Attributed Grammars

An attributed grammar (or **attribute grammar**) is a grammar along with a set of **rules** for each production. Each rule defines an attribute of a grammar symbol in the production, often in terms of other attributes of grammar symbols in the production. For example,

production	**rules**
$E_1 \rightarrow (E_2)$	E_1.type := E_2.type
	E_1.isConstant := E_2.isConstant

Where subscripts have been used, as is conventional, to distinguish between two instances of the nonterminal E in the production.

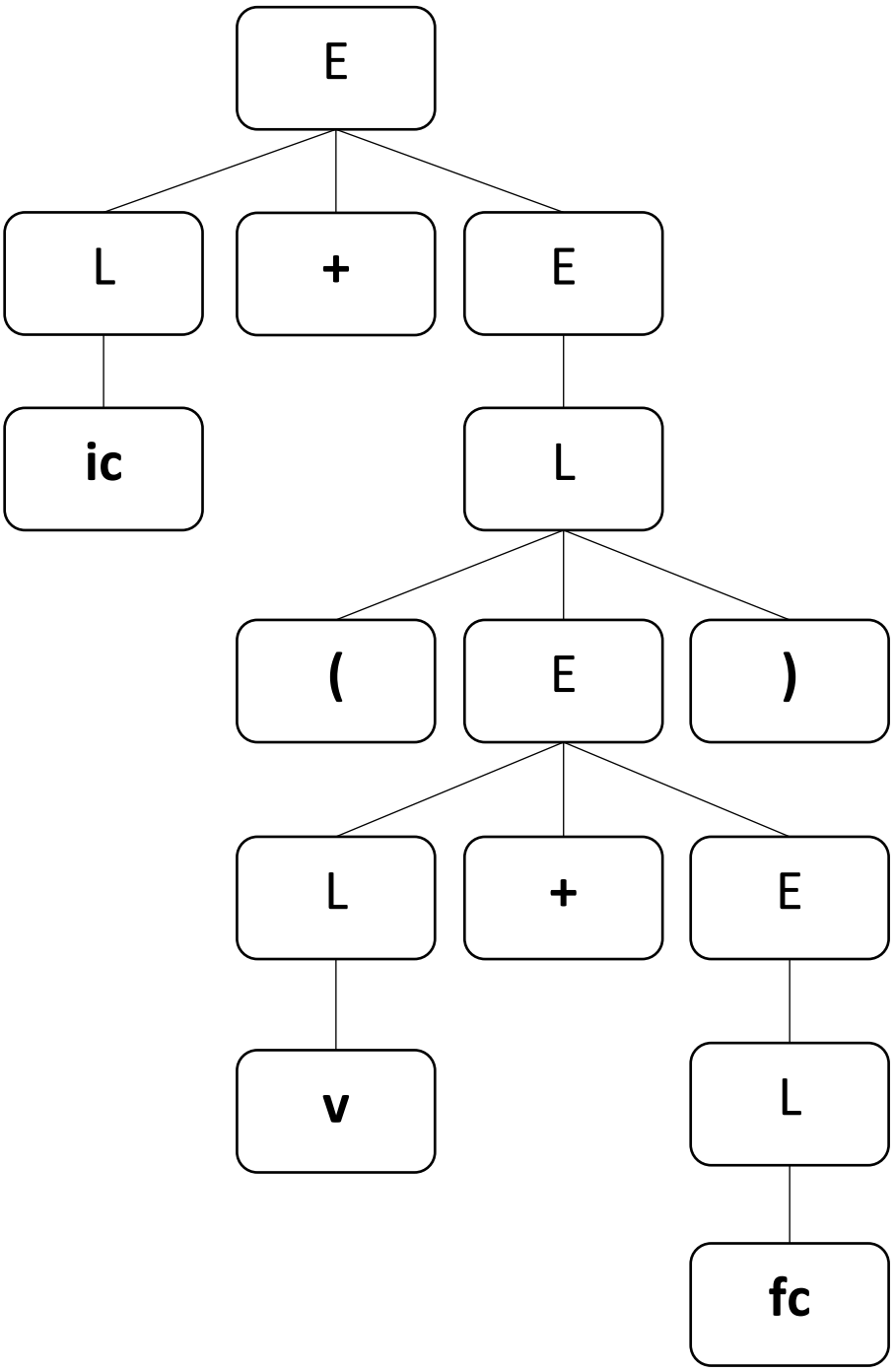
If a rule gives a grammar symbol an attribute, then each instance of that grammar symbol in the parse tree gets its own instance of that attribute. In the above grammar, all E's would get an attribute *type* and an attribute *isConstant*.

The rules are declarative. They imply no specific evaluation order—they require only that the attributes in the right-hand side of the rule have already been evaluated.

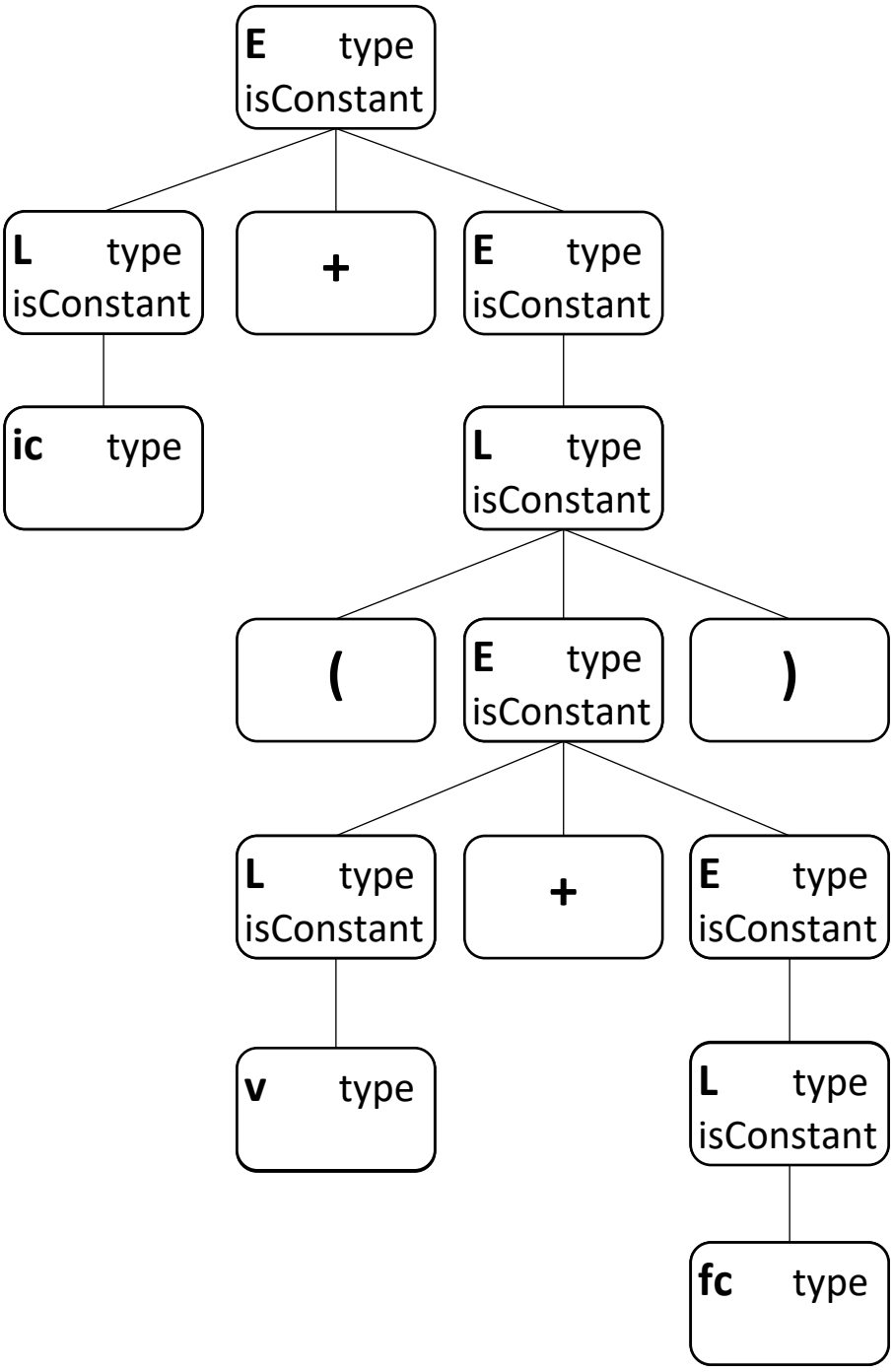
Let's examine a small attributed grammar.

production	**rules**
$E_1 \rightarrow L$ **+** E_2	E_1.type := if (L.type == **int** && E_2.type == **int**) then **int** else **float** E_1.isConstant := L.isConstant && E_2.isConstant
$E \rightarrow L$	E.type := L.type E.isConstant := L.isConstant
$L \rightarrow$ (E)	L.type := E.type L.isConstant := E.isConstant
$L \rightarrow$ **v**	L.type := v.type v.type := v.getBinding().getType() L.isConstant := **false**
$L \rightarrow$ **ic**	L.type := ic.type ic.type := **int** L.isConstant := **true**
$L \rightarrow$ **fc**	L.type := fc.type fc.type := **float** L.isConstant := **true**

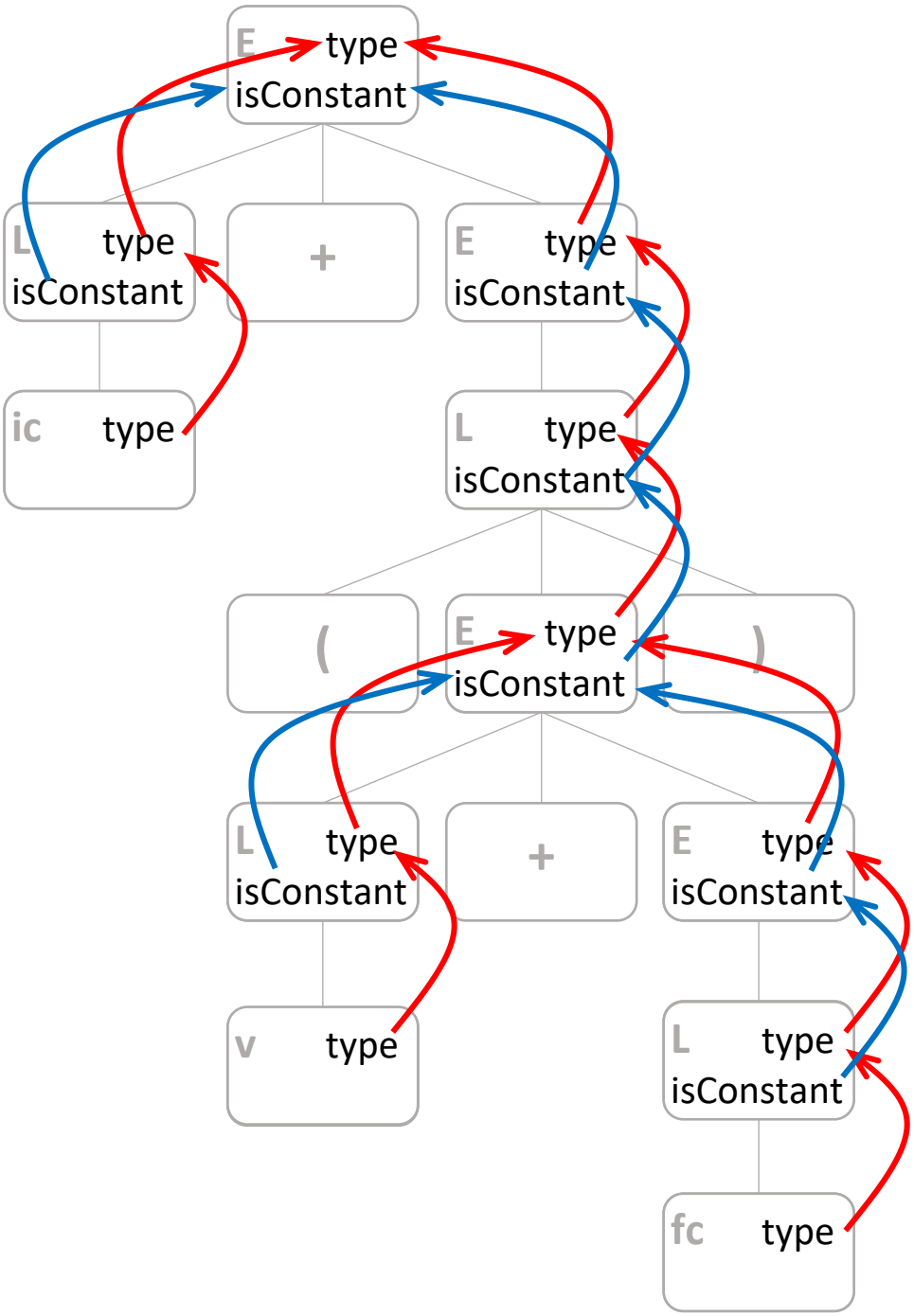
Consider parsing **ic +** (**v + fc**) where the variable **v** is of type **int**. Here's the parse tree:



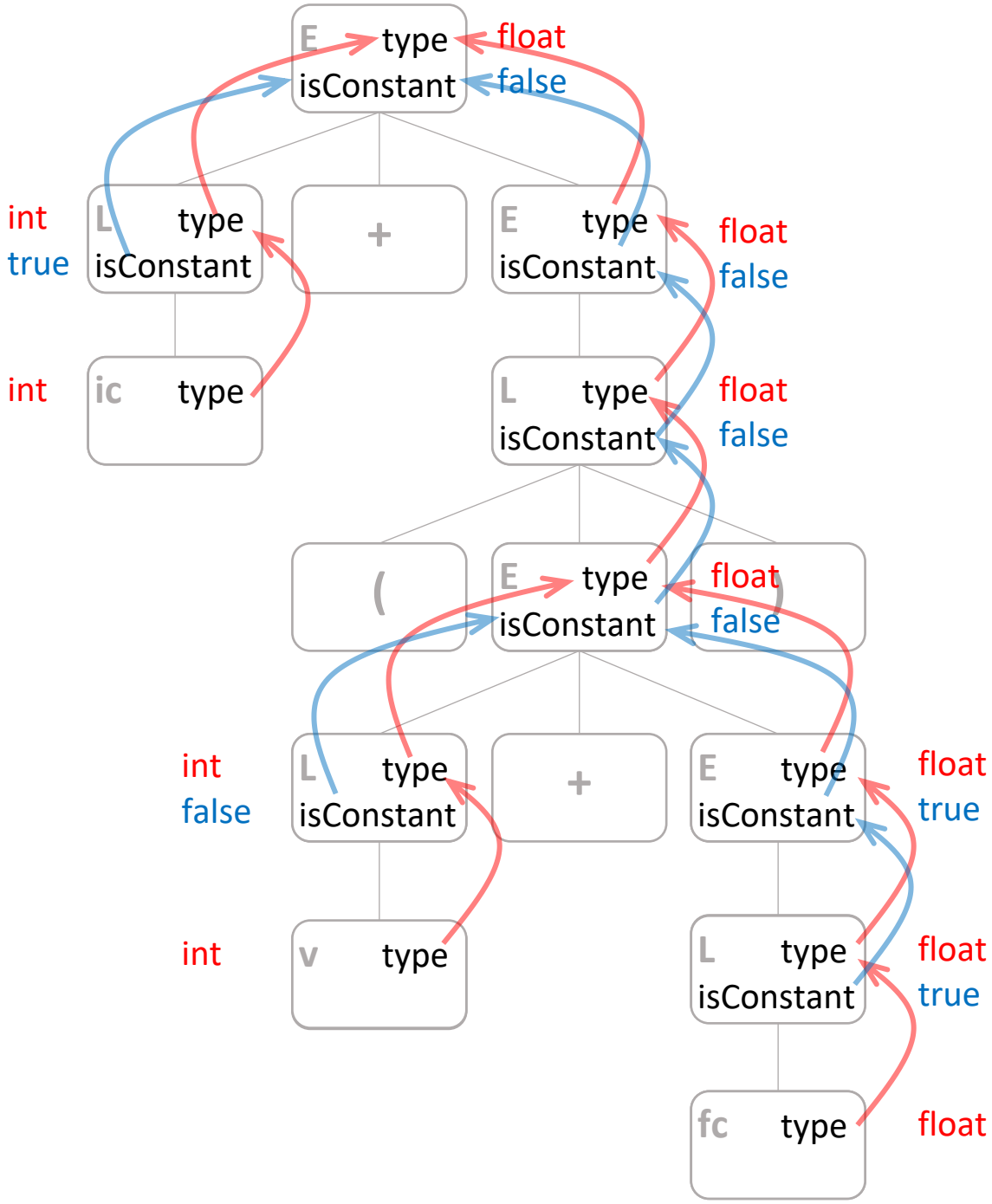
Here's the parse tree with all the attributes at each node.



Now we will draw a digraph on the attributes where an edge from a to b means that the value of a is required to compute the value of b.



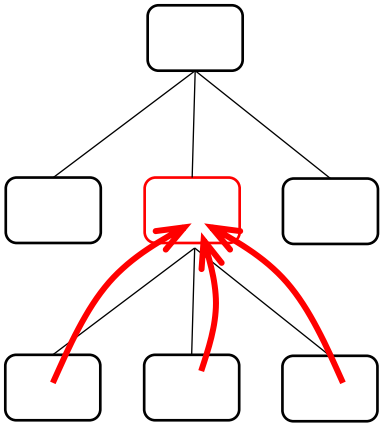
Finally, here is the diagram with all the values of the attributes computed.



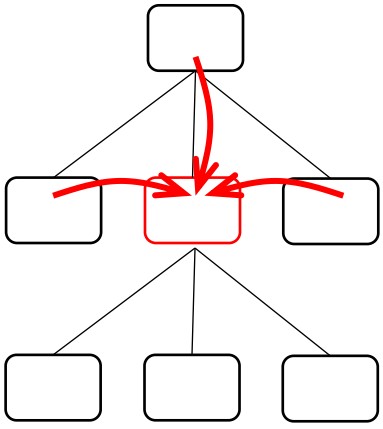
If the digraph is not a directed acyclic graph (DAG), then the attribution scheme has a problem and needs correction.

In this attributed grammar, all the flow of values goes **up** the tree. (This type of grammar is called **S-attributed.**) Not all attributed grammars are S-attributed. An example of a non-S-attributed grammar is given in **Section 4.3** of the text; **please read** this section and **Section 4.4**.

In general, if an attribute is computed from attributes of symbols below it in the tree, it is called a **synthesized** or **synthetic** attribute. If it is computed using attributes of its siblings and its parents, it is called an **inherited** attribute.

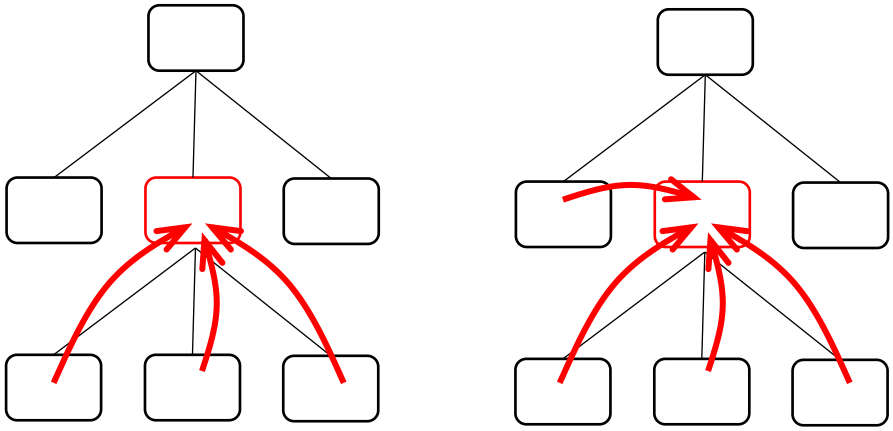


Synthesized



Inherited

An attributed grammar is called **L-attributed** if every attribute at a node is computable if one knows the value of all the attributes of the node's children and all the attributes of the node's *left* siblings. The class of L-attributed grammars contains the class of S-attributed grammars.



S-attributed L-attributed

With an L-attributed grammar, one can evaluate the attributes in a straightforward left-to-right depth first search.

If a grammar is not L-attributed, then the compiler must determine the evaluation ordering in another way; at worst it needs to do a **topological sort** on the attribute digraph.

The book talks about **ad-hoc syntax-directed translation.**

Syntax-directed translation is translation based on the grammar. Attributed grammars as thus syntax-directed translation but they are systematic rather than ad-hoc.

The Visitor mechanism in our compiler is based on a visit routine for each type of node in the AST, which is essentially each grammar symbol. It is therefore syntax-directed translation. It is ad-hoc, though it is very similar to (and can be based on) attributed grammars. It is not restricted to L-attributed grammars, though: it can pass information down the tree if necessary. It also uses some nonlocal information, in the form of symbol tables located at nodes elsewhere in the tree.

In semantic analysis, most of our attributes are types that flow up the tree (S-attributed). However, there is some more general flow going on when declaring variables: the type information must go from a right child to a left child. We may see more non-S-attributed flow of attributes in the upcoming milestones.

Attributed grammars have been used to try to automate semantic analysis. If they permit some function calls, such as to store and retrieve symbol table values, they can completely describe most semantic analysis. Oftentimes, though, semantic analysis in broken down into several subphases, and attributed grammars used for some but not all phases.

Attributed grammars can also be used for intermediate code generation. The main approach is to have an attribute called “code” for most nodes in the parse tree. Does this sound familiar? In our project, we attach the code attribute to nodes by using a hash table with nodes as keys.

The promise of automatic semantic analysis and intermediate code generation via attributed grammars has generated a wide variety of research and different schemes for evaluating attributed grammars have been proposed. These can be grouped into three major categories:

Dynamic Methods: Use the structure of a particular attributed parse tree to determine the evaluation order. One can, for instance, keep a worklist of attributes (or more properly, attributes at a particular parse tree node) to be evaluated, placing a particular attribute into the worklist as soon as all of its predecessors are evaluated. This entails checking all successors of a just-evaluated attribute to see if they are ready to enter the worklist. A related scheme is to build the entire attribute dependence graph and topologically sort it, using the topological order to evaluate attributes.

Oblivious Methods: Here the order of evaluation is independent of the attribute grammar and the parse tree. Examples include repeated left-to-right passes, repeated right-to-left passes, and alternating passes. In each pass, the evaluator evaluates any attributes that are ready. These are typically simple and easy to implement, but are not the most efficient.

Rule-based Methods: These do a static analysis of the attribute grammar to construct rules about an evaluation order, then use the parse tree to guide application of the rules. An example rule might be “visit the second child of a declaration node to compute its type before visiting the first child to set its type.” The rule tells you an order for computing attributes, and the parse tree having a declaration node guides you to use that rule at that place.

Simple cost estimation

B : Block (of statements)
A : Assignment statement
E : Expression
T : Term
F : Factor

production	**rules**
$B_0 \rightarrow B_1\ A$	B_0.cost = B_1.cost + A.cost
B → A	B.cost = A.cost
A → var = E	A.cost = Cost(store) + E.cost
$E_0 \rightarrow E_1 + T$	E_0.cost = E_1.cost + Cost(add) + T.cost
$E_0 \rightarrow E_1 - T$	E_0.cost = E_1.cost + Cost(sub) + T.cost
E → T	E.cost = T.cost
$T_0 \rightarrow T_1 * F$	T_0.cost = T_1.cost + Cost(mult) + F.cost
$T_0 \rightarrow T_1 / F$	T_0.cost = T_1.cost + Cost(div) + F.cost
T → F	T.cost = F.cost
F → (E)	F.cost = E.cost
F → num	F.cost = Cost(loadImm)
F → var	F.cost = Cost(load)

production	rules
$B_0 \rightarrow B_1\ A$	B_0.cost = B_1.cost + A.cost
$B \rightarrow A$	B.cost = A.cost
$A \rightarrow$ var = E	A.cost = Cost(store, var.type) + E.cost + PromotionCost(E.type, var.type)
$E_0 \rightarrow E_1 + T$	E_0.type = CompatibleType(E_1.type, T.type) E_0.cost = E_1.cost + Cost(add, E_0.type) + T.cost + PromotionCost(E_1.type, E_0.type) + PromotionCost(T.type, E_0.type)
$E_0 \rightarrow E_1 - T$	E_0.type = CompatibleType(E_1.type, T.type) E_0.cost = E_1.cost + Cost(sub, E_0.type) + T.cost + PromotionCost(E_1.type, E_0.type) + PromotionCost(T.type, E_0.type)
$E \rightarrow T$	E.cost = T.cost
$T_0 \rightarrow T_1 * F$	T_0.type = CompatibleType(T_1.type, F.type) T_0.cost = T_1.cost + Cost(mult, T_0.type) + F.cost+ PromotionCost(T_1.type, T_0.type) + PromotionCost(F.type, T_0.type)
$T_0 \rightarrow T_1 / F$	T_0.type = CompatibleType(T_1.type, F.type) T_0.cost = T_1.cost + Cost(div, T_0.type) + F.cost + PromotionCost(T_1.type, T_0.type) + PromotionCost(F.type, T_0.type)
$T \rightarrow F$	T.cost = F.cost
$F \rightarrow (E)$	F.cost = E.cost
$F \rightarrow$ num	F.cost = Cost(loadImm, num.type)
$F \rightarrow$ var	F.cost = Cost(load, var.type)

Now suppose instead of adding types we want to keep closer track of the number of loads necessary; variable loads take a lot of time, even if the variable's memory is in cache. When given a block like

```
x = y + 14
z = 3 * y
```

most compilers would load y only once, into a register, and then use that register twice. So we might want a rule to say that any variable is loaded only once per block. To do this, we must keep track of names of variables that have been loaded in a block. So the rule for F → var should be something like:

```
if ( var.name has not been loaded )
    then F.cost = Cost(load)
    else F.cost = 0
```

To implement the "not been loaded" test, we can add an attribute that holds the set of variables already loaded. The rules must pass this set through each assignment to each F. This is most easily done by having two sets per node—one that holds the set before traversing the subtree rooted at this node, and one that holds the set after.

This leads to the following set of rules.

production	rules
$B_0 \rightarrow B_1\ A$	B_0.cost = B_1.cost + A.cost A.before = B_1.after B_0.after = A.after
$B \rightarrow A$	B.cost = A.cost A.before = Ø B.after = A.after
$A \rightarrow$ var = E	A.cost = Cost(store) + E.cost E.before = A.before A.after = E.after
$E_0 \rightarrow E_1 + T$	E_0.cost = E_1.cost + Cost(add) + T.cost E_1.before = E_0.before T.before = E_1.after E_0.after = T.after
$E_0 \rightarrow E_1 - T$	E_0.cost = E_1.cost + Cost(sub) + T.cost E_1.before = E_0.before T.before = E_1.after E_0.after = T.after
$E \rightarrow T$	E.cost = T.cost T.before = E.before E.after = T.after
$T_0 \rightarrow T_1 * F$	T_0.cost = T_1.cost + Cost(mult) + F.cost T_1.before = T_0.before F.before = T_1.after T_0.after = F.after
$T_0 \rightarrow T_1 / F$	T_0.cost = T_1.cost + Cost(div) + F.cost T_1.before = T_0.before

	F.before = T_1.after T_0.after = F.after
T → F	T.cost = F.cost F.before = T.before T.after = F.after
F → (E)	F.cost = E.cost E.before = F.before F.after = E.after
F → num	F.cost = Cost(loadImm) F.after = F.before
F → var	**if** var.lexeme not in F.before **then** F.cost = Cost(load) F.after = F.before ∪ {var.lexeme} **else** F.cost = 0 F.after = F.before

That's getting to be a fairly complex specification, and we don't have types in there, and we haven't taken into account the effect of storing a variable on before or after. This set of rules has over three times as many as the simple rule set, and each rule must be written, understood, and evaluated. This rule set uses synthesized and inherited attributes, so a simple bottom-up evaluation of attributes will fail. Also, the rules that manipulate the

before and after attributes require a fair amount of attention—the kind of low-level detail that we would hope to avoid by using a system based on high-level specifications.

It is for these types of reasons that attributed grammars are not used for more tasks in semantic analysis and code generation. Your text has a fuller exposition on the problems with attributed grammars.