COS320: Compiling Techniques

Zak Kincaid

March 25, 2024
Parsing III: LR parsing
Bottom-up parsing

- Stack holds a word in \((N \cup \Sigma)^*\) such that it is possible to derive the part of the input string that has been consumed from its reverse.
- At any time, may read a letter from input string and push it on top of the stack.
- At any time, may non-deterministically choose a rule \(A ::= \gamma_1...\gamma_n\) and apply it in reverse: pop \(\gamma_n...\gamma_1\) off the top of the stack, and push \(A\).
- Accept when stack just contains start non-terminal.

\[
\begin{align*}
<S> &::= <B>+<S> | <B> \\
<B> &::= (<S>) | x
\end{align*}
\]
\[ <S> ::= <B> + <S> \mid <B> \]
\[ <B> ::= ( <S> ) \mid x \]

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( \epsilon )</td>
<td>( (x+x)+x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( $ )</td>
<td>( (x+x)+x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( () )</td>
<td>( x+x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( x() )</td>
<td>( +x+x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( +&lt;B&gt;() )</td>
<td>( x) +x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( x+&lt;B&gt;() )</td>
<td>( ) +x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( &lt;B&gt;+$ )</td>
<td>( ) +x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( &lt;S&gt;+$ )</td>
<td>( ) +x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( &lt;S&gt;$ )</td>
<td>( +x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( &lt;B&gt;$ )</td>
<td>( +x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( +&lt;B&gt;$ )</td>
<td>( x )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( x+&lt;B&gt;$ )</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( &lt;B&gt;+$ )</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( &lt;S&gt;+$ )</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( &lt;S&gt;$ )</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>( q_f )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
</tr>
</tbody>
</table>
LL vs LR

- LL parsers are top-down, LR parsers are bottom-up
- Easier to write LR grammars
  - Every LL(k) grammar is also LR(k), but not vice versa.
  - No need to eliminate left (or right) recursion
  - No need to left-factor
- Harder to write LR parsers
  - But parser generators will do it for us!
Bottom-up PDA has two kinds of actions:

- **Shift**: move lookahead token to the top of the stack
- **Reduce**: remove $\gamma_n, ..., \gamma_1$ from the top of the stack, replace with $A$ (where $A ::= \gamma_1 ... \gamma_n$ is a rule of the grammar)
- Just as for LL parsing, the trick is to resolve non-determinism.
  - When should the parser shift?
  - When should the parser reduce?

\[
\begin{align*}
\text{alt} &= \langle B \rangle + \langle S \rangle | \langle B \rangle \\
\langle B \rangle &= ( \langle S \rangle ) | x
\end{align*}
\]
Roadmap to LR parsing

1. “Greedy” determinization: warm-up (not examinable material)
2. LR(0): LR parsing with 0 tokens of lookahead – not used in practice.
3. SLR (Simple LR): LR(0) + lookahead to resolve some nondeterminism
4. LR(1): Add one token of lookahead to LR construction
5. LALR(1): simple, practical optimization of LR(1) (but less powerful!)
Determinizing the bottom-up PDA

- **Intuition**: reduce greedily
  - If any reduce action applies, then apply it
    - Actually, a bit more nuanced: only apply reduction action if it is “relevant” (can eventually lead to the input word being accepted)
  - If no reduce action applies, then shift
- Can use the states of the PDA to implement greedy strategy
  - State tracks top few symbols of the stack – enough to know if a reduction rule applies.
\[ <S'> ::= <S>$ \]

\[ <S> ::= a<S>a | b \]
\[
\begin{align*}
\langle S' \rangle & ::= \langle S \rangle \$ \\
\langle S \rangle & ::= a \langle S \rangle a \mid b
\end{align*}
\]
\[ <S^\prime> ::= <S> \$ \]

\[ <S> ::= a<S>a | b \]
\[ <S'> ::= <S>$ \]

\[ <S> ::= a<S>a \mid b \]

S is on top of the stack, but what's underneath?
\[ \langle S' \rangle \coloneqq \langle S \rangle \$ \]
\[ \langle S \rangle \coloneqq a \langle S \rangle a \mid b \]
\[
\begin{align*}
\langle S' \rangle &::= \langle S \rangle \$ \\
\langle S \rangle &::= a \langle S \rangle a \mid b
\end{align*}
\]
\[ <S'> ::= <S>$ \]

\[ <S> ::= a<S>a | b \]
LR parsing

- Greedy strategy matches right-hand-sides of all rules against the top of the stack
  - Consider \( <S> ::= <A><B> , <A> ::= a , <B> ::= a \)
  - a on top of stack \( \Rightarrow \) conflict between reductions \( <A> ::= a \) and \( <B> ::= a \)
- LR parsing is partially greedy: only apply reduction action if it is “relevant” (can eventually lead to the input word being accepted)
  - E.g., apply \( <A> ::= a \) reduction to the first \( a \) that we push on the stack, but not the second.
- \( LR(k) = LR \) with \( k \)-symbol lookahead
LR(0) parsing

\[
\begin{align*}
  <S> & ::= (<L>) | x \\
  <L> & ::= <S> | <L>; <S>
\end{align*}
\]

- An **LR(0) item** of a grammar \( G = (N, \Sigma, R, S) \) is of the form \( A ::= \gamma_1 \cdots \gamma_i \bullet \gamma_{i+1} \cdots \gamma_n \), where \( A ::= \gamma_1 \cdots \gamma_n \) is a rule of \( G \)
  - \( \gamma_1 \cdots \gamma_i \) derives part of the word that has already been read
  - \( \gamma_{i+1} \cdots \gamma_n \) derives part of the word that remains to be read
  - LR(0) items \( \sim \) states of an NFA that determines when a reduction applies to the top of the stack
- LR(0) items for the above grammar:
  - \( <S> ::= \bullet (<L>), <S> ::= (\bullet <L>), <S> ::= (<L> \bullet), <S> ::= (<L>) \bullet, \)
  - \( <S> ::= \bullet x, <S> ::= x \bullet, \)
  - \( <L> ::= \bullet <S>, <L> ::= <S> \bullet, \)
  - \( <L> ::= \bullet <L>; <S>, <L> ::= <L> \bullet; <S>, <L> ::= <L>; \bullet <S>, <L> ::= <L>; <S> \bullet, \)
For any set of items $I$, define $\text{closure}(I)$ to be the least set of items such that

- $\text{closure}(I)$ contains $I$
- If $\text{closure}(I)$ contains an item of the form $A ::= \alpha \bullet B \beta$ where $B$ is a non-terminal, then $\text{closure}(I)$ contains $B ::= \bullet \gamma$ for all $B ::= \gamma \in R$

$\text{closure}(I)$ saturates $I$ with all items that may be relevant to reducing via $I$

- E.g., $\text{closure}([<S> ::= (\bullet<L>)] ) = [<S> ::= (\bullet<L>), <L> ::= \bullet<S>, <L> ::= \bullet<L>; <S>, <S> ::= \bullet(<L>)<S> ::= \bullet x]$

- Part of the not-quite greedy strategy: don’t try to reduce using all rules all the time, track only a relevant subset
For any set of items $I$, define $\text{closure}(I)$ to be the least set of items such that

- $\text{closure}(I)$ contains $I$
- If $\text{closure}(I)$ contains an item of the form $A ::= \alpha \cdot B \beta$ where $B$ is a non-terminal, then $\text{closure}(I)$ contains $B ::= \gamma \cdot \gamma$ for all $B ::= \gamma \in R$

$\text{closure}(I)$ saturates $I$ with all items that may be relevant to reducing via $I$

- E.g., $\text{closure}\left(\{<S> ::= (\bullet <L>), \bullet <L>, <L> ::= \bullet <S>, \bullet <L>; <S>, <S> ::= \bullet (<L>) <S> ::= \bullet x\} \right) = \{<S> ::= (<L> \bullet <S>), <L> ::= <L> \bullet ; <S>, \} \}

- Part of the not-quite greedy strategy: don’t try to reduce using all rules all the time, track only a relevant subset

For any item set $I$, and (terminal or non-terminal) symbol $\gamma \in N \cup \Sigma$ define $\text{goto}(I, \gamma) = \text{closure}(\{A ::= \alpha \gamma \cdot \beta | A ::= \alpha \cdot \gamma \beta \in I\})$

- I.e., $\text{goto}(I, \gamma)$ is the result of “moving $\cdot$ across $\gamma$”
- E.g., $\text{goto}(\text{closure}(\{<S> ::= (\bullet <L>), \bullet <L> \}), <L>) = \{<S> ::= (<L> \bullet), <L> ::= <L> \bullet ; <S>, \}$
Mechanical construction of LR(0) parsers

1. Add a new production \( S' ::= S\$ \) to the grammar.
   - \( S' \) is new start symbol
   - \$ marks end of word

2. Stack alphabet = closed item sets, starting with \( \text{closure}(\{ S' ::= \bullet S\$ \}) \)

3. Construct transitions as follows: for each closed item set \( I \),
   - For each item of the form \( A ::= \gamma_1 \ldots \gamma_n \bullet \) in \( I \), add reduce transition
     \[
     \epsilon, IJ_1 \ldots J_{n-1} K \rightarrow K'K, \text{ where } K' = \text{goto}(K, A)
     \]
   - For each item of the form \( A ::= \gamma \bullet a\beta \) in \( I \) with \( a \in \Sigma \), add a shift transition
     \[
     a, I \rightarrow I' I \text{ where } I' = \text{goto}(I, a)
     \]

Resulting automaton is deterministic \iff grammar is LR(0)
Conflicts

• Recall: Automaton is deterministic $\iff$ grammar is LR(0)

• Two different types of transitions:
  • Reduce transitions, from items of the form $A ::= \gamma \bullet$
  • Shift transitions, from items of the form $A ::= \gamma \bullet a\beta$, where $a$ is a terminal
  • (No transitions generated by items of the form $A ::= \gamma \bullet A\beta$ where $A$ is a non-terminal)
Conflicts

- Recall: Automaton is deterministic $\iff$ grammar is LR(0)
- Two different types of transitions:
  - *Reduce* transitions, from items of the form $A ::= \gamma\bullet$
  - *Shift* transitions, from items of the form $A ::= \gamma \bullet a\beta$, where $a$ is a terminal
  - (No transitions generated by items of the form $A ::= \gamma \bullet A\beta$ where $A$ is a non-terminal)
- *Reduce/reduce conflict*: state has two or more items of the form $A ::= \gamma\bullet$ (choice of reduction is non-deterministic!)
Conflicts

• Recall: Automaton is deterministic $\iff$ grammar is LR(0)

• Two different types of transitions:
  • *Reduce* transitions, from items of the form $A ::= \gamma \bullet$
  • *Shift* transitions, from items of the form $A ::= \gamma \bullet a\beta$, where $a$ is a terminal
  • (No transitions generated by items of the form $A ::= \gamma \bullet A\beta$ where $A$ is a non-terminal)

• **Reduce/reduce conflict**: state has two or more items of the form $A ::= \gamma \bullet$ (choice of reduction is non-deterministic!)

• **Shift/reduce conflict**: state has an item of the form $A ::= \gamma \bullet$ and one of the form $A ::= \gamma \bullet a\beta$ (choice of whether to shift or reduce is non-deterministic!)
Simple LR (SLR)

- Simple LR is a straightforward extension of LR(0) with a lookahead token
- Idea: proceed exactly as LR(0), but eliminate (some) conflicts using lookahead token
  - For each item of the form $A ::= \gamma_1 \ldots \gamma_n$ in $I$, add reduce transition
    
    $\epsilon, IJ_1 \ldots J_{n-1} K \rightarrow K' K$, where $K' = \text{goto}(K, A)$

    with any lookahead token in follow($A$)
Simple LR (SLR)

• Simple LR is a straight-forward extension of LR(0) with a lookahead token
• Idea: proceed exactly as LR(0), but eliminate (some) conflicts using lookahead token
  • For each item of the form $A ::= \gamma_1...\gamma_n$ in $I$, add reduce transition
    
    $\epsilon, IJ_1...J_{n-1}K \rightarrow K'K$, where $K' = \text{goto}(K, A)$

    with any lookahead token in $\text{follow}(A)$

• Example: the following grammar is SLR, but not LR(0)

  \[
  \begin{align*}
  <S> & ::= <T>b \\
  <T> & ::= a<T> | \epsilon 
  \end{align*}
  \]

  Consider: $\text{closure}(<S'> ::= \bullet <S>$) contains $<T> ::= \bullet$ and $<T> ::= \bullet a<T>$.

• SLR parser generators: Jison
LR(1) parser construction

- LR(1) parser generators: Menhir, Bison
- An LR(1) item of a grammar $G = (N, \Sigma, R, S)$ is of the form $(A ::= \gamma_1 \cdots \gamma_i \bullet \gamma_{i+1} \cdots \gamma_n, a)$, where $A ::= \gamma_1 \cdots \gamma_n$ is a rule of $G$ and $a \in \Sigma$
  - $\gamma_1 \cdots \gamma_i$ derives part of the word that has already been read
  - $\gamma_{i+1} \cdots \gamma_n$ derives part of the word that remains to be read
  - $a$ is a lookahead symbol
- For any set of items $I$, define $\text{closure}(I)$ to be the least set of items such that
  - $\text{closure}(I)$ contains $I$
  - If $\text{closure}(I)$ contains an item of the form $(A ::= \alpha \bullet B\beta, a)$ where $B$ is a non-terminal, then $\text{closure}(I)$ contains $(B ::= \bullet \gamma, b)$ for all $B ::= \gamma \in R$ and all $b \in \text{first}(\beta a)$.
- Construct PDA as in LR(0)
LALR(1)

- LR(1) transition tables can be very large
- LALR(1) ("lookahead LR(1)") make transition table smaller by merging states (that is, closed itemsets) that are identical except for lookahead
- Merging states can create reduce/reduce conflicts. Say that a grammar is LALR(1) if this merging doesn’t create conflicts.
- LALR(1) parser generators: Bison, Yacc, ocamlyacc, Jison
Summary of parsing

- For any $k$, $LL(k)$ grammars are $LR(k)$
- $SLR$ grammars are $LALR(1)$ are $LR(1)$
- In terms of language expressivity, there is an SLR (and therefore LALR(1) and LR(1) grammar for any context-free language that can be accepted by a deterministic pushdown automaton).
- Not every deterministic context free language is $LL(k)$: $\{a^n b^n : n \in \mathbb{N}\} \cup \{a^n c^n : n \in \mathbb{N}\}$ is DCFL but not $LL(k)$ for any $k$.\(^1\)

---

\(^1\)John C. Beatty, *Two iteration theorems for the LL(k) Languages*