COMPILER DESIGN

MODULE-3

BOTTOM UP PARSING

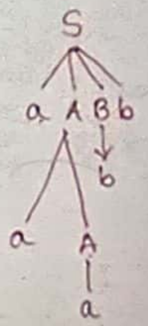
Bottom-Up Parsing:

- An attempt to reduce the input string 'w' to the start symbol of a grammar by tracing out right most derivations of 'w' in reverse.

For eg:- Consider the grammar, $S \rightarrow aABb$ and the input string is 'aaabb'

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$



aaabb
aaAbb
aAbb
aABb
S

reduction.

$S \rightarrow aABb \rightarrow aAbb \rightarrow aaAbb \rightarrow aaabb$

Eg:- Shift Reduce Parser, Operator Precedance, LR parser.

Handle

Handle of a string is a substring that matches a right hand side of a production & it is reduced to nonterminal on the LHS of production

Handle Pruning

A rightmost derivation in reverse is obtained by Handle Pruning.

For eg:- Consider the grammar,

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

Right Sentential form	Handle	Reducing Productions
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

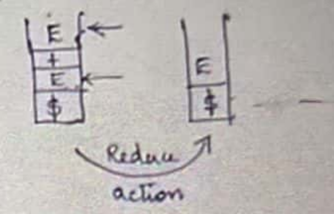
Shift Reduce Parsing

- Bottom up Parsing

Stack	Input
$	w$
⋮	⋮
$S	$

Stack	i/p buffer
$	abc$
⋮	⋮
$S	$

$E \rightarrow E+E$



Actions

1) Shift → The next input symbol is shifted onto the top of the stack.

2) Reduce → 'β' is reduced to left hand side of the production [A → β] when handle 'β' appears on top of the stack.

3) Accept → Announces successful completion of parsing.

4) Error → Discover a syntax error has occurred and calls an error recovery routine.

Eg: 2

Perform Shift Reduce parsing of i/p string, $w = cdcd$ using the grammar

$S \longrightarrow CC$

$C \longrightarrow cC / d$

Stack	i/p buffer	Action
\$	cdcd\$	Shift
\$c	dcd\$	Shift
\$cd	cd\$	Reduce by $C \to d$
\$cC	cd\$	Reduce by $C \to cC$
\$C	cd\$	Shift
\$Cc	d\$	Shift
\$Ccd	\$	Reduce by $C \to d$
\$CcC	\$	Reduce by $C \to cC$
\$CC	\$	Reduce by $S \to CC$
\$S	\$	Accept

using the grammar

$E \longrightarrow E+T / T$

$T \longrightarrow T*F / F$

33315:2022080912:55

$F \longrightarrow (E) / id$.

Stack	Input Buffer	Action
$	id * id $	Shift
$ id	* id $	Reduce by F → id
$ F	* id $	Reduce by T → F
$ T	* id $	Shift
$ T *	id $	Shift
$ T * id	$	Reduce by F → id
$ T * F	$	Reduce by T → T * F
$ T	$	Reduce by E → T.
$ E	$	Accept

$ E *

33315:2022080913:20

Conflicts in Shift Reduce Parsing:

1) Shift / Reduce Conflicts

Every SR parser can reach a configuration knowing the stack and the next input symbols, cannot decide whether to shift or reduce.

2) Reduce / Reduce Conflicts

Every SR parser can reach a configuration, knowing the entire stack and the next 'k' input symbols, it can't decide which of several reductions to make.

OPERATOR PRECEDENCE PARSING

33315:2022080913:29

* Bottom-up Parser.
* Uses Operator Grammars.

Operator Grammars.

In this grammar, no production rule can have
- 'ε' at the right hand side
- two adjacent non-terminals at the right side.

For eg:-

1) $E \longrightarrow AB$
$A \longrightarrow a$
$B \longrightarrow b$

Not operator grammar

2) $E \longrightarrow EOE$
$E \longrightarrow id$
$O \longrightarrow + / *$

Not Operator grammar

3) $E \longrightarrow E+E$
$E \longrightarrow E*E$
$E \longrightarrow id$

Operator grammar.

Precedance Relations

* 3 precedence relations between certain pair of terminals

$a \lessdot b$ → a yields precedence to 'b'

$a \doteq b$ → a has same precedence as 'b'

$a \gtrdot b$ → a takes precedence over 'b'

Precedence relation is used to find the handle of a right sentential form with $\lessdot$ marking the left end and $\gtrdot$ marking the right end.

ie we insert the precedence relation between the pair of terminals

$a \lessdot b \rightarrow$ a yields precedence to 'b'

$a \doteq b \rightarrow$ a has same precedence as 'b'

$a \gtrdot b \rightarrow$ a takes precedence over 'b'

Precedence relation is used to find the handle of a right sentential form with $\lessdot$ marking the left end and $\gtrdot$ marking the right end

ie We insert the precedence relation between the pair of terminals

Steps to perform Operator Precedence Parsing:

1) Check whether the given grammar is Operator grammar or not

2) Construct Operator precedence relation table.

3) Parse the given input string using Operator precedence Parsing algorithm

4) Generate Parse tree.

I Check whether the given grammar is Operator grammar or not. If not convert it into Operator grammar.

Here in this problem, the first production is not in operator grammar (adjacent non terminals are present). So we can rewrite that production as $E \rightarrow E+E \,/\, E*E \,/\, id$. (Replace A with its RHS)

Now the grammar is Operator grammar.

II Construct the precedence relation table.

	id	+	*	$
id	–	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	Accept

Rules

1) id has more precedence than all other terminals.
2) $ has lower precedence than all other terminals.
3) If both operators are having equal precedence then use associativity rule.

+ ⋗ + + ⋗ +

* ⋗ * * ⋗ *

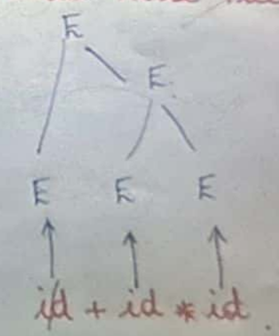
33315:2022080914:01

Stack	Input	Action
$	id+id*id$	Shift
$~~id~~	+id*id$	Reduce by E→id
$	+id*id$	Shift
$+	id*id$	Shift
$+~~id~~	*id$	Reduce by E→id
$+	*id$	Shift
$+*	id$	Shift
$+~~*id~~	$	Reduce by E→id
$+~~*~~	$	Reduce by E→E*E
$~~+~~	$	Reduce by E→E+E
$	$	Accept

Precedence relation table

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

IV. Generate Parse Tree



15:2022080914:16

Disadvantage of Operator Precedence Parsing:

1) If the no: of operators in the given grammar is 'n' then the size of precedence relation table is $O(n^2)$

www.trio.education

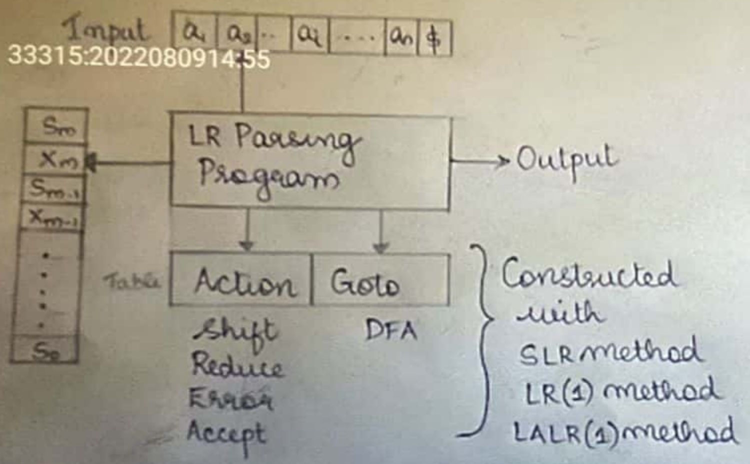
LR Parsing

* Most efficient method of Bottom-up Parsing
* Used to parse large class of context-free grammar.

In LR(k) parsing,

L stands for left to right scanning of input symbols.
R stands for right constructing rightmost derivation in reverse.
k stands for number of input symbols of lookahead that are used in making parsing decision.

Model of an LR Parser:



* Input Buffer → The parsing program reads characters from an i/p buffer one at a time.

* Stack → Parsing pgm uses a stack to store strings of the form $S_0 X_1 S_1 X_2 S_2 X_3 \ldots X_m S_m$ where S_m is on the top. X_i is a grammar symbol and S_i is the state. Combination of state symbol on the stack top and current input symbol are used to index parsing table & determine the parsing decisi…

* Parsing Table

Consists of two parts

1) Action
2) Goto

Action :- This function takes a state i and a terminal 'a' as arguments. The value of ACTION[i, a] can have any one of the four forms.

a) Shift j where j is a state
b) Reduce by a grammar production, $A \longrightarrow \beta$
c) Accept
d) Error.

Goto :-

This function takes a state and grammar symbol as argument and produces a state as output. ie $GOTO[I_i, A] = I_j$ where 'j' is a state.

Different types of LR parsers

1) Simple LR (SLR)
2) Canonical LR (CLR)
3) Lookahead LR (LALR)

All LR parsers use the same Parsing algorithm but the difference is only in terms of construction of LR Parsing tables.

SLR parser	LR(1) Parser/ Canonical LR Parser	LALR (1) Parser
1) Works on small class of grammars	1) Works on complete set of LR(1) grammar	1) Works on intermediate size of grammar
2) Few no. of states hence very small table	2) Generates large table and large number of states.	2) No. of states are same as in SLR(1)
3) Simple & fast construction	3) Slow construction	3) Intermediate in power and cost between other 2 parsers.
4) Least powerful	4) Most powerful and most expensive	

33315:2022080915:20

Constructing SLR parsing Table

* LR parser using SLR parsing table is called an SLR parser
* A grammar for which an SLR parser can be constructed is an SLR grammar
* **LR(0) Item**

An LR(0) item of a grammar G is a production of G with a dot at the some position of the right side

Eg:- for the productions $A \longrightarrow aBb$.

Possible LR(0) items are
$A \longrightarrow .aBb$
$A \longrightarrow a.Bb$
$A \longrightarrow aB.b$
$A \longrightarrow aBb.$

An item shows how much of a production we have seen till the current point in the parsing procedure.

* A production rule of the form $A \longrightarrow \epsilon$ yields only one item $A \longrightarrow .$
* A collection of sets of LR(0) items (the canonical LR(0) collection) is the basis for constructing SLR parsers

To construct the canonical LR(0) collection for a grammar we define
* Augmented grammar
* Two functions - closure and goto

5:2022080915:21

Augmented Grammar (G')

A grammar G with a new production rule $S' \longrightarrow S$ where S' is the new starting symbol.

$G' = G \cup \{S' \rightarrow S\}$

This is done to signal to the parser when the parsing should stop to announce acceptance of the input. $S' \rightarrow S$

Closure

If I is the set of LR(0) items, then closure (I) is the set of items constructed from I by the two rules. I

1) Initially, every LR(0) item in I is added to closure (I).
2) If $A \longrightarrow \alpha . B\beta$ is in closure (I) and $B \rightarrow \gamma$ is a production rule of G then $B \longrightarrow .\gamma$ will be in closure (I). $A \rightarrow a.B$

* Apply this rule untill no more new LR(0) items can be added to closure (I). $B \rightarrow .b$

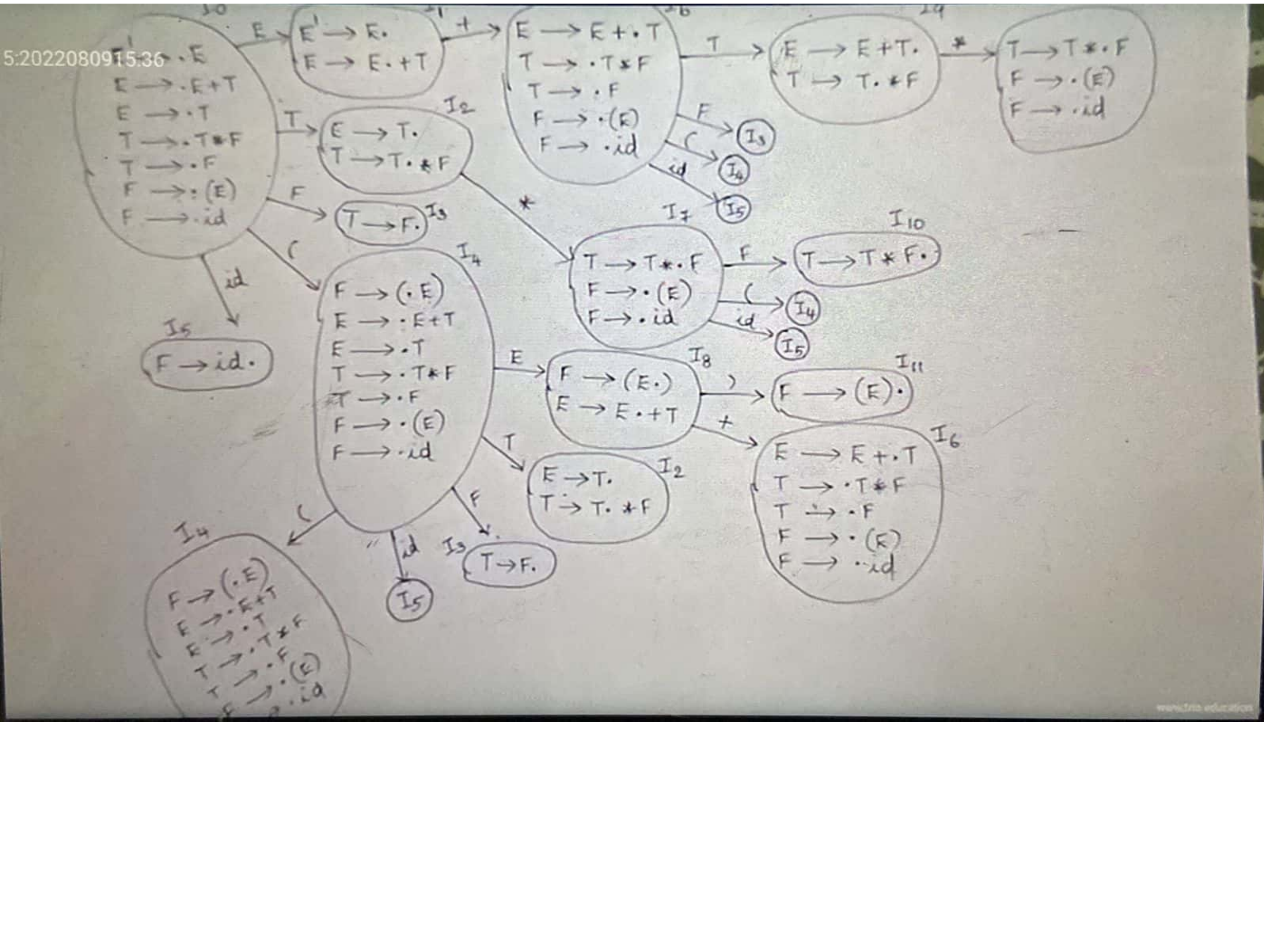
Q) Construct the canonical LR(0) collection of items and SLR parsing table for the given grammar.

$$\left.\begin{array}{l} E \longrightarrow E+T \\ E \longrightarrow T \\ T \longrightarrow T*F \\ T \longrightarrow F \\ F \longrightarrow (E) \\ F \longrightarrow id \end{array}\right\} G$$

1) Augmented grammar can be written as

$$\left.\begin{array}{l} E' \longrightarrow E \\ E \longrightarrow E+T \\ E \longrightarrow T \\ T \longrightarrow T*F \\ T \longrightarrow F \\ F \longrightarrow (E) \\ F \longrightarrow id \end{array}\right\} G' = G \cup \{E' \rightarrow E\}$$

In order to construct the cannonical collection of LR(0) items for an augmented grammar, G' find closure $(E' \longrightarrow \cdot E)$


5:20220809 15:36
I0
E' → ·E
E → ·E+T
E → ·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id
I1
E' → E·
E → E·+T
I6
E → E+·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id
I9
E → E+T·
T → T·*F
T → T*·F
F → ·(E)
F → ·id
I2
E → T·
T → T·*F
I3
T → F·
I7
T → T*·F
F → ·(E)
F → ·id
I10
T → T*F·
I4
F → (·E)
E → ·E+T
E → ·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id
I5
F → id·
I8
F → (E·)
E → E·+T
I11
F → (E)·

Constructing SLR parsing table

Input: An augmented grammar

Output: SLR parsing table functions action and goto for G'

Method:

1) Construct $C = \{I_0, I_1, \dots I_n\}$, the collection set of LR(0) items for G'.

2) State 'i' is constructed from I_i. The parsing actions are determined as follows:

   (a) If $[A \rightarrow \alpha \cdot a\beta]$ is in I_i and goto $(I_i, a) = I_j$ then set action$[i, a]$ to shift j. where 'a' is a terminal

   (b) If $[A \rightarrow \alpha \cdot]$ is in I_i then set action$[i, a]$ to "reduce $\underset{r_1}{A} \rightarrow \alpha$" for all 'a' in FOLLOW(A)

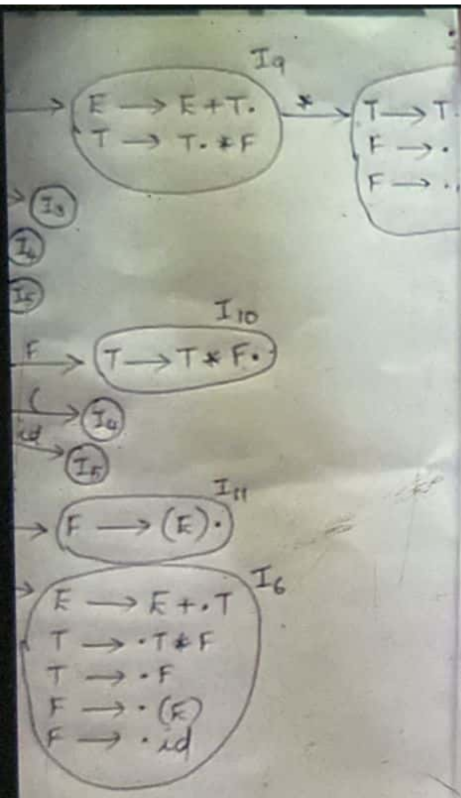
   (c) If $[S' \rightarrow S \cdot]$ is in I_i, then set action$[i, \$]$ to "accept"

   If any conflicts are generated by above rules, then the grammar is not SLR(1).

3) For all nonterminals, A, if GOTO$(I_i, A) = I_j$ then GOTO$[I, A] = j$

4) All entries defined by rules (2) & (3) are made "error"

5) The initial state is constructed from set of items containing $[S' \rightarrow S]$



SLR PARSING TABLE

	ACTION						GOTO		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5				S4		8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

a) Number the given gramma

1) $E \to E+T$
2) $E \to T$
3) $T \to T*F$
4) $T \to F$
5) $F \to (E)$
6) $F \to id$

b) Find FOLLOW of nonter
the given grammar

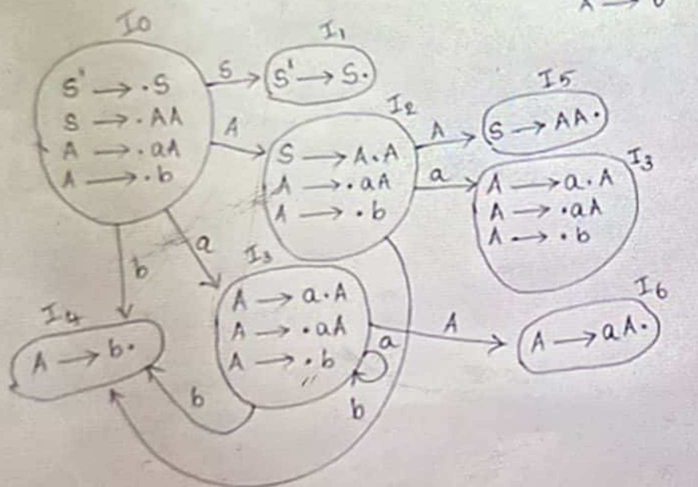
FOLLOW (E) = {$,), +}
FOLLOW (T) = {$,), +,
FOLLOW (F) = {*, +,),

Q2) Construct the SLR parsing table for the given grammar. OR Construct the Canonical LR(0) collection of items for the given grammar.

$S \rightarrow AA$

$A \rightarrow aA \mid b$

1) Construct Augmented grammar, G' : $S' \rightarrow S$, $A \rightarrow aA$, $S \rightarrow AA$, $A \rightarrow b$



SLR Parsing Table

	Action			Goto	
State	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

1) Number the productions

1) $S \rightarrow AA$
2) $A \rightarrow aA$
3) $A \rightarrow b$

2) FOLLOW of

$S = \{\$\}$

$A = \{\$, a, b\}$

Q8) Construct the SLR parsing table for the given grammar.

OR

Construct the canonical LR(0) collection of items for the given grammar

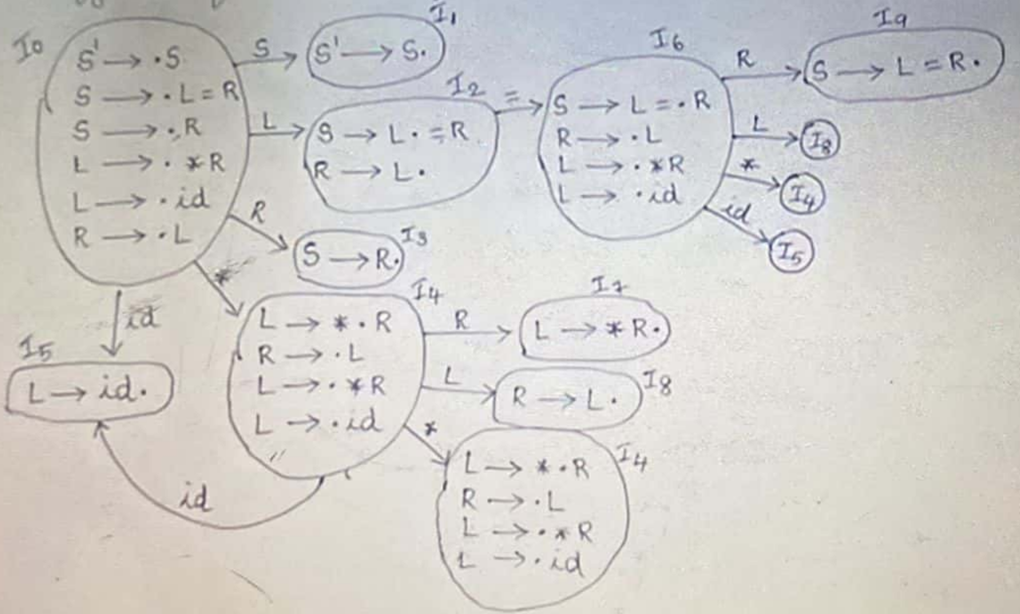
$$\left.\begin{array}{l} S \longrightarrow L = R \\ S \longrightarrow R \\ L \longrightarrow * R \\ L \longrightarrow id \\ R \longrightarrow L \end{array}\right\} G + \{S' \longrightarrow S\} \Big\} G'$$

Also identify a shift reduce conflicts in the LR(0) collection constructed above (KTU, APRIL 2018)

$S \rightarrow L = R$
$S \rightarrow R$
$L \rightarrow * R$
$L \rightarrow id$
$R \rightarrow L$

$G + \{S' \rightarrow S\}$ → G'

Also identify a shift reduce conflicts in the LR(0) collection constructed above (KTU, APRIL 2018)



SLR Parsing Table

	Action				Goto		
State	=	*	id	$	S	L	R
0		S4	S5		1	2	3
1				Accept			
2	S6/r5			r5			
3				r2			
4		S4	S5			8	7
5.	r4			r4			
6		S4	S5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

b) Find FOLLOW of all the nonterminals in the given grammar

FOLLOW(S) = { $ }

FOLLOW(L) = { = , $ }

FOLLOW(R) = { $, = }

a) Number the productions in the given grammar.

1) $S \rightarrow L = R$
2) $S \rightarrow R$
3) $L \rightarrow *R$
4) $L \rightarrow id$
5) $R \rightarrow L$

33315-2022081010:47

Conflicts in SLR Parser

1) Shift/Reduce Conflicts

If a state doesn't know whether it will make a shift operation or reduction for a terminal, we say that there is a shift/reduce conflict.

2) Reduce/Reduce Conflicts

If a state doesn't know whether it will make a reduction operation using the production rule 'i' or 'j' for a terminal, then we say that there is reduce/reduce conflict.

If the SLR parsing table of a grammar G has a conflict, then the grammar is not SLR grammar.

CANONICAL LR (CLR) PARSING TABLES

While constructing SLR parsing table for the grammar,

$S \rightarrow L = R$
$S \rightarrow R$
$L \rightarrow *R$
$L \rightarrow id$
$R \rightarrow L$

where in state 2 and i/p symbol is '=', the entry in the parsing table Action[2, =] = S6/r5 which leads to error or conflict. ie The reduction by 5th production is invalid when the input symbol is '='. Inorder to avoid these invalid reductions, it is required to carry more information in the state. Extra information is incorporated into state by redefining items to include terminal symbol as the second component. ie $\boxed{[A \rightarrow \alpha \cdot \beta, a]}$ where $A \rightarrow \alpha \cdot \beta$ is a production and 'a' is a terminal or \$. Such an object is called an LR(1) item.

'1' refers to length of second component called as lookahead of the item.

Here in CLR parser, an item of the form $[A \rightarrow \alpha \cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is 'a'.

$\boxed{\text{LR(1) item} = \text{LR(0) item} + \text{lookahead.}}$

Cannonical collection of Sets of LR(1) items

The construction of the cannonical collection of the sets of LR(1) items are similar to the construction of cannonical collection of the sets of LR(0) items except some difference in the goto and closure function.

Goto

If I is a set of LR(1) items and x is a grammar symbol then goto (I, x) is defined as

- If $A \to \alpha \cdot X\beta, a$ is in I then every item in closure ($\{A \to \alpha X \cdot \beta, a\}$) will be in goto (I, x)

Closure (I)

- Every LR(1) item in I is in closure (I).
- If $A \to \alpha \cdot B\beta, a$ in closure (I) and $B \to \gamma$ is a production rule of G then $B \to \cdot\gamma, b$ will be in closure (I) for each terminal 'b' in FIRST(βa)

$A \to a \cdot B, \underline{b}$ FIRST(b)

$B \to \cdot a, \underline{b}$

$B \to a$

Q) Construct the canonical collection of set of LR(1) items for the given grammar

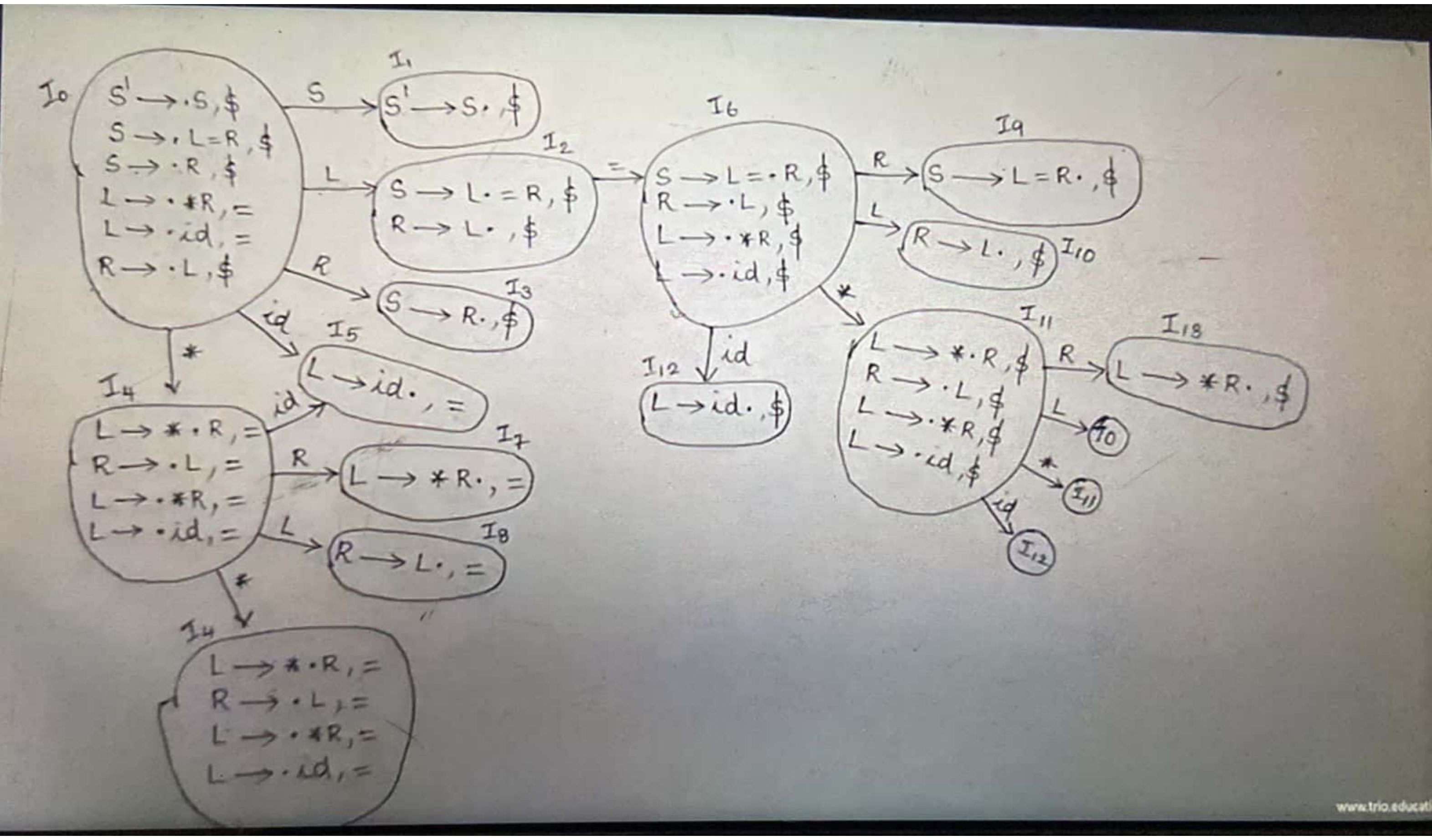
OR

Construct the CLR parsing table for the given grammar.

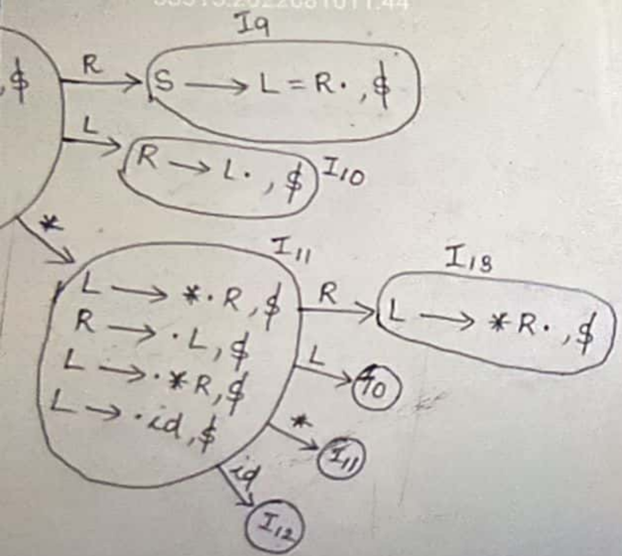
$S \longrightarrow L = R$
$S \longrightarrow R$
$L \longrightarrow * R$
$L \longrightarrow id$
$R \longrightarrow L$

Solution:- 1) Construct the Augmented grammar by adding the production $S' \longrightarrow S, \bullet$

2) Compute the closure of $(\{S' \longrightarrow \cdot S, \$\})$



I0
S' → ·S, $
S → ·L=R, $
S → ·R, $
L → ·*R, =
L → ·id, =
R → ·L, $
I1
S' → S·, $
I2
S → L·=R, $
R → L·, $
I3
S → R·, $
I4
L → *·R, =
R → ·L, =
L → ·*R, =
L → ·id, =
I5
L → id·, =
I6
S → L=·R, $
R → ·L, $
L → ·*R, $
L → ·id, $
I7
L → *R·, =
I8
R → L·, =
I9
S → L=R·, $
I10
R → L·, $
I11
L → *·R, $
R → ·L, $
L → ·*R, $
L → ·id, $
I12
L → id·, $
I13
L → *R·, $
S
L
R
id
*
=



CLR (Canonical LR) Parsing Table

	ACTION				GOTO		
State	=	*	id	$	S	L	R
0		S4	S5		1	2	3
1				Accept			
2	S6			r5			
3				r2			
4		S4	S5			8	7
5	r4						
6		S11	S12			10	9
7	r3						
8	r5						
9				r1			
10				r5		10	13
11		S11	S12	r4			
12				r3			
13							

1) Number ...

1) S → L ...
2) S → R
3) L → * ...
4) L → i ...
5) R → ...

Features of CLR(1) Parsing Table.

1) No: of States in Canonical LR(1) is increased compared to SLR due to lookaheads.

2) No: of reduce actions in CLR(1) is reduced so conflicts also get reduced.

3) Blank space in the table is increased & hence error detecting capability of CLR parser is also increased.

To reduce the size of CLR(1) parsing table, we can merge the states in the LR(1) sets which differs only in look aheads. Thus we go for constructing LALR parsing Table.

o functions are same
with same
ut different
in CL(1)

eg for CLR(1) parser,
d I_{11} are same
d component or
different. So merge
LALR.

4 11

$I_{12} = I_{512}$

$I_{13} = I_{713}$

$I_{10} = I_{810}$

reduce the size of LALR parsing tables.

LALR Parsing Table for the grammar

$S \to L = R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

Step 1:- CLR parsing table is ↓

	Action				Goto		
State	=	*	id	$	S	L	R
0		S4	S5		1	2	3
1				Accept			
2	S6			r5			
3				r2			
4		S4	S5			8	7
5	r4						
6		S11	S12			10	9
7	r3						
8	r5						
9				r1			
10				r5			
11		S11	S12			10	13
12				r4			
13				r3			

Step 1 Construct the set of LR(1) items and if no conflicts arise, merge set with common cores.

Step 2: Draw the table with combined states. ↓

	Action				Goto		
State	=	*	id	$	S	L	R
0		S_{411}	S_{512}		1	2	3
1				Accept			
2	S6			r5			
3				r2			
411		S_{411}	S_{512}			810	713
512	r4			r4			
6		S_{411}	S_{512}			810	9
713	r3			r3			
810	r5			r5			
9				r1			

LALR parsing Table