LALR Parsing Table construction

- **LALR** stands for **LookAhead LR.**

- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- ***yacc*** creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

Creating LALR Parsing Tables

Canonical LR(1) Parser ➔ LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrink process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L \bullet =R,\$$ ➔ $S \rightarrow L \bullet =R$ ← Core

$R \rightarrow L \bullet ,\$$ $R \rightarrow L \bullet$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

$I_1{:}L \rightarrow id \bullet ,=$ A new state: $I_{12}{:}\ L \rightarrow id \bullet ,=$

➔ $L \rightarrow id \bullet ,\$$

$I_2{:}L \rightarrow id \bullet ,\$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

  $C=\{I_0,...,I_n\} \Rightarrow C'=\{J_1,...,J_m\}$ where $m \leq n$
- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores

    $\Rightarrow$ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
  - So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.
- If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

$$A \to \alpha \bullet ,a \qquad \text{and} \qquad B \to \beta \bullet a\gamma ,b$$

- This means that a state of the canonical LR(1) parser must have:

$$A \to \alpha \bullet ,a \qquad \text{and} \qquad B \to \beta \bullet a\gamma ,c$$

But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)

Reduce/Reduce Conflict

- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

$$I_1 : A \to \alpha\bullet,a \qquad\qquad I_2: A \to \alpha\bullet,b$$
$$B \to \beta\bullet,b \qquad\qquad B \to \beta\bullet,c$$
$$\Downarrow$$
$$I_{12}: A \to \alpha\bullet,a/b \qquad \Rightarrow \text{reduce/reduce conflict}$$
$$B \to \beta\bullet,b/c$$

Canonical LALR(1) Collection – Example2

S' → S
1) S → L=R
2) S → R
3) L→ *R
4) L → id
5) R → L

I_0:S' → •S,$
S → •L=R,$
S → •R,$
L → •*R,$/=
L → •id,$/=
R → •L,$

$I_0 \xrightarrow{S}$ I_1:S' → S•,$

$I_0 \xrightarrow{*}$ I_{411}

$I_0 \xrightarrow{L}$ I_2:S → L•=R,$ → to I_6
R → L•,$

$I_0 \xrightarrow{R}$ I_3:S → R•,$

$I_0 \xrightarrow{id}$ I_{512}

I_{411}:L → *•R,$/=
R → •L,$/=
L→ •*R,$/=
L → •id,$/=

I_{411}: R → to I_{713}; L → to I_{810}; * → to I_{411}; id → to I_{512}

I_{512}:L → id•,$/=

I_6:S → L=•R,$
R → •L,$
L → •*R,$
L → •id,$

I_6: R → to I_9; L → to I_{810}; * → to I_{411}; id → to I_{512}

I_9:S → L=R•,$

I_{713}:L → *R•,$/=

I_{810}: R → L•,$/=

Same Cores
I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LALR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce conflict

$\Downarrow$

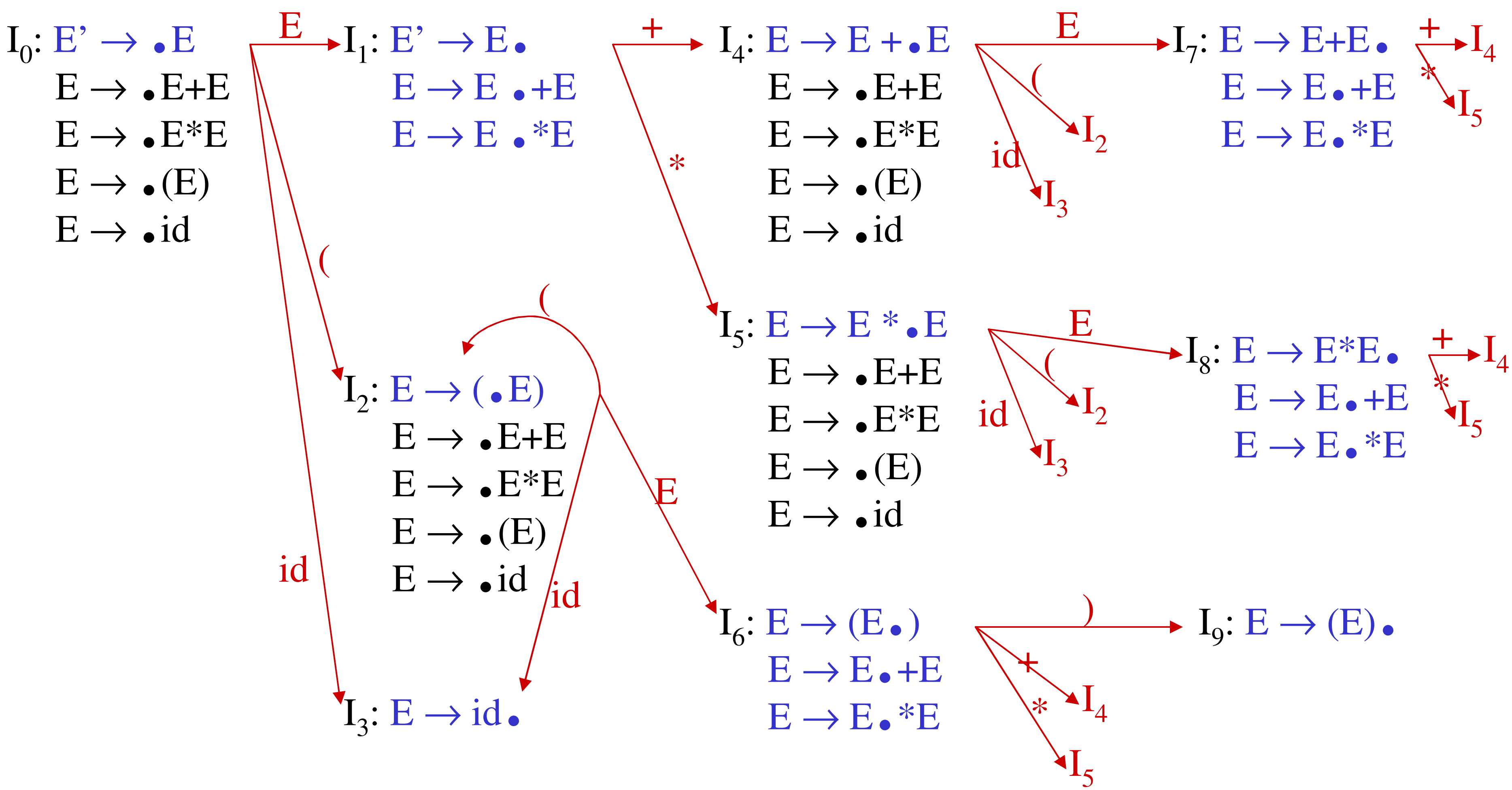
so, it is a LALR(1) grammar

Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.
- Can we create LR-parsing tables for ambiguous grammars ?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  - At the end, we will have again an unambiguous grammar.
- Why we want to use an ambiguous grammar?
  - Some of the ambiguous grammars are **much natural**, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.
- Ex.

$$E \rightarrow E+E \mid E^*E \mid (E) \mid id \quad \Rightarrow \quad \begin{array}{l} E \rightarrow E+T \mid T \\ T \rightarrow T^*F \mid F \\ F \rightarrow (E) \mid id \end{array}$$

Sets of LR(0) Items for Ambiguous Grammar



SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $, +, *,) }

State I_7 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{E} I_7$$

when current token is +

shift ➔ + is right-associative

reduce ➔ + is left-associative

when current token is *

shift ➔ * has higher precedence than +

reduce ➔ + has higher precedence than *

SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $, +, *,) }

State I_8 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{*} I_5 \xrightarrow{E} I_7$$

when current token is *
- shift ➔ * is right-associative
- reduce ➔ * is left-associative

when current token is +
- shift ➔ + has higher precedence than *
- reduce ➔ * has higher precedence than +

SLR-Parsing Tables for Ambiguous Grammar

	Action							Goto
	id	**+**	*****	**(**	**)**	**$**		**E**
0	s3			s2				1
1		s4	s5			acc		
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		

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state s).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow A.
  - The symbol a is simply in FOLLOW(A), but this may not work for all situations.
- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing.
- This nonterminal A is normally is a basic programming block (there can be more than one choice for A).
  - stmt, expr, block, ...

Phrase-Level Error Recovery in LR Parsing

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - missing operand
  - unbalanced right parenthesis