CSE 401 – Compilers

Lecture 7: LR Parsing (part II)
Michael Ringenburg
Winter 2013

Reminders/Announcements

• Project part 1 is due on Monday.
• Part 2 will be assigned early next week.
  – Will be due 2 weeks after it is assigned.
• Will also assign homework 2 (parsing) next week.
  – Will be due 1 week after it is assigned.
• Midterm in class on February 15.
Review From Last Week

• Right-sentential form: $\alpha$ is a right-sentential form of the grammar $G = \langle N, \Sigma, P, S \rangle$ if $S \Rightarrow_{rm}^* \alpha$

- $S ::= aABe$
- $A ::= Abc | b$
- $B ::= d$

- Handle: The handle of a right-sentential form is the substring corresponding to the right hand side of the production that produced it from the previous step in the rightmost derivation.
### Review From Last Week

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aABe</td>
<td>a, aA, aAB, aABe</td>
</tr>
<tr>
<td>aAde</td>
<td>a, aA, aAd</td>
</tr>
<tr>
<td>abde</td>
<td>a, ab</td>
</tr>
<tr>
<td>aAbcde</td>
<td>a, aA, aAb, aAbc</td>
</tr>
<tr>
<td>abbcde</td>
<td>a, ab</td>
</tr>
<tr>
<td>aAbcbcde</td>
<td>a, aA, aAb, aAbc</td>
</tr>
<tr>
<td>abbcbcde</td>
<td>a, ab</td>
</tr>
</tbody>
</table>

... **Bold red**: Handle

- Viable prefix: a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form.
Review From Last Week

- Viable prefix: a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form.

• Viable prefixes and handles of a CFG are a regular language, thus can recognize with a DFA.
Review From Last Week

• Basic idea: Reduce by handle when we reach state corresponding to viable prefix that goes all the way to the end of a handle. Otherwise, shift.

• But, recall from last week that this involved a lot of DFA transitions at every step – not \( O(n) \).
• Consider what happens before and after a shift ...

Winter 2013  UW CSE 401 (Michael Ringenburg)
Consider what happens before and after a shift ...

Winter 2013  UW CSE 401 (Michael Ringenburg)
### Trace

- Consider what happens before and after a shift ...

- Repeat all of the states from before the shift, and then make one more transition.
• Consider what happens before and after a shift ...
• Repeat all of the states from before the shift, and then make one more transition.

Winter 2013           UW CSE 401 (Michael Ringenburg)

18
• Consider what happens before and after a reduce...
• Consider what happens before and after a reduce...

• Pop the handle off the stack
Trace

$S ::= aA\text{be}$
$A ::= Abc \mid b$
$B ::= d$

Stack | Input
------|--------
$\$     | abbcde$
$a$     | bccde$
$ab$    | cde$
$aa$    | de$
$aaA$   | e$
$aaAB$  | e$

- Pop the handle off the stack

Trace

$S ::= aA\text{be}$
$A ::= Abc \mid b$
$B ::= d$

Stack | Input
------|--------
$\$     | abbcde$
$a$     | bccde$
$ab$    | cde$
$aa$    | de$
$aaA$   | e$
$aaAB$  | e$

- Repeat all of the states up to the start of the handle
Trace

\[ S ::= aABe \]
\[ A ::= Abc \mid b \]
\[ B ::= d \]

Stack | Input
--- | ---
$ | abbcde$
$a | bbcde$
$ab | bcde$
$aA | bcde$
$aAb | cde$
$aAbc | de$
$aA | de$
$aAd | e$
$aAB | e$

- Then make a single transition corresponding to new nonterminal.

Avoiding DFA Rescanning

- Observation 1: no need to restart the DFA after a shift. Stay in same state and process next (shifted) token.
- Observation 2: after a reduction, the contents of the stack prior to the handle are the same as before. After that, new stack will contain a single non-terminal.
  - Scanning the new stack will take us through the same transitions up to the beginning of the handle. Then, one more transition for the new non-terminal.
  - Can record state numbers on the stack with each symbol, and go directly to the appropriate state when we pop the handle from the stack.
New Stack

- Change the stack to contain pairs of states and symbols from the grammar
  \[ s_0 X_1 s_1 X_2 s_2 \ldots X_n s_n \]
  - State \( s_0 \) is the start state
  - When we add a symbol to the stack, push the symbol \textit{plus} new FA state
  - If \( X_i \) is the beginning of the handle that we reduce, popping it will reveal \( s_{i-1} \), which is precisely the state the FA was in prior to reading the handle.
- Optimization: in an actual parser, only the state numbers need to be pushed, since they implicitly contain the symbol information (actually, items).

Trace

\[
\begin{align*}
S &::= aABe \\
A &::= Abc \mid b \\
B &::= d
\end{align*}
\]

Stack | Input
---|---
\( s_1 \) | abcde$

Shift first symbol on to stack.
Shifted ‘a’ and state $s_2$ on to stack, followed ‘a’ transition. Shift again.

Shifted ‘b’ and state $s_4$ on to stack, followed ‘b’ transition. Now, reduce.
Trace

$S ::= aA\bar{b}e$

$A ::= Abc \mid b$

$B ::= d$

Stack Input
$s_1$ abbcde$
$s_1s_2$ bbcde$
$s_1bs_4$ bcde$
$s_1s_2$ bcde$

Pop handle (b), revealing new state $s_2$. Go there.

---

Trace

$S ::= aA\bar{b}e$

$A ::= Abc \mid b$

$B ::= d$

Stack Input
$s_1$ abbcde$
$s_1s_2$ bbcde$
$s_1s_2bs_4$ bcde$
$s_1s_2$ bcde$

- Push nonterminal on left of reduction (A) onto stack, transition on new nonterminal, and push new state.
- Each step – read an input (shift) or produce an output (reduce). $O(n)$, where $n$ is input + output.
Encoding the DFA in a Table

- Given these optimizations, and a stack containing states, a shift-reduce parser’s DFA can be encoded in two tables
  - action table rows contain state and columns contain input symbols. Encodes what to do given current state and next symbol (e.g., shift and go to state 4).
  - goto table rows contain uncovered states (states revealed after pop) and columns contain nonterminals. Encodes transition to take after a reduction, given uncovered state and new nonterminal.
    - Based on transition we’d take from uncovered state if we saw the nonterminal. E.g., reduce to A and uncover s2, goto s3
- Note necessity of the stack ... can’t be done with just an FA, because language grammars are not regular.

Action Table Actions (1)

- Given the current state and input symbol, the main possible actions are
  - si – shift the input symbol and state i onto the stack (i.e., shift and move to state i)
  - rj – reduce using grammar production j
    - The production number tells us how many <symbol, state> pairs to pop off the stack
### Action Table Actions (2)

- Other possible *action* table entries
  - *accept*
  - *blank* – no transition – syntax error
    - A LR parser will detect an error as soon as possible on a left-to-right scan
    - A real compiler needs to produce an error message, recover, and continue parsing when this happens. Various strategies exist for this, e.g., advance past next semicolon token.

### Goto

- When a reduction is performed using production \( A ::= \beta, |\beta| <\text{symbol, state}> \) pairs are popped from the stack revealing a state `uncovered_s` on the top of the stack.
- `goto[uncovered_s, A]` is the new state to push on the stack when reducing with production \( A ::= \beta \) (after popping handle and pushing \( A \))
Dijkstra

“Thou shalt not use goto”

Dijkstra, revisited

“Thou shalt not use goto, except as a table in an LR parser.”
LR Parse Table for

1. \( S ::= aABe \)
2. \( A ::= Abc \)
3. \( A ::= b \)
4. \( B ::= d \)

---

Table

<table>
<thead>
<tr>
<th>State</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s6</td>
<td>s5</td>
</tr>
<tr>
<td>4</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>5</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>6</td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>8</td>
<td>s9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>r1</td>
</tr>
</tbody>
</table>

Winter 2013

UW CSE 401 (Michael Ringenburg)
LR Parsing Algorithm Pseudocode

```java
word = scanner.getToken();
while (true) {
    s = state on top of stack;
    if (action[s, word] = si) {
        push; push i; // i is state
        word = scanner.getToken();
    } else if (action[s, word] = rj) {
        pop 2 * length of right side of
        production j;
        uncovered_s = top of stack;
        push left side A of production / ;
        push state goto[uncovered_s, A];
    }
} else if (action[s, word] = accept) {
    return;
} else {
    // no entry in action table
    report syntax error;
    halt or attempt recovery;
}

Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>abbcde$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>d</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>e</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>ac</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>s2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>s4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>s6</td>
<td>s5</td>
</tr>
<tr>
<td>4</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>5</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>6</td>
<td>s7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>8</td>
<td>s9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
</tbody>
</table>

Winter 2013 UW CSE 401 (Michael Ringenburg)
LR States – Where do they come from?

• Idea is that each state of this DFA encodes
  – The set of all possible productions that we could be looking at, given the current state of the parse, and
  – Where we are in the right hand side of each of those productions
    • Part-way through: shift
    • All the way through: reduce
    • Preview: Could there be some ambiguity here?
      – Reduce-Reduce and Shift-Reduce conflicts

Items

• An item is a production with a dot in the right hand side
• Example: Items for production $A ::= XY$
  $A ::= \cdot XY$
  $A ::= X.Y$
  $A ::= XY.$
• Idea: The dot represents a position in the production
DFA with items for

\[ S ::= aABe \]
\[ A ::= Abc | b \]
\[ B ::= d \]