Syntax Analysis, VII
The Canonical LR(1) Table Construction

Comp 412

Copyright 2019, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 3 in EaC2e
Table-Driven LR Parsers

A table-driven LR(1) parser is a bottom-up shift-reduce parser

Back to the Meta Issue

- The compiler writer creates a grammar at **design time**
- The parser generator builds **ACTION** and **GOTO** tables at **build time**
- The compiler uses those tables to parse at **compile time**
Building **LR(1)** Tables

**How do we generate the ACTION and GOTO tables?**

- Use the grammar to build a model of the control automaton
- Encode actions & transitions in ACTION & GOTO tables
- If construction succeeds, the grammar is **LR(1)**
  - “Succeeds” means defines each table entry uniquely

**The Big Picture**

- Model the state of the parser with **LR(1)** items
- Use two functions $\text{goto}(s, X)$ and $\text{closure}(s)$
  - $\text{goto}()$ is analogous to $\text{move}()$ in the subset construction
  - Given a partial state, $\text{closure}()$ adds all the items implied by the partial state
- Build up the states and transition functions of the automaton
- Use this information to fill in the ACTION and GOTO tables
LR(1) Table Construction

To understand the algorithms, we need to understand the data structure that they use: sets of LR(1) items

An LR(1) item is a pair \([P, \delta]\), where
- \(P\) is a production \(A \rightarrow \beta\) with a \(\bullet\) at some position in the RHS
- \(\delta\) is a single symbol lookahead \((\text{symbol} = \text{word or EOF})\)

The LR(1) table construction algorithm models a configuration of the parser, or a state of the parser, as a set of LR(1) items

- Each item represents a specific production by which the parser might reduce in the future
- Each item’s placeholder represents the progress toward that reduction

The construction ties builds up a set of configurations that model all the possible behaviors implied by the grammar
LR(1) Items

An LR(1) item is a pair \([P, \delta]\), where

- \(P\) is a production \(A \rightarrow \beta\) with a \(\bullet\) at some position in the RHS
- \(\delta\) is a single symbol lookahead \((symbol \equiv \text{word or EOF})\)

The \(\bullet\) in an item indicates the position of the top of the stack.

\([A \rightarrow \bullet \beta \gamma, a]\) means that the input seen so far is consistent with the use of \(A \rightarrow \beta \gamma\) immediately after the symbol on top of the stack.

We call an item like this a **possibility**.

\([A \rightarrow \beta \bullet \gamma, a]\) means that the input seen so far is consistent with the use of \(A \rightarrow \beta \gamma\) at this point in the parse, *and* that the parser has already recognized \(\beta\) (that is, \(\beta\) is on top of the stack).

We call an item like this a **partially complete** item.

\([A \rightarrow \beta \gamma \bullet, a]\) means that the parser has seen \(\beta \gamma\), *and* that a lookahead symbol of \(a\) is consistent with reducing to \(A\).

This item is **complete**.

LR(k) parsers rely on items with a lookahead of \(\leq k\) symbols. That leads to LR(k) items, with correspondingly longer \(\delta\).
LR(1) Items

The production \( A \rightarrow B_1 B_2 B_3 \) with lookahead \( a \), can give rise to 4 items

\[
[A \rightarrow \bullet B_1 B_2 B_3, a], [A \rightarrow B_1 \bullet B_2 B_3, a], [A \rightarrow B_1 B_2 \bullet B_3, a], \text{ & } [A \rightarrow B_1 B_2 B_3 \bullet, a]
\]

The set of LR(1) items for a grammar is \textit{finite}.

What’s the point of all these lookahead symbols?

• Carry them along to help choose the correct reduction

• Lookaheads are bookkeeping, unless item has \( \bullet \) at right end
  – Has no direct use in \( [A \rightarrow \beta \bullet \gamma, a] \)
  – In \( [A \rightarrow \beta \bullet, a] \), a lookahead of \( a \) implies a reduction by \( A \rightarrow \beta \)
  – For \( \{ [A \rightarrow \beta \bullet, a], [B \rightarrow \gamma \bullet \delta, b] \} \), \( a \Rightarrow reduce \) to \( A \); \( \text{FIRST}(\delta) \Rightarrow shift \)

\( \Rightarrow \) Limited right context is enough to pick the actions
**LR(1) Items**

The set of LR(1) items for a grammar is finite

Consider the *SheepNoise* grammar

<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>SheepNoise</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal</td>
<td>SheepNoise</td>
</tr>
<tr>
<td>1</td>
<td>SheepNoise</td>
<td>SheepNoise baa</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>baa</td>
</tr>
</tbody>
</table>

It gives rise to 7 LR(1) items

- \([\text{Goal } \rightarrow \bullet \text{ SheepNoise}]\)
- \([\text{Goal } \rightarrow \bullet \text{ SheepNoise } \text{ baa}]\)
- \([\text{Goal } \rightarrow \bullet \text{ baa}]\)
- \([\text{Goal } \rightarrow \text{ SheepNoise } \bullet]\)
- \([\text{Goal } \rightarrow \text{ SheepNoise } \text{ baa } \bullet]\)
- \([\text{Goal } \rightarrow \text{ SheepNoise } \bullet \text{ baa}]\)
- \([\text{Goal } \rightarrow \text{ baa } \bullet]\)

7 is finite. Why does this matter?

A **configuration** of the LR(1) parser is represented as a set of LR(1) items—the collection of productions by which the parser might reduce in the future, given the context seen so far, with the placeholder showing the progress in recognizing each possible handle.

The number configurations (subsets of the set of LR(1) items) is, again, finite.
LR(1) Items: Why should you know this stuff?

Debugging a grammar

- When you build an LR(1) parser, it is possible (likely) that the initial grammar is not LR(1)
- The tools will provide you with debugging output
- To the right is a sample of bison’s output for the if-then-else grammar

```
state 10

  4 stmt : IF EXPR THEN stmt .
  5    | IF EXPR THEN stmt . ELSE stmt

ELSE    shift, and go to state 11

ELSE    [reduce using rule 4 (stmt)]
$default reduce using rule 4 (stmt)
```

The state is described by its LR(1) items

```
goal : stmt_list
stmt_list : stmt_list stmt
            | stmt
stmt : IF EXPR THEN stmt
             | IF EXPR THEN stmt ELSE stmt
             | OTHER
```
LR(1) Table Construction

**High-level overview**

1. Build the Canonical Collection of Sets of LR(1) Items, \( I \)
   
   a. Begin in an appropriate state, \( s_0 \)
      
      - \([S' \rightarrow \cdot S, \text{EOF}]\), along with any equivalent items
      - Derive equivalent items as \( \text{closure}( s_0 ) \)
   
   b. Repeatedly compute, for each \( s_k \), and each \( X \), \( \text{goto}(s_k, X) \)
      
      - If the set is not already in the collection, add it
      - Record all the transitions created by \( \text{goto}( ) \)
      
      This eventually reaches a fixed point

2. Fill in the table from the Canonical Collection of Sets of LR(1) items

---

The sets in the canonical collection form the states of the control automaton.

The construction traces the automaton’s transitions.
LR(1) Table Construction

High-level overview

1. Build the Canonical Collection of Sets of LR(1) Items, \( I \)
   a. Begin in an appropriate state, \( s_0 \)
      - \([S' \rightarrow S, \text{EOF}]\), along with any equivalent items
      - Derive equivalent items as \( \text{closure}(s_0) \)
   b. Repeatedly compute, for each \( s_k \), and each \( X \), \( \text{goto}(s_k, X) \)
      - If the set is not already in the collection, add it
      - Record all the transitions created by \( \text{goto}() \)
      This eventually reaches a fixed point

2. Fill in the table from the Canonical Collection of Sets of LR(1) items

Let’s build the tables for the left-recursive SheepNoise grammar \((S' \text{ is Goal})\)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>Goal</th>
<th>→</th>
<th>SheepNoise</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SheepNoise</td>
<td>→</td>
<td>SheepNoise baa</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>baa</td>
</tr>
</tbody>
</table>
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**

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

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

Lookahead is calculated on the item being expanded, not prod’n being added. See note on p. 128 in *EaC2e*
Computing Closures

Generating Closures is the place where a human is most likely to make a mistake

- With everything going on in the construction, it is easy to lose track of $\delta a$ and the fact that it refers to the item, not the current production

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

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

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

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

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

\[
\text{if } [B \rightarrow \cdot \tau, b] \notin s
\]

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

- The lookahead computation is a great example of why these table constructions should be done by computers, not human beings
Example From SheepNoise

Initial step builds the item \([\text{Goal} \rightarrow \bullet \text{SheepNoise}, \text{EOF}]\) and takes its \textit{Closure}( )

\textit{Closure}( \([\text{Goal} \rightarrow \bullet \text{SheepNoise}, \text{EOF}]\) )

<table>
<thead>
<tr>
<th>Item</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{Goal} \rightarrow \bullet \text{SheepNoise}, \text{EOF}])</td>
<td>Original item</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa}, \text{EOF}])</td>
<td>ITER 1, PR 1, (\delta_a) is \text{EOF}</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{baa}, \text{EOF}])</td>
<td>ITER 1, PR 2, (\delta_a) is \text{EOF}</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa}])</td>
<td>ITER 2, PR 1, (\delta_a) is \text{baa} \text{EOF}</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{baa, baa}])</td>
<td>ITER 2, PR 2, (\delta_a) is \text{baa} \text{EOF}</td>
</tr>
</tbody>
</table>

So, \(S_0\) is

\[
\{ \ [\text{Goal} \rightarrow \bullet \text{SheepNoise,EOF}], \ [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa,EOF}], \\
[\text{SheepNoise} \rightarrow \bullet \text{baa,EOF}], \ [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa,baa}], \\
[\text{SheepNoise} \rightarrow \bullet \text{baa,baa}] \ \} 
\]
Computing Gotos

\textit{Goto}(s,x) computes the state that the parser would reach if it recognized an x while in state s

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

The Algorithm

\begin{align*}
\textit{Goto}( s, X ) \\
new &\leftarrow \emptyset \\
\forall \text{ items } [A\rightarrow\beta \times \delta, a] \in s \\
new &\leftarrow new \cup \{ [A\rightarrow\beta X \times \delta, a] \} \\
\text{return } \textit{Closure}( \textit{new} )
\end{align*}

- \textit{Goto}() models a transition in the automaton
- Straightforward computation
- \textit{Goto}() is \textbf{not} a fixed-point method (but it calls \textit{Closure}())

Items introduced by moving the placeholder are called \textbf{core} items.
Items introduced by closure are called \textbf{non-core} items.

\textit{Goto} in this construction is analogous to \textit{Move} in the subset construction.
Example from SheepNoise

Assume that \( S_0 \) is

\[
\{ \begin{array}{l}
[ \text{Goal} \rightarrow \bullet \text{SheepNoise, EOF} ], \\
[ \text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, EOF} ], \\
[ \text{SheepNoise} \rightarrow \bullet \text{baa, EOF} ], \\
[ \text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa} ], \\
[ \text{SheepNoise} \rightarrow \bullet \text{baa, baa} ] \\
\end{array} \}
\]

**Goto( \( S_0, \text{ baa } \))**

- Loop produces

<table>
<thead>
<tr>
<th>Item</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{SheepNoise} \rightarrow \text{ baa } \bullet, \text{ EOF}])</td>
<td>Item 3 in ( s_0 )</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \text{ baa } \bullet, \text{ baa}])</td>
<td>Item 5 in ( s_0 )</td>
</tr>
</tbody>
</table>

- **Closure** adds nothing since \( \bullet \) is at end of \( rhs \) in each item

In the construction, this produces \( s_2 \)

\[
\{ [\text{SheepNoise} \rightarrow \text{ baa } \bullet, \{ \text{EOF}, \text{ baa} \}] \} 
\]

New, but *obvious*, notation for two distinct items

\[
[\text{SheepNoise} \rightarrow \text{ baa } \bullet, \text{ EOF}] \text{ & } [\text{SheepNoise} \rightarrow \text{ baa } \bullet, \text{ baa}] 
\]

<table>
<thead>
<tr>
<th>0</th>
<th>Goal</th>
<th>→</th>
<th>SheepNoise</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SheepNoise</td>
<td>→</td>
<td>SheepNoise baa</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>baa</td>
</tr>
</tbody>
</table>
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

\[
\begin{align*}
s_0 & \leftarrow \text{Closure}( \{ [S' \rightarrow \bullet S, \text{EOF}] \} ) \\
S & \leftarrow \{ s_0 \} \\
k & \leftarrow 1
\end{align*}
\]

while ( $S$ is still changing )

\[
\begin{align*}
\forall s_i \in S \text{ and } \forall x \in ( T \cup NT ) \\
s_k & \leftarrow \text{Goto}(s_i, x) \\
\text{record } s_i \rightarrow s_k \text{ on } x \\
\text{if } s_k \notin S \text{ then } \\
S & \leftarrow S \cup \{ s_k \} \\
k & \leftarrow k + 1
\end{align*}
\]

- 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.
The algorithm is structured to make obvious the fact that it is a monotonic fixed point algorithm.

\[
\begin{align*}
& s_0 \leftarrow \emptyset \\
& \text{for each production } S' \rightarrow \alpha \\
& \quad s_0 \leftarrow s_0 \cup \{ [S' \rightarrow \bullet \alpha, \text{EOF}] \} \\
& s_0 \leftarrow \text{Closure}( \{ [S' \rightarrow \bullet S, \text{EOF}] \} ) \\
& S \leftarrow \{ s_0 \} \\
& \quad \text{while } (S \text{ is still changing}) \\
& \quad \quad \text{for each unmarked set } s_i \text{ in } S \\
& \quad \quad \quad \text{mark } s_i \text{ as processed} \\
& \quad \quad \quad \text{for each } x \text{ following } a \bullet \text{ in } s_i \\
& \quad \quad \quad \quad \text{temp} \leftarrow \text{Goto}(s_i, x) \\
& \quad \quad \quad \quad \text{if } \text{temp} \notin S \text{ then} \\
& \quad \quad \quad \quad \quad S \leftarrow S \cup \{ \text{temp} \} \\
& \quad \quad \quad \quad \text{record transition } s_i \rightarrow \text{temp}
\end{align*}
\]

- It does lots of redundant work
  - Iterate over all \( s_j \in S \)
  - Iterate over all \( x \in (T \cup NT) \)
- In practice, we can reformulate the algorithm to eliminate much of the redundant work
  - Process each \( s_j \in S \) just once
  - Only look at symbols that appear after \( \bullet \)
- Asymptotically, these things do not matter
- Practically, they help a lot
Example from SheepNoise

**Starts with** $S_0$

$S_0 : \{ \text{[Goal} \rightarrow \bullet \text{SheepNoise, EOF]}, \text{[SheepNoise} \rightarrow \bullet \text{SheepNoise baa, EOF]},$

$\text{[SheepNoise} \rightarrow \bullet \text{ baa, EOF]}, \text{[SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa]},$

$\text{[SheepNoise} \rightarrow \bullet \text{ baa, baa]} \} \}

**Iteration 1 computes**

$S_1 = \text{Goto}(S_0, \text{SheepNoise}) =$

$\{ \text{[Goal} \rightarrow \text{SheepNoise} \bullet, \text{EOF]}, \text{[SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{ baa, EOF]},$

$\text{[SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{ baa, baa]} \} \}

$S_2 = \text{Goto}(S_0, \text{ baa}) = \{ \text{[SheepNoise} \rightarrow \text{ baa} \bullet, \text{EOF]},$

$\text{[SheepNoise} \rightarrow \text{ baa} \bullet, \text{ baa]} \} \}

**Iteration 2 computes**

$S_3 = \text{Goto}(S_1, \text{ baa}) = \{ \text{[SheepNoise} \rightarrow \text{SheepNoise} \text{ baa} \bullet, \text{EOF]},$

$\text{[SheepNoise} \rightarrow \text{SheepNoise} \text{ baa} \bullet, \text{ baa]} \} \}

\begin{align*}
0 & \text{ Goal} & \rightarrow & \text{SheepNoise} \\
1 & \text{SheepNoise} & \rightarrow & \text{SheepNoise baa} \\
2 & | & \text{ baa} & \end{align*}
Example from SheepNoise

\[ S_0 : \{ [\text{Goal} \rightarrow \bullet \text{SheepNoise, EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, EOF}], \\
[\text{SheepNoise} \rightarrow \bullet \text{ baa, EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa}], \\
[\text{SheepNoise} \rightarrow \bullet \text{ baa, baa}] \} \]

\[ S_1 = \text{Goto}(S_0, \text{SheepNoise}) = \\
\{ [\text{Goal} \rightarrow \text{SheepNoise} \bullet, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{ baa, EOF}], \\
[\text{SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{ baa, baa}] \} \]

\[ S_2 = \text{Goto}(S_0, \text{ baa}) = \{ [\text{SheepNoise} \rightarrow \text{ baa} \bullet, \text{EOF}], [\text{SheepNoise} \rightarrow \text{ baa} \bullet, \text{ baa}] \} \]

\[ S_3 = \text{Goto}(S_1, \text{ baa}) = \{ [\text{SheepNoise} \rightarrow \text{SheepNoise baa} \bullet, \text{EOF}], \\
[\text{SheepNoise} \rightarrow \text{SheepNoise baa} \bullet, \text{ baa}] \} \]
Filling in the ACTION and GOTO Tables

The Table Construction Algorithm

\( \forall \) set \( S_x \in S \)
\( \forall \) item \( i \in S_x \)
\( \text{if } i \text{ is } [A \rightarrow \beta \bullet a \delta, b] \text{ and } \text{goto}(S_x, a) = S_k, a \in T \)
\( \text{then } \text{ACTION}[x, a] \leftarrow \text{“shift } k \text{”} \)
\( \text{else if } i \text{ is } [S' \rightarrow S \bullet, \text{EOF}] \)
\( \text{then } \text{ACTION}[x, \text{EOF}] \leftarrow \text{“accept”} \)
\( \text{else if } i \text{ is } [A \rightarrow \beta \bullet, a] \)
\( \text{then } \text{ACTION}[x, a] \leftarrow \text{“reduce } A \rightarrow \beta \text{”} \)
\( \forall \ n \in NT \)
\( \text{if } \text{goto}(S_x, n) = S_k \)
\( \text{then } \text{GOTO}[x, n] \leftarrow k \)

Many items generate no table entry

\( \rightarrow \) Placeholder before a \( NT \) does not generate an ACTION table entry
\( \rightarrow \) \textbf{Closure}() instantiates \textbf{FIRST}(X) directly for \( [A \rightarrow \beta \bullet X \delta, a] \)
Example from SheepNoise

\[ S_0 : \{ [Goal \rightarrow \bullet SheepNoise, \textbf{EOF}], [SheepNoise \rightarrow \bullet SheepNoise \textit{ baa}, \textbf{EOF}], [SheepNoise \rightarrow \bullet \textit{ baa}, \textbf{EOF}], [SheepNoise \rightarrow \bullet \textit{ baa}, \textit{ baa}], [SheepNoise \rightarrow \bullet \textit{ baa}], [SheepNoise \rightarrow \bullet \textit{ baa}, \textit{ baa}] \} \]

\[ S_1 = \text{Goto}(S_0, \textit{SheepNoise}) = \]
\[ \{ [Goal \rightarrow \textit{SheepNoise} \bullet, \textbf{EOF}], [SheepNoise \rightarrow \textit{SheepNoise} \bullet \textit{ baa}, \textbf{EOF}], [SheepNoise \rightarrow \textit{SheepNoise} \bullet \textit{ baa}, \textit{ baa}] \} \]

\[ S_2 = \text{Goto}(S_0, \textit{baa}) = \{ [\textit{SheepNoise} \rightarrow \textit{baa} \bullet, \textbf{EOF}], [\textit{SheepNoise} \rightarrow \textit{baa} \bullet, \textit{ baa}] \} \]

\[ S_3 = \text{Goto}(S_1, \textit{baa}) = \{ [\textit{SheepNoise} \rightarrow \textit{SheepNoise} \textit{ baa} \bullet, \textbf{EOF}], [\textit{SheepNoise} \rightarrow \textit{SheepNoise} \textit{ baa} \bullet, \textit{ baa}] \} \]

- before \( T \Rightarrow \text{shift } k \)

so, ACTION\([s_0, \text{baa}]\) is "shift \( S_2 \)" (clause 1)

(items define same entry)
**Example from SheepNoise**

\[ S_0 : \{ [\text{Goal} \rightarrow \bullet \text{SheepNoise}, \text{EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa}, \text{EOF}], \\
\quad [\text{SheepNoise} \rightarrow \bullet \text{baa}, \text{EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa}], \\
\quad [\text{SheepNoise} \rightarrow \bullet \text{baa, baa}] \} \]

\[ S_1 = \text{Goto}(S_0, \text{SheepNoise}) = \]
\[ \{ [\text{Goal} \rightarrow \text{SheepNoise }\bullet, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise baa, EOF}], \\
\quad [\text{SheepNoise} \rightarrow \text{SheepNoise baa, baa}] \} \]

\[ S_2 = \text{Goto}(S_0, \text{baa}) = \{ [\text{SheepNoise} \rightarrow \text{baa }\bullet, \text{EOF}], \\
\quad [\text{SheepNoise} \rightarrow \text{baa }\bullet, \text{baa}] \} \]

\[ S_3 = \text{Goto}(S_1, \text{baa}) = \{ [\text{SheepNoise} \rightarrow \text{SheepNoise baa }\bullet, \text{EOF}], \\
\quad [\text{SheepNoise} \rightarrow \text{SheepNoise baa }\bullet, \text{baa}] \} \]

so, ACTION[S_1, baa] is “shift S_3” (clause 1)
Example from SheepNoise

$S_0 : \{ [Goal \rightarrow \bullet SheepNoise, \text{EOF}], [SheepNoise \rightarrow \bullet SheepNoise \text{ baa, EOF}],
\quad [SheepNoise \rightarrow \bullet \text{ baa, EOF}], [SheepNoise \rightarrow \bullet SheepNoise \text{ baa, baa}],
\quad [SheepNoise \rightarrow \bullet \text{ baa, baa}] \}$

$S_1 = \text{Goto}(S_0, SheepNoise) = \{ [Goal \rightarrow SheepNoise \bullet, \text{EOF}], [SheepNoise \rightarrow SheepNoise \bullet \text{ baa, EOF}],
\quad [SheepNoise \rightarrow SheepNoise \bullet \text{ baa, baa}] \}$

$S_2 = \text{Goto}(S_0, \text{ baa}) = \{ [SheepNoise \rightarrow \text{ baa } \bullet, \text{ EOF}],
\quad [SheepNoise \rightarrow \text{ baa } \bullet, \text{ baa}] \}$

$S_3 = \text{Goto}(S_1, \text{ baa}) = \{ [SheepNoise \rightarrow SheepNoise \text{ baa } \bullet, \text{ EOF}],
\quad [SheepNoise \rightarrow SheepNoise \text{ baa } \bullet, \text{ baa}] \}$

so, ACTION[$S_1, \text{EOF}$] is “accept” (clause 2)
Example from SheepNoise

\[ S_0 : \{ [Goal \rightarrow \bullet \text{SheepNoise}, \text{EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise} \text{ baa, EOF}], \\
[\text{SheepNoise} \rightarrow \bullet \text{ baa, EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa}], \\
[\text{SheepNoise} \rightarrow \bullet \text{ baa, baa}] \} \]

\[ S_1 = \text{Goto}(S_0, \text{SheepNoise}) = \]
\[ \{ [Goal \rightarrow \text{SheepNoise} \bullet, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{ baa, EOF}], \\
[\text{SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{ baa, baa}] \} \]

\[ S_2 = \text{Goto}(S_0, \text{ baa}) = \{ [\text{SheepNoise} \rightarrow \text{ baa} \bullet, \text{EOF}], \\
[\text{SheepNoise} \rightarrow \text{ baa} \bullet, \text{ baa}] \} \]

\[ S_3 = \text{Goto}(S_1, \text{ baa}) = \{ [\text{SheepNoise} \rightarrow \text{SheepNoise baa} \bullet, \text{EOF}], \\
[\text{SheepNoise} \rightarrow \text{SheepNoise baa} \bullet, \text{ baa}] \} \]

ACTION[\text{S}_3, \text{EOF}] \text{ is “reduce 1” (clause 3)} \text{ (baa, too) }

so, ACTION[\text{S}_2, \text{EOF}] \text{ is “reduce 2” (baa, too) (clause 3)}

COMP 412, Fall 2019
Building the Goto Table

**State**  

$S_0 : \{ [Goal \rightarrow \bullet \text{SheepNoise, EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise, baa, EOF}], [\text{SheepNoise} \rightarrow \bullet \text{baa, EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise, baa, baa}], [\text{SheepNoise} \rightarrow \bullet \text{baa, baa}] \}$

**$S_1 = Goto(S_0, \text{SheepNoise}) =$**

$\{ [Goal \rightarrow \text{SheepNoise} \bullet, EOF], [\text{SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{SheepNoise, baa, EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{baa, baa}] \}$

**$S_2 = Goto(S_0, \text{baa}) =$**

$\{ [\text{SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{baa, EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{baa, baa}] \}$

**$S_3 = Goto(S_1, \text{baa}) =$**

$\{ [\text{SheepNoise} \rightarrow \text{SheepNoise} \text{SheepNoise, baa, EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise} \text{SheepNoise, baa, baa}] \}$

The Goto table holds just the entries for nonterminal symbols.  
*(the baa column went into Action)*

<table>
<thead>
<tr>
<th>State</th>
<th>SN</th>
<th>baa</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$S_1$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>—</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$S_3$</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Goto Relationships**
ACTION & GOTO Tables

Here are the tables for the left-recursive *SheepNoise* grammar

The Tables

<table>
<thead>
<tr>
<th>ACTION TABLE</th>
<th>GOTO TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State</strong></td>
<td><strong>EOF</strong></td>
</tr>
<tr>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td><em>accept</em></td>
</tr>
<tr>
<td>2</td>
<td><em>reduce 2</em></td>
</tr>
<tr>
<td>3</td>
<td><em>reduce 1</em></td>
</tr>
</tbody>
</table>

The Grammar

0  

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>→</td>
<td><strong>SheepNoise</strong></td>
</tr>
<tr>
<td>1</td>
<td><strong>SheepNoise</strong></td>
<td>→</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember, this is the left-recursive *SheepNoise*; EaC2e shows the right-recursive version.
What can go wrong?

What if a set $s$ contains $[A \rightarrow \beta \cdot a \gamma, b]$ and $[B \rightarrow \beta \cdot, a]$?

- First item generates “shift”, second generates “reduce”
- Both define $\text{ACTION}[s, a]$ — cannot do both actions
- This is a fundamental ambiguity, called a \textbf{shift/reduce error or conflict}
- Modify the grammar to eliminate it \textit{(if-then-else)}
- Shifting will often resolve it correctly

What if a set $s$ contains $[A \rightarrow \gamma \cdot, a]$ and $[B \rightarrow \gamma \cdot, a]$?

- Each generates “reduce”, but with a different production
- Both define $\text{ACTION}[s, a]$ — cannot do both reductions
- This is a fundamental ambiguity, called a \textbf{reduce/reduce error or conflict}
- Modify the grammar to eliminate it \textit{(PL/I’s overloading of (...) )}

\textit{In either case, the grammar is not LR(1)}
Building the Canonical Collection

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

Repeatedly construct new states, until all are found

The algorithm

\[
\begin{align*}
  s_0 & \leftarrow \text{closure} \left( [S' \rightarrow S, \text{EOF}] \right) \\
  S & \leftarrow \{ s_0 \} \\
  k & \leftarrow 1
\end{align*}
\]

while ( \( S \) is still changing )

\[
\begin{align*}
  \forall s_j \in S \text{ and } \forall x \in (T \cup NT) \\
  s_k & \leftarrow \text{goto}(s_j, x) \\
  \text{record } s_j \rightarrow s_k \text{ on } x
\end{align*}
\]

if \( s_k \notin S \) then

\[
\begin{align*}
  S & \leftarrow S \cup \{ s_k \} \\
  k & \leftarrow k + 1
\end{align*}
\]

Remember this comment about implementing the equality test at the bottom of the algorithm to build the Canonical Collection of Sets of LR(1) Items?

- Only need to compare core items (see slide 14) — the rest will follow
- Represent items as a triple \((R, P, L)\)
  - \( R \) is the rule or production
  - \( P \) is the position of the placeholder
  - \( L \) is the lookahead symbol
- Order items, then
  1. Compare set cardinalities
  2. Compare (in order) by \( R, P, L \)

This membership / equality test requires careful and/or clever implementation.