Bottom-Up Parsing II

Lecture 8

Review: Shift-Reduce Parsing

Bottom-up parsing uses two actions:

* **Shift**
  
  \[ ABC|xyz \Rightarrow ABC|xyz \]

* **Reduce**
  
  \[ Cb|lijk \Rightarrow CbA|lijk \]

Recall: The Stack

- Left string can be implemented by 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
  - pushes a non-terminal on the stack
  - production lhs

Key Issue

- How do we decide when to shift or reduce?
- Example grammar:
  
  - \[ E \rightarrow T + E | T \]
  - \[ T \rightarrow \text{int} \ast T | \text{int} | (E) \]
- Consider step \[ \text{int} | \ast \text{int} + \text{int} \]
  - We could reduce by \[ T \rightarrow \text{int} \]
    giving \[ T | \ast \text{int} + \text{int} \]
  - A fatal mistake!
    - No way to reduce to the start symbol \[ E \]

Handles

- Intuition: Want to reduce only if the result can still be reduced to the start symbol
- Assume a rightmost derivation
  
  \[ S \Rightarrow^* \alpha X \omega \Rightarrow \alpha \beta \omega \]
- Then \[ X \rightarrow \beta \] in the position after \[ \alpha \] is a handle of \[ \alpha \beta \omega \]

Handles (Cont.)

- Handles formalize the intuition
  - A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)
  - We only want to reduce at handles
- Note: We have said what a handle is, not how to find handles
Important Fact #2

Important Fact #2 about bottom-up parsing:

*In shift-reduce parsing, handles appear only at the top of the stack, never inside*

Why?

* Informal induction on # of reduce moves:
  * True initially, stack is empty
  * Immediately after reducing a handle
    - right-most non-terminal on top of the stack
    - next handle must be to right of right-most non-terminal, because this is a right-most derivation
    - Sequence of shift moves reaches next handle

Summary of Handles

* In shift-reduce parsing, handles always appear at the top of the stack
* Handles are never to the left of the rightmost non-terminal
  - Therefore, shift-reduce moves are sufficient; the need never move left
* Bottom-up parsing algorithms are based on recognizing handles

Recognizing Handles

* There are no known efficient algorithms to recognize handles
  * Solution: use heuristics to guess which stacks are handles
  * On some CFGs, the heuristics always guess correctly
    - For the heuristics we use here, these are the SLR grammars
    - Other heuristics work for other grammars

Grammars

| All CFGs | Unambiguous CFGs | SLR CFGs |

Viable Prefixes

* It is not obvious how to detect handles
  * At each step the parser sees only the stack, not the entire input; start with that . . .
* $\alpha$ is a viable prefix if there is an $\omega$ such that $\alpha|\omega$ is a state of a shift-reduce parser
Huh?

- What does this mean? A few things:
  - A viable prefix does not extend past the right end of the handle
  - It’s a viable prefix because it is a prefix of the handle
  - As long as a parser has viable prefixes on the stack no parsing error has been detected

Important Fact #3

Important Fact #3 about bottom-up parsing:

For any grammar, the set of viable prefixes is a regular language

Important Fact #3 (Cont.)

- Important Fact #3 is non-obvious
- We show how to compute automata that accept viable prefixes

Items

- An item is a production with a “.” somewhere on the rhs
- The items for \( T \rightarrow (E) \)
  - \( T \rightarrow (E) \)
  - \( T \rightarrow (.E) \)
  - \( T \rightarrow (E.) \)
  - \( T \rightarrow (E). \)

Items (Cont.)

- The only item for \( X \rightarrow e \) is \( X \rightarrow . \)
- Items are often called “LR(0) items”

Intuition

- The problem in recognizing viable prefixes is that the stack has only bits and pieces of the rhs of productions
  - If it had a complete rhs, we could reduce
- These bits and pieces are always prefixes of rhs of productions
Example

Consider the input (int)

- Then (E) is a state of a shift-reduce parse
- (E) is a prefix of the rhs of $T \to (E)$
  - Will be reduced after the next shift
- Item $T \to (E)$ says that so far we have seen (E of this production and hope to see )

---

Generalization

- The stack may have many prefixes of rhs’s
  - Prefix, Prefix\textsubscript{2} \ldots Prefi x\textsubscript{n}, Prefix\textsubscript{n}
- Let Prefix\textsubscript{i} be a prefix of rhs of $X_i \to \alpha_i$
  - Prefix\textsubscript{i} will eventually reduce to $X_i$
  - The missing part of $\alpha\textsubscript{i-1}$ starts with $X_i$
    - i.e. there is a $X_i \to$ Prefix\textsubscript{i-1} $X_i \beta$ for some $\beta$
- Recursively, Prefix\textsubscript{k+1} Prefix\textsubscript{n} eventually reduces to the missing part of $\alpha_k$

---

An Example

Consider the string (int * int):

- (int *| int) is a state of a shift-reduce parse
- "(" is a prefix of the rhs of $T \to (E)$
- "e" is a prefix of the rhs of $E \to T$
- "int *" is a prefix of the rhs of $T \to$ int * T

---

Recognizing Viable Prefixes

Idea: To recognize viable prefixes, we must

- Recognize a sequence of partial rhs’s of productions, where
- Each sequence can eventually reduce to part of the missing suffix of its predecessor

---

An NFA Recognizing Viable Prefixes

1. Add a dummy production $S' \to S$ to $G$
2. The NFA states are the items of $G$
   - Including the extra production
3. For item $E \to \alpha X \beta$ add transition
   - $E \to \alpha X \beta \rightarrow^X E \to \alpha X \beta$
4. For item $E \to \alpha X \beta$ and production $X \to \gamma$ add
   - $E \to \alpha X \beta \rightarrow^{\epsilon} X \rightarrow \gamma$

---
An NFA Recognizing Viable Prefixes (Cont.)

5. Every state is an accepting state

6. Start state is $S' \rightarrow S$
The states of the DFA are "canonical collections of items" or "canonical collections of LR(0) items".

The Dragon book gives another way of constructing LR(0) items.
Items Valid for a Prefix

An item $I$ is valid for a viable prefix $\alpha$ if the DFA recognizing viable prefixes terminates on input $\alpha$ in a state $s$ containing $I$.

The items in $s$ describe what the top of the item stack might be after reading input $\alpha$.

Valid Items Example

- An item is often valid for many prefixes.
- Example: The item $T \rightarrow (E)$ is valid for prefixes $((...$.

Valid Items for (((...

LR(0) Parsing

- Idea: Assume
  - stack contains $\alpha$
  - next input is $t$
  - DFA on input $\alpha$ terminates in state $s$
- Reduce by $X \rightarrow \beta$ if
  - $s$ contains item $X \rightarrow \beta$.
- Shift if
  - $s$ contains item $X \rightarrow \beta, t\omega$
  - equivalent to saying $s$ has a transition labeled $t$.

LR(0) Conflicts

- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
    - $X \rightarrow \beta$, and $Y \rightarrow \omega$.
- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
    - $X \rightarrow \beta$, and $Y \rightarrow \omega, t\delta$.
SLR

- LR = "Left-to-right scan"
- SLR = "Simple LR"
- SLR improves on LR(0) shift/reduce heuristics
  - Fewer states have conflicts

SLR Parsing

- Idea: Assume
  - stack contains $\alpha$
  - next input is $t$
  - DFA on input $\alpha$ terminates in state $s$
- Reduce by $X \rightarrow \beta$ if
  - $s$ contains item $X \rightarrow \beta$
- Shift if
  - $s$ contains item $X \rightarrow \beta, t$\no

SLR Parsing (Cont.)

- If there are conflicts under these rules, the grammar is not SLR
- The rules amount to a heuristic for detecting handles
  - The SLR grammars are those where the heuristics detect exactly the handles

SLR Conflicts

Follow(E) = { '(', $\}$
Follow(T) = { '*', ')', $\}$

No conflicts with SLR rules!

Precedence Declarations Digression

- Lots of grammars aren't SLR
  - including all ambiguous grammars
- We can parse more grammars by using precedence declarations
  - Instructions for resolving conflicts

Precedence Declarations (Cont.)

- Consider our favorite ambiguous grammar:
  - $E \rightarrow E \cdot E \mid E + E \mid (E) \mid \text{int}$
- The DFA for this grammar contains a state with the following items:
  - $E \rightarrow E \cdot E \mid E \rightarrow E \cdot E$
    - shift/reduce conflict!
- Declaring "*" has higher precedence than +" resolves this conflict in favor of reducing
Precedence Declarations (Cont.)

• The term "precedence declaration" is misleading.

• These declarations do not define precedence; they define conflict resolutions.
  - Not quite the same thing!

Naïve SLR Parsing Algorithm

1. Let M be DFA for viable prefixes of G
2. Let \([x_1...x_n]\) be initial configuration
3. Repeat until configuration is \(S\)$
   - Let \(\alpha\) be current configuration
   - Run \(M\) on current stack \(\alpha\)
     - If \(M\) rejects \(\alpha\), report parsing error
       - Stack \(\alpha\) is not a viable prefix
     - If \(M\) accepts \(\alpha\) with items \(I\), let \(a\) be next input
       - Shift if \(X \rightarrow \beta \cdot a \in I\)
       - Reduce if \(X \rightarrow \beta \in I\) and \(a \in \text{Follow}(X)\)
     - Report parsing error if neither applies

Notes

• If there is a conflict in the last step, grammar is not SLR(k)

• \(k\) is the amount of lookahead
  - In practice \(k = 1\)

SLR Example

<table>
<thead>
<tr>
<th>Configuration DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{int} \cdot \text{int}$)</td>
<td>1 shift</td>
</tr>
</tbody>
</table>

SLR Example

<table>
<thead>
<tr>
<th>Configuration DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{int} \cdot \text{int}$)</td>
<td>1 shift</td>
</tr>
<tr>
<td>(\text{int} \cdot \text{int}$)</td>
<td>3 shift</td>
</tr>
<tr>
<td>(\text{int} \cdot \text{int}$)</td>
<td>3 not in Follow(T) shift</td>
</tr>
</tbody>
</table>
SLR Example

**Configuration DFA Halt State Action**

| int * int$ | 1  | shift |
| int | * int$ | 3 | not in Follow(T) shift |
| int * int$ | 11 | shift |
SLR Example

Configuration DFA Halt State | Action
---|---
| int * int$ | 1 | shift |
| int | * int$ | 3 | * not in Follow(T) | shift |
| int * | int$ | 11 | shift |
| int * int | $ | 3 | $ ∈ Follow(T) | red. T→int |

Configuration int * int$ | 2
| S→E | 5 |
| E→T | 6 |
| E→T+E | 7 |
| T→E | 8 |
| T→int * T | 9 |
| T→(E) | 10 |

Configuration int * int$ | 2
| S→E | 5 |
| E→T | 6 |
| E→T+E | 7 |
| T→E | 8 |
| T→int * T | 9 |
| T→(E) | 10 |

SLR Example

Configuration DFA Halt State | Action
---|---
| int * int$ | 1 | shift |
| int | * int$ | 3 | * not in Follow(T) | shift |
| int * | int$ | 11 | shift |
| int * int | $ | 3 | $ ∈ Follow(T) | red. T→int |
| int * T | $ | 4 | $ ∈ Follow(T) | red. T→int*T |
### Configuration \( \text{int} \times T \$

1. \( S \rightarrow \text{E} \)
2. \( \text{E} \rightarrow \text{T} \)
3. \( \text{E} \rightarrow \text{T} \)
4. \( \text{T} \rightarrow \text{int} \times \text{T} \)
5. \( \text{T} \rightarrow \text{int} \times \text{T} \)
6. \( \text{T} \rightarrow \text{int} \times \text{T} \)
7. \( \text{T} \rightarrow \text{int} \times \text{T} \)
8. \( \text{T} \rightarrow \text{int} \times \text{T} \)
9. \( \text{T} \rightarrow \text{int} \times \text{T} \)
10. \( \text{T} \rightarrow \text{int} \times \text{T} \)

### Configuration \( \text{int} \times T \$

1. \( S \rightarrow \text{E} \)
2. \( \text{E} \rightarrow \text{T} \)
3. \( \text{E} \rightarrow \text{T} \)
4. \( \text{T} \rightarrow \text{int} \times \text{T} \)
5. \( \text{T} \rightarrow \text{int} \times \text{T} \)
6. \( \text{T} \rightarrow \text{int} \times \text{T} \)
7. \( \text{T} \rightarrow \text{int} \times \text{T} \)
8. \( \text{T} \rightarrow \text{int} \times \text{T} \)
9. \( \text{T} \rightarrow \text{int} \times \text{T} \)
10. \( \text{T} \rightarrow \text{int} \times \text{T} \)

### Configuration \( \text{int} \times T \$

1. \( S \rightarrow \text{E} \)
2. \( \text{E} \rightarrow \text{T} \)
3. \( \text{E} \rightarrow \text{T} \)
4. \( \text{T} \rightarrow \text{int} \times \text{T} \)
5. \( \text{T} \rightarrow \text{int} \times \text{T} \)
6. \( \text{T} \rightarrow \text{int} \times \text{T} \)
7. \( \text{T} \rightarrow \text{int} \times \text{T} \)
8. \( \text{T} \rightarrow \text{int} \times \text{T} \)
9. \( \text{T} \rightarrow \text{int} \times \text{T} \)
10. \( \text{T} \rightarrow \text{int} \times \text{T} \)

### Configuration \( \text{int} \times T \$

1. \( S \rightarrow \text{E} \)
2. \( \text{E} \rightarrow \text{T} \)
3. \( \text{E} \rightarrow \text{T} \)
4. \( \text{T} \rightarrow \text{int} \times \text{T} \)
5. \( \text{T} \rightarrow \text{int} \times \text{T} \)
6. \( \text{T} \rightarrow \text{int} \times \text{T} \)
7. \( \text{T} \rightarrow \text{int} \times \text{T} \)
8. \( \text{T} \rightarrow \text{int} \times \text{T} \)
9. \( \text{T} \rightarrow \text{int} \times \text{T} \)
10. \( \text{T} \rightarrow \text{int} \times \text{T} \)

### SLR Example

<table>
<thead>
<tr>
<th>Configuration DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{int} \times \text{int}$</td>
<td>shift</td>
</tr>
<tr>
<td>( \text{int} \times \text{int}$</td>
<td>shift</td>
</tr>
<tr>
<td>( \text{int} \times \text{int}$</td>
<td>shift</td>
</tr>
<tr>
<td>( \text{int} \times \text{int}$</td>
<td>red, T→int</td>
</tr>
<tr>
<td>( \text{int} \times \text{T}$</td>
<td>red, T→int*T</td>
</tr>
<tr>
<td>( \text{T}$</td>
<td>red, E→T</td>
</tr>
</tbody>
</table>

### Configuration \( \text{T}\$

1. \( S \rightarrow \text{E} \)
2. \( \text{E} \rightarrow \text{T} \)
3. \( \text{E} \rightarrow \text{T} \)
4. \( \text{T} \rightarrow \text{int} \times \text{T} \)
5. \( \text{T} \rightarrow \text{int} \times \text{T} \)
6. \( \text{T} \rightarrow \text{int} \times \text{T} \)
7. \( \text{T} \rightarrow \text{int} \times \text{T} \)
8. \( \text{T} \rightarrow \text{int} \times \text{T} \)
9. \( \text{T} \rightarrow \text{int} \times \text{T} \)
10. \( \text{T} \rightarrow \text{int} \times \text{T} \)
Notes

- Skipped using extra start state $S'$ in this example to save space on slides
- Rerunning the automaton at each step is wasteful
  - Most of the work is repeated

An Improvement

- Remember the state of the automaton on each prefix of the stack
- Change stack to contain pairs $(\text{Symbol, DFA State})$

An Improvement (Cont.)

- For a stack $(\text{sym}_1, \text{state}_1) \ldots (\text{sym}_n, \text{state}_n)$
  $\text{state}_n$ is the final state of the DFA on $\text{sym}_1 \ldots \text{sym}_n$
- Detail: The bottom of the stack is $(\text{any}, \text{start})$
  - any is any dummy symbol
  - start is the start state of the DFA

Goto Table

- Define $\text{goto}[i, A] = j$ if state, $A$ state
- $\text{goto}$ is just the transition function of the DFA
  - One of two parsing tables
Refined Parser Moves

- **Shift x**
  - Push (a, x) on the stack
  - a is current input
  - x is a DFA state

- **Reduce X → α**
  - As before

- **Accept**
- **Error**

---

Action Table

For each state s, and terminal a

- If s has item X → α, a, and goto[i,a] = j then
  - action[i,a] = shift j

- If s has item X → α, and a ∈ Follow(X) and X ≠ S’ then
  - action[i,a] = reduce X → α

- If s has item S’ → S, then
  - action[i,$] = accept
- Otherwise, action[i,a] = error

---

SLR Parsing Algorithm

Let I = w$ be initial input
Let i = 0
Let DFA state i have item S’ → S
Let stack = (dummy, i)
repeat
  case action[top_state(stack), I[j]] of
    shift k:  push (I[j++], k)
    reduce X → A:
    pop |A| pairs,
    push (X, goto[top_state(stack), X])
    accept: halt normally
    error: halt and report error

---

Notes on SLR Parsing Algorithm

- Note that the algorithm uses only the DFA states and the input
  - The stack symbols are never used!

- However, we still need the symbols for semantic actions

---

More Notes

- Some common constructs are not SLR(1)

- LR(1) is more powerful
  - Build lookahead into the items
  - An LR(1) item is a pair: LR(0) item x lookahead
  - [T → . int * T, $] means
    - After seeing T → int * T reduce if lookahead is $
  - More accurate than just using follow sets
  - Take a look at the LR(1) automaton for your parser!