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

Comp 412

Copyright 2018, 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
Computing Closures

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

- Any item \([A \rightarrow \beta \bullet B \delta, a]\) where \(B \in NT\) implies \([B \rightarrow \bullet \tau, x]\) 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**

```
Closure( s )
while ( s is still changing )
    \(\forall\) items \([A \rightarrow \beta \bullet B \delta, a]\) \(\in s\)
        lookahead \(\leftarrow\) FIRST(\(\delta a\)) \(//\) \(\delta\) might be \(\varepsilon\)
    \(\forall\) productions \(B \rightarrow \tau \in P\)
    \(\forall\) \(b \in\) lookahead
        if \([B \rightarrow \bullet \tau, b]\) \(\notin s\)
            then \(s \leftarrow s \cup \{ [B \rightarrow \bullet \tau, b]\} \)
```

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

Generate new lookaheads. See note on p. 128
Computing Gotos

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

- *Goto*( { [A→β • Xδ, a] }, X ) produces { [A→βX • δ, a] } *(obviously)*
- It finds all such items & uses *Closure()* to fill out the state

The Algorithm

```plaintext
Goto( s, X )
new ← ∅
∀ items [A→β • Xδ, a] ∈ s
new ← new ∪ { [A→βX • δ, 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**

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

- 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.
### Filling in the ACTION and GOTO Tables

#### The Table Construction Algorithm

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

### Many items generate no table entry

- Placeholder before a NT does not generate an ACTION table entry
- **Closure**() instantiates FIRST(X) directly for \([A \rightarrow \beta \cdot X \delta, a]\)
Another Example  

(grammar & sets)

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>

0  Goal  →  Expr
1  Expr  →  Term - Expr
2  |  Term
3  Term  →  Factor * Term
4  |  Factor
5  Factor →  id
Simplified Expression Grammar

Initialization Step

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

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

**Item in black** is the initial item.
**Items in gray** are added by \text{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**

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

\[ s_6 \leftarrow \text{goto}(s_3, \ast) \]

**Iteration 3**

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

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

**Goal, \ast, \text{&} - generate empty sets**

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

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

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

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

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

\[ s_3 \leftarrow \text{goto}(s_0, \text{Factor}) \]
\[ \{ [\text{Term} \rightarrow \text{Factor }\bullet \text{ * }\text{Term }\text{, EOF}], [\text{Term} \rightarrow \text{Factor }\bullet \text{ * }\text{Term }\text{, –}], \]
\[ [\text{Term} \rightarrow \text{Factor }\bullet, \text{ EOF}], [\text{Term} \rightarrow \text{Factor }\bullet, \text{ –}] \} \]

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

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

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

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

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

\[ s_7 \leftarrow \text{goto}(s_5, \text{Expr}) \]
\[ \{ [\text{Expr} \rightarrow \text{Term} \rightarrow \text{Expr} \cdot, \text{EOF}] \} \]
\[ \text{goto}(s_5, \text{Term}) \text{ recreates } s_2 \]
\[ \text{goto}(s_5, \text{Factor}) \text{ recreates } s_3 \]
\[ \text{goto}(s_5, \text{id}) \text{ recreates } s_4 \]

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

*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>—</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></td>
<td>8</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
Simplified Expression Grammar

The algorithm produces the following tables

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>id: 4</td>
<td>Expr: 1, Term: 2, Factor: 3</td>
</tr>
<tr>
<td>( s_1 )</td>
<td></td>
<td>acc</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>id: 5</td>
<td>Term: 3</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>Term: 5, Factor: 5</td>
<td></td>
</tr>
<tr>
<td>( s_4 )</td>
<td>Term: 6, Factor: 6</td>
<td></td>
</tr>
<tr>
<td>( s_5 )</td>
<td>id: 4</td>
<td>Expr: 7, Term: 2, Factor: 3</td>
</tr>
<tr>
<td>( s_6 )</td>
<td>id: 4</td>
<td>Factor: 8, Factor: 3</td>
</tr>
<tr>
<td>( s_7 )</td>
<td></td>
<td>Factor: 2</td>
</tr>
<tr>
<td>( s_8 )</td>
<td>Term: 4, Factor: 4</td>
<td></td>
</tr>
</tbody>
</table>

Notice that none of these are shift actions, obviously.
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 parseable, 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

§ 3.6.2 in EaC2e
Shrinking the Grammar

The Classic Expression Grammar

```
0 Goal → Expr
1 Expr → Expr + Term
2 | Expr - Term
3 | Term
4 Term → Term * Factor
5 | Term / Factor
6 | Factor
7 Factor → (Expr)
8 | number
9 | id
```

Canonical construction produces 32 states

- 32 x (9 + 3) = 384 ACTION/GOTO entries
- Large table, but still just 1.5kb

**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

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

(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

§ 3.6.2 in EaC2e

COMP 412, Fall 2018
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
            end if
        end for
    end if

for k ← 0 to MaxRow
    if Table[i,k] = Table[j,k] then
        same ← false
        break
    end if
end for

if same = true then
    MapTo[j] ← i
end if
```

Basic Table Compression Algorithm

- Same algorithm applies to ACTION & GOTO
  - Also columns in DFA transition tables
- 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
§ 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

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

classic space-time tradeoff

Fewer grammars, same languages
LR(1) versus LL(1)

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></td>
<td></td>
<td>s₄</td>
<td>s₅</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></td>
<td></td>
<td>s₁₀</td>
<td></td>
</tr>
<tr>
<td>s₇</td>
<td></td>
<td></td>
<td></td>
<td>s₁₁</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>r 5</td>
<td>r 7</td>
<td></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></td>
<td></td>
<td>s₁₄</td>
<td></td>
</tr>
<tr>
<td>s₁₃</td>
<td></td>
<td></td>
<td></td>
<td>s₁₅</td>
<td></td>
</tr>
<tr>
<td>s₁₄</td>
<td></td>
<td></td>
<td></td>
<td>r 4</td>
<td></td>
</tr>
<tr>
<td>s₁₅</td>
<td></td>
<td></td>
<td></td>
<td>r 6</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>

- **0**: Start → A
- **1**: | B
- **2**: A → ( A )
- **3**: | a
- **4**: B → ( B )
- **5**: | a

---

**COMP 412, Fall 2018**

24
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 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:

\[
\text{instruction\_list} : \begin{array}{c}
\text{instruction} \\
| \text{label\_def \ instruction} \\
| \text{instruction \ instruction\_list} \\
| \text{label\_def \ instruction \ instruction\_list}
\end{array}
\]

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

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

• 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

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

right recursion

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

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

left recursion

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

See pp. 145-146 in EaC2e
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($k$) ⊆ LR(1)

The inclusion hierarchy for context-free languages
The inclusion hierarchy for context-free grammars

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