## 5.4 The Concept of Binding

A **binding** is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol. The time at which a binding takes place is called **binding time**. Binding and binding times are prominent concepts in the semantics of programming languages. Bindings can take place at language design time, language implementation time, compile time, load time, link time, or run time. For example, the asterisk symbol (`*`) is usually bound to the multiplication operation at language design time. A data type, such as `int` in C, is bound to a range of possible values at language implementation time. At compile time, a variable in a Java program is bound to a particular data type. A variable may be bound to a storage cell when the program is loaded into memory. That same binding does not happen until run time in some cases, as with variables declared in Java methods. A call to a library subprogram is bound to the subprogram code at link time.

Consider the following C++ assignment statement:

```
count = count + 5;
```

Some of the bindings and their binding times for the parts of this assignment statement are as follows:

- The type of `count` is bound at compile time.
- The set of possible values of `count` is bound at compiler design time.
- The meaning of the operator symbol + is bound at compile time, when the types of its operands have been determined.
- The internal representation of the literal `5` is bound at compiler design time.
- The value of `count` is bound at execution time with this statement.

A complete understanding of the binding times for the attributes of program entities is a prerequisite for understanding the semantics of a programming language. For example, to understand what a subprogram does, one must understand how the actual parameters in a call are bound to the formal parameters in its definition. To determine the current value of a variable, it may be necessary to know when the variable was bound to storage and with which statement or statements.

### 5.4.1 Binding of Attributes to Variables

A binding is **static** if it first occurs before run time begins and remains unchanged throughout program execution. If the binding first occurs during run time or can change in the course of program execution, it is called **dynamic**. The physical binding of a variable to a storage cell in a virtual memory environment is complex, because the page or segment of the address space in which the cell resides may be moved in and out of memory many times during program execution. In a sense, such variables are bound and unbound repeatedly. These

bindings, however, are maintained by computer hardware, and the changes are invisible to the program and the user. Because they are not important to the discussion, we are not concerned with these hardware bindings. The essential point is to distinguish between static and dynamic bindings.

## 5.4.2 Type Bindings

Before a variable can be referenced in a program, it must be bound to a data type. The two important aspects of this binding are how the type is specified and when the binding takes place. Types can be specified statically through some form of explicit or implicit declaration.

### 5.4.2.1 Static Type Binding

An **explicit declaration** is a statement in a program that lists variable names and specifies that they are a particular type. An **implicit declaration** is a means of associating variables with types through default conventions, rather than declaration statements. In this case, the first appearance of a variable name in a program constitutes its implicit declaration. Both explicit and implicit declarations create static bindings to types.

Most widely used programming languages that use static type binding exclusively and were designed since the mid-1960s require explicit declarations of all variables (Visual Basic and ML are two exceptions).

Implicit variable type binding is done by the language processor, either a compiler or an interpreter. There are several different bases for implicit variable type bindings. The simplest of these is naming conventions. In this case, the compiler or interpreter binds a variable to a type based on the syntactic form of the variable's name.

Although they are a minor convenience to programmers, implicit declarations can be detrimental to reliability because they prevent the compilation process from detecting some typographical and programmer errors.

Some of the problems with implicit declarations can be avoided by requiring names for specific types to begin with particular special characters. For example, in Perl any name that begins with `$` is a scalar, which can store either a string or a numeric value. If a name begins with `@`, it is an array; if it begins with a `%`, it is a hash structure.[3] This creates different namespaces for different type variables. In this scenario, the names `@apple` and `%apple` are unrelated, because each is from a different namespace. Furthermore, a program reader always knows the type of a variable when reading its name.

Another kind of implicit type declarations uses context. This is sometimes called **type inference**. In the simpler case, the context is the type of the value assigned to the variable in a declaration statement. For example, in C# a **`var`** declaration of a variable must include an initial value, whose type is taken as the type of the variable. Consider the following declarations:

3. Both arrays and hashes are considered types—both can store any scalar in their elements.

```
var sum = 0;
var total = 0.0;
var name = "Fred";
```

The types of `sum`, `total`, and `name` are **`int`**, **`float`**, and **`string`**, respectively. Keep in mind that these are statically typed variables—their types are fixed for the lifetime of the unit in which they are declared.

Visual Basic and the functional languages ML, Haskell, OCaml, and F# also use type inferencing. In these functional languages, the context of the appearance of a name is the basis for determining its type. This kind of type inferencing is discussed in detail in Chapter 15.

#### 5.4.2.2 Dynamic Type Binding

With dynamic type binding, the type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name. Instead, the variable is bound to a type when it is assigned a value in an assignment statement. When the assignment statement is executed, the variable being assigned is bound to the type of the value of the expression on the right side of the assignment. Such an assignment may also bind the variable to an address and a memory cell, because different type values may require different amounts of storage. Any variable can be assigned any type value. Furthermore, a variable's type can change any number of times during program execution. It is important to realize that the type of a variable whose type is dynamically bound may be temporary.

When the type of a variable is statically bound, the name of the variable can be thought of being bound to a type, in the sense that the type and name of a variable are simultaneously bound. However, when a variable's type is dynamically bound, its name can be thought of as being only temporarily bound to a type. In reality, the names of variables are never bound to types. Names can be bound to variables and variables can be bound to types.

Languages in which types are dynamically bound are dramatically different from those in which types are statically bound. The primary advantage of dynamic binding of variables to types is that it provides more programming flexibility. For example, a program to process numeric data in a language that uses dynamic type binding can be written as a generic program, meaning that it is capable of dealing with data of any numeric type. Whatever type data is input will be acceptable, because the variables in which the data are to be stored can be bound to the correct type when the data is assigned to the variables after input. By contrast, because of static binding of types, one cannot write a C program to process data without knowing the type of that data.

Before the mid-1990s, the most commonly used programming languages used static type binding, the primary exceptions being some functional languages such as Lisp. However, since then there has been a significant shift to languages that use dynamic type binding. In Python, Ruby, JavaScript, and PHP, type binding is dynamic. For example, a JavaScript script may contain the following statement:

```
list = [10.2, 3.5];
```

Regardless of the previous type of the variable named `list`, this assignment causes it to become the name of a single-dimensioned array of length 2. If the statement

```
list = 47;
```

followed the previous example assignment, `list` would become the name of a scalar variable.

The option of dynamic type binding was included in C# 2010. A variable can be declared to use dynamic type binding by including the **`dynamic`** reserved word in its declaration, as in the following example:

```
dynamic  any;
```

This is similar, although also different from declaring `any` to have type **`object`**. It is similar in that `any` can be assigned a value of any type, just as if it were declared **`object`**. It is different in that it is not useful for several different situations of interoperation; for example, with dynamically typed languages such as IronPython and IronRuby (.NET versions of Python and Ruby, respectively). However, it is useful when data of unknown type come into a program from an external source. Class members, properties, method parameters, method return values, and local variables can all be declared **`dynamic`**.

In pure object-oriented languages—for example, Ruby—all variables are references and do not have types; all data are objects and any variable can reference any object. Variables in such languages are, in a sense, all the same type—they are references. However, unlike the references in Java, which are restricted to referencing one specific type of value, variables in Ruby can reference any object.

There are two disadvantages to dynamic type binding. First, it causes programs to be less reliable, because the error-detection capability of the compiler is diminished relative to a compiler for a language with static type bindings. Dynamic type binding allows any variable to be assigned a value of any type. Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type. For example, suppose that in a particular JavaScript program, `i` and `x` are currently the names of scalar numeric variables and `y` is currently the name of an array. Furthermore, suppose that the program needs the assignment statement

```
i = x;
```

but because of a keying error, it has the assignment statement

```
i = y;
```

In JavaScript (or any other language that uses dynamic type binding), no error is detected in this statement by the interpreter—the type of the variable named `i` is simply changed to an array. But later uses of `i` will expect it to be a scalar, and correct results will be impossible. In a language with static type binding,

such as Java, the compiler would detect the error in the assignment `i = y`, and the program would not get to execution.

Note that this disadvantage is also present to some extent in some languages that use static type binding, such as C and C++, which in many cases automatically convert the type of the RHS of an assignment to the type of the LHS.

Perhaps the greatest disadvantage of dynamic type binding is cost. The cost of implementing dynamic attribute binding is considerable, particularly in execution time. Type checking must be done at run time. Furthermore, every variable must have a run-time descriptor associated with it to maintain the current type. The storage used for the value of a variable must be of varying size, because different type values require different amounts of storage.

Finally, languages that have dynamic type binding for variables are usually implemented using pure interpreters rather than compilers. Computers do not have instructions whose operand types are not known at compile time. Therefore, a compiler cannot build machine instructions for the expression A + B if the types of A and B are not known at compile time. Pure interpretation typically takes at least 10 times as long as it does to execute equivalent machine code. Of course, if a language is implemented with a pure interpreter, the time to do dynamic type binding is hidden by the overall time of interpretation, so it seems less costly in that environment. On the other hand, languages with static type bindings are seldom implemented by pure interpretation, because programs in these languages can be easily translated to very efficient machine code versions.

### 5.4.3 Storage Bindings and Lifetime

The fundamental character of an imperative programming language is in large part determined by the design of the storage bindings for its variables. It is therefore important to have a clear understanding of these bindings.

The memory cell to which a variable is bound somehow must be taken from a pool of available memory. This process is called **allocation**. **Deallocation** is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory.

The **lifetime** of a variable is the time during which the variable is bound to a specific memory location. So, the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell. To investigate storage bindings of variables, it is convenient to separate scalar (unstructured) variables into four categories, according to their lifetimes. These categories are named static, stack-dynamic, explicit heap-dynamic, and implicit heap-dynamic. In the following sections, we discuss the definitions of these four categories, along with their purposes, advantages, and disadvantages.

#### 5.4.3.1 Static Variables

**Static variables** are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates. Statically bound variables have several valuable

applications in programming. Globally accessible variables are often used throughout the execution of a program, thus making it necessary to have them bound to the same storage during that execution. Sometimes it is convenient to have subprograms that are history sensitive. Such a subprogram must have local static variables.

One advantage of static variables is efficiency. All addressing of static variables can be direct;[4] other kinds of variables often require indirect addressing, which is slower. Also, no run-time overhead is incurred for allocation and deallocation of static variables, although this time is often negligible.

One disadvantage of static binding to storage is reduced flexibility; in particular, a language that has only static variables cannot support recursive subprograms. Another disadvantage is that storage cannot be shared among variables. For example, suppose a program has two subprograms, both of which require large arrays. Furthermore, suppose that the two subprograms are never active at the same time. If the arrays are static, they cannot share the same storage for their arrays.

C and C++ allow programmers to include the **`static`** specifier on a variable definition in a function, making the variables it defines static. Note that when the **`static`** modifier appears in the declaration of a variable in a class definition in C++, Java, and C#, it also implies that the variable is a class variable, rather than an instance variable. Class variables are created statically some time before the class is first instantiated.

#### 5.4.3.2 Stack-Dynamic Variables

**Stack-dynamic variables** are those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound. **Elaboration** of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached. Therefore, elaboration occurs during run time. For example, the variable declarations that appear at the beginning of a Java method are elaborated when the method is called and the variables defined by those declarations are deallocated when the method completes its execution.

As their name indicates, stack-dynamic variables are allocated from the run-time stack.

Some languages—for example, C++ and Java—allow variable declarations to occur anywhere a statement can appear. In some implementations of these languages, all of the stack-dynamic variables declared in a function or method (not including those declared in nested blocks) may be bound to storage at the beginning of execution of the function or method, even though the declarations of some of these variables do not appear at the beginning. In such cases, the

4. In some implementations, static variables are addressed through a base register, making accesses to them as costly as for stack-allocated variables.

variable becomes visible at the declaration, but the storage binding (and initialization, if it is specified in the declaration) occurs when the function or method begins execution. The fact that storage binding of a variable takes place before it becomes visible does not affect the semantics of the language.

The advantages of stack-dynamic variables are as follows: To be useful, at least in most cases, recursive subprograms require some form of dynamic local storage so that each active copy of the recursive subprogram has its own version of the local variables. These needs are conveniently met by stack-dynamic variables. Even in the absence of recursion, having stack-dynamic local storage for subprograms is not without merit, because all subprograms share the same memory space for their locals.

The disadvantages, relative to static variables, of stack-dynamic variables are the run-time overhead of allocation and deallocation, possibly slower accesses because indirect addressing is required, and the fact that subprograms cannot be history sensitive. The time required to allocate and deallocate stack-dynamic variables is not significant, because all of the stack-dynamic variables that are declared at the beginning of a subprogram are allocated and deallocated together, rather than by separate operations.

In Java, C++, and C#, variables defined in methods are by default stack dynamic.

All attributes other than storage are statically bound to stack-dynamic scalar variables. That is not the case for some structured types, as is discussed in Chapter 6. Implementation of allocation/deallocation processes for stack-dynamic variables is discussed in Chapter 10.

#### 5.4.3.3 Explicit Heap-Dynamic Variables

**Explicit heap-dynamic variables** are nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer. These variables, which are allocated from and deallocated to the heap, can only be referenced through pointer or reference variables. The heap is a collection of storage cells whose organization is highly disorganized due to the unpredictability of its use. The pointer or reference variable that is used to access an explicit heap-dynamic variable is created as any other scalar variable. An explicit heap-dynamic variable is created by either an operator (for example, in C++) or a call to a system subprogram provided for that purpose (for example, in C).

In C++, the allocation operator, named **`new`**, uses a type name as its operand. When executed, an explicit heap-dynamic variable of the operand type is created and its address is returned. Because an explicit heap-dynamic variable is bound to a type at compile time, that binding is static. However, such variables are bound to storage at the time they are created, which is during run time.

In addition to a subprogram or operator for creating explicit heap-dynamic variables, some languages include a subprogram or operator for explicitly destroying them.

As an example of explicit heap-dynamic variables, consider the following C++ code segment:

```
int  *intnode;       // Create a pointer
intnode =  new int; // Create the heap-dynamic variable
. . .
delete  intnode;    // Deallocate the heap-dynamic variable
                        // to which intnode points
```

In this example, an explicit heap-dynamic variable of **int** type is created by the **new** operator. This variable can then be referenced through the pointer, `intnode`. Later, the variable is deallocated by the **delete** operator. C++ requires the explicit deallocation operator **delete**, because it does not use implicit storage reclamation, such as garbage collection.

In Java, all data except the primitive scalars are objects. Java objects are explicitly heap dynamic and are accessed through reference variables. Java has no way of explicitly destroying a heap-dynamic variable; rather, implicit garbage collection is used. Garbage collection is discussed in Chapter 6.

C# has both explicit heap-dynamic and stack-dynamic objects, all of which are implicitly deallocated. In addition, C# supports C++-style pointers. Such pointers are used to reference heap, stack, and even static variables and objects. These pointers have the same dangers as those of C++, and the objects they reference on the heap are not implicitly deallocated. Pointers are included in C# to allow C# components to interoperate with C and C++ components. To discourage their use, and also to make clear to any program reader that the code uses pointers, the header of any method that defines a pointer must include the reserved word **unsafe**.

Explicit heap-dynamic variables are often used to construct dynamic structures, such as linked lists and trees, that need to grow and/or shrink during execution. Such structures can be built conveniently using pointers or references and explicit heap-dynamic variables.

The disadvantages of explicit heap-dynamic variables are the difficulty of using pointer and reference variables correctly, the cost of references to the variables, and the complexity of the required storage management implementation. This is essentially the problem of heap management, which is costly and complicated. Implementation methods for explicit heap-dynamic variables are discussed at length in Chapter 6.

#### 5.4.3.4 Implicit Heap-Dynamic Variables

**Implicit heap-dynamic variables** are bound to heap storage only when they are assigned values. In fact, all their attributes are bound every time they are assigned. For example, consider the following JavaScript assignment statement:

```
highs = [74, 84, 86, 90, 71];
```

Regardless of whether the variable named `highs` was previously used in the program or what it was used for, it is now an array of five numeric values.

The advantage of such variables is that they have the highest degree of flexibility, allowing highly generic code to be written. One disadvantage of implicit heap-dynamic variables is the run-time overhead of maintaining all the dynamic attributes, which could include array subscript types and ranges, among others. Another disadvantage is the loss of some error detection by the compiler, as discussed in Section 5.4.2.2.

## 5.5 Scope

One of the important factors in understanding variables is scope. The **scope** of a variable is the range of statements in which the variable is visible. A variable is **visible** in a statement if it can be referenced or assigned in that statement.

The scope rules of a language determine how a particular occurrence of a name is associated with a variable, or in the case of a functional language, how a name is associated with an expression. In particular, scope rules determine how references to variables declared outside the currently executing subprogram or block are associated with their declarations and thus their attributes (blocks are discussed in Section 5.5.2). A clear understanding of these rules for a language is therefore essential to the ability to write or read programs in that language.

A variable is **local** in a program unit or block if it is declared there. The nonlocal variables of a program unit or block are those that are visible within the program unit or block but are not declared there. Global variables are a special category of nonlocal variables, which are discussed in Section 5.5.4.

Scoping issues of classes, packages, and namespaces are discussed in Chapter 11.

### 5.5.1 Static Scope

ALGOL 60 introduced the method of binding names to nonlocal variables called **static scoping**,[5] which has been copied by many subsequent imperative languages and many nonimperative languages as well. Static scoping is so named because the scope of a variable can be statically determined—that is, prior to execution. This permits a human program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.

There are two categories of static-scoped languages: those in which subprograms can be nested, which creates nested static scopes, and those in which subprograms cannot be nested. In the latter category, static scopes are also created by subprograms but nested scopes are created only by nested class definitions and blocks.

Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python allow nested subprograms, but the C-based languages do not.

5. Static scoping is sometimes called *lexical scoping*.

Our discussion of static scoping in this section focuses on those languages that allow nested subprograms. Initially, we assume that *all* scopes are associated with program units and that all referenced nonlocal variables are declared in other program units.[6] In this chapter, it is assumed that scoping is the only method of accessing nonlocal variables in the languages under discussion. This is not true for all languages. It is not even true for all languages that use static scoping, but the assumption simplifies the discussion here.

When the reader of a program finds a reference to a variable, the attributes of the variable can be determined by finding the statement in which it is declared (either explicitly or implicitly). In static-scoped languages with nested subprograms, this process can be thought of in the following way. Suppose a reference is made to a variable `x` in subprogram `sub1`. The correct declaration is found by first searching the declarations of subprogram `sub1`. If no declaration is found for the variable there, the search continues in the declarations of the subprogram that declared subprogram `sub1`, which is called its **static parent**. If a declaration of `x` is not found there, the search continues to the next-larger enclosing unit (the unit that declared `sub1`'s parent), and so forth, until a declaration for `x` is found or the largest unit's declarations have been searched without success. In that case, an undeclared variable error is reported. The static parent of subprogram `sub1`, and its static parent, and so forth up to and including the largest enclosing subprogram, are called the **static ancestors** of `sub1`. Actual implementation techniques for static scoping, which are discussed in Chapter 10, are usually much more efficient than the process just described.

Consider the following JavaScript function, `big`, in which the two functions `sub1` and `sub2` are nested:

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

Under static scoping, the reference to the variable `x` in `sub2` is to the `x` declared in the procedure `big`. This is true because the search for `x` begins in the procedure in which the reference occurs, `sub2`, but no declaration for `x` is found there. The search continues in the static parent of `sub2`, `big`, where the declaration of `x` is found. The `x` declared in `sub1` is ignored, because it is not in the static ancestry of `sub2`.

6. Nonlocal variables not defined in other program units are discussed in Section 5.5.4.

In some languages that use static scoping, regardless of whether nested subprograms are allowed, some variable declarations can be hidden from some other code segments. For example, consider again the JavaScript function `big`. The variable `x` is declared in both `big` and in `sub1`, which is nested inside `big`. Within `sub1`, every simple reference to `x` is to the local `x`. Therefore, the outer `x` is hidden from `sub1`.

### 5.5.2 Blocks

Many languages allow new static scopes to be defined in the midst of executable code. This powerful concept, introduced in ALGOL 60, allows a section of code to have its own local variables whose scope is minimized. Such variables are typically stack dynamic, so their storage is allocated when the section is entered and deallocated when the section is exited. Such a section of code is called a **block**. Blocks provide the origin of the phrase **block-structured language**.

The C-based languages allow any compound statement (a statement sequence surrounded by matched braces) to have declarations and thereby define a new scope. Such compound statements are called blocks. For example, if `list` were an integer array, one could write the following:

```
if (list[i] < list[j]) {
  int temp;
  temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

The scopes created by blocks, which could be nested in larger blocks, are treated exactly like those created by subprograms. References to variables in a block that are not declared there are connected to declarations by searching enclosing scopes (blocks or subprograms) in order of increasing size.

Consider the following skeletal C function:

```
void sub() {
  int count;
  . . .
  while (. . .) {
    int count;
    count++;
    . . .
  }
  . . .
}
```

The reference to `count` in the **`while`** loop is to that loop's local `count`. In this case, the `count` of `sub` is hidden from the code inside the **`while`** loop. In general, a declaration for a variable effectively hides any declaration of a variable with

the same name in a larger enclosing scope.[7] Note that this code is legal in C and C++ but illegal in Java and C#. The designers of Java and C# believed that the reuse of names in nested blocks was too error prone to be allowed.

Although JavaScript uses static scoping for its nested functions, nonfunction blocks cannot be defined in the language.

Most functional programming languages include a construct that is related to the blocks of the imperative languages, usually named *let*. These constructs have two parts, the first of which is to bind names to values, usually specified as expressions. The second part is an expression that uses the names defined in the first part. Programs in functional languages are comprised of expressions, rather than statements. Therefore, the final part of a `let` construct is an expression, rather than a statement. In Scheme, a `let` construct is a call to the function `LET` with the following form:

```
(LET (
   (name1 expression1)
   . . .
  (namen expressionn))
   expression
)
```

The semantics of the call to `LET` is as follows: The first *n* expressions are evaluated and the values are assigned to the associated names. Then, the final expression is evaluated and the return value of `LET` is that value. This differs from a block in an imperative language in that the names are of values; they are not variables in the imperative sense. Once set, they cannot be changed. However, they are like local variables in a block in an imperative language in that their scope is local to the call to `LET`. Consider the following call to `LET`:

```
(LET (
  (top (+ a b))
  (bottom (- c d)))
  (/ top bottom)
)
```

This call computes and returns the value of the expression `(a + b) / (c - d)`.

In ML, the form of a **`let`** construct is as follows:

```
let
 val name1 = expression1
 ...
 val namen = expressionn
in
 expression
end;
```

7. As discussed in Section 5.5.4, in C++, such hidden global variables can be accessed in the inner scope using the scope operator (`::`).

Each **val** statement binds a name to an expression. As with Scheme, the names in the first part are like the named constants of imperative languages; once set, they cannot be changed.[8] Consider the following **let** construct:

```
let
  val  top = a + b
  val bottom = c - d
in
  top / bottom
end;
```

The general form of a **let** construct in F# is as follows:

**let** left_side = expression

The left_side of **let** can be a name or a tuple pattern (a sequence of names separated by commas).

The scope of a name defined with **let** inside a function definition is from the end of the defining expression to the end of the function. The scope of **let** can be limited by indenting the following code, which creates a new local scope. Although any indentation will work, the convention is that the indentation is four spaces. Consider the following code:

```
let n1 =
    let  n2 = 7
    let  n3 = n2 + 3
    n3;;
let n4 = n3 + n1;;
```

The scope of `n1` extends over all of the code. However, the scope of `n2` and `n3` ends when the indentation ends. So, the use of `n3` in the last **let** causes an error. The last line of the **let** `n1` scope is the value bound to `n1`; it could be any expression.

Chapter 15 includes more details of the let constructs in Scheme, ML, Haskell, and F#.

### 5.5.3 Declaration Order

In C89, as well as in some other languages, all data declarations in a function except those in nested blocks must appear at the beginning of the function. However, some languages—for example, C99, C++, Java, JavaScript, and C#—allow variable declarations to appear anywhere a statement can appear in a program unit. Declarations may create scopes that are not associated with compound statements or subprograms. For example, in C99, C++, and Java, the scope of all local variables is from their declarations to the ends of the blocks in which those declarations appear.

8. In Chapter 15, we will see that they can be reset, but that the process actually creates a new name.

In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block, as long as it is not in a nested block. The same is true for methods. Recall that C# does not allow the declaration of a variable in a nested block to have the same name as a variable in a nesting scope. For example, consider the following C# code:

```
{int x;
   ...
   {int x;  // Illegal
   ...
   }
   ...
}
```

Because the scope of a declaration is the whole block, the following nested declaration of `x` is also illegal:

```
{
   {int  x;  // Illegal
   ...
   }
   int  x;
}
```

Note that C# still requires that all be declared before they are used. Therefore, although the scope of a variable extends from the declaration to the top of the block or subprogram in which that declaration appears, the variable still cannot be used above its declaration.

In JavaScript, local variables can be declared anywhere in a function, but the scope of such a variable is always the entire function. If used before its declaration in the function, such a variable has the value `undefined`. The reference is not illegal.

The **`for`** statements of C++, Java, and C# allow variable definitions in their initialization expressions. In early versions of C++, the scope of such a variable was from its definition to the end of the smallest enclosing block. In the standard version, however, the scope is restricted to the **`for`** construct, as is the case with Java and C#. Consider the following skeletal method:

```
void fun() {
   . . .
   for (int  count = 0; count < 10; count++){
      . . .
   }
   . . .
}
```

In later versions of C++, as well as in Java and C#, the scope of `count` is from the **`for`** statement to the end of its body (the right brace).

### 5.5.4 Global Scope

Some languages, including C, C++, PHP, JavaScript, and Python, allow a program structure that is a sequence of function definitions, in which variable definitions can appear outside the functions. Definitions outside functions in a file create global variables, which potentially can be visible to those functions.

C and C++ have both declarations and definitions of global data. Declarations specify types and other attributes but do not cause allocation of storage. Definitions specify attributes *and* cause storage allocation. For a specific global name, a C program can have any number of compatible declarations, but only a single definition.

A declaration of a variable outside function definitions specifies that the variable is defined in a different file. A global variable in C is implicitly visible in all subsequent functions in the file, except those that include a declaration of a local variable with the same name. A global variable that is defined after a function can be made visible in the function by declaring it to be external, as in the following:

```
extern int  sum;
```

In C99, definitions of global variables usually have initial values. Declarations of global variables never have initial values. If the declaration is outside function definitions, it need not include the **`extern`** qualifier.

This idea of declarations and definitions carries over to the functions of C and C++, where prototypes declare names and interfaces of functions but do not provide their code. Function definitions, on the other hand, are complete.

In C++, a global variable that is hidden by a local with the same name can be accessed using the scope operator (`::`). For example, if `x` is a global that is hidden in a function by a local named `x`, the global could be referenced as `::x`.

PHP statements can be interspersed with function definitions. Variables in PHP are implicitly declared when they appear in statements. Any variable that is implicitly declared outside any function is a global variable; variables implicitly declared in functions are local variables. The scope of global variables extends from their declarations to the end of the program but skips over any subsequent function definitions. So, global variables are not implicitly visible in any function. Global variables can be made visible in functions in their scope in two ways: (1) If the function includes a local variable with the same name as a global, that global can be accessed through the `$GLOBALS` array, using the name of the global as a string literal subscript, and (2) if there is no local variable in the function with the same name as the global, the global can be

made visible by including it in a `global` declaration statement. Consider the following example:

```
$day = "Monday";
$month = "January";

function calendar() {
  $day = "Tuesday";
  global $month;
  print "local day is $day  ";
  $gday = $GLOBALS['day'];
  print "global day is $gday <br \>";
  print "global month is $month  ";
}

calendar();
```

Interpretation of this code produces the following:

```
local day is Tuesday
global day is Monday
global month is January
```

The global variables of JavaScript are very similar to those of PHP, except that there is no way to access a global variable in a function that has declared a local variable with the same name.

The visibility rules for global variables in Python are unusual. Variables are not normally declared, as in PHP. They are implicitly declared when they appear as the targets of assignment statements. A global variable can be referenced in a function, but a global variable can be assigned in a function only if it has been declared to be global in the function. Consider the following examples:

```
day = "Monday"

def tester():
print "The global day is:", day

tester()
```

The output of this script, because globals can be referenced directly in functions, is as follows:

```
The global day is: Monday
```

The following script attempts to assign a new value to the global `day`:

```
day = "Monday"

def  tester():
  print "The global day is:", day
  day = "Tuesday"
  print "The new value of day is:", day

tester()
```

This script creates an `UnboundLocalError` error message, because the assignment to `day` in the second line of the body of the function makes `day` a local variable, which makes the reference to `day` in the first line of the body of the function an illegal forward reference to the local.

The assignment to `day` can be to the global variable if `day` is declared to be global at the beginning of the function. This prevents the assignment to `day` from creating a local variable. This is shown in the following script:

```
day = "Monday"

def tester():
  global  day
  print  "The global day is:", day
  day = "Tuesday"
  print  "The new value of day is:", day

tester()
```

The output of this script is as follows:

```
The global day is: Monday
The new value of day is: Tuesday
```

Functions can be nested in Python. Variables defined in nesting functions are accessible in a nested function through static scoping, but such variables must be declared **`nonlocal`** in the nested function.[9] An example skeletal program in Section 5.7 illustrates accesses to nonlocal variables.

All names defined outside function definitions in F# are globals. Their scope extends from their definitions to the end of the file.

Declaration order and global variables are also issues in the class and member declarations in object-oriented languages. These are discussed in Chapter 12.

9. The **`nonlocal`** reserved word was introduced in Python 3.

### 5.5.5 Evaluation of Static Scoping

Static scoping provides a method of nonlocal access that works well in many situations. However, it is not without its problems. First, in most cases it allows more access to both variables and subprograms than is necessary. It is simply too crude a tool for concisely specifying such restrictions. Second, and perhaps more important, is a problem related to program evolution. Software is highly dynamic—programs that are used regularly continually change. These changes often result in restructuring, thereby destroying the initial structure that restricted variable and subprogram access in a static-scoped language. To avoid the complexity of maintaining these access restrictions, developers often discard structure when it gets in the way. Thus, getting around the restrictions of static scoping can lead to program designs that bear little resemblance to the original, even in areas of the program in which changes have not been made. Designers are encouraged to use far more globals than are necessary. All subprograms can end up being nested at the same level, in the main program, using globals instead of deeper levels of nesting.[10] Moreover, the final design may be awkward and contrived, and it may not reflect the underlying conceptual design. These and other defects of static scoping are discussed in detail in Clarke, Wileden, and Wolf (1980). An alternative to the use of static scoping to control access to variables and subprograms is an encapsulation construct, which is included in many newer languages. Encapsulation constructs are discussed in Chapter 11.

### 5.5.6 Dynamic Scope

The scope of variables in APL, SNOBOL4, and the early versions of Lisp is dynamic. Perl and Common Lisp also allow variables to be declared to have dynamic scope, although the default scoping mechanism in these languages is static. **Dynamic scoping** is based on the calling sequence of subprograms, not on their spatial relationship to each other. Thus, the scope can be determined only at run time.

Consider again the function `big` from Section 5.5.1, which is reproduced here, minus the function calls:

```
function big() {
  function sub1() {
    var x = 7;
  }
  function sub2() {
    var y = x;
    var z = 3;
  }
  var x = 3;
}
```

10. Sounds like the structure of a C program, doesn't it?

Assume that dynamic-scoping rules apply to nonlocal references. The meaning of the identifier `x` referenced in `sub2` is dynamic—it cannot be determined at compile time. It may reference the variable from either declaration of `x`, depending on the calling sequence.

One way the correct meaning of `x` can be determined during execution is to begin the search with the local declarations. This is also the way the process begins with static scoping, but that is where the similarity between the two techniques ends. When the search of local declarations fails, the declarations of the dynamic parent, or calling function, are searched. If a declaration for `x` is not found there, the search continues in that function's dynamic parent, and so forth, until a declaration for `x` is found. If none is found in any dynamic ancestor, it is a run-time error.

Consider the two different call sequences for `sub2` in the earlier example. First, `big` calls `sub1`, which calls `sub2`. In this case, the search proceeds from the local procedure, `sub2`, to its caller, `sub1`, where a declaration for `x` is found. So, the reference to `x` in `sub2` in this case is to the `x` declared in `sub1`. Next, `sub2` is called directly from `big`. In this case, the dynamic parent of `sub2` is `big`, and the reference is to the `x` declared in `big`.

Note that if static scoping were used, in either calling sequence discussed, the reference to `x` in `sub2` would be to `big`'s `x`.

Perl's dynamic scoping is unusual—in fact, it is not exactly like that discussed in this section, although the semantics are often that of traditional dynamic scoping (see Programming Exercise 1).

### 5.5.7 Evaluation of Dynamic Scoping

The effect of dynamic scoping on programming is profound. When dynamic scoping is used, the correct attributes of nonlocal variables visible to a program statement cannot be determined statically. Furthermore, a reference to the name of such a variable is not always to the same variable. A statement in a subprogram that contains a reference to a nonlocal variable can refer to different nonlocal variables during different executions of the subprogam. Several kinds of programming problems follow directly from dynamic scoping.

First, during the time span beginning when a subprogram begins its execution and ending when that execution ends, the local variables of the subprogram are all visible to any other executing subprogram, regardless of its textual proximity or how execution got to the currently executing subprogram. There is no way to protect local variables from this accessibility. Subprograms are *always* executed in the environment of all previously called subprograms that have not yet completed their executions. As a result, dynamic scoping results in less reliable programs than static scoping.

A second problem with dynamic scoping is the inability to type check references to nonlocals statically. This problem results from the inability to statically find the declaration for a variable referenced as a nonlocal.

Dynamic scoping also makes programs much more difficult to read, because the calling sequence of subprograms must be known to determine the

meaning of references to nonlocal variables. This task can be virtually impossible for a human reader.

Finally, accesses to nonlocal variables in dynamic-scoped languages take far longer than accesses to nonlocals when static scoping is used. The reason for this is explained in Chapter 10.

On the other hand, dynamic scoping is not without merit. In many cases, the parameters passed from one subprogram to another are variables that are defined in the caller. None of these needs to be passed in a dynamically scoped language, because they are implicitly visible in the called subprogram.

It is not difficult to understand why dynamic scoping is not as widely used as static scoping. Programs in static-scoped languages are easier to read, are more reliable, and execute faster than equivalent programs in dynamic-scoped languages. It was precisely for these reasons that dynamic scoping was replaced by static scoping in most current dialects of Lisp. Implementation methods for both static and dynamic scoping are discussed in Chapter 10.

## 5.6 Scope and Lifetime

Sometimes the scope and lifetime of a variable appear to be related. For example, consider a variable that is declared in a Java method that contains no method calls. The scope of such a variable is from its declaration to the end of the method. The lifetime of that variable is the period of time beginning when the method is entered and ending when execution of the method terminates. Although the scope and lifetime of the variable are clearly not the same, because static scope is a textual, or spatial, concept whereas lifetime is a temporal concept, they at least appear to be related in this case.

This apparent relationship between scope and lifetime does not hold in other situations. In C and C++, for example, a variable that is declared in a function using the specifier **`static`** is statically bound to the scope of that function and is also statically bound to storage. So, its scope is static and local to the function, but its lifetime extends over the entire execution of the program of which it is a part.

Scope and lifetime are also unrelated when subprogram calls are involved. Consider the following C++ functions:

```
void printheader() {
  . . .
 }  /* end of printheader */
void compute() {
  int sum;
  . . .
  printheader();
 } /* end of compute */
```

The scope of the variable `sum` is completely contained within the `compute` function. It does not extend to the body of the function `printheader`, although `printheader` executes in the midst of the execution of `compute`. However, the

lifetime of `sum` extends over the time during which `printheader` executes. Whatever storage location `sum` is bound to before the call to `printheader`, that binding will continue during and after the execution of `printheader`.

## 5.7 Referencing Environments

The **referencing environment** of a statement is the collection of all variables that are visible in the statement. The referencing environment of a statement in a static-scoped language is the variables declared in its local scope plus the collection of all variables of its ancestor scopes that are visible. In such a language, the referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to variables from other scopes during run time. Techniques for implementing references to nonlocal variables in both static- and dynamic-scoped languages are discussed in Chapter 10.

In Python, scopes can be created by function definitions. The referencing environment of a statement includes the local variables, plus all of the variables declared in the functions in which the statement is nested (excluding variables in nonlocal scopes that are hidden by declarations in nearer functions). Each function definition creates a new scope and thus a new environment. Consider the following Python skeletal program:

```
g = 3; # A global

def sub1():
    a = 5; # Creates a local
    b = 7; # Creates another local
    . . . <------------------------------ 1
  def  sub2():
  global g;  # Global g is now assignable here
    c = 9;  # Creates a new local
    . . . <------------------------------ 2
    def  sub3():
      nonlocal  c:  # Makes nonlocal c visible here
      g = 11;  # Creates a new local
      . . . <------------------------------ 3
```

The referencing environments of the indicated program points are as follows:

| *Point* | *Referencing Environment* |
|---|---|
| 1 | local `a` and `b` (of `sub1`), global `g` for reference, but not for assignment |
| 2 | local `c` (of `sub2`), global `g` for both reference and for assignment |
| 3 | nonlocal `c` (of `sub2`), local `g` (of `sub3`) |