Bottom-Up (LR) Parsing
Compilation

Source code

Tokens

Syntax tree

Machine code

Lexing

Parsing

Code Generation & Optimization

"Front end"

"Back end"

Current focus

char data[20];
int main() {
    float x;
    x = 42.0;
    return 7;
}
Overview

- Two general parsing approaches
  - Top-down: begin with start symbol (root of parse tree), and gradually expand non-terminals
  - Bottom-up: begin with terminals (leaves of parse tree), and gradually connect using non-terminals
Shift-Reduce Parsing

- **Top-down (LL) parsers**
  - Left-to-right scan, Leftmost derivation
  - Recursive routines, one per non-terminal (*recursive descent*)
  - Implicit stack (system call stack)
  - Requires more restrictive grammars
  - Simpler to understand and possible to hand-code

- **Bottom-up (LR) parsers**
  - Left-to-right scan, (reverse) Rightmost derivation
  - "Shift"/push terminals and non-terminals onto a stack
  - "Reduce"/pop to replace *handles* with non-terminals
  - Less restrictive grammars
  - Harder to understand and nearly always auto-generated
  - Very efficient
Shift-Reduce Parsing

- shift 'a'
  - reduce (V → a)
- V
  - shift '='
- V =
  - shift 'b'
- V = b
  - reduce (V → b)
- V = V
  - reduce (E → V)

(handles are underlined)

A → V = E
E → E + V
V → a | b | c

(shift = push, reduce = popN)
LR Parsing

- Creating an LR parser (pushdown automaton)
  - Build item sets
    - An item uses a dot (•) to represent parser status: "A → a • S b"
      - Dots on the left end: "possibilities"
      - Dots in the middle: "partially-complete"
      - Dots on the right end: "complete"
    - Item sets represent closures of parser states
    - Similar to NFA state collections in subset construction
  - Build ACTION / GOTO tables
    - Encodes stack and transition decisions (replaces δ in FA)
    - ACTION(state, terminal) = { shift/push, reduce/pop, accept }
    - GOTO(state, non-terminal) = state
LR Parsing

• How much lookahead do we need?
  – Depends on how complicated the grammar is
  – LR(k) – multiple lookaheads (not necessary)
  – **LR(1)** – single lookahead (*our textbook covers this!*)
    • Very general and very powerful
  – LR(0) – no lookahead
    • Severely restricted; most "interesting" grammars aren't LR(0)
LR Parsing

- **LR(1)** grammars and parsers
  - **Left-to-right** scan of the input string
  - **Rightmost** derivation
  - 1 symbol of lookahead
  - Less restricted form of context-free grammar
    - Support for most language features
    - Efficient parsing

Context-Free Hierarchy

- Context-Free
- LR(1)
- LL(1)
- Regular
LR Parser Variants

- **LR(1)** – single lookahead (*our textbook covers this!*)
  - Very general and very powerful
  - Lots of item sets; tedious to construct by hand
  - Overkill for most practical languages

- **LALR(1)** – special case of LR(1) that merges some states
  - Less powerful, but easier to manage

- **SLR(1)** – special case of LR(1) w/o explicit lookahead
  - Uses **FOLLOW** sets to disambiguate
  - Even less powerful, but much easier to understand
  - Slightly counterintuitive: all LR(1) languages have SLR(1) grammars
  - So SLR(1) is sufficiently general for our purposes
  - Use LR(0) item sets and generate SLR(1) ACTION/GOTO tables
LR(0) Item Sets

- LR(0) item sets and automaton
  - Start with an item representing "• S"
  - Form new sets by “moving the dot”
  - Take the closure to add more states if the dot lies to the left of a non-terminal
    - (Denoted here in blue)
  - Convert to finite automaton for recognizing handles by adding transitions
    - Each set becomes a state
    - “Moving the dot” = state transition + stack push

\[
\begin{align*}
S & \rightarrow a \ S \ b \\
& \mid a \ b \\
\end{align*}
\]

\[
\begin{align*}
CC_0: & \quad • \ s \\
S \rightarrow & • a \ S \ b \\
& S \rightarrow • a \ b \\
CC_1: & \quad S \cdot \\
CC_2: & \quad S \rightarrow a \cdot S \ b \\
& S \rightarrow • a \ S \ b \\
& S \rightarrow • a \ b \\
CC_3: & \quad S \rightarrow a \ S \cdot b \\
CC_4: & \quad S \rightarrow a \ b \cdot \\
CC_5: & \quad S \rightarrow a \ S \ b \cdot 
\end{align*}
\]
Create **ACTION** and **GOTO** tables

- For each item set $i$
  - If an item matches $A \rightarrow \beta \cdot c \ y$
    - $\text{ACTION}(i, c) = \text{"shift" to corresponding item set ("move the dot")}$
  - If an item matches $A \rightarrow \beta \cdot$
    - $\text{ACTION}(i, x) = \text{"reduce A → β" for all x in FOLLOW(A) ("backtrack in FA")}$
  - If an item matches $A \rightarrow \beta \cdot B \ y$
    - $\text{GOTO}(i, B) = \text{corresponding item set ("move the dot")}$
  - $\text{ACTION}(S', \$) = \text{"accept"}$
SLR(1) parsing

\[ S \rightarrow a \, S \, b \]
\[ | \quad a \, b \]

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>shift(2)</td>
<td>shift(3)</td>
<td>$</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>shift(2)</td>
<td>shift(4)</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>shift(5)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>reduce($\rightarrow a , S , b$)</td>
<td>reduce($\rightarrow a , b$)</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>reduce($\rightarrow a , a , S , b$)</td>
<td>reduce($\rightarrow a , a , b$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LR(1) parsing

\[ S \rightarrow a \ S \ b \]
\[ | \ a \ b \]

State Transition Table:

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>shift(2)</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>shift(3)</td>
<td>shift(4)</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>shift(6)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>reduce(S → a b)</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>shift(5)</td>
<td>shift(8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>reduce(S → a S b)</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>shift(9)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>reduce(S → a b)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>reduce(S → a S b)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- State 0: \([S \rightarrow \ast a S b, S]\) [\(S \rightarrow \ast a S b, S\)] [\(S \rightarrow \ast a b, S\)] [\(\ast S, S\)]
- State 1: \([S \ast, S]\)
- State 2: \([S \rightarrow a S \ast b, S]\) [\(S \rightarrow a S b, S\)] [\(S \rightarrow a a b, S\)] [\(S \rightarrow a a b, S\)]
- State 3: \([S \rightarrow a S b \ast, S]\)
- State 4: \([S \rightarrow a b \ast, S]\)
- State 5: \([S \rightarrow a b \ast, S]\) [\(S \rightarrow a b \ast, S\)] [\(S \rightarrow a a b, S\)] [\(S \rightarrow a a b, S\)]
- State 6: \([S \rightarrow a S b \ast, b]\)
- State 7: \([S \rightarrow a S b \ast, b]\)
- State 8: \([S \rightarrow a S b \ast, b]\)
- State 9: \([S \rightarrow a S b \ast, b]\)
**LR Conflicts**

- **Shift/reduce**
  - Can be resolved by always shifting or by grammar modification
- **Reduce/reduce**
  - Requires grammar modification to fix

\[
\begin{align*}
A & \rightarrow V = E \\
E & \rightarrow E + V \\
E & \rightarrow V \\
V & \rightarrow a \mid b \mid c
\end{align*}
\]

*Shift/reduce conflict in LR(0)*

\[
\begin{align*}
A & \rightarrow x \ A \ x \\
A & \rightarrow \\
B & \rightarrow x \\
C & \rightarrow x
\end{align*}
\]

*Shift/reduce conflict (all LR)*

*Observation: none of these languages are LL(1) either!*

\[
\begin{align*}
A & \rightarrow B \mid C \\
B & \rightarrow x \\
C & \rightarrow x
\end{align*}
\]

*Reduce/reduce conflict (all LR)*