Bottom-Up (LR) Parsing

https://xkcd.com/859/
Compilation

Source code

Lexing

Tokens

Parsing

Syntax tree

Current focus

"Front end"

"Back end"

Machine code

Code Generation & Optimization

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)

- shift '='
- reduce (V → V)

- shift '+'
- reduce (V → E +)

- shift 'c'
- reduce (V → c)

- shift 'b'
- reduce (V → b)

- V = E
- reduce (E → E + V)

- V = E
- reduce (V = E)

A → V = E
E → E + V
| V
V → a | b | c

(handles are underlined)

shift = push, reduce = popN
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
LR Parser Variants

- LR(k) – multiple lookaheads (not necessary)
- 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) – no lookahead
  - Severely restricted; most "interesting" grammars aren't LR(0)
LR Parsing

• Creating an LR parser (pushdown automaton)
  - Build item sets
    • An item uses a dot (\cdot) 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 \( \delta \) in FA)
    • \texttt{ACTION}(state, terminal) = \{ \texttt{shift/push, reduce/pop, accept} \}
    • \texttt{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
  - 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
    - (Non-kernel items, 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

S → a S b
| a b

I₀:    • S
S → • a S b
S → • a b

I₁:  S •

I₂:  S → a • S b
S → a • b
S → • a S b
S → • a b

I₃:  S → a S • b

I₄:  S → a b •

I₅:  S → a S b •
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', $) = "accept"
SLR(1) parsing

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

\[ S \rightarrow a \ b \]

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

Diagram:

- State 0: \( S \rightarrow a \cdot S \cdot b \)
- State 1: \( S \rightarrow S \cdot \)
- State 2: \( S \rightarrow a \cdot S \cdot b \)
- State 3: \( S \rightarrow a \cdot S \cdot b \)
- State 4: \( S \rightarrow a \cdot b \cdot \)
- State 5: \( S \rightarrow a S \cdot b \cdot \)
## SLR(1) parsing

The grammar for the language is:

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

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>
LR(1) parsing

\[ S \to a \quad S \quad b \quad | \quad a \quad b \]
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)

\[
\begin{align*}
A & \rightarrow x \ A \ x \\
A & \rightarrow \\
\text{Shift/reduce conflict (all LR)}
\end{align*}
\]

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