Bottom-up parsing

- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - can handle left-recursive grammars, virtually all programming languages
  - Easier to express programming-language syntax
- Shift-reduce parsers
  - construct right-most derivation of program
  - automatic parser generators (e.g., yacc, CUP, ocamlyacc)

Top-down parsing

(1+2+(3+4))+5

- $S \rightarrow S+E \rightarrow E \rightarrow (S)+E$
- $(S+E)+E \rightarrow (S)+E+E$
- $(E+E)+E \rightarrow (1+E)+E$
- $(1+2+E)+E$...
- In left-most derivation, entire tree above a token (2) has to be expanded when encountered
- Must be able to predict productions!

Progress of bottom-up parsing

(1+2+(3+4))+5

S $\rightarrow$ S+E $\rightarrow$ E $\rightarrow$ (S $\rightarrow$ n ($S)$)

1 $\rightarrow$ 1+2+3+4 $\rightarrow$ (1+2+(3+4))+5

S+E $\rightarrow$ n ($S$) $\rightarrow$ 1+2+3+4 $\rightarrow$ (1+2+(3+4))+5

S $\rightarrow$ (S+E)+5 $\rightarrow$ (S+E)+5 $\rightarrow$ (1+2+3+4)+5

E $\rightarrow$ (1+2+3+4)+5 $\rightarrow$ (1+2+3+4)+5

S $\rightarrow$ (S+E)+5 $\rightarrow$ (S+E)+5 $\rightarrow$ (1+2+3+4)+5

S $\rightarrow$ (S+E)+5 $\rightarrow$ (S+E)+5 $\rightarrow$ (1+2+3+4)+5

E $\rightarrow$ (1+2+3+4)+5 $\rightarrow$ (1+2+3+4)+5

S $\rightarrow$ (S+E)+5 $\rightarrow$ (S+E)+5 $\rightarrow$ (1+2+3+4)+5
Bottom-up parsing

- \((1+2+(3+4))+5 \leftarrow (E+2+(3+4))+5 \leftarrow (S+2+(3+4))+5 \leftarrow (S+E+(3+4))+5 \ldots\)

- Advantage of bottom-up parsing: select productions using more information

Top-down vs. Bottom-up

Bottom-up: Don’t need to figure out as much of the parse tree for a given amount of input

Shift-reduce parsing

- Parsing is a sequence of \textit{shift} and \textit{reduce} operations
- Parser state is a stack of terminals and non-terminals (grows to the right)
- Unconsumed input is a string of terminals
- Current derivation step is always stack+input

Problem

- How do we know which action to take -- whether to shift or reduce, and which production?

- Sometimes can reduce but \textit{shouldn’t}.
  - e.g., \(X \rightarrow \varepsilon\) can \textit{always} be reduced
  - Sometimes can reduce in more than one way.

Action-Selection Problem

- Given stack \(a\) and look-ahead symbol \(b\), should parser:
  - \texttt{shift} \(b\) onto the stack (making it \(\sigma b\))
  - \texttt{reduce} some production \(X \rightarrow \gamma\) assuming that stack has the form \(\alpha \gamma\) (making it \(\alpha X\))
**Parser States**

- **Goal:** know which reductions are legal at any given point.
- **Idea:** summarize all possible stacks $\sigma$ as a finite parser state
  - Parser state is computed by a DFA that reads in the stack $\sigma$
  - Accept states of DFA: unique reduction!
- Summarizing discards information
  - affects what grammars parser handles
  - affects size of DFA (number of states)

**LR(0) parser**

- **Left-to-right scanning, Right-most derivation, “zero” look-ahead characters**
- Too weak to handle most language grammars (e.g., “sum” grammar)
- But will help us understand shift-reduce parsing...

**An LR(0) grammar: non-empty lists**

$$S \rightarrow (L) \mid id$$

$$L \rightarrow S \mid L, S$$

$x, (y,z), w)$

$(((x)))) (x, (y, (z, w)))$

**LR(0) states**

- A state is a set of *items* keeping track of progress on possible upcoming reductions
- An LR(0) *item* is a production from the language with a separator “.” somewhere in the RHS of the production

- Stuff before “.” is already on stack (beginnings of possible γ’s to be reduced)
- Stuff after “.”: what we might see next

**Full DFA**

- **DFA-ish**
  - $S \rightarrow (L) \mid id$
  - $L \rightarrow S \mid L, S$

- **Final state**

- **States:**
  - State 1: $S \rightarrow S, S$
  - State 2: $S \rightarrow id$
  - State 3: $S \rightarrow (L) S$
  - State 4: $S \rightarrow L, S$
  - State 5: $S \rightarrow (L), S$
  - State 6: $S \rightarrow (L)$
  - State 7: $S \rightarrow L, S$
  - State 8: $S \rightarrow (L), S$
  - State 9: $L \rightarrow L, S$

- **Transitions:**
  - $S \rightarrow (L) \mid id$
  - $L \rightarrow S \mid L, S$
  - $S \rightarrow id$
  - $S \rightarrow (L)$
  - $L \rightarrow (L)$
  - $S \rightarrow id$
  - $L \rightarrow id$
  - $L \rightarrow S$
  - $S \rightarrow S$
  - $L \rightarrow S$

- **Reduce-only state:** reduce
- **Shift-only state:** shift
- **Error state:** syntax error
- **Current state:** push stack through DFA
Parsing example: \(((x), y)\)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>((x), y)</td>
<td>Shift, gate 3</td>
</tr>
<tr>
<td>(L)</td>
<td>((x), y)</td>
<td>Shift, gate 3</td>
</tr>
<tr>
<td>(L)</td>
<td>(x), y)</td>
<td>Shift, gate 2</td>
</tr>
<tr>
<td>(L)</td>
<td>(y)</td>
<td>Reduce (S \rightarrow \text{id})</td>
</tr>
<tr>
<td>(L)</td>
<td>(y)</td>
<td>Reduce (S \rightarrow \text{id})</td>
</tr>
<tr>
<td>(L)</td>
<td>(L)</td>
<td>Shift, gate 5</td>
</tr>
<tr>
<td>(L)</td>
<td>(y)</td>
<td>Shift, gate 6</td>
</tr>
<tr>
<td>(L)</td>
<td>(y)</td>
<td>Shift, gate 7</td>
</tr>
<tr>
<td>(L)</td>
<td>(S)</td>
<td>Reduce (L \rightarrow S)</td>
</tr>
<tr>
<td>(L)</td>
<td>(S)</td>
<td>Reduce (L \rightarrow S)</td>
</tr>
<tr>
<td>(L)</td>
<td>(S)</td>
<td>Reduce (L \rightarrow S)</td>
</tr>
<tr>
<td>(L)</td>
<td>(L)</td>
<td>Shift, gate 5</td>
</tr>
<tr>
<td>(L)</td>
<td>(L)</td>
<td>Shift, gate 6</td>
</tr>
<tr>
<td>(L)</td>
<td>(L)</td>
<td>Reduce (S \rightarrow \text{id})</td>
</tr>
<tr>
<td>(S)</td>
<td>(S)</td>
<td>Shift, gate 4</td>
</tr>
<tr>
<td>(S)</td>
<td>()</td>
<td>Done</td>
</tr>
</tbody>
</table>

Start State & Closure

\[ S \rightarrow (L) \mid \text{id} \]

DFA start state

CLOSURE

Constructing a DFA to read stack:

- First step: augment grammar with production \(S' \rightarrow S \text{ } S\)
- Start state of DFA: empty stack \(S \rightarrow \cdot, S\)
- CLOSURE of a state adds items for all productions whose LHS occurs in an item in the state, just after "."
- A set of possible productions to be reduced next
- Added items have the "." located at the beginning: no symbols for these items on the stack yet

Applying terminal symbols

\[ S \rightarrow \cdot, S \]
\[ S \rightarrow (L) \]
\[ S \rightarrow \text{id} \]

In new state, include all items that have appropriate input symbol just after dot, advance dot in those items, and take closure.

Applying non-terminals

\[ S \rightarrow (L) \]
\[ L \rightarrow S \]

- Non-terminals on stack treated just like terminals (but added by reductions)

Applying reduce actions

\[ S \rightarrow \cdot, S \]
\[ S \rightarrow (L) \]
\[ S \rightarrow \text{id} \]

\[ S \rightarrow \cdot, L \]
\[ L \rightarrow S \]
\[ L \rightarrow L, S \]
\[ L \rightarrow L, L, S \]
\[ L \rightarrow L, L, L, S \]

- Pop RHS off stack, replace with LHS \(X \rightarrow \gamma\), rerun DFA (e.g. \(x\))

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action -- in those states, always reduce ignoring lookahead
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Choose based on lookahead.
An LR(0) grammar?

\[ S \rightarrow S + E \mid E \]
\[ E \rightarrow n \mid (S) \]

- Left-associative version: LR(0)
- Right-associative version
  - not LR(0)

\[ S \rightarrow E + S \mid E \]
\[ E \rightarrow n \mid (S) \]

LR(0) construction

\[ S \rightarrow E + S \mid E \]
\[ E \rightarrow n \mid (S) \]

SLR grammars

- Idea: Only add reduce action to table if lookahead symbol is in the FOLLOW set of the non-terminal being reduced
- Eliminates some conflicts
- \( FOLLOW(S) = \{ \$, \) \}
- Many language grammars are SLR

\[ S \rightarrow . \]
\[ S \rightarrow E \]
\[ S \rightarrow E + S \]
\[ S \rightarrow . E + S \]
\[ E \rightarrow . num \]
\[ E \rightarrow . (S) \]

What to do in state 2?

1. \( S \rightarrow . \)
2. \( S \rightarrow E + S \)
3. \( S \rightarrow E + S \)

\[ S \rightarrow \]