Bottom-Up Parsing II

Lecture 8

Review: Shift-Reduce Parsing

Bottom-up parsing uses two actions:

**Shift**

\[ ABC|xyz \Rightarrow ABC|x|yz \]

**Reduce**

\[ Cbxy|ijk \Rightarrow CbA|ijk \]

Recall: The Stack

- Left string can be implemented by a stack
  - Top of the stack is the | (vertical bar)
- Shift pushes a terminal on the stack
- Reduce pushes a non-terminal on the stack
  - Production rhs
  - Production lhs

Key Issue

- How do we decide when to shift or reduce?
- Example grammar:
  \[
  \begin{align*}
  E &\rightarrow T + E | T \\
  T &\rightarrow \text{int} * T | \text{int} | (E)
  \end{align*}
  \]
- Consider step \[ \text{int} | * \text{int} + \text{int} \]
  - We could reduce by \[ T \rightarrow \text{int} \] giving \[ T | * \text{int} + \text{int} \]
  - 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
  \[ S \rightarrow^{*} \alpha X_0 \rightarrow \alpha \beta_0 \]
- Then \( X \rightarrow \beta \) in the position after \( \alpha \) is a handle of \( \alpha \beta_0 \)

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
  - Will generate conflicts
- 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 $T \rightarrow (E)$ are $T \rightarrow (E)$, $T \rightarrow (.E)$, $T \rightarrow (E.)$, $T \rightarrow (E)$.

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 → (E)
  - Will be reduced after the next shift
- Item T → (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_1 Prefix_2 ... Prefix_n
- Let Prefix_i be a prefix of rhs of X_i → α_i
  - Prefix_i will eventually reduce to X_i
  - The missing part of α_i-1 starts with X_i
  - i.e. there is a X_i → Prefix_i X_i β for some β
- Recursively, Prefix_i-1 Prefix_i eventually reduces to the missing part of α_k

An Example

Consider the string (int * int):
( (int | int)

“(” is a prefix of the rhs of T → (E)
“ε” is a prefix of the rhs of E → T
“int *” is a prefix of the rhs of T → int * T

An Example (Cont.)

The “stack of items”
T → (E)
E → .T
T → int * .T

Says
We’ve seen “(” of T → (E)
We’ve seen ε of E → T
We’ve seen int * of T → 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’ → S to G
2. The NFA states are the items of G
   - Including the extra production
3. For item E → α.Xβ add transition
   E → α.Xβ →^X E → α.Xβ
4. For item E → α.Xβ and production X → γ add
   E → α.Xβ →^X X → γ
An NFA Recognizing Viable Prefixes (Cont.)

5. Every state is an accepting state
6. Start state is $S' \rightarrow S$
NFA for Viable Prefixes in Detail (5)

NFA for Viable Prefixes in Detail (6)

NFA for Viable Prefixes in Detail (7)

NFA for Viable Prefixes in Detail (8)

NFA for Viable Prefixes in Detail (9)

NFA for Viable Prefixes in Detail (10)
Lingo

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

Valid Items

Item \( X \rightarrow \beta \gamma \) is valid for a viable prefix \( \alpha \beta \) if
\[ S' \rightarrow^* \alpha X \omega \rightarrow \alpha \beta \gamma \omega \]
by a right-most derivation

After parsing \( \alpha \beta \), the valid items are the
possible tops of the stack of 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
  \[
  (\quad)
  \]
  \[
  (\quad)
  \]
  \[
  (\quad)
  \]
  \[
  \ldots
  \]

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 $X \rightarrow \beta, t, \omega$

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

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

SLR Conflicts

Follow(E) = { ‘)’, $ } 
Follow(T) = { ‘*’, ‘)’, $ }

No conflicts with SLR rules!

Precedence Declarations (Cont.)

- Consider our favorite ambiguous grammar:
  - $E \rightarrow E + E | E * E | (E) | \text{int}$
- The DFA for this grammar contains a state with the following items:
  - $E \rightarrow E + E \quad E \rightarrow E * E \quad E \rightarrow (E) \quad E \rightarrow \text{int}$
  - 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 $\alpha|\omega$ 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 $X \rightarrow \beta \in I$
       - 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</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{int} * \text{int}$</td>
<td>1</td>
<td>shift</td>
</tr>
</tbody>
</table>

SLR Example

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

<table>
<thead>
<tr>
<th>Configuration DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$ 1</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>*</td>
</tr>
<tr>
<td>int *</td>
<td>int$ 11</td>
</tr>
</tbody>
</table>

Configuration int * int$

Configuration int * int$

Configuration int * int$

Configuration int * int$
### 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 $ in Follow(T) red, T→int

---

### Configuration int * int

**Configuration DFA Halt State**  
**Action**

- `| int * int $` 1 shift
- `int * int | $` 3 * not in Follow(T) shift
- `int * int |` 11 shift
- `int * int | $` 3 $ in Follow(T) red, T→int

---

### 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 $ in Follow(T) red, T→int
- `int * T | $` 4 $ in Follow(T) red, T→int*T
SLR Example

<table>
<thead>
<tr>
<th>Configuration DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>* int$</td>
</tr>
<tr>
<td>int * int$</td>
<td>shift</td>
</tr>
<tr>
<td>int * int</td>
<td>$</td>
</tr>
<tr>
<td>int * T</td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>$</td>
</tr>
</tbody>
</table>
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
  - $(\text{Symbol}, \text{DFA State})$

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 $(\text{any}, \text{start})$
  - $\text{any}$ is any dummy symbol
  - $\text{start}$ is the start state of the DFA

Goto Table

- Define $\text{goto}[i, A] = j$ if state, $\rightarrow^A \text{state}_j$
- $\text{goto}$ 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** \(X \rightarrow \alpha\)
  - As before
- **Accept**
- **Error**

Action Table

For each state \(s_i\) and terminal \(a\)
- If \(s_i\) has item \(X \rightarrow \alpha . a \beta\) and \(\text{goto}[i,a] = j\) then
  - \(\text{action}[i,a] = \text{shift} j\)
- If \(s_i\) has item \(X \rightarrow \alpha\) and \(a \in \text{Follow}(X)\) and \(X \neq S'\) then
  - \(\text{action}[i,a] = \text{reduce} X \rightarrow \alpha\)
- If \(s_i\) has item \(S' \rightarrow S\) then
  - \(\text{action}[i,$$] = \text{accept}\)
- Otherwise, \(\text{action}[i,a] = \text{error}\)

SLR Parsing Algorithm

Let \(I = w\) be initial input
Let \(j = 0\)
Let DFA state \(i\) have item \(S' \rightarrow .S\)
Let stack = \(\langle \text{dummy}, 1 \rangle\)
repeat
  case \(\text{action}[\text{top_state}(\text{stack}), I[j]]\) of
  shift \(k\): push \((I[j++], k)\)
  reduce \(X \rightarrow \alpha\): pop \(|\alpha|\) pairs,
  push \((X, \text{goto}[\text{top_state}(\text{stack}), X])\)
  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 \rightarrow \text{int} * T, \$_\] means
    - After seeing \(T \rightarrow \text{int} * T\) reduce if lookahead is \$_\)
  - More accurate than just using follow sets
  - Take a look at the LR(1) automaton for your parser!