Syntax Analysis – Part IV
Bottom-Up Parsing

EECS 483 – Lecture 7
University of Michigan
Wednesday, January 30, 2008

Announcements: Turning in Project 1

- Yuan Lin is helping with grading
  linyz@umich.edu 😊
- Anonymous ftp to www.eecs.umich.edu
  » login: anonymous
  » pw: your email addr
- cd groups/eecs483
- put uniquename.l
- put uniquename.y
- Note, you won’t be able to “get” or “rm” any files in the directory – try if you wish
- If you make a mistake, then put uniquename2.l and send Yuan mail (linyz@umich.edu)
- Grading signup sheet available next wk
More details: Turning in Project 1

- Instructions for uploading projects:
- ftp to www.eecs.umich.edu
- when prompted for "Name", type: anonymous
- when prompted for "Password", type your email address
- when you see the "ftp>" prompt, type cd groups/eecs483
- then type put <your project name> <your project name>
- you will get back something saying transfer complete as long as everything went well

Grammars

- Have been using grammar for language “sums with parentheses” (1+2+(3+4))+5
- Started with simple, right-associative grammar
  - S → E + S | E
  - E → num | (S)
- Transformed it to an LL(1) by left factoring:
  - S → ES’
  - S’ → ε | +S
  - E → num | (S)
- What if we start with a left-associative grammar?
  - S → S + E | E
  - E → num | (S)
Reminder: Left vs Right Associativity

Consider a simpler string on a simpler grammar: “1 + 2 + 3 + 4”

Right recursion: right associative

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

Left recursion: left associative

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

Left Recursion

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

“1 + 2 + 3 + 4”

<table>
<thead>
<tr>
<th>derived string</th>
<th>lookahead</th>
<th>read/unread</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>1</td>
<td>1+2+3+4</td>
</tr>
<tr>
<td>S+E</td>
<td>1</td>
<td>1+2+3+4</td>
</tr>
<tr>
<td>S+E+E</td>
<td>1</td>
<td>1+2+3+4</td>
</tr>
<tr>
<td>S+E+E+E</td>
<td>1</td>
<td>1+2+3+4</td>
</tr>
<tr>
<td>E+E+E</td>
<td>1</td>
<td>1+2+3+4</td>
</tr>
<tr>
<td>1+E+E</td>
<td>2</td>
<td>1+2+3+4</td>
</tr>
<tr>
<td>1+2+E</td>
<td>3</td>
<td>1+2+3+4</td>
</tr>
<tr>
<td>1+2+3+E</td>
<td>4</td>
<td>1+2+3+4</td>
</tr>
<tr>
<td>1+2+3+4</td>
<td>$</td>
<td>1+2+3+4</td>
</tr>
</tbody>
</table>

Is this right? If not, what’s the problem?
Left-Recursive Grammars

- Left-recursive grammars don’t work with top-down parsers: we don’t know when to stop the recursion
- Left-recursive grammars are NOT LL(1)!
  - $S \rightarrow S\alpha$
  - $S \rightarrow \beta$
- In parse table
  - Both productions will appear in the predictive table at row $S$ in all the columns corresponding to $\text{FIRST}(\beta)$

Eliminate Left Recursion

- Replace
  - $X \rightarrow X\alpha_1 \mid \ldots \mid X\alpha_m$
  - $X \rightarrow \beta_1 \mid \ldots \mid \beta_n$
- With
  - $X \rightarrow \beta_1X' \mid \ldots \mid \beta_nX'$
  - $X' \rightarrow \alpha_1X' \mid \ldots \mid \alpha_mX' \mid \epsilon$
- See complete algorithm in Dragon book
Class Problem

Transform the following grammar to eliminate left recursion:

E → E + T | T
T → T * F | F
F → (E) | num

Creating an LL(1) Grammar

- Start with a left-recursive grammar
  - S → S + E
  - S → E
  - and apply left-recursion elimination algorithm
    - S → ES'
    - S' → +ES' | ε

- Start with a right-recursive grammar
  - S → E + S
  - S → E
  - and apply left-factoring to eliminate common prefixes
    - S → ES'
    - S' → +S | ε
Top-Down Parsing Summary

Language grammar -> Left-recursion elimination
                   -> Left factoring

LL(1) grammar

predictive parsing table
     FIRST, FOLLOW

recursive-descent parser

parser with AST gen

New Topic: Bottom-Up Parsing

- A more powerful parsing technology
- LR grammars – more expressive than LL
  - Construct right-most derivation of program
  - Left-recursive grammars, virtually all programming languages are left-recursive
  - Easier to express syntax
- Shift-reduce parsers
  - Parsers for LR grammars
  - Automatic parser generators (yacc, bison)
Bottom-Up Parsing (2)

- Right-most derivation – Backwards
  - Start with the tokens
  - End with the start symbol
  - Match substring on RHS of production, replace by LHS

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

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

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

\[
\leftarrow (S+(S+E))+5 \leftarrow (S+(S))+5 \leftarrow (S+E)+5 \leftarrow (S)+5
\]

\[
\leftarrow E+5 \leftarrow S+E \leftarrow S
\]

Bottom-Up Parsing (3)

- Advantage of bottom-up parsing: can postpone the selection of productions until more of the input is scanned

\[
S \rightarrow S + E | E
\]

\[
E \rightarrow \text{num} | (S)
\]

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

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

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

Advantage of bottom-up parsing: can postpone the selection of productions until more of the input is scanned
Top-Down Parsing

S \rightarrow S + E \mid E
E \rightarrow \text{num} \mid (S)

S \rightarrow S+E \rightarrow E+E \rightarrow (S)+E \rightarrow (S+E)+E
\rightarrow (S+E+E)+E \rightarrow (E+E)+E
\rightarrow (1+E+E)+E \rightarrow (1+2+E)+E \ldots

In left-most derivation, entire tree above token (2) has been expanded when encountered

Top-Down vs Bottom-Up

Bottom-up: Don’t need to figure out as much of he parse tree for a given amount of input \rightarrow More time to decide what rules to apply

Top-down

Bottom-up
Terminology: LL vs LR

- **LL(k)**
  - Left-to-right scan of input
  - Left-most derivation
  - k symbol lookahead
  - [Top-down or predictive] parsing or LL parser
  - Performs pre-order traversal of parse tree

- **LR(k)**
  - Left-to-right scan of input
  - Right-most derivation
  - k symbol lookahead
  - [Bottom-up or shift-reduce] parsing or LR parser
  - Performs post-order traversal of parse tree

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 step = 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>$(E)$</td>
<td>$(1+2+(3+4)) + 5$</td>
</tr>
<tr>
<td>$(E+2+(3+4)) + 5$</td>
<td>$(S)$</td>
<td>$(E+2+(3+4)) + 5$</td>
</tr>
<tr>
<td>$(S+2+(3+4)) + 5$</td>
<td>$(S+E)$</td>
<td>$(S+2+(3+4)) + 5$</td>
</tr>
<tr>
<td>$(S+E+(3+4)) + 5$</td>
<td></td>
<td>$(S+E+(3+4)) + 5$</td>
</tr>
</tbody>
</table>
Shift-Reduce Actions

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

<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 β from top of stack with non-terminal symbol X corresponding to the production: \( X \rightarrow β \) (e.g., pop β, 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 ( \rightarrow ) S+E</td>
</tr>
<tr>
<td>(S)</td>
<td>+(3+4))+5</td>
<td></td>
</tr>
</tbody>
</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>(1+2+(3+4))+5</td>
<td>(</td>
<td>1+2+(3+4))+5</td>
<td>shift</td>
</tr>
<tr>
<td>(1+2+(3+4))+5</td>
<td>(</td>
<td>1+2+(3+4))+5</td>
<td>shift</td>
</tr>
<tr>
<td>(1+2+(3+4))+5</td>
<td>(1</td>
<td>+2+(3+4))+5</td>
<td>reduce E ( \rightarrow ) num</td>
</tr>
<tr>
<td>(E+2+(3+4))+5</td>
<td>(E</td>
<td>+2+(3+4))+5</td>
<td>reduce S ( \rightarrow ) E</td>
</tr>
<tr>
<td>(S+2+(3+4))+5</td>
<td>(S</td>
<td>+2+(3+4))+5</td>
<td>shift</td>
</tr>
<tr>
<td>(S+2+(3+4))+5</td>
<td>(S+</td>
<td>2+(3+4))+5</td>
<td>shift</td>
</tr>
<tr>
<td>(S+2+(3+4))+5</td>
<td>(S+2</td>
<td>+(3+4))+5</td>
<td>reduce E ( \rightarrow ) num</td>
</tr>
<tr>
<td>(S+E+2+(3+4))+5</td>
<td>(S+E</td>
<td>+(3+4))+5</td>
<td>reduce S ( \rightarrow ) S+E</td>
</tr>
<tr>
<td>(S+2+(3+4))+5</td>
<td>(S</td>
<td>+(3+4))+5</td>
<td>shift</td>
</tr>
<tr>
<td>(S+2+(3+4))+5</td>
<td>(S+</td>
<td>(3+4))+5</td>
<td>shift</td>
</tr>
<tr>
<td>(S+2+(3+4))+5</td>
<td>(S+</td>
<td>3+(4))+5</td>
<td>shift</td>
</tr>
<tr>
<td>(S+2+(3+4))+5</td>
<td>(S+</td>
<td>+(4))+5</td>
<td>reduce E ( \rightarrow ) num</td>
</tr>
</tbody>
</table>
Potential Problems

- How do we know which action to take: whether to shift or reduce, and which production to apply
- Issues
  - Sometimes can reduce but should not
  - Sometimes can reduce in different ways

Action Selection Problem

- Given stack $\beta$ and look-ahead symbol $b$, should parser:
  - Shift $b$ onto the stack making it $\beta b$?
  - Reduce $X \rightarrow \gamma$ assuming that the stack has the form $\beta = \alpha \gamma$ making it $\alpha X$?
- If stack has the form $\alpha \gamma$, should apply reduction $X \rightarrow \gamma$ (or shift) depending on stack prefix $\alpha$?
  - $\alpha$ is different for different possible reductions since $\gamma$'s have different lengths
LR Parsing Engine

- Basic mechanism
  - Use a set of parser states
  - Use stack with alternating symbols and states
    - E.g., 1 ( 6 S 10 + 5 (blue = state numbers)
  - Use parsing table to:
    - Determine what action to apply (shift/reduce)
    - Determine next state

- The parser actions can be precisely determined from the table

LR Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Terminals</th>
<th>Non-terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Next action and next state</td>
<td>Next state</td>
</tr>
<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] = s(S’) then shift:
    - push(C), push(S’)
  - If Table[S,C] = XÆ α then reduce:
    - pop(2*|α|), S’= top(), push(X), push(Table[S’,X])
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></td>
<td>( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>id $</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>2</td>
<td>$\rightarrow$id</td>
<td>$\rightarrow$id</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>4</td>
<td>s6 accept</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$\rightarrow$S(L)</td>
<td>$\rightarrow$S(L)</td>
</tr>
<tr>
<td>6</td>
<td>L$\rightarrow$S</td>
<td>L$\rightarrow$S</td>
</tr>
<tr>
<td>7</td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>g7 g5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>g9</td>
<td></td>
</tr>
</tbody>
</table>

LR(k) Grammars

- LR(k) = Left-to-right scanning, right-most derivation, k lookahead chars
- Main cases
  - LR(0), LR(1)
  - Some variations SLR and LALR(1)
- Parsers for LR(0) Grammars:
  - Determine the actions without any lookahead
  - 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 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

- Sub-string before “.” is already on the stack (beginnings of possible \( \gamma \)’s to be reduced)
- Sub-string after “.”: what we might see next
Class Problem

For the production, 
\[ E \rightarrow \text{num} \mid (S) \]

Two items are: 
\[ E \rightarrow \text{num} \]
\[ E \rightarrow (\cdot, S) \]

Are there any others? If so, what are they? If not, why?

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)))

Parse tree for
\[ (a, (b,c), d) \]
Start State and 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 \( \text{Closure}(S) = S \)
  - Then for each item in \( S \):
    - \( X \rightarrow \alpha . Y \beta \)
    - Add items for all the productions \( Y \rightarrow \gamma \) to the closure of \( S \): \( Y \rightarrow . \gamma \)

Closure Example

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

DFA start state \( S' \rightarrow . S \) $

- 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 state S and a symbol Y
  » If the item $[X \rightarrow \alpha \cdot Y \cdot \beta]$ is in I, then
  » $\text{Goto}(I, Y) = \text{Closure}( [X \rightarrow \alpha \cdot Y \cdot \beta] )$

Class Problem

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T \cdot F \mid F$
$F \rightarrow (E) \mid \text{id}$

1. If $I = \{ [E' \rightarrow E] \}$, then $\text{Closure}(I) = ??$

2. If $I = \{ [E' \rightarrow E \cdot], [E \rightarrow E \cdot + T] \}$, then $\text{Goto}(I,+) = ??$