Bottom-Up (LR) Parsing
Compilation

Source code

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

Tokens

Lexing

Parsing

Syntax tree

"Front end"

Current focus

Code Generation & Optimization

"Back end"

Machine code

"Front end"
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)

shift = push, reduce = popN

```
A → V = E
| V
V → a | b | c
```

“a = b + c”
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(k) – multiple lookaheads (not necessary)
- LR(1) – single lookahead (*EAC 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 (*Dragon book covers this!*)
  - 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) – no lookahead
  - Severely restricted; most "interesting" grammars aren't LR(0)
LR Parsing

- Creating an LR parser (**pushdown automaton**)
  - Build item sets from grammar productions
    - 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 multiple parser states (build by taking closure)
      - Similar to NFA state collections in subset construction
  - Build **ACTION / GOTO** tables
    - Encodes stack and transition decisions (like δ in FA)
    - **ACTION**(state, terminal) = { *shift/push, reduce/pop, accept* }
    - **GOTO**(state, non-terminal) = state
LR(0) Item Sets

• LR(0) item sets and automaton
  - Start with an item representing “•S” or “S' → •S”
    - The latter is an augmented grammar
    - The Dragon book uses it; the online tool doesn’t
  - Take the closure to add more states if the dot lies immediately to the left of a non-terminal
    - (Non-kernel items, denoted here in blue)
  - Form new sets by “moving the dot” (and take the closure)
  - Convert to finite automaton for recognizing handles by adding transitions
    - Each set becomes a state
    - “Moving the dot” = state transition + stack push

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

\[
\begin{align*}
I_0: & \quad • \; S \\
S & \rightarrow • \; a \; S \; b \\
S & \rightarrow • \; a \; b \\
I_1: & \quad S \; • \\
I_2: & \quad S \rightarrow a \; • \; S \; b \\
S & \rightarrow • \; a \; S \; b \\
S & \rightarrow • \; a \; b \\
I_3: & \quad S \rightarrow a \; S \; • \; b \\
I_4: & \quad S \rightarrow a \; b \; • \\
I_5: & \quad S \rightarrow a \; S \; b \; • \\
\end{align*}
\]
SLR(1) Tables

- Create **ACTION** and **GOTO** tables

- For each item set i
  - If an item matches $A \rightarrow \beta \cdot c \gamma$
    - **ACTION**$(i, c)$ = "shift" to corresponding item set ("move the dot")
  - If an item matches $A \rightarrow \beta \cdot$
    - **ACTION**$(i, x)$ = "reduce $A \rightarrow \beta$" for all $x$ in FOLLOW$(A)$ ("backtrack in FA")
  - If an item matches $A \rightarrow \beta \cdot B \gamma$
    - **GOTO**$(i, B)$ = corresponding item set ("move the dot")

- **ACTION**$(\{S \cdot\}, \$$) = "accept"

- Any empty **ACTION** entry = “error” (usually left blank)
SLR(1) Parsing

• **Push** state 0 onto the stack

• Repeat until next action is accept or error:
  - Look up next action in **ACTION** table
    • Row is the current state (top of stack)
    • Column is the next input (terminal or $)
  - If action is **shift(s)**:
    • **Push** state s onto stack
    • Consume one token from input
  - If action is **reduce(A → β)**:
    • **Pop** one state for each terminal or non-terminal in β
    • Look up next state in **GOTO** table and **push** it onto the stack
      - Row is the current state (top of stack, after popping β)
      - Column is A (newly-reduced non-terminal)
Example

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

\[ | \, a \, b \]

\[ \text{FOLLOW}(S) = \{ b, \$ \} \]
Example

S → a S b
| a b

Parsing for “a a b b”:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Symbols</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ 0</td>
<td>$</td>
<td>a a b b $</td>
<td>shift(2)</td>
</tr>
<tr>
<td>$ 0 2</td>
<td>a</td>
<td>a b b $</td>
<td>shift(2)</td>
</tr>
<tr>
<td>$ 0 2 2</td>
<td>a a</td>
<td>b b $</td>
<td>shift(4)</td>
</tr>
<tr>
<td>$ 0 2 2 4</td>
<td>a a b</td>
<td>b $</td>
<td>reduce(S → a b)</td>
</tr>
<tr>
<td>$ 0 2 3</td>
<td>a S</td>
<td>b $</td>
<td>shift(5)</td>
</tr>
<tr>
<td>$ 0 2 3 5</td>
<td>a S b</td>
<td>$</td>
<td>reduce(S → a S b)</td>
</tr>
<tr>
<td>$ 0 1</td>
<td>S</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

(Dragon Book version)
Example

S → a S b
|   a b

Parsing for “a a b b”:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a a b b $</td>
<td>shift(2)</td>
<td>-</td>
</tr>
<tr>
<td>0 2</td>
<td>a b b $</td>
<td>shift(2)</td>
<td>-</td>
</tr>
<tr>
<td>0 2 2</td>
<td>b b $</td>
<td>shift(4)</td>
<td>-</td>
</tr>
<tr>
<td>0 2 2 4</td>
<td>b $</td>
<td>reduce(S → a b)</td>
<td>3</td>
</tr>
<tr>
<td>0 2 3</td>
<td>b $</td>
<td>shift(5)</td>
<td>-</td>
</tr>
<tr>
<td>0 2 3 5</td>
<td>$</td>
<td>reduce(S → a S b)</td>
<td>1</td>
</tr>
<tr>
<td>0 1</td>
<td>$</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>
LR Conflicts

• Shift/reduce
  – Can be resolved by always shifting or by grammar modification

• Reduce/reduce
  – Requires grammar modification to fix

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

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

*Shift/reduce conflict in LR(0)*

*Shift/reduce conflict (all LR)*

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

*Reduce/reduce conflict (all LR)*