CS2 Language Processing note 11

Fixing problems with grammars

In Note 10 of this thread, we mentioned three steps involved in the construction of a parser for a given grammar G, and discussed the third of these steps (the parsing algorithm) in some detail. In the present note, we will first discuss one small detail of the parsing algorithm which we omitted in Note 10. The rest of the note will then be devoted to the first of the three steps: the problem of eliminating undesirable features from a grammar G in order to obtain an equivalent $LL(1)$ grammar G'. Once you have followed the material in this note, you should be able to identify and correct various problems with grammars: *ambiguity*, *common prefixes* and *left recursion*.

The end-of-input marker

Recall that in the parse tables we gave in Note 10, we had one row for each nonterminal symbol of the grammar, and one column for each terminal symbol. For the examples we gave, this was enough, but in general we need to include one extra column, which we label with a special symbol \$ called the *end of input marker*. The purpose of this extra column is as follows. Just as the entry in the table for a nonterminal A and a terminal a tells us which production to apply if we are expecting an A and we see a as the next symbol, so the table entry for A and \$ tells us which production to apply if we are expecting an A and we have encountered the *end* of the input. For many grammars (such as the examples considered in Note 10), such a situation can never arise when parsing well-formed sentences of the language; however, there are occasions when this additional column is necessary, as the following example shows.

Consider the language consisting of sequences of zero or more occurrences of the symbol x. We can give a context-free grammar for this language as follows:

$$S \longrightarrow \varepsilon \mid \mathsf{x}\, S$$

Now consider how the algorithm of Note 10 should work for this language. If we are expecting an S (that is, we have S on the top of the stack) and we encounter an x in the input, it is clear that the production to apply is $S \longrightarrow \mathsf{x}\ S$. However, if we are expecting an S and we encounter the end of the input, this does not mean we have a parsing error — it just means that the production we need to apply is $S \longrightarrow \varepsilon$. The parse table for the above grammar therefore looks like this:

	x	\$
S	$S \longrightarrow \mathsf{x}S$	$S \longrightarrow \varepsilon$

Officially the parse tables in Note 10 should also have a column labelled \$, though in these cases this column should be empty.

To incorporate this modification into our parsing algorithm, we just need to treat \$ as an extra terminal symbol (which we assume is different from all the existing terminals), and make sure we explicitly add \$ at the end of the input string. With this minor modification, the parsing algorithm works exactly as we have described it in Note 10.

Ambiguity

We now move on to consider the question: how can we transform a grammar that is not $LL(1)$ (and hence not suitable for predictive parsing) into an equivalent grammar that is? ("Equivalent" here means that the two grammars must define the same language.) We examine in turn three common reasons why a grammar might fail to be $LL(1)$: it might suffer from ambiguity, common prefixes or left recursion.

A grammar is said to be *ambiguous* if there is any sentence with more than one parse tree. Any parser for an ambiguous grammar has to choose somehow which tree to return. There are a number of solutions to this — e.g., the parser could pick one arbitrarily, or we could provide some hints about which to choose — but it is probably best to rewrite the grammar so that it is not ambiguous. Note that no ambiguous grammar can be $LL(1)$, since our parsing algorithm for $LL(1)$ grammars builds the only possible parse tree for a sentence in a deterministic way. (In general, an ambiguity in a grammar will manifest itself in the fact that there are two productions competing for the same cell in the parse table.)

There is no general method for removing ambiguity, and the best approach is usually to step back and consider what language a grammar was intended to capture in the first place. For example, here is a grammar for lists of arguments to a function.

$$\mathit{arguments} \longrightarrow (\ \mathit{list}\ ) \qquad\qquad \mathit{list} \longrightarrow \mathsf{arg} \mid \mathit{list}, \mathit{list}$$

This is ambiguous because of the production $\mathit{list} \longrightarrow \mathit{list},\mathit{list}$. Any sentence with more than two variables, such as $(\mathsf{arg}, \mathsf{arg}, \mathsf{arg})$, will have multiple parse trees. All we need, though, is a grammar that can express "one or more comma-separated occurrences of arg". The following unambiguous grammar does just this.

$$\mathit{arguments} \longrightarrow (\ \mathsf{arg}\ \mathit{rest}\ ) \qquad\qquad \mathit{rest} \longrightarrow \varepsilon \mid\ ,\mathsf{arg}\ \mathit{rest}$$

Figure 11.1 gives examples of how to carry out this construction in various situations.

Arithmetic expressions provide another standard example where what appears to be the simplest grammar turns out to be problematic.

$$\begin{array}{rcl} E & \longrightarrow & E + E \mid E - E \\ & \mid & E * E \mid E \;/\; E \\ & \mid & (E) \mid \mathsf{num} \end{array}$$

Ambiguous	**Language**	**Unambiguous**
$A \longrightarrow B \mid AA$	Lists of one or more B's.	$A \longrightarrow BC$ $C \longrightarrow A \mid \varepsilon$
$A \longrightarrow B \mid AA \mid \varepsilon$	Lists of zero or more B's	$A \longrightarrow BA \mid \varepsilon$
$A \longrightarrow B \mid A;A$	Lists of one or more B's, with punctuation.	$A \longrightarrow BC$ $C \longrightarrow ;A \mid \varepsilon$

Figure 11.1: Fixing some simple ambiguities in a grammar

A sentence like "num + num ∗ num − num" can be parsed many different ways, each with a different parse tree. Seeing this as arithmetic, we know that the "right" thing to do is the multiplication first, followed by the addition and then the subtraction. But how can we capture this in a grammar?

One solution is to add fresh nonterminals that enforce *precedence* between operators, and make them *left associative*. In the following equivalent grammar, '∗' and '/' bind more tightly than '+' and '−'; and operators of equal binding strength act from the left first.

$$\begin{array}{lcccccc} E & \longrightarrow & E+T & \mid & E-T & \mid & T \\ T & \longrightarrow & T*F & \mid & T\,/\,F & \mid & F \\ F & \longrightarrow & (E) & \mid & \mathsf{num} & & \end{array}$$

Here T stands for *term*, and F for *factor*. This is now unambiguous, and also captures more precisely the usual arithmetic conventions on how to interpret expressions like this.

Common prefixes and left factoring

Ambiguity is a problem for any parsing algorithm. The algorithm for predictive parsing also has difficulties with various other features of grammars. For example, consider the following productions describing the use of a **loop** command.

$$\begin{aligned} C \longrightarrow \mathsf{code} \mid &\ \textbf{loop}\ \mathsf{code}\ \textbf{while}\ \mathsf{test} \\ \mid &\ \textbf{loop}\ \mathsf{code}\ \textbf{until}\ \mathsf{test} \end{aligned}$$

There is no ambiguity here, but a predictive parser, trying to expand the nonterminal C, cannot tell which production to choose when the next token is **loop**. We fix this by *left factoring* the grammar: taking out the part common to the conflicting productions, and delaying the point when a predictive parser has to choose between them.

$$\begin{aligned} C &\longrightarrow \mathsf{code} \mid \textbf{loop}\ \mathsf{code}\ T \\ T &\longrightarrow \textbf{while}\ \mathsf{test} \mid \textbf{until}\ \mathsf{test} \end{aligned}$$

The same technique can also be applied to the grammar given on page 3 of Note 9 (see Exercise 2 below).

Left recursion

A more serious problem for predictive parsing arises when a grammar is *left recursive*. This means that there is some nonterminal A such that $A \Rightarrow^* At$ for some sentential form t. This will put a predictive parser into an endless loop, as it repeatedly expands A without consuming any input tokens.

Left recursion can be systematically eliminated from any grammar. The simplest case is *immediate* left-recursion, when it happens in a single step. Suppose we have a left-recursive nonterminal A with the following productions

$$A \longrightarrow At_1 \mid At_2 \mid \ldots \mid At_m \mid u_1 \mid u_2 \mid \ldots \mid u_n$$

where $t_1, \ldots, t_m, u_1, \ldots, u_n$ are sentential forms and none of the u_i begin with A. As with ambiguous grammars, we do best to step back and try to read what this grammar really means. In this case, an A can expand to some u followed by a sequence of zero or more t's; for example as $A \Rightarrow^* u_2t_3t_4t_3$. With this reading, it is possible to see that the following productions give an equivalent grammar.

$$\begin{aligned} A &\longrightarrow u_1A' \mid u_2A' \mid \cdots \mid u_nA' \\ A' &\longrightarrow t_1A' \mid t_2A' \mid \cdots \mid t_mA' \mid \varepsilon \end{aligned}$$

Now the nonterminal A generates the same set of strings but it is no longer immediately left-recursive. (It is also fairly easy to eliminate "non-immediate" forms of left recursion from a grammar, but we will not give the method here.)

The revised grammar for arithmetic expressions given above is not ambiguous, but it is still left recursive. Applying the same method gives an equivalent grammar that is suitable for predictive parsing.

$$\begin{array}{lllllll} E & \longrightarrow & TE' & \qquad & E' & \longrightarrow & +\,TE' \mid -\,TE' \mid \varepsilon \\ T & \longrightarrow & FT' & & T' & \longrightarrow & *\,FT' \mid /\,FT' \mid \varepsilon \\ F & \longrightarrow & (E) \mid \mathsf{num} & & & & \end{array}$$

Unfortunately this also renders the grammar less comprehensible to the reader. This is the price we pay for the (relative) simplicity of the $LL(1)$ parsing algorithm.

Exercises

1. Find two parse trees for the sentence $(\mathsf{arg}, \mathsf{arg}, \mathsf{arg})$ under the first grammar given on page 2. Then give the unique parse tree for the second grammar.
2. Apply left factoring to the grammar given on page 3 of Note 9, in order to fix the "lookahead" problem discussed there and give a grammar suitable for predictive parsing.
3. There are three grammars in this note for arithmetic expressions. Give parse trees in each of them for the sentence $\mathsf{num}{+}\mathsf{num}{*}\mathsf{num}$. For the first, ambiguous, grammar, you should find two different parse trees.

John Longley 2002, Ian Stark 2001