CS412/413
Introduction to Compilers
Tim Teitelbaum

Lecture 10: LR Parsing
February 12, 2007

LR(0) Parsing Summary
• LR(0) item = a production with a dot in RHS
• LR(0) state = set of LR(0) items valid for viable prefixes
• Compute LR(0) states and build DFA:
  - Start state: \( V(\epsilon) = \{ [S' \rightarrow S] \} \)
  - Other states: \( V(\alpha X) = V(\alpha) \rightarrow X \)
• Build the LR(0) parsing table from the DFA
• Use the LR(0) parsing table to determine whether to reduce or to shift

LR(0) Limitations
• An LR(0) machine only works if each state with a reduce action has only one possible reduce action and no shift action
• With some grammars, construction gives states with shift/reduce or reduce/reduce conflicts
• Need to use look-ahead to choose

LR(0) Parsing Table

A Non-LR(0) Grammar
• Grammar for addition of numbers:
  \[ S \rightarrow S + E | E \]
  \[ E \rightarrow \text{num} \]
• Left-associative version is LR(0)
• Right-associative version is not LR(0)
  \[ S \rightarrow E + S | E \]
  \[ E \rightarrow \text{num} \]
SLR(1) Parsing

- SLR Parsing = easy extension of LR(0)
  - For each reduction $A \rightarrow \beta$, look at the next symbol $c$
  - Apply reduction only if $c$ is in FOLLOW(A), or $c=\varepsilon$ and $S \Rightarrow \gamma A$

- SLR parsing table eliminates some conflicts
  - Same as LR(0) table except reduction rows
  - Adds reductions $A \rightarrow \beta$ only in the columns of symbols in FOLLOW(A)

- Example:
  - FOLLOW(S) = {} but $S \Rightarrow \gamma E$

<table>
<thead>
<tr>
<th>num</th>
<th>+</th>
<th>ε</th>
<th>E</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s4</td>
<td>g2</td>
<td>g6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>S→E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S→E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S→E+S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>accept</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SLR(k)

- Use the LR(0) machine states as rows of table
- Let Q be a state and $u$ be a lookahead string
  - Action(Q,$u$) = shift Goto(Q,$b$) if Q contains an item of the form $[A \rightarrow \beta_1 B \gamma_1,\ldots,B \gamma_k]$, with $u$ in FIRST$_k(b_1 FOLL OW_k(A))$
  - Action(Q,$u$) = accept if Q = $[[S \rightarrow \gamma]]$ and $u=\varepsilon$
  - Action(Q,$u$) = reduce $i$ if Q contains the item $[A \rightarrow \beta_1 B \gamma_1,\ldots,B \gamma_k]$, where $A \rightarrow \beta$ is the $i$th production of G and $u$ in FOLLOW$_k(A)$, or $u=\varepsilon$ and $S \Rightarrow \gamma A$
  - Action(Q,$u$) = error otherwise
- G is SLR(k) iff the Action function given above is single-valued for all Q and u, i.e., there are no shift-reduce or reduce-reduce conflicts.

LR(1) Parsing

- Get as much power as possible out of 1 lookahead symbol parsing table
- LR(1) grammar = recognizable by a shift/reduce parser with 1-symbol lookahead
- LR(1) parsing uses similar concepts as LR(0)
  - Parse states = sets of items
  - LR(1) item = LR(0) item + look-ahead symbol following the production

LR(1) States

- LR(1) state = set of LR(1) items
- LR(1) item = $[A \rightarrow \alpha B \beta_1 b_1,\ldots,b_n]$ where $b$ in $\Sigma \cup \{\varepsilon\}$
- Meaning: $\alpha$ already matched at top of the stack; next expect to see $\beta$.
- Shorthand notation
  - $[A \rightarrow \alpha B \beta_1]$ means:
    - $[A \rightarrow \alpha B \beta_1 b_1]$ if $b_1$ is present
    - $[A \rightarrow \alpha B \beta_1]$ if $b_1$ is missing

LR(1) Closure

- LR(1) closure operation on set of items S
  - For each item in S: $[A \rightarrow \alpha B \beta_1 b_1]$
  - and for each production $B \rightarrow \gamma$, add the following item to S: $[B \rightarrow \gamma \text{ FIRST}(\beta_1 b_1)]$, or $[B \rightarrow \gamma \varepsilon]$, if $\text{ FIRST}(\beta_1 b_1) = \{\}$
  - Repeat until nothing changes
- Similar to LR(0) closure, but also keeps track of the look-ahead symbol
LR(1) Start State

• Initial state: start with \([S' \rightarrow S \epsilon]\), then apply the closure operation.

• Example: sum grammar

\[
\begin{align*}
S' & \rightarrow S \\
S & \rightarrow E+S | E \\
E & \rightarrow \text{num} +, E | \text{num} \\
S' & \rightarrow S \\
S & \rightarrow E+S | E \\
E & \rightarrow \text{num} +, E | \text{num}
\end{align*}
\]

LR(1) Goto Operation

• LR(1) goto operation = describes transitions between LR(1) states.

• Algorithm: for a state \(S\) and a symbol \(Y\)

\[
S'= \{ \left[ A \rightarrow \alpha Y \beta \right] \left| \left[ A \rightarrow \alpha \beta \right] \in S \}
\]

Goto(S, ' +')

LR(1) DFA Construction

• If \(S' = \text{Goto}(S,X)\) then add an edge labeled \(X\) from \(S\) to \(S'\)

LR(1) Reductions

• Reductions correspond to LR(1) items of the form \([A \rightarrow \beta ; x]\)

LR(1) Parsing Table Construction

• Same as construction of LR(0) parsing table, except for reductions.

• If \([A \rightarrow \beta ; b] \in \text{state } Q\), then:

\[
\text{Action}(Q, b) = \text{Reduce}(A \rightarrow \beta)
\]
LR(1) but not SLR(1)

- Let G have productions
  \[ S \rightarrow aAb \mid Ac \]
  \[ A \rightarrow a \mid \varepsilon \]
- \( V(a) = \{ \]
  \[ [ S \rightarrow aAb ] \]
  \[ [ A \rightarrow a ] \]
  \[ [ A \rightarrow \varepsilon ] \]
- FOLLOW(A) = \{b, c\}

LALR(1) Grammars

- Problem with LR(1): too many states
- LALR(1) Parsing (Look-Ahead LR)
  - Construct LR(1) DFA and then merge any two LR(1) states whose items are identical except look-ahead
  - Results in smaller parser tables
  - Theoretically less powerful than LR(1)
- LALR(1) Grammar = a grammar whose LALR(1) parsing table has no conflicts

Classification of Grammars

LR(1) \(\subset\) LR(k) \(\subset\) LL(k) \(\subset\) LL(k+1) \(\subset\) LR(0) \(\subset\) SLR(1) \(\subset\) LALR(1) \(\subset\) LR(1)

Automate the Parsing Process

- Can automate:
  - The construction of LR parsing tables
  - The construction of shift-reduce parsers based on these parsing tables
- Automatic parser generators: yacc, bison, CUP
- LALR(1) parser generators
  - Not much difference compared to LR(1) in practice
  - Smaller parsing tables than LR(1)
  - Augment LALR(1) grammar specification with declarations of precedence, associativity
- output: LALR(1) parser program

Associativity

\[ S \rightarrow S + E \mid E \]
\[ E \rightarrow \text{num} \]
\[ E \rightarrow E + E \]

What happens if we run this grammar through LALR construction?

Shift/Reduce Conflict

\[ E \rightarrow E + E \]
\[ E \rightarrow \text{num} \]

\[ [ E \rightarrow E + E \mid + ] \]
\[ [ E \rightarrow E + E \mid +, \varepsilon ] \]

shift/reduce conflict

shift: \(1 + (2 + 3)\)
reduce: \((1 + 2) + 3\)

\(1 + 2 + 3\)
Grammar in CUP

nonterminal E; terminal PLUS, LPAREN...
precedence left PLUS;

"when shifting a '+' conflicts with reducing a production, choose reduce"

E ::= E PLUS E
| LPAREN E RPAREN
| NUMBER ;

Precedence

• CUP can also handle operator precedence

E → E + E | T
T → T × T | num | ( E )

E → E + E | E × E
| num | ( E )

Conflicts without Precedence

E → E + E | E × E
| num | ( E )

Precedence in CUP

precedence left PLUS;
precedence left TIMES; // TIMES > PLUS
E ::= E PLUS E | E TIMES E | ...

RULE: in conflict, choose reduce if last terminal of production has higher precedence than symbol to be shifted; choose shift if vice-versa. In tie, use associativity (left or right) given by precedence rule

reduce E → E × E
Shift ×

Summary

• Look-ahead information makes SLR(1), LALR(1), LR(1) grammars expressive
• Automatic parser generators support LALR(1) grammars
• Precedence, associativity declarations simplify grammar writing