## Introduction to Bottom-Up Parsing

<table>
<thead>
<tr>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Review LL parsing</td>
</tr>
<tr>
<td>• Shift-reduce parsing</td>
</tr>
<tr>
<td>• The LR parsing algorithm</td>
</tr>
<tr>
<td>• Constructing LR parsing tables</td>
</tr>
</tbody>
</table>

### Top-Down Parsing: Review

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

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

- 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

\[
E \rightarrow T^* E \\
T \rightarrow \text{int}^* T \\
T \rightarrow \text{int} \\
\]

- 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 \( b \)

\[
\text{int}^* \text{int} + \text{int} 
\]

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 \( b \) 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 \( S \rightarrow^* \beta A \gamma \)
  - 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 \( A \rightarrow \alpha \) can be used if \( b \) can start a string derived from \( \alpha \)
   - We say that \( b \in \text{First}(\alpha) \)

Or...
Constructing Predictive Parsing Tables (Cont.)

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

First Sets: Example

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

- First sets
  
  \[
  \begin{align*}
  \text{First}(\text{( }) &= \{ \text{( } \} \\
  \text{First}(\text{)} &= \{ \text{, ( } \} \\
  \text{First}(\text{int }) &= \{ \text{int} \} \\
  \text{First}(\text{+ }) &= \{ + \} \\
  \text{First}(\text{* }) &= \{ * \}
  \end{align*}
  \]

Computing First Sets

- Definition
  
  \[
  \text{First}(X) = \{ b \mid X \rightarrow^* b \alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \}
  \]

- Algorithm sketch
  
  1. \( \text{First}(b) = \{ b \} \)
  2. \( \varepsilon \in \text{First}(X) \) if \( X \rightarrow \varepsilon \) is a production
  3. \( \varepsilon \in \text{First}(X) \) if \( X \rightarrow A_1 \ldots A_n \) and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)
  4. \( \text{First}(\alpha) \subseteq \text{First}(X) \) if \( X \rightarrow A_1 \ldots A_n \alpha \) and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)

Computing Follow Sets

- Definition
  
  \[
  \text{Follow}(X) = \{ b \mid S \rightarrow^* \beta X b\delta \}
  \]

- Intuition
  
  - If \( X \rightarrow A B \) then \( \text{First}(B) \subseteq \text{Follow}(A) \) and \( \text{Follow}(X) \subseteq \text{Follow}(B) \)
  - Also if \( B \rightarrow^* \varepsilon \) 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($\beta$) - {ε} $\subseteq$ 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 ε $\in$ First($\beta$)

Follow Sets: Example

• Recall the grammar
  
  \[
  E \rightarrow T X \\
  X \rightarrow + E | \epsilon \\
  T \rightarrow (E) | \text{int} \ Y \\
  Y \rightarrow * T | \epsilon
  \]

• Follow sets
  
  \[
  \begin{align*}
  \text{Follow( + )} &= \{ \text{int}, ( \} \\
  \text{Follow( * )} &= \{ \text{int}, ( \} \\
  \text{Follow( ( )} &= \{ \text{int}, ( \} \\
  \text{Follow( E )} &= \{ ), \} \\
  \text{Follow( X )} &= \{ $, ) \} \\
  \text{Follow( T )} &= \{ +, ), $ \} \\
  \text{Follow( Y )} &= \{ +, ), $ \} \\
  \text{Follow( int )} &= \{ *, +, ), $ \}
  \end{align*}
  \]

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 ε $\in$ First($\alpha$), for each $b \in \text{Follow}(A$) do
    
    $T[A, b] = \alpha$
  - If ε $\in$ First($\alpha$) and $\epsilon \in \text{Follow}(A)$ do
    
    $T[A, \epsilon] = \alpha$

Constructing LL(1) Tables: Example

• Recall the grammar
  
  \[
  E \rightarrow T X \\
  X \rightarrow + E | \epsilon \\
  T \rightarrow (E) | \text{int} \ Y \\
  Y \rightarrow * T | \epsilon
  \]

• 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 $Y \rightarrow \epsilon$?
  
  - 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:

str w input string of terminals
repeat
  - Identify β in str such that A → β is a production (i.e., str = αβγ)
  - Replace β by A in str (i.e., str w = αAγ)
until str = S (the start symbol)
  OR all possibilities are exhausted

A Bottom-up Parse in Detail (1)

E → E + (E) | int
int + (int) + (int)

A Bottom-up Parse in Detail (2)

E → E + (E) | int
int + (int) + (int)
E + (int) + (int)

A Bottom-up Parse in Detail (3)

E → E + (E) | int
int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
A Bottom-up Parse in Detail (4)

$E \rightarrow E + (E) \mid int$

- $int + (int) + (int)$
- $E + (int) + (int)$
- $E + (E) + (int)$
- $E + (int)$

A Bottom-up Parse in Detail (5)

$E \rightarrow E + (E) \mid int$

- $int + (int) + (int)$
- $E + (int) + (int)$
- $E + (E) + (int)$
- $E + (int)$
- $E + (E)$

A Bottom-up Parse in Detail (6)

$E \rightarrow E + (E) \mid int$

- $int + (int) + (int)$
- $E + (int) + (int)$
- $E + (E) + (int)$
- $E + (int)$
- $E + (E)$
- $E$

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

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: $Ix_1x_2 \ldots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

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

Shift:
**Shift**: Move $I$ one place to the right
- Shifts a terminal to the left string

In general:
$E + (I \text{ int } ) \Rightarrow E + (\text{ int } I )$

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

\[
C_{bxy} \Rightarrow C_{bA}
\]

Shift-Reduce Example

<table>
<thead>
<tr>
<th>Action</th>
<th>Symbol</th>
<th>State</th>
<th>Action</th>
<th>Symbol</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>int</td>
<td></td>
<td>Reduce</td>
<td>E</td>
<td>int</td>
</tr>
<tr>
<td>Shift</td>
<td>int</td>
<td>int</td>
<td>Reduce</td>
<td>E</td>
<td>int</td>
</tr>
<tr>
<td>Shift</td>
<td>int</td>
<td>int</td>
<td>Reduce</td>
<td>E</td>
<td>int</td>
</tr>
</tbody>
</table>
**Shift-Reduce Example**

1. int + (int) + (int) $  
   \text{shift}

2. int I + (int) + (int) $  
   \text{reduce } E \rightarrow \text{int}

3. E I + (int) + (int) $  
   \text{shift 3 times}

4. E + (int I) + (int) $  
   \text{reduce } E \rightarrow \text{int}

5. E + (E I) + (int) $  
   \text{shift}

6. E + (E I) + (int) $  
   \text{reduce } E \rightarrow \text{int}

7. E + (int I) + (int) $  
   \text{shift 3 times}

8. E + (int I) + (int) $  
   \text{reduce } E \rightarrow \text{int}

9. E + (E I) + (int) $  
   \text{shift}

10. E + (E I) + (int) $  
    \text{reduce } E \rightarrow E + (E)$

11. E I + (int) $  
    \text{shift 3 times}

12. E I + (int) $  
    \text{reduce } E \rightarrow \text{int}

13. E + (E I) + (int) $  
    \text{shift}

14. E + (E I) + (int) $  
    \text{reduce } E \rightarrow E + (E)$

15. E I + (int) $  
    \text{shift 3 times}
**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 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
```

```
int
E → int 
on $, +
accept
on $, +
E → E + (E)
on $, +
E → E + (E)
on $, +
E → int
```

```
Representing the DFA
```

- Parsers represent the DFA as a 2D table
  - 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>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></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></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r_E→int</td>
<td>r_E→int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>s8</td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r_E→E+(E)</td>
<td>r_E→E+(E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></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?

```plaintext
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++], k \rangle
    reduce X → A:
      pop |A| pairs,
      push \langle X, Goto[top_state(stack), X] \rangle
    accept: halt normally
    error: halt and report error
```