Bottom-Up Parsing

Lecture 6

Dr. Sean Peisert – ECS 142 – Spring 2009
Bottom-Up Parsing

• Preferred method in practice

• Also called “LR parsing”
  • L means that tokens are read left to right
  • R means that it constructs a rightmost derivation

• We’ll talk about “LL parsing” later
Idea

• LR parsing reduces a string to the start symbol by inverting productions

• str ← input string of terminals
• repeat
  • Identify ß in str such that A → ß is a production (i.e., str = α ß γ)
  • Replace ß by A in str (i.e., str becomes α A γ)
• until str = S
bottom-up parse diagram
Important Fact #1

- An LR parser traces a rightmost derivation in reverse.
Where Do Reductions Happen?

- Let $\alpha \beta \gamma$ be a step of a bottom-up parse
- Assume the next reduction is by $A \rightarrow \beta$
- Then $\gamma$ is a string of terminals!

- Why? Because $\alpha A \gamma \rightarrow \alpha \beta \gamma$ is a step in a right-most derivation (you would have reduced $\gamma$ otherwise).
Notation

- Split the string into two substrings
- Right substring (a string of terminals) is as yet unexamined by parser.
- Left substring has terminals and non-terminals
- The dividing point is marked by a \( \triangleright \)
- The \( \triangleright \) is not part of the string.
- Initially, all input is examined: \( \triangleright x_1, x_2 \ldots x_n \)
Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:
  - *Shift*
  - *Reduce*
Shift

• *Shift*: Move ▷ one place to the right

• Shifts a terminal to the left string

\[ E + (▷ \text{ int}) \Rightarrow E + (\text{int ▷}) \]
• *Reduce*: Apply an inverse production at the right end of the left string

If $E \rightarrow E + (E)$ is a production, then:

$$E + (E + (E) \uparrow) \Rightarrow E + (E \uparrow)$$
shift-reduce diagram
The Stack

- Left string can be implemented as a stack
- Top of the stack is the ▷
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-termianl on the stack (production lhs)
When to Shift vs. Reduce?

- Decide based on the left string (the stack)
- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The DFA input is the stack
  - The language consists of terminals and non-terminals
- We run the DFA on the stack and we examine the resulting state \( X \) and the token \( T \) after
  - If \( X \) has a transition labeled \( T \) then shift
  - If \( X \) is labeled with “\( A \rightarrow \beta \) on \( T \)” then reduce
LR(1) parsing diagram
Representing the DFA

- Parsers represent the DFA as a 2D table.
- Recall table-driven lexing
- Rows correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for terminals: action table
  - Those for non-terminals: goto table
representing the DFA diagram
LR Parsing Algorithm

- After a shift or reduce action, rerun the DFA on the entire stack
- Remember for each stack element to which state it brings the DFA
- LR parser maintains a stack
  \[<\text{sym}_1, \text{state}_1> \ldots <\text{sym}_n, \text{state}_n>\]
- \(\text{state}_k\) is the final state of the DFA on \(<\text{sym}_1 \ldots \text{sym}_k>\)
LR Parsing Algorithm

Let I = w$ be initial input
Let j = 0
Let DFA state 0 be the start state
Let stack = (dummy, 0)

repeat

case action[ top_state(stack),I[j] ] of

shift k: push (I[j++],k)
reduce X → α:

    pop | α | pairs,
    push (X, Goto[top_state(stack), X])
accept: halt normally
error: halt and report error
LR Parsing Notes

- Can parse more grammars than LL (next lecture)
- Most programming languages are LR
- Can be described as a simple table
- There are tools for building the table
- How is the table constructed?
Status

- Today 11:55pm: Project 1 Due
- Project 2 Due Friday, Apr. 24, 11:55pm