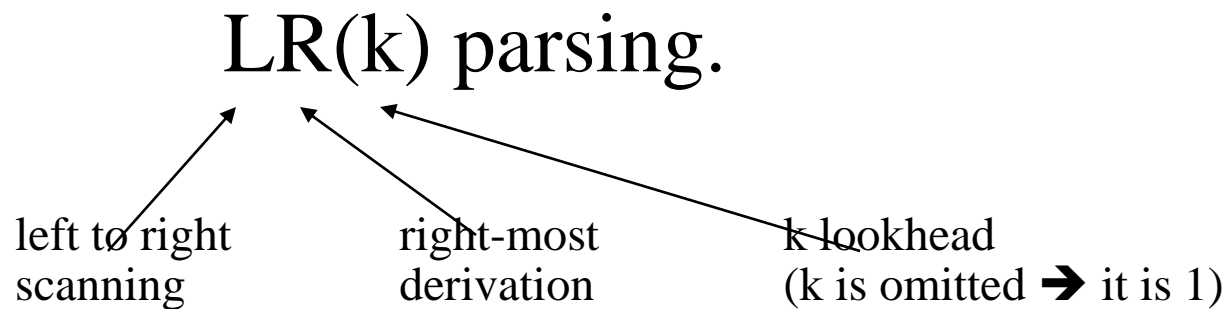


# LR Parsing

Dewan Tanvir Ahmed  
Assistant Professor, CSE  
BUET

# LR Parsers

- The most powerful shift-reduce parsing (yet efficient) is:

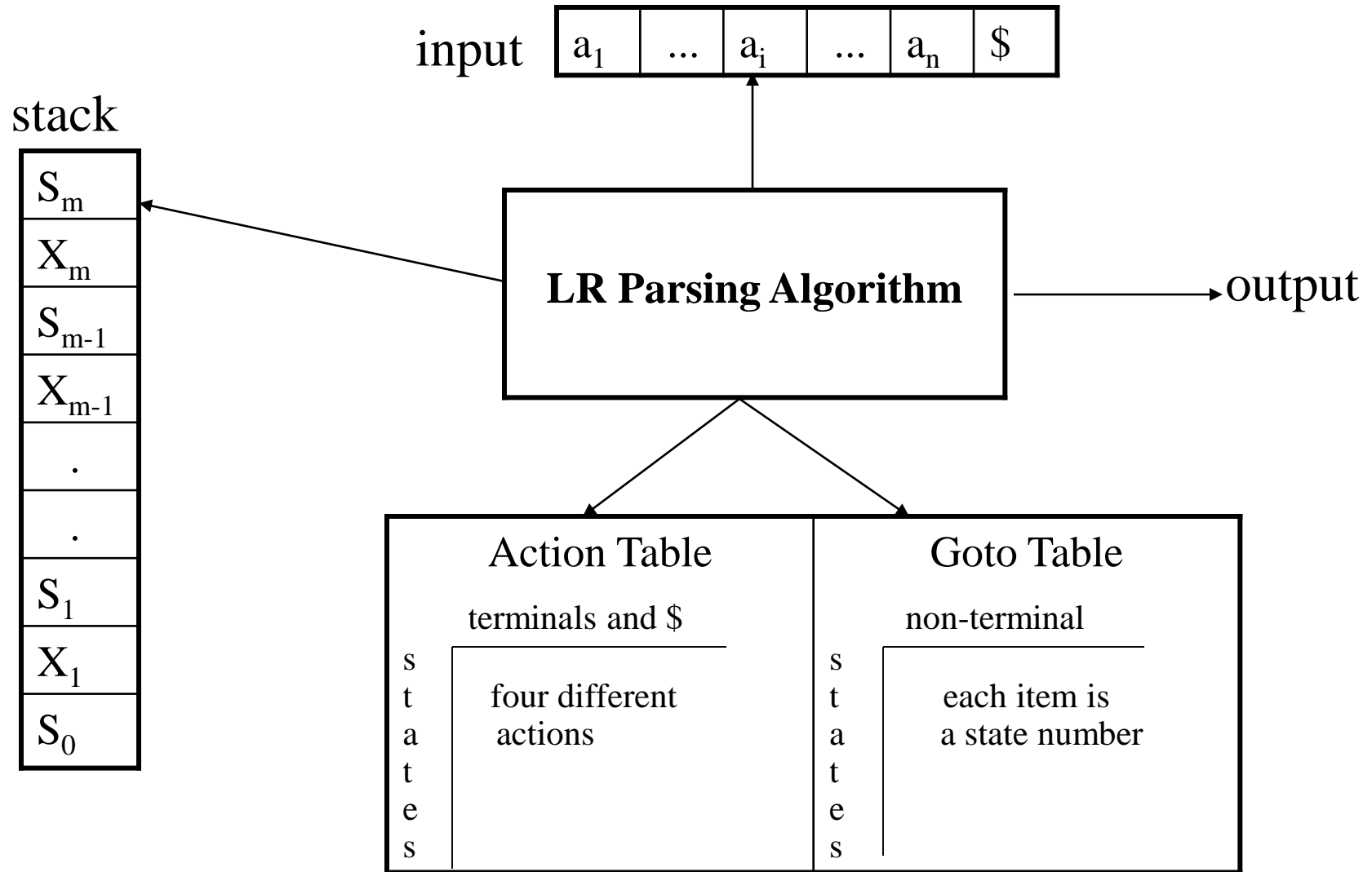


- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.  
**LL(1)-Grammars  $\subset$  LR(1)-Grammars**
  - An LR-parser can detect a syntactic error as soon as it is possible to do so in a left-to-right scan of the input.

# LR Parsers

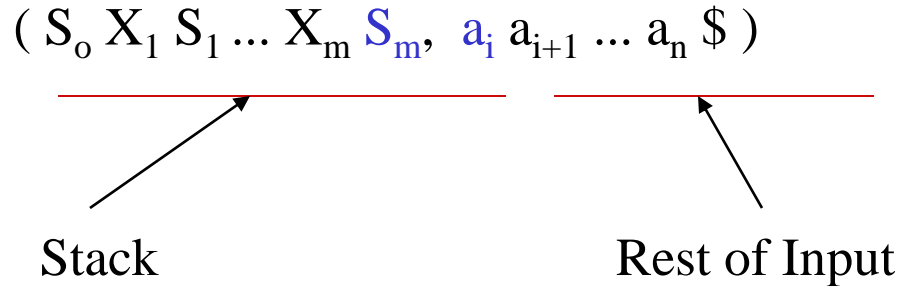
- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-head LR parser)
  - SLR, LR and LALR work same (they use the same algorithm), only their parsing tables differ.

# LR Parsing Algorithm



# A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:



- $S_m$  and  $a_i$  decides the parser action by consulting the parsing action table. (*Initially, Stack contains just  $S_o$*  )
- A configuration of a LR parsing represents the right sentential form:

$$X_1 \dots X_m a_i a_{i+1} \dots a_n \$$$

# Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

$(S_o X_1 S_1 \dots X_m S_m, a_i a_{i+1} \dots a_n \$) \rightarrow (S_o X_1 S_1 \dots X_m S_m \mathbf{a_i s}, a_{i+1} \dots a_n \$)$

2. **reduce  $A \rightarrow \beta$**  (or **rn** where n is a production number)

- pop  $2|\beta|$  ( $=r$ ) items from the stack;
- then push **A** and **s** where **s=goto[s<sub>m-r</sub>,A]**

$(S_o X_1 S_1 \dots X_m S_m, a_i a_{i+1} \dots a_n \$) \rightarrow (S_o X_1 S_1 \dots X_{m-r} \mathbf{S_{m-r} A s}, a_i \dots a_n \$)$

- Output is the reducing production reduce  $A \rightarrow \beta$

3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

# Reduce Action

- pop  $2|\beta|$  ( $=r$ ) items from the stack; let us assume that  $\beta = Y_1 Y_2 \dots Y_r$
- then push  $A$  and  $s$  where  $s = \text{goto}[s_{m-r}, A]$

$$\begin{aligned}
 & ( S_o X_1 S_1 \dots X_{m-r} \textcolor{blue}{S}_{m-r} \textcolor{red}{Y}_1 \textcolor{red}{S}_{m-r+1} \dots \textcolor{red}{Y}_r \textcolor{red}{S}_m, a_i a_{i+1} \dots a_n \$ ) \\
 & \quad \rightarrow ( S_o X_1 S_1 \dots X_{m-r} \textcolor{blue}{S}_{m-r} \textcolor{red}{A} s, a_i \dots a_n \$ )
 \end{aligned}$$

- In fact,  $Y_1 Y_2 \dots Y_r$  is a handle.

$$X_1 \dots X_{m-r} \textcolor{red}{A} a_i \dots a_n \$ \Rightarrow X_1 \dots X_m \textcolor{red}{Y}_1 \dots \textcolor{red}{Y}_r a_i a_{i+1} \dots a_n \$$$

# (SLR) Parsing Tables for Expression Grammar

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

Action Table

Goto Table

state	id	+	*	(	)	\$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
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				



# Actions of A (S)LR-Parser -- Example

<u>stack</u>	<u>input</u>	<u>action</u>	<u>output</u>
0	id*id+id\$	shift 5	
0id5	*id+id\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0F3	*id+id\$	reduce by $T \rightarrow F$	$T \rightarrow F$
0T2	*id+id\$	shift 7	
0T2*7	id+id\$	shift 5	
0T2*7id5	+id\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0T2*7F10	+id\$	reduce by $T \rightarrow T * F$	$T \rightarrow T * F$
0T2	+id\$	reduce by $E \rightarrow T$	$E \rightarrow T$
0E1	+id\$	shift 6	
0E1+6	id\$	shift 5	
0E1+6id5	\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0E1+6F3	\$	reduce by $T \rightarrow F$	$T \rightarrow F$
0E1+6T9	\$	reduce by $E \rightarrow E + T$	$E \rightarrow E + T$
0E1	\$	accept	

# Actions of A (S)LR-Parser -- Example

<u>stack</u>	<u>input</u>	<u>action</u>	<u>output</u>
0	id*id+id\$	shift 5	
0id5	*id+id\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0F3	*id+id\$	reduce by $T \rightarrow F$	$T \rightarrow F$
0T2	*id+id\$	shift 7	
0T2*7	id+id\$	shift 5	
0T2*7id5	+id\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0T2*7F10	+id\$	reduce by $T \rightarrow T * F$	$T \rightarrow T * F$
0T2	+id\$	reduce by $E \rightarrow T$	$E \rightarrow T$
0E1	+id\$	shift 6	
0E1+6	id\$	shift 5	
0E1+6id5	\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0E1+6F3	\$	reduce by $T \rightarrow F$	$T \rightarrow F$
0E1+6T9	\$	reduce by $E \rightarrow E + T$	$E \rightarrow E + T$
0E1	\$	accept	

- |                          |                        |
|--------------------------|------------------------|
| 1) $E \rightarrow E + T$ | 4) $T \rightarrow F$   |
| 2) $E \rightarrow T$     | 5) $F \rightarrow (E)$ |
| 3) $T \rightarrow T * F$ | 6) $F \rightarrow id$  |

Action Table

Goto Table

state	id	+	*	(	)	\$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
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				

# Constructing SLR Parsing Tables – LR(0) Item

- An **LR(0) item** of a grammar G is a production of G a dot at the some position of the right side.
- Ex:     $A \rightarrow aBb$                   Possible LR(0) Items:                   $A \rightarrow \bullet aBb$   
                                       (four different possibility)                   $A \rightarrow a\bullet Bb$   
    $A \rightarrow aB\bullet b$   
    $A \rightarrow aBb\bullet$
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- Augmented Grammar:*  
 $G'$  is G with a new production rule  $S' \rightarrow S$  where  $S'$  is the new starting symbol.

# The Closure Operation

- If  $I$  is a set of LR(0) items for a grammar  $G$ , then  $\text{closure}(I)$  is the set of LR(0) items constructed from  $I$  by the two rules:
  1. Initially, every LR(0) item in  $I$  is added to  $\text{closure}(I)$ .
  2. If  $A \rightarrow \alpha \bullet B \beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production rule of  $G$ ; then  $B \rightarrow \bullet \gamma$  will be in the  $\text{closure}(I)$ . We will apply this rule until no more new LR(0) items can be added to  $\text{closure}(I)$ .

What is happening by  $B \rightarrow \bullet \gamma$  ?

# The Closure Operation -- Example

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{id}$

$\text{closure}(\{E' \rightarrow \bullet E\}) =$

$\{ E' \rightarrow \bullet E \} \longleftarrow \text{kernel items}$

$E \rightarrow \bullet E+T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T*F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet \text{id} \}$

# Computation of Closure

function closure ( I )

begin

$J := I;$

    repeat

        for each item  $A \rightarrow \alpha.B\beta$  in J and each production

$B \rightarrow \gamma$  of G such that  $B \rightarrow \cdot \gamma$  is not in J do

                add  $B \rightarrow \cdot \gamma$  to J

    until no more items can be added to J

end

# Goto Operation

- If  $I$  is a set of LR(0) items and  $X$  is a grammar symbol (terminal or non-terminal), then  $\text{goto}(I, X)$  is defined as follows:
  - If  $A \rightarrow \alpha \bullet X \beta$  in  $I$  then every item in  $\text{closure}(\{A \rightarrow \alpha X \bullet \beta\})$  will be in  $\text{goto}(I, X)$ .
  - If  $I$  is the set of items that are valid for some viable prefix  $\gamma$ , then  $\text{goto}(I, X)$  is the set of items that are valid for the viable prefix  $\gamma X$ .

## Example:

$I = \{ \begin{array}{l} E' \rightarrow \bullet E, \quad E \rightarrow \bullet E + T, \quad E \rightarrow \bullet T, \\ T \rightarrow \bullet T * F, \quad T \rightarrow \bullet F, \\ F \rightarrow \bullet (E), \quad F \rightarrow \bullet \text{id} \end{array} \}$

$\text{goto}(I, E) = \{ E' \rightarrow E \bullet, E \rightarrow E \bullet + T \}$

$\text{goto}(I, T) = \{ E \rightarrow T \bullet, T \rightarrow T \bullet * F \}$

$\text{goto}(I, F) = \{ T \rightarrow F \bullet \}$

$\text{goto}(I, () = \{ F \rightarrow ( \bullet E), E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, \\ F \rightarrow \bullet (E), F \rightarrow \bullet \text{id} \}$

$\text{goto}(I, \text{id}) = \{ F \rightarrow \text{id} \bullet \}$

# Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar  $G$ , we will create the canonical LR(0) collection of the grammar  $G'$ .
- **Algorithm:**
  - $C$  is  $\{ \text{closure}(\{S' \rightarrow \bullet S\}) \}$
  - repeat** the followings until no more set of LR(0) items can be added to  $C$ .
    - for each**  $I$  in  $C$  and each grammar symbol  $X$ 
      - if**  $\text{goto}(I, X)$  is not empty and not in  $C$ 
        - add  $\text{goto}(I, X)$  to  $C$
- $\text{goto}$  function is a DFA on the sets in  $C$ .



# The Canonical LR(0) Collection -- Example

$I_0: E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_1: E' \rightarrow E.$

$I_6: E \rightarrow E+.T$

$E \rightarrow E.+T$

$I_2: E \rightarrow T.$

$T \rightarrow T.*F$

$I_3: T \rightarrow F.$

$I_4: F \rightarrow (.E)$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_5: F \rightarrow id.$

$I_9: E \rightarrow E+T.$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_7: T \rightarrow T*.F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_8: F \rightarrow (E.)$

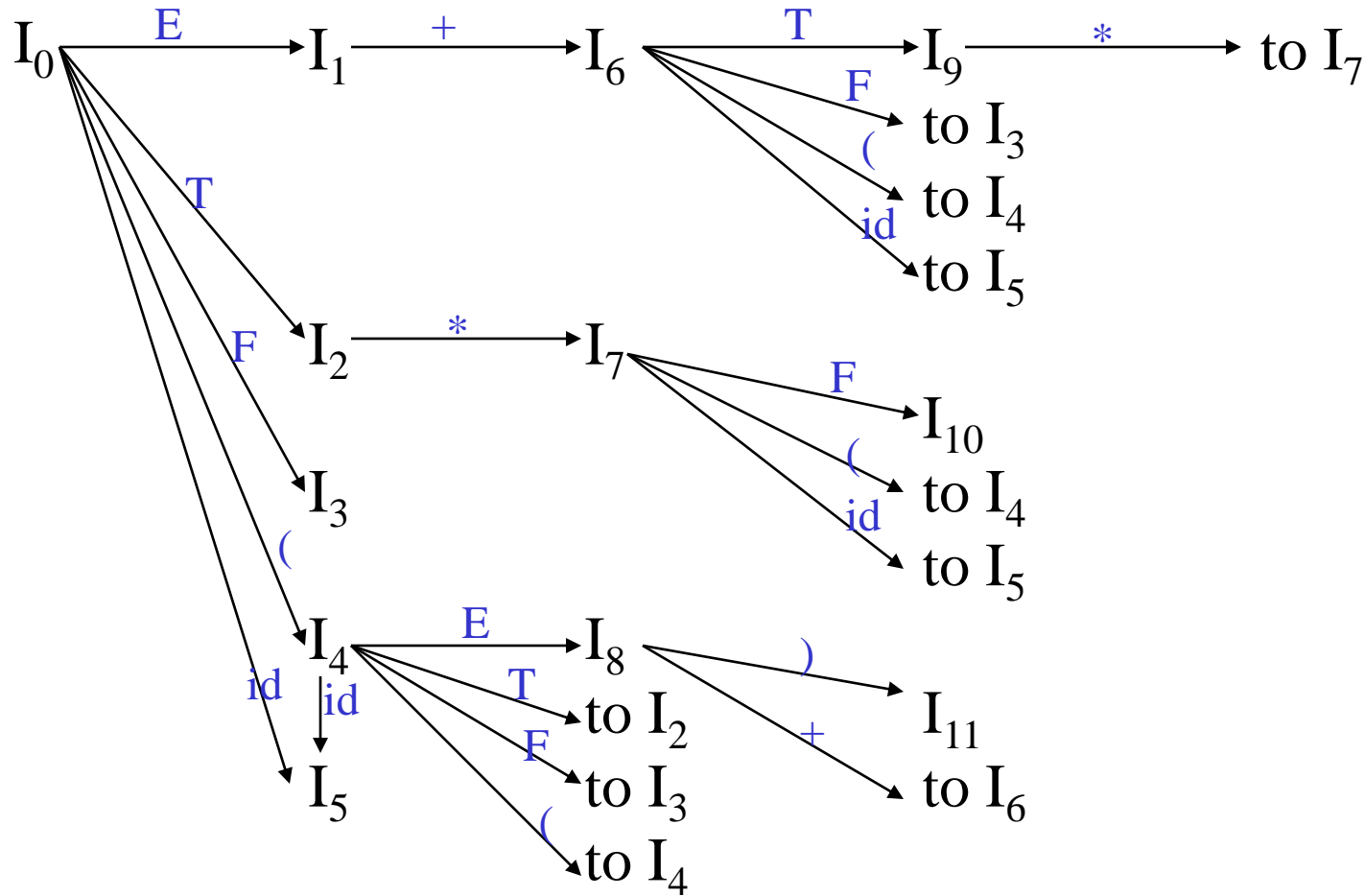
$E \rightarrow E.+T$

$T \rightarrow T.*F$

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

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

# Transition Diagram (DFA) of Goto Function



# Constructing SLR Parsing Table

(of an augmented grammar  $G'$ )

1. Construct the canonical collection of sets of LR(0) items for  $G'$ .  $C \leftarrow \{I_0, \dots, I_n\}$
2. Create the parsing action table as follows
  - If  $a$  is a terminal,  $A \rightarrow \alpha.a\beta$  in  $I_i$  and  $\text{goto}(I_i, a) = I_j$  then  $\text{action}[i, a]$  is *shift j*.
  - If  $A \rightarrow \alpha.$  is in  $I_i$ , then  $\text{action}[i, a]$  is *reduce*  $A \rightarrow \alpha$  for all  $a$  in  $\text{FOLLOW}(A)$  where  $A \neq S'$ .
  - If  $S' \rightarrow S.$  is in  $I_i$ , then  $\text{action}[i, \$]$  is *accept*.
  - If any conflicting actions generated by these rules, the grammar is not SLR(1).
3. Create the parsing goto table
  - for all non-terminals  $A$ , if  $\text{goto}(I_i, A) = I_j$  then  $\text{goto}[i, A] = j$
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains  $S' \rightarrow .S$

# Parsing Tables of Expression Grammar

Action Table

Goto Table

state	id	+	*	(	)	\$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
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				

# SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar  $G$  is called as the SLR(1) parser for  $G$ .
- If a grammar  $G$  has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

# shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.
- If a state does not know whether it will make a reduction operation using the production rule  $i$  or  $j$  for a terminal, we say that there is a **reduce/reduce conflict**.
- If the SLR parsing table of a grammar  $G$  has a conflict, we say that that grammar is not SLR grammar.

# Conflict Example

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow \text{id}$

$R \rightarrow L$

$I_0: S' \rightarrow .S$

$S \rightarrow .L=R$

$S \rightarrow .R$

$L \rightarrow .*R$

$L \rightarrow .\text{id}$

$R \rightarrow .L$

$I_1: S' \rightarrow S.$

$I_2: S \rightarrow L.=R$   
 $R \rightarrow L.$

$I_3: S \rightarrow R.$

$I_4: L \rightarrow *.R$

$R \rightarrow .L$

$L \rightarrow .*R$

$L \rightarrow .\text{id}$

$I_5: L \rightarrow \text{id}.$

$I_6: S \rightarrow L=.R$

$R \rightarrow .L$

$L \rightarrow .*R$

$L \rightarrow .\text{id}$

$I_7: L \rightarrow *.R.$

$I_8: R \rightarrow L.$

$I_9: S \rightarrow L=R.$

**Problem**

$\text{FOLLOW}(R) = \{=, \$\}$

$=$   $\rightarrow$  shift 6

$\rightarrow$  reduce by  $R \rightarrow L$

shift/reduce conflict

**Action[2,=] = shift 6**

**Action[2,=] = reduce by  $R \rightarrow L$**

[  $S \Rightarrow L=R \Rightarrow *R=R$  ] so follow(R) contains, =

# Conflict Example2

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

$I_0: S' \rightarrow .S$

$S \rightarrow .AaAb$

$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

**Problem**

$\text{FOLLOW}(A) = \{a, b\}$

$\text{FOLLOW}(B) = \{a, b\}$

a  $\rightarrow$  reduce by  $A \rightarrow \varepsilon$

$\searrow$  reduce by  $B \rightarrow \varepsilon$

reduce/reduce conflict

b  $\rightarrow$  reduce by  $A \rightarrow \varepsilon$

$\searrow$  reduce by  $B \rightarrow \varepsilon$

reduce/reduce conflict



# Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state  $i$  makes a reduction by  $A \rightarrow \alpha$  when the current token is **a**:
  - if the  $A \rightarrow \alpha.$  in the  $I_i$  and **a** is FOLLOW( $A$ )
- In some situations,  $\beta A$  cannot be followed by the terminal **a** in a right-sentential form when  $\beta\alpha$  and the state  $i$  are on the top stack. This means that making reduction in this case is not correct.
- Back to Slide no 22.

# LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:  
$$A \rightarrow \alpha \cdot \beta, a$$
where **a** is the look-head of the LR(1) item  
(**a** is a terminal or end-marker.)
- Such an object is called LR(1) item.
  - 1 refers to the length of the second component
  - The lookahead has no effect in an item of the form  $[A \rightarrow \alpha \cdot \beta, a]$ , where  $\beta$  is not  $\epsilon$ .
  - But an item of the form  $[A \rightarrow \alpha \cdot, a]$  calls for a reduction by  $A \rightarrow \alpha$  only if the next input symbol is  $a$ .
  - The set of such  $a$ 's will be a subset of FOLLOW(A), but it could be a proper subset.

## LR(1) Item (cont.)

- When  $\beta$  ( in the LR(1) item  $A \rightarrow \alpha.\beta,a$  ) is not empty, the look-head does not have any affect.
- When  $\beta$  is empty ( $A \rightarrow \alpha.,a$  ), we do the reduction by  $A \rightarrow \alpha$  only if the next input symbol is **a** (**not for any terminal in FOLLOW(A)**).
- A state will contain  $A \rightarrow \alpha.,a_1$  where  $\{a_1, \dots, a_n\} \subseteq \text{FOLLOW}(A)$

...

$A \rightarrow \alpha.,a_n$

# Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

**closure(I)** is: ( where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if  $A \rightarrow \alpha \cdot B \beta, a$  in closure(I) and  $B \rightarrow \gamma$  is a production rule of G; then  $B \rightarrow \cdot \gamma, b$  will be in the closure(I) for each terminal b in FIRST( $\beta a$ ) .

## goto operation

- If  $I$  is a set of LR(1) items and  $X$  is a grammar symbol (terminal or non-terminal), then  $\text{goto}(I, X)$  is defined as follows:
  - If  $A \rightarrow \alpha.X\beta, a$  in  $I$   
then every item in  $\text{closure}(\{A \rightarrow \alpha X.\beta, a\})$  will be in  $\text{goto}(I, X)$ .

# Construction of The Canonical LR(1) Collection

- *Algorithm:*

$C$  is { closure(  $\{S' \rightarrow .S, \$\}$  ) }

**repeat** the followings until no more set of LR(1) items can be added to  $C$ .

**for each**  $I$  in  $C$  and each grammar symbol  $X$

**if** goto( $I, X$ ) is not empty and not in  $C$

            add goto( $I, X$ ) to  $C$

- goto function is a DFA on the sets in  $C$ .

# A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \rightarrow \alpha \cdot \beta, a_1$$

...

$$A \rightarrow \alpha \cdot \beta, a_n$$

can be written as

$$A \rightarrow \alpha \cdot \beta, a_1/a_2/.../a_n$$

# Canonical LR(1) Collection -- Example

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

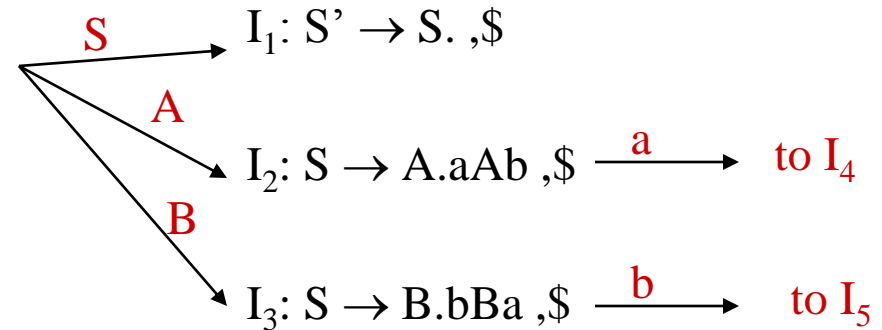
$I_0: S' \rightarrow .S, \$$

$S \rightarrow .AaAb, \$$

$S \rightarrow .BbBa, \$$

$A \rightarrow ., a$

$B \rightarrow ., b$



$I_4: S \rightarrow Aa.Ab, \$ \xrightarrow{A} I_6: S \rightarrow AaA.b, \$ \xrightarrow{a} I_8: S \rightarrow AaAb., \$$   
 $A \rightarrow ., b$

$I_5: S \rightarrow Bb.Ba, \$ \xrightarrow{B} I_7: S \rightarrow BbB.a, \$ \xrightarrow{b} I_9: S \rightarrow BbBa., \$$   
 $B \rightarrow ., a$

**closure(I)** is: ( where I is a set of LR(1) items)

every LR(1) item in I is in closure(I)

if  $A \rightarrow \alpha.B\beta, a$  in closure(I) and  $B \rightarrow \gamma$  is a production rule of G; then  $B \rightarrow \gamma, b$  will be in the closure(I) for each terminal b in FIRST( $\beta a$ )



# An Example

- 1.  $S' \rightarrow S$
- 2.  $S \rightarrow C C$
- 3.  $C \rightarrow c C$
- 4.  $C \rightarrow d$

## An Example

1.  $S' \rightarrow S$
2.  $S \rightarrow C C$
3.  $C \rightarrow c C$
4.  $C \rightarrow d$

$I_0$ :  $\text{closure}(\{(S' \rightarrow \bullet S, \$)\}) =$   
 $(S' \rightarrow \bullet S, \$)$   
 $(S \rightarrow \bullet C C, \$)$   
 $(C \rightarrow \bullet c C, c/d)$   
 $(C \rightarrow \bullet d, c/d)$

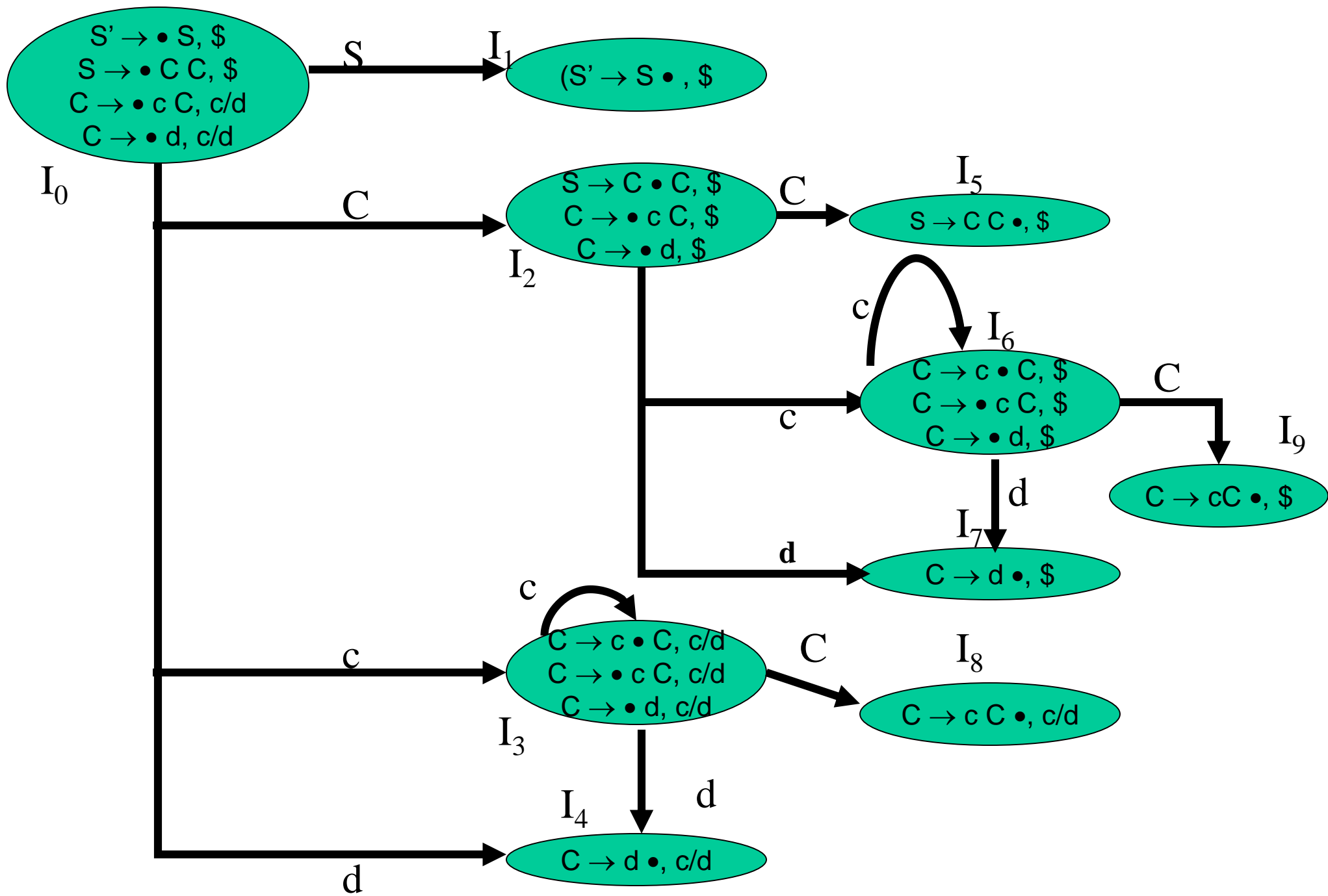
$I_1$ :  $\text{goto}(I_0, S) = (S' \rightarrow S \bullet, \$)$

$I_2$ :  $\text{goto}(I_0, C) =$   
 $(S \rightarrow C \bullet C, \$)$   
 $(C \rightarrow \bullet c C, \$)$   
 $(C \rightarrow \bullet d, \$)$

$I_3$ :  $\text{goto}(I_1, c) =$   
 $(C \rightarrow c \bullet C, c/d)$   
 $(C \rightarrow \bullet c C, c/d)$   
 $(C \rightarrow \bullet d, c/d)$

$I_4$ :  $\text{goto}(I_1, d) =$   
 $(C \rightarrow d \bullet, c/d)$

$I_5$ :  $\text{goto}(I_3, C) =$   
 $(S \rightarrow C C \bullet, \$)$



# An Example

$l_6$ : goto( $l_3$ , c) =  
( $C \rightarrow c \bullet C$ , \$)  
( $C \rightarrow \bullet c C$ , \$)  
( $C \rightarrow \bullet d$ , \$)

$l_7$ : goto( $l_3$ , d) =  
( $C \rightarrow d \bullet$ , \$)

$l_8$ : goto( $l_4$ , C) =  
( $C \rightarrow c C \bullet$ , c/d)

: goto( $l_4$ , c) =  $l_4$

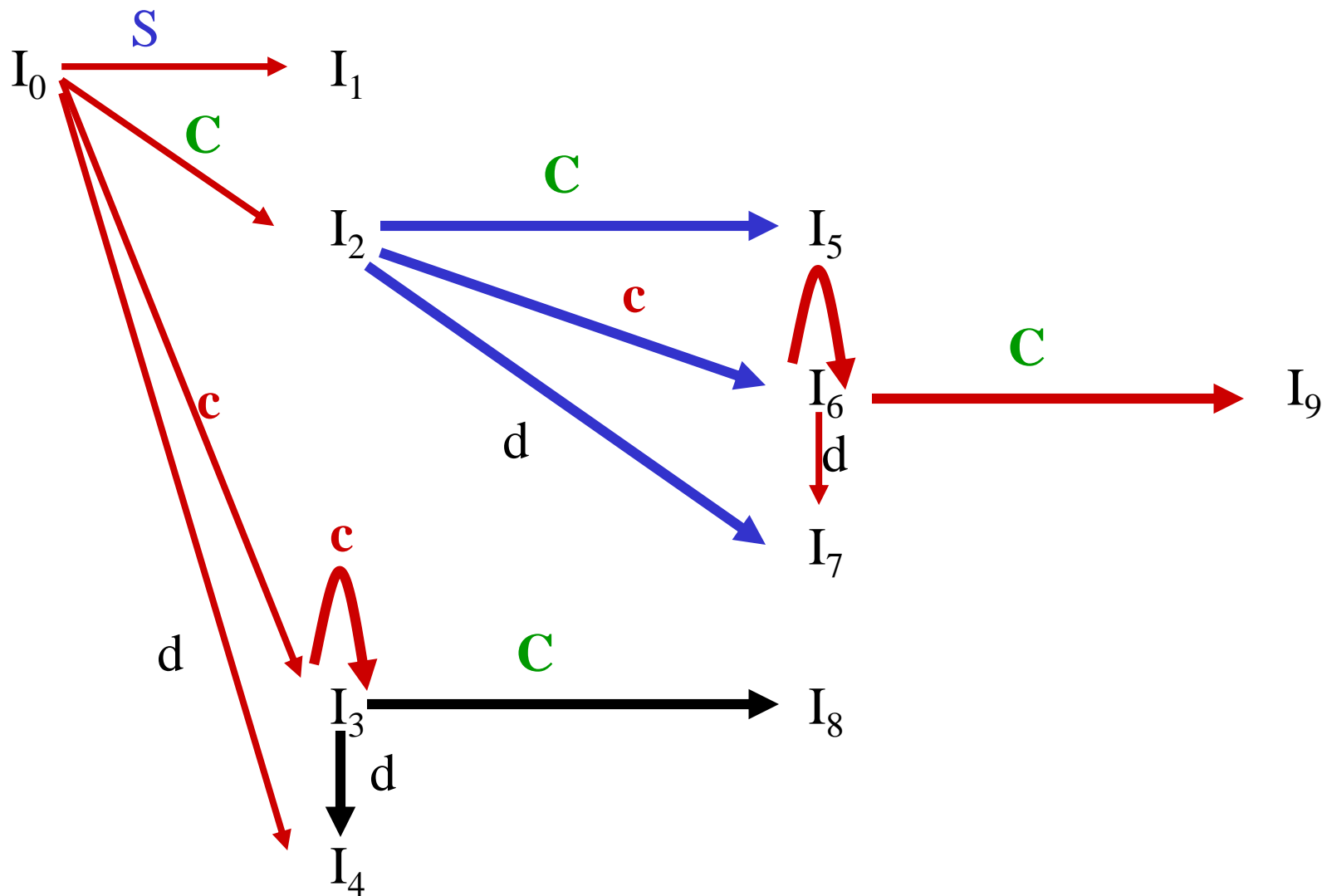
: goto( $l_4$ , d) =  $l_5$

$l_9$ : goto( $l_7$ , c) =  
( $C \rightarrow c C \bullet$ , \$)

: goto( $l_7$ , c) =  $l_7$

: goto( $l_7$ , d) =  $l_8$

# An Example



# An Example

	c	d	\$	S	C
0	s3	s4		g1	g2
1			a		
2	s6	s7			g5
3	s3	s4			g8
4	r3	r3			
5			r1		
6	s6	s7			g9
7			r3		
8	r2	r2			
9			r2		

# The Core of LR(1) Items

- The **core** of a set of LR(1) Items is the set of their first components (i.e., LR(0) items)
- The core of the set of LR(1) items

$$\{ (C \rightarrow c \bullet C, c/d), \\ (C \rightarrow \bullet c C, c/d), \\ (C \rightarrow \bullet d, c/d) \}$$

$$\text{is } \{ C \rightarrow c \bullet C, \\ C \rightarrow \bullet c C, \\ C \rightarrow \bullet d \}$$

# Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for  $G'$ .  
 $C \leftarrow \{I_0, \dots, I_n\}$
2. Create the parsing action table as follows
  - If  $a$  is a terminal,  $A \rightarrow \alpha \cdot a \beta$ ,  $b$  in  $I_i$  and  $\text{goto}(I_i, a) = I_j$  then  $\text{action}[i, a]$  is *shift j*.
  - If  $A \rightarrow \alpha \cdot$ ,  $a$  is in  $I_i$ , then  $\text{action}[i, a]$  is *reduce  $A \rightarrow \alpha$*  where  $A \neq S'$ .
  - If  $S' \rightarrow S \cdot, \$$  is in  $I_i$ , then  $\text{action}[i, \$]$  is *accept*.
  - If any conflicting actions generated by these rules, the grammar is not LR(1).
3. Create the parsing goto table
  - for all non-terminals  $A$ , if  $\text{goto}(I_i, A) = I_j$  then  $\text{goto}[i, A] = j$
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains  $S' \rightarrow \cdot S, \$$



# LALR Parsing Tables

1. **LALR** stands for **Lookahead LR**.
2. LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
3. The number of states in SLR and LALR parsing tables for a grammar  $G$  are equal.
4. But LALR parsers recognize more grammars than SLR parsers.
5. *yacc* creates a LALR parser for the given grammar.
6. 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:  $S \rightarrow L \bullet = R, \$$   $\rightarrow$   $S \rightarrow L \bullet = R$   $\leftarrow$  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.

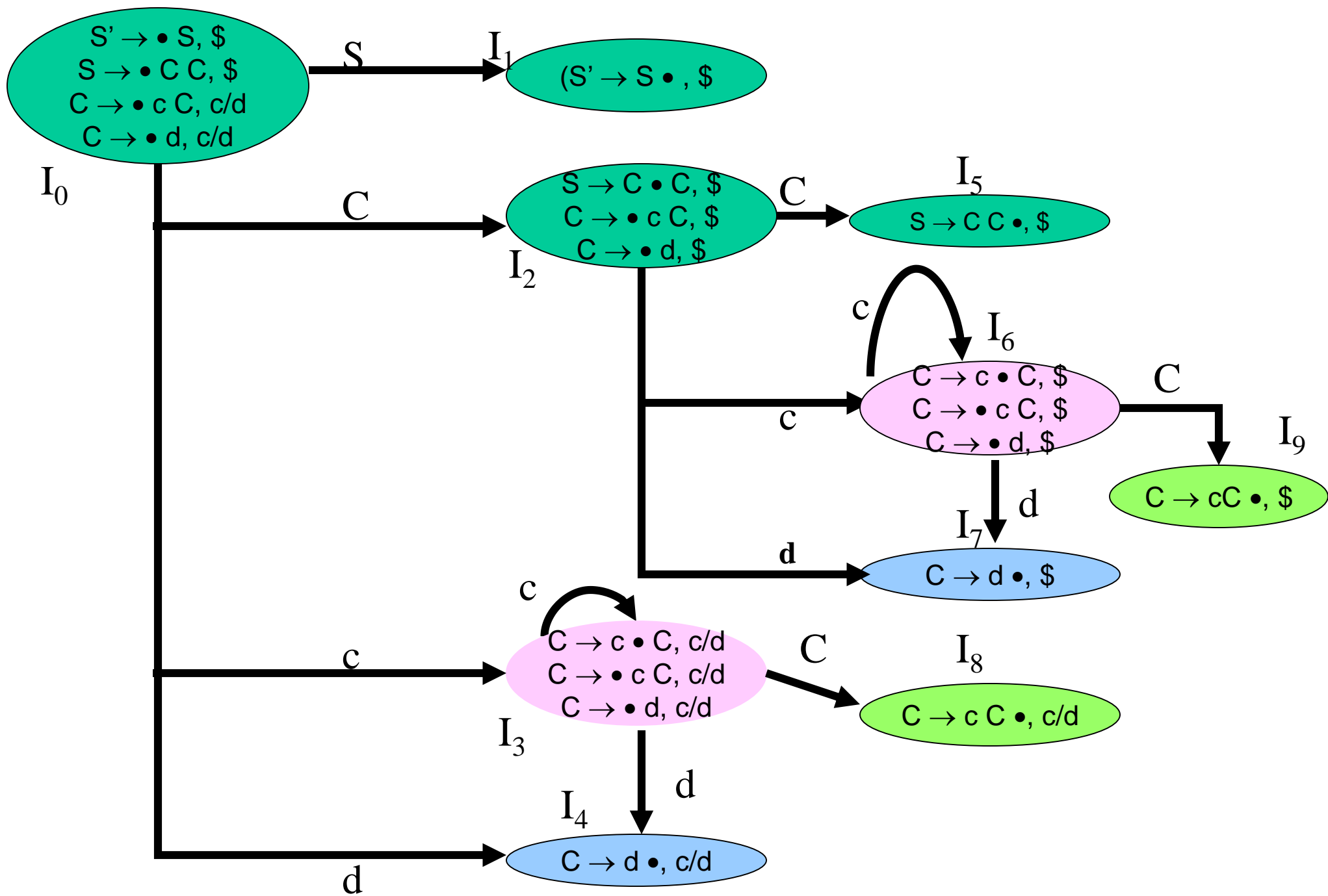
$I_1: L \rightarrow id \bullet, =$       A new state:       $I_{12}: L \rightarrow id \bullet, =$   
 $\rightarrow$        $L \rightarrow id \bullet, \$$

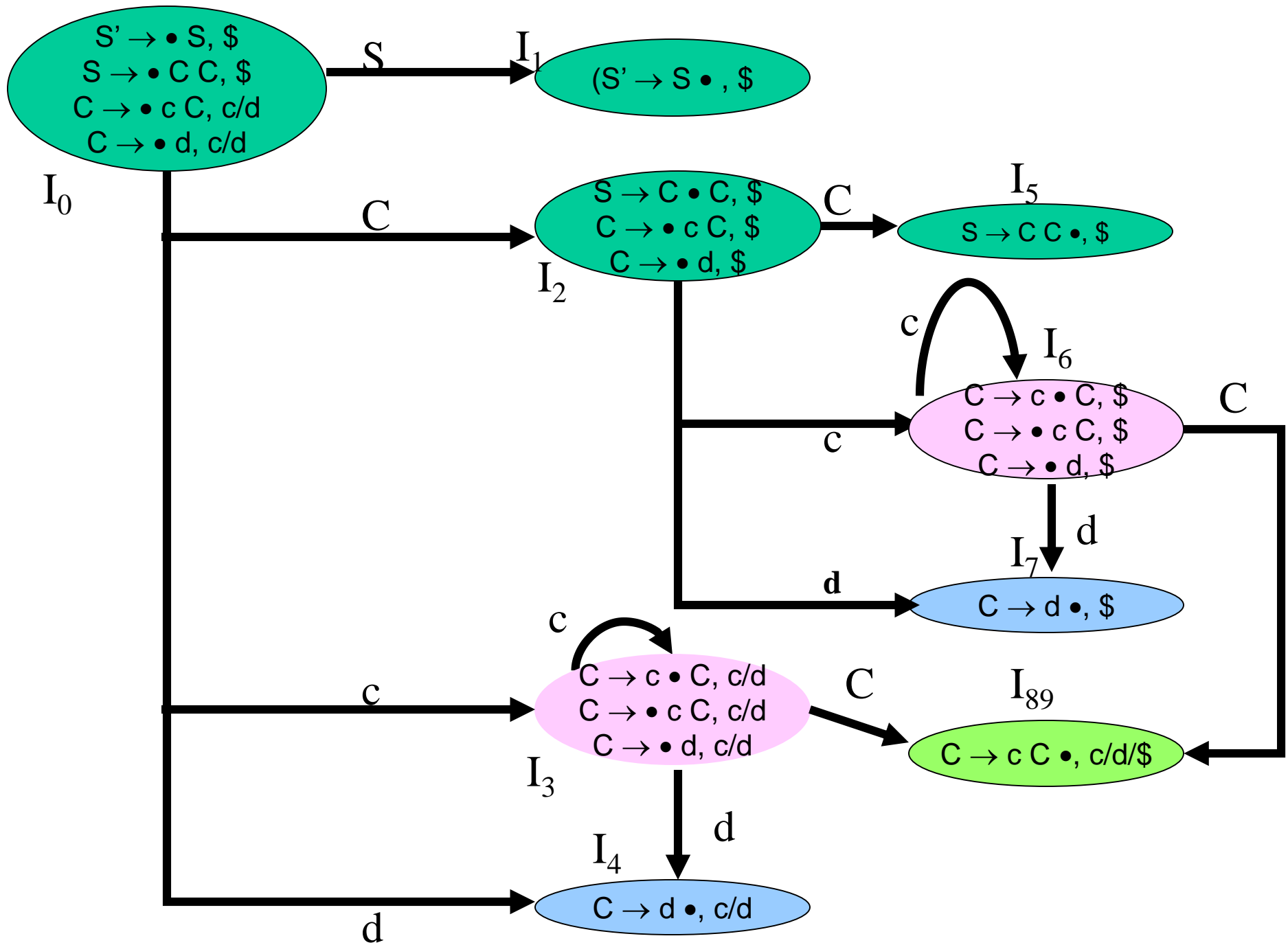
$I_2: L \rightarrow id \bullet, \$$       have same core, merge them

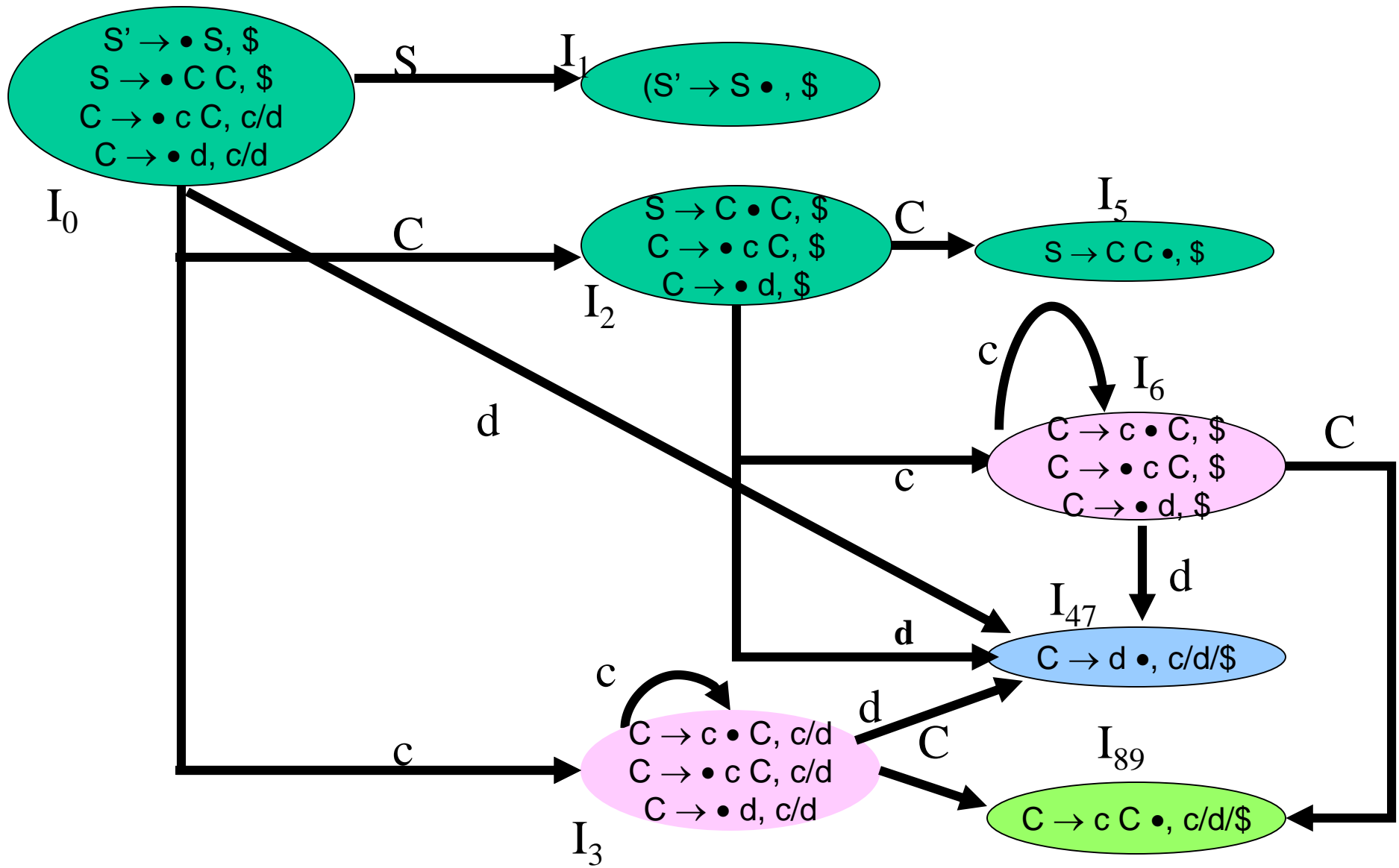
- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

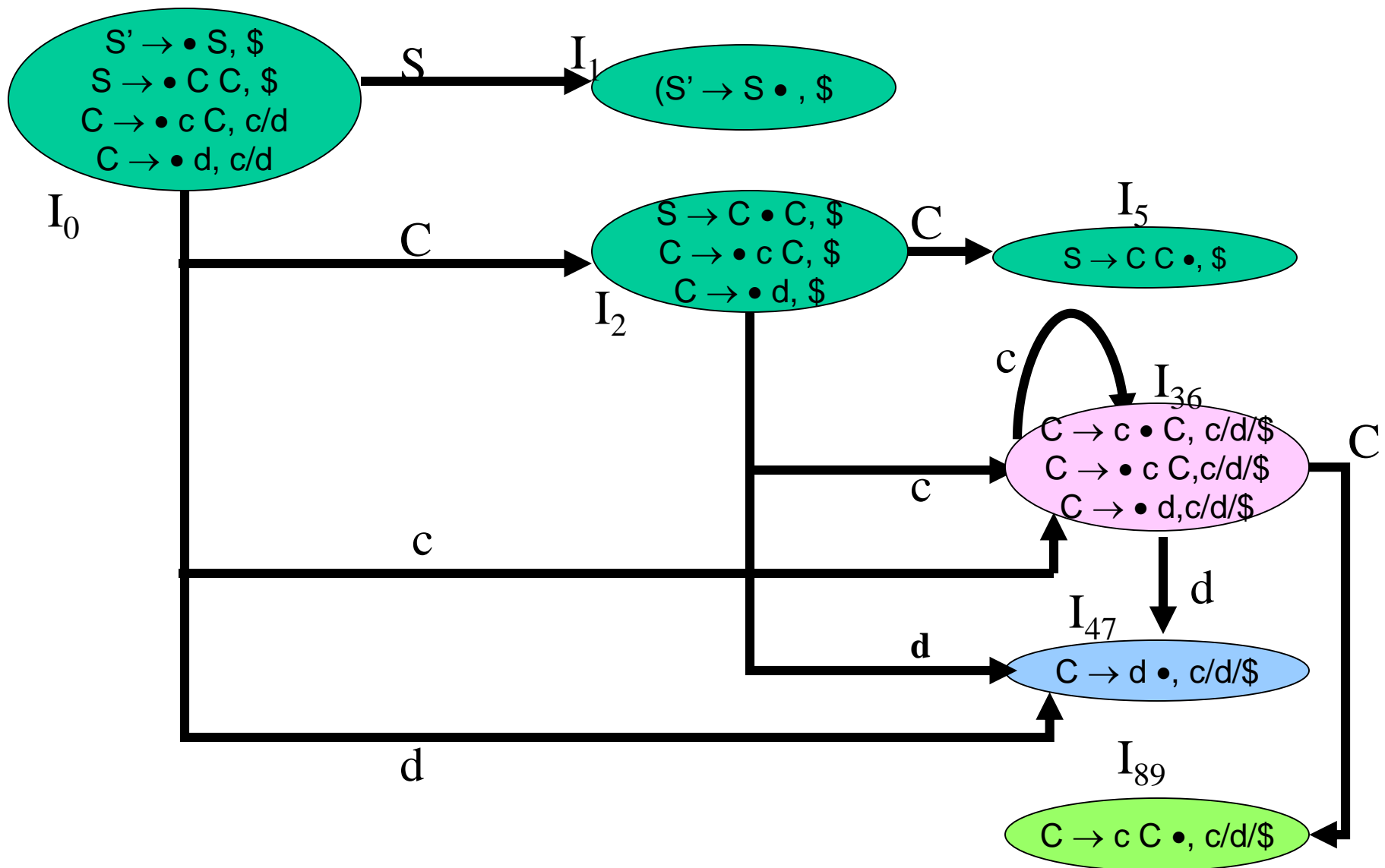
# Creation of LALR Parsing Tables

1. Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
2. For each core present; find all sets having that same core; replace those sets having same cores with a single set which is their union.  
$$C = \{I_0, \dots, I_n\} \rightarrow C' = \{J_1, \dots, J_m\} \quad \text{where } m \leq n$$
3. Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  1. Note that: If  $J = I_1 \cup \dots \cup I_k$  since  $I_1, \dots, I_k$  have same cores  
 $\rightarrow$  cores of  $\text{goto}(I_1, X), \dots, \text{goto}(I_k, X)$  must be same.
  1. So,  $\text{goto}(J, X) = K$  where  $K$  is the union of all sets of items having same cores as  $\text{goto}(I_1, X)$ .
4. 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)











# LALR Parse Table

	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

# 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:

$$A \rightarrow \alpha \bullet, a \quad \text{and} \quad B \rightarrow \beta \bullet a \gamma, b$$

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

$$A \rightarrow \alpha \bullet, a \quad \text{and} \quad B \rightarrow \beta \bullet a \gamma, c$$

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.

$I_1 : A \rightarrow \alpha \bullet, a$

$B \rightarrow \beta \bullet, b$

$I_2 : A \rightarrow \alpha \bullet, b$

$B \rightarrow \beta \bullet, c$



$I_{12} : A \rightarrow \alpha \bullet, a/b$

$B \rightarrow \beta \bullet, b/c$

➔ reduce/reduce conflict

# Canonical LALR(1) Collection – Example2

$$S' \rightarrow S$$
$$I_0:S' \rightarrow \bullet S, \$$$

1)  $S \rightarrow L=R$

$$S \rightarrow \bullet L=R, \$$$
$$2) S \rightarrow R$$
$$S \rightarrow \bullet R, \$$$

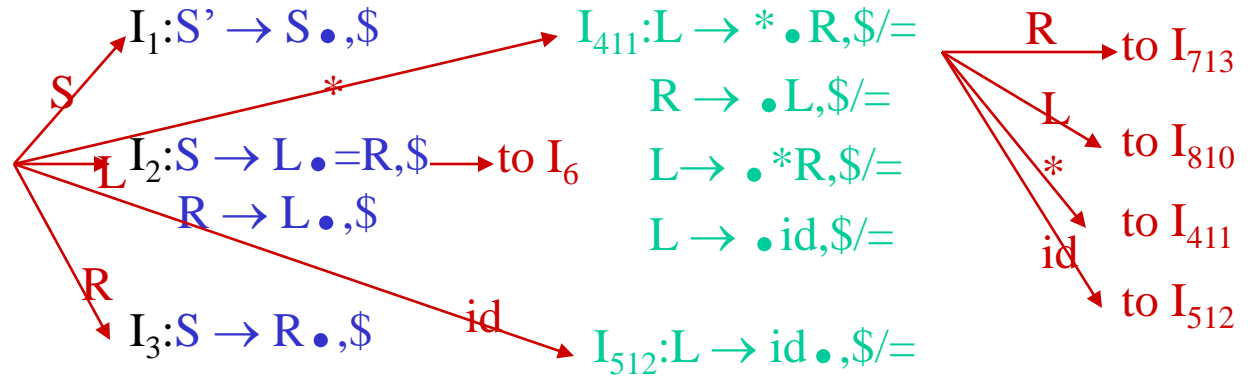
3)  $L \rightarrow *R$

$$L \rightarrow \bullet * R, \$/=$$

4)  $L \rightarrow \text{id}$

$$L \rightarrow \bullet \text{id}, \$ / =$$

5)  $R \rightarrow L$

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

$$I_6: S \rightarrow L = \bullet R, \$$$
$$R \rightarrow \bullet L, \$$$
$$L \rightarrow \bullet * R, \$$$
$$L \rightarrow \bullet \text{id}, \$$$

$R \rightarrow I_9$

to  $I_{810}$

to  $I_{411}$ 

id  $\searrow$  to  $I_{512}$

$$I_9: S \rightarrow L=R \bullet, \$$$

## Same Cores

## I<sub>4</sub> and I<sub>11</sub>

## I<sub>5</sub> and I<sub>12</sub>

## I<sub>7</sub> and I<sub>13</sub>

## I<sub>8</sub> and I<sub>10</sub>

$$I_{713}:L \rightarrow *R \bullet, \$/=$$
$$I_{810}: R \rightarrow L_{\bullet, \$/=}$$

# 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



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.

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

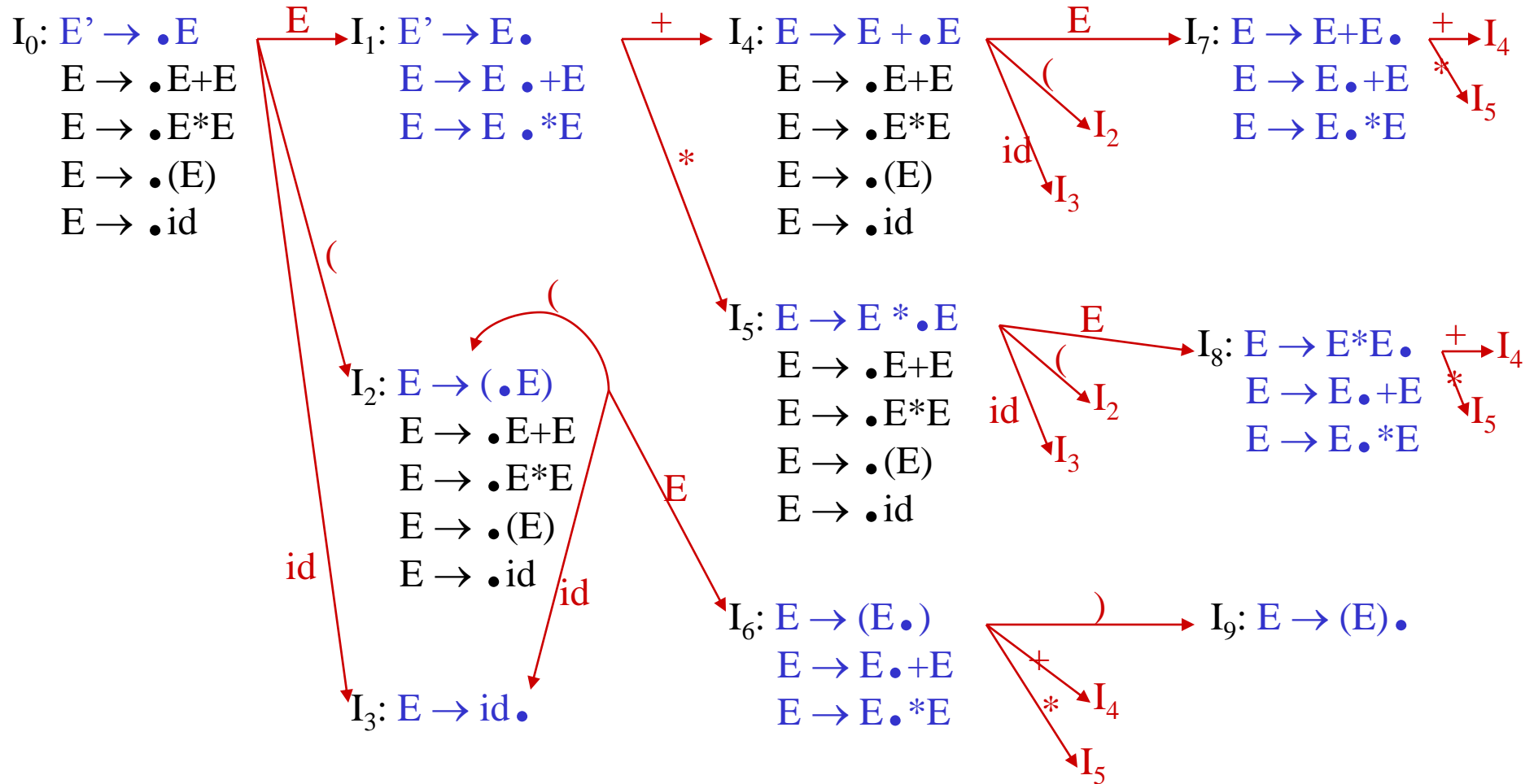


$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{id}$

# Sets of LR(0) Items for Ambiguous Grammar



# SLR-Parsing Tables for Ambiguous Grammar

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

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  $\rightarrow$   $+$  is right-associative

reduce  $\rightarrow$   $+$  is left-associative

when current token is  $*$

shift  $\rightarrow$   $*$  has higher precedence than  $+$

reduce  $\rightarrow$   $+$  has higher precedence than  $*$



# SLR-Parsing Tables for Ambiguous Grammar

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

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

$$I_0 \xrightarrow{E} I_1 \xrightarrow{*} I_5 \xrightarrow{E} I_8$$

when current token is  $*$

shift  $\rightarrow$   $*$  is right-associative

reduce  $\rightarrow$   $*$  is left-associative

when current token is  $+$

shift  $\rightarrow$   $+$  has higher precedence than  $*$

reduce  $\rightarrow$   $*$  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

The End