguous]

S/R favoured shift

R/R favoured first reduce

: (conflicts can be resolved by taking meaningful decisions)

Summary :

Advantages of using Ambiguous Grammar