Chapter 4

Context-Sensitive Analysis

CHAPTER OVERVIEW

An input program that is grammatically correct may still contain serious errors that would prevent compilation. To detect such errors, a compiler performs a further level of checking that involves considering each statement in its actual context. These checks find errors of type and of agreement.

This chapter introduces two techniques for context-sensitive checking. Attribute grammars are a functional formalism for specifying context-sensitive computation. Ad hoc syntax-directed translation provides a simple framework where the compiler writer can hang arbitrary code snippets to perform these checks.

Keywords: Semantic Elaboration, Type Checking, Attribute Grammars, Ad Hoc Syntax Directed Translation

4.1 INTRODUCTION

The compiler's ultimate task is to translate the input program into a form that can execute directly on the target machine. For this purpose, it needs knowledge about the input program that goes well beyond syntax. The compiler must build up a large base of knowledge about the detailed computation encoded in the input program. It must know what values are represented, where they reside, and how they flow from name to name. It must understand the structure of the computation. It must analyze how the program interacts with external files and devices. All of these facts can be derived from the source code, using contextual knowledge. Thus, the compiler must perform deeper analysis than is typical for a scanner or a parser.

These kinds of analysis are either performed alongside parsing or in a post-pass that traverses the IR produced by the parser. We call this analysis either

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00004-9

"context-sensitive analysis," to differentiate it from parsing, or "semantic elaboration," since its elaborates the IR. This chapter explores two techniques for organizing this kind of analysis in a compiler: an automated approach based on attribute grammars and an ad hoc approach that relies on similar concepts.

Conceptual Roadmap

To accumulate the contextual knowledge needed for further translation, the compiler must develop ways of viewing the program other than syntax. It uses abstractions that represent some aspect of the code, such as a type system, a storage map, or a control-flow graph. It must understand the program's name space: the kinds of data represented in the program, the kinds of data that can be associated with each name and each expression, and the mapping from a name's appearance in the code back to a specific instance of that name. It must understand the flow of control, both within procedures and across procedures. The compiler will have an abstraction for each of these categories of knowledge.

This chapter focuses on mechanisms that compilers use to derive context-sensitive knowledge. It introduces one of the abstractions that the compiler manipulates during semantic elaboration, the type system. (Others are introduced in later chapters.) Next, the chapter presents a principled automatic approach to implementing these computations in the form of *attribute grammars*. It then presents the most widely used technique, *ad hoc syntax-directed translation*, and compares the strengths and weaknesses of these two tools. The advanced topics section includes brief descriptions of situations that present harder problems in type inference, along with a final example of ad hoc syntax-directed translation.

Overview

Consider a single name used in the program being compiled; let's call it *x*. Before the compiler can emit executable target-machine code for computations involving *x*, it must have answers to many questions.

- *What kind of value is stored in x?* Modern programming languages use a plethora of data types, including numbers, characters, boolean values, pointers to other objects, sets (such as `{red, yellow, green}`), and others. Most languages include compound objects that aggregate individual values; these include arrays, structures, sets, and strings.
- *How big is x?* Because the compiler must manipulate *x*, it needs to know the length of *x*'s representation on the target machine. If *x* is a number, it might be one word (an integer or floating-point number), two

words (a double-precision floating-point number or a complex number), or four words (a quad-precision floating-point number or a double-precision complex number). For arrays and strings, the number of elements might be fixed at compile time or it might be determined at runtime.

- *If x is a procedure, what arguments does it take? What kind of value, if any, does it return?* Before the compiler can generate code to invoke a procedure, it must know how many arguments the code for the called procedure expects, where it expects to find those arguments, and what kind of value it expects in each argument. If the procedure returns a value, where will the calling routine find that value, and what kind of data will it be? (The compiler must ensure that the calling procedure uses the value in a consistent and safe manner. If the calling procedure assumes that the return value is a pointer that it can dereference, and the called procedure returns an arbitrary character string, the results may not be predictable, safe, or consistent.)
- *How long must x's value be preserved?* The compiler must ensure that *x*'s value remains accessible for any part of the computation that can legally reference it. If *x* is a local variable in Pascal, the compiler can easily overestimate *x*'s interesting lifetime by preserving its value for the duration of the procedure that declares *x*. If *x* is a global variable that can be referenced anywhere, or if it is an element of a structure explicitly allocated by the program, the compiler may have a harder time determining its lifetime. The compiler can always preserve *x*'s value for the entire computation; however, more precise information about *x*'s lifetime might let the compiler reuse its space for other values with nonconflicting lifetimes.
- *Who is responsible for allocating space for x (and initializing it)?* Is space allocated for *x* implicitly, or does the program explicitly allocate space for it? If the allocation is explicit, then the compiler must assume that *x*'s address cannot be known until the program runs. If, on the other hand, the compiler allocates space for *x* in one of the runtime data structures that it manages, then it knows more about *x*'s address. This knowledge may let it generate more efficient code.

The compiler must derive the answers to these questions, and more, from the source program and the rules of the source language. In an Algol-like language, such as Pascal or C, most of these questions can be answered by examining the declarations for *x*. If the language has no declarations, as in APL, the compiler must either derive this kind of information by analyzing the program, or it must generate code that can handle any case that might arise.

Many, if not all, of these questions reach beyond the context-free syntax of the source language. For example, the parse trees for x ← y and x ← z differ only in the text of the name on the right-hand side of the assignment. If x and y are integers while z is a character string, the compiler may need to emit different code for x ← y than for x ← z. To distinguish between these cases, the compiler must delve into the program's meaning. Scanning and parsing deal solely with the program's form; the analysis of meaning is the realm of *context-sensitive analysis*.

To see this difference between syntax and meaning more clearly, consider the structure of a program in most Algol-like languages. These languages require that every variable be declared before it is used and that each use of a variable be consistent with its declaration. The compiler writer can structure the syntax to ensure that all declarations occur before any executable statement. A production such as

ProcedureBody → *Declarations Executables*

where the nonterminals have the obvious meanings, ensures that all declarations occur before any executable statements. This syntactic constraint does nothing to check the deeper rule—that the program actually declares each variable before its first use in an executable statement. Neither does it provide an obvious way to handle the rule in C++ that requires declaration before use for some categories of variables, but lets the programmer intermix declarations and executable statements.

To solve this particular problem, the compiler typically creates a table of names. It inserts a name on declaration; it looks up the name at each reference. A lookup failure indicates a missing declaration.

This ad hoc solution bolts onto the parser, but uses mechanisms well outside the scope of context-free languages.

Enforcing the "declare before use" rule requires a deeper level of knowledge than can be encoded in the context-free grammar. The context-free grammar deals with syntactic categories rather than specific words. Thus, the grammar can specify the positions in an expression where a variable name may occur. The parser can recognize that the grammar allows a variable name to occur, and it can tell that one has occurred. However, the grammar has no way to match one instance of a variable name with another; that would require the grammar to specify a much deeper level of analysis—an analysis that can account for context and that can examine and manipulate information at a deeper level than context-free syntax.

4.2 AN INTRODUCTION TO TYPE SYSTEMS

Type
an abstract category that specifies properties held in common by all its members

Common types include *integer*, *list*, and *character*.

Most programming languages associate a collection of properties with each data value. We call this collection of properties the value's *type*. The type specifies a set of properties held in common by all values of that type. Types can be specified by membership; for example, an integer might be any whole number i in the range $-2^{31} \leq i < 2^{31}$, or red

might be a value in an enumerated type `colors`, defined as the set {`red`, `orange`, `yellow`, `green`, `blue`, `brown`, `black`, `white`}. Types can be specified by rules; for example, the declaration of a structure in C defines a type. In this case, the type includes any object with the declared fields in the declared order; the individual fields have types that specify the allowable ranges of values and their interpretation. (We represent the type of a structure as the product of the types of its constituent fields, in order.) Some types are predefined by a programming language; others are constructed by the programmer. The set of types in a programming language, along with the rules that use types to specify program behavior, are collectively called a *type system*.

4.2.1 The Purpose of Type Systems

Programming-language designers introduce type systems so that they can specify program behavior at a more precise level than is possible in a context-free grammar. The type system creates a second vocabulary for describing both the form and behavior of valid programs. Analyzing a program from the perspective of its type system yields information that cannot be obtained using the techniques of scanning and parsing. In a compiler, this information is typically used for three distinct purposes: safety, expressiveness, and runtime efficiency.

Ensuring Runtime Safety

A well-designed type system helps the compiler detect and avoid runtime errors. The type system should ensure that programs are well behaved—that is, the compiler and runtime system can identify all ill-formed programs before they execute an operation that causes a runtime error. In truth, the type system cannot catch all ill-formed programs; the set of ill-formed programs is not computable. Some runtime errors, such as dereferencing an out-of-bounds pointer, have obvious (and often catastrophic) effects. Others, such as mistakenly interpreting an integer as a floating-point number, can have subtle and cumulative effects. The compiler should eliminate as many runtime errors as it can using type-checking techniques.

Type inference
the process of determining a type for each name and each expression in the code

To accomplish this, the compiler must first infer a type for each expression. These inferred types expose situations in which a value is incorrectly interpreted, such as using a floating-point number in place of a boolean value. Second, the compiler must check the types of the operands of each operator against the rules that define what the language allows. In some cases, these rules might require the compiler to convert values from one representation to another. In other circumstances, they may forbid such a conversion and simply declare that the program is ill formed and, therefore, not executable.

+	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	*illegal*
complex	complex	complex	*illegal*	complex

■ **FIGURE 4.1** Result Types for Addition in FORTRAN 77.

Implicit conversion
Many languages specify rules that allow an operator to combine values of different type and require that the compiler insert conversions as needed.

The alternative is to require the programmer to write an *explicit conversion* or *cast*.

In many languages, the compiler can infer a type for every expression. FORTRAN 77 has a particularly simple type system with just a handful of types. Figure 4.1 shows all the cases that can arise for the + operator. Given an expression `a + b` and the types of `a` and `b`, the table specifies the type of `a + b`. For an integer `a` and a double-precision `b`, `a + b` produces a double-precision result. If, instead, `a` were complex, `a + b` would be illegal. The compiler should detect this situation and report it before the program executes—a simple example of type safety.

For some languages, the compiler cannot infer types for all expressions. APL, for example, lacks declarations, allows a variable's type to change at any assignment, and lets the user enter arbitrary code at input prompts. While this makes APL powerful and expressive, it ensures that the implementation must do some amount of runtime type inference and checking. The alternative, of course, is to assume that the program behaves well and ignore such checking. In general, this leads to bad behavior when a program goes awry. In APL, many of the advanced features rely heavily on the availability of type and dimension information.

Safety is a strong reason for using typed languages. A language implementation that guarantees to catch most type-related errors before they execute can simplify the design and implementation of programs. A language in which every expression can be assigned an unambiguous type is called a *strongly typed* language. If every expression can be typed at compile time, the language is *statically typed*; if some expressions can only be typed at runtime, the language is *dynamically typed*. Two alternatives exist: an *untyped* language, such as assembly code or BCPL, and a *weakly typed* language—one with a poor type system.

Improving Expressiveness

A well-constructed type system allows the language designer to specify behavior more precisely than is possible with context-free rules. This capability lets the language designer include features that would be impossible

to specify in a context-free grammar. An excellent example is *operator overloading*, which gives context-dependent meanings to an operator. Many programming languages use + to signify several kinds of addition. The interpretation of + depends on the types of its operands. In typed languages, many operators are overloaded. The alternative, in an untyped language, is to provide lexically different operators for each case.

Operator overloading
An operator that has different meanings based on the types of its arguments is "overloaded."

For example, in BCPL, the only type is a "cell." A cell can hold any bit pattern; the interpretation of that bit pattern is determined by the operator applied to the cell. Because cells are essentially untyped, operators cannot be overloaded. Thus, BCPL uses + for integer addition and #+ for floating-point addition. Given two cells `a` and `b`, both `a + b` and `a #+ b` are valid expressions, neither of which performs any conversion on its operands.

In contrast, even the oldest typed languages use overloading to specify complex behavior. As described in the previous section, FORTRAN has a single addition operator, +, and uses type information to determine how it should be implemented. ANSI C uses function prototypes—declarations of the number and type of a function's parameters and the type of its returned value—to convert arguments to the appropriate types. Type information determines the effect of autoincrementing a pointer in C; the amount of the increment is determined by the pointer's type. Object-oriented languages use type information to select the appropriate implementation at each procedure call. For example, Java selects between a default constructor and a specialized one by examining the constructor's argument list.

Generating Better Code

A well-designed type system provides the compiler with detailed information about every expression in the program—information that can often be used to produce more efficient translations. Consider implementing addition in FORTRAN 77. The compiler can completely determine the types of all expressions, so it can consult a table similar to the one in Figure 4.2. The code on the right shows the ILOC operation for the addition, along with the conversions specified in the FORTRAN standard for each mixed-type expression. The full table would include all the cases from Figure 4.1.

In a language with types that cannot be wholly determined at compile time, some of this checking might be deferred until runtime. To accomplish this, the compiler would need to emit code similar to the pseudo-code in Figure 4.3. The figure only shows the code for two numeric types, integer and real. An actual implementation would need to cover the entire set of possibilities. While this approach ensures runtime safety, it adds significant

Type of			Code
a	b	a + b	
integer	integer	integer	iADD r_a, r_b ⇒ r_{a+b}
integer	real	real	i2f f_a ⇒ r_{a_f} fADD r_{a_f}, r_b ⇒ r_{a_f+b}
integer	double	double	i2d r_a ⇒ r_{a_d} dADD r_{a_d}, r_b ⇒ r_{a_d+b}
real	real	real	fADD r_a, r_b ⇒ r_{a+b}
real	double	double	r2d r_a ⇒ r_{a_d} dADD r_{a_d}, r_b ⇒ r_{a_d+b}
double	double	double	dADD r_a, r_b ⇒ r_{a+b}

■ **FIGURE 4.2** Implementing Addition in FORTRAN 77.

overhead to each operation. One goal of compile-time checking is to provide such safety without the runtime cost.

Notice that runtime type checking requires a runtime representation for type. Thus, each variable has both a value field and a tag field. The code that performs runtime checking—the nested if-then-else structure in Figure 4.3—relies on the tag fields, while the arithmetic uses the value fields. With tags, each data item needs more space, that is, more bytes in memory. If a variable is stored in a register, both its value and its tag will need registers. Finally, tags must be initialized, read, compared, and written at runtime. All of those activities add overhead to a simple addition operation.

The benefit of keeping a in a register comes from speed of access. If a's tag is in RAM, that benefit is lost.

An alternative is to use part of the space in a to store the tag and to reduce the range of values that a can hold.

Runtime type checking imposes a large overhead on simple arithmetic and on other operations that manipulate data. Replacing a single addition, or a conversion and an addition, with the nest of if-then-else code in Figure 4.3 has a significant performance impact. The size of the code in Figure 4.3 strongly suggests that operators such as addition be implemented as procedures and that each instance of an operator be treated as a procedure call. In a language that requires runtime type checking, the costs of runtime checking can easily overwhelm the costs of the actual operations.

Performing type inference and checking at compile time eliminates this kind of overhead. It can replace the complex code of Figure 4.3 with the fast, compact code of Figure 4.2. From a performance perspective, compile-time checking is *always* preferable. However, language design determines whether or not that is possible.

```
// partial code for "a+b ⇒ c"
if (tag(a) = integer) then
     if (tag(b) = integer) then
          value(c) = value(a) + value(b);
          tag(c) = integer;
     else if (tag(b) = real) then
          temp = ConvertToReal(a);
          value(c) = temp + value(b);
          tag(c) = real;
     else if (tag(b) = ...) then
          // handle all other types ...
     else
          signal runtime type fault
else if (tag(a) = real) then
     if (tag(b) = integer) then
          temp = ConvertToReal(b);
          value(c) = value(a) + temp;
          tag(c) = real;
     else if (tag(b) = real) then
          value(c) = value(a) + value(b);
          tag(c) = real;
     else if (tag(b) = ...) then
          // handle all other types ...
     else
          signal runtime type fault
else if (tag(a) = ...) then
     // handle all other types ...
else
     signal illegal tag value;
```

■ **FIGURE 4.3** Schema for Implementing Addition with Runtime Type Checking.

Type Checking

To avoid the overhead of runtime type checking, the compiler must analyze the program and assign a type to each name and each expression. It must check these types to ensure that they are used in contexts where they are legal. Taken together, these activities are often called *type checking*. This is an unfortunate misnomer, because it lumps together the separate activities of type inference and identifying type-related errors.

The programmer should understand how type checking is performed in a given language and compiler. A strongly typed, statically checkable language might be implemented with runtime checking (or with no checking). An untyped language might be implemented in a way that catches certain kinds of errors. Both ML and Modula-3 are good examples of strongly typed languages that can be statically checked. Common Lisp has a strong type system that must be checked dynamically. ANSI C is a typed language, but some implementations do a poor job of identifying type errors.

The theory underlying type systems encompasses a large and complex body of knowledge. This section provides an overview of type systems and introduces some simple problems in type checking. Subsequent sections use simple problems of type inference as examples of context-sensitive computations.

4.2.2 Components of a Type System

A type system for a typical modern language has four major components: a set of base types, or built-in types; rules for constructing new types from the existing types; a method for determining if two types are equivalent or compatible; and rules for inferring the type of each source-language expression. Many languages also include rules for the implicit conversion of values from one type to another based on context. This section describes each of these in more detail, with examples from popular programming languages.

Base Types

Most programming languages include base types for some, if not all, of the following kinds of data: numbers, characters, and booleans. These types are directly supported by most processors. Numbers typically come in several forms, such as integers and floating-point numbers. Individual languages add other base types. Lisp includes both a rational number type and a recursive type `cons`. Rational numbers are, essentially, pairs of integers interpreted as ratios. A `cons` is defined as either the designated value `nil` or `(cons first rest)` where `first` is an object, `rest` is a `cons`, and `cons` creates a list from its arguments.

The precise definitions for base types, and the operators defined for them, vary across languages. Some languages refine these base types to create more; for example, many languages distinguish between several types of numbers in their type systems. Other languages lack one or more of these base types. For example, C has no string type, so C programmers use an array of characters instead. Almost all languages include facilities to construct more complex types from their base types.

Numbers

Almost all programming languages include one or more kinds of numbers as base types. Typically, they support both limited-range integers and approximate real numbers, often called *floating-point* numbers. Many programming languages expose the underlying hardware implementation by creating distinct types for different hardware implementations. For example, C, C++, and Java distinguish between signed and unsigned integers.

FORTRAN, PL/I, and C expose the size of numbers. Both C and FORTRAN specify the length of data items in relative terms. For example, a `double` in FORTRAN is twice the length of a `real`. Both languages, however, give the compiler control over the length of the smallest category of number. In contrast, PL/I declarations specify a length in bits. The compiler maps this desired length onto one of the hardware representations. Thus, the IBM 370 implementation of PL/I mapped both a `fixed binary(12)` and a `fixed binary(15)` variable to a 16-bit integer, while a `fixed binary(31)` became a 32-bit integer.

Some languages specify implementations in detail. For example, Java defines distinct types for signed integers with lengths of 8, 16, 32, and 64 bits. Respectively, they are `byte`, `short`, `int`, and `long`. Similarly, Java's `float` type specifies a 32-bit IEEE floating-point number, while its `double` type specifies a 64-bit IEEE floating-point number. This approach ensures identical behavior on different architectures.

Scheme takes a different approach. The language defines a hierarchy of number types but lets the implementor select a subset to support. However, the standard draws a careful distinction between exact and inexact numbers and specifies a set of operations that should return an exact number when all of its arguments are exact. This provides a degree of flexibility to the implementer, while allowing the programmer to reason about when and where approximation can occur.

Characters

Many languages include a character type. In the abstract, a character is a single letter. For years, due to the limited size of the Western alphabets, this led to a single-byte (8-bit) representation for characters, usually mapped into the ASCII character set. Recently, more implementations—both operating system and programming language—have begun to support larger character sets expressed in the Unicode standard format, which requires 16 bits. Most languages assume that the character set is ordered, so that standard comparison operators, such as <, =, and >, work intuitively, enforcing lexicographic ordering. Conversion between a character and an integer appears in some languages. Few other operations make sense on character data.

Booleans

Most programming languages include a boolean type that takes on two values: `true` and `false`. Standard operations provided for booleans include `and`, `or`, `xor`, and `not`. Boolean values, or boolean-valued expressions, are often used to determine the flow of control. C considers boolean values as a subrange of the unsigned integers, restricted to the values zero (`false`) and one (`true`).

Compound and Constructed Types

While the base types of a programming language usually provide an adequate abstraction of the actual kinds of data handled directly by the hardware, they are often inadequate to represent the information domain needed by programs. Programs routinely deal with more complex data structures, such as graphs, trees, tables, arrays, records, lists, and stacks. These structures consist of one or more objects, each with its own type. The ability to construct new types for these compound or aggregate objects is an essential feature of many programming languages. It lets the programmer organize information in novel, program-specific ways. Tying these organizations to the type system improves the compiler's ability to detect ill-formed programs. It also lets the language express higher-level operations, such as a whole-structure assignment.

Take, for example, Lisp, which provides extensive support for programming with lists. Lisp's list is a constructed type. A list is either the designated value `nil` or `(cons first rest)` where `first` is an object, `rest` is a list, and `cons` is a constructor that creates a list from its two arguments. A Lisp implementation can check each call to `cons` to ensure that its second argument is, in fact, a list.

Arrays

Arrays are among the most widely used aggregate objects. An array groups together multiple objects of the same type and gives each a distinct name—albeit an implicit, computed name rather than an explicit, programmer-designated, name. The C declaration `int a[100][200];` sets aside space for $100 \times 200 = 20{,}000$ integers and ensures that they can be addressed using the name `a`. The references `a[1][17]` and `a[2][30]` access distinct and independent memory locations. The essential property of an array is that the program can compute names for each of its elements by using numbers (or some other ordered, discrete type) as subscripts.

Support for operations on arrays varies widely. FORTRAN 90, PL/I, and APL all support assignment of whole or partial arrays. These languages support element-by-element application of arithmetic operations to arrays. For

10×10 arrays `x`, `y`, and `z`, indexed from 1 to 10, the statement `x = y + z` would overwrite each `x[i,j]` with `y[i,j] + z[i,j]` for all $1 \leq i, j \leq 10$. APL takes the notion of array operations further than most languages; it includes operators for inner product, outer product, and several kinds of reductions. For example, the sum reduction of `y`, written `x ← +/y`, assigns `x` the scalar sum of the elements of `y`.

An array can be viewed as a constructed type because we construct an array by specifying the type of its elements. Thus, a 10×10 array of integers has type *two-dimensional array of integers*. Some languages include the array's dimensions in its type; thus a 10×10 array of integers has a different type than a 12×12 array of integers. This lets the compiler catch array operations in which dimensions are incompatible as a type error. Most languages allow arrays of any base type; some languages allow arrays of constructed types as well.

Strings

Some programming languages treat strings as a constructed type. PL/I, for example, has both bit strings and character strings. The properties, attributes, and operations defined on both of these types are similar; they are properties of a string. The range of values allowed in any position differs between a bit string and a character string. Thus, viewing them as *string of bit* and *string of character* is appropriate. (Most languages that support strings limit the built-in support to a single string type—the character string.) Other languages, such as C, support character strings by handling them as arrays of characters.

A true string type differs from an array type in several important ways. Operations that make sense on strings, such as concatenation, translation, and computing the length, may not have analogs for arrays. Conceptually, string comparison should work from lexicographic order, so that `"a" < "boo"` and `"fee" < "fie"`. The standard comparison operators can be overloaded and used in the natural way. Implementing comparison for an array of characters suggests an equivalent comparison for an array of numbers or an array of structures, where the analogy to strings may not hold. Similarly, the actual length of a string may differ from its allocated size, while most uses of an array use all the allocated elements.

Enumerated Types

Many languages allow the programmer to create a type that contains a specific set of constant values. An *enumerated type*, introduced in Pascal, lets the programmer use self-documenting names for small sets of constants. Classic examples include days of the week and months. In C syntax, these might be

```
enum WeekDay {Monday, Tuesday, Wednesday,
              Thursday, Friday, Saturday, Sunday};

enum Month {January, February, March, April,
            May, June, July, August, September,
            October, November, December};
```

The compiler maps each element of an enumerated type to a distinct value. The elements of an enumerated type are ordered, so comparisons between elements of the same type make sense. In the examples, `Monday < Tuesday` and `June < July`. Operations that compare different enumerated types make no sense—for example, `Tuesday > September` should produce a type error, Pascal ensures that each enumerated type behaves as if it were a subrange of the integers. For example, the programmer can declare an array indexed by the elements of an enumerated type.

Structures and Variants

Structures, or *records*, group together multiple objects of arbitrary type. The elements, or members, of the structure are typically given explicit names. For example, a programmer implementing a parse tree in C might need nodes with both one and two children.

```
struct Node1 {
  struct   Node1 *left;
  unsigned Operator;
  int      Value
}
```

```
struct Node2 {
  struct   Node2 *left;
  struct   Node2 *right;
  unsigned Operator;
  int      Value
}
```

The type of a structure is the ordered product of the types of the individual elements that it contains. Thus, we might describe the type of a `Node1` as `(Node1 *)` × `unsigned` × `int`, while a `Node2` would be `(Node2 *)` × `(Node2 *)` × `unsigned` × `int`. These new types should have the same essential properties that a base type has. In C, autoincrementing a pointer to a `Node1` or casting a pointer into a `Node1 *` has the desired effect—the behavior is analogous to what happens for a base type.

Many programming languages allow the creation of a type that is the union of other types. For example, some variable `x` can have the type `integer` or `boolean` or `WeekDay`. In Pascal, this is accomplished with variant records—a *record* is the Pascal term for a structure. In C, this is accomplished with a `union`. The type of a `union` is the alternation of its component types; thus our variable `x` has type `integer` ∪ `boolean` ∪ `WeekDay`. Unions can also

AN ALTERNATIVE VIEW OF STRUCTURES

The classical view of structures treats each kind of structure as a distinct type. This approach to structure types follows the treatment of other aggregates, such as arrays and strings. It seems natural. It makes distinctions that are useful to the programmer. For example, a tree node with two children probably should have a different type than a tree node with three children; presumably, they are used in different situations. A program that assigns a three-child node to a two-child node should generate a type error and a warning message to the programmer.

From the perspective of the runtime system, however, treating each structure as a distinct type complicates the picture. With distinct structure types, the heap contains an arbitrary set of objects drawn from an arbitrary set of types. This makes it difficult to reason about programs that deal directly with the objects on the heap, such as a garbage collector. To simplify such programs, their authors sometimes take a different approach to structure types.

This alternate model considers all structures in the program as a single type. Individual structure declarations each create a variant form of the type *structure*. The type *structure*, itself, is the union of all these variants. This approach lets the program view the heap as a collection of objects of a single type, rather than a collection of many types. This view makes code that manipulates the heap much simpler to analyze and optimize.

include structures of distinct types, even when the individual structure types have different lengths. The language must provide a mechanism to reference each field unambiguously.

Pointers

Pointers are abstract memory addresses that let the programmer manipulate arbitrary data structures. Many languages include a pointer type. Pointers let a program save an address and later examine the object that it addresses. Pointers are created when objects are created (`new` in Java or `malloc` in C). Some languages provide an operator that returns the address of an object, such as C's `&` operator.

The address operator, when applied to an object of type *t*, returns a value of type *pointer to t*.

To protect programmers from using a pointer to type *t* to reference a structure of type *s*, some languages restrict pointer assignment to "equivalent" types. In these languages, the pointer on the left-hand side of an assignment must have the same type as the expression on the right-hand side. A program can legally assign a *pointer to integer* to a variable declared as *pointer to integer* but not to one declared as *pointer to pointer to integer* or *pointer to boolean*.

These latter assignments are either illegal or require an explicit conversion by the programmer.

Polymorphism
A function that can operate on arguments of different types is a *polymorphic* function.

If the set of types must be specified explicitly, the function uses *ad hoc polymorphism*; if the function body does not specify types, it uses *parametric polymorphism*.

Of course, the mechanism for creating new objects should return an object of the appropriate type. Thus, Java's `new` explicitly creates a typed object; other languages use a polymorphic routine that takes the return type as a parameter. ANSI C handles this in an unusual way: The standard allocation routine `malloc` returns a *pointer to* `void`. This forces the programmer to cast the value returned by each call to `malloc`.

Some languages allow direct manipulation of pointers. Arithmetic on pointers, including autoincrement and autodecrement, allow the program to construct new pointers. C uses the type of a pointer to determine autoincrement and decrement magnitudes. The programmer can set a pointer to the start of an array; autoincrementing advances the pointer from one element in the array to the next element.

Type safety with pointers relies on an implicit assumption that addresses correspond to typed objects. The ability to construct new pointers seriously reduces the ability of both the compiler and its runtime system to reason about pointer-based computations and to optimize such code. (See, for example, Section 8.4.1.)

Type Equivalence

A critical component of any type system is the mechanism that it uses to decide whether or not two different type declarations are equivalent. Consider the two declarations in C shown in the margin. Are `Tree` and `STree` the same type? Are they equivalent? Any programming language with a nontrivial type system must include an unambiguous rule to answer this question for arbitrary types.

```
struct Tree {
  struct Tree *left;
  struct Tree *right;
  int value
}

struct STree {
  struct STree *left;
  struct STree *right;
  int value
}
```

Historically, two general approaches have been tried. The first, *name equivalence*, asserts that two types are equivalent if and only if they have the same name. Philosophically, this rule assumes that the programmer can select any name for a type; if the programmer chooses different names, the language and its implementation should honor that deliberate act. Unfortunately, the difficulty of maintaining consistent names grows with the size of the program, the number of authors, and the number of distinct files of code.

The second approach, *structural equivalence*, asserts that two types are equivalent if and only if they have the same structure. Philosophically, this rule asserts that two objects are interchangeable if they consist of the same set of fields, in the same order, and those fields all have equivalent types. Structural equivalence examines the essential properties that define the type.

REPRESENTING TYPES

As with most objects that a compiler must manipulate, types need an internal representation. Some languages, such as FORTRAN 77, have a small fixed set of types. For these languages, a small integer tag is both efficient and sufficient. However, many modern languages have open-ended type systems. For these languages, the compiler writer needs to design a structure that can represent arbitrary types.

If the type system is based on name equivalence, any number of simple representations will suffice, as long as the compiler can use the representation to trace back to a representation of the actual structure. If the type system is based on structural equivalence, the representation of the type must encode its structure. Most such systems build trees to represent types. They construct a tree for each type declaration and compare tree structures to test for equivalence.

Each policy has strengths and weaknesses. Name equivalence assumes that identical names occur as a deliberate act; in a large programming project, this requires discipline to avoid unintentional clashes. Structural equivalence assumes that interchangeable objects can be used safely in place of one another; if some of the values have "special" meanings, this can create problems. (Imagine two hypothetical, structurally identical types. The first holds a system I/O control block, while the second holds the collection of information about a bit-mapped image on the screen. Treating them as distinct types would allow the compiler to detect a misuse—passing the I/O control block to a screen refresh routine—while treating them as the same type would not.)

Inference Rules

In general, type inference rules specify, for each operator, the mapping between the operand types and the result type. For some cases, the mapping is simple. An assignment, for example, has one operand and one result. The result, or left-hand side, must have a type that is compatible with the type of the operand, or right-hand side. (In Pascal, the subrange `1..100` is compatible with the integers since any element of the subrange can be assigned safely to an integer.) This rule allows assignment of an integer value to an integer variable. It forbids assignment of a structure to an integer variable, without an explicit conversion that makes sense of the operation.

The relationship between operand types and result types is often specified as a recursive function on the type of the expression tree. The function computes the result type of an operation as a function of the types of its

operands. The functions might be specified in tabular form, similar to the table in Figure 4.1. Sometimes, the relationship between operand types and result types is specified by a simple rule. In Java, for example, adding two integer types of different precision produces a result of the more precise (longer) type.

The inference rules point out type errors. Mixed-type expressions may be illegal. In FORTRAN 77, a program cannot add a `double` and a `complex`. In Java, a program cannot assign a number to a character. These combinations should produce a type error at compile time, along with a message that indicates how the program is ill formed.

Some languages require the compiler to perform implicit conversions. The compiler must recognize certain combinations of mixed-type expressions and handle them by inserting the appropriate conversions. In FORTRAN, adding an integer and a floating-point number forces conversion of the integer to floating-point form before the addition. Similarly, Java mandates implicit conversions for integer addition of values with different precision. The compiler must coerce the less precise value to the form of the more precise value before addition. A similar situation arises in Java with integer assignment. If the right-hand side is less precise, it is converted to the more precise type of the left-hand side. If, however, the left-hand side is less precise than the right-hand side, the assignment produces a type error unless the programmer inserts an explicit cast operation to change its type and coerce its value.

Declarations and Inference

As previously mentioned, many programming languages include a "declare before use" rule. With mandatory declarations, each variable has a well-defined type. The compiler needs a way to assign types to constants. Two approaches are common. Either a constant's form implies a specific type—for example, `2` is an integer and `2.0` is a floating-point number—or the compiler infers a constant's type from its usage—for example, `sin(2)` implies that `2` is a floating-point number, while `x` ← `2`, for integer `x`, implies that `2` is an integer. With declared types for variables, implied types for constants, and a complete set of type-inference rules, the compiler can assign types to any expression over variables and constants. Function calls complicate the picture, as we shall see.

This scheme overloads 2 with different meanings in different contexts. Experience suggests that programmers are good at understanding this kind of overloading.

Some languages absolve the programmer from writing any declarations. In these languages, the problem of type inference becomes substantially more intricate. Section 4.5 describes some of the problems that this creates and some of the techniques that compilers use to address them.

CLASSIFYING TYPE SYSTEMS

Many terms are used to describe type systems. In the text, we have introduced the terms *strongly typed*, *untyped*, and *weakly typed* languages. Other distinctions between type systems and their implementations are important.

Checked versus Unchecked Implementations The implementation of a programming language may elect to perform enough checking to detect and to prevent all runtime errors that result from misuse of a type. (This may actually exclude some value-specific errors, such as division by zero.) Such an implementation is called *strongly checked*. The opposite of a strongly checked implementation is an *unchecked implementation*—one that assumes a well-formed program. Between these poles lies a spectrum of *weakly checked implementations* that perform partial checking.

Compile Time versus Runtime Activity A strongly typed language may have the property that all inference and all checking can be done at compile time. An implementation that actually does all this work at compile time is called *statically typed* and *statically checked*. Some languages have constructs that must be typed and checked at runtime. We term these languages *dynamically typed* and *dynamically checked*. To confuse matters further, of course, a compiler writer can implement a strongly typed, statically typed language with dynamic checking. Java is an example of a language that could be statically typed and checked, except for an execution model that keeps the compiler from seeing all the source code at once. This forces it to perform type inference as classes are loaded and to perform some of the checking at runtime.

Inferring Types for Expressions

The goal of type inference is to assign a type to each expression that occurs in a program. The simplest case for type inference occurs when the compiler can assign a type to each base element in an expression—that is, to each leaf in the parse tree for an expression. This requires declarations for all variables, inferred types for all constants, and type information about all functions.

Conceptually, the compiler can assign a type to each value in the expression during a simple postorder tree walk. This should let the compiler detect every violation of an inference rule, and report it *at compile time*. If the language lacks one or more of the features that make this simple style of inference possible, the compiler will need to use more sophisticated techniques. If

compile time type inference becomes too difficult, the compiler writer may need to move some of the analysis and checking to runtime.

Type inference for expressions, in this simple case, directly follows the expression's structure. The inference rules describe the problem in terms of the source language. The evaluation strategy operates bottom up on the parse tree. For these reasons, type inference for expressions has become a classic example problem to illustrate context-sensitive analysis.

Interprocedural Aspects of Type Inference

Type inference for expressions depends, inherently, on the other procedures that form the executable program. Even in the simplest type systems, expressions contain function calls. The compiler must check each of those calls. It must ensure that each actual parameter is type compatible with the corresponding formal parameter. It must determine the type of any returned value for use in further inference.

Type signature
a specification of the types of the formal parameters and return value(s) of a function

To analyze and understand procedure calls, the compiler needs a *type signature* for each function. For example, the `strlen` function in C's standard library takes an operand of type `char *` and returns an `int` that contains its length in bytes, excluding the terminating character. In C, the programmer can record this fact with a *function prototype* that looks like:

Function prototype
The C language includes a provision that lets the programmer declare functions that are not present. The programmer includes a skeleton declaration, called a *function prototype*.

```
unsigned int strlen(const char *s);
```

This prototype asserts that `strlen` takes an argument of type `char *`, which it does not modify, as indicated by the `const` attribute. The function returns a nonnegative integer. Writing this in a more abstract notation, we might say that

```
strlen : const char * → unsigned int
```

which we read as "`strlen` is a function that takes a constant-valued character string and returns an unsigned integer." As a second example, the classic Scheme function `filter` has the type signature

`filter`: $(\alpha \rightarrow boolean) \times list\ of\ \alpha \rightarrow list\ of\ \alpha$

That is, `filter` is a function that takes two arguments. The first should be a function that maps some type α into a boolean, written $(\alpha \rightarrow boolean)$, and the second should be a list whose elements are of the same type α. Given arguments of those types, `filter` returns a list whose elements have type α. The function `filter` exhibits *parametric polymorphism*: its result type is a function of its argument types.

To perform accurate type inference, the compiler needs a type signature for every function. It can obtain that information in several ways. The compiler can eliminate separate compilation, requiring that the entire program be presented for compilation as a unit. The compiler can require the programmer to provide a type signature for each function; this usually takes the form of mandatory function prototypes. The compiler can defer type checking until either link time or runtime, when all such information is available. Finally, the compiler writer can embed the compiler in a program-development system that gathers the requisite information and makes it available to the compiler on demand. All of these approaches have been used in real systems.

SECTION REVIEW

A type system associates with each value in the program some textual name, a type, that represents a set of common properties held by all values of that type. The definition of a programming language specifies interactions between objects of the same type, such as legal operations on values of a type, and between objects of different type, such as mixed-type arithmetic operations. A well-designed type system can increase the expressiveness of a programming language, allowing safe use of features such as overloading. It can expose subtle errors in a program long before they become puzzling runtime errors or wrong answers. It can let the compiler avoid runtime checks that waste time and space.

A type system consists of a set of base types, rules for constructing new types from existing ones, a method for determining equivalence of two types, and rules for inferring the types of each expression in a program. The notions of base types, constructed types, and type equivalence should be familiar to anyone who has programmed in a high-level language. Type inference plays a critical role in compiler implementation.

Review Questions

1. For your favorite programming language, write down the base types in its type system. What rules and constructs does the language allow to build aggregate types? Does it provide a mechanism for creating a procedure that takes a variable number of arguments, such as `printf` in the C standard I/O library?
2. What kinds of information must the compiler have to ensure type safety at procedure calls? Sketch a scheme based on the use of function prototypes. Sketch a scheme that can check the validity of those function prototypes.

4.3 THE ATTRIBUTE-GRAMMAR FRAMEWORK

Attribute
a value attached to one or more of the nodes in a parse tree

One formalism that has been proposed for performing context-sensitive analysis is the *attribute grammar*, or attributed context-free grammar. An attribute grammar consists of a context-free grammar augmented by a set of rules that specify computations. Each rule defines one value, or *attribute*, in terms of the values of other attributes. The rule associates the attribute with a specific grammar symbol; each instance of the grammar symbol that occurs in a parse tree has a corresponding instance of the attribute. The rules are functional; they imply no specific evaluation order and they define each attribute's value uniquely.

To make these notions concrete, consider a context-free grammar for signed binary numbers. Figure 4.4 defines the grammar $SBN = (T,NT,S,P)$. *SBN* generates all signed binary numbers, such as `-101`, `+11`, `-01`, and `+11111001100`. It excludes unsigned binary numbers, such as `10`.

From *SBN*, we can build an attribute grammar that annotates *Number* with the value of the signed binary number that it represents. To build an attribute grammar from a context-free grammar, we must decide what attributes each node needs, and we must elaborate the productions with rules that define values for these attributes. For our attributed version of *SBN*, the following attributes are needed:

Symbol	Attributes
Number	`value`
Sign	`negative`
List	`position, value`
Bit	`position, value`

In this case, no attributes are needed for the terminal symbols.

Figure 4.5 shows the productions of *SBN* elaborated with attribution rules. Subscripts are added to grammar symbols whenever a specific symbol

$$P = \left\{ \begin{array}{lcl} \textit{Number} & \rightarrow & \textit{Sign List} \\ \textit{Sign} & \rightarrow & + \\ & | & - \\ \textit{List} & \rightarrow & \textit{List Bit} \\ & | & \textit{Bit} \\ \textit{Bit} & \rightarrow & 0 \\ & | & 1 \end{array} \right\}$$

$$T = \{+, -, 0, 1\}$$

$$NT = \{\textit{Number}, \textit{Sign}, \textit{List}, \textit{Bit}\}$$

$$S = \{\textit{Number}\}$$

■ **FIGURE 4.4** An Attribute Grammar for Signed Binary Numbers.

	Production	Attribution Rules

1	*Number* → *Sign List*	*List*.`position` ← *0*
		`if` *Sign*.`negative`
		`then` *Number*.`value` ← -*List*.`value`
		`else` *Number*.`value` ← *List*.`value`

2	*Sign* → +	*Sign*.`negative` ← `false`
3	*Sign* → -	*Sign*.`negative` ← `true`
4	*List* → *Bit*	*Bit*.`position` ← *List*.`position` *List*.`value` ← *Bit*.`value`
5	$List_0 \rightarrow List_1\ Bit$	$List_1$.`position` ← $List_0$.`position`+*1* *Bit*.`position` ← $List_0$.`position` $List_0$.`value` ← $List_1$.`value`+*Bit*.`value`
6	*Bit* → 0	*Bit*.`value` ← *0*
7	*Bit* → 1	*Bit*.`value` ← $2^{Bit.position}$

■ **FIGURE 4.5** Attribute Grammar for Signed Binary Numbers.

appears multiple times in a single production. This practice disambiguates references to that symbol in the rules. Thus, the two occurrences of *List* in production 5 have subscripts, both in the production and in the corresponding rules.

The rules add attributes to the parse tree nodes by their names. An attribute mentioned in a rule must be instantiated for every occurrence of that kind of node.

Each rule specifies the value of one attribute in terms of literal constants and the attributes of other symbols in the production. A rule can pass information from the production's left-hand side to its right-hand side; a rule can also pass information in the other direction. The rules for production 4 pass information in both directions. The first rule sets *Bit*.`position` to *List*.`position`, while the second rule sets *List*.`value` to *Bit*.`value`. Simpler attribute grammars can solve this particular problem; we have chosen this one to demonstrate particular features of attribute grammars.

Given a string in the *SBN* grammar, the attribution rules set *Number*.`value` to the decimal value of the binary input string. For example, the string `-101` causes the attribution shown in Figure 4.6a. (The names for `value`, `number`, and `position` are truncated in the figure.) Notice that *Number*.`value` has the value `-5`.

To evaluate an attributed parse tree for some sentence in $L(SBN)$, the attributes specified in the various rules are instantiated for each node in

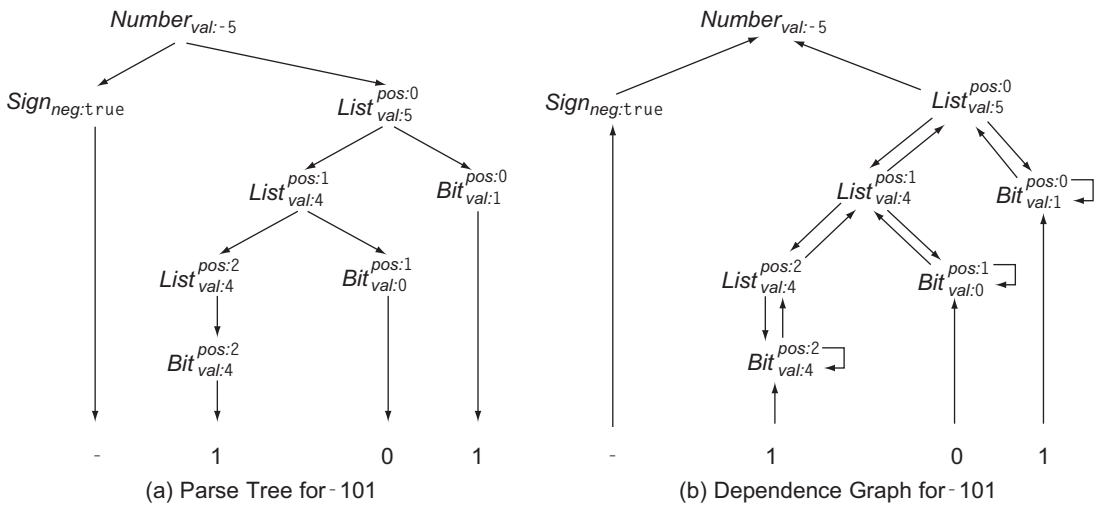
the parse tree. This creates, for example, an attribute instance for both `value` and `position` in each *List* node. Each rule implicitly defines a set of dependences; the attribute being defined depends on each argument to the rule. Taken over the entire parse tree, these dependences form an *attribute-dependence graph*. Edges in the graph follow the flow of values in the evaluation of a rule; an edge from $node_i.field_j$ to $node_k.field_l$ indicates that the rule defining $node_k.field_l$ uses the value of $node_i.field_j$ as one of its inputs. Figure 4.6b shows the attribute-dependence graph induced by the parse tree for the string `-101`.

The bidirectional flow of values that we noted earlier (in, for example, production 4) shows up in the dependence graph, where arrows indicate both flow upward toward the root (*Number*) and flow downward toward the leaves. The *List* nodes show this effect most clearly. We distinguish between attributes based on the direction of value flow. *Synthesized attributes* are defined by bottom-up information flow; a rule that defines an attribute for the production's left-hand side creates a synthesized attribute. A synthesized attribute can draw values from the node itself, its descendants in the parse tree, and constants. *Inherited attributes* are defined by top-down and lateral information flow; a rule that defines an attribute for the production's right-hand side creates an inherited attribute. Since the attribution rule can name any symbol used in the corresponding production, an inherited attribute can draw values from the node itself, its parent and its siblings in the parse tree,

Synthesized attribute
an attribute defined wholly in terms of the attributes of the node, its children, and constants

Inherited attribute
an attribute defined wholly in terms of the node's own attributes and those of its siblings or its parent in the parse tree (plus constants)

The rule *node.field* ← 1 can be treated as either synthesized or inherited.



■ **FIGURE 4.6** Attributed Tree for the Signed Binary Number −101.

and constants. Figure 4.6b shows that the `value` and `negative` attributes are synthesized, while the `position` attribute is inherited.

Any scheme for evaluating attributes must respect the relationships encoded implicitly in the attribute-dependence graph. Each attribute must be defined by some rule. If that rule depends on the values of other attributes, it cannot be evaluated until all those values have been defined. If the rule depends on no other attribute values, then it must produce its value from a constant or some external source. As long as no rule relies on its own value, the rules should uniquely define each value.

Circularity
An attribute grammar is circular if it can, for some inputs, create a cyclic dependence graph.

Of course, the syntax of the attribution rules allows a rule to reference its own result, either directly or indirectly. An attribute grammar containing such rules is ill formed. We say that such rules are *circular* because they can create a cycle in the dependence graph. For the moment, we will ignore circularity; Section 4.3.2 addresses this issue.

The dependence graph captures the flow of values that an evaluator must respect in evaluating an instance of an attributed tree. If the grammar is noncircular, it imposes a partial order on the attributes. This partial order determines when the rule defining each attribute can be evaluated. Evaluation order is unrelated to the order in which the rules appear in the grammar.

Consider the evaluation order for the rules associated with the uppermost *List* node—the right child of *Number*. The node results from applying production five, *List* → *List Bit*; applying that production adds three rules to the evaluation. The two rules that set inherited attributes for the *List* node's children must execute first. They depend on the value of *List*.`position` and set the `position` attributes for the node's subtrees. The third rule, which sets the *List* node's `value` attribute, cannot execute until the two subtrees both have defined `value` attributes. Since those subtrees cannot be evaluated until the first two rules at the *List* node have been evaluated, the evaluation sequence will include the first two rules early and the third rule much later.

To create and use an attribute grammar, the compiler writer determines a set of attributes for each symbol in the grammar and designs a set of rules to compute their values. These rules specify a computation for any valid parse tree. To create an implementation, the compiler writer must create an evaluator; this can be done with an ad hoc program or by using an evaluator generator—the more attractive option. The evaluator generator takes as input the specification for the attribute grammar. It produces the code for an evaluator as its output. This is the attraction of attribute grammars for the compiler writer; the tools take a high-level, nonprocedural specification and automatically produce an implementation.

One critical insight behind the attribute-grammar formalism is the notion that the attribution rules can be associated with productions in the context-free grammar. Since the rules are functional, the values that they produce are independent of evaluation order, for any order that respects the relationships embodied in the attribute-dependence graph. In practice, any order that evaluates a rule only after all of its inputs have been defined respects the dependences.

4.3.1 Evaluation Methods

The attribute-grammar model has practical use only if we can build evaluators that interpret the rules to evaluate an instance of the problem automatically—a specific parse tree, for example. Many attribute evaluation techniques have been proposed in the literature. In general, they fall into three major categories.

1. *Dynamic Methods* These techniques use the structure of a particular attributed parse tree to determine the evaluation order. Knuth's original paper on attribute grammars proposed an evaluator that operated in a manner similar to a dataflow computer architecture—each rule "fired" as soon as all its operands were available. In practical terms, this might be implemented using a queue of attributes that are ready for evaluation. As each attribute is evaluated, its successors in the attribute dependence graph are checked for "readiness" (see Section 12.3). A related scheme would build the attribute dependence graph, topologically sort it, and use the topological order to evaluate the attributes.
2. *Oblivious Methods* In these methods, the order of evaluation is independent of both the attribute grammar and the particular attributed parse tree. Presumably, the system's designer selects a method deemed appropriate for both the attribute grammar and the evaluation environment. Examples of this evaluation style include repeated left-to-right passes (until all attributes have values), repeated right-to-left passes, and alternating left-to-right and right-to-left passes. These methods have simple implementations and relatively small runtime overheads. They lack, of course, any improvement that can be derived from knowledge of the specific tree being attributed.
3. *Rule-Based Methods* Rule-based methods rely on a static analysis of the attribute grammar to construct an evaluation order. In this framework, the evaluator relies on grammatical structure; thus, the parse tree guides the application of the rules. In the signed binary number example, the evaluation order for production 4 should use the first rule to set *Bit*.`position`, recurse downward to *Bit*, and, on return, use *Bit*.`value` to set *List*.`value`. Similarly, for production 5, it should evaluate the first

two rules to define the `position` attributes on the right-hand side, then recurse downward to each child. On return, it can evaluate the third rule to set the *List*.`value` field of the parent *List* node. Tools that perform the necessary static analysis offline can produce fast rule-based evaluators.

4.3.2 Circularity

Circular attribute grammars can give rise to cyclic attribute-dependence graphs. Our models for evaluation fail when the dependence graph contains a cycle. A failure of this kind in a compiler causes serious problems—for example, the compiler might not be able to generate code for its input. The catastrophic impact of cycles in the dependence graph suggests that this issue deserves close attention.

If a compiler uses attribute grammars, it must handle circularity in an appropriate way. Two approaches are possible.

1. *Avoidance* The compiler writer can restrict the attribute grammar to a class that cannot give rise to circular dependence graphs. For example, restricting the grammar to use only synthesized and constant attributes eliminates any possibility of a circular dependence graph. More general classes of noncircular attribute grammars exist; some, like *strongly noncircular attribute grammars,* have polynomial-time tests for membership.
2. *Evaluation* The compiler writer can use an evaluation method that assigns a value to every attribute, even those involved in cycles. The evaluator might iterate over the cycle and assign appropriate or default values. Such an evaluator would avoid the problems associated with a failure to fully attribute the tree.

In practice, most attribute-grammar systems restrict their attention to non-circular grammars. The rule-based evaluation methods may fail to construct an evaluator if the attribute grammar is circular. The oblivious methods and the dynamic methods will attempt to evaluate a circular dependence graph; they will simply fail to define some of the attribute instances.

4.3.3 Extended Examples

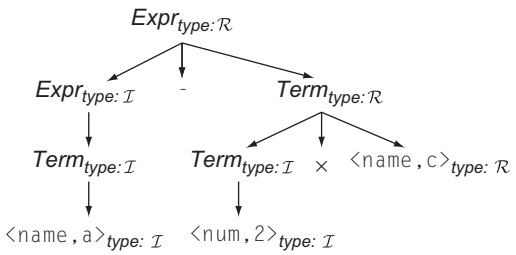
To better understand the strengths and weaknesses of attribute grammars as a tool, we will work through two more detailed examples that might arise in a compiler: inferring types for expression trees in a simple, Algol-like language, and estimating the execution time, in cycles, for a straight-line sequence of code.

Inferring Expression Types

Any compiler that tries to generate efficient code for a typed language must confront the problem of inferring types for every expression in the program. This problem relies, inherently, on context-sensitive information; the type associated with a `name` or `num` depends on its identity—its textual name—rather than its syntactic category.

Consider a simplified version of the type inference problem for expressions derived from the classic expression grammar given in Chapter 3. Assume that the expressions are represented as parse trees, and that any node representing a `name` or `num` already has a *type* attribute. (We will return to the problem of getting the type information into these *type* attributes later in the chapter.) For each arithmetic operator in the grammar, we need a function that maps the two operand types to a result type. We will call these functions $\mathcal{F}_+$, $\mathcal{F}_-$, $\mathcal{F}_\times$, and $\mathcal{F}_\div$; they encode the information found in tables such as the one shown in Figure 4.1. With these assumptions, we can write simple attribution rules that define a *type* attribute for each node in the tree. Figure 4.7 shows the attribution rules.

If `a` has type `integer` (denoted $\mathcal{I}$) and `c` has type `real` (denoted $\mathcal{R}$), then this scheme generates the following attributed parse tree for the input string `a - 2×c`:



The leaf nodes have their *type* attributes initialized appropriately. The remainder of the attributes are defined by the rules from Figure 4.7, with the assumption that $\mathcal{F}_+$, $\mathcal{F}_-$, $\mathcal{F}_\times$, and $\mathcal{F}_\div$ reflect the FORTRAN 77 rules.

A close look at the attribution rules shows that all the attributes are synthesized attributes. Thus, all the dependences flow from a child to its parent in the parse tree. Such grammars are sometimes called *S-attributed grammars*. This style of attribution has a simple, rule-based evaluation scheme. It meshes well with bottom-up parsing; each rule can be evaluated when the parser reduces by the corresponding right-hand side. The attribute-grammar paradigm fits this problem well. The specification is short. It is easily understood. It leads to an efficient evaluator.

Careful inspection of the attributed expression tree shows two cases in which an operation has an operand whose type is different from the type of the

Production			Attribution Rules
$Expr_0$	$\rightarrow$	$Expr_1 + Term$	$Expr_0.type \leftarrow \mathcal{F}_+(Expr_1.type, Term.type)$
	|	$Expr_1 - Term$	$Expr_0.type \leftarrow \mathcal{F}_-(Expr_1.type, Term.type)$
	|	$Term$	$Expr_0.type \leftarrow Term.type$
$Term_0$	$\rightarrow$	$Term_1\ Factor$	$Term_0.type \leftarrow \mathcal{F}_\times(Term_1.type, Factor.type)$
	|	$Term_1\ Factor$	$Term_0.type \leftarrow \mathcal{F}_\div(Term_1.type, Factor.type)$
	|	$Factor$	$Term_0.type \leftarrow Factor.type$
$Factor$	$\rightarrow$	($Expr$)	$Factor.type \leftarrow Expr.type$
	|	num	num.type *is already defined*
	|	name	name.type *is already defined*

■ **FIGURE 4.7** Attribute Grammar to Infer Expression Types.

operation's result. In FORTRAN 77, this requires the compiler to insert a conversion operation between the operand and the operator. For the *Term* node that represents the multiplication of 2 and c, the compiler would convert 2 from an integer representation to a real representation. For the *Expr* node at the root of the tree, the compiler would convert a from an integer to a real. Unfortunately, changing the parse tree does not fit well into the attribute-grammar paradigm.

To represent these conversions in the attributed tree, we could add an attribute to each node that holds its converted type, along with rules to set the attributes appropriately. Alternatively, we could rely on the process that generates code from the tree to compare the two types—parent and child—during the traversal and insert the necessary conversion. The former approach adds some work during attribute evaluation, but localizes all of the information needed for a conversion to a single parse-tree node. The latter approach defers that work until code generation, but does so at the cost of distributing the knowledge about types and conversions across two separate parts of the compiler. Either approach will work; the difference is largely a matter of taste.

A Simple Execution-Time Estimator

As a second example, consider the problem of estimating the execution time of a sequence of assignment statements. We can generate a sequence of assignments by adding three new productions to the classic expression grammar:

$$\begin{array}{lcl} Block & \rightarrow & Block\ Assign \\ & | & Assign \end{array}$$

$$Assign \rightarrow \texttt{name} = Expr;$$

Production			Attribution Rules
$Block_0$	→	$Block_1$ *Assign*	{ $Block_0$. *cost* ← $Block_1$. *cost* + *Assign* . *cost* }
	|	*Assign*	{ $Block_0$. *cost* ← *Assign* . *cost* }
Assign	→	`name` = *Expr;*	{ *Assign* . *cost* ← *Cost*(store) + *Expr* . *cost* }
$Expr_0$	→	$Expr_1$ + *Term*	{ $Expr_0$. *cost* ← $Expr_1$. *cost* + *Cost*(add) + *Term* . *cost* }
	|	$Expr_1$ − *Term*	{ $Expr_0$. *cost* ← $Expr_1$. *cost* + *Cost*(sub) + *Term* . *cost* }
	|	*Term*	{ $Expr_0$. *cost* ← *Term* . *cost* }
$Term_0$	→	$Term_1$ × *Factor*	{ $Term_0$. *cost* ← $Term_1$. *cost* + *Cost*(mult) + *Factor* . *cost* }
	|	$Term_1$ ÷ *Factor*	{ $Term_0$. *cost* ← $Term_1$. *cost* + *Cost*(div) + *Factor* . *cost* }
	|	*Factor*	{ $Term_0$. *cost* ← *Factor* . *cost* }
Factor	→	`(Expr)`	{ *Factor* . *cost* ← *Expr* . *cost* }
	|	`num`	{ *Factor* . *cost* ← *Cost*(loadI) }
	|	`name`	{ *Factor* . *cost* ← *Cost*(load) }

■ **FIGURE 4.8** Simple Attribute Grammar to Estimate Execution Time.

where *Expr* is from the expression grammar. The resulting grammar is simplistic in that it allows only simple identifiers as variables and it contains no function calls. Nonetheless, it is complex enough to convey the issues that arise in estimating runtime behavior.

Figure 4.8 shows an attribute grammar that estimates the execution time of a block of assignment statements. The attribution rules estimate the total cycle count for the block, assuming a single processor that executes one operation at a time. This grammar, like the one for inferring expression types, uses only synthesized attributes. The estimate appears in the `cost` attribute of the topmost *Block* node of the parse tree. The methodology is simple. Costs are computed bottom up; to read the example, start with the productions for *Factor* and work your way up to the productions for *Block*. The function `Cost` returns the latency of a given ILOC operation.

Improving the Execution-Cost Estimator

To make this example more realistic, we can improve its model for how the compiler handles variables. The initial version of our cost-estimating attribute grammar assumes that the compiler naively generates a separate `load` operation for each reference to a variable. For the assignment `x = y + y`, the model counts two load operations for `y`. Few compilers would generate a redundant load for `y`. More likely, the compiler would generate a sequence such as:

```
loadAI  r_arp, @y ⇒ r_y
add     r_y, r_y  ⇒ r_x
storeAI r_x       ⇒ r_arp, @x
```

that loads `y` once. To approximate the compiler's behavior better, we can modify the attribute grammar to charge only a single load for each variable used in the block. This requires more complex attribution rules.

To account for loads more accurately, the rules must track references to each variable by the variable's name. These names are extra-grammatical, since the grammar tracks the syntactic category `name` rather than individual names such as `x`, `y`, and `z`. The rule for `name` should follow the general outline:

```
if ( name has not been loaded )
    then Factor.cost ← Cost(load);
    else Factor.cost ← 0;
```

The key to making this work is the test "`name has not been loaded`."

To implement this test, the compiler writer can add an attribute that holds the set of all variables already loaded. The production *Block* → *Assign* can initialize the set. The rules must thread the expression trees to pass the set through each assignment. This suggests augmenting each node with two sets, `Before` and `After`. The `Before` set for a node contains the lexemes of all `name`s that occur earlier in the *Block*; each of these must have been loaded already. A node's `After` set contains all the names in its `Before` set, plus any `name`s that would be loaded in the subtree rooted at that node.

The expanded rules for *Factor* are shown in Figure 4.9. The code assumes that it can obtain the textual name—the lexeme—of each `name`. The first production, which derives `(` *Expr* `)`, copies the `Before` set down into the *Expr* subtree and copies the `After` set up to the *Factor*. The second production, which derives `num`, simply copies its parent's `Before` set into its parent's `After` set. `num` must be a leaf in the tree; therefore, no further actions are needed. The final production, which derives `name`, performs the critical work. It tests the `Before` set to determine whether or not a load is needed and updates the parent's `cost` and `After` attributes accordingly.

To complete the specification, the compiler writer must add rules that copy the `Before` and `After` sets around the parse tree. These rules, sometimes called *copy rules*, connect the `Before` and `After` sets of the various *Factor* nodes. Because the attribution rules can reference only local attributes—defined as the attributes of a node's parent, its siblings, and its children—the attribute grammar must explicitly copy values around the parse tree to

Production	Attribution Rules
Factor → (*Expr*)	{ *Factor*.*cost* ← *Expr*.*cost*; *Expr*.*Before* ← *Factor*.*Before*; *Factor*.*After* ← *Expr*.*After*}
| num	{ *Factor*.*cost* ← Cost(loadI); *Factor*.*After* ← *Factor*.*Before*}
| name	{ if (name.*lexeme* ∉ *Factor*.*Before*) then *Factor*.*cost* ← *Cost*(load); *Factor*.*After* ← *Factor*.*Before* ∪ {name.*lexeme*} else *Factor*.*cost* ← *0*; *Factor*.*After* ← *Factor*.*Before*}

■ **FIGURE 4.9** Rules to Track Loads in *Factor* Productions.

ensure that they are local. Figure 4.10 shows the required rules for the other productions in the grammar. One additional rule has been added; it initializes the *Before* set of the first *Assign* statement to ∅.

This model is much more complex than the simple model. It has over three times as many rules; each rule must be written, understood, and evaluated. It uses both synthesized and inherited attributes, so the simple bottom-up evaluation strategy will no longer work. Finally, the rules that manipulate the *Before* and *After* sets require a fair amount of attention—the kind of low-level detail that we would hope to avoid by using a system based on high-level specifications.

Back to Inferring Expression Types

In the initial discussion about inferring expression types, we assumed that the attributes name.*type* and num.*type* were already defined by some external mechanism. To fill in those values using an attribute grammar, the compiler writer would need to develop a set of rules for the portion of the grammar that handles declarations.

Those rules would need to record the type information for each variable in the productions associated with the declaration syntax. The rules would need to collect and aggregate that information so that a small set of attributes contained the necessary information on all the declared variables. The rules would need to propagate that information up the parse tree to a node that is an ancestor of all the executable statements, and then to copy it downward into each expression. Finally, at each leaf that is a name or num, the rules would need to extract the appropriate facts from the aggregated information.

Production	Attribution Rules
$Block_0 \rightarrow Block_1\ Assign$	{ $Block_0.cost \leftarrow Block_1.cost + Assign.cost$; $Assign.Before \leftarrow Block_1.After$; $Block_0.After \leftarrow Assign.After$
| $Assign$	{ $Block_0.cost \leftarrow Assign.cost$; $Assign.Before \leftarrow \emptyset$; $Block_0.After \leftarrow Assign.After$ }
$Assign \rightarrow$ `name` = $Expr$;	{ $Assign.cost \leftarrow Cost($`store`$) + Expr.cost$; $Expr.Before \leftarrow Assign.Before$; $Assign.After \leftarrow Expr.After$ }
$Expr_0 \rightarrow Expr_1 + Term$	{ $Expr_0.cost \leftarrow Expr_1.cost + Cost($`add`$) + Term.cost$; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$ }
| $Expr_1 - Term$	{ $Expr_0.cost \leftarrow Expr_1.cost + Cost($`sub`$) + Term.cost$; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$ }
| $Term$	{ $Expr_0.cost \leftarrow Term.cost$; $Term.Before \leftarrow Expr_0.Before$; $Expr_0.After \leftarrow Term.After$ }
$Term_0 \rightarrow Term_1 \times Factor$	{ $Term_0.cost \leftarrow Term_1.cost + Cost($`mult`$) + Factor.cost$; $Term_1.Before \leftarrow Term_0.Before$; $Factor.Before \leftarrow Term_1.After$; $Term_0.After \leftarrow Factor.After$ }
| $Term_1 \div Factor$	{ $Term_0.cost \leftarrow Term_1.cost + Cost($`div`$) + Factor.cost$; $Term_1.Before \leftarrow Term_0.Before$; $Factor.Before \leftarrow Term_1.After$; $Term_0.After \leftarrow Factor.After$ }
| $Factor$	{ $Term_0.cost \leftarrow Factor.cost$; $Factor.Before \leftarrow Term_0.Before$; $Term_0.After \leftarrow Factor.After$ }

■ **FIGURE 4.10** Copy Rules to Track Loads.

The resulting set of rules would be similar to those that we developed for tracking loads but would be more complex at the detailed level. These rules also create large, complex attributes that must be copied around the parse tree. In a naive implementation, each instance of a copy rule would create a new copy. Some of these copies could be shared, but many of the versions created by merging information from multiple children will differ (and, thus, need to be distinct copies). The same problem arises with the `Before` and `After` sets in the previous example.

A Final Improvement to the Execution-Cost Estimator

While tracking loads improved the fidelity of the estimated execution costs, many further refinements are possible. Consider, for example, the impact of finite register sets on the model. So far, our model has assumed that the target computer provides an unlimited set of registers. In reality, computers provide small register sets. To model the capacity of the register set, the estimator could limit the number of values allowed in the `Before` and `After` sets.

As a first step, we must replace the implementation of `Before` and `After`. They were implemented with arbitrarily sized sets; in this refined model, the sets should hold exactly k values, where k is the number of registers available to hold the values of variables. Next, we must rewrite the rules for the production *Factor* → `name` to model register occupancy. If a value has not been loaded, and a register is available, it charges for a simple load. If a load is needed, but no register is available, it can evict a value from some register and charge for the load. The choice of which value to evict is complex; it is discussed in Chapter 13. Since the rule for *Assign* always charges for a store, the value in memory will be current. Thus, no store is needed when a value is evicted. Finally, if the value has already been loaded and is still in a register, then no cost is charged.

This model complicates the rule set for *Factor* → `name` and requires a slightly more complex initial condition (in the rule for *Block* → *Assign*). It does not, however, complicate the copy rules for all the other productions. Thus, the accuracy of the model does not add significantly to the complexity of using an attribute grammar. All of the added complexity falls into the few rules that directly manipulate the model.

4.3.4 Problems with the Attribute-Grammar Approach

The preceding examples illustrate many of the computational issues that arise in using attribute grammars to perform context-sensitive computations on parse trees. Some of these pose particular problems for the use of attribute grammars in a compiler. In particular, most applications of attribute grammars in the front end of a compiler assume that the results of attribution must be preserved, typically in the form of an attributed parse tree. This section details the impact of the problems that we have seen in the preceding examples.

Handling Nonlocal Information

Some problems map cleanly onto the attribute-grammar paradigm, particularly those problems in which all information flows in the same direction. However, problems with a complex pattern of information flow can be difficult to express as attribute grammars. An attribution rule can name only

values associated with a grammar symbol that appears in the same production; this constrains the rule to using only nearby, or local, information. If the computation requires a nonlocal value, the attribute grammar must include copy rules to move those values to the points where they are used.

Copy rules can swell the size of an attribute grammar; compare Figures 4.8, 4.9, and 4.10. The implementor must write each of those rules. In the evaluator, each of the rules must be executed, creating new attributes and additional work. When information is aggregated, as in the declare-before-use rule or the framework for estimating execution times, a new copy of the information must be made each time a rule changes an aggregate's value. These copy rules add another layer of work to the tasks of writing and evaluating an attribute grammar.

Storage Management

For realistic examples, evaluation produces large numbers of attributes. The use of copy rules to move information around the parse tree can multiply the number of attribute instances that evaluation creates. If the grammar aggregates information into complex structures—to pass declaration information around the parse tree, for example—the individual attributes can be large. The evaluator must manage storage for attributes; a poor storage-management scheme can have a disproportionately large negative impact on the resource requirements of the evaluator.

If the evaluator can determine which attribute values can be used after evaluation, it may be able to reuse some of the attribute storage by reclaiming space for values that can never again be used. For example, an attribute grammar that evaluated an expression tree to a single value might return that value to the process that invoked it. In this scenario, the intermediate values calculated at interior nodes might be dead—never used again—and, thus, candidates for reclamation. On the other hand, if the tree resulting from attribution is persistent and subject to later inspection—as might be the case in an attribute grammar for type inference—then the evaluator must assume that a later phase of the compiler can traverse the tree and inspect arbitrary attributes. In this case, the evaluator cannot reclaim the storage for any of the attribute instances.

This problem reflects a fundamental clash between the functional nature of the attribute-grammar paradigm and the imperative use to which it might be put in the compiler. The possible uses of an attribute in later phases of the compiler have the effect of adding dependences from that attribute to uses not specified in the attribute grammar. This bends the functional paradigm and removes one of its strengths: the ability to automatically manage attribute storage.

Instantiating the Parse Tree

An attribute grammar specifies a computation relative to the parse tree for a valid sentence in the underlying grammar. The paradigm relies, inherently, on the availability of the parse tree. The evaluator might simulate the parse tree, but it must behave as if the parse tree exists. While the parse tree is useful for discussions of parsing, few compilers actually build a parse tree.

Some compilers use an abstract syntax tree (AST) to represent the program being compiled. The AST has the essential structure of the parse tree but eliminates many of the internal nodes that represent nonterminal symbols in the grammar (see the description starting on page 226 of Section 5.2.1). If the compiler builds an AST, it could use an attribute grammar tied to a grammar for the AST. However, if the compiler has no other use for the AST, then the programming effort and compile-time cost associated with building and maintaining the AST must be weighed against the benefits of using the attribute-grammar formalism.

Locating the Answers

One final problem with attribute-grammar schemes for context-sensitive analysis is more subtle. The result of attribute evaluation is an attributed tree. The results of the analysis are distributed over that tree, in the form of attribute values. To use these results in later passes, the compiler must traverse the tree to locate the desired information.

The compiler can use carefully constructed traversals to locate a particular node, which requires walking from the root of the parse tree down to the appropriate location—on each access. This makes the code both slower and harder to write, because the compiler must execute each of these traversals and the compiler writer must construct each of them. The alternative is to copy the important answers to a point in the tree where they are easily found, typically the root. This introduces more copy rules, exacerbating that problem.

Breakdown of the Functional Paradigm

One way to address all of these problems is to add a central repository for attributes. In this scenario, an attribute rule can record information directly into a global table, where other rules can read the information. This hybrid approach can eliminate many of the problems that arise from nonlocal information. Since the table can be accessed from any attribution rule, it has the effect of providing local access to any information already derived.

Adding a central repository for facts complicates matters in another way. If two rules communicate through a mechanism other than an attribution

rule, the implicit dependence between them is removed from the attribute dependence graph. The missing dependence should constrain the evaluator to ensure that the two rules are processed in the correct order; without it, the evaluator may be able to construct an order that, while correct for the grammar, has unintended behavior because of the removed constraint. For example, passing information between the declaration syntax and an executable expression through a table might allow the evaluator to process declarations after some or all of the expressions that use the declared variables. If the grammar uses copy rules to propagate that same information, those rules constrain the evaluator to orders that respect the dependences embodied by those copy rules.

SECTION REVIEW

Attribute grammars provide a functional specification that can be used to solve a variety of problems, including many of the problems that arise in performing context-sensitive analysis. In the attribute-grammar approach, the compiler writer produces succinct rules to describe the computation; the attribute-grammar evaluator then provides the mechanisms to perform the actual computation. A high-quality attribute-grammar system would simplify the construction of the semantic elaboration section of a compiler.

The attribute-grammar approach has never achieved widespread popularity for a number of mundane reasons. Large problems, such as the difficulty of performing nonlocal computation and the need to traverse the parse tree to discover answers to simple questions, have discouraged the adoption of these ideas. Small problems, such as space management for short-lived attributes, evaluator efficiency, and the lack of widely-available, open-source attribute-grammar evaluators have also made these tools and techniques less attractive.

Review Questions

1. From the "four function calculator" grammar given in the margin, construct an attribute-grammar scheme that attributes each *Calc* node with the specified computation, displaying the answer on each reduction to *Expr*.
2. The "define-before-use" rule specifies that each variable used in a procedure must be declared before it appears in the text. Sketch an attribute-grammar scheme for checking that a procedure conforms with this rule. Is the problem easier if the language requires that all declarations precede any executable statement?

$$
\begin{aligned}
\mathit{Calc} &\rightarrow \mathit{Expr} \\
\mathit{Expr} &\rightarrow \mathit{Expr} + \mathit{Term} \\
&\mid \mathit{Expr} - \mathit{Term} \\
&\mid \mathit{Term} \\
\mathit{Term} &\rightarrow \mathit{Term} \times \mathtt{num} \\
&\mid \mathit{Term} \div \mathtt{num} \\
&\mid \mathtt{num}
\end{aligned}
$$

Four Function Calculator

4.4 AD HOC SYNTAX-DIRECTED TRANSLATION

The rule-based evaluators for attribute grammars introduce a powerful idea that serves as the basis for the ad hoc techniques used for context-sensitive analysis in many compilers. In the rule-based evaluators, the compiler writer specifies a sequence of actions that are associated with productions in the grammar. The underlying observation, that the actions required for context-sensitive analysis can be organized around the structure of the grammar, leads to a powerful, albeit ad hoc, approach to incorporating this kind of analysis into the process of parsing a context-free grammar. We refer to this approach as ad hoc syntax-directed translation.

In this scheme, the compiler writer provides snippets of code that execute at parse time. Each snippet, or *action*, is directly tied to a production in the grammar. Each time the parser recognizes that it is at a particular place in the grammar, the corresponding action is invoked to perform its task. To implement this in a top-down, recursive-descent parser, the compiler writer simply adds the appropriate code to the parsing routines. The compiler writer has complete control over when the actions execute. In a bottom-up, shift-reduce parser, the actions are performed each time the parser performs a reduce action. This is more restrictive, but still workable.

To make this concrete, consider reformulating the signed binary number example in an ad hoc syntax-directed translation framework. Figure 4.11 shows one such framework. Each grammar symbol has a single value associated with it, denoted `val` in the code snippets. The code snippet for each rule defines the value associated with the symbol on the rule's left-hand side. Rule 1 simply multiplies the value for *Sign* with the value for *List*. Rules 2 and 3 set the value for *Sign* appropriately, just as rules 6 and 7 set the value for each instance of *Bit*. Rule 4 simply copies the value from *Bit* to *List*. The real work occurs in rule 5, which multiplies the accumulated value of the leading bits (in *List*.`val`) by two, and then adds in the next bit.

So far, this looks quite similar to an attribute grammar. However, it has two key simplifications. Values flow in only one direction, from leaves to root. It allows only a single value per grammar symbol. Even so, the scheme in Figure 4.11 correctly computes the value of the signed binary number. It leaves that value at the root of the tree, just like the attribute grammar for signed binary numbers.

These two simplifications make possible an evaluation method that works well with a bottom-up parser, such as the LR(1) parsers described in Chapter 3. Since each code snippet is associated with the right-hand side of a specific production, the parser can invoke the action each time it reduces by

	Production			Code Snippet
1	*Number*	→	*Sign List*	*Number*.val ← *Sign*.val × *List*.val
2	*Sign*	→	+	*Sign*.val ← 1
3	*Sign*	→	-	*Sign*.val ← -1
4	*List*	→	*Bit*	*List*.val ← *Bit*.val
5	$List_0$	→	$List_1$ *Bit*	$List_0$.val ← 2 × $List_1$.val + *Bit*.val
6	*Bit*	→	0	*Bit*.val ← 0
7	*Bit*	→	1	*Bit*.val ← 1

■ **FIGURE 4.11** Ad Hoc Syntax-Directed Translation for Signed Binary Numbers.

that production. This requires minor modifications to the reduce action in the skeleton LR(1) parser shown in Figure 3.15.

```
else if Action[s,word] = "reduce A→β" then
    invoke the appropriate reduce action
    pop 2 × |β| symbols
    s ← top of stack
    push A
    push Goto[s,A]
```

The parser generator can gather the syntax-directed actions together, embed them in a case statement that switches on the number of the production being reduced, and place the case statement just before it pops the right-hand side from the stack.

The translation scheme shown in Figure 4.11 is simpler than the scheme used to explain attribute grammars. Of course, we can write an attribute grammar that applies the same strategy. It would use only synthesized attributes. It would have fewer attribution rules and fewer attributes than the one shown in Figure 4.5. We chose the more complex attribution scheme to illustrate the use of both synthesized and inherited attributes.

4.4.1 Implementing Ad Hoc Syntax-Directed Translation

To make ad hoc syntax-directed translation work, the parser must include mechanisms to pass values from their definitions in one action to their uses in another, to provide convenient and consistent naming, and to allow for actions that execute at other points in the parse. This section describes mechanisms for handling these issues in a bottom-up, shift-reduce parser.

Analogous ideas will work for top-down parsers. We adopt a notation introduced in the Yacc system, an early and popular LALR(1) parser generator distributed with the Unix operating system. The Yacc notation has been adopted by many subsequent systems.

Communicating between Actions

To pass values between actions, the parser must have a methodology for allocating space to hold the values produced by the various actions. The mechanism must make it possible for an action that uses a value to find it. An attribute grammar associates the values (attributes) with nodes in the parse tree; tying the attribute storage to the tree nodes' storage makes it possible to find attribute values in a systematic way. In ad hoc syntax-directed translation, the parser may not construct the parse tree. Instead, the parser can integrate the storage for values into its own mechanism for tracking the state of the parse—its internal stack.

Recall that the skeleton LR(1) parser stored two values on the stack for each grammar symbol: the symbol and a corresponding state. When it recognizes a handle, such as a *List Bit* sequence to match the right-hand side of rule 5, the first pair on the stack represents the *Bit*. Underneath that lies the pair representing the *List*. We can replace these ⟨*symbol, state*⟩ pairs with triples, ⟨*value, symbol, state*⟩. This provides a single value attribute per grammar symbol—precisely what the simplified scheme needs. To manage the stack, the parser pushes and pops more values. On a reduction by $A \rightarrow \beta$, it pops $3 \times |\beta|$ items from the stack, rather than $2 \times |\beta|$ items. It pushes the value along with the symbol and state.

This approach stores the values at easily computed locations relative to the top of the stack. Each reduction pushes its result onto the stack as part of the triple that represents the left-hand side. The action reads the values for the right-hand side from their relative positions in the stack; the i^{th} symbol on the right-hand side has its value in the i^{th} triple from the top of the stack. Values are restricted to a fixed size; in practice, this limitation means that more complex values are passed using pointers to structures.

To save storage, the parser could omit the actual grammar symbols from the stack. The information necessary for parsing is encoded in the state. This shrinks the stack and speeds up the parse by eliminating the operations that stack and unstack those symbols. On the other hand, the grammar symbol can help in error reporting and in debugging the parser. This tradeoff is usually decided in favor of not modifying the parser that the tools produce—such modifications must be reapplied each time the parser is regenerated.

Naming Values

To simplify the use of stack-based values, the compiler writer needs a notation for naming them. Yacc introduced a concise notation to address this problem. The symbol $$ refers to the result location for the current production. Thus, the assignment `$$ = 0;` would push the integer value zero as the result corresponding to the current reduction. This assignment could implement the action for rule 6 in Figure 4.11. For the right-hand side, the symbols `$1`, `$2`, ..., `$n` refer to the locations for the first, second, through n^{th} symbols in the right-hand side, respectively.

Rewriting the example from Figure 4.11 in this notation produces the following specification:

	Production	Code Snippet
1	*Number* → *Sign List*	$\$\$ \leftarrow \$1 \times \$2$
2	*Sign* → +	$\$\$ \leftarrow 1$
3	*Sign* → -	$\$\$ \leftarrow -1$
4	*List* → *Bit*	$\$\$ \leftarrow \$1$
5	$List_0$ → $List_1$ *Bit*	$\$\$ \leftarrow 2 \times \$1 + \$2$
6	*Bit* → 0	$\$\$ \leftarrow 0$
7	*Bit* → 1	$\$\$ \leftarrow 1$

Notice how compact the code snippets are. This scheme has an efficient implementation; the symbols translate directly into offsets from the top of the stack. The notation $\$1$ indicates a location $3 \times |\beta|$ slots below the top of the stack, while a reference to $\$i$ designates the location $3 \times (|\beta| - i + 1)$ slots from the top of the stack. Thus, the positional notation allows the action snippets to read and write the stack locations directly.

Actions at Other Points in the Parse

Compiler writers might also need to perform an action in the middle of a production or on a shift action. To accomplish this, compiler writers can transform the grammar so that it performs a reduction at each point where an action is needed. To reduce in the middle of a production, they can break the production into two pieces around the point where the action should execute. A higher-level production that sequences the first part, then the second part, is added. When the first part reduces, the parser invokes the action. To force actions on shifts, a compiler writer can either move them into the scanner or add a production to hold the action. For example, to perform an action

whenever the parser shifts the terminal symbol *Bit*, a compiler writer can add a production

ShiftedBit → *Bit*

and replace every occurrence of *Bit* with *ShiftedBit*. This adds an extra reduction for every terminal symbol. Thus, the additional cost is directly proportional to the number of terminal symbols in the program.

4.4.2 Examples

To understand how ad hoc syntax-directed translation works, consider rewriting the execution-time estimator using this approach. The primary drawback of the attribute-grammar solution lies in the proliferation of rules to copy information around the tree. This creates many additional rules in the specification and duplicates attribute values at many nodes.

To address these problems in an ad hoc syntax-directed translation scheme, the compiler writer typically introduces a central repository for information about variables, as suggested earlier. This eliminates the need to copy values around the trees. It also simplifies the handling of inherited values. Since the parser determines evaluation order, we do not need to worry about breaking dependences between attributes.

Most compilers build and use such a repository, called a *symbol table*. The symbol table maps a name into a variety of annotations such as a type, the size of its runtime representation, and the information needed to generate a runtime address. The table may also store a number of type-dependent fields, such as the type signature of a function or the number of dimensions and their bounds for an array. Section 5.5 and Appendix B.4 delve into symbol-table design more deeply.

Load Tracking, Revisited

Consider, again, the problem of tracking `load` operations that arose as part of estimating execution costs. Most of the complexity in the attribute grammar for this problem arose from the need to pass information around the tree. In an ad hoc syntax-directed translation scheme that uses a symbol table, the problem is easy to handle. The compiler writer can set aside a field in the table to hold a boolean that indicates whether or not that identifier has already been charged for a `load`. The field is initially set to *`false`*. The critical code is associated with the production *Factor* → `name`. If the `name`'s symbol table entry indicates that it has not been charged for a `load`, then cost is updated and the field is set to *`true`*.

Production			Syntax-Directed Actions
$Block_0$	→	$Block_1$ *Assign*	
	|	*Assign*	
Assign	→	`name = Expr ;`	{ *cost* = *cost* + *Cost*(store) }
Expr	→	*Expr* + *Term*	{ *cost* = *cost* + *Cost*(add) }
	|	*Expr* − *Term*	{ *cost* = *cost* + *Cost*(sub) }
	|	*Term*	
Term	→	*Term* × *Factor*	{ *cost* = *cost* + *Cost*(mult) }
	|	*Term* ÷ *Factor*	{ *cost* = *cost* + *Cost*(div) }
	|	*Factor*	
Factor	→	(*Expr*)	
	|	`num`	{ *cost* = *cost* + *Cost*(loadI) }
	|	`name`	{ *if* name's *symbol table field indicates that it has not been loaded then* *cost* = *cost* + *Cost*(load) *set the field to true* }

■ **FIGURE 4.12** Tracking Loads with Ad Hoc Syntax-Directed Translation.

Figure 4.12 shows this case, along with all the other actions. Because the actions can contain arbitrary code, the compiler can accumulate *cost* in a single variable, rather than creating a *cost* attribute at each node in the parse tree. This scheme requires fewer actions than the attribution rules for the simplest execution model, even though it can provide the accuracy of the more complex model.

Notice that several productions have no actions. The remaining actions are simple, except for the action taken on a reduction by `name`. All of the complication introduced by tracking loads falls into that single action; contrast that with the attribute-grammar version, where the task of passing around the *Before* and *After* sets came to dominate the specification. The ad hoc version is cleaner and simpler, in part because the problem fits nicely into the evaluation order dictated by the reduce actions in a shift-reduce parser. Of course, the compiler writer must implement the symbol table or import it from some library of data-structure implementations.

Clearly, some of these strategies could also be applied in an attribute-grammar framework. However, they violate the functional nature of the attribute grammar. They force critical parts of the work out of the attribute-grammar framework and into an ad hoc setting.

The scheme in Figure 4.12 ignores one critical issue: initializing `cost`. The grammar, as written, contains no production that can appropriately initialize `cost` to zero. The solution, as described earlier, is to modify the grammar in a way that creates a place for the initialization. An initial production, such as *Start* → *CostInit Block*, along with *CostInit* → ϵ, does this. The framework can perform the assignment `cost` ← `0` on the reduction from ϵ to *CostInit*.

Type Inference for Expressions, Revisited

The problem of inferring types for expressions fit well into the attribute-grammar framework, as long as we assumed that leaf nodes already had type information. The simplicity of the solution shown in Figure 4.7 derives from two principal facts. First, because expression types are defined recursively on the expression tree, the natural flow of information runs bottom up from the leaves to the root. This biases the solution toward an *S*-attributed grammar. Second, expression types are defined in terms of the syntax of the source language. This fits well with the attribute-grammar framework, which implicitly requires the presence of a parse tree. All the type information can be tied to instances of grammar symbols, which correspond precisely to nodes in the parse tree.

We can reformulate this problem in an ad hoc framework, as shown in Figure 4.13. It uses the type inference functions introduced with Figure 4.7. The resulting framework looks similar to the attribute grammar for the same purpose from Figure 4.7. The ad hoc framework provides no real advantage for this problem.

Production			Syntax-Directed Actions
Expr	→	*Expr* − *Term*	{ $$ ← $\mathcal{F}_+$($1,$3) }
	|	*Expr* − *Term*	{ $$ ← $\mathcal{F}_-$($1,$3) }
	|	*Term*	{ $$ ← $1 }
Term	→	*Term* × *Factor*	{ $$ ← $\mathcal{F}_\times$($1,$3) }
	|	*Term* ÷ *Factor*	{ $$ ← $\mathcal{F}_\div$($1,$3) }
	|	*Factor*	{ $$ ← $1 }
Factor	→	(*Expr*)	{ $$ ← $2 }
	|	num	{ $$ ← *type of the* num }
	|	name	{ $$ ← *type of the* name }

■ **FIGURE 4.13** Ad Hoc Framework for Inferring Expression Types.

Production			Syntax-Directed Actions
Expr	→	*Expr* + *Term*	{ $$ ← MakeNode$_2$ (plus, $1, $3); $$.type ← $\mathcal{F}_+$($1.type, $3.type) }
	|	*Expr* − *Term*	{ $$ ← MakeNode$_2$(minus, $1, $3); $$.type ← $\mathcal{F}_-$($1.type, $3.type) }
	|	*Term*	{ $$ ← $1 }
Term	→	*Term* × *Factor*	{ $$ ← MakeNode$_2$(times, $1, $3); $$.type ← $\mathcal{F}_\times$($1.type, $3.type) }
	|	*Term* ÷ *Factor*	{ $$ ← MakeNode$_2$(divide, $1, $3); $$.type ← $\mathcal{F}_\div$($1.type, $3.type) }
	|	*Factor*	{ $$ ← $1 }
Factor	→	(*Expr*)	{ $$ ← $2 }
	|	num	{ $$ ← MakeNode$_0$(number); $$.text ← *scanned text;* $$.type ← *type of the number* }
	|	name	{ $$ ← MakeNode$_0$(identifier); $$.text ← *scanned text;* $$.type ← *type of the identifier* }

■ **FIGURE 4.14** Building an Abstract Syntax Tree and Inferring Expression Types.

Building an Abstract Syntax Tree

Compiler front ends must build an intermediate representation of the program for use in the compiler's middle part and its back end. Abstract syntax trees are a common form of tree-structured IR. The task of building an AST fits neatly into an ad hoc syntax-directed translation scheme.

Assume that the compiler has a series of routines named MakeNode$_i$, for $0 \le i \le 3$. The routine takes, as its first argument, a constant that uniquely identifies the grammar symbol that the new node will represent. The remaining i arguments are the nodes that head each of the i subtrees. Thus, MakeNode$_0$ (number) constructs a leaf node and marks it as representing a num. Similarly,

The MakeNode routines can implement the tree in any appropriate way. For example, they might map the structure onto a binary tree, as discussed in Section B.3.1.

```
MakeNode2(Plus,MakeNode0(number,) MakeNode0(number))
```

builds an AST rooted in a node for plus with two children, each of which is a leaf node for num.

To build an abstract syntax tree, the ad hoc syntax-directed translation scheme follows two general principles:

1. For an operator, it creates a node with a child for each operand. Thus, `2 + 3` creates a binary node for + with the nodes for `2` and `3` as children.
2. For a useless production, such as *Term* → *Factor*, it reuses the result from the *Factor* action as its own result.

In this manner, it avoids building tree nodes that represent syntactic variables, such as *Factor*, *Term*, and *Expr*. Figure 4.14 shows a syntax-directed translation scheme that incorporates these ideas.

Generating ILOC *for Expressions*

As a final example of manipulating expressions, consider an ad hoc framework that generates ILOC rather than an AST. We will make several simplifying assumptions. The example limits its attention to integers; handling other types adds complexity, but little insight. The example also assumes that all values can be held in registers—both that the values fit in registers and that the ILOC implementation provides more registers than the computation will use.

Code generation requires the compiler to track many small details. To abstract away most of these bookkeeping details (and to defer some deeper issues to following chapters), the example framework uses four supporting routines.

1. *Address* takes a variable name as its argument. It returns the number of a register that contains the value specified by `name`. If necessary, it generates code to load that value.
2. *Emit* handles the details of creating a concrete representation for the various ILOC operations. It might format and print them to a file. Alternatively, it might build an internal representation for later use.
3. *NextRegister* returns a new register number. A simple implementation could increment a global counter.
4. *Value* takes a number as its argument and returns a register number. It ensures that the register contains the number passed as its argument. If necessary, it generates code to move that number into the register.

Figure 4.15 shows the syntax-directed framework for this problem. The actions communicate by passing register names in the parsing stack. The actions pass these names to `Emit` as needed, to create the operations that implement the input expression.

Production	Syntax-Directed Actions
Expr → *Expr* + *Term*	{ *$$* ← *NextRegister*; *Emit*(*add*, *$1*, *$3*, *$$*) }
| *Expr* − *Term*	{ *$$* ← *NextRegister*; *Emit*(*sub*, *$1*, *$3*, *$$*) }
| *Term*	{ *$$* ← *$1* }
Term → *Term* × *Factor*	{ *$$* ← *NextRegister*; *Emit*(*mult*, *$1*, *$3*, *$$*) }
| *Term* ÷ *Factor*	{ *$$* ← *NextRegister*; *Emit*(*div*, *$1*, *$3*, *$$*) }
| *Factor*	{ *$$* ← *$1* }
Factor → (*Expr*)	{ *$$* ← *$2* }
| num	{ *$$* ← *Value*(*scanned text*); }
| name	{ *$$* ← *Address*(*scanned text*); }

■ **FIGURE 4.15** Emitting ILOC for Expressions.

Processing Declarations

Of course, the compiler writer can use syntax-directed actions to fill in much of the information that resides in the symbol table. For example, the grammar fragment shown in Figure 4.16 describes a limited subset of the syntax for declaring variables in C. (It omits `typedefs`, `structs`, `unions`, the type qualifiers `const`, `restrict`, and `volatile`, as well as the details of the initialization syntax. It also leaves several nonterminals unelaborated.) Consider the actions required to build symbol-table entries for each declared variable. Each *Declaration* begins with a set of one or more qualifiers that specify the variable's type and storage class. These qualifiers are followed by a list of one or more variable names; each variable name can include specifications about indirection (one or more occurrences of `*`), about array dimensions, and about initial values for the variable.

For example, the *StorageClass* production allows the programmer to specify information about the lifetime of a variable's value; an `auto` variable has a lifetime that matches the lifetime of the block that declares it, while `static` variables have lifetimes that span the program's entire execution. The `register` specifier suggests to the compiler that the value should be kept in a location that can be accessed quickly—historically, a hardware register. The `extern` specifier tells the compiler that declarations of the same name in different compilation units are to be linked as a single object.

DeclarationList	→	*DeclarationList Declaration*
	|	*Declaration*
Declaration	→	*SpecifierList InitDeclaratorList* ;
SpecifierList	→	*Specifier SpecifierList*
	|	*Specifier*
Specifier	→	*StorageClass*
	|	*TypeSpecifier*
StorageClass	→	`auto`
	|	`static`
	|	`extern`
	|	`register`
TypeSpecifier	→	`void`
	|	`char`
	|	`short`
	|	`int`
	|	`long`
	|	`signed`
	|	`unsigned`
	|	`float`
	|	`double`
InitDeclaratorList	→	*InitDeclaratorList* , *InitDeclarator*
	|	*InitDeclarator*
InitDeclarator	→	*Declarator* = *Initializer*
	|	*Declarator*
Declarator	→	*Pointer DirectDeclarator*
	|	*DirectDeclarator*
Pointer	→	*
	|	* *Pointer*
DirectDeclarator	→	`ident`
	|	(*Declarator*)
	|	*DirectDeclarator* ( )
	|	*DirectDeclarator* (*ParameterTypeList*)
	|	*DirectDeclarator* (*IdentifierList*)
	|	*DirectDeclarator* [ ]
	|	*DirectDeclarator* [*ConstantExpr*]

■ **FIGURE 4.16** A Subset of C's Declaration Syntax.

While such restrictions can be encoded in the grammar, the standard writers chose to leave it for semantic elaboration to check, rather than complicate an already large grammar.

The compiler must ensure that each declared name has at most one storage class attribute. The grammar places the specifiers before a list of one or more names. The compiler can record the specifiers as it processes them and apply them to the names when it later encounters them. The grammar admits an arbitrary number of *StorageClass* and *TypeSpecifier* keywords; the standard limits the ways that the actual keywords can be combined. For example, it allows only one *StorageClass* per declaration. The compiler must enforce

WHAT ABOUT CONTEXT-SENSITIVE GRAMMARS?

Given the progression of ideas from the previous chapters, it might seem natural to consider the use of context-sensitive languages to perform context-sensitive checks, such as type inference. After all, we used regular languages to perform lexical analysis and context-free languages to perform syntax analysis. A natural progression might suggest the study of context-sensitive languages and their grammars. Context-sensitive grammars can express a larger family of languages than can context-free grammars.

However, context-sensitive grammars are not the right answer for two distinct reasons. First, the problem of parsing a context-sensitive grammar is P-Space complete. Thus, a compiler that used such a technique could run *very* slowly. Second, many of the important questions are difficult, if not impossible, to encode in a context-sensitive grammar. For example, consider the issue of declaration before use. To write this rule into a context-sensitive grammar would require the grammar to encode each distinct combination of declared variables. With a sufficiently small name space (for example, Dartmouth BASIC limited the programmer to single-letter names, with an optional single digit), this might be manageable; in a modern language with a large name space, the set of names is too large to encode in a context-sensitive grammar.

this restriction through context-sensitive checking. Similar restrictions apply to *TypeSpecifier*s. For example, `short` is legal with `int` but not with `float`.

To process declarations, the compiler must collect the attributes from the qualifiers, add any indirection, dimension, or initialization attributes, and enter the variable in the table. The compiler writer might set up a properties structure whose fields correspond to the properties of a symbol-table entry. At the end of a *Declaration*, it can initialize the values of each field in the structure. As it reduces the various productions in the declaration syntax, it can adjust the values in the structure accordingly.

- On a reduction of `auto` to *StorageClass*, it can check that the field for storage class has not already been set, and then set it to `auto`. Similar actions for `static`, `extern`, and `register` complete the handling of those properties of a name.
- The type specifier productions will set other fields in the structure. They must include checks to insure that only valid combinations occur.
- Reduction from `ident` to *DirectDeclarator* should trigger an action that creates a new symbol-table entry for the name and copies the current settings from the properties structure into that entry.

- Reducing by the production

  $$InitDeclaratorList \rightarrow InitDeclaratorList\ ,\ InitDeclarator$$

  can reset the properties fields that relate to the specific name, including those set by the *Pointer*, *Initializer*, and *DirectDeclarator* productions.

By coordinating a series of actions across the productions in the declaration syntax, the compiler writer can arrange to have the properties structure contain the appropriate settings each time a name is processed.

When the parser finishes building the *DeclarationList*, it has built a symbol-table entry for each variable declared in the current scope. At that point, it may need to perform some housekeeping chores, such as assigning storage locations to declared variables. This can be done in an action for the production that reduces the *DeclarationList*. If necessary, that production can be split to create a convenient point for the action.

SECTION REVIEW

The introduction of parser generators created the need for a mechanism to tie context-sensitive actions to the parse-time behavior of the compiler. Ad hoc syntax-directed translation, as described in this section, evolved to fill that need. It uses some of the same intuitions as the attribute-grammar approach. It allows only one evaluation order. It has a limited name space for use in the code snippets that form semantic actions.

Despite these limitations, the power of allowing arbitrary code in semantic actions, coupled with support for this technique in widely used parser generators, has led to widespread use of ad hoc syntax-directed translation. It works well in conjunction with global data structures, such as a symbol table, to perform nonlocal communication. It efficiently and effectively solves a class of problems that arise in building a compiler's front end.

Calc → *Expr*

Expr → *Expr* + *Term*
| *Expr* − *Term*
| *Term*

Term → *Term* × num
| *Term* ÷ num
| num

Four function calculator

Hint: Recall that an attribute grammar does not specify order of evaluation.

Review Questions

1. Consider the problem of adding ad hoc actions to an LL(1) parser generator. How would you modify the LL(1) skeleton parser to include user-defined actions for each production?
2. In review question 1 for Section 4.3, you built an attribute-grammar framework to compute values in the "four function calculator" grammar. Now, consider implementing a calculator widget for the desktop on your personal computer. Contrast the utility of your attribute grammar and your ad hoc syntax-directed translation scheme for the calculator implementation.

4.5 ADVANCED TOPICS

This chapter has introduced the basic notions of type theory and used them as one motivating example for both attribute-grammar frameworks and for ad hoc syntax-directed translation. A deeper treatment of type theory and its applications could easily fill an entire volume.

The first subsection lays out some language design issues that affect the way that a compiler must perform type inference and type checking. The second subsection looks at a problem that arises in practice: rearranging a computation during the process of building the intermediate representation for it.

4.5.1 Harder Problems in Type Inference

Strongly typed, statically checked languages can help the programmer produce valid programs by detecting large classes of erroneous programs. The same features that expose errors can improve the compiler's ability to generate efficient code for a program by eliminating runtime checks and exposing where the compiler can specialize special case code for some construct to eliminate cases that cannot occur at runtime. These facts account, in part, for the growing role of type systems in modern programming languages.

Our examples, however, have made assumptions that do not hold in all programming languages. For example, we assumed that variables and procedures are declared—the programmer writes down a concise and binding specification for each name. Varying these assumptions can radically change the nature of both the type-checking problem and the strategies that the compiler can use to implement the language.

Some programming languages either omit declarations or treat them as optional information. Scheme programs lack declarations for variables. Smalltalk programs declare classes, but an object's class is determined only when the program instantiates that object. Languages that support separate compilation—compiling procedures independently and combining them at link time to form a program—may not require declarations for independently compiled procedures.

In the absence of declarations, type checking is harder because the compiler must rely on contextual clues to determine the appropriate type for each name. For example, if `i` is used as an index for some array `a`, that might constrain `i` to have a numeric type. The language might allow only integer subscripts; alternatively, it might allow any type that can be converted to an integer.

Typing rules are specified by the language definition. The specific details of those rules determine how difficult it is to infer a type for each variable.

This, in turn, has a direct effect on the strategies that a compiler can use to implement the language.

Type-Consistent Uses and Constant Function Types

Consider a declaration-free language that requires consistent use of variables and functions. In this case, the compiler can assign each name a general type and narrow that type by examining each use of the name in context. For example, a statement such as `a` ← `b`×`3.14159` provides evidence that `a` and `b` are numbers and that `a` must have a type that allows it to hold a decimal number. If `b` also appears in contexts where an integer is expected, such as an array reference `c(b)`, then the compiler must choose between a noninteger number (for `b`×`3.14159`) and an integer (for `c(b)`). With either choice, it will need a conversion for one of the uses.

If functions have return types that are both known and constant—that is, a function `fee` always returns the same type—then the compiler can solve the type inference problem with an iterative fixed-point algorithm operating over a lattice of types.

Type-Consistent Uses and Unknown Function Types

If the type of a function varies with the function's arguments, then the problem of type inference becomes more complex. This situation arises in Scheme, for example. Scheme's library procedure `map` takes as arguments a function and a list. It returns the result of applying the function argument to each element of the list. That is, if the argument function takes type α to β, then `map` takes a list of α to a list of β. We would write its type signature as

`Map` can also handle functions with multiple arguments. To do so, it takes multiple argument lists and treats them as lists of arguments, in order.

$$\texttt{map}: (\alpha \rightarrow \beta) \times \textit{list of } \alpha \rightarrow \textit{list of } \beta$$

Since `map`'s return type depends on the types of its arguments, a property known as *parametric polymorphism*, the inference rules must include equations over the space of types. (With known, constant return types, functions return values in the space of types.) With this addition, a simple iterative fixed-point approach to type inference is not sufficient.

The classic approach to checking these more complex systems relies on unification, although clever type-system design and type representations can permit the use of simpler or more efficient techniques.

Dynamic Changes in Type

If a variable's type can change during execution, other strategies may be required to discover where type changes occur and to infer appropriate types.

In principle, a compiler can rename the variables so that each definition site corresponds to a unique name. It can then infer types for those names based on the context provided by the operation that defines each name.

To infer types successfully, such a system would need to handle points in the code where distinct definitions must merge due to the convergence of different control-flow paths, as with ϕ-functions in static single assignment form (see Sections 5.4.2 and 9.3). If the language includes parametric polymorphism, the type-inference mechanism must handle it, as well.

The classic approach to implementing a language with dynamically changing types is to fall back on interpretation. Lisp, Scheme, Smalltalk, and APL all have similar problems. The standard implementation practice for these languages involves interpreting the operators, tagging the data with their types, and checking for type errors at runtime.

In APL, the programmer can easily write a program where `a×b` multiplies integers the first time it executes and multiplies multidimensional arrays of floating-point numbers the next time. This led to a body of research on check elimination and check motion. The best APL systems avoided most of the checks that a naive interpreter would need.

4.5.2 Changing Associativity

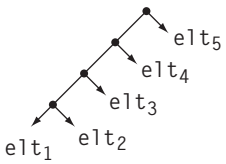
As we saw in Section 3.5.4, associativity can make a difference in numerical computation. Similarly, it can change the way that data structures are built. We can use syntax-directed actions to build representations that reflect a different associativity than the grammar would naturally produce.

In general, left-recursive grammars naturally produce left associativity, while right-recursive grammars naturally produce right associativity. To see this, consider the left-recursive and right-recursive list grammars, augmented with syntax-directed actions to build lists, shown at the top of Figure 4.17. The actions associated with each production build a list representation. Assume that *L(x,y)* is a list constructor; it can be implemented as *MakeNode*$_2$*(cons,x,y)*. The lower part of the figure shows the result of applying the two translation schemes to an input consisting of five `elt`s.

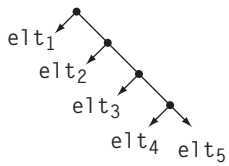
The two trees are, in many ways, equivalent. An in-order traversal of both trees visits the leaf nodes in the same order. If we add parentheses to reflect the tree structure, the left-recursive tree is $((((\texttt{elt}_1,\texttt{elt}_2),\texttt{elt}_3),\texttt{elt}_4),\texttt{elt}_5)$ while the right-recursive tree is $(\texttt{elt}_1,(\texttt{elt}_2,(\texttt{elt}_3,(\texttt{elt}_4,\texttt{elt}_5))))$. The ordering produced by left recursion corresponds to the classic left-to-right ordering for algebraic operators. The ordering produced by right recursion corresponds to the notion of a list found in Lisp and Scheme.

Production	Actions
List → *List* `elt`	{*$$* ← `L(`*$1*`,`*$2*`)`}
| `elt`	{*$$* ← *$1*}

Production	Actions
List → `elt` *List*	{*$$* ← `L(`*$1*`,`*$2*`)`}
| `elt`	{*$$* ← *$1*}



Left Recursion



Right Recursion

■ **FIGURE 4.17** Recursion versus Associativity.

Sometimes, it is convenient to use different directions for recursion and associativity. To build the right-recursive tree from the left-recursive grammar, we could use a constructor that adds successive elements to the end of the list. A straightforward implementation of this idea would have to walk the list on each reduction, making the constructor itself take $\mathbf{O}(n^2)$ time, where n is the length of the list. To avoid this overhead, the compiler can create a list header node that contains pointers to both the first and last nodes in the list. This introduces an extra node to the list. If the system constructs many short lists, the overhead may be a problem.

A solution that we find particularly appealing is to use a list header node during construction and discard it after the list has been built. Rewriting the grammar to use an ϵ-production makes this particularly clean.

Grammar	Actions
List → ϵ	{ *$$* ← *MakeListHeader ( )* }
| *List* `elt`	{ *$$* ← *AddToEnd(*$1*,*$2*)* }
Quux → *List*	{ *$$* ← *RemoveListHeader(*$1*)* }

A reduction with the ϵ-production creates the temporary list header node; with a shift-reduce parser, this reduction occurs first. The *List* → *List* `elt` production invokes a constructor that relies on the presence of the temporary header node. When *List* is reduced on the right-hand side of any other production, the corresponding action invokes a function that discards the temporary header and returns the first element of the list.

This approach lets the parser reverse the associativity at the cost of a small constant overhead in both space and time. It requires one more reduction per list, for the ϵ-production. The revised grammar admits an empty list, while

the original grammar did not. To remedy this problem, `RemoveListHeader` can explicitly check for the empty case and report the error.

4.6 SUMMARY AND PERSPECTIVE

In Chapters 2 and 3, we saw that much of the work in a compiler's front end can be automated. Regular expressions work well for lexical analysis. Context-free grammars work well for syntax analysis. In this chapter, we examined two ways to perform context-sensitive analysis: attribute-grammar formalism and an ad hoc approach. For context-sensitive analysis, unlike scanning and parsing, formalism has not displaced the ad hoc approach.

The formal approach, using attribute grammars, offers the hope of writing high-level specifications that produce reasonably efficient executables. While attribute grammars are not the solution to every problem in context-sensitive analysis, they have found application in several domains, ranging from theorem provers to program analysis. For problems in which the attribute flow is mostly local, attribute grammars work well. Problems that can be formulated entirely in terms of one kind of attribute, either inherited or synthesized, often produce clean, intuitive solutions when cast as attribute grammars. When the problem of directing the flow of attributes around the tree with copy rules comes to dominate the grammar, it is probably time to step outside the functional paradigm of attribute grammars and introduce a central repository for facts.

The ad hoc technique, syntax-directed translation, integrates arbitrary snippets of code into the parser and lets the parser sequence the actions and pass values between them. This approach has been widely embraced because of its flexibility and its inclusion in most parser-generator systems. The ad hoc approach sidesteps the practical problems that arise from nonlocal attribute flow and from the need to manage attribute storage. Values flow in one direction alongside the parser's internal representation of its state (synthesized values for bottom-up parsers and inherited for top-down parsers). These schemes use global data structures to pass information in the other direction and to handle nonlocal attribute flow.

In practice, the compiler writer often tries to solve several problems at once, such as building an intermediate representation, inferring types, and assigning storage locations. This tends to create significant attribute flows in both directions, pushing the implementor toward an ad hoc solution that uses some central repository for facts, such as a symbol table. The justification for solving many problems in one pass is usually compile-time efficiency. However, solving the problems in separate passes can

often produce solutions that are easier to understand, to implement, and to maintain.

This chapter introduced the ideas behind type systems as an example of the kind of context-sensitive analysis that a compiler must perform. The study of type theory and type-system design is a significant scholarly activity with a deep literature of its own. This chapter scratched the surface of type inference and type checking, but a deeper treatment of these issues is beyond the scope of this text. In practice, the compiler writer needs to study the type system of the source language thoroughly and to engineer the implementation of type inference and type checking carefully. The pointers in this chapter are a start, but a realistic implementation requires more study.

CHAPTER NOTES

Type systems have been an integral part of programming languages since the original FORTRAN compiler. While the first type systems reflected the resources of the underlying machine, deeper levels of abstraction soon appeared in type systems for languages such as Algol 68 and Simula 67. The theory of type systems has been actively studied for decades, producing a string of languages that embodied important principles. These include Russell [45] (parametric polymorphism), CLU [248] (abstract data types), Smalltalk [162] (subtyping through inheritance), and ML [265] (thorough and complete treatment of types as first-class objects). Cardelli has written an excellent overview of type systems [69]. The APL community produced a series of classic papers that dealt with techniques to eliminate runtime checks [1, 35, 264, 349].

Attribute grammars, like many ideas in computer science, were first proposed by Knuth [229, 230]. The literature on attribute grammars has focused on evaluators [203, 342], on circularity testing [342], and on applications of attribute grammars [157, 298]. Attribute grammars have served as the basis for several successful systems, including Intel's Pascal compiler for the 80286 [142, 143], the Cornell Program Synthesizer [297] and the Synthesizer Generator [198, 299].

Ad hoc syntax-directed translation has always been a part of the development of real parsers. Irons described the basic ideas behind syntax-directed translation to separate a parser's actions from the description of its syntax [202]. Undoubtedly, the same basic ideas were used in hand-coded precedence parsers. The style of writing syntax-directed actions that we describe was introduced by Johnson in Yacc [205]. The same notation has been carried forward into more recent systems, including `bison` from the Gnu project.

EXERCISES

1. In Scheme, the + operator is overloaded. Given that Scheme is dynamically typed, describe a method to type check an operation of the form `(+ a b)` where `a` and `b` may be of any type that is valid for the + operator. Section 4.2

2. Some languages, such as APL or PHP, neither require variable declarations nor enforce consistency between assignments to the same variable. (A program can assign the integer 10 to × and later assign the string value "book" to × in the same scope.) This style of programming is sometimes called *type juggling*.

   Suppose that you have an existing implementation of a language that has no declarations but requires type-consistent uses. How could you modify it to allow type juggling?

3. Based on the following evaluation rules, draw an annotated parse tree that shows how the syntax tree for `a - (b + c)` is constructed. Section 4.3

Production	Evaluation Rules
$E_0 \rightarrow E_1 + T$	{ E_0.*nptr* ← *mknode*(+, E_1.*nptr*, T.*nptr*) }
$E_0 \rightarrow E_1 - T$	{ E_0.*nptr* ← *mknode*(-, E_1.*nptr*, T.*nptr*) }
$E_0 \rightarrow T$	{ E_0.*nptr* ← T.*nptr* }
$T \rightarrow (E)$	{ T.*nptr* ← E.*nptr* }
$T \rightarrow$ id	{ T.*nptr* ← *mkleaf*(id, id.*entry*) }

4. Use the attribute-grammar paradigm to write an interpreter for the classic expression grammar. Assume that each `name` has a `value` attribute and a `lexeme` attribute. Assume that all attributes are already defined and that all values will always have the same type.

5. Write a grammar to describe all binary numbers that are multiples of four. Add attribution rules to the grammar that will annotate the start symbol of a syntax tree with an attribute `value` that contains the decimal value of the binary number.

6. Using the grammar defined in the previous exercise, build the syntax tree for the binary number `11100`.
   a. Show all the attributes in the tree with their corresponding values.
   b. Draw the attribute dependence graph for the syntax tree and classify all attributes as being either synthesized or inherited.

Section 4.4

7. A Pascal program can declare two integer variables `a` and `b` with the syntax

```
var a, b: int
```

This declaration might be described with the following grammar:

VarDecl	→	`var` *IDList* : *TypeID*
IDList	→	*IDList*, *ID*
	|	*ID*

where *IDList* derives a comma-separated list of variable names and *TypeID* derives a valid Pascal type. You may find it necessary to rewrite the grammar.

a. Write an attribute grammar that assigns the correct data type to each declared variable.

b. Write an ad hoc syntax-directed translation scheme that assigns the correct data type to each declared variable.

c. Can either scheme operate in a single pass over the syntax tree?

8. Sometimes, the compiler writer can move an issue across the boundary between context-free and context-sensitive analysis. Consider, for example, the classic ambiguity that arises between function invocation and array references in FORTRAN 77 (and other languages). These constructs might be added to the classic expression grammar using the productions:

Factor	→	`name (` *ExprList* `)`
ExprList	→	*ExprList* , *Expr*
	|	*Expr*

Here, the only difference between a function invocation and an array reference lies in how the `name` is declared.

In previous chapters, we have discussed using cooperation between the scanner and the parser to disambiguate these constructs. Can the problem be solved during context-sensitive analysis? Which solution is preferable?

9. Sometimes, a language specification uses context-sensitive mechanisms to check properties that can be tested in a context-free way. Consider the grammar fragment in Figure 4.16 on page 208. It allows an arbitrary number of *StorageClass* specifiers when, in fact, the standard restricts a declaration to a single *StorageClass* specifier.

a. Rewrite the grammar to enforce the restriction grammatically.

b. Similarly, the language allows only a limited set of combinations of *TypeSpecifier*. `long` is allowed with either `int` or `float`; `short` is allowed only with `int`. Either `signed` or `unsigned` can appear

with any form of `int`. `signed` may also appear on `char`. Can these restrictions be written into the grammar?

Hint: The scanner returned a single token type for any of the *StorageClass* values and another token type for any of the *TypeSpecifiers*.

c. Propose an explanation for why the authors structured the grammar as they did.

d. Do your revisions to the grammar change the overall speed of the parser? In building a parser for C, would you use the grammar like the one in Figure 4.16, or would you prefer your revised grammar? Justify your answer.

10. Object-oriented languages allow operator and function overloading. In these languages, the function name is not always a unique identifier, since you can have multiple related definitions, as in

Section 4.5

```
void Show(int);
void Show(char *);
void Show(float);
```

For lookup purposes, the compiler must construct a distinct identifier for each function. Sometimes, such overloaded functions will have different return types, as well. How would you create distinct identifiers for such functions?

11. Inheritance can create problems for the implementation of object-oriented languages. When object type *A* is a parent of object type *B*, a program can assign a "pointer to *B*" to a "pointer to *A*," with syntax such as `a` ← `b`. This should not cause problems since everything that *A* can do, *B* can also do. However, one cannot assign a "pointer to *A*" to a "pointer to *B*," since object class *B* can implement methods that object class *A* does not.

Design a mechanism that can use ad hoc syntax-directed translation to determine whether or not a pointer assignment of this kind is allowed.