CS415 Compilers
Syntax Analysis

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

• Homework #3 is due today!

• Homework #4 (last before midterm) has been posted. Deadline: Sunday, March 4.

• Upcoming deadlines:
  • Project 1 code: March 2
  • Project 1 report: March 4
  • Homework #4: March 4

• Midterm: Wednesday, March 7, class time slot, in Tillett hall (two rooms); closed book, closed notes, 80 minutes
Parsing
(Syntax Analysis)

Top-Down Parsing
EAC Chapters 3.3
Parsing Techniques: Top-down parsers

**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 $A \to \delta$

$$S \Rightarrow^{*_{lm}} x A \beta \Rightarrow_{lm} x \delta \beta \Rightarrow^{*_{lm}} x y$$
Remember the expression grammar?

Version with precedence

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal → Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr → Expr + Term</td>
</tr>
<tr>
<td>3</td>
<td>Expr - Term</td>
</tr>
<tr>
<td>4</td>
<td>Term</td>
</tr>
<tr>
<td>5</td>
<td>Term → Term * Factor</td>
</tr>
<tr>
<td>6</td>
<td>Term / Factor</td>
</tr>
<tr>
<td>7</td>
<td>Factor</td>
</tr>
<tr>
<td>8</td>
<td>Factor → number</td>
</tr>
<tr>
<td>9</td>
<td>id</td>
</tr>
</tbody>
</table>

And the input $x - 2 * y$
Left Recursion

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

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

where \( Fie \) is a new non-terminal

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

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Expr} + \text{Term} & \text{Term} & \rightarrow \text{Term} * \text{Factor} \\
& \mid \text{Expr} - \text{Term} & & \mid \text{Term} / \text{Factor} \\
& \mid \text{Term} & & \mid \text{Factor}
\end{align*}
\]

Applying the transformation yields

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Term} \text{Expr'} \\
(\text{Expr'}) & \rightarrow + \text{Term} \text{Expr'} \\
& \mid - \text{Term} \text{Expr'} \\
& \mid \varepsilon
\end{align*}
\]

\[
\begin{align*}
\text{Term} & \rightarrow \text{Factor} \text{Term'} \\
(\text{Term'}) & \rightarrow * \text{Factor} \text{Term'} \\
& \mid / \text{Factor} \text{Term'} \\
& \mid \varepsilon
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>1</th>
<th>Goal</th>
<th>( \rightarrow ) Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Expr</td>
<td>( \rightarrow ) Term Expr'</td>
</tr>
<tr>
<td>3</td>
<td>Expr'</td>
<td>( \rightarrow ) + Term Expr'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \rightarrow ) - Term Expr'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \rightarrow ) ( \varepsilon )</td>
</tr>
<tr>
<td>4</td>
<td>Term</td>
<td>( \rightarrow ) Factor Term'</td>
</tr>
<tr>
<td>5</td>
<td>Term'</td>
<td>( \rightarrow ) * Factor Term'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \rightarrow ) / Factor Term'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \rightarrow ) ( \varepsilon )</td>
</tr>
<tr>
<td>6</td>
<td>Factor</td>
<td>( \rightarrow ) number</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>( \rightarrow ) id</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>( \rightarrow ) ( Expr )</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 p.103 EAC
Roadmap (Where are we?)

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)
Picking the “Right” Production

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* $A \rightarrow \alpha \mid \beta$, *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 (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 \text{FIRST}(X) for all grammar symbols X:

1. if X is a terminal (token), \text{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 \text{FIRST}(X):
   
   \[
   \begin{align*}
   \text{if } X \rightarrow Y_1 Y_2 \ldots Y_k \text{ then} \\
   &a \in \text{FIRST}(X) \text{ if } a \in \text{FIRST}(Y_i) \text{ and} \\
   &\varepsilon \in \text{FIRST}(Y_j) \text{ for all } 1 \leq j < i \\
   &\varepsilon \in \text{FIRST}(X) \text{ if } \varepsilon \in \text{FIRST}(Y_i) \text{ for all } 1 \leq i \leq k
   \end{align*}
   \]

end iterate

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

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

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

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

2. \(\epsilon \in \text{FIRST}(\alpha)\) if \(\epsilon \in \text{FIRST}(X_i)\) for all \(1 \leq i \leq n\)
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, $\alpha \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* \gamma$, 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
More Syntax Analysis (top-down and bottom-up)

Read EaC: Chapter 3.3 and 3.4