CS412/413

Introduction to Compilers
Radu Rugina

Lecture 8: Bottom-up Parsing
6 Feb 02

Shift-reduce Parsing

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

<table>
<thead>
<tr>
<th>Derivation step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1+2+(3+4))+5</td>
<td>1</td>
<td>2</td>
<td>shift 1</td>
</tr>
<tr>
<td>(1+2+(3+4))+5</td>
<td>1</td>
<td>2</td>
<td>reduce E → num</td>
</tr>
</tbody>
</table>

Shift-reduce Actions

- Parsing is a sequence of shifts and reduces
- Shift: move look-ahead token to stack
  
  stack | input | action
  
  (S+E) | +3+4 | reduce S → S+E

LR Parsing Engine

- Basic mechanism:
  - Use a set of parser states
  - Use a stack with alternating symbols and states
  - E.g.: (S S in + 5)
  - 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

Shift-reduce Parsing

<table>
<thead>
<tr>
<th>derivation</th>
<th>stack</th>
<th>input stream</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → S+E</td>
<td>S+E</td>
<td>S+num</td>
<td>reduce E → num</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Next action and next state</th>
<th>Next state</th>
<th>Goto table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-terminals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action table</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Algorithm: look at entry for current state S and input terminal C
  - IfTable(S,C) = u(S) then shift:
    - push(u), push(S)

  - IfTable(S,C) = X → α then reduce:
    - pop(α), S = box, push(α), pushTable(S, X)
LR Parsing Table Example

<table>
<thead>
<tr>
<th>( )</th>
<th>id</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S-&gt;id S-&gt;id S-&gt;id S-&gt;id S-&gt;id S-&gt;id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td>accept</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>L-&gt;S L-&gt;S L-&gt;S L-&gt;S L-&gt;S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>L-&gt;L,S L-&gt;L,S L-&gt;L,S L-&gt;L,S L-&gt;L,S</td>
<td></td>
<td></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 look-ahead symbol
  - Will help us understand shift-reduce parsing

Building LR(0) Parsing Tables

- To build the parsing table:
  - Define states of the parser
  - Build a DFA to describe the transitions between states
  - Use the DFA to build the parsing table
- Each LR(0) state is a set of LR(0) items:
  - An LR(0) item: $X \rightarrow \alpha \beta$, where $X \rightarrow \alpha \beta$ is a production in the grammar
  - The LR(0) items keep track of the progress on all of the possible upcoming productions
  - The item $X \rightarrow \alpha \beta$ abstracts the fact that the parser already matched the string $\alpha$ at the top of the stack

Example LR(0) State

- An LR(0) item is a production from the language with a separator "." somewhere in the RHS of the production

\[
\text{state} \quad \begin{array}{c}
E \rightarrow \text{num} ; \\
E \rightarrow ( : S ) \end{array} \quad \text{Item}
\]

- Sub-string before "." is already on stack (beginnings of possible $\gamma$s to be reduced)
- Sub-string after "." : what we might see next

LR(0) Grammar

- Nested lists:
  \[
  S \rightarrow ( L ) \mid \text{id} \\
  L \rightarrow S \mid L , S 
  \]
- Examples:
  - (a, b, c)
  - ((a,b), (c,d), (e,f))
  - (a, (b,c,d), (f,g))

Start State & Closure

- Start state
  - Augment grammar with production $S' \rightarrow S\,\$. 
  - Start state of DFA has empty stack: $S' \rightarrow .\,\$. 
- Closure of a parser state:
  - Start with $\text{Closure}(S) = S$
  - Then for each item in $S$:
    - $X \rightarrow \alpha : Y \beta$
      - Add the items for all the productions $Y \rightarrow \gamma$ to the closure of $S$:
        - $Y \rightarrow \gamma$
Closure Example

\[
S \rightarrow (L) \mid \text{id} \\
L \rightarrow S \mid L, S
\]

DFA start state

\[
S' \rightarrow S \implies \text{closure} \\
S' \rightarrow S\$
\]

- Set of possible productions to be reduced
- Added items have the "," located at the beginning
- No symbols for these items on the stack yet

The Goto Operation

- Goto operation describes transitions between parser states, which are sets of items
- Algorithm for a state S and a symbol Y
  \[
  S' = (X \rightarrow \alpha Y \beta | X \rightarrow \alpha \mid \alpha, Y, \beta \in S) \\
  \text{Goto}(S, X) = \text{Closure}(S')
  \]

Goto: Terminal Symbols

Grammar:

\[
S \rightarrow \ (.L) \\
L \rightarrow ., S \\
L \rightarrow ., S, L \\
S \rightarrow (.L) \mid \text{id} \\
L \rightarrow S \mid L, S
\]

In new state, include all items that have appropriate input symbol just after dot, advance dot in those items, and take closure.

Goto: Non-terminal Symbols

( Same algorithm for transitions on non-terminals)

Applying Reduce Actions

\[
\text{Pop RHS off stack, replace with LHS } X \rightarrow \gamma \text{, then rerun DFA (e.g. } X)\]

Full DFA (Appel p. 63)

Grammar:

\[
S \rightarrow (L) \mid \text{id} \\
L \rightarrow S \mid L, S
\]
**Parsing Example: ((a),b)**

<table>
<thead>
<tr>
<th>derivation</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a)b)</td>
<td>i</td>
<td>((a)b)</td>
<td>shift go to 3</td>
</tr>
<tr>
<td>((a)b)</td>
<td>i</td>
<td>(a)b</td>
<td>shift go to 3</td>
</tr>
<tr>
<td>((a)b)</td>
<td>i</td>
<td>a</td>
<td>shift go to 2</td>
</tr>
<tr>
<td>((a)b)</td>
<td>i</td>
<td>b</td>
<td>reduce S ⇒ id</td>
</tr>
<tr>
<td>((a)b)</td>
<td>i</td>
<td>(a)b</td>
<td>shift go to 6</td>
</tr>
<tr>
<td>((a)b)</td>
<td>i</td>
<td>A</td>
<td>reduce S ⇒ (L)</td>
</tr>
<tr>
<td>(S,b)</td>
<td>i</td>
<td>b</td>
<td>reduce S ⇒ id</td>
</tr>
<tr>
<td>(L,b)</td>
<td>i</td>
<td>b</td>
<td>shift go to 9</td>
</tr>
<tr>
<td>(L,b)</td>
<td>i</td>
<td>L</td>
<td>reduce S ⇒ id</td>
</tr>
<tr>
<td>(L)</td>
<td>i</td>
<td>L</td>
<td>shift go to 6</td>
</tr>
<tr>
<td>(L)</td>
<td>i</td>
<td>A</td>
<td>reduce S ⇒ (L)</td>
</tr>
<tr>
<td>S</td>
<td>i</td>
<td>A</td>
<td>done</td>
</tr>
</tbody>
</table>

**Reductions**

- On reducing X → γ with stack αρ:
  - Pop γ off stack, revealing prefix α and state
  - Take single step in DFA from top state
  - Push X onto stack with new DFA state

- Example:
  - ((a),b) → (a),b shift, go to 2
  - ((a),b) → (a),b reduce S ⇒ (L)
  - ((S),b) → (S),b reduce L ⇒ S

---

**Build the Parsing Table**

- States in the table = states in the DFA
- For a transition S → S' on terminal C:
  
  \[ \text{Shift}(S') \subseteq \text{Table}(S, C) \]

- For a transition S → S' on non-terminal N:
  
  \[ \text{Goto}(S') \subseteq \text{Table}(S, N) \]

- If S is a reduction state X → γ then:
  
  \[ \text{Reduce}(X → γ) \subseteq \text{Table}(S, *) \]

---

**Computed LR Parsing Table**

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s1</td>
<td>s2</td>
<td>g7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td>g5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s6</td>
<td>s8</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s1</td>
<td>s7</td>
<td>g6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S→(L)</td>
<td>S→(L)</td>
<td>S→(L)</td>
<td>S→(L)</td>
</tr>
<tr>
<td>7</td>
<td>L→S</td>
<td>L→S</td>
<td>L→S</td>
<td>L→S</td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>L→L</td>
<td>L→L</td>
<td>L→L</td>
<td>L→L</td>
</tr>
</tbody>
</table>

---

**LR(0) Summary**

- LR(0) parsing recipe:
  - Start with an LR(0) grammar
  - Compute LR(0) states and build DFA:
    - Use the closure operation to compute states
    - Use the goto operation to compute transitions between states
  - Build the LR(0) parsing table from the DFA
  - This process can be automated, i.e. we can build parser generator tools

---

**LR(0) Limitations**

- An LR(0) machine only works if states with reduce actions have a single reduce action -- in those states, always reduce ignoring lookahead
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use look-ahead to choose

<table>
<thead>
<tr>
<th></th>
<th>shift/reduce</th>
<th>reduce/reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>L→L</td>
<td>S→L,S</td>
<td>S→S,L</td>
</tr>
</tbody>
</table>
LR(0) Parsing Table

<table>
<thead>
<tr>
<th></th>
<th>( )</th>
<th>id</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S → id</td>
<td>S → id</td>
<td>S → id</td>
<td>S → id</td>
<td>S → id</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td>accept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S → (L)</td>
<td>S → (L)</td>
<td>S → (L)</td>
<td>S → (L)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>L → S</td>
<td>L → S</td>
<td>L → S</td>
<td>L → S</td>
<td>L → S</td>
</tr>
<tr>
<td>7</td>
<td>s3</td>
<td>s2</td>
<td>g9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L → S</td>
<td>L → S</td>
<td>L → S</td>
<td>L → S</td>
<td>L → S</td>
</tr>
</tbody>
</table>

A Non-LR(0) Grammar

- Grammar for addition of numbers:
  \[ S \rightarrow S + E \mid E \]
  \[ E \rightarrow \text{num} \mid (S) \]
- Left-associative is LR(0)
- Right-associative version is not LR(0)
  \[ S \rightarrow E + S \mid E \]
  \[ E \rightarrow \text{num} \mid (S) \]

LR(0) Parsing Table

1. \[ S \rightarrow E + S \mid E \]
   \[ E \rightarrow \text{num} \mid (S) \]
2. \[ S \rightarrow E + S \]
3. \[ S \rightarrow E + S \]
4. \[ S \rightarrow E + S \]
5. \[ S \rightarrow E + S \]
6. \[ S \rightarrow E + S \]
7. \[ S \rightarrow E + S \]
8. \[ S \rightarrow E + S \]
9. \[ S \rightarrow E + S \]

What to do in state 2?
1. \[ S \rightarrow E + S \]
2. \[ S \rightarrow E + S \]

Next Time

- Learn about other kinds of LR parsing:
  - SLR = improved LR(0)
  - LR(1) = 1 character lookahead
  - LALR(1) = Look-Ahead LR(1)
- Basic ideas are the same as for LR(0)
  - Parser states with LR items
  - DFA with transitions between parser states
  - Parser table with shift/reduce/goto actions