Introduction to Bottom-Up Parsing

Outline

- Review LL parsing
- Shift-reduce parsing
- The LR parsing algorithm
- Constructing LR parsing tables

Top-Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

```
E
  T
+ E
```

```
int * int + int
```
Top-Down Parsing: Review

• Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

- The leaves at any point form a string $\beta A \gamma$
  - $\beta$ contains only terminals
  - The input string is $\beta b \delta$
  - The prefix $\beta$ matches
  - The next token is $b$

$E \rightarrow I \ T^* I$
$E \rightarrow I \ T$

Predictive Parsing: Review

• A predictive parser is described by a table
  - For each non-terminal $A$ and for each token $b$ we specify a production $A \rightarrow \alpha$
  - When trying to expand $A$ we use $A \rightarrow \alpha$ if $b$ follows next

- Once we have the table
  - The parsing algorithm is simple and fast
  - No backtracking is necessary

Constructing Predictive Parsing Tables

1. Consider the state $S \rightarrow^* \beta A \gamma$
   - With $b$ the next token
   - Trying to match $\beta b \delta$

There are two possibilities:

• $b$ belongs to an expansion of $A$
  - Any $A \rightarrow \alpha$ can be used if $b$ can start a string derived from $\alpha$
  - We say that $b \in \text{First}(\alpha)$

Or...
2. \( b \) does not belong to an expansion of \( A \)
- The expansion of \( A \) is empty and \( b \) belongs to an
  expansion of \( \gamma \)
- Means that \( b \) can appear after \( A \) in a derivation of
  the form \( S \rightarrow^* \beta A \delta \)
- We say that \( b \in \text{Follow}(A) \) in this case

What productions can we use in this case?
- Any \( A \rightarrow \alpha \) can be used if \( \alpha \) can expand to \( \epsilon \)
- We say that \( \epsilon \in \text{First}(A) \) in this case

First Sets: Example
- Recall the grammar
  \[
  \begin{align*}
  E & \rightarrow T X & X & \rightarrow + E | \epsilon \\
  T & \rightarrow ( E ) | \text{int} \ Y & Y & \rightarrow * T | \epsilon
  \end{align*}
  \]
- First sets
  \[
  \begin{align*}
  \text{First}(\ (\ ) &= \{\ (\ )\} & \text{First}(\ T) &= \{\text{int}, (\ )\} \\
  \text{First}(\ ) &= \{\ )\} & \text{First}(\ E) &= \{\text{int}, (\ )\} \\
  \text{First}(\ \text{int}) &= \{\ \text{int}\} & \text{First}(\ X) &= \{+, \epsilon\} \\
  \text{First}(\ +) &= \{+, \epsilon\} & \text{First}(\ Y) &= \{*, \epsilon\} \\
  \text{First}(\ *) &= \{*, \epsilon\}
  \end{align*}
  \]

Computing First Sets
- **Definition**
  \[
  \text{First}(X) = \{ b \mid X \rightarrow^* b \alpha \} \cup \{ \epsilon \mid X \rightarrow^* \epsilon \}
  \]
- **Algorithm sketch**
  1. \( \text{First}(b) = \{ b \} \)
  2. \( \epsilon \in \text{First}(X) \) if \( X \rightarrow \epsilon \) is a production
  3. \( \epsilon \in \text{First}(X) \) if \( X \rightarrow A_1 \ldots A_n \)
     and \( \epsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)
  4. \( \text{First}(\alpha) \subseteq \text{First}(X) \) if \( X \rightarrow A_1 \ldots A_n \alpha \)
     and \( \epsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)

Computing Follow Sets
- **Definition**
  \[
  \text{Follow}(X) = \{ b \mid S \rightarrow^* \beta X \delta \}
  \]
- **Intuition**
  - If \( X \rightarrow A B \) then \( \text{First}(B) \subseteq \text{Follow}(A) \)
    and \( \text{Follow}(X) \subseteq \text{Follow}(B) \)
  - Also if \( B \rightarrow^* \epsilon \) then \( \text{Follow}(X) \subseteq \text{Follow}(A) \)
  - If \( S \) is the start symbol then \( $ \in \text{Follow}(S) \)
Computing Follow Sets (Cont.)

Algorithm sketch
1. $ \in \text{Follow}(S)$
2. First($\beta$) - $\{\varepsilon\} \subseteq \text{Follow}(X)$
   - For each production $A \rightarrow \alpha X \beta$
3. Follow($A$) $\subseteq$ Follow($X$)
   - For each production $A \rightarrow \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$

Follow Sets: Example

- Recall the grammar
  \[
  E \rightarrow TX \\
  T \rightarrow (E) \mid \text{int } Y \\
  Y \rightarrow *T \mid \varepsilon
  \]
- Follow sets
  \[
  \begin{align*}
  \text{Follow}(+) &= \{\text{int, (}\} \\
  \text{Follow}(\cdot) &= \{\text{int, (}\} \\
  \text{Follow}(\emptyset) &= \{\text{int, (}\} \\
  \text{Follow}(E) &= \{\), $\} \\
  \text{Follow}(X) &= \{\$\} \\
  \text{Follow}(T) &= \{\), $\} \\
  \text{Follow}(Y) &= \{\), $\} \\
  \text{Follow}(\text{int}) &= \{\$, $\}
  \end{align*}
  \]

Constructing LL(1) Parsing Tables

- Construct a parsing table $T$ for CFG $G$
- For each production $A \rightarrow \alpha$ in $G$ do:
  - For each terminal $b \in \text{First}(\alpha)$ do
    - $T[A, b] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $b \in \text{Follow}(A)$ do
    - $T[A, b] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    - $T[A, \$] = \alpha$

Constructing LL(1) Tables: Example

- Recall the grammar
  \[
  E \rightarrow TX \\
  T \rightarrow (E) \mid \text{int } Y \\
  Y \rightarrow *T \mid \varepsilon
  \]
- Where in the line of $Y$ we put $Y \rightarrow *T$?
  - In the lines of First($\text{int}$) = $\{\star\}$
- Where in the line of $Y$ we put $Y \rightarrow \varepsilon$?
  - In the lines of Follow($Y$) = $\{\$, $\}$
Notes on LL(1) Parsing Tables

• If any entry is multiply defined then $G$ is not LL(1)
  - If $G$ is ambiguous
  - If $G$ is left recursive
  - If $G$ is not left-factored
  - And in other cases as well

• For some grammars there is a simple parsing strategy: *Predictive parsing*

• Most programming language grammars are not LL(1)
• Thus, we need more powerful parsing strategies

Bottom-Up Parsing

• Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
  - Preferred method in practice

• Also called LR parsing
  - L means that tokens are read left to right
  - R means that it constructs a rightmost derivation!

An Introductory Example

• LR parsers don't need left-factored grammars and can also handle left-recursive grammars

• Consider the following grammar:
  \[
  E \rightarrow E + ( E ) | \text{int}
  \]
  - Why is this not LL(1)?

• Consider the string: \text{int} + ( \text{int} ) + ( \text{int} )
The Idea

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

\[
\text{str } w \text{ input string of terminals}
\]

\[
\text{repeat}
\]

- Identify \( \beta \) in \( \text{str} \) such that \( A \rightarrow \beta \) is a production (i.e., \( \text{str} = \alpha \beta \gamma \))

- Replace \( \beta \) by \( A \) in \( \text{str} \) (i.e., \( \text{str} w = \alpha A \gamma \))

until \( \text{str} = S \) (the start symbol)

OR all possibilities are exhausted

A Bottom-up Parse in Detail (1)

\[
\text{int } + (\text{int}) + (\text{int})
\]

A Bottom-up Parse in Detail (2)

\[
\text{int } + (\text{int}) + (\text{int})
\]

\[
E + (\text{int}) + (\text{int})
\]

A Bottom-up Parse in Detail (3)

\[
E + (\text{int}) + (\text{int})
\]

\[
E + (E) + (\text{int})
\]
A Bottom-up Parse in Detail (4)

\[ \text{int} + (\text{int}) + (\text{int}) \]
\[ E + (\text{int}) + (\text{int}) \]
\[ E + (E) + (\text{int}) \]
\[ E + (\text{int}) \]

A Bottom-up Parse in Detail (5)

\[ \text{int} + (\text{int}) + (\text{int}) \]
\[ E + (\text{int}) + (\text{int}) \]
\[ E + (E) + (\text{int}) \]
\[ E + (\text{int}) \]

A Bottom-up Parse in Detail (6)

\[ \text{int} + (\text{int}) + (\text{int}) \]
\[ E + (\text{int}) + (\text{int}) \]
\[ E + (E) + (\text{int}) \]
\[ E + (\text{int}) \]
\[ E + (E) \]

A rightmost derivation in reverse

A Bottom-up Parse in Detail (6)

Important Fact #1

Important Fact #1 about bottom-up parsing:

An LR parser traces a rightmost derivation in reverse.
**Where Do Reductions Happen**

Important Fact #1 has an interesting consequence:
- Let $\alpha\beta\gamma$ be a step of a bottom-up parse
- Assume the next reduction is by using $A \rightarrow \beta$
- Then $\gamma$ is a string of terminals

Why? Because $\alpha A \gamma \rightarrow \alpha \beta \gamma$ is a step in a right-most derivation

**Notation**

- Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals
- The dividing point is marked by a $I$
  - The $I$ is not part of the string
- Initially, all input is unexamined: $I x_1 x_2 \ldots x_n$

**Shift-Reduce Parsing**

Bottom-up parsing uses only two kinds of actions:

- **Shift**: Move $I$ one place to the right
  - Shifts a terminal to the left string
  
  $E + (I \text{ int }) \Rightarrow E + (\text{ int } I)$

  In general:
  
  $ABC I xyz \Rightarrow ABCx I yz$

- **Reduce**
Reduce

Reduction: Apply an inverse production at the right end of the left string
- If \( E \rightarrow E + (E) \) is a production, then

\[
E + (E + (E)) \Rightarrow E + (E)
\]

In general, given \( A \rightarrow xy \), then:

\[
Cbxy \Rightarrow CbA
\]

Shift-Reduce Example

\[
i \text{int } + \text{(int)} + \text{(int)} \\
\text{shift}
\]

\[
i \text{int } + \text{(int)} + \text{(int)} \\
\text{reduce}\ E \rightarrow \text{int}
\]

\[
i \text{int } + \text{(int)} + \text{(int)}
\]

\[
i \text{int } + \text{(int)} + \text{(int)}
\]

\[
i \text{int } + \text{(int)} + \text{(int)}
\]
Shift-Reduce Example

\[
\text{int } + (\text{int }) + (\text{int}) \\
\text{shift 3 times}
\]

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

\[
\text{int } + (\text{int }) + (\text{int}) \\
\text{shift 3 times}
\]

\[
\text{E } + (\text{E} ) + (\text{E}) \\
\text{reduce E} \rightarrow \text{int}
\]
Shift-Reduce Example

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

\[
\begin{align*}
1 & \text{ int } + (\text{int}) + (\text{int})$
\text{shift} \\
& \text{int } 1 + (\text{int}) + (\text{int})$
\text{reduce } E \rightarrow \text{int} \\
& E 1 + (\text{int}) + (\text{int})$
\text{shift 3 times} \\
& E + (\text{int } 1) + (\text{int})$
\text{reduce } E \rightarrow \text{int} \\
& E + (E 1) + (\text{int})$
\text{shift} \\
& E + (E) 1 + (\text{int})$
\text{reduce } E \rightarrow E + (E) \\
& E 1 + (\text{int})$
\text{shift 3 times} \\
& E + (\text{int } 1)$
\text{reduce } E \rightarrow \text{int} \\
& E + (E 1)$
\text{shift} \\
& E + (E) 1$
\text{reduce } E \rightarrow E + (E) \\
& E 1$
\text{reduce } E \rightarrow \text{int} \\
& E + (E 1)$
\text{shift} \\
& E + (E) 1$
\text{reduce } E \rightarrow E + (E) \\
& E 1$
\text{accept}
\end{align*}
\]

Shift-Reduce Example

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

\[
\begin{align*}
1 & \text{ int } + (\text{int}) + (\text{int})$
\text{shift} \\
& \text{int } 1 + (\text{int}) + (\text{int})$
\text{reduce } E \rightarrow \text{int} \\
& E 1 + (\text{int}) + (\text{int})$
\text{shift 3 times} \\
& E + (\text{int } 1) + (\text{int})$
\text{reduce } E \rightarrow \text{int} \\
& E + (E 1) + (\text{int})$
\text{shift} \\
& E + (E) 1 + (\text{int})$
\text{reduce } E \rightarrow E + (E) \\
& E 1 + (\text{int})$
\text{shift 3 times} \\
& E + (\text{int } 1)$
\text{reduce } E \rightarrow \text{int} \\
& E + (E 1)$
\text{shift} \\
& E + (E) 1$
\text{reduce } E \rightarrow E + (E) \\
& E 1$
\text{accept}
\end{align*}
\]
### The Stack

- Left string can be implemented by a stack
  - Top of the stack is the $I$
- 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 Question: To Shift or to Reduce?

**Idea:** use a finite automaton (DFA) to decide when to shift or reduce
- The input is the stack
- The language consists of terminals and non-terminals

- We run the DFA on the stack and we examine the resulting state $X$ and the token $tok$ after $I$
  - If $X$ has a transition labeled $tok$ then shift
  - If $X$ is labeled with “$A \rightarrow \beta$ on tok” then reduce

### LR(1) Parsing: An Example

#### LR(1) Parsing: An Example

<table>
<thead>
<tr>
<th>State</th>
<th>Token</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>int + (int) + (int)$</td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td>E → int on $, +</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>E + (int)$</td>
<td>shift (x3)</td>
</tr>
<tr>
<td>5</td>
<td>E + (E) + (int)$</td>
<td>E → E+(E)</td>
</tr>
<tr>
<td>6</td>
<td>E + (int)$</td>
<td>shift (x3)</td>
</tr>
<tr>
<td>7</td>
<td>E + (int)</td>
<td>E → int</td>
</tr>
<tr>
<td>8</td>
<td>E + (E)</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>E + (E)$</td>
<td>E → E+(E)</td>
</tr>
<tr>
<td>10</td>
<td>E → E + (E)</td>
<td>accept</td>
</tr>
</tbody>
</table>

### Representing the DFA

- Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for terminals: action table
  - Those for non-terminals: goto table
Representing the DFA: Example

- The table for a fragment of our DFA:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>+</th>
<th>( )</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated

- Remember for each stack element on which state it brings the DFA

- LR parser maintains a stack

\[
\langle \text{sym}_1, \text{state}_1 \rangle \ldots \langle \text{sym}_n, \text{state}_n \rangle
\]

\(\text{state}_k\) is the final state of the DFA on \(\text{sym}_1 \ldots \text{sym}_k\)

LR Parsers

- Can be used to parse more grammars than LL

- Most programming languages grammars are LR

- LR Parsers can be described as a simple table

- There are tools for building the table

- How is the table constructed?