CS412/CS413
Introduction to Compilers
Tim Teitelbaum
Lecture 8: LR parsing
February 7, 2007

Bottom-up Parsing

- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - Scan the input from left to right and determine a right-most derivation of program (backwards)
  - Allows left-recursive grammars, virtually all programming languages
  - Easier to express programming language syntax
- Shift-reduce parsers
  - Parsers for LR grammars
  - Automatic parser generators (e.g., yacc, CUP)

Bottom-up Parsing

- Right-most derivation -- backward
  - Start with the tokens; end with the start symbol

\[
(1 + 2 + (3 + 4)) + 5 \leftarrow (S + 2 + (3 + 4)) + 5 \leftarrow (S + (3 + 4)) + 5 \leftarrow (S + (S + 4)) + 5 \leftarrow (S + (S + E)) + 5 \leftarrow (S + E) + 5 \leftarrow S + 5 \leftarrow S + E + 5 \leftarrow S + (S + E) + 5 \leftarrow S + (S + (E + 4)) + 5 \leftarrow S + (S + E + 4) + 5 \leftarrow S + (S + (S + 4)) + 5 \leftarrow S + (S + (E + 4)) + 5 \leftarrow S + (S + (S + 4)) + 5 \leftarrow S + (S + (E + 4)) + 5 \leftarrow S + (S + (S + 4)) + 5 \leftarrow S + (S + (E + 4)) + 5
\]

Top-down Parsing

- In left-most derivation, entire tree above a token (2) has been expanded when it is encountered
Top-down vs. Bottom-up

**Bottom-up:** Don't need to figure out as much of the parse tree for a given amount of input.

### Shift-reduce Parsing

- Parsing actions: a sequence of shift and reduce operations.
- Parser state: a stack of terminals and non-terminals (grows to the right).
- Current derivation is always stack+input.

<table>
<thead>
<tr>
<th>Derivation step</th>
<th>Stack</th>
<th>Unconsumed Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1+2+(3+4))+5</td>
<td>(1</td>
<td>+2+(3+4)))+5</td>
</tr>
<tr>
<td>(1+2+(3+4))+5</td>
<td>(1+2</td>
<td>+2+(3+4)))+5</td>
</tr>
<tr>
<td>(S+E+(3+4))+5</td>
<td>(S+E</td>
<td>+3+4))+5</td>
</tr>
<tr>
<td>(S+E+(3+4))+5</td>
<td>(S+E</td>
<td>+3+4)+5</td>
</tr>
</tbody>
</table>

- Parsing is a sequence of shifts and reduces.
- **Shift:** move look-ahead token to stack.
  
  Stack: \( \{ 1 \} \), Input: \(+2+(3+4)\), Action: Shift 1

- **Reduce:** Replace symbols \( \beta \) from top of stack with non-terminal symbol \( A \), corresponding to production \( A \rightarrow \beta \) (pop \( \beta \), push \( A \)).
  
  Stack: \( \{ S \} \), Input: \(+3+4\), Action: Reduce \( S \rightarrow S + E \)

- **Issues:**
  - Sometimes can reduce but shouldn’t
  - Sometimes can reduce in different ways

### Action Selection Problem

- Given stack \( \sigma \) and look-ahead symbol \( b \), should parser:
  - Shift \( b \) onto the stack (making it \( \sigma b \))
  - Reduce \( A \rightarrow \beta \) assuming that stack has the form \( \alpha \beta \) (making it \( \alpha A \))
LR Parsing Engine

- Basic mechanism:
  - Use a set of parser states
  - Use a stack of states
  - Use a parsing table to:
    - Determine what action to apply (shift/reduce)
    - Determine the next state
- The parser actions can be precisely determined from the table

LR Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action to take and next state to enter</th>
<th>Non-terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Action table</td>
<td>Goto table</td>
</tr>
</tbody>
</table>

- Algorithm: look at entry for current state Q and input terminal c
  - If Table[Q,c] = shift(Q') then shift:
    - push(Q')
  - If Table[Q,c] = A → α then reduce:
    - pop(|α|); Q = top(); push(Table[Q',A])

LR(1) Parsing Table Example

<table>
<thead>
<tr>
<th>State</th>
<th>Action to take and next state to enter</th>
<th>Non-terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Action table</td>
<td>Goto table</td>
</tr>
<tr>
<td>1</td>
<td>s3, 3, ε, S, L</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S → id, S → id, S → id, S → id, S → id</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3, 3, ε, S, L</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s6, S, ε, S, L</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S → (L), S → (L), S → (L), S → (L)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>L → S, L → S, L → S, L → S, L → S</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s3, s2, ε, S, L</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L → LS, L → LS, L → LS, L → LS, L → LS</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>s3, s2, ε, S, L</td>
<td></td>
</tr>
</tbody>
</table>

LR(k) Grammars

- LR(k) = Left-to-right scanning, Right-most derivation, k look-ahead characters
- Main cases: LR(0), LR(1), and some variations (SLR and LALR(1))
- Parsers for LR(0) Grammars:
  - Determine the actions without any lookahead symbol
  - Will help us understand shift-reduce parsing
- Read: CUP User Manual