40-414 Compiler Design

Bottom-Up Parsing-II

Lecture 7
LR (k) parsing (Revisited)

- SLR - Simple LR parser
- LR - most general LR parser
- LALR - intermediate LR parser (Look-Ahead LR)
- SLR, LR, and LALR work exactly the same; only their parse tables are different.
LR(1) Grammar

 Canonical sets of LR(1) items

• Number of states much larger than in the SLR construction
  \( LR(1) = \text{Order of thousands for a standard prog. Lang.} \)
  \( SLR(1) = \text{Order of hundreds for a standard prog. Lang.} \)

LALR(1) (lookahead-LR)

• A tradeoff:
  - Collapse states of the LR(1) table that have the same core (the "LR(0)" part of each state)
  - LALR never introduces a Shift/Reduce Conflict if LR(1) doesn’t.
  - It might introduce a Reduce/Reduce Conflict (that did not exist in the LR(1))
  - Still much better than SLR(1) (larger set of languages)
  - but smaller than LR(1)
Conflict Example

\[
\begin{align*}
S \rightarrow L=R & \quad I_0: \quad S' \rightarrow S \quad I_1: S' \rightarrow S \quad I_6: S \rightarrow L=R \quad I_9: S \rightarrow L=R. \\
S \rightarrow R & \quad S \rightarrow L=R \\
L \rightarrow *R & \quad S \rightarrow R \\
L \rightarrow id & \quad L \rightarrow *R \\
R \rightarrow L & \quad L \rightarrow *R \\
& \quad R \rightarrow L. \\
& \quad R \rightarrow L. \\
& \quad I_3: S \rightarrow R. \\
& \quad I_4: L \rightarrow *R \quad I_7: L \rightarrow *R. \\
& \quad R \rightarrow L. \\
FOLLOW(R) = \{=,\$\} & \quad L \rightarrow *R \\
= & \quad L \rightarrow .id \\
\text{shift 6} & \quad L \rightarrow .id \\
\text{reduce by } R \rightarrow L & \quad I_8: R \rightarrow L. \\
\text{shift/reduce conflict} & \quad I_5: L \rightarrow id.
\end{align*}
\]
Conflict Example

\[ S \rightarrow \text{AaAb} \]
\[ S \rightarrow \text{BbBa} \]
\[ A \rightarrow \varepsilon \]
\[ B \rightarrow \varepsilon \]

Initial derivation:
\[ I_0: \quad S' \rightarrow .S \]
\[ \quad S \rightarrow .\text{AaAb} \]
\[ \quad S \rightarrow .\text{BbBa} \]
\[ \quad A \rightarrow . \]
\[ \quad B \rightarrow . \]

Problem:
\[ \text{FOLLOW}(A) = \{a,b\} \]
\[ \text{FOLLOW}(B) = \{a,b\} \]
\[ a \xrightarrow{\text{reduce by } A \rightarrow \varepsilon} \]
\[ \quad \text{reduce by } B \rightarrow \varepsilon \]
\[ \text{reduce/reduce conflict} \]
\[ b \xrightarrow{\text{reduce by } A \rightarrow \varepsilon} \]
\[ \quad \text{reduce by } B \rightarrow \varepsilon \]
\[ \text{reduce/reduce conflict} \]
**LookAhead versus Follow**

In SLR method, the state $i$ makes a reduction by $A \rightarrow \alpha$ when the current token is $a$:

if the $A \rightarrow \alpha \cdot$ in the $l_i$ 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.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>RMD:</th>
<th>RMD:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1) S \rightarrow AaAb$</td>
<td>$S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$</td>
<td>$S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$</td>
</tr>
<tr>
<td>$2) S \rightarrow BbBa$</td>
<td>Parse (Reverse of RMD):</td>
<td>Parse (Reverse of RMD):</td>
</tr>
<tr>
<td>$3) A \rightarrow \varepsilon$</td>
<td>$Aab \Rightarrow \varepsilon \ ab$ (correct follow is $a$)</td>
<td>$Bba \Rightarrow \varepsilon \ ba$</td>
</tr>
<tr>
<td>$4) B \rightarrow \varepsilon$</td>
<td>$AaAb \Rightarrow Aa \ \varepsilon \ b$ (correct follow is $b$)</td>
<td>$BbBa \Rightarrow Bb \ \varepsilon \ a$</td>
</tr>
</tbody>
</table>
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 lookahead of the LR(1) item

\( a \) is a terminal or end-marker.)
LR(1) items

• When $\beta$ (in the LR(1) item $A \to \alpha\cdot\beta,a$) is not empty, the lookahead $a$ does not have any affect.

• When $\beta$ is empty ($A \to \alpha\cdot,a$), we do the reduction by $A \to \alpha$ only if the next input symbol is $a$ (not for any terminal in FOLLOW($A$)).

• A state will contain $A \to \alpha\cdot,a_1$ where $\{a_1,\ldots,a_n\} \subseteq$ FOLLOW($A$)

  ...

  $A \to \alpha\cdot,a_n$
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.

\textbf{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(\( \beta a \)).
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:
  \[ C \text{ is } \{ \text{closure}({S'\rightarrow S,\$}) \} \]
  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

\[ A \rightarrow \alpha \cdot \beta , a_1 \]

\[ \ldots \]

\[ A \rightarrow \alpha \cdot \beta , a_n \]

can be written as

\[ A \rightarrow \alpha \cdot \beta , \{ a_1, a_2, \ldots, a_n \} \]
Canonical LR(1) Collection - Example

S → AaAb
S → BbBa
A → ε
B → ε

I₀: S' → .S ,$
S → .AaAb ,$
S → .BbBa ,$
A → . ,a
B → . ,b

I₁: S' → S . ,$
S → AaAb ,$
A → . ,a
I₂: S → A.aAb ,$
B → . ,b

I₃: S → B.bBa ,$
B → . ,a

I₄: S → Aa.Ab ,$
A → . ,b

I₅: S → Bb.Ba ,$
B → . ,a

I₆: S → AaA.b ,$
I₇: S → BbB.a ,$
I₈: S → AaAb. ,$
I₉: S → BbBa. ,$

→ to I₄
→ to I₅
→ to I₆
→ to I₇
→ to I₈
→ to I₉
Canonical LR(1) Collection - Example 2

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

I₆: S' → .S,$
I₇: L → *R., {$,=}
I₈: R → L., {$,=}
I₉: S → L=R.,$
I₁₀: R → L.,$
I₁₁: L → *R.,$
I₁₂: L → id.,$
I₁₃: L → *R.,$
I₁₄: L → *.R, {$,=}
I₁₅: L → .R, {$,=}
I₁₆: L → .L, {$,=}
I₁₇: L → .id, {$,=}
I₁₈: L → .L, {$,=}
I₁₉: L → .id, {$,=}
I₂₀: L → .R.$

R → .L, {$,=}
L → *.R, {$,=}
L → .id, {$,=}
L → .L,$

I₁: S' → S.,$
I₂: S → L.=R,$
I₃: S → R.,$
I₄: L → *.R, {$,=}
I₅: L → id., {$,=}

R → .L, {$,=}
L → *.R, {$,=}
L → .id, {$,=}
L → .L,$

I₁: S' → S.,$
I₂: S → L.=R,$
I₃: S → R.,$
I₄: L → *.R, {$,=}
I₅: L → id., {$,=}

R → .L, {$,=}
L → *.R, {$,=}
L → .id, {$,=}
L → .L,$

I₁: S' → S.,$
I₂: S → L.=R,$
I₃: S → R.,$
I₄: L → *.R, {$,=}
I₅: L → id., {$,=}

R → .L, {$,=}
L → *.R, {$,=}
L → .id, {$,=}
L → .L,$

I₁: S' → S.,$
I₂: S → L.=R,$
I₃: S → R.,$
I₄: L → *.R, {$,=}
I₅: L → id., {$,=}

R → .L, {$,=}
L → *.R, {$,=}
L → .id, {$,=}
L → .L,$

I₁: S' → S.,$
I₂: S → L.=R,$
I₃: S → R.,$
I₄: L → *.R, {$,=}
I₅: L → id., {$,=}

R → .L, {$,=}
L → *.R, {$,=}
L → .id, {$,=}
L → .L,$

I₁: S' → S.,$
I₂: S → L.=R,$
I₃: S → R.,$
I₄: L → *.R, {$,=}
I₅: L → id., {$,=}

R → .L, {$,=}
L → *.R, {$,=}
L → .id, {$,=}
L → .L,$

I₁: S' → S.,$
I₂: S → L.=R,$
I₃: S → R.,$
I₄: L → *.R, {$,=}
I₅: L → id., {$,=}

R → .L, {$,=}
L → *.R, {$,=}
L → .id, {$,=}
L → .L,$

I₁: S' → S.,$
I₂: S → L.=R,$
I₃: S → R.,$
I₄: L → *.R, {$,=}
I₅: L → id., {$,=}

R → .L, {$,=}
L → *.R, {$,=}
L → .id, {$,=}
L → .L,$
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, $A \rightarrow \alpha \cdot a \beta, b$ in $I_i$ and $\text{goto}(I_i, a) = I_j$ then action\[i, a\] is shift $j$.
     • If $A \rightarrow \alpha \cdot a$ is in $I_i$, then action\[i, a\] is reduce $A \rightarrow \alpha$ where $A \neq S'$.
     • If $S' \rightarrow S \cdot, \_$ 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 $\text{goto}(I_i, A) = I_j$ then $\text{goto}[i, A] = j$

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

5. Initial state of the parser contains $S' \rightarrow .S, _$
LR(1) Parsing Tables

<table>
<thead>
<tr>
<th>id</th>
<th>*</th>
<th>=</th>
<th>$</th>
<th>S</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s6</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
<td>8</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s12</td>
<td>s11</td>
<td>10</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>s12</td>
<td>s11</td>
<td>10</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

no shift/reduce or no reduce/reduce conflict

so, it is a LR(1) grammar
LALR(1) Parsing Tables

- 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.
LALR(1) 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: \[ S \rightarrow L \cdot = R, $ \rightarrow S \rightarrow L \cdot = R \quad (Core) \]

- 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.

\[ I_1: L \rightarrow id \cdot, = \]

\[ \rightarrow \text{A new state: } I_{12}: L \rightarrow id \cdot, \{=, $\} \]

\[ I_2: L \rightarrow id \cdot, $ \quad (\text{have same core, merge the lookaheads}) \]

- 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.
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:

\[ A \rightarrow \alpha.,a \quad \text{and} \quad B \rightarrow \beta.a_{\gamma},b \]

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

\[ A \rightarrow \alpha.,a \quad \text{and} \quad B \rightarrow \beta.a_{\gamma},c \]

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.

\[
I_1 : A \rightarrow \alpha, a \\
B \rightarrow \beta, b
\]

\[
I_2 : A \rightarrow \alpha, b \\
B \rightarrow \beta, c
\]

\[
\downarrow
\]

\[
I_{12} : A \rightarrow \alpha, \{a, b\} \xrightarrow{\text{reduce/reduce conflict}} \\
B \rightarrow \beta, \{b, c\}
\]
Canonical LALR(1) Collection

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

I₀:S' → ⋅S,$
I₁:S' → S ⋅,$
I₂:S → L ⋅=R,$ → I₆
R → ⋅L,$
R → L ⋅,$
I₃:S → R ⋅,$
I₄₁₁:L → ⋅ ⋅ R, {$,=}$
R → ⋅L, {$,=}$
L → ⋅*R, {$,=}$
L → ⋅id, {$,=}$
I₅₁₂:L → id ⋅, {$,=}$
I₆:S → L ⋅=R,$
I₇₁₃:L → *R ⋅, {$,=}$
R → L ⋅,$
R → id,$
L → *R,$
L → id,$
I₈₁₀: R → L ⋅, {$,=}$
I₉:S → L=R ⋅,$
I₄₁₁:*L, {$,=}$
I₅₁₂:*id, {$,=}$

Same Cores
I₄ and I₁₁
I₅ and I₁₂
I₇ and I₁₃
I₈ and I₁₀
## LALR(1) Parse Table

<table>
<thead>
<tr>
<th>id</th>
<th>*</th>
<th>=</th>
<th>$</th>
<th>S</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s512</td>
<td>s411</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>s6</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>411</td>
<td>s512</td>
<td>s411</td>
<td></td>
<td>810</td>
<td>713</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s512</td>
<td>s411</td>
<td></td>
<td>810</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>713</td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>810</td>
<td></td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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?
  o Yes, but they will have conflicts.
  o We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  o At the end, we will have again an unambiguous grammar.

• Why we want to use an ambiguous grammar?
  o Some of the ambiguous grammars are much natural, and a corresponding unambiguous grammar can be very complex.
  o Usage of an ambiguous grammar may eliminate unnecessary reductions.

• Ex.

\[
\begin{align*}
E \rightarrow & \ E+T \mid T \\
E \rightarrow & \ E+E \mid E*E \mid (E) \mid \text{id} \\
T \rightarrow & \ T*F \mid F \\
F \rightarrow & \ (E) \mid \text{id}
\end{align*}
\]
Sets of LR(0) Items for Ambiguous Grammar

$I_0$: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot id$

$I_1$: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot +E$
$E \rightarrow E \cdot *E$

$I_2$: $E \rightarrow (\cdot E)$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot id$

$I_3$: $E \rightarrow id \cdot$

$I_4$: $E \rightarrow E + \cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot id$

$I_5$: $E \rightarrow E \cdot \cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot id$

$I_6$: $E \rightarrow (E \cdot)$
$E \rightarrow E \cdot +E$
$E \rightarrow E \cdot *E$

$I_7$: $E \rightarrow E+\cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot id$

$I_8$: $E \rightarrow E \cdot \cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot id$

$I_9$: $E \rightarrow (E \cdot)$

$I_{14}$
FOLLOW(E) = { $, +, *, ) } 

State \text{I}_7 \text{ has shift/reduce conflicts for symbols } + \text{ and } * .  

\begin{align*}  
\text{I}_0 & \xrightarrow{E} \text{I}_1 \xrightarrow{+} \text{I}_4 \xrightarrow{E} \text{I}_7 
\end{align*}  

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

<table>
<thead>
<tr>
<th>Action</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td></td>
<td>s2</td>
<td></td>
<td></td>
<td>$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td></td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td></td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
<td></td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
<td>s5</td>
<td></td>
<td></td>
<td>s9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>r1</td>
<td>s5</td>
<td></td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>r2</td>
<td>r2</td>
<td></td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
</tbody>
</table>
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/Local 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