Top-Down Parsing and Intro to Bottom-Up Parsing

Lecture 7

Predictive Parsers

- Like recursive-descent but parser can “predict” which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means “left-to-right” scan of input
  - L means “leftmost derivation”
  - k means “predict based on k tokens of lookahead”
  - In practice, LL(1) is used

LL(1) vs. Recursive Descent

- In recursive-descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- In LL(1),
  - At each step, only one choice of production
    - That is
      - When a non-terminal A is leftmost in a derivation
      - The next input symbol is t
      - There is a unique production A → α to use
        - Or no production to use (an error state)
- LL(1) is a recursive descent variant without backtracking

Predictive Parsing and Left Factoring

- Recall the grammar
  \[
  E \rightarrow T \cdot E \mid T \\
  T \rightarrow \text{int} \mid \text{int} \cdot T \mid (E)
  \]
- Hard to predict because
  - For T two productions start with \text{int}
  - For E it is not clear how to predict
- We need to \textbf{left-factor} the grammar

Left-Factoring Example

- Recall the grammar
  \[
  E \rightarrow T \cdot X \\
  X \rightarrow + E \mid \epsilon \\
  T \rightarrow \text{(E)} \mid \text{int} \cdot Y \\
  Y \rightarrow * T \mid \epsilon
  \]
- Factor out common prefixes of productions
  \[
  E \rightarrow T \cdot X \\
  X \rightarrow * E \mid \epsilon \\
  T \rightarrow \text{(E)} \cdot \text{int} \cdot Y \\
  Y \rightarrow * T \mid \epsilon
  \]

LL(1) Parsing Table Example

- Left-factored grammar
  \[
  E \rightarrow T \cdot X \\
  X \rightarrow + E \mid \epsilon \\
  T \rightarrow \text{(E)} \mid \text{int} \cdot Y \\
  Y \rightarrow * T \mid \epsilon
  \]
- The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>T</th>
<th>X</th>
<th>*</th>
<th>( )</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td>T</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>-</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int</td>
<td>Y</td>
<td></td>
<td>*</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>*</td>
<td>T</td>
<td></td>
<td>( E )</td>
<td></td>
</tr>
</tbody>
</table>
LL(1) Parsing Table Example (Cont.)

- Consider the \([E, int]\) entry
  - “When current non-terminal is \(E\) and next input is int, use production \(E \rightarrow TX\)”
  - This can generate an int in the first position

- Consider the \([Y, +]\) entry
  - “When current non-terminal is \(Y\) and current token is +, get rid of \(Y\)”
  - \(Y\) can be followed by + only if \(Y \rightarrow \varepsilon\)

LL(1) Parsing Tables. Errors

- Blank entries indicate error situations

- Consider the \([E, \varepsilon]\) entry
  - “There is no way to derive a string starting with \(\varepsilon\) from non-terminal \(E\)”

Using Parsing Tables

- Method similar to recursive descent, except
  - For the leftmost non-terminal \(S\)
  - We look at the next input token \(a\)
  - And choose the production shown at \([S, a]\)

- A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet matched against the input
  - Top of stack = leftmost pending terminal or non-terminal

- Reject on reaching error state
- Accept on end of input & empty stack

LL(1) Parsing Algorithm

initialize stack = \(<S \>$) and next repeat case stack of
\(<X, \text{rest}>\) : if \(T[X, \text{next}] = Y_1...Y_n\) then stack \(\leftarrow <Y_1...Y_n, \text{rest}>\);
else error ();
\(<t, \text{rest}>\) : if \(t == \text{next}++\) then stack \(\leftarrow <\text{rest}>\);
else error ();
until stack == \(<>\)

LL(1) Parsing Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E &gt;$</td>
<td>int * int $</td>
<td>(TX )&gt;</td>
</tr>
<tr>
<td>(TX &gt;$</td>
<td>int * int $</td>
<td>(int Y)</td>
</tr>
<tr>
<td>(int YX &gt;$</td>
<td>int * int $</td>
<td>terminal</td>
</tr>
<tr>
<td>(XY &gt;$</td>
<td>* int $</td>
<td>(* T)</td>
</tr>
<tr>
<td>(* TX &gt;$</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>(TX &gt;$</td>
<td>int $</td>
<td>(int Y)</td>
</tr>
<tr>
<td>(int YX &gt;$</td>
<td>int $</td>
<td>terminal</td>
</tr>
<tr>
<td>(XY &gt;$</td>
<td>$</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>(X &gt;$</td>
<td>$</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>($)</td>
<td>$</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>
Constructing Parsing Tables: The Intuition

- Consider non-terminal A, production A → α, & token t
- T[A,t] = α in two cases:
  - If α →* t β
    - α can derive t in the first position
    - We say that t ∈ First(α)
  - If A → α and α →* ε and S →* β A t δ
    - Useful if stack has A, input is t, and A cannot derive t
    - In this case only option is to get rid of A (by deriving ε)
    - We say t ∈ Follow(A)

Computing First Sets

Definition
First(X) = { t | X →* t α } ∪ { ε | X →* ε }

Algorithm sketch:
1. First(t) = { t }
2. ε ∈ First(X)
   - if X → ε
   - if X → A₁...Aᵣ and ε ∈ First(Aᵢ) for 1 ≤ i ≤ r
3. First(α) ⊆ First(X)
   - and ε ∈ First(Aᵢ) for 1 ≤ i ≤ r

First Sets. Example

- Recall the grammar
  E → TX
  T → (E) | int Y
  X → + E | ε
  Y → * T | ε

- First sets
  First( ) = { ( ) } First( ) = { ( , ) } First( ) = { ( , ) } First( ) = { ( , ) } First( ) = { ( ) }

Follow Sets. Example

- Recall the grammar
  E → TX
  T → (E) | int Y
  X → * E | ε

- Follow sets
  Follow( ) = { ( ) } Follow( ) = { ( , ) } Follow( ) = { ( , ) } Follow( ) = { ( , ) } Follow( ) = { ( , ) } Follow( ) = { ( , ) } Follow( ) = { ( , ) } Follow( ) = { ( , ) }
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 $t \in \text{First}(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    - $T[A, \$] = \alpha$

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
- Most programming language CFGs are not LL(1)

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
- Concepts today, algorithms next time

An Introductory Example

- Bottom-up parsers don’t need left-factored grammars
- Revert to the “natural” grammar for our example:
  
  \[
  \begin{align*}
  E &\rightarrow T \cdot E \mid T \\
  T &\rightarrow \text{int} \cdot T \mid \text{int} \mid (E)
  \end{align*}
  \]
- Consider the string: \text{int} \cdot \text{int} + \text{int}

The Idea

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

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

Observation

- Read the productions in reverse (from bottom to top)
- This is a rightmost derivation!

\[
\begin{align*}
\text{int} \cdot \text{int} + \text{int} &\rightarrow \text{int} \\
\text{int} \cdot T \cdot \text{int} &\rightarrow \text{int} \cdot T \\
T \cdot \text{int} &\rightarrow \text{int} \\
T \cdot T &\rightarrow E \rightarrow T \\
T \cdot E &\rightarrow E \rightarrow \text{int} \cdot \text{int} \cdot \text{int} + \text{int} \rightarrow \text{int} \cdot \text{int} + \text{int} \rightarrow \text{int}
\end{align*}
\]
Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

A Bottom-up Parse

A Bottom-up Parse in Detail (1)

A Bottom-up Parse in Detail (2)

A Bottom-up Parse in Detail (3)

A Bottom-up Parse in Detail (4)
A Bottom-up Parse in Detail (5)

\[
\begin{align*}
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{T} + \text{int} \\
\text{T} & + \text{int} \\
\text{T} & + \text{T} \\
\text{T} & + \text{E} \\
\text{E} & \\
\end{align*}
\]

A Bottom-up Parse in Detail (6)

\[
\begin{align*}
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{T} + \text{int} \\
\text{T} & + \text{int} \\
\text{T} & + \text{T} \\
\text{T} & + \text{E} \\
\text{E} & \\
\end{align*}
\]

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

Questions

• Does this algorithm terminate?
• How fast is the algorithm?
• Does the algorithm handle all cases?
• How do we choose the substring to reduce at
each step?

Where Do Reductions Happen?

Important Fact #1 has an interesting
consequence:
- 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 \( \alpha X_0 \rightarrow \alpha \beta \omega \) 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 \( | \)
  - 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

Reduce

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

\[ ABC \rightarrow ABCx \rightarrow ABC | yz \]

Reduce

• Apply an inverse production at the right end of the left string
  - If \( A \rightarrow xy \) is a production, then

\[ Cbxy | ijk \rightarrow CbA | ijk \]

The Example with Reductions Only

\[
\begin{align*}
\text{int} \ast \text{int} | \ast \text{int} & \quad \text{reduce } T \rightarrow \text{int} \\
\text{int} \ast T | \ast \text{int} & \quad \text{reduce } T \rightarrow \text{int} \ast T \\
\text{T} \ast \text{int} | & \quad \text{reduce } T \rightarrow \text{int} \\
\text{T} \ast \text{T} | & \quad \text{reduce } E \rightarrow \text{T} \\
\text{T} \ast \text{E} | & \quad \text{reduce } E \rightarrow \text{T} \ast \text{E} \\
\text{E} | & \quad \text{reduce } E \rightarrow \text{T} \ast \text{E}
\end{align*}
\]

The Example with Shift-Reduce Parsing

\[
\begin{align*}
\text{int} \ast \text{int} \ast \text{int} & \quad \text{shift} \\
\text{int} \ast \text{int} \ast \text{int} & \quad \text{shift} \\
\text{int} \ast \text{int} \ast \text{int} & \quad \text{shift} \\
\text{int} \ast \text{int} \ast \text{int} & \quad \text{reduce } T \rightarrow \text{int} \\
\text{int} \ast \text{T} \ast \text{int} & \quad \text{reduce } T \rightarrow \text{int} \ast T \\
\text{T} \ast \text{int} | & \quad \text{shift} \\
\text{T} \ast \text{int} | & \quad \text{shift} \\
\text{T} \ast \text{T} | & \quad \text{reduce } E \rightarrow \text{T} \\
\text{T} \ast \text{E} | & \quad \text{reduce } E \rightarrow \text{T} \ast \text{E} \\
\text{E} | & \quad \text{reduce } E \rightarrow \text{T} \ast \text{E}
\end{align*}
\]

A Shift-Reduce Parse in Detail (1)

\[
\begin{align*}
\text{int} \ast \text{int} \ast \text{int}
\end{align*}
\]
A Shift-Reduce Parse in Detail (8)

T

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

A Shift-Reduce Parse in Detail (9)

T

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

A Shift-Reduce Parse in Detail (10)

T

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

A Shift-Reduce Parse in Detail (11)

E

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

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)

Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse
  - 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
  - You will see such conflicts in your project!
  - More next time...