Bottom-Up Parsing (cont.)

CS2210
Lecture 6

LR Grammars
- A grammar for which an LR parsing table can be constructed
- LR(0) and LR(1) typically of interest
  - What about LL(0)?
- Parsing tables from LR grammars
  - SLR (simple LR) tables
    - Many grammars for which it is not possible
  - Canonical LR tables
    - Works on "most" grammars
  - LALR (= lookahead LR) tables
    - Often used in practice because significantly smaller than canonical LR

LR(0) items
- A -> XYZ
- Items:
  - A -> .XYZ
  - A -> X, YZ
  - A -> XY.Z
  - A -> XYZ.
- Represents how much of a production we have seen in parsing
Parse Table Construction

- Build a DFA to recognize viable prefixes
  - set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser
  - Items are ~ states of an NFA for viable prefixes
  - Group items together (subset construction) to get DFA
  - Define a canonical collection of LR(0) items

- Helpers:
  - Augmented grammar
  - S start symbol of G, S'->S new start symbol S'
  - Closure function
  - Goto function

Closure

- I set of items of G
- Closure(I)
  - Initially I
  - Repeat
    - If A->α,X,β in closure(I) and B->γ is a production add B->γ to closure
    - Until no new items get added

Goto

- Goto(I,X), I set of items, X grammar symbol
- Goto(I,X) := closure({A->α,X,β| A->α,β ∈ I})
- For valid items I for viable prefix γ, then goto(I,X) = valid items for viable prefix γX
Sets of Items Construction

procedure items(G');
begin
C := closure({S->.S});
repeat
for each set of items I in C and each
  grammar symbol X such that goto(I,X) is not
  empty and not in C do
    add goto(I,X) to C
  until no more changes to C
end

Sets of Items Construction
Example

E' -> E 10 =
E -> E + T | T 11 =
T -> T * F | F 12 =
F -> (E) | id 13 =

SLR Table Construction

Input G' augmented grammar
  1. Construct C = {I_0, ..., I_n}, collection of LR(0) items
  2. State i is constructed from I_i with parsing actions:
     a) if A->a is in I_i and goto(I_i,a) = I_j then action[i,a] := shift j
     b) if A->a is in I_i then action[i,a] := reduce A->a for
     all a in Follow(A)
     c) if S'->S is in I_i set action[I,$] = accept
  3. Goto transitions for state I are constructed for all nonterminal A as follows: if goto[I,A] = I_j then
    goto[i,A] := j
  4. Undefined entries are set to error
  5. The initial state of the parser is the one constructed
     from the items containing S'->.S
SLR Table Example

Same grammar...

Parsing Example

Grammar that are not SLR

- There are non-ambiguous grammars for which SLR tables have multiple entries
  - Grammar 4.20 in Aho
- Alternatives
  - Canonical LR
    - Most powerful
  - LALR
    - Smaller parsing tables
Building LR Parsing Tables

- Problem in SLR
  - If \( A \rightarrow \alpha \) is in Items set and \( a \in \text{Follow}(A) \) then we reduce by \( A \rightarrow \alpha \) but \( \beta \) on the stack is such that \( \beta A \) cannot be followed by \( a \).

- Solution
  - Carry more information in state on stack

---

SLR Problem Example

**Grammar:**

\[
\begin{align*}
S & \rightarrow L = R \\
S & \rightarrow R \\
L & \rightarrow *R \\
L & \rightarrow id \\
R & \rightarrow L \\
\end{align*}
\]

**Shift-reduce conflict:**

Viable prefix \( L = R \) should not reduce to \( R = R \)

**Items:**

- \( \text{I}_0 = \{S \rightarrow ., S \rightarrow .L=R, S \rightarrow .R, L \rightarrow .*R, L \rightarrow .id, R \rightarrow .L\} \)
- \( \text{I}_1 = \{S \rightarrow .\} \)
- \( \text{I}_2 = \{S \rightarrow .L.=R, R \rightarrow .L.\} \)
- \( \text{I}_3 = \{S \rightarrow .R.\} \)
- \( \text{I}_4 = \{L \rightarrow .*R, R \rightarrow .L, L \rightarrow ..*R, L \rightarrow .id\} \)
- \( \text{I}_5 = \{L \rightarrow .id.\} \)
- \( \text{I}_6 = \{S \rightarrow .L.=R, R \rightarrow .L, L \rightarrow .*R, L \rightarrow .id\} \)
- \( \text{I}_7 = \{L \rightarrow .*R.\} \)
- \( \text{I}_8 = \{R \rightarrow .L.\} \)
- \( \text{I}_9 = \{S \rightarrow .L=R.\} \)

---

SLR Problem Example (2)

**Parse id =id**

**Stack**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Error (expect to see a $)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>id=id</td>
<td>( \in \text{Follow}(L) )</td>
</tr>
<tr>
<td>s05s5</td>
<td>=id</td>
<td>( \in \text{Follow}(L) )</td>
</tr>
<tr>
<td>s06l62</td>
<td>=id</td>
<td>( \in \text{Follow}(L) )</td>
</tr>
<tr>
<td>s06l63</td>
<td>=id</td>
<td>( \in \text{Follow}(L) )</td>
</tr>
<tr>
<td>s06s61</td>
<td>=id</td>
<td>( \in \text{Follow}(L) )</td>
</tr>
</tbody>
</table>

**Items:**

- \( \text{I}_0 = \{S \rightarrow ., S \rightarrow .L=R, S \rightarrow .R, L \rightarrow .*R, L \rightarrow .id, R \rightarrow .L\} \)
- \( \text{I}_1 = \{S \rightarrow .\} \)
- \( \text{I}_2 = \{S \rightarrow .L.=R, R \rightarrow .L.\} \)
- \( \text{I}_3 = \{S \rightarrow .R.\} \)
- \( \text{I}_4 = \{L \rightarrow .*R, R \rightarrow .L, L \rightarrow ..*R, L \rightarrow .id\} \)
- \( \text{I}_5 = \{L \rightarrow .id.\} \)
- \( \text{I}_6 = \{S \rightarrow .L.=R, R \rightarrow .L, L \rightarrow .*R, L \rightarrow .id\} \)
- \( \text{I}_7 = \{L \rightarrow .*R.\} \)
- \( \text{I}_8 = \{R \rightarrow .L.\} \)
- \( \text{I}_9 = \{S \rightarrow .L=R.\} \)
LR(1) Items

- New item form: [A -> α, β, a] a terminal (including $)
- a represents (1 token) lookahead
- Used only if β = ε, i.e., we use lookahead to decide whether to reduce (or shift/error)
- a ∈ Follow(A)

LR(1) Items Construction

```plaintext
function closure(I):
    begin
        repeat
            for each item [A->α, β, a] in I,
                each production B->γ
            add [B->γ, b] to I (if not present)
        until no more items can be added to I
    end;

procedure items(G');
    begin
        C := {closure({S->.S, $})};
        repeat
            for each set of items I in C
                each grammar symbol X such that
goto(I,X) is not empty
        add goto(I,X) to C (if not there)
        until no more items can be added to C
    end
```

Example

- Consider same grammar again:
  S -> L = R
  S -> R
  L -> *R
  L -> id
  R -> L

```plaintext
function goto(I, X):
    begin
        let J be the set
            of items
            such that
            X = X
        goto(I, X) to J (if not there)
        until no more items can be added to J
    end
```
LALR Parsing

- Advantages
  - Smaller parsing tables than canonical LR
  - 100s versus 1,000s for typical language
  - Most PL constructs can be parsed with LALR
  - Algorithm of choice in practice

- Construction idea
  - Merge different items with same core, e.g., L->id., $ and L->id.,)
  - May reduce where an error should be reported
    - But reduction will lead to a parse error later
    - Merging does not produce new shift-reduce conflicts
      - Possible reduce-reduce conflicts; merge only if no conflicts

LALR Table Construction

- Simple but space / time consuming
- Build LR tables
- Merge item sets with identical cores
  - If conflict results: grammar not LALR

Efficient LALR Construction

- Ideas:
  - Represent only kernels for items set I, i.e., either S^->.S or item with "." not leftmost
  - Can generate all items in I from kernel
  - Can generate parsing actions from kernel

- Details in the book
Parsing with more Lookahead

- LR(k)
  - Closure: $A \rightarrow \alpha . B \gamma$, a predict $B \rightarrow \gamma$, y where $y \in \text{First}_k(Bx)$
  - Reduce: $A \rightarrow \alpha . x$ reduce on lookahead $x$
  - Shift on lookahead $x$ for $A \rightarrow \alpha . a \beta$, $y$, where $a$ is a terminal and $x \in \text{First}_k(a \beta y)$
- SLR(k)
  - Reduce: $A \rightarrow \alpha$ if lookahead $x \in \text{Follow}_k(A)$
  - Shift on lookahead $x$ for $A \rightarrow \alpha . a \beta$, a terminal and $x \in \text{First}_k(a \beta \text{Follow}_k(A))$
- LALR(k): merge LR(k) machine states with same core

LR(k) in Practice

- LR(k), SLR(k) are not used in practice
  - Tables too large
  - Not necessary in practice since most grammars can be made LR(1) and even LALR(1)
- Some parser generators for LALR(2)
  - Useful if too lazy to rewrite the grammar

Language Class Hierarchy
Parsing with Ambiguous Grammars

- Use non-grammatical means to resolve conflict
- Want to have just ONE parse tree!
- Useful because grammar can be
  - "more natural"
  - Smaller
- Conflict resolution:
  - Precedence
  - Associativity
  - Shift versus reduce

Precedence Example

$$E \rightarrow E + E \mid E \cdot E \mid (E) \mid id$$

$$versus$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T \cdot F \mid F$$

$$F \rightarrow (E) \mid id$$

Parse for id+id*id

<table>
<thead>
<tr>
<th>I0</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
<th>I5</th>
<th>I6</th>
<th>I7</th>
<th>I8</th>
<th>I9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0E</td>
<td>E+</td>
<td>E+E</td>
<td>E*</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>1E</td>
<td>E+</td>
<td>E+E</td>
<td>E*</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>2E</td>
<td>E+</td>
<td>E+E</td>
<td>E*</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>3E</td>
<td>E+</td>
<td>E+E</td>
<td>E*</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>4E</td>
<td>E+</td>
<td>E+E</td>
<td>E*</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>5E</td>
<td>E+</td>
<td>E+E</td>
<td>E*</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>6E</td>
<td>E+</td>
<td>E+E</td>
<td>E*</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>7E</td>
<td>E+</td>
<td>E+E</td>
<td>E*</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>8E</td>
<td>E+</td>
<td>E+E</td>
<td>E*</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>9E</td>
<td>E+</td>
<td>E+E</td>
<td>E*</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

Parser Generators

- Yacc, bison, LLGen, LRGen etc
- Specify grammar
  - Produce a table-drive parser
  - Typically can use precedence & associativity rules and allow shift-reduce conflicts
- Project uses Cup a parser generator for Java