Chapter 5

Syntax-Directed Translation

This chapter develops the theme of Section 2.3: the translation of languages guided by context-free grammars. The translation techniques in this chapter will be applied in Chapter 6 to type checking and intermediate-code generation. The techniques are also useful for implementing little languages for specialized tasks; this chapter includes an example from typesetting.

We associate information with a language construct by attaching *attributes* to the grammar symbol(s) representing the construct, as discussed in Section 2.3.2. A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE	(5.1)
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel \text{'+'}$	

This production has two nonterminals, E and T; the subscript in E_1 distinguishes the occurrence of E in the production body from the occurrence of E as the head. Both E and T have a string-valued attribute *code*. The semantic rule specifies that the string $E.code$ is formed by concatenating $E_1.code$, $T.code$, and the character '+'. While the rule makes it explicit that the translation of E is built up from the translations of E_1, T, and '+', it may be inefficient to implement the translation directly by manipulating strings.

From Section 2.3.5, a syntax-directed translation scheme embeds program fragments called semantic actions within production bodies, as in

$$E \rightarrow E_1 + T \ \{ \text{ print '+' } \} \tag{5.2}$$

By convention, semantic actions are enclosed within curly braces. (If curly braces occur as grammar symbols, we enclose them within single quotes, as in

'{' and '}'.) The position of a semantic action in a production body determines the order in which the action is executed. In production (5.2), the action occurs at the end, after all the grammar symbols; in general, semantic actions may occur at any position in a production body.

Between the two notations, syntax-directed definitions can be more readable, and hence more useful for specifications. However, translation schemes can be more efficient, and hence more useful for implementations.

The most general approach to syntax-directed translation is to construct a parse tree or a syntax tree, and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree. In many cases, translation can be done during parsing, without building an explicit tree. We shall therefore study a class of syntax-directed translations called "L-attributed translations" (L for left-to-right), which encompass virtually all translations that can be performed during parsing. We also study a smaller class, called "S-attributed translations" (S for synthesized), which can be performed easily in connection with a bottom-up parse.

5.1 Syntax-Directed Definitions

A *syntax-directed definition* (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. Attributes may be of any kind: numbers, types, table references, or strings, for instance. The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

5.1.1 Inherited and Synthesized Attributes

We shall deal with two kinds of attributes for nonterminals:

1. A *synthesized attribute* for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

2. An *inherited attribute* for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

An Alternative Definition of Inherited Attributes

No additional translations are enabled if we allow an inherited attribute $B.c$ at a node N to be defined in terms of attribute values at the children of N, as well as at N itself, at its parent, and at its siblings. Such rules can be "simulated" by creating additional attributes of B, say $B.c_1, B.c_2, \ldots$. These are synthesized attributes that copy the needed attributes of the children of the node labeled B. We then compute $B.c$ as an inherited attribute, using the attributes $B.c_1, B.c_2, \ldots$ in place of attributes at the children. Such attributes are rarely needed in practice.

While we do not allow an inherited attribute at node N to be defined in terms of attribute values at the children of node N, we do allow a synthesized attribute at node N to be defined in terms of inherited attribute values at node N itself.

Terminals can have synthesized attributes, but not inherited attributes. Attributes for terminals have lexical values that are supplied by the lexical analyzer; there are no semantic rules in the SDD itself for computing the value of an attribute for a terminal.

Example 5.1: The SDD in Fig. 5.1 is based on our familiar grammar for arithmetic expressions with operators + and *. It evaluates expressions terminated by an endmarker **n**. In the SDD, each of the nonterminals has a single synthesized attribute, called *val*. We also suppose that the terminal **digit** has a synthesized attribute *lexval*, which is an integer value returned by the lexical analyzer.

	PRODUCTION	SEMANTIC RULES
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (\ E\ )$	$F.val = E.val$
7)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}$.lexval

Figure 5.1: Syntax-directed definition of a simple desk calculator

The rule for production 1, $L \to E\ \mathbf{n}$, sets $L.val$ to $E.val$, which we shall see is the numerical value of the entire expression.

Production 2, $E \to E_1 + T$, also has one rule, which computes the *val* attribute for the head E as the sum of the values at E_1 and T. At any parse-

tree node N labeled E, the value of *val* for E is the sum of the values of *val* at the children of node N labeled E and T.

Production 3, $E \rightarrow T$, has a single rule that defines the value of *val* for E to be the same as the value of *val* at the child for T. Production 4 is similar to the second production; its rule multiplies the values at the children instead of adding them. The rules for productions 5 and 6 copy values at a child, like that for the third production. Production 7 gives $F.val$ the value of a digit, that is, the numerical value of the token **digit** that the lexical analyzer returned. □

An SDD that involves only synthesized attributes is called *S-attributed*; the SDD in Fig. 5.1 has this property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

For simplicity, the examples in this section have semantic rules without side effects. In practice, it is convenient to allow SDD's to have limited side effects, such as printing the result computed by a desk calculator or interacting with a symbol table. Once the order of evaluation of attributes is discussed in Section 5.2, we shall allow semantic rules to compute arbitrary functions, possibly involving side effects.

An S-attributed SDD can be implemented naturally in conjunction with an LR parser. In fact, the SDD in Fig. 5.1 mirrors the Yacc program of Fig. 4.58, which illustrates translation during LR parsing. The difference is that, in the rule for production 1, the Yacc program prints the value $E.val$ as a side effect, instead of defining the attribute $L.val$.

An SDD without side effects is sometimes called an *attribute grammar*. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

5.1.2 Evaluating an SDD at the Nodes of a Parse Tree

To visualize the translation specified by an SDD, it helps to work with parse trees, even though a translator need not actually build a parse tree. Imagine therefore that the rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an *annotated parse tree*.

How do we construct an annotated parse tree? In what order do we evaluate attributes? Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends. For example, if all attributes are synthesized, as in Example 5.1, then we must evaluate the *val* attributes at all of the children of a node before we can evaluate the *val* attribute at the node itself.

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree; the evaluation of S-attributed definitions is discussed in Section 5.2.3.

For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order in which to evaluate attributes at nodes. For instance, consider nonterminals A and B, with synthesized and inherited attributes $A.s$ and $B.i$, respectively, along with the production and rules

PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i;$
	$B.i = A.s + 1$

These rules are circular; it is impossible to evaluate either $A.s$ at a node N or $B.i$ at the child of N without first evaluating the other. The circular dependency of $A.s$ and $B.i$ at some pair of nodes in a parse tree is suggested by Fig. 5.2.

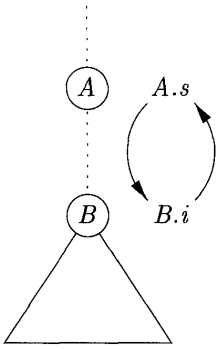

Figure 5.2: The circular dependency of $A.s$ and $B.i$ on one another

It is computationally difficult to determine whether or not there exist any circularities in any of the parse trees that a given SDD could have to translate.[1] Fortunately, there are useful subclasses of SDD's that are sufficient to guarantee that an order of evaluation exists, as we shall see in Section 5.2.

Example 5.2: Figure 5.3 shows an annotated parse tree for the input string $3 * 5 + 4$ **n**, constructed using the grammar and rules of Fig. 5.1. The values of *lexval* are presumed supplied by the lexical analyzer. Each of the nodes for the nonterminals has attribute *val* computed in a bottom-up order, and we see the resulting values associated with each node. For instance, at the node with a child labeled $*$, after computing $T.val = 3$ and $F.val = 5$ at its first and third children, we apply the rule that says $T.val$ is the product of these two values, or 15. □

Inherited attributes are useful when the structure of a parse tree does not "match" the abstract syntax of the source code. The next example shows how inherited attributes can be used to overcome such a mismatch due to a grammar designed for parsing rather than translation.

[1] Without going into details, while the problem is decidable, it cannot be solved by a polynomial-time algorithm, even if $\mathcal{P} = \mathcal{NP}$, since it has exponential time complexity.

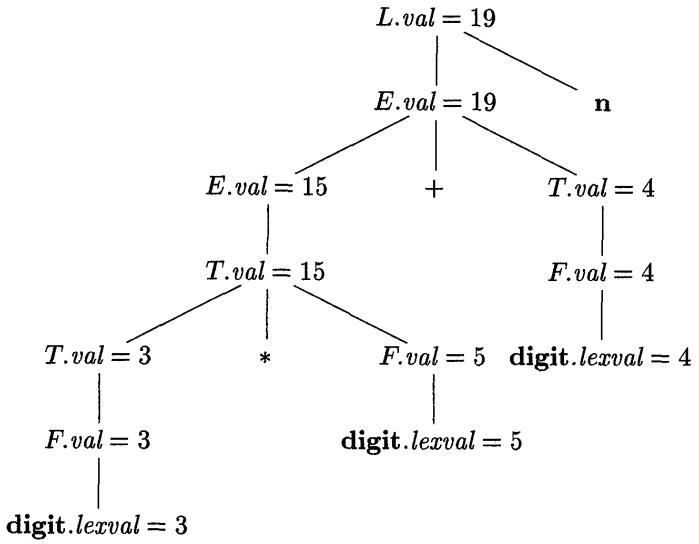

Figure 5.3: Annotated parse tree for $3 * 5 + 4\,\mathbf{n}$

Example 5.3: The SDD in Fig. 5.4 computes terms like $3 * 5$ and $3 * 5 * 7$. The top-down parse of input $3 * 5$ begins with the production $T \rightarrow F\,T'$. Here, F generates the digit 3, but the operator $*$ is generated by T'. Thus, the left operand 3 appears in a different subtree of the parse tree from $*$. An inherited attribute will therefore be used to pass the operand to the operator.

The grammar in this example is an excerpt from a non-left-recursive version of the familiar expression grammar; we used such a grammar as a running example to illustrate top-down parsing in Section 4.4.

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow * F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Figure 5.4: An SDD based on a grammar suitable for top-down parsing

Each of the nonterminals T and F has a synthesized attribute *val*; the terminal **digit** has a synthesized attribute *lexval*. The nonterminal T' has two attributes: an inherited attribute *inh* and a synthesized attribute *syn*.

The semantic rules are based on the idea that the left operand of the operator $*$ is inherited. More precisely, the head T' of the production $T' \rightarrow * \ F \ T_1'$ inherits the left operand of $*$ in the production body. Given a term $x * y * z$, the root of the subtree for $* y * z$ inherits x. Then, the root of the subtree for $* z$ inherits the value of $x * y$, and so on, if there are more factors in the term. Once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.

To see how the semantic rules are used, consider the annotated parse tree for $3 * 5$ in Fig. 5.5. The leftmost leaf in the parse tree, labeled **digit**, has attribute value $lexval = 3$, where the 3 is supplied by the lexical analyzer. Its parent is for production 4, $F \rightarrow$ **digit**. The only semantic rule associated with this production defines $F.val =$ **digit**.$lexval$, which equals 3.

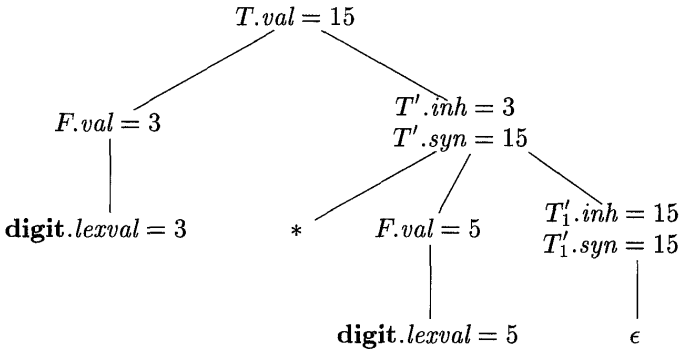

Figure 5.5: Annotated parse tree for $3 * 5$

At the second child of the root, the inherited attribute $T'.inh$ is defined by the semantic rule $T'.inh = F.val$ associated with production 1. Thus, the left operand, 3, for the $*$ operator is passed from left to right across the children of the root.

The production at the node for T' is $T' \rightarrow * F T_1'$. (We retain the subscript 1 in the annotated parse tree to distinguish between the two nodes for T'.) The inherited attribute $T_1'.inh$ is defined by the semantic rule $T_1'.inh = T'.inh \times F.val$ associated with production 2.

With $T'.inh = 3$ and $F.val = 5$, we get $T_1'.inh = 15$. At the lower node for T_1', the production is $T' \rightarrow \epsilon$. The semantic rule $T'.syn = T'.inh$ defines $T_1'.syn = 15$. The *syn* attributes at the nodes for T' pass the value 15 up the tree to the node for T, where $T.val = 15$. □

5.1.3 Exercises for Section 5.1

Exercise 5.1.1: For the SDD of Fig. 5.1, give annotated parse trees for the following expressions:

a) $(3 + 4) * (5 + 6)$ **n**.

b) $1 * 2 * 3 * (4 + 5)$ **n**.

c) $(9 + 8 * (7 + 6) + 5) * 4$ **n**.

Exercise 5.1.2: Extend the SDD of Fig. 5.4 to handle expressions as in Fig. 5.1.

Exercise 5.1.3: Repeat Exercise 5.1.1, using your SDD from Exercise 5.1.2.

5.2 Evaluation Orders for SDD's

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

In this section, in addition to dependency graphs, we define two important classes of SDD's: the "S-attributed" and the more general "L-attributed" SDD's. The translations specified by these two classes fit well with the parsing methods we have studied, and most translations encountered in practice can be written to conform to the requirements of at least one of these classes.

5.2.1 Dependency Graphs

A *dependency graph* depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules. In more detail:

- For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.
- Suppose that a semantic rule associated with a production p defines the value of synthesized attribute $A.b$ in terms of the value of $X.c$ (the rule may define $A.b$ in terms of other attributes in addition to $X.c$). Then, the dependency graph has an edge from $X.c$ to $A.b$. More precisely, at every node N labeled A where production p is applied, create an edge to attribute b at N, from the attribute c at the child of N corresponding to this instance of the symbol X in the body of the production.[2]
- Suppose that a semantic rule associated with a production p defines the value of inherited attribute $B.c$ in terms of the value of $X.a$. Then, the dependency graph has an edge from $X.a$ to $B.c$. For each node N labeled B that corresponds to an occurrence of this B in the body of production p, create an edge to attribute c at N from the attribute a at the node M

[2]Since a node N can have several children labeled X, we again assume that subscripts distinguish among uses of the same symbol at different places in the production.

that corresponds to this occurrence of X. Note that M could be either the parent or a sibling of N.

Example 5.4: Consider the following production and rule:

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

At every node N labeled E, with children corresponding to the body of this production, the synthesized attribute *val* at N is computed using the values of *val* at the two children, labeled E and T. Thus, a portion of the dependency graph for every parse tree in which this production is used looks like Fig. 5.6. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid. □

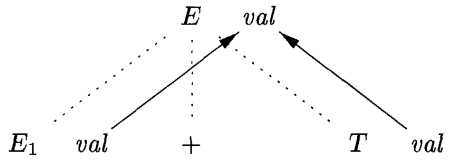

Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

Example 5.5: An example of a complete dependency graph appears in Fig. 5.7. The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree in Fig. 5.5.

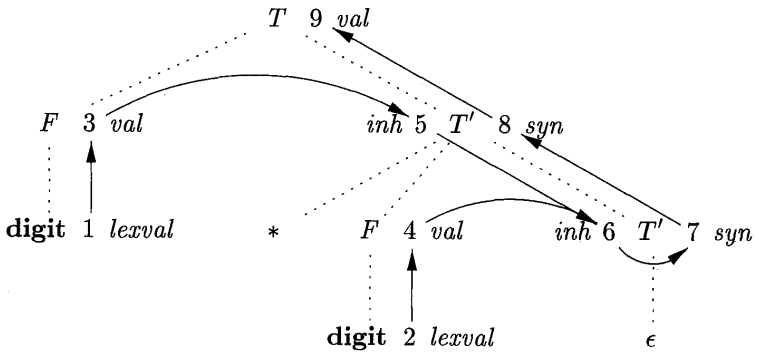

Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

Nodes 1 and 2 represent the attribute *lexval* associated with the two leaves labeled **digit**. Nodes 3 and 4 represent the attribute *val* associated with the two nodes labeled F. The edges to node 3 from 1 and to node 4 from 2 result

from the semantic rule that defines $F.val$ in terms of **digit**.*lexval*. In fact, $F.val$ equals **digit**.*lexval*, but the edge represents dependence, not equality.

Nodes 5 and 6 represent the inherited attribute $T'.inh$ associated with each of the occurrences of nonterminal T'. The edge to 5 from 3 is due to the rule $T'.inh = F.val$, which defines $T'.inh$ at the right child of the root from $F.val$ at the left child. We see edges to 6 from node 5 for $T'.inh$ and from node 4 for $F.val$, because these values are multiplied to evaluate the attribute inh at node 6.

Nodes 7 and 8 represent the synthesized attribute syn associated with the occurrences of T'. The edge to node 7 from 6 is due to the semantic rule $T'.syn = T'.inh$ associated with production 3 in Fig. 5.4. The edge to node 8 from 7 is due to a semantic rule associated with production 2.

Finally, node 9 represents the attribute $T.val$. The edge to 9 from 8 is due to the semantic rule, $T.val = T'.syn$, associated with production 1. □

5.2.2 Ordering the Evaluation of Attributes

The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of a parse tree. If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N. Thus, the only allowable orders of evaluation are those sequences of nodes $N_1, N_2, \ldots, N_k$ such that if there is an edge of the dependency graph from N_i to N_j, then $i < j$. Such an ordering embeds a directed graph into a linear order, and is called a *topological sort* of the graph.

If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort. To see why, since there are no cycles, we can surely find a node with no edge entering. For if there were no such node, we could proceed from predecessor to predecessor until we came back to some node we had already seen, yielding a cycle. Make this node the first in the topological order, remove it from the dependency graph, and repeat the process on the remaining nodes.

Example 5.6: The dependency graph of Fig. 5.7 has no cycles. One topological sort is the order in which the nodes have already been numbered: $1, 2, \ldots, 9$. Notice that every edge of the graph goes from a node to a higher-numbered node, so this order is surely a topological sort. There are other topological sorts as well, such as $1, 3, 5, 2, 4, 6, 7, 8, 9$. □

5.2.3 S-Attributed Definitions

As mentioned earlier, given an SDD, it is very hard to tell whether there exist any parse trees whose dependency graphs have cycles. In practice, translations can be implemented using classes of SDD's that guarantee an evaluation order,

since they do not permit dependency graphs with cycles. Moreover, the two classes introduced in this section can be implemented efficiently in connection with top-down or bottom-up parsing.

The first class is defined as follows:

- An SDD is *S-attributed* if every attribute is synthesized.

Example 5.7: The SDD of Fig. 5.1 is an example of an S-attributed definition. Each attribute, *L.val*, *E.val*, *T.val*, and *F.val* is synthesized. □

When an SDD is S-attributed, we can evaluate its attributes in any bottom-up order of the nodes of the parse tree. It is often especially simple to evaluate the attributes by performing a postorder traversal of the parse tree and evaluating the attributes at a node N when the traversal leaves N for the last time. That is, we apply the function *postorder*, defined below, to the root of the parse tree (see also the box "Preorder and Postorder Traversals" in Section 2.3.4):

```
postorder(N) {
    for ( each child C of N, from the left ) postorder(C);
    evaluate the attributes associated with node N;
}
```

S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal. Specifically, postorder corresponds exactly to the order in which an LR parser reduces a production body to its head. This fact will be used in Section 5.4.2 to evaluate synthesized attributes and store them on the stack during LR parsing, without creating the tree nodes explicitly.

5.2.4 L-Attributed Definitions

The second class of SDD's is called *L-attributed definitions*. The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed"). More precisely, each attribute must be either

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1 X_2 \cdots X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:
   (a) Inherited attributes associated with the head A.
   (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i.

(c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.

Example 5.8: The SDD in Fig. 5.4 is L-attributed. To see why, consider the semantic rules for inherited attributes, which are repeated here for convenience:

PRODUCTION	SEMANTIC RULE
$T \rightarrow F\ T'$	$T'.inh = F.val$
$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$

The first of these rules defines the inherited attribute $T'.inh$ using only $F.val$, and F appears to the left of T' in the production body, as required. The second rule defines $T'_1.inh$ using the inherited attribute $T'.inh$ associated with the head, and $F.val$, where F appears to the left of T'_1 in the production body.

In each of these cases, the rules use information "from above or from the left," as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed. □

Example 5.9: Any SDD containing the following production and rules cannot be L-attributed:

PRODUCTION	SEMANTIC RULES
$A \rightarrow B\ C$	$A.s = B.b;$
	$B.i = f(C.c, A.s)$

The first rule, $A.s = B.b$, is a legitimate rule in either an S-attributed or L-attributed SDD. It defines a synthesized attribute $A.s$ in terms of an attribute at a child (that is, a symbol within the production body).

The second rule defines an inherited attribute $B.i$, so the entire SDD cannot be S-attributed. Further, although the rule is legal, the SDD cannot be L-attributed, because the attribute $C.c$ is used to help define $B.i$, and C is to the right of B in the production body. While attributes at siblings in a parse tree may be used in L-attributed SDD's, they must be to the left of the symbol whose attribute is being defined. □

5.2.5 Semantic Rules with Controlled Side Effects

In practice, translations involve side effects: a desk calculator might print a result; a code generator might enter the type of an identifier into a symbol table. With SDD's, we strike a balance between attribute grammars and translation schemes. Attribute grammars have no side effects and allow any evaluation order consistent with the dependency graph. Translation schemes impose left-to-right evaluation and allow semantic actions to contain any program fragment; translation schemes are discussed in Section 5.4.

We shall control side effects in SDD's in one of the following ways:

- Permit incidental side effects that do not constrain attribute evaluation. In other words, permit side effects when attribute evaluation based on any topological sort of the dependency graph produces a "correct" translation, where "correct" depends on the application.
- Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order. The constraints can be thought of as implicit edges added to the dependency graph.

As an example of an incidental side effect, let us modify the desk calculator of Example 5.1 to print a result. Instead of the rule $L.val = E.val$, which saves the result in the synthesized attribute $L.val$, consider:

	PRODUCTION	SEMANTIC RULE
1)	$L \rightarrow E$ **n**	$print(E.val)$

Semantic rules that are executed for their side effects, such as $print(E.val)$, will be treated as the definitions of dummy synthesized attributes associated with the head of the production. The modified SDD produces the same translation under any topological sort, since the print statement is executed at the end, after the result is computed into $E.val$.

Example 5.10: The SDD in Fig. 5.8 takes a simple declaration D consisting of a basic type T followed by a list L of identifiers. T can be **int** or **float**. For each identifier on the list, the type is entered into the symbol-table entry for the identifier. We assume that entering the type for one identifier does not affect the symbol-table entry for any other identifier. Thus, entries can be updated in any order. This SDD does not check whether an identifier is declared more than once; it can be modified to do so.

	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow$ **int**	$T.type =$ integer
3)	$T \rightarrow$ **float**	$T.type =$ float
4)	$L \rightarrow L_1$, **id**	$L_1.inh = L.inh$
		$addType(\textbf{id}.entry, L.inh)$
5)	$L \rightarrow$ **id**	$addType(\textbf{id}.entry, L.inh)$

Figure 5.8: Syntax-directed definition for simple type declarations

Nonterminal D represents a declaration, which, from production 1, consists of a type T followed by a list L of identifiers. T has one attribute, $T.type$, which is the type in the declaration D. Nonterminal L also has one attribute, which we call *inh* to emphasize that it is an inherited attribute. The purpose of $L.inh$

is to pass the declared type down the list of identifiers, so that it can be added to the appropriate symbol-table entries.

Productions 2 and 3 each evaluate the synthesized attribute $T.type$, giving it the appropriate value, integer or float. This type is passed to the attribute $L.inh$ in the rule for production 1. Production 4 passes $L.inh$ down the parse tree. That is, the value $L_1.inh$ is computed at a parse-tree node by copying the value of $L.inh$ from the parent of that node; the parent corresponds to the head of the production.

Productions 4 and 5 also have a rule in which a function *addType* is called with two arguments:

1. **id**.*entry*, a lexical value that points to a symbol-table object, and

2. $L.inh$, the type being assigned to every identifier on the list.

We suppose that function *addType* properly installs the type $L.inh$ as the type of the represented identifier.

A dependency graph for the input string **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$ appears in Fig. 5.9. Numbers 1 through 10 represent the nodes of the dependency graph. Nodes 1, 2, and 3 represent the attribute *entry* associated with each of the leaves labeled **id**. Nodes 6, 8, and 10 are the dummy attributes that represent the application of the function *addType* to a type and one of these *entry* values.

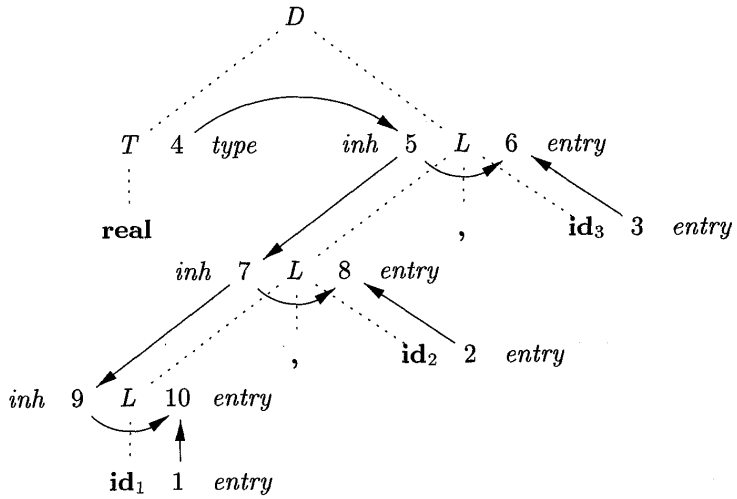

Figure 5.9: Dependency graph for a declaration **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

Node 4 represents the attribute $T.type$, and is actually where attribute evaluation begins. This type is then passed to nodes 5, 7, and 9 representing $L.inh$ associated with each of the occurrences of the nonterminal L. □

5.2.6 Exercises for Section 5.2

Exercise 5.2.1: What are all the topological sorts for the dependency graph of Fig. 5.7?

Exercise 5.2.2: For the SDD of Fig. 5.8, give annotated parse trees for the following expressions:

a) `int a, b, c.`

b) `float w, x, y, z.`

Exercise 5.2.3: Suppose that we have a production $A \rightarrow BCD$. Each of the four nonterminals A, B, C, and D have two attributes: s is a synthesized attribute, and i is an inherited attribute. For each of the sets of rules below, tell whether (*i*) the rules are consistent with an S-attributed definition (*ii*) the rules are consistent with an L-attributed definition, and (*iii*) whether the rules are consistent with any evaluation order at all?

a) $A.s = B.i + C.s$.

b) $A.s = B.i + C.s$ and $D.i = A.i + B.s$.

c) $A.s = B.s + D.s$.

! d) $A.s = D.i$, $B.i = A.s + C.s$, $C.i = B.s$, and $D.i = B.i + C.i$.

! **Exercise 5.2.4:** This grammar generates binary numbers with a "decimal" point:

$$S \rightarrow L \,.\, L \mid L$$
$$L \rightarrow L\ B \mid B$$
$$B \rightarrow 0 \mid 1$$

Design an L-attributed SDD to compute $S.val$, the decimal-number value of an input string. For example, the translation of string `101.101` should be the decimal number 5.625. *Hint*: use an inherited attribute $L.side$ that tells which side of the decimal point a bit is on.

!! **Exercise 5.2.5:** Design an S-attributed SDD for the grammar and translation described in Exercise 5.2.4.

!! **Exercise 5.2.6:** Implement Algorithm 3.23, which converts a regular expression into a nondeterministic finite automaton, by an L-attributed SDD on a top-down parsable grammar. Assume that there is a token **char** representing any character, and that **char**.*lexval* is the character it represents. You may also assume the existence of a function *new*() that returns a new state, that is, a state never before returned by this function. Use any convenient notation to specify the transitions of the NFA.

5.3 Applications of Syntax-Directed Translation

The syntax-directed translation techniques in this chapter will be applied in Chapter 6 to type checking and intermediate-code generation. Here, we consider selected examples to illustrate some representative SDD's.

The main application in this section is the construction of syntax trees. Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree. (Chapter 6 also discusses approaches to intermediate-code generation that apply an SDD without ever constructing a tree explicitly.)

We consider two SDD's for constructing syntax trees for expressions. The first, an S-attributed definition, is suitable for use during bottom-up parsing. The second, L-attributed, is suitable for use during top-down parsing.

The final example of this section is an L-attributed definition that deals with basic and array types.

5.3.1 Construction of Syntax Trees

As discussed in Section 2.8.2, each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression $E_1 + E_2$ has label $+$ and two children representing the subexpressions E_1 and E_2.

We shall implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an *op* field that is the label of the node. The objects will have additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function *Leaf*(*op*, *val*) creates a leaf object. Alternatively, if nodes are viewed as records, then *Leaf* returns a pointer to a new record for a leaf.

- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function *Node* takes two or more arguments: $Node(op, c_1, c_2, \ldots, c_k)$ creates an object with first field *op* and k additional fields for the k children $c_1, \ldots, c_k$.

Example 5.11: The S-attributed definition in Fig. 5.10 constructs syntax trees for a simple expression grammar involving only the binary operators $+$ and $-$. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute *node*, which represents a node of the syntax tree.

Every time the first production $E \rightarrow E_1 + T$ is used, its rule creates a node with '$+$' for *op* and two children, $E_1.node$ and $T.node$, for the subexpressions. The second production has a similar rule.

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**, **id**.$entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**, **num**.$val)$

Figure 5.10: Constructing syntax trees for simple expressions

For production 3, $E \rightarrow T$, no node is created, since $E.node$ is the same as $T.node$. Similarly, no node is created for production 4, $T \rightarrow (E)$. The value of $T.node$ is the same as $E.node$, since parentheses are used only for grouping; they influence the structure of the parse tree and the syntax tree, but once their job is done, there is no further need to retain them in the syntax tree.

The last two T-productions have a single terminal on the right. We use the constructor *Leaf* to create a suitable node, which becomes the value of $T.node$.

Figure 5.11 shows the construction of a syntax tree for the input $a - 4 + c$. The nodes of the syntax tree are shown as records, with the *op* field first. Syntax-tree edges are now shown as solid lines. The underlying parse tree, which need not actually be constructed, is shown with dotted edges. The third type of line, shown dashed, represents the values of $E.node$ and $T.node$; each line points to the appropriate syntax-tree node.

At the bottom we see leaves for a, 4 and c, constructed by *Leaf*. We suppose that the lexical value **id**.*entry* points into the symbol table, and the lexical value **num**.*val* is the numerical value of a constant. These leaves, or pointers to them, become the value of $T.node$ at the three parse-tree nodes labeled T, according to rules 5 and 6. Note that by rule 3, the pointer to the leaf for a is also the value of $E.node$ for the leftmost E in the parse tree.

Rule 2 causes us to create a node with *op* equal to the minus sign and pointers to the first two leaves. Then, rule 1 produces the root node of the syntax tree by combining the node for $-$ with the third leaf.

If the rules are evaluated during a postorder traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown in Fig. 5.12 ends with p_5 pointing to the root of the constructed syntax tree. □

With a grammar designed for top-down parsing, the same syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees.

Example 5.12: The L-attributed definition in Fig. 5.13 performs the same translation as the S-attributed definition in Fig. 5.10. The attributes for the grammar symbols E, T, **id**, and **num** are as discussed in Example 5.11.

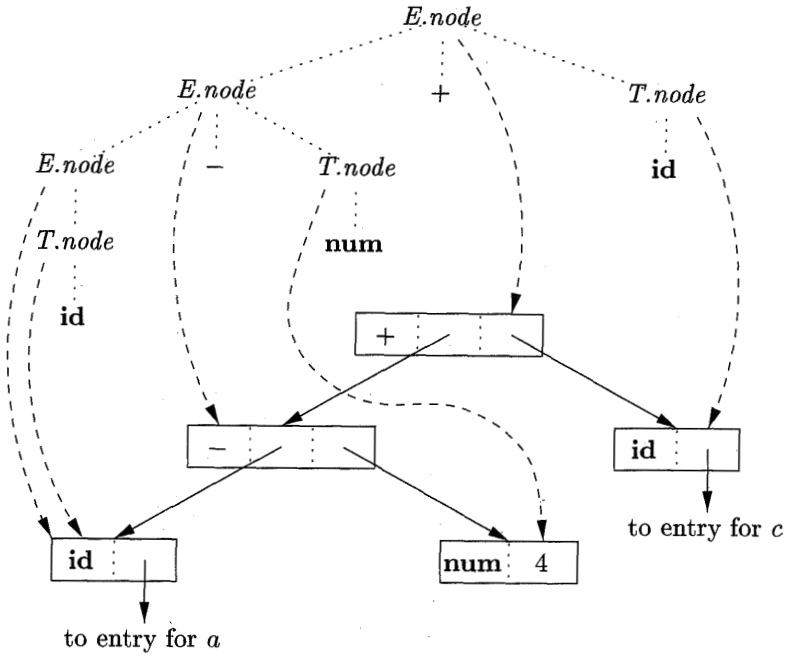

Figure 5.11: Syntax tree for $a - 4 + c$

1) $p_1 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}a})$;
2) $p_2 = \textbf{new}\ \mathit{Leaf}(\textbf{num}, 4)$;
3) $p_3 = \textbf{new}\ \mathit{Node}('-', p_1, p_2)$;
4) $p_4 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}c})$;
5) $p_5 = \textbf{new}\ \mathit{Node}('+', p_3, p_4)$;

Figure 5.12: Steps in the construction of the syntax tree for $a - 4 + c$

The rules for building syntax trees in this example are similar to the rules for the desk calculator in Example 5.3. In the desk-calculator example, a term $x * y$ was evaluated by passing x as an inherited attribute, since x and $* y$ appeared in different portions of the parse tree. Here, the idea is to build a syntax tree for $x + y$ by passing x as an inherited attribute, since x and $+ y$ appear in different subtrees. Nonterminal E' is the counterpart of nonterminal T' in Example 5.3. Compare the dependency graph for $a - 4 + c$ in Fig. 5.14 with that for $3 * 5$ in Fig. 5.7.

Nonterminal E' has an inherited attribute *inh* and a synthesized attribute *syn*. Attribute $E'.inh$ represents the partial syntax tree constructed so far. Specifically, it represents the root of the tree for the prefix of the input string that is to the left of the subtree for E'. At node 5 in the dependency graph in Fig. 5.14, $E'.inh$ denotes the root of the partial syntax tree for the identifier a; that is, the leaf for a. At node 6, $E'.inh$ denotes the root for the partial syntax

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh =$ **new** $Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh =$ **new** $Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (\ E\ )$	$T.node = E.node$
6)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**$,$ **id**$.entry)$
7)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**$,$ **num**$.val)$

Figure 5.13: Constructing syntax trees during top-down parsing

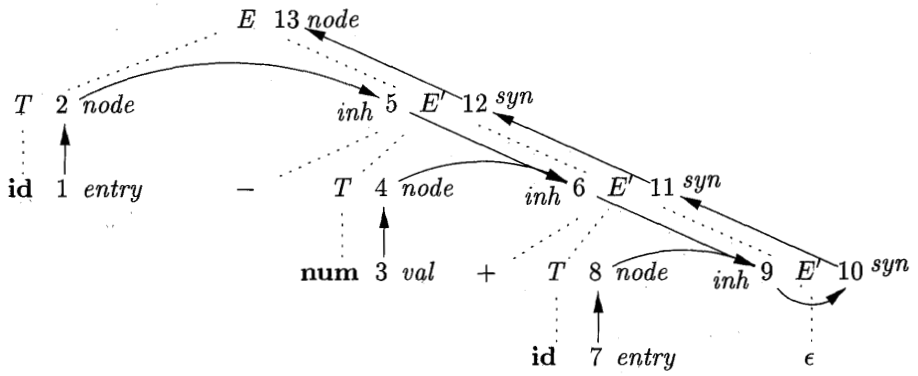

Figure 5.14: Dependency graph for $a - 4 + c$, with the SDD of Fig. 5.13

tree for the input $a - 4$. At node 9, $E'.inh$ denotes the syntax tree for $a - 4 + c$.

Since there is no more input, at node 9, $E'.inh$ points to the root of the entire syntax tree. The *syn* attributes pass this value back up the parse tree until it becomes the value of $E.node$. Specifically, the attribute value at node 10 is defined by the rule $E'.syn = E'.inh$ associated with the production $E' \rightarrow \epsilon$. The attribute value at node 11 is defined by the rule $E'.syn = E'_1.syn$ associated with production 2 in Fig. 5.13. Similar rules define the attribute values at nodes 12 and 13. □

5.3.2 The Structure of a Type

Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input; attributes can then be used to carry informa-

tion from one part of the parse tree to another. The next example shows how a mismatch in structure can be due to the design of the language, and not due to constraints imposed by the parsing method.

Example 5.13: In C, the type **int** [2][3] can be read as, "array of 2 arrays of 3 integers." The corresponding type expression *array*(2, *array*(3, *integer*)) is represented by the tree in Fig. 5.15. The operator *array* takes two parameters, a number and a type. If types are represented by trees, then this operator returns a tree node labeled *array* with two children for a number and a type.

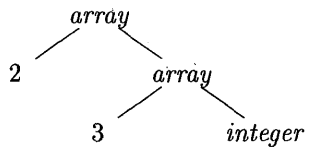

Figure 5.15: Type expression for **int**[2][3]

With the SDD in Fig. 5.16, nonterminal T generates either a basic type or an array type. Nonterminal B generates one of the basic types **int** and **float**. T generates a basic type when T derives $B\,C$ and C derives ϵ. Otherwise, C generates array components consisting of a sequence of integers, each integer surrounded by brackets.

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array($**num**$.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

The nonterminals B and T have a synthesized attribute t representing a type. The nonterminal C has two attributes: an inherited attribute b and a synthesized attribute t. The inherited b attributes pass a basic type down the tree, and the synthesized t attributes accumulate the result.

An annotated parse tree for the input string **int**[2][3] is shown in Fig. 5.17. The corresponding type expression in Fig. 5.15 is constructed by passing the type *integer* from B, down the chain of C's through the inherited attributes b. The array type is synthesized up the chain of C's through the attributes t.

In more detail, at the root for $T \rightarrow B\ C$, nonterminal C inherits the type from B, using the inherited attribute $C.b$. At the rightmost node for C, the

production is $C \rightarrow \epsilon$, so $C.t$ equals $C.b$. The semantic rules for the production $C \rightarrow [\ \mathbf{num}\ ]\ C_1$ form $C.t$ by applying the operator *array* to the operands $\mathbf{num}.val$ and $C_1.t$. □

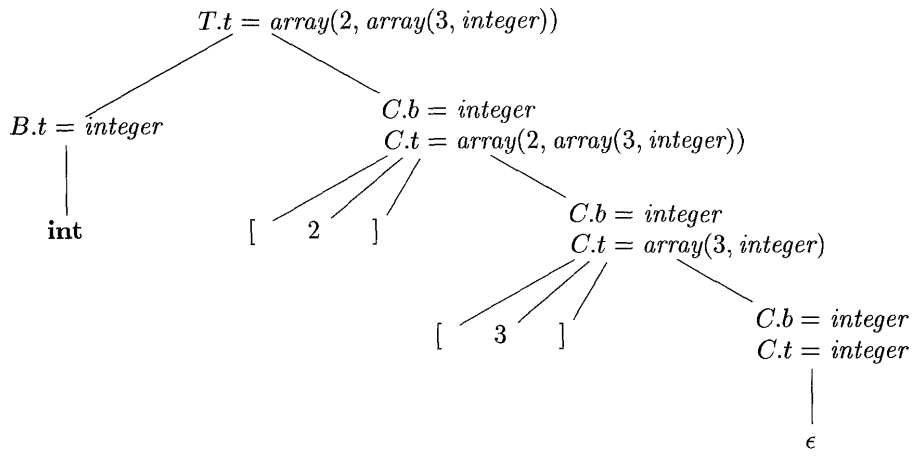

Figure 5.17: Syntax-directed translation of array types

5.3.3 Exercises for Section 5.3

Exercise 5.3.1: Below is a grammar for expressions involving operator + and integer or floating-point operands. Floating-point numbers are distinguished by having a decimal point.

$$E \rightarrow E\ +\ T \mid T$$
$$T \rightarrow \mathbf{num}\ .\ \mathbf{num} \mid \mathbf{num}$$

a) Give an SDD to determine the type of each term T and expression E.

b) Extend your SDD of (a) to translate expressions into postfix notation. Use the unary operator **intToFloat** to turn an integer into an equivalent float.

! **Exercise 5.3.2:** Give an SDD to translate infix expressions with + and * into equivalent expressions without redundant parentheses. For example, since both operators associate from the left, and * takes precedence over +, $((a*(b+c))*(d))$ translates into $a * (b + c) * d$.

! **Exercise 5.3.3:** Give an SDD to differentiate expressions such as $x * (3 * x + x * x)$ involving the operators + and *, the variable x, and constants. Assume that no simplification occurs, so that, for example, $3 * x$ will be translated into $3 * 1 + 0 * x$.