CHAPTER 5

Syntax-Directed Translation

This chapter develops the theme of Section 2.3, the translation of languages guided by context-free grammars. We associate information with a programming language construct by attaching attributes to the grammar symbols representing the construct. Values for attributes are computed by "semantic rules" associated with the grammar productions.

There are two notations for associating semantic rules with productions, syntax-directed definitions and translation schemes. Syntax-directed definitions are high-level specifications for translations. They hide many implementation details and free the user from having to specify explicitly the order in which translation takes place. Translation schemes indicate the order in which semantic rules are to be evaluated, so they allow some implementation details to be shown. We use both notations in Chapter 6 for specifying semantic checking, particularly the determination of types, and in Chapter 8 for generating intermediate code.

Conceptually, with both syntax-directed definitions and translation schemes, we parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse-tree nodes (see Fig. 5.1). Evaluation of the semantic rules may generate code, save information in a symbol table, issue error messages, or perform any other activities. The translation of the token stream is the result obtained by evaluating the semantic rules.

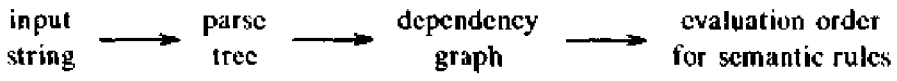


Fig. 5.1. Conceptual view of syntax-directed translation.

An implementation does not have to follow the outline in Fig. 5.1 literally. Special cases of syntax-directed definitions can be implemented in a single pass by evaluating semantic rules during parsing, without explicitly constructing a parse tree or a graph showing dependencies between attributes. Since single-pass implementation is important for compile-time efficiency, much of this

chapter is devoted to studying such special cases. One important subclass, called the "L-attributed" definitions, encompasses virtually all translations that can be performed without explicit construction of a parse tree.

5.1 SYNTAX-DIRECTED DEFINITIONS

A syntax-directed definition is a generalization of a context-free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called the synthesized and inherited attributes of that grammar symbol. If we think of a node for the grammar symbol in a parse tree as a record with fields for holding information, then an attribute corresponds to the name of a field.

An attribute can represent anything we choose: a string, a number, a type, a memory location, or whatever. The value of an attribute at a parse-tree node is defined by a semantic rule associated with the production used at that node. The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree; the value of an inherited attribute is computed from the values of attributes at the siblings and parent of that node.

Semantic rules set up dependencies between attributes that will be represented by a graph. From the dependency graph, we derive an evaluation order for the semantic rules. Evaluation of the semantic rules defines the values of the attributes at the nodes in the parse tree for the input string. A semantic rule may also have side effects, e.g., printing a value or updating a global variable. Of course, an implementation need not explicitly construct a parse tree or a dependency graph; it just has to produce the same output for each input string.

A parse tree showing the values of attributes at each node is called an annotated parse tree. The process of computing the attribute values at the nodes is called *annotating* or *decorating* the parse tree.

Form of a Syntax-Directed Definition

In a syntax-directed definition, each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form $b := f(c_1, c_2, \ldots, c_k)$ where f is a function, and either

1. b is a synthesized attribute of A and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production, or
2. b is an inherited attribute of one of the grammar symbols on the right side of the production, and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production.

In either case, we say that attribute b *depends* on attributes $c_1, c_2, \ldots, c_k$. An *attribute grammar* is a syntax-directed definition in which the functions in semantic rules cannot have side effects.

Functions in semantic rules will often be written as expressions. Occasionally, the only purpose of a semantic rule in a syntax-directed definition is to create a side effect. Such semantic rules are written as procedure calls or program fragments. They can be thought of as rules defining the values of dummy synthesized attributes of the nonterminal on the left side of the associated production; the dummy attribute and the := sign in the semantic rule are not shown.

Example 5.1. The syntax-directed definition in Fig. 5.2 is for a desk-calculator program. This definition associates an integer-valued synthesized attribute called *val* with each of the nonterminals E, T, and F. For each E, T, and F-production, the semantic rule computes the value of attribute *val* for the nonterminal on the left side from the values of *val* for the nonterminals on the right side.

PRODUCTION	SEMANTIC RULES
$L \rightarrow E$ **n**	*print*(*E.val*)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **digit**	$F.val :=$ **digit**.*lexval*

Fig. 5.2. Syntax-directed definition of a simple desk calculator.

The token **digit** has a synthesized attribute *lexval* whose value is assumed to be supplied by the lexical analyzer. The rule associated with the production $L \rightarrow E$ **n** for the starting nonterminal L is just a procedure that prints as output the value of the arithmetic expression generated by E; we can think of this rule as defining a dummy attribute for the nonterminal L. A Yacc specification for this desk calculator was presented in Fig. 4.56 to illustrate translation during LR parsing. □

In a syntax-directed definition, terminals are assumed to have synthesized attributes only, as the definition does not provide any semantic rules for terminals. Values for attributes of terminals are usually supplied by the lexical analyzer, as discussed in Section 3.1. Furthermore, the start symbol is assumed not to have any inherited attributes, unless otherwise stated.

Synthesized Attributes

Synthesized attributes are used extensively in practice. A syntax-directed definition that uses synthesized attributes exclusively is said to be an *S-attributed definition*. A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node

bottom up, from the leaves to the root. Section 5.3 describes how an LR-parser generator can be adapted to mechanically implement an S-attributed definition based on an LR grammar.

Example 5.2. The S-attributed definition in Example 5.1 specifies a desk calculator that reads an input line containing an arithmetic expression involving digits, parentheses, the operators + and *, followed by a newline character **n**, and prints the value of the expression. For example, given the expression 3*5+4 followed by a newline, the program prints the value 19. Figure 5.3 contains an annotated parse tree for the input 3*5+4**n**. The output, printed at the root of the tree, is the value of *E.val* at the first child of the root.

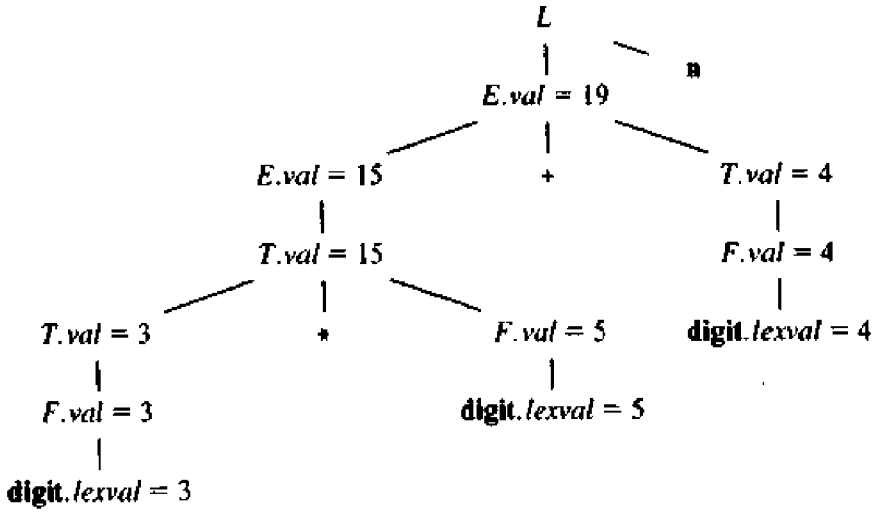


Fig. 5.3. Annotated parse tree for 3*5+4 n.

To see how attribute values are computed, consider the leftmost bottommost interior node, which corresponds to the use of the production $F \rightarrow$ **digit**. The corresponding semantic rule, *F.val* := **digit**.*lexval*, defines the attribute *F.val* at that node to have the value 3 because the value of **digit**.*lexval* at the child of this node is 3. Similarly, at the parent of this *F*-node, the attribute *T.val* has the value 3.

Now consider the node for the production $T \rightarrow T * F$. The value of the attribute *T.val* at this node is defined by

PRODUCTION	SEMANTIC RULE
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$

When we apply the semantic rule at this node, $T_1.val$ has the value 3 from the left child and *F.val* the value 5 from the right child. Thus, *T.val* acquires the value 15 at this node.

The rule associated with the production for the starting nonterminal $L \rightarrow E$ **n** prints the value of the expression generated by *E*. □

Inherited Attributes

An inherited attribute is one whose value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of that node. Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears. For example, we can use an inherited attribute to keep track of whether an identifier appears on the left or right side of an assignment in order to decide whether the address or the value of the identifier is needed. Although it is always possible to rewrite a syntax-directed definition to use only synthesized attributes, it is often more natural to use syntax-directed definitions with inherited attributes.

In the following example, an inherited attribute distributes type information to the various identifiers in a declaration.

Example 5.3. A declaration generated by the nonterminal D in the syntax-directed definition in Fig. 5.4 consists of the keyword **int** or **real**, followed by a list of identifiers. The nonterminal T has a synthesized attribute *type*, whose value is determined by the keyword in the declaration. The semantic rule $L.in := T.type$, associated with production $D \rightarrow T\,L$, sets inherited attribute $L.in$ to the type in the declaration. The rules then pass this type down the parse tree using the inherited attribute $L.in$. Rules associated with the productions for L call procedure *addtype* to add the type of each identifier to its entry in the symbol table (pointed to by attribute *entry*).

PRODUCTION	SEMANTIC RULES
$D \rightarrow T\,L$	$L.in := T.type$
$T \rightarrow$ **int**	$T.type := integer$
$T \rightarrow$ **real**	$T.type := real$
$L \rightarrow L_1$, **id**	$L_1.in := L.in$ *addtype*(**id**.*entry*, $L.in$)
$L \rightarrow$ **id**	*addtype*(**id**.*entry*, $L.in$)

Fig. 5.4. Syntax-directed definition with inherited attribute $L.in$.

Figure 5.5 shows an annotated parse tree for the sentence **real** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$. The value of $L.in$ at the three L-nodes gives the type of the identifiers $\mathbf{id}_1$, $\mathbf{id}_2$, and $\mathbf{id}_3$. These values are determined by computing the value of the attribute $T.type$ at the left child of the root and then evaluating $L.in$ top-down at the three L-nodes in the right subtree of the root. At each L-node we also call the procedure *addtype* to insert into the symbol table the fact that the identifier at the right child of this node has type real. □

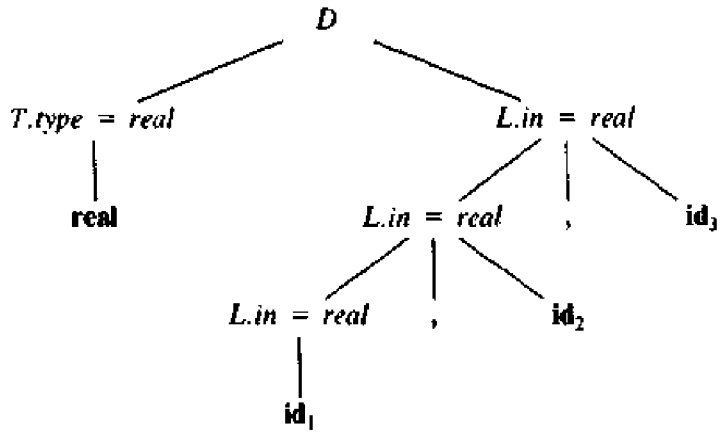


Fig. 5.5. Parse tree with inherited attribute *in* at each node labeled *L*.

Dependency Graphs

If an attribute b at a node in a parse tree depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c. The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a *dependency graph*.

Before constructing a dependency graph for a parse tree, we put each semantic rule into the form $b := f(c_1, c_2, \ldots, c_k)$, by introducing a dummy synthesized attribute b for each semantic rule that consists of a procedure call. The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c. In more detail, the dependency graph for a given parse tree is constructed as follows.

```
for each node n in the parse tree do
    for each attribute a of the grammar symbol at n do
        construct a node in the dependency graph for a;
for each node n in the parse tree do
    for each semantic rule b := f(c1, c2, . . . , ck)
            associated with the production used at n do
        for i := 1 to k do
            construct an edge from the node for ci to the node for b;
```

For example, suppose $A.a := f(X.x, Y.y)$ is a semantic rule for the production $A \rightarrow XY$. This rule defines a synthesized attribute $A.a$ that depends on the attributes $X.x$ and $Y.y$. If this production is used in the parse tree, then there will be three nodes $A.a$, $X.x$, and $Y.y$ in the dependency graph with an edge to $A.a$ from $X.x$ since $A.a$ depends on $X.x$, and an edge to $A.a$ from $Y.y$ since $A.a$ also depends on $Y.y$.

If the production $A \rightarrow XY$ has the semantic rule $X.i := g(A.a, Y.y)$ associated with it, then there will be an edge to $X.i$ from $A.a$ and also an edge to $X.i$ from $Y.y$, since $X.i$ depends on both $A.a$ and $Y.y$.

Example 5.4. Whenever the following production is used in a parse tree, we add the edges shown in Fig. 5.6 to the dependency graph.

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + E_2$	$E.val := E_1.val + E_2.val$

The three nodes of the dependency graph marked by • represent the synthesized attributes $E.val$, $E_1.val$, and $E_2.val$ at the corresponding nodes in the parse tree. The edge to $E.val$ from $E_1.val$ shows that $E.val$ depends on $E_1.val$ and the edge to $E.val$ from $E_2.val$ shows that $E.val$ also depends on $E_2.val$. The dotted lines represent the parse tree and are not part of the dependency graph. □

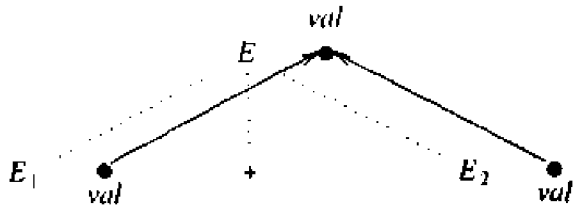


Fig. 5.6. $E.val$ is synthesized from $E_1.val$ and $E_2.val$.

Example 5.5. Figure 5.7 shows the dependency graph for the parse tree in Fig. 5.5. Nodes in the dependency graphs are marked by numbers; these numbers will be used below. There is an edge to node 5 for $L.in$ from node 4 for $T.type$ because the inherited attribute $L.in$ depends on the attribute $T.type$ according to the semantic rule $L.in := T.type$ for the production $D \rightarrow TL$. The two downward edges into nodes 7 and 9 arise because $L_1.in$ depends on $L.in$ according to the semantic rule $L_1.in := L.in$ for the production $L \rightarrow L_1$, **id**. Each of the semantic rules *addtype* (**id**.*entry*, $L.in$) associated with the L-productions leads to the creation of a dummy attribute. Nodes 6, 8, and 10 are constructed for these dummy attributes. □

Evaluation Order

A *topological sort* of a directed acyclic graph is any ordering $m_1, m_2, \ldots, m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes; that is, if $m_i \rightarrow m_j$ is an edge from m_i to m_j, then m_i appears before m_j in the ordering.

Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated. That is, in the topological sort, the dependent attributes $c_1, c_2, \ldots, c_k$ in a semantic rule $b := f(c_1, c_2, \ldots, c_k)$ are available at a node before f is evaluated.

The translation specified by a syntax-directed definition can be made precise

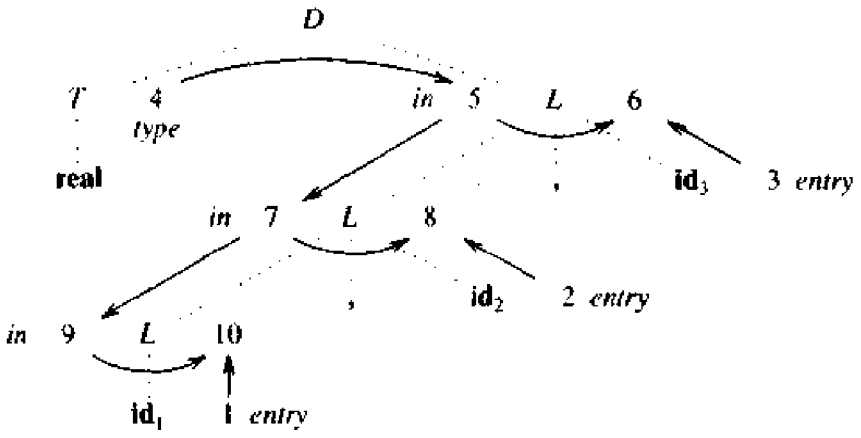


Fig. 5.7. Dependency graph for parse tree of Fig. 5.5.

as follows. The underlying grammar is used to construct a parse tree for the input. The dependency graph is constructed as discussed above. From a topological sort of the dependency graph, we obtain an evaluation order for the semantic rules. Evaluation of the semantic rules in this order yields the translation of the input string.

Example 5.6. Each of the edges in the dependency graph in Fig. 5.7 goes from a lower-numbered node to a higher-numbered node. Hence, a topological sort of the dependency graph is obtained by writing down the nodes in the order of their numbers. From this topological sort, we obtain the following program. We write a_n for the attribute associated with the node numbered n in the dependency graph.

$a_4 := real;$
$a_5 := a_4;$
$addtype(\mathbf{id}_3.entry, a_5);$
$a_7 := a_5;$
$addtype(\mathbf{id}_2.entry, a_7);$
$a_9 := a_7;$
$addtype(\mathbf{id}_1.entry, a_9);$

Evaluating these semantic rules stores the type *real* in the symbol-table entry for each identifier. □

Several methods have been proposed for evaluating semantic rules:

1. *Parse-tree methods.* At compile time, these methods obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input. These methods will fail to find an evaluation order only if the dependency graph for the particular parse tree under consideration has a cycle.

2. *Rule-based methods.* At compiler-construction time, the semantic rules associated with productions are analyzed, either by hand, or by a specialized tool. For each production, the order in which the attributes associated with that production are evaluated is predetermined at compiler-construction time.

3. *Oblivious methods.* An evaluation order is chosen without considering the semantic rules. For example, if translation takes place during parsing, then the order of evaluation is forced by the parsing method, independent of the semantic rules. An oblivious evaluation order restricts the class of syntax-directed definitions that can be implemented.

Rule-based and oblivious methods need not explicitly construct the dependency graph at compile time, so they can be more efficient in their use of compile time and space.

A syntax-directed definition is said to be *circular* if the dependency graph for some parse tree generated by its grammar has a cycle. Section 5.10 discusses how to test a syntax-directed definition for circularity.

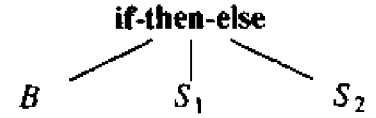
5.2 CONSTRUCTION OF SYNTAX TREES

In this section, we show how syntax-directed definitions can be used to specify the construction of syntax trees and other graphical representations of language constructs.

The use of syntax trees as an intermediate representation allows translation to be decoupled from parsing. Translation routines that are invoked during parsing must live with two kinds of restrictions. First, a grammar that is suitable for parsing may not reflect the natural hierarchical structure of the constructs in the language. For example, a grammar for Fortran may view a subroutine as consisting simply of a list of statements. However, analysis of the subroutine may be easier if we use a tree representation that reflects the nesting of DO loops. Second, the parsing method constrains the order in which nodes in a parse tree are considered. This order may not match the order in which information about a construct becomes available. For this reason, compilers for C usually construct syntax trees for declarations.

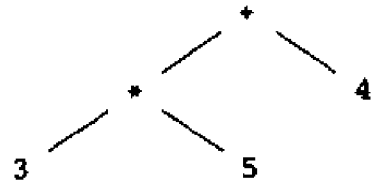
Syntax Trees

An (abstract) syntax tree is a condensed form of parse tree useful for representing language constructs. The production $S \rightarrow$ **if** B **then** S_1 **else** S_2 might appear in a syntax tree as



In a syntax tree, operators and keywords do not appear as leaves, but rather are associated with the interior node that would be the parent of those leaves

in the parse tree. Another simplification found in syntax trees is that chains of single productions may be collapsed; the parse tree of Fig. 5.3 becomes the syntax tree



Syntax-directed translation can be based on syntax trees as well as parse trees. The approach is the same in each case; we attach attributes to the nodes as in a parse tree.

Constructing Syntax Trees for Expressions

The construction of a syntax tree for an expression is similar to the translation of the expression into postfix form. We construct subtrees for the subexpressions by creating a node for each operator and operand. The children of an operator node are the roots of the nodes representing the subexpressions constituting the operands of that operator.

Each node in a syntax tree can be implemented as a record with several fields. In the node for an operator, one field identifies the operator and the remaining fields contain pointers to the nodes for the operands. The operator is often called the *label* of the node. When used for translation, the nodes in a syntax tree may have additional fields to hold the values (or pointers to values) of attributes attached to the node. In this section, we use the following functions to create the nodes of syntax trees for expressions with binary operators. Each function returns a pointer to a newly created node.

1. *mknode*(*op*, *left*, *right*) creates an operator node with label *op* and two fields containing pointers to *left* and *right*.
2. *mkleaf* (**id**, *entry*) creates an identifier node with label **id** and a field containing *entry*, a pointer to the symbol-table entry for the identifier.
3. *mkleaf* (**num**, *val*) creates a number node with label **num** and a field containing *val*, the value of the number.

Example 5.7. The following sequence of functions calls creates the syntax tree for the expression a - 4 + c in Fig. 5.8. In this sequence, $p_1, p_2, \ldots, p_5$ are pointers to nodes, and *entrya* and *entryc* are pointers to the symbol-table entries for identifiers a and c, respectively.

(1) p_1 := *mkleaf* (**id**, *entrya*);
(2) p_2 := *mkleaf* (**num**, 4);
(3) p_3 := *mknode*('-', p_1, p_2);
(4) p_4 := *mkleaf* (**id**, *entryc*);
(5) p_5 := *mknode*('+', p_3, p_4);

The tree is constructed bottom up. The function calls *mkleaf* (**id**, *entrya*) and *mkleaf* (**num**, 4) construct the leaves for a and 4; the pointers to these

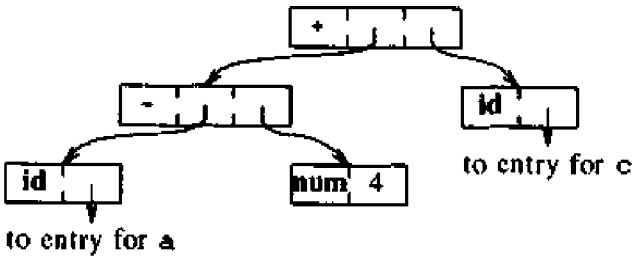


Fig. 5.8. Syntax tree for a-4+c.

nodes are saved using p_1 and p_2. The call *mknode*('-', p_1, p_2) then constructs the interior node with the leaves for a and 4 as children. After two more steps, p_5 is left pointing to the root. □

A Syntax-Directed Definition for Constructing Syntax Trees

Figure 5.9 contains an S-attributed definition for constructing a syntax tree for an expression containing the operators + and −. It uses the underlying productions of the grammar to schedule the calls of the functions *mknode* and *mkleaf* to construct the tree. The synthesized attribute *nptr* for E and T keeps track of the pointers returned by the function calls.

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 + T$	$E.nptr := mknode('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr := mknode('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow (E)$	$T.nptr := E.nptr$
$T \rightarrow$ **id**	$T.nptr := mkleaf$ (**id**, **id**.*entry*)
$T \rightarrow$ **num**	$T.nptr := mkleaf$ (**num**, **num**.*val*)

Fig. 5.9. Syntax-directed definition for constructing a syntax tree for an expression.

Example 5.8. An annotated parse tree depicting the construction of a syntax tree for the expression a-4+c is shown in Fig. 5.10. The parse tree is shown dotted. The parse-tree nodes labeled by the nonterminals E and T use the synthesized attribute *nptr* to hold a pointer to the syntax-tree node for the expression represented by the nonterminal.

The semantic rules associated with the productions $T \rightarrow$ **id** and $T \rightarrow$ **num** define attribute *T.nptr* to be a pointer to a new leaf for an identifier and a number, respectively. Attributes **id**.*entry* and **num**.*val* are the lexical values assumed to be returned by the lexical analyzer with the tokens **id** and **num**.

In Fig. 5.10, when an expression E is a single term, corresponding to a use

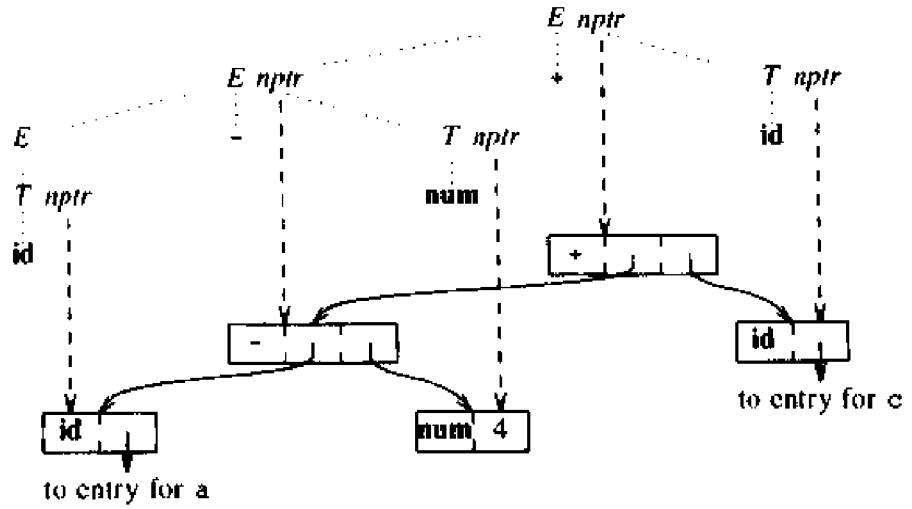


Fig. 5.10. Construction of a syntax-tree for a-4+c.

of the production $E \rightarrow T$, the attribute $E.nptr$ gets the value of $T.nptr$. When the semantic rule $E.nptr := mknode('-', E_1.nptr, T.nptr)$ associated with the production $E \rightarrow E_1 - T$ is invoked, previous rules have set $E_1.nptr$ and $T.nptr$ to be pointers to the leaves for a and 4, respectively.

In interpreting Fig. 5.10, it is important to realize that the lower tree, formed from records is a "real" syntax tree that constitutes the output, while the dotted tree above is the parse tree, which may exist only in a figurative sense. In the next section, we show how an S-attributed definition can be simply implemented using the stack of a bottom-up parser to keep track of attribute values. In fact, with this implementation, the node-building functions are invoked in the same order as in Example 5.7. □

Directed Acyclic Graphs for Expressions

A directed acyclic graph (hereafter called a *dag*) for an expression identifies the common subexpressions in the expression. Like a syntax tree, a dag has a node for every subexpression of the expression; an interior node represents an operator and its children represent its operands. The difference is that a node in a dag representing a common subexpression has more than one "parent;" in a syntax tree, the common subexpression would be represented as a duplicated subtree.

Figure 5.11 contains a dag for the expression

```
a + a * ( b - c ) + ( b - c ) * d
```

The leaf for a has two parents because a is common to the two subexpressions a and a*(b-c). Likewise, both occurrences of the common subexpression b-c are represented by the same node, which also has two parents.

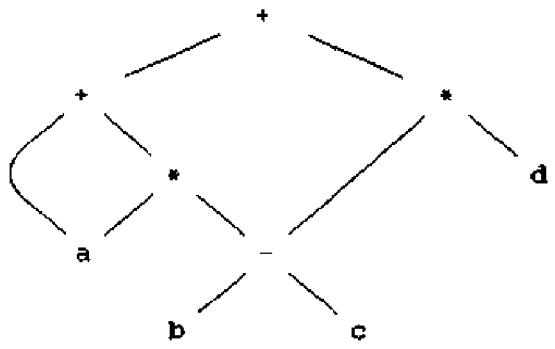


Fig. 5.11. Dag for the expression a+a*(b-c)+(b-c)*d.

The syntax-directed definition of Fig. 5.9 will construct a dag instead of a syntax tree if we modify the operations for constructing nodes. A dag is obtained if the function constructing a node first checks to see whether an identical node already exists. For example, before constructing a new node with label *op* and fields with pointers to *left* and *right*, *mknode*(*op*, *left*, *right*) can check whether such a node has already been constructed. If so, *mknode*(*op*, *left*, *right*) can return a pointer to the previously constructed node. The leaf-constructing functions *mkleaf* can behave similarly.

Example 5.9. The sequence of instructions in Fig. 5.12 constructs the dag in Fig. 5.11, provided *mknode* and *mkleaf* create new nodes only when necessary, returning pointers to existing nodes with the correct label and children whenever possible. In Fig. 5.12, *a*, *b*, *c*, and *d* point to the symbol-table entries for identifiers a, b, c, and d.

(1) p_1 := *mkleaf* (**id**, *a*);
(2) p_2 := *mkleaf* (**id**, *a*);
(3) p_3 := *mkleaf* (**id**, *b*);
(4) p_4 := *mkleaf* (**id**, *c*);
(5) p_5 := *mknode*('-', p_3, p_4);
(6) p_6 := *mknode*('*', p_2, p_5);
(7) p_7 := *mknode*('+', p_1, p_6);
(8) p_8 := *mkleaf* (**id**, *b*);
(9) p_9 := *mkleaf* (**id**, *c*);
(10) p_{10} := *mknode*('-', p_8, p_9);
(11) p_{11} := *mkleaf* (**id**, *d*);
(12) p_{12} := *mknode*('*', p_{10}, p_{11});
(13) p_{13} := *mknode*('+', p_7, p_{12});

Fig. 5.12. Instructions for constructing the dag of Fig. 5.11.

When the call *mkleaf* (**id**, *a*) is repeated on line 2, the node constructed by the previous call *mkleaf* (**id**, *a*) is returned, so $p_1 = p_2$. Similarly, the nodes returned on lines 8 and 9 are the same as those returned on lines 3 and 4, respectively. Hence, the node returned on line 10 must be the same one constructed by the call of *mknode* on line 5. □

In many applications, nodes are implemented as records stored in an array, as in Fig. 5.13. In the figure, each record has a label field that determines the

nature of the node. We can refer to a node by its index or position in the array. The integer index of a node is often called a *value number* for historical reasons. For example, using value numbers, we can say node 3 has label +, its left child is node 1, and its right child is node 2. The following algorithm can be used to create nodes for a dag representation of an expression.

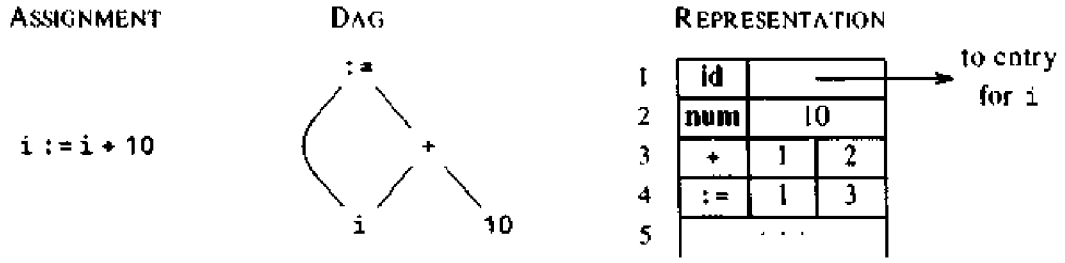


Fig. 5.13. Nodes in a dag for `i := i + 10` allocated from an array.

Algorithm 5.1. Value-number method for constructing a node in a dag.

Suppose that nodes are stored in an array, as in Fig. 5.13, and that each node is referred to by its value number. Let the *signature* of an operator node be a triple $<op, l, r>$ consisting of its label op, left child l, and right child r.

Input. Label op, node l, and node r.

Output. A node with signature $<op, l, r>$.

Method. Search the array for a node m with label op, left child l, and right child r. If there is such a node, return m; otherwise, create a new node n with label op, left child l, right child r, and return n.

An obvious way to determine if node m is already in the array is to keep all previously created nodes on a list and to check each node on the list to see if it has the desired signature. The search for m can be made more efficient by using k lists, called buckets, and using a hashing function h to determine which bucket to search.[1]

The hash function h computes the number of a bucket from the value of op, l, and r. It will always return the same bucket number, given the same arguments. If m is not in the bucket $h(op, l, r)$, then a new node n is created and added to this bucket, so subsequent searches will find it there. Several signatures may hash into the same bucket number, but in practice we expect each bucket to contain a small number of nodes.

Each bucket can be implemented as a linked list as shown in Fig. 5.14.

[1] Any data structure that implements dictionaries in the sense of Aho, Hopcroft, and Ullman [1983] suffices. The important property of the structure is that given a key, i.e., a label op and two nodes l and r, we can rapidly obtain a node m with signature $<op, l, r>$, or determine that none exists.

Each cell in a linked list represents a node. The bucket headers, consisting of pointers to the first cell in a list, are stored in an array. The bucket number returned by $h(op, l, r)$ is an index into this array of bucket headers.

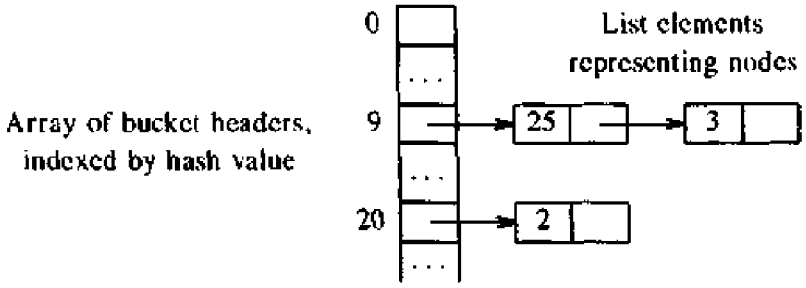


Fig. 5.14. Data structure for searching buckets.

This algorithm can be adapted to apply to nodes that are not allocated sequentially from an array. In many compilers, nodes are allocated as they are needed, to avoid preallocating an array that may hold too many nodes most of the time and not enough nodes some of the time. In this case, we cannot assume that nodes are in sequential storage, so we have to use pointers to refer to nodes. If the hash function can be made to compute the bucket number from a label and pointers to children, then we can use pointers to nodes instead of value numbers. Otherwise, we can number the nodes in any way and use this number as the value number of the node. □

Dags can also be used to represent sets of expressions, since a dag can have more than one root. In Chapters 9 and 10, the computations performed by a sequence of assignment statements will be represented as a dag.

5.3 BOTTOM-UP EVALUATION OF S-ATTRIBUTED DEFINITIONS

Now that we have seen how to use syntax-directed definitions to specify translations, we can begin to study how to implement translators for them. A translator for an arbitrary syntax-directed definition can be difficult to build. However, there are large classes of useful syntax-directed definitions for which it is easy to construct translators. In this section, we examine one such class: the S-attributed definitions, that is, the syntax-directed definitions with only synthesized attributes. The following sections consider the implementation of definitions that have inherited attributes as well.

Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed. The parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack. Whenever a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production. This section shows how the parser stack can be extended to hold the values of these synthesized attributes. We shall see in

Section 5.6 that this implementation also supports some inherited attributes.

Only synthesized attributes appear in the syntax-directed definition in Fig. 5.9 for constructing the syntax tree for an expression. The approach of this section can therefore be applied to construct syntax trees during bottom-up parsing. As we shall see in Section 5.5, the translation of expressions during top-down parsing often uses inherited attributes. We therefore defer translation during top-down parsing until after "left-to-right" dependencies are examined in the next section.

Synthesized Attributes on the Parser Stack

A translator for an S-attributed definition can often be implemented with the help of an LR-parser generator, such as the one discussed in Section 4.9. From an S-attributed definition, the parser generator can construct a translator that evaluates attributes as it parses the input.

A bottom-up parser uses a stack to hold information about subtrees that have been parsed. We can use extra fields in the parser stack to hold the values of synthesized attributes. Figure 5.15 shows an example of a parser stack with space for one attribute value. Let us suppose, as in the figure, that the stack is implemented by a pair of arrays *state* and *val*. Each *state* entry is a pointer (or index) to an LR(1) parsing table. (Note that the grammar symbol is implicit in the state and need not be stored in the stack.) It is convenient, however, to refer to the state by the unique grammar symbol that it covers when placed on the parsing stack as described in Section 4.7. If the ith *state* symbol is A, then $val[i]$ will hold the value of the attribute associated with the parse tree node corresponding to this A.

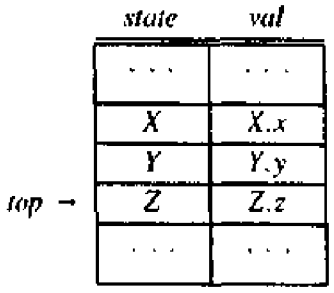


Fig. 5.15. Parser stack with a field for synthesized attributes.

The current top of the stack is indicated by the pointer *top*. We assume that synthesized attributes are evaluated just before each reduction. Suppose the semantic rule $A.a := f(X.x, Y.y, Z.z)$ is associated with the production $A \rightarrow XYZ$. Before XYZ is reduced to A, the value of the attribute $Z.z$ is in $val[top]$, that of $Y.y$ in $val[top-1]$, and that of $X.x$ in $val[top-2]$. If a symbol has no attribute, then the corresponding entry in the *val* array is undefined. After the reduction, *top* is decremented by 2, the state covering A is

put in *state*[*top*] (i.e., where X was), and the value of the synthesized attribute $A.a$ is put in *val*[*top*].

Example 5.10. Consider again the syntax-directed definition of the desk calculator in Fig. 5.2. The synthesized attributes in the annotated parse tree of Fig. 5.3 can be evaluated by an LR parser during a bottom-up parse of the input line 3*5+4**n**. As before, we assume that the lexical analyzer supplies the value of attribute **digit**.*lexval*, which is the numeric value of each token representing a digit. When the parser shifts a **digit** onto the stack, the token **digit** is placed in *state*[*top*] and its attribute value is placed in *val*[*top*].

We can use the techniques of Section 4.7 to construct an LR parser for the underlying grammar. To evaluate attributes, we modify the parser to execute the code fragments shown in Fig. 5.16 just before making the appropriate reduction. Note that we can associate attribute evaluation with reductions, because each reduction determines the production to be applied. The code fragments have been obtained from the semantic rules in Fig. 5.2 by replacing each attribute by a position in the *val* array.

PRODUCTION	CODE FRAGMENT
$L \rightarrow E$ **n**	*print*(*val*[*top*])
$E \rightarrow E_1 + T$	*val*[*ntop*] := *val*[*top* − 2] + *val*[*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val*[*ntop*] := *val*[*top* − 2] × *val*[*top*]
$T \rightarrow F$	
$F \rightarrow (\ E\ )$	*val*[*ntop*] := *val*[*top* − 1]
$F \rightarrow$ **digit**	

Fig. 5.16. Implementation of a desk calculator with an LR parser.

The code fragments do not show how the variables *top* and *ntop* are managed. When a production with r symbols on the right side is reduced, the value of *ntop* is set to $top - r + 1$. After each code fragment is executed, *top* is set to *ntop*.

Figure 5.17 shows the sequence of moves made by the parser on input 3*5+4**n**. The contents of the *state* and *val* fields of the parsing stack are shown after each move. We again take the liberty of replacing stack states by their corresponding grammar symbols. We take the further liberty of showing, instead of token **digit**, the actual input digit.

Consider the sequence of events on seeing the input symbol 3. In the first move, the parser shifts the state corresponding to the token **digit** (whose attribute value is 3) onto the stack. (The state is represented by 3 and the value 3 is in the *val* field.) On the second move, the parser reduces by the production $F \rightarrow$ **digit** and implements the semantic rule $F.val :=$ **digit**.*lexval*. On the third move the parser reduces by $T \rightarrow F$. No code fragment is associated with

INPUT	*state*	*val*	PRODUCTION USED
3*5+4 n	_	_	
*5+4 n	3	3	
*5+4 n	*F*	3	*F* → **digit**
*5+4 n	*T*	3	*T* → *F*
5+4 n	*T* *	3 _	
+4 n	*T* * 5	3 _ 5	
+4 n	*T* * *F*	3 _ 5	*F* → **digit**
+4 n	*T*	15	*T* → *T* * *F*
+4 n	*E*	15	*E* → *T*
4 n	*E* +	15 _	
n	*E* + 4	15 _ 4	
n	*E* + *F*	15 _ 4	*F* → **digit**
n	*E* + *T*	15 _ 4	*T* → *F*
n	*E*	19	*E* → *E* + *T*
	E n	19 _	
	L	19	*L* → *E* n

Fig. 5.17. Moves made by translator on input 3*5+4 n.

this production, so the *val* array is left unchanged. Note that after each reduction the top of the *val* stack contains the attribute value associated with the left side of the reducing production. □

In the implementation sketched above, code fragments are executed just before a reduction takes place. Reductions provide a "hook" on which actions consisting of arbitrary code fragments can be hung. That is, we can allow the user to associate an action with a production that is executed when a reduction according to that production takes place. Translation schemes considered in the next section provide a notation for interleaving actions with parsing. In Section 5.6, we shall see how a larger class of syntax-directed definitions can be implemented during bottom-up parsing.

5.4 L-ATTRIBUTED DEFINITIONS

When translation takes place during parsing, the order of evaluation of attributes is linked to the order in which nodes of a parse tree are "created" by the parsing method. A natural order that characterizes many top-down and bottom-up translation methods is the one obtained by applying the procedure *dfvisit* in Fig. 5.18 to the root of a parse tree. We call this evaluation order the *depth-first order*. Even if the parse tree is not actually constructed, it is useful to study translation during parsing by considering depth-first evaluation of attributes at the nodes of a parse tree.

```
procedure dfvisit(n: node);
begin
    for each child m of n, from left to right do begin
        evaluate inherited attributes of m;
        dfvisit(m)
    end;
    evaluate synthesized attributes of n
end
```

Fig. 5.18. Depth-first evaluation order for attributes in a parse tree.

We now introduce a class of syntax-directed definitions, called L-attributed definitions, whose attributes can always be evaluated in depth-first order. (The L is for "left," because attribute information appears to flow from left to right.) Implementation of progressively larger classes of L-attributed definitions is covered in the next three sections of this chapter. L-attributed definitions include all syntax-directed definitions based on LL(1) grammars; Section 5.5 gives a method for implementing such definitions in a single pass using predictive parsing methods. A larger class of L-attributed definitions is implemented in Section 5.6 during bottom-up parsing, by extending the translation methods of Section 5.3. A general method for implementing all L-attributed definitions is outlined in Section 5.7.

L-Attributed Definitions

A syntax-directed definition is *L-attributed* if each inherited attribute of X_j, $1 \leq j \leq n$, on the right side of $A \rightarrow X_1 X_2 \cdots X_n$, depends only on

1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$ to the left of X_j in the production and
2. the inherited attributes of A.

Note that every S-attributed definition is L-attributed, because the restrictions (1) and (2) apply only to inherited attributes.

Example 5.11. The syntax-directed definition in Fig. 5.19 is not L-attributed because the inherited attribute $Q.i$ of the grammar symbol Q depends on the attribute $R.s$ of the grammar symbol to its right. Other examples of definitions that are not L-attributed can be found in Sections 5.8 and 5.9. □

Translation Schemes

A translation scheme is a context-free grammar in which attributes are associated with the grammar symbols and semantic actions enclosed between braces {} are inserted within the right sides of productions, as in Section 2.3. We

PRODUCTION	SEMANTIC RULES
$A \rightarrow L\ M$	$L.i := l(A.i)$
	$M.i := m(L.s)$
	$A.s := f(M.s)$
$A \rightarrow Q\ R$	$R.i := r(A.i)$
	$Q.i := q(R.s)$
	$A.s := f(Q.s)$

Fig. 5.19. A non-L-attributed syntax-directed definition.

shall use translation schemes in this chapter as a useful notation for specifying translation during parsing.

The translation schemes considered in this chapter can have both synthesized and inherited attributes. In the simple translation schemes considered in Chapter 2, the attributes were of string type, one for each symbol, and for every production $A \rightarrow X_1 \cdots X_n$, the semantic rule formed the string for A by concatenating the strings for $X_1, \ldots, X_n$, in order, with some optional additional strings in between. We saw that we could perform the translation by simply printing the literal strings in the order they appeared in the semantic rules.

Example 5.12. Here is a simple translation scheme that maps infix expressions with addition and subtraction into corresponding postfix expressions. It is a slight reworking of the translation scheme (2.14) from Chapter 2.

$$\begin{array}{lcl} E & \rightarrow & T\ R \\ R & \rightarrow & \textbf{addop}\ T\ \{print(\textbf{addop}.lexeme)\}\ R_1 \mid \epsilon \\ T & \rightarrow & \textbf{num}\ \{print(\textbf{num}.val)\} \end{array} \tag{5.1}$$

Figure 5.20 shows the parse tree for the input 9-5+2 with each semantic action attached as the appropriate child of the node corresponding to the left side of their production. In effect, we treat actions as though they are terminal symbols, a viewpoint that is a convenient mnemonic for establishing when the actions are to be executed. We have taken the liberty of showing the actual numbers and additive operator in place of the tokens **num** and **addop**. When performed in depth-first order, the actions in Fig. 5.20 print the output 95-2+. □

When designing a translation scheme, we must observe some restrictions to ensure that an attribute value is available when an action refers to it. These restrictions, motivated by L-attributed definitions, ensure that an action does not refer to an attribute that has not yet been computed.

The easiest case occurs when only synthesized attributes are needed. For this case, we can construct the translation scheme by creating an action

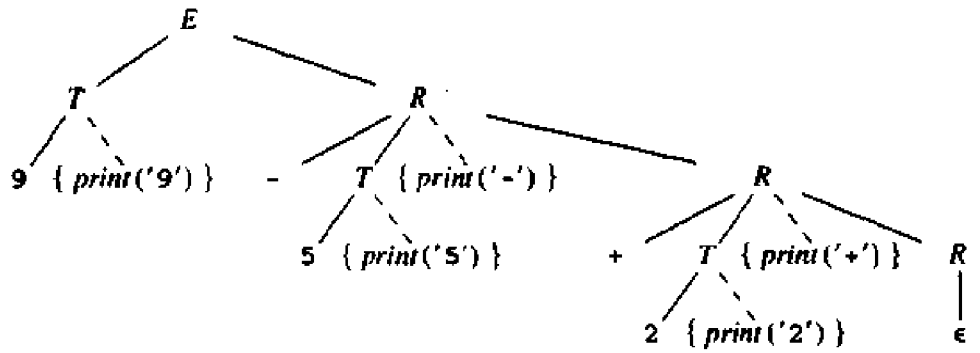


Fig. 5.20. Parse tree for 9-5+2 showing actions.

consisting of an assignment for each semantic rule, and placing this action at the end of the right side of the associated production. For example, the production and semantic rule

PRODUCTION	SEMANTIC RULE
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$

yield the following production and semantic action

$$T \rightarrow T_1 * F \ \{ T.val := T_1.val \times F.val \}$$

If we have both inherited and synthesized attributes, we must be more careful:

1. An inherited attribute for a symbol on the right side of a production must be computed in an action before that symbol.
2. An action must not refer to a synthesized attribute of a symbol to the right of the action.
3. A synthesized attribute for the nonterminal on the left can only be computed after all attributes it references have been computed. The action computing such attributes can usually be placed at the end of the right side of the production.

In the next two sections, we show how a translation scheme satisfying these three requirements can be implemented by generalizations of top-down and bottom-up parsers.

The following translation scheme does not satisfy the first of these three requirements.

$$S \rightarrow A_1 A_2 \quad \{ A_1.in := 1; A_2.in := 2 \}$$
$$A \rightarrow a \quad \{ print(A.in) \}$$

We find that the inherited attribute $A.in$ in the second production is not defined when an attempt is made to print its value during a depth-first traversal of the parse tree for the input string aa. That is, a depth-first traversal

starts at S and visits the subtrees for A_1 and A_2 before the values of $A_1.in$ and $A_2.in$ are set. If the action defining the values of $A_1.in$ and $A_2.in$ is embedded before the A's on the right side of $S \rightarrow A_1 A_2$, instead of after, then $A.in$ will be defined each time $print(A.in)$ occurs.

It is always possible to start with an L-attributed syntax-directed definition and construct a translation scheme that satisfies the three requirements above. The next example illustrates this construction. It is based on the mathematics-formatting language EQN, described briefly in Section 1.2. Given the input

```
E sub 1 .val
```

EQN places E, 1, and $.val$ in the relative positions and sizes shown in Fig. 5.21. Notice that the subscript 1 is printed in a smaller size and font, and is moved down relative to E and $.val$.

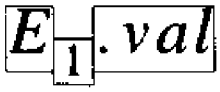


Fig. 5.21. Syntax-directed placement of boxes.

Example 5.13. From the L-attributed definition in Fig. 5.22 we shall construct the translation scheme in Fig. 5.23. In the figures, the nonterminal B (for box) represents a formula. The production $B \rightarrow B\ B$ represents the juxtaposition of two boxes, and $B \rightarrow B$ **sub** B represents the placement of the second subscript box in a smaller size than the first box in the proper relative position for a subscript.

The inherited attribute ps (for point size) affects the height of a formula. The rule for production $B \rightarrow$ **text** causes the normalized height of the text to be multiplied by the point size to get the actual height of the text. The attribute h of **text** is obtained by table lookup, given the character represented by the token **text**. When production $B \rightarrow B_1 B_2$ is applied, B_1 and B_2 inherit the point size from B by copy rules. The height of B, represented by the synthesized attribute ht, is the maximum of the heights of B_1 and B_2.

When production $B \rightarrow B_1$ **sub** B_2 is used, the function *shrink* lowers the point size of B_2 by 30%. The function *disp* allows for the downward displacement of the box B_2 as it computes the height of B. The rules that generate the actual typesetter commands as output are not shown.

The definition in Fig. 5.22 is L-attributed. The only inherited attribute is ps of the nonterminal B. Each semantic rule defines ps only in terms of the inherited attribute of the nonterminal on the left of the production. Hence, the definition is L-attributed.

The translation scheme in Fig. 5.23 is obtained by inserting assignments corresponding to the semantic rules in Fig. 5.22 into the productions,

PRODUCTION	SEMANTIC RULES
$S \rightarrow B$	$B.ps := 10$
	$S.ht := B.ht$
$B \rightarrow B_1\ B_2$	$B_1.ps := B.ps$
	$B_2.ps := B.ps$
	$B.ht := max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ **sub** B_2	$B_1.ps := B.ps$
	$B_2.ps := shrink(B.ps)$
	$B.ht := disp(B_1.ht, B_2.ht)$
$B \rightarrow$ **text**	$B.ht :=$ **text**.$h \times B.ps$

Fig. 5.22. Syntax-directed definition for size and height of boxes.

$S \rightarrow$ $\{ B.ps := 10 \}$
$\quad B$ $\{ S.ht := B.ht \}$

$B \rightarrow$ $\{ B_1.ps := B.ps \}$
$\quad B_1$ $\{ B_2.ps := B.ps \}$
$\quad B_2$ $\{ B.ht := max(B_1.ht, B_2.ht) \}$

$B \rightarrow$ $\{ B_1.ps := B.ps \}$
$\quad B_1$
$\quad$ **sub** $\{ B_2.ps := shrink(B.ps) \}$
$\quad B_2$ $\{ B.ht := disp(B_1.ht, B_2.ht) \}$

$B \rightarrow$ **text** $\{ B.ht :=$ **text**.$h \times B.ps \}$

Fig. 5.23. Translation scheme constructed from Fig. 5.22.

following the three requirements given above. For readability, each grammar symbol in a production is written on a separate line and actions are shown to the right. Thus,

$$S \rightarrow \{ B.ps := 10 \}\ B\ \{ S.ht := B.ht \}$$

is written as

$S \rightarrow$ $\{ B.ps := 10 \}$
$\quad B$ $\{ S.ht := B.ht \}$

Note that actions setting inherited attributes $B_1.ps$ and $B_2.ps$ appear just before B_1 and B_2 on the right sides of productions. □

5.5 TOP-DOWN TRANSLATION

In this section, L-attributed definitions will be implemented during predictive parsing. We work with translation schemes rather than syntax-directed definitions so we can be explicit about the order in which actions and attribute evaluations take place. We also extend the algorithm for left-recursion elimination to translation schemes with synthesized attributes.

Eliminating Left Recursion from a Translation Scheme

Since most arithmetic operators associate to the left, it is natural to use left-recursive grammars for expressions. We now extend the algorithm for eliminating left recursion in Sections 2.4 and 4.3 to allow for attributes when the underlying grammar of a translation scheme is transformed. The transformation applies to translation schemes with synthesized attributes. It allows many of the syntax-directed definitions of Sections 5.1 and 5.2 to be implemented using predictive parsing. The next example motivates the transformation.

Example 5.14. The translation scheme in Fig. 5.24 is transformed below into the translation scheme in Fig. 5.25. The new scheme produces the annotated parse tree of Fig. 5.26 for the expression 9-5+2. The arrows in the figure suggest a way of determining the value of the expression.

$$
\begin{array}{ll}
E \rightarrow E_1 + T & \{\ E.val := E_1.val + T.val\ \} \\
E \rightarrow E_1 - T & \{\ E.val := E_1.val - T.val\ \} \\
E \rightarrow T & \{\ E.val := T.val\ \} \\
T \rightarrow (\ E\ ) & \{\ T.val := E.val\ \} \\
T \rightarrow \mathbf{num} & \{\ T.val := \mathbf{num}.val\ \}
\end{array}
$$

Fig. 5.24. Translation scheme with left-recursive grammar.

In Fig. 5.26, the individual numbers are generated by T, and $T.val$ takes its value from the lexical value of the number, given by attribute **num**.*val*. The 9 in the subexpression 9-5 is generated by the leftmost T, but the minus operator and 5 are generated by the R at the right child of the root. The inherited attribute $R.i$ obtains the value 9 from $T.val$. The subtraction 9−5 and the passing of the result 4 down to the middle node for R are done by embedding the following action between T and R_1 in $R \rightarrow -TR_1$:

$$\{\ R_1.i := R.i - T.val\ \}$$

A similar action adds 2 to the value of 9-5, yielding the result $R.i = 6$ at the bottom node for R. The result is needed at the root as the value of $E.val$; the synthesized attribute s for R, not shown in Fig. 5.26, is used to copy the result up to the root. □

For top-down parsing, we can assume that an action is executed at the time that a symbol in the same position would be expanded. Thus, in the second

$$
\begin{array}{lll}
E \rightarrow T & \{ & R.i := T.val \} \\
\quad R & \{ & E.val := R.s \} \\
R \rightarrow + & & \\
\quad T & \{ & R_1.i := R.i + T.val \} \\
\quad R_1 & \{ & R.s := R_1.s \} \\
R \rightarrow - & & \\
\quad T & \{ & R_1.i := R.i - T.val \} \\
\quad R_1 & \{ & R.s := R_1.s \} \\
R \rightarrow \epsilon & \{ & R.s := R.i \} \\
T \rightarrow (& & \\
\quad E & & \\
\quad) & \{ & T.val := E.val \} \\
T \rightarrow \textbf{num} & \{ & T.val := \textbf{num}.val \}
\end{array}
$$

Fig. 5.25. Transformed translation scheme with right-recursive grammar.

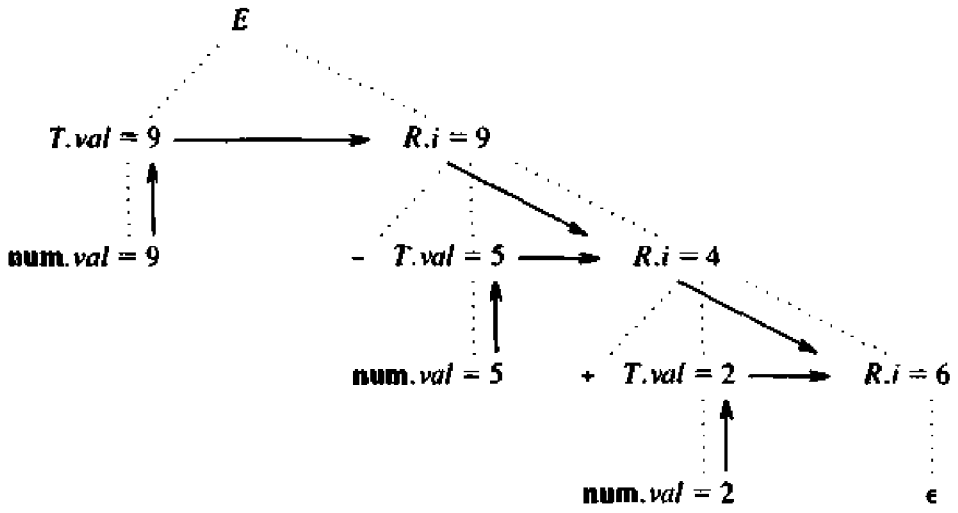


Fig. 5.26. Evaluation of the expression 9-5+2.

production in Fig. 5.25, the first action (assignment to $R_1.i$) is done after T has been fully expanded to terminals, and the second action is done after R_1 has been fully expanded. As noted in the discussion of L-attributed definitions in Section 5.4, an inherited attribute of a symbol must be computed by an action appearing before the symbol and a synthesized attribute of the nonterminal on the left must be computed after all the attributes it depends on have been computed.

In order to adapt other left-recursive translation schemes for predictive parsing, we shall express the use of attributes $R.i$ and $R.s$ in Fig. 5.25 more abstractly. Suppose we have the following translation scheme

$$\begin{array}{ll} A \rightarrow A_1\, Y & \{\ A.a := g(A_1.a,\ Y.y)\ \} \\ A \rightarrow X & \{\ A.a := f(X.x)\ \} \end{array} \tag{5.2}$$

Each grammar symbol has a synthesized attribute written using the corresponding lower case letter, and f and g are arbitrary functions. The generalization to additional A-productions and to productions with strings in place of symbols X and Y can be done as in Example 5.15, below.

The algorithm for eliminating left recursion in Section 2.4 constructs the following grammar from (5.2):

$$\begin{array}{l} A \rightarrow X\, R \\ R \rightarrow Y\, R \mid \epsilon \end{array} \tag{5.3}$$

Taking the semantic actions into account, the transformed scheme becomes

$$\begin{array}{lll} A \rightarrow X & \{\ R.i := f(X.x)\ \} & \\ \quad R & \{\ A.a := R.s\ \} & \\ R \rightarrow Y & \{\ R_1.i := g(R.i,\ Y.y)\ \} & \\ \quad R_1 & \{\ R.s := R_1.s\ \} & \\ R \rightarrow \epsilon & \{\ R.s := R.i\ \} & \end{array} \tag{5.4}$$

The transformed scheme uses attributes i and s for R, as in Fig. 5.25. To see why the results of (5.2) and (5.4) are the same, consider the two annotated parse trees in Fig. 5.27. The value of $A.a$ is computed according to (5.2) in Fig. 5.27(a). Figure 5.27(b) contains the computation of $R.i$ down the tree according to (5.4). The value of $R.i$ at the bottom is passed up unchanged as $R.s$, and it becomes the correct value of $A.a$ at the root ($R.s$ is not shown in Fig. 5.27(b)).

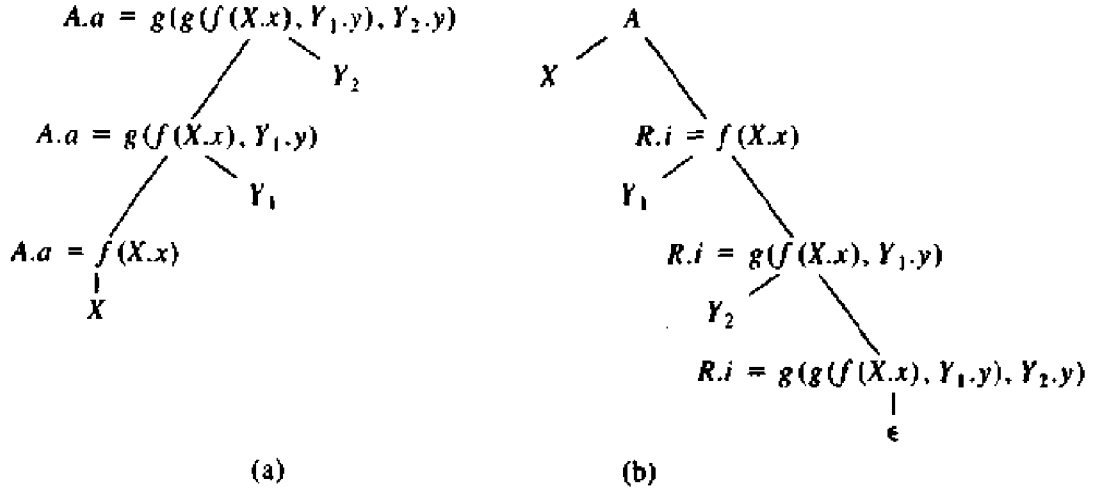


Fig. 5.27. Two ways of computing an attribute value.

Example 5.15. If the syntax-directed definition in Fig. 5.9 for constructing syntax trees is converted into a translation scheme, then the productions and

semantic actions for E become:

$$
\begin{array}{lll}
E \rightarrow E_1 + T & \{ & E.nptr := mknode('+', E_1.nptr, T.nptr) \} \\
E \rightarrow E_1 - T & \{ & E.nptr := mknode('-', E_1.nptr, T.nptr) \} \\
E \rightarrow T & \{ & E.nptr := T.nptr \}
\end{array}
$$

When left recursion is eliminated from this translation scheme, nonterminal E corresponds to A in (5.2) and the strings $+T$ and $-T$ in the first two productions correspond to Y; nonterminal T in the third production corresponds to X. The transformed translation scheme is shown in Fig. 5.28. The productions and semantic actions for T are similar to those in the original definition in Fig. 5.9.

$$
\begin{array}{lll}
E \rightarrow T & \{ & R.i := T.nptr \} \\
\quad R & \{ & E.nptr := R.s \} \\
R \rightarrow + & & \\
\quad T & \{ & R_1.i := mknode('+', R.i, T.nptr) \} \\
\quad R_1 & \{ & R.s := R_1.s \} \\
R \rightarrow - & & \\
\quad T & \{ & R_1.i := mknode('-', R.i, T.nptr) \} \\
\quad R_1 & \{ & R.s := R_1.s \} \\
R \rightarrow \epsilon & \{ & R.s := R.i \} \\
T \rightarrow (& & \\
\quad E & & \\
\quad) & \{ & T.nptr := E.nptr \} \\
T \rightarrow \textbf{id} & \{ & T.nptr := mkleaf(\textbf{id}, \textbf{id}.entry) \} \\
T \rightarrow \textbf{num} & \{ & T.nptr := mkleaf(\textbf{num}, \textbf{num}.val) \}
\end{array}
$$

Fig. 5.28. Transformed translation scheme for constructing syntax trees.

Figure 5.29 shows how the actions in Fig. 5.28 construct a syntax tree for a-4+c. Synthesized attributes are shown to the right of the node for a grammar symbol, and inherited attributes are shown to the left. A leaf in the syntax tree is constructed by actions associated with the productions $T \rightarrow$ **id** and $T \rightarrow$ **num**, as in Example 5.8. At the leftmost T, attribute $T.nptr$ points to the leaf for a. A pointer to the node for a is inherited as attribute $R.i$ on the right side of $E \rightarrow TR$.

When the production $R \rightarrow -TR_1$ is applied at the right child of the root, $R.i$ points to the node for a, and $T.nptr$ to the node for 4. The node for a-4 is constructed by applying *mknode* to the minus operator and these pointers.

Finally, when production $R \rightarrow \epsilon$ is applied, $R.i$ points to the root of the entire syntax tree. The entire tree is returned through the s attributes of the nodes for R (not shown in Fig. 5.29) until it becomes the value of $E.nptr$. □

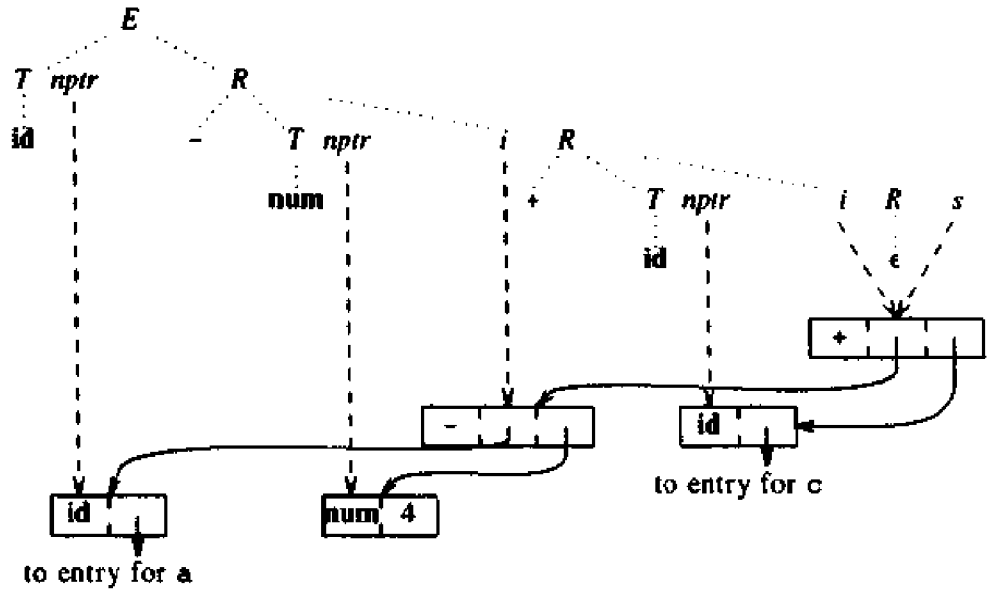


Fig. 5.29. Use of inherited attributes to construct syntax trees.

Design of a Predictive Translator

The next algorithm generalizes the construction of predictive parsers to implement a translation scheme based on a grammar suitable for top-down parsing.

Algorithm 5.2. Construction of a predictive syntax-directed translator.

Input. A syntax-directed translation scheme with an underlying grammar suitable for predictive parsing.

Output. Code for a syntax-directed translator.

Method. The technique is a modification of the predictive-parser construction in Section 2.4.

1. For each nonterminal A, construct a function that has a formal parameter for each inherited attribute of A and that returns the values of the synthesized attributes of A (possibly as a record, as a pointer to a record with a field for each attribute, or using the call-by-reference mechanism for passing parameters, discussed in Section 7.5). For simplicity, we assume that each nonterminal has just one synthesized attribute. The function for A has a local variable for each attribute of each grammar symbol that appears in a production for A.

2. As in Section 2.4, the code for nonterminal A decides what production to use based on the current input symbol.

3. The code associated with each production does the following. We consider the tokens, nonterminals, and actions on the right side of the production from left to right.

i) For token X with synthesized attribute x, save the value of x in the variable declared for $X.x$. Then generate a call to match token X and advance the input.

ii) For nonterminal B, generate an assignment $c := B(b_1, b_2, \ldots, b_k)$ with a function call on the right side, where $b_1, b_2, \ldots, b_k$ are the variables for the inherited attributes of B and c is the variable for the synthesized attribute of B.

iii) For an action, copy the code into the parser, replacing each reference to an attribute by the variable for that attribute. □

Algorithm 5.2 is extended in Section 5.7 to implement any L-attributed definition, provided a parse tree has already been constructed. In Section 5.8, we consider ways of improving the translators constructed by Algorithm 5.2. For example, it may be possible to eliminate copy statements of the form $x := y$ or to use a single variable to hold the values of several attributes. Some such improvements can also be done automatically using the methods of Chapter 10.

Example 5.16. The grammar in Fig. 5.28 is LL(1), and hence suitable for top-down parsing. From the attributes of the nonterminals in the grammar, we obtain the following types for the arguments and results of the functions for E, R, and T. Since E and T do not have inherited attributes, they have no arguments.

```
function E : ↑ syntax_tree_node;
function R(i: ↑ syntax_tree_node): ↑ syntax_tree_node;
function T : ↑ syntax_tree_node;
```

We combine two of the R-productions in Fig. 5.28 to make the translator smaller. The new productions use token **addop** to represent + and −:

$$
\begin{array}{lll}
R \rightarrow \textbf{addop} & & \\
\quad T & \{\ R_1.i := \mathit{mknode}(\textbf{addop}.\mathit{lexeme}, R.i, T.\mathit{nptr})\ \} & \\
\quad R_1 & \{\ R.s := R_1.s\ \} & (5.5) \\
R \rightarrow \epsilon & \{\ R.s := R.i\ \} &
\end{array}
$$

The code for R is based on the parsing procedure in Fig. 5.30. If the lookahead symbol is **addop**, then the production $R \rightarrow \textbf{addop}\, T\, R$ is applied by using the procedure *match* to read the next input token after **addop**, and then calling the procedures for T and R. Otherwise, the procedure does nothing, to mimic the production $R \rightarrow \epsilon$.

The procedure for R in Fig. 5.31 contains code for evaluating attributes. The lexical value *lexval* of the token **addop** is saved in *addoplexeme*, **addop** is matched, T is called, and its result is saved using *nptr*. Variable *i1* corresponds to the inherited attribute $R_1.i$, and *s1* to the synthesized attribute $R_1.s$. The **return** statement returns the value of s just before control leaves the function. The functions for E and T are constructed similarly. □

```
procedure R;
begin
    if lookahead = addop then begin
        match(addop); T; R
    end
    else begin /* do nothing */
    end
end;
```

Fig. 5.30. Parsing procedure for the productions $R \rightarrow \textbf{addop}\ T\ R \mid \epsilon$.

```
function R(i: ↑ syntax_tree_node): ↑ syntax_tree_node;
    var nptr, il, sl, s: ↑ syntax_tree_node;
        addoplexeme : char;
begin
    if lookahead = addop then begin
        /* production R → addop T R */
        addoplexeme := lexval;
        match(addop);
        nptr := T;
        il := mknode(addoplexeme, i, nptr);
        sl := R(il);
        s := sl
    end
    else s := i;   /* production R → ε */
    return s
end;
```

Fig. 5.31. Recursive-descent construction of syntax trees.

5.6 BOTTOM-UP EVALUATION OF INHERITED ATTRIBUTES

In this section, we present a method to implement L-attributed definitions in the framework of bottom-up parsing. The method is capable of handling all L-attributed definitions considered in the previous section in that it can implement any L-attributed definition based on an LL(1) grammar. It can also implement many (but not all) L-attributed definitions based on LR(1) grammars. The method is a generalization of the bottom-up translation technique introduced in Section 5.3.

Removing Embedding Actions from Translation Schemes

In the bottom-up translation method of Section 5.3, we relied upon all translation actions being at the right end of the production, while in the predictive-

parsing method of Section 5.5 we needed to embed actions at various places within the right side. To begin our discussion of how inherited attributes can be handled bottom up, we introduce a transformation that makes all embedded actions in a translation scheme occur at the right ends of their productions.

The transformation inserts new *marker* nonterminals generating ϵ into the base grammar. We replace each embedded action by a distinct marker nonterminal M and attach the action to the end of the production $M \rightarrow \epsilon$. For example, the translation scheme

$$
\begin{array}{lcl}
E & \rightarrow & T\ R \\
R & \rightarrow & +\ T\ \{print('+')\}\ R \mid -\ T\ \{print('-')\}\ R \mid \epsilon \\
T & \rightarrow & \textbf{num}\ \{print(\textbf{num}.val)\}
\end{array}
$$

is transformed using marker nonterminals M and N into

$$
\begin{array}{lcl}
E & \rightarrow & T\ R \\
R & \rightarrow & +\ T\ M\ R \mid -\ T\ N\ R \mid \epsilon \\
T & \rightarrow & \textbf{num}\ \{print(\textbf{num}.val)\} \\
M & \rightarrow & \epsilon\ \{print('+')\} \\
N & \rightarrow & \epsilon\ \{print('-')\}
\end{array}
$$

The grammars in the two translation schemes accept exactly the same language and, by drawing a parse tree with extra nodes for the actions, we can show that the actions are performed in the same order. Actions in the transformed translation scheme terminate productions, so they can be performed just before the right side is reduced during bottom-up parsing.

Inheriting Attributes on the Parser Stack

A bottom-up parser reduces the right side of production $A \rightarrow XY$ by removing X and Y from the top of the parser stack and replacing them by A. Suppose X has a synthesized attribute $X.s$, which the implementation of Section 5.3 kept along with X on the parser stack.

Since the value of $X.s$ is already on the parser stack before any reductions take place in the subtree below Y, this value can be inherited by Y. That is, if inherited attribute $Y.i$ is defined by the copy rule $Y.i := X.s$, then the value $X.s$ can be used where $Y.i$ is called for. As we shall see, copy rules play an important role in the evaluation of inherited attributes during bottom-up parsing.

Example 5.17. The type of an identifier can be passed by copy rules using inherited attributes as shown in Fig. 5.32 (adapted from Fig. 5.7). We shall first examine the moves made by a bottom-up parser on the input

```
real p, q, r
```

Then we show how the value of attribute $T.type$ can be accessed when the productions for L are applied. The translation scheme we wish to implement is

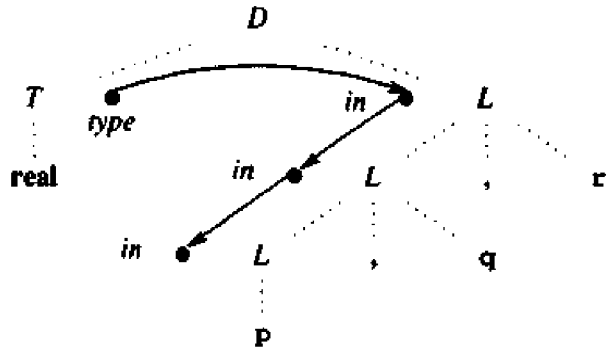


Fig. 5.32. At each node for L, $L.in = T.type$.

$D \rightarrow T$	{ $L.in := T.type$ }
$\quad L$	
$T \rightarrow$ **int**	{ $T.type := integer$ }
$T \rightarrow$ **real**	{ $T.type := real$ }
$L \rightarrow$	{ $L_1.in := L.in$ }
$\quad L_1$, **id**	{ $addtype$(**id**.$entry$, $L.in$) }
$L \rightarrow$ **id**	{ $addtype$(**id**.$entry$, $L.in$) }

If we ignore the actions in the above translation scheme, the sequence of moves made by the parser on the input of Fig. 5.32 is as in Fig. 5.33. For clarity, we show the corresponding grammar symbol instead of a stack state and the actual identifier instead of the token **id**.

INPUT	*state*	PRODUCTION USED
real p,q,r	–	
p,q,r	**real**	
p,q,r	T	$T \rightarrow$ **real**
,q,r	T p	
,q,r	$T\ L$	$L \rightarrow$ **id**
q,r	$T\ L$,	
,r	$T\ L$, q	
,r	$T\ L$	$L \rightarrow L$, **id**
r	$T\ L$,	
	$T\ L$, r	
	$T\ L$	$L \rightarrow L$, **id**
	D	$D \rightarrow T\ L$

Fig. 5.33. Whenever a right side for L is reduced, T is just below the right side.

Suppose as in Section 5.3 that the parser stack is implemented as a pair of

arrays *state* and *val*. If *state*[*i*] is for grammar symbol X, then *val*[*i*] holds a synthesized attribute $X.s$. The contents of the *state* array are shown in Fig. 5.33. Note that every time the right side of a production for L is reduced in Fig. 5.33, T is in the stack just below the right side. We can use this fact to access the attribute value $T.type$.

The implementation in Fig. 5.34 uses the fact that attribute $T.type$ is at a known place in the *val* stack, relative to the top. Let *top* and *ntop* be the indices of the top entry in the stack just before and after a reduction takes place, respectively. From the copy rules defining $L.in$, we know that $T.type$ can be used in place of $L.in$.

When the production $L \rightarrow$ **id** is applied, **id**.*entry* is on top of the *val* stack and $T.type$ is just below it. Hence, *addtype*(*val*[*top*], *val*[*top* − 1]) is equivalent to *addtype*(**id**.*entry*, $T.type$). Similarly, since the right side of the production $L \rightarrow L$, **id** has three symbols, $T.type$ appears in *val*[*top* − 3] when this reduction takes place. The copy rules involving $L.in$ are eliminated because the value of $T.type$ in the stack is used instead. □

PRODUCTION	CODE FRAGMENT
$D \rightarrow T\ L$;	
$T \rightarrow$ **int**	*val*[*ntop*] := *integer*
$T \rightarrow$ **real**	*val*[*ntop*] := *real*
$L \rightarrow L$, **id**	*addtype*(*val*[*top*], *val*[*top* − 3])
$L \rightarrow$ **id**	*addtype*(*val*[*top*], *val*[*top* − 1])

Fig. 5.34. The value of $T.type$ is used in place of $L.in$.

Simulating the Evaluation of Inherited Attributes

Reaching into the parser stack for an attribute value works only if the grammar allows the position of the attribute value to be predicted.

Example 5.18. As an instance where we cannot predict the position, consider the following translation scheme:

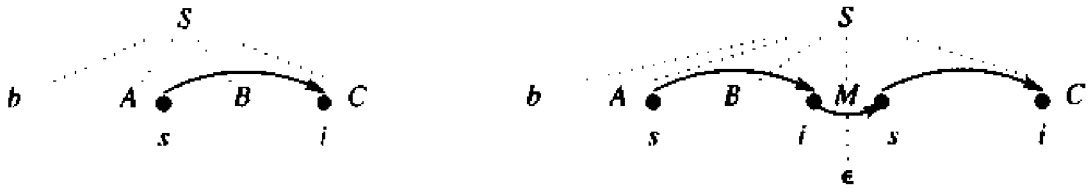
PRODUCTION	SEMANTIC RULES	
$S \rightarrow aAC$	$C.i := A.s$	
$S \rightarrow bABC$	$C.i := A.s$	(5.6)
$C \rightarrow c$	$C.s := g(C.i)$	

C inherits the synthesized attribute $A.s$ by a copy rule. Note that there may or may not be a B between A and C in the stack. When reduction by $C \rightarrow c$ is performed, the value of $C.i$ is either in *val*[*top* − 1] or in *val*[*top* − 2], but it is not clear which case applies.

In Fig. 5.35, a fresh marker nonterminal M is inserted just before C on the

right side of the second production in (5.6). If we are parsing according to production $S \rightarrow bABMC$, then $C.i$ inherits the value of $A.s$ indirectly through $M.i$ and $M.s$. When the production $M \rightarrow \epsilon$ is applied, a copy rule $M.s := M.i$ ensures that the value $M.s = M.i = A.s$ appears just before the part of the stack used for parsing the subtree for C. Thus, the value of $C.i$ can be found in $val[top - 1]$ when $C \rightarrow c$ is applied, independent of whether the first or second productions in the following modification of (5.6) are used.

PRODUCTION	SEMANTIC RULES
$S \rightarrow aAC$	$C.i := A.s$
$S \rightarrow bABMC$	$M.i := A.s; C.i := M.s$
$C \rightarrow c$	$C.s := g(C.i)$
$M \rightarrow \epsilon$	$M.s := M.i$



(a) original production (b) modified dependencies

Fig. 5.35. Copying an attribute value through a marker M.

Marker nonterminals can also be used to simulate semantic rules that are not copy rules. For example, consider

PRODUCTION	SEMANTIC RULES	
$S \rightarrow aAC$	$C.i := f(A.s)$	(5.7)

This time the rule defining $C.i$ is not a copy rule, so the value of $C.i$ is not already in the val stack. This problem can also be solved using a marker.

PRODUCTION	SEMANTIC RULES	
$S \rightarrow aANC$	$N.i := A.s; C.i := N.s$	(5.8)
$N \rightarrow \epsilon$	$N.s := f(N.i)$	

The distinct nonterminal N inherits $A.s$ by a copy rule. Its synthesized attribute $N.s$ is set to $f(A.s)$; then $C.i$ inherits this value using a copy rule. When we reduce by $N \rightarrow \epsilon$, we find the value of $N.i$ in the place for $A.s$, that is, in $val[top - 1]$. When we reduce by $S \rightarrow aANC$, the value of $C.i$ is also found in $val[top - 1]$, because it is $N.s$. Actually, we do not need $C.i$ at this time; we needed it during the reduction of a terminal string to C, when its value was safely stored on the stack with N.

Example 5.19. Three marker nonterminals L, M, and N are used in Fig. 5.36 to ensure that the value of inherited attribute $B.ps$ appears at a known position in the parser stack while the subtree for B is being reduced. The original

attribute grammar appears in Fig. 5.22 and its relevance to text formatting is explained in Example 5.13.

PRODUCTION	SEMANTIC RULES
$S \rightarrow L\ B$	$B.ps := L.s$
	$S.ht := B.ht$
$L \rightarrow \epsilon$	$L.s := 10$
$B \rightarrow B_1\ M\ B_2$	$B_1.ps := B.ps$
	$M.i := B.ps$
	$B_2.ps := M.s$
	$B.ht := max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ **sub** $N\ B_2$	$B_1.ps := B.ps$
	$N.i := B.ps$
	$B_2.ps := N.s$
	$B.ht := disp(B_1.ht, B_2.ht)$
$B \rightarrow$ **text**	$B.ht := \mathbf{text}.h \times B.ps$
$M \rightarrow \epsilon$	$M.s := M.i$
$N \rightarrow \epsilon$	$N.s := shrink(N.i)$

Fig. 5.36. All inherited attributes are set by copy rules.

Initialization is done using L. The production for S is $S \rightarrow L\ B$ in Fig. 5.36, so L will remain on the stack while the subtree below B is reduced. The value 10 of the inherited attribute $B.ps = L.s$ is entered into the parser stack by the rule $L.s := 10$ associated with $L \rightarrow \epsilon$.

Marker M in $B \rightarrow B_1\ M\ B_2$ plays a role similar to that of M in Fig. 5.35; it ensures that the value of $B.ps$ appears just below B_2 in the parser stack. In production $B \rightarrow B_1$ **sub** $N\ B_2$, the nonterminal N is used as it is in (5.8). N inherits, via the copy rule $N.i := B.ps$, the attribute value that $B_2.ps$ depends on, and synthesizes the value of $B_2.ps$ by the rule $N.s := shrink(N.i)$. The consequence, which we leave as an exercise, is that the value of $B.ps$ is always immediately below the right side when we reduce to B.

Code fragments implementing the syntax-directed definition of Fig. 5.36 are shown in Fig. 5.37. All inherited attributes are set by copy rules in Fig. 5.36, so the implementation obtains their values by keeping track of their position in the *val* stack. As in previous examples, *top* and *ntop* give the indices of the top of the stack before and after a reduction, respectively. □

Systematic introduction of markers, as in the modification of (5.6) and (5.7), can make it possible to evaluate L-attributed definitions during LR parsing. Since there is only one production for each marker, a grammar

PRODUCTION	CODE FRAGMENT
$S \rightarrow L\ B$	$val[ntop] := val[top]$
$L \rightarrow \epsilon$	$val[ntop] := 0$
$B \rightarrow B_1\ M\ B_2$	$val[ntop] := max(val[top-2], val[top])$
$B \rightarrow B_1$ **sub** $N\ B_2$	$val[ntop] := disp(val[top-3], val[top])$
$B \rightarrow$ **text**	$val[ntop] := val[top] \times val[top-1]$
$M \rightarrow \epsilon$	$val[ntop] := val[top-1]$
$N \rightarrow \epsilon$	$val[ntop] := shrink(val[top-2])$

Fig. 5.37. Implementation of the syntax-directed definition in Fig. 5.36.

remains LL(1) when markers are added. Any LL(1) grammar is also an LR(1) grammar so no parsing conflicts arise when markers are added to an LL(1) grammar. Unfortunately, the same cannot be said of all LR(1) grammars; that is, parsing conflicts can arise if markers are introduced in certain LR(1) grammars.

The ideas of the previous examples can be formalized in the following algorithm.

Algorithm 5.3. Bottom-up parsing and translation with inherited attributes.

Input. An L-attributed definition with an underlying LL(1) grammar.

Output. A parser that computes the values of all attributes on its parsing stack.

Method. Let us assume for simplicity that every nonterminal A has one inherited attribute $A.i$, and every grammar symbol X has a synthesized attribute $X.s$. If X is a terminal, then its synthesized attribute is really the lexical value returned with X by the lexical analyzer; that lexical value appears on the stack, in an array *val*, as in the previous examples.

For every production $A \rightarrow X_1 \cdots X_n$, introduce n new marker nonterminals, $M_1, \ldots, M_n$, and replace the production by $A \rightarrow M_1X_1 \cdots M_nX_n$.[2] The synthesized attribute $X_j.s$ will go on the parser stack in the *val* array entry associated with X_j. The inherited attribute $X_j.i$, if there is one, appears in the same array, but associated with M_j.

An important invariant is that as we parse, the inherited attribute $A.i$, if it exists, is found in the position of the *val* array immediately below the position for M_1. As we assume the start symbol has no inherited attribute, there is no problem with the case when the start symbol is A, but even if there were such

[2] Although inserting M_1 before X_1 simplifies the discussion of marker nonterminals, it has the unfortunate side effect of introducing parsing conflicts into a left-recursive grammar. See Exercise 5.21. As noted below, M_1 can be eliminated.

an inherited attribute, it could be placed below the bottom of the stack. We may prove the invariant by an easy induction on the number of steps in the bottom-up parse, noting the fact that inherited attributes are associated with the marker nonterminals M_j, and that the attribute $X_j.i$ is computed at M_j before we begin the reduction to X_j.

To see that the attributes can be computed as intended during a bottom-up parse, consider two cases. First, if we reduce to a marker nonterminal M_j, we know which production $A \to M_1 X_1 \cdots M_n X_n$ that marker belongs to. We therefore know the positions of any attributes that the inherited attribute $X_j.i$ needs for its computation. $A.i$ is in $val[top-2j+2]$, $X_1.i$ is in $val[top-2j+3]$, $X_1.s$ is in $val[top-2j+4]$, $X_2.i$ is in $val[top-2j+5]$, and so on. Therefore, we may compute $X_j.i$ and store it in $val[top+1]$, which becomes the new top of stack after the reduction. Notice how the fact that the grammar is LL(1) is important, or else we might not be sure that we were reducing ϵ to one particular marker nonterminal, and thus could not locate the proper attributes, or even know what formula to apply in general. We ask the reader to take on faith, or derive a proof of the fact that every LL(1) grammar with markers is still LR(1).

The second case occurs when we reduce to a nonmarker symbol, say by production $A \to M_1 X_1 \cdots M_n X_n$. Then, we have only to compute the synthesized attribute $A.s$; note that $A.i$ was already computed, and lives at the position on the stack just below the position into which we insert A itself. The attributes needed to compute $A.s$ are clearly available at known positions on the stack, the positions of the X_j's, during the reduction.

The following simplifications reduce the number of markers; the second avoids parsing conflicts in left-recursive grammars.

1. If X_j has no inherited attribute, we need not use marker M_j. Of course, the expected positions for attributes on the stack will change if M_j is omitted, but this change can be incorporated easily into the parser.

2. If $X_1.i$ exists, but is computed by a copy rule $X_1.i = A.i$, then we can omit M_1, since we know by our invariant that $A.i$ will already be located where we want it, just below X_1 on the stack, and this value can therefore serve for $X_1.i$ as well. □

Replacing Inherited by Synthesized Attributes

It is sometimes possible to avoid the use of inherited attributes by changing the underlying grammar. For example, a declaration in Pascal can consist of a list of identifiers followed by a type, e.g., `m, n : integer`. A grammar for such declarations may include productions of the form

$D \to L$: T
$T \to$ **integer** | **char**
$L \to L$, **id** | **id**

Since identifiers are generated by L but the type is not in the subtree for L, we cannot associate the type with an identifier using synthesized attributes alone. In fact, if nonterminal L inherits a type from T to its right in the first production, we get a syntax-directed definition that is not L-attributed, so translations based on it cannot be done during parsing.

A solution to this problem is to restructure the grammar to include the type as the last element of the list of identifiers:

$D \rightarrow \textbf{id}\ L$
$L \rightarrow \textbf{,}\ \textbf{id}\ L \mid \textbf{:}\ T$
$T \rightarrow \textbf{integer} \mid \textbf{char}$

Now, the type can be carried along as a synthesized attribute *L.type*. As each identifier is generated by L, its type can be entered into the symbol table.

A Difficult Syntax-Directed Definition

Algorithm 5.3, for implementing inherited attributes during bottom-up parsing, extends to some, but not all, LR grammars. The L-attributed definition in Fig. 5.38 is based on a simple LR(1) grammar, but it cannot be implemented, as is, during LR parsing. Nonterminal L in $L \rightarrow \epsilon$ inherits the count of the number of 1's generated by S. Since the production $L \rightarrow \epsilon$ is the first that a bottom-up parser would reduce by, the translator at that time cannot know the number of 1's in the input.

PRODUCTION	SEMANTIC RULES
$S \rightarrow L$	$L.count := 0$
$L \rightarrow L_1\ 1$	$L_1.count := L.count + 1$
$L \rightarrow \epsilon$	$print(L.count)$

Fig. 5.38. Difficult syntax-directed definition.

5.7 RECURSIVE EVALUATORS

Recursive functions that evaluate attributes as they traverse a parse tree can be constructed from a syntax-directed definition using a generalization of the techniques for predictive translation in Section 5.5. Such functions allow us to implement syntax-directed definitions that cannot be implemented simultaneously with parsing. In this section, we associate a single translation function with each nonterminal. The function visits the children of a node for the nonterminal in some order determined by the production at the node; it is not necessary that the children be visited in a left-to-right order. In Section 5.10, we shall see how the effect of translation during more than one pass can be achieved by associating multiple procedures with nonterminals.

Left-to Right Traversals

In Algorithm 5.2, we showed how an L-attributed definition based on an LL(1) grammar can be implemented by constructing a recursive function that parses and translates each nonterminal. All L-attributed syntax-directed definitions can be implemented if a similar recursive function is invoked on a node for that nonterminal in a previously constructed parse tree. By looking at the production at the node, the function can determine what its children are. The function for a nonterminal A takes a node and the values of the inherited attributes for A as arguments, and returns the values of the synthesized attributes for A as results.

The details of the construction are exactly as in Algorithm 5.2, except for step 2 where the function for a nonterminal decides what production to use based on the current input symbol. The function here employs a case statement to determine the production used at a node. We give an example to illustrate the method.

Example 5.20. Consider the syntax-directed definition for determining the size and height of formulas in Fig. 5.22. The nonterminal B has an inherited attribute ps and a synthesized attribute ht. Using Algorithm 5.2, modified as mentioned above, we construct the function for B shown in Fig. 5.39.

Function B takes a node n and a value corresponding to $B.ps$ at the node as arguments, and returns a value corresponding to $B.ht$ at node n. The function has a case for each production with B on the left. The code corresponding to each production simulates the semantic rules associated with the production. The order in which the rules are applied must be such that inherited attributes of a nonterminal are computed before the function for the nonterminal is called.

In the code corresponding to the production $B \rightarrow B$ **sub** B, variables ps, $ps1$, and $ps2$ hold the values of the inherited attributes $B.ps$, $B_1.ps$, and $B_2.ps$. Similarly ht, $ht1$, and $ht2$ hold the values of $B.ht$, $B_1.ht$, and $B_2.ht$. We use the function $child(m, i)$ to refer to the ith child of node m. Since B_2 is the label of the third child of node n, the value of $B_2.ht$ is determined by the function call $B(child(n, 3), ps2)$. □

Other Traversals

Once an explicit parse tree is available, we are free to visit the children of a node in any order. Consider the non-L-attributed definition of Example 5.21. In a translation specified by this definition, the children of a node for one production need to be visited from left to right, while the children of a node for the other production need to be visited from right to left.

This abstract example illustrates the power of using mutually recursive functions for evaluating the attributes at the nodes of a parse tree. The functions need not depend on the order in which the parse tree nodes are created. The main consideration for evaluation during a traversal is that the inherited attributes at a node be computed before the node is first visited and that the

```
function B(n, ps);
    var ps1, ps2, ht1, ht2;
begin
    case production at node n of
    'B → B1 B2':
        ps1 := ps;
        ht1 := B(child(n, 1), ps1);
        ps2 := ps;
        ht2 := B(child(n, 2), ps2);
        return max(ht1, ht2);
    'B → B1 sub B2':
        ps1 := ps;
        ht1 := B(child(n, 1), ps1);
        ps2 := shrink(ps);
        ht2 := B(child(n, 3), ps2);
        return disp(ht1, ht2);
    'B → text':
        return ps × text.h;
    default:
        error
    end
end;
```

Fig. 5.39. Function for nonterminal B in Fig. 5.22.

synthesized attributes be computed before we leave the node for the last time.

Example 5.21. Each of the nonterminals in Fig. 5.40 has an inherited attribute i and a synthesized attribute s. The dependency graphs for the two productions are also shown. The rules associated with $A \rightarrow L\,M$ set up left-to-right dependencies and the rules associated with $A \rightarrow Q\,R$ set up right-to-left dependencies.

The function for nonterminal A is shown in Fig. 5.41; we assume that functions for L, M, Q, and R can be constructed. Variables in Fig. 5.41 are named after the nonterminal and its attribute; e.g., *li* and *ls* are the variables corresponding to $L.i$ and $L.s$.

The code corresponding to the production $A \rightarrow L\,M$ is constructed as in Example 5.20. That is, we determine the inherited attribute of L, call the function for L to determine the synthesized attribute of L, and repeat the process for M. The code corresponding to $A \rightarrow Q\,R$ visits the subtree for R before it visits the subtree for Q. Otherwise, the code for the two productions is very similar. □

PRODUCTION	SEMANTIC RULES
$A \rightarrow L\ M$	$L.i := l(A.i)$
	$M.i := m(L.s)$
	$A.s := f(M.s)$
$A \rightarrow Q\ R$	$R.i := r(A.i)$
	$Q.i := q(R.s)$
	$A.s := f(Q.s)$

Fig. 5.40. Productions and semantic rules for nonterminal A.

```
function A(n, ai);
begin
    case production at node n of
    'A → L M':          /* left-to-right order */
        li := l(ai);
        ls := L(child(n, 1), li);
        mi := m(ls);
        ms := M(child(n, 2), mi)
        return f(ms);
    'A → Q R':          /* right-to-left order */
        ri := r(ai);
        rs := R(child(n, 2), ri);
        qi := q(rs);
        qs := Q(child(n, 1), qi);
        return f(qs);
    default:
        error
    end
end;
```

Fig. 5.41. Dependencies in Fig. 5.40 determine the order children are visited in.

5.8 SPACE FOR ATTRIBUTE VALUES AT COMPILE TIME

In this section we consider the compile-time assignment of space for attribute values. We shall use information from the dependency graph for a parse tree, so the approach of this section is suitable for parse-tree methods that determine the evaluation order from the dependency graph. In the next section we consider the case in which the evaluation order can be predicted in advance, so we can decide on the space for attributes once and for all when the compiler is constructed.

Given a not necessarily depth-first order of evaluation for attributes, the *lifetime* of an attribute begins when the attribute is first computed and ends when all attributes that depend on it have been computed. We can conserve space by holding an attribute value only during its lifetime.

In order to emphasize that the techniques in this section apply to any evaluation order, we shall consider the following syntax-directed definition that is not L-attributed, for passing type information to identifiers in a declaration.

Example 5.22. The syntax-directed definition in Fig. 5.42 is an extension of that in Fig. 5.4 to allow declarations of the form

```
real c[12][31];                    (5.9)
int  x[3], y[5];                   (5.10)
```

A parse tree for (5.10) is shown by the dotted lines in Fig. 5.43(a). The numbers at the nodes are discussed in the next example. As in Example 5.3, the type obtained from T is inherited by L and passed down towards the identifiers in the declaration. An edge from $T.type$ to $L.in$ shows that $L.in$ depends on $T.type$. The syntax-directed definition in Fig. 5.42 is not L-attributed because $I_1.in$ depends on **num**.*val* and **num** is to the right of I_1 in $I \rightarrow I_1$ [**num**]. □

Assigning Space for Attributes at Compile Time

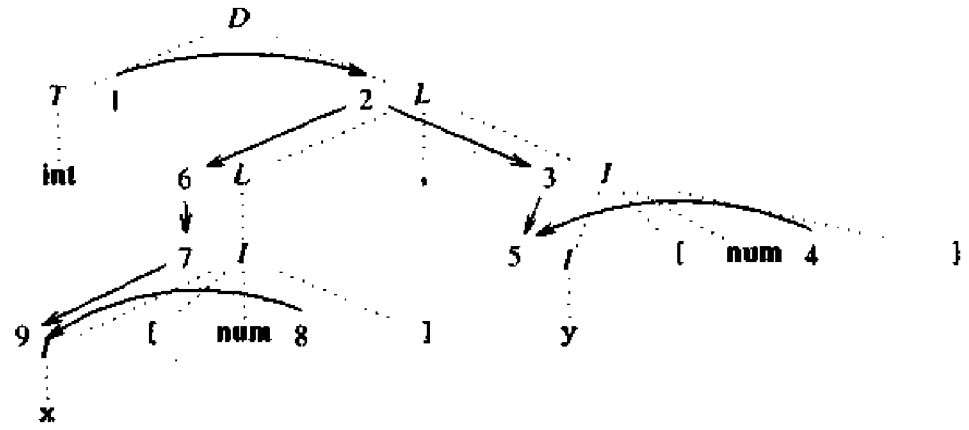
Suppose we are given a sequence of registers to hold attribute values. For convenience, we assume that each register can hold any attribute value. If attributes represent different types, then we can form groups of attributes that take the same amount of storage and consider each group separately. We rely on information about the lifetimes of attributes to determine the registers into which they are evaluated.

Example 5.23. Suppose that attributes are evaluated in the order given by the node numbers in the dependency graph of Fig. 5.43,[3] constructed in the last example. The lifetime of each node begins when its attribute is evaluated and

[3] The dependency graph in Fig. 5.43 does not show nodes corresponding to the semantic rule *addtype*(**id**.*entry*, *I.in*) because no space is allocated for dummy attributes. Note, however, that this semantic rule must not be evaluated until after the value of *I.in* is available. An algorithm to determine this fact must work with a dependency graph containing nodes for this semantic rule.

PRODUCTION	RULES
$D \rightarrow T\ L$	$L.in := T.type$
$T \rightarrow$ **int**	$T.type := integer$
$T \rightarrow$ **real**	$T.type := real$
$L \rightarrow L_1$, I	$L_1.in := L.in$ $I.in := L.in$
$L \rightarrow I$	$I.in := L.in$
$I \rightarrow I_1$ [**num**]	$I_1.in := array($**num**$.val, I.in)$
$I \rightarrow$ **id**	$addtype($**id**$.entry, I.in)$

Fig. 5.42. Passing the type to identifiers in a declaration.



(a) Dependency graph for a parse tree

(b) Nodes in order of evaluation (a)

Fig. 5.43. Determining lifetimes of attribute values.

ends when its attribute is used for the last time. For example, the lifetime of node 1 ends when 2 is evaluated because 2 is the only node that depends on 1. The lifetime of 2 ends when 6 is evaluated. □

A method for evaluating attributes that uses as few registers as possible is given in Fig. 5.44. We consider the nodes of dependency graph D for a parse

tree in the order they are to be evaluated. Initially, we have a pool of registers $r_1, r_2, \ldots$. If attribute b is defined by the semantic rule $b := f(c_1, c_2, \ldots, c_k)$, then the lifetime of one or more of $c_1, c_2, \ldots, c_k$ might end with the evaluation of b; registers holding such attributes are returned after b is evaluated. Whenever possible, b is evaluated into a register that held one of $c_1, c_2, \ldots, c_k$.

```
for each node m in m_1, m_2, ..., m_N do begin
    for each node n whose lifetime ends with the evaluation of m do
        mark n's register;
    if some register r is marked then begin
        unmark r;
        evaluate m into register r;
        return marked registers to the pool
    end
    else /* no registers were marked */
        evaluate m into a register from the pool;
    /* actions using the value of m can be inserted here */
    if the lifetime of m has ended then
        return m's register to the pool
end
```

Fig. 5.44. Assigning attribute values to registers.

Registers used during an evaluation of the dependency graph of Fig. 5.43 are shown in Fig. 5.45. We start by evaluating node 1 to register r_1. The lifetime of node 1 ends when 2 is evaluated, so 2 is evaluated into r_1. Node 3 gets a fresh register r_2, because node 6 will need the value of 2.

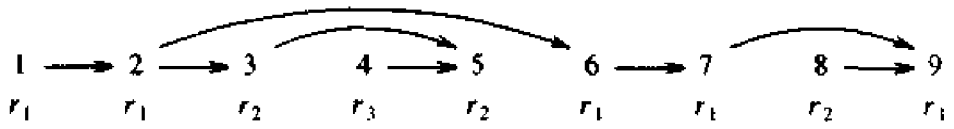


Fig. 5.45. Registers used for attribute values in Fig. 5.43.

Avoiding Copies

We can improve the method of Fig. 5.44 by treating copy rules as a special case. A copy rule has the form $b := c$, so if the value of c is in register r, then the value of b already appears in register r. The number of attributes defined by copy rules can be significant, so we wish to avoid making explicit copies.

A set of nodes having the same value forms an equivalence class. The method of Fig. 5.44 can be modified as follows to hold the value of an equivalence class in a register. When node m is considered, we first check if it is defined by a copy rule. If it is, then its value must already be in a register, and m joins the equivalence class with values in that register. Furthermore, a register is returned to the pool only at the end of the lifetimes of all nodes with values in the register.

Example 5.24. The dependency graph in Fig. 5.43 is redrawn in Fig. 5.46, with an equal sign before each node defined by a copy rule. From the syntax-directed definition in Fig. 5.42, we find that the type determined at node 1 is copied to each element in the list of identifiers, resulting in nodes 2, 3, 6, and 7 of Fig. 5.43 being copies of 1.

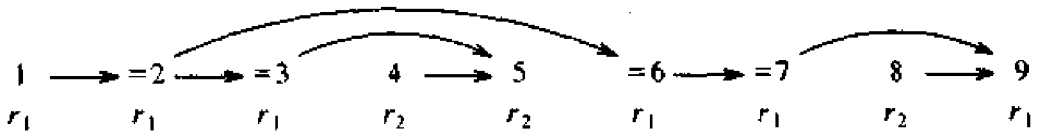


Fig. 5.46. Registers used, taking copy rules into account.

Since 2 and 3 are copies of 1, their values are taken from register r_1 in Fig. 5.46. Note that the lifetime of 3 ends when 5 is evaluated, but register r_1 holding the value of 3 is not returned to the pool because the lifetime of 2 in its equivalence class has not ended.

The following code shows how the declaration (5.10) of Example 5.22 might be processed by a compiler:

```
r1 := integer;          /* evaluates nodes 1, 2, 3, 6, 7 */
r2 := 5;                /* evaluates node 4 */
r2 := array(r2, r1);    /* type of y */
adddtype(y, r2);
r2 := 3;                /* evaluates node 8 */
r2 := array(r2, r1);    /* type of x */
adddtype(x, r2);
```

In the above, x and y point to the symbol-table entries for x and y, and procedure *addtype* must be called at the appropriate times to add the types of x and y to their symbol-table entries. □

5.9 ASSIGNING SPACE AT COMPILER-CONSTRUCTION TIME

Although it is possible to hold all attribute values on a single stack during a traversal, we can sometimes avoid making copies by using multiple stacks. In general, if dependencies between attributes make it inconvenient to place certain attribute values on a stack, we can save them at nodes in an explicitly

constructed syntax tree.

We have already seen the use of a stack to hold attribute values during bottom-up parsing in Sections 5.3 and 5.6. A stack is also used implicitly by a recursive-descent parser to keep track of procedure calls; this issue will be discussed in Chapter 7.

The use of a stack can be combined with other techniques for saving space. The print actions used extensively in the translation schemes in Chapter 2 emit string-valued attributes to an output file whenever possible. While constructing syntax trees in Section 5.2, we passed pointers to nodes instead of complete subtrees. In general, rather than passing large objects, we can save space by passing pointers to them. These techniques will be applied in Examples 5.27 and 5.28.

Predicting Lifetimes from the Grammar

When the evaluation order for attributes is obtained from a particular traversal of the parse tree, we can predict the lifetimes of attributes at compiler-construction time. For example, suppose children are visited from left to right during a depth-first traversal, as in Section 5.4. Starting at a node for production $A \rightarrow B\,C$, the subtree for B is visited, the subtree for C is visited, and then we return to the node for A. The parent of A cannot refer to the attributes of B and C, so their lifetimes must end when we return to A. Note that these observations are based on the production $A \rightarrow B\,C$ and the order in which the nodes for these nonterminals are visited. We do not need to know about the subtrees at B and C.

With any evaluation order, if the lifetime of attribute c is contained in that of b, then the value of c can be held in a stack above the value of b. Here b and c do not have to be attributes of the same nonterminal. For the production $A \rightarrow B\,C$, we can use a stack during a depth-first traversal in the following way.

Start at the node for A with the inherited attributes of A already on the stack. Then evaluate and push the values of the inherited attributes of B. These attributes remain on the stack as we traverse the subtree of B, returning with the synthesized attributes of B above them. This process is repeated with C; that is, we push its inherited attributes, traverse its subtree and return with its synthesized attributes on top. Writing $\mathrm{I}(X)$ and $\mathrm{S}(X)$ for the inherited and synthesized attributes of X, respectively, the stack now contains

$$\mathrm{I}(A),\ \mathrm{I}(B),\ \mathrm{S}(B),\ \mathrm{I}(C),\ \mathrm{S}(C) \tag{5.11}$$

All of the attribute values needed to compute the synthesized attributes of A are now on the stack, so we can return to A with the stack containing

$$\mathrm{I}(A),\ \mathrm{S}(A)$$

Notice that the number (and presumably the size) of inherited and synthesized attributes of a grammar symbol is fixed. Thus, at each step of the above process we know how far down into the stack we have to reach to find

an attribute.

Example 5.25. Suppose that attribute values for the typesetting translation of Fig. 5.22 are held in a stack as discussed above. Starting at a node for production $B \rightarrow B_1 B_2$ with $B.ps$ on top of the stack, the stack contents before and after visiting a node are shown in Fig. 5.47 to the left and right of the node, respectively. As usual stacks grow downwards.

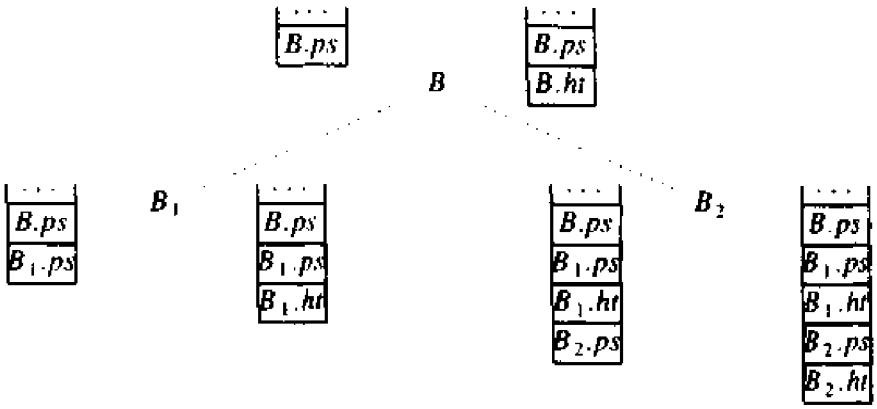


Fig. 5.47. Stack contents before and after visiting a node.

Note that just before a node for nonterminal B is visited for the first time, its ps attribute is on top of the stack. Just after the last visit, i.e., when the traversal moves up from that node, its ht and ps attributes are in the top two positions of the stack. □

When an attribute b is defined by a copy rule $b := c$ and the value of c is on top of the stack of attribute values, it may not be necessary to push a copy of c onto the stack. There may be more opportunities for eliminating copy rules if more than one stack is used to hold attribute values. In the next example, we use separate stacks for synthesized and inherited attributes. A comparison with Example 5.25 shows that more copy rules can be eliminated if separate stacks are used.

Example 5.26. With the syntax-directed definition of Fig. 5.22, suppose we use separate stacks for the inherited attribute ps and the synthesized attribute ht. We maintain the stacks so that $B.ps$ is on top of the ps stack just before B is first visited and just after B is last visited. $B.ht$ will be on top of the ht stack just after B is visited.

With separate stacks we can take advantage of both the copy rules $B_1.ps := B.ps$ and $B_2.ps := B.ps$ associated with $B \rightarrow B_1 B_2$. As shown in Fig. 5.48, we do not need to push $B_1.ps$ because its value is already on top of the stack as $B.ps$.

A translation scheme based on the syntax-directed definition of Fig. 5.22 is shown in Fig. 5.49. Operation $push(v, s)$ pushes the value v onto stack s and

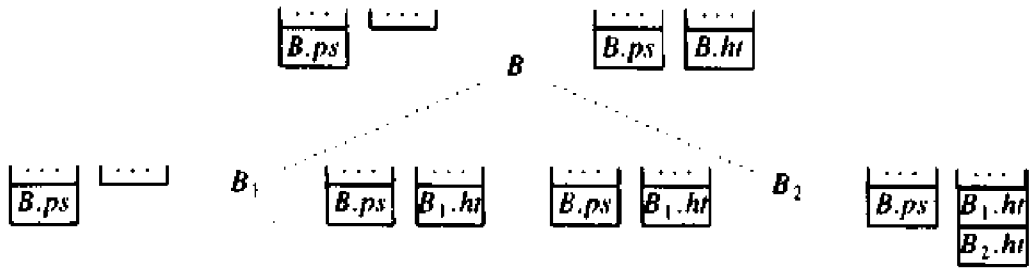


Fig. 5.48. Using separate stacks for attributes *ps* and *ht*

pop(*s*) pops the value on top of stack *s*. We use *top*(*s*) to refer to the top element of stack *s*. □

$S \rightarrow$ { *push*(10, *ps*) }
B

$B \rightarrow B_1$
B_2 { *h*2 := *top*(*ht*); *pop*(*ht*);
*h*1 := *top*(*ht*); *pop*(*ht*);
push(*max*(*h*1, *h*2), *ht*) }

$B \rightarrow B_1$
sub { *push*(*shrink*(*top*(*ps*)), *ps*) }
B_2 { *pop*(*ps*);
*h*2 := *top*(*ht*); *pop*(*ht*);
*h*1 := *top*(*ht*); *pop*(*ht*);
push(*disp*(*h*1, *h*2), *ht*) }

$B \rightarrow$ **text** { *push*(**text**.*h* × *top*(*ps*), *ht*) }

Fig. 5.49. Translation scheme maintaining stacks *ps* and *ht*.

The next example combines the use of a stack for attribute values with actions to emit code.

Example 5.27. Here, we consider techniques for implementing a syntax-directed definition specifying the generation of intermediate code. The value of a boolean expression *E* **and** *F* is false if *E* is false. In C, subexpression *F* must not be evaluated if *E* is false. The evaluation of such boolean expressions is considered in Section 8.4.

Boolean expressions in the syntax-directed definition in Fig. 5.50 are constructed from identifiers and the **and** operator. Each expression *E* inherits two labels *E.true* and *E.false* marking the points control must jump to if *E* is true and false, respectively.

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1$ **and** E_2	$E_1.true := newlabel$ $E_1.false := E.false$ $E_2.true := E.true$ $E_2.false := E.false$ $E.code := E_1.code \parallel gen(\text{'}\textbf{label}\text{'}\ E_1.true) \parallel E_2.code$
$E \rightarrow$ **id**	$E.code := gen(\text{'}\textbf{if}\text{'}\ \textbf{id}.place\ \text{'}\textbf{goto}\text{'}\ E.true) \parallel$ $gen(\text{'}\textbf{goto}\text{'}\ E.false)$

Fig. 5.50. Short-circuit evaluation of boolean expressions.

Suppose $E \rightarrow E_1$ **and** E_2. If E_1 evaluates to false, then control flows to the inherited label $E.false$; otherwise, E_1 evaluates to true so control flows to the code for evaluating E_2. A new label generated by function *newlabel* marks the beginning of the code for E_2. Individual instructions are formed using *gen*. For further discussion of the relevance of Fig. 5.50 to intermediate code generation see Section 8.4.

The syntax-directed definition in Fig. 5.50 is L-attributed, so we can construct a translation scheme for it. The translation scheme in Fig. 5.51 uses a procedure *emit* to generate and emit instructions incrementally. Also shown in the figure are actions for setting the values of the inherited attributes, inserted before the appropriate grammar symbol as discussed in Section 5.4.

$E \rightarrow$ { $E_1.true := newlabel$;
$E_1.false := E.false$ }
E_1
and { $emit(\text{'}\textbf{label}\text{'}\ E_1.true)$ };
$E_2.true := E.true$;
$E_2.false := E.false$ }
E_2

$E \rightarrow$ **id** { $emit(\text{'}\textbf{if}\text{'}\ \textbf{id}.place\ \text{'}\textbf{goto}\text{'}\ E.true)$;
$emit(\text{'}\textbf{goto}\text{'}\ E.false)$ }

Fig. 5.51. Emitting code for boolean expressions.

The translation scheme in Fig. 5.52 goes further; it uses separate stacks to hold the values of the inherited attributes $E.true$ and $E.false$. As in Example 5.26, copy rules have no effect on the stacks. To implement the rule $E_1.true := newlabel$, a new label is pushed onto the true stack before E_1 is visited. The lifetime of this label ends with the action $emit(\text{'}\textbf{label}\text{'}\ top(true))$, corresponding to $emit(\text{'}\textbf{label}\text{'}\ E_1.true)$, so the true stack is popped after the action. The *false* stack does not change in this example, but it is needed when the **or** operator is allowed in addition to the **and** operator. □

$E \rightarrow$ { *push*(*newlabel*, *true*) }
E_1
and { *emit*('**label**' *top*(*true*));
pop(*true*) }
E_2

$E \rightarrow$ **id** { *emit*('**if**' **id**.*place* '**goto**' *top*(*true*));
emit('**goto**' *top*(*false*)) }

Fig. 5.52. Emitting code for boolean expressions.

Nonoverlapping Lifetimes

A single register is a special case of a stack. If each *push* operation is followed by a *pop*, then there can be at most one element in the stack at a time. In this case, we can use a register instead of a stack. In terms of lifetimes, if the lifetimes of two attributes do not overlap, their values can be held in the same register.

Example 5.28. The syntax-directed definition in Fig. 5.53 constructs syntax trees for list-like expressions with operators at just one precedence level. It is taken from the translation scheme in Fig. 5.28.

PRODUCTION	SEMANTIC RULES
$E \rightarrow T\ R$	$R.i := T.nptr$ $E.nptr := R.s$
$R \rightarrow$ **addop** $T\ R_1$	$R_1.i := mknode$(**addop**.*lexeme*, $R.i$, $T.nptr$) $R.s := R_1.s$
$R \rightarrow \epsilon$	$R.s := R.i$
$T \rightarrow$ **num**	$T.nptr := mkleaf$(**num**, **num**.*val*)

Fig. 5.53. A syntax-directed definition adapted from Fig. 5.28.

We claim that the lifetime of each attribute of R ends when the attribute that depends on it is evaluated. We can show that, for any parse tree, the R attributes can be evaluated into the same register r. The following reasoning is typical of that needed to analyze grammars. The induction is on the size of the subtree attached at R in the parse tree fragment in Fig. 5.54.

The smallest subtree is obtained if $R \rightarrow \epsilon$ is applied, in which case $R.s$ is a copy of $R.i$, so both their values are in register r. For a larger subtree, the production at the root of the subtree must be for $R \rightarrow$ **addop** $T\ R_1$. The

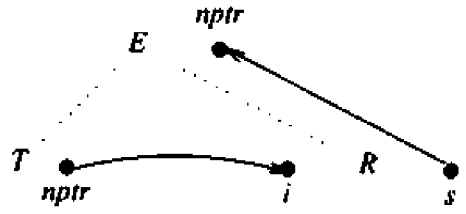


Fig. 5.54. Dependency graph for $E \rightarrow TR$.

lifetime of $R.i$ ends when $R_1.i$ is evaluated, so $R_1.i$ can be evaluated into register r. From the inductive hypothesis, all the attributes for instances of nonterminal R in the subtree for R_1 can be assigned the same register. Finally, $R.s$ is a copy of $R_1.s$, so its value is already in r.

The translation scheme in Fig. 5.55 evaluates the attributes in the attribute grammar of Fig. 5.53, using register r to hold the values of the attributes $R.i$ and $R.s$ for all instances of nonterminal R.

$E \rightarrow T$ { $r := T.nptr$ /* r now holds $R.i$ */ }
R { $E.nptr := r$ /* r has returned with $R.s$ */ }
$R \rightarrow$ **addop**
T { $r :=$ *mknode*(**addop**.*lexeme*, r, $T.nptr$) }
R
$R \rightarrow \epsilon$
$T \rightarrow$ **num** { $T.nptr :=$ *mkleaf*(**num**, **num**.*val*) }

Fig. 5.55. Transformed translation scheme for constructing syntax trees.

For completeness, we show in Fig. 5.56 the code for implementing the translation scheme above; it is constructed according to Algorithm 5.2. Nonterminal R no longer has attributes, so R becomes a procedure rather than a function. Variable r was made local to function E, so it is possible for E to be called recursively, although we do not need to do so in the scheme of Fig. 5.55. This code can be improved further by eliminating tail recursion and then replacing the remaining call of R by the body of the resulting procedure, as in Section 2.5. □

5.10 ANALYSIS OF SYNTAX-DIRECTED DEFINITIONS

In Section 5.7, attributes were evaluated during a traversal of a tree using a set of mutually recursive functions. The function for a nonterminal mapped the values of the inherited attributes at a node to the values of the synthesized attributes at that node.

```
function E: ↑ syntax_tree_node;
    var r: ↑ syntax_tree_node;
        addoplexeme : char;

    procedure R;
    begin
        if lookahead = addop then begin
            addoplexeme := lexval;
            match(addop);
            r := mknode(addoplexeme, r, T);
            R
        end
    end;

begin
    r := T; R
    return r
end;
```

Fig. 5.56. Compare procedure R with the code in Fig. 5.31.

The approach of Section 5.7 extends to translations that cannot be performed during a single depth-first traversal. Here, we shall use a separate function for each synthesized attribute of each nonterminal, although groups of synthesized attributes can be evaluated by a single function. The construction in Section 5.7 deals with the special case in which all synthesized attributes form one group. The grouping of attributes is determined from the dependencies set up by the semantic rules in a syntax-directed definition. The following abstract example illustrates the construction of a recursive evaluator.

Example 5.29. The syntax-directed definition in Fig. 5.57 is motivated by a problem we shall consider in Chapter 6. Briefly, the problem is as follows. An "overloaded" identifier can have a set of possible types; as a result, an expression can have a set of possible types. Context information is used to select one of the possible types for each subexpression. The problem can be solved by making a bottom-up pass to synthesize the set of possible types, followed by a top-down pass to narrow down the set to a single type.

The semantic rules in Fig. 5.57 are an abstraction of this problem. Synthesized attribute s represents the set of possible types and inherited attribute i represents the context information. An additional synthesized attribute t that cannot be evaluated in the same pass as s might represent the generated code or the type selected for a subexpression. Dependency graphs for the productions in Fig. 5.57 are shown in Fig. 5.58. □

PRODUCTION	SEMANTIC RULES
$S \rightarrow E$	$E.i := g(E.s)$
	$S.r := E.t$
$E \rightarrow E_1 E_2$	$E.s := fs(E_1.s, E_2.s)$
	$E_1.i := fi1(E.i)$
	$E_2.i := fi2(E.i)$
	$E.t := ft(E_1.t, E_2.t)$
$E \rightarrow$ id	$E.s :=$ id.s
	$E.t := h(E.i)$

Fig. 5.57. Synthesized attributes s and t cannot be evaluated together.

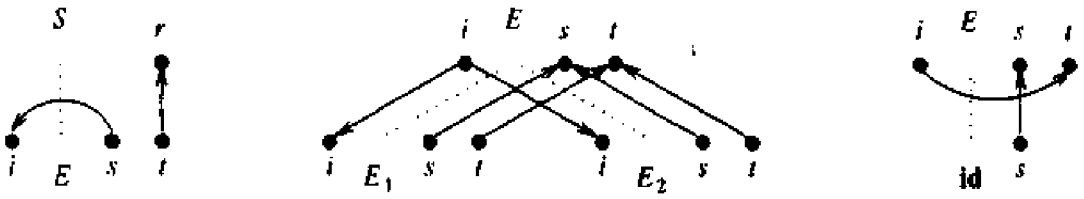


Fig. 5.58. Dependency graphs for productions in Fig. 5.57.

Recursive Evaluation of Attributes

The dependency graph for a parse tree is formed by pasting together smaller graphs corresponding to the semantic rules for a production. The dependency graph D_p for production p is based only on the semantic rules for a single production, i.e., on the semantic rules for the synthesized attributes of the left side and the inherited attributes of the grammar symbols on the right side of the production. That is, the graph D_p shows local dependencies only. For example, all edges in the dependency graph for $E \rightarrow E_1 E_2$ in Fig. 5.58 are between instances of the same attribute. From this dependency graph, we cannot tell that the s attributes must be computed before the other attributes.

A close look at the dependency graph for the parse tree in Fig. 5.59 shows that the attributes of each instance of nonterminal E must be evaluated in the order $E.s$, $E.i$, $E.t$. Note that all attributes in Fig. 5.59 can be evaluated in three passes: a bottom-up pass to evaluate the s attributes, a top-down pass to evaluate the i attributes, and a final bottom-up pass to evaluate the t attributes.

In a recursive evaluator, the function for a synthesized attribute takes the values of some of the inherited attributes as parameters. In general, if synthesized attribute $A.a$ can depend on inherited attribute $A.b$, then the function

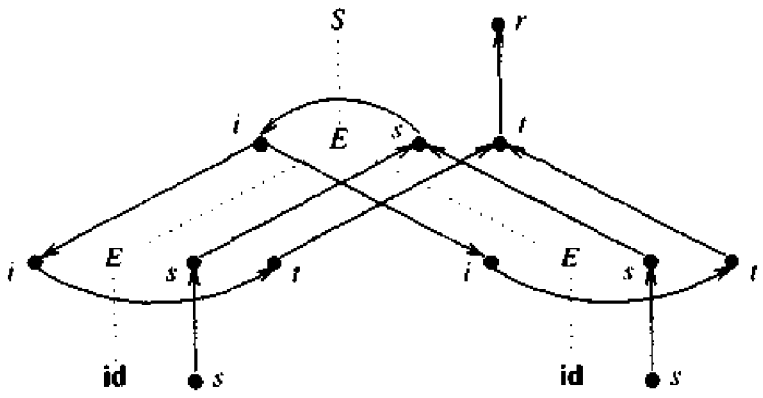


Fig. 5.59. Dependency graph for a parse tree.

for $A.a$ takes $A.b$ as a parameter. Before analyzing dependencies, we consider an example showing their use.

Example 5.30. The functions Es and Et in Fig. 5.60 return the values of the synthesized attributes s and t at a node n labeled E. As in Section 5.7, there is a case for each production in the function for a nonterminal. The code executed in each case simulates the semantic rules associated with the production in Fig. 5.57.

From the above discussion of the dependency graph in Fig. 5.59 we know that attribute $E.t$ at a node in a parse tree may depend on $E.i$. We therefore pass inherited attribute i as a parameter to the function Et for attribute t. Since attribute $E.s$ does not depend on any inherited attributes, function Es has no parameters corresponding to attribute values. □

Strongly Noncircular Syntax-Directed Definitions

Recursive evaluators can be constructed for a class of syntax-directed definitions, called "strongly noncircular" definitions. For a definition in this class, the attributes at each node for a nonterminal can be evaluated according to the same (partial) order. When we construct the function for a synthesized attribute of the nonterminal, this order is used to select the inherited attributes that become the parameters of the function.

We give a definition of this class and show that the syntax-directed definition in Fig. 5.57 falls in the class. We then give an algorithm for testing circularity and strong noncircularity, and show how the implementation of Example 5.30 extends to all strongly noncircular definitions.

Consider nonterminal A at a node n in a parse tree. The dependency graph for the parse tree may in general have paths that start at an attribute of node n, go through attributes of other nodes in the parse tree, and end at another attribute of n. For our purposes, it is enough to look at paths that stay within the part of the parse tree below A. A little thought reveals that such paths go

```
function Es(n);
begin
    case production at node n of
    'E → E1 E2':
        s1 := Es(child(n, 1));
        s2 := Es(child(n, 2));
        return fs(s1, s2);
    'E → id':
        return id.s;
    default:
        error
    end
end;

function Et(n, i);
begin
    case production at node n of
    'E → E1 E2':
        i1 := fi1(i);
        t1 := Et(child(n, 1), i1);
        i2 := fi2(i);
        t2 := Et(child(n, 2), i2);
        return ft(t1, t2);
    'E → id':
        return h(i);
    default:
        error
    end
end;

function Sr(n);
begin
    s := Es(child(n, 1));
    i := g(s);
    t := Et(child(n, 1), i);
    return t
end;
```

Fig. 5.60. Functions for synthesized attributes in Fig. 5.57.

from some inherited attribute of A to some synthesized attribute of A. We shall make an estimate (possibly too pessimistic) of the set of such paths by considering partial orders on the attributes of A.

Let production p have nonterminals $A_1, A_2, \ldots, A_n$ occurring on the right side. Let RA_j be a partial order on the attributes of A_j, for $1 \leq j \leq n$. We

write $D_p[RA_1, RA_2, \ldots, RA_n]$ for the graph obtained by adding edges to D_p as follows: if RA_j orders attribute $A_j.b$ before $A_j.c$ then add an edge from $A_j.b$ to $A_j.c$.

A syntax-directed definition is said to be *strongly noncircular* if for each nonterminal A we can find a partial order RA on the attributes of A such that for each production p with left side A and nonterminals $A_1, A_2, \ldots, A_n$ occurring on the right side

1. $D_p[RA_1, RA_2, \ldots, RA_n]$ is acyclic, and
2. if there is an edge from attribute $A.b$ to $A.c$ in $D_p[RA_1, RA_2, \ldots, RA_n]$, then RA orders $A.b$ before $A.c$.

Example 5.31. Let p be the production $E \rightarrow E_1E_2$ from Fig. 5.57, whose dependency graph D_p is in the center of Fig. 5.58. Let RE be the partial order (total order in this case) $s \rightarrow i \rightarrow t$. There are two occurrences of nonterminals on the right side of p, written E_1 and E_2, as usual. Thus, RE_1 and RE_2 are the same as RE, and the graph $D_p[RE_1, RE_2]$ is as shown in Fig. 5.61.

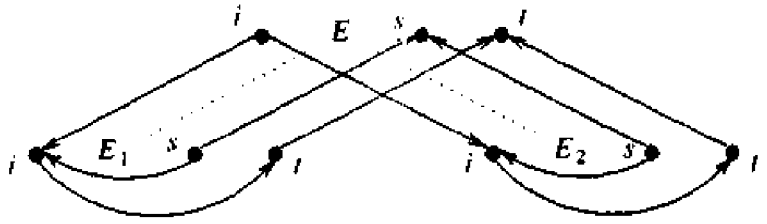


Fig. 5.61. Augmented dependency graph for a production.

Among the attributes associated with the root E in Fig. 5.61, the only paths are from i to t. Since RE makes i precede t, there is no violation of condition (2). □

Given a strongly noncircular definition and partial order RA for each nonterminal A, the function for synthesized attribute s of A takes arguments as follows: if RA orders inherited attribute i before s, then i is an argument of the function, otherwise not.

A Circularity Test

A syntax-directed definition is said to be circular if the dependency graph for some parse tree has a cycle; circular definitions are ill-formed and meaningless. There is no way we can begin to compute any of the attribute values on the cycle. Computing the partial orders that ensure that a definition is strongly noncircular is closely related to testing if a definition is circular. We shall therefore first consider a test for circularity.

Example 5.32. In the following syntax-directed definition, paths between the attributes of A depend on which production is applied. If $A \rightarrow 1$ is applied,

then $A.s$ depends on $A.i$; otherwise, it does not. For complete information about the possible dependencies we therefore have to keep track of sets of partial orders on the attributes of a nonterminal.

PRODUCTION	SEMANTIC RULES
$S \rightarrow A$	$A.i := c$
$A \rightarrow 1$	$A.s := f(A.i)$
$A \rightarrow 2$	$A.s := d$

□

The idea behind the algorithm in Fig. 5.62 is as follows. We represent partial orders by directed acyclic graphs. Given dags on the attributes of symbols on the right side of a production, we can determine a dag for the attributes of the left side as follows.

```
for grammar symbol X do
    ℱ(X) has a single graph with the attributes of X and no edges;
repeat
    change := false;
    for production p given by A → X1X2 · · · Xk do begin
        for dags G1∈ℱ(X1), . . . , Gk∈ℱ(Xk) do begin
            D := Dp;
            for edge b → c in Gj, 1≤j≤k do
                add an edge in D between attributes b and c of Xj;
            if D has a cycle then
                fail the circularity test
            else begin
                G := a new graph with nodes for the attributes
                    of A and no edges;
                for each pair of attributes b and c of A do
                    if there is a path in D from b to c then
                        add b → c to G;
                if G is not already in ℱ(A) then begin
                    add G to ℱ(A);
                    change := true
                end
            end
        end
    end
until change = false
```

Fig. 5.62. A circularity test.

Let production p be $A \rightarrow X_1X_2 \cdots X_k$ with dependency graph D_p. Let D_j be a dag for X_j, $1 \leq j \leq k$. Each edge $b \rightarrow a$ in D_j is temporarily added in to

the dependency graph D_p for the production. If the resulting graph has a cycle, then the syntax-directed definition is circular. Otherwise, paths in the resulting graph determine a new dag on the attributes of the left side of the production, and the resulting dag is added to $\mathcal{F}(A)$.

The circularity test in Fig. 5.62 takes time exponential in the number of graphs in the sets $\mathcal{F}(X)$ for any grammar symbol X. There are syntax-directed definitions that cannot be tested for circularity in polynomial time.

We can convert the algorithm in Fig. 5.62 into a more efficient test if a syntax-directed definition is strongly noncircular, as follows. Instead of maintaining a family of graphs $\mathcal{F}(X)$ for each X, we summarize the information in the family by keeping a single graph $F(X)$. Note that each graph in $\mathcal{F}(X)$ has the same nodes for the attributes of X, but may have different edges. $F(X)$ is the graph on the nodes for the attributes of X that has an edge between $X.b$ and $X.c$ if any graph in $\mathcal{F}(X)$ does. $F(X)$ represents a "worst-case estimate" of dependencies between attributes of X. In particular, if $F(X)$ is acyclic, then the syntax-directed definition is guaranteed to be noncircular. However, the converse need not be true; i.e., if $F(X)$ has a cycle, it is not necessarily true that the syntax-directed definition is circular.

The modified circularity test constructs acyclic graphs $F(X)$ for each X if it succeeds. From these graphs we can construct an evaluator for the syntax-directed definition. The method is a straightforward generalization of Example 5.30. The function for synthesized attribute $X.s$ takes as arguments all and only the inherited attributes that precede s in $F(X)$. The function, called at node n, calls other functions that compute the needed synthesized attributes at the children of n. The routines to compute these attributes are passed values for the inherited attributes they need. The fact that the strong noncircularity test succeeded guarantees that these inherited attributes can be computed.

EXERCISES

5.1 For the input expression `(4*7+1)*2`, construct an annotated parse tree according to the syntax-directed definition of Fig. 5.2

5.2 Construct the parse tree and syntax tree for the expression `((a)+(b))` according to
a) the syntax-directed definition of Fig. 5.9, and
b) the translation scheme of Fig. 5.28.

5.3 Construct the dag and identify the value numbers for the subexpressions of the following expression, assuming + associates from the left: `a+a+(a+a+a+(a+a+a+a))`.

***5.4** Give a syntax-directed definition to translate infix expressions into infix expressions without redundant parentheses. For example, since + and * associate to the left, `((a*(b+c))*(d))` can be rewritten as `a*(b+c)*d`.