CS415 Compilers
Syntax Analysis
Part 3

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

- Third homework due on Sunday, March 7.
- First project deadline extension: Thursday (03/04) for code, and Saturday (03/06) for report.
Parsing
(Syntax Analysis)

Top-Down Parsing
EAC Chapters 3.3
**LL(1), recursive descent**

- 1 input symbol lookahead
- construct leftmost derivation (forwards)
- input: read left-to-right

\[ S \Rightarrow^{*_{lm}} x A \beta \Rightarrow_{lm} x \delta \beta \Rightarrow^{*_{lm}} x y \]
**LL(1), recursive descent**

- 1 input symbol lookahead
- Construct leftmost derivation (forwards)
- Input: read left-to-right

**Rule** $A \rightarrow \delta$

$$S \Rightarrow_{lm}^{*} \text{ x } A \beta \Rightarrow_{lm} \text{ x } \delta \beta \Rightarrow_{lm}^{*} \text{ x } y$$

Diagram:

```
  S
 /|
/ A
/  \
A  \beta
  \delta
```

Input: read left-to-right
Remember the expression grammar?

Version with precedence

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal → Expr</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Expr → Expr + Term</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Expr – Term</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Term</td>
</tr>
<tr>
<td>5</td>
<td>Term → Term * Factor</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Term / Factor</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Factor</td>
</tr>
<tr>
<td>8</td>
<td>Factor → number</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>id</td>
</tr>
</tbody>
</table>

And the input $x - 2 * y$
Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is left recursive if $\exists A \in NT$ such that

$\exists$ a derivation $A \Rightarrow^+ A\alpha$, for some string $\alpha \in (NT \cup T)^+$

Our expression grammar is left recursive

• This can lead to non-termination in a top-down parser
• For a top-down parser, any recursion must be right recursion
• We would like to convert the left recursion to right recursion

Non-termination is a bad property in any part of a compiler
To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

\[ Fee \rightarrow Fee \alpha \]
\[ \quad \mid \beta \]

where neither \( \alpha \) nor \( \beta \) start with \( Fee \)

We can rewrite this as

\[ Fee \rightarrow \beta \text{ Fie} \]
\[ \text{Fie} \rightarrow \alpha \text{ Fie} \]
\[ \quad \mid \varepsilon \]

where \( \text{Fie} \) is a new non-terminal

This accepts the same language, but uses only right recursion
The expression grammar contains two cases of left recursion

```
Expr  →  Expr  + Term  Term  →  Term  *  Factor
      |  Expr  – Term          |  Term  /  Factor
      |  Term
```

Applying the transformation yields

```
Expr  →  Term  Expr’  Term  →  Factor  Term’
Expr’  |  +  Term  Expr’  Term’  |  *  Factor  Term’
      |  –  Term  Expr’  |  /  Factor  Term’
      |  ε
```

These fragments use only right recursion
Substituting them back into the grammar yields

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Goal</strong> → <strong>Expr</strong></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><strong>Expr</strong> → <strong>Term Expr’</strong></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><strong>Expr’</strong> → <strong>+ Term Expr’</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>– Term Expr’</strong></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><strong>ε</strong></td>
</tr>
<tr>
<td>5</td>
<td><strong>Term</strong> → <strong>Factor Term’</strong></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><strong>Term’</strong> → *** Factor**</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Term’</strong></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td><strong>/ Factor</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Term’</strong></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td><strong>ε</strong></td>
</tr>
<tr>
<td>9</td>
<td><strong>Factor</strong> → <strong>number</strong></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td><strong>id</strong></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>( <strong>Expr</strong> )</td>
</tr>
</tbody>
</table>

- This grammar is correct, if somewhat non-intuitive.
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.
- General left recursion removal algorithm in EAC
We set out to study parsing

- Specifying syntax
  - Context-free grammars
  - Ambiguity

- Top-down parsers
  - Algorithm & its problem with left recursion
  - Left-recursion removal
  - Left factoring (will discuss later)

- Predictive top-down parsing
  - The LL(1) condition
  - Table-driven LL(1) parsers
  - Recursive descent parsers
    - Syntax directed translation (example)
If it picks the wrong production, a top-down parser may backtrack. Alternative is to look ahead in input & use context to pick correctly.

How much lookahead is needed?
- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami algorithm or Earley’s algorithm

Fortunately,
- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are $LL(1)$ and $LR(1)$ grammars.
Basic idea

*Given* $A \rightarrow \alpha | \beta$, the parser should be able to choose between $\alpha$ & $\beta$.

**FIRST sets**

For some rhs $\alpha \in G$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear as the first (terminal) symbol in some string that derives from $\alpha$.

That is, $\alpha \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* a \gamma$, for some $\gamma$. 
The FIRST Set - 1 symbol lookahead

\[ a \in \text{FIRST}_1(\alpha) \iff \alpha \Rightarrow^* a \gamma, \text{ for some } \gamma \]

To build FIRST(X) for all grammar symbols X:
1. if X is a terminal (token),FIRST(X) := \{ X \}
2. if X \rightarrow \varepsilon, then \varepsilon \in \text{FIRST}(X)
3. iterate until no more terminals or \varepsilon can be added to any FIRST(X):
   if X \rightarrow Y_1 Y_2 \ldots Y_k then
   a \in \text{FIRST}(X) if a \in \text{FIRST}(Y_i) and
   \varepsilon \in \text{FIRST}(Y_j) for all 1 \leq j < i
   \varepsilon \in \text{FIRST}(X) if \varepsilon \in \text{FIRST}(Y_i) for all 1 \leq i \leq k
   end iterate

Note: if \varepsilon \notin \text{FIRST}(Y_1), then \text{FIRST}(Y_i) is irrelevant, for 1 < i
Syntax Analysis

(top-down)
Read EaC: Chapter 3.3

(bottom-up)
Read EaC: Chapter 3.4