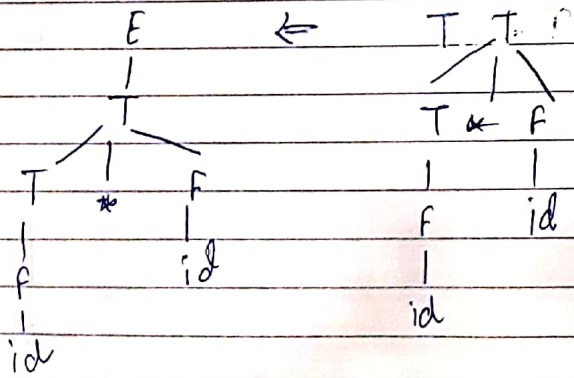
UNIT-III : SYNTAX ANALYSIS - I

Bottom-Up Parsing	Top-down Parsing
↳ Unambiguous ↳ Left tailoring	↳ Unambiguous ↳ Left recursion elimination ↳ Left Tailoring
Shift-Reduce parsing ↳ LR Grammar	Recursive-Descent parsing ↳ LL(1) Grammar

Example: $E \to E+T \mid T$ input id * id
$T \to T*F \mid F$
$F \to (E) \mid id$

Soln: Construction of Bottom-up tree. (starting from the leaf node to root)

id * id ⇒ F * id ⇒ T * id ⇒ T * F



⇒ RMD for id * id

$E \xrightarrow{rmd} T$
$\Rightarrow T*F$
$\Rightarrow T*id$
$\Rightarrow F*id$
$\Rightarrow id*id$

Bottom-up parsing

** Bottom-up approach is constructing / writing RMD in reverse i.e

id * id
F * id
T * id
T * F
T
E

Bottom-up parsing

→ In case of bottom-up approach the body is replaced with head.

* **Handle Pruning** (Imp)

→ Handle is the substring / Body replaced by head.

→ Consider the following example:
$id_1 * id_2$, $F * id_2$, $T * id_2$, $T * F$, T, E

Right Sentential form	Handle	Reducing production
$id_1 * id_2$	id_1	$F \to id$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$
E		

⇒ Handle Pruning is construction of rightmost derivation in reverse. **

* Shift-reduce Parsing

⇒ Generalized Bottom-up Parsing
⇒ Shifting is pushing the symbol onto the Stack.
⇒ Reducing " popping " " from " "

	STACK	INPUT
Initial config-uration	$	w$
Final config	$S	$

Example: w = id * id

STACK	INPUT	ACTION
$	$id_1 * id_2$ $	Shift
$id_1	* id_2 $	reduce by $F \to id$
$F	* id_2 $	reduce by $T \to F$
$T	* id_2 $	Shift
$T*	id_2 $	Shift
$T*id_2	$	reduce by, $F \to id$
$T*F	$	reduce by, $T \to T*F$
$T	$	reduce by, $E \to T$
Final configuration $E	$	Accept

* Shift Reduce conflicts (conflicts during shift-reduce parsing)

Example: stmt → if expr then stmt
| if expr then stmt else stmt
| other.

STACK	INPUT	ACTION
$ if expr then stmt	else $	

→ The conflict here is that whether to reduce the STACK top to head or to shift INPUT "else" to stack.

⇒ LR parsing (to handle shift reduce parsing) uses set of states for shift-reduce conflicts.
⇒ These states consists set of items (productions).

Eg:- A → XYZ } items (not ordinary production it consists of dots)

A → ·XYZ A → XY·Z (processed string derivable by XY.)
A → X·YZ A → XYZ·

dots → tells how many inputs are processed.

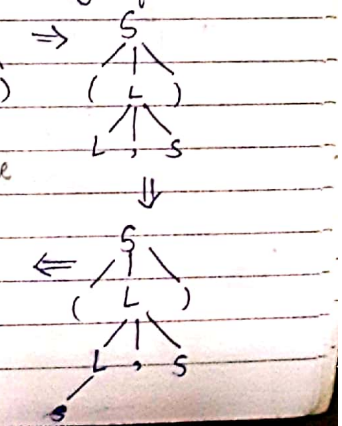
Example: Write LMD & RMD for the string (a,(a,a)) using the grammar:-

S → (L) | a
L → L,S | S

Soln

$S \overset{LMD}{\Rightarrow} (L)$
$\overset{LMD}{\Rightarrow} (L,S)$
$\overset{LMD}{\Rightarrow} (S,S)$
$\overset{LMD}{\Rightarrow} (a,S)$
$\overset{LMD}{\Rightarrow} (a,(L))$
$\overset{LMD}{\Rightarrow} (a,(L,S))$
$\overset{LMD}{\Rightarrow} (a,(S,S))$
$\overset{LMD}{\Rightarrow} (a,(a,S))$
$\overset{LMD}{\Rightarrow} (a,(a,a))$

Top-down parsing for LMD.



$S \Rightarrow^{RMD} (L)$
$\Rightarrow (L,S)$
$\Rightarrow (L,(L))$
$\Rightarrow (L,(L,a))$
$\Rightarrow (L,(S,a))$
$\Rightarrow (L,(a,a))$
$\Rightarrow (S,(a,a))$
$\Rightarrow (a,(a,a))$

Bottom-up Parsing

$(a,(a,a)) \Rightarrow (S,(a,a))$

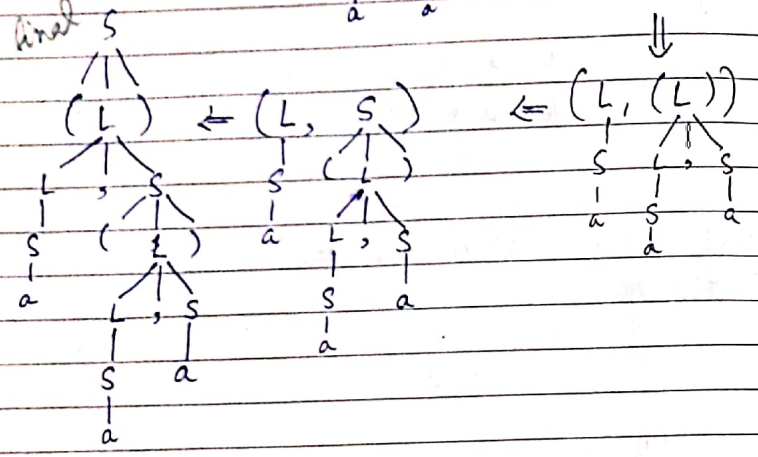
$(L,(S,a)) \Leftarrow (L,(a,a))$

$(L,(L,a)) \Rightarrow (L,(L,S))$

$(L,(L)) \Leftarrow (L,S)$

$(L,L) \Leftarrow (L,(L))$

final S



Example: Use Shift reduce parsing to process the input:
$(a,(a,a))\$$

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

Soln: Step 1: Right Most derivation for easier purpose.

$S \Rightarrow^{RMD} (L)$
$\to (L,S)$
$\to (L,(L))$
$\to (L,(L,S))$
$\Rightarrow (L,(L,a))$
$\Rightarrow (L,(S,a))$
$\Rightarrow (L,(a,a))$
$\Rightarrow (S,(a,a))$
$\Rightarrow (a,(a,a))$

STACK	INPUT	ACTION
$	(a,(a,a))$	Shift
$(	a,(a,a))$	Shift
$(a	,(a,a))$	reduce by $S \to a$
$(S	,(a,a))$	reduce by $L \to S$
$(L	,(a,a))$	Shift
$(L,	(a,a))$	Shift
$(L,(	a,a))$	Shift
$(L,(a	,a))$	reduce $S \to a$
$(L,(S	,a))$	reduce $L \to S$
$(L,(L	,a))$	Shift
$(L,(L,	a))$	Shift
$(L,(L,a	))$	reduce by $S \to a$
$(L,(L,S	))$	reduce by $L \to L,S$
$(L,(L	))$	Shift
$(L,(L)	)$	reduce by $S \to (L)$
$(L,S	)$	Shift
$(L,S)	$	reduce by $S \to (L,S)$
$(L)	$	reduce by $S \to (L)$
$S	$	

* Items & LR(0) automaton (To solve shift reduce parsing problem)

LR (k) → How many symbols of lookahead we need to make decision
↑ writing the RMD in reverse
↑ Scanning the i/p from left to right

→ LR comes in two flavour i.e LR(0) & LR(1) ↑ input symbols of lookahead
→ When we say LR it is by default LR(0).

$A \to \cdot XYZ$
$A \to \cdot XYZ$ (no input is processed)

Input is processed which is derivable by X

$A \rightarrow XYZ \cdot$ (Input is processed which is derivable by Z)

(Input is processed which is derivable by Y)

(When the dot is reached, the XYZ is reduced to A).

→ We can resolve shift reduce conflict using dot. When the dot is reached at the end, then only the expression is reduced.

→ Collection of LR(0) items :- We will have set of states. (Each state contains set of items.)

→ We'll construct automata i.e. either DFA or NFA

→ But for LR(0) we will construct DFA.

→ Each item is a production in the form of Head → Body (In the body there will be a dot always present, if there is a dot it is called the item).

→ One set of collection of LR(0) items is called as canonical LR items. For canonical LR items we construct an automaton called DFA

* Augmented Grammar : Suppose G is a Grammar then G' is Augmented Grammar.

If start symbol of G is S

Then start symbol of G' is given by $S' \rightarrow S$

(The extra production $S' \rightarrow S$ is to know when to stop parsing as it is when the i/p is accepted)

Example:

G	G'
$E \rightarrow E+T \mid T$	$E' \rightarrow E$
$T \rightarrow T*F \mid F$	$E \rightarrow E+T \mid T$
$F \rightarrow (E) \mid id$	$T \rightarrow T*F \mid F$
	$F \rightarrow (E) \mid id$

* Closure of Item Sets :- If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules.

1. Initially, add every item in I to Closure(I)
2. If $A \rightarrow \alpha \cdot B\beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot\gamma$ to CLOSURE(I) if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I)

Ex: i) $E' \rightarrow E$
$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

$I : \{ E' \rightarrow .E \}$

Compute CLOSURE(I).

Soln: CLOSURE(I) :
$E' \rightarrow \cdot E$ (add E productions)
$E \rightarrow \cdot E+T$ (Add E & T productions)
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$ (Add T & F productions)
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$ (Terminals so end of discuss)
$F \rightarrow \cdot id$

(Closure of I items and we are done)

ii) Compute closure of $E \rightarrow \cdot T$

Closure $(E \rightarrow \cdot T)$ =
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

Ex: $S \rightarrow (L) \mid a$
$L \rightarrow L, S \mid a$

compute closure of $S \rightarrow (\cdot L)$

Automata for LR Grammar is constructed using
- Augmented Grammar
- Closure
- GOTO.

Soln: CLOSURE $(S \to (\cdot L))$: $S \to (\cdot L)$
$L \to \cdot L,S$
$L \to \cdot a$

ii) Compute CLOSURE $(S \to \cdot a)$: $S \to \cdot a$

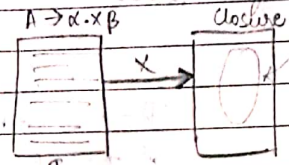
Note: Kernel Items: The initial item, $S' \to \cdot S$ and all items whose dots are not at the left end.

Non-kernel Items: all items with their dots at the left end, except for $S' \to S$.

1:03 08

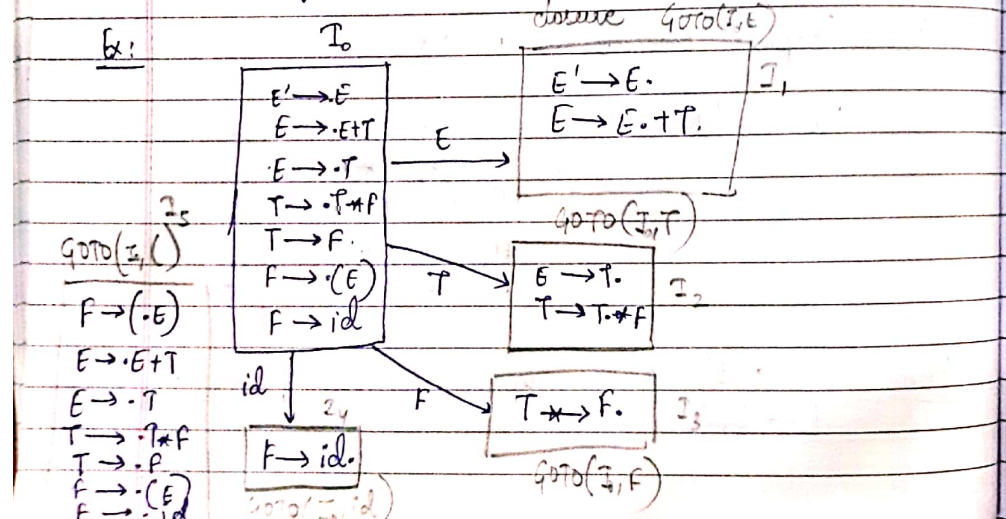
* The function GOTO (To create transitions of the state diagram).

GOTO(I, X).



from state I, if we follow X Grammar symbol, then which state to go, will be given by GOTO function.

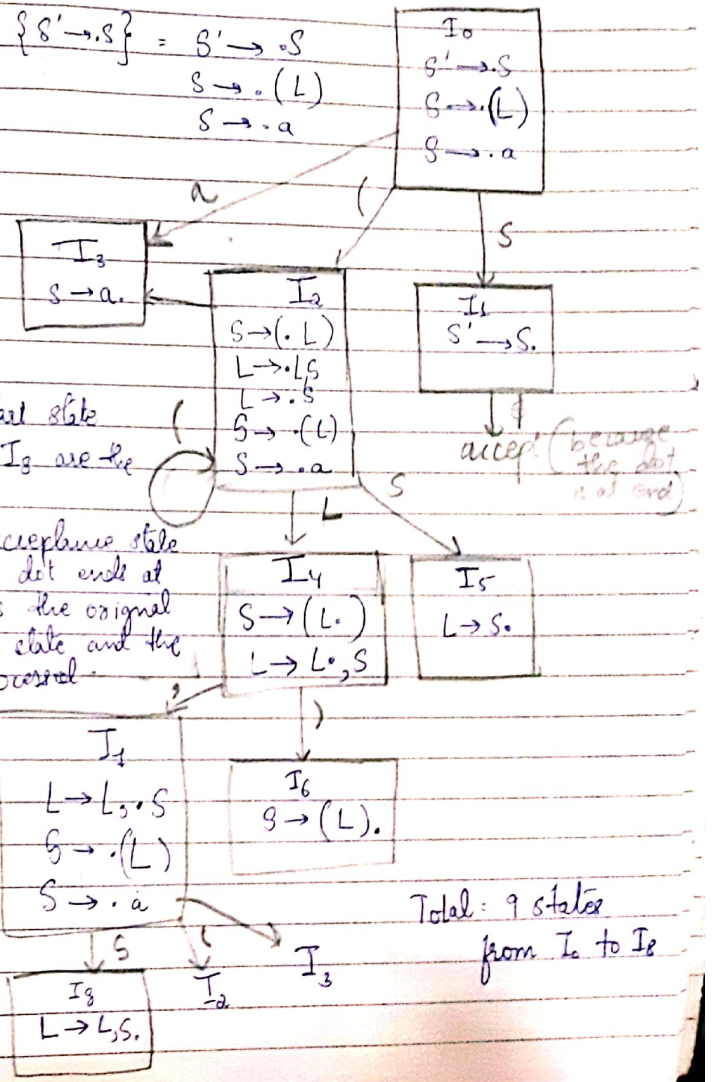
Ex:



* Construct canonical LR(0) automata (or) Construct canonical LR(0) sets of items.

1) $S' \to S$
$S \to (L) \mid a$
$L \to L,S \mid S$.

Soln: closure $\{S' \to .S\} = S' \to .S$
$S \to .(L)$
$S \to .a$



→ I_0 is the start state
→ I_1, I_5, I_7 I_8 are the final state
⇒ I_5 is the acceptance state because the dot ends at 'S' which is the original grammar start state and the input is processed.

Scanned with CamScanner

* Shift-reduce parser on the input id * id using LR(0) automata. (Ex: SS ask someone)
↳ The state diagram to be referred

Line	Stack (Initial state → 0; State number)	Symbol	Input	Action
1	0	$	id * id $	Shift to 5
2	0 5	$ id	* id $	reduce by F → id
3	0 3	$ F	* id $	reduce by T → F
4	0 2	$ T	* id $	Shift to 7.
5	0 2 7	$ T *	id $	Shift to 5
6	0 2 7 5	$ T * id	$	reduce by F → id
7	0 2 7 10	$ T * F	$	reduce by T → T * F
8	0 2	$ T	$	reduce by E → T
9.	0 1	$ E	$	Accept

↑ from I1 we read $ so accept.

* Structure of LR Parsing Table

→ ACTION table
- ACTION takes in two parameters ACTION[i, a]
- i refers to the row & 'a' to the column
- i is the state, a is the terminal symbol or $.

- 4 Posibilities of ACTION[i, a]
  i) ACTION[i, a] = Shift j
  ii) ACTION[i, a] = Reduce A → β
  iii) ACTION[i, a] = Accept (When $ is read)
  iv) Error (call error recovery routine).

* LR-parser configuration

($S_0 S_1 S_2 S_3 \dots S_m$, $a_i a_{i+1} \dots a_n \$$)
- Stack contents (States pushed onto the stack)
- Remaining inputs

($X_1 X_2 \dots X_m$, $a_i a_{i+1} \dots a_n \$$)
- Grammar symbols

* Behaviour of LR parser

i) Shift → Eg: ACTION[s_m, a_i] = Shift j
ii) Reduce - Here S_m is the state, shift j indicates
iii) Accept that j will be pushed onto the stack.
iv) Error - So the configuration for this is:
($S_0 S_1 \dots S_m j$, $a_{i+1} \dots a_n \$$) (a_i is popped from input which corresponds to j.)
(j is pushed)

* LR-Parsing program

→ It is used for LR(0), SLR, LALR.

INPUT: An input string w and an LR-parsing table with function ACTION and GOTO for a Grammar G.

OUTPUT: If w is in L(G), the reduction steps of a bottom-up parse for w, otherwise an error indication

METHOD: Initially, the parser has S_0 on its stack, where S_0 is the initial state and w$ in the input buffer. The parser then executes the program

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
    let s be the state on top of the stack;
    if ( ACTION[s, a] == shift t ) {
        push t onto the stack;
        let a be the next input symbol;
    }
    else if ( ACTION[s, a] == reduce A → β ) {
        pop |β| symbols of the stack;
        let state t now be on top of the stack;
        push GOTO(t, A) onto the stack;
        output the production A → β;
    } else if ( ACTION(s, a) == accept ) break;
    else call error-recovery routine;
}
```

Ex 1: Parsing table for the following expression Grammar

(1) $E \to E+T$ (4) $T \to F$

(2) $E \to T$ (5) $F \to (E)$

(3) $T \to T*F$ (6) $F \to id$

⇒ The codes for the actions are:

1. s_i means shift and stack state i
2. r_j means reduce by the production numbered j
3. acc means accept
4. blank means error

Suppose we are in state i and there is a transition for that i/p symbol then we do shift else reduce

Simple LR & lookAhead LR → LR(0)
Canonical LR → LR(1)

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s_5			s_4			1	2	3
1		s_6				acc			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Fig: Parsing table for expression Grammar (This for LR parsing)

* Constructing SLR (Simple LR) Parsing table:

→ All these LR algorithms, the Driver program remains same whereas the parsing tab algorithm changes

INPUT: An augmented Grammar G'
OUTPUT: The SLR-parsing table function ACTION and GOTO for G'
METHOD: See Txt book

→ When · dot is at the end, you should take follow of A in the production A → a· and Action is to reduce A → a

→ When $S' \to S.$ is in I_i then set ACTION[i, $] to accept.

→ When $A \to \alpha . a\beta$ then ACTION[i, a] to Shift j. a should be terminal.

Example: Construct SLR(1) parsing table for the following Grammar:

$S \to BB$

$B \to aB \mid b$

Soln: LR(0) items set

Step 1: Augmented Grammar

$S' \to S$ ← Dont consider while number

$S \to BB$ (1)

$B \to aB$ (2)

$B \to b$ (3)

closure($\{S' \to .S\}$)

Step 2: Construct LR(0) items.

I_0:
$S' \to .S$
$S \to .BB$
$B \to .aB$
$B \to .b$

I_1: $S' \to S.$ — $ → accept

I_2: $S \to B.B$, $B \to .aB$, $B \to .b$

I_3: $B \to a.B$, $B \to .aB$, $B \to .b$

I_4: $B \to b.$

I_5: $S \to BB.$

I_6: $B \to aB.$

we take these transitions because in I_0 we have S, B, a & b after the dot.

* when dots is at the end then there's no transition from that item.

Step 3: Construct SLR parsing table

(from I_0 we have transition on a to I_3, so shift 3)

STATE	ACTION			GOTO	
	a	b	$	S	B
0	S_3	S_4		1	2
1			accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

(from I_0 we have a transition on 'S' to I_1, so 1 indicates the state I_1)

(For I_4 the dot is at end so we do reduction, and the production 3 is used, here we have to take the follow of B and apply r_3 to all the follow elements)

(Apply follow to 6 as terminal)

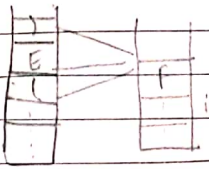
* Steps to construct SLR(1) parsing table
1. Augmented Grammar
2. Find FOLLOW of each non-terminal.
3. Construct LR(0) automata
4. Construct the table with STATE, ACTION, GOTO

* Viable Prefix :- It is a prefix on the top of the stack to be reduced.

Eg: $E \to E+T \mid T$, $T \to T*F \mid F$, $F \to (E) \mid id$.

Assume a RMD as follows:

$E \Rightarrow \ldots \Rightarrow F * id \Rightarrow (E) * id \Rightarrow \ldots$



Here (E) is reduced to F, hence (, E, (is the viable prefix

FOLLOW(S) = {$}

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

→ LR(0) items completed
→ Now ↓ we have LR(1) items

⇒ **Defn:** Viable prefix is the one in which the prefix of the right most sentential forms which are present on top of the stack. (RMD)

→ Dots '.' and viable prefix helps in deciding to whether shift or reduce

* **More powerful LR Parser**

i) Canonical LR (CLR) → or LR - set of items LR(1) items
ii) LALR (Lookahead LR) → Set of items of LR(0) - less no of states

* **Canonical LR(1) items** (1 is for the length of the 2nd argument)

⇒ General format of each item in case of LR(1) is:

$$[A \to \alpha \cdot B\beta, a]$$

where, $A \to \alpha B\beta$ is a production
a is a terminal or $.

⇒ To compute closure of the LR(1) items.

Ex i) Grammar, G: $S \to CC$
$C \to cC \mid d$

Soln: Augmented Grammar, G': $S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

first item for any LR(1) grammar

CLOSURE { $S' \to .S, \$$ }

I_0	
Initially	$S' \to .S, \$$
a)	$S \to .CC, \$$
b)	$C \to .cC, c/d$
b)	$C \to .d, c/d$

a) Compare $S' \to .S, \$$ with $A \to \alpha .B\beta, a$

Here, $A = S'$; $\alpha = \varepsilon$; $B = S$, $\beta = \varepsilon$, $a = \$$
According to the algorithm, Compute:
i) FIRST(βa) = FIRST($\varepsilon \$$) = {$}, b is the elements
ii) Add $B \to .\gamma, b$ to set I, calculate for
⇒ On comparison, we add: FIRST(βa)
$S \to .CC, \$$ to I_0

b) Compare $S \to .CC, \$$ with $A \to \alpha .B\beta, a$

Here, $A = S$; $\alpha = \varepsilon$; $B = C$; $\beta = C$; $a = \$$
According to the algorithm, compute
i) FIRST(βa) = FIRST($C \$$) = {c, d}
ii) Add $B \to .\gamma, b$ to set I,
⇒ On comparison, we add:
$C \to .cC, c$
$C \to .cC, d$
$C \to d, c$
$C \to d, d$

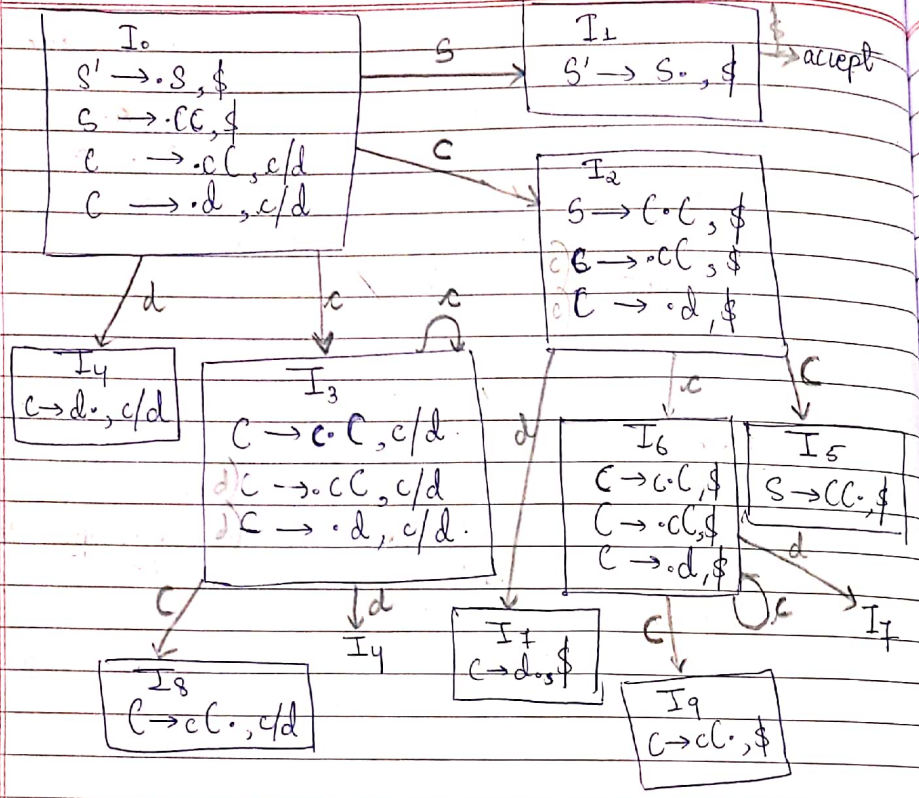
⇒ To compute GOTO of the LR(1) items: Same as LR(0) items

* Example to construct LR(1) item sets / automata for the following Grammar / Canonical LR(1) / LR(1) GOTO graph.

1] Augmented, G': $S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

c) Compare $S \to C \cdot C, \$$ with $A \to \alpha \cdot B\beta, a$
→ $A = S$, $\alpha = C$, $B = C$, $\beta = \epsilon$, $a = \$$
→ $FIRST(\epsilon\$) = \$$
→ Add $C \to \cdot cC, \$$ and $C \to \cdot d, \$$

Soln:



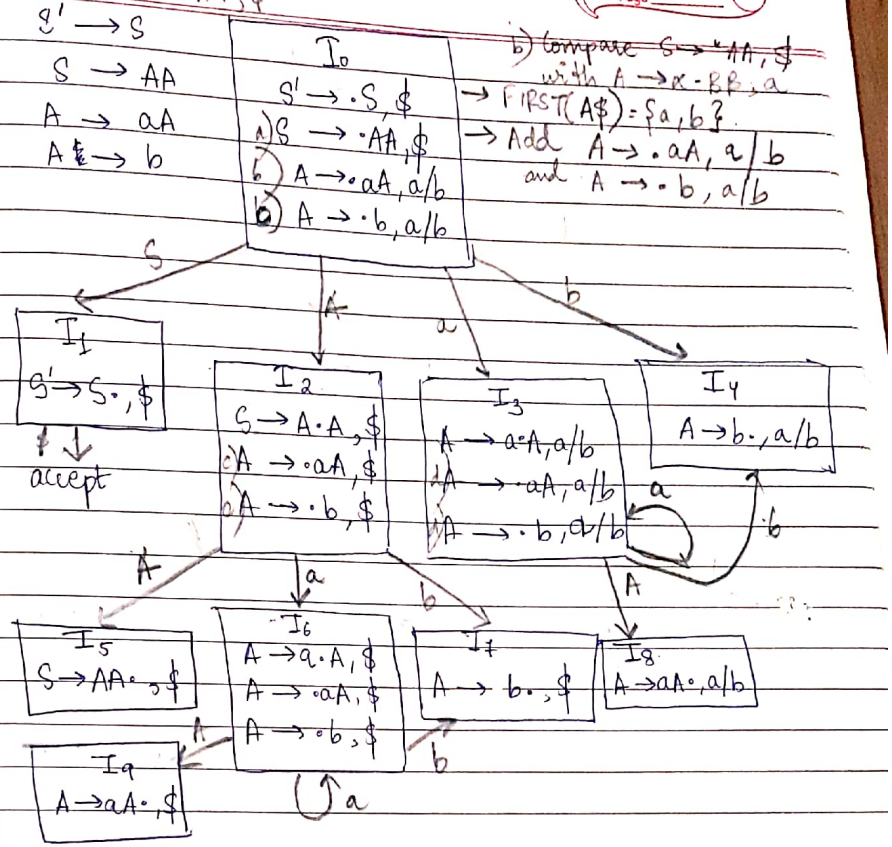
$I_0 \dots I_8, I_9$ are LR(1) items.

d) Compare $C \to c \cdot C, c$ with
$A \to \alpha \cdot B\beta, a$
→ $A = C$; $\alpha = c$; $B = C$; $\beta = \epsilon$; $a = c$
$FIRST(\epsilon c) = c$
→ Add $C \to \cdot cC, c$ and $C \to \cdot d, c$
⇒ Similarly with $C \to c \cdot C, d$

a) Compare $S' \to \cdot S, \$$ with $A \to \alpha \cdot B\beta, a$
→ $A = S'$; $\alpha = \epsilon$; $B = S$; $\beta = \epsilon$; $a = \$$
→ $FIRST(\epsilon\$) = \$$
→ Add $S \to \cdot AA, \$$

2

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



b) ~~Compare $S \to \cdot AA, \$$ with $A \to \alpha \cdot B\beta, a$~~
→ $FIRST(A\$) = \{a, b\}$
→ Add $A \to \cdot aA, a/b$ and $A \to \cdot b, a/b$

c) Compare $S \to A \cdot A, \$$ with $A \to \alpha \cdot B\beta, a$
→ $FIRST(\epsilon\$) = \$$
→ Add $A \to \cdot aA, \$$ & $A \to \cdot b, \$$

d) Compare $A \to a \cdot A, a/b$ with $A \to \alpha \cdot B\beta, a$
→ $FIRST(\epsilon a) = \{a\}$ $FIRST(\epsilon b) = \{b\}$
→ Add $A \to \cdot aA, a/b$ and $A \to \cdot b, a/b$

* Canonical LR(1) Parsing Tables

Ex 1] $S \to CC$ (1)
$C \to cC$ (2)
$C \to d$ (3)

Soln: Step 1: Construct LR(1) items - already done. check previous example.

Step 2: CLR parsing table

STATE	ACTION			GOTO	
	c	d	$	S	C
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

* Look Ahead LR (Few states & much easier than SLR & CLR)

→ Required both LR(0) & LR(1) item sets.
→ In LALR we dub states/items when the core part i.e LR(0) part of LR(1) is same.
For eg:- $C \to d\cdot, c/d$
$C \to d\cdot, \$$
Here $C \to d\cdot$ is same whereas the part after "," differ, hence we dub these two items.

→ LALR Parsing table based on above example

STATE	ACTION			GOTO	
	c	d	$	S	C
0	S36	S47		1	2
1			acc		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Ex1: Construct LALR(1) parsing table for the following grammar.

G: $S \to L=R$ (1)
$S \to R$ (2)
$L \to *R$ (3)
$L \to id$ (4)
$R \to L$ (5)

Augmented G' is
$S' \to S$
$S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

a) Compare $S \to \cdot L=R, \$$ with $A \to \alpha \cdot B\beta, a$
FIRST(=$) = {=}
Add $L \to \cdot *R, =$ & $L \to \cdot id, =$

b) Compare $S \to \cdot R, \$$ with $A \to \alpha \cdot B\beta, a$
FIRST($\$) = \$
Add $R \to \cdot L, \$$

c) Compare $R \to \cdot L$ with $A \to \alpha \cdot B\beta a$
FIRST($\$) = \$, Add $L \to \cdot *R, \$$ and $L \to \cdot id, \$$

Soln Steps: Construct LR(1) items collection.