- Hence these gotos can not be applied to avoid repetition.
- There is no point applying goto on I_5. Hence now we will move ahead by applying goto on I_6 for T.

goto (I_6, T)

I_9 : $E \rightarrow E + T \bullet$

$T \rightarrow T \bullet * F$

Then

goto (I_7, F)

I_{10} : $T \rightarrow T * F \bullet$

Then

goto $(I_8,))$

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

- Applying goto on I_9, I_{10}, I_{11} is of no use.
- Thus there is no item that can be added in this set of items.
- The collection of set of items is from I_0 to I_{11}.

states	Action						Goto		
	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			

Parsing table for expression grammar

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

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

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

Construction of SLR parsing table :→

Input - An augmented grammar G'.

~~Output~~ -

Output - The SLR - parsing table functions Action and goto for G'.

Method -

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

2. state i is constructed from I_i. The passing actions for state i are determined as follows:

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

(b) If $[A \to \alpha .]$ is on I_i, then set action $[i, a]$ to "reduce $A \to \alpha$" for all a in follow (A); here A maynotbe S'.

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

If any conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule : If goto $(I_i, A) = I_j$, then goto $(i, A) = j$.

4. All entries not defined by rules (2) and (3) are made error.

5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot S]$.

moves of an LR parser on id * id + id using parsing table

	stack	Symbols	Input	Action
(1)	0		id * id + id $	shift
(2)	05	id	* id + id $	reduce by F→id
(3)	03	F	* id + id $	reduce by T→F
(4)	02	T	* id + id $	shift
(5)	027	T *	id + id $	shift
(6)	0275	T * id	+ id $	reduce by F→id
(7)	0270 10	T * F	+ id $	" T→T*F
(8)	02	T	+ id $	" E→T
(9)	01	E	+ id $	shift
(10)	016	E +	id $	shift
(11)	0165	E + id	$	reduce by F→id
(12)	0163	E + F	$	" T→F
(13)	0169	E + T	$	" E→E+T
(14)	01	E	$	Accept

...ible Prefixes →

- The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- They are defined as follows: a viable prefix is a prefix of a right sentential form that does not continue past the right end of the right-most handle of that sentential form.
- By this definition, it is always possible to add terminal symbols to the end of a viable prefix to obtain a right-sentential form.
- SLR parsing is based on the fact that LR(0) automata recognize viable prefixes.
- We say item $A \to \beta_1 \cdot \beta_2$ is valid for a (viable) prefix $\alpha\beta_1$ if there is a derivation $S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$.
- An item will be valid for many viable prefixes.
- The fact that $A \to \beta_1 \cdot \beta_2$ is valid for $\alpha\beta_1$ tells us a lot about whether to shift or reduce when we find $\alpha\beta_1$ on the parsing stack.
- In particular if $\beta_2 \neq \epsilon$, then it suggests that we have not yet shifted the handle onto the stack, so shift is our move.

- If $\beta_2 = \epsilon$, then it looks as if $A \to \beta_1$ is handle, and we should reduce by this pro

Canonical - LR Parser

- The canonical - LR or just LR method, which makes full use of the look ahead symbol(s).
- This method uses a large set of items, called the LR(1) items.

Canonical LR(1) items :- →

(i) For the grammar G initially add $S' \to \cdot S$ in the set of item C.

(ii) For each set of items I_i in C and for each grammar symbol X (may be terminal or non terminal) add closure (I_i, X).

- This process should be repeated by applying goto (I_i, X) for each X in I_i such that goto (I_i, X) is not empty and not in C.
- The set of items has to constructed until no more set of items can be added to C.

(iii) The closure function can be computed as follows

for each item $A \to \alpha \cdot X\beta, a$ and rule $X \to \gamma$ and $b \in first(\beta, a)$ such that $X \to \cdot\gamma$ and b is not in I then add $X \to \cdot\gamma, b$ to I.

(iv) similarly the goto function can be computed as: for each item $[A \to \alpha \cdot X\beta, a]$ is in I and rule $[A \to \alpha X \cdot \beta, a]$ is not in goto items then add $[A \to \alpha X \cdot \beta, a]$ to goto items.

This process is repeated until no more set of items can be added to the collection C.

Algo.

```
set of Items closure (I) {
        repeat for (each item [A → α·Bβ, a] in I)
          for (each production B → γ in G')
            for (each terminal b in first (βa))
              add [B → ·γ, b] to set I;
        until no more items are added to I;
        return I;
    }
set of Items goto (I, X) {
        initialize J to be the empty set;
        for (each item [A → α·Xβ, a] in I)
            add item [A → αX·β, a] to set J;
        return closure (J);
    }
```

```
void items(G') {
    Initialize C to { closure({[S' → ·S, $]}).
    repeat
        for (each set of items I in C)
            for (each grammar symbol X)
                if (goto(I, X) is not empty and not in C)
                    add goto(I, X) to C;
    until no new sets of items are added to C;
}
```

Sets-of-LR(1)-items construction for grammar G'

Ex-

$S \to CC$

$C \to aC \mid d$

Initially add $S' \to \cdot S, \$$ as the first rule in I_0

None match

$S' \to \cdot S, \$$ with

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

Hence $S' \to \cdot S, \$$

$A \to \alpha \cdot X\beta, a$

$A \rightleftharpoons S', \quad \alpha = \epsilon, \quad X = S, \quad \beta = \epsilon, \quad a = \$$

If there is a production $X \to Y, b$ then add $X \to \cdot Y, b$

$\therefore \quad S \to \cdot CC \qquad b \in first(\beta, a)$

$b \in First(\epsilon \$)$ as $\epsilon \$ = \$$

$b \in First(\$)$

$b = \{\$\}$

$S \to \cdot CC$, \$ will be added in I_0.

Now $S \to \cdot CC$, \$ is in I_0 we will match it with

$A \to \alpha \cdot X \beta, a$

$A = S'$, $\alpha = \epsilon$, $X = C$, $\beta = C$, $a = \$$

If there is a production $X \to \gamma, b$ then add

$X \to \cdot \gamma, b$

$\therefore$ $C \to \cdot aC$ $\quad b \in \text{first}(\beta, a)$

$C \to \cdot d$ $\quad b \in \text{first}(C\,\$)$

$b \in \text{first}(C)$ as $\text{first}(C) = \{a, d\}$

$b = \{a, d\}$

$\therefore$ $C \to \cdot aC$, a or d will be added in I_0.

Similarly $C \to \cdot d$, a/d will be added in I_0.

Hence I_0:

$S' \to \cdot S, \$$

$S \to \cdot CC, \$$

$C \to \cdot aC, a/d$

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

Here a/d is used to denote a or d. That means for the production $C \to \cdot aC$, a and $C \to \cdot aC, d$.

Now apply goto on I_0.

$S' \to \cdot S, \$$

$S \to \cdot CC, \$$

$C \to \cdot aC, a/d$

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

Hence

goto (I_0, S)

I_1: $S' \to S \cdot, \$$

Now apply goto on C in I_0.

$S \to C \cdot C, \$$ add in I_2. Now as after dot C comes we will add the rules of C.

$X = C, \beta = \epsilon, a = \$$

$X \to \cdot \gamma, b$ where $b \in$ first (β, a)

$C \to \cdot aC$ $b \in$ first $(\epsilon, \$)$

$C \to \cdot d$ $b \in$ first $(\$)$

Hence

$C \to \cdot aC, \$$ and $C \to \cdot d, \$$ will be added to I_2.

I_2: goto (I_0, C)

$S \to C \cdot C, \$$

$C \to \cdot aC, \$$

$C \to \cdot d, \$$

we we will apply goto on a of I_0 for rule $C \to \cdot aC, a/d$ that becomes $C \to \cdot aC, a/d$. will be added in I_3.

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

As $A = C$, $\alpha = a$, $X = C$, $\beta = \epsilon$, $a = a/d$

Hence $X \to \cdot Y, b$

$C \to \cdot aC$ $\quad b \in \text{first}(\beta, a)$

$C \to \cdot d$ $\quad b \in \text{first}(\epsilon a)$ or $\text{first}(\epsilon d)$

$b = a/d$

Hence

I_3: goto(I_0, a)
$C \to a \cdot C, a/d$
$C \to \cdot aC, a/d$
$C \to \cdot d, a/d$

Now if we apply goto on d of I_0 in the rule $C \to \cdot d, a/d$ then we get $C \to d \cdot, a/d$ hence I_4 becomes

I_4: goto (I_0, d)
$C \to d \cdot, a/d$

As applying gotos on I_0 is over. we will move to I_1 but we get no new production from I_1 we will apply goto on C in I_2. And there is no closure possible in this state.

I_5: goto (I_2, C)
$S \to CC \cdot \$$

we will apply goto on a from I_2 on the rule $C \to \cdot aC, \$$ we get the state I_6.

$C \to a \cdot C, \$$

$A \to \alpha \cdot X\beta, a$

$A \to C, \alpha = a, X = C, \beta = G, a = \$$

$X \to \cdot\gamma$

$C \to \cdot aC$ and $C \to \cdot d$

$b \in First(\beta . a)$

$b \in First(G\$)$

$b = \$$

I_6 : goto (I_2, a)

$C \to a \cdot C, \$$

$C \to \cdot aC, \$$

$C \to \cdot d, \$$

* I_3 and I_6 are different because the second component in I_3 and I_6 is different.

Apply goto on d of I_2 for the rule $C \to \cdot d, \$$.

I_7 : goto (I_2, d)

$C \to d \cdot, \$$

Now if we apply goto on a and d of I_3 we will get I_3 and I_4 respectively and there is no

int in repeating the states. so we will apply goto on I_6 for C for the rule.

$C \rightarrow a \cdot C, \$$

I_9: goto (I_6, C)

$C \rightarrow aC \cdot \$$

For remaining states I_7, I_8 and I_9 we cannot apply goto. Hence the process construction of set of LR(1) items is completed. Thus the set of LR(1) items consists I_0 to I_9 states.

I_0:

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

$S \rightarrow \cdot CC, \$$

$C \rightarrow \cdot aC, a/d$

$C \rightarrow \cdot d, a/d$

I_1: goto (I_0, S)

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

I_2: goto (I_0, C)

$S \rightarrow C \cdot C, \$$

$C \rightarrow \cdot aC, \$$

$C \rightarrow d, \$$

I_3: goto (I_0, a)

$C \rightarrow a \cdot C, a/d$

$C \rightarrow \cdot aC, a/d$

$C \rightarrow \cdot d, a/d$

I_4: goto (I_0, d)

$C \rightarrow d \cdot, a/d$

I_5: goto (I_2, C)

$S \rightarrow CC \cdot, \$$

I_6: goto (I_2, a)

$C \rightarrow a \cdot C, \$$

$C \rightarrow \cdot aC, \$$

$C \rightarrow \cdot d, \$$

I_7: goto (I_2, d)

$C \rightarrow d \cdot, \$$

I_8: goto (I_3, C)

$C \rightarrow aC \cdot, a/d$

I_9: goto (I_6, C)

$C \rightarrow aC \cdot, \$$

Construction of Canonical LR Parsing Table

Algo - Construction of canonical-LR passing tables.

Input - An augmented grammar G'.

output - The canonical -LR parsing table functions Action and goto for G'.

Method -

1. construct $C' = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items for G'.

2. State i of the parser is constructed from I_i.

The parsing action for state i is determined as follows-

(a) If $[A \to \alpha \cdot a\beta, b]$ is in I_i and goto $(I_i, a) = I_j$, then set action $[i, a]$ to "shift j". Here a must be a terminal.

(b) If this is a production $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S'$ then set action $[i, a]$ to "reduce $A \to \alpha$".

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

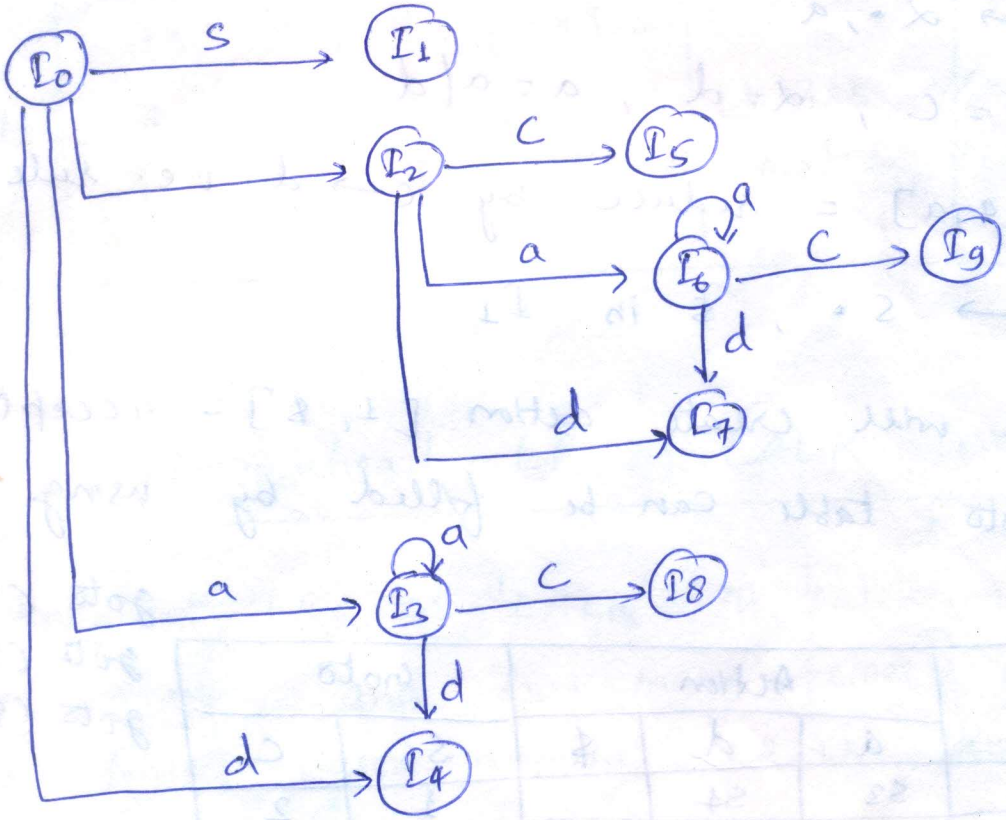
If any conflicting actions ~~for state~~ result from the above rules, we say the grammar is not LR(1).

The goto transitions for state i are constructed for all nonterminals A using the rule: If goto $(I_i, A) = I_j$, then goto $[i, A] = j$

4. All entries not defined by rules (2) and (3) are made 'error'.

5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot S, \$]$.

The DFA for the set of items can be drawn as follows –



DFA [goto graph]

consider I_0 in which there is a rule matching with $[A \rightarrow \alpha \cdot a\beta, b]$ as –

$C \rightarrow \cdot aC, a/d$ and if the goto is applied on a

then we get the state I_3. Hence we will create entry action [0, a] = shift 3.

similarly in I_0

$C \to \cdot d \quad a/d$

$A \to \alpha \cdot a\beta, b$

$A = C, \alpha = \epsilon, a = d, \beta = \epsilon, b = a/d$

goto $(I_0, d) = I_4$

hence action [0, d] = shift 4

For state I_4

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

$A \to \alpha \cdot, a$

$A = C, \alpha = d, a = a/d$

action [4, a] = reduce by $C \to d$ i.e rule 3

$S' \to S \cdot, \$$ in I_1

So we will create action [1, $] = accept

The goto table can be filled by using the goto functions

goto $(I_0, S) = I_1$
goto $(I_0, C) = I_2$
goto $(I_2, C) = I_5$

	Action			Goto	
	a	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Using the parsing table we can parse the input string aadd as

Stack	Input buffer	Action table	goto table	Passing Action
$0	aadd $	action [0,a] = s3		shift
$0a3	add $	" [3,a] = s3		shift
$0a3a3	dd $	" [3,d] = s4	~~[3,C] = 8~~	~~Reduce by~~ shift c→d
$0a3a3d4	d$	" [4,d] = r3	[3,C] = 8	Reduce by c→d
$0a3a3 C8	d$	" [8,d] = r2	[3,C] = 8	Reduce by C→aC
$0a3C8	d$	" [8,d] = r2	[0,C] = 2	Reduce by C→aC
$0C2	d$	" [2,d] = s7		shift
$0C2d7	$	" [7,$] = r3	[2,C] = 5	Reduce by c→d
$0C2C5	$	action [5,$] = r1	[0,S] = 1	Reduce by S→CC
$0S1	$	accept		

Lookahead - LR or LALR Parser

- The 'lookahead - LR' or 'LALR' is based on the LR(0) sets of items, and has many fewer states than typical parsers based on the LR(1) items.
- By carefully introducing lookaheads into the LR(0) items, we can handle many more grammars with the LALR method than with the SLR method, and build parsing tables that are no bigger than the SLR tables.

construction set of LR(1) items along with the lookahea

- The construction of LR(1) items is same as discussed earlier but difference is that: on construction of LR(1) items for LR parser, we have differed the two states of the second component is different.
- But on this case we will merge the two states by merging of first and second components from both the states.

Eg.

$S \to CC$

$C \to aC$

$C \to d$

I_0:
$S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot aC, a/d$
$C \to \cdot d, a/d$

I_1: goto (I_0, S)
$S' \to S\cdot, \$$

I_2: goto (I_0, C)
$S \to C\cdot C, \$$
$C \to \cdot aC, \$$
$C \to \cdot d, \$$

I_3: goto (I_0, a)
$C \to a\cdot C, a/d$
$C \to \cdot aC, a/d$
$C \to \cdot d, a/d$

I_4: goto (I_0, d)
$C \to d\cdot, a/d$

I_5: goto (I_2, C)
$S \to CC\cdot, \$$

I_6: goto I_2, a
$C' \to a\cdot C, \$$
$C \to \cdot aC, \$$
$C \to \cdot d, \$$

I_7 : goto (I_2, d)

$C \to d \bullet, \$$

I_8 : goto (I_3, C)

$C \to aC \bullet, a/d$

I_9 : goto (I_6, C)

$C \to aC \bullet, \$$

Now we will merge states 3,6 then 4,7 and 8,9.

I_0 :

$S' \to \bullet S, \$$

$S \to \bullet CC, \$$

$C \to \bullet aC, a/d$

$C \to \bullet d, a/d$

I_1 : goto (I_0, S)

$S' \to S \bullet, \$$

I_2 : goto (I_0, C)

$S \to C \bullet C, \$$

$C \to \bullet aC, \$$

$C \to \bullet d, \$$

I_{36} : goto (I_0, a)

$C \to a \bullet C, a/d/\$$

$C \to \bullet aC, a/d/\$$

$C \to \bullet d, a/d/\$$

I_{47} : goto (I_0, d)

$C \to d \bullet, a/d/\$$

I_5 : goto (I_2, C)

$S \to CC \bullet, \$$

I_{89} : goto (I_3, C)

$C \to aC \bullet a/d/\$$

→ We have merged two states I_3 and I_6 and made the second component as a or d or $.

→ The production rule will remain as it is. similarly in I_4 and I_7.

→ The set of items consists of states { I_0, I_1, I_2, I_{36}, I_{47}, I_5, I_{89} }.

Construction of LALR parsing table -

Input - An augmented grammar G_1'.

output - The LALR parsing table functions action and goto for G_1'.

method -

(I) Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.

(II) For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

(III) Let $C' = \{J_0, J_1, \ldots J_m\}$ be the resulting sets of LR(1) items.

- The parsing actions for state i are constructed from J_i in the same manner as in algo for canonical - LR parsing table.
- If there is a parsing action conflict, the algo. fails to produce a parser, and the grammar is said not to be LALR(1).

(IV) The goto table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \ldots \cup I_k$, then the cores of goto (I_1, X), goto (I_2, X) .. goto (I_k, X) are the same. Since $I_1, I_2, \ldots I_k$ all have the same core.

Let K be the union of all sets of items having the same core as goto (I_i, X). Then goto $(I, X) = K$.

passing table of previous e.g.

- consider the state I_0 there is a match with the rule $[A \to \alpha \cdot a\beta, b]$ and goto $(I_i, a) = I_j$.
- $C \to \cdot aC$, $a/d/\$$ and if the goto is applied on a then we get the state I_{36}.
- Hence we will create entry action $[0, a] = shift\ 36$.

→ In I_0

$C \to \cdot d$ a/d

$A \to \alpha \cdot a\beta, b$

$A = C$, $\alpha = \epsilon$, $a = d$, $\beta = \epsilon$, $b = a/d$

goto $(I_0, d) = I_{47}$

Hence $(I_0, d) = I_{47}$

Hence action $[0, d] = shift\ 47$

→ for state I_{47}

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

$A \to \alpha \cdot, a$

$A = C$, $\alpha = d$, $a = a/d/\$$

action $[47, a]$ = reduce by $C \to d$ i.e. rule 3

" $[47, d]$ = " " $C \to d$ " "

" $[47, \$]$ = " " $C \to d$ " "

$S' \to S\bullet, \$$ in I_1

So we will create action $[1, \$]$ = accept

The goto table can be filled by using the goto functions.

For instance goto $(I_0, S) = I_1$. Hence goto $[0, S] = 1$

Continuing in this fashion we can fill up the LR(1) parsing table as follows.

State	Action			goto	
	a	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Parsing the input string using LALR parser →

i/p string – aadd

stack	Input buffer	Action table	goto table	Parsing Action
$ 0	aadd $	action [0,a] = s36		
$0a36	add $	" [36,a] = s36		shift
$0a36a36	dd$	" [36, d] = s47		"
$0a36a36d47	d $	" [47,d] = r36	[36,C] = 89	Reduce by C→d
$0a36a36C89	d$	" [89,d] = r2	[36,C] = 89	" C→aC
$0a36C89	d$	action [89,d] = r2	[0,C] = 2	" C→aC
$0C2	d$	" [2,d] = s47		shift
$0C2d47	$	" [47,$] = r36	[2,C] = 5	Reduce by C→d
$0C2C5	$	" [5,$] = r1	[0,S] = 1	" S→CC
$0S1	$	accept		

A comparison of predictive parser with shift reduce parsers

Predictive Parser

Top down (LL) Parser

stack predicts what is to come

The stack initially contains the start symbol of the grammar

The stack is empty when the accept state is reached

shift Reduce Parser

Bottom-up (LR) Parser

stack shows what has been seen so far

stack is initially empty

The stack contains the start symbol of the gramm or when the accept state is reached.

Input tokens are popped off the stack	Input tokens are pushed on stack
Left sides of productions are popped off the stack	Right sides of productions are popped off the stack.
Right sides of productions are pushed on the stack	Left sides of productions are pushed on the stack.

Comparison of LR parsers

SLR Parser	LALR Parser	Canonical LR Parser
SLR parser is smallest on size	The LALR and SLR have the same size	LR parser or canonical LR parser is largest on size
It is an easiest method based on Follow funct.	This method is applicable to wider class than SLR	This method is most powerful than SLR and LALR.
This method exposes less syntactic features than that of LR parser.	most of the syntactic features of a language are expressed in LALR	This method exposes less syntactic features than that of LR parser.
Error detection is not immediate in SLR	Error detection is not immediate in LALR	Immediate error detection is done by LR parser.
It requires less time and space complexity	The time and space complexity is more but efficient method exist for constructing LALR parsers directly	The time and space complexity is more for canonical LR parser

Ambiguous Grammar And LR Parsers

It is a fact that every ambiguous grammar fails to be LR.

- If ambiguous grammar is used for parsing, it won't recognize the intended language.

Advantage -

- Ambiguous grammars provide much shorter and more natural specification for constructs like expressions as compared to unambiguous grammars.
- Ambiguous grammars helps in isolating few syntactic constructs for special case optimization.
- If we use ambiguous grammar for parsing, to remove conflicts, new productions can be added to the given grammar and can also specify special case construct.

using precedence and associativity to resolve parsing action conflicts -

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

Now we will build the set of LR(0) items for this grammar-

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

I_2: goto $(I_0, ()$
$E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot id$

I_3: goto (I_0, id)
$E \to id \cdot$

I_4: goto $(I_1, +)$
$E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot id$

I_5: goto $(I_1, *)$
$E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot id$

I_6: goto (I_2, E)
$E \to (E \cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_7: goto (I_4, E)
$E \to E + E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_8: goto (I_5, E)
$E \to E * E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_9: goto $(I_6,))$
$E \to (E) \cdot$

Now compute Follow (E) = {+, *,), $}

arsing Table -

State	Action						Goto
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5			Accept	
2	S3			S2			6
3		r4	r4		r4	r4	
4	S3			S2			7
5	S3			S2			8
6		S4	S5		S9		
7		S4 or r1	S5 or r1		r1	r1	
8		S4 or r2	S5 or r2		r2	r2	
9		r3	r3		r3	r3	

- Shift/Reduce conflicts occur at state 7 and 8.

Now consider one string "id + id * id".

Stack	Input	Action with conflict resolution
$0	id+id * id $	shift
$0id3	+id * id $	Reduce by E→id
$0E1	+id * id $	shift
$0E1+4	id *id $	shift
$0E1+4id3	* id$	Reduce by E→id
$0E1+4E7	* id$	conflict can be resolved by shift 5
$0E1+4E7*5	id $	shift
$0E1+4E7*5id3	$	Reduce by E→id
$0E1+4E7*5E8	$	" " E→E*E
$0E1+4E7	$	" " E→E+E
$0E1	$	Accept

- As * has precedence over + we have to perform multiplication operation first.
- For that it is necessary to push * on the top of the stack.
- The stack positions will be 

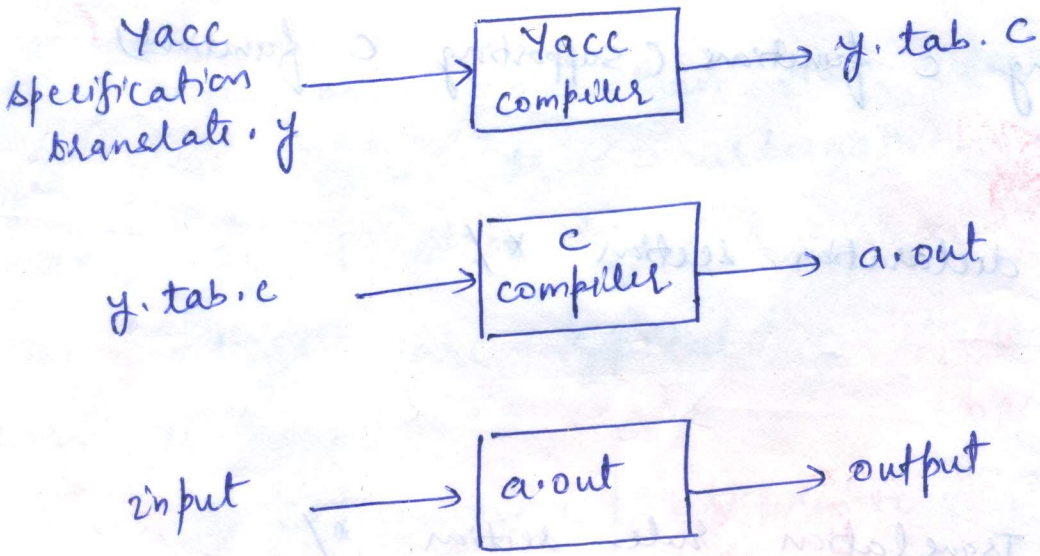
- By this we can perform E*E first and then E+E.
- Therefore parsing conflict can be resolved by assigning shift operation. Hence action [7, *] = S5.

Finally the ambiguous grammar without any conflict has following SLR parse table as -

State	Action						Goto
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5			Accept	
2	S3			S2			6
3		r4	r4		r4	r4	
4	S3			S2			7
5	S3			S2			8
6		S4	S5		S9		
7		r1	S5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Parser Generators (YACC)

- Certain automation tools for parser generation are available.
- YACC is one such automatic tool for generating the parser program.
- Basically YACC (Yet Another compiler compiler) is LALR parser generator.
- The YACC can report conflict or ambiguities in the form of error messages.
- LEX and YACC work together to analyse the program syntactically.



creating an input/output translator with Yacc

- A file, say translate.y, containing a Yacc specification of the translator is prepared.

- The UNIX system command yacc translate.y transforms the file translate.y into C program called y.tab.c using the LALR method.
- By compiling y.tab.c along with the ly library that contains the LR passing program using the command cc y.tab.c -ly we obtain the desired object program a.out that performs the translation specified by the original Yacc program.

YACC specifications

- The YACC specification file consists of three parts

(I) Declaration part (ordinary C declarations)
(II) Translation rule (context free Grammar)
(III) supporting C functions (supporting C functions)

```
%{
  /* declaration section */
%}
%%
/* Translation rule section */
%%
/* Required C functions */
```

Declaration part –

- In this section ordinary C declarations can be put.
- We can also declare grammar tokens in this section.
- The declaration of tokens should be with in %{ and %}.

(ii) The translation rule section –

- It consists of all the production rules of context free grammar with corresponding actions. For instance

rule 1	action 1
rule 2	action 2
⋮	⋮
rule n	action n

- If there are more than one alternatives to a single rule then those alternatives should be separated by | character.
- The actions are typical C statements. If CFG is

LHS → alternative 1 | alternative 2 — | alternative n

then

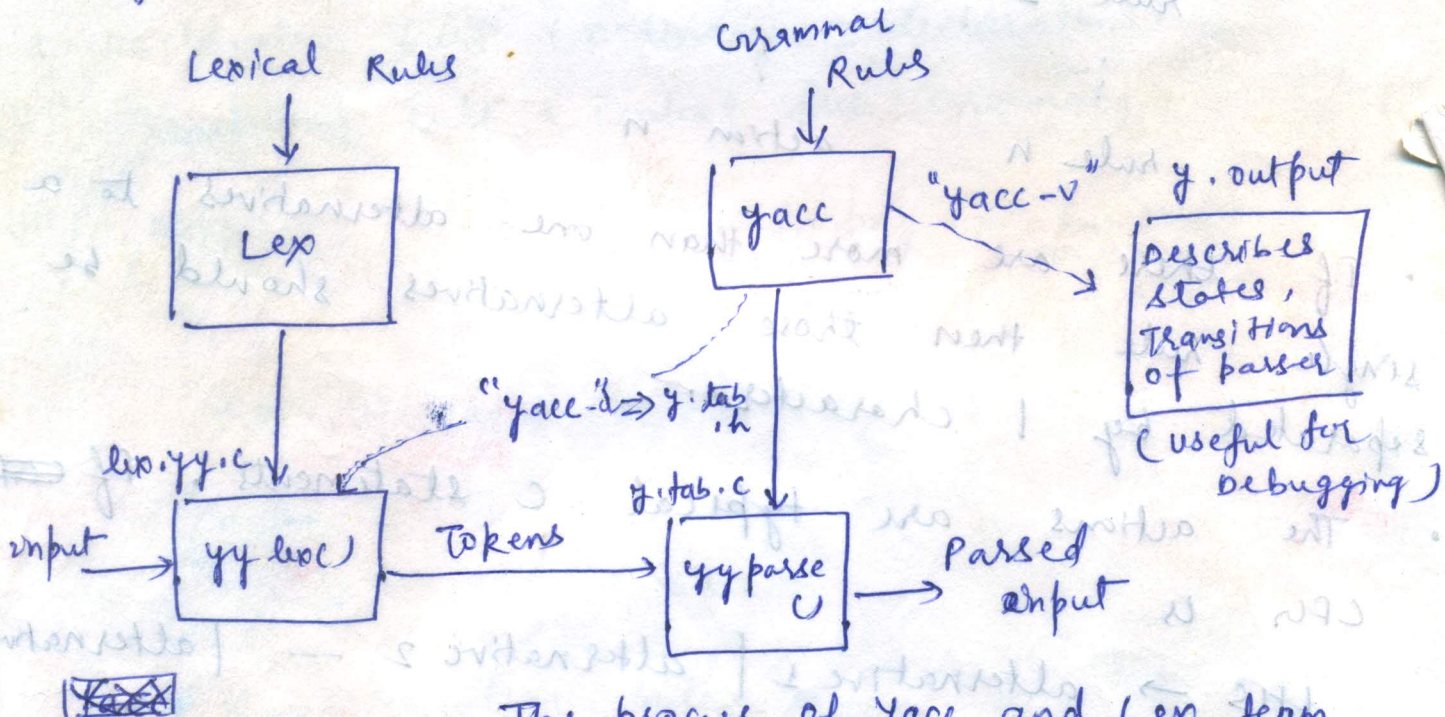
LHS : alternative 1 { action 1 }
| alternative 2 { action 2 }
⋮
alternative n { action n }
;

(iii) C functions section :-

- This section consists of one main function in which the routine yyparse() will be called.
- It also consists of required C functions.

LEX and Yacc as a team -

- Yacc assumes the existence of a function 'int yylex()' that implements the scanner (lex).
- Lex return value indicates the type of token found and other values communicated to the parser using yytext, yylval.



The process of Yacc and Lex team

Yacc determines integer representations for tokens.

(i) communicated to scanner in file y.tab.h
- use "yacc -d" to produce y.tab.h

(ii) Token encoding: - 'end of file' represented by 0
- a character literal : its ASCII value
- other tokens: assigned numbers ≥ 257