4 Program Semantics

4.6 Attribute Grammars

In this section we examine *attribute grammars*, an alternate formalism for describing and implementing the semantics of a programming language. Intuitively, we can think of attribute grammars as a generalization of action routines in which the compiler designer no longer needs to specify exactly when to execute each routine—and in which the execution need not necessarily be interleaved with parsing. Alternatively, we can think of attribute grammars as a more imperative alternative to inference rules: instead of providing rules that indicate what can be inferred about the meaning of nodes in a syntax tree, we provide code to compute the values of attributes (fields) of tree nodes, either in a syntax tree or in the original parse tree.

EXAMPLE 4.24
Bottom-up CFG for constant expressions

As a starting point, in a parse tree context, consider an LR (bottom-up) grammar for arithmetic expressions composed of constants with precedence and associativity, adapted from Example 2.8:[1]

$E \longrightarrow E$ `+` T
$E \longrightarrow E$ `-` T
$E \longrightarrow T$
$T \longrightarrow T$ `*` F
$T \longrightarrow T$ `/` F
$T \longrightarrow F$
$F \longrightarrow$ `-` F
$F \longrightarrow$ `(` E `)`
$F \longrightarrow$ `const`

1 The addition of semantic rules tends to make attribute grammars quite a bit more verbose than context-free grammars. For the sake of brevity, many of the examples in this section use very short symbol names: *E* instead of *expr*, *TT* instead of *term_tail*.

1.	$E_1 \longrightarrow E_2$ `+` T	▷ E_1.val := sum(E_2.val, T.val)
2.	$E_1 \longrightarrow E_2$ `-` T	▷ E_1.val := difference(E_2.val, T.val)
3.	$E \longrightarrow T$	▷ E.val := T.val
4.	$T_1 \longrightarrow T_2$ `*` F	▷ T_1.val := product(T_2.val, F.val)
5.	$T_1 \longrightarrow T_2$ `/` F	▷ T_1.val := quotient(T_2.val, F.val)
6.	$T \longrightarrow F$	▷ T.val := F.val
7.	$F_1 \longrightarrow$ `-` F_2	▷ F_1.val := additive_inverse(F_2.val)
8.	$F \longrightarrow$ `(` E `)`	▷ F.val := E.val
9.	$F \longrightarrow$ `const`	▷ F.val := const.val

Figure 4.16 A simple attribute grammar for constant expressions, using the standard arithmetic operations. Each semantic rule is introduced by a ▷ sign.

EXAMPLE 4.25
Bottom-up AG for constant expressions

This grammar will generate all properly formed constant expressions over the basic arithmetic operators, but it says nothing about their meaning. To tie these expressions to mathematical concepts (as opposed to, say, floor tile patterns or dance steps), we could use inference rules, as discussed in the main text, but we can also use *attributes*. In our expression grammar, we associate a val attribute with each E, T, F, and `const` in the grammar. The intent is that for any symbol S, S.val will be the meaning, as an arithmetic value, of the token string derived from S. We assume that the val of a `const` is provided to us by the scanner. We must then invent a set of rules for each production, to specify how the vals of different symbols are related. The resulting *attribute grammar* (AG) is shown in Figure C-4.16.

In this simple grammar, every production has a single rule. We shall see more complicated grammars later, in which productions can have several rules. The rules come in two forms. Those in productions 3, 6, 8, and 9 are known as *copy rules*; they specify that one attribute should be a copy of another. The other rules invoke *semantic functions* (sum, quotient, additive_inverse, etc.). In this example, the semantic functions are all familiar arithmetic operations. In general, they can be arbitrarily complex functions specified by the language designer. Each semantic function takes an arbitrary number of arguments (each of which must be an attribute of a symbol in the current production—no global variables are allowed), and each computes a single result, which must likewise be assigned into an attribute of a symbol in the current production. When more than one symbol of a production has the same name, subscripts are used to distinguish them. These subscripts are solely for the benefit of the semantic functions; they are not part of the context-free grammar itself. ■

In a strict definition of attribute grammars, copy rules and semantic function calls are the only two kinds of permissible rules. In our examples we use a ▷ symbol to introduce each code fragment corresponding to a single rule. In practice, it is common to allow rules to consist of small fragments of code in some well-defined notation (e.g., the language in which a compiler is being written), so that simple

EXAMPLE 4.26
Top-down AG to count the elements of a list

semantic functions can be written out "in-line." In this relaxed notation, the rule for the first production in Figure C-4.16 might be simply E_1.val := E_2.val + T.val. As another example, suppose we wanted to count the elements of a comma-separated list:

$L \longrightarrow$ `id` LT	▷ L.c := 1 + LT.c
$LT \longrightarrow$ `,` L	▷ LT.c := L.c
$LT \longrightarrow \varepsilon$	▷ LT.c := 0

Here the rule on the first production sets the c (count) attribute of the left-hand side to one more than the count of the tail of the right-hand side. Like explicit semantic functions, in-line rules are not allowed to refer to any variables or attributes outside the current production. We will relax this restriction when we relate attribute grammars to action routines in Subsection C-4.6.2. ■

Neither the notation for semantic functions (whether in-line or explicit) nor the types of the attributes themselves is intrinsic to the notion of an attribute grammar. The purpose of the grammar is simply to associate meaning with the nodes of a parse tree or syntax tree. Toward that end, we can use any notation and types whose meanings are already well defined. In Examples C-4.25 and C-4.26, we associated numeric values with the symbols in a CFG—and thus with parse tree nodes—using semantic functions drawn from ordinary arithmetic. In a compiler or interpreter for a full programming language, the attributes of tree nodes might include

- for an identifier, a reference to information about it in the symbol table
- for an expression, its type
- for a statement or expression, a reference to corresponding code in the compiler's intermediate form
- for almost any construct, an indication of the file name, line, and column where the corresponding source code begins
- for any internal node, a list of semantic errors found in the subtree below

For purposes other than translation—for example, in a theorem prover or machine-independent language definition—attributes might be drawn from the disciplines of denotational, operational, or axiomatic semantics. Operational semantics were discussed in Section 4.3; interested readers can find references to other alternatives in the Bibliographic Notes at the end of the chapter.

4.6.1 Evaluating Attributes

EXAMPLE 4.27
Decoration of a parse tree

The process of evaluating attributes is called *annotation* or *decoration* of the parse tree (it also applies to syntax trees, as we shall see in Section C-4.6.3). Figure C-4.17 shows how to decorate the parse tree for the expression `(1 + 3) * 2`, using the AG of Figure C-4.16. Once decoration is complete, the value of the overall expression can be found in the val attribute of the root of the tree. ■

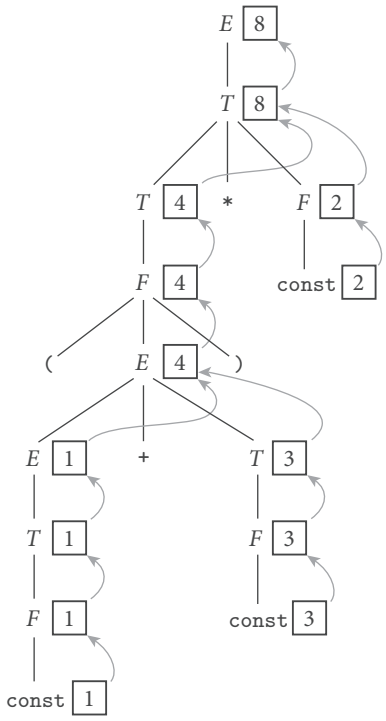


Figure 4.17 Decoration of a parse tree for `(1 + 3) * 2`, using the attribute grammar of Figure C-4.16. The val attributes of symbols are shown in boxes. Curving arrows show the attribute flow, which is strictly upward in this case. Each box holds the output of a single semantic rule; the arrow(s) entering the box indicate the input(s) to the rule. At the second level of the tree, for example, the two arrows pointing into the box with the 8 represent application of the rule T_1.val := product(T_2.val, F.val).

Synthesized Attributes

The attribute grammar of Figure C-4.16 is very simple. Each symbol has at most one attribute (the punctuation marks have none). Moreover, they are all so-called *synthesized attributes*: their values are calculated (synthesized) only in productions in which their symbol appears on the left-hand side. For annotated parse trees like the one in Figure C-4.17, this means that the *attribute flow*—the pattern in which information moves from node to node—is entirely bottom-up.

An attribute grammar in which all attributes are synthesized is said to be *S-attributed*. The arguments to semantic functions in an S-attributed grammar are always attributes of symbols on the right-hand side of the current production, and the return value is always placed into an attribute of the left-hand side of the production. Tokens (terminals) often have intrinsic properties (e.g., the character-string representation of an identifier or the value of a numeric constant); in a compiler these are synthesized attributes initialized by the scanner.

Inherited Attributes

When we considered the construction of syntax trees during top-down parsing (Example 4.6 and Figure 4.6), we found that we needed to place action routines *within* the right-hand sides of productions, so that the left operands of an arithmetic operator could be passed into the subtree that would contain the right operand. In a similar vein—and for similar reasons—we will encounter situations in which attribute values will need to be calculated when their symbol is on the right-hand side of the current production. Such attributes are said to be *inherited*. They allow contextual information to flow into a symbol from above or from the side, so that the rules of that production can be enforced in different ways (or generate different values) depending on surrounding context. Symbol table information is commonly passed from symbol to symbol by means of inherited attributes. Inherited attributes of the root of the parse tree can also be used to represent the external environment (characteristics of the target machine, command-line arguments to the compiler, etc.).

EXAMPLE 4.28

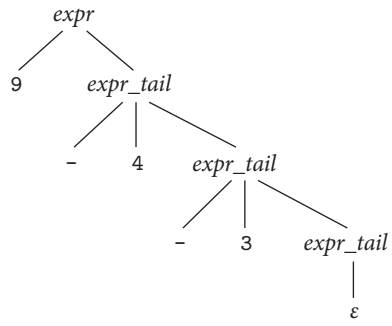
Top-down CFG and parse tree for subtraction

As a simple example of inherited attributes, consider the following fragment of an LL(1) expression grammar (here covering only subtraction):

expr ⟶ `const` *expr_tail*

expr_tail ⟶ `-` `const` *expr_tail* | ε

For the expression `9 - 4 - 3`, we obtain the following parse tree:



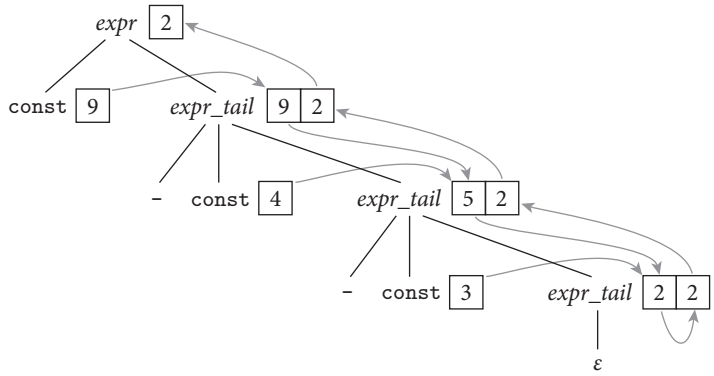
If we want to create an attribute grammar that accumulates the value of the overall expression into the root of the tree, we have a problem: because subtraction is left associative, we cannot summarize the right subtree of the root with a single numeric value. If we want to decorate the tree bottom-up, with an S-attributed grammar, we must be prepared to describe an arbitrary number of right operands in the attributes of the top-most *expr_tail* node (see Exercise C-4.23). This is indeed possible, but it defeats the purpose of the formalism: in effect, it requires us to embed the entire tree into the attributes of a single node, and do all the real work inside a single semantic function at the root.

EXAMPLE 4.29

Decoration with left-to-right attribute flow

If, however, we are allowed to pass attribute values not only bottom-up but also left-to-right in the tree, then we can pass the `9` into the top-most *expr_tail* node, where it can be combined (in proper left-associative fashion) with the `4`. The

resulting 5 can then be passed into the middle *expr_tail* node, combined with the 3 to make 2, and then passed upward to the root:



EXAMPLE 4.30
Top-down AG for subtraction

To effect this style of decoration, we need the following attribute rules:

expr ⟶ `const` *expr_tail*
- ▷ expr_tail.st := const.val
- ▷ expr.val := expr_tail.val

$expr\_tail_1$ ⟶ `-` `const` $expr\_tail_2$
- ▷ $expr\_tail_2$.st := $expr\_tail_1$.st − const.val
- ▷ $expr\_tail_1$.val := $expr\_tail_2$.val

expr_tail ⟶ ε
- ▷ expr_tail.val := expr_tail.st

In each of the first two productions, the first rule serves to copy the left context (value of the expression so far) into a "subtotal" (st) attribute; the second rule copies the final value from the right-most leaf back up to the root. In the *expr_tail* nodes of the picture in Example C-4.29, the left box holds the st attribute; the right holds val.

EXAMPLE 4.31
Top-down AG for constant expressions

We can flesh out the grammar fragment of Example C-4.28 to produce a more complete expression grammar, as shown (with shorter symbol names) in Figure C-4.18. The underlying CFG for this grammar accepts the same language as the one in Figure C-4.16, but where that one was SLR(1), this one is LL(1). Attribute flow for a parse of `(1 + 3) * 2`, using the LL(1) grammar, appears in Figure C-4.19. As in the grammar fragment of Example C-4.30, the value of the left operand of each operator is carried into the *TT* and *FT* productions by the st (subtotal) attribute. The relative complexity of the attribute flow arises from the fact that operators are left associative, but the grammar cannot be left recursive: the left and right operands of a given operator are thus found in separate productions. Grammars to perform semantic analysis for practical languages generally require some non-S-attributed flow.

1. $E \longrightarrow$ `T TT`
   ▷ TT.st := T.val ▷ E.val := TT.val
2. $TT_1 \longrightarrow$ `+ T` TT_2
   ▷ TT_2.st := TT_1.st + T.val ▷ TT_1.val := TT_2.val
3. $TT_1 \longrightarrow$ `- T` TT_2
   ▷ TT_2.st := TT_1.st − T.val ▷ TT_1.val := TT_2.val
4. $TT \longrightarrow \varepsilon$
   ▷ TT.val := TT.st
5. $T \longrightarrow F\ FT$
   ▷ FT.st := F.val ▷ T.val := FT.val
6. $FT_1 \longrightarrow$ `* F` FT_2
   ▷ FT_2.st := FT_1.st × F.val ▷ FT_1.val := FT_2.val
7. $FT_1 \longrightarrow$ `/ F` FT_2
   ▷ FT_2.st := FT_1.st ÷ F.val ▷ FT_1.val := FT_2.val
8. $FT \longrightarrow \varepsilon$
   ▷ FT.val := FT.st
9. $F_1 \longrightarrow$ `-` F_2
   ▷ F_1.val := − F_2.val
10. $F \longrightarrow$ `( E )`
    ▷ F.val := E.val
11. $F \longrightarrow$ `const`
    ▷ F.val := const.val

Figure 4.18 **An attribute grammar for constant expressions based on an LL(1) CFG.** In this grammar several productions have two semantic rules.

Attribute Flow

Just as a context-free grammar does not specify how it should be parsed, an attribute grammar does not specify the order in which attribute rules should be invoked. Put another way, both notations are *declarative*: they define a set of valid parse trees, but they don't say how to build or decorate them. Among other things, this means that the order in which attribute rules are listed for a given production is immaterial; attribute flow may require them to execute in any order. If, in Figure C-4.18, we were to reverse the order in which the rules appear in productions 1, 2, 3, 5, 6, and/or 7 (listing the rule for symbol.val first), it would be a purely cosmetic change; the grammar would not be altered.

We say an attribute grammar is *well defined* if its rules determine a unique set of values for the attributes of every possible parse tree. An attribute grammar is *noncircular* if it never leads to a parse tree in which there are cycles in the attribute flow graph—that is, if no attribute, in any parse tree, ever depends (transitively) on itself. (A grammar can be circular and still be well defined if attributes are guaranteed to converge to a unique value.) As a general rule, practical attribute grammars tend to be noncircular.

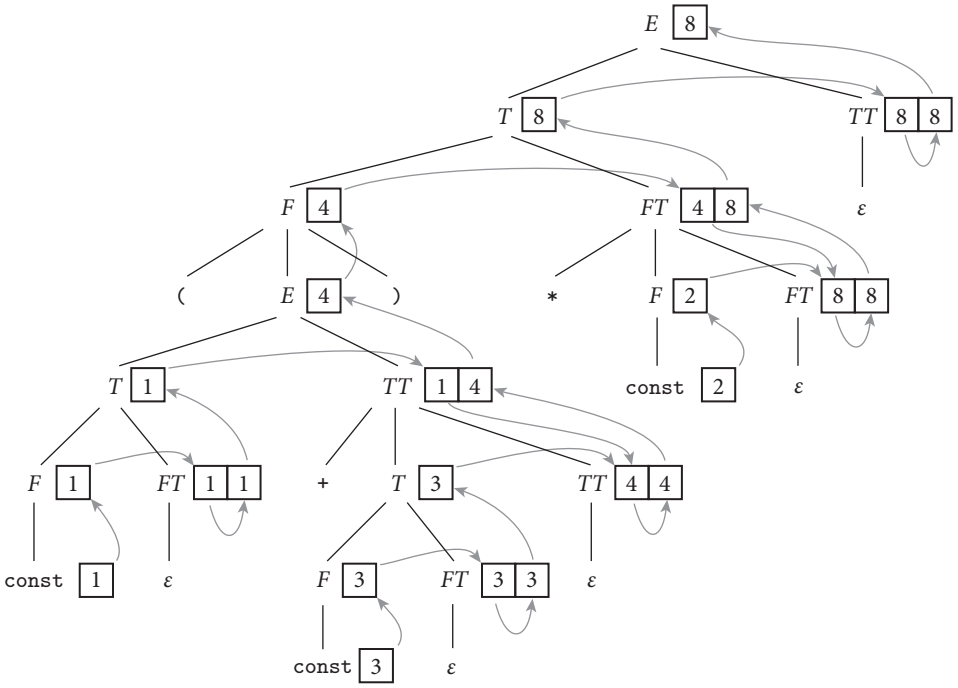


Figure 4.19 Decoration of a top-down parse tree for `(1 + 3) * 2`, using the AG of Figure C-4.18. Curving arrows again indicate attribute flow; the arrow(s) entering a given box represent the application of a single semantic rule. Flow in this case is no longer strictly bottom-up, but it is still left-to-right. At *FT* and *TT* nodes, the left box holds the st attribute; the right holds `val`.

An algorithm that decorates parse trees by invoking the rules of an attribute grammar in an order consistent with the tree's attribute flow is called a *translation scheme*. Perhaps the simplest scheme is one that makes repeated passes over a tree, invoking any semantic function whose arguments have all been defined, and stopping when it completes a pass in which no values change. Such a scheme is said to be *oblivious*, in the sense that it exploits no special knowledge of either the parse tree or the grammar. It will halt only if the grammar is well defined. Better performance, at least for noncircular grammars, may be achieved by a *dynamic* scheme that tailors the evaluation order to the structure of a given parse tree—for example, by constructing a topological sort of the attribute flow graph and then invoking rules in an order consistent with the sort.

The fastest translation schemes, however, tend to be *static*—based on an analysis of the structure of the attribute grammar itself, and then applied mechanically to any tree arising from the grammar. Like LL and LR parsers, linear-time static translation schemes can be devised only for certain restricted classes of grammars. S-attributed grammars, such as the one in Figure C-4.16, form the simplest such class. Because attribute flow in an S-attributed grammar is strictly bottom-up,

attributes can be evaluated by visiting the nodes of the parse tree in exactly the same order that those nodes are generated by an LR-family parser. In fact, the attributes can be evaluated on the fly during a bottom-up parse, thereby interleaving parsing and semantic analysis (attribute evaluation).

The attribute grammar of Figure C-4.18 is a good bit messier than that of Figure C-4.16, but it is still *L-attributed*: its attributes can be evaluated by visiting the nodes of the parse tree in a single left-to-right, depth-first traversal (the same order in which they are visited during a top-down parse—see Figure C-4.19). If we say that an attribute A.s *depends on* an attribute B.t if B.t is ever passed to a semantic function that returns a value for A.s, then we can define L-attributed grammars more formally with the following two rules: (1) each synthesized attribute of a left-hand-side symbol depends only on that symbol's own inherited attributes or on attributes (synthesized or inherited) of the production's right-hand-side symbols, and (2) each inherited attribute of a right-hand-side symbol depends only on inherited attributes of the left-hand-side symbol or on attributes (synthesized or inherited) of symbols to its left in the right-hand side.

Because L-attributed grammars permit rules that initialize attributes of the left-hand side of a production using attributes of symbols on the right-hand side, every S-attributed grammar is also an L-attributed grammar. The reverse is not the case: S-attributed grammars do not permit the initialization of attributes on the right-hand side, so there are L-attributed grammars that are not S-attributed.

S-attributed attribute grammars are the most general class of attribute grammars for which evaluation can be implemented on the fly during an LR parse. L-attributed grammars are the most general class for which evaluation can be implemented on the fly during an LL parse. If we interleave semantic analysis (and possibly intermediate code generation) with parsing, then a bottom-up parser must in general be paired with an S-attributed translation scheme; a top-down parser must be paired with an L-attributed translation scheme. (Depending on the structure of the grammar, it is often possible for a bottom-up parser to accommodate some non-S-attributed attribute flow; we consider this possibility in Section C-4.6.4.) If we choose to separate parsing and semantic analysis into separate passes, then the code that builds the parse tree or syntax tree must still use an S-attributed or L-attributed translation scheme (as appropriate), but the semantic analyzer can use a more powerful scheme if desired. There are certain tasks that are easiest to accomplish with a non-L-attributed scheme. Examples include the generation of code for "short-circuit" Boolean expressions (to be discussed in Sections 6.1.5 and 6.4.1) and the type checking of mutually recursive functions (Section 3.3.3).

Building a Syntax Tree

EXAMPLE 4.32

Bottom-up and top-down AGs to build a syntax tree

If we choose not to interleave parsing and semantic analysis, we still need to add attribute rules to the context-free grammar, but they serve only to create the syntax tree—not to enforce semantic rules or generate code. Figures C-4.20 and C-4.21 contain bottom-up and top-down attribute grammars, respectively, to build a syntax tree for constant expressions. The attributes in these grammars hold neither numeric values nor target code fragments; instead they point to nodes

$E_1 \longrightarrow E_2$ `+` T
▷ E_1.ptr := bin_op(E_2.ptr, ''+'', T.ptr)

$E_1 \longrightarrow E_2$ `-` T
▷ E_1.ptr := bin_op(E_2.ptr, ''–'', T.ptr)

$E \longrightarrow T$
▷ E.ptr := T.ptr

$T_1 \longrightarrow T_2$ `*` F
▷ T_1.ptr := bin_op(T_2.ptr, ''×'', F.ptr)

$T_1 \longrightarrow T_2$ `/` F
▷ T_1.ptr := bin_op(T_2.ptr, ''÷'', F.ptr)

$T \longrightarrow F$
▷ T.ptr := F.ptr

$F_1 \longrightarrow$ `-` F_2
▷ F_1.ptr := un_op(''+/–'', F_2.ptr)

$F \longrightarrow$ `(` E `)`
▷ F.ptr := E.ptr

$F \longrightarrow$ `const`
▷ F.ptr := int_lit(const.val)

Figure 4.20 **Bottom-up (S-attributed) attribute grammar to construct a syntax tree.** The symbol +/– is used (as it is on calculators) to indicate change of sign.

of the syntax tree. Function int_lit returns a pointer to a newly allocated syntax tree node containing the value of a constant. Functions un_op and bin_op return pointers to newly allocated syntax tree nodes containing a unary or binary operator, respectively, and pointers to the supplied operand(s). Bottom-up and top-down construction of syntax trees for `(1 + 3) * 2` is analogous to that of Figures 4.5 and 4.8, respectively, in the main text.

4.6.2 Action Routines and Attribute Grammars

The astute reader will have noticed the similarity between Figures 4.4 and C-4.20, and between Figures 4.6 and C-4.21. Indeed, the action routines we introduced in Section 4.2 are simply an implementation, provided by most parser-generator tools, of attribute grammars with a manually specified static translation scheme. Each action routine is a semantic function that the programmer (grammar writer) instructs the compiler to execute at a particular point in the parse.

One difference between the action routines of Figures 4.4 and 4.6 and the semantic functions of attribute grammars is that the former just return a value, while the former can set multiple attributes. While some tools (e.g., the University of Minnesota's attribute grammar-based Silver system) allow an action routine to modify multiple attributes, many popular parser generators, including `yacc/bison`

$E \longrightarrow$ T TT
▷ TT.st := T.ptr
▷ E.ptr := TT.ptr

$TT_1 \longrightarrow$ `+` T TT_2
▷ TT_2.st := bin_op(TT_1.st, ''+'', T.ptr)
▷ TT_1.ptr := TT_2.ptr

$TT_1 \longrightarrow$ `-` T TT_2
▷ TT_2.st := bin_op(TT_1.st, ''–'', T.ptr)
▷ TT_1.ptr := TT_2.ptr

$TT \longrightarrow$ ε
▷ TT.ptr := TT.st

$T \longrightarrow$ F FT
▷ FT.st := F.ptr
▷ T.ptr := FT.ptr

$FT_1 \longrightarrow$ `*` F FT_2
▷ FT_2.st := bin_op(FT_1.st, ''×'', F.ptr)
▷ FT_1.ptr := FT_2.ptr

$FT_1 \longrightarrow$ `/` F FT_2
▷ FT_2.st := bin_op(FT_1.st, ''÷'', F.ptr)
▷ FT_1.ptr := FT_2.ptr

$FT \longrightarrow$ ε
▷ FT.ptr := FT.st

$F_1 \longrightarrow$ `-` F_2
▷ F_1.ptr := un_op(''+/–'', F_2.ptr)

$F \longrightarrow$ `(` E `)`
▷ F.ptr := E.ptr

$F \longrightarrow$ `const`
▷ F.ptr := int_lit(const.val)

Figure 4.21 Top-down (L-attributed) attribute grammar to construct a syntax tree. Here the st attribute, like the ptr attribute (and unlike the st attribute of Figure C-4.18), is a pointer to a syntax tree node.

and JavaCC, provide only the simpler return-value mechanism. In these tools, an action routine that needs to modify more than one attribute can return a record with a separate field for each.

CHECK YOUR UNDERSTANDING

39. What is an *attribute grammar*?

40. What is the difference between *synthesized* and *inherited* attributes?

41. Give two examples of information that is typically passed through inherited attributes.

42. What is *attribute flow*?
43. What does it mean for an attribute grammar to be *S-attributed*? *L-attributed*? *Noncircular*? What is the significance of these grammar classes?
44. What is the difference between a semantic function and an action routine?

4.6.3 Semantic Analysis with Attribute Grammars

In our discussion so far we have used attribute grammars solely to decorate parse trees. Attribute grammars can also be used, however, to decorate syntax trees. To define semantic analyses over syntax trees using attribute grammars, we can simply attach semantic rules to the productions of an abstract grammar. These rules define the attribute flow of a syntax tree in exactly the same way that semantic rules attached to the productions of a context-free grammar are used to define the attribute flow of a parse tree. We will use an abstract grammar in the remainder of this section to perform static semantic checking. Additional semantic rules could be used to generate intermediate code.

EXAMPLE 4.33
Abstract AG for the calculator language with types

A complete abstract attribute grammar for our calculator language with types can be constructed using the node classes, variants, and attributes shown in Figure C-4.22. The grammar itself appears in Figure C-4.23. Once decorated, the *program* node at the root of the syntax tree will contain a list, in a synthesized attribute, of all static semantic errors in the program. (The list will be empty if the program is free of such errors.) Each *stmt* or *expr* node has an inherited attribute symtab that contains a list, with types, of all identifiers declared to the left in the tree. Each *stmt* node also has an inherited attribute errors_in that lists all static semantic errors found to its left in the tree, and a synthesized attribute errors_out to propagate the final error list back to the root. Each *expr* node has one synthesized attribute that indicates its type and another that contains a list of any static semantic errors found inside. To avoid cascading messages when an error is found early in

DESIGN & IMPLEMENTATION

4.6 Attribute evaluators

Automatic evaluators based on formal attribute grammars are popular in language research projects because they save developer time when the language definition changes. They are popular in syntax-based editors and incremental compilers because they save execution time: when a small change is made to a program, the evaluator may be able to "patch up" tree decorations significantly faster than it could rebuild them from scratch. For the typical compiler, however, semantic analysis based on a formal attribute grammar is overkill: it has higher overhead than action routines or ad-hoc traversal of a syntax tree, and doesn't really save the compiler writer that much work.

Class of node	Variants	Attributes	
		Inherited	**Synthesized**
program	—	—	location, errors
stmt	*int_decl, real_decl, assign, read, write, null*	symtab, errors_in	location, errors_out
expr	*int_lit, real_lit, var, bin_op, float, trunc*	symtab	location, type, errors name (*var* only)

Figure 4.22 Classes of nodes for the abstract attribute grammar of Figure C-4.23. All variants of a given class have all the class's attributes.

our pass over the syntax tree, we adopt the technique introduced in Section 4.4.2: we associate a pseudotype called error with any symbol table entry or expression for which we have already generated a message.

Though it takes a bit of checking to verify the fact, our attribute grammar is noncircular and well defined. No attribute is ever assigned a value more than once. (The helper routines at the end of Figure C-4.23 should be thought of as macros, rather than semantic functions. For the sake of brevity we have passed them entire tree nodes as arguments. Each macro calculates the values of two different attributes. Under a strict formulation of attribute grammars each macro would be replaced by two separate semantic functions, one per calculated attribute.)

EXAMPLE 4.34

Decorating a tree with the AG of the previous Example

Figure C-4.24 uses the grammar of Figure C-4.23 to decorate the syntax tree of Figure 4.2. The pattern of attribute flow appears considerably messier than in previous examples in this section, but this is simply because type checking is more complicated than calculating constants or building a syntax tree. Symbol table information flows along the chain of *stmt*s and down into *expr* trees. The *int_decl* and *real_decl* nodes add new information; other nodes simply pass the table along. Ideally, when an undeclared identifier is encountered, we would enter it into the symbol table with an "error" designation, to suppress further messages about the same identifier; we have not shown that code here.

Type information is synthesized at *var*, *assign*, and *expr* leaves by looking up an identifier's name in the symbol table. The information then propagates upward within an expression tree, and is used to type-check operators and assignments (the latter don't appear in this example). Error messages flow along the chain of *stmt*s via the errors_in attributes, and then back to the root via the errors_out attributes. Messages also flow up out of *expr* trees. Wherever a type check is performed, the type attribute may be used to help create a new message to be appended to the growing message list.

In our example grammar we accumulate error messages into a synthesized attribute of the root of the syntax tree. In an ad hoc attribute evaluator we might be tempted to print these messages on the fly as the errors are discovered. In practice, however, particularly in a multipass compiler, it makes sense to buffer the messages, so they can be interleaved with messages produced by other phases of the compiler, and printed in program order at the end of compilation.

program ⟶ *stmt*
- ▷ stmt.symtab := null
- ▷ program.errors := stmt.errors_out
- ▷ stmt.errors_in := null

$stmt_1$ ⟶ `int` *id* $stmt_2$
- ▷ declare_name(id.name, $stmt_1$, $stmt_2$, int)
- ▷ $stmt_1$.errors_out := $stmt_2$.errors_out

$stmt_1$ ⟶ `real` *id* $stmt_2$
- ▷ declare_name(id.name, $stmt_1$, $stmt_2$, real)
- ▷ $stmt_1$.errors_out := $stmt_2$.errors_out

$stmt_1$ ⟶ `read` *id* $stmt_2$
- ▷ $stmt_2$.symtab := $stmt_1$.symtab
- ▷ if ⟨id.name, ?⟩ ∈ $stmt_1$.symtab
  - $stmt_2$.errors_in := $stmt_1$.errors_in
  - else
    - $stmt_2$.errors_in := $stmt_1$.errors_in + [id.name ''undefined at'' id.location]
- ▷ $stmt_1$.errors_out := $stmt_2$.errors_out

$stmt_1$ ⟶ `write` *expr* $stmt_2$
- ▷ expr.symtab := $stmt_1$.symtab
- ▷ $stmt_2$.symtab := $stmt_1$.symtab
- ▷ $stmt_2$.errors_in := $stmt_1$.errors_in + expr.errors
- ▷ $stmt_1$.errors_out := $stmt_2$.errors_out

$stmt_1$ ⟶ *id* := *expr* $stmt_2$
- ▷ expr.symtab := $stmt_1$.symtab
- ▷ $stmt_2$.symtab := $stmt_1$.symtab
- ▷ if ⟨id.name, A⟩ ∈ $stmt_1$.symtab –– for some type A
  - if A ≠ error and expr.type ≠ error and A ≠ expr.type
    - $stmt_2$.errors_in := $stmt_1$.errors_in + [''type clash at'' :=.location]
  - else
    - $stmt_2$.errors_in := $stmt_1$.errors_in + expr.errors
  - else
    - $stmt_2$.errors_in := $stmt_1$.errors_in + [id.name ''undefined at'' id.location] + expr.errors
- ▷ $stmt_1$.errors_out := $stmt_2$.errors_out

null : stmt ⟶ ε
- ▷ stmt.errors_out := stmt.errors_in

Figure 4.23 Attribute grammar to decorate an abstract syntax tree for the calculator language with types. We use square brackets to delimit error messages and pointed brackets to delimit symbol table entries. Juxtaposition indicates concatenation within error messages; the '+' and '–' operators indicate insertion and removal in lists. We assume that every node has been initialized by the scanner or by action routines in the parser to contain an indication of the location (line and column) at which the corresponding construct appears in the source (see Exercise C-4.36). The '?' symbol is used as a "wild card"; it matches any type. *(continued)*

```
expr ⟶ var
    ▷ if ⟨var.name, A⟩ ∈ expr.symtab          -- for some type A
          expr.errors := null
          expr.type := A
      else
          expr.errors := [var.name ''undefined at'' var.location]
          expr.type := error

expr ⟶ n
    ▷ expr.type := int

expr ⟶ r
    ▷ expr.type := real

expr1 ⟶ expr2 op expr3
    ▷ expr2.symtab := expr1.symtab
    ▷ expr3.symtab := expr1.symtab
    ▷ check_types(expr1, expr2, op, expr3)

expr1 ⟶ float(expr2)
    ▷ expr2.symtab := expr1.symtab
    ▷ convert_type(expr2, expr1, int, real, ''float of non-int'')

expr1 ⟶ trunc(expr2)
    ▷ expr2.symtab := expr1.symtab
    ▷ convert_type(expr2, expr1, real, int, ''trunc of non-real'')
```

Figure 4.23 *(continued on next page)*

One could convert our attribute grammar into executable code using an automatic attribute evaluator generator. Alternatively, one could create an ad hoc evaluator in the form of mutually recursive subroutines (Exercise C-4.35). In the latter case attribute flow would be explicit in the calling sequence of the routines. We could then choose if desired to keep the symbol table in global variables, rather than passing it from node to node through attributes. Most compilers employ the ad hoc approach.

CHECK YOUR UNDERSTANDING

45. What patterns of attribute flow can be captured easily with action routines?
46. Some compilers perform all semantic checks and intermediate code generation in action routines. Others use action routines to build a syntax tree and then perform semantic checks and intermediate code generation in separate traversals of the syntax tree. Discuss the tradeoffs between these two strategies.
47. What sort of information do action routines typically keep in global variables, rather than in attributes?
48. How can a semantic analyzer avoid the generation of cascading error messages?

```
macro declare_name(name, cur_stmt, next_stmt : syntax_tree_node; t : type)
    if ⟨name, ?⟩ ∈ cur_stmt.symtab
        next_stmt.errors_in := cur_stmt.errors_in + [''redefinition of'' name ''at'' cur_stmt.location]
        next_stmt.symtab := cur_stmt.symtab − ⟨name, ?⟩ + ⟨name, error⟩
    else
        next_stmt.errors_in := cur_stmt.errors_in
        next_stmt.symtab := cur_stmt.symtab + ⟨name, t⟩
macro check_types(result, operand1, op, operand2)
    if operand1.type = error or operand2.type = error
        result.type := error
        result.errors := operand1.errors + operand2.errors
    else if operand1.type ≠ operand2.type
        result.type := error
        result.errors := operand1.errors + operand2.errors + [''type clash at'' op.location]
    else
        result.type := operand1.type
        result.errors := operand1.errors + operand2.errors
macro convert_type(old_expr, new_expr : syntax_tree_node; from_t, to_t : type; msg : string)
    if old_expr.type = from_t or old_expr.type = error
        new_expr.errors := old_expr.errors
        new_expr.type := to_t
    else
        new_expr.errors := old_expr.errors + [msg ''at'' old_expr.location]
        new_expr.type := error
```

Figure 4.23 *(continued)*

4.6.4 Space Management for Attributes

Any attribute evaluation method requires space to hold the attributes of the grammar symbols. In an attribute grammar based on the abstract grammar of explicit syntax trees, the obvious approach is to store attributes in the nodes of the tree themselves. In a context-free grammar with action routines, the analogous approach applies only if we are building an explicit parse tree—and usually we're not. This means we need to find a way to keep track of the attributes of symbols we have seen (or predicted) but not yet finished parsing. The details differ in bottom-up and top-down parsers.

For a bottom-up parser with an S-attributed grammar, it is straightforward to maintain an *attribute stack* that directly mirrors the parse stack: next to every state number on the parse stack is an attribute record for the symbol we shifted when we entered that state. Entries in the attribute stack are pushed and popped automatically by the parser driver; space management is not an issue for the writer of action routines. Complications arise if we try to achieve the effect of inherited attributes, but these can be accommodated within the basic attribute-stack framework.

For a top-down parser with an L-attributed grammar, we have two principal options. The first option is automatic, but more complex than for bottom-up

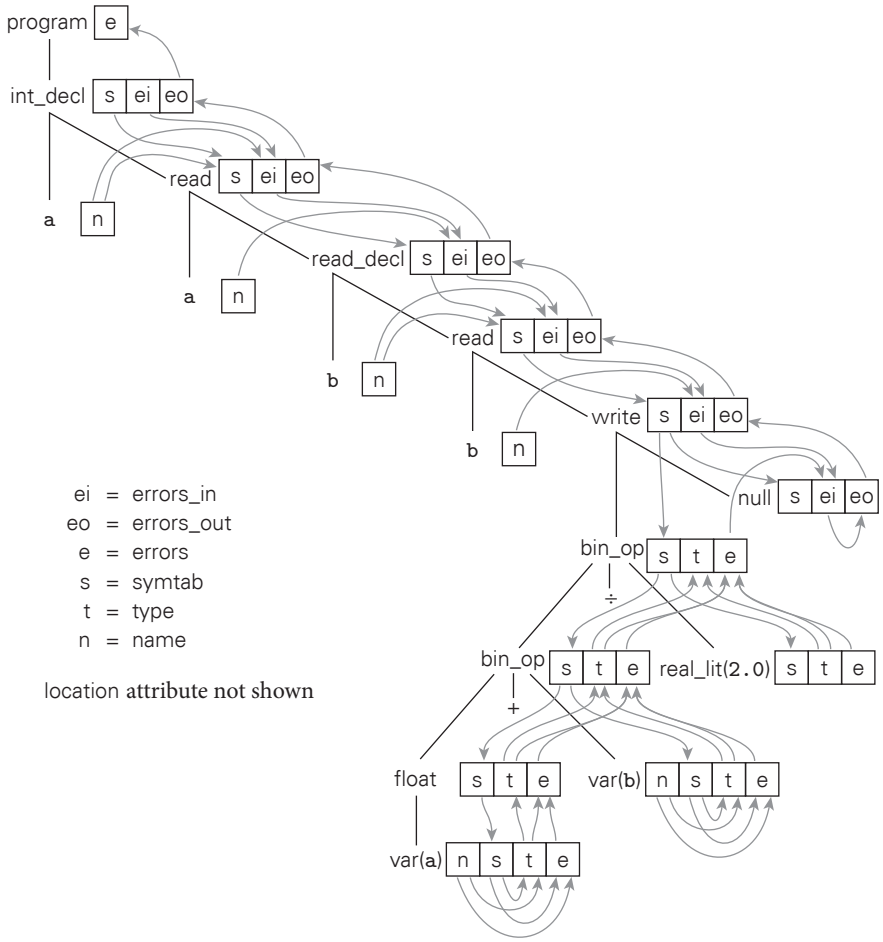


Figure 4.24 Decoration of the syntax tree of Figure 4.2, using the grammar of Figure C-4.23. Location information, which we assume has been initialized in every node by the parser, contributes to error messages, but does not otherwise propagate through the tree.

grammars. It still uses an attribute stack, but one that does not mirror the parse stack, because it must store information about symbols that have already been parsed. The second option has lower space overhead, and saves time by "short-cutting" copy rules, but requires action routines to allocate and deallocate space for attributes explicitly.

In both bottom-up and top-down parsers, it is common for some of the contextual information for action routines to be kept in global variables. The symbol table in particular is usually global. Rather than pass its full contents through attributes from one production to the next, we pass an indication of the currently active

1. (
2. (1
3. (F_1
4. (T_1
5. (E_1
6. (E_1 +
7. (E_1 + 3
8. (E_1 + F_3
9. (E_1 + T_3
10. (E_4
11. (E_4)
12. F_4
13. T_4
14. T_4 *
15. T_4 * 2
16. T_4 * F_2
17. T_8
18. E_8

Figure 4.25 Parse/attribute stack trace for `(1 + 3) * 2`, using the grammar of Figure C-4.16. Subscripts represent val attributes; they are not meant to distinguish among instances of a symbol.

scope. Lookups in the global table then use this scope information to obtain the right referencing environment.

In this subsection, we consider attribute space management in more detail. Using bottom-up and top-down grammars for arithmetic expressions, we illustrate automatic management for both bottom-up and top-down parsers, as well as the ad hoc option for top-down parsers.

Bottom-Up Evaluation

EXAMPLE 4.35

Stack trace for bottom-up parse, with action routines

Figure C-4.25 shows a trace of the parse and attribute stack for `(1 + 3) * 2`, using the attribute grammar of Figure C-4.16. For the sake of clarity, we show a single, combined stack for the parser and attribute evaluator, and we omit the CFSM state numbers.

It is easy to evaluate the attributes of symbols in this grammar, because the grammar is S-attributed. In an automatically generated parser, such as those produced by `yacc/bison`, the attribute rules associated with the productions of the grammar in Figure C-4.16 would constitute action routines, to be executed when their productions are recognized. For `yacc/bison`, they would be written in C, with "pseudostructs" to name the attribute records of the symbols in each production. Attributes of the left-hand side symbol would be accessed as fields of the pseudostruct `$$`. Attributes of right-hand side symbols would be accessed as fields of the pseudostructs `$1`, `$2`, etc. To get from line 9 to line 10, for example, in the trace of Figure C-4.25, we would use an action routine version of the first rule of the grammar in Figure C-4.16: $$.val = $1.val + $3.val. ■

When a bottom-up action routine is executed, the attribute records for symbols on the right-hand side of the production can be found in the top few entries of the attribute stack. The attribute record for the symbol on the left-hand side of the production (i.e., `$$`) will not yet lie in the stack: it is the task of the action routine to initialize this record. After the action routine completes, the parser pops the right-hand side records off the attribute stack and replaces them with `$$`. In `yacc/bison`, if no action routine is specified for a given production, the default action is to "copy" `$1` into `$$`. Since `$$` will occupy the same location, once pushed, that `$1` occupied before being popped, this "copy" can be effected without doing any work.

EXAMPLE 4.36
Finding inherited attributes in "buried" records

Inherited Attributes. Unfortunately, it is not always easy to write an S-attributed grammar. A simple example in which inherited attributes are desirable arises in C or Fortran-style variable declarations, in which a type name precedes the list of variable names:

dec $\longrightarrow$ *type id_list*
id_list $\longrightarrow$ `id`
id_list $\longrightarrow$ *id_list* `,` `id`

Let us assume that *type* has a synthesized attribute tp that contains a pointer to the symbol table entry for the type in question. Ideally, we should like to pass this attribute into *id_list* as an inherited attribute, so that we may enter each newly declared identifier into the symbol table, complete with type indication, as it is encountered. When we recognize the production *id_list* $\longrightarrow$ `id`, we know that the top record on the attribute stack will be the one for `id`. But we know more than this: the next record down must be the one for *type*. To find the type of the new entry to be placed in the symbol table, we may safely inspect this "buried" record. Though it does not belong to a symbol of the current production, we can count on its presence because there is no other way to reach the *id_list* $\longrightarrow$ `id` production.

Now what about the `id` in *id_list* $\longrightarrow$ *id_list* `,` `id`? This time the top three records on the attribute stack will be for the right-hand symbols `id`, `,`, and *id_list*. Immediately below them, however, we can still count on finding the entry for *type*, waiting for the *id_list* to be completed so that *dec* can be recognized. Using nonpositive indices for pseudostructs below the current production, we can write action routines as follows:

dec $\longrightarrow$ *type id_list*
id_list $\longrightarrow$ `id` { declare_id ($1.name, $0.tp) }
id_list $\longrightarrow$ *id_list* `,` `id` { declare_id ($3.name, $0.tp) }

Records deeper in the attribute stack could be accessed as $–1, $–2, and so on. While *id_list* appears in two places in this grammar fragment, both occurrences are guaranteed to lie above a *type* record in the attribute stack, the first because it lies next to *type* in a right-hand side, and the second by induction, because it is the beginning of the yield of the first. ■

Unfortunately, there are grammars in which a symbol that needs inherited attributes occurs in productions in which the underlying symbols are not the same. We can still handle inherited attributes in such cases, but only by modifying the underlying context-free grammar. An example can be found in languages like Perl, in which the meaning of an expression (and of the identifiers and operators within it) depends on the *context* in which that expression appears. Some Perl contexts expect arrays. Others expect numbers, strings, or Booleans. To correctly analyze an expression, we must pass the expectations of the context into the expression subtree as inherited attributes. Here is a grammar fragment that captures the problem:

EXAMPLE 4.37
Grammar fragment requiring context

```
stmt → id := expr
     → ...
     → if expr then stmt
expr → ...
```

Within the production for *expr*, the parser doesn't know whether the surrounding context is an assignment or the condition of an `if` statement. If it is a condition, then the expected type of the expression is Boolean. If it is an assignment, then the expected type is that of the identifier on the assignment's left-hand side. This identifier can be found two records below the current production in the attribute stack.

Semantic Hooks. To allow these cases to be treated uniformly, we can add *semantic hook*, or "marker" symbols to the grammar. Semantic hooks generate ε, and thus do not alter the language defined by the grammar; their only purpose is to hold inherited attributes:

EXAMPLE 4.38
Semantic hooks for context

```
stmt → id := A expr
     → ...
     → if B expr then stmt
A → ε { $$.tp := $-1.tp }
B → ε { $$.tp := Boolean }
expr → ... { if $0.tp = Boolean then ... }
```

Since the epsilon production for a semantic hook can provide an action routine, it is tempting to think of semantic hooks as a general technique to insert action routines in the middle of bottom-up productions. Unfortunately this is not the case: semantic hooks can be used only in places where the parser can be sure that it is in a given production. Placing a semantic hook anywhere else will break the "LR-ness" of the grammar, causing the parser generator to reject the modified grammar. Consider the following example:

EXAMPLE 4.39
Semantic hooks that break an LR CFG

```
1. stmt → l_val := expr
2.      → id args
3. l_val → id quals
```

4. *quals* ⟶ *quals* `.` `id`
5. ⟶ *quals* `(` *expr_list* `)`
6. ⟶ ε
7. *args* ⟶ `(` *expr_list* `)`
8. ⟶ ε

An *l-value* in this grammar is a "qualified" identifier: an identifier followed by optional array subscript and record field qualifiers.[2] We have assumed that the language follows the notation of Fortran and Ada, in which parentheses delimit both procedure call arguments and array subscripts. In the case of procedure calls, it would be natural to want an action routine to pass the symbol-table index of the subroutine into the argument list as an inherited attribute, so that it can be used to check the number and types of arguments:

stmt ⟶ `id` *A* *args*
A ⟶ ε { $$.proc_index := lookup ($0.name) }

If we try this, however, we will run into trouble, because the procedure call

```
foo(1, 2, 3);
```

and the array element assignment

```
foo(1, 2, 3) := 4;
```

begin with the same sequence of tokens. Until it sees the token after the closing parenthesis, the parser cannot tell whether it is working on production 1 or production 2. The presence of *A* in production 2 will therefore lead to a shift-reduce conflict; after seeing an `id`, the parser will not know whether to recognize *A* or shift `(`. ■

Left Corners. In general, the right-hand side of a production in a context-free grammar is said to consist of the *left corner* and the *trailing part*. In the left corner we cannot be sure which production we are parsing; in the trailing part the production is uniquely determined. In an LL(1) grammar, the left corner is always empty. In an LR(1) grammar, it can consist of up to the entire right-hand side. Semantic hooks can safely be inserted in the trailing part of a production, but not in the left corner. `Yacc/bison` recognizes this fact explicitly by allowing action routines to be embedded in right-hand sides. It automatically converts the production

EXAMPLE 4.40
Action routines in the trailing part

2 In general, an l-value in a programming language is anything to which a value can be assigned (i.e., anything that can appear on the left-hand side of an assignment). From a low-level point of view, this is basically an address. An r-value is anything that can appear on the right-hand side of an assignment. From a low-level point of view, this is a value that can be stored at an address. We will discuss l-values and r-values further in Section 6.1.2.

$$S \longrightarrow \alpha \ \{ \text{ your code here } \} \ \beta$$

to

$$S \longrightarrow \alpha \ A \ \beta$$
$$A \longrightarrow \varepsilon \ \{ \text{ your code here } \}$$

for some new, distinct symbol A. If the action routine is not in the trailing part, the resulting grammar will not be LALR(1), and `yacc`/`bison` will produce an error message.

EXAMPLE 4.41

Left factoring in lieu of semantic hooks

In our procedure call and array subscript example, we cannot place a semantic hook before the *args* of production 2 because this location is in the left corner. If we wish to look up a procedure name in the symbol table before we parse the arguments, we will need to combine the productions for statements that can begin with an identifier, in a manner reminiscent of the *left factoring* discussed in Section 2.3.2:

$$\begin{aligned}
stmt &\longrightarrow \texttt{id}\ A\ quals\ assign_opt \\
A &\longrightarrow \varepsilon \ \{ \ \$\$.\text{id_index} := \text{lookup } (\$0.\text{name}) \ \} \\
quals &\longrightarrow quals\ \texttt{.}\ \texttt{id} \\
&\longrightarrow quals\ \texttt{(}\ expr_list\ \texttt{)} \\
&\longrightarrow \varepsilon \\
assign_opt &\longrightarrow \texttt{:=}\ expr \\
&\longrightarrow \varepsilon
\end{aligned}$$

This change eliminates the shift-reduce conflict, but at the expense of combining the entire grammar subtrees for procedure call arguments and array subscripts. To use the modified grammar we shall have to write action routines for *quals* that work for both kinds of constructs, and this can be a major nuisance. Users of LR-family parser generators often find that there is a tension between the desire for grammar clarity and parsability on the one hand and the need for semantic hooks to set inherited attributes on the other.

Top-Down Evaluation

Top-down parsers, as discussed in Chapter 2, come in two principal varieties: recursive descent and table driven. Attribute management in recursive descent parsers is almost trivial: inherited attributes of symbol *foo* take the form of parameters passed into the parsing routine named `foo`; synthesized attributes are the return parameters. These synthesized attributes can then be passed as inherited attributes to symbols later in the current production, or returned as synthesized attributes of the current left-hand side.

Attribute space management for automatically generated top-down parsers is somewhat more complex. Because they allow action routines at arbitrary locations in a right-hand side, top-down parsers avoid the need to modify the grammar in order to insert semantic hooks. (Of course, because they must have empty left corners, top-down grammars can be harder to write in the first place.) Because the parse stack describes the future, instead of the past, we cannot employ an attribute

$E \longrightarrow T$ { TT.st := T.val }^{1} TT { E.val := TT.val }^{2}
$TT_1 \longrightarrow$ `+` T { TT_2.st := TT_1.st + T.val }^{3} TT_2 { TT_1.val := TT_2.val }^{4}
$TT_1 \longrightarrow$ `-` T { TT_2.st := TT_1.st − T.val }^{5} TT_2 { TT_1.val := TT_2.val }^{6}
$TT \longrightarrow \varepsilon$ { TT.val := TT.st }^{7}
$T \longrightarrow F$ { FT.st := F.val }^{8} FT { T.val := FT.val }^{9}
$FT_1 \longrightarrow$ `*` F { FT_2.st := FT_1.st × F.val }^{10} FT_2 { FT_1.val := FT_2.val }^{11}
$FT_1 \longrightarrow$ `/` F { FT_2.st := FT_1.st ÷ F.val }^{12} FT_2 { FT_1.val := FT_2.val }^{13}
$FT \longrightarrow \varepsilon$ { FT.val := FT.st }^{14}
$F_1 \longrightarrow$ `-` F_2 { F_1.val := − F_2.val }^{15}
$F \longrightarrow$ `(` E `)` { F.val := E.val }^{16}
$F \longrightarrow$ `const` { F.val := C.val }^{17}

Figure 4.26 LL(1) grammar for constant expressions, with action routines. The boldface superscripts are for reference in Figure C-4.27.

stack that simply mirrors the parse stack. Our two principal options are to equip the parser with a (more complicated) algorithm for automatic space management, or to require action routines to manage space explicitly.

Automatic Management. Automatic management of attribute space for top-down parsing is more complicated than it is for bottom-up parsing. It is also more space intensive. We can still use an attribute stack, but it has to contain all of the symbols in all of the productions between the root of the (hypothetical) parse tree and the current point in the parse. All of the right-hand side symbols of a given production are adjacent in the stack; the left-hand side is buried in the right-hand side of a deeper (closer to the root) production.

EXAMPLE 4.42
Operation of an LL attribute stack

Figure C-4.26 contains an LL(1) grammar for constant expressions, with action routines. Figure C-4.27 uses this grammar to trace the operation of a top-down attribute stack on the sample input `(1 + 3) * 2`. The left-hand column shows the parse stack. The right-hand column shows the attribute stack. Three global pointers index into the attribute stack. One (shown as an "arrow-boxed" `L` in the trace) identifies the record in the attribute stack that holds the attributes of the left-hand side symbol of the current production. The second (shown as an arrow-boxed `R` in the trace) identifies the first symbol on the right-hand side of the production. `L` and `R` allow the action routines to find the attributes of the symbols of the current production. The third pointer (shown as an arrow-boxed `N` in the trace) identifies the first symbol within the right-hand side that has not yet been completely parsed. It allows the parser to update `L` correctly when a production is predicted.

At any given time, the attribute stack contains all symbols of all productions on the path between the root of the parse tree and the symbol currently at the top of the parse stack. Figure C-4.28 identifies these symbols graphically at the point in Figure C-4.27 immediately above the eight elided lines. Symbols to the left in the parse tree have already been reclaimed; those to the right have yet to be allocated.

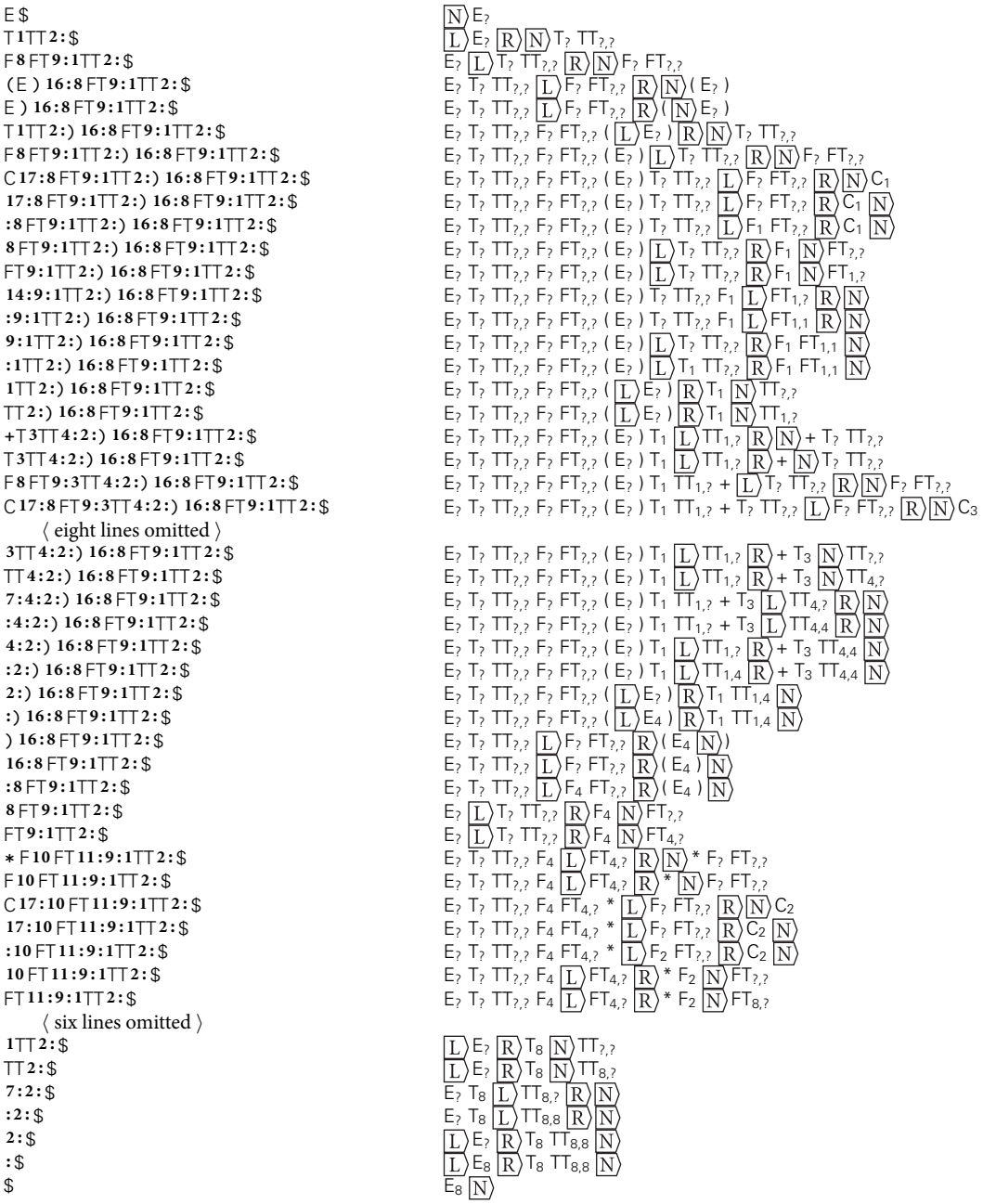


Figure 4.27 **Trace of the parse stack (left) and attribute stack (right) for `(1 + 3) * 2`, using the grammar (and action routine numbers) of Figure C-4.26.** Subscripts in the attribute stack indicate the values of attributes. For symbols with two attributes, st comes first.

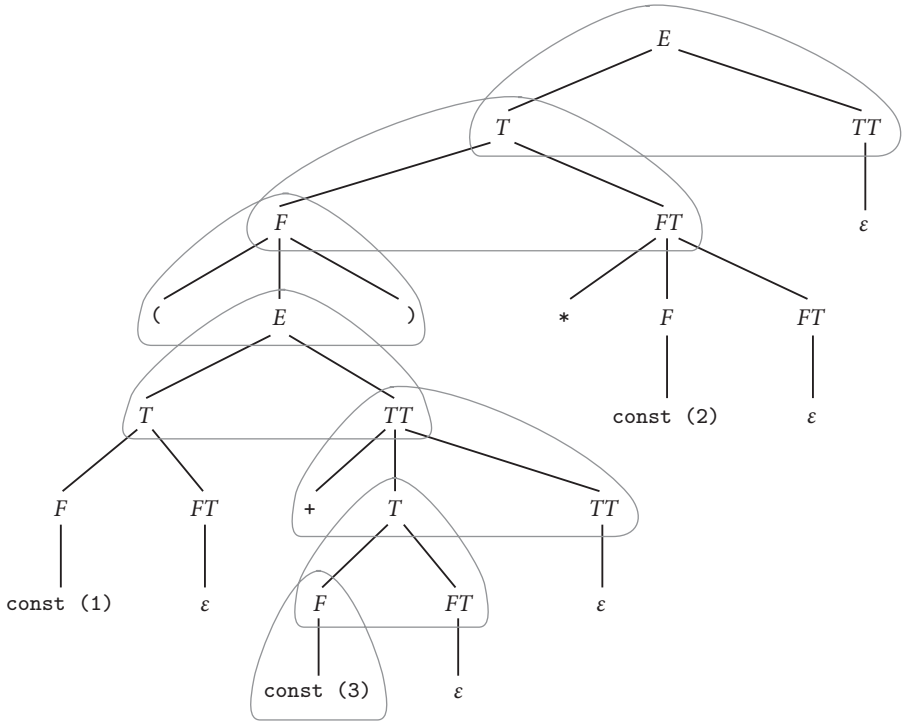


Figure 4.28 Productions with symbols currently in the attribute stack during a parse of `(1 + 3) * 2` (using the grammar of Figure C-4.26), at the point where we are about to parse the `3`. In Figure C-4.27 this point corresponds to the line immediately above the eight elided lines.

At start-up, the attribute stack contains a record for the start symbol, pointed at by `N`. When we push the right-hand side of a predicted production onto the parse stack, we add an "end-of-production" marker, represented by a colon in the trace. At the same time, we push records for the right-hand-side symbols onto the attribute stack. (These are *added* to the attribute stack; they do not replace the left-hand side.) Prior to pushing these entries, we save the current `L` and `R` pointers in another stack (not shown). We then set `L` to the old `N`, and make `R` and `N` point to the newly pushed right-hand side.

When we see an action symbol at the top of the parse stack (shown in the trace as a small bold number), we pop it and execute the corresponding action routine. When we match a terminal at the top of the parse stack, we pop it and move `N` forward one record in the attribute stack. When we see an end-of-production marker at the top of the parse stack, we pop it, set `N` to the attribute record following the one currently pointed at by `L`, pop everything from `R` forward off of the attribute stack, and restore the most recently saved values of `L` and `R`.

$E \longrightarrow T\ TT$
$TT \longrightarrow$ `+` T { bin_op (''+'') } TT
$TT \longrightarrow$ `-` T { bin_op (''−'') } TT
$TT \longrightarrow \varepsilon$
$T \longrightarrow F\ FT$
$FT \longrightarrow$ `*` F { bin_op (''×'') } FT
$FT \longrightarrow$ `/` F { bin_op (''÷'') } FT
$FT \longrightarrow \varepsilon$
$F \longrightarrow$ `-` F { un_op (''+/−'') }
$F \longrightarrow$ `(` E `)`
$F \longrightarrow$ `const` { push_leaf (cur_tok.val) }

Figure 4.29 Ad hoc management of attribute space in an LL(1) grammar to build a syntax tree.

It should be emphasized that while the trace is long and tedious, its complexity is completely hidden from the writer of action routines. Once the space management routines are integrated with the driver for a top-down parser generator, all the compiler writer sees is the grammar of Figure C-4.26. When the compiler writer refers to attributes of the symbol on the left-hand side of a production, the parser generator will access entry `L` in the attribute stack; when the compiler writer refers to attributes of the kth symbol on the right-hand side, the parser generator will access entry `R`$-k-1$. In comparing Figures C-4.25 and C-4.27, one should also note that reduction and execution of a production's action routine are shown as a single step in the LR trace; they are shown separately in the LL trace, making that trace appear more complex than it really is.

Ad Hoc Management. One drawback of automatic space management for top-down grammars is the frequency with which the compiler writer must specify copy routines. Of the 17 action routines in Figure C-4.26, 12 simply move information from one place to another. The time required to execute these routines can be minimized by copying pointers, rather than large records, but compiler writers may still consider the copies a nuisance.

EXAMPLE 4.43
Ad hoc management of a semantic stack

An alternative is to manage space explicitly within the action routines, pushing and popping an ad hoc *semantic stack* only when information is generated or consumed. Using this technique, we can replace the action routines of Figure C-4.26 with the simpler version shown in Figure C-4.29. Variable cur_tok is assumed to contain the synthesized attributes of the most recently matched token. The semantic stack contains pointers to syntax tree nodes. The push_leaf routine creates a node for a specified constant and pushes a pointer to it onto the semantic stack. The un_op routine pops the top pointer off the stack, makes it the child of a newly created node for the specified unary operator, and pushes a pointer to that node back on the stack. The bin_op routine pops the top *two* pointers off the semantic stack and pushes a pointer to a newly created node for the specified binary operator.

When the parse of E is completed, a pointer to a syntax tree describing its yield will be found in the top-most record on the semantic stack. ■

The advantage of ad hoc space management is clearly the smaller number of rules and the elimination of the inherited attributes used to represent left context. The disadvantage is that the compiler writer must be aware of what is in the semantic stack at all times, and must remember to push and pop it when appropriate.

One further advantage of an ad hoc semantic stack is that it allows action routines to push or pop an arbitrary number of records. With automatic space management, the number of records that can be seen by any one routine is limited by the number of symbols in the current production. The difference is particularly important in the case of productions that generate lists. In Example C-4.36 we saw an SLR(1) grammar for declarations in the style of C and Fortran, in which the type name precedes the list of identifiers. Here is an LL(1) grammar fragment for a language in the style of Pascal and Ada, in which the variables precede the type:

EXAMPLE 4.44
Processing lists with an attribute stack

```
dec → id_list : type
id_list → id id_list_tail
id_list_tail → , id_list
             → ε
```

Without resorting to non-L-attributed flow (see Exercise C-4.41), we cannot pass the declared type into *id_list* as an inherited attribute. Instead, we must save up the list of identifiers and enter them into the symbol table *en masse* when the type is finally encountered. With automatic management of space for attributes, the action routines would look something like this:

```
dec → id_list : type { declare_vars(id_list.chain, type.tp) }
id_list → id id_list_tail { id_list.chain := append(id.name, id_list_tail.chain) }
id_list_tail → , id_list { id_list_tail.chain := id_list.chain }
             → ε { id_list_tail.chain := null }
```

■

EXAMPLE 4.45
Processing lists with a semantic stack

With ad hoc management of space, we can get by without the linked list:

```
dec → { push(marker) }
      id_list : type
      { pop(tp)
        pop(name)
        while name ≠ marker
            declare_var(name, tp)
            pop(name) }
id_list → id { push(cur_tok.name) } id_list_tail
id_list_tail → , id_list
             → ε
```

■

Neither automatic nor ad hoc management of attribute space in top-down parsers is clearly superior to the other. The ad hoc approach eliminates the need

for many copy rules and inherited attributes, and is consequently somewhat more time and space efficient. It also allows lists to be embedded in the semantic stack. On the other hand, it requires that the programmer who writes the action routines be continually aware of what is in the stack and why, in order to push and pop it appropriately. In the final analysis, the choice is an engineering tradeoff driven by the particular needs of the project.

CHECK YOUR UNDERSTANDING

49. Explain how to manage space for synthesized attributes in a bottom-up parser.
50. Explain how to manage space for inherited attributes in a bottom-up parser.
51. Define *left corner* and *trailing part*.
52. Under what circumstances can an action routine be embedded in the right-hand side of a production in a bottom-up parser? Equivalently, under what circumstances can a marker symbol be embedded in a right-hand side without rendering the grammar non-LR?
53. Summarize the tradeoffs between automatic and ad hoc management of space for attributes in a top-down parser.
54. At any given point in a top-down parse, which symbols will have attribute records in an automatically managed attribute stack?

4 Program Semantics

4.8 Exercises

4.22 Basic results from automata theory tell us that the language $L = \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n = \{\varepsilon, \mathtt{abc}, \mathtt{aabbcc}, \mathtt{aaabbbccc}, \ldots\}$ is not context free. It can be captured, however, using an attribute grammar. Give an underlying CFG and a set of attribute rules that associates a Boolean attribute ok with the root R of each parse tree, such that R.ok = true if and only if the string corresponding to the fringe of the tree is in L.

4.23 Write an S-attributed attribute grammar, based on the CFG of Example C-4.28, that accumulates the value of the overall expression into the root of the tree. You will need to use dynamic memory allocation so that individual attributes can hold an arbitrary amount of information.

4.24 Suppose that we want to translate constant expressions into the postfix, or "reverse Polish" notation of logician Jan Łukasiewicz. Postfix notation does not require parentheses. It appears in stack-based languages such as Postscript, Forth, and the P-code and Java bytecode intermediate forms mentioned in Section 1.4. It also served, historically, as the input language of certain hand-held calculators made by Hewlett-Packard. When given a number, a postfix calculator would push the number onto an internal stack. When given an operator, it would pop the top two numbers from the stack, apply the operator, and push the result. The display would show the value at the top of the stack. To compute $2 \times (15 - 3)/4$, for example, one would push `2 E 1 5 E 3 E - * 4 E /` (here E is the "enter" key, used to end the string of digits that constitute a number).

Using the underlying CFG of Figure C-4.16, write an attribute grammar that will associate with the root of the parse tree a sequence of postfix calculator button pushes, seq, that will compute the arithmetic value of the tokens derived from that symbol. You may assume the existence of a function buttons(c) that returns a sequence of button pushes (ending with E on a

postfix calculator) for the constant `c`. You may also assume the existence of a concatenation function for sequences of button pushes.

4.25 Repeat the previous exercise using the underlying CFG of Figure C-4.18.

4.26 Consider the following grammar for reverse Polish arithmetic expressions:

E ⟶ *E E op* | `id`
op ⟶ `+` | `-` | `*` | `/`

Assuming that each `id` has a synthesized attribute name of type string, and that each *E* and *op* has an attribute val of type string, write an attribute grammar that arranges for the val attribute of the root of the parse tree to contain a translation of the expression into conventional infix notation. For example, if the leaves of the tree, left to right, were "`A A B - * C /`", then the val field of the root would be "`( ( A * ( A - B ) ) / C )`". As an extra challenge, write a version of your attribute grammar that exploits the usual arithmetic precedence and associativity rules to use as few parentheses as possible.

4.27 To reduce the likelihood of typographic errors, the digits comprising most credit card numbers are designed to satisfy the so-called *Luhn formula*, standardized by ANSI in the 1960s, and named for IBM mathematician Hans Peter Luhn. Starting at the right, we double every other digit (the second-to-last, fourth-to-last, etc.). If the doubled value is 10 or more, we add the resulting digits. We then sum together all the digits. In any valid number the result will be a multiple of 10. For example, 1234 5678 9012 3456 becomes 2264 1658 9022 6416, which sums to 64, so this is not a valid number. If the last digit had been 2, however, the sum would have been 60, so the number would potentially be valid.

Give an attribute grammar for strings of digits that accumulates into the root of the parse tree a Boolean value indicating whether the string is valid according to Luhn's formula. Your grammar should accommodate strings of arbitrary length.

4.28 Consider the following CFG for floating-point constants, without exponential notation. (Note that this exercise is somewhat artificial: the language in question is regular, and would be handled by the scanner of a typical compiler.)

C ⟶ *digits* . *digits*
digits ⟶ *digit more_digits*
more_digits ⟶ *digits* | ε
digit ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Augment this grammar with attribute rules that will accumulate the value of the constant into a val attribute of the root of the parse tree. Your answer should be S-attributed.

4.29 One potential criticism of the obvious solution to the previous problem is that the values in internal nodes of the parse tree do not reflect the value, in context, of the fringe below them. Create an alternative solution that addresses this criticism. More specifically, create your grammar in such a way that the val of an internal node is the sum of the `val`s of its children. Illustrate your solution by drawing the parse tree and attribute flow for `12.34`. (Hint: You will probably want a different underlying CFG, and non-L-attributed flow.)

4.30 Consider the following attribute grammar for variable declarations, based on the CFG of Exercise 2.11:

decl ⟶ `ID` *decl_tail*
- ▷ decl.t := decl_tail.t
- ▷ decl_tail.in_tab := insert (decl.in_tab, ID.n, decl_tail.t)
- ▷ decl.out_tab := decl_tail.out_tab

decl_tail ⟶ `,` *decl*
- ▷ decl_tail.t := decl.t
- ▷ decl.in_tab := decl_tail.in_tab
- ▷ decl_tail.out_tab := decl.out_tab

decl_tail ⟶ `: ID ;`
- ▷ decl_tail.t := ID.n
- ▷ decl_tail.out_tab := decl_tail.in_tab

Show a parse tree for the string `A, B : C;`. Then, using arrows and textual description, specify the attribute flow required to fully decorate the tree. (Hint: Note that the grammar is *not* L-attributed.)

4.31 A CFG-based attribute evaluator capable of handling non-L-attributed attribute flow needs to take a parse tree as input. Explain how to build a parse tree automatically during a top-down or bottom-up parse (i.e., without explicit action routines).

4.32 Write an LL(1) grammar with action routines and automatic attribute space management that generates the reverse Polish translation described in Exercise C-4.24.

4.33 (a) Write a context-free grammar for `case` or `switch` statements in the style of Pascal or C. Add semantic functions to ensure that the same label does not appear on two different arms of the construct.

(b) Replace your semantic functions with action routines that can be evaluated during parsing.

4.34 Write an algorithm to determine whether the rules of an arbitrary attribute grammar are noncircular. (Your algorithm will require exponential time in the worst case [JOR75].)

4.35 Rewrite the attribute grammar of Figure C-4.23 in the form of an ad hoc tree traversal consisting of mutually recursive subroutines in your favorite programming language. Keep the symbol table in a global variable, rather than passing it through arguments.

4.36 Augment the attribute grammar of Figure C-4.20, Figure C-4.21 to initialize a synthesized attribute in every syntax tree node that indicates the location (line and column) at which the corresponding construct appears in the source program. You may assume that the scanner initializes the location of every token.

4.37 Modify the CFG and attribute grammar of Figures 4.1 and C-4.23 to permit mixed integer and real expressions, without the need for `float` and `trunc`. You will want to add an annotation to any node that must be coerced to the opposite type, so that the code generator will know to generate code to do so. Be sure to think carefully about your coercion rules. In the expression `my_int + my_real`, for example, how will you know whether to coerce the integer to be a real, or to coerce the real to be an integer?

4.38 A potential objection to the abstract attribute grammar of Example C-4.33 is that it repeatedly copies the entire symbol table from one node to another. In this particular tiny language, it is easy to see that the referencing environment never shrinks: the symbol table changes only with the addition of new identifiers. Exploiting this observation, show how to modify the pseudocode of Figure C-4.23 so that it copies only pointers, rather than the entire symbol table.

4.39 Your solution to the previous exercise probably doesn't generalize to languages with nontrivial scoping rules. Explain how an AG such as that in Figure C-4.23 might be modified to use a global symbol table similar to the one described in Section C-3.4.1. Among other things, you should consider nested scopes, the hiding of names in outer scopes, and the requirement (not enforced by the table of Section C-3.4.1) that variables be declared before they are used.

4.40 Repeat Exercise C-4.32 using ad hoc attribute space management. Instead of accumulating the translation into a data structure, write it to a file on the fly.

4.41 Rewrite the grammar for declarations of Example C-4.44 without the requirement that your attribute flow be L-attributed. Try to make the grammar as simple and elegant as possible (you shouldn't need to accumulate lists of identifiers).

4.42 Fill in the missing lines in Figure C-4.27.

4.43 Consider the following grammar with action routines:

params ⟶ *mode* `ID` *par_tail*
{ params.list := insert(⟨mode.val, ID.name⟩, par_tail.list) }

par_tail ⟶ `,` *params* { par_tail.list := params.list }

⟶ { par_tail.list := null }

mode ⟶ `IN` { mode.val := IN }

⟶ `OUT` { mode.val := OUT }

⟶ `IN OUT` { mode.val := IN_OUT }

Suppose we are parsing the input `IN a, OUT b`, and that our compiler uses an automatically maintained attribute stack to hold the active slice of the parse tree. Show the contents of this attribute stack immediately before the parser predicts the production *par_tail* $\longrightarrow \varepsilon$. Be sure to indicate where $\boxed{\text{L}}\rangle$ and $\boxed{\text{R}}\rangle$ point in the attribute stack. Also show the stack of saved $\boxed{\text{L}}\rangle$ and $\boxed{\text{R}}\rangle$ values, showing where each points in the attribute stack. You may ignore the $\boxed{\text{N}}\rangle$ pointer.

4.44 One problem with automatic space management for attributes in a top-down parser occurs in lists and sequences. Consider for example the following grammar:

block $\longrightarrow$ `begin` *stmt_list* `end`
stmt_list $\longrightarrow$ *stmt stmt_list_tail*
stmt_list_tail $\longrightarrow$ `;` *stmt_list* | ε
stmt $\longrightarrow$ ...

After predicting the final statement of an n-statement block, the attribute stack will contain the following (line breaks and indentation are for clarity only):

block begin stmt_list end
    stmt stmt_list_tail ; stmt_list
    stmt stmt_list_tail ; stmt_list
    stmt stmt_list_tail ; stmt_list
    { n times }

If the attribute stack is of finite size, it is guaranteed to overflow for some long but valid block of straight-line code. The problem is especially unfortunate since, with the exception of the accumulated output code, none of the repeated symbols in the attribute stack contains any useful attributes once its substructure has been parsed.

Suggest a technique to "squeeze out" useless symbols in the attribute stack, dynamically. Ideally, your technique should be amenable to automatic implementation, so it does not constitute a burden on the compiler writer.

Also, suppose you are using a compiler with a top-down parser that employs an automatically managed attribute stack, but does not squeeze out useless symbols. What could you do if your program caused the compiler to run out of stack space? How could you modify your program to "get around" the problem?

4 Program Semantics

4.9 Explorations

4.50 One of the most influential applications of attribute grammars was the Cornell Synthesizer Generator [Rep84, RT88]. Learn how the Generator used attribute grammars not only for incremental update of semantic information in a program under edit, but also for automatic creation of language based editors from formal language specifications. How general is this technique? What applications might it have beyond syntax-directed editing of computer programs?

4.51 The attribute grammars used in this chapter are all quite simple. Most are S- or L-attributed. All are noncircular. Are there any practical uses for more complex attribute grammars? How about automatic attribute evaluators? Using the Bibliographic Notes as a starting point, conduct a survey of attribute evaluation techniques. Where is the line between practical techniques and intellectual curiosities?

4.52 As described in Section C-4.6.4, `yacc/bison` will refuse to accept action routines in the left corner of a production. Is there any way around this problem? Can you imagine implementing an extended version of the tool that would permit action routines in arbitrary locations? What would be the challenges? The cost?

4.53 Learn how attribute space is managed in the ANTLR parser generator. How does it compare to the techniques described in Section C-4.6.4 for top-down parsing?