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

$$E \to E + E \mid E * E \mid (E) \mid a$$

to the equivalent unambiguous grammar

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid a \end{aligned}$$

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 +.

$$E \to E + E \mid E * E \mid (E) \mid a$$

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid a \end{aligned}$$

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.

$E \to E + E \mid E * E \mid (E) \mid a \quad \text{FOLLOW}(E) = \{+, *,), \$\}$

I_0: $\{E' \to \cdot E,\ E \to \cdot E + E,\ E \to \cdot E * E,\ E \to \cdot(E),\ E \to \cdot a\}$

I_1: $\{E' \to E \cdot,\ E \to E \cdot + E,\ E \to E \cdot * E\}$

I_2: $\{E \to (\cdot E),\ E \to \cdot E + E,\ E \to \cdot E * E,\ E \to \cdot(E),\ E \to \cdot a\}$

I_3: $\{E \to a \cdot\}$

I_4: $\{E \to E + \cdot E,\ E \to \cdot E + E,\ E \to \cdot E * E,\ E \to \cdot(E),\ E \to \cdot a\}$

I_5: $\{E \to E * \cdot E,\ E \to \cdot E + E,\ E \to \cdot E * E,\ E \to \cdot(E),\ E \to \cdot a\}$

I_6: $\{E \to (E \cdot),\ E \to E \cdot + E,\ E \to E \cdot * E\}$

I_7: $\{E \to E + E \cdot,\ E \to E \cdot + E,\ E \to E \cdot * E\}$

I_8: $\{E \to E * E \cdot,\ E \to E \cdot + E,\ E \to E \cdot * E\}$

I_9: $\{E \to (E) \cdot\}$

	action						goto
STATE	a	$+$	$*$	$($	$)$	\$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		s4,r1	s5,r1		r1	r1	
8		s4,r2	s5,r2		r2	r2	
9		r3	r3		r3	r3	

$E \to E + E \mid E * E \mid (E) \mid a$

	action						goto
STATE	a	$+$	$*$	$($	$)$	\$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		s4,**r1**	**s5**,r1		r1	r1	
8		s4,r2	s5,r2		r2	r2	
9		r3	r3		r3	r3	

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 $+$.

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

	action						goto
STATE	a	$+$	$*$	$($	$)$	\$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		s4,**r1**	**s5**,r1		r1	r1	
8		s4,**r2**	s5,**r2**		r2	r2	
9		r3	r3		r3	r3	

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

Among the state 8 actions is r2, so we must have $E * E$ 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 $*$ over $+$.

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

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

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

	action						goto
STATE	a	$+$	$*$	$($	$)$	\$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

"Dangling else" ambiguity

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

$$\begin{aligned} \textit{stmt} \;\rightarrow\; & \textbf{if expr then } \textit{stmt} \\ \mid\; & \textbf{if expr then } \textit{stmt} \textbf{ else } \textit{stmt} \\ \mid\; & \textbf{other} \end{aligned}$$

Consider the two parse trees for

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:

$$\begin{aligned} \textit{stmt} &\rightarrow \textit{matched-stmt} \mid \textit{unmatched-stmt} \\ \textit{matched-stmt} &\rightarrow \textbf{if expr then } \textit{matched-stmt} \textbf{ else } \textit{matched-stmt} \\ &\mid \textbf{other} \\ \textit{unmatched-stmt} &\rightarrow \textbf{if expr then } \textit{stmt} \\ &\mid \textbf{if expr then } \textit{matched-stmt} \textbf{ else } \textit{unmatched-stmt} \end{aligned}$$

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

$$\begin{aligned} \textit{stmt} \;\rightarrow\; & \textbf{if expr then } \textit{stmt} \\ \mid\; & \textbf{if expr then } \textit{stmt} \textbf{ else } \textit{stmt} \\ \mid\; & \textbf{other} \end{aligned}$$

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?

$$
\begin{aligned}
\mathit{stmt} \;\rightarrow\; & \textbf{if expr then } \mathit{stmt} \\
\mid\; & \textbf{if expr then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
\mid\; & \textbf{other}
\end{aligned}
$$

$$S \rightarrow iS \mid iSeS \mid o$$

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: $\{S \rightarrow i \cdot S,\ S \rightarrow i \cdot SeS,\ S \rightarrow \cdot iS,\ S \rightarrow \cdot iSeS,\ S \rightarrow \cdot o\}$

I_3: $\{S \rightarrow o \cdot\}$

I_4: $\{S \rightarrow iS \cdot,\ S \rightarrow iS \cdot eS\}$

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

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)?

$$S \rightarrow iS \mid iSeS \mid o$$

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

Here's an example Yacc program for this grammar:

```
%%
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 */
  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.

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

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
iioeo
ok
>
```


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

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'
   ;
%%
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");
}
```

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:

```
%left '+'
%left '*'
%%
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.