CS412/413
Introduction to Compilers
Radu Rugina

Lecture 8: Bottom-up Parsing
5 Feb 03

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></td>
<td>(1+2+3+4)+5</td>
</tr>
<tr>
<td>(1+2+3+4)+5</td>
<td></td>
<td>(E       +2+3+4)+5</td>
</tr>
<tr>
<td>(S+E       +3+4)+5</td>
<td></td>
<td>(S       +2+3+4)+5</td>
</tr>
<tr>
<td>(S+E       +3+4)+5</td>
<td></td>
<td>(S       +2+3+4)+5</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
  (      1+2+3+4) +5  shift 1
  (1      +2+3+4) +5
- Reduce: Replace symbols γ from top of stack with non-terminal symbol X, corresponding to production X → γ
  (pop γ, push X)
  stack  input  action
  (S+E  +(3+4)+5)  reduce S → S+E
  (S       +(3+4)+5)

Shift-reduce Parsing

The LR Parsing Engine
- Basic mechanism:
  - Use a set of parser states
  - Use a stack with alternating symbols and states
    - E.g.: (  S  §  +  5
  - 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>Next state</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 S and input terminal C
  - If Table(S, C) = a(S) then shift:
    - push(C), push(0)
  - If Table(S, C) = X → γ then reduce:
    - pop(2*{a}), S = top(j), push(X), push(Table[S: X])
LR Parsing Table Example

<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>g4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S→id</td>
<td>S→id</td>
<td>S→s3</td>
<td>s3</td>
<td>g7</td>
</tr>
<tr>
<td>3</td>
<td>S→id</td>
<td>S→id</td>
<td>S→s2</td>
<td>s2</td>
<td>g5</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) S→(L) S→(L) S→(L)</td>
<td>S→(L) S→(L) S→(L) S→(L)</td>
<td>g9</td>
<td></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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L→L,S</td>
<td>L→L,S</td>
<td>L→L,S</td>
<td>L→L,S</td>
<td>L→L,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: $X \rightarrow \alpha . \beta$, 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

```
state          item
E → num ;   E → ( \{ , S) ;
```

- 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:
  - $S \rightarrow (L) | \text{id}$
  - $L \rightarrow S | L | S$
- Examples
  - (a, b, c)
  - ((a,b), (c,d), (e,f))
  - (a, (b,c,d), ((f,g)))

Parse tree for $(a, (b,c,d), ((f,g)))$

```
S
 / \ /
L S  L S
 /    /
S  id
 |    /
( a,b,c,d)
|      /
( (a,b)
|      /
S  id
 |    /
( c,d)
|      /
( e,f)
|      /
( f,g)
```

Start State & Closure

- Start state
  - Augment grammar with production $S \rightarrow S \ S$
  - Start state of DFA has empty stack: $S \rightarrow S \ S$
- Closure of a parser state:
  - Start with 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 → (L) | id
L → S | L, S

DFA start state
S' → S

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 → a Y b | X → a Y b ∈ S}
  - Goto(S, X) = 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→γ), then rerun DFA (e.g. (X))

Full DFA

Grammar:
S → (L) | id
L → S | L, S

States causing reductions

Final state
### Parsing Example: \((a,b)\)

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

\[
L \rightarrow S \mid L,S
\]

<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, goto 3</td>
</tr>
<tr>
<td>((a,b))</td>
<td>(i_1)</td>
<td>((a,b))</td>
<td>shift, goto 3</td>
</tr>
<tr>
<td>((a,b))</td>
<td>(i_1, a)</td>
<td>(b)</td>
<td>reduce (S \rightarrow id)</td>
</tr>
<tr>
<td>((a,b))</td>
<td>(i_1, b)</td>
<td>(a)</td>
<td>reduce (L \rightarrow S)</td>
</tr>
<tr>
<td>((a,b))</td>
<td>(i_1, L_a)</td>
<td>(b)</td>
<td>shift, goto 6</td>
</tr>
<tr>
<td>((a,b))</td>
<td>(i_1, a)</td>
<td>(b)</td>
<td>reduce (S \rightarrow L(L))</td>
</tr>
<tr>
<td>((L))</td>
<td>(i_1, L_a)</td>
<td>(b)</td>
<td>shift, goto 6</td>
</tr>
<tr>
<td>((L))</td>
<td>(i_1, L_b)</td>
<td>(b)</td>
<td>reduce (S \rightarrow L(L))</td>
</tr>
<tr>
<td>(S)</td>
<td>(i_4)</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 \(C\):
  
  Shift\((S') \subseteq Table[S,C]\)

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

- If \(S\) is a reduction state \(X \rightarrow \gamma\) then:
  
  Reduce\((X \rightarrow \gamma) \subseteq Table[S,\gamma]\)

---

### 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>(s_3)</td>
<td>(s_2)</td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(S \rightarrow id)</td>
<td>(S \rightarrow id)</td>
<td>g7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(s_3)</td>
<td>(s_2)</td>
<td>g5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(s_6)</td>
<td>(s_8)</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(S \rightarrow L(L))</td>
<td>(S \rightarrow L(L))</td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(L \rightarrow S)</td>
<td>(L \rightarrow S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(L \rightarrow S)</td>
<td>(L \rightarrow S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(s_3)</td>
<td>(s_2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>(L \rightarrow L(S))</td>
<td>(L \rightarrow L(S))</td>
<td></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 -- 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 \rightarrow L, S)</td>
<td>(L \rightarrow L, S)</td>
<td>(S \rightarrow S, L)</td>
</tr>
<tr>
<td>(L \rightarrow S, L)</td>
<td>(L \rightarrow S, L)</td>
<td>(L \rightarrow S)</td>
</tr>
</tbody>
</table>
LR(0) Parsing 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

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