regular expressions than from arbitrary grammars.

There are no firm guidelines as to what to put into the lexical rules, as opposed to the syntactic rules. Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords, and white space. Grammars, on the other hand, are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on. These nested structures cannot be described by regular expressions.

4.3.2 Eliminating Ambiguity

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example, we shall eliminate the ambiguity from the following "dangling-else" grammar:

$$\begin{array}{rcl} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{other} \end{array} \tag{4.14}$$

Here "**other**" stands for any other statement. According to this grammar, the compound conditional statement

$$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$$

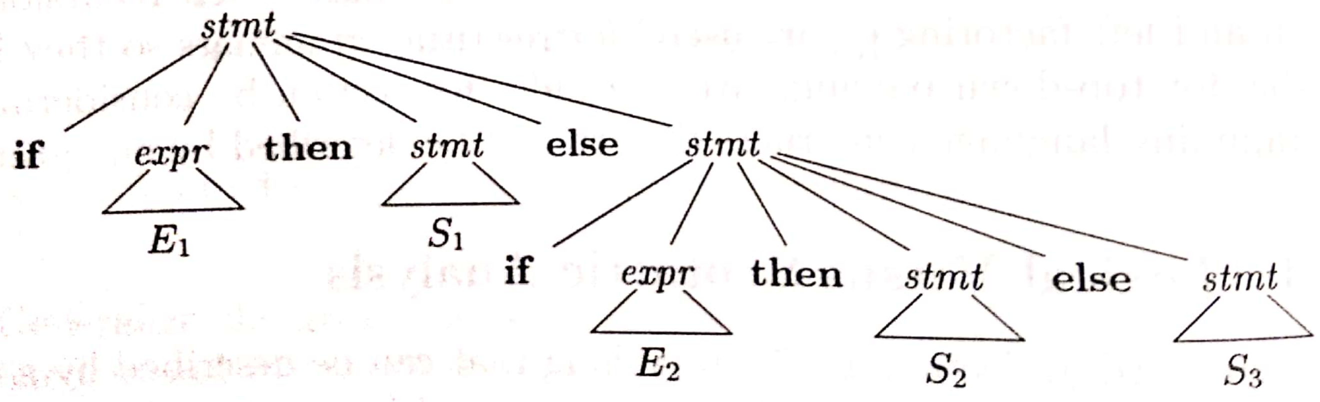

Figure 4.8: Parse tree for a conditional statement

has the parse tree shown in Fig. 4.8.[1] Grammar (4.14) is ambiguous since the string

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2 \tag{4.15}$$

has the two parse trees shown in Fig. 4.9.

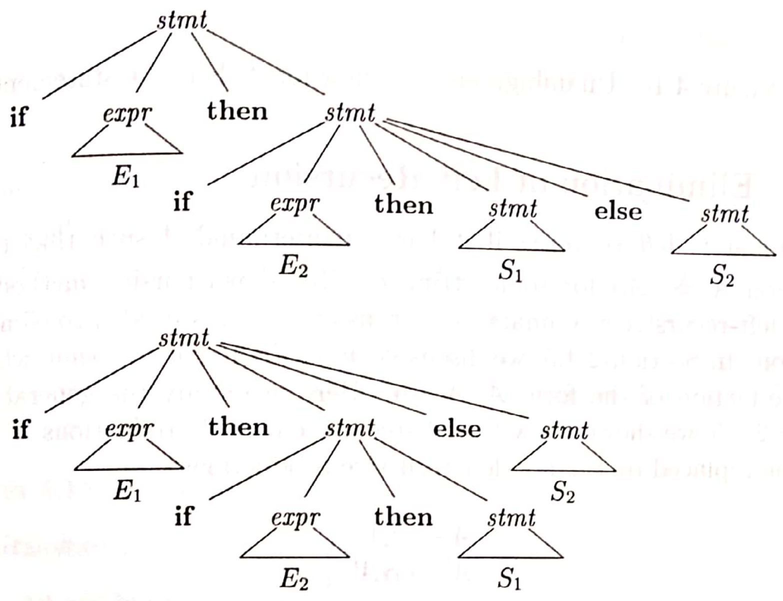

Figure 4.9: Two parse trees for an ambiguous sentence

In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is, "Match each **else** with the closest unmatched **then**."[2] This disambiguating rule can theoretically be incorporated directly into a grammar, but in practice it is rarely built into the productions.

Example 4.16: We can rewrite the dangling-else grammar (4.14) as the following unambiguous grammar. The idea is that a statement appearing between a **then** and an **else** must be "matched"; that is, the interior statement must not end with an unmatched or open **then**. A matched statement is either an **if-then-else** statement containing no open statements or it is any other kind of unconditional statement. Thus, we may use the grammar in Fig. 4.10. This grammar generates the same strings as the dangling-else grammar (4.14), but it allows only one parsing for string (4.15); namely, the one that associates each **else** with the closest previous unmatched **then**. □

[1]The subscripts on E and S are just to distinguish different occurrences of the same nonterminal, and do not imply distinct nonterminals.

[2]We should note that C and its derivatives are included in this class. Even though the C family of languages do not use the keyword **then**, its role is played by the closing parenthesis for the condition that follows **if**.

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

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 $A \overset{+}{\Rightarrow} A\alpha$ 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 $A \to A\alpha$. Here we stud In