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

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

int \ast \text{int} + \text{int}

• 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 \( b \)
Top-Down Parsing: Review

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

\[
E \to T \ast E \\
T \to \text{int} \ast T \downarrow \text{int} \\
E \to \text{int} + \text{int} \\
\]

- 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 \( b \).

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 \( A \to \alpha \).
  - When trying to expand \( A \) we use \( A \to \alpha \) if \( b \) is the token that follows next.

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

Constructing Predictive Parsing Tables

Consider the state \( S \to \ast \beta A \gamma \):

- With \( b \) the next token.
- Trying to match \( \beta b \delta \).

There are two possibilities:

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

Or...
2. Token $b$ 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 $S \rightarrow^* \beta Ab\omega$
   - 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 $\varepsilon$
   - We say that $\varepsilon \in \text{First}(A)$ in this case

---

First Sets: Example

- Recall the grammar
  
  $\begin{align*}
  &E \rightarrow T \ X \\
  &T \rightarrow (E) \mid \text{int} \\
  &X \rightarrow + E \mid \varepsilon \\
  &Y \rightarrow * T \mid \varepsilon
  \end{align*}$

- First sets
  
  $\begin{align*}
  &\text{First}(\ ) = \{ \}\quad &\text{First}(T) = \{ \text{int}, () \} \\
  &\text{First}(\ ) = \{ () \} &\text{First}(E) = \{ \text{int}, () \} \\
  &\text{First}(\text{int}) = \{ \text{int} \} &\text{First}(X) = \{ +, \varepsilon \} \\
  &\text{First}(+) = \{ + \} &\text{First}(Y) = \{ *, \varepsilon \} \\
  &\text{First}(* ) = \{ * \}
  \end{align*}$
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. $\text{Follow}(A) \subseteq \text{Follow}(X)$
   - For each production $A \rightarrow \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$

Follow Sets: Example

• Recall the grammar
  $E \rightarrow T X$
  $X \rightarrow + E | \varepsilon$
  $T \rightarrow ( E ) | \text{int} Y$
  $Y \rightarrow * T | \varepsilon$

• Follow sets
  $\text{Follow}( + ) = \{ \text{int, (} \}$
  $\text{Follow}( * ) = \{ \text{int, (} \}$
  $\text{Follow}( ( ) = \{ \text{int, (} \}$
  $\text{Follow}( E ) = \{ ), $ \}
  $\text{Follow}( X ) = \{ +, \varepsilon \}$
  $\text{Follow}( Y ) = \{ +, \varepsilon \}$
  $\text{Follow}( \text{int} ) = \{ *, +, \varepsilon \}$

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 $\varepsilon \in \text{Follow}(A)$ do $T[A, \varepsilon] = \alpha$

Constructing LL(1) Tables: Example

• Recall the grammar
  $E \rightarrow T X$
  $X \rightarrow + E | \varepsilon$
  $T \rightarrow ( E ) | \text{int} Y$
  $Y \rightarrow * T | \varepsilon$

• Where in the line of $Y$ do we put $Y \rightarrow * T$?
  - In the lines of First(*$T$) = \{ * \}

• Where in the line of $Y$ do we put $Y \rightarrow \varepsilon$?
  - In the lines of Follow($Y$) = \{ $, +, \varepsilon \}$
### 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 + ( int ) + ( 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., \( \text{str } w = \alpha A \gamma \))

\[ \text{until } \text{str} = S \text{ (the start symbol)} \]
\[ \text{OR all possibilities are exhausted} \]

A Bottom-up Parse in Detail (1)

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

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

A Bottom-up Parse in Detail (2)

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

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

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

A Bottom-up Parse in Detail (3)

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

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

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

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

E → E + (E) | int

E + (int) + (int)
E + (E) + (int)
E + (int)

A rightmost derivation in reverse

A Bottom-up Parse in Detail (5)

E → E + (E) | int

E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)

A rightmost derivation in reverse

Important Fact #1 about Bottom-up Parsing

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

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

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

- **Shift**
- **Reduce**

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:

$ABC I \text{xyz} \Rightarrow ABCx I \text{yz}$

Notaion

- 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: $I x_1 x_2 \ldots x_n$
**Reduce**

*Reduce*: 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**

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

<table>
<thead>
<tr>
<th>Shift-Reduce Example</th>
<th>Shift-Reduce Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>Shift</td>
</tr>
<tr>
<td>\text{int} (\text{int}) + (\text{int}$)</td>
<td>\text{int} + (\text{int}) + (\text{int}$)</td>
</tr>
<tr>
<td>\text{int} (\text{int}) + (\text{int}$)</td>
<td>\text{int} + (\text{int}) + (\text{int}$)</td>
</tr>
</tbody>
</table>

\[
\text{shift 3 times}
\]

\[
\text{reduce } E \rightarrow \text{int}
\]

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

\[
\text{shift 3 times}
\]
Shift-Reduce Example

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ E \rightarrow \text{int} + (E) ]</td>
<td>Shif</td>
<td>Shift 3 times</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
</tbody>
</table>

Shift-Reduce Example

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ E \rightarrow \text{int} + (E) ]</td>
<td>Shif</td>
<td>Shift 3 times</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
</tbody>
</table>

Shift-Reduce Example

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ E \rightarrow \text{int} + (E) ]</td>
<td>Shif</td>
<td>Shift 3 times</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
</tbody>
</table>

Shift-Reduce Example

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ E \rightarrow \text{int} + (E) ]</td>
<td>Shif</td>
<td>Shift 3 times</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
<tr>
<td>[ E + \text{int} \rightarrow \text{int} ]</td>
<td>Red</td>
<td>Reduce ( E \rightarrow \text{int} )</td>
</tr>
</tbody>
</table>
Shift-Reduce Example

```
E → E + ( E ) | int
```

```
I int + (int) + (int)
int I + (int) + (int)
E I + (int) + (int)
E + (int I) + (int)
E + (E I) + (int)
E + (E) I + (int)
E I + (int)
E + (E I)
E + (E) I
E I
```

Shift-Reduce Example

```
E → E + ( E ) | int
```

```
I int + (int) + (int)
int I + (int) + (int)
E I + (int) + (int)
shift 3 times
E + (int I) + (int)
reduce E → int
E + (E I) + (int)
shift
E + (E) I + (int)
reduce E → int
E I + (int)
shift 3 times
E + (int I)
reduce E → int
E + (E I)
shift
E + (E) I
reduce E → int
E I
accept
```
**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 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 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**

```
int E \rightarrow int
  on $, +
  accept
on $

E \rightarrow E + (E)
  on $, +

int + (int) + (int)$ shift
int E \rightarrow int
E I + (int) + (int)$ shift(x3)
E + (int I ) + (int)$ E \rightarrow int
E + (E I ) + (int)$ shift
E + (E) I + (int)$ E \rightarrow E+(E)
E I + (int)$ shift (x3)
E + (int I )$ E \rightarrow int
E + (E I)$ shift
E + (E) I$ E \rightarrow E+(E)
E I$ accept
```

---

**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
    - action = shift or reduce
  - 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>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>s4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td>g6</td>
</tr>
<tr>
<td>5</td>
<td>rE→int</td>
<td>rE→int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s8</td>
<td></td>
<td></td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>rE→E+(E)</td>
<td>rE→E+(E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

E → E + (E) on $, +

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
  \[ (\text{sym}_1, \text{state}_1) \ldots (\text{sym}_n, \text{state}_n) \]
  state_k is the final state of the DFA on \(\text{sym}_1 \ldots \text{sym}_k\)

The LR Parsing Algorithm

let \(I = w\$\) be initial input
let \(j = 0\)
let DFA state 0 be the start state
let stack = \(\langle \text{dummy}, 0 \rangle\)

repeat
  case action[top_state(stack), I[j]] of
    shift \(k\): push \(\langle I[j+1], k \rangle\)
    reduce \(X \rightarrow A\):
      pop \(|A|\) pairs,
      push \(\langle X, \text{Goto}[\text{top_state(stack)}, X] \rangle\)
    accept: halt normally
    error: halt and report error

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?