end marker $.

LR(1) items = LR(0) items + lookahead

How to add lookahead with the production?

Case 1: $A \to \alpha . BC, a$

Suppose this is 0th production.

Now '.' precedes B, so we have to write B's production as well

$B \to .D$ [1st production]

The look ahead of this production is given as we look at previous productions ie., 0th. whatever is after B, we find First of that value that is look a head of 1st production.

assume First(C) = d,

$B \to .D, d$

Case 2: Now if the 0th production was like $A \to \alpha . B, a$

Here after B there's nothing after B so lookahead is

$B \to .D, a$

Case 3: Assume $A \to a/b$

$A \to a, \$$

$A \to b, \$$

Steps to Construct CLR parsing table:

1. Writing augmented grammar.
2. LR(1) collection of items to be found.
3. Defining 2 functions: goto[list of terminals] and action[list of nonterminals] in CLR Parsing.

Ex: $S \to CC$ String = dd

$C \to cC/d$

Step 1: Write the Augmented grammar and LR item.

$S' \to .S$ 0

$S \to .CC$ 1

$C \to .cC$ 2

$C \to .d$ 3

$I_0 \to S' \to .S, \$$

$S' \to .CC, \$$

$C \to .cC, c/d$

$C \to .d, c/d$

goto(I_0, S) :

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

goto (I_0, C) :

$I_2 \rightarrow S \rightarrow C.C$, $\$$

$C \rightarrow .CC$, C/d $\$$

$C \rightarrow .d$, C/d $\$$

goto(I_0, c) :

$I_3 \rightarrow C \rightarrow c.C$, c/d

$C \rightarrow .cC$, c/d

$C \rightarrow .d$, c/d

goto(I_0, d) :

$I_4 \rightarrow C \rightarrow d.$, C/d

goto(I_2, C) :

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

goto (I_2, c)

$I_6 -$ $\begin{cases} C \rightarrow c.C, \$ \\ C \rightarrow .CC, \$ \\ C \rightarrow .d, \$ \end{cases}$

goto (I_2, d) :

I_4 : $C \to d.\ ,\ \$$

goto (I_3, C)

$I_8 \to C \to cC.\ ,\ \$$ C/d

goto (I_3, C)

$C \to c \cdot C, C/d$

$C \to . cC, C/d$ } I_3

$C \to . d\ , C/d$

goto (I_3, d)

$C \to d.; C/d$ — I_4

goto (I_6, C)

$C \to cC.\ ,\ \$$ — I_9

goto (I_6, C)

$C \to c.C, \$$

$C \to . cC, \$$ } I_6

$C \to . d, \$$

goto (I_6, d)

$C \to d.\ ,\ \$$ — I_7

Step 8: Construction of CLR Parse table.

	Terminal			Nonterminals	
	Action			goto	
~~Ao~~	c	d	$	S	C
I_0	S_3	S_4		I_1	I_2
I_1			Accepted		
I_2	S_6	S_7			I_5
I_3	S_3	S_4			I_8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			I_9
I_7			r_3		
I_8	r_2	r_2			
I_9	r_2		r_2		

Reduce can be find out by seeing '.' operator is present on last of the grammar.

Stack	Input string	Action
$0	dd$	Shift 4
$0d4	d$	Reduce C→d if we have one symbol on right side then pop 2 symbols.
$0C2	d$	Shift 7
$0C2d7	$	Reduce C→d
$0C2C5	$	Reduce S→CC
$0S1	$	Accepted.

3) SLR Parser:

SLR is Simple LR. It is the smallest class of grammar having few number of states. SLR is very easy to construct and similar to LR parsing.

Steps to construct SLR parse table:

1. Writing augmented grammar.
2. LR(0) Collection of items
3. Find follow of LHS of production
4. Defining 2 functions: goto and action.

Ex: $S \to AS/b$

$A \to SA/a$

Step 1: Writing Augmented Grammar

$S' \to S$

$S \to AS$

$S \to b$

$A \to SA$

$A \to a$

Step 2: Calculating Canonical collection of LR(0) items.

$I_0 \to S' \to .S$ 0

$S \to .AS$ 1

$S \to .b$ 2

$A \to .SA$ 3

$A \to .a$ 4

goto(I_0, S)

I_1 $S' \to .S = S' \to S.$ (.? present at final then it is final item)

$A \to .SA = A \to S.A$

$A \to .SA$

$A \to .a$

$S \to .AS$

$S \to .b$

goto(I_0, A)

$S \to .AS = S \to A.S$

I_2 — $S \to .AS$

$S \to .b$

$A \to .SA$

$A \to .a$

goto(I_0, b)

$I_3 - S \to b.$

goto(I_0, a)

$I_4 - S \to a.$

goto(I_1, A)

I_5 $A \to SA.$

~~$A \to S.A$~~

$S \to A.S$

$S \to .AS$

$S \to .b$

$A \to .SA$

$A \to .a$

goto(I_1, S)

I_6 $A \to S.A$ $S \to .AS$

$A \to .SA$ $S \to .b$

$A \to .a$

goto (I_1, a):

I_4 — $A \to a\cdot$

goto (I_1, b)

I_3 — $S \to b\cdot$

goto (I_2, S):

I_7 $S \to AS\cdot$

$A \to S\cdot A$

$A \to \cdot SA$

$A \to \cdot A$

$S \to \cdot AS$

$S \to \cdot b$

goto (I_2, A)

I_2 — $S \to A\cdot S$

$S \to \cdot AS$

$S \to \cdot b$

$A \to \cdot SA$

$A \to \cdot a$

goto (I_2, a)

$A \to a\cdot$ — I_4

goto (I_2, b)

$S \to b\cdot$ — I_3

goto(I_5, S)

$S \to AS.$
$A \to S.A$
$A \to .SA$
$A \to a$
$S \to .AS$
$S \to .b$
} I_4

goto(I_5, a)

$A \to a.$ I_4

goto(I_5, b)

$S \to b.$ — I_3

goto(I_5, A)

$S \to A.S$
$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$
} I_2

goto(I_6, A)

I_5 $A \to SA.$
$S \to A.S$
$S \to .AS$
$S \to .b$

$A \to .SA$
$A \to .a$

goto (I_6, S)

$A \to S.A$
$A \to .SA$
$A \to a$
$S \to .AS$
$S \to b$
} I_6

goto (I_6, a)

$A \to a.$ — I_4

goto (I_6, b)

$S \to b.$ — I_3

goto (I_7, A)

$A \to SA.$
$S \to A.S$
} I_5

goto (I_7, S)

$A \to S.A$
$A \to .SA$
$A \to .a$
} I_6

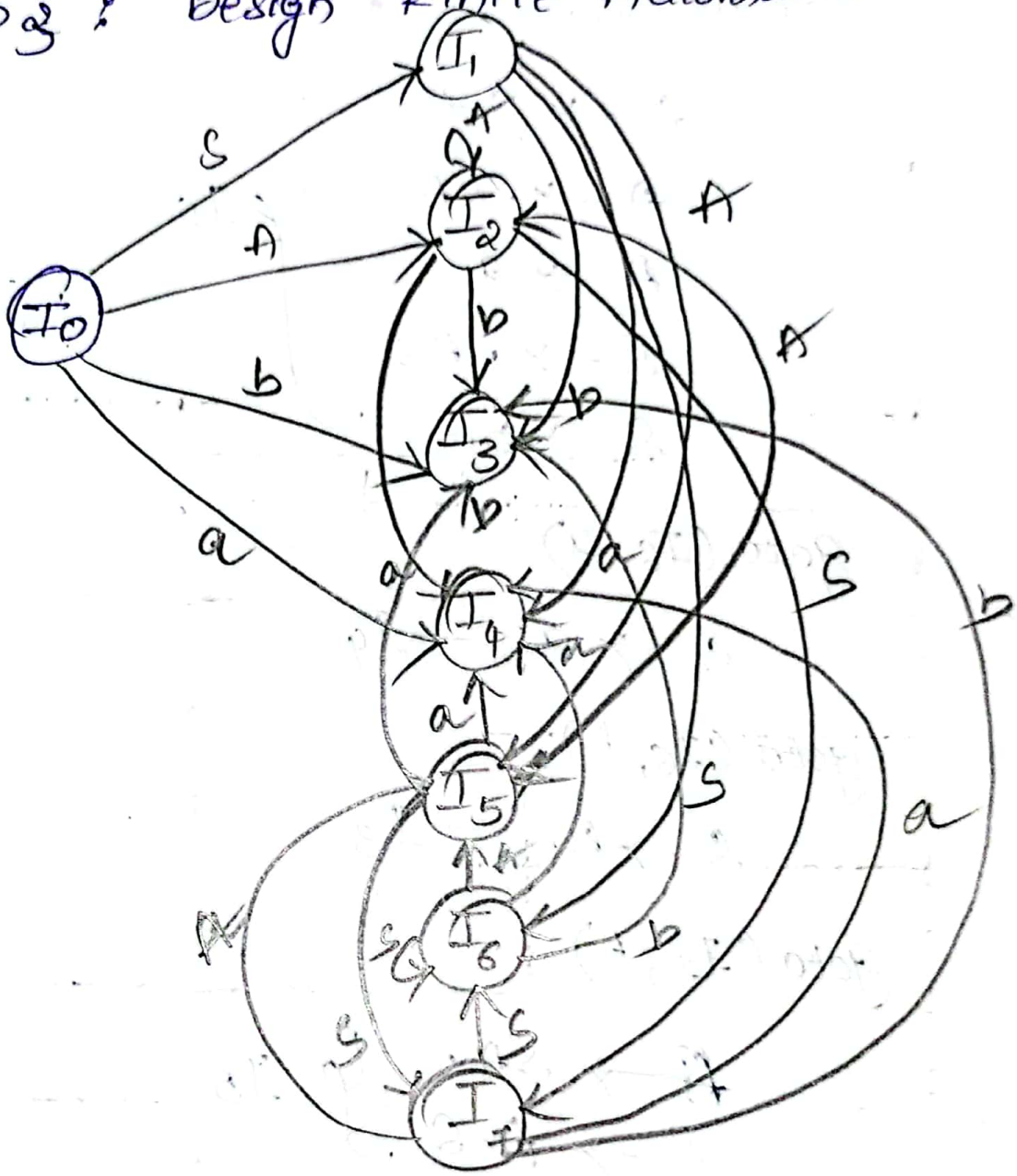
goto (I_7, a)

$A \to a.$ — I_4

goto (I_7, b)

$S \to b.$ — I_3

Step 3 : Design Finite Automata



Step 4: First(S) = {b, a}

First(A) = {a, b}

Follow(S) = {$, a, b}

Follow(A) = {a, b}

Construction of parse table.

	Action			Goto	
	a	b	$	S	A
I_0	S_4	S_3		I_1	I_2
I_1	S_4	S_3	Accepted	I_5	I_5
I_2	S_4	S_3		I_7	I_2
I_3	r_2	r_2	r_2		
I_4	r_4	r_4			
I_5	S_4/r_3	S_3/r_3		I_7	I_2
I_6	S_4	S_3		I_6	I_5
I_7	S_4/r_1	S_3/r_1	r_1	I_6	I_5

Step 5: Check whether the string will accept by the parser or not.

String = abab

Stack	I/ps	Action
$0	abab$	Shift 4
$0a4	bab$	Reduce A→a
$0A2	bab$	Shift 3
$0A2b3	ab$	Reduce S→b
$0A2S7	ab$	Reduce S→AS
$0S1	ab$	Shift 4
$0S1a4	b$	Reduce A→a
$0S1	b$	~~Shift 5~~
$0S1A5	b$	Reduce A→S A
$0A2	b$	Shift 3
$0A2b3	$	S→b
$0A2S7	$	Reduce S→AS
$0S1	$	Accepted.

4) LALR Parser

LALR Parser is lookahead LR parser. It is the most powerful parser which can handle large classes of grammar. The size of CLR parsing is quite large as compared to other parsing table. LALR reduces the size of this table. LALR works similar to CLR. It combines the similar states of CLR parsing table into single state.

LR(1) Items = LR(0) Items + lookahead

Steps for constructing the LALR parsing table:

1. Writing the augmented grammar.
2. LR(1) collection of items to be found.
3. Defining 2 functions: goto and action.

Ex: $S \rightarrow L = R / R$

$L \rightarrow *R / id$

$R \rightarrow L$

String = $id = id$

Step 1: Augmented Grammar

$S' \rightarrow S$

$S \rightarrow L = R / R$

$L \rightarrow *R / id$

$R \rightarrow L$

Step 2: Calculating LR(0) items.

I_0 $S' \rightarrow \cdot S, \$$ —— 0

$S \rightarrow \cdot L = R, \$$ —— 1

$S \rightarrow \cdot R, \$$ —— 2

$L \rightarrow \cdot *R, =/\$$ —— 3

$L \rightarrow \cdot id, =/\$$ —— 4

$R \rightarrow \cdot L, \$$ —— 5

goto (I_0, S)

I_1 $S \rightarrow L \cdot = R, \$$

$S' \rightarrow S \cdot, \$$

goto (I_0, L)

I_2 $S \rightarrow L \cdot = R, \$$

$R \rightarrow L \cdot, \$$

goto (I_0, R)

I_3> $S \to R.$, $

goto(I_0, *)

I_4> $L \to *.R$, =/$

$R \to .L$, =/$

$L \to .*R$, =/$

$L \to .id$, =/$

goto (I_0, id)

I_5> $L \to id.$, $

goto (I_2, =)

I_6> $S \to L = .R$, $

$R \to .L$, $

$L \to .*R$, $

$L \to .id$, $

goto (I_4, R)

I_7> $L \to *R.$, =/$

$R \to$

goto (I_4, L)

$R \to L.$, =/$ — I_8

goto (I_4, *)

$L \to *.R$, $ — I_4

goto(I_4, id)

$L \rightarrow id \cdot ; / \$$ — I_5

goto(I_6, R)

$S \rightarrow L = R \cdot , \$$ — I_9

goto(I_6, L)

$R \rightarrow L \cdot , \$$ — I_{10}

goto(I_6, *)

$L \rightarrow * \cdot R , \$$

$R \rightarrow \cdot L , \$$

$L \rightarrow \cdot * R , \$$

$L \rightarrow \cdot id , \$$

} I_{10}

goto(I_6, id)

$L \rightarrow id \cdot , \$$ — I_{12}

goto(I_{10}, R)

$L \rightarrow * R \cdot ; \$$ — I_{13}

goto(I_{11}, L)

$R \rightarrow L \cdot , \$$ — I_{10}

goto(I_{11}, *)

$L \rightarrow * \cdot R , \$$ — I_{11}

goto (I_{11}, id)

$L \rightarrow id \cdot, \$ \quad - I_{12}$

Combine the similar States grammar

$I_4 \& I_{11} \rightarrow I_{411}$

$I_5 \& I_{12} \rightarrow I_{512}$

$I_7 \& I_{13} \rightarrow I_{713}$

$I_8 \& I_{10} \rightarrow I_{810}$

Step 3: Construct CALR parse table

	Action				goto		
	id	=	*	$	S	L	R
0	S_{512}		S_{411}		1	2	3
1				Accept			
2		S_6		r_5			
3				r_2			
411	S_{512}		S_{411}			810	713
512		r_4		r_4			
6	S_{512}		411			810	9
713		r_3		r_3			
810		r_5		r_5			
9				r_1			

Stack	I/p String	Action
$0	id=id$	Shift 5\12
$0id5\12	=id$	Shift reduce $\rightarrow id
$0R5\12	=id$	Shift 6
$0\12=6	id$	Shift 5\12
$0\12=6id5\12	$	reduce by $\rightarrow i
$0\12=6i8\12	$	reduce by $\rightarrow L
$0\12=6R9	$	reduce $\rightarrow L=R
$0S1	$	Accepted.

Ex: $S \rightarrow CC$
$C \rightarrow aC/b$
String: bb

Ex: $E \rightarrow E+T/T$
$T \rightarrow T*F/F$
$F \rightarrow (E)/id$
String: id*id+id

VI) Parser Generators :—

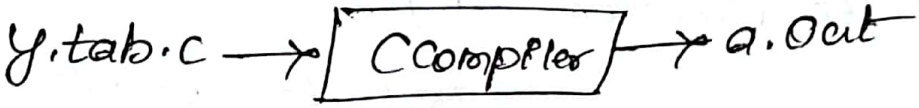
(or)

YACC Tool (or) YACC Parser Generator

(Yet Another Compiler Compiler)

1. YACC provides a tool to produce a parser for a given grammar.
2. YACC is a program designed to compile a LALR (1) grammar.
3. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
4. The input of YACC is the rule (of) grammar and the output is a (progra-m.

It is a tool to produce a parser. It accepts input from lexical analyzer (tokens) and produce parsers. It is also called as parser generators.



Structure of YACC tool.

YACC program similarly contain 3 sections.

1. declarations
2. translation rules
3. Auxiliary functions.

The Declaration Part:

There are two sections in the declarations part of Yacc program; In the first section, we put ordinary C declarations, delimited by %{ and %}, and in second section contains only include statement that causes the C preprocessor to include the standard header file <ctype.h>.

The translation Rules Part:

In this part the YACC Specification after the first %% pair, we put the translation rules. Each rules contain of a grammar production and the associated semantic actions.

<head> → <body$_1$> | <body$_2$> | .... | <body$_n$>

would be written as in YACC

<head> : <body1> { <Semantic action> }
| <body2> { <Semantic action> }
⋮
| <bodyn> { <Semantic action> }

Ex: $E \rightarrow E + T \mid T$

expr : expr '+' term { $$ = $1 + $3; }
| term
;

The Auxiliary functions:

It is mainly used to define C functions yylex() will be written in auxiliary functions.

YACC Program of Simple Desk Calculator

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

→ translate.y

```
%{
#include<ctype.h>  — ctype stands for conversion.
%}
%token DIGIT
%%
line : expr '\n' { printf("%d \n", $1)
expr : expr '+' term { $$ = $1 + $3 ;}
     | term ;
term : term '*' factor { $$ = $1 * $3}
     | factor }
factor : '(' expr ')' { $$ = $2 ;}
       | DIGIT ;
%%  — lexical analyzer
yylex()
{ int c ;
  c = getchar() ;
```

```
if(isdigit(c))
{
yylval = c - '0';
return c;
}
```