# LR Parsers

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

LR(k) parsing.

left to right scanning

right-most derivation

k lookhead (k is omitted → it is 1)

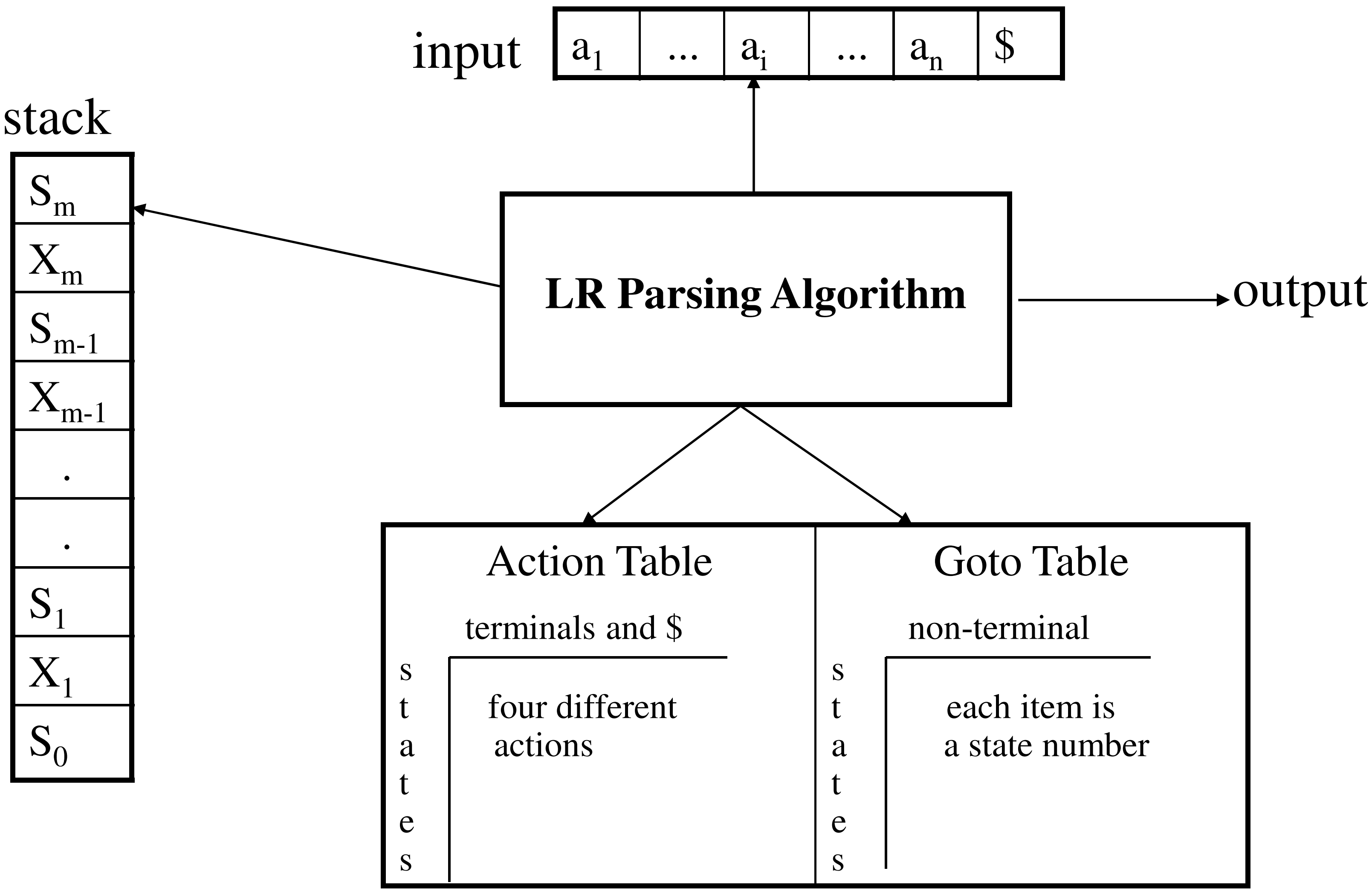
- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    LL(1)-Grammars $\subset$ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so 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 used the same algorithm), only their parsing tables are different.

# LR Parsing Algorithm



# A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$( \underbrace{S_o\ X_1\ S_1 \ldots X_m\ S_m}_{\text{Stack}},\ \underbrace{a_i\ a_{i+1} \ldots a_n\ \$}_{\text{Rest of Input}} )$$

- $S_m$ and $a_i$ decides the parser action by consulting the parsing action table. (*Initial Stack* contains just $S_o$ )

- A configuration of a LR parsing represents the right sentential form:

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

# Actions of A LR-Parser

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

   $( S_o\ X_1\ S_1 \ldots X_m\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$ ) \Rightarrow ( S_o\ X_1\ S_1 \ldots X_m\ S_m\ a_i\ s,\ a_{i+1} \ldots a_n\ \$ )$

2. **reduce A→β** (or **`rn`** where n is a production number)
   - pop 2|β| (=r) items from the stack;
   - then push **A** and **s** where **s=goto[$s_{m-r}$,A]**

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

   - Output is the reducing production reduce A→β

3. **Accept** – Parsing successfully completed

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

# Reduce Action

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

$$( S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ Y_1\ S_{m-r+1} \ldots Y_r\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$\ )$$

$$\rightarrow ( S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ A\ s,\ a_i \ldots a_n\ \$\ )$$

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

$$X_1 \ldots X_{m-r}\ A\ a_i \ldots a_n\ \$ \Rightarrow X_1 \ldots X_m\ Y_1 \ldots Y_r\ a_i\ a_{i+1} \ldots a_n\ \$$$

# (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

| stack | input | action | output |
|---|---|---|---|
| 0 | id*id+id$ | shift 5 | |
| 0id5 | *id+id$ | reduce by F→id | F→id |
| 0F3 | *id+id$ | reduce by T→F | T→F |
| 0T2 | *id+id$ | shift 7 | |
| 0T2*7 | id+id$ | shift 5 | |
| 0T2*7id5 | +id$ | reduce by F→id | F→id |
| 0T2*7F10 | +id$ | reduce by T→T*F | T→T*F |
| 0T2 | +id$ | reduce by E→T | E→T |
| 0E1 | +id$ | shift 6 | |
| 0E1+6 | id$ | shift 5 | |
| 0E1+6id5 | $ | reduce by F→id | F→id |
| 0E1+6F3 | $ | reduce by T→F | T→F |
| 0E1+6T9 | $ | reduce by E→E+T | E→E+T |
| 0E1 | $ | accept | |

# 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 ***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 closure(I).
2. If $\mathbf{A \to \alpha \bullet B \beta}$ is in closure(I) and $\mathbf{B \to \gamma}$ is a production rule of G; then $\mathbf{B \to \bullet \gamma}$ will be in the closure(I). We will apply this rule until no more new LR(0) items can be added to closure(I).

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

# The Closure Operation -- Example

| | | |
|---|---|---|
| $E' \to E$ | closure($\{E' \to \bullet E\}$) = | |
| $E \to E+T$ | $\{$ $E' \to \bullet E$ | ← kernel items |
| $E \to T$ | $E \to \bullet E+T$ | |
| $T \to T*F$ | $E \to \bullet T$ | |
| $T \to F$ | $T \to \bullet T*F$ | |
| $F \to (E)$ | $T \to \bullet F$ | |
| $F \to id$ | $F \to \bullet (E)$ | |
| | $F \to \bullet id$ $\}$ | |

# Computation of Closure

```
function closure ( I )
begin
        J := I;
        repeat
                for each item A → α.Bβ in J and each production
                          B→γ of G such that B→.γ is not in J do
                                  add B→.γ 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 goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha \bullet X\beta$ in I then every item in **closure({$A \rightarrow \alpha X \bullet \beta$})** will be in goto(I,X).
  - If I is the set of items that are valid for some viable prefix $\gamma$, then goto(I,X) is the set of items that are valid for the viable prefix $\gamma X$.

## Example:

I ={ $E' \rightarrow \bullet E$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$,
$T \rightarrow \bullet T*F$, $T \rightarrow \bullet F$,
$F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }

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

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

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

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

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

# 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 { 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** 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.

# The Canonical LR(0) Collection -- Example

$I_0$: E’ → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

$I_1$: E’ → E.
E → E.+T

$I_2$: E → T.
T → T.*F

$I_3$: T → F.

$I_4$: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

$I_5$: F → id.

$I_6$: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

$I_7$: T → T*.F
F → .(E)
F → .id

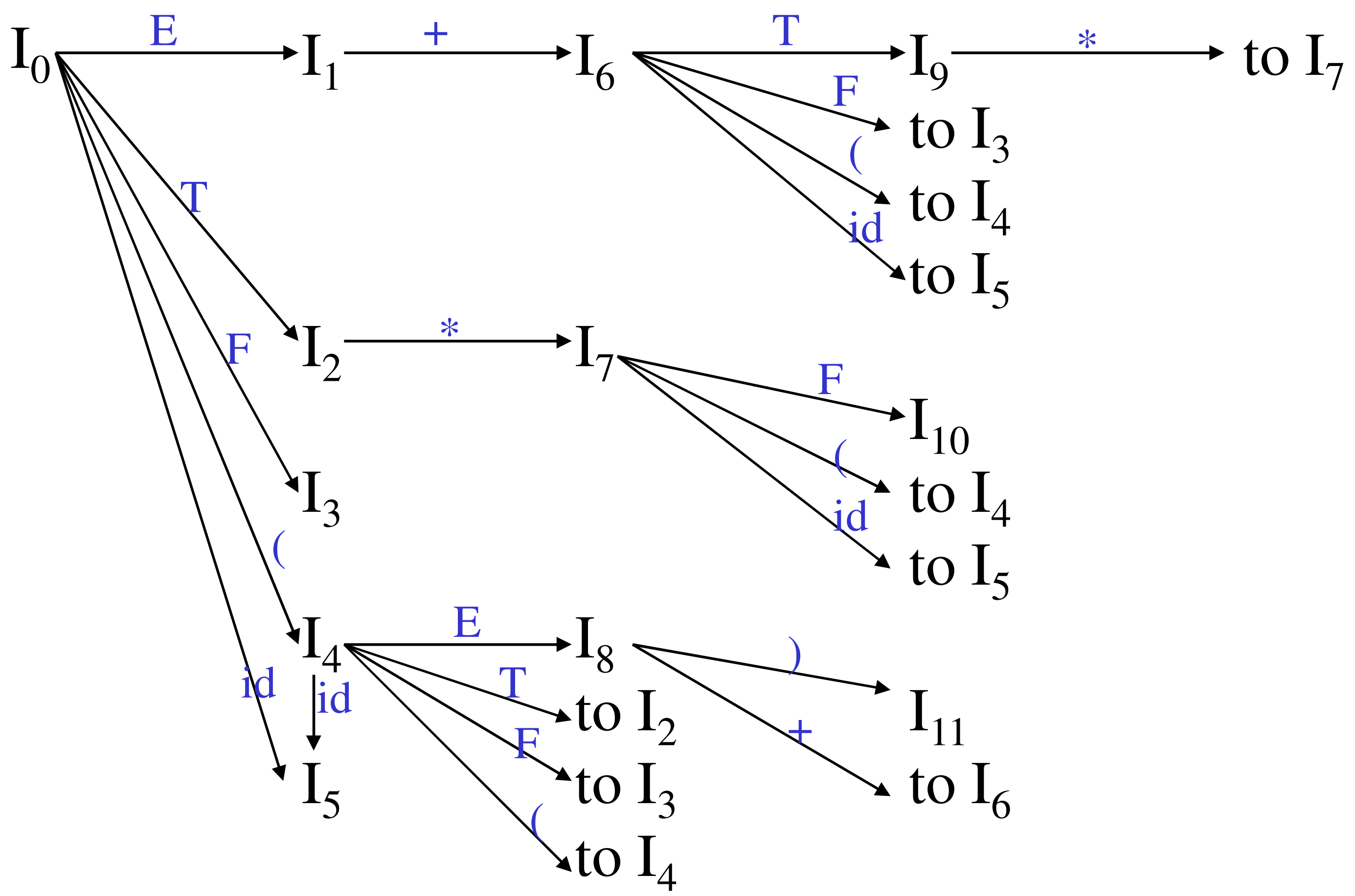
$I_8$: F → (E.)
E → E.+T

$I_9$: E → E+T.
T → T.*F

$I_{10}$: T → T*F.

$I_{11}$: F → (E).

# Transition Diagram (DFA) of Goto Function



# Constructing SLR Parsing Table

**(of an augumented grammar G')**

1. Construct the canonical collection of sets of LR(0) items for G'. $\mathbf{C \leftarrow \{I_0,...,I_n\}}$

2. Create the parsing action table as follows
   - If a is a terminal, $\mathbf{A \rightarrow \alpha.a\beta}$ in $I_i$ and goto($I_i$,a)=$I_j$ then action[i,a] is ***shift j.***
   - If $\mathbf{A \rightarrow \alpha.}$ is in $I_i$ , then **action[i,a]** is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ for all a in **FOLLOW**(A) where $\mathbf{A \neq S'}$.
   - If $\mathbf{S' \rightarrow S.}$ is in $I_i$ , then 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 goto($I_i$,A)=$I_j$ then 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

$S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

$I_0$: $S' \to .S$
$S \to .L=R$
$S \to .R$
$L \to .*R$
$L \to .id$
$R \to .L$

$I_1$: $S' \to S.$

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

$I_3$: $S \to R.$

$I_4$: $L \to *.R$
$R \to .L$
$L \to .*R$
$L \to .id$

$I_5$: $L \to id.$

$I_6$: $S \to L=.R$
$R \to .L$
$L \to .*R$
$L \to .id$

$I_7$: $L \to *R.$

$I_8$: $R \to L.$

$I_9$: $S \to L=R.$

Problem

FOLLOW(R)={=,$}

= → shift 6
→ reduce by $R \to L$

shift/reduce conflict

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

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

[ $S \Rightarrow L=R \Rightarrow *R=R$ ] 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

FOLLOW(A)={a,b}

FOLLOW(B)={a,b}

a → reduce by $A \rightarrow \varepsilon$

a → reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

b → reduce by $A \rightarrow \varepsilon$

b → 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:

  $A \rightarrow \alpha \bullet \beta, a$ 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 $[A \rightarrow \alpha.\beta,a]$, where $\beta$ is not $\in$.
  - But an item of the form $[A \rightarrow \alpha.,a]$ 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 $A \rightarrow \alpha \bullet \beta, a$ ) is not empty, the look-head does not have any affect.
- When $\beta$ is empty ($A \rightarrow \alpha \bullet, a$ ), 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 $A \rightarrow \alpha \bullet, a_1$ where $\{a_1, \ldots, a_n\} \subseteq$ FOLLOW(A)

$$\ldots$$

$$A \rightarrow \alpha \bullet, a_n$$

# 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 $A \rightarrow \alpha \bullet B\beta, a$ 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($\beta a$) .

# goto operation

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

# Construction of The Canonical LR(1) Collection

- ***Algorithm***:

  ***C*** is { closure({S'→.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

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

$$\ldots$$

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

can be written as

$$A \rightarrow \alpha \bullet \beta, a_1/a_2/\ldots/a_n$$

# Canonical LR(1) Collection -- Example

$S \to AaAb$

$S \to BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

$I_0$: $S' \to .S\ ,\$$
$S \to .AaAb\ ,\$$
$S \to .BbBa\ ,\$$
$A \to .\ ,a$
$B \to .\ ,b$

$\xrightarrow{S}$ $I_1$: $S' \to S.\ ,\$$

$\xrightarrow{A}$ $I_2$: $S \to A.aAb\ ,\$$ $\xrightarrow{a}$ to $I_4$

$\xrightarrow{B}$ $I_3$: $S \to B.bBa\ ,\$$ $\xrightarrow{b}$ to $I_5$

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

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

# An Example

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

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

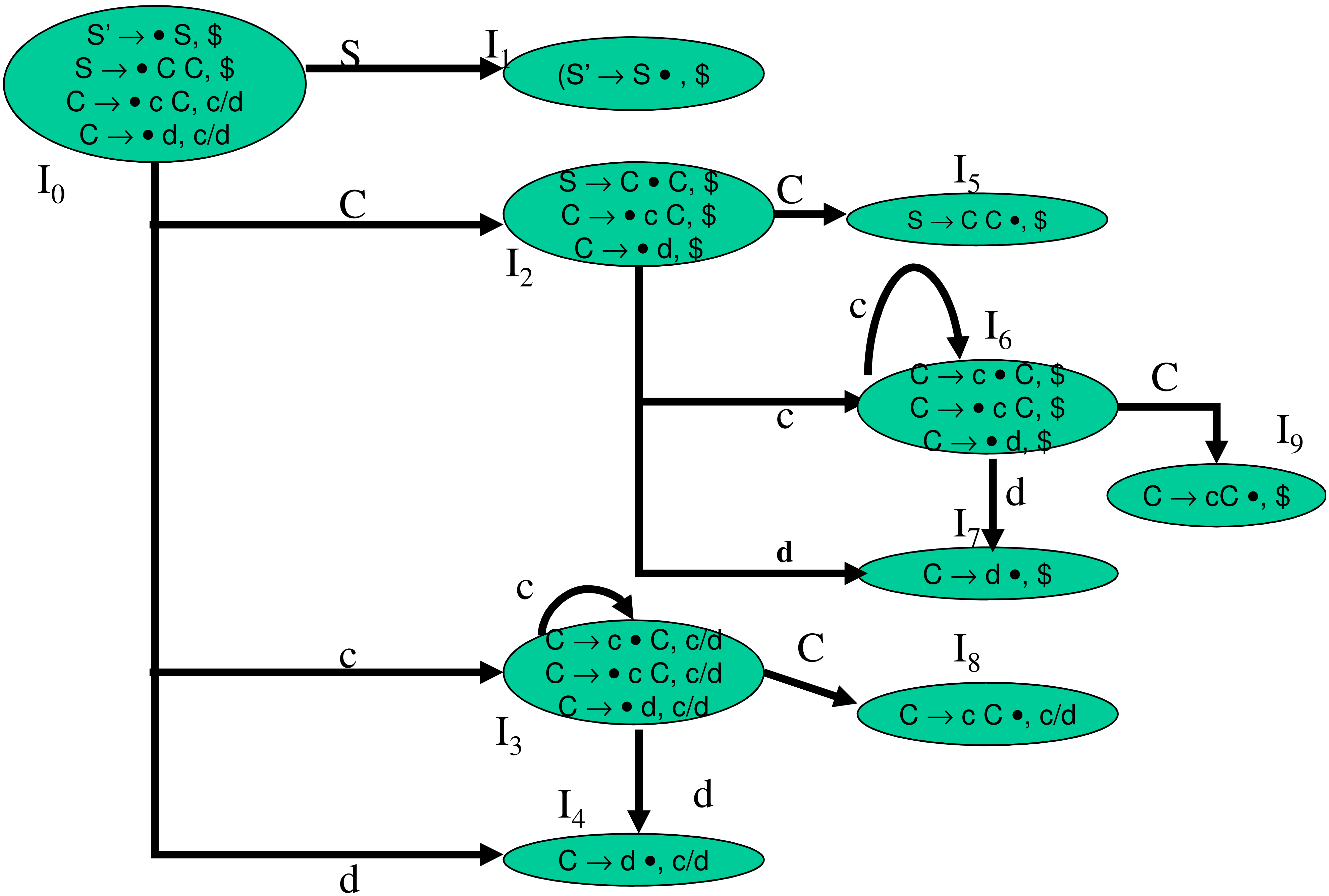
$I_1$: goto$(I_1, S)$ = $(S' \to S\ \bullet, \$)$

$I_2$: goto$(I_1, C)$ =
  $(S \to C\ \bullet\ C, \$)$
  $(C \to \bullet\ c\ C, \$)$
  $(C \to \bullet\ d, \$)$

$I_3$: goto$(I_1, c)$ =
  $(C \to c\ \bullet\ C, c/d)$
  $(C \to \bullet\ c\ C, c/d)$
  $(C \to \bullet\ d, c/d)$

$I_4$: goto$(I_1, d)$ =
  $(C \to d\ \bullet, c/d)$

$I_5$: goto$(I_3, C)$ =
  $(S \to C\ C\ \bullet, \$)$



S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
I2
S → C • C, $
C → • c C, $
C → • d, $
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
c
C
I9
C → cC •, $
d
I7
d
C → d •, $
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
c
C
I8
C → c C •, c/d
d
I4
C → d •, c/d
d

# An Example

$I_6$: goto($I_3$, c) =
$(C \rightarrow c \bullet C, \$)$
$(C \rightarrow \bullet c\ C, \$)$
$(C \rightarrow \bullet d, \$)$

$I_7$: goto($I_3$, d) =
$(C \rightarrow d \bullet, \$)$

$I_8$: goto($I_4$, C) =
$(C \rightarrow c\ C \bullet, c/d)$

: goto($I_4$, c) = $I_4$

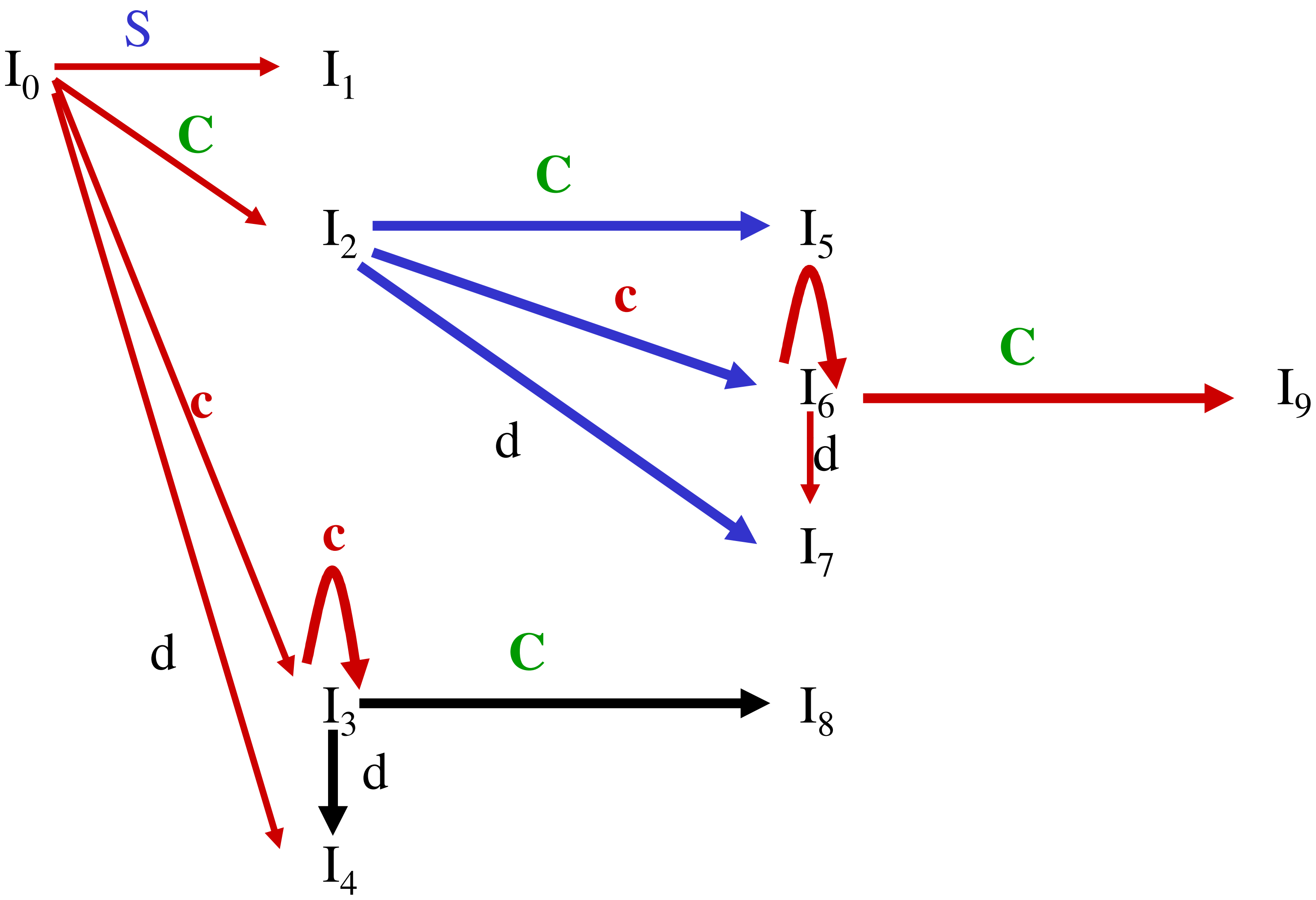
: goto($I_4$, d) = $I_5$

$I_9$: goto($I_7$, c) =
$(C \rightarrow c\ C \bullet, \$)$

: goto($I_7$, c) = $I_7$

: goto($I_7$, d) = $I_8$

# 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

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

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'.

   $C \leftarrow \{I_0,...,I_n\}$

2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha \bullet a\beta,b$ in $I_i$ and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha \bullet ,a$ is in $I_i$ , then action[i,a] is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ where $A \neq S'$.
   - If $S' \rightarrow S \bullet ,\$$ is in $I_i$ , then 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 $goto(I_i,A)=I_j$ then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow .S,\$$

# 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 shrik 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

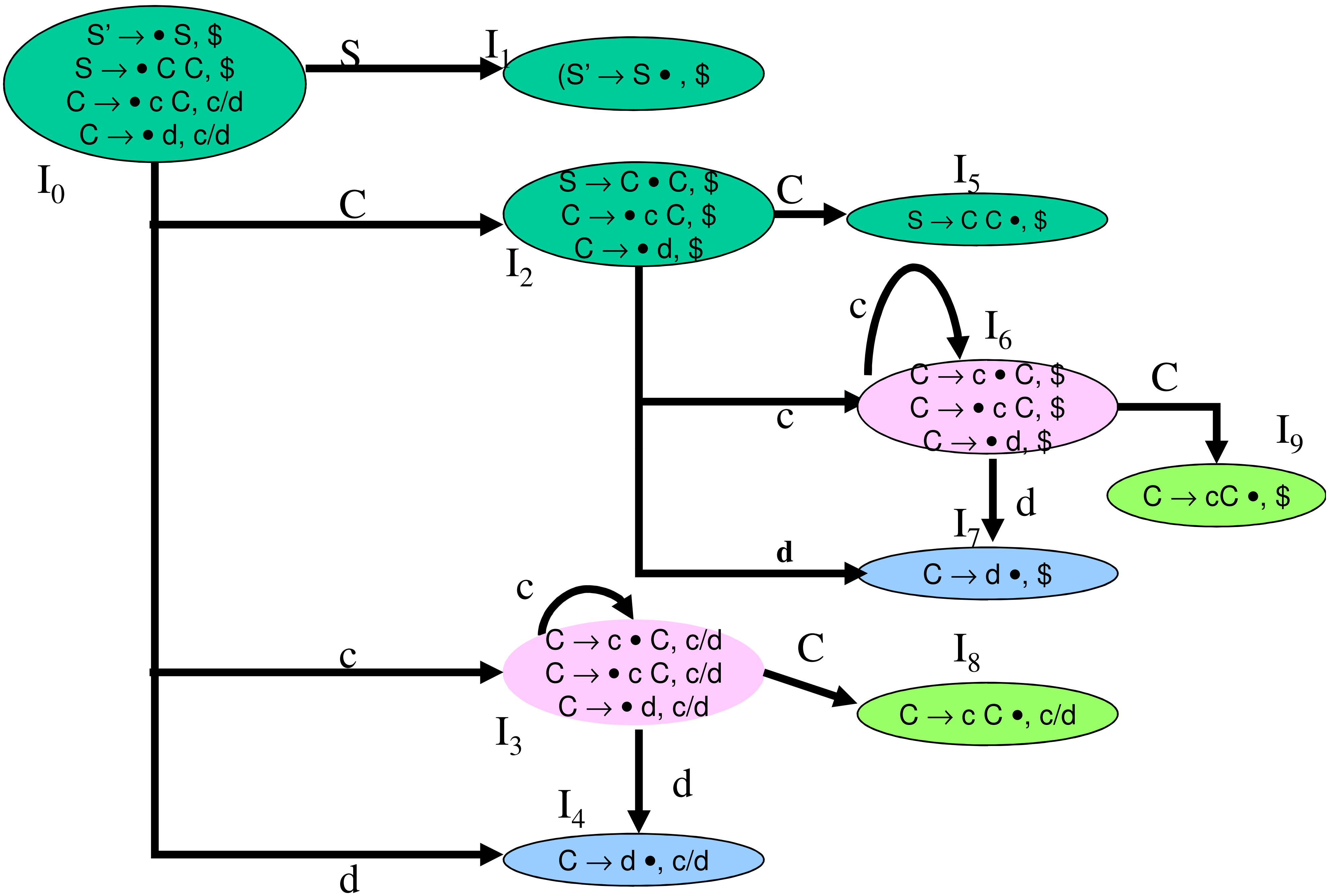
- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

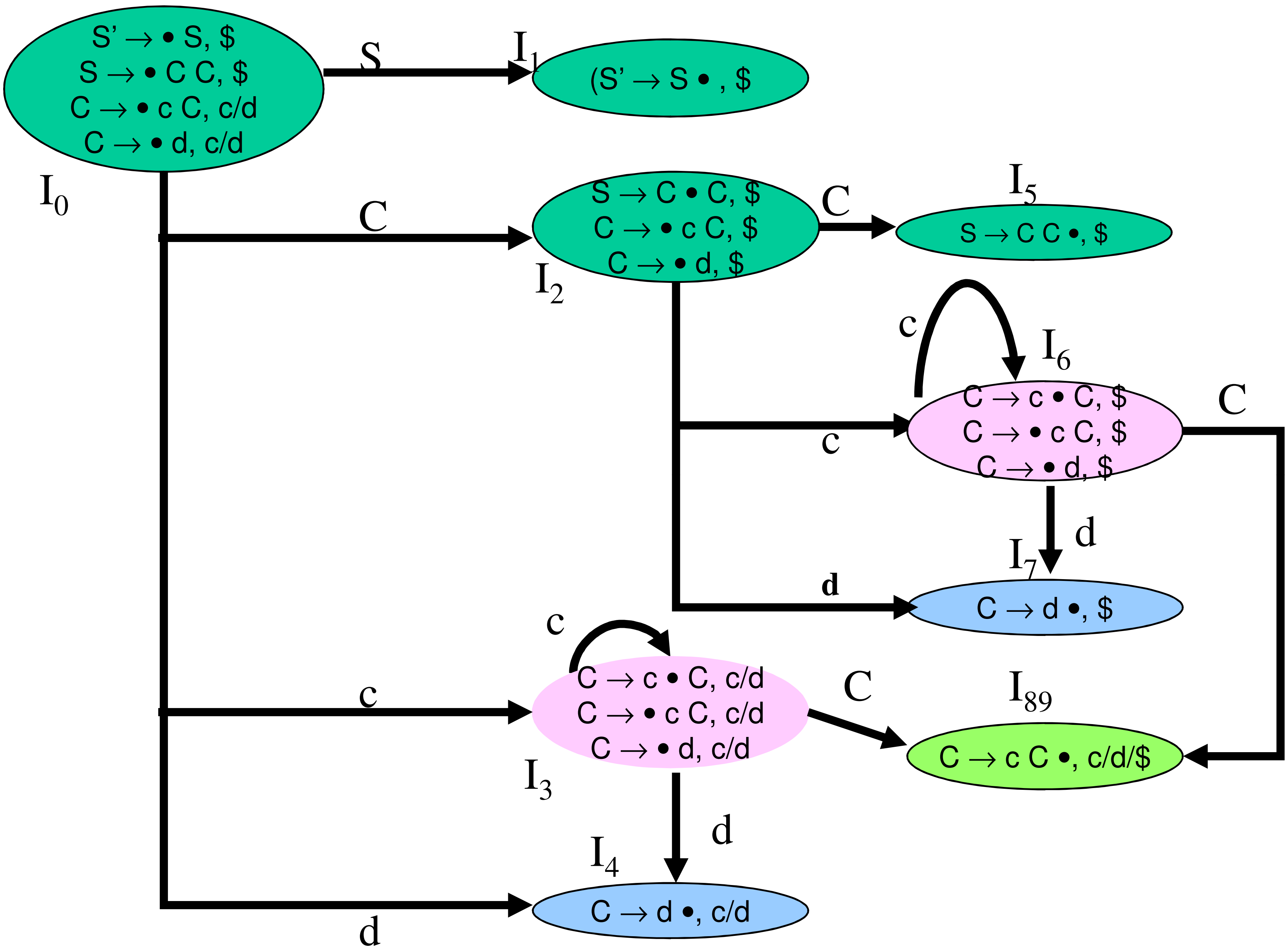
# Creation of LALR Parsing Tables

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.

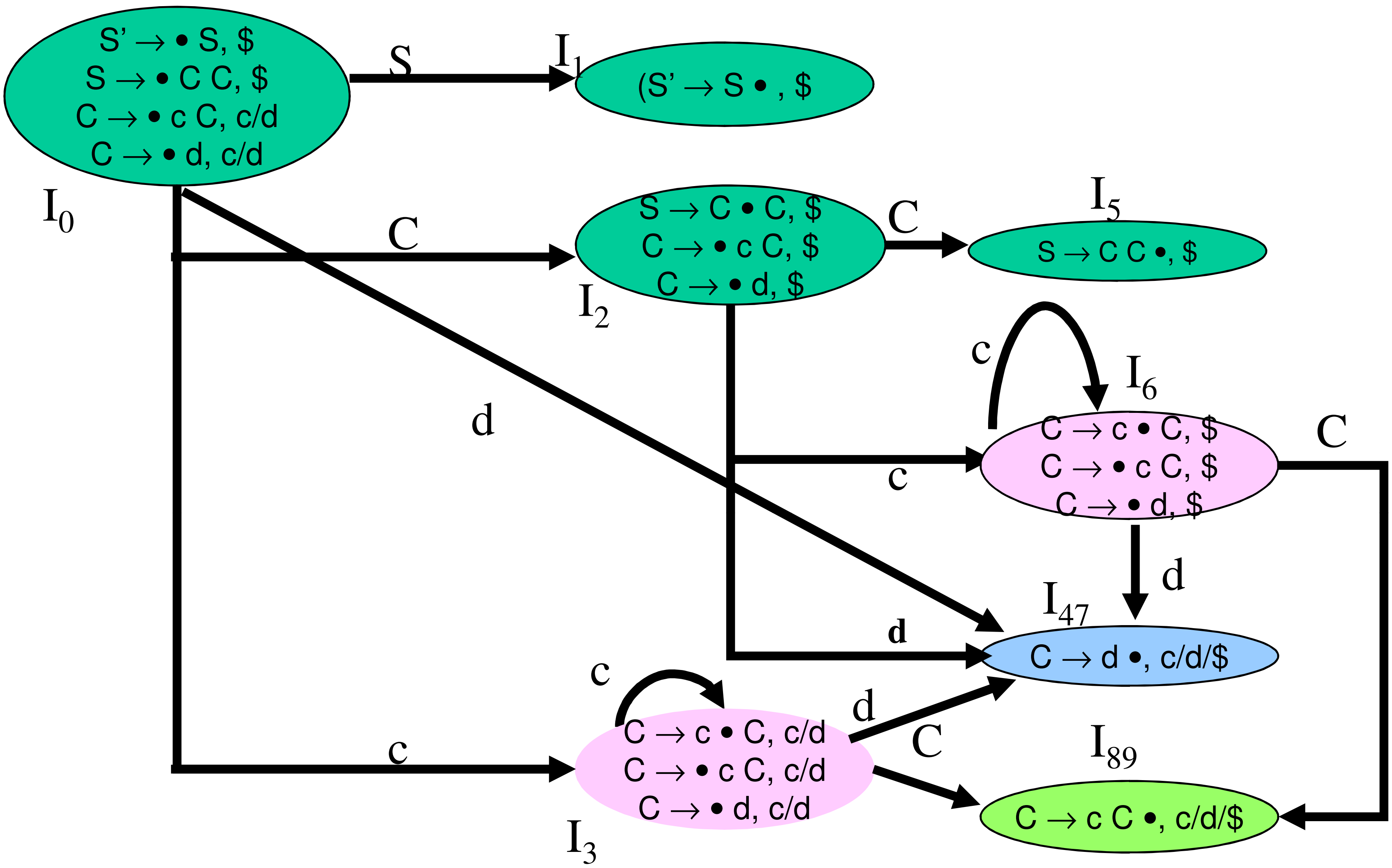
   $C=\{I_0,...,I_n\}$ ➔ $C'=\{J_1,...,J_m\}$ where $m \leq n$
3. Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
   1. Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores

      ➔ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
   1. So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.
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)

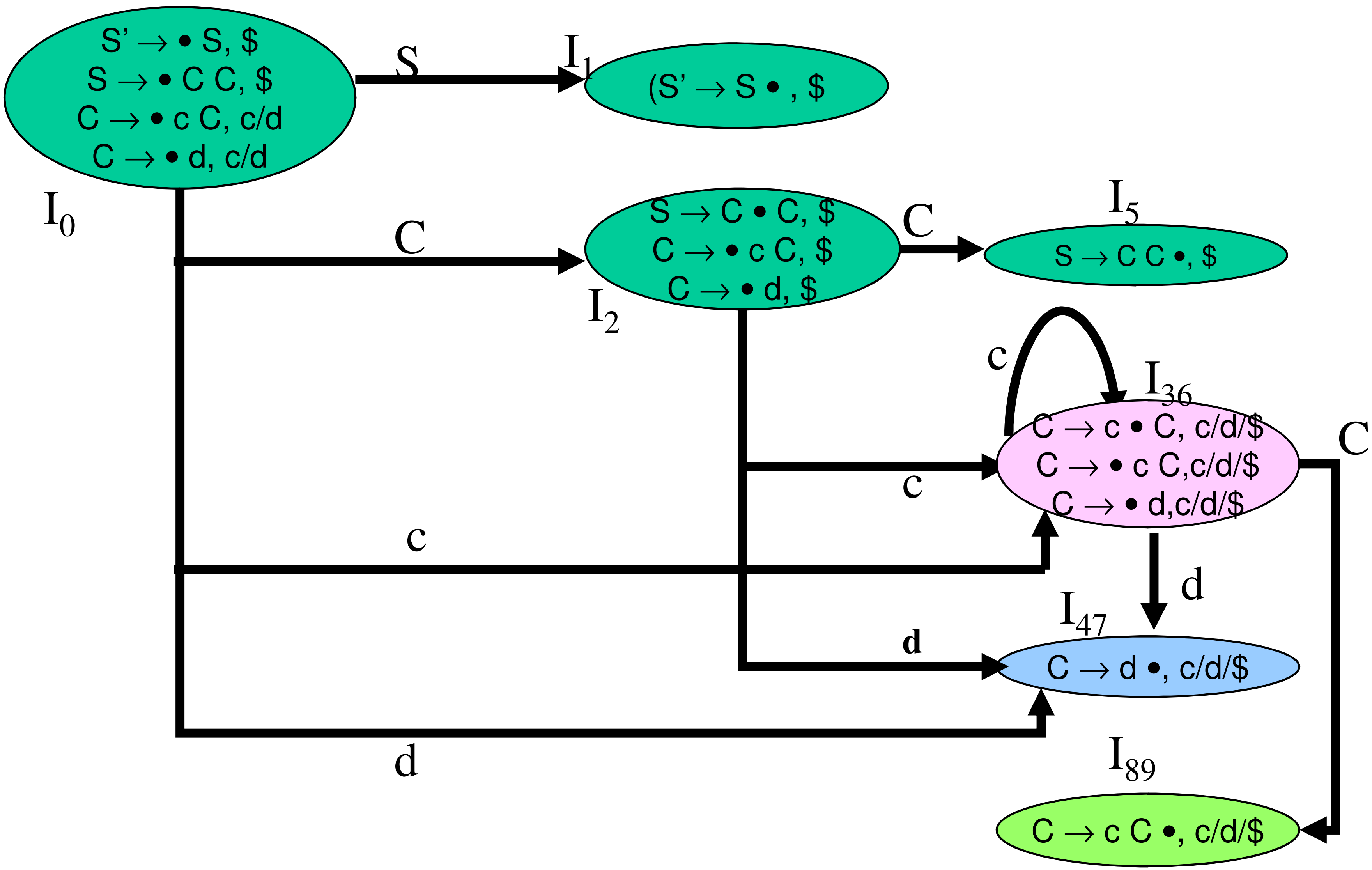

S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
c
C
I9
C → cC •, $
d
I7
d
C → d •, $
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
c
I3
C
I8
C → c C •, c/d
d
I4
C → d •, c/d
d



S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
c
C
d
I7
d
C → d •, $
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
c
C
I89
C → c C •, c/d/$
d
I4
C → d •, c/d
d



S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
C
c
d
d
d
I47
C → d •, c/d/$
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
c
d
C
I89
C → c C •, c/d/$



S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I36
C → c • C, c/d/$
C → • c C,c/d/$
C → • d,c/d/$
C
c
c
d
I47
d
C → d •, c/d/$
d
I89
C → c C •, c/d/$

# 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:

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

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

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

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.

$$I_1 : A \to \alpha\bullet, a \qquad\qquad I_2: A \to \alpha\bullet, b$$
$$\quad B \to \beta\bullet, b \qquad\qquad\quad B \to \beta\bullet, c$$

$$\Downarrow$$

$$I_{12}: A \to \alpha\bullet, a/b$$ ➔ reduce/reduce conflict
$$B \to \beta\bullet, b/c$$

# Canonical LALR(1) Collection – Example2

$S' \to S$
1) $S \to L=R$
2) $S \to R$
3) $L \to *R$
4) $L \to id$
5) $R \to L$

$I_0$: $S' \to \bullet S, \$$
$S \to \bullet L=R, \$$
$S \to \bullet R, \$$
$L \to \bullet *R, \$/=$
$L \to \bullet id, \$/=$
$R \to \bullet L, \$$

$I_0 \xrightarrow{S} I_1$: $S' \to S \bullet, \$$

$I_0 \xrightarrow{L} I_2$: $S \to L \bullet =R, \$ \longrightarrow$ to $I_6$
$R \to L \bullet, \$$

$I_0 \xrightarrow{R} I_3$: $S \to R \bullet, \$$

$I_0 \xrightarrow{*} I_{411}$: $L \to * \bullet R, \$/=$
$R \to \bullet L, \$/=$
$L \to \bullet *R, \$/=$
$L \to \bullet id, \$/=$

$I_{411}$: $\xrightarrow{R}$ to $I_{713}$; $\xrightarrow{L}$ to $I_{810}$; $\xrightarrow{*}$ to $I_{411}$; $\xrightarrow{id}$ to $I_{512}$

$I_0 \xrightarrow{id} I_{512}$: $L \to id \bullet, \$/=$

$I_6$: $S \to L= \bullet R, \$$
$R \to \bullet L, \$$
$L \to \bullet *R, \$$
$L \to \bullet id, \$$

$I_6$: $\xrightarrow{R}$ to $I_9$; $\xrightarrow{L}$ to $I_{810}$; $\xrightarrow{*}$ to $I_{411}$; $\xrightarrow{id}$ to $I_{512}$

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

$I_{713}$: $L \to *R \bullet, \$/=$

$I_{810}$: $R \to L \bullet, \$/=$

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.

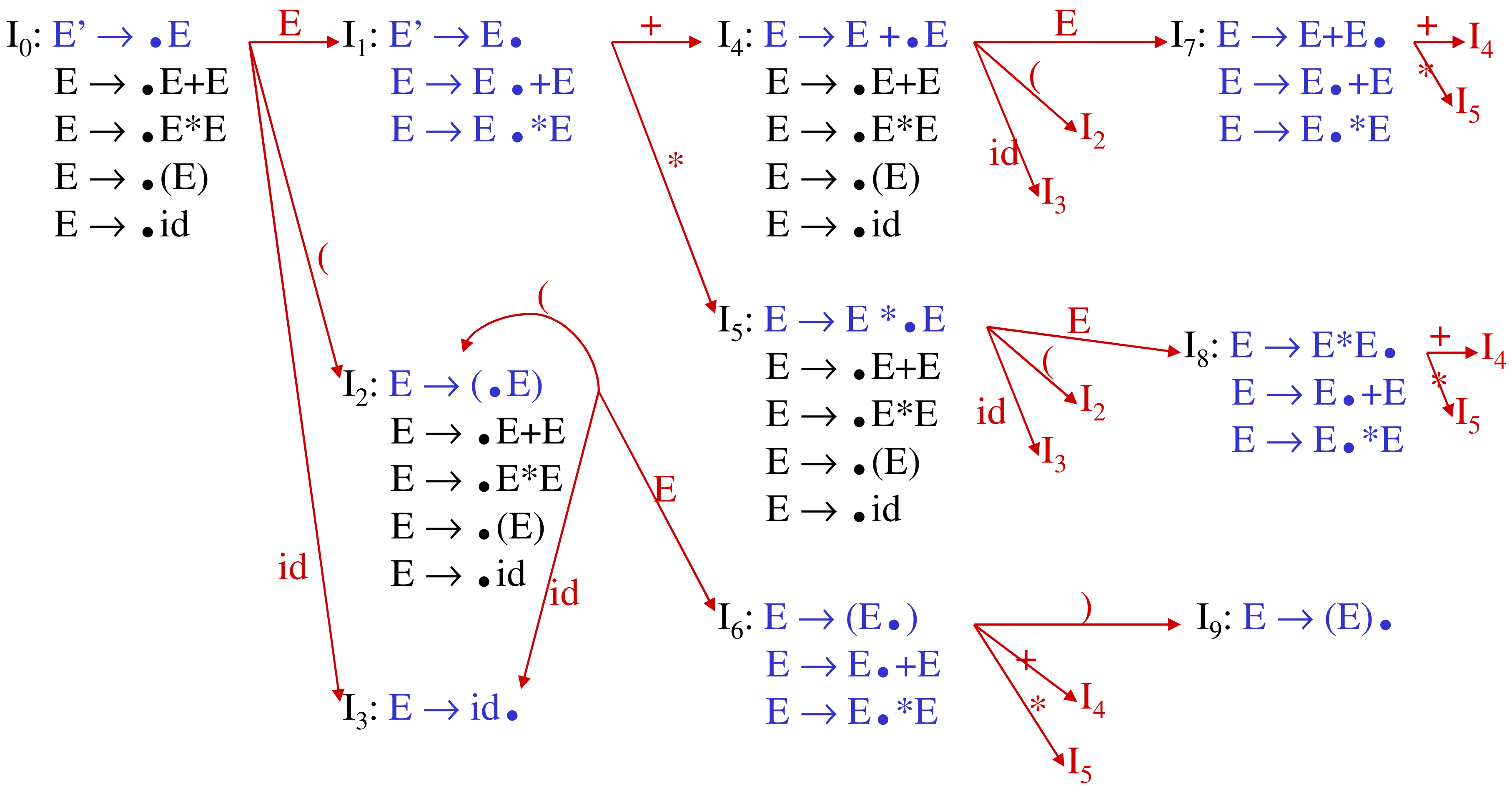
$E \rightarrow E+E \mid E*E \mid (E) \mid id$ ➔

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

# 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 `*`.

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

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

# The End