**Bottom-up parsing**

Construct parse tree for input from leaves up
- **reducing** a string of tokens to single start symbol
  (inverse of deriving a string of tokens from start symbol)

“Shift-reduce” strategy:
- read (“shift”) tokens until seen r.h.s. of “correct” production
- reduce handle to l.h.s. nonterminal, then continue
- done when all input read and reduced to start nonterminal

**LR parsing**

LR(\(k\)) parsing
- Left-to-right scan of input, **Rightmost derivation**
- \(k\) tokens of lookahead

Strictly more general than LL(\(k\))
- gets to look at whole rhs of production before deciding what to do, not just first \(k\) tokens of rhs
- can handle left recursion and common prefixes fine

Still as efficient as any top-down or bottom-up parsing method

Complex to implement
- need automatic tools to construct parser from grammar

**LR parsing tables**

Construct parsing tables implementing a FSA with a stack
- rows: states of parser
- columns: token(s) of lookahead
- entries: action of parser
  - shift, goto state \(X\)
  - reduce production \(X ::= \text{RHS}\)
  - accept
  - error

Algorithm to construct FSA similar to algorithm to build DFA from NFA
- each state represents set of possible places in parsing

LR(\(k\)) algorithm builds huge tables
LALR(\(k\)) algorithm has fewer states \(\Rightarrow\) smaller tables
- less general than LR(\(k\)), but still good in practice
- size of tables acceptable in practice

\(k = 1\) in practice
- most parser generators, including yacc and jflex, are LALR(1)

**LR(0) parser generation**

Example grammar:
\[
P ::= S \$
\]
// always add this production
\[
S ::= \text{beep} \mid \{ L \}
\]
\[
L ::= S \mid L ; S
\]

Key idea:
simulate where input might be in grammar as it reads tokens

“Where input might be in grammar” captured by set of **items**, which forms a state in the parser’s FSA
- LR(0) item: \(lhs ::= \text{ rhs}\) production, with dot in rhs somewhere marking what’s been read (shifted) so far
- LR(\(k\)) item: also add \(k\) tokens of lookahead to each item

Initial item:
\[
P ::= . S \$
\]
Closure

Initial state is closure of initial item
- closure: if dot before non-terminal, add all productions for non-terminal with dot at the start
- "epsilon transitions"

Initial state (1):

\[
\begin{align*}
P & ::= . S \$ \\
S & ::= . \text{beep} \\
S & ::= . \{ L \}
\end{align*}
\]

State transitions

Given set of items, compute new state(s) for each symbol (terminal and non-terminal) after dot
- state transitions correspond to shift actions

New item derived from old item by shifting dot over symbol
- do closure to compute new state

Initial state (1):

\[
\begin{align*}
P & ::= . S \$ \\
S & ::= . \text{beep} \\
S & ::= . \{ L \}
\end{align*}
\]

State (2) reached on transition that shifts \textit{S}:

\[
\begin{align*}
P & ::= . S \$ \\
S & ::= . \text{beep} \\
S & ::= . \{ L \}
\end{align*}
\]

State (3) reached on transition that shifts \textit{beep}:

\[
\begin{align*}
P & ::= . S \$ \\
S & ::= . \text{beep} \\
S & ::= . \{ L \}
\end{align*}
\]

State (4) reached on transition that shifts \textit{\{}:

\[
\begin{align*}
P & ::= . S \$ \\
S & ::= . \text{beep} \\
S & ::= . \{ L \}
\end{align*}
\]

Accepting transitions

If state has \textit{P ::= ... . \$} item, then add transition labeled \textit{\$} to the accept action

Example:

\[
\begin{align*}
P & ::= . S \$ \\
S & ::= \text{beep}
\end{align*}
\]

has transition labeled \textit{\$} to accept action

Reducing states

If state has \textit{lhs ::= rhs .} item, then it has a reduce \textit{lhs ::= rhs} action

Example:

\[
\begin{align*}
P & ::= . S \$ \\
S & ::= \text{beep}
\end{align*}
\]

has reduce \textit{S ::= beep} action

No label; this state always reduces this production
- what if other items in this state shift, or accept?
- what if other items in this state reduce differently?
Rest of the states (part 1)

State (4): if shift beep, goto State (3)
State (4): if shift ¤, goto State (4)
State (4): if shift s, goto State (5)
State (4): if shift L, goto State (6)

State (5):
\[ L ::= S . \]

State (6):
\[ S ::= \{ L \} . \]
\[ L ::= L ; S \]

State (6): if shift \}, goto State (7)
State (6): if shift \; , goto State (8)

State (7):
\[ S ::= \{ L \} . \]

State (8):
\[ L ::= L ; S . \]
\[ S ::= \} . beep \]
\[ S ::= \{ L \} . \]

State (8): if shift beep, goto State (3)
State (8): if shift ¤, goto State (4)
State (8): if shift s, goto State (9)

State (9):
\[ L ::= L ; S . \]

(whew)

Building table from the states & transitions

Create a row for each state
Create a column for each terminal, non-terminal, and $$

For every "state (i): if shift \( X \) goto state (j)" transition:

- if \( X \) is a terminal, put "shift, goto \( J \)" action in row \( i \), column \( X \)
- if \( X \) is a non-terminal, put "goto \( J \)" action in row \( i \), column \( X \)

For every "state (i): if $$ accept" transition:

- put "accept" action in row \( i \), column $$

For every "state (i): reduce \( LHS ::= RHS \)" action:

- put "reduce \( LHS ::= RHS \)" action in all columns of row \( i \)

Table for this grammar

<table>
<thead>
<tr>
<th>State</th>
<th>{ }</th>
<th>beep</th>
<th>;</th>
<th>S</th>
<th>L</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s,g4</td>
<td>s,g3</td>
<td>g2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a!</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>reduce S ::= beep</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s,g4</td>
<td>s,g3</td>
<td>g5</td>
<td>g6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>reduce L ::= S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s,g7</td>
<td>s,g8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>reduce S ::= { L }</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s,g4</td>
<td>s,g3</td>
<td>g9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>reduce L ::= L ; S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

Input: { beep ; { beeeep } } $

Problems in shift-reduce parsing

Can write grammars that cannot be handled with shift-reduce parsing

Shift/reduce conflict:
- state has both shift action(s) and reduce actions

Reduce/reduce conflict:
- state has more than one reduce action

Shift/reduce conflicts

LR(0) example:
\[
E ::= E + T | T
\]

State:
\[
E ::= E . + T \\
E ::= T .
\]

Can shift +
Can reduce \( E ::= T \)

LR(k) example:
\[
S ::= if \ E \ then \ S \ |
\quad if \ E \ then \ S \ else \ S \ |
\]

State:
\[
S ::= if \ E \ then \ S . \\
S ::= if \ E \ then \ S . \ else \ S
\]

Can shift else
Can reduce \( S ::= if \ E \ then \ S \)

Avoiding shift/reduce conflicts

Can rewrite grammar to remove conflict
- E.g. MatchedStmt vs. UnmatchedStmt

Can resolve in favor of shift action
- tries to find longest r.h.s. before reducing
- works well in practice
- yacc, jflex, et al. do this
Reduce/reduce conflicts

Example:
Stmt ::= Type id ; | LHS = Expr ; | ...
... 
LHS ::= id | LHS [ Expr ] | ...
... 
Type ::= id | Type [ ] | ...

State:
Type ::= id .
LHS ::= id .

Can reduce Type ::= id
Can reduce LHS ::= id

Avoiding reduce/reduce conflicts

Can rewrite grammar to remove conflict
- can be hard
  - e.g. C/C++ declaration vs. expression problem
  - e.g. MiniJava array declaration vs. array store problem

Can resolve in favor of one of the reduce actions
- but which?
  - yacc, jflex, et al. pick reduce action for production listed textually first in specification