Syntax Analysis, VII

One more LR(1) example, plus some more stuff

Comp 412
Computing Closures

**Closure(s)** adds all the **possibilities** for the items already in s

- Any item \([A \rightarrow \beta \cdot B \delta, a]\) where \(B \in NT\) implies \([B \rightarrow \cdot \tau, x]\) for each production that has \(B\) on the *lhs*, and each \(x \in \text{FIRST}(\delta a)\)
- Since \(\beta B \delta\) is valid, any way to derive \(\beta B \delta\) is valid, too

**The Algorithm**

\[
\text{Closure}( s )
\]

\[
\text{while ( } s \text{ is still changing )}
\]

\[
\forall \text { items } [A \rightarrow \beta \cdot B \delta, a] \in s
\]

\[
\text{lookahead } \leftarrow \text{FIRST}(\delta a) \text{ // } \delta \text{ might be } \epsilon
\]

\[
\forall \text { productions } B \rightarrow \tau \in P
\]

\[
\forall b \in \text{lookahead}
\]

\[
\text{if } [B \rightarrow \cdot \tau, b] \not\in s
\]

\[
\text{then } s \leftarrow s \cup \{ [B \rightarrow \cdot \tau, b] \}
\]

- Classic fixed-point method
- Halts because \(s \subseteq I\), the set of all items (finite)
- Worklist version is faster
- **Closure** “fills out” a state \(s\)

Generate new lookaheads.
See note on p. 128

COMP 412, Fall 2017
Computing Gotos

**Goto** computes the state that the parser would reach if it recognized an $x$ while in state $s$

- **Goto** ( $\{ [A\rightarrow \beta \cdot X\delta, a] \}$, $X$ ) produces $\{ [A\rightarrow \beta X \cdot \delta, a] \}$ (obviously)
- It finds all such items & uses **Closure()** to fill out the state

The Algorithm

Goto( $s$, $X$ )

$\text{new } \leftarrow \emptyset$

$\forall \text{ items } [A\rightarrow \beta \cdot X\delta, a] \in s$

$\text{new } \leftarrow \text{new } \cup \{ [A\rightarrow \beta X \cdot \delta, a] \}$

return **Closure** ( new )

- **Goto** ( ) models a transition in the automaton
- Straightforward computation
- **Goto** ( ) is not a fixed-point method (but it calls **Closure()**)

Goto in this construction is analogous to Move in the subset construction.
Building the Canonical Collection

Start from \( s_0 = \text{Closure}( [S' \rightarrow \bullet S, \text{EOF}] ) \)

Repeatedly construct new states, until all are found

**The Algorithm**

\[
\begin{align*}
 s_0 & \leftarrow \text{Closure}( \{ [S' \rightarrow \bullet S, \text{EOF}] \} ) \\
 S & \leftarrow \{ s_0 \} \\
 k & \leftarrow 1 \\
 \text{while (} S \text{ is still changing) } & \\
 & \forall s_j \in S \text{ and } \forall x \in (T \cup NT) \\
 & \quad s_k \leftarrow \text{Goto}(s_j, x) \\
 & \quad \text{record } s_j \rightarrow s_k \text{ on } x \\
 & \quad \text{if } s_k \notin S \text{ then} \\
 & \quad S \leftarrow S \cup \{ s_k \} \\
 & \quad k \leftarrow k + 1
\end{align*}
\]

- Fixed-point computation
- Loop adds to \( S \) (monotone)
- \( S \subseteq 2^{\text{ITEMS}} \), so \( S \) is finite

**Worklist version is faster because it avoids duplicated effort**

This membership / equality test requires careful and/or clever implementation.
Filling in the ACTION and GOTO Tables

The Table Construction Algorithm

\( \forall \text{ set } S_x \in S \)
\( \forall \text{ item } i \in S_x \)
\( \text{if } i \text{ is } [A \rightarrow \beta \cdot a \delta, b] \text{ and } \text{goto}(S_x, a) = S_k, \ a \in T \)
then ACTION\([x, a]\) \leftarrow “shift k”

else if \( i \text{ is } [S' \rightarrow S \cdot, \text{EOF}] \)
then ACTION\([x, \text{EOF}]\) \leftarrow “accept”

else if \( i \text{ is } [A \rightarrow \beta \cdot a] \)
then ACTION\([x, a]\) \leftarrow “reduce A \rightarrow \beta”

\( \forall \ n \in NT \)
if \( \text{goto}(S_x, n) = S_k \)
then GOTO\([x, n]\) \leftarrow k

Many items generate no table entry

\( \rightarrow \) Placeholder before a NT does not generate an ACTION table entry

\( \rightarrow \textbf{Closure}( ) \) instantiates FIRST\( (X) \) directly for \([A \rightarrow \beta \cdot X \delta, a]\)
Another Example

*Simplified, right recursive expression grammar*

0  Goal    $\rightarrow$  Expr
1  Expr    $\rightarrow$  Term  -  Expr
2       |  Term
3  Term    $\rightarrow$  Factor  *  Term
4       |  Factor
5  Factor  $\rightarrow$  id

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>{ id }</td>
</tr>
<tr>
<td>Expr</td>
<td>{ id }</td>
</tr>
<tr>
<td>Term</td>
<td>{ id }</td>
</tr>
<tr>
<td>Factor</td>
<td>{ id }</td>
</tr>
<tr>
<td>-</td>
<td>{ - }</td>
</tr>
<tr>
<td>*</td>
<td>{ * }</td>
</tr>
<tr>
<td>id</td>
<td>{ id }</td>
</tr>
</tbody>
</table>
Simplified Expression Grammar

Initialization Step

\[ s_0 \leftarrow \text{closure}( \{ [\text{Goal} \rightarrow \bullet \text{Expr} , \text{EOF}] \} ) \]

\{ [\text{Goal} \rightarrow \bullet \text{Expr} , \text{EOF}],

[\text{Expr} \rightarrow \bullet \text{Term} \rightarrow \text{Expr} , \text{EOF}],

[\text{Expr} \rightarrow \bullet \text{Term} , \text{EOF}],

[\text{Term} \rightarrow \bullet \text{Factor} * \text{Term} , \text{EOF}],

[\text{Term} \rightarrow \bullet \text{Factor} * \text{Term} , -],

[\text{Term} \rightarrow \bullet \text{Factor} , \text{EOF}],

[\text{Term} \rightarrow \bullet \text{Factor} , -],

[\text{Factor} \rightarrow \bullet \text{id} , \text{EOF}],

[\text{Factor} \rightarrow \bullet \text{id} , -],

[\text{Factor} \rightarrow \bullet \text{id} , *] \}

\[ S \leftarrow \{ s_0 \} \]

Item in **black** is the initial item.
Items in **gray** are added by closure().
Simplified Expression Grammar

Iteration 1

\[ s_1 \leftarrow \text{goto}(s_0, \text{Expr}) \]
\[ s_2 \leftarrow \text{goto}(s_0, \text{Term}) \]
\[ s_3 \leftarrow \text{goto}(s_0, \text{Factor}) \]
\[ s_4 \leftarrow \text{goto}(s_0, \text{id}) \]

Iteration 2

\[ s_5 \leftarrow \text{goto}(s_2, -) \]
\[ s_6 \leftarrow \text{goto}(s_3, *) \]

Iteration 3

\[ s_7 \leftarrow \text{goto}(s_5, \text{Expr}) \]
\[ s_8 \leftarrow \text{goto}(s_6, \text{Term}) \]

Goal, *, & - generate empty sets.

Goal, Expr, Term, Factor, & id generate empty sets.

Goal, *, & - generate empty sets. Term, Factor, & id start to re-create existing sets.
Simplified Expression Grammar

\[ s_0 \leftarrow \text{closure}( \{ [\text{Goal} \rightarrow \cdot \text{Expr}, \text{EOF}] \} ) \]
\[ \{ [\text{Goal} \rightarrow \cdot \text{Expr}, \text{EOF}], \\
   [\text{Expr} \rightarrow \cdot \text{Term} \cdot \text{Expr}, \text{EOF}], [\text{Expr} \rightarrow \cdot \text{Term}, \text{EOF}], \\
   [\text{Term} \rightarrow \cdot \text{Factor} * \text{Term}, \text{EOF}], [\text{Term} \rightarrow \cdot \text{Factor} * \text{Term}, -], \\
   [\text{Term} \rightarrow \cdot \text{Factor}, \text{EOF}], [\text{Term} \rightarrow \cdot \text{Factor}, -], \\
   [\text{Factor} \rightarrow \cdot \text{id}, \text{EOF}], [\text{Factor} \rightarrow \cdot \text{id}, -], [\text{Factor} \rightarrow \cdot \text{id}, *] \} \]

\[ s_1 \leftarrow \text{goto}(s_0, \text{Expr}) \]
\[ \{ [\text{Goal} \rightarrow \text{Expr} \cdot, \text{EOF}] \} \]

\[ s_2 \leftarrow \text{goto}(s_0, \text{Term}) \]
\[ \{ [\text{Expr} \rightarrow \text{Term} \cdot \cdot \cdot \text{Expr}, \text{EOF}], [\text{Expr} \rightarrow \text{Term} \cdot \cdot \cdot \text{Term}, \text{EOF}] \} \]

\[ s_3 \leftarrow \text{goto}(s_0, \text{Factor}) \]
\[ \{ [\text{Term} \rightarrow \text{Factor} \cdot \cdot \cdot \text{Factor} \cdot \text{Term}, \text{EOF}], [\text{Term} \rightarrow \text{Factor} \cdot \cdot \cdot \text{Factor} \cdot \text{Term}, -], \\
   [\text{Term} \rightarrow \text{Factor} \cdot \cdot \cdot \text{Factor}, \text{EOF}], [\text{Term} \rightarrow \text{Factor} \cdot \cdot \cdot \text{Factor}, -] \} \]

Items in \textbf{black} are core items, generated by moving the placeholder.
Items in \textbf{gray} are added by \text{closure}().
Simplified Expression Grammar

\[ s_4 \leftarrow \text{goto}(s_0, \text{id}) \]
\[ \{ [\text{Factor} \rightarrow \text{id} \cdot, \text{EOF}], [\text{Factor} \rightarrow \text{id} \cdot, -], [\text{Factor} \rightarrow \text{id} \cdot, *] \} \]

\[ s_5 \leftarrow \text{goto}(s_2, -) \]
\[ \{ [\text{Expr} \rightarrow \text{Term} - \cdot \text{Expr} , \text{EOF}], [\text{Expr} \rightarrow \cdot \text{Term} - \text{Expr} , \text{EOF}], [\text{Expr} \rightarrow \cdot \text{Term} , \text{EOF}], [\text{Term} \rightarrow \cdot \text{Factor} * \text{Term} , -], [\text{Term} \rightarrow \cdot \text{Factor} , -], [\text{Term} \rightarrow \cdot \text{Factor} * \text{Term} , \text{EOF}], [\text{Term} \rightarrow \cdot \text{Factor} , \text{EOF}], [\text{Factor} \rightarrow \cdot \text{id} , *], [\text{Factor} \rightarrow \cdot \text{id} , -], [\text{Factor} \rightarrow \cdot \text{id} , \text{EOF}] \} \]

\[ s_6 \leftarrow \text{goto}(s_3, *) \]
\[ \{ [\text{Term} \rightarrow \text{Factor} * \cdot \text{Term} , \text{EOF}], [\text{Term} \rightarrow \text{Factor} * \cdot \text{Term} , -], [\text{Term} \rightarrow \cdot \text{Factor} * \text{Term} , \text{EOF}], [\text{Term} \rightarrow \cdot \text{Factor} * \text{Term} , -], [\text{Term} \rightarrow \cdot \text{Factor} , \text{EOF}], [\text{Term} \rightarrow \cdot \text{Factor} , -], [\text{Factor} \rightarrow \cdot \text{id} , \text{EOF}], [\text{Factor} \rightarrow \cdot \text{id} , -], [\text{Factor} \rightarrow \cdot \text{id} , *] \} \]

Items in black are core items, generated by moving the placeholder.
Items in gray are added by closure().
Simplified Expression Grammar

\[ s_7 \leftarrow \text{goto}(s_5, \text{Expr}) \]
\{ [\text{Expr} \rightarrow \text{Term} - \text{Expr} \bullet, \text{EOF}] \}

\text{goto}(s_5, \text{Term}) \text{ recreates } s_2

\text{goto}(s_5, \text{Factor}) \text{ recreates } s_3

\text{goto}(s_5, \text{id}) \text{ recreates } s_4

\[ s_8 \leftarrow \text{goto}(s_6, \text{Term}) \]
\{ [\text{Term} \rightarrow \text{Factor} \ast \text{Term} \bullet, \text{EOF}], [\text{Term} \rightarrow \text{Factor} \ast \text{Term} \bullet, -] \}

\text{goto}(s_6, \text{Term}) \text{ recreates } s_3

\text{goto}(s_6, \text{id}) \text{ recreates } s_4

The next iteration creates no new sets.

Items in black are core items, generated by moving the placeholder. Items in gray are added by closure().
## Simplified Expression Grammar

### The Goto Relationship

*(recorded during the construction)*

<table>
<thead>
<tr>
<th>State</th>
<th>Expr</th>
<th>Term</th>
<th>Factor</th>
<th>-</th>
<th>*</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>$s_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>$s_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>$s_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_5$</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>$s_6$</td>
<td></td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>$s_7$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_8$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

COMP 412, Fall 2017
The algorithm produces the following tables

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td></td>
</tr>
<tr>
<td>$s_0$</td>
<td>s 4</td>
<td></td>
</tr>
<tr>
<td>$s_1$</td>
<td></td>
<td>acc</td>
</tr>
<tr>
<td>$s_2$</td>
<td>s 5</td>
<td>r 3</td>
</tr>
<tr>
<td>$s_3$</td>
<td>r 5</td>
<td>s 6</td>
</tr>
<tr>
<td>$s_4$</td>
<td>r 6</td>
<td>r 6</td>
</tr>
<tr>
<td>$s_5$</td>
<td>s 4</td>
<td></td>
</tr>
<tr>
<td>$s_6$</td>
<td>s 4</td>
<td></td>
</tr>
<tr>
<td>$s_7$</td>
<td></td>
<td>r 2</td>
</tr>
<tr>
<td>$s_8$</td>
<td>r 4</td>
<td>r 4</td>
</tr>
</tbody>
</table>
Brief Commercial: Why Are We Doing This?

The goal of this exercise is to automate construction of parsers

- Compiler writer provides a **CFG** written in modified **BNF**
- Tools provide an efficient and correct parser
  - *One that works well with an automatically generated scanner*
- **LR** parser generators accept the largest class of grammars that are deterministically parsable, and they are highly efficient
  - *Generated parsers are preferable to hand-coded ones for large grammars*
Three classic options:

- Combine terminals such as number & identifier, + & -, * & /
  - Directly removes a column, may remove a row
  - For expression grammar, 198 (vs. 384) table entries

- Combine rows or columns
  - Implement identical rows once & remap states
  - Requires extra indirection on each lookup
  - Use separate mappings for ACTION & GOTO

- Use another construction algorithm
  - Both LALR(1) and SLR(1) produce smaller tables
    - LALR(1) represents each state with its “core” items
    - SLR(1) uses LR(0) items and the FOLLOW set
  - Implementations are readily available
Shrinking the Grammar

The Classic Expression Grammar

0  \textbf{Goal} \rightarrow \textit{Expr}
1  \textit{Expr} \rightarrow \textit{Expr} + \textit{Term}
2  \mid \textit{Expr} - \textit{Term}
3  \mid \textit{Term}
4  \textbf{Term} \rightarrow \textit{Term} * \textit{Factor}
5  \mid \textit{Term} / \textit{Factor}
6  \mid \textit{Factor}
7  \textbf{Factor} \rightarrow \{ \textit{Expr} \}
8  \mid \text{number}
9  \mid \text{id}

Canonical construction produces 32 states

\begin{itemize}
\item 32 \times (9 + 3) = 384 ACTION/GOTO entries
\item Large table, but still just 1.5kb
\end{itemize}

FIGURE 3.31  Action Table for the Classic Expression Grammar.
Shrinking the Grammar

We can combine some of the syntactically equivalent symbols

- Combine + and − into AddSub
- Combine * and / into MulDiv
- Combine identifier and number into Val

The “Reduced” Expression Grammar

This grammar has

- Fewer terminals
- Fewer productions

Which leads to

- Fewer columns in ACTION
- Fewer states, which leads to fewer rows in both tables
Shrinking the Grammar

The Resulting Tables

- 22 states
- 22 * (6 + 3) = 198 ACTION/GOTO entries
- 48.4% reduction \( \frac{(384 - 198)}{384} \)
- Builds (essentially) the same parse tree

### ACTION Table

<table>
<thead>
<tr>
<th></th>
<th>eof</th>
<th>addsub</th>
<th>muldiv</th>
<th>(</th>
<th>)</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s 4</td>
<td>s 5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s 6</td>
<td>r 3</td>
<td>r 3</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r 5</td>
<td>r 5</td>
<td>r 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s 11</td>
<td>s 12</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r 7</td>
<td>r 7</td>
<td>r 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s 4</td>
<td>s 5</td>
<td>13</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s 4</td>
<td>s 5</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s 15</td>
<td>s 16</td>
<td>18</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r 3</td>
<td>s 17</td>
<td>r 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r 5</td>
<td>r 5</td>
<td>r 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>s 11</td>
<td>s 12</td>
<td>19</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>r 7</td>
<td>r 7</td>
<td>r 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>r 2</td>
<td>r 2</td>
<td>s 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>r 4</td>
<td>r 4</td>
<td>r 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>s 11</td>
<td>s 12</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>r 6</td>
<td>r 6</td>
<td>r 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>s 15</td>
<td>s 21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>r 2</td>
<td>s 17</td>
<td>r 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>r 4</td>
<td>r 4</td>
<td>r 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>r 6</td>
<td>r 6</td>
<td>r 6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### GOTO Table

<table>
<thead>
<tr>
<th>Expr</th>
<th>Term</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) **Action and Goto Tables for the Reduced Expression Grammar**

*FIGURE 3.33* The Reduced Expression Grammar and its Tables.
Shrinking the ACTION and GOTO Tables

Three classic options:

- Combine terminals such as number & identifier, + & -, * & /
  - Directly removes a column, may remove a row
  - For expression grammar, 198 (vs. 384) table entries

- Combine rows or columns
  - Implement identical rows once & remap states
  - Requires extra indirection on each lookup
  - Use separate mappings for ACTION & GOTO

- Use another construction algorithm
  - Both LALR(1) and SLR(1) produce smaller tables
    - LALR(1) represents each state with its “core” items
    - SLR(1) uses LR(0) items and the FOLLOW set
  - Implementations are readily available

left-recursive expression grammar with precedence, see §3.6.2 in EAC

classic space-time tradeoff

Fewer grammars, same languages

§ 3.6.2 in EAC2e

COMP 412, Fall 2017
LR($k$) versus LL($k$)

Finding the next step in a derivation

LR($k$) $\Rightarrow$ Each reduction in the parse is detectable with
→ the complete left context,
→ the reducible phrase, itself, and
→ the $k$ terminal symbols to its right

LL($k$) $\Rightarrow$ Parser must select the expansion based on
→ The complete left context
→ The next $k$ terminals

Thus, LR($k$) examines more context

The question is, do languages fall in the gap between LR($k$) and LL($k$)?
The following **LR(1)** grammar has no **LL(1)** counterpart

- The Canonical Collection has 18 sets of LR(1) Items
  - It is not a simple grammar
  - It is, however, LR(1)

- It requires an arbitrary lookahead to choose between \( A \) & \( B \)
- An **LR(1)** parser can carry the left context (the ‘(‘ s) until it sees \( a \) or \( b \)
- The table construction will handle it
- In contrast, an **LL(1)** parser cannot decide whether to expand \( Goal \) by \( A \) or \( B \)
  
  → *No amount of massaging the grammar and no amount of lookahead will resolve this problem*

---

More precisely, the language described by this **LR(1)** grammar cannot be described with an **LL(1)** grammar. In fact, the language has no **LL(\( k \))** grammar, for finite \( k \).
# ACTION & GOTO Tables for Waite’s Example

<table>
<thead>
<tr>
<th>State</th>
<th>EOF</th>
<th>(</th>
<th>)</th>
<th>a</th>
<th>}</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td></td>
<td>s 4</td>
<td>s 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁</td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₂</td>
<td>r 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₃</td>
<td>r 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₄</td>
<td>s 8</td>
<td>s 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₅</td>
<td>r 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₆</td>
<td></td>
<td>s 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₇</td>
<td></td>
<td>s 11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₈</td>
<td>s 8</td>
<td>s 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₉</td>
<td></td>
<td>r 5</td>
<td>r 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₀</td>
<td>r 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₁</td>
<td>r 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₂</td>
<td></td>
<td>s 14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₃</td>
<td></td>
<td>s 15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₅</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>s₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₃</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₄</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>s₅</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₆</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₇</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₈</td>
<td>12</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>s₉</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₃</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₄</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₅</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Initial State:** s₀
- **Start Rule:**
  - 0: $\text{Start} \rightarrow A$
  - 1: $A \rightarrow \{ A \}$
  - 2: $B \rightarrow \{ B \}$
  - 3: $\vdash a$
  - 4: $\vdash a$

21
LR($k$) versus LL($k$)

Other Non-LL Grammars


Example from Lewis, Rosenkrantz, & Stearns book, “Compiler Design Theory,” (1976), Figure 13.1

This grammar is actually LR(0)
LR(k) versus LL(k)

Finding the next step in a derivation

LR(k) ⇒ Each reduction in the parse is detectable with
→ the complete left context,
→ the reducible phrase, itself, and
→ the k terminal symbols to its right

LL(k) ⇒ Parser must select the expansion based on
→ The complete left context
→ The next k terminals

Thus, LR(k) examines more context

“... in practice, programming languages do not actually seem to fall in the gap between LL(1) languages and deterministic languages”

Left Recursion versus Right Recursion

- **Right recursion**
  - Required for termination in top-down parsers
  - Uses (on average) more stack space
  - Naïve right recursion produces right-associativity

- **Left recursion**
  - Works fine in bottom-up parsers
  - Limits required stack space
  - Naïve left recursion produces left-associativity

- **Rule of thumb**
  - Left recursion for bottom-up parsers
  - Right recursion for top-down parsers
Left Recursion versus Right Recursion

A real example, from the lab 1 ILOC simulator’s front end

The simulator was built by two of my successful Ph.D.s

• It is actually a more complex piece of software than you might guess
• The front end is an LR(1) parser, generated by Bison
• The grammar contained the following productions:

\[
\begin{align*}
\text{instruction\_list} & : \quad \text{instruction} \\
& \mid \quad \text{label\_def} \ \text{instruction} \\
& \mid \quad \text{instruction} \ \text{instruction\_list} \\
& \mid \quad \text{label\_def} \ \text{instruction} \ \text{instruction\_list} \\
\end{align*}
\]

When my colleague first ran the timing blocks through the simulator, it exploded with the error message “memory exhausted”.

⇒ What happened?
Left Recursion versus Right Recursion

A real example, from the lab 1 simulator’s front end

```
instruction_list : instruction
  | label_def instruction
  | instruction instruction_list
  | label_def instruction instruction_list
```

• The parse stack overflowed as it tried to instantiate the instruction_list
Left Recursion versus Right Recursion

A real example, from the lab 1 simulator’s front end

```
instruction_list  :  instruction
|  label_def instruction
|  instruction  instruction_list
|  label_def  instruction  instruction_list
```

• The parse stack overflowed as it tried to instantiate the `instruction_list`
• The fix was easy

```
instruction_list  :  instruction
|  label_def instruction
|  instruction_list  instruction
|  instruction_list  label_def  instruction
```

• This grammar has (small) bounded stack space & (thus) scales well

See pp. 145-146 in EaC2e
Error Detection and Recovery

Error Detection

• Recursive descent
  – Parser takes the last else clause in a routine
  – Compiler writer can code almost any arbitrary action

• Table-driven LL(1)
  – In state $s_i$ facing word $x$, entry is an error
  – Report the error, valid entries in row for $s_i$ encode possibilities

• Table-driven LR(1)
  – In state $s_i$ facing word $x$, entry is an error
  – Report the error, shift states in row encode possibilities
  – Can precompute better messages from LR(1) items
Error Detection and Recovery

Error Recovery

• Table-driven LL(1)
  – Treat as missing token, e.g. ‘)’, ⇒ expand by desired symbol
  – Treat as extra token, e.g., ‘x + y’, ⇒ pop stack and move ahead

• Table-driven LR(1)
  – Treat as missing token, e.g. ‘)’, ⇒ shift the token
  – Treat as extra token, e.g., ‘x + y’, ⇒ don’t shift the token
One common strategy is “hard token” recovery

Skip ahead in input until we find some “hard” token, e.g. ‘;’

- ‘;’ separates statements, makes a logical break in the parse
- Resynchronize state, stack, and input to point after hard token
  \[\text{\(\rightarrow\)}\ \text{LL(1): \text{pop stack until we find a row with entry for ‘;}\)}\]
  \[\text{\(\rightarrow\)}\ \text{LR(1): \text{pop stack until we find a state with a reduction on ‘;}\)}\]
- Does not correct the input, rather it allows parse to proceed

\[
\begin{align*}
\text{NT} & \leftarrow \text{pop()} \\
\text{repeat until Table}[\text{NT},’;’] & \neq \text{error} \\
& \quad \text{NT} \leftarrow \text{pop()} \\
\text{token} & \leftarrow \text{NextToken()} \\
\text{repeat until token} & = ‘;’ \\
& \quad \text{token} \leftarrow \text{NextToken()}
\end{align*}
\]

Resynchronizing an \textbf{LL(1)} parser

\[
\begin{align*}
\text{repeat until token} & = ‘;’ \\
& \quad \text{shift token} \\
& \quad \text{shift \(s_e\)} \\
& \quad \text{token} \leftarrow \text{NextToken()} \\
& \quad \text{reduce by error production} \\
& \quad // \text{pops all that state off stack}
\end{align*}
\]

Resynchronizing an \textbf{LR(1)} parser
Hierarchy of Context-Free Languages

- Context-free languages
  - Deterministic languages \((LR(k))\)
    - \(LL(k)\) languages
    - \(LL(1)\) languages
    - Simple precedence languages
    - Operator precedence languages

LR\((k) \equiv LR(1)\)

The inclusion hierarchy for context-free languages
Hierarchy of Context-Free Grammars

- Operator precedence includes some ambiguous grammars
- Note sub-categories of LR(1)
- LL(1) is a subset of SLR(1)

The inclusion hierarchy for context-free grammars