Syntax Analysis – Part V
Finish LR(0) Parsing
Start on LR(1) Parsing

EECS 483 – Lecture 8
University of Michigan
Monday, February 4, 2008

Announcements/Reading

- Reading
  » Today 4.5, 4.7
From Last Time: Shift-Reduce Parsing

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

derivation stack input stream action
(1+2+(3+4))+5 ( ) (1+2+(3+4))+5 shift
(1+2+(3+4))+5 (1) 1+2+(3+4))+5 shift
(1+2+(3+4))+5 (E) +2+(3+4))+5 reduce E \rightarrow \text{num}
(E+2+(3+4))+5 (S) +2+(3+4))+5 reduce S \rightarrow E
(S+2+(3+4))+5 (S+) 2+(3+4))+5 shift
(S+2+(3+4))+5 (S+2) +2+(3+4))+5 reduce E \rightarrow \text{num}
(S+2+(3+4))+5 (S+E) +2+(3+4))+5 reduce S \rightarrow S+E
(S+2+(3+4))+5 (S) +2+(3+4))+5 shift
(S+2+(3+4))+5 (S+) +2+(3+4))+5 reduce E \rightarrow \text{num}
(S+2+(3+4))+5 (S+3) +2+(3+4))+5 shift
(S+2+(3+4))+5 (S+(3+4))+5 reduce E \rightarrow \text{num}
...

From Last Time: LR Parsing Table

Example

We want to derive this in an algorithmic fashion

<table>
<thead>
<tr>
<th>State</th>
<th>Input terminal</th>
<th>Non-terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>id, S</td>
</tr>
<tr>
<td>1</td>
<td>s2</td>
<td>S \rightarrow id</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>id, S</td>
</tr>
<tr>
<td>2</td>
<td>s2</td>
<td>S \rightarrow id</td>
</tr>
<tr>
<td>3</td>
<td>accept</td>
<td>S \rightarrow id</td>
</tr>
<tr>
<td>4</td>
<td>s6</td>
<td>S \rightarrow (L)</td>
</tr>
<tr>
<td>5</td>
<td>s8</td>
<td>S \rightarrow (L)</td>
</tr>
<tr>
<td>6</td>
<td>s3</td>
<td>L \rightarrow S</td>
</tr>
<tr>
<td>7</td>
<td>s2</td>
<td>L \rightarrow S</td>
</tr>
<tr>
<td>8</td>
<td>L \rightarrow L, S</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>L \rightarrow L, S</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Input terminal</th>
<th>Non-terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>id, S</td>
</tr>
<tr>
<td>1</td>
<td>s2</td>
<td>S \rightarrow id</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>id, S</td>
</tr>
<tr>
<td>2</td>
<td>s2</td>
<td>S \rightarrow id</td>
</tr>
<tr>
<td>3</td>
<td>accept</td>
<td>S \rightarrow id</td>
</tr>
<tr>
<td>4</td>
<td>s6</td>
<td>S \rightarrow (L)</td>
</tr>
<tr>
<td>5</td>
<td>s8</td>
<td>S \rightarrow (L)</td>
</tr>
<tr>
<td>6</td>
<td>s3</td>
<td>L \rightarrow S</td>
</tr>
<tr>
<td>7</td>
<td>s2</td>
<td>L \rightarrow S</td>
</tr>
<tr>
<td>8</td>
<td>L \rightarrow L, S</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>L \rightarrow L, S</td>
<td></td>
</tr>
</tbody>
</table>
From Last Time: Start State and Closure

- Start state
  - Augment grammar with production: \(S' \rightarrow S \, S\)
  - Start state of DFA has empty stack: \(S' \rightarrow . \, S\)

- Closure of a parser state:
  - Start with \(\text{Closure}(S) = S\)
  - Then for each item in \(S\):
    - \(X \rightarrow \alpha \cdot Y \beta\)
    - Add items for all the productions \(Y \rightarrow \gamma\) to the closure of \(S\): \(Y \rightarrow . \, \gamma\)

Closure

\[
\begin{align*}
S &\rightarrow (L) \mid \text{id} \\
L &\rightarrow S \mid L,S \\
\end{align*}
\]

DFA start state

- Set of possible productions to be reduced next
- Closure of a parser state, \(S\):
  - Start with \(\text{Closure}(S) = S\)
  - Then for each item in \(S\):
    - \(X \rightarrow \alpha \cdot Y \beta\)
    - Add items for all the productions \(Y \rightarrow \gamma\) to the closure of \(S\): \(Y \rightarrow . \, \gamma\)
The Goto Operation

- Goto operation = describes transitions between parser states, which are sets of items
- Algorithm: for state S and a symbol Y
  » If the item \([X \rightarrow \alpha \cdot Y \beta]\) is in I, then
  » \(\text{Goto}(I, Y) = \text{Closure(} [X \rightarrow \alpha \cdot Y \cdot \beta ] \text{)}\)

Goto: Terminal Symbols

In new state, include all items that have appropriate input symbol just after dot, advance do in those items and take closure
Goto: Non-terminal Symbols

```
Goto: Non-terminal Symbols

S' → S $  
S → (L)  
S → (L)  
S → id  
S → id  
L → S  
L → S  
L → S  
L → S  
L → S  
L → S  

same algorithm for transitions on non-terminals
```

Class Problem

```
Class Problem

E' → E  
E → E + T | T  
T → T * F | F  
F → (E) | id  

1. If I = { [E' → E] }, then Closure(I) = ??

2. If I = { [E' → E .], [E → E . + T] }, then Goto(I+) = ??
```
Applying Reduce Actions

![Diagram of grammar rules and reduce actions]

Grammar:
- $S \rightarrow (L)$
- $S \rightarrow \text{id}$
- $L \rightarrow S | L,S$

Executing reduce actions:
- $S \rightarrow \text{id}$
- $L \rightarrow S$
- $S \rightarrow (L)$
- $S \rightarrow .(L)$

States causing reductions:
- Dot has reached the end!

Reduce RHS off stack, replace with LHS $X (X \rightarrow \beta)$, then rerun DFA (e.g., $(x)$)

Reductions

- On reducing $X \rightarrow \beta$ with stack $\alpha\beta$
  - Pop $\beta$ off stack, revealing prefix $\alpha$ and state
  - Take single step in DFA from top state
  - Push $X$ onto stack with new DFA state

Example

<table>
<thead>
<tr>
<th>derivation</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a),b)</td>
<td>1 (3 3 a),b</td>
<td>shift, goto 2</td>
<td></td>
</tr>
<tr>
<td>((a),b)</td>
<td>1 (3 3 a 2),b</td>
<td>reduce $S \rightarrow \text{id}$</td>
<td></td>
</tr>
<tr>
<td>((S),b)</td>
<td>1 (3 3 S 7),b</td>
<td>reduce $L \rightarrow S$</td>
<td></td>
</tr>
</tbody>
</table>
Full DFA

Grammar

S → (L) | id
L → S | 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>1</td>
<td>((a),b)</td>
<td>shift, goto 3</td>
</tr>
<tr>
<td>((a),b)</td>
<td>1(3(a))</td>
<td>(a),b</td>
<td>shift, goto 3</td>
</tr>
<tr>
<td>((a),b)</td>
<td>1(3(a),b)</td>
<td>a,b</td>
<td>shift, goto 2</td>
</tr>
<tr>
<td>((a),b)</td>
<td>1(3(a),b)</td>
<td>b</td>
<td>reduce S→id</td>
</tr>
<tr>
<td>((S),b)</td>
<td>1(3(3(S)),b)</td>
<td>b</td>
<td>reduce L→S</td>
</tr>
<tr>
<td>((L),b)</td>
<td>1(3(3(L),b)</td>
<td>b</td>
<td>shift, goto 6</td>
</tr>
<tr>
<td>((L),b)</td>
<td>1(3(3(L),b)</td>
<td>b)</td>
<td>reduce S→(L)</td>
</tr>
<tr>
<td>(S,b)</td>
<td>1(3(S),b)</td>
<td>b</td>
<td>reduce L→S</td>
</tr>
<tr>
<td>(L,b)</td>
<td>1(3(L),b)</td>
<td>b</td>
<td>shift, goto 8</td>
</tr>
<tr>
<td>(L,b)</td>
<td>1(3(L),b)</td>
<td>b</td>
<td>shift, goto 9</td>
</tr>
<tr>
<td>(L,b)</td>
<td>1(3(L),b)</td>
<td>b</td>
<td>reduce S→id</td>
</tr>
<tr>
<td>(L,S)</td>
<td>1(3(L),S)</td>
<td>b</td>
<td>reduce L→L,S</td>
</tr>
<tr>
<td>(L)</td>
<td>1(3(L))</td>
<td>b</td>
<td>shift, goto 6</td>
</tr>
<tr>
<td>(L)</td>
<td>1(3(L))</td>
<td>b</td>
<td>reduce S→(L)</td>
</tr>
<tr>
<td>S</td>
<td>1S4</td>
<td>$</td>
<td>done</td>
</tr>
</tbody>
</table>
Building the Parsing Table

- States in the table = states in the DFA
- For transition $S \rightarrow S'$ on terminal $C$:
  - Table[$S, C$] += Shift($S'$)
- For transition $S \rightarrow S'$ on non-terminal $N$:
  - Table[$S, N$] += Goto($S'$)
- If $S$ is a reduction state $X \rightarrow \beta$ then:
  - Table[$S, *$] += Reduce($X \rightarrow \beta$)

Computed LR Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Input terminal</th>
<th>Non-terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>2</td>
<td>S $\rightarrow$ id</td>
<td>S $\rightarrow$ id</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>4</td>
<td>s6</td>
<td>s8</td>
</tr>
<tr>
<td>5</td>
<td>S $\rightarrow$ (L)</td>
<td>S $\rightarrow$ (L)</td>
</tr>
<tr>
<td>6</td>
<td>L $\rightarrow$ S</td>
<td>L $\rightarrow$ S</td>
</tr>
<tr>
<td>7</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>8</td>
<td>L $\rightarrow$ L,S</td>
<td>L $\rightarrow$ L,S</td>
</tr>
<tr>
<td>9</td>
<td>L $\rightarrow$ L,S</td>
<td>L $\rightarrow$ L,S</td>
</tr>
</tbody>
</table>

winter 2008

- 14 -
LR(0) Summary

- LR(0) parsing recipe:
  - Start with LR(0) grammar
  - Compute LR(0) states and build DFA:
    - Use the closure operation to compute states
    - Use the goto operation to compute transitions
  - Build the LR(0) parsing table from the DFA
- This can be done automatically

Class Problem

Generate the DFA for the following grammar:

\[
S \rightarrow E + S \mid E \\
E \rightarrow \text{num}
\]
LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action
  - Always reduce regardless of lookahead
- With a more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use lookahead to choose

<table>
<thead>
<tr>
<th></th>
<th>shift/reduce</th>
<th>reduce/reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L \rightarrow L, S</td>
<td>L \rightarrow L, S, L</td>
<td>L \rightarrow S, L, L</td>
</tr>
<tr>
<td>S \rightarrow S, S, L</td>
<td>L \rightarrow S, L</td>
<td>L \rightarrow S</td>
</tr>
</tbody>
</table>

A Non-LR(0) Grammar

- Grammar for addition of numbers
  - S \rightarrow S + E | E
  - E \rightarrow num
- Left-associative version is LR(0)
- Right-associative is **not LR(0)** as you saw with the previous class problem
  - S \rightarrow E + S | E
  - E \rightarrow num
LR(0) Parsing Table

1  \[S' \to . S \$\]  
     \[S \to . E + S\]  
     \[S \to . E\]  
     \[E \to . num\]  

2  \[E \to . E + S\]  
     \[S \to . E\]  

3  \[E \to + S\]  
     \[S \to E + S\]  
     \[S \to E\]  
     \[E \to . num\]  

4  \[E \to . num\]  

5  \[S \to E + S\]  

Shift or reduce in state 2?

<table>
<thead>
<tr>
<th>num</th>
<th>+</th>
<th>$</th>
<th>E</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S → E</td>
<td></td>
<td></td>
<td>g2</td>
</tr>
</tbody>
</table>

Grammar

\[S \to E + S \mid E\]  
\[E \to . num\]

Solve Conflict With Lookahead

- 3 popular techniques for employing lookahead of 1 symbol with bottom-up parsing
  - SLR – Simple LR
  - LALR – LookAhead LR
  - LR(1)
- Each as a different means of utilizing the lookahead
  - Results in different processing capabilities
SLR Parsing

- SLR Parsing = Easy extension of LR(0)
  - For each reduction $X \rightarrow \beta$, look at next symbol $C$
  - Apply reduction only if $C$ is in FOLLOW($X$)
- SLR parsing table eliminates some conflicts
  - Same as LR(0) table except reduction rows
  - Adds reductions $X \rightarrow \beta$ only in the columns of symbols in FOLLOW($X$)

Example: FOLLOW($S$) = \{$\}$

<table>
<thead>
<tr>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow E \cdot S \mid E$</td>
</tr>
<tr>
<td>$E \rightarrow$ num</td>
</tr>
</tbody>
</table>

| num | + | $|$ | $E$ | $S$ |
|-----|---|-----|-----|-----|
| 1   | s4|     | g2  | g6  |
| 2   | s3| S→E |     |

SLR Parsing Table

- Reductions do not fill entire rows as before
- Otherwise, same as LR(0)

| num | + | $|$ | $E$ | $S$ |
|-----|---|-----|-----|-----|
| 1   | s4|     | g2  | g6  |
| 2   | s3| S→E |     |
| 3   | s4|     | g2  | g5  |
| 4   | E→num| E→num| S→E+S | s7 | accept |
Class Problem

Consider:

\[ S \rightarrow L = R \]
\[ S \rightarrow R \]
\[ L \rightarrow *R \]
\[ L \rightarrow \text{ident} \]
\[ R \rightarrow L \]

Think of \( L \) as l-value, \( R \) as r-value, and \( * \) as a pointer dereference.

When you create the states in the SLR(1) DFA, 2 of the states are the following:

\[
\begin{array}{c}
S \rightarrow L . = R \\
R \rightarrow L . \\
S \rightarrow R .
\end{array}
\]

Do you have any shift/reduce conflicts? (Not as easy as it looks)

LR(1) Parsing

- Get as much as possible out of 1 lookahead symbol parsing table
- LR(1) grammar = recognizable by a shift/reduce parser with 1 lookahead
- LR(1) parsing uses similar concepts as LR(0)
  - Parser states = set of items
  - LR(1) item = LR(0) item + lookahead symbol possibly following production
    - LR(0) item: \( S \rightarrow . S + E \)
    - LR(1) item: \( S \rightarrow . S + E \star \)
    - Lookahead only has impact upon REDUCE operations, apply when lookahead = next input
LR(1) States

- LR(1) state = set of LR(1) items
- LR(1) item = \((X \rightarrow \alpha \cdot \beta , y)\)
  - Meaning: \(\alpha\) already matched at top of the stack, next expect to see \(\beta y\)
- Shorthand notation
  - \((X \rightarrow \alpha \cdot \beta , \{x_1, ..., x_n\})\)
  - means:
    - \((X \rightarrow \alpha \cdot \beta , x_1)\)
    - ...
    - \((X \rightarrow \alpha \cdot \beta , x_n)\)
- Need to extend closure and goto operations

---

LR(1) Closure

- LR(1) closure operation:
  - Start with \(\text{Closure}(S) = S\)
  - For each item in \(S\):
    - \(X \rightarrow \alpha \cdot Y \beta , z\)
    - and for each production \(Y \rightarrow \gamma\), add the following item to the closure of \(S\): \(Y \rightarrow \cdot \gamma , \text{FIRST}(\beta z)\)
  - Repeat until nothing changes
- Similar to LR(0) closure, but also keeps track of lookahead symbol

---
LR(1) Start State

- Initial state: start with \((S' \rightarrow S, \$)\), then apply closure operation
- Example: sum grammar

\[
\begin{align*}
S' & \rightarrow S \$
S & \rightarrow E + S \mid E \\
E & \rightarrow \text{num}
\end{align*}
\]

\[
\begin{array}{|c|}
\hline
S' \rightarrow .S, \$
\hline
\end{array}
\quad \text{closure}
\]

\[
\begin{align*}
S' & \rightarrow .S, \$
S & \rightarrow .E + S, \$
S & \rightarrow .E, \$
E & \rightarrow .\text{num}, +, \$
\end{align*}
\]

LR(1) Goto Operation

- LR(1) goto operation = describes transitions between LR(1) states
- Algorithm: for a state \(S\) and a symbol \(Y\) (as before)
  - If the item \([X \rightarrow \alpha \cdot Y \beta]\) is in \(I\), then
  - \(\text{Goto}(I, Y) = \text{Closure}( [X \rightarrow \alpha \cdot Y \beta] )\)

\[
\begin{array}{|c|c|}
\hline
\text{S1} & \text{Goto(S1, ‘+’)} \quad \text{S2} \\
S & \rightarrow E . + S, \$
S & \rightarrow E ., \$
\hline
\end{array}
\]

Grammar:

\[
\begin{align*}
S' & \rightarrow S S \\
S & \rightarrow E + S \mid E \\
E & \rightarrow \text{num}
\end{align*}
\]
Class Problem

1. Compute: Closure(I = \{S \Rightarrow E + . S , $\})
   \[ S' \Rightarrow S \, $ \]

2. Compute: Goto(I, num)
   \[ S \Rightarrow E + S \mid E \]

3. Compute: Goto(I, E)
   \[ E \Rightarrow \text{num} \]