The scheme in Figure 4.12 ignores one critical issue: initializing `cost`. The grammar, as written, contains no production that can appropriately initialize `cost` to zero. The solution, as described earlier, is to modify the grammar in a way that creates a place for the initialization. An initial production, such as *Start* → *CostInit Block*, along with *CostInit* → ϵ, does this. The framework can perform the assignment `cost` ← 0 on the reduction from ϵ to *CostInit*.

Type Inference for Expressions, Revisited

The problem of inferring types for expressions fit well into the attribute-grammar framework, as long as we assumed that leaf nodes already had type information. The simplicity of the solution shown in Figure 4.7 derives from two principal facts. First, because expression types are defined recursively on the expression tree, the natural flow of information runs bottom up from the leaves to the root. This biases the solution toward an S-attributed grammar. Second, expression types are defined in terms of the syntax of the source language. This fits well with the attribute-grammar framework, which implicitly requires the presence of a parse tree. All the type information can be tied to instances of grammar symbols, which correspond precisely to nodes in the parse tree.

We can reformulate this problem in an ad hoc framework, as shown in Figure 4.13. It uses the type inference functions introduced with Figure 4.7. The resulting framework looks similar to the attribute grammar for the same purpose from Figure 4.7. The ad hoc framework provides no real advantage for this problem.

Production			Syntax-Directed Actions
Expr	→	*Expr* – *Term*	{ $\$\$ \leftarrow \mathcal{F}_{+}(\$1,\$3)$ }
	|	*Expr* – *Term*	{ $\$\$ \leftarrow \mathcal{F}_{-}(\$1,\$3)$ }
	|	*Term*	{ $\$\$ \leftarrow \$1$ }
Term	→	*Term* × *Factor*	{ $\$\$ \leftarrow \mathcal{F}_{\times}(\$1,\$3)$ }
	|	*Term* ÷ *Factor*	{ $\$\$ \leftarrow \mathcal{F}_{\div}(\$1,\$3)$ }
	|	*Factor*	{ $\$\$ \leftarrow \$1$ }
Factor	→	(*Expr*)	{ $\$\$ \leftarrow \$2$ }
	|	num	{ $\$\$ \leftarrow$ *type of the* num }
	|	name	{ $\$\$ \leftarrow$ *type of the* name }

■ **FIGURE 4.13** Ad Hoc Framework for Inferring Expression Types.

Production			Syntax-Directed Actions
Expr	→	*Expr* + *Term*	{ `$$` ← `MakeNode`$_2$ `(plus, $1, $3);` `$$.type` ← $\mathcal{F}_{+}$`($1.type, $3.type)` }
	|	*Expr* – *Term*	{ `$$` ← `MakeNode`$_2$`(minus, $1, $3);` `$$.type` ← $\mathcal{F}_{-}$`($1.type,$3.type)` }
	|	*Term*	{ `$$` ← `$1` }
Term	→	*Term* × *Factor*	{ `$$` ← `MakeNode`$_2$`(times, $1, $3);` `$$.type` ← $\mathcal{F}_{\times}$`($1.type, $3.type)` }
	|	*Term* ÷ *Factor*	{ `$$` ← `MakeNode`$_2$`(divide, $1, $3);` `$$.type` ← $\mathcal{F}_{\div}$`($1.type, $3.type)` }
	|	*Factor*	{ `$$` ← `$1` }
Factor	→	(*Expr*)	{ `$$` ← `$2` }
	|	num	{ `$$` ← `MakeNode`$_0$`(number);` `$$.text` ← *scanned text;* `$$.type` ← *type of the number* }
	|	name	{ `$$` ← `MakeNode`$_0$`(identifier);` `$$.text` ← *scanned text;* `$$.type` ← *type of the identifier* }

■ **FIGURE 4.14** Building an Abstract Syntax Tree and Inferring Expression Types.

Building an Abstract Syntax Tree

Compiler front ends must build an intermediate representation of the program for use in the compiler's middle part and its back end. Abstract syntax trees are a common form of tree-structured IR. The task of building an AST fits neatly into an ad hoc syntax-directed translation scheme.

Assume that the compiler has a series of routines named `MakeNode`$_i$, for $0 \le i \le 3$. The routine takes, as its first argument, a constant that uniquely identifies the grammar symbol that the new node will represent. The remaining i arguments are the nodes that head each of the i subtrees. Thus, `MakeNode`$_0$ `(number)` constructs a leaf node and marks it as representing a num. Similarly,

`MakeNode`$_2$`(Plus, MakeNode`$_0$`(number,) MakeNode`$_0$`(number))`

builds an AST rooted in a node for `plus` with two children, each of which is a leaf node for num.

The `MakeNode` routines can implement the tree in any appropriate way. For example, they might map the structure onto a binary tree, as discussed in Section B.3.1.