LR Parsing

CS 440: Programming Languages and Translators, Spring 2019
Lecture 15, Wed Mar 13

Notation

* $a, b, \ldots \in T$ (a and b stand for members of T; a, b, \ldots are members of T) Was $\Sigma$ for regular expressions.
* $x, y, \ldots \in T^*$
* $A, B, \ldots \in V$
* $W, X, \ldots \in T \mid V$
* $\alpha, \beta, \ldots \in (T \mid V)^*$ Sentential form

I. Intro

* In bottom-up parsing, we build the trees nearest the terminal characters of input string first, then trees higher up, making our way to the root.
* Bottom-up parsing generates a tree in a reverse rightmost derivation. Usually use shift-reduce parsing for this
* Shift-reduce parsers are the kind of bottom-up parser normally seen when people talk about "bottom-up parsing".
* A shift-reduce parser a stack to hold the symbols we've seen but haven't completely parsed.
* Parser has two basic operations
  * Shift - the lookahead symbol onto the stack of unprocessed items
  * Reduce - The top of the stack matches the rhs of a rule; pop them off and push on the lhs nonterminal.
* Differences between shift-reduce parsers
  * How and when to reduce
  * How do we handle conflicts between shifting and reduction or between two reductions

LR Parsing

* In an LR parser, in addition to the stack of things already parsed, the parser maintains a state.
  * The state summarizes the context we're in (what we know about the input seen/parsed so far)
  * The state is used as part of deciding when to shift and reduce.
* Whenever we shift a symbol onto the stack, we also include the current state with it.
* This way, when some initial part of the stack is popped off for a reduce operation, the exposed old state tells us what we were thinking when we came across the first symbol of input that was generated by the popped-off rhs.
  * For example, if we start a parse for $A$ using $A \rightarrow B C$ and pause to parse $B \rightarrow \beta$, then when we reduce the $\beta$ to get $B$, we have to know that we want to go back to $A \rightarrow B C$. This is done by having the $A \rightarrow B C$ state (along with the info that we paused for $B$) on the stack.
  * (It's similar to having a return address for a subroutine call.)
Implications of having an LR parser state

* The state of an LR parser is like the state of a finite automaton, so we can easily parse regular languages.
  * But since an LR parser has a stack, it can parse languages that a finite automaton can't.
  * Technically, an LR parser is a kind of PDA (Push-Down Automaton).
  * The class of PDAs can parse any context-free grammar, but the class of LR parsers parses a smaller set than that (ones with deterministic context-free grammars).

LR(0) Items as States

* In LR parsing, a state keeps track of where we think we are in the parsing of some rhs of a rule.
  * Specifically, a state corresponds to an LR(0) item.
  * The format of an LR(0) item is that it's a grammar rule where a special dot symbol (typically •) indicates where we think we are. As we find the items on the rhs, we move the dot rightwards. When the dot reaches the end of the rule, we can reduce.
  * E.g., for the rule $A \rightarrow B \ C$, we have three LR(0) items
    
    - $A \rightarrow • B \ C$ (we're looking for a $B$)
    - $A \rightarrow B • C$ (we've found a $B$ and now are looking for a $C$)
    - $A \rightarrow B C •$ (we've found the $C$ and can reduce)

* To reduce, we pop off the $C$ item and $B$ item from the stack and push $A$ onto the stack.
  * If the parse is producing a parse tree, this is where we'd build a tree for $A \rightarrow B \ C$.
  * In general, for a rule $A \rightarrow \alpha$, if $\alpha$ is a sequence of $n$ symbols, we get $n+1$ LR(0) items from the rule.
    * The exception is $A \rightarrow \epsilon$: the items $A \rightarrow • \epsilon$ and $A \rightarrow \epsilon •$ are equivalent.
  * The reason for pushing the state onto the stack along with the unprocessed items comes from how we have to jump from one kind of LR(0) item to another as we look for different kinds of right-hand sides.
  * E.g., take $A \rightarrow • B \ C$. Say we look for a $B$ using the rule a rule $B \rightarrow x$.
    * Before we see the $x$, we're in state $B \rightarrow • x$ and after we see the $x$, we're in state $B \rightarrow x •$.
    * If we reduce using $B \rightarrow x •$ (by popping off $x$ and pushing on $B$), we need to go back to the $A \rightarrow • B C$ state and advance to $A \rightarrow B • C$.
    * The state $A \rightarrow • B C$ is pushed onto the stack so that we know that's what we were trying to parse when we went off searching for a $B$, as opposed to some arbitrary other rule that uses a $B$, say $D \rightarrow z B D$.
    * This lets us avoid having to use states that keep track of all the proposed decisions we've made so far in the parse. In stack terms, only the top part of the stack is important; anything under $B C$ is irrelevant for purposes of deciding to reduce using $A \rightarrow B C$. 
* How do we find the states and transitions for a shift-reduce parser?

* _First_ look at nondeterministic version, then go to deterministic version.

* For every grammar rule $A \rightarrow X \ y \ Z$ take its LR(0) items and make them states of the parser.
  
  * E.g., $S \rightarrow \bullet X \ y \ Z$ and $S \rightarrow X \ \bullet \ y \ Z$ and $S \rightarrow X \ y \ \bullet \ Z$ and $S \rightarrow X \ y \ Z \ \bullet$
  
  * Link each state to the next via a transition labeled with the symbol to the right of the dot.

  * E.g., $(S \rightarrow \bullet X \ y \ Z) \rightarrow_{x} (S \rightarrow X \ \bullet \ y \ Z) \rightarrow_{y} (S \rightarrow X \ y \ \bullet \ Z) \rightarrow_{z} (S \rightarrow X \ y \ Z \ \bullet)$

  * The last item (with the dot at the end) ends the chain. If we get to this state we reduce.

* Take each state that has a dot next to a nonterminal and add an $\varepsilon$ transition to the initial state for that nonterminal.
  
  * E.g., state $(A \rightarrow ... \ \bullet \ \bullet ...)$ gets an $\varepsilon$ transition to every state $(B \rightarrow \bullet \ \alpha)$

* **Example:** $S \rightarrow a \ A$, $A \rightarrow c$

  * Initially we have $(S \rightarrow \bullet \ a \ A) \rightarrow_{a} (S \rightarrow a \ \bullet \ A) \rightarrow_{A} (S \rightarrow a \ A \ \bullet)$

    
    ↓ $\varepsilon$

    Need to add $\varepsilon$ transition

    $(A \rightarrow \bullet \ c) \rightarrow_{c} (A \rightarrow c \ \bullet)$

* So parser action will look like

<table>
<thead>
<tr>
<th>Stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a c</td>
<td>Shift a</td>
</tr>
<tr>
<td>a c</td>
<td>c</td>
<td>Shift c</td>
</tr>
<tr>
<td>a c</td>
<td>$\varepsilon$</td>
<td>Reduce via $A \rightarrow c \ \bullet$</td>
</tr>
<tr>
<td>a A</td>
<td>$\varepsilon$</td>
<td>Reduce via $S \rightarrow a \ A \ \bullet$</td>
</tr>
<tr>
<td>S</td>
<td>$\varepsilon$</td>
<td>Success!</td>
</tr>
</tbody>
</table>

* For a parse to be successful, we have to have empty input and a stack with just the start symbol.

  * If the input is too short, the stack isn't just the start symbol when we end.
  
  * If the input is too long, there's input remaining when we get to a stack with just the start symbol.

* Notice nondeterminism about whether we wanted $(S \rightarrow a \ \bullet \ A)$ or $(A \rightarrow \bullet \ c)$ was resolved by looking at next input. Since $c \not\in \text{First}(A)$, we can't use $S \rightarrow a \ \bullet \ A$ and must use $A \rightarrow \bullet \ c$ instead.

* Make the PDA deterministic

  * Go from nondeterministic automaton to deterministic automaton using the same technique as for the NFA to DFA change: A state in the new machine is a set of states in the old machine.

  * If you have an item with an $\varepsilon$ transition to another item, then both items go into one new state.

  $$(S \rightarrow \bullet \ a \ A) \rightarrow_{a} (S \rightarrow a \ \bullet \ A) \rightarrow_{A} (S \rightarrow a \ A \ \bullet)$$

    ↓ $\varepsilon$

    $$(A \rightarrow \bullet \ c) \rightarrow_{c} (A \rightarrow c \ \bullet)$$

becomes

$${(S \rightarrow \bullet \ a \ A)} \rightarrow_{a} {(S \rightarrow a \ \bullet \ A, A \rightarrow \bullet \ c)} \rightarrow_{A} {S \rightarrow a \ A \ \bullet}$$
When parsing, what do we do when we hit $A \rightarrow c \bullet$? We reduce, but then where do we go?

* Have to keep track of which state we were in when we took the $\rightarrow c$ path.

* Stack will have state numbers in addition to symbols.

Parse looks like

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$a \ c$</td>
<td>Shift a</td>
</tr>
<tr>
<td>0 a 1</td>
<td>$c$</td>
<td>Shift c</td>
</tr>
<tr>
<td>0 a 1 c 2</td>
<td>$\varepsilon$</td>
<td>Reduce via $A \rightarrow c \bullet$</td>
</tr>
<tr>
<td>0 a 1 A 3</td>
<td>$\varepsilon$</td>
<td>Reduce via $S \rightarrow a A$</td>
</tr>
<tr>
<td>0 S 4</td>
<td>$\varepsilon$</td>
<td>Accept</td>
</tr>
</tbody>
</table>

State 0: $\{ S \rightarrow \bullet a A \}$
State 1: $\{ S \rightarrow a \bullet A, A \rightarrow \bullet c \}$
State 2: $\{ A \rightarrow c \bullet \}$
State 3: $\{ S \rightarrow a A \bullet \}$
State 4: $\{ S \bullet \}$
State 5: $\{ S' \rightarrow S \bullet \}$

Difficult making sure we stop when we hit $S \rightarrow \ldots$. Also, there can be more than one $S \rightarrow \text{rhs1} \lor \text{rhs2} \ldots$ rule.

Add a new start symbol $S'$ to the grammar $S' \rightarrow S \$ as its only rule, $\$ = end of input. (Same as in LL parse)

For parse to succeed, we need to end with an empty stack and empty input otherwise we have too much or too little input to parse correctly.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S'$ 0</td>
<td>$a \ c $</td>
<td>Shift a</td>
</tr>
<tr>
<td>$S'$ 0 a 1</td>
<td>$c $</td>
<td>Shift c</td>
</tr>
<tr>
<td>$S'$ 0 a 1 c 2</td>
<td>$$</td>
<td>Reduce via $A \rightarrow c \bullet$</td>
</tr>
<tr>
<td>$S'$ 0 a 1 A 3</td>
<td>$$</td>
<td>Reduce via $S \rightarrow a A \bullet$</td>
</tr>
<tr>
<td>$S'$ 0 S 4</td>
<td>$$</td>
<td>Shift $$</td>
</tr>
<tr>
<td>$S'$ 0 S' 6</td>
<td>$\varepsilon$</td>
<td>Accept</td>
</tr>
</tbody>
</table>

$S' \rightarrow S \$ Accept

Note we can have shift on one symbol to one state and shift on the same symbol to a different state

* $\{ C \rightarrow \alpha \bullet d \beta, D \rightarrow \alpha \bullet d \delta \} \rightarrow_d \{ C \rightarrow \alpha \bullet d \beta, D \rightarrow \alpha \bullet d \delta \}$

* We didn't know which rule we were parsing, shifted a d, and we still don't know which rule we're in.

* It's important that both the $C \rightarrow \alpha \bullet d \beta$ and $D \rightarrow \alpha \bullet d \delta$ items be in the target state because we don't know.

* If we try one transition $\{ C \rightarrow \alpha \bullet d \beta, D \rightarrow \alpha \bullet d \delta \} \rightarrow_d \{ C \rightarrow \alpha \bullet d \beta \}$ along with $\rightarrow_d \{ D \rightarrow \alpha \bullet d \delta \}$, we'd still have a nondeterministic automaton.
When to reduce?

* **SLR - Simple LR parser** - uses the simplest technique: "Reduce when the next terminal symbol is in the \textit{Follow} set for this nonterminal".

  * E.g., we're in state $A \to \alpha \bullet$ and the next input symbol is $b$ and $b \in \text{Follow}(A)$, so we reduce.
  * Advantage: we can reuse the First and Follow set ideas from LL parsing. The question $? \in \text{Follow}(A)$ is easy to answer.

  * Might provide a way to arbitrate reduce/reduce conflicts, e.g., if you're in a state like $\{D \to \alpha \bullet, \ E \to \beta \bullet\}$:

    * Look at $\text{Follow}(D)$ and $\text{Follow}(E)$. If they're disjoint, then you can use the next input character to decide which rule to reduce which. (If $\text{Follow}(D) \cap \text{Follow}(E) \neq \emptyset$, then if the next input character is in that set, you still have a reduce/reduce conflict.)

* **The problem with the SLR parser** is that it might be too eager to reduce. Maybe you're not in a context where the next input character can follow this particular use of the nonterminal in the parse.

  * Example: Grammar $S \to A \ b \ A \ c$, $A \to a$

    * On input $a \ b \ a \ c$, SLR parsing works fine. (shift $a$, reduce using $A \to a$, shift $b$, shift $a$, reduce using $A \to a$, shift $c$, reduce using $S \to \ldots$). $\text{Follow}(A) = \{b, c\}$

    * On input $a \ c \ a \ c$, we'll reduce the first $a$ to $A$ even though there's no way to continue the parse with $S \to A \bullet b \ A \ c$ and input $c \ a \ c$. You'll eventually end with a stack that contains $A \ c \ A \ c$ and nothing to reduce it with, so at the end of the parse you'll generate an error message. It's at the end of the parse, so it could be far away from the actual error.

  * More complicated rules for deciding to reduce cover more cases but make for a more complicated automaton (and parsing table).