LR Parsing, part 2
CS 440: Programming Languages and Translators, Spring 2019
Lecture 16, Mon Mar 25

Notation
* a, b, ... ∈ T (a and b stand for members of T; a, b, ... are members of T) Was Σ for regular expressions.
* x, y, ... ∈ T*
* A, B, ... ∈ V
* W, X, ... ∈ T | V
* α, β, ... ∈ (T | V)*  Sentential form

Review
* LR parsing uses shift-reduce parsing. An LR parser is a finite state automaton plus a stack that holds symbols that haven’t yet been completely parsed interleaved with the parser states we were at when we pushed the symbol onto the stack. The stack states get used when we reduce (replace symbols at the top of the stack with the nonterminal that generates them), to get back to the state we were at before looking for the nonterminal we just found. So that we can be sure we’ve reached the exact end of a derivation, we add a rule S’ → S $ where S’ is the new start symbol, S is the original start symbol, and $ is a new terminal symbol that indicates end of input.
* LR parsing uses LR(0) items as states: Each item is a grammar rule where a • indicates where we think we are in the rule. The dot begins at the left end of the rhs of the rule and moves right as we find symbols; we reduce when the dot reaches the end of the rule. E.g., there’s a transition on a for (A → α • a β) to (A → α a • β)
* The parser automaton is nondeterministic because every state of the form (A → ... • B ....) gets an ε-transition to every state of the form (B → • α). We get an equivalent deterministic automaton by using the usual NFA → DFA transformation.
* A small example from last time: [this is from the updated version of Lecture 15; the old one had S’ as the original top nonterminal] The rules are S’ → S $, S → a A, A → c.

<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 State 0: {S’ → • S $, S → • a A}</td>
</tr>
<tr>
<td>0 a 1</td>
<td>c $</td>
<td>Shift c State 1: {S → a • A, A → • c}</td>
</tr>
<tr>
<td>0 a 1 c 2</td>
<td>$</td>
<td>Reduce via A → c • State 2: {A → c •}</td>
</tr>
<tr>
<td>0 a 1 A 3</td>
<td>$</td>
<td>Reduce via S → a A • State 3: {S → a A •}</td>
</tr>
<tr>
<td>0 S 4</td>
<td>$</td>
<td>Shift $ State 4: {S’ → S • $}</td>
</tr>
<tr>
<td>0 S 4 $ 5</td>
<td>ε</td>
<td>Reduce via S’ → S $ State 5: {S’ → S $ •}</td>
</tr>
<tr>
<td>0 S’ 6</td>
<td>ε</td>
<td>Accept, since input is ε</td>
</tr>
</tbody>
</table>

* We ended by starting to look at the question of when to do a reduction.
**LR(0) Parser**

* The zero in LR(0) indicates we don't use any lookahead at all. As a result, we always reduce when we reach the end of a rule. This makes for an extremely simple parser, but the set of grammars that can be parsed this way is relatively small.

* Consider the grammar $S' \rightarrow S \$, $S \rightarrow A \ a$, $S \rightarrow B \ b$, $A \rightarrow c$, $B \rightarrow c$ [make this SLR(1) but not LR(0) Mon 3/25, 15:44]

  * Activity question: What is the LR automaton for this grammar?

  * If we have input $c \ x$ (where $x$ is $a$ or $b$) then after we shift the $c$, we're ready to reduce $A \rightarrow c \ •$ and also $B \rightarrow c \ •$. In an LR(0) parser, we have a reduce-reduce conflict because we have no lookahead information to tell us that $x$ gives us information about which rule to reduce with.

**SLR(1) - Simple LR parser**

* A Simple LR parser\(^1\) gets to use one symbol of lookahead to eliminate the reduce-reduce conflict.

  * Uses the simplest technique: "Reduce when the next terminal symbol is in the Follow set for this nonterminal".

  * Since $Follow(A) = \{a\}$ and $Follow(B) = \{b\}$, we can use $x = a$ or $x = b$ on input $c \ x$ to tell us which rule to reduce with. (If $x$ is neither of these, we can generate an error message.)

  * More generally, we still have a reduce-reduce conflict on the symbols in $Follow(A) \cap Follow(B)$. If that set is empty, we've eliminated the conflict.

  * Advantage: we can reuse the First and Follow set ideas from LL parsing. The question $? \in Follow(A)$ is easy to answer.

  * More grammars are SLR than LR(0), but many grammars are LR(1) without being SLR. The problem is that in general, the lookahead character can depend on how you got to the rule you want to reduce. We had an example at the end of last time: For Grammar $S \rightarrow A \ b \ A \ c$, $A \rightarrow a$, $Follow(A) = \{b, c\}$, so one SLR state is $\{A \rightarrow a \ •\}$. On input $a \ c \ ...$, we got here by shifting $a$. With an SLR parser, since the lookahead character $c$ is in $Follow(A)$, we'll reduce and continue on from state $\{S \rightarrow A \ • \ b \ A \ c\}$, but with input $c$, our parse will fail because we don't have a $b$ to transition to $\{S \rightarrow A \ b \ • \ A \ c\}$.

**Canonical LR(1) Parser**

* The opposite of an SLR(1) parser is the canonical LR(1) parser (abbreviated CLR(1) or just LR(1)).

  * In a CLR parser, each LR(0) item for a state gets annotated with a lookahead character that tells us it's okay to reduce.

  * Going back to the grammar $S \rightarrow A \ b \ A \ c$, $A \rightarrow a$, we get two versions of the $A \rightarrow a$ rule:\(^2\)

    * $A \rightarrow \ • \ a\$, $b$ and $A \rightarrow a \ •$, $b$ and

\(^1\) Plain "SLR" means "SLR(1)". In the unlikely case that you mean, say, SLR(2), you have to leave in the (2).

\(^2\) I've left out the top rule $S' \rightarrow S \$ because we don't reduce at the end of it, we accept the input.
* \( A \to \bullet a, c \) and \( A \to a \bullet, c \)

* There are transitions
  * \( S \to \bullet A b A c, \$ \) with \( \epsilon \)-transition to \( A \to \bullet a, b \) and
  * \( S \to A b \bullet A c, \$ \) with \( \epsilon \)-transition to \( A \to \bullet a, c \)

* On correct input \( a b a c \$ \), the nondeterministic automaton goes through the sequence of states
  * \( S' \to \bullet S \$ \) (take \( \epsilon \) transition)
  * \( S \to \bullet A b A c, \$ \) (take \( \epsilon \) transition)
  * \( A \to \bullet a, b \) (shift since next character \( a \))
  * \( A \to a \bullet, b \) (reduce, since next character is \( b \))
  * \( S \to A \bullet b A c, \$ \) (shift next character \( b \))
  * \( S \to A b \bullet A c, \$ \) (take \( \epsilon \) transition)
  * \( A \to \bullet a, c \) (shift next character \( a \))
  * \( A \to a \bullet, c \) (reduce, since next character is \( c \))
  * \( S \to A b A c \bullet, \$ \) (reduce, since next character is \$ \)
  * \( S' \to S \bullet \$ \) (accept)

* On incorrect input \( a c \ldots \), the automaton is stuck at state \( A \to a \bullet, b \) because the next character is \( c \), not \( b \). We can generate an error message.

* The initial DFA state would have the four NFA states
  * \{ \( S' \to \bullet S \$ \); \( S \to \bullet A b A c, \$ \); \( A \to \bullet a, b \); \( A \to a \bullet, c \) \}

* If we see an \( a \), we take a transition to \{ \( A \to a \bullet, b \); \( A \to a \bullet, c \) \}

**Building CLR(1) Parsing Transitions**

* An LR(0) item has the form \( A \to \alpha \bullet \beta \) where \( \alpha \) or \( \beta \) are (possibly missing) sequences of terminals and/or nonterminals. (If both \( \alpha \) and \( \beta \) are missing, we write \( A \to \bullet \epsilon \) and \( A \to \epsilon \bullet \))

* A CLR(1) item is similar but includes an expected lookahead character: \( A \to \alpha \bullet \beta, a \) where \( a \) is a terminal.

* The starting state is \( S' \to \bullet S \$ \) (with no lookahead; a special case)
  * For an item \( A \to \alpha \bullet b \beta, a \), on input \( b \) we transition to \( A \to \alpha b \bullet \beta, a \). (Recall \( \alpha \) and/or \( \beta \) can be missing.)
  * For an item \( A \to \alpha \bullet B \beta, a \), for every character \( c \in \text{First}(\beta) \), we create a state \( B \to \bullet \gamma, c \) and add an \( \epsilon \)-transition to it from \( A \to \alpha \bullet B \beta, a \).
    * If \( \epsilon \in \text{First}(\beta) \), we add a transition \( B \to \bullet \gamma, a \). (Recall \( a \) was the lookahead in the item \( A \to \alpha \bullet B \beta, a \).)
    * A shorter way to say this is to look at \( c \in \text{First}(\beta) a \); this includes characters in \( \text{First}(\beta) \), and if \( \beta \to \epsilon \), we also get \( a \in \text{First}(\beta) a \).

v. Tue 3/26, 18:00

© James Sasaki, 2019
This process is called "splitting the LR(0) states". In the worst case, if we have \( n \) terminal symbols in the language and \( r \) for the size of the set of all LR(0) items, then there can be \( n \times r \) CLR(1) items.

* Note \( r \) can be pretty large too: For any rule \( A \to \alpha \) (without dots) there are \( (\text{length}(\alpha) + 1) \) LR(0) items (where length simply means the number of symbols in \( \alpha \)).

* Altogether, the jump from LR(0) items to CLR(1) items can introduce a huge number of new states, many of which might be redundant.

* Going from the NFA to DFA automaton can merge LR(1) items with the same LR(0) "core" but different lookahead characters. But it doesn't necessarily get rid of all the redundancy in states.

* In practice, CLR(1) parsing isn't practical because the parsing table gets too big.

**State Merging and LALR(1) Parsing**

* In LALR(1) parsing ("LA" = "Look Ahead", "LR" = "LR parsing", certain sets of CLR(1) states are merged together into one new state.

* The criterion is that the LR(0) "core items" must match.

  * Erase the lookahead part of the CLR(1) items.
  * The results are the core LR(0) items.
  * If two states have exactly the same set of core items, we merge their corresponding CLR(1) states together (so the new state will have the same core items but with possibly more lookahead symbols attached.

* The advantage is that the parsing tables become small enough to be practical, and the set of grammars that are LALR(1) is big enough for computer language parsing.

* We'll look at an example of an LALR(1) grammar next time

**Building the Parsing Tables (for SLR grammar)**

* To describe the SLR Characteristic Finite State Machine (CFSM)\(^3\) and its parse tables, we rely on two functions, \( \text{next}(I, X) \) and \( \text{closure}(I) \). \( I \) is a automaton state (a set of LR(0) items) and \( X \) is a terminal or nonterminal symbol.

  * \( \text{closure}(I) \) is \( I \) plus all the LR(0) items that (in the NFA) are \( \epsilon \)-reachable from the states of \( I \).

    * The closure is used to build the deterministic automaton states from the nondeterministic automaton

      \[ \text{closure}(I) = I \cup \{ \; C \to \gamma \mid A \rightarrow \alpha \cdot C \beta \in I \} \]

  * \( \text{next}(I, X) \) tells us what state to go to if we see symbol \( X \) in state \( I \).

    * \( \text{next}(I, X) = \text{closure}(\{ A \rightarrow \alpha X \cdot \beta \mid \text{all } A \rightarrow \alpha \cdot X \beta \in I \}) \)

    * If \( X \) is a terminal symbol, then \( \text{next}(I, X) \) specifies the state to shift to on \( X \).

    * If \( X \) is a nonterminal, then \( \text{next}(I, X) \) specifies the state to go if we've come back momentarily to state \( I \) after a reduction for \( X \).

\(^3\) CFSM is another name for an LR DFA.
Constructing the SLR DFA

* The algorithm for building the deterministic automaton for an SLR parser works by building the start state for the DFA and then repeatedly finding all the successor states from states we haven't already processed.

\[ I_0 \leftarrow \text{closure } \{ S' \rightarrow \bullet S \} \text{, marked unprocessed} \]

States ← \{I_0\} // I_0 is start state for automaton

while exists unprocessed state \( I \) in States

for each terminal and nonterminal \( X \) in the language

. Calculate \( J = \text{next}(I, X) \)

. If \( J \) not already in States

. Add \( J \) to States, marked unprocessed

Mark \( I \) as processed

Table-Based LR Parsing

* Two tables are used to control LR parsing

* The Action Table: maps a state and (the next) terminal symbol to the action to take

* Shift and go to state \( n \)

* Reduce using rule \( r \) (and then use the Go To table (below))

* Accept the input (at the end of the parse)

* Error (bad combination of state and input symbol)

* The Go To Table: maps a state and nonterminal to a new state.

* The table is used after a reduction: After we pop off the rhs symbols but before we push on the lhs nonterminal, we look at the exposed state number — it's the state we were in before we started to parse the nonterminal we just successfully parsed. We index into the Go To table with that state and the nonterminal we reduced to get the new state we should go to. Push the nonterminal onto the stack and go to that new state.

Construct (SLR) Action Table (mostly shift and reduce actions)

Say States = \{I_0, I_1, \ldots, I_N\}

for each \( i \) in 0 .. \( N \) // build row for \( I_i \)

for each terminal \( a \in \Sigma \) // build columns for this row

. if next(\( I_i, a \)) = \{I_k\}

. Set Action Table(\( i, a \)) = "shift \( k \"

. else (next(\( I_i, a \)) = \{\})

. skip // empty Action Table(\( i, a \)) will indicate an error

for each item \( A \rightarrow \alpha \bullet \) in \( I_i \)

. for each terminal \( a \in \text{Follow}(A) \) // The underlined part for an SLR(1) parser

. let \( r \) = rule number for \( A \rightarrow \alpha \)

. Set Action Table(\( i, a \)) = "reduce \( r \"

Find \( I_m \) = the state containing \( S' \rightarrow S \bullet \$ \) and set Action Table(\( m, \$ \)) = "accept"