CS412/413

Introduction to Compilers
Radu Rugina

Lecture 8: Bottom-up Parsing
11 Feb '04

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>Unconsumed input</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1+2*(3+4))+5</td>
<td>(1</td>
<td>1+2*(3+4))+5</td>
</tr>
<tr>
<td>(E+2*(3+4))+5</td>
<td>(E</td>
<td>+2*(3+4))+5</td>
</tr>
<tr>
<td>(S+2*(3+4))+5</td>
<td>(S</td>
<td>+2*(3+4))+5</td>
</tr>
</tbody>
</table>

CS 412/413 Spring 2004 Introduction to Compilers

Shift-reduce Actions

- Parsing is a sequence of shifts and reduces
- Shift: move look-ahead token to stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>1+2*(3+4))+5</td>
<td>shift 1</td>
</tr>
<tr>
<td>(1</td>
<td>+2*(3+4))+5</td>
<td></td>
</tr>
</tbody>
</table>

- Reduce: Replace symbols γ from top of stack with non-terminal symbol X, corresponding to production X → γ (pop γ, push X)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S+E</td>
<td>+(3+4))+5</td>
<td>reduce S → S+E</td>
</tr>
<tr>
<td>(S</td>
<td>+(3+4))+5</td>
<td></td>
</tr>
</tbody>
</table>

CS 412/413 Spring 2004 Introduction to Compilers

LR Parsing Engine

- Basic mechanism:
  - Use a set of parser states
  - Use a stack with alternating symbols and states
    - Ex: (S 10 + S)
  - Use a parsing table:
    - Determine what action to apply (shift/reduce)
    - Determine the next state

The parser actions can be precisely determined from the table

The LR Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Next action and next state</th>
<th>Non-terminals</th>
<th>Next state</th>
</tr>
</thead>
</table>

- Algorithm: look at entry for current state S and input terminal C
  - If Table(S,C) = d(S') then shift:
    - push(C), push(S')
  - If Table(S,C) = X then reduce:
    - pop(2*(ax)), S'=top(), push(X), push(Table(S’|X))

CS 412/413 Spring 2004 Introduction to Compilers
### LR Parsing Table Example

<table>
<thead>
<tr>
<th>State</th>
<th>Production</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$S \rightarrow id$, $S \rightarrow ,$</td>
<td>$S$</td>
</tr>
<tr>
<td>2</td>
<td>$S \rightarrow id$, $S \rightarrow S$</td>
<td>$S$</td>
</tr>
<tr>
<td>3</td>
<td>$S \rightarrow id$, $S \rightarrow S$</td>
<td>$S$</td>
</tr>
<tr>
<td>4</td>
<td>$S \rightarrow id$, $S \rightarrow S$</td>
<td>$S$</td>
</tr>
<tr>
<td>5</td>
<td>$S \rightarrow id$, $S \rightarrow S$</td>
<td>$S$</td>
</tr>
<tr>
<td>6</td>
<td>$S \rightarrow \lambda$, $S \rightarrow \lambda$</td>
<td>$S$</td>
</tr>
<tr>
<td>7</td>
<td>$S \rightarrow \lambda$, $S \rightarrow \lambda$</td>
<td>$S$</td>
</tr>
<tr>
<td>8</td>
<td>$S \rightarrow \lambda$, $S \rightarrow \lambda$</td>
<td>$S$</td>
</tr>
<tr>
<td>9</td>
<td>$S \rightarrow \lambda$, $S \rightarrow \lambda$</td>
<td>$S$</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

### 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: $X \rightarrow \alpha \cdot \beta$, where $X \rightarrow \alpha \cdot \beta$ is a production in the grammar
  - The LR(0) items keep track of the progress on all of the possible upcoming productions
  - The item $X \rightarrow \alpha \cdot \beta$ 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
- State: $S \rightarrow (\cdot)$
- Item: $L \rightarrow (\cdot)$
- Sub-string before "." is already on stack
- Sub-string after ".": what we might see next

### LR(0) Grammar

- **Nested lists:**
  - $S \rightarrow (\cdot)$ | $id$
  - $L \rightarrow S \mid L \mid S$
- **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: $S \rightarrow S$
- **Closure of a parser state:**
  - Start with Closure($S$) = $S$
  - Then for each item in $S:
    - $X \rightarrow \alpha \cdot \beta$
    - Add the items for all the productions $Y \rightarrow \gamma$ to the closure of $S$: $Y \rightarrow \gamma$
Closure Example

\[ S \rightarrow (L) \mid id \]
\[ L \rightarrow S \mid L, S \]

DFA start state
\[ S' \rightarrow .S$ \]
\[ S' \rightarrow .S$ \]
\[ S \rightarrow (L) \]
\[ S \rightarrow .(L) \]
\[ S \rightarrow .id \]

- Set of possible productions to be reduced next
- 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 \( Y \)
  \[ S' = \{ X \rightarrow \alpha Y \cdot \beta \mid X \rightarrow \alpha \cdot Y \beta \in S \} \]
  \[ \text{Goto}(S, Y) = \text{Closure}(S') \]

Goto: Terminal Symbols

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

- Pop RHS off stack, replace with LHS X \((X \rightarrow \gamma)\), then rerun DFA (e.g. \((X)\))

Full DFA

Grammar:
\[ S \rightarrow (L) \mid id \]
\[ L \rightarrow S \mid L, S \]
Parsing Example: \(((a),b)\)

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

<table>
<thead>
<tr>
<th>derivation</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(((a),b)) ← (i) (((a),b))</td>
<td>(i)</td>
<td>(((a),b))</td>
<td>shift, goto 3</td>
</tr>
<tr>
<td>(((a),b)) ← (i) (((a),b))</td>
<td>(i)</td>
<td>(((a),b))</td>
<td>shift, goto 3</td>
</tr>
<tr>
<td>(((a),b)) ← (i) (((a),b))</td>
<td>(i)</td>
<td>(((a),b))</td>
<td>shift, goto 2</td>
</tr>
<tr>
<td>(((a),b)) ← (i) (((a),b))</td>
<td>(i)</td>
<td>(((a),b))</td>
<td>reduce S→id</td>
</tr>
<tr>
<td>(((a),b)) ← (i) (((a),b))</td>
<td>(i)</td>
<td>(((a),b))</td>
<td>reduce L→S</td>
</tr>
<tr>
<td>(((a),b)) ← (i) (((a),b))</td>
<td>(i)</td>
<td>(((a),b))</td>
<td>shift, goto 6</td>
</tr>
<tr>
<td>((L),b) ← (i) ((L),b)</td>
<td>(i)</td>
<td>((L),b)</td>
<td>reduce L→S</td>
</tr>
<tr>
<td>((L),b) ← (i) ((L),b)</td>
<td>(i)</td>
<td>((L),b)</td>
<td>shift, goto 8</td>
</tr>
<tr>
<td>((S),b) ← (i) ((S),b)</td>
<td>(i)</td>
<td>((S),b)</td>
<td>reduce S→iL</td>
</tr>
<tr>
<td>((L),b) ← (i) ((L),b)</td>
<td>(i)</td>
<td>((L),b)</td>
<td>reduce L→iL</td>
</tr>
<tr>
<td>((S),b) ← (i) ((S),b)</td>
<td>(i)</td>
<td>((S),b)</td>
<td>shift, goto 6</td>
</tr>
<tr>
<td>((S),b) ← (i) ((S),b)</td>
<td>(i)</td>
<td>((S),b)</td>
<td>reduce S→iL</td>
</tr>
<tr>
<td>(S) ← (S)</td>
<td>(S)</td>
<td>(S)</td>
<td>done</td>
</tr>
</tbody>
</table>

Build the Parsing Table

- States in the table = states in the DFA
- For a transition \(S \rightarrow S'\) on terminal C:
  \(\text{Shift}(S') \subseteq \text{Table}[S,C]\)
- For a transition \(S \rightarrow S'\) on non-terminal N:
  \(\text{Goto}(S') \subseteq \text{Table}[S,N]\)
- If \(S\) is a reduction state \(X \rightarrow Y\) then:
  \(\text{Reduce}(X \rightarrow Y) \subseteq \text{Table}[S,\_\_]

Computed LR Parsing Table

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>,</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></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td>s6</td>
<td>s8</td>
<td>accept</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>g9</td>
</tr>
<tr>
<td>7</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L→iLS</td>
<td>L→iLS</td>
<td>L→iLS</td>
<td>L→iLS</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>L→iLS</td>
<td>L→iLS</td>
<td>L→iLS</td>
<td>L→iLS</td>
<td></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 -- if 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

\[
\begin{align*}
\text{ok} & \quad \text{shift/reduce} \quad \text{reduce/reduce} \\
L \rightarrow L, S, & \quad L \rightarrow L, S, \quad L \rightarrow S, L, \quad L \rightarrow S, L, \\
S \rightarrow S, L, & \quad S \rightarrow S, L, \quad L \rightarrow S, L, \quad L \rightarrow S.
\end{align*}
\]
**LR(0) Parsing Table**

<table>
<thead>
<tr>
<th></th>
<th>( )</th>
<th>id</th>
<th>$</th>
<th>S (L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td>S</td>
<td>L</td>
</tr>
<tr>
<td>2</td>
<td>S=id</td>
<td>id</td>
<td>S</td>
<td>id</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>id</td>
<td>S</td>
<td>id</td>
</tr>
<tr>
<td>4</td>
<td>s4</td>
<td>s2</td>
<td>S</td>
<td>id</td>
</tr>
<tr>
<td>5</td>
<td>s5</td>
<td>s2</td>
<td>S</td>
<td>id</td>
</tr>
<tr>
<td>6</td>
<td>S=(L)</td>
<td>S=(L)</td>
<td>S=(L)</td>
<td>S=(L)</td>
</tr>
<tr>
<td>7</td>
<td>L&gt;=$</td>
<td>L&gt;$</td>
<td>L&gt;$</td>
<td>L&gt;$</td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td>L</td>
<td>&gt;=$</td>
</tr>
</tbody>
</table>

**A Non-LR(0) Grammar**

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

**LR(0) Parsing Table**

\[
\begin{align*}
S & \rightarrow E + S | E \\
E & \rightarrow \text{num} | (S)
\end{align*}
\]

**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