**TOP-DOWN VS. BOTTOM-UP PARSING**

**Top-down:**
- Construct tree from root to leaves.
- “Guess” which RHS to substitute for non-terminal.
- Produces left-most derivation.
- Recursive-descent, LL parsers.
- “Easy” for humans.

**Bottom-up:**
- Construct tree from leaves to root.
- “Guess” which rule to “reduce” terminals.
- Produces reverse right-most derivation.
- Shift-reduce, LR, LALR, etc.
- yacc or CUP parser generator.
- “Harder” for humans.
- Can parse a larger set of languages than top-down.

---

**BOTTOM-UP PARSE EXAMPLE**

```
S  →  if E then S else S | while E do S | print | ε
E  →  true | false | id
```

If `id_b` then while true do else print

---

**LEFT-MOST VS. RIGHT-MOST DERIVATIONS**

```
S
⇒_{lm} if E then S else S
⇒_{lm} if id_b then S else S
⇒_{lm} if id_b then while E do S else S
⇒_{lm} if id_b then while true do S else S
⇒_{lm} if id_b then while true do else S
⇒_{lm} if id_b then while true do else print

⇐_{rm} if E then S else print
⇐_{rm} if E then while E do S else print
⇐_{rm} if E then while E do print
⇐_{rm} if E then while S else print
⇐_{rm} if E then S else S
⇐_{rm} S
```
There are many bottom-up parsing algorithms, suitable for different subsets of CFG's. Basic idea: Given input string \( w \), "reduce" it to the goal (start) symbol, by looking for substrings that match production right-hand sides.

Example:

\[
\begin{align*}
S & \rightarrow aAcBe \\
A & \rightarrow Ab \mid b \\
B & \rightarrow d
\end{align*}
\]

"Right sentential form" Reduction

\[
\begin{align*}
abbcde & \\
abbcde & A \rightarrow b \\
Aabccde & A \rightarrow Ab \\
AaCcde & B \rightarrow d \\
S & S \rightarrow aAcBe
\end{align*}
\]

Steps correspond to a right-most derivation in reverse.

We must choose the production to use wisely! We don’t always making progress by reducing with a production even when its right-hand sides match the input.

Example:

\[
\begin{align*}
abbcde & \\
aAbccde & A \rightarrow b \\
aAaCcde & A \rightarrow b
\end{align*}
\]

Stuck!

A handle is a substring that

- is the right-hand side of some production; and
- whose replacement by the production's left-hand side is a (reverse) step in a rightmost derivation.

If grammar is unambiguous, handle is unique.
More formally, a handle is a production $A \rightarrow \beta$ and a position in the current right-sentential form $\alpha \beta w$ such that:

$S \Rightarrow_{rm} \alpha Aw \Rightarrow_{rm} \alpha \beta w$

For example grammar, if current right-sentential form is $\alpha A \beta de$ then the handle is $A \rightarrow Ab$ at the marked position.

Note that $w$ never contains non-terminals.

**Handle Pruning**

Idea: Keep removing handles, replacing them with corresponding left-hand side of production, until we reach $S$.

Another example:

$E \rightarrow E+E | E*E | (E) | id$

**Shift-reduce Parsing**

Happily, we can use parser generators that compute the handles for us.

Will concentrate on shift-reduce machine framework used for bottom-up parsing, so that we can understand generator behavior.

Have stack to hold grammar symbols and input buffer to hold string to be parsed.

Machine actions:

- **Shift** input symbols from buffer to stack until a handle is formed.
- **Reduce** handle by replacing grammar symbols at top of stack by l.h.s. of production.
- **Accept** on successful completion of parse.
- **Fail** on syntax error.

Why a stack?

Because handles always appear at the top of a stack, i.e., there’s no need to look deeper into the “state.” This is just a fact about rightmost derivations.
**Shift-Reduce Parsing Example**

\[ E \rightarrow E \mid E \mid (E) \mid \text{id} \]

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input Buffer</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ $</td>
<td>( a \cdot b \cdot c $</td>
<td>Shift</td>
</tr>
<tr>
<td>$E$</td>
<td>( +b \cdot c $</td>
<td>Reduce: ( E \rightarrow \text{id} )</td>
</tr>
<tr>
<td>$E+$</td>
<td>( +b \cdot c $</td>
<td>Shift</td>
</tr>
<tr>
<td>$E+E$</td>
<td>( \ast c $</td>
<td>Reduce: ( E \rightarrow \text{id} )</td>
</tr>
<tr>
<td>$E+E*$</td>
<td>( \ast c $</td>
<td>Shift (*’)</td>
</tr>
<tr>
<td>$E+E* $</td>
<td>( c $</td>
<td>Shift</td>
</tr>
<tr>
<td>$E+E* $</td>
<td>( \ast c $</td>
<td>Reduce: ( E \rightarrow \text{id} )</td>
</tr>
<tr>
<td>$E+E* $</td>
<td>( \ast c $</td>
<td>Reduce: ( E \rightarrow E \cdot E )</td>
</tr>
<tr>
<td>$E+* $</td>
<td>( \ast c $</td>
<td>Reduce: ( E \rightarrow E \cdot E )</td>
</tr>
<tr>
<td>$E+* $</td>
<td>$ $</td>
<td>Accept</td>
</tr>
</tbody>
</table>

Gives \( E \Rightarrow_{\text{rm}} E \cdot E \Rightarrow_{\text{rm}} E \cdot E \cdot E \Rightarrow_{\text{rm}} E \cdot E \) \( \cdot E \cdot E \Rightarrow_{\text{rm}} E \cdot E \cdot E \Rightarrow_{\text{rm}} E \cdot E \cdot E \cdot E \Rightarrow_{\text{rm}} E \cdot E \cdot E \cdot E \Rightarrow_{\text{rm}} a \cdot b \cdot c \).

Why not \( E \Rightarrow_{\text{rm}} E \cdot E \Rightarrow_{\text{rm}} E \cdot E \cdot E \Rightarrow_{\text{rm}} E \cdot E \cdot E \cdot E \Rightarrow_{\text{rm}} E \cdot E \cdot E \cdot E \Rightarrow_{\text{rm}} a \cdot b \cdot c \)??

**Conflicts**

Ambiguous grammars lead to **parsing conflicts**. Can fix by rewriting grammar or by making appropriate **choice of action** during parsing.

**Shift/Reduce** conflicts: should we shift or reduce?
- (See previous example)
- Dangling \textit{else} is another example.

**Reduce/Reduce** conflicts: which production should we reduce with?

Example:

\[
\begin{align*}
\text{stmt} & \rightarrow \text{id} (\text{expr}) \quad (a(i) \text{ is procedure call}) \\
\text{expr} & \rightarrow \text{id} (\text{expr}) \mid \text{id} \quad (a(i) \text{ is array subscript})
\end{align*}
\]

Stack | Action
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$...a(i)$</td>
<td>Reduce by ??</td>
</tr>
</tbody>
</table>

Should we reduce to \textit{stmt} or to \textit{expr}? Need to know the type of \textit{a}: is it an array or a function? This information must flow from declaration of \textit{a} to this use, typically via a symbol table.

**LR Parsing**

**LR parsers** are most general non-backtracking shift-reduce parsers known.
- \textit{L} stands for “Left-to-right scan of input.”
- \textit{R} stands for “Rightmost derivation (in reverse).”

Efficient implementations are possible.

Any LL grammar is also LR (and so are many others).

Suffices for almost all programming language CFG’s.

Disadvantage: Extremely tedious to build by hand, so need a generator.

**LR Parser Engine**

Idea: Implement shift-reduce parser using a\textit{ DFA} to choose actions based on contents of stack plus zero or more symbols of lookahead.

Components of machine:
- Input buffer.
- Stack of \textit{states} (and grammar symbols). States “summarize” stack contents.
- Parsing tables, which encode DFA.
- Driver routine (fixed for all grammars)

Machine is efficient because actions are determined by input and state at top of stack.

If each entry in \textit{LR} parsing table is uniquely defined, grammar is an **LR grammar**.
LR GRAMMARS

In an \( LR(k) \) grammar, parsing moves are determined by state on top of stack and next \( k \) symbols of input. \((k = 0, 1 \text{ usually enough.})\)

\( LR(k) \) grammars don’t suffice for, e.g., dangling else construct, but it (and others) can be handled by making a choice of table entry (e.g., Shift or Reduce).

\( LR \) comes in different varieties, based on table construction method, each able to parse a somewhat different set of languages:

- \( SLR \) small tables, simple languages
- \( LR(1) \) large tables, more languages
- \( LALR(1) \) same size tables as \( SLR \), but more languages (\( CUP \) uses these)

\( LR \) parsers have more information available than \( LL \) parsers when choosing a production:

- \( LR(k) \) knows everything derived from r.h.s. plus \( k \) lookahead symbols.
- \( LL(k) \) just knows \( k \) lookahead symbols into what’s derived from r.h.s.