! b) The grammar of Fig. 4.7 and its generalization in part (a) allow declarations that are contradictory and/or redundant, such as:

```
declare foo real fixed real floating
```

We could insist that the syntax of the language forbid such declarations; that is, every declaration generated by the grammar has exactly one value for each of the n options. If we do, then for any fixed n there is only a finite number of legal declarations. The language of legal declarations thus has a grammar (and also a regular expression), as any finite language does. The obvious grammar, in which the start symbol has a production for every legal declaration has $n!$ productions and a total production length of $O(n \times n!)$. You must do better: a total production length that is $O(n2^n)$.

!! c) Show that any grammar for part (b) must have a total production length of at least 2^n.

d) What does part (c) say about the feasibility of enforcing nonredundancy and noncontradiction among options in declarations via the syntax of the programming language?

4.3 Writing a Grammar

Grammars are capable of describing most, but not all, of the syntax of programming languages. For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar. Therefore, the sequences of tokens accepted by a parser form a superset of the programming language; subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

This section begins with a discussion of how to divide work between a lexical analyzer and a parser. We then consider several transformations that could be applied to get a grammar more suitable for parsing. One technique can eliminate ambiguity in the grammar, and other techniques — left-recursion elimination and left factoring — are useful for rewriting grammars so they become suitable for top-down parsing. We conclude this section by considering some programming language constructs that cannot be described by any grammar.

4.3.1 Lexical Versus Syntactic Analysis

As we observed in Section 4.2.7, everything that can be described by a regular expression can also be described by a grammar. We may therefore reasonably ask: "Why use regular expressions to define the lexical syntax of a language?" There are several reasons.

1. Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.

2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.

3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.

4. More efficient lexical analyzers can be constructed automatically from 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

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

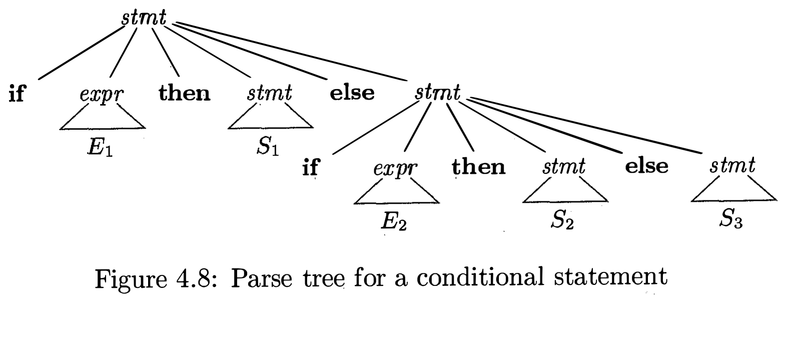

Figure 4.8: Parse tree for a conditional statement

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.

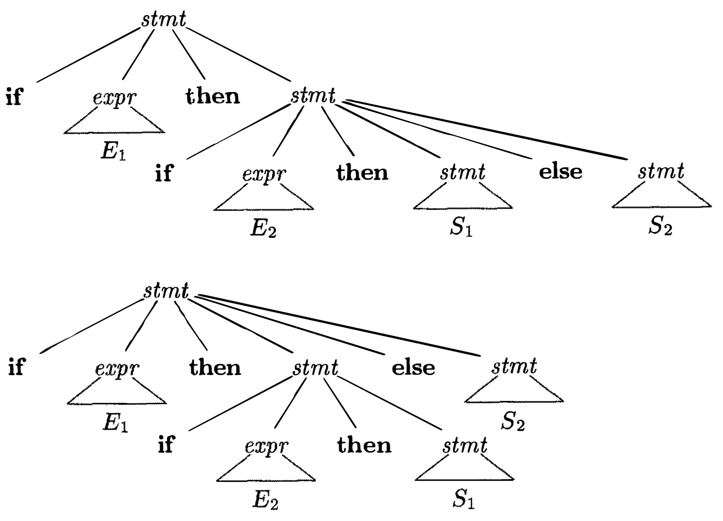

Figure 4.9: Two parse trees for an ambiguous sentence

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**.

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

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 study the general case. In Section 2.4.5, we showed how the left-recursive pair of productions $A \to A\alpha \mid \beta$ could be replaced by the non-left-recursive productions:

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

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,

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

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

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

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

$$\begin{array}{l} A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}$$

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 $A \overset{+}{\Rightarrow} A$) 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 $A_1, A_2, \ldots, A_n$.
2)  for ( each $i$ from 1 to $n$ ) {
3)          for ( each $j$ from 1 to $i - 1$ ) {
4)                  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 current $A_j$-productions
5)          }
6)          eliminate the immediate left recursion among the $A_i$-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 $A_i \to A_m\alpha$, 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 $A \to \epsilon$ 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 the outer loop for $i = 1$. For $i = 2$, we substitute for S in $A \to S\ d$ to obtain the following A-productions.

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

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

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

□

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

$$\begin{array}{lcl} \mathit{stmt} & \to & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\ & \mid & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \end{array}$$

on seeing the input **if**, we cannot immediately tell which production to choose to expand *stmt*. In general, if $A \to \alpha\beta_1 \mid \alpha\beta_2$ 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

$$\begin{array}{l} A \to \alpha A' \\ A' \to \beta_1 \mid \beta_2 \end{array}$$

Algorithm 4.21: 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 of 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

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

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

Example 4.22: The following grammar abstracts the "dangling-else" problem:

$$\begin{array}{l} S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a \\ E \rightarrow b \end{array} \qquad (4.23)$$

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

$$\begin{array}{l} S \rightarrow i\ E\ t\ S\ S' \mid a \\ S' \rightarrow e\ S \mid \epsilon \\ E \rightarrow b \end{array} \qquad (4.24)$$

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, these grammars are both ambiguous, and on input e, it will not be clear which alternative for S' should be chosen. Example 4.33 discusses a way out of this dilemma. □

4.3.5 Non-Context-Free Language Constructs

A few syntactic constructs found in typical programming languages cannot be specified using grammars alone. Here, we consider two of these constructs, using simple abstract languages to illustrate the difficulties.

Example 4.25: The language in this example abstracts the problem of checking that identifiers are declared before they are used in a program. The language consists of strings of the form wcw, where the first w represents the declaration of an identifier w, c represents an intervening program fragment, and the second w represents the use of the identifier.

The abstract language is $L_1 = \{wcw \mid w \text{ is in } (\mathbf{a}|\mathbf{b})^*\}$. L_1 consists of all words composed of a repeated string of a's and b's separated by c, such as $aabcaab$. 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 C and Java, which require declaration of identifiers before their use and which allow identifiers of arbitrary length.

For this reason, a grammar for C or Java does not distinguish among identifiers that are different character strings. 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 are declared before they are used. □

Example 4.26: The non-context-free language in this example abstracts the problem of checking that the number of formal parameters in the declaration of a function agrees with the number of actual parameters in a use of the function. The language consists of strings of the form $a^n b^m c^n d^m$. (Recall a^n means a written n times.) Here a^n and b^m could represent the formal-parameter lists of two functions declared to have n and m arguments, respectively, while c^n and d^m represent the actual-parameter lists in calls to these two functions.

The abstract language is $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$. That is, L_2 consists of strings in the language generated by the regular expression $\mathbf{a^*b^*c^*d^*}$ such that the number of a's and c's are equal and the number of b's and d's are equal. This language is not context free.

Again, the typical syntax of function declarations and uses does not concern itself with counting the number of parameters. For example, a function call in C-like language might be specified by

$$\begin{array}{rcl} stmt & \rightarrow & \mathbf{id}\ (\ expr\_list\ ) \\ expr\_list & \rightarrow & expr\_list\ ,\ expr \\ & | & expr \end{array}$$

with suitable productions for *expr*. Checking that the number of parameters in a call is correct is usually done during the semantic-analysis phase. □

4.3.6 Exercises for Section 4.3

Exercise 4.3.1: The following is a grammar for regular expressions over symbols a and b only, using $+$ in place of $|$ for union, to avoid conflict with the use of vertical bar as a metasymbol in grammars:

$$\begin{array}{lcl} rexpr & \rightarrow & rexpr + rterm \mid rterm \\ rterm & \rightarrow & rterm\ rfactor \mid rfactor \\ rfactor & \rightarrow & rfactor\ * \mid rprimary \\ rprimary & \rightarrow & \mathbf{a} \mid \mathbf{b} \end{array}$$

a) Left factor this grammar.

b) Does left factoring make the grammar suitable for top-down parsing?

c) In addition to left factoring, eliminate left recursion from the original grammar.

d) Is the resulting grammar suitable for top-down parsing?

Exercise 4.3.2: Repeat Exercise 4.3.1 on the following grammars:

a) The grammar of Exercise 4.2.1.

b) The grammar of Exercise 4.2.2(a).

c) The grammar of Exercise 4.2.2(c).

d) The grammar of Exercise 4.2.2(e).

e) The grammar of Exercise 4.2.2(g).

! **Exercise 4.3.3:** The following grammar is proposed to remove the "dangling-else ambiguity" discussed in Section 4.3.2:

$$\begin{array}{lcl} \mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\ & | & \mathit{matchedStmt} \\ \mathit{matchedStmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matchedStmt} \textbf{ else } \mathit{stmt} \\ & | & \textbf{other} \end{array}$$

Show that this grammar is still ambiguous.

4.4 Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first, as discussed in Section 2.3.4). Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Example 4.27: The sequence of parse trees in Fig. 4.12 for the input **id+id*id** is a top-down parse according to grammar (4.2), repeated here:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \ |\ \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \ |\ \epsilon \\ F & \rightarrow & (\ E\ )\ |\ \textbf{id} \end{array} \tag{4.28}$$

This sequence of trees corresponds to a leftmost derivation of the input. □

At each step of a top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say A. Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.

The section begins with a general form of top-down parsing, called recursive-descent parsing, which may require backtracking to find the correct A-production to be applied. Section 2.4.2 introduced predictive parsing, a special case of recursive-descent parsing, where no backtracking is required. Predictive parsing chooses the correct A-production by looking ahead at the input a fixed number of symbols, typically we may look only at one (that is, the next input symbol).