Plan for Today

Ambiguous Grammars

Disambiguating ambiguous grammars

Left-factoring for predictive parsers
Ambiguous Grammars

Ambiguous grammar:
>2+ parse trees for 1 sentence

Expression grammar

<table>
<thead>
<tr>
<th>Rule</th>
<th>Parse Tree 1</th>
<th>Parse Tree 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E * E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E → E + E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E → E - E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E → ( E )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E → ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E → NUM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

String

42 + 7 * 6

what about 42-7-6?
Goal: disambiguate the grammar

Cause
- the grammar did not specify the precedence nor the associativity of the operators +, -, *

Two Options
- keep the ambiguous grammar, but add extra directives to the parser (many LALR parser generators can do this)
- Rewrite the grammar, making the precedence and associativity explicit in the grammar.
Unambiguous grammar for simple expressions

Grammar

\( E \rightarrow E + T \mid E - T \mid T \)

\( T \rightarrow T \times F \mid F \)

\( F \rightarrow ( E ) \mid ID \mid NUM \)

String

42+7*6

How is the precedence encoded?

How is the associativity encoded?
When Predictive Parsing works, when it does not

What about our expression grammar:

\[ E \rightarrow E + T \mid E - T \mid T \]
\[ T \rightarrow T \ast F \mid F \]
\[ F \rightarrow (E) \mid \text{ID} \mid \text{NUM} \]

Predictive parser

- The E method cannot decide looking one token ahead whether to predict E+T, E-T, or T.
- Same problem for T.

Predictive parsing works for grammars where

- The first terminal symbol of each sub expression
- provides enough information to decide which production to use.
Left recursion and Predictive parsing

What happens to the recursive descent parser if we have a left recursive production rule, e.g.  \[ E \rightarrow E + T | T \]

E calls E calls E forever

To eliminate left recursion we rewrite the grammar:

from:

\[
\begin{align*}
E & \rightarrow E + T | E - T | T \\
T & \rightarrow T \ast F | F \\
F & \rightarrow (E) | ID | NUM
\end{align*}
\]

to:

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T E' | - T E' | \epsilon \\
T & \rightarrow F T' \\
T' & \rightarrow \ast F T' | \epsilon \\
F & \rightarrow (E) | ID | NUM
\end{align*}
\]

replacing left recursion \( X \rightarrow X \gamma | \alpha \) (where \( \alpha \) does not start with \( X \)) with right recursion, \( X \rightarrow \alpha X' \), \( X' \rightarrow \gamma X' | \epsilon \), that can be produced right recursively. Now we can augment the grammar \( (S \rightarrow E\$) \), compute nullable, FIRST and FOLLOW, and produce an LL(1) predictive parse table.
Left Factoring

Left recursion does not work for predictive parsing. Neither does a grammar that has a non-terminal with two productions that start with a common phrase, so we left factor the grammar:

\[
S \rightarrow \alpha \beta_1 \\
S \rightarrow \alpha \beta_2
\]

Left refactor:

\[
S \rightarrow \alpha S' \\
S' \rightarrow \beta_1 | \beta_2
\]

E.g.: if statement:

\[
S \rightarrow \text{IF } t \text{ THEN } S \text{ ELSE } S | \text{IF } t \text{ THEN } S | o
\]

becomes

\[
S \rightarrow \text{IF } t \text{ THEN } S \ X | o \\
X \rightarrow \text{ELSE } S | \varepsilon
\]

When building the predictive parse table, there will be a multiple entries. **WHY?**
Dangling else problem: ambiguity

Given

\[
S \rightarrow \text{IF } t \text{ THEN } S \ X \mid o \\
X \rightarrow \text{ELSE } S \mid \varepsilon
\]

construct two parse trees for

\[
\text{IF } t \text{ THEN IF } t \text{ THEN } o \text{ ELSE } o
\]

Which is the correct parse tree? (C, Java rules)
Dangling else disambiguation

The correct parse tree is:

```
S
  /|
/   \
IF t THEN S X ε
  \
  \. ELSE S
```

We can get this parse tree by removing the \( X \rightarrow \varepsilon \) rule in the multiple entry slot in the parse tree.