CSCI 742 - Compiler Construction

Lecture 20
Parsing Wrap-up
Instructor: Hossein Hojjat

March 10, 2017
LR(0) Automaton Example

+  ×  num  $  S  R  L

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
<td>g3</td>
<td>g2</td>
<td>g1</td>
</tr>
<tr>
<td>1</td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>accept</td>
</tr>
<tr>
<td>4</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s5</td>
<td>s4</td>
<td>g7</td>
<td>g6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s5</td>
<td>s4</td>
<td>g9</td>
<td>g6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
</tbody>
</table>

r1  $S \rightarrow L \times R$

r2  $S \rightarrow R$

r3  $L \rightarrow \text{num}$

r4  $L \rightarrow + R$

r5  $R \rightarrow L$

r6  $L \rightarrow \text{num}$

r7  $L \rightarrow + R$

r8  $S \rightarrow L \times R$

r9  $R \rightarrow L \times R$
SLR Automaton Example

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>×</th>
<th>num</th>
<th>$</th>
<th>S</th>
<th>R</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
<td>g3</td>
<td>g2</td>
<td>g1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>r5/s8</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>accept</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s5</td>
<td>s4</td>
<td>g7</td>
<td>g6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s5</td>
<td>s4</td>
<td>g9</td>
<td>g6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FOLLOW(S) = $
FOLLOW(L) =
FOLLOW(R) = \{ \times, \}$

r1 \( S \to L \times R \)
r2 \( S \to R \)
r3 \( L \to \text{num} \)
r4 \( L \to + R \)
r5 \( R \to L \)

0 \( S' \to S\$ \)
1 \( R \to L \cdot \)
2 \( S \to \text{num} \)
3 \( S' \to S \cdot $ \)
4 \( L \to + R \)
5 \( R \to L \cdot \)
6 \( R \to L \cdot \)
7 \( R \to L \cdot \)
8 \( S \to L \times R \)
9 \( R \to L \times R \cdot \)
Grammar is not LR(0) and SLR, but it is LR(1)

There is no more shift/reduce conflict in the automaton:
LALR

- Drawback: LR(1) parse engine has a large number of states
- LALR (Look-Ahead LR parser): Simple technique to eliminate states
- If two LR(1) states are identical except for the look ahead symbol of their items, merge them
- Result is LALR(1) DFA
- It is more memory efficient, typically merges several LR(1) states
- May also have more reduce/reduce conflicts
- Power of LALR parsing is enough for many mainstream computer languages
- Several automatic parser generators such as Yacc or GNU Bison
- Consider for example these two LR(1) states

\[
\begin{align*}
X \rightarrow & \alpha \cdot, a \\
Y \rightarrow & \beta \cdot, c
\end{align*}
\]

\[
\begin{align*}
X \rightarrow & \alpha \cdot, b \\
Y \rightarrow & \beta \cdot, d
\end{align*}
\]

- They will be merged into the following LALR(1) states

\[
\begin{align*}
X \rightarrow & \alpha \cdot, \{a, b\} \\
Y \rightarrow & \beta \cdot, \{c, d\}
\end{align*}
\]
“Modern Compiler Implementation in Java”,
Andrew W. Appel, Jens Palsberg
• Task of a parser:
  find a derivation of a string in given a context-free grammar
• CYK recognizes languages defined by context-free grammars
  - cubic time $O(n^3)$
• Restricted forms of CFG can be parsed in linear time:
  - LL (left to right, left-most derivation)
  - LR (left to right, reverse right-most derivation)
• Simple top-down parser: LL(1)
  - Basic recursive-descent implementation
• More powerful parser: LR(1), bottom-up
• An efficiency hack on top of LR(1): LALR(1)
What to expect next?

- Is “x” an array, integer or a function? Is it declared?
- Is the expression “x + z” type-consistent?
- In “x[i]”, is “x” an array? Does it have the correct number of dimensions?
- Where can “x” be stored? (register, local, global, heap, static)
- How many arguments does “f()” take? What about “printf ()”? 


KEEP CALM AND HAVE A SAFE SPRING BREAK