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.

Bottom-up can parse a larger set of languages than top-down.

Both work for most (but not all) features of most computer languages.

Bottom-up Parse Example

\[ S \rightarrow \text{if } E \text{ then } S \text{ else } S | \text{while } E \text{ do } S | \text{print} \]
\[ E \rightarrow \text{true} | \text{false} | \text{id} \]

if id then while true do print else print

Parse Tree:
**Bottom-up Parsing**

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 r.h.s.’s.

Example:

$$S \rightarrow aAcBe$$

$$A \rightarrow Ab \mid b$$

$$B \rightarrow d$$

“Right sentential form”  Reduction

\[
\begin{align*}
ab\bar{c}de & \rightarrow A \rightarrow b \\
ab\bar{c}de & \rightarrow A \rightarrow Ab \\
a\bar{c}de & \rightarrow B \rightarrow d \\
S & \rightarrow S \rightarrow aAcBe
\end{align*}
\]

Steps correspond to a right-most derivation in reverse.

Note: must choose r.h.s. wisely!

**“Handle Pruning”**

Idea: Keep removing handles, replacing them with corresponding l.h.s. of production, until we reach $S$.

Another example:

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

Right-sentential form  Handle  Reducing production

\[
\begin{align*}
 a+b*c & \rightarrow a \quad E \rightarrow id \\
E+b*c & \rightarrow b \quad E \rightarrow id \\
E+E*E & \rightarrow c \quad E \rightarrow id \\
E+E*E & \rightarrow E*E \quad E \rightarrow E*E \\
E+E & \rightarrow E \quad E \rightarrow E+E
\end{align*}
\]

Note that grammar is ambiguous, so there are actually two handles at next-to-last step.

Big question: How do we identify handles?

- We will not answer in this course (see Cooper and Torczon section 3.5).

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

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

**Handles**

Don’t always make progress by replacing a substring with the l.h.s. of a matching production.

Example:

\[
\begin{align*}
ab\bar{c}de & \rightarrow a\bar{c}de \rightarrow A \rightarrow b \\
ab\bar{c}de & \rightarrow a\bar{c}de \rightarrow A \rightarrow Ab \\
a\bar{c}de & \rightarrow a\bar{c}de \rightarrow A \rightarrow Ab \\
S & \rightarrow S \rightarrow S \rightarrow aAcBe
\end{align*}
\]

A **handle** is a substring that

- is the r.h.s. of some production; and
- whose replacement by the production’s l.h.s. 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 $a\beta\bar{w}$ such that:

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

For example grammar, if current right-sentential form is $a\bar{c}de$

then the handle is $A \rightarrow Ab$ at the marked position.

Note that $w$ never contains non-terminals.

**Shift-reduce Parsing**

Machine framework common to bottom-up parsers.

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 gramming 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**

\[ E \rightarrow E + E | E \cdot E | (E) | id \]

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

We can perform “semantic actions” (e.g., build parse tree nodes) when reduce actions are performed.

Again – we haven’t said how actions are chosen. (In general, based on stack and input.)

Machine execution shown corresponds to this derivation:

\[ E \Rightarrow_{rm} E + E \Rightarrow_{rm} E + E \cdot E \Rightarrow_{rm} E + E \cdot c \Rightarrow_{rm} E + b \cdot c \]
\[ \Rightarrow_{rm} a + b \cdot c \]

What about \[ E \Rightarrow_{rm} E \cdot E \Rightarrow_{rm} E \cdot c \Rightarrow_{rm} E + E \cdot c \Rightarrow_{rm} E + b \cdot c \Rightarrow_{rm} a + b \cdot c \] ?! See “*ed Shift action.

**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 else is another example.

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

Example:

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input Buffer</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ldots a(i) \ldots$</td>
<td>$\ldots$</td>
<td>Reduce by ??</td>
</tr>
</tbody>
</table>

Should we reduce to param or to expr? Need to know the type of a: is it an array or a function?

This info. must flow from declaration of a to this use, typically via a symbol table.

**LR Parsing**

**LR parsers** are most general non-backtracking shift-reduce parsers known.

- **L** stands for “Left-to-right scan of input.”
- **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.

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

**Components of machine:**

- Input buffer.
- Stack of 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 and.

**LR Grammars**

If each entry in LR parsing table is uniquely defined, grammar is an LR grammar.

In an LR($k$) grammar, parsing moves are determined by state on top of stack and next $k$ symbols of input. ($k = 0, 1$ 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** knows everything derived from r.h.s. plus $k$ lookahead symbols.
- **LL** just knows $k$ lookahead symbols into what’s derived from r.h.s.