Bottom-Up Parsing Algorithms
Overview

Matthew Dwyer
324E Nichols Hall
dwyer@cis.ksu.edu
http://www.cis.ksu.edu/~dwyer
Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
- Bottom-up is the preferred method in practice
- We’ll explain the intuition, you should read the book to understand the algorithm
An Introductory Example

• Bottom-up parsers don’t need left-factored grammars
• Hence we can revert to the “natural” grammar for our example:
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \times T \mid \text{int} \mid (E) \]
• Consider the string: \text{int} \times \text{int} + \text{int}
The Idea

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

\[
\begin{align*}
\text{int} \times \text{int} + \text{int} & \quad T \rightarrow \text{int} \\
\text{int} \times T + \text{int} & \quad T \rightarrow \text{int} \times T \\
T + \text{int} & \quad T \rightarrow \text{int} \\
T + T & \quad E \rightarrow T \\
T + E & \quad E \rightarrow T + E \\
E & \\
\end{align*}
\]
For You To Do

• **Question**: find the rightmost derivation of the string `int * int + int`
Observation

• Read the productions found by bottom-up parse in reverse (i.e., from bottom to top)

• This is a rightmost derivation!

\[
\begin{align*}
\text{int} \times \text{int} + \text{int} & \quad \text{T} \rightarrow \text{int} \\
\text{int} \times \text{T} + \text{int} & \quad \text{T} \rightarrow \text{int} \times \text{T} \\
\text{T} + \text{int} & \quad \text{T} \rightarrow \text{int} \\
\text{T} + \text{T} & \quad \text{E} \rightarrow \text{T} \\
\text{T} + \text{E} & \quad \text{E} \rightarrow \text{T} + \text{E} \\
\text{E} &
\end{align*}
\]
A bottom-up parser traces a rightmost derivation in reverse

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

int * int + int
A Bottom-up Parse in Detail (2)

\[ \text{int} \times \text{int} + \text{int} \]

\[ \text{int} \times T + \text{int} \]
A Bottom-up Parse in Detail (3)

\[ \text{int} \times \text{int} + \text{int} \]
\[ \text{int} \times \text{T} + \text{int} \]
\[ \text{T} + \text{int} \]
int * int + int
int * T + int
T + int
T + T
int * int + int
int * T + int
T + int
T + T
T + E

A Bottom-up Parse in Detail (5)
A Bottom-up Parse in Detail (6)

int * int + int
int * T + int
T + int
T + T
T + E
E

```
E
  ┌── T
  │   ┌── int *
  │   │   └── int
  │   └── int + int
  └── int
```
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
For You To Do

Do you see any problems with this algorithm?

Think about performance and completeness
Where Do Reductions Happen

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 $\alpha X \omega \rightarrow \alpha \beta \omega$ is a step in a right-most derivation
Idea

• Split string into two substrings
  – Right substring: as yet unexamined by parsing (a string of terminals)
  – Left substring: has terminals and non-terminals
• The dividing point is marked by a |
  – The | 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**
- **Reduce**
Shift

• *Shift*: Move | one place to the right
  – Shifts a terminal to the left string

\[ \text{ABC|xyz} \Rightarrow \text{ABCx|yz} \]
Reduce

• Apply an *inverse production* at the right end of the left string
  – If $A \rightarrow xy$ is a production, then

$$Cbxy|ijk \Rightarrow CbA|ijk$$

  – $xy$ is called a *handle*
The Example with Reductions

\[
\begin{align*}
\text{reduce } T & \rightarrow \text{int} \\
\text{reduce } T & \rightarrow \text{int} * T \\
\text{reduce } T & \rightarrow \text{int} \\
\text{reduce } E & \rightarrow T \\
\text{reduce } E & \rightarrow T + E
\end{align*}
\]
The Example with Shift-Reduce Parsing

\[ \text{int} \ast \text{int} + \text{int} \quad \text{shift} \]
\[ \text{int} \ast \text{int} + \text{int} \quad \text{shift} \]
\[ \text{int} \ast \text{int} + \text{int} \quad \text{shift} \]
\[ \text{int} \ast \text{int} + \text{int} \quad \text{reduce } T \rightarrow \text{int} \]
\[ \text{int} \ast \text{int} + \text{int} \quad \text{reduce } T \rightarrow \text{int} \ast T \]
\[ T + \text{int} \quad \text{shift} \]
\[ T + \text{int} \quad \text{shift} \]
\[ T + \text{int} \quad \text{reduce } T \rightarrow \text{int} \]
\[ T + \text{int} \quad \text{reduce } T \rightarrow \text{int} \]
\[ T + \text{int} \quad \text{reduced } E \rightarrow T \]
\[ T + \text{int} \quad \text{reduced } E \rightarrow T + E \]
\[ E \quad \text{reduce } E \rightarrow T + E \]
A Shift-Reduce Parse in Detail

\( \text{int} \times \text{int} + \text{int} \)
A Shift-Reduce Parse in Detail (2)

\[ \text{int} \ast \text{int} + \text{int} \]
\[ \text{int} \mid \ast \text{int} + \text{int} \]
A Shift-Reduce Parse in Detail

(3)

\[
\text{int} * \text{int} + \text{int} \\
\text{int} \mid * \text{int} + \text{int} \\
\text{int} \mid \text{int} + \text{int}
\]
A Shift-Reduce Parse in Detail

(4)

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

(5)

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

T

int * int + int

↑
A Shift-Reduce Parse in Detail

(6)

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

(7)

\[ \text{int} \times \text{int} + \text{int} \]
\[ \text{int} \times \text{int} + \text{int} \]
\[ \text{int} \times \text{int} + \text{int} \]
\[ \text{int} \times \text{int} + \text{int} \]
\[ \text{T} + \text{int} \]
\[ \text{T} + \text{int} \]

\[ T \]
\[ T \]
\[ \text{int} \times \text{int} \]
\[ \text{int} + \text{int} \]
\[ \text{int} + \text{int} \]

CIS 706 Translators I
A Shift-Reduce Parse in Detail

(8)

| int * int + int |
int | * int + int |
int * | int + int |
int * int | + int |
int * T | + int |
T | + int |
T + | int |
T + int |
A Shift-Reduce Parse in Detail

(9)

\[ T + \text{int} \mid \text{int} + T \]
\[ \text{int} \ast \text{int} + \text{int} \]
\[ \text{int} \ast \mid \text{int} + \text{int} \]
\[ \text{int} \ast \text{int} \mid + \text{int} \]
\[ \text{int} \ast T \mid + \text{int} \]

CIS 706 Translators I
A Shift-Reduce Parse in Detail

\[ \text{int} * \text{int} + \text{int} \]
\[ \text{int} | * \text{int} + \text{int} \]
\[ \text{int} * \text{int} | + \text{int} \]
\[ \text{int} * \text{T} | + \text{int} \]
\[ \text{T} | + \text{int} \]
\[ \text{T} + | \text{int} \]
\[ \text{T} + \text{int} | \]
\[ \text{T} + \text{T} | \]
\[ \text{T} + \text{E} | \]
A Shift-Reduce Parse in Detail

(11)

\[
\begin{align*}
E & \rightarrow T + E \\
T & \rightarrow T + T \\
T & \rightarrow T + int \\
T & \rightarrow int * T + int \\
T & \rightarrow int * int + int \\
T & \rightarrow int * int + int \\
T & \rightarrow T + int \\
T & \rightarrow T + int \\
T & \rightarrow T + T \\
T & \rightarrow T + E \\
E & \rightarrow int \\
E & \rightarrow int \\
E & \rightarrow int \\
E & \rightarrow int \\
\end{align*}
\]
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 of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)
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: No way to reduce to the start symbol E

• This is resolved by various bottom-up parsing algorithms
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:

\[
E \quad \rightarrow \quad E + E \\
\mid \quad E * E \\
\mid \quad (E) \\
\mid \quad \text{int}
\]
One Shift-Reduce Parse

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

\[
| \text{int * int + int} \quad \text{shift} \\
\ldots \\
E \ast E \mid + \text{int} \quad \text{shift} \\
E \ast E + \mid \text{int} \quad \text{shift} \\
E \ast E + \text{int} \mid \text{reduce } E \rightarrow \text{int} \\
E \ast E + E \mid \text{reduce } E \rightarrow E + E \\
E \ast E \mid \text{reduce } E \rightarrow E \ast E \\
E \mid \text{reduce } E \rightarrow E
\]
Example Notes

- In the second step $E \cdot E \mid + \text{ int}$ we can either shift or reduce by $E \rightarrow E \cdot E$
- Choice determines associativity of $+$ and $\cdot$
- 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 $E \ast E \mid + \text{int}$
• More precisely, precedence declaration is used to resolve conflict between reducing a * and shifting a +
Precedence Declarations Revisited (Cont.)

• The term “precedence declaration” is a bit misleading

• These declarations do not define precedence; they define conflict resolutions
  – Not quite the same thing!