7.8 Eliminating Ambiguity

Ambiguity is of several kinds. *Lexical ambiguity* or *word-sense ambiguity* arises when a word in a natural language or a token or symbol in a programming language has multiple meanings. Lexical ambiguity has been pretty much eliminated in programming languages using reserved and key words and allowing overloading of operators and method names only where the context helps us to disambiguate the meaning. *Structural ambiguity* or *syntactic ambiguity* arises when there are multiple parse trees, as in the examples from the previous section. Structural ambiguities exist in programming languages, for example, in nesting of if-then-else statements and in precedence of operators in arithmetic expressions.

In general, there are three ways of working around such ambiguities:

1. Relying on the programmer to always disambiguate by providing parentheses or brackets appropriately (usually not a good choice).
2. Using operator precedence rules outside of the grammar, that is, in later stages of the compiler.
3. Re-writing the grammar by introducing new non-terminals to encode the rules of operator precedence in the grammar itself.

EXAMPLE 7.17

Let us see how to re-write the expression grammar from Example 7.15 to make it unambiguous. The ambiguity arose in the grammar because the productions for different arithmetic operators could be applied in any order. Rules of precedence which are applied conventionally in programming languages tell us that * and / have a higher precedence than + and –. These rules can be incorporated into the grammar itself by introducing new variables for intermediate nodes in the parse tree as follows:

$S \rightarrow (S) \mid S + T \mid S - T \mid T$	An expression has a + or – operator or is just a term
$T \rightarrow T * F \mid T / F \mid F$	A term has a * or / operator or is just a factor
$F \rightarrow$ variable | constant	A factor is just a variable or a constant

Notice that no variable occurs twice on the right-hand side of any production rule, thereby eliminating any structural ambiguity. The string $a + b / c$ from Example 7.16 has only one leftmost derivation now:

$$S \Rightarrow S + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T / F \Rightarrow a + F / F \Rightarrow a + b / F \Rightarrow a + b / c$$

From the parse tree for this derivation, shown in Fig. 7.6, it is clear that the division must be performed before the addition (if a leftmost derivation is the convention in the given programming language), thereby making the expression unambiguous even in the absence of parentheses in the expression.

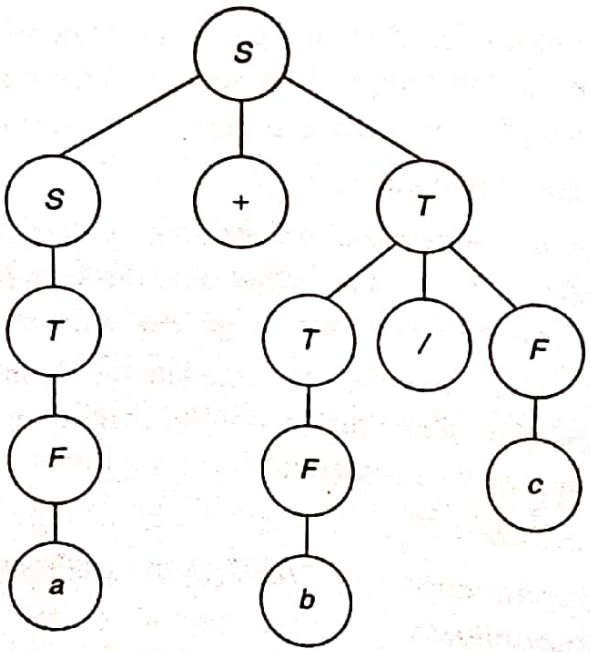

FIGURE 7.6 Parse tree for leftmost derivation using unambiguous grammar.