Introduction to Bottom-Up Parsing

Outline

- The strategy: *shift-reduce* parsing
- A key concept: *handles*
- Ambiguity and precedence declarations

Predictive Parsing Summary

- *First* and *Follow* sets are used to construct predictive tables
  - For non-terminal $A$ and input $t$, use a production $A \rightarrow \alpha$ where $t \in \text{First}(\alpha)$
  - For non-terminal $A$ and input $t$, if $t \in \text{Follow}(\alpha)$ and $\varepsilon \in \text{First}(A)$, use a production $A \rightarrow \alpha$ where $\varepsilon \in \text{First}(\alpha)$

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing.
  - Don’t need left-factored grammars.
  - Left recursion fine.
  - Just as efficient.
  - Builds on ideas in top-down parsing.
- Bottom-up parsing is the preferred method in practice.
  - Automatic parser generators: *YACC*, *Bison*, ...
An Introductory Example

- Revert to the "natural" grammar for our example:
  \[
  \begin{align*}
  E &\rightarrow T + E | T \\
  T &\rightarrow \text{int} * T | \text{int} | (E)
  \end{align*}
  \]
- Consider the string: \text{int} * \text{int} + \text{int}

The Idea

Bottom-up parsing reduces a string to the start symbol by inverting productions:

<table>
<thead>
<tr>
<th>Production</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{int} * \text{int} + \text{int}</td>
<td>\text{int}</td>
</tr>
<tr>
<td>\text{int} * \text{int} + \text{T}</td>
<td>\text{T}</td>
</tr>
<tr>
<td>\text{T} + \text{int}</td>
<td>\text{T}</td>
</tr>
<tr>
<td>\text{T} + \text{E}</td>
<td>\text{E}</td>
</tr>
<tr>
<td>\text{E}</td>
<td>\text{E}</td>
</tr>
</tbody>
</table>

Observation

- Read the sequence of productions in reverse (from bottom to top)
- This is a rightmost derivation!

Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse.

LR-parser
A Bottom-up Parse

\[
\begin{align*}
E & \rightarrow T + E \\
T & \rightarrow T + T \\
E & \rightarrow T + E \\
T & \rightarrow \text{int} \\
E & \rightarrow \text{int} \\
\end{align*}
\]

A Bottom-up Parse in Detail (1)

\[
\begin{align*}
E & \rightarrow \text{int} + \text{int} \\
T & \rightarrow \text{int} * \text{int} \\
T & \rightarrow \text{int} + \text{int} \\
\end{align*}
\]

A Bottom-up Parse in Detail (2)

\[
\begin{align*}
E & \rightarrow \text{int} + \text{int} \\
T & \rightarrow \text{int} * \text{int} \\
\end{align*}
\]

A Bottom-up Parse in Detail (3)

\[
\begin{align*}
E & \rightarrow \text{int} + \text{int} \\
T & \rightarrow \text{int} * \text{int} \\
\end{align*}
\]
A Bottom-up Parse in Detail (4)

\[
\begin{align*}
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast T + \text{int} \\
T & + \text{int} \\
T & + T
\end{align*}
\]

A Bottom-up Parse in Detail (5)

\[
\begin{align*}
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast T + \text{int} \\
T & + \text{int} \\
T & + T \\
T & + E
\end{align*}
\]

A Bottom-up Parse in Detail (6)

\[
\begin{align*}
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast T + \text{int} \\
T & + T \\
T & + \text{E} \\
E
\end{align*}
\]

A Trivial Bottom-Up Parsing Algorithm

Let \( I = \) input string

repeat

pick a non-empty substring \( \beta \) of \( I \)

where \( X \rightarrow \beta \) is a production

if no such \( \beta \), backtrack

replace one \( \beta \) by \( X \) in \( I \)

until \( I = \) "S" (the start symbol) or all possibilities are exhausted
Questions

• Does this algorithm terminate?
• How fast is the algorithm?
• Does the algorithm deal with all cases?
• How do we choose the substring to reduce at each step?

Where Do Reductions Happen

"Important Fact #1" has an interesting consequence:

- Let $\alpha \beta \omega$ be a step of a bottom-up parse.
- Assume the next reduction is by $X \rightarrow \beta$.
- Then $\omega$ is a string of terminals.

Why? Because $aX_0 \rightarrow a\beta_0$ is a step in a right-most derivation.

Notation

• Idea: Split string into two substrings.
  - Right substring is as yet unexamined by parser (hence is a string of terminals).
  - Left substring has terminals and non-terminals.

• The dividing point is marked by $\mid$.
  - The $\mid$ is not part of the string.

• Initially, all input is unexamined. $x_1x_2 \ldots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

• Shift: Move $\mid$ one place to the right.
  - Shifts a terminal to the left string
    $ABC\mid xyz \Rightarrow ABCx\mid yz$

• Reduce: Apply an inverse production at the right end of the left string.
  - If $A \rightarrow xy$ is a production, then
    $Cbxy\mid ijk \Rightarrow CbA\mid ijk$
The Example with Reductions Only

\[
\begin{align*}
\text{reduce } T & \rightarrow \text{int } T + \text{int} \\
\text{reduce } T & \rightarrow \text{int } T \text{ int} \\
T + \text{int} & \rightarrow \text{reduce } T \rightarrow \text{int} \\
T + T & \rightarrow \text{reduce } E \rightarrow T \\
T + E & \rightarrow \text{reduce } E \rightarrow T + E \\
E & \\
\end{align*}
\]

The Example with Shift-Reduce Parsing

\[
\begin{align*}
\text{reduce } T & \rightarrow \text{int } T + \text{int} \\
\text{reduce } T & \rightarrow \text{int } T \text{ int} \\
\text{shift } T & \rightarrow \text{int } T \text{ int} \\
\text{shift } \text{int} & \rightarrow \text{int } T \text{ int} \\
\text{shift } \text{int} & \rightarrow \text{int } T \text{ int} \\
\text{shift } \text{int} & \rightarrow \text{int } T \text{ int} \\
\text{shift } \text{int} & \rightarrow \text{int } T \text{ int} \\
\text{reduce } T & \rightarrow \text{int } T \text{ int} \\
\text{reduce } T & \rightarrow \text{int } T \text{ int} \\
\text{reduce } T & \rightarrow \text{int } T \text{ int} \\
\text{reduce } E & \rightarrow T + E \\
E & \\
\end{align*}
\]

A Shift-Reduce Parse in Detail (1)

\[
\begin{align*}
\text{int } * \text{ int } + \text{int} & \\
\text{int } * \text{ int } + \text{int} & \\
\end{align*}
\]

A Shift-Reduce Parse in Detail (2)

\[
\begin{align*}
\text{int } * \text{ int } + \text{int} & \\
\text{int } * \text{ int } + \text{int} & \\
\end{align*}
\]
A Shift-Reduce Parse in Detail (3)

\[
\begin{align*}
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{int} \ast \text{int} + \text{int} \\
\uparrow
\end{align*}
\]

A Shift-Reduce Parse in Detail (4)

\[
\begin{align*}
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{int} \ast \text{int} + \text{int} \\
\uparrow
\end{align*}
\]

A Shift-Reduce Parse in Detail (5)

\[
\begin{align*}
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{T} \\
\text{int} \ast \text{int} + \text{int} \\
\uparrow
\end{align*}
\]

A Shift-Reduce Parse in Detail (6)

\[
\begin{align*}
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{T} \\
\text{int} \ast \text{int} + \text{int} \\
\uparrow
\end{align*}
\]
A Shift-Reduce Parse in Detail (11)

<table>
<thead>
<tr>
<th>int * int + int</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
<tr>
<td>int * int + int</td>
</tr>
<tr>
<td>int * T</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>T + int</td>
</tr>
<tr>
<td>T + T</td>
</tr>
<tr>
<td>T + E</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>

The Stack

- Left string can be implemented by a stack
- Top of the stack is the |
- Shift pushes a terminal on the stack.
- Reduce pops 0 or more symbols off the stack (production rhs) and pushes a non-terminal on the stack (production lhs).

Shift-Reduce Parser

- Stack
- Current Symbol
- Parser Engine
- Parser Action

Key Issue

- How do we decide when to shift or reduce?
  - Consider step int * int + int
  - We could reduce by T → int giving T | * int + int
  - A fatal mistake: Because there is no way to reduce to the start symbol E.

```
E → T+E | T
T → int * T | int | (E)
```
Handles

- **Intuition:** Want to reduce only if the result can still be reduced to the start symbol.

- Assume a rightmost derivation: \( S \Rightarrow^* \alpha X_\omega \rightarrow \alpha \beta _\omega \)

- Then \( \alpha \beta \) is a **handle** of \( \alpha \beta _\omega \).

Handles (Cont.)

- A **handle** is a string that can be reduced, and that also allows further reductions back to the start symbol.

- We only want to reduce at handles.

- **Note:** We have said what a handle is, not how to find handles.

Important Fact #2

**Important Fact #2 about bottom-up parsing:**

*In shift-reduce parsing, handles appear only at the top of the stack, never inside.*

Why?

- Informal induction on # of reduce moves:
  - True initially, stack is empty
  - Immediately after reducing a handle
    - right-most non-terminal on top of the stack.
    - next handle must be to right of right-most non-terminal, because this is a right-most derivation.
    - Sequence of shift moves reaches next handle.
Summary of Handles

- In shift-reduce parsing, handles always appear at the top of the stack.
- Handles are never to the left of the rightmost non-terminal.
  - Therefore, shift-reduce moves are sufficient; the \| need never move left.
- Bottom-up parsing algorithms are based on recognizing handles.

Conflicts

- **Generic shift-reduce strategy:**
  - If there is a handle on top of the stack, reduce
  - Otherwise, shift
- **But what if there is a choice?**
  - If it is legal to shift or reduce, there is a shift-reduce conflict.
  - If it is legal to reduce by two different productions, there is a reduce-reduce conflict.

Source of Conflicts

- Ambiguous grammars always cause conflicts.
- But beware, so do many non-ambiguous grammars.

Consider our favorite ambiguous grammar:

\[
E \rightarrow E + E \\
| E \ast E \\
| (E) \\
| \text{int}
\]

One Shift-Reduce Parse

\[
\begin{align*}
& \text{int} \ast \text{int} + \text{int} \quad \text{shift} \\
& \quad \ldots \\
& \quad E \ast E \ast \text{int} \quad \text{reduce } E \rightarrow E \ast E \\
& \quad E \ast \text{int} \quad \text{shift} \\
& \quad E + \text{int} \quad \text{reduce } E \rightarrow \text{int} \\
& \quad E + E \quad \text{reduce } E \rightarrow E + E \\
& \quad E \\
\end{align*}
\]
Another Shift-Reduce Parse

<table>
<thead>
<tr>
<th>int * int + int</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>E * E</td>
<td>int</td>
</tr>
<tr>
<td>E * E +</td>
<td>int</td>
</tr>
<tr>
<td>E * E + int</td>
<td>reduce E → int</td>
</tr>
<tr>
<td>E * E + E</td>
<td>reduce E → E + E</td>
</tr>
<tr>
<td>E</td>
<td>reduce E → E * E</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

Example Notes

• In the second step E * E | + int, we can either shift or reduce by E → E * E.

• Choice determines associativity and precedence of + and *.

• As noted previously, grammar can be rewritten to enforce precedence.
• Precedence declarations are an alternative.

Precedence Declarations Revisited

• Precedence declarations cause shift-reduce parsers to resolve conflicts in certain ways.

• Declaring "** has greater precedence than +" causes parser to reduce at E * E | + int.

• More precisely, precedence declaration is used to resolve conflict between reducing a * and shifting a +.