CS 415: Lecture 5

- Syntax analysis, part 2, top-down parsing

Derivations

- Given a grammar, valid sentences can be derived by repeated substitution
  - `goal`
  - `expr`
  - `expr <op> expr`
  - `expr <op> expr <op> expr`
  - `<id,> <op> expr <op> expr`
  - `<id,> + expr <op> expr`
  - `<id,> + <num,2> <op> expr`
  - `<id,> + <num,2> * expr`
  - `<id,> + <num,2> * id`

- We have derived sentence `x + 2 * y`
  - `goal => id + num * id`
Notation & Terminology

- **Derivation**
  - \( \alpha \Rightarrow^* \beta \)
    - \( \beta \) derives from \( \alpha \) in zero or more steps
  - \( \alpha \Rightarrow \beta \)
    - \( \beta \) derives from \( \alpha \) in one or more steps
  - \( \alpha \Rightarrow^* \) \( \alpha \) for any string \( \alpha \)
  - If \( \alpha \Rightarrow^* \beta \) and \( \beta \Rightarrow^* \gamma \), then \( \alpha \Rightarrow^* \gamma \)
- **If \( G \) is a grammar, then we say that \( L(G) \) is the language generated by \( G \)**
  - A string \( w \) is in \( G \) if and only if \( S \Rightarrow^* w \)
  - If \( S \Rightarrow^* \alpha \), where \( \alpha \) may contain nonterminals, then we say that \( \alpha \) is a sentential form of \( G \)

Leftmost and Rightmost Derivation

- **Previous example was a leftmost derivation**
  - Why is it called leftmost?
- **We can also do a rightmost derivation**
  - \(<\text{goal}>\)
  - \(<\text{expr}>\)
  - \(<\text{expr}> <\text{op}> <\text{expr}>\)
  - \(<\text{expr}> <\text{op}> <\text{id},y>\)
  - \(<\text{expr}> <\text{id},y>\)
  - \(<\text{expr}> <\text{op}> <\text{expr}> <\text{id},y>\)
  - \(<\text{expr}> <\text{op}> <\text{num},2> <\text{id},y>\)
  - \(<\text{expr}> <\text{op}> <\text{num},2> <\text{id},y>\)
  - \(<\text{id},x> <\text{op}> <\text{num},2> <\text{id},y>\)
Precedence

- Adding precedence
  - `<goal>::=<expr>`
  - `<expr>::=<expr>+<term>|<expr>-<term>|<term>`
  - `<term>::=<term>*<factor>|<term>/<factor>|<factor>`
  - `<factor>::=number|id`
- This grammar enforces a precedence on the derivation
  - terms must be derived from expressions
  - forces the "correct" tree

Treewalk evaluation would give "wrong" answer!

Our simple grammar has no notion of precedence (or implied order of evaluation)

Can we fix this problem?
Derivation Using New Grammar

What does the derivation of \( x + 2 \times y \) look like now?

Parse Tree

What does the parse tree look like now?
Ambiguity

- If a grammar produces more than one parse tree for some sentence, it is said to be *ambiguous*.
  - We care more about leftmost and rightmost derivations.
  - If a grammar has multiple leftmost derivations for a single sentential form, then the grammar is ambiguous.
  - Equivalent definition for rightmost derivation.

Example
- \[ \text{stmt} ::= \text{if expr then stmt} \]
  \[ | \text{if expr then stmt} \text{ else stmt} \text{ else stmt} \]
  \[ | ... /\* other statements */ \]
- Let's derive the sentential form:
  - if E1 then if E2 then S1 else S2
- What's the problem?

Eliminating Ambiguities

- We'd like to remove ambiguities.
- Can we do this by rearranging the grammar?
- Let's try for the previous example ...
New Grammar

- \( \text{stmt} ::= \text{ms} \mid \text{us} \)
- \( \text{ms} ::= \text{if} \ \text{expr} \ \text{then} \ \text{ms} \ \text{else} \ \text{ms} \mid \ldots \ \text{other} \ \text{stmts} \)
- \( \text{us} ::= \text{if} \ \text{expr} \ \text{then} \ \text{stmt} \)
  - \( \text{if} \ \text{expr} \ \text{then} \ \text{ms} \ \text{else} \ \text{us} \)

This grammar generates the same language as the previous ambiguous grammar but applies the common sense rule:
- Match each else with the closest unmatched then

Can we describe all context-free languages with unambiguous grammars?
- No ... but, in practice, we can design the language carefully so that there's an unambiguous grammar to generate it

Ambiguity

- When we say a grammar is ambiguous, we are referring to a confusion in the context-free specification
- Context-sensitive confusion can arise from overloading
  - Example: \( a = f(17) \)
  - In many Algol-like languages, \( f \) could be either a function or a subscripted variable
  - Disambiguating this statement requires context
    - Need values of declarations
    - Not context-free
  - Rather than complicate parsing, we will handle this separately (semantic analysis)
Parse Tree

- “Graphical” representation for a derivation; filters out the choice regarding non-terminal replacement order
- Fringe or frontier of parse tree: labels of leaf nodes of (partial) parse tree read from left to right

Two Approaches to Parsing

- Top-down parsers
  - Start at the root of the derivation tree and fill in
  - Picks a production and tries to match the input
  - May require backtracking
  - Can parse some grammars without backtracking (predictive parsers)
- Bottom-up parsers
  - Start at the leaves and fill in
  - Start in a state valid for legal first tokens
  - As input is consumed, change state to encode possibilities (recognize valid prefixes)
- We’ll start with top-down parsers
Top-down Parsing

- A (left-most) top-down parser starts with the root of the parse tree. It is labeled with the start symbol of the grammar.
- To build a parse tree, it repeats the following steps until the fringe of the parse tree matches the input string:
  - At leftmost node labeled (non-terminal) A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate subtree.
  - When a terminal is added to the fringe that doesn't match the input string, backtrack.
  - Find the next node to be expanded.
- The key, of course, is selecting the right production in the first step.
  - Guided by input string

Simple Expression Grammar

- Recall our grammar for simple expressions:
  1. <goal> ::= <expr>
  2. <expr> ::= <expr> + <term>
  3. | <expr> - <term>
  4. | <term>
  5. <term> ::= <term> * <factor>
  6. | <term> / <factor>
  7. | <factor>
  8. <factor> ::= number
  9. | id
- Let's try to parse the input string "x - 2 * y"
### Example

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>Sentential form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>&lt;goal&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>1</td>
<td>&lt;expr&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; + &lt;term&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>4</td>
<td>&lt;term&gt; + &lt;term&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>7</td>
<td>&lt;factor&gt; + &lt;term&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; + &lt;term&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>-</td>
<td>&lt;id&gt; + &lt;term&gt;</td>
<td>(x \uparrow 2 \ast y)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>Sentential form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>&lt;expr&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>3</td>
<td>&lt;expr&gt; - &lt;term&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>4</td>
<td>&lt;term&gt; - &lt;term&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>7</td>
<td>&lt;factor&gt; - &lt;term&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;term&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
<tr>
<td>-</td>
<td>&lt;id&gt; - &lt;term&gt;</td>
<td>(x \uparrow 2 \ast y)</td>
</tr>
</tbody>
</table>

### Example (Cont’d)

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>Sentential form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>&lt;id&gt; - &lt;factor&gt;</td>
<td>(x \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;num&gt;</td>
<td>(x \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>-</td>
<td>&lt;id&gt; - &lt;num&gt;</td>
<td>(x - 2 \uparrow y)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>Sentential form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>&lt;id&gt; - &lt;term&gt; * &lt;factor&gt;</td>
<td>(x \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id&gt; - &lt;factor&gt; * &lt;factor&gt;</td>
<td>(x \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;num&gt; * &lt;factor&gt;</td>
<td>(x - 2 \uparrow y)</td>
</tr>
<tr>
<td>-</td>
<td>&lt;id&gt; - &lt;num&gt; * &lt;id&gt;</td>
<td>(x - 2 \uparrow y)</td>
</tr>
<tr>
<td>-</td>
<td>&lt;id&gt; - &lt;num&gt; * &lt;id&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
</tbody>
</table>
Example (Cont'd)

Another possible parse for $x - 2 \ast y$

<table>
<thead>
<tr>
<th>Prod'n</th>
<th>Sentential form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>&lt;goal&gt;</td>
<td>$x - 2 \ast y$</td>
</tr>
<tr>
<td>1</td>
<td>&lt;expr&gt;</td>
<td>$x - 2 \ast y$</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; + &lt;term&gt;</td>
<td>$x - 2 \ast y$</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; + &lt;term&gt; + &lt;term&gt;</td>
<td>$x - 2 \ast y$</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; + &lt;term&gt; + ...</td>
<td>$x - 2 \ast y$</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; + &lt;term&gt; + ...</td>
<td>$x - 2 \ast y$</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td>$x - 2 \ast y$</td>
</tr>
</tbody>
</table>

Left Recursion

- If the parser makes the wrong choice, the expansion will not terminate
  - Uh oh!!
- Top-down parsers cannot handle left-recursion in a grammar
  - A grammar is left recursive if $\exists A \in NT$ such that $\exists a$ derivation $A \Rightarrow A\alpha$ for some string $\alpha$
- What does the previous derivation example say about our expression grammar?
Eliminating Immediate Left Recursion

- To remove immediate left recursion, we can transform the grammar
- Consider the grammar fragment where $\alpha$ and $\beta$ do not start with `<foo>`
  - `<foo> ::= <foo> $\alpha$ | $\beta$
- This is obviously left recursive, right? How can we rewrite it to get rid of the left recursion?

---

Eliminating Immediate Left Recursion

- Rewrite
  - `<foo> ::= <foo> $\alpha$ | $\beta$
- as
  - `<foo> ::= $\beta$ <bar>
  - `<bar> ::= $\alpha$ <bar> | $\epsilon$

- What about the following grammar fragments?
  - `<foo> ::= <foo> $\alpha$ | $\beta$ | $\gamma$
  - `<foo> ::= <foo> $\alpha$ | <foo> $\beta$ | $\gamma$
Eliminating Left Recursion - Example

Our expression grammar contains two cases of left recursion

\[
\begin{align*}
<\text{expr}> & ::= <\text{expr}> + <\text{term}> \\
& \quad | <\text{expr}> - <\text{term}> \\
& \quad | <\text{term}> \\
<\text{term}> & ::= <\text{term}> \ast <\text{factor}> \\
& \quad | <\text{term}> / <\text{factor}> \\
& \quad | <\text{factor}>
\end{align*}
\]

Eliminating Left Recursion - Example

Applying the transformation gives

\[
\begin{align*}
<\text{expr}> & ::= <\text{term}> <\text{expr'}> \\
<\text{expr'}> & ::= + <\text{term}> <\text{expr'}> \\
& \quad | - <\text{term}> <\text{expr'}> \\
& \quad | \epsilon \\
<\text{term}> & ::= <\text{factor}> <\text{term'}> \\
<\text{term'}> & ::= \ast <\text{factor}> <\text{term'}> \\
& \quad | / <\text{factor}> <\text{term'}> \\
& \quad | \epsilon
\end{align*}
\]
Eliminating Left Recursion

Why can’t we use the following grammar instead?

1 \<goal> ::= \<expr>
2 \<expr> ::= \<term> + \<expr>
3 \<term> ::= \<term> - \<expr>
4 \<term> ::= \<term>
5 \<term> ::= \<factor> * \<term>
6 \<term> ::= \<factor> / \<term>
7 \<term> ::= \<factor>
8 \<factor> ::= num
9 \<factor> ::= id

Algorithm for Eliminating Left Recursion

Arrange the non-terminals in some order \(A_1, A_2, \ldots, A_n\)
for \(i \leftarrow 1\) to \(n\)
for \(j \leftarrow 1\) to \(i - 1\)
   replace each production of the form
   \(A_i ::= A_j \gamma\) with the productions
   \(A_i ::= \delta_1 \gamma | \delta_2 \gamma | \ldots | \delta_k \gamma\)
   where \(A_j ::= \delta_1 | \delta_2 | \ldots | \delta_k\) are all current \(A_j\) productions
endfor
eliminate any immediate left recursion on \(A_i\) using the direct transformation
endfor

This works when the grammar has no cycles \((A \Rightarrow A)\) or \(\varepsilon\) productions
Eliminating Left Recursion

- How does the previous algorithm work?
  - Impose arbitrary order on the non-terminals
  - Outer loop cycles through NT in order
  - Inner loop ensures that a production expanding $A_i$ has no non-terminal $A_j$ with $j < i$ as the first symbol on its right hand side (it forward substitutes those away)
  - Last step in the outer loop converts any direct recursion on $A_i$ to right recursion using the simple transformation shown earlier
  - New non-terminals are added at the end of the order and only involve right recursion

How Much Lookahead Is Needed?

- Top-down parsers may need to backtrack when they select the wrong production
- Do we need arbitrary lookahead to parse CFGs?
  - In general, yes
  - There are algorithms for this (e.g., Earley)
- Fortunately
  - Large subclasses of CFGs can be parsed with limited lookahead
  - Most programming language constructs can be expressed in a grammar that falls into these subclasses
  - Among interesting subclasses are LL(1) and LR(1)