Eg 3):- S → TL;
T → int/float
L → L, id / id

w = int id id

(4) S → 0S0 / 1S1 / 2

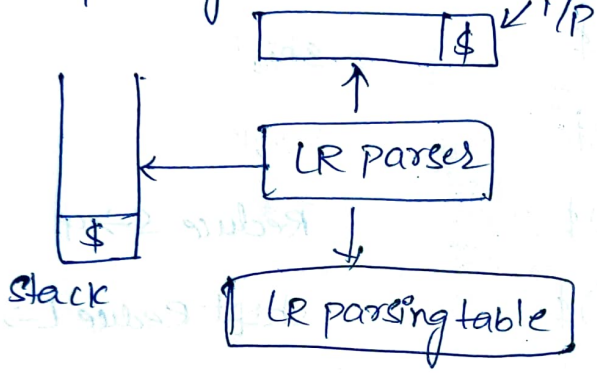
w = 10201

Introduction to LR parsing:- LR parser is a type of Bottom-up Paser

→ LR(k) parses scans i/p from left to Right order.

→ It use Right most Derivation in Reverse ~~order~~ to reduce i/p string to start symbol of gramm

K → No. of lookahead symbols that are used to make parsing decision



Stack – Datastructure used to store grammar symbols.

i/p – " " " " " i/p string to be parsed

LR paser → uses algorithm to make decisions.

LR parsing table – is constructed by using LR(0) items.

– These table uses two functions (i) closure (ii) Action.

Benefits of LR(k) parsing:-

→ most generec NON-Backtracking shift Reduced parsing Technique.

→ These parsers can recognize all programming languages for which CFG can be written.

→ They are capable of detecting syntactic errors as soon as possible while scanning of i/p.

Simple LR parsing:-

Eg:- $E \to E+T$
$E \to T$
$T \to T*F$
$T \to F$
$F \to (E)$
$F \to id$

① Augmented Grammars
$E' \to E$
$E \to E+T$
$E \to T$
$T \to T*F$
$T \to F$
$F \to (E)/id$

Steps to construct SLR parser
① Introduce augumented grammar
② Calculate canonical collection of LR(0) items.
③ Construct SLR parsing table by using (i) Goto (ii) Action functions

I0:
$E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

Goto $(I_0, E) \to I_1$
$E' \to E.$
$E \to E.+T$

$(I_0, T) \to I_2$
$E \to T.$
$T \to T.*F$

$(I_0, F) \to I_3$
$T \to F.$

$(I_0, ( ) \to I_4$
$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

$(I_0, id) \to I_5$
$F \to id.$

$(I_1, +) \to I_6$
$E \to E+.T$
$T \to .F$
$T \to .T*F$
$F \to .(E)$
$F \to .id$

$(I_2, *) \to I_7$
$T \to T*.F$
$F \to .(E)$
$F \to .id$

$(I_4, E) \to I_8$
$F \to (E.)$
$E \to E.+T$

$(I_4, T) \to I_2$
$E \to T.$
$T \to T.*F$

$(I_4, F) \to I_3$
$T \to F.$

$(I_4, ( ) \to I_4$
$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

$(I_4, id) \to I_5$
$F \to id.$

$(I_6, T) \to I_9$
$E \to E+T.$
$T \to T.*F$

$(I_6, F) \to I_3$
$T \to F.$

$(I_6, ( ) \to I_4$
$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to id$

$(I_6, id) \to I_5$
$F \to id.$

$(I_7, F) \to I_{10}$
$T \to T*F.$

$(I_7, ( ) \to I_4$
$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

$(I_7, id) \to I_5$
$F \to id.$

$(I_8,)) \to I_{11}$
$F \to (E).$

$(I_8, +) \to I_6$
$E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

$(I_9, *) \to I_7$

$T \to T * \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		r_2	S7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S5			S4			8	2	3
5	S5	r_6	r_6		r_6	r_6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r_1	S7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

1. $I_1 : E' \to E \cdot$
2. $I_2 : E \to T \cdot$
3. $I_3 : T \to F \cdot$ (r_4)
4. $I_5 : F \to id \cdot$
5. $I_9 : E \to E + T \cdot$

$I_{10} : T \to T * F \cdot$

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

Follow(E') = $

$E' \to \cdot E$

1 $E \to E + T$

2 $E \to T$ (r_2)

3 $T \to T * F$

4 $T \to F$ (r_4)

5 $F \to (E)$

6 $F \to id$

Follow(E') = $

Follow (E) = {+,), $}

Follow(T) = {$, +,), *}

Follow (F) = {$, +,), *}

(2, $) (2, +) (2,)) — r_2

(3, $) (3, +) (3,)) (3, *) — r_3

(5, $) (5, +) (5,)) (5, *) — r_6

$I_1 : S' \to S \cdot$ 2 & 4

w = id * id + id

STACK	INPUT	ACTION
$0	id*id+id$	shift 5
$0id5	*id+id$	reduce 6 F→id
$0F3	*id+id$	reduce 4 T→F
$0T2	*id+id$	S7
$0T2*7	id+id$	S5
$0T2*7id5	+id$	r6 F→id
$0T2*7F10	+id$	r3 T→T*F
$0T2	+id$	r2 E→T
$0E1	+id$	S6
$0E1+6	id$	S5
$0E1+6id5	$	r6 F→id
$0E1+6F3	$	r4 T→F
$0E1+6T9	$	r1 E→E+T
$0E1	$	Acceptance

Eg:-2 S → As/b

A → SA/a

w = abab

→ shift → while performing shifting first we have to shift i/p symbol on to the top of the stack & then shift the state Numbers.

→ reduce → while performing reduce operation. (if F→id we have reducing), if Right hand side contains one symbol we need to pop two symbols from top of the stack.

(ii) CLR LR(0) parser:-

Steps for construct LR(0) parser

(1) Introduce Augumented grammar

(2) calculate canonical collection of LR(0) items.

(3) construct LR(0) parsing table by using (i) Goto (ii) Action functions.

w = string = aabb$

Eg:- $S \to AA$

$A \to aA/b$

Augumented grammar

$S' \to S$

$S \to AA$

Closure:-

LR(0) items I_0

$S' \to .S$

$S \to .AA$

$A \to .aA/.b$

Goto

$(I_0, S) \to I_1$

$S' \to S.$

Goto $(I_0, A) \to I_2$

$S \to A.A$

$A \to .aA/.b$

Goto $(I_0, a) \to I_3$

$A \to a.A$

$A \to .aA/.b$

$(I_0, b) \to I_4$

$A \to b.$

$(I_2, A) \to I_5$

$S \to AA.$

$(I_2, a) \to I_3$

$A \to a.A$

$A \to .aA/.b$

$(I_2, b) \to I_4$

$A \to b.$

$(I_3, a) \to I_3$

$A \to a.A$

$A \to .aA/.b$

$(I_3, A) \to I_6$

$A \to aA.$

$(I_3, b) \to I_4$

$A \to b.$

LR(0) parsing Table:

	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_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

write down the states that contains "." at the end of the production.

$I_1 \to S' \to S.$

$I_4 \to A \to b.$

$I_6 \to A \to aA.$

$I_5 \to S \to AA.$

Given Grammar

$S \to AA$ — (1)

$A \to aA$ — (2)

$A \to b$ — (3)

→ The given string w = aa bb ⇒ w = aabb$

Stack	Input	Action
$0	aabb$	shift 3
$0a3	abb$	shift 3
$0a3a3	bb$	shift 4
$0a3a3b4	b$	reduce r3 A → b
$0a3a3	b$	shift 4
$0a3a3b4	$	

→ while performing shifting it we have to shift i/p symbol on to the top of the stack & then state Numbers

Stack	Input	Action
$0	aabb$	S3
$0a3	abb$	S3
$0a3a3	bb$	S4
$0a3a3b4	b$	reduce r3 A → b
$0a3a3A6	b$	r2 A → aA
$0a3A6	b$	r2 A → aA
$0A2	b$	S4
$0A2b4	$	r3 A → b
$0A2A5	$	r1 S → AA
$0S1	$	Accepted.

String w = aabb is parsed though the grammar by using LR(0) parser.

Canonical LR parsing :-

Step 1 :- Augumented grammar

Eg :- $S \to CC$
$C \to ac$
$C \to d$

$S' \to S$
$S \to CC$
$C \to ac$
$C \to d$

$w = add$

Step 2 : LR(1) items = LR(0) + lookahead

I_0 : $S' \to .S, \$$
$S \to .CC, \$$
$C \to .ac, a/d$
$C \to .d, a/d$

$(I_0, S) \to I_1$
$S' \to S., \$$

$(I_0, C) \to I_2$
$S \to C.C, \$$
$C \to .ac, \$$
$C \to .d, \$$

$(I_0, a) \to I_3$
$C \to a.c, a/d$
$C \to .ac, a/d$
$C \to .d, a/d$

$(I_0, d) \to I_4$
$C \to d., a/d$

$(I_2, C) \to I_5$
$S \to CC., \$$

$(I_2, a) \to I_6$
$S \to a.c, \$$
$C \to .ac, \$$
$C \to .d, \$$

$(I_2, d) \to I_7$
$C \to d., \$$

$(I_3, C) \to I_8$
$C \to ac., a/d$

$(I_3, a) \to I_3$
$C \to a.c, a/d$
$C \to .ac, a/d$
$C \to .d, a/d$

$(I_3, d) \to I_4$
$C \to d., a/d$

$(I_6, C) \to I_9$
$S \to ac., \$$

$(I_6, a) \to I_6$
$C \to a.c, \$$
$C \to .ac, \$$
$C \to .d, \$$

$(I_6, d) \to I_7$
$C \to d., \$$

Step3:- construction of CLR parsing Table.

	Action			Goto	
	a	d	$	S	C
0	S_3	S_4		1	2
1			Accepted		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3 (C→d)	r_3 (C→d)			
5			r_1 (S→CC)		
6	S_6	S_7			9
7			r_3 (C→d)		
8	r_2 (C→ac)	r_2 (C→ac)			
9			r_2 (C→ac)		

1: S→CC
2: C→ac
3: C→d

I_1: S'→S.$
I_4: C→d., a/d
I_5: S→CC., $
I_7: C→d., $
I_8: C→ac., a/d
I_9: S→ac., $

LALR Table:- $I_3 = I_6$
$I_4 = I_7$
$I_8 = I_9$

	Action			Goto	
	a	d	$	S	C
0	S_{36}	S_{47}		1	2
1			Acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

LALR parsing table

w = add

Stack	i/p buffer	Action
$0	add$	S36
$0a36	dd$	S47
$0a36d47	d$	r_3 C→d
$0a36C89	d$	r_2 C→ac
$0C2	d$	S47
$0C2d47	$	r_3 C→d
$0C2C5	$	r_1 S→CC
$0S1	$	Accepted

→ In LALR parses combine the same states that are having different lookahead symbols.

Canonical LR Parser:- (CLR parser)

Eg: $S \to CC$
$C \to cC$
$C \to d$

Step1:- Augumented grammar

$S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

Step2:- Calculating LR(1) items

$I_0 \to S' \to .S, \$$
$S \to .CC, \$$
$C \to .cC, c/d$
$C \to .d, c/d$

$(I_0, S) \to I_1$
$S' \to S., \$$

$(I_0, C) \to I_2$
$S \to C.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

$(I_0, c) \to I_3$
$C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$

$(I_0, d) \to I_4$
$C \to d., c/d$

$(I_2, C) \to I_5$
$S \to CC., \$$

$(I_2, c) \to I_6$
$C \to c.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

$(I_2, d) \to I_7$
$C \to d., \$$

$(I_3, C) \to I_8$
$C \to CC., c/d$

$(I_3, c) \to I_3$
$C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$

$(I_3, d) \to I_4$
$C \to d., c/d$

$(I_6, C) \to I_9$
$C \to CC., \$$

$(I_6, c) \to I_6$
$C \to c.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

$(I_6, d) \to I_7$
$C \to d., \$$

construction of CLR parsing tables:-

	Action			Goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			Accepted		
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_1	r_1			

1 $S' \to S$
2 $S \to CC$ — 1
3 $C \to cC$ — 2
4 $C \to d$ — 3

I_1: $S' \to S., \$$
I_4: $C \to d., c/d$
I_5: $CC., \$$
I_7: $C \to d., \$$
I_8: $C \to CC, c/d$
I_9: $C \to cC, \$$

Parse the input string w = dd$

Stack	Input	String
$0	dd$	shift 4
$0d4	d$	reduce 3 C→d
$0C2	d$	shift 7
$0C2d7	$	reduce 3 C→d
$0C2C5	$	reduce 2, S→CC
$0S1	$	Accepted

↳ So, the given i/p string is accepted by the grammar.

↳

LALR parser := (Look Ahead LR parser)

Eg(1) := steps to construct LALR parser

(1) Introduce Augumented grammar
(2) calculate the LR(1) items
(3) Contruction of LALR parse table.
(4) Parsing of given i/p string.

Eg1 := construct LALR parser for

S→CC
C→aC
C→d

and also parse the string w=add.

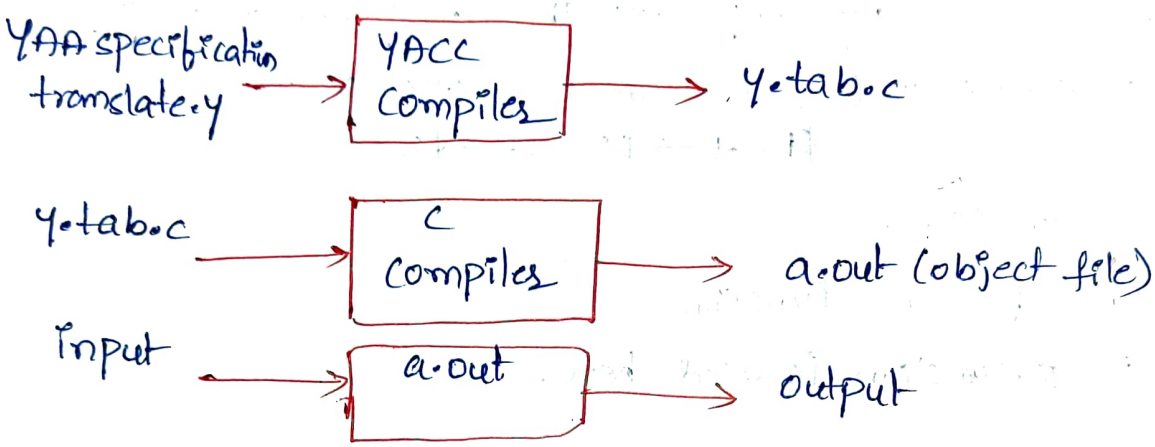
Parser generator :-

→ We use LALR parser generator YACC.

→ YACC → "Yet another compiler compiler"

→ YACC is a tool to generate parser, YACC accepts an tokens as i/p and produces parse tree as o/p.

The parser generator YACC :-



→ translate.y consists of YACC specification

Structure of YACC program :-

it has three parts.

```
declarations
%%
    translation Rules
%%
    Auxilliary functions
```

Declarations :-

→ used to declare the C variables and constants, & headers files are also specified here.

Syntax :- `%{ ... %}`

Eg :-
```
%{
    int a,b;
    Const int a=20;
    #include<stdio.h>
%}
```

(i) Translation Rules:-

→ Translation Rules are enclosed between "%{" & "%}"

Syntax:- head → body1 | body2 | body3. Eg:- C → aa|bb

head : body1 {semantic action}
| body2 {semantic action}
| body3 {semantic action}

$$ → represent LHS attribute value, to access the non-terminal value we use $$

→ In translation Rules we use two symbols $$, $i $i →

(ii) Auxilliary function:- $i → represent the ith symbol of body.

Eg:- E → E+T (if we want to access E, we have to use $1, + → $2, T → $3)

(iii) Auxilliary function:-

→ used to define "C" function.

→ yylex is function is used here.

Eg:- YACC specification (program) of simple desk calculator.

Ex:- E → E+E/T
T → T*F/F
F → (E)/id.

Declaration part:-

```
%{
   #include<ctype.h>
%}
% token DIGIT → specification of RE
%%                    ↗ it specifies I/P to desk calculator is an exp followed by \n
line : expr '\n' {printf("%d\n", $1);}
expr : expr '+' term { $$ = $1+$3; } $
     | term
term : term '*' factor { $$ = $1 * $3; }
     | factor
```

```
factor : '(' expr ')' { $$ = $2; }
       | DIGIT;
%%
yylex()
{
  int c;
  c = getchar();
  if(isdigit(c))
  {
    yylval = c - '0';
    return DIGIT;
  }
  return c;
}
```

yylex → is a lexical analyzer function which takes our source program as input and produces token as o/p

1) Using Ambiguous Grammars

→ For language constructs like expressions, an ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar.

→ Another use of ambiguous grammar is in isolating commonly occurring syntatic constructs for special-case optimization, we can add new productions to grammar.

→ Sometimes ambiguity rules allow only one parse tree, in such case it is unambiguous, it is possible to design an LR parser that allows same ambiguity resolving choices.

Precedence & Associativity to Resolve Conflicts:-

→ Consider ambiguous grammar with operators '+' & '*'

$$E \rightarrow E+E \mid E*E \mid (E) \mid id \quad \text{—(1)}$$

→ $E \rightarrow E+T$, $T \rightarrow T*F$, generates same language gives lower precedence to '+' than *, makes left associative. [use ambiguous grammar]

→ First we change associativity and precedence of operators + & * without disturbing above grammar.

→ Second, the parser for the unambiguous grammar will spend a substantial fraction of its time reducing by the productions $E \rightarrow T$ & $T \rightarrow F$

→ $E' \rightarrow E$, (1) is ambiguous there will be parsing-action conflicts when we try to produce LR parsing table.

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

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

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

I_3 : $E \to id \cdot$

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

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

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

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

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

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

LR(0) items of augmented exp grammar

Conflict occur in I_7 & I_8 $E \to E+E \cdot$, $E \to E*E \cdot$

PREFIX	STACK	INPUT
E+E	0147	*id$

If i/p is id + id.

State	ACTION						GOTO
	id	+	*	(	)	$	
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_5		r_1	r_1	
8		r_2	r_2		r_2	r_2	
9		r_3	r_3		r_3	r_3	

Fig :- parsing table for grammar

"Dangling-Else" Ambiguity :-

stmt → if expr then stmt else stmt
| if expr then stmt
| other

Consider grammar, an abstraction of this grammar where 'i' stands for if expr then, e stands for else and 'a' stands for "all other production".

S' → S (augmented grammar)
S → iSeS / iS / a

I_0 : S' → ·S
S → ·iSeS
S → ·iS
S → ·a

I_1 : S' → S·

I_2 : S → i·SeS
S → i·S
S → ·iSeS
S → ·iS
S → ·a

I_3 : S → a·

I_4 : S → iS·eS

I_5 : S → iSe·S
S → ·iSeS
S → ·iS
S → ·a

I_6 : S → iSeS·

Fig :- LR(0) states for augmented grammar

if expr then stmt — (2)

Ambiguity arises in I_4 shift/reduce conflict

S → iS·eS calls for a shift of e

FOLLOW(S) = {e, $}, S → iS· call for reduction by S → iS on i/p 'e'

In (2) should we shift else onto stack or reduce if expr then stmt, we should shift else because it is associated with previous then.

SLR parsing table is Constructed.

STATE	ACTION				GOTO
	i	e	a	$	S
0	S2		S3		1
1				accepted	
2	S2		S3		4
3		r3		r3	
4		S5		r2	
5	S2		S3		6
6		r1		r1	

input is iiaea, At line (5) state 4 selects the shift action on input e, whereas at line (9) state 4 calls for reduction by S→is on input $.

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		iiaea $	Shift
(2)	02	i	iaea $	shift
(3)	022	ii	aea $	shift
(4)	0223	iia	ea $	shift
(5)	0224	iiS	ea $	reduce by S→a
(6)	02245	iiSe	a $	shift
(7)	022453	iiSea	$	reduce by S→a
(8)	022456	iiSeS	$	reduce by S→iSeS
(9)	024	iS	$	reduce by S→iS
(10)	01	S	$	accept

Fig:- parsing actions on input iiaea