CS415 Compilers
Syntax Analysis
Top-Down Parsing

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University.
Top-Down Parsing
(Syntax Analysis)

EAC Chapters 3.1 - 3.3
LL(1), recursive descent

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

$S \Rightarrow^{*_{lm}} x A \beta \Rightarrow^{l_m}_{lm} x \delta \beta \Rightarrow^{*_{lm}}_{lm} x y$

Diagram:

S

A

\(\beta\)

\(\delta\)

X

Y
**LL(1), recursive descent**

1 input symbol lookahead

construct leftmost derivation (forwards)

input: read left-to-right

rule $A \rightarrow \delta$

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

Diagram:

```
  S
  |
  v
 A
  |
  v
 β
  |
  v
 δ
```

Input: read left-to-right
A top-down parser starts with the root of the parse tree. The root node is labeled with the goal symbol of the grammar.

**Top-down parsing algorithm:**

1. Construct the root node of the parse tree.
2. Repeat until the fringe of the parse tree matches the input string.
   - **At a node labeled A**, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child.
   - **When a terminal symbol is added to the fringe and it doesn’t match the fringe**, backtrack.
   - **Find the next node to be expanded** (label ∈ NT)

- **The key is picking the right production in step 1**
  - That choice should be guided by the input string.
Example

Trying again with “2” in \( x - 2 * y \):

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(&lt;id,x&gt; - Term * Factor)</td>
<td>(x - 2 * y)</td>
</tr>
<tr>
<td>7</td>
<td>(&lt;id,x&gt; - Factor * Factor)</td>
<td>(x - 2 * y)</td>
</tr>
<tr>
<td>8</td>
<td>(&lt;id,x&gt; - &lt;num,2&gt; * Factor)</td>
<td>(x - 2 * y)</td>
</tr>
<tr>
<td>9</td>
<td>(&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;)</td>
<td>(x - 2 * y)</td>
</tr>
<tr>
<td></td>
<td>(&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;)</td>
<td>(x - 2 * y)</td>
</tr>
</tbody>
</table>

GUESS : SUCCESS
Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

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

\[ \exists \text{ a derivation } A \Rightarrow^+ A\alpha, \text{ for some string } \alpha \in (NT \cup T)^+ \]

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

\[
\text{Fee} \rightarrow \text{Fee} \; \alpha \\
| \quad \beta
\]

where neither \( \alpha \) nor \( \beta \) start with \( \text{Fee} \)

We can rewrite this as

\[
\text{Fee} \rightarrow \beta \; \text{Fie} \\
\text{Fie} \rightarrow \alpha \; \text{Fie} \\
| \quad \epsilon
\]

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

\[
\text{Expr} \quad \rightarrow \quad \text{Expr} \ + \ \text{Term} \quad \quad \quad \text{Term} \quad \rightarrow \quad \text{Term} \ * \ \text{Factor} \\
\quad \quad \quad \quad \quad \text{|} \quad \text{Expr} \ - \ \text{Term} \quad \quad \quad \quad \quad \text{|} \quad \text{Term} \ / \ \text{Factor} \\
\quad \quad \quad \quad \quad \text{|} \quad \text{Term} \quad \quad \quad \quad \quad \text{|} \quad \text{Factor}
\]

Applying the transformation yields

\[
\text{Expr} \quad \rightarrow \quad \text{Term} \ \text{Expr}' \quad \quad \quad \text{Term} \quad \rightarrow \quad \text{Factor} \ \text{Term}' \\
\quad \quad \quad \quad \quad \text{Expr}' \quad \rightarrow \quad + \ \text{Term} \ \text{Expr}' \quad \quad \quad \text{Term}' \quad \rightarrow \quad * \ \text{Factor} \ \text{Term}' \\
\quad \quad \quad \quad \quad \text{|} \quad - \ \text{Term} \ \text{Expr}' \quad \quad \quad \quad \quad \text{|} \quad / \ \text{Factor} \ \text{Term}' \\
\quad \quad \quad \quad \quad \text{|} \quad \varepsilon \quad \quad \quad \quad \quad \text{|} \quad \varepsilon
\]

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

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.
- General left recursion removal algorithm fig 3.6 EAC

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Goal</strong> $\rightarrow$ <strong>Expr</strong></td>
</tr>
<tr>
<td>2</td>
<td><strong>Expr</strong> $\rightarrow$ <strong>Term</strong> <strong>Expr</strong>$'$</td>
</tr>
<tr>
<td>3</td>
<td><strong>Expr</strong>$'$ $\rightarrow$ + <strong>Term</strong> <strong>Expr</strong>$'$</td>
</tr>
<tr>
<td>4</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>$</td>
</tr>
<tr>
<td>6</td>
<td><strong>Term</strong> $\rightarrow$ <strong>Factor</strong> <strong>Term</strong>$'$</td>
</tr>
<tr>
<td>7</td>
<td><strong>Term</strong>$'$ $\rightarrow$ * <strong>Factor</strong> <strong>Term</strong>$'$</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
</tr>
<tr>
<td>9</td>
<td>$</td>
</tr>
<tr>
<td>10</td>
<td><strong>Factor</strong> $\rightarrow$ <strong>number</strong></td>
</tr>
<tr>
<td>11</td>
<td>$</td>
</tr>
</tbody>
</table>
|12 |   $|$ ($$**Expr$$)$
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, $a \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* a \gamma$, for some $\gamma$
a \in \text{FIRST}(\alpha) \text{ iff } \alpha \Rightarrow^* a\gamma, \text{ for some } \gamma

To build \text{FIRST}(\alpha) for \alpha = X_1 X_2 \ldots X_n:

1. \( x \in \text{FIRST}(\alpha) \) if \( x \in \text{FIRST}(X_i) \) and 
   \[ \varepsilon \in \text{FIRST}(X_j) \text{ for all } 1 \leq j < i \]

2. \( \varepsilon \in \text{FIRST}(\alpha) \) if \( \varepsilon \in \text{FIRST}(X_i) \) for all \( 1 \leq i \leq n \)
The FIRST Set

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

To build \text{FIRST}(X) for all grammar symbols X:

1. if X is a terminal (token), \text{FIRST}(X) := \{ X \}
2. if X ::= ε, then ε ∈ \text{FIRST}(X)

3. iterate until no more terminals or ε can be added to any \text{FIRST}(X):
   
   if \ X ::= Y_1 Y_2 \ldots Y_k \ then
   a ∈ \text{FIRST}(X) if \ a ∈ \text{FIRST}(Y_i) \ and
   ε ∈ \text{FIRST}(Y_j) \text{ for all } 1 \leq j < i
   ε ∈ \text{FIRST}(X) if ε ∈ \text{FIRST}(Y_i) \text{ for all } 1 \leq i \leq k

end iterate

Note: if ε ∉ \text{FIRST}(Y_1), then \text{FIRST}(Y_i) is irrelevant, for 1 < i
Basic idea

*Given* $A \rightarrow \alpha \mid \beta$, 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 symbol in some string that derives from $\alpha$

That is, $a \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* a \gamma$, for some $\gamma$

**The LL(1) Property**

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

This is almost correct, but not quite
For a non-terminal $A$, define $\text{FOLLOW}(A)$ as

$$\text{FOLLOW}(A) := \text{the set of terminals that can appear immediately to the right of } A \text{ in some sentential form.}$$

Thus, a non-terminal’s $\text{FOLLOW}$ set specifies the tokens that can legally appear after it; a terminal has no $\text{FOLLOW}$ set.
The FOLLOW Set

To build FOLLOW(X) for all non-terminal X:

1. Place eof in FOLLOW(<goal>)
   iterate until no more terminals or ε can be added to any FOLLOW(X):
2. If $A \rightarrow \alpha B \beta$ then
   put \{FIRST(\beta) - ε\} in FOLLOW(B)
3. If $A \rightarrow \alpha B$ then
   put FOLLOW(A) in FOLLOW(B)
4. If $A \rightarrow \alpha B \beta$ and ε \in FIRST(\beta) then
   put FOLLOW(A) in FOLLOW(B)
More Syntax Analysis

Read EaC: Chapter 3.1-3.3