3 Names, Scopes, and Bindings

3.4 Implementing Scope

For both static and dynamic scoping, a language implementation must keep track of the name-to-object bindings in effect at each point in the program. The principal difference is *time*: with static scope the compiler uses a *symbol table* to track bindings at compile time; with dynamic scoping the interpreter or run-time system uses an *association list* or *central reference table* to track bindings at run time.

3.4.1 Symbol Tables

In a language with static scoping, the compiler uses an insert operation to place a name-to-object binding into the symbol table for each newly encountered declaration. When it encounters the use of a name that should already have been declared, the compiler uses a lookup operation to search for an existing binding. It is tempting to try to accommodate the visibility rules of static scoping by performing a remove operation to delete a name from the symbol table at the end of its scope. Unfortunately, several factors make this straightforward approach impractical:

- The ability of inner declarations to hide outer ones in most languages with nested scopes means that the symbol table has to be able to contain an arbitrary number of mappings for a given name. The lookup operation must return the innermost mapping, and outer mappings must become visible again at end of scope.
- Records (structures) and classes have some of the properties of scopes, but do not share their nicely nested structure. When it sees a record declaration, the semantic analyzer must remember the names of the record's fields (recursively, if records are nested). At the end of the declaration, the field names must become invisible. Later, however, whenever a variable of the record type appears in the program text (as in `my_rec.field_name`), the record fields

must suddenly become visible again for the part of the reference after the dot. In Pascal and other languages with `with` statements (Section 7.3.3), field names must become visible in a multi-statement context.

- As noted in Section 3.3.3, names are sometimes used before they are declared. Algol and C, for example, permit *forward references* to labels. Pascal permits forward references in pointer declarations. Modula-3 permits forward references of all kinds.
- As noted in Section 3.3.3, C, C++, and Ada distinguish between the declaration of an object and its definition. Pascal has a similar mechanism for mutually recursive subroutines. When it sees a declaration, the compiler must remember any nonvisible details, so that it can check the eventual definition for consistency. This operation is similar to remembering the field names of records.
- While it may be desirable to forget names at the end of their scope, and even to reclaim the space they occupy in the symbol table, information about them may need to be saved for use by a *symbolic debugger*. The debugger is a tool that allows the user to manipulate a running program: starting it, stopping it, and reading and writing its data. In order to parse high level commands from the user (e.g., to print the value of `my_firm^.revenues[1999]`), the debugger must have access to the compiler's symbol table. To make it available at run time, the compiler typically saves the table in a hidden portion of the final machine-language program.

EXAMPLE 3.44

The LeBlanc-Cook symbol table

To accommodate these concerns, most compilers never delete anything from the symbol table. Instead, they manage visibility using enter_scope and leave_scope operations. Implementations vary from compiler to compiler; the approach described here is due to LeBlanc and Cook [CL83].

Each scope, as it is encountered, is assigned a serial number. The outermost scope (the one that contains the predefined identifiers), is given number 0. The scope containing programmer-declared global names is given number 1. Additional scopes are given successive numbers as they are encountered. All serial numbers are distinct; they do not represent the level of lexical nesting, except in as much as nested subroutines naturally end up with numbers higher than those of surrounding scopes.

All names, regardless of scope, are entered into a single large hash table, keyed by name. Each entry in the table then contains the symbol name, its category (variable, constant, type, procedure, field name, parameter, etc.), scope number, type (a pointer to another symbol table entry), and additional, category-specific fields.

In addition to the hash table, the symbol table has a *scope stack* that indicates, in order, the scopes that compose the current referencing environment. As the semantic analyzer scans the program, it pushes and pops this stack whenever it enters or leaves a scope, respectively. Entries in the scope stack contain the scope number, an indication of whether the scope is closed, and in some cases further information.

```
procedure lookup(name)
    pervasive := best := null
    apply hash function to name to find appropriate chain
    foreach entry e on chain
        if e.name = name        -- not something else with same hash value
            if e.scope = 0
                pervasive := e
            else
                foreach scope s on scope stack, top first
                    if s.scope = e.scope
                        best := e        -- closer instance
                        exit inner loop
                    elsif best ≠ null and then s.scope = best.scope
                        exit inner loop        -- won't find better
                    if s.closed
                        exit inner loop        -- can't see farther
    if best ≠ null
        while best is an import or export entry
            best := best.real_entry
        return best
    elsif pervasive ≠ null
        return pervasive
    else
        return null        -- name not found
```

Figure 3.18 LeBlanc-Cook symbol table lookup operation.

To look up a name in the table, we scan down the appropriate hash chain looking for entries that match the name we are trying to find. For each matching entry, we scan down the scope stack to see if the scope of that entry is visible. We look no deeper in the stack than the top-most closed scope. Imports and exports are made visible outside their normal scope by creating additional entries in the table; these extra entries contain pointers to the real entries. We don't have to examine the scope stack at all for entries with scope number 0: they are pervasive. Pseudocode for the lookup algorithm appears in Figure 3.18. ■

EXAMPLE 3.45

Symbol table for a sample program

The lower right portion of Figure 3.19 contains the skeleton of a Modula-2 program. The remainder of the figure shows the configuration of the symbol table for the referencing environment of the `with` statement in procedure `P2`. The scope stack contains four entries representing, respectively, the `with` statement, procedure `P2`, module `M`, and the global scope. The scope for the `with` statement indicates the specific record variable to which names (fields) in this scope belong. The outermost, pervasive scope is not explicitly represented.

All of the entries for a given name appear on the same hash chain, since the table is keyed on name. In this example, `A2`, `F2`, and `T` have also ended up on a single chain, due to hash collisions. Variables `V` and `I` (`M`'s `I`) have extra entries, to make them visible across the boundary of closed scope `M`. When we are inside

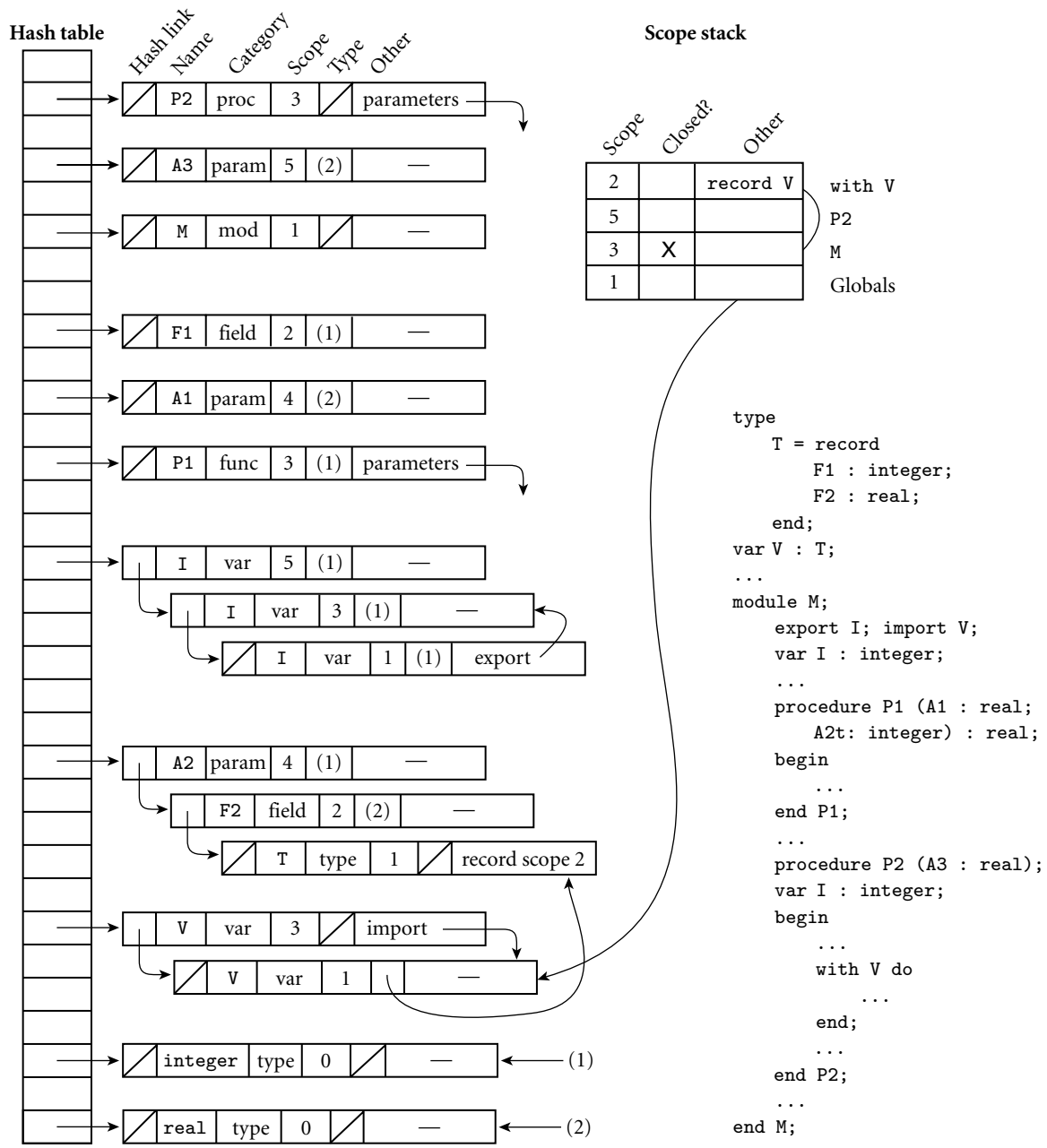

Figure 3.19 **LeBlanc-Cook symbol table for an example program in a language like Modula-2.** The scope stack represents the referencing environment of the `with` statement in procedure `P2`. For the sake of clarity, the many pointers from type fields to the symbol table entries for `integer` and `real` are shown as parenthesized (1)s and (2)s, rather than as arrows.

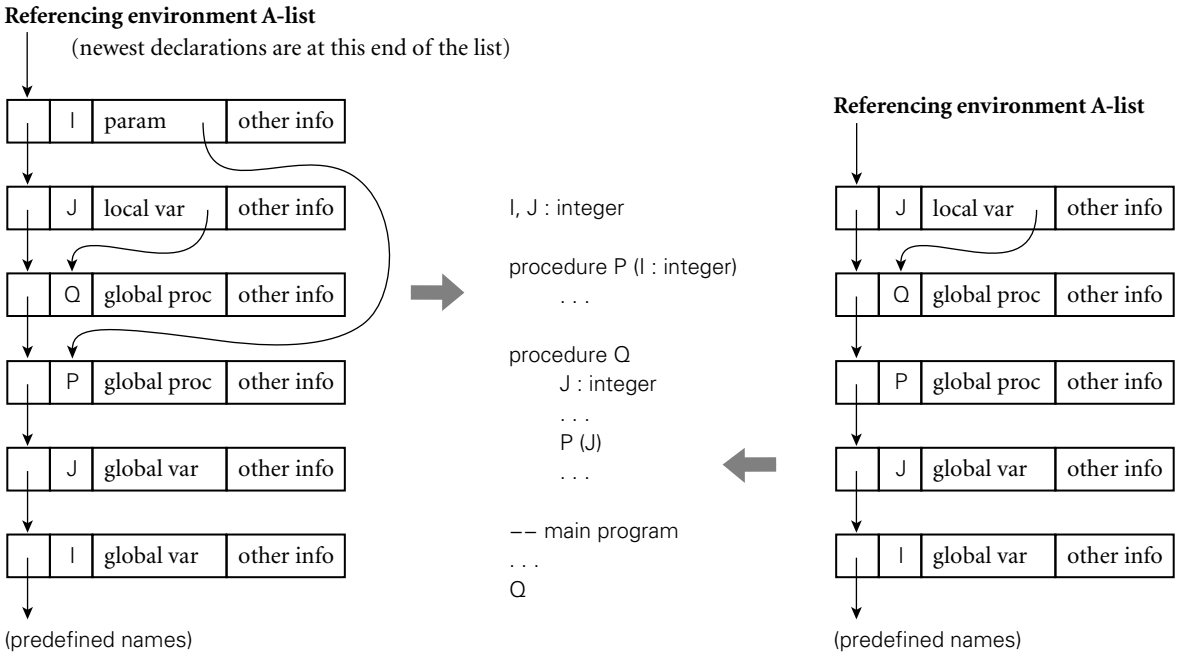

Figure 3.20 **Dynamic scoping with an association list.** The left side of the figure shows the referencing environment at the point in the code indicated by the adjacent grey arrow: after the main program calls Q and it in turn calls P. When searching for I, one will find the parameter at the beginning of the A-list. The right side of the figure shows the environment at the other grey arrow: after P returns to Q. When searching for I, one will find the global definition.

P2, a lookup operation on `I` will find P2's `I`; neither of the entries for M's `I` will be visible. The entry for type `T` indicates the scope number to be pushed onto the scope stack during `with` statements. The entry for each subroutine contains the head pointer of a list that links together the subroutine's parameters, for use in analyzing calls (additional links of these chains are not shown). During code generation, many symbol table entries would contain additional fields, for such information as size and run-time address. ■

3.4.2 Association Lists and Central Reference Tables

Pictorial representations of the two principal implementations of dynamic scoping appear in Figures 3.20 and 3.21. Association lists are simple and elegant, but can be very inefficient. Central reference tables resemble a simplified LeBlanc-Cook symbol table, without the separate scope stack; they require more work at scope entry and exit than do association lists, but they make *lookup* operations fast.

EXAMPLE 3.46
A-list lookup in Lisp

A-lists are widely used for dictionary abstractions in Lisp; they are supported by a rich set of built-in functions in most Lisp dialects. It is therefore natural

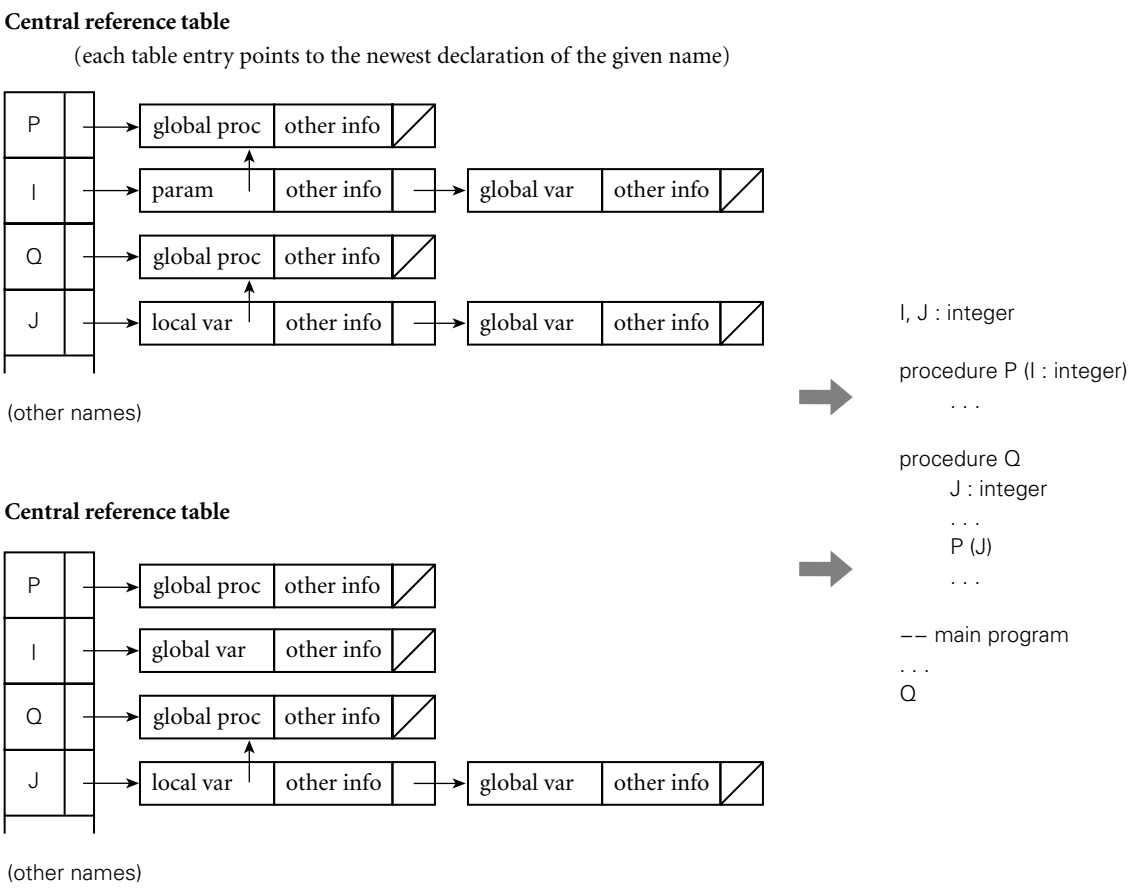

Figure 3.21 **Dynamic scoping with a central reference table.** The upper half of the figure shows the referencing environment at the point in the code indicated by the upper grey arrow: after the main program calls Q and it in turn calls P. When searching for I, one will find the parameter at the beginning of the chain in the I slot of the table. The lower half of the figure shows the environment at the lower grey arrow: after P returns to Q. When searching for I, one will find the global definition.

for Lisp interpreters to use an A-list to keep track of name-value bindings, and even to make this list explicitly visible to the running program. Since bindings are created when entering a scope, and destroyed when leaving or returning from a scope, the A-list functions as a stack. When execution enters a scope at run time, the interpreter pushes bindings for names declared in that scope onto the top of the A-list. When execution finally leaves a scope, these bindings are removed. To look up the meaning of a name in an expression, the interpreter searches from the top of the list until it finds an appropriate binding (or reaches the end of the list, in which case an error has occurred). Each entry in the list contains whatever information is needed to perform semantic checks (e.g., type checking, which we will consider in Section 7.2) and to find variables and other objects that occupy

memory locations. In the left half of Figure 3.20, the first (top) entry on the A-list represents the most recently encountered declaration: the I in procedure P. The second entry represents the J in procedure Q. Below these are the global symbols, Q, P, J, and I, and (not shown explicitly) any predefined names provided by the Lisp interpreter.

The problem with using an association list to represent a program's referencing environment is that it can take a long time to find a particular entry in the list, particularly if it represents an object declared in a scope encountered early in the program's execution, and now buried deep in the list. A central reference table is designed for faster access. It has one slot for every distinct name in the program. The table slot in turn contains a list (stack) of declarations encountered at run time, with the most recent occurrence at the beginning of the list. Looking up a name is now easy: the current meaning is found at the beginning of the list in the appropriate slot in the table. In the upper part of Figure 3.21, the first entry on the I list is the I in procedure P; the second is the global I. If the program is compiled and the set of names is known at compile time, then each name can have a statically assigned slot in the table, which the compiled code can refer to directly. If the program is not compiled, or the set of names is not statically known, then a hash function will need to be used at run time to find the appropriate slot.

EXAMPLE 3.47
Central reference table

When control enters a new scope at run time, entries must be pushed onto the beginning of every list in the central reference table whose name is (re)declared in that scope. When control leaves a scope for the final time, these entries must be popped. The work involved is somewhat more expensive than pushing and popping an A-list, but not dramatically more so, and lookup operations are now much faster. In contrast to the symbol table of a compiler for a language with static scoping, central reference table entries for a given scope do not need to be saved when the scope completes execution; the space can be reclaimed.

Within the Lisp community, implementation of dynamic scoping via an association list is sometimes called *deep binding*, because the lookup operation may need to look arbitrarily deep in the list. Implementation via a central reference table is sometimes called *shallow binding*, because it finds the current association at the head of a given reference chain. Unfortunately, the terms "deep and shallow binding" are also more widely used for a completely different purpose, discussed in Section 3.6. To avoid potential confusion, some authors use "deep and shallow access" [Seb08] or "deep and shallow search" [Fin96] for the implementations of dynamic scoping.

Closures with Dynamic Scoping

(This subsection is best read after Section 3.6.1.)

EXAMPLE 3.48
A-list closures

If an association list is used to represent the referencing environment of a program with dynamic scoping, the referencing environment in a closure can be represented by a top-of-stack (beginning of A-list) pointer (Figure 3.22). When a subroutine is called through a closure, the main pointer to the referencing environment A-list is temporarily replaced by the pointer from the closure, making

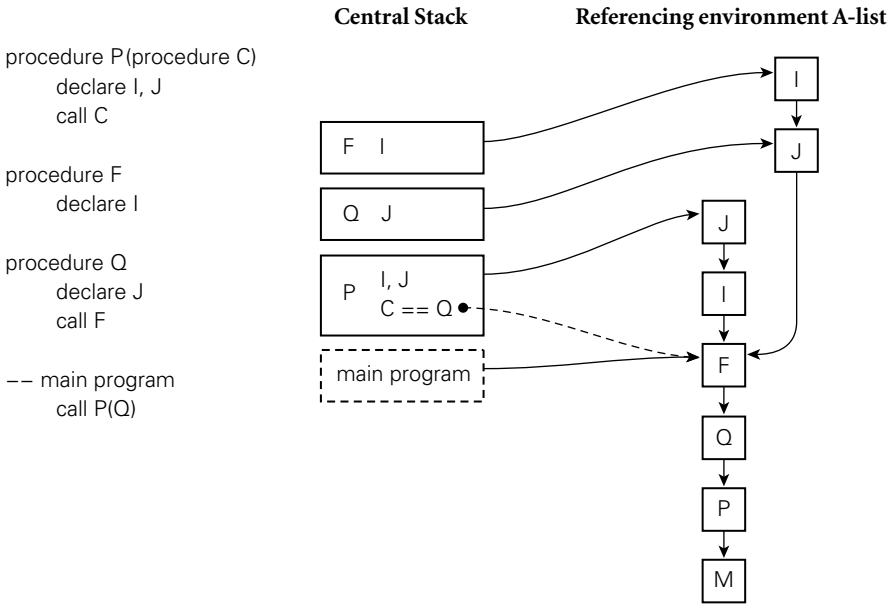

Figure 3.22 **Capturing the A-list in a closure.** Each frame in the stack has a pointer to the current beginning of the A-list, which the run-time system uses to look up names. When the main program passes Q to P with deep binding, it bundles its A-list pointer in Q's closure (dashed arrow). When P calls C (which is Q), it restores the bundled pointer. When Q elaborates its declaration of J (and F elaborates its declaration of I), the A-list is temporarily bifurcated.

any bindings created since the closure was created (P's I and J in the figure) temporarily invisible. New bindings created *within* the subroutine (or in any subroutine it calls) are pushed using the temporary pointer. Because the A-list is represented by pointers (rather than an array), the effect is to have two lists—one representing the caller's referencing environment and the other temporary referencing environment resulting from use of the closure—that share their older entries. When Q returns to P in our example, the original head of the A-list will be restored, making P's I and J visible again. ■

With a central reference table implementation of dynamic scoping, the creation of a closure is more complicated. In the general case, it may be necessary to copy the entire main array of the central table and the first entry on each of its lists. Space and time overhead may be reduced if the compiler or interpreter is able to determine that only some of the program's names will be used by the subroutine in the closure (or by things that the subroutine may call). In this case, the environment can be saved by copying the first entries of the lists for only the names that will be used. When the subroutine is called through the closure, these entries can then be pushed onto the beginnings of the appropriate lists in the central reference table. Additional code must be executed to remove them again after the subroutine returns.

CHECK YOUR UNDERSTANDING

44. List the basic operations provided by a symbol table.

45. Outline the implementation of a LeBlanc-Cook style symbol table.

46. Why don't compilers generally remove names from the symbol table at the ends of their scopes?

47. Describe the *association list* (*A-list*) and *central reference table* data structures used to implement dynamic scoping. Summarize the tradeoffs between them.

48. Explain how to implement deep binding by capturing the referencing environment A-list in a closure. Why are closures harder to build with a central reference table?