Elimination of Left Recursion

A grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α. Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed. In Section 2.4, we discussed simple left recursion, where there was one production of the form $A \rightarrow A\alpha$. Here we study the general case. In Section 2.4, we showed how the left-recursive pair of productions $A \rightarrow A\alpha \mid \beta$ could be replaced by the non-left-recursive productions

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

without changing the set of strings derivable from A. This rule by itself suffices in many grammars.

Example 4.8. Consider the following grammar for arithmetic expressions.

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{4.10}$$

Eliminating the *immediate left recursion* (productions of the form $A \rightarrow A\alpha$) to the productions for E and then for T, we obtain

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{4.11}$$

□

No matter how many A-productions there are, we can eliminate immediate left recursion from them by the following technique. First, we group the A-productions as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where no β_i begins with an A. Then, we replace the A-productions by

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

The nonterminal A generates the same strings as before but is no longer left recursive. This procedure eliminates all immediate left recursion from the A and A' productions (provided no α_i is ϵ), but it does not eliminate left recursion involving derivations of two or more steps. For example, consider the grammar

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned} \tag{4.12}$$

The nonterminal S is left-recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not

immediately left recursive.

Algorithm 4.1, below, will systematically eliminate left recursion from a grammar. It is guaranteed to work if the grammar has no cycles (derivations of the form $A \stackrel{+}{\Rightarrow} A$) or ϵ-productions (productions of the form $A \to \epsilon$). Cycles can be systematically eliminated from a grammar as can ϵ-productions (see Exercises 4.20 and 4.22).

Algorithm 4.1. Eliminating left recursion.

Input. Grammar G with no cycles or ϵ-productions.

Output. An equivalent grammar with no left recursion.

Method. Apply the algorithm in Fig. 4.7 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

1. Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2. **for** $i := 1$ **to** n **do begin**
   for $j := 1$ **to** $i - 1$ **do begin**
   replace each production of the form $A_i \to A_j\gamma$
   by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$,
   where $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all the current A_j-productions;
   end
   eliminate the immediate left recursion among the A_i-productions
   end

Fig. 4.7. Algorithm to eliminate left recursion from a grammar.

The reason the procedure in Fig. 4.7 works is that after the $i - 1^{st}$ iteration of the outer **for** loop in step (2), any production of the form $A_k \to A_l\alpha$, where $k < i$, must have $l > k$. As a result, on the next iteration, the inner loop (on j) progressively raises the lower limit on m in any production $A_i \to A_m\alpha$, until we must have $m \geq i$. Then, eliminating immediate left recursion for the A_i-productions forces m to be greater than i.

Example 4.9. Let us apply this procedure to grammar (4.12). Technically, Algorithm 4.1 is not guaranteed to work, because of the ϵ-production, but in this case the production $A \to \epsilon$ turns out to be harmless.

We order the nonterminals S, A. There is no immediate left recursion among the S-productions, so nothing happens during step (2) for the case $i = 1$. For $i = 2$, we substitute the S-productions in $A \to Sd$ to obtain the following A-productions.

$$A \to Ac \mid Aad \mid bd \mid \epsilon$$

Eliminating the immediate left recursion among the A-productions yields the following grammar.

$$
\begin{aligned}
S &\rightarrow Aa \mid b \\
A &\rightarrow bdA' \mid A' \\
A' &\rightarrow cA' \mid adA' \mid \epsilon
\end{aligned}
$$

□

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

For example, if we have the two productions

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

on seeing the input token **if**, we cannot immediately tell which production to choose to expand *stmt*. In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$. However, we may defer the decision by expanding A to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β_1 or to β_2. That is, left-factored, the original productions become

$$
\begin{aligned}
A &\rightarrow \alpha A' \\
A' &\rightarrow \beta_1 \mid \beta_2
\end{aligned}
$$

Algorithm 4.2. Left factoring a grammar.

Input. Grammar G.

Output. An equivalent left-factored grammar.

Method. For each nonterminal A find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$, i.e., there is a nontrivial common prefix, replace all the A productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$ where γ represents all alternatives that do not begin with α by

$$
\begin{aligned}
A &\rightarrow \alpha A' \mid \gamma \\
A' &\rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n
\end{aligned}
$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

Example 4.10. The following grammar abstracts the dangling-else problem:

$$
\begin{aligned}
S &\rightarrow iEtS \mid iEtSeS \mid a \\
E &\rightarrow b
\end{aligned}
\tag{4.13}
$$

Here i, t, and e stand for **if**, **then** and **else**, E and S for "expression" and "statement." Left-factored, this grammar becomes:

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned} \tag{4.14}$$

Thus, we may expand S to $iEtSS'$ on input i, and wait until $iEtS$ has been seen to decide whether to expand S' to eS or to ϵ. Of course, grammars (4.13) and (4.14) are both ambiguous, and on input e, it will not be clear which alternative for S' should be chosen. Example 4.19 discusses a way out of this dilemma. □

Non-Context-Free Language Constructs

It should come as no surprise that some languages cannot be generated by any grammar. In fact, a few syntactic constructs found in many programming languages cannot be specified using grammars alone. In this section, we shall present several of these constructs, using simple abstract languages to illustrate the difficulties.

Example 4.11. Consider the abstract language $L_1 = \{wcw \mid w \text{ is in } (a|b)^*\}$. L_1 consists of all words composed of a repeated string of a's and b's separated by a c, such as $aabcaab$. It can be proven this language is not context free. This language abstracts the problem of checking that identifiers are declared before their use in a program. That is, the first w in wcw represents the declaration of an identifier w. The second w represents its use. While it is beyond the scope of this book to prove it, the non-context-freedom of L_1 directly implies the non-context-freedom of programming languages like Algol and Pascal, which require declaration of identifiers before their use, and which allow identifiers of arbitrary length.

For this reason, a grammar for the syntax of Algol or Pascal does not specify the characters in an identifier. Instead, all identifiers are represented by a token such as **id** in the grammar. In a compiler for such a language, the semantic analysis phase checks that identifiers have been declared before their use. □

Example 4.12. The language $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is not context free. That is, L_2 consists of strings in the language generated by the regular expression $a^*b^*c^*d^*$ such that the number of a's and c's are equal and the number of b's and d's are equal. (Recall a^n means a written n times.) L_2 abstracts the problem of checking that the number of formal parameters in the declaration of a procedure agrees with the number of actual parameters in a use of the procedure. That is, a^n and b^m could represent the formal parameter lists in two procedures declared to have n and m arguments, respectively. Then c^n and d^m represent the actual parameter lists in calls to these two procedures.

Again note that the typical syntax of procedure definitions and uses does not concern itself with counting the number of parameters. For example, the CALL statement in a Fortran-like language might be described