Using ambiguous grammars

No ambiguous grammar is LR. Nonetheless, ambiguous grammars are often used in LR parsing.

Why? Because ambiguous grammars are often simpler.

For instance, compare

\[
E \rightarrow E + E \mid E * E \mid (E) \mid a
\]

to the equivalent unambiguous grammar

\[
E \rightarrow E + T \mid T \\
T \rightarrow T * F \mid F \\
F \rightarrow (E) \mid a
\]

The second grammar has important virtues though:

1. It is unambiguous.
2. It reflects the fact that operators + and * are left-associative.
3. It reflects the fact that * has higher precedence than +.

\[
E \rightarrow E + E \mid E * E \mid (E) \mid a \\
E \rightarrow E + T \mid T \\
T \rightarrow T * F \mid F \\
F \rightarrow (E) \mid a
\]

In practice, we can obtain the same advantages for the first grammar.

In particular, we can stipulate — as a guide to using the first grammar — that * and + are left-associative and that * has precedence over +.

And if we consider only parse trees that reflect these stipulations, the grammar is rendered unambiguous “in practice.” (Moreover, the language generated by the grammar is unchanged!)

That is, for every sentence generated by the grammar, there is exactly one parse tree that respects the stipulations.

Consider alternative parse trees for \(a + a + a\) and \(a + a * a\), for example.

There are two parse trees for each, only one of which satisfies the stipulations.
\[ E \rightarrow E + E \mid E \ast E \mid (E) \mid a \quad \text{FOLLOW}(E) = \{+, *, ), \}$

\[ I_0: \{E' \rightarrow E, E \rightarrow \cdot E + E, E \rightarrow \cdot E \ast E, E \rightarrow \cdot (E), E \rightarrow \cdot a\} \]

\[ I_1: \{E' \rightarrow E \cdot, E \rightarrow E \cdot + E, E \rightarrow E \cdot E\} \]

\[ I_2: \{E \rightarrow (\cdot E), E \rightarrow \cdot E + E, E \rightarrow \cdot E \ast E, E \rightarrow (\cdot (E), E \rightarrow \cdot a\} \]

\[ I_3: \{E \rightarrow a \cdot \} \]

\[ I_4: \{E \rightarrow E + E \cdot, E \rightarrow E \cdot E + E, E \rightarrow E \ast E \ast E \rightarrow (\cdot (E), E \rightarrow \cdot a\} \]

\[ I_5: \{E \rightarrow E \ast E \cdot, E \rightarrow E \cdot E + E, E \rightarrow E \ast E, E \rightarrow (\cdot (E), E \rightarrow \cdot a\} \]

\[ I_6: \{E \rightarrow E \rightarrow (\cdot E \cdot), E \rightarrow E \cdot + E, E \rightarrow E \cdot E\} \]

\[ I_7: \{E \rightarrow E + E \cdot, E \rightarrow E \cdot + E, E \rightarrow E \cdot E\} \]

\[ I_8: \{E \rightarrow E \ast E \cdot, E \rightarrow E \cdot + E, E \rightarrow E \ast E\} \]

\[ I_9: \{E \rightarrow (E) \cdot \} \]

<table>
<thead>
<tr>
<th>STATE</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td>+</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
<td>s5</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>+</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
<td>+</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
<td>s5</td>
</tr>
<tr>
<td>7</td>
<td>s4,r1</td>
<td>s5,r1</td>
</tr>
<tr>
<td>8</td>
<td>s4,r2</td>
<td>s5,r2</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
<td>r3</td>
</tr>
</tbody>
</table>

We'll obtain essentially the same conflicts in an LR or LALR table for this grammar — the conflicts arise because of the ambiguity of the grammar.

The conflicts can be resolved by associativity and precedence.

For example, in state 7 it must be the case that we have \( E + E \) on top of the stack, since one of the state 7 actions is r1.

If the lookahead is + we have a shift/reduce conflict. Here we want to reduce, to reflect the fact that + is left-associative.

If the lookahead is * we again have a shift/reduce conflict. In this case we should shift, since * has higher precedence than +.
$E \rightarrow E + E \mid E \ast E \mid (E) \mid a$

<table>
<thead>
<tr>
<th>STATE</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
<td>s5</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
<td>s5</td>
</tr>
<tr>
<td>7</td>
<td>s4, r1</td>
<td>s5, r1</td>
</tr>
<tr>
<td>8</td>
<td>s4, r2</td>
<td>s5, r2</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
<td>r3</td>
</tr>
</tbody>
</table>

Similar observations apply to the shift/reduce conflicts in state 8.

Among the state 8 actions is r2, so we must have $E \ast E$ on top of the stack:

If the lookahead is $+$ we have a shift/reduce conflict. Here we want to reduce, to reflect the precedence of $\ast$ over $+$. 

If the lookahead is $\ast$ we again have a shift/reduce conflict. Here we want to reduce also, this time to reflect the fact that $\ast$ is left-associative.

So by stipulating that $+$ and $\ast$ are left-associative and that $\ast$ takes precedence over $+$, we can decide how to eliminate the conflicts in the SLR table for

$E \rightarrow E + E \mid E \ast E \mid (E) \mid a$

<table>
<thead>
<tr>
<th>STATE</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
<td>s5</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
<td>s5</td>
</tr>
<tr>
<td>7</td>
<td>r1</td>
<td>s5</td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
<td>r3</td>
</tr>
</tbody>
</table>
“Dangling else” ambiguity

When parsing if-then-else constructs, it is customary to associate each else with the closest preceding unmatched then.

\[
stmt \rightarrow \text{if expr then stmt} \\
| \text{if expr then stmt else stmt} \\
| \text{other}
\]

Consider the two parse trees for

\[
\text{if expr then if expr then other else other}
\]

The preference for matching the closest unmatched then is a bit awkward to express directly in a grammar:

\[
stmt \rightarrow \text{matched-stmt | unmatched-stmt} \\
\text{matched-stmt} \rightarrow \text{if expr then matched-stmt else matched-stmt} \\
| \text{other} \\
\text{unmatched-stmt} \rightarrow \text{if expr then stmt} \\
| \text{if expr then matched-stmt else unmatched-stmt}
\]

This grammar is SLR, I believe, but again we can instead can work directly with the ambiguous grammar…

\[
stmt \rightarrow \text{if expr then stmt} \\
| \text{if expr then stmt else stmt} \\
| \text{other}
\]

First simplify the representation:

\[
S \rightarrow iS \mid iSeS \mid o
\]

An example of the problem with ambiguity in the simplified grammar:

\[
S \Rightarrow iSeS \Rightarrow iSeo \Rightarrow iiSeo \Rightarrow iioeo
\]

\[
S \Rightarrow iS \Rightarrow iiSeS \Rightarrow iiSeo \Rightarrow iioeo
\]

Which of these rightmost derivations do we prefer?
\[ stmt \rightarrow \text{if expr then stmt} \]
\[ \quad | \text{if expr then stmt else stmt} \]
\[ \quad | \text{other} \]
\[ S \rightarrow iS | iSeS | o \]

\( I_0: \{S' \rightarrow \cdot S, \ S \rightarrow \cdot iS, \ S \rightarrow \cdot iSeS, \ S \rightarrow \cdot o\} \)

\( I_1: \{S' \rightarrow S \cdot \} \)

\( I_2: \{S \rightarrow i \cdot S, \ S \rightarrow i \cdot iSeS, \ S \rightarrow iS, \ S \rightarrow iSeS, \ S \rightarrow \cdot o\} \)

\( I_3: \{S \rightarrow o \cdot \} \)

\( I_4: \{S \rightarrow iS \cdot , \ S \rightarrow iS \cdot eS\} \)

\( I_5: \{S \rightarrow iSe \cdot S, \ S \rightarrow iS, \ S \rightarrow iSeS, \ S \rightarrow \cdot o\} \)

\( I_6: \{S \rightarrow iSeS \cdot \} \)

\( \text{FOLLOW}(S) = \{e, \$\} \)

Where is the shift/reduce conflict?

How should we resolve it (to attach each else to the nearest preceding unmatched then)?

We'll be working a bit with Yacc — an LALR parser generator.

Here's an example Yacc program for this grammar:

```yacc
%%
S : 'i' S
   | 'i' S 'e' S
   | 'o';

%%
yylex() {
    int c;
    while ((c=getchar()) == ' '); /* skip blanks */
    if (c == '\n') return 0;    /* convenient for interactive use */
    if (c == ' ') return c;
    return c;
}

yyerror(char *s) {
    printf("%s\n", s);
}

main() {
    if (yyparse() == 0) printf("ok\n");
}

Yacc informs the user that this grammar has 1 shift/reduce conflict.
Here’s a first attempt at a Yacc program for the simple, ambiguous expression grammar we considered earlier:

```
%%
E : E + E | E * E | (E) | a
```

By default Yacc resolves shift/reduce conflicts in favor of shift, so the Yacc-generated parser in this case is the one we want.

```
> yacc ites.y
yacc: 1 shift/reduce conflict.
> cc -o itesy y.tab.c
> itesy
iiioeo
ok
> 
```

As expected, Yacc finds 4 shift/reduce conflicts. But recall that for this grammar we do not always prefer shift over reduce.
In fact, in three of the four shift/reduce conflicts in this grammar we prefer to reduce. Our decision in each case was determined by associativity and precedence of + and \(*\).

Yacc provides a simple mechanism for specifying associativity and precedence of tokens:

```plaintext
%%
E : E '+' E
   | E '*' E
   | '(' E ')' 
   | 'a'

%%
yylex() {
    int c;
    while ((c=getchar()) == ' ');
    if (c == '\n') return 0;
    return c;
}
yyerror(char *s) {
    printf("%s\n", s);
}
main() {
    if (yyparse() == 0) printf("ok\n");
}
```

---

For next time...

Look at Yacc document available via course web page.

Read Section 4.9.