LR Parsing
Bottom-Up Parsing

Outline

• No Stopping The Parsing!
• Bottom-Up Parsing
  • LR Parsing
    - Shift and Reduce
    - LR(1) Parsing Algorithm
• LR(1) Parsing Tables

In One Slide

• An LR(1) parser reads tokens from left to right and constructs a bottom-up rightmost derivation. LR(1) parsers shift terminals and reduce the input by application productions in reverse. LR(1) parsing is fast and easy, and uses a finite automaton with a stack. LR(1) works fine if the grammar is left-recursive, or not left-factored.
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:

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

  - Why is this **not** LL(1)? (Guess before I show you!)

- Consider the string: \( \text{int} + ( \text{int} ) + ( \text{int} ) \)

The Idea

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

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

  \[ \text{repeat} \]

  - 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., \( \text{str} \) becomes \( \alpha A \gamma \))

  until \( \text{str} = S \)
A Bottom-up Parse in Detail (1)

```
int + (int) + (int)
```

```
int + (int) + (int)
```

A Bottom-up Parse in Detail (2)

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

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

A Bottom-up Parse in Detail (3)

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

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

\[
\begin{align*}
\text{int + (int) * (int)} \\
E + (int) * (int) \\
E + (E) * (int) \\
E + (int) \\
E \\
\end{align*}
\]

A Bottom-up Parse in Detail (5)

\[
\begin{align*}
\text{int + (int) * (int)} \\
E + (int) * (int) \\
E + (E) * (int) \\
E + (int) \\
E + (E) \\
\end{align*}
\]

A Bottom-up Parse in Detail (6)

\[
\begin{align*}
\text{int + (int) * (int)} \\
E + (int) * (int) \\
E + (E) * (int) \\
E + (int) \\
E + (E) \\
\end{align*}
\]

A rightmost derivation in reverse
Important Fact

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 αβγ be a step of a bottom-up parse
- Assume the next reduction is by A → β
- Then γ is a string of terminals!

Why? Because αAγ → αβγ is a step in a rightmost derivation

Notation

- Idea: Split the string into two substrings
  - Right substring (a string of terminals) is as yet unexamined by parser
  - 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 new: ▶x₁x₂...xₙ
Shift-Reduce Parsing

• Bottom-up parsing uses only two kinds of actions:
  
  \textit{Shift}

  \textit{Reduce}

\textbf{Shift}

\textit{Shift}: Move \(\rightarrow\) one place to the right
- Shifts a terminal to the left string

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

\[ \Rightarrow \]

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

\textbf{Reduce}

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

\[ E + (E + (E) \rightarrow) \]

\[ \Rightarrow \]

\[ E + (T \rightarrow) \]
Shift-Reduce Example

- `int * (int) + (int)$` shift

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

Shift-Reduce Example

- `int * (int) + (int)$` shift
- `int * (int) + (int)$` red. $E \rightarrow \text{int}$

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

Shift-Reduce Example

- `int * (int) + (int)$` shift
- `int * (int) + (int)$` red. $E \rightarrow \text{int}$
- `E * (int) + (int)$` shift 3 times

\[
E
\]

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

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

\[
E \\
\text{int} + \{ \text{int} \} + \{ \text{int} \}
\]
Shift-Reduce Example

\[ int + (int) \rightarrow (int) \]
\[ int \rightarrow (int) + (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (E) \]
\[ E \rightarrow int \]

\[ E \rightarrow (E) \]
\[ E \rightarrow (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (E) \]
\[ E \rightarrow int \]

Shift-Reduce Example

\[ int + (int) \rightarrow (int) \]
\[ int \rightarrow (int) + (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (E) \]
\[ E \rightarrow int \]

\[ E \rightarrow (E) \]
\[ E \rightarrow (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (E) \]
\[ E \rightarrow int \]

Shift-Reduce Example

\[ int + (int) \rightarrow (int) \]
\[ int \rightarrow (int) + (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (E) \]
\[ E \rightarrow int \]

\[ E \rightarrow (E) \]
\[ E \rightarrow (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (int) \]
\[ E \rightarrow (int) + (int) \]
\[ E \rightarrow (E) + (E) \]
\[ E \rightarrow int \]
Shift-Reduce Example

1. int + (int) + (int) $\rightarrow$ shift
2. int $\rightarrow$ (int) + (int) $\rightarrow$ red. $E \rightarrow$ int
3. $E \rightarrow$ (int) + (int) $\rightarrow$ shift 3 times
4. $E \rightarrow$ (int) + (int) $\rightarrow$ red. $E \rightarrow$ int
5. $E \rightarrow$ (int) + (int) $\rightarrow$ shift
6. $E \rightarrow$ (int) + (int) $\rightarrow$ red. $E \rightarrow$ int
7. $E \rightarrow$ (E) $\rightarrow$ (int) $\rightarrow$ shift
8. $E \rightarrow$ (E) $\rightarrow$ (int) $\rightarrow$ red. $E \rightarrow$ int
9. $E \rightarrow$ (E) $\rightarrow$ (int) $\rightarrow$ shift
10. $E \rightarrow$ (E) $\rightarrow$ (int) $\rightarrow$ red. $E \rightarrow$ int
11. $E \rightarrow$ (E) $\rightarrow$ (int) $\rightarrow$ shift
12. $E \rightarrow$ (E) $\rightarrow$ (int) $\rightarrow$ red. $E \rightarrow$ int
13. $E \rightarrow$ (E) $\rightarrow$ (int) $\rightarrow$ shift
14. $E \rightarrow$ (E) $\rightarrow$ (int) $\rightarrow$ red. $E \rightarrow$ int
15. $E \rightarrow$ (E) $\rightarrow$ (int) $\rightarrow$ shift
16. $E \rightarrow$ (E) $\rightarrow$ (int) $\rightarrow$ red. $E \rightarrow$ int

The Stack

- Left string can be implemented as a stack
  - Top of the stack is the $\downarrow$
- **Shift pushes** a terminal on the stack
- **Reduce pops** 0 or more symbols from the stack (production RHS) and **pushes** a non-terminal on the stack (production LHS)
Key Issue: When to Shift or Reduce?

- Decide based on the left string (the stack)
- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The DFA input is the stack
  - DFA language consists of terminals and nonterminals

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

LR(1) Parsing Example

```
<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Input</th>
<th>Shift/Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int</td>
<td>$</td>
<td>accept</td>
</tr>
<tr>
<td>1</td>
<td>int</td>
<td>(</td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>(</td>
<td>shift(x3)</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>)</td>
<td>shift(x)</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>)</td>
<td>shift</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>$</td>
<td>shift</td>
</tr>
<tr>
<td>6</td>
<td>int</td>
<td>)</td>
<td>shift</td>
</tr>
<tr>
<td>7</td>
<td>int</td>
<td>$</td>
<td>shift</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td>+</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>int</td>
<td>$</td>
<td>accept</td>
</tr>
<tr>
<td>10</td>
<td>E</td>
<td>+</td>
<td>shift</td>
</tr>
<tr>
<td>11</td>
<td>E</td>
<td>+</td>
<td>shift</td>
</tr>
</tbody>
</table>
```

Representing the DFA

- Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
- Lines (rows) 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
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

- **Optimization**: remember for each stack element to 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\_n is the final state of the DFA on sym\_1 \ldots sym\_n

---

The LR Parsing Algorithm

Let S = w$ be initial input
Let j = 0
Let DFA state 0 be the start state
Let stack = < dummy, 0 >

\[ \text{repeat} \]
  \[ \text{match action[top_state(stack), S[j]] with} \]
  | \text{shift k: push } < S[j++], k > |
  | \text{reduce } X \rightarrow \alpha: |
  | \quad \text{pop } \langle \alpha \rangle \text{ pairs,} |
  | \quad \text{push } < X, \text{Goto[top_state(stack), X] } > |
  | \text{accept: halt normally} |
  | \text{error: halt and report error} \]
LR Parsing Notes

• Can be used to parse more grammars than LL
• Most PL grammars are LR
• Can be described as a simple table
• There are tools for building the table
  - Often called “yacc” or “bison”
• How is the table constructed? Next time!

Homework

• Thursday: WA2 due
  - You may work in pairs.
• Thursday: Read 2.3.4-2.3.5, 2.4.2-2.4.3
• Next Friday: WA3 due
  - Parsing!