Note: Grammars, like recursive algorithms, must have carefully chosen base cases. You must ensure that, when a string is decomposed far enough, it will always reach the form of one of the grammar's base cases.

CHECK POINT

Question 1 Consider the language of these character strings: $, cc$d, cccc$dd, cccccc$ddd, and so on. Write a recursive grammar for this language.

5.2 Algebraic Expressions

VideoNote
Processing expressions

One of the tasks a compiler must perform is to recognize and evaluate algebraic expressions. For example, consider the C++ assignment statement

```
y = x + z * (w / k + z * (7 - 6));
```

A C++ compiler must determine whether the right side is a syntactically legal algebraic expression; if so, the compiler must then indicate how to compute the expression's value.

There are several common definitions for a "syntactically legal" algebraic expression. Some definitions force an expression to be fully parenthesized—that is, to have parentheses around each pair of operands together with their operator. Thus, you would have to write `((a * b) * c)` rather than `a * b * c`. In general, the stricter a definition, the easier it is to recognize a syntactically legal expression. On the other hand, conforming to overly strict rules of syntax is an inconvenience for programmers.

This section presents three different languages for algebraic expressions. The expressions in these languages are easy to recognize and evaluate but are generally inconvenient to use. However, these languages provide us with good, nontrivial applications of grammars. We will see other languages of algebraic expressions whose members are difficult to recognize and evaluate but are convenient to use. To avoid unnecessary complications, assume that you have only the binary operators +, –, *, and / (no unary operators or exponentiation). Also, assume that all operands in the expression are single-letter identifiers.

5.2.1 Kinds of Algebraic Expressions

The algebraic expressions you learned about in school are called **infix expressions**. The term "infix" indicates that every binary operator appears *between* its operands. For example, in the expression

$$a + b$$

the operator + is between its operands a and b. This convention necessitates associativity rules, precedence rules, and the use of parentheses to avoid ambiguity. For example, the expression

$$a + b * c$$

is ambiguous. What is the second operand of the +? Is it b or is it $(b * c)$? Similarly, the first operand of the * could be either b or $(a + b)$. The rule that * has higher precedence than + removes the ambiguity by specifying that b is the first operand of the * and that $(b * c)$ is the second operand of the +. If you want another interpretation, you must use parentheses:

$$(a + b) * c$$

Even with precedence rules, an expression like

$a / b * c$

is ambiguous. Typically, / and * have equal precedence, so you could interpret the expression either as $(a / b) * c$ or as $a / (b * c)$. The common practice is to *associate from left to right*, thus yielding the first interpretation.

Two alternatives to the traditional infix convention are **prefix** and **postfix expressions**. Under these conventions, an operator appears either before its operands (prefix) or after its operands (postfix). Thus, the infix expression

In a prefix expression, an operator precedes its operands

$a + b$

is written in prefix form as

$+ \; a \; b$

In a postfix expression, an operator follows its operands

and in postfix form as

$a \; b \; +$

To further illustrate the conventions, consider the two interpretations of the infix expression $a + b * c$ just considered. You write the expression

$a + (b * c)$

in prefix form as

$+ \; a \; * \; bc$

The + appears before its operands *a* and (* *b c*), and the * appears before its operands *b* and *c*. The same expression is written in postfix form as

$a \; b \; c \; * \; +$

The * appears after its operands *b* and *c,* and the + appears after its operands *a* and (*b c* *).

Similarly, you write the expression

$(a + b) * c$

in prefix form as

$* \; + \; a \; b \; c$

The * appears before its operands (+ *a b*) and *c*, and the + appears before its operands *a* and *b*. The same expression is written in postfix form as

$a \; b \; + \; c \; *$

The + appears after its operands *a* and *b*, and the * appears after its operands (*a b* +) and *c*.

If the infix expression is fully parenthesized, converting it to either prefix or postfix form is straightforward. Because each operator then corresponds to a pair of parentheses, you simply move the operator to the position marked by either the open parenthesis "("—if you want to convert to prefix form—or the close parenthesis ")"—if you want to convert to postfix form. This position either precedes or follows the operands of the operator. All parentheses would then be removed.

For example, consider the fully parenthesized infix expression

$((a + b) * c)$

To convert this expression to prefix form, you first move each operator to the position marked by its corresponding open parenthesis:

Converting to prefix form

$$\begin{array}{l}((a\ b)c)\\ \downarrow\downarrow\\ *+\end{array}$$

Next, you remove the parentheses to get the desired prefix expression:

$$*\ +\ a\ b\ c$$

Similarly, to convert the infix expression to postfix form, you move each operator to the position marked by its corresponding close parenthesis:

Converting to postfix form

$$\begin{array}{l}((ab)c)\\ \ \ \downarrow\downarrow\\ \ \ +\ *\end{array}$$

Then you remove the parentheses:

$$a\ b\ +\ c\ *$$

When an infix expression is not fully parenthesized, these conversions are more complex. Chapter 6 discusses the general case of converting an infix expression to postfix form.

Prefix and postfix expressions never need precedence rules, association rules, or parentheses

The advantage of prefix and postfix expressions is that they never need precedence rules, association rules, or parentheses. Therefore, the grammars for prefix and postfix expressions are quite simple. In addition, the algorithms that recognize and evaluate these expressions are relatively straightforward.

5.2.2 Prefix Expressions

A grammar that defines the language of all prefix expressions is

$$\begin{aligned}&\langle prefix\rangle = \langle identifier\rangle \mid \langle operator\rangle\ \langle prefix\rangle\ \langle prefix\rangle\\ &\langle operator\rangle = + \mid - \mid * \mid /\\ &\langle identifier\rangle = a \mid b \mid \ldots \mid z\end{aligned}$$

From this grammar, you can construct a recursive algorithm that recognizes whether a string is a prefix expression. If the string is of length 1, it is a prefix expression if and only if the string is a single lowercase letter. Strings of length 1 can be the base case. If the length of the string is greater than 1, then for it to be a legal prefix expression, it must be of the form

$$\langle operator\rangle\ \langle prefix\rangle\ \langle prefix\rangle$$

Thus, the algorithm must check to see whether

- The first character of the string is an operator

and

- The remainder of the string consists of two consecutive prefix expressions

The first task is trivial, but the second is a bit tricky. How can you tell whether you are looking at two consecutive prefix expressions? A key observation is that if you add *any* string of nonblank characters to the end of a prefix expression, you will no longer have a prefix expression. That is, if E is