Syntax Analysis, VII

One more LR(1) example, plus some final thoughts

Comp 412
Computing Closures

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

- Any item *A → β • Bδ, a* where *B ∈ NT* implies *B → • τ, x* for each production that has *B* on the *lhs*, and each *x ∈ FIRST(δa)*
- Since *βBδ* is valid, any way to derive *βBδ* is valid, too

**The Algorithm**

```plaintext
Closure( s )
while ( s is still changing )
  ∀ items [A → β • Bδ, a] ∈ s
    lookahead ← FIRST(δa) // δ might be ε
  ∀ productions B → τ ∈ P
    ∀ b ∈ lookahead
      if [B → • τ, b] ∉ s
        then s ← s ∪ { [B → • τ, b] }
```

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

Lookahead is calculated on the item being expanded. See note on p. 128
Computing Gotos

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

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

The Algorithm

$$Goto( s, X )$$

$\text{new} \leftarrow \emptyset$

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

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

$\text{return } Closure( \text{new} )$

- $Goto( )$ 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

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

Slightly different formulation than in last lecture. Simple changes eliminate lots of duplicate work.

See updated slide in Lecture 15.
The Table Construction Algorithm

∀ set \( S_x \in S \)
∀ item \( i \in S_x \)

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

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

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

∀ \( n \in NT \)

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

Many items generate no table entry

→ Placeholder before a \( NT \) does not generate an \( \text{ACTION} \) table entry

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

Simplified, right recursive expression grammar

<table>
<thead>
<tr>
<th></th>
<th>SYMBOL</th>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal</td>
<td>{ id  }</td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
<td>{ id  }</td>
</tr>
<tr>
<td>2</td>
<td>Term</td>
<td>{ id  }</td>
</tr>
<tr>
<td>3</td>
<td>Term</td>
<td>{ id  }</td>
</tr>
<tr>
<td>4</td>
<td>Factor</td>
<td>{ id  }</td>
</tr>
<tr>
<td>5</td>
<td>Factor</td>
<td>{ id  }</td>
</tr>
</tbody>
</table>
# Simplified Expression Grammar

## Initialization Step

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

\[
\{ [\text{Goal} \rightarrow \bullet \text{Expr}, \text{EOF}], \\
[\text{Expr} \rightarrow \bullet \text{Term} - \text{Expr}, \text{EOF}], [\text{Expr} \rightarrow \bullet \text{Term}, \text{EOF}], \\
[\text{Term} \rightarrow \bullet \text{Factor} \ast \text{Term}, \text{EOF}], [\text{Term} \rightarrow \bullet \text{Factor} \ast \text{Term}, -], \\
[\text{Term} \rightarrow \bullet \text{Factor}, \text{EOF}], [\text{Term} \rightarrow \bullet \text{Factor}, -], \\
[\text{Factor} \rightarrow \bullet \text{id}, \text{EOF}], [\text{Factor} \rightarrow \bullet \text{id}, -], [\text{Factor} \rightarrow \bullet \text{id}, \ast] \} \}
\]

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

---

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal</td>
<td>( \rightarrow )</td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
<td>( \rightarrow )</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>( \mid )</td>
</tr>
<tr>
<td>3</td>
<td>Term</td>
<td>( \rightarrow )</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>( \mid )</td>
</tr>
<tr>
<td>5</td>
<td>Factor</td>
<td>( \rightarrow )</td>
</tr>
</tbody>
</table>

**Item in black is the initial item.**

**Items in gray are added by closure().**
Simplified Expression Grammar

Iteration 1

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

Goal, * , & - generate empty sets

Iteration 2

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

Goal, Expr, Term, Factor, & \text{id} generate empty sets

Iteration 3

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

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

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

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

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

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

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

The Details

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

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

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

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

The Details

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

\[ s_8 \leftarrow \text{goto}(s_6, \text{Term}) \]
\[ \{ [\text{Term} \rightarrow \text{Factor} \ast \text{Term} \cdot, \text{EOF}], [\text{Term} \rightarrow \text{Factor} \ast \text{Term} \cdot, \cdot] \} \]
\text{goto}(s_6, \text{Term}) \text{ recreates } s_3
\text{goto}(s_6, \text{id}) \text{ recreates } s_4

\text{The next iteration creates no new sets.}

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

#### The Goto Relationship

*(recorded during the construction)*

<table>
<thead>
<tr>
<th>State</th>
<th>—</th>
<th>*</th>
<th>id</th>
<th>Expr</th>
<th>Term</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td></td>
<td></td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</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>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$s_6$</td>
<td></td>
<td></td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>3</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>

*Into shifts in the ACTION Table*

*Into the GOTO Table*
The algorithm produces the following tables

<table>
<thead>
<tr>
<th>State</th>
<th>id</th>
<th>-</th>
<th>*</th>
<th>EOF</th>
<th>GOTO</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>s 4</td>
<td></td>
<td></td>
<td>EOF</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$s_1$</td>
<td></td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>S 5</td>
<td>R 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td>R 5</td>
<td>S 6</td>
<td>R 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_4$</td>
<td>R 6</td>
<td>R 6</td>
<td>R 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_5$</td>
<td>S 4</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>$s_6$</td>
<td>S 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>$s_7$</td>
<td></td>
<td></td>
<td></td>
<td>R 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_8$</td>
<td>R 4</td>
<td>R 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
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
Shrinking the Grammar

The Classic Expression Grammar

<table>
<thead>
<tr>
<th>State</th>
<th>Action Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>acc s 7 s 8</td>
</tr>
<tr>
<td>1</td>
<td>r 4 r 4 r 4 s 9 s 10</td>
</tr>
<tr>
<td>2</td>
<td>r 7 r 7 r 7 r 7</td>
</tr>
<tr>
<td>3</td>
<td>r 9 r 9 r 9 r 9</td>
</tr>
<tr>
<td>4</td>
<td>r 10 r 10 r 10 r 10</td>
</tr>
<tr>
<td>5</td>
<td>s 4 s 5 s 6</td>
</tr>
<tr>
<td>6</td>
<td>s 4 s 5 s 6</td>
</tr>
<tr>
<td>7</td>
<td>s 4 s 5 s 6</td>
</tr>
<tr>
<td>8</td>
<td>s 4 s 5 s 6</td>
</tr>
<tr>
<td>9</td>
<td>s 4 s 5 s 6</td>
</tr>
<tr>
<td>10</td>
<td>s 4 s 5 s 6</td>
</tr>
<tr>
<td>11</td>
<td>r 2 r 2 r 2 s 9 s 10</td>
</tr>
<tr>
<td>12</td>
<td>r 3 r 3 r 3 s 9 s 10</td>
</tr>
<tr>
<td>13</td>
<td>r 5 r 5 r 5 r 5 r 5</td>
</tr>
<tr>
<td>14</td>
<td>r 6 r 6 r 6 r 6 r 6</td>
</tr>
<tr>
<td>15</td>
<td>s 21 s 22 s 23</td>
</tr>
<tr>
<td>16</td>
<td>s 24 s 25 r 4</td>
</tr>
<tr>
<td>17</td>
<td>r 10 r 10 r 10 r 10</td>
</tr>
<tr>
<td>18</td>
<td>r 2 r 2 r 2 s 9 s 10</td>
</tr>
<tr>
<td>19</td>
<td>r 3 r 3 r 3 s 9 s 10</td>
</tr>
<tr>
<td>20</td>
<td>r 5 r 5 r 5 r 5 r 5</td>
</tr>
<tr>
<td>21</td>
<td>r 6 r 6 r 6 r 6 r 6</td>
</tr>
<tr>
<td>22</td>
<td>s 14 s 15 s 16</td>
</tr>
<tr>
<td>23</td>
<td>s 14 s 15 s 16</td>
</tr>
<tr>
<td>24</td>
<td>s 14 s 15 s 16</td>
</tr>
<tr>
<td>25</td>
<td>s 14 s 15 s 16</td>
</tr>
<tr>
<td>26</td>
<td>s 21 s 22 s 31</td>
</tr>
<tr>
<td>27</td>
<td>r 2 r 2 s 24 s 25 r 2</td>
</tr>
<tr>
<td>28</td>
<td>r 3 r 3 s 24 s 25 r 3</td>
</tr>
<tr>
<td>29</td>
<td>r 5 r 5 r 5 r 5 r 5</td>
</tr>
<tr>
<td>30</td>
<td>r 6 r 6 r 6 r 6 r 6</td>
</tr>
<tr>
<td>31</td>
<td>r 8 r 8 r 8 r 8 r 8</td>
</tr>
</tbody>
</table>

Canonical construction produces 32 states

- 32 x (9 + 3) = 384 ACTION/GOTO entries
- Large table, but still just 1.5kb
We can combine some of the syntactically equivalent symbols

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

This grammar has
- Fewer terminals
- Fewer productions

Which leads to
- Fewer columns in ACTION
- Fewer states, which leads to fewer rows in both tables

The “Reduced” Expression Grammar
Shrinking the Grammar

The Resulting Tables

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

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

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

\[\text{FIGURE 3.33} \quad \text{The Reduced Expression Grammar and its Tables.}\]
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

§ 3.6.2 in EaC2e
Shrinking The Action and Goto Tables

The parser generator can combine identical columns & identical rows

```plaintext
for i ← 0 to MaxCol
    MapTo[i] ← i

for i ← 0 to MaxCol
    if MapTo[i] = i then
        for j ← i to MaxCol
            if MapTo[j] = j then
                same ← true
            for k ← 0 to MaxRow
                if Table[i,k] = Table[j,k] then
                    same ← false
                    break
            if same = true then
                MapTo[j] ← i
```

Basic Table Compression Algorithm

- Same algorithm applies to ACTION & GOTO as the DFA table
- Adds a level of indirection, as in a DFA’s character classifier
  - More complex table encoding
  - Tradeoff loads vs. table size
- Significant table-size reductions without grammar changes
- Can improve efficiency by using a quick inequality test
  - pop count of non-error states
  - more complex signatures (hash)
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 in construction and the FOLLOW set in table-filling algorithm
  – Implementations are readily available

Option at Parser-Generator Design-time

§ 3.6.2 in EaC2e
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

"3.6.2 in EAC2e"
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)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>Goal</strong></td>
<td>→ S</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>→ A</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>→ ( A )</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>→ ( B ]</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>b</td>
</tr>
</tbody>
</table>

- 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>States</th>
<th>Action</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>EOF</td>
<td>s 4, s 5</td>
</tr>
<tr>
<td>$s_1$</td>
<td>acc</td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>r 2</td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td>r 3</td>
<td></td>
</tr>
<tr>
<td>$s_4$</td>
<td>s 8, s 9</td>
<td></td>
</tr>
<tr>
<td>$s_5$</td>
<td>r 5</td>
<td></td>
</tr>
<tr>
<td>$s_6$</td>
<td></td>
<td>s 10</td>
</tr>
<tr>
<td>$s_7$</td>
<td></td>
<td>s 11</td>
</tr>
<tr>
<td>$s_8$</td>
<td>s 8, s 9</td>
<td></td>
</tr>
<tr>
<td>$s_9$</td>
<td></td>
<td>r 5, r 7</td>
</tr>
<tr>
<td>$s_{10}$</td>
<td>r 4</td>
<td></td>
</tr>
<tr>
<td>$s_{11}$</td>
<td>r 6</td>
<td></td>
</tr>
<tr>
<td>$s_{12}$</td>
<td></td>
<td>s 14</td>
</tr>
<tr>
<td>$s_{13}$</td>
<td></td>
<td>s 15</td>
</tr>
<tr>
<td>$s_{14}$</td>
<td></td>
<td>r 4</td>
</tr>
<tr>
<td>$s_{15}$</td>
<td></td>
<td>r 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>States</th>
<th>Action</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td></td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>$s_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_4$</td>
<td></td>
<td>6, 7</td>
</tr>
<tr>
<td>$s_5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_6$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_7$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_8$</td>
<td></td>
<td>12, 13</td>
</tr>
<tr>
<td>$s_9$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_{10}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_{11}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_{12}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_{13}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_{14}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_{15}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example Transitions:
- **Start** → A
- A → ( A )
- a
- B → ( B )
- a
LR($k$) versus LL($k$)

The Bottom Line

“... 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 front end of the lab 2 ILOC simulator

The simulator was built by two of my more 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:

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

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

\[
\text{instruction\_list} : \text{instruction} \\
| \text{label\_def instruction} \\
| \text{instruction instruction\_list} \\
| \text{label\_def instruction instruction\_list}
\]

- 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

\[
\text{instruction\_list} : \text{instruction} \\
| \text{label\_def instruction} \\
| \text{instruction \_list} \\
| \text{label\_def instruction instruction\_list} \\
\]

• The parse stack overflowed as it tried to instantiate the instruction\_list
• The fix was easy

\[
\text{instruction\_list} : \text{instruction} \\
| \text{label\_def instruction} \\
| \text{instruction\_list instruction} \\
| \text{instruction\_list label\_def instruction} \\
\]

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

See pp. 145-146 in EaC2e
Left Recursion versus Right Recursion

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

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

• But, the tool wants the right-associative data structure
  – List traversal should find the first element of the list before the second
  – \((elt_1, (elt_2, (elt_1, ... )))\)

• The tool needs to build the right-associative list from the left-associative grammar
  – We will come back to this idea in ad-hoc syntax directed translation
Hierarchy of Context-Free Languages

Context-free languages

\[ \text{Deterministic languages} \quad (\text{LR}(k)) \]

\[ \text{LL}(k) \text{ languages} \]
\[ \text{LL}(1) \text{ languages} \]

\[ \text{Simple precedence languages} \]

\[ \text{Operator precedence languages} \]

LR\((k) \equiv \text{LR}(1)\)

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

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

The inclusion hierarchy for context-free grammars