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)
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$	

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)	$F \to$ **digit**	$F.val =$ **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.

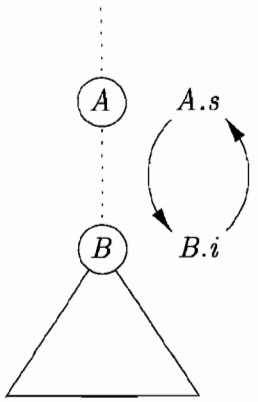

Figure 5.2: The circular dependency of $A.s$ and $B.i$ on one another

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.

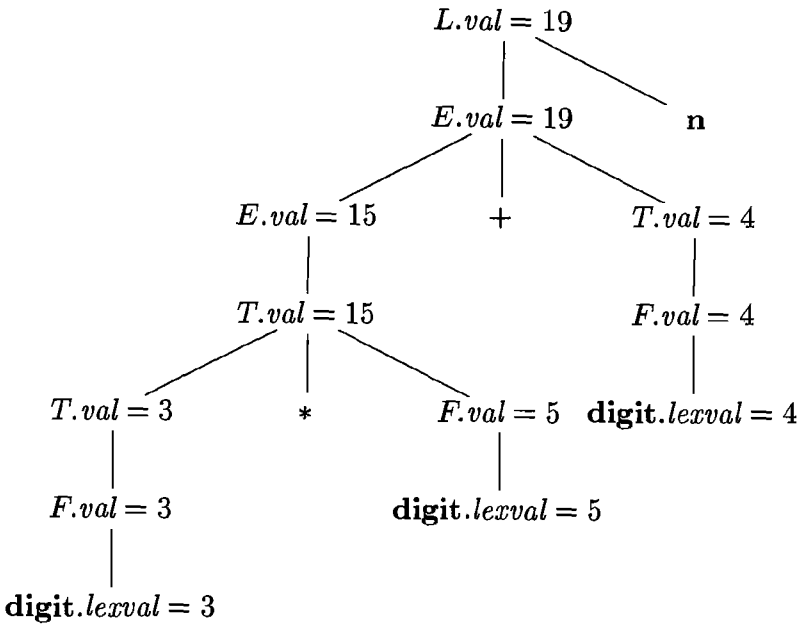

Figure 5.3: Annotated parse tree for $3 * 5 + 4\,\mathbf{n}$

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 $T \to F\,T'$. 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)	$T \to F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to * F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

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.

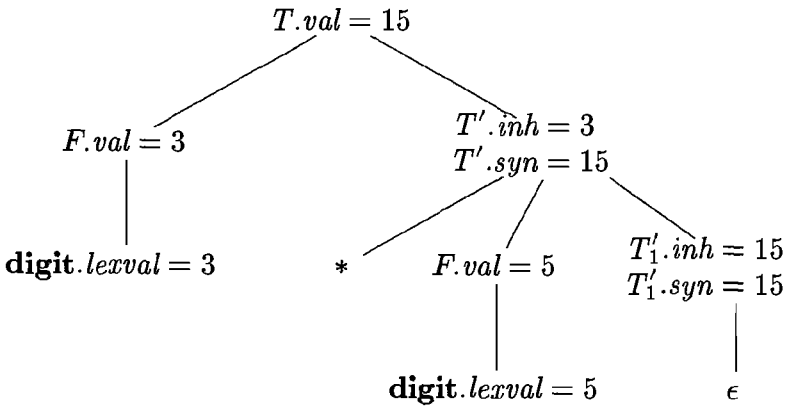

Figure 5.5: Annotated parse tree for $3 * 5$

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. □

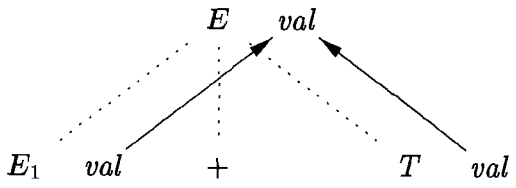

Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

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.

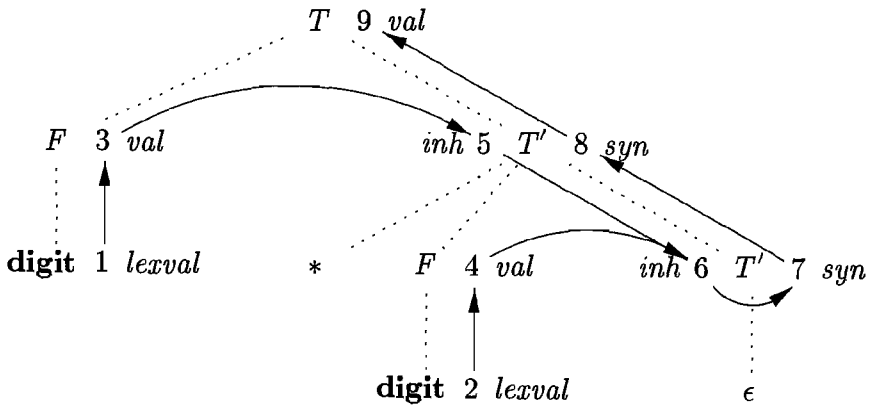

Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

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(**id**.*entry*, $L.inh$)
5)	$L \rightarrow$ **id**	*addType*(**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.

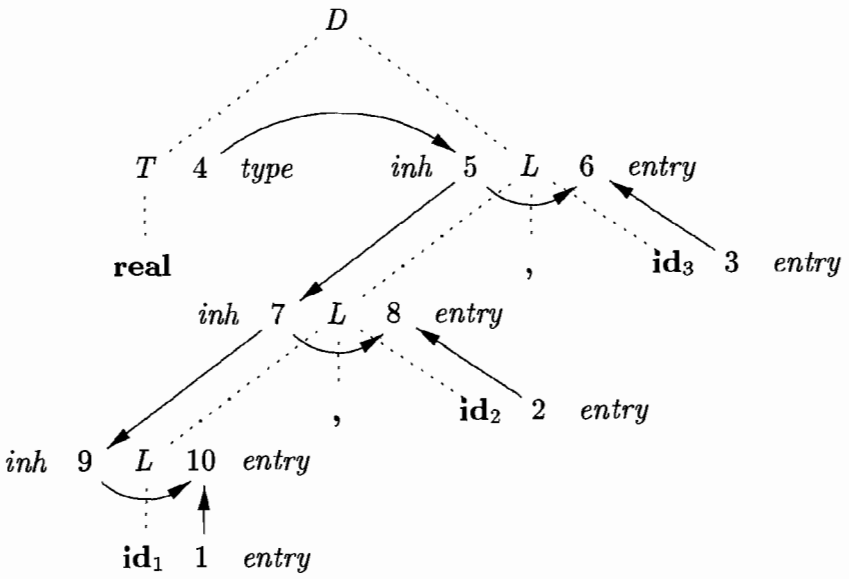

Figure 5.9: Dependency graph for a declaration **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

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.

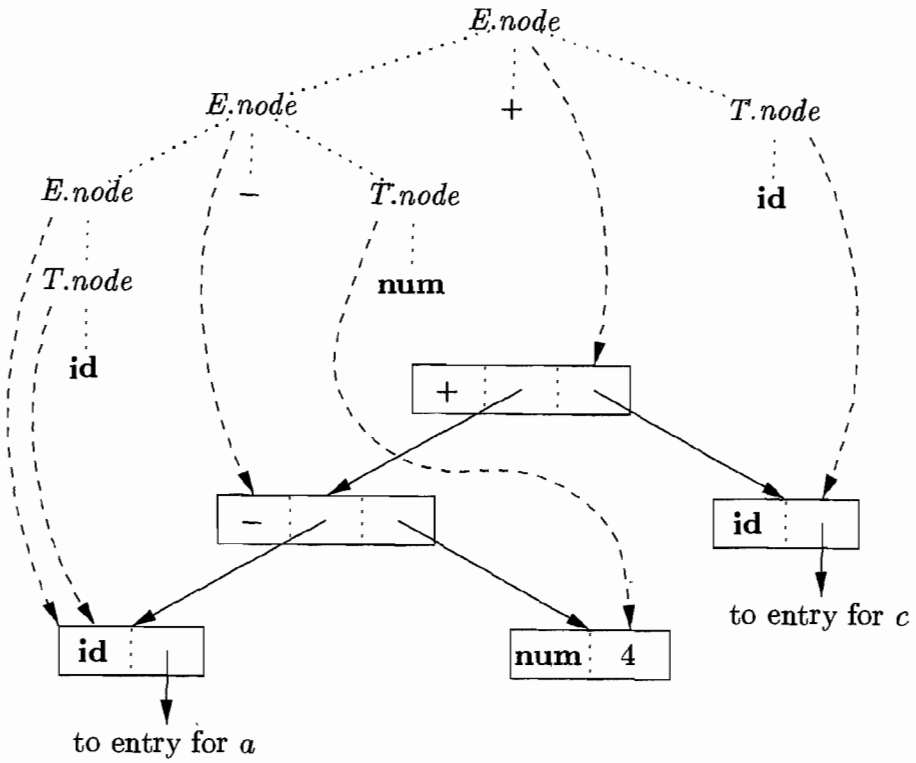

Figure 5.11: Syntax tree for $a - 4 + c$

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)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (\ E\ )$	$T.node = E.node$
6)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.13: Constructing syntax trees during top-down parsing

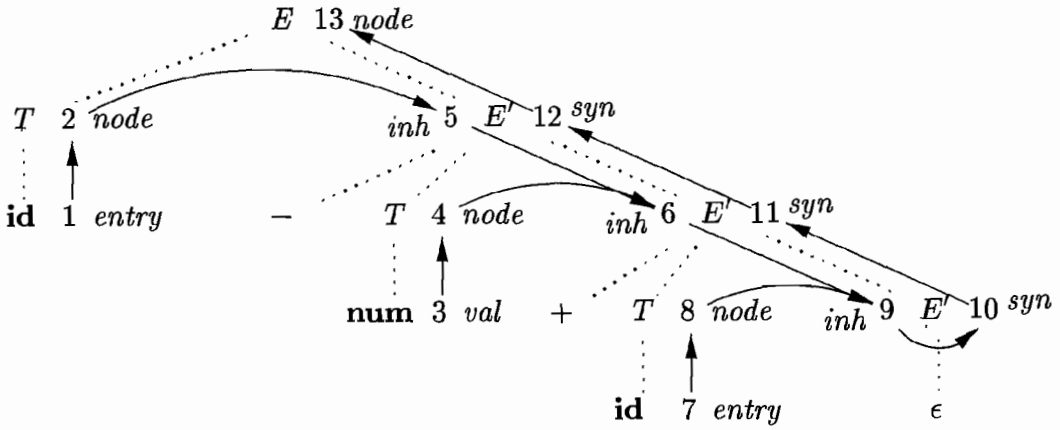

Figure 5.14: Dependency graph for $a - 4 + c$, with the SDD of Fig. 5.13

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.

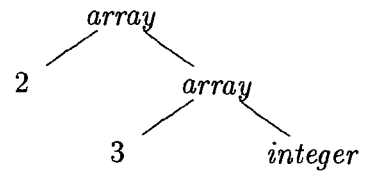

Figure 5.15: Type expression for **int**[2][3]

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 BC 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 $C_1.t$. $\square$

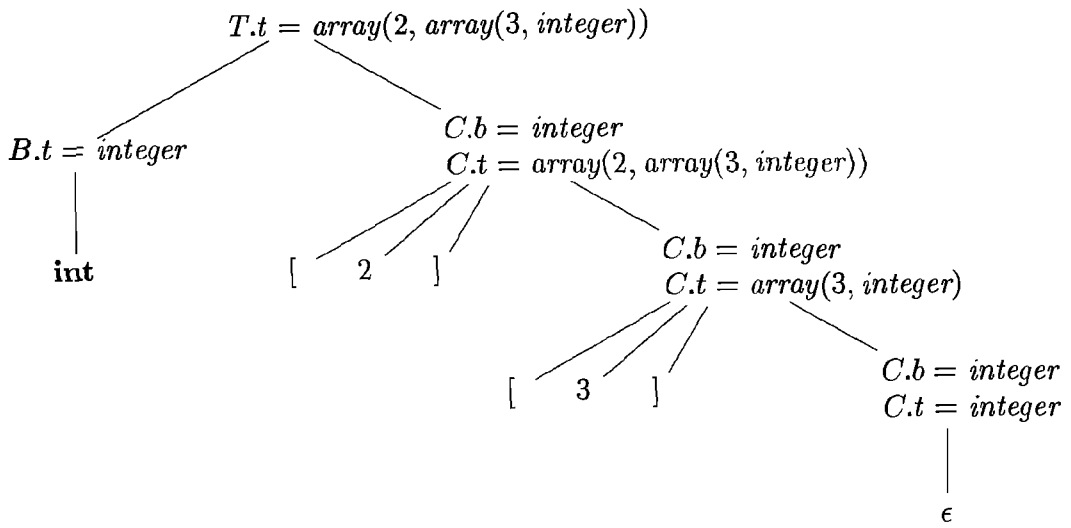

Figure 5.17: Syntax-directed translation of array types

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 $3 * 1 + 0 * x$.

5.4 Syntax-Directed Translation Schemes

Syntax-directed translation schemes are a complementary notation to syntax-directed definitions. All of the applications of syntax-directed definitions in Section 5.3 can be implemented using syntax-directed translation schemes.

From Section 2.3.5, a *syntax-directed translation scheme* (SDT) is a context-free grammar with program fragments embedded within production bodies. The program fragments are called *semantic actions* and can appear at any position within a production body. By convention, we place curly braces around actions; if braces are needed as grammar symbols, then we quote them.

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal. An example appears in Section 5.4.3.

Typically, SDT's are implemented during parsing, without building a parse tree. In this section, we focus on the use of SDT's to implement two important classes of SDD's:

1. The underlying grammar is LR-parsable, and the SDD is S-attributed.

2. The underlying grammar is LL-parsable, and the SDD is L-attributed.

We shall see how, in both these cases, the semantic rules in an SDD can be converted into an SDT with actions that are executed at the right time. During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched.

SDT's that can be implemented during parsing can be characterized by introducing distinct *marker nonterminals* in place of each embedded action; each marker M has only one production, $M \to \epsilon$. If the grammar with marker nonterminals can be parsed by a given method, then the SDT can be implemented during parsing.

5.4.1 Postfix Translation Schemes

By far the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production. SDT's with all actions at the right ends of the production bodies are called *postfix SDT's*.

Example 5.14: The postfix SDT in Fig. 5.18 implements the desk calculator SDD of Fig. 5.1, with one change: the action for the first production prints a value. The remaining actions are exact counterparts of the semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser. □

$$
\begin{array}{lcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \text{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}
$$

Figure 5.18: Postfix SDT implementing the desk calculator

5.4.2 Parser-Stack Implementation of Postfix SDT's

Postfix SDT's can be implemented during LR parsing by executing the actions when reductions occur. The attribute(s) of each grammar symbol can be put on the stack in a place where they can be found during the reduction. The best plan is to place the attributes along with the grammar symbols (or the LR states that represent these symbols) in records on the stack itself.

In Fig. 5.19, the parser stack contains records with a field for a grammar symbol (or parser state) and, below it, a field for an attribute. The three grammar symbols XYZ are on top of the stack; perhaps they are about to be reduced according to a production like $A \rightarrow XYZ$. Here, we show $X.x$ as the one attribute of X, and so on. In general, we can allow for more attributes, either by making the records large enough or by putting pointers to records on the stack. With small attributes, it may be simpler to make the records large enough, even if some fields go unused some of the time. However, if one or more attributes are of unbounded size — say, they are character strings — then it would be better to put a pointer to the attribute's value in the stack record and store the actual value in some larger, shared storage area that is not part of the stack.

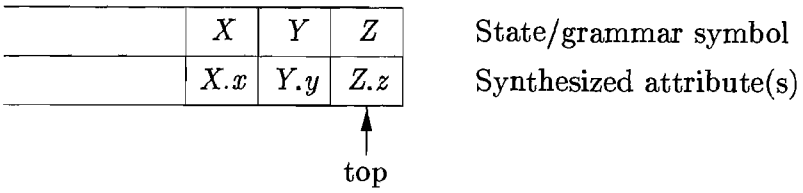

Figure 5.19: Parser stack with a field for synthesized attributes

If the attributes are all synthesized, and the actions occur at the ends of the productions, then we can compute the attributes for the head when we reduce the body to the head. If we reduce by a production such as $A \rightarrow XYZ$, then we have all the attributes of X, Y, and Z available, at known positions on the stack, as in Fig. 5.19. After the action, A and its attributes are at the top of the stack, in the position of the record for X.

Example 5.15: Let us rewrite the actions of the desk-calculator SDT of Ex-

ample 5.14 so that they manipulate the parser stack explicitly. Such stack manipulation is usually done automatically by the parser.

PRODUCTION	ACTIONS
$L \rightarrow E$ **n**	{ print($stack[top-1].val$); $top = top - 1$; }
$E \rightarrow E_1 + T$	{ $stack[top-2].val = stack[top-2].val + stack[top].val$; $top = top - 2$; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ $stack[top-2].val = stack[top-2].val \times stack[top].val$; $top = top - 2$; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ $stack[top-2].val = stack[top-1].val$; $top = top - 2$; }
$F \rightarrow$ **digit**	

Figure 5.20: Implementing the desk calculator on a bottom-up parsing stack

Suppose that the stack is kept in an array of records called *stack*, with *top* a cursor to the top of the stack. Thus, $stack[top]$ refers to the top record on the stack, $stack[top-1]$ to the record below that, and so on. Also, we assume that each record has a field called *val*, which holds the attribute of whatever grammar symbol is represented in that record. Thus, we may refer to the attribute $E.val$ that appears at the third position on the stack as $stack[top-2].val$. The entire SDT is shown in Fig. 5.20.

For instance, in the second production, $E \rightarrow E_1 + T$, we go two positions below the top to get the value of E_1, and we find the value of T at the top. The resulting sum is placed where the head E will appear after the reduction, that is, two positions below the current top. The reason is that after the reduction, the three topmost stack symbols are replaced by one. After computing $E.val$, we pop two symbols off the top of the stack, so the record where we placed $E.val$ will now be at the top of the stack.

In the third production, $E \rightarrow T$, no action is necessary, because the length of the stack does not change, and the value of $T.val$ at the stack top will simply become the value of $E.val$. The same observation applies to the productions $T \rightarrow F$ and $F \rightarrow$ **digit**. Production $F \rightarrow (E)$ is slightly different. Although the value does not change, two positions are removed from the stack during the reduction, so the value has to move to the position after the reduction.

Note that we have omitted the steps that manipulate the first field of the stack records — the field that gives the LR state or otherwise represents the grammar symbol. If we are performing an LR parse, the parsing table tells us what the new state is every time we reduce; see Algorithm 4.44. Thus, we may

simply place that state in the record for the new top of stack. □

5.4.3 SDT's With Actions Inside Productions

An action may be placed at any position within the body of a production. It is performed immediately after all symbols to its left are processed. Thus, if we have a production $B \rightarrow X \ \{a\} \ Y$, the action a is done after we have recognized X (if X is a terminal) or all the terminals derived from X (if X is a nonterminal). More precisely,

- If the parse is bottom-up, then we perform action a as soon as this occurrence of X appears on the top of the parsing stack.
- If the parse is top-down, we perform a just before we attempt to expand this occurrence of Y (if Y a nonterminal) or check for Y on the input (if Y is a terminal).

SDT's that can be implemented during parsing include postfix SDT's and a class of SDT's considered in Section 5.5 that implements L-attributed definitions. Not all SDT's can be implemented during parsing, as we shall see in the next example.

Example 5.16: As an extreme example of a problematic SDT, suppose that we turn our desk-calculator running example into an SDT that prints the prefix form of an expression, rather than evaluating the expression. The productions and actions are shown in Fig. 5.21.

1)	L	$\rightarrow$	E **n**
2)	E	$\rightarrow$	{ print('+'); } $E_1 + T$
3)	E	$\rightarrow$	T
4)	T	$\rightarrow$	{ print('*'); } $T_1 * F$
5)	T	$\rightarrow$	F
6)	F	$\rightarrow$	(E)
7)	F	$\rightarrow$	**digit** { print(**digit**.*lexval*); }

Figure 5.21: Problematic SDT for infix-to-prefix translation during parsing

Unfortunately, it is impossible to implement this SDT during either top-down or bottom-up parsing, because the parser would have to perform critical actions, like printing instances of * or +, long before it knows whether these symbols will appear in its input.

Using marker nonterminals M_2 and M_4 for the actions in productions 2 and 4, respectively, on input 3, a shift-reduce parser (see Section 4.5.3) has conflicts between reducing by $M_2 \rightarrow \epsilon$, reducing by $M_4 \rightarrow \epsilon$, and shifting the digit. □

Any SDT can be implemented as follows:

1. Ignoring the actions, parse the input and produce a parse tree as a result.
2. Then, examine each interior node N, say one for production $A \to \alpha$. Add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and actions of α.
3. Perform a preorder traversal (see Section 2.3.4) of the tree, and as soon as a node labeled by an action is visited, perform that action.

For instance, Fig. 5.22 shows the parse tree for expression $3 * 5 + 4$ with actions inserted. If we visit the nodes in preorder, we get the prefix form of the expression: $+ * 3\ 5\ 4$.

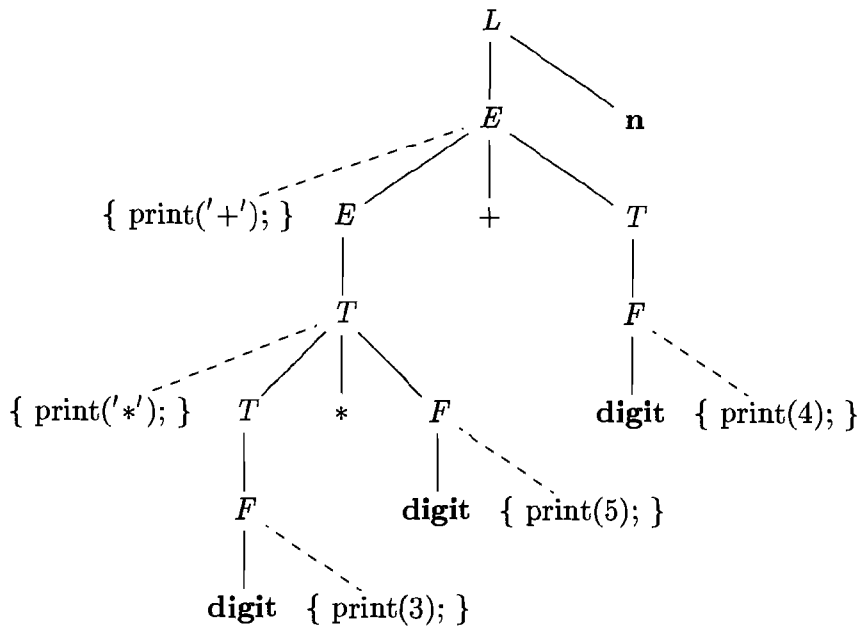

Figure 5.22: Parse tree with actions embedded

5.4.4 Eliminating Left Recursion From SDT's

Since no grammar with left recursion can be parsed deterministically top-down, we examined left-recursion elimination in Section 4.3.3. When the grammar is part of an SDT, we also need to worry about how the actions are handled.

First, consider the simple case, in which the only thing we care about is the order in which the actions in an SDT are performed. For example, if each action simply prints a string, we care only about the order in which the strings are printed. In this case, the following principle can guide us:

- When transforming the grammar, treat the actions as if they were terminal symbols.

This principle is based on the idea that the grammar transformation preserves the order of the terminals in the generated string. The actions are therefore executed in the same order in any left-to-right parse, top-down or bottom-up.

The "trick" for eliminating left recursion is to take two productions

$$A \rightarrow A\alpha \mid \beta$$

that generate strings consisting of a β and any number of α's, and replace them by productions that generate the same strings using a new nonterminal R (for "remainder") of the first production:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned}$$

If β does not begin with A, then A no longer has a left-recursive production. In regular-definition terms, with both sets of productions, A is defined by $\beta(\alpha)^*$. See Section 4.3.3 for the handling of situations where A has more recursive or nonrecursive productions.

Example 5.17: Consider the following E-productions from an SDT for translating infix expressions into postfix notation:

$$\begin{array}{lcll} E & \rightarrow & E_1 + T & \{\ \text{print}('+');\ \} \\ E & \rightarrow & T & \end{array}$$

If we apply the standard transformation to E, the remainder of the left-recursive production is

$$\alpha \ = \ + \ T \ \{\ \text{print}('+');\ \}$$

and β, the body of the other production is T. If we introduce R for the remainder of E, we get the set of productions:

$$\begin{array}{lcl} E & \rightarrow & T\ R \\ R & \rightarrow & +\ T\ \{\ \text{print}('+');\ \}\ R \\ R & \rightarrow & \epsilon \end{array}$$

□

When the actions of an SDD compute attributes rather than merely printing output, we must be more careful about how we eliminate left recursion from a grammar. However, if the SDD is S-attributed, then we can always construct an SDT by placing attribute-computing actions at appropriate positions in the new productions.

We shall give a general schema for the case of a single recursive production, a single nonrecursive production, and a single attribute of the left-recursive nonterminal; the generalization to many productions of each type is not hard, but is notationally cumbersome. Suppose that the two productions are

$$\begin{array}{lcl} A & \rightarrow & A_1\ Y\ \{A.a = g(A_1.a, Y.y)\} \\ A & \rightarrow & X\ \{A.a = f(X.x)\} \end{array}$$

Here, $A.a$ is the synthesized attribute of left-recursive nonterminal A, and X and Y are single grammar symbols with synthesized attributes $X.x$ and $Y.y$, respectively. These could represent a string of several grammar symbols, each with its own attribute(s), since the schema has an arbitrary function g computing $A.a$ in the recursive production and an arbitrary function f computing $A.a$ in the second production. In each case, f and g take as arguments whatever attributes they are allowed to access if the SDD is S-attributed.

We want to turn the underlying grammar into

$$\begin{array}{lcl} A & \rightarrow & X\ R \\ R & \rightarrow & Y\ R \mid \epsilon \end{array}$$

Figure 5.23 suggests what the SDT on the new grammar must do. In (a) we see the effect of the postfix SDT on the original grammar. We apply f once, corresponding to the use of production $A \rightarrow X$, and then apply g as many times as we use the production $A \rightarrow AY$. Since R generates a "remainder" of Y's, its translation depends on the string to its left, a string of the form $XYY\cdots Y$. Each use of the production $R \rightarrow YR$ results in an application of g. For R, we use an inherited attribute $R.i$ to accumulate the result of successively applying g, starting with the value of $A.a$.

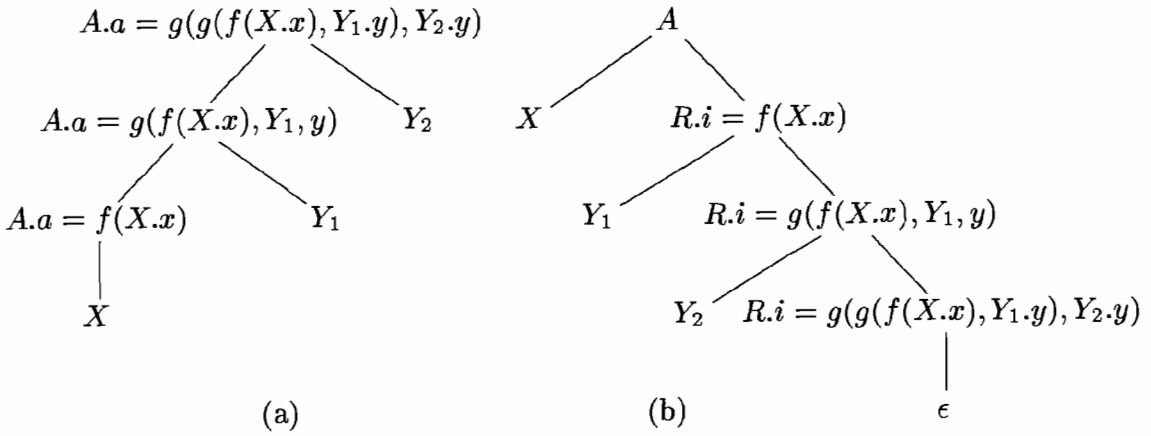

Figure 5.23: Eliminating left recursion from a postfix SDT

In addition, R has a synthesized attribute $R.s$, not shown in Fig. 5.23. This attribute is first computed when R ends its generation of Y symbols, as signaled by the use of production $R \rightarrow \epsilon$. $R.s$ is then copied up the tree, so it can become the value of $A.a$ for the entire expression $XYY\cdots Y$. The case where A generates XYY is shown in Fig. 5.23, and we see that the value of $A.a$ at the root of (a) has two uses of g. So does $R.i$ at the bottom of tree (b), and it is this value of $R.s$ that gets copied up that tree.

To accomplish this translation, we use the following SDT:

$$
\begin{array}{lcl}
A & \rightarrow & X \ \{R.i = f(X.x)\} \ R \ \{A.a = R.s\} \\
R & \rightarrow & Y \ \{R_1.i = g(R.i, Y.y\} \ R_1 \ \{R.s = R_1.s\} \\
R & \rightarrow & \epsilon \ \{R.s = R.i\}
\end{array}
$$

Notice that the inherited attribute $R.i$ is evaluated immediately before a use of R in the body, while the synthesized attributes $A.a$ and $R.s$ are evaluated at the ends of the productions. Thus, whatever values are needed to compute these attributes will be available from what has been computed to the left.

5.4.5 SDT's for L-Attributed Definitions

In Section 5.4.1, we converted S-attributed SDD's into postfix SDT's, with actions at the right ends of productions. As long as the underlying grammar is LR, postfix SDT's can be parsed and translated bottom-up.

Now, we consider the more general case of an L-attributed SDD. We shall assume that the underlying grammar can be parsed top-down, for if not it is frequently impossible to perform the translation in connection with either an LL or an LR parser. With any grammar, the technique below can be implemented by attaching actions to a parse tree and executing them during preorder traversal of the tree.

The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A in the body of the production. If several inherited attributes for A depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.

2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.

We shall illustrate these principles with two extended examples. The first involves typesetting. It illustrates how the techniques of compiling can be used in language processing for applications other than what we normally think of as programming languages. The second example is about the generation of intermediate code for a typical programming-language construct: a form of while-statement.

Example 5.18: This example is motivated by languages for typesetting mathematical formulas. `Eqn` is an early example of such a language; ideas from `Eqn` are still found in the TEX typesetting system, which was used to produce this book.

We shall concentrate on only the capability to define subscripts, subscripts of subscripts, and so on, ignoring superscripts, built-up fractions, and all other mathematical features. In the `Eqn` language, one writes `a sub i sub j` to set the expression a_{i_j}. A simple grammar for *boxes* (elements of text bounded by a rectangle) is

$$B \rightarrow B_1\ B_2 \mid B_1\ \textbf{sub}\ B_2 \mid (\ B_1\ ) \mid \textbf{text}$$

Corresponding to these four productions, a box can be either

1. Two boxes, juxtaposed, with the first, B_1, to the left of the other, B_2.
2. A box and a subscript box. The second box appears in a smaller size, lower, and to the right of the first box.
3. A parenthesized box, for grouping of boxes and subscripts. Eqn and TEX both use curly braces for grouping, but we shall use ordinary, round parentheses to avoid confusion with the braces that surround actions in SDT's.
4. A text string, that is, any string of characters.

This grammar is ambiguous, but we can still use it to parse bottom-up if we make subscripting and juxtaposition right associative, with **sub** taking precedence over juxtaposition.

Expressions will be typeset by constructing larger boxes out of smaller ones. In Fig. 5.24, the boxes for E_1 and *.height* are about to be juxtaposed to form the box for E_1*.height*. The left box for E_1 is itself constructed from the box for E and the subscript 1. The subscript 1 is handled by shrinking its box by about 30%, lowering it, and placing it after the box for E. Although we shall treat *.height* as a text string, the rectangles within its box show how it can be constructed from boxes for the individual letters.

Figure 5.24: Constructing larger boxes from smaller ones

In this example, we concentrate on the vertical geometry of boxes only. The horizontal geometry — the widths of boxes — is also interesting, especially when different characters have different widths. It may not be readily apparent, but each of the distinct characters in Fig. 5.24 has a different width.

The values associated with the vertical geometry of boxes are as follows:

a) The *point size* is used to set text within a box. We shall assume that characters not in subscripts are set in 10 point type, the size of type in this book. Further, we assume that if a box has point size p, then its subscript box has the smaller point size $0.7p$. Inherited attribute $B.ps$ will represent the point size of block B. This attribute must be inherited, because the context determines by how much a given box needs to be shrunk, due to the number of levels of subscripting.

b) Each box has a *baseline*, which is a vertical position that corresponds to the bottoms of lines of text, not counting any letters, like "g" that extend below the normal baseline. In Fig. 5.24, the dotted line represents the baseline for the boxes E, *.height*, and the entire expression. The baseline for the box containing the subscript 1 is adjusted to lower the subscript.

c) A box has a *height*, which is the distance from the top of the box to the baseline. Synthesized attribute $B.ht$ gives the height of box B.

d) A box has a *depth*, which is the distance from the baseline to the bottom of the box. Synthesized attribute $B.dp$ gives the depth of box B.

The SDD in Fig. 5.25 gives rules for computing point sizes, heights, and depths. Production 1 is used to assign $B.ps$ the initial value 10.

	PRODUCTION	SEMANTIC RULES
1)	$S \rightarrow B$	$B.ps = 10$
2)	$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
3)	$B \rightarrow B_1$ **sub** B_2	$B_1.ps = B.ps$ $B_2.ps = 0.7 \times B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4)	$B \rightarrow (\ B_1\ )$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5)	$B \rightarrow$ **text**	$B.ht = getHt(B.ps, \textbf{text}.lexval)$ $B.dp = getDp(B.ps, \textbf{text}.lexval)$

Figure 5.25: SDD for typesetting boxes

Production 2 handles juxtaposition. Point sizes are copied down the parse tree; that is, two sub-boxes of a box inherit the same point size from the larger box. Heights and depths are computed up the tree by taking the maximum. That is, the height of the larger box is the maximum of the heights of its two components, and similarly for the depth.

Production 3 handles subscripting and is the most subtle. In this greatly simplified example, we assume that the point size of a subscripted box is 70% of the point size of its parent. Reality is much more complex, since subscripts cannot shrink indefinitely; in practice, after a few levels, the sizes of subscripts

shrink hardly at all. Further, we assume that the baseline of a subscript box drops by 25% of the parent's point size; again, reality is more complex.

Production 4 copies attributes appropriately when parentheses are used. Finally, production 5 handles the leaves that represent text boxes. In this matter too, the true situation is complicated, so we merely show two unspecified functions *getHt* and *getDp* that examine tables created with each font to determine the maximum height and maximum depth of any characters in the text string. The string itself is presumed to be provided as the attribute *lexval* of terminal **text**.

Our last task is to turn this SDD into an SDT, following the rules for an L-attributed SDD, which Fig. 5.25 is. The appropriate SDT is shown in Fig. 5.26. For readability, since production bodies become long, we split them across lines and line up the actions. Production bodies therefore consist of the contents of all lines up to the head of the next production. □

	PRODUCTION	ACTIONS
1)	$S \rightarrow$	$\{\ B.ps = 10;\ \}$
	B	
2)	$B \rightarrow$	$\{ B_1.ps = B.ps;\ \}$
	B_1	$\{ B_2.ps = B.ps;\ \}$
	B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht);$
		$\quad B.dp = \max(B_1.dp, B_2.dp);\ \}$
3)	$B \rightarrow$	$\{ B_1.ps = B.ps;\ \}$
	B_1 **sub**	$\{ B_2.ps = 0.7 \times B.ps;\ \}$
	B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$
		$\quad B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps);\ \}$
4)	$B \rightarrow$ (	$\{ B_1.ps = B.ps;\ \}$
	B_1)	$\{\ B.ht = B_1.ht;$
		$\quad B.dp = B_1.dp;\ \}$
5)	$B \rightarrow$ **text**	$\{\ B.ht = getHt(B.ps, \textbf{text}.lexval);$
		$\quad B.dp = getDp(B.ps, \textbf{text}.lexval);\ \}$

Figure 5.26: SDT for typesetting boxes

Our next example concentrates on a simple while-statement and the generation of intermediate code for this type of statement. Intermediate code will be treated as a string-valued attribute. Later, we shall explore techniques that involve the writing of pieces of a string-valued attribute as we parse, thus avoiding the copying of long strings to build even longer strings. The technique was introduced in Example 5.17, where we generated the postfix form of an infix

expression "on-the-fly," rather than computing it as an attribute. However, in our first formulation, we create a string-valued attribute by concatenation.

Example 5.19: For this example, we only need one production:

$$S \rightarrow \textbf{while} \ (\ C \ ) \ S_1$$

Here, S is the nonterminal that generates all kinds of statements, presumably including if-statements, assignment statements, and others. In this example, C stands for a conditional expression — a boolean expression that evaluates to true or false.

In this flow-of-control example, the only things we ever generate are labels. All the other intermediate-code instructions are assumed to be generated by parts of the SDT that are not shown. Specifically, we generate explicit instructions of the form **label** L, where L is an identifier, to indicate that L is the label of the instruction that follows. We assume that the intermediate code is like that introduced in Section 2.8.4.

The meaning of our while-statement is that the conditional C is evaluated. If it is true, control goes to the beginning of the code for S_1. If false, then control goes to the code that follows the while-statement's code. The code for S_1 must be designed to jump to the beginning of the code for the while-statement when finished; the jump to the beginning of the code that evaluates C is not shown in Fig. 5.27.

We use the following attributes to generate the proper intermediate code:

1. The inherited attribute $S.next$ labels the beginning of the code that must be executed after S is finished.

2. The synthesized attribute $S.code$ is the sequence of intermediate-code steps that implements a statement S and ends with a jump to $S.next$.

3. The inherited attribute $C.true$ labels the beginning of the code that must be executed if C is true.

4. The inherited attribute $C.false$ labels the beginning of the code that must be executed if C is false.

5. The synthesized attribute $C.code$ is the sequence of intermediate-code steps that implements the condition C and jumps either to $C.true$ or to $C.false$, depending on whether C is true or false.

The SDD that computes these attributes for the while-statement is shown in Fig. 5.27. A number of points merit explanation:

- The function *new* generates new labels.

- The variables $L1$ and $L2$ hold labels that we need in the code. $L1$ is the beginning of the code for the while-statement, and we need to arrange

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$	$L1 = new();$
	$L2 = new();$
	$S_1.next = L1;$
	$C.false = S.next;$
	$C.true = L2;$
	$S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code$

Figure 5.27: SDD for while-statements

that S_1 jumps there after it finishes. That is why we set $S_1.next$ to $L1$. $L2$ is the beginning of the code for S_1, and it becomes the value of $C.true$, because we branch there when C is true.

- Notice that $C.false$ is set to $S.next$, because when the condition is false, we execute whatever code must follow the code for S.
- We use $\parallel$ as the symbol for concatenation of intermediate-code fragments. The value of $S.code$ thus begins with the label $L1$, then the code for condition C, another label $L2$, and the code for S_1.

This SDD is L-attributed. When we convert it into an SDT, the only remaining issue is how to handle the labels $L1$ and $L2$, which are variables, and not attributes. If we treat actions as dummy nonterminals, then such variables can be treated as the synthesized attributes of dummy nonterminals. Since $L1$ and $L2$ do not depend on any other attributes, they can be assigned to the first action in the production. The resulting SDT with embedded actions that implements this L-attributed definition is shown in Fig. 5.28. □

$S \rightarrow \textbf{while}\ ($	$\{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \}$
$C\ )$	$\{\ S_1.next = L1;\ \}$
S_1	$\{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;\ \}$

Figure 5.28: SDT for while-statements

5.4.6 Exercises for Section 5.4

Exercise 5.4.1: We mentioned in Section 5.4.2 that it is possible to deduce, from the LR state on the parsing stack, what grammar symbol is represented by the state. How would we discover this information?

Exercise 5.4.2: Rewrite the following SDT:

$$A \rightarrow A\ \{a\}\ B \mid A\ B\ \{b\} \mid 0$$
$$B \rightarrow B\ \{c\}\ A \mid B\ A\ \{d\} \mid 1$$

so that the underlying grammar becomes non-left-recursive. Here, a, b, c, and d are actions, and 0 and 1 are terminals.

Exercise 5.4.3: The following SDT computes the value of a string of 0's and 1's interpreted as a positive, binary integer.

$$\begin{array}{rcl} B & \rightarrow & B_1\ 0\ \{B.val = 2 \times B_1.val\} \\ & | & B_1\ 1\ \{B.val = 2 \times B_1.val + 1\} \\ & | & 1\ \{B.val = 1\} \end{array}$$

Rewrite this SDT so the underlying grammar is not left recursive, and yet the same value of $B.val$ is computed for the entire input string.

Exercise 5.4.4: Write L-attributed SDD's analogous to that of Example 5.19 for the following productions, each of which represents a familiar flow-of-control construct, as in the programming language C. You may need to generate a three-address statement to jump to a particular label L, in which case you should generate **goto** L.

a) $S \rightarrow$ **if** (C) S_1 **else** S_2

b) $S \rightarrow$ **do** S_1 **while** (C)

c) $S \rightarrow$ '{' L '}'; $L \rightarrow L\ S \mid \epsilon$

Note that any statement in the list can have a jump from its middle to the next statement, so it is not sufficient simply to generate code for each statement in order.

Exercise 5.4.5: Convert each of your SDD's from Exercise 5.4.4 to an SDT in the manner of Example 5.19.

Exercise 5.4.6: Modify the SDD of Fig. 5.25 to include a synthesized attribute $B.le$, the length of a box. The length of the concatenation of two boxes is the sum of the lengths of each. Then add your new rules to the proper positions in the SDT of Fig. 5.26

Exercise 5.4.7: Modify the SDD of Fig. 5.25 to include superscripts denoted by operator **sup** between boxes. If box B_2 is a superscript of box B_1, then position the baseline of B_2 0.6 times the point size of B_1 above the baseline of B_1. Add the new production and rules to the SDT of Fig. 5.26.

5.5 Implementing L-Attributed SDD's

Since many translation applications can be addressed using L-attributed definitions, we shall consider their implementation in more detail in this section. The following methods do translation by traversing a parse tree:

1. *Build the parse tree and annotate.* This method works for any noncircular SDD whatsoever. We introduced annotated parse trees in Section 5.1.2.

2. *Build the parse tree, add actions, and execute the actions in preorder.* This approach works for any L-attributed definition. We discussed how to turn an L-attributed SDD into an SDT in Section 5.4.5; in particular, we discussed how to embed actions into productions based on the semantic rules of such an SDD.

In this section, we discuss the following methods for translation during parsing:

3. *Use a recursive-descent parser* with one function for each nonterminal. The function for nonterminal A receives the inherited attributes of A as arguments and returns the synthesized attributes of A.

4. *Generate code on the fly*, using a recursive-descent parser.

5. *Implement an SDT in conjunction with an LL-parser.* The attributes are kept on the parsing stack, and the rules fetch the needed attributes from known locations on the stack.

6. *Implement an SDT in conjunction with an LR-parser.* This method may be surprising, since the SDT for an L-attributed SDD typically has actions in the middle of productions, and we cannot be sure during an LR parse that we are even in that production until its entire body has been constructed. We shall see, however, that if the underlying grammar is LL, we can always handle both the parsing and translation bottom-up.

5.5.1 Translation During Recursive-Descent Parsing

A recursive-descent parser has a function A for each nonterminal A, as discussed in Section 4.4.1. We can extend the parser into a translator as follows:

a) The arguments of function A are the inherited attributes of nonterminal A.

b) The return-value of function A is the collection of synthesized attributes of nonterminal A.

In the body of function A, we need to both parse and handle attributes:

1. Decide upon the production used to expand A.

2. Check that each terminal appears on the input when it is required. We shall assume that no backtracking is needed, but the extension to recursive-descent parsing with backtracking can be done by restoring the input position upon failure, as discussed in Section 4.4.1.

3. Preserve, in local variables, the values of all attributes needed to compute inherited attributes for nonterminals in the body or synthesized attributes for the head nonterminal.

4. Call functions corresponding to nonterminals in the body of the selected production, providing them with the proper arguments. Since the underlying SDD is L-attributed, we have already computed these attributes and stored them in local variables.

Example 5.20: Let us consider the SDD and SDT of Example 5.19 for while-statements. A pseudocode rendition of the relevant parts of the function S appears in Fig. 5.29.

```
string S(label next) {
        string Scode, Ccode; /* local variables holding code fragments */
        label L1, L2; /* the local labels */
        if ( current input == token while ) {
                advance input;
                check '(' is next on the input, and advance;
                L1 = new();
                L2 = new();
                Ccode = C(next, L2);
                check ')' is next on the input, and advance;
                Scode = S(L1);
                return("label" || L1 || Ccode || "label" || L2 || Scode);
        }
        else /* other statement types */
}
```

Figure 5.29: Implementing while-statements with a recursive-descent parser

We show S as storing and returning long strings. In practice, it would be far more efficient for functions like S and C to return pointers to records that represent these strings. Then, the return-statement in function S would not physically concatenate the components shown, but rather would construct a record, or perhaps tree of records, expressing the concatenation of the strings represented by *Scode* and *Ccode*, the labels $L1$ and $L2$, and the two occurrences of the literal string `"label"`. □

Example 5.21: Now, let us take up the SDT of Fig. 5.26 for typesetting boxes. First, we address parsing, since the underlying grammar in Fig. 5.26 is ambiguous. The following transformed grammar makes juxtaposition and subscripting right associative, with **sub** taking precedence over juxtaposition:

$$
\begin{array}{lcl}
S & \rightarrow & B \\
B & \rightarrow & T\ B_1 \mid T \\
T & \rightarrow & F\ \textbf{sub}\ T_1 \mid F \\
F & \rightarrow & (\ B\ ) \mid \textbf{text}
\end{array}
$$

The two new nonterminals, T and F, are motivated by terms and factors in expressions. Here, a "factor," generated by F, is either a parenthesized box or a text string. A "term," generated by T, is a "factor" with a sequence of subscripts, and a box generated by B is a sequence of juxtaposed "terms."

The attributes of B carry over to T and F, since the new nonterminals also denote boxes; they were introduced simply to aid parsing. Thus, both T and F have an inherited attribute *ps* and synthesized attributes *ht* and *dp*, with semantic actions that can be adapted from the SDT in Fig. 5.26.

The grammar is not yet ready for top-down parsing, since the productions for B and T have common prefixes. Consider T, for instance. A top-down parser cannot choose between the two productions for T by looking one symbol ahead in the input. Fortunately, we can use a form of left-factoring, discussed in Section 4.3.4, to make the grammar ready. With SDT's, the notion of common prefix applies to actions as well. Both productions for T begin with the nonterminal F inheriting attribute *ps* from T.

The pseudocode in Fig. 5.30 for $T(ps)$ folds in the code for $F(ps)$. After left-factoring is applied to $T \rightarrow F\ \textbf{sub}\ T_1 \mid F$, there is only one call to F; the pseudocode shows the result of substituting the code for F in place of this call.

The function T will be called as $T(10.0)$ by the function for B, which we do not show. It returns a pair consisting of the height and depth of the box generated by nonterminal T; in practice, it would return a record containing the height and depth.

Function T begins by checking for a left parenthesis, in which case it must have the production $F \rightarrow (\ B\ )$ to work with. It saves whatever the B inside the parentheses returns, but if that B is not followed by a right parenthesis, then there is a syntax error, which must be handled in a manner not shown.

Otherwise, if the current input is **text**, then the function T uses *getHt* and *getDp* to determine the height and depth of this text.

T then decides whether the next box is a subscript and adjusts the point size, if so. We use the actions associated with the production $B \rightarrow B\ \textbf{sub}\ B$ in Fig. 5.26 for the height and depth of the larger box. Otherwise, we simply return what F would have returned: $(h1, d1)$. □

5.5.2 On-The-Fly Code Generation

The construction of long strings of code that are attribute values, as in Example 5.20, is undesirable for several reasons, including the time it could take to copy or move long strings. In common cases such as our running code-generation example, we can instead incrementally generate pieces of the code into an array or output file by executing actions in an SDT. The elements we need to make this technique work are:

```
(float, float) T(float ps) {
        float h1, h2, d1, d2; /* locals to hold heights and depths */
        /* start code for F(ps) */
        if ( current input == '(' ) {
                advance input;
                (h1, d1) = B(ps);
                if (current input != ')' ) syntax error: expected ')';
                advance input;
        }
        else if ( current input == text ) {
                let lexical value text.lexval be t;
                advance input;
                h1 = getHt(ps, t);
                d1 = getDp(ps, t);
        }
        else syntax error: expected text or '(';
        /* end code for F(ps) */
        if ( current input == sub ) {
                advance input;
                (h2, d2) = T(0.7 * ps);
                return (max(h1, h2 - 0.25 * ps), max(d1, d2 + 0.25 * ps));
        }
        return (h1, d1);
}
```

Figure 5.30: Recursive-descent typesetting of boxes

1. There is, for one or more nonterminals, a *main* attribute. For convenience, we shall assume that the main attributes are all string valued. In Example 5.20, the attributes $S.code$ and $C.code$ are main attributes; the other attributes are not.

2. The main attributes are synthesized.

3. The rules that evaluate the main attribute(s) ensure that

   (a) The main attribute is the concatenation of main attributes of nonterminals appearing in the body of the production involved, perhaps with other elements that are not main attributes, such as the string **label** or the values of labels $L1$ and $L2$.

   (b) The main attributes of nonterminals appear in the rule in the same order as the nonterminals themselves appear in the production body.

As a consequence of the above conditions, the main attribute can be constructed by emitting the non-main-attribute elements of the concatenation. We can rely

The Type of Main Attributes

Our simplifying assumption that main attributes are of string type is really too restrictive. The true requirement is that the type of all the main attributes must have values that can be constructed by concatenation of elements. For instance, a list of objects of any type would be appropriate, as long as we represent these lists in a way that allows elements to be efficiently appended to the end of the list. Thus, if the purpose of the main attribute is to represent a sequence of intermediate-code statements, we could produce the intermediate code by writing statements to the end of an array of objects. Of course the requirements stated in Section 5.5.2 still apply to lists; for example, main attributes must be assembled from other main attributes by concatenation in order.

on the recursive calls to the functions for the nonterminals in a production body to emit the value of their main attribute incrementally.

Example 5.22: We can modify the function of Fig. 5.29 to emit elements of the main translation *S.code* instead of saving them for concatenation into a return value of *S.code*. The revised function S appears in Fig. 5.31.

```
void S(label next) {
        label L1, L2; /* the local labels */
        if ( current input == token while ) {
                advance input;
                check '(' is next on the input, and advance;
                L1 = new();
                L2 = new();
                print("label", L1);
                C(next, L2);
                check ')' is next on the input, and advance;
                print("label", L2);
                S(L1);
        }
        else /* other statement types */
}
```

Figure 5.31: On-the-fly recursive-descent code generation for while-statements

In Fig. 5.31, S and C now have no return value, since their only synthesized attributes are produced by printing. Further, the position of the print statements is significant. The order in which output is printed is: first `label` $L1$, then the code for C (which is the same as the value of *Ccode* in Fig. 5.29), then

`label` $L2$, and finally the code from the recursive call to S (which is the same as *Scode* in Fig. 5.29). Thus, the code printed by this call to S is exactly the same as the value of *Scode* that is returned in Fig. 5.29). □

Incidentally, we can make the same change to the underlying SDT: turn the construction of a main attribute into actions that emit the elements of that attribute. In Fig. 5.32 we see the SDT of Fig. 5.28 revised to generate code on the fly.

$$
\begin{array}{lll}
S \rightarrow \textbf{while} (& \{\ L1 = new();\ L2 = new();\ C.false = S.next; \\
 & \quad C.true = L2;\ print(\texttt{"label"}, L1);\ \} \\
C\ ) & \{\ S_1.next = L1;\ print(\texttt{"label"}, L2);\ \} \\
S_1 &
\end{array}
$$

Figure 5.32: SDT for on-the-fly code generation for while statements

5.5.3 L-Attributed SDD's and LL Parsing

Suppose that an L-attributed SDD is based on an LL-grammar and that we have converted it to an SDT with actions embedded in the productions, as described in Section 5.4.5. We can then perform the translation during LL parsing by extending the parser stack to hold actions and certain data items needed for attribute evaluation. Typically, the data items are copies of attributes.

In addition to records representing terminals and nonterminals, the parser stack will hold *action-records* representing actions to be executed and *synthesize-records* to hold the synthesized attributes for nonterminals. We use the following two principles to manage attributes on the stack:

- The inherited attributes of a nonterminal A are placed in the stack record that represents that nonterminal. The code to evaluate these attributes will usually be represented by an action-record immediately above the stack record for A; in fact, the conversion of L-attributed SDD's to SDT's ensures that the action-record will be immediately above A.
- The synthesized attributes for a nonterminal A are placed in a separate synthesize-record that is immediately below the record for A on the stack.

This strategy places records of several types on the parsing stack, trusting that these variant record types can be managed properly as subclasses of a "stack-record" class. In practice, we might combine several records into one, but the ideas are perhaps best explained by separating data used for different purposes into different records.

Action-records contain pointers to code to be executed. Actions may also appear in synthesize-records; these actions typically place copies of the synthesized attribute(s) in other records further down the stack, where the value of

that attribute will be needed after the synthesize-record and its attributes are popped off the stack.

Let us take a brief look at LL parsing to see the need to make temporary copies of attributes. From Section 4.4.4, a table-driven LL parser mimics a leftmost derivation. If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols α such that $S \stackrel{*}{\underset{lm}{\Rightarrow}} w\alpha$, where S is the start symbol. When the parser expands by a production $A \rightarrow B\,C$, it replaces A on top of the stack by $B\,C$.

Suppose nonterminal C has an inherited attribute $C.i$. With $A \rightarrow B\,C$, the inherited attribute $C.i$ may depend not only on the inherited attributes of A, but on all the attributes of B. Thus, we may need to process B completely before $C.i$ can be evaluated. We therefore save temporary copies of all the attributes needed to evaluate $C.i$ in the action-record that evaluates $C.i$. Otherwise, when the parser replaces A on top of the stack by $B\,C$, the inherited attributes of A will have disappeared, along with its stack record.

Since the underlying SDD is L-attributed, we can be sure that the values of the inherited attributes of A are available when A rises to the top of the stack. The values will therefore be available in time to be copied into the action-record that evaluates the inherited attributes of C. Furthermore, space for the synthesized attributes of A is not a problem, since the space is in the synthesize-record for A, which remains on the stack, below B and C, when the parser expands by $A \rightarrow B\,C$.

As B is processed, we can perform actions (through a record just above B on the stack) that copy its inherited attributes for use by C, as needed, and after B is processed, the synthesize-record for B can copy its synthesized attributes for use by C, if needed. Likewise, synthesized attributes of A may need temporaries to help compute their value, and these can be copied to the synthesize-record for A as B and then C are processed. The principle that makes all this copying of attributes work is:

- All copying takes place among the records that are created during one expansion of one nonterminal. Thus, each of these records knows how far below it on the stack each other record is, and can write values into the records below safely.

The next example illustrates the implementation of inherited attributes during LL parsing by diligently copying attribute values. Shortcuts or optimizations are possible, particularly with copy rules, which simply copy the value of one attribute into another. Shortcuts are deferred until Example 5.24, which also illustrates synthesize-records.

Example 5.23: This example implements the the SDT of Fig. 5.32, which generates code on the fly for the while-production. This SDT does not have synthesized attributes, except for dummy attributes that represent labels.

Figure 5.33(a) shows the situation as we are about to use the while-production to expand S, presumably because the lookahead symbol on the input is

while. The record at the top of stack is for S, and it contains only the inherited attribute $S.next$, which we suppose has the value x. Since we are now parsing top-down, we show the stack top at the left, according to our usual convention.

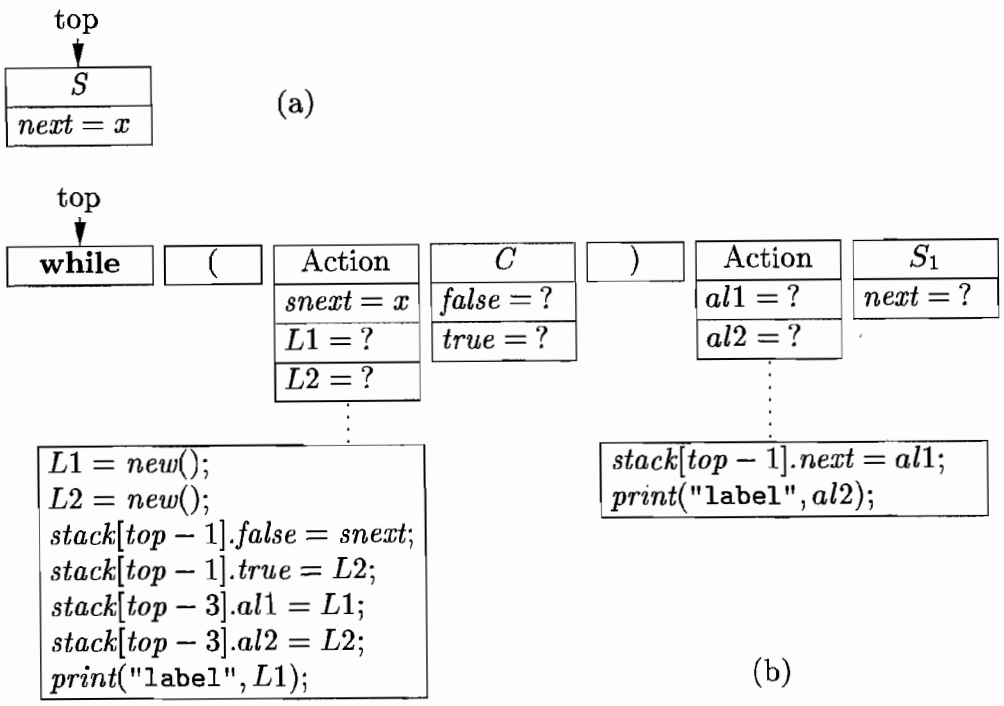

Figure 5.33: Expansion of S according to the while-statement production

Figure 5.33(b) shows the situation immediately after we have expanded S. There are action-records in front of the nonterminals C and S_1, corresponding to the actions in the underlying SDT of Fig. 5.32. The record for C has room for inherited attributes *true* and *false*, while the record for S_1 has room for attribute *next*, as all S-records must. We show values for these fields as ?, because we do not yet know their values.

The parser next recognizes **while** and (on the input and pops their records off the stack. Now, the first action is at the top, and it must be executed. This action-record has a field *snext*, which holds a copy of the inherited attribute $S.next$. When S is popped from the stack, the value of $S.next$ is copied into the field *snext* for use during the evaluation of the inherited attributes for C. The code for the first action generates new values for $L1$ and $L2$, which we shall suppose are y and z, respectively. The next step is to make z the value of $C.true$. The assignment $stack[top - 1].true = L2$ is written knowing it is only executed when this action-record is at the top of stack, so $top - 1$ refers to the record below it — the record for C.

The first action-record then copies $L1$ into field $al1$ in the second action, where it will be used to evaluate $S_1.next$. It also copies $L2$ into a field called $al2$ of the second action; this value is needed for that action-record to print its output properly. Finally, the first action-record prints `label` y to the output.

The situation after completing the first action and popping its record off

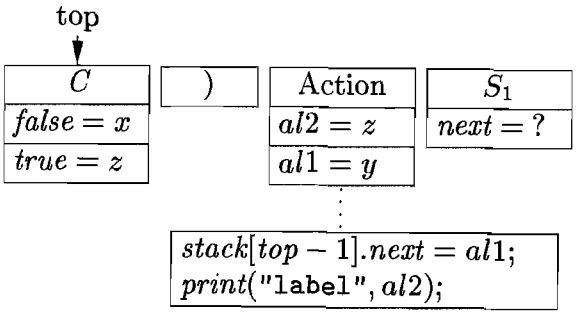

Figure 5.34: After the action above C is performed

the stack is shown in Fig. 5.34. The values of inherited attributes in the record for C have been filled in properly, as have the temporaries $al1$ and $al2$ in the second action record. At this point, C is expanded, and we presume that the code to implement its test containing jumps to labels x and z, as appropriate, is generated. When the C-record is popped from the stack, the record for) becomes top and causes the parser to check for) on its input.

With the action above S_1 at the top of the stack, its code sets $S_1.next$ and emits `label` z. When that is done, the record for S_1 becomes the top of stack, and as it is expanded, we presume it correctly generates code that implements whatever kind of statement it is and then jump to label y. □

Example 5.24: Now, let us consider the same while-statement, but with a translation that produces the output $S.code$ as a synthesized attribute, rather than by on-the-fly generation. In order to follow the explanation, it is useful to bear in mind the following invariant or inductive hypothesis, which we assume is followed for every nonterminal:

- Every nonterminal that has code associated with it leaves that code, as a string, in the synthesize-record just below it on the stack.

Assuming this statement is true, we shall handle the while-production so it maintains this statement as an invariant.

Figure 5.35(a) shows the situation just before S is expanded using the production for while-statements. At the top of the stack we see the record for S; it has a field for its inherited attribute $S.next$, as in Example 5.23. Immediately below that record is the synthesize-record for this occurrence of S. The latter has a field for $S.code$, as all synthesize-records for S must have. We also show it with some other fields for local storage and actions, since the SDT for the while production in Fig. 5.28 is surely part of a larger SDT.

Our expansion of S is based on the SDT of Fig. 5.28, and it is shown in Fig. 5.35(b). As a shortcut, during the expansion, we assume that the inherited attribute $S.next$ is assigned directly to $C.false$, rather than being placed in the first action and then copied into the record for C.

Let us examine what each record does when it becomes the top of stack. First, the **while** record causes the token **while** to be matched with the input,

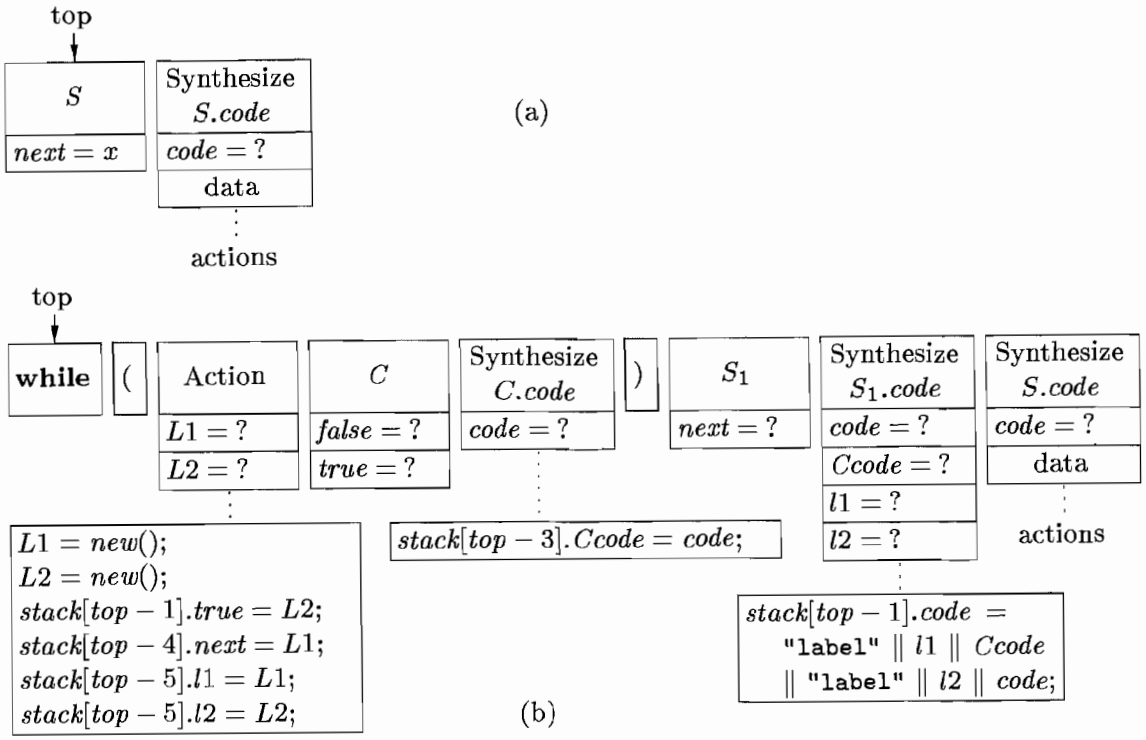

Figure 5.35: Expansion of S with synthesized attribute constructed on the stack

which it must, or else we would not have expanded S in this way. After **while** and (are popped off the stack, the code for the action-record is executed. It generates values for $L1$ and $L2$, and we take the shortcut of copying them directly to the inherited attributes that need them: $S_1.next$ and $C.true$. The last two steps of the action cause $L1$ and $L2$ to be copied into the record called "Synthesize $S_1.code$."

The synthesize-record for S_1 does double duty: not only will it hold the synthesized attribute $S_1.code$, but it will also serve as an action-record to complete the evaluation of the attributes for the entire production $S \rightarrow$ **while** $(\ C\ )\ S_1$. In particular, when it gets to the top, it will compute the synthesized attribute $S.code$ and place its value in the synthesize-record for the head S.

When C becomes the top of the stack, it has both its inherited attributes computed. By the inductive hypothesis stated above, we suppose it correctly generates code to execute its condition and jump to the proper label. We also assume that the actions performed during the expansion of C correctly place this code in the record below, as the value of synthesized attribute $C.code$.

After C is popped, the synthesize-record for $C.code$ becomes the top. Its code is needed in the synthesize-record for $S_1.code$, because that is where we concatenate all the code elements to form $S.code$. The synthesize-record for $C.code$ therefore has an action to copy $C.code$ into the synthesize-record for $S_1.code$. After doing so, the record for token) reaches the top of stack, and causes a check for) on the input. Assuming that test succeeds, the record for S_1 becomes the top of stack. By our inductive hypothesis, this nonterminal is

Can We Handle L-Attributed SDD's on LR Grammars?

In Section 5.4.1, we saw that every S-attributed SDD on an LR grammar can be implemented during a bottom-up parse. From Section 5.5.3 every L-attributed SDD on an LL grammar can be parsed top-down. Since LL grammars are a proper subset of the LR grammars, and the S-attributed SDD's are a proper subset of the L-attributed SDD's, can we handle every LR grammar and L-attributed SDD bottom-up?

We cannot, as the following intuitive argument shows. Suppose we have a production $A \to B\,C$ in an LR-grammar, and there is an inherited attribute $B.i$ that depends on inherited attributes of A. When we reduce to B, we still have not seen the input that C generates, so we cannot be sure that we have a body of production $A \to B\,C$. Thus, we cannot compute $B.i$ yet, since we are unsure whether to use the rule associated with this production.

Perhaps we could wait until we have reduced to C, and know that we must reduce $B\,C$ to A. However, even then, we do not know the inherited attributes of A, because even after reduction, we may not be sure of the production body that contains this A. We could reason that this decision, too, should be deferred, and therefore further defer the computation of $B.i$. If we keep reasoning this way, we soon realize that we cannot make any decisions until the entire input is parsed. Essentially, we have reached the strategy of "build the parse tree first and then perform the translation."

expanded, and the net effect is that its code is correctly constructed and placed in the field for *code* in the synthesize-record for S_1.

Now, all the data fields of the synthesize-record for S_1 have been filled in, so when it becomes the top of stack, the action in that record can be executed. The action causes the labels and code from $C.code$ and $S_1.code$ to be concatenated in the proper order. The resulting string is placed in the record below; that is, in the synthesize-record for S. We have now correctly computed $S.code$, and when the synthesize-record for S becomes the top, that code is available for placement in another record further down the stack, where it will eventually be assembled into a larger string of code implementing a program element of which this S is a part. □

5.5.4 Bottom-Up Parsing of L-Attributed SDD's

We can do bottom-up every translation that we can do top-down. More precisely, given an L-attributed SDD on an LL grammar, we can adapt the grammar to compute the same SDD on the new grammar during an LR parse. The "trick" has three parts:

1. Start with the SDT constructed as in Section 5.4.5, which places embedded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized attributes.

2. Introduce into the grammar a marker nonterminal in place of each embedded action. Each such place gets a distinct marker, and there is one production for any marker M, namely $M \to \epsilon$.

3. Modify the action a if marker nonterminal M replaces it in some production $A \to \alpha\,\{a\}\,\beta$, and associate with $M \to \epsilon$ an action a' that

   (a) Copies, as inherited attributes of M, any attributes of A or symbols of α that action a needs.

   (b) Computes attributes in the same way as a, but makes those attributes be synthesized attributes of M.

   This change appears illegal, since typically the action associated with production $M \to \epsilon$ will have to access attributes belonging to grammar symbols that do not appear in this production. However, we shall implement the actions on the LR parsing stack, so the necessary attributes will always be available a known number of positions down the stack.

Example 5.25: Suppose that there is a production $A \to B\,C$ in an LL grammar, and the inherited attribute $B.i$ is computed from inherited attribute $A.i$ by some formula $B.i = f(A.i)$. That is, the fragment of an SDT we care about is

$$A \to \{B.i = f(A.i);\}\ B\ C$$

We introduce marker M with inherited attribute $M.i$ and synthesized attribute $M.s$. The former will be a copy of $A.i$ and the latter will be $B.i$. The SDT will be written

$$\begin{aligned} A &\to M\ B\ C \\ M &\to \{M.i = A.i;\ M.s = f(M.i);\} \end{aligned}$$

Notice that the rule for M does not have $A.i$ available to it, but in fact we shall arrange that every inherited attribute for a nonterminal such as A appears on the stack immediately below where the reduction to A will later take place. Thus, when we reduce ϵ to M, we shall find $A.i$ immediately below it, from where it may be read. Also, the value of $M.s$, which is left on the stack along with M, is really $B.i$ and properly is found right below where the reduction to B will later occur. □

Example 5.26: Let us turn the SDT of Fig. 5.28 into an SDT that can operate with an LR parse of the revised grammar. We introduce a marker M before C and a marker N before S_1, so the underlying grammar becomes

Why Markers Work

Markers are nonterminals that derive only ϵ and that appear only once among all the bodies of all productions. We shall not give a formal proof that, when a grammar is LL, marker nonterminals can be added at any position in the body, and the resulting grammar will still be LR. The intuition, however, is as follows. If a grammar is LL, then we can determine that a string w on the input is derived from nonterminal A, in a derivation that starts with production $A \rightarrow \alpha$, by seeing only the first symbol of w (or the following symbol if $w = \epsilon$). Thus, if we parse w bottom-up, then the fact that a prefix of w must be reduced to α and then to S is known as soon as the beginning of w appears on the input. In particular, if we insert markers anywhere in α, the LR states will incorporate the fact that this marker has to be there, and will reduce ϵ to the marker at the appropriate point on the input.

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{while} \ (\ M\ C\ )\ N\ S_1 \\
M & \rightarrow & \epsilon \\
N & \rightarrow & \epsilon
\end{array}
$$

Before we discuss the actions that are associated with markers M and N, let us outline the "inductive hypothesis" about where attributes are stored.

1. Below the entire body of the while-production — that is, below **while** on the stack — will be the inherited attribute *S.next*. We may not know the nonterminal or parser state associated with this stack record, but we can be sure that it will have a field, in a fixed position of the record, that holds *S.next* before we begin to recognize what is derived from this S.

2. Inherited attributes *C.true* and *C.false* will be just below the stack record for C. Since the grammar is presumed to be LL, the appearance of **while** on the input assures us that the while-production is the only one that can be recognized, so we can be sure that M will appear immediately below C on the stack, and M's record will hold the inherited attributes of C.

3. Similarly, the inherited attribute S_1.*next* must appear immediately below S_1 on the stack, so we may place that attribute in the record for N.

4. The synthesized attribute *C.code* will appear in the record for C. As always when we have a long string as an attribute value, we expect that in practice a pointer to (an object representing) the string will appear in the record, while the string itself is outside the stack.

5. Similarly, the synthesized attribute S_1.*code* will appear in the record for S_1.

Let us follow the parsing process for a while-statement. Suppose that a record holding $S.next$ appears on the top of the stack, and the next input is the terminal **while**. We shift this terminal onto the stack. It is then certain that the production being recognized is the while-production, so the LR parser can shift "(" and determine that its next step must be to reduce ϵ to M. The stack at this time is shown in Fig. 5.36. We also show in that figure the action that is associated with the reduction to M. We create values for $L1$ and $L2$, which live in fields of the M-record. Also in that record are fields for $C.true$ and $C.false$. These attributes must be in the second and third fields of the record, for consistency with other stack records that might appear below C in other contexts and also must provide these attributes for C. The action completes by assigning values to $C.true$ and $C.false$, one from the $L2$ just generated, and the other by reaching down the stack to where we know $S.next$ is found.

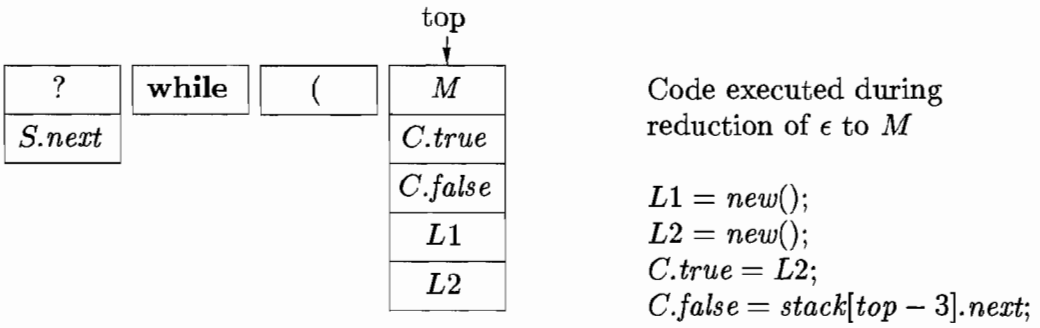

Figure 5.36: LR parsing stack after reduction of ϵ to M

We presume that the next inputs are properly reduced to C. The synthesized attribute $C.code$ is therefore placed in the record for C. This change to the stack is shown in Fig. 5.37, which also incorporates the next several records that are later placed above C on the stack.

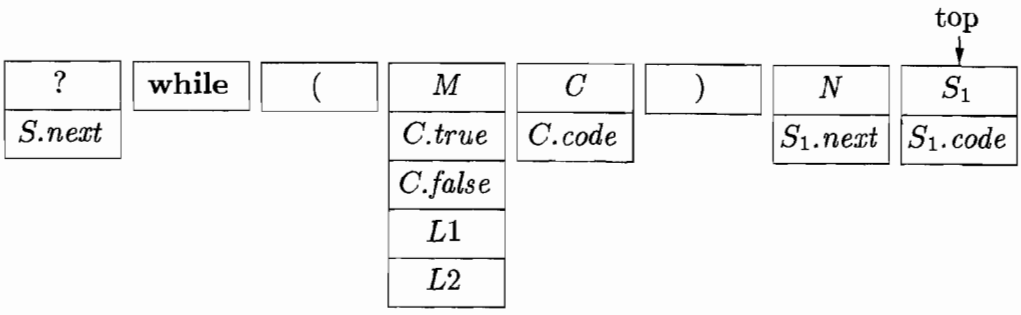

Figure 5.37: Stack just before reduction of the while-production body to S

Continuing with the recognition of the while-statement, the parser should next find ")" on the input, which it pushes onto the stack in a record of its own. At that point, the parser, which knows it is working on a while-statement because the grammar is LL, will reduce ϵ to N. The single piece of data associated with N is the inherited attribute $S_1.next$. Note that this attribute needs

to be in the record for N because that will be just below the record for S_1. The code that is executed to compute the value of $S_1.next$ is

$$S_1.next = stack[top - 3].L1;$$

This action reaches three records below N, which is at the top of stack when the code is executed, and retrieves the value of $L1$.

Next, the parser reduces some prefix of the remaining input to S, which we have consistently referred to as S_1 to distinguish it from the S at the head of the production. The value of $S_1.code$ is computed and appears in the stack record for S_1. This step takes us to the condition that is illustrated in Fig. 5.37.

At this point, the parser will reduce everything from **while** to S_1 to S. The code that is executed during this reduction is:

$$\begin{aligned} tempCode &= \textbf{label} \parallel stack[top - 4].L1 \parallel stack[top - 3].code \parallel \\ &\quad \textbf{label} \parallel stack[top - 4].L2 \parallel stack[top].code; \\ top &= top - 5; \\ stack[top].code &= tempCode; \end{aligned}$$

That is, we construct the value of $S.code$ in a variable $tempCode$. That code is the usual, consisting of the two labels $L1$ and $L2$, the code for C and the code for S_1. The stack is popped, so S appears where **while** was. The value of the code for S is placed in the *code* field of that record, where it can be interpreted as the synthesized attribute $S.code$. Note that we do not show, in any of this discussion, the manipulation of LR states, which must also appear on the stack in the field that we have populated with grammar symbols. □

5.5.5 Exercises for Section 5.5

Exercise 5.5.1: Implement each of your SDD's of Exercise 5.4.4 as a recursive-descent parser in the style of Section 5.5.1.

Exercise 5.5.2: Implement each of your SDD's of Exercise 5.4.4 as a recursive-descent parser in the style of Section 5.5.2.

Exercise 5.5.3: Implement each of your SDD's of Exercise 5.4.4 with an LL parser in the style of Section 5.5.3, with code generated "on the fly."

Exercise 5.5.4: Implement each of your SDD's of Exercise 5.4.4 with an LL parser in the style of Section 5.5.3, but with code (or pointers to the code) stored on the stack.

Exercise 5.5.5: Implement each of your SDD's of Exercise 5.4.4 with an LR parser in the style of Section 5.5.4.

Exercise 5.5.6: Implement your SDD of Exercise 5.2.4 in the style of Section 5.5.1. Would an implementation in the style of Section 5.5.2 be any different?

5.6 Summary of Chapter 5

- *Inherited and Synthesized Attributes*: Syntax-directed definitions may use two kinds of attributes. A synthesized attribute at a parse-tree node is computed from attributes at its children. An inherited attribute at a node is computed from attributes at its parent and/or siblings.

- *Dependency Graphs*: Given a parse tree and an SDD, we draw edges among the attribute instances associated with each parse-tree node to denote that the value of the attribute at the head of the edge is computed in terms of the value of the attribute at the tail of the edge.

- *Cyclic Definitions*: In problematic SDD's, we find that there are some parse trees for which it is impossible to find an order in which we can compute all the attributes at all nodes. These parse trees have cycles in their associated dependency graphs. It is intractable to decide whether an SDD has such circular dependency graphs.

- *S-Attributed Definitions*: In an S-attributed SDD, all attributes are synthesized.

- *L-Attributed Definitions*: In an L-attributed SDD, attributes may be inherited or synthesized. However, inherited attributes at a parse-tree node may depend only on inherited attributes of its parent and on (any) attributes of siblings to its left.

- *Syntax Trees*: Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.

- *Implementing S-Attributed SDD's*: An S-attributed definition can be implemented by an SDT in which all actions are at the end of the production (a "postfix" SDT). The actions compute the synthesized attributes of the production head in terms of synthesized attributes of the symbols in the body. If the underlying grammar is LR, then this SDT can be implemented on the LR parser stack.

- *Eliminating Left Recursion From SDT's*: If an SDT has only side-effects (no attributes are computed), then the standard left-recursion-elimination algorithm for grammars allows us to carry the actions along as if they were terminals. When attributes are computed, we can still eliminate left recursion if the SDT is a postfix SDT.

- *Implementing L-attributed SDD's by Recursive-Descent Parsing*: If we have an L-attributed definition on a top-down parsable grammar, we can build a recursive-descent parser with no backtracking to implement the translation. Inherited attributes become arguments of the functions for their nonterminals, and synthesized attributes are returned by that function.

- *Implementing L-Attributed SDD's on an LL Grammar*: Every L-attributed definition with an underlying LL grammar can be implemented along with the parse. Records to hold the synthesized attributes for a nonterminal are placed below that nonterminal on the stack, while inherited attributes for a nonterminal are stored with that nonterminal on the stack. Action records are also placed on the stack to compute attributes at the appropriate time.

- *Implementing L-Attributed SDD's on an LL Grammar, Bottom-Up*: An L-attributed definition with an underlying LL grammar can be converted to a translation on an LR grammar and the translation performed in connection with a bottom-up parse. The grammar transformation introduces "marker" nonterminals that appear on the bottom-up parser's stack and hold inherited attributes of the nonterminal above it on the stack. Synthesized attributes are kept with their nonterminal on the stack.

5.7 References for Chapter 5

Syntax-directed definitions are a form of inductive definition in which the induction is on the syntactic structure. As such they have long been used informally in mathematics. Their application to programming languages came with the use of a grammar to structure the Algol 60 report.

The idea of a parser that calls for semantic actions can be found in Samelson and Bauer [8] and Brooker and Morris [1]. Irons [2] constructed one of the first syntax-directed compilers, using synthesized attributes. The class of L-attributed definitions comes from [6].

Inherited attributes, dependency graphs, and a test for circularity of SDD's (that is, whether or not there is some parse tree with no order in which the attributes can be computed) are from Knuth [5]. Jazayeri, Ogden, and Rounds [3] showed that testing circularity requires exponential time, as a function of the size of the SDD.

Parser generators such as Yacc [4] (see also the bibliographic notes in Chapter 4) support attribute evaluation during parsing.

The survey by Paakki [7] is a starting point for accessing the extensive literature on syntax-directed definitions and translations.

1. Brooker, R. A. and D. Morris, "A general translation program for phrase structure languages," *J. ACM* **9**:1 (1962), pp. 1–10.

2. Irons, E. T., "A syntax directed compiler for Algol 60," *Comm. ACM* **4**:1 (1961), pp. 51–55.

3. Jazayeri, M., W. F. Odgen, and W. C. Rounds, "The intrinsic exponential complexity of the circularity problem for attribute grammars," *Comm. ACM* **18**:12 (1975), pp. 697–706.