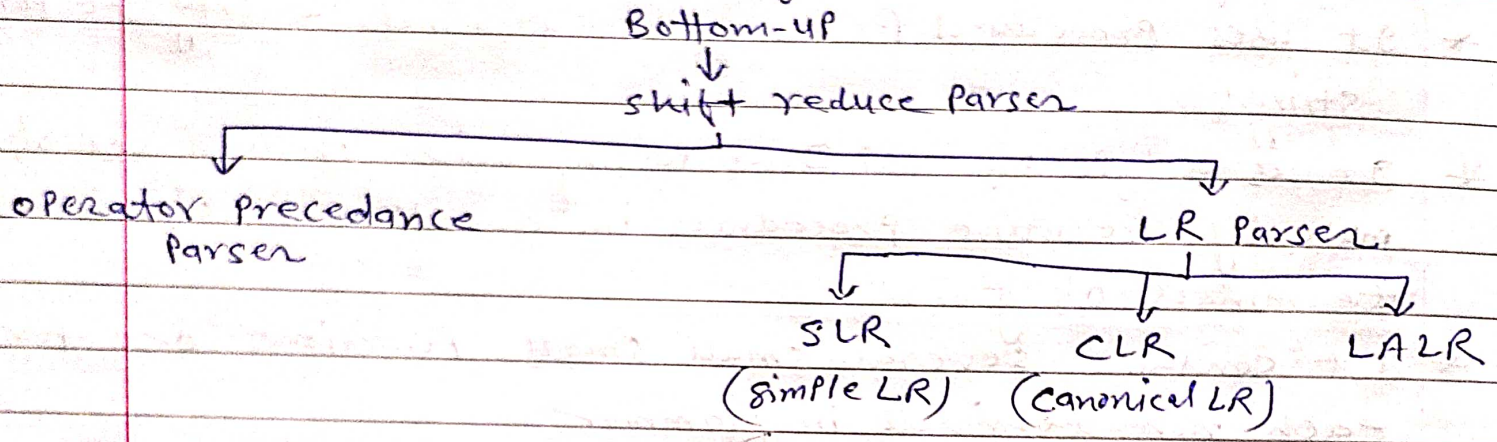
Bottom-up Parser

→ Bottom-up Parser (Parsing) can be defined as attempt to reduce the input string (w) to the starting symbol of the grammar by tracking of the right most derivation of string (w) of reverse

→ Bottom-up design from leaf to root.

→ Bottom-up parser use right most derivation (RMD)



* Operator precedance ⇒ It is the kind of bottom-up Parser we can design operator precedance Parser ~~when~~ when the grammar is operator precedance grammar.

Condition for operator precedance grammar ⇒ A grammar is said to be operator precedance when

(i) These should not be ∈ production in RHS of any production

(ii) These should not be two or more consecutive (consecutive) variables

Example:- S → AB/a X S → AbB/Ab/a/ab
B → bB/b

~~constitutive~~ Variable
Consecutive (लगातार)

* Sign conversion ~~for operator~~ ⇒ [dot use for it is the way of representation

<· , ·> , ≐

<· ⇒ Representation of less than sign

·> ⇒ Representation of greater than sign

≐ ⇒ Representation of show equality.

* Priority of operator precedance ⇒

(), ↑ (power), *, /, +, −

Date:

* **Advantage of Bottom-up Parsing ⇒**

→ It simple in design

→ Ambiguous grammer :- Operator precedance Parsing can handle ambiguous grammer also even though we can eliminate ambiguity from ambiguous grammar of programming language but it is not advisable hence we need a parsing technique that can handle ambiguous grammar

→ It also known as shift reduce Parsing.

Construction of operator precedance Parsing Tree ⇒

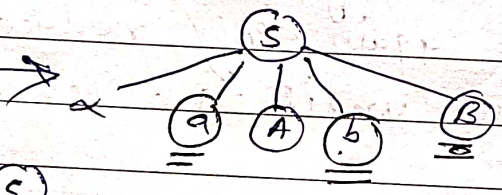
Ambiguous CFG ⇒ General programming lang. precedance

$* > +$, $* > -$, $+ = -$

Un-ambiguous CFG ⇒

(i) $a \doteq b$

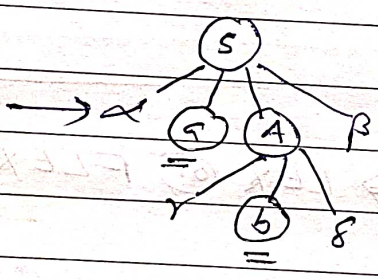
$S \to \alpha a A b \beta$



(ii) $a \lessdot b$

$S \to \alpha a A \beta$

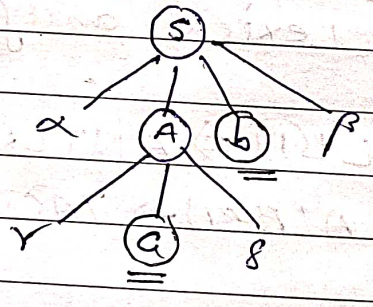
$A \to \gamma b \delta$



(iii) $a \gtrdot b$

$S \to A b \beta$

$A \to \gamma a \delta$



Note :-

* Identifier is higher precedance in all other operator
* $ is lesser precedance than then all other operator
* If grammar contains both left recursion & right recursion then it is ambiguous grammar
* If a grammar contain either left recursion or right recursion & also follow chain rule then it is unambiguous grammar.

[Associative / Association] :- ↑ - right associative because in grammar ↑ is right recursion ($A \to B \uparrow A / B$)

@ left associative $[B \to B @ C]$. $ → right associative $[C \to D \$ C]$

— right associative $[D \to C - D]$

* Construct operator precedance parsing table ⇒

Calculate Leading & Trailing ⇒

Leading :- Rules:-

(i) If the production of the type $A \to \alpha b \beta$. where α is the single variable or ϵ then b will be including in Leading of A.

(ii) If $A \to B\alpha$ is the production then all the leading symbol of B will be including Leading of A.

Trailing ⇒ Rules:-

(i) If $A \to \alpha b \beta$ is the production then β is the single variable or ϵ then b will be Trailing of A.

(ii) If $A \to \alpha B$ then all the trailing symbol of B will be including Trailing of A.

Q.1] Calculate the Leading & trailing of given grammar

$A \rightarrow BaGd/be$, $B \rightarrow aG/d$, $G \rightarrow c$

Sol^n Leading of A :-

$L(A) = L(B) \cup L(aGd) \cup L(be)$ ——— (1)

$L(B) = L(aG) \cup L(d) = \{a\} \cup \{d\} = \{a, d\}$

$L(G) = L(c) = \{c\}$

from equation (1)

$L(A) = \{a, d\} \cup \{a\} \cup \{b\} = \{a, b, d\}$

Trailing of A :-

$T(A) = T(BaGd) \cup T(be) = \{d\} \cup \{e\} = \{d, e\}$

$T(B) = T(aG) \cup T(d)$ ——— (2)

$T(G) = \{c\}$

from equation (2)

$T(A) = \{c\} \cup \{a\} \cup \{d\} = \{a, c, d\}$

Q.2] Calculate the Leading & Trailing of given grammar

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

Variable	Leading	Trailing
E	{+, *, (, id}	{+, *,), id}
T	{*, (, id}	{*,), id}
F	{(, id}	{), id}

Q. Calculate the ~~first & follow~~ Leading & Trailing.

$E \rightarrow E+E/E*E/id$

V	L	T
E	{+, *, id}	{+, *, id}

Q.1) $E \to E+T/T$, $T \to T*F/F$, $F \to (E)/id$ construct operator precedance relation table.

Soln

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

	+	*	(	)	id	$
+	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$\cdot>$	$\cdot>$	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
(	$<\cdot$	$<\cdot$	$<\cdot$	$=$	$<\cdot$	
)	$\cdot>$	$\cdot>$		$\cdot>$		$\cdot>$
id	$\cdot>$	$\cdot>$		$\cdot>$		$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$		$<\cdot$	

Disadvantages ⇒ of relation table ⇒

* Huge (विशाल) space require
* space means space complexity $O(n^2)$
* function table ⇒ we reduce the complexity of $O(n^2)$ so we used the function table. we construct passing table in terms of function called function table. It's complexity is $O(n)$

we use the function f & g

F for point to stack TOP value.

g for point to input tape symbol.

* function graph ⇒

First we construct function graph then function table.

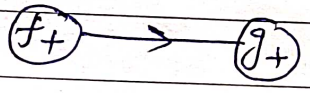
Q.2] Construct function graph & function table for the given operator precedance relation table

f \ g	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

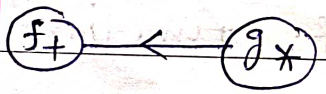
Operator Precedance graph (function graph) ⇒ Operator Precedance graph is a kind of directed graph. The every terminal of this graph having the two corresponding vertex f & g.

Example:

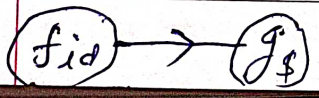
+ Vs + ⇒ ·>

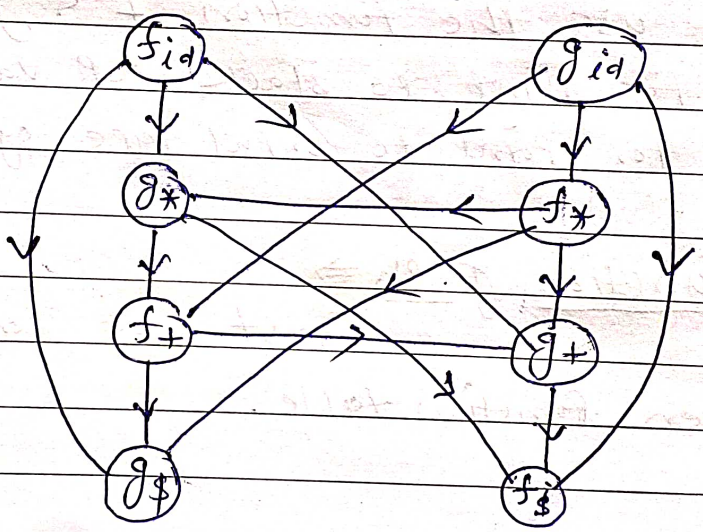


+ Vs * ⇒ <·



id Vs $ ⇒ ·>





Date: YOUVA

function table :-

Maximum traversing length in table

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

Here 0,0

because $f_\$$ & $g_\$$ ~~से को~~ No out going edges from this node so length will be zero

---

* Cycle in graph ⇒

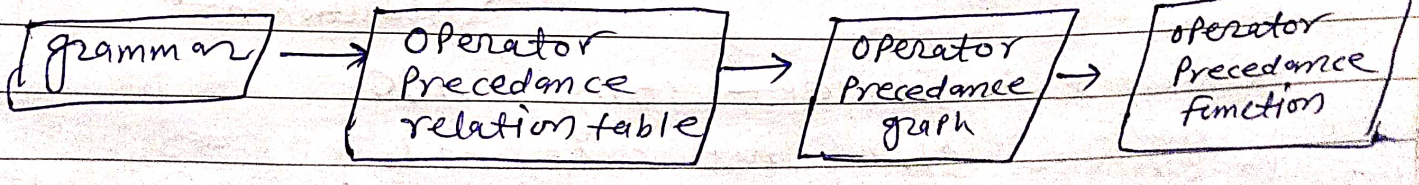
Traveling from source vertex & again reach on traveling (start node / from start node) on destination node means source & destination vertex are same & traveling continues & ∞ times is called cycle in graph.

If the cycles are present in the operator precedance graph then these relation table is not possible to construct function table.

* Dis advantages of function table ⇒

* Parsing operation is slow (means number of memory ~~clete~~ cycle is more)
* Not direct entry in table means it's depends on the some rules.

How to design function table ⇒



LR Parser

SLR for LR(0) collection

CLR for LR(1) collection

~~LARL~~ LALR

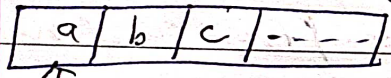
LR(K)

First L :- Left to Right Scanning of I/P string

second R :- Reverse right most derivation

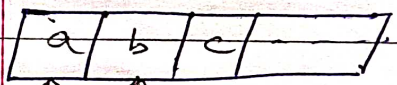
Third K :- Look ahead symbol

LR(1) means (Single head) look a head is one



only read one symbol.

LR(2) means (Single head) look a head is two



only two reading head symbol

SLR (Simple LR Parser) / LR(0) Parser) ⇒

→ Reduce operation all terminal में only follow में SLR

Operation types (Performed) —
- → Shift (S)
- → Reduce (R)
- → Accept (Acc)
- → Error (Er)

Date: YOUVA

* **Handler** (sequence of grammar symbols) ⇒ It is present in the form of RHS of any position. It is always appears in the stack top during parsing (we can not include state no. forget it see only grammar symbol ↑)

$S \to \alpha A \beta$ & $A \to \gamma$ so

$S \to \alpha\gamma\beta$ then $A \to \gamma$ is a handler of $\alpha\gamma\beta$ in the position following α or position preceding β.

* **Handle Pruning** ⇒ when ever a handle is detected perform the reduction. This is equivalent to performing right most derivation called as handle pruning.

$w = id + id * id$

$w = id + id * E$ ∵ $E \to id$

$w = id + E * E$ ∵ $E \to id$

$w = E + E * E$ ∵ $E \to id$

$w = E + E$ ∵ $E \to E * E$

$w = E$ or $E = w$ ∵ $E \to E + E$

Q.3) Construct of SLR(1) parser / LR(0) item collection when grammar is given

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

Sol^{n}

Step-I :- Calculate first & follow from given grammar

Variable	first	follow
E	{ (, id }	{ +,), $ }
T	{ (, id }	{ +, *,), $ }
F	{ (, id }	{ +, *,), $ }

Step-2 :- Do numbering of each production as

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

Step-3 :- make a augumented grammar means (Augument) add new starting variable

$E' \to E$

Date: YOUVA

Step-4 :- Calculate (Find out) canonical item collection for LR(0) item set

item :- . (dot will be include/including in RHS of the production)

Complete item :- If dot (.) goes to end of the production is called complete item

A → aBc (production)

↓ item

A → . aBc or A → a (or) A → ∈

A → a.Bc .A → . a A → . ∈

A → aB.c A → a. A → .

A → aBc. // called complete item

Note :- No. of items formed = n+1

No. of complete item is only one

No. of incomplete item is n

Canonical item collection for LR(0) set ⇒ (we get)

Closure Property :- If . Variable [.V] then convert all the items into production with dot (.) operator on RHS of all productions

Goto Property :- shift dot (.) operator one position right hand side & create new state & again see closure property.

GIRIRAJ - R.G.G.

Page No.:
Date:
YOUVA

Kernel item ⇒ An LR(0) item set is called kernel item if dot (·) is not at the left end (S'→·S) is called kernel item.
when dot at the left end is called non kernel item

I_0 :- First we have to convert augmented grammar

$E' \rightarrow \cdot E$

I_0 :- $E' \rightarrow \cdot E$

Apply closure properties $E \rightarrow \cdot E+T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T*F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

I_1 :- Apply goto property means dot move one position Right hand side of each production

$E' \rightarrow E \cdot$

$E \rightarrow E \cdot +T$

I_2 :- $E \rightarrow T \cdot$

$T \rightarrow T \cdot *F$

I_3 :- $T \rightarrow F \cdot$

I_4 :- $F \rightarrow (\cdot E)$

$E \rightarrow \cdot E+T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T*F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

I_5 :- $F \rightarrow id \cdot$

I_6 :- $E \rightarrow E+\cdot T$

$T \rightarrow \cdot T*f$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id.$

I_7 :- $T \rightarrow T*\cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

I_8 :- $F \rightarrow (E \cdot)$

$E \rightarrow E \cdot +T$

I_9 :- $E \rightarrow E+T \cdot$

$T \rightarrow T \cdot *F$

I_{10} :- $T \rightarrow T*F \cdot$

I_{11} :- $F \rightarrow (E) \cdot$

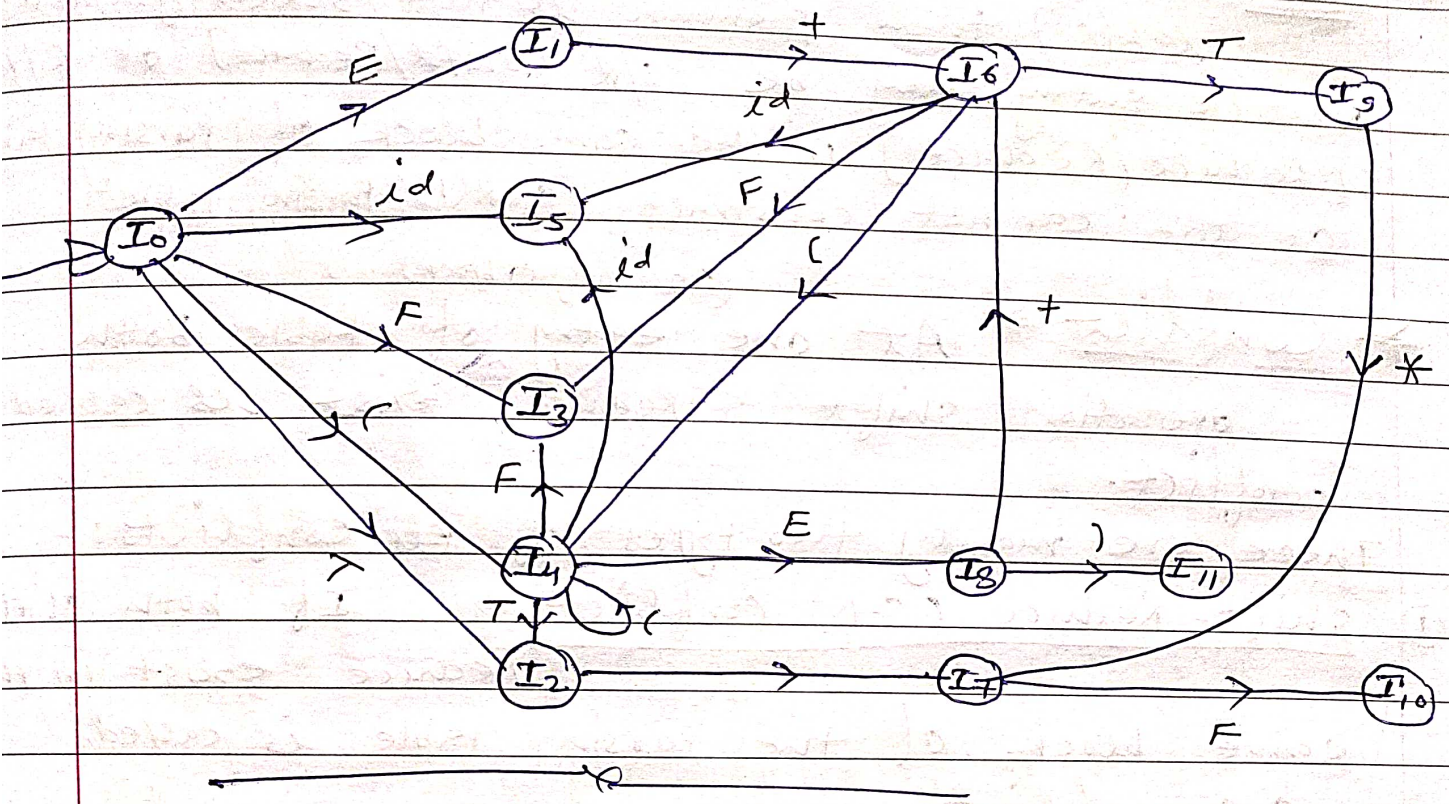
Step-5 :- construct parsing table

Date: youvA

Parsing table

S.No.	Action table						goto table		
item collection	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Accept			
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			

Goto graph ⇒



.2] Design SLR(1) Parser for the following grammar

$S \to 0S_0$, $S \to 1S_1$, $S \to 10$

step-I

Variable	First	follow
S	{0,1}	{0,1,$}

step-II :- make Augmented grammar $S' \to S$

step-III :- Do numbering

1. $S \to 0S_0$ 2. $S \to 1S_1$ 3. $S \to 10$

step-IV :- calculate canonical items collection of LR(0) itemset

I_0 :- $S' \to .S$

$S \to .0S_0$

$S \to .1S_1$

$S \to .10$

I_1 :- $S' \to S.$

I_2 :- $S \to 0.S_0$

$S \to .0S_0$

$S \to .1S_1$

$S \to .10$

I_3 :- $S \to 1.S_1$

$S \to .0S_0$

$S \to .1S_1$

$S \to .10$

I_4 :- $S \to 1.0$

I_5 :- $S \to 0S.0$

I_6 :- $S \to 1S.1$

I_7 :- $S \to 10.$

I_8 :- $S \to 0S0.$

I_9 :- $1S1.$

Parsing table sath goto graph.

P.T.O

Q.3]

$S \to L=R$

$S \to R$

$L \to *R$

$L \to id$

$R \to L$

Total item collection I_0 to I_9

Date:

CLR [Canonical LR Parser / LR(1)] ⇒ It is the same way as SLR Parser. SLR Parser working ways as:-

- Find out the first & follow [calculate both]
- Augument grammar
- Do numbering of production
- calculate canonical item collections / LR(1) collection item.
- construct CLR(1)/CLR Parsing table.

LR(k) item set ⇒ $A \to \alpha \cdot B\beta, K$ [string of length k or k look ahead symbol]

LR(2) item set ⇒ $A \to \alpha \cdot B\beta, ab$ [string of ~~length~~ two terminals look ahead 2]

LR(1) item set ⇒ $A \to \alpha \cdot B\beta, a$ [string of one terminal look ahead 1]

Rules:-

$A \to \alpha \cdot B\beta, a$
$A \to \alpha \cdot B\beta, b$
} Both are different item set

Use of look ahead symbol only when we get complete item [only on case of reduce operation entry]

Example:- $A \to \alpha \cdot B\beta, a$ where First(β) = {b}

$B \to \cdot\gamma, b$ First(β, a) = {b} then

Q.1) Design CLR(1) Parser for the grammar

$S \to GG$, $G \to cG$, $G \to d$

Soln

Step-I:- calculate First & follow of given grammar

Variable	First	follow
S	{c, d}	{$}
G	{c, d}	{c, d, $}

Step-II:- Do number of given productions

1. $S \to GG$
   $G \to cG$
   $G \to d$

~~**Step-III:-** make Augmented grammar~~ $S' \to S$

Step-IV :- Calculate Canonical item (LR(1) item collection)

I_0 :- $S' \rightarrow .S, \$$
$S \rightarrow .CC, \$$
$C \rightarrow .cC, c/d$
$C \rightarrow .d, c/d$

First $(.\$) = \{\$\}$ because here β is null (ϵ) so First of $(\$) = \{\$\}$

I_1 :- $S' \rightarrow S., \$$

I_2 :- $S \rightarrow C.C, \$$
$C \rightarrow .cC, \$$
$C \rightarrow .d, \$$

I_3 :- $C \rightarrow c.C, c/d$
$C \rightarrow .cC, c/d$
$C \rightarrow .d, c/d$

I_4 :- $C \rightarrow d., c/d$

I_5 :- $S \rightarrow CC., \$$

I_6 :- $C \rightarrow c.C, \$$
$C \rightarrow .cC, \$$
$C \rightarrow .d, \$$

I_7 :- $C \rightarrow d., \$$

I_8 :- $C \rightarrow cC., c/d$

I_9 :- $C \rightarrow cC., \$$

Parsing table :- CLR Parsing table

State no	Action table			goto table	
	c	d	$	S	C
0	S_3	S_4		1	2
1			Accept		
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		

* LALR Parser [Look-ahead LR Parser] ⇒ This parser is 100% same as canonical LR(1) Parser ~~that~~ but in case of LALR Parser we remove the similar state that one subset of other states in LR(1) collection of items then table is constructed.

~~Example: super set state~~ Firstly mostly language based on LALR Parser because of less size [CLR(1) ✗] & high accuracy (SLR ✓)

In terms of power

$$CLR(1) > LALR(1) > SLR(1)$$

(Q.1) Design LALR(1) Parser for the following grammar

$E \to E+T/T/TF$

$T \to F$ & $F \to F*/a/b$

Sol^n Step-I ⇒ calculate First & follow of the given grammar

Variable	First	follow
E	{a, b}	{+, $}
T	{a, b}	{+, a, b, $}
F	{a, b}	{+, *, a, b, $}

Step-II ⇒ Do the numbering of the given grammar production

1. $E \to E+T$ 2. $E \to T$ 3. $E \to TF$
4. $T \to F$ 5. $F \to F*$ 6. $F \to a$ 7. $F \to b$

Step-III ⇒ Make Augumented grammar

$E' \to E$

Step-IV ⇒ calculate LR(1) canonical item collection ~~set~~

I_0:- $E' \to \cdot E, \$$ — $ according to $E' \to \cdot E, \$ = \{\$\}$

$E \to \cdot E+T, \$/+$ ← + according to $E \to \cdot E+T, \$ = \{+\}$

$E \to \cdot T, \$/+$

$E \to \cdot TF, \$/+$ — a & b according to $E \to \cdot TF, \$/+ = \$F(F)$

$T \to \cdot F, \$/+/a/b/*$ → * according to $F \to \cdot F* = F(*) = \{*\}$

$T \to \cdot F*, \$/+/a/b/*$

$F \to \cdot a, \$/+/a/b/*$

$F \to \cdot b, \$/+/a/b/*$

I_1 :- $E' \to E \cdot, \$$
$E \to E \cdot + T, \$/+$

I_2 :- $E \to T \cdot, \$/+$
$E \to T \cdot F, \$/+$
$F \to \cdot F *, \$/+$
$F \to \cdot a, \$/+$
$F \to \cdot b, \$/+$

I_3 :- $T \to F \cdot, \$/+/a/b$
$F \to F \cdot *, \$/+/*/a/b$

I_4 :- $F \to a \cdot, \$/+/*/a/b$

I_5 :- $F \to b \cdot, \$/+/*/a/b$

I_6 :- $E \to E + \cdot T, \$/+$
$T \to \cdot F, \$/+$
$F \to \cdot F *, \$/+$
$F \to \cdot a, \$/+$
$F \to \cdot b, \$/+$

I_7 :- $E \to TF \cdot, \$/+$
$F \to F \cdot *, \$/+$

I_8 :- $F \to a \cdot, \$/+$

I_9 :- $F \to b \cdot, \$/+$

I_{10} :- $F \to F * \cdot, \$/+/*/a/b$

I_{11} :- $E \to E + T \cdot, \$/+$

I_{12} :- $T \to F \cdot, \$/+$
$F \to F \cdot *, \$/+$

I_{13} :- $F \to F * \cdot, \$/+$

Here $I_3 = I_{12}$, $I_4 = I_8$, $I_5 = I_9$
$I_{10} = I_{13}$ are same &
I_3, I_4, I_5, I_{10} are super set &
I_{12}, I_8, I_9, I_{13} are subset so we merge this item collections

$I_{312} \Rightarrow T \to F \cdot, \$/+/a/b$
$F \to F \cdot *, \$/+/*/a/b$

$I_{48} \Rightarrow F \to a \cdot, \$/+/*/a/b$

$I_{59} \Rightarrow F \to b \cdot, \$/+/*/a/b$

$I_{1013} \Rightarrow F \to F * \cdot, \$/+/*/a/b$

LALR Parsing table

item/ state no.	Action table					goto table		
	+	*	a	b	$	E	T	F
0			S_4	S_5		1	2	3
1	S_6				Accept			
2	R_2/S_8				R_2/S_9			7
312	R_4	S_{10}/S_{13}	R_4	R_4	R_4			
48	R_6	R_6	R_6	R_6	R_6			
59	R_7	R_7	R_7	R_7	R_7			
6	S_8				S_9		11	12
7	R_3	S_{13}			R_3			
1013	R_5	R_5	R_5	R_5	R_5			
11	R_1				R_1			

* Parsing with ambiguous grammar ⇒ There are certain types of ambiguous grammars that are useful in the specification & implementation of languages. An ambiguous grammar provides a shorter more nature specification than any equivalent unambiguous grammar.

Q.1) Consider ambiguous grammar

$E \to E+E$, $E \to E*E$ $E \to (E) / id$

Soln

Variable	First	follow
E	{(, id}	{+, *,), $}

1. $E \to E+E$, 2. $E \to E*E$, 3. $E \to (E)$ 4. $E \to id$

LR(0) item collections :-

$I_0 \Rightarrow$ $E' \to \cdot E$
$E \to \cdot E+E$
$E \to \cdot E*E$
$E \to \cdot (E)$
$E \to \cdot id$

$I_1 \Rightarrow$ $E' \to E \cdot$
$E \to E \cdot +E$
$E \to E \cdot *E$

$I_2 \Rightarrow$ $E \to (\cdot E)$
$E \to \cdot E+E$
$E \to \cdot E*E$
$E \to \cdot (E)$
$E \to \cdot id$

$I_3 \Rightarrow$ $E \to id \cdot$

$I_4 \Rightarrow$ $E \to E+\cdot E$
$E \to \cdot E+E$
$E \to \cdot E*E$
$E \to \cdot (E)$
$E \to \cdot id$

$I_5 \Rightarrow$ $E \to E*\cdot E$
$E \to \cdot E+E$
$E \to \cdot E*E$
$E \to \cdot (E)$
$E \to \cdot id$

$I_6 \Rightarrow$ $E \to (E \cdot)$
$E \to E \cdot +E$
$E \to E \cdot *E$

$I_7 \Rightarrow$ $E \to E+E \cdot$
$E \to E \cdot +E$
$E \to E \cdot *E$

$I_8 \Rightarrow$ $E \to E*E \cdot$
$E \to E \cdot +E$
$E \to E \cdot *E$

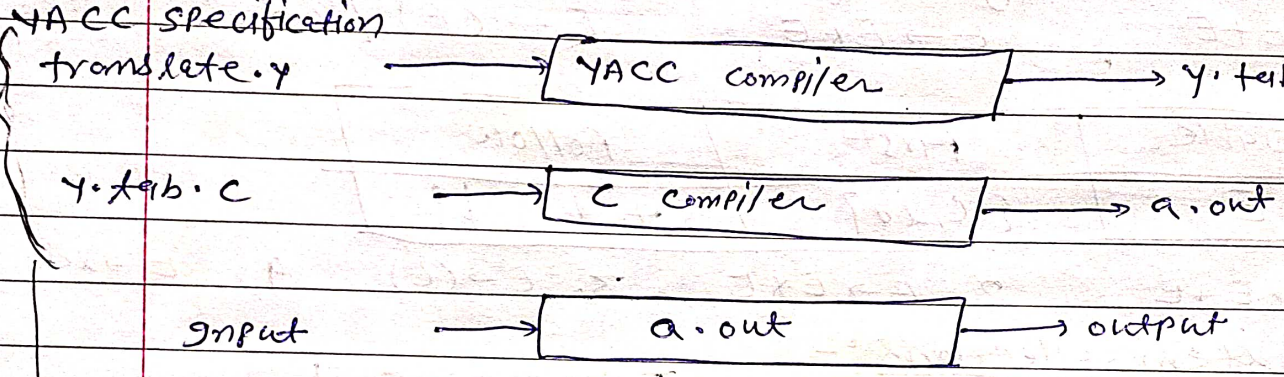
$I_9 \Rightarrow$ $E \to (E) \cdot$

Parsing table

item no / state no	Action						goto
	id	+	*	(	)	$	E
0	S_3			S_2			1
1		S_4	S_5			Accept	
2	S_3			S_2			6
3		R_4	R_4		R_4	R_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		R_1/S_4	R_1/S_5		R_1	R_1	
8		R_2/S_4	R_2/S_5		R_2	R_2	
9		R_3	R_3		R_3	R_3	

* LALR Parser generator "YACC" ⇒ A Parser generator can be used to facilitate the construction of the front end of a compiler.

LALR Parser generator "YACC" stands for another compiler-compiler. YACC constructed by S.C Johnson in 1970. YACC ~~constructed~~ is avaidable unix system command with the help of "YACC" to implement of hundreds (100) of compiler.



A translater can be constructed using "YACC" by first, a file "translate.y" containing a "YACC" specification of the translator is prepared. The unix command

YACC translate.y

Transforms the file translate.y into a C program called y.tab.C using the LALR method. The program y.tab.C representation of an LALR Parser written in C by Compiling y.tab.c along with the ly library that contains LR parsing program using this command cc y.tab.c -ly

we obtain the desired object program "a.out" that performs the translation ~~specified~~ specified by the original "YACC" program.

A "YACC" source program has three parts:-

1. Declarations %% %%
2. Translation rules %% %%
3. Supporting C-routines

→ diagram यहाँ बनाये