$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{matched\_stmt} \\
 & | & \mathit{open\_stmt} \\
\mathit{matched\_stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matched\_stmt} \textbf{ else } \mathit{matched\_stmt} \\
 & | & \textbf{other} \\
\mathit{open\_stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
 & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matched\_stmt} \textbf{ else } \mathit{open\_stmt}
\end{array}
$$

Figure 4.10: Unambiguous grammar for if-then-else statements

4.3.3 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 is needed to eliminate left recursion. In Section 2.4.5, we discussed *immediate left recursion*, where there is a production of the form $A \rightarrow A\alpha$. Here, we study the general case. In Section 2.4.5, we showed how the left-recursive pair of productions $A \rightarrow A\alpha \mid \beta$ could be replaced by the non-left-recursive productions:

$$
\begin{array}{l}
A \rightarrow \beta A' \\
A' \rightarrow \alpha A' \mid \epsilon
\end{array}
$$

without changing the strings derivable from A. This rule by itself suffices for many grammars.

Example 4.17: The non-left-recursive expression grammar (4.2), repeated here,

$$
\begin{array}{l}
E \rightarrow T\ E' \\
E' \rightarrow +\ T\ E' \\
T \rightarrow F\ T' \\
T' \rightarrow *\ F\ T' \\
F \rightarrow (\ E\ ) \mid \textbf{id}
\end{array}
$$

is obtained by eliminating immediate left recursion from the expression grammar (4.1). The left-recursive pair of productions $E \rightarrow E + T \mid T$ are replaced by $E \rightarrow T\ E'$ and $E' \rightarrow +\ T\ E' \mid \epsilon$. The new productions for T and T' are obtained similarly by eliminating immediate left recursion. □

Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions. First, group the 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, replace the A-productions by

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

The nonterminal A generates the same strings as before but is no longer left recursive. This procedure eliminates all 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

$$\begin{array}{l} S \to A\ a \mid b \\ A \to A\ c \mid S\ d \mid \epsilon \end{array} \tag{4.18}$$

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

Algorithm 4.19, below, systematically eliminates 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 eliminated systematically from a grammar, as can ϵ-productions (see Exercises 4.4.6 and 4.4.7).

Algorithm 4.19: 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. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
1)  arrange the nonterminals in some order A1, A2, ..., An.
2)  for ( each i from 1 to n ) {
3)      for ( each j from 1 to i - 1 ) {
4)          replace each production of the form Ai → Aj γ by the
                productions Ai → δ1 γ | δ2 γ | ··· | δk γ, where
                Aj → δ1 | δ2 | ··· | δk are all current Aj-productions
5)      }
6)      eliminate the immediate left recursion among the Ai-productions
7)  }
```

Figure 4.11: Algorithm to eliminate left recursion from a grammar

The procedure in Fig. 4.11 works as follows. In the first iteration for $i = 1$, the outer for-loop of lines (2) through (7) eliminates any immediate left recursion among A_1-productions. Any remaining A_1 productions of the form $A_1 \to A_l \alpha$ must therefore have $l > 1$. After the $i-1$st iteration of the outer for-loop, all nonterminals A_k, where $k < i$, are "cleaned"; that is, any production $A_k \to A_l \alpha$, must have $l > k$. As a result, on the ith iteration, the inner loop

of lines (3) through (5) progressively raises the lower limit in any production $A_i \rightarrow A_m\alpha$, until we have $m \geq i$. Then, eliminating immediate left recursion for the A_i productions at line (6) forces m to be greater than i.

Example 4.20: Let us apply Algorithm 4.19 to the grammar (4.18). Technically, the algorithm is not guaranteed to work, because of the ϵ-production, but in this case, the production $A \rightarrow \epsilon$ 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 the outer loop for $i = 1$. For $i = 2$, we substitute for S in $A \rightarrow S\ d$ to obtain the following A-productions.

$$A \rightarrow A\ c \mid A\ a\ d \mid b\ d \mid \epsilon$$

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

$$\begin{aligned} S &\rightarrow A\ a \mid b \\ A &\rightarrow b\ d\ A' \mid A' \\ A' &\rightarrow c\ A' \mid a\ d\ A' \mid \epsilon \end{aligned}$$

□

4.3.4 Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

For example, if we have the two productions

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

on seeing the input **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 $\alpha\beta_2$. 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 4.21: Left factoring a grammar.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.