8
Symbol Tables

Chapter Chapter:global:seven considered the construction of an **abstract syntax tree** (AST) as an artifact of a top-down or bottom-up parse. On its own, a top-down or bottom-up parser cannot fully accomplish the compilation of modern programming languages. The AST serves to represent the source program and to coordinate information contributed by the various passes of a compiler. This chapter presents one such pass—the harvesting of **symbols** from an AST. Most programming languages allow the declaration, definition, and use of symbolic names to represent constants, variables, methods, types, and objects. The compiler checks that such names are used correctly, based on the programming language's definition. This chapter describes the organization and implementation of a **symbol table**. This structure records the names and important attributes of a program's names. Examples of such attributes include a name's type, scope, and accessibility.

We are interested in two aspects of a symbol table: its use and its organization. Section 8.1 defines a simple symbol table interface and shows how to use this interface to manage symbols for a block-structured language. Section 8.2 explains the effects of program *scopes* on symbol-table management. Section 8.3 examines various implementations of a symbol table. Advanced topics, such as type definitions, inheritance, and overloading are considered in Section 8.4.

8.1 Use of a Symbol Table

In this section, we consider how to construct a symbol table for a simple, block-structured language. Assuming an AST has been constructed as described in Chapter Chapter:global:seven, we **walk** (make a **pass** over) the AST to

- process symbol declarations and

- connect each symbol reference with its declaration.

Symbol references are connected with declarations through the symbol table. An AST node that mentions a symbol by name is enriched with a reference to the name's entry in the symbol table. If such a connection cannot be made, then the offending reference is improperly declared and an error message is issued. Otherwise, subsequent passes can use the symbol table reference to obtain information about the symbol, such as its type, storage requirements, and accessibility.

The block-structured program shown in Figure 8.1(a) contains two nested scopes. Although the program uses keywords such as `float` and `int`, no symbol table action is required for these symbols if they are recognized by the scanner as the terminal symbols float and int, respectively. Most programming-language grammars demand such precision of the scanner to avoid ambiguity in the grammar.

The program in Figure 8.1(a) begins by *importing* function definitions for `f` and `g`. The compiler finds these, determines that their return types are `void`, and then records the functions in the symbol table as its first two entries. The declarations for `w` and `x` in the outer scope are entered as symbols 3 and 4, respectively. The inner scope's redeclaration of `x` and its declaration of `z` are entered as symbols 5 and 6, respectively. The AST in Figure 8.1(b) refers to symbols by name. In Figure 8.1(d), the names are replaced by symbol table references. In particular, the references to `x`—shown only by name in Figure 8.1(b)—are declaration-specific in Figure 8.1(d). In the symbol table, the references contain the original name as well as type information processed from the symbol declarations.

8.1.1 Static Scoping

Modern programming languages offer **scopes** to confine the activity of a name to a prescribed region of a program. A name may be declared at most once in any given scope. For statically scoped, block-structured languages, references are typically resolved to the declaration in their closest containing scope. Additionally, most languages contain directives to promote a given declaration or reference to the program's **global scope**—a name space shared by all compilation units. Section 8.4.4 discusses these issues in greater detail.

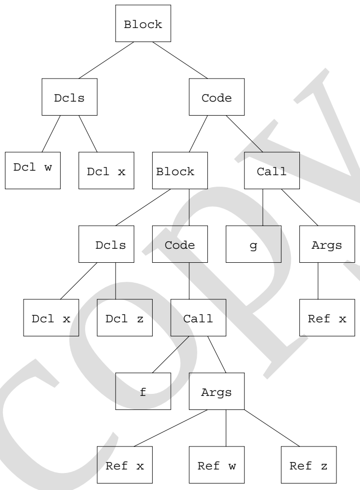
Proper use of scopes results in programs whose behavior is easier to understand. Figure 8.1(a) shows a program with two nested scopes. Declared in the program's outer scope, `w` is available in both scopes. The activity of `z` is confined to the inner scope. The name `x` is available in both scopes, but the *meaning* of `x` changes—the inner scope redeclares `x`. In general, the **static scope** of an identifier includes its defining block as well as any contained blocks that do not themselves contain a declaration for the identifier.

Languages such as C, Pascal, and Java use the following keywords or punctuation to define static scopes.

- For Pascal, the reserved keywords `begin` and `end` open and close scopes, respectively. Within each scope, types, variables, and methods can be declared.

```
import f(float, float, float)
import g(int)
{
  int w,x
  {
     float x,z
     f(x,w,z)
  }
  g(x)
}
```

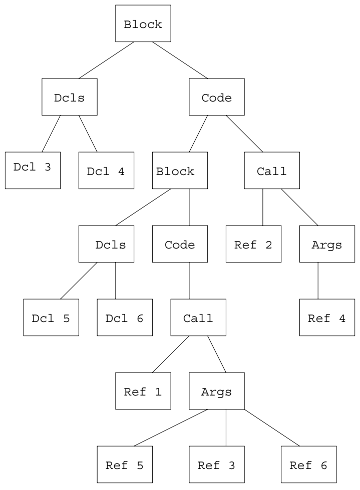
(a)



(b)

Symbol Number	Symbol Name	Attributes
1	f	void func(float, float, float)
2	g	void func(int)
3	w	int
4	x	int
5	x	float
6	z	float

(c)



(d)

Figure 8.1: Symbol table processing for a block-structured program.

- For C and Java, scopes are opened and closed by the appropriate braces, as shown in Figure 8.1(a). In these languages, a scope can declare types and variables. However, methods (function definitions) cannot appear in inner scopes.

As considered in Exercises 1 and 2, compilers sometimes create scopes to make room for temporaries and other compiler-generated names. For example, the contents of a register may be temporarily deposited in a compiler-generated name to free the register for some other task. When the task is finished, the register is reloaded from the temporary storage. A scope nicely delimits the useful life of such temporaries.

In some languages, references to names in outer scopes can incur overhead at runtime. As discussed in Chapter Chapter:global:eleven, an important consideration is whether a language allows the *nested declaration* of methods. C and Java prohibit this, while Pascal allows methods to be defined in any scope. For C and Java, a method's local variables can be **flattened** by renaming nested symbols and moving their declaration to the method's outermost scope. Exercise 3 considers this in greater detail.

8.1.2 A Symbol Table Interface

A symbol table is responsible for tracking *which* declaration is in effect when a reference to the symbol is encountered. In this section, we define a symbol-table interface for processing symbols in a block-structured, statically scoped language. The methods in our interface are as follows.

INCRNESTLEVEL( ) opens a new scope in the symbol table. New symbols are entered in the resulting scope.

DECRNESTLEVEL( ) closes the most recently opened scope in the symbol table. Symbol references subsequently revert to outer scopes.

ENTERSYMBOL(*name*, *type*) enters the *name* in the symbol table's current scope. The parameter *type* conveys the data type and access attributes of *name*'s declaration.

RETRIEVESYMBOL(*name*) returns the symbol table's currently valid declaration for *name*. If no declaration for *name* is currently in effect, then a *null* pointer can be returned.

To illustrate the use of this interface, Figure 8.2 contains code to build the symbol table for the AST shown in Figure 8.1. The code is specialized to the type of the AST node. Actions can be performed both before and after a given node's children are visited. Prior to visiting the children of a Block node, Step **1** increases the static scope depth. After the subtree of the Block is processed, Step **2** abandons the scope. The code for Ref retrieves the symbol's current definition in the symbol table. If none exists, then an error message is issued.

```
procedure BuildSymbolTable( )
    call ProcessNode( ASTroot )
end
procedure ProcessNode( node )
    switch (node.kind)
        case Block
            call symtab.IncrNestLevel( )                         1
        case Dcl
            call symtab.EnterSymbol( name, type )
        case Ref
            sym ← symtab.RetrieveSymbol( name )
            if sym = ⊥
            then  call error( “Undeclared symbol” )
    foreach c ∈ Children(node) do call ProcessNode( c )
    switch (node.kind)
        case Block
            call symtab.DecrNestLevel( )                         2
end
```

Figure 8.2: Building the symbol table.

8.2 Block-Structured Languages and Scope Management

Most programming languages allow scopes to be nested statically, based on concepts introduced by Algol 60. Languages that allow nested name scopes are known as **block-structured languages**. While the mechanisms that open and close scopes can vary by language, we assume that the IncrNestLevel and DecrNestLevel methods are the uniform mechanism for opening and closing scopes in the symbol table. In this section, we consider various language constructs that call for the opening and closing of scopes. We also consider the issue of allocating a symbol table per scope as compared with a single, global symbol table.

8.2.1 Scopes

Every symbol reference in an AST occurs in the context of defined scopes. The scope defined by the *innermost* such context is known as the **current scope**. The scopes defined by the current scope and its surrounding scopes are known as the **open** or **currently active scopes**. All other scopes are said to be **closed**. Based on these definitions, current, open, and closed scopes are not fixed attributes; instead, they are defined relative to a particular point in the program.

Following are some common **visibility rules** that define the interpretation of a name in the presence of multiple scopes.

- At any point in the text of a program, the accessible names are those that are declared in the current scope and in all other open scopes.
- If a name is declared in more than one open scope, a reference to the name is resolved to the *innermost* declaration—the one that most closely surrounds the reference.
- New declarations can be made only in the current scope.

Most languages offer mechanisms to install or resolve symbol names in the outermost, program-global scope. In C, names bearing the `extern` attribute are resolved globally. In Java, a class can reference any class's `public static` fields, but these fields do not populate a single, flat name space. Instead, each such field must be fully qualified by its containing class.

Programming languages have evolved to allow various useful levels of scoping. C and C++ offer a compilation-unit scope—names declared outside of all methods are available within the compilation unit's methods. Java offers a package-level scope—classes can be organized into packages that can access the package-scoped methods and fields. In C, every function definition is available in the global scope, unless the definition has the `static` attribute. In C++ and Java, names declared within a class are available to all methods in the class. In Java and C++, a class's fields and methods bearing the `protected` attribute are available to the class's subclasses. The parameters and local variables of a method are available within the given method. Finally, names declared within a statement-block are available in all contained blocks, unless the name is redeclared in an inner scope.

8.2.2 One Symbol Table or Many?

There are two common approaches to implementing block-structured symbol tables, as follows.

- A symbol table is associated with each scope.
- All symbols are entered in a single, global table.

A single symbol table must accommodate multiple, active declarations of the same symbol. On the other hand, searching for a symbol can be faster in a single symbol table. We next consider this issue in greater detail.

An Individual Table for Each Scope

If an individual table is created for each scope, some mechanism must be in place to ensure that a search produces the name defined by the nested-scope rules. Because name scopes are opened and closed in a **last-in, first-out** (LIFO) manner, a stack is an appropriate data structure for organizing such a search. Thus, a **scope stack** of symbol tables can be maintained, with one entry in the stack for each open scope. The innermost scope appears at the top of the stack. The next containing scope is second from the top, and so forth. When a new scope is opened, INCRNESTLEVEL creates and pushes a new symbol table on the stack. When a scope is closed, DECRNESTLEVEL, the top symbol table is popped.

A disadvantage of this approach is that we may need to search for a name in a number of symbol tables before the symbol is found. The cost of this stack search varies from program to program, depending on the number of nonlocal name references and the depth of nesting of open scopes. In fact, studies have shown that most lookups of symbols in block-structured languages return symbols in the inner- or outer-most scopes. With a table per scope, intermediate scopes must be checked before an outermost declaration can be returned.

An example of this symbol table organization is shown in Figure 8.8.

One Symbol Table

In this organization, all names in a compilation unit's scopes are entered into the same table. If a name is declared in different scopes, then the scope name or depth helps identify the name uniquely in the table. With a single symbol table, RETRIEVESYMBOL need not chain through scope tables to locate a name. Section 8.3.3 describes this kind of symbol table in greater detail. Such a symbol table is shown in Figure 8.7.

8.3 Basic Implementation Techniques

Any implementation of the interface presented in Section 8.1 must correctly insert and find symbols. Depending on the number of names that must be accommodated and other performance considerations, a variety of implementations is possible. Section 8.3.1 examines some common approaches for organizing symbols in a symbol table. Section 8.3.2 considers how to represent the symbol names themselves. Based on this discussion, Section 8.3.3 proposes an efficient symbol table implementation.

8.3.1 Entering and Finding Names

We begin by considering various approaches for *organizing* the symbols in the symbol table. For each approach, we examine the time needed to

- insert symbols,
- retrieve symbols, and
- maintain scopes.

These actions are not typically performed with equal frequency. Each scope can declare a name at most once, but names can be referenced multiple times. It is therefore reasonable to expect that RETRIEVESYMBOL is called more frequently than the other methods in our symbol table interface. Thus, we pay particular attention to the cost of retrieving symbols.

Unordered List

This is the simplest possible storage mechanism. The only data structure required is an array, with insertions occurring at the next available location. For added flexibility, a linked list avoids the limitations imposed by a fixed-size array. In this representation, ENTERSYMBOL inserts a name at the head of the unordered list. The scope name (or depth) is recorded with the name. This allows ENTERSYMBOL to detect if the same name is entered twice in the same scope—a situation disallowed by most programming languages. RETRIEVESYMBOL searches for a name from the head of the list toward its tail, so that the closest, active declaration of the name is encountered first.

All names for a given scope appear adjacently in the unordered list. Thus, INCRNESTLEVEL can annotate the list with a marker to show where the new scope begins. DECRNESTLEVEL can then delete the currently active symbols at the head of the list.

Although insertion is fast, retrieval of a name from the outermost scope can require scanning the entire unordered list. This approach is therefore impractically slow except for the smallest of symbol tables.

Ordered List

If a list of n distinct names is maintained alphabetically, binary search can find any name in $O(\log n)$ time. In the unordered list, declarations from the same scope appear in sequence—an unlikely situation for the ordered list. How should we organize the ordered list to accommodate declarations of a name in multiple scopes? Exercise 4 investigates the potential performance of storing all names in a single, ordered list. Because RETRIEVESYMBOL accesses the *currently active* declaration for a name, a better data structure is an ordered list of *stacks*. Each stack represents one currently active name; the stacks are ordered by their representative names.

RETRIEVESYMBOL locates the appropriate stack using binary search. The currently active declaration appears on top of the located stack. DECRNESTLEVEL must pop those stacks containing declarations for the abandoned scope. To facilitate this, each symbol can be recorded along with its scope name or depth, as established by INCRNESTLEVEL. DECRNESTLEVEL can then examine each stack in the list and pop those stacks whose top symbol is declared in the abandoned scope. When a stack becomes empty, it can be removed from the ordered list. Figure 8.3 shows such a symbol table for the example in Figure 8.1, at the point where method `f` is invoked.

A more efficient approach avoids touching each stack when a scope is abandoned. The idea is to maintain a separate linking of symbol table entries that are declared at the same scope level. Section 8.3.3 presents this organization in greater detail.

The details of maintaining a symbol table using ordered lists is left as Exercise 5. Although ordered lists offer fast retrieval, insertion into an ordered list

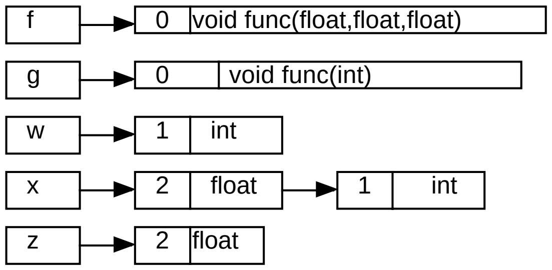


Figure 8.3: An ordered list of symbol stacks.

is relatively expensive. Thus, ordered lists are advantageous when the space of symbols is known in advance—such is the case for reserved keywords.

Binary Search Trees

Binary search trees are designed to combine the efficiency of a linked data structure for insertion with the efficiency of binary search for retrieval. Given random inputs, it is expected that a name can be inserted or found in $O(\log n)$ time, where n is the number of names in the tree. Unfortunately, average-case performance does not necessarily hold for symbol tables—programmers do not choose identifier names at random! Thus, a tree of n names could have depth n, causing name lookup to take $O(n)$ time.

An advantage of binary search trees is their simple, widely known implementation. This simplicity and the common perception of reasonable average-case performance make binary search trees a popular technique for implementing symbol tables. As with the list structures, each name (node) in the binary search tree is actually a *stack* of currently active scopes that declare the name.

Balanced Trees

The worst-case scenario for a binary search tree can be avoided if a search tree can be maintained in *balanced* form. The time spent balancing the tree can be amortized over all operations so that a symbol can be inserted or found in $O(\log n)$ time, where n is the number of names in the tree. Examples of such trees include **red-black trees** [?] and **splay trees** [?]. Exercises 10 and 11 further explore symbol table implementations based on balanced-tree structures.

Hash Tables

Hash tables are the most common mechanism for managing symbol tables, owing to their excellent performance. Given a sufficiently large table, a good hash function, and appropriate collision-handling techniques, insertion or retrieval can be performed in *constant* time, regardless of the number of entries in the table. The implementation discussed in Section 8.3.3 uses a hash table, with collisions handled by chaining. Hash tables are widely implemented. Some languages (including Java) contain hash table implementations in their core library. The implementation details for hash tables are covered well in most books on elementary data structures and algorithms [?].

8.3.2 The Name Space

At some point, a symbol table entry must represent the *name* of its symbol. Each name is essentially a string of characters. However, by taking the following properties into consideration, an efficient implementation of the name space can be obtained.

- The name of a symbol does not change during compilation. Thus, the strings associated with symbol table names are **immutable**—once allocated, they do not change.
- Although scopes come and go, the symbol names persist throughout compilation. Scope creation and deletion affects the set of currently available symbols, obtainable through RETRIEVESYMBOL. However, a scope's symbols are not completely forgotten when the scope is abandoned. Space must be reserved for the symbols at runtime, and the symbols may require initialization. Thus, the symbols' strings occupy storage that persists throughout compilation.
- There can be great variance in the *length* of identifier names. Short names—perhaps only a single character—are typically used for iteration variables and temporaries. Global names in a large software system tend to be descriptive and much longer in length. For example, the `X` windowing system contains names such as `VisibilityPartiallyObscured`.
- Unless an ordered list is maintained, comparisons of symbol names involve only equality and inequality.

The above points argue in favor of one logical name space, as shown in Figure 8.4, in which names are inserted but never deleted.

In Figure 8.4, each string referenced by a pair of fields. One field specifies the string's origin in the string buffer, and the other field specifies the string's length. If the names are managed so that the buffer contains at most one occurrence of any name, then the equality of two strings can be tested by comparing the strings' references. If they differ in origin or length, the strings cannot be the same. The Java `String` class contains the method `intern` that maps any string to a unique reference for the string.

The strings in Figure 8.4 do not share any common characters. Exercise 16 considers a stringspace that stores shared substrings more compactly. In some languages, the suffix of a name can suffice to locate the name. For example, a

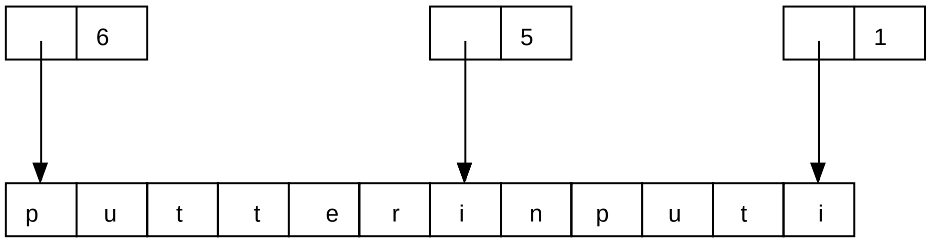


Figure 8.4: Name space for symbols `putter`, `input`, and `i`.

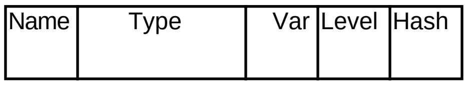


Figure 8.5: A symbol table entry.

reference of `String` in a Java program defaults to `java.lang.String`. Exercise 6 considers the organization of name spaces to accommodate such access.

8.3.3 An Efficient Symbol Table Implementation

We have examined issues of symbol management and representation. Based on the discussion up to this point, we next present an efficient symbol table implementation. Figure 8.5 shows the layout of a symbol table entry, each containing the following fields.

Name is a reference to the symbol name space, organized as described in Section 8.3.2. The name is required to locate the symbol in a chain of symbols with the same hash-table location.

Type is a reference to the type information associated with the symbol's declaration. Such information is processed as described in Chapter Chapter:global:nine.

Hash threads symbols whose names hash to the same value. In practice, such symbols are doubly linked to facilitate symbol deletion.

Var is a reference to the next outer declaration of this same name. When the scope containing this declaration is abandoned, the referenced declaration becomes the currently active declaration for the name. Thus, this field essentially represents a stack of scope declarations for its symbol name.

Level threads symbols declared in the same scope. This field facilitates symbol deletion when a scope is abandoned.

There are two *index* structures for the symbol table: a hash table and a scope display. The hash table allows efficient lookup and entry of names, as described in Section 8.3.1. The **scope display** maintains a list of symbols that are declared at the same level. In particular, the *i*th entry of the scope display references a list of symbols currently active at scope depth *i*. Such symbols are linked by their Level field. Moreover, each active symbol's Var field is essentially a stack of declarations for the associated variable.

Figure 8.6 shows the pseudocode for this symbol table implementation. Figure 8.7 shows the table that results from applying this implementation to the example in Figure 8.1, at the point where method `f` is invoked. Figure 8.7 assumes the following unlikely situation with respect to the hash function.

- `f` and `g` hash to the same location.
- `w` and `z` hash to the same location.
- All symbols are clustered in the same part of the table.

The code in Figure 8.6 relies on the following methods.

DELETE(*sym*) removes the symbol table entry *sym* from the collision chain found at *HashTable*.GET(*sym*.*name*). The symbol is not destroyed—it is simply removed from the collision chain. In particular, its Var and Level fields remains intact.

ADD(*sym*) adds the symbol entry *sym* to the collision chain at *HashTable*.GET(*sym*.*name*). Prior to the call to ADD, there is no entry in the table for *sym*.

When DECRNESTLEVEL is invoked to abandon the currently active scope, each symbol in the scope is visited by the loop at Step **3**. Each such symbol is removed from the hash table at Step **4**. If an outer scope definition exists for the symbol, the definition is inserted into the hash table at Step **5**. Thus, the Var field serves to maintain a stack of active scope declarations for each symbol. The Level field allows DECRNESTLEVEL to operate in time proportional to the number of symbols affected by abandoning the current scope. Amortized over all symbols, this adds a constant overhead to the management of each declared symbol.

RETRIEVESYMBOL examines a collision chain to find the desired symbol. The loop at Step **6** accesses all symbols that hash to the same table location—the chain that should contain the desired *name*. Step **7** follows the entries' Hash fields until the chain is exhausted or the symbol is located. A properly managed hash table should have very short collision chains. Thus, we expect that only a few iterations of the loop at Step **6** should be necessary to locate a symbol or to detect that the symbol has not been properly declared.

ENTERSYMBOL first locates the currently active definition for *name*, should any exist in the table. Step **9** checks that no declaration already exists in the current scope. A new symbol table entry is generated at Step **10**. The symbol is added to those in the current scope by linking it into the scope display. The remaining code inserts the new symbol into the table. If an active scope contains a definition of the symbol name, that name is removed from the table and referenced by the Var field of the new symbol.

Recalling the discussion of Section 8.2.2, an alternative approach segregates symbols by scope. A stack of symbol tables results—one per scope—as shown in Figure 8.8. The code to manage such a structure is left as Exercise 19.

8.4 Advanced Features

We next examine how to extend the simple symbol table framework to accommodate advanced features of modern programming languages. Extensions to our simple framework fall generally in the following categories:

- Name augmentation (overloading)
- Name hiding and promotion
- Modification of search rules

In each case, it is appropriate to rethink the symbol table design to arrive at an efficient, correct implementation of the desired features. In the following sections, we discuss the essential issues associated with each feature. However, we leave the details of the implementation as exercises.

8.4.1 Records and Typenames

Languages such as C and Pascal allow aggregate data structures using the `struct` and `record` type constructors. Because such structures can be nested, access to a field may involve navigating through many containers before the field can be reached. In Pascal, Ada, and C, such fields are accessed by completely specifying the containers and the field. Thus, the reference `a.b.c.d` accesses field `b` of record `a`, field `c` of record `b`, and finally field `d` of record `c`. COBOL and PL/I allow intermediate containers to be omitted—if the reference can be unambiguously resolved. In such languages, `a.b.c.d` might be abbreviated as `a.d` or `c.d`. This idea has not met with general acceptance, partly because programs that use such abbreviations are difficult to read. It is also possible that `a.d` is a *mistake*, but the compiler silently inteprets the reference by filling in missing containers.

Records can be nested arbitrarily deep. Thus, records are typically implemented using a tree. Each record is represented as a node; its children represent the record's subfield. Alternatively, a record can be represented by a symbol table whose entries are the record's subfields. Exercise 14 considers the implementation of records in symbol tables.

C offers the `typedef` construct, which establishes a name as an *alias* for a type. As with record types, it is convenient to establish an entry in the symbol table for

```
procedure IncrNestLevel( )
    $Depth \leftarrow Depth + 1$
    $ScopeDisplay[Depth] \leftarrow \perp$
end
procedure DecrNestLevel( )
    foreach $sym \in ScopeDisplay[Depth]$ do                                  3
        $prevsym \leftarrow sym.var$
        call Delete( $sym$ )                                                  4
        if $prevsym \neq \perp$                                               5
        then  call Add( $prevsym$ )
    $Depth \leftarrow Depth - 1$
end
function RetrieveSymbol( $name$ ) : $Symbol$
    $sym \leftarrow HashTable$.Get( $name$ )
    while $sym \neq \perp$ do                                                 6
        if $sym.name = name$
        then  return ( $sym$ )
        $sym \leftarrow sym$.Hash                                             7
    return ( $\perp$ )                                                        8
end
procedure EnterSymbol( $name, type$ )
    $oldsym \leftarrow$ RetrieveSymbol( $name$ )
    if $oldsym \neq \perp$ and $oldsym.depth = Depth$                         9
    then  call Error( "Duplicate declaration of $name$" )
    $newsym \leftarrow$ CreateNewSymbol( $name, type$ )                       10
    /*                    Add to scope display                    */
    $newsym$.Level $\leftarrow ScopeDisplay[Depth]$
    $ScopeDisplay[Depth] \leftarrow newsym$
    /*                    Add to hash table                       */
    if $oldsym = \perp$
    then  call Add( $newsym$ )
    else
        call Delete( $oldsym$ )
        call Add( $newsym$ )
    $newsym.var \leftarrow oldsym$
end
```

Figure 8.6: Symbol table management.

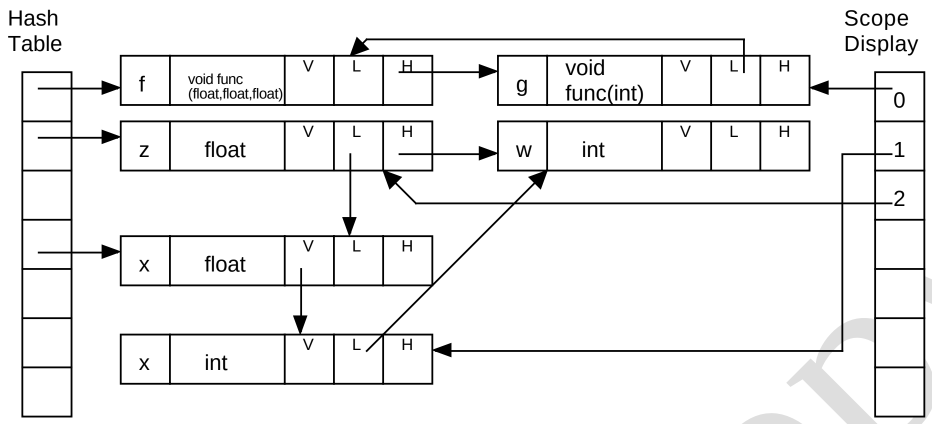


Figure 8.7: Detailed layout of the symbol table for Figure 8.1. The V, L, and H fields abbreviate the Var, Level, and Hash fields, respectively.

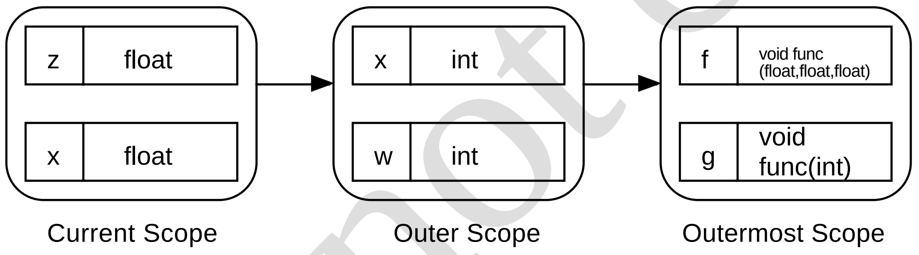


Figure 8.8: A stack of symbol tables, one per scope.

the `typedef`. In fact, most C compilers use scanners that must distinguish between ordinary names and typenames. This is typically accomplished through a "back door" call to the symbol table to lookup each identifier. If the symbol table shows that the name as an active typename, then the scanner returns a typename token. Otherwise, an ordinary identifier token is returned.

8.4.2 Overloading and Type Hierarchies

The notion of an *identifier* has thus far been restricted to a string containing the identifier's name. Situations can arise where the name alone is insufficient to locate a desired symbol. Object-oriented languages such as C++ and Java allow **method overloading**. A method can be defined multiple times, provided that each definition has a unique *type signature*. The **type signature** of a method includes the number and types of its parameters and its return type. With overloading, a program can contain the methods `print(int)` as well as `print(String)`.

When the type signatures are included, the compiler comes to view a method definition not only in terms of its name but also in terms of the its type signature. The symbol table must be capable of entering and retrieving the appropriate symbol for a method. In one approach to method overloading, the type signature of a method is encoded along with its name. For example, the method `println` that accepts an integer and returns nothing is encoded as `println(I)V`. It then becomes necessary to include a method's type signature each time such symbols are retrieved from the symbol table. Alternatively, the symbol table could simply record a method along with a list of of its overloaded definitions. In the AST, method invocations point to the entire list of method definitions. Subsequently, semantic processing scans the list to make certain that a valid definition of the method occurs for each invocation.

Some languages, such as C++ and Ada, allow operator symbols to be overloaded. For example, the meaning of + could change if its arguments are matrices instead of scalars. A program could provide a method that overloads + and performs matrix addition on matrix arguments. The symbol table for such languages must be capable of determining the definition of an operator symbol in every scope.

Ada allows literals to be overloaded. For example, the symbol `Diamonds` could participate simultaneously in two different enumeration types: as a playing card and as a commodity.

Pascal and FORTRAN employ a small degree of overloading in that the same symbol can represent the invocation of a method as well as the value of the method's result. For such languages, the symbol table contains two entries. One represents the method while the other represents a name for the value returned by the method. It is clear in context whether the name means the value returned by the method or the method itself. As demonstrated in Figure 8.1, semantic processing makes an explicit connection between a name and its symbol.

C also has overloading to a certain degree. A program can use the same name as a local variable, a `struct` name, and a label. Although it is unwise to write such confusing programs, C allows this because it is clear in each context which definition of a name is intended (see Exercise 13).

Languages such as Java and C++ offer type extension through subclassing. The symbol table could contain a method `resize(Shape)`, while the program invokes the method `resize(Rectangle)`. If `Rectangle` is a subclass of `Shape`, then the invocation should resolve to the method `resize(Shape)`. However, if the program contains a `resize` method for both `Rectangle` and `Shape` types, then resolution should choose the method whose formal parameters most closely match the types of the supplied parameters.

In modern, object-oriented languages, a method call's resolution is determined not only by its type signature, but also by its situation in the type hierarchy. For example, consider a Java class `A` and its subclass `B`. Suppose classes `A` and `B` both define the method `sqrt(int)`. Given an instance `b` of class `B`, an invocation of `((A) b).sqrt()` causes `B`'s `sqrt` to execute, even though the instance is apparently converted to an object of type A. This is because the *runtime* type identification of the instance determines which class's `sqrt` is executed. At compile-time it is generally undecidable which version of `sqrt` will execute at runtime. However, the compiler can check that a valid definition of `sqrt` is present for this example. The instance of type B is regarded as an instance of class A after the cast. The compiler can check that class A provides a suitable definition of `sqrt`. If so, then all of A's subclasses provide a default definition by inheritance, and perhaps a more specialized definition by overiding the method. In fact, Java provides no mechanism for directly invoking A's definition of `sqrt` on an object whose runtime type is B. In contrast, C++ allows a *nonvirtual* call to A's definition of `sqrt`. In fact, such is the default for a B object once it is cast to type A. Chapter Chapter:global:thirteen considers these issues in greater detail.

8.4.3 Implicit Declarations

In some languages, the appearance of a name in a certain context serves to declare the name as well. As a common example, consider the use of *labels* in C. A label is introduced as an identifier followed by a colon. Unlike Pascal, the program need not declare the use of such labels in advance. In FORTRAN, the type of an identifier for which no declaration is offered can be inferred from the identifier's first letter. In Ada, a `for loop` index is implicitly declared to be of the same type as the range specifier. Moreover, a *new scope* is opened for the loop so that the loop index cannot clash with an existing variable (see Exercise 12).

Implicit declarations are almost always introduced in a programming language for convenience—the convenience of those who *use* rather than *implement* the language. Taking this point further, implicit declarations may ease the task of *writing* programs at the expense of those who must later read them. In any event, the compiler is responsible for supporting such features.

8.4.4 Export and Import Directives

Export rules allow a programmer to specify that some local scope names are to become visible outside that scope. This selective visibility is in contrast to the usual block-structured scope rule, which causes local scope names to be *invisible* outside the scope. Export rules are typically associated with modularization features such such as Ada packages, C++ classes, C compilation units, and Java classes. These language features help a programmer organize a program's files by their functionality.

In Java, the `public` attribute causes the associated field or method to be known outside its class. To prevent name clashes, each class can situate itself in a package hierarchy through the `package` directive. Thus, the `String` class provided in the Java core library is actually part of the `java.lang` package. In contrast, all methods in C are known outside of their compilation units, *unless* the `static` attribute is bestowed. The `static` methods are available only within their compilation units.

With an export rule, each compilation unit advertises its offerings. In a large software system, the space of available global names can become polluted and disorganized. To manage this, compilation units are typically required to specify which names they wish to *import*. In C and C++, the use of a header file includes declarations of methods and structures that can be used by the compilation unit. In Java, the `import` directive specifies the classes and packages that a compilation unit might access. Ada's `use` directive serves essentially the same purpose.

To process the export and import directives, the compiler typically examines the import directives to obtain a list of potentially external references. These references are then examined by the compiler to determine their validity, to the extent that is possible at compile time. In Java, the `import` directives serve to initialize the symbol table so that references to abbreviated classes (`String` for `java.lang.String`) can be resolved.

8.4.5 Altered Search Rules

Pascal's `with` statement is a good example of a feature that alters the way in which symbols are found in the symbol table. If a Pascal program contains the phrase `with` *R* `do` *S*, then the statements in *S* first try to resolve references within the record *R*. If no valid reference is found in record *R*, then the symbol table is searched as usual. Inside the `with` block, fields that would otherwise be invisible are made visible. This feature prevents the programmer from frequently restating *R* inside *S*. Moreover, the compiler can usually generate faster code, since it is likely that record *R* is mentioned frequently in *S*.

Forward references also affect a symbol table's search rules. Consider a set of recursive data structures or methods. In the program, the set of definitions must be presented in some linear order. It is inevitable that a portion of the program will reference a definition that has not yet been processed. Forward references suspend the compiler's skepticism concerning undeclared symbols. A forward reference is essentially a promise that a complete definition will eventually be provided.

Some languages require that forward references be announced. In C, it is considered good style to *declare* an undefined function so that its types are known at forward references. In fact, some compilers require such declarations. On the other hand, a C structure may contain a field that is a pointer to itself. For example, each element in a linked list contains a pointer to another element. It is customary to process such forward references in two passes. The first pass makes note of types that should be checked in the second pass.

Summary

Although the interface for a symbol table is quite simple, the details underlying a symbol table's implementation play a significant role in the performance of the symbol table. Most modern programming languages are statically scoped. The symbol table organization presented in this chapter efficiently represents scope-declared symbols in a block-structured language. Each language places its own requirements on how symbols can be declared and used. Most languages include rules for symbol promotion to a global scope. Issues such as inheritance, over-loading, and aggregate data types should be considered when designing a symbol table.

Exercises

1. Consider a program in which the variable `x` is declared as a method's parameter *and* as one of the method's local variables. A programming language includes **parameter hiding** if the local variable's declaration can mask the parameter's declaration. Otherwise, the situation described in this exercise results in a multiply defined symbol. With regard to the symbol table interface presented in Section 8.1.2, explain the implicit scope-changing actions that must be taken if the language calls for

   (a) parameter hiding

   (b) no parameter hiding.

2. Code-generation sequences often require the materialization of temporary storage. For example, a complex arithmetic expression may require temporaries to hold intermediate values. Hopefully, machine registers can accommodate such short-lived values. However, compilers must sometimes plan to **spill** values out of registers into temporary storage.

   (a) Where are symbol table actions needed to allocate storage for temporaries?

   (b) How can implicit scoping serve to limit the effective lifetime of temporaries?

3. Consider the following C program.

```
int func() {
   int x, y;
   x = 10;
   {
      int x;
      x = 20;
      y = x;
   }
   return(x * y);
}
```

   The identifier `x` is declared in the method's outer scope. Another declaration occurs within the nested scope that assigns `y`. For C, the nested declaration of `x` could be regarded as a declaration of some other name, provided that the inner scope's reference to `x` is appropriately renamed. Design an AST pass that

   - Renames and moves nested variable declarations to the method's outermost scope

- Appropriately renames symbol references to preserve the meaning of the program

4. Recalling the discussion of Section 8.3.1, suppose that *all* active names are contained in a single, ordered list. An identifier would appear k times in the list if there are currently k active scopes that declare the identifier.

   (a) How would you implement the methods defined in Section 8.1.2 so that that RETRIEVESYMBOL finds the appropriate declaration of a given identifier?

   (b) Explain why the lookup time does or does not remain $O(\log(n))$ for a list of n entries.

5. Design and implement a symbol table using the ordered list data structure suggested in Section 8.3.1. Discuss the time required to perform each of the methods defined in Section 8.1.2.

6. Some languages contain names whose suffix is sufficient to locate the name. In Java, the classes in the package `java.lang` are available, so that the `java.lang.Integer` class can be referenced simply as `Integer`. This feature is also available for any explicitly imported classes. Similarly, Pascal's `with` statement allows field names to be abbreviated in the applicable blocks.

   Design a name space that supports the efficient retrieval of such abbreviated names, under the following conditions. Be sure to document any changes you wish to make to the symbol table interface given in Section 8.1.2.

7. Section 8.4.1 states that some languages allow a series of field references to be abbreviated, providing the abbreviation can uniquely locate the desired field. Certainly, the last field of such a reference must appear in the abbreviation. Moreover, the first field is necessary to distinguish the reference from other instances of the same type. We therefore assume that the first and last fields must appear in an abbreviation.

   As an example, the reference `a.b.c.d` could be abbreviated `a.d` if records `a` and `b` do not contain their own `d` field.

   Design an algorithm to allow such abbreviations, taking into consideration that the first and last fields of the reference cannot be omitted. Integrate your solution into RETRIEVESYMBOL in Section 8.3.3.

8. Consider a language whose field references are the reverse of those found in C and Pascal. A C reference to `a.b.c.d` would be specified as `d.c.b.a`. Describe the modifications you would make to your symbol table to accommodate such languages.

9. Program a symbol table manager using the interface described in Section 8.1.2 and the implementation described in Section 8.3.3.

10. Program a symbol table manager using the interface described in Section 8.1.2. Maintain the symbol table using *red-black trees* [?].

11. Program a symbol table manager using the interface described in Section 8.1.2. Maintain the symbol table using *splay trees* [?].

12. Describe how you would use the symbol table interface given in Section 8.1.2 to localize the declaration and effects of a loop iteration variable. As an example, consider the variable `i` in the Java-like statement

    ```
    for (int i=1; i<10; ++i) {...}.
    ```

    In this exercise, we seek the following.

    - The declaration of `i` could not possibly conflict with any other declaration.
    - The effects on this `i` are confined to the body of the loop. That is, the scope of `i` includes the expressions of the `for` statement as well as its body, represented above as {...}. The value of the loop's iteration variable is undefined when the loop exits.

13. As mentioned in Section 8.4.2, C allows the same identifier to appear as a `struct` name, a label, and an ordinary variable name. Thus, the following is a valid C program:

    ```
    main() {
      struct xxx {
        int a,b;
      } c;
      int xxx;

      xxx:
        c.a = 1;
    }
    ```

    In C, the structure name never appears without the terminal `struct` preceding it. The label can only be used as the target of a `goto` statement.

    Explain how to use the symbol table interface given in Section 8.1.2 to allow all three varieties of `xxx` to coexist in the same scope.

14. Using the symbol table interface given in Section 8.1.2, describe how to implement records (`structs` in C) under each of the following assumptions.

    (a) All records and fields are entered in a single symbol table.

    (b) A record is represented by its own symbol table, whose contents are the record's subfields.

15. Describe how to implement typenames (`typedef` in C) in a standard symbol table. Consider the following program fragment:

```
typedef
   struct {
     int x,y;
   } *Pair;

Pair *(pairs[23]);
```

The `typedef` establishes `Pair` as a typename, defined as a *pointer* to a record of two integers. The declaration for `pairs` uses the typename, but adds one more level of indirection to an array of 23 `Pairs`. Your design for `typedef` must be accommodate further type construction using `typedefs`.

16. Each string in Figure 8.4 occupies its own space in the string buffer. Suppose the strings were added in the following order: `i`, `input`, and `putter`. If the strings could share common characters, then these three strings can be represented using only 8 character slots.

    Design a symbol table string-space manager that allows strings to overlap, retaining the representation of each string as an offset and length pair, as described in Section 8.3.2.

17. The two data structures most commonly used to implement symbol tables in production compilers are binary search and hash tables. What are the advantages and disadvantages of using these data structures for symbol tables?

18. Describe two alternative approaches to handling multiple scopes in a symbol table, and list the actions required to open and close a scope for each alternative. Trace the sequence of actions that would be performed for each alternative during compilation of the program in Figure 8.1.

19. Figure 8.6 provides code to create a single symbol table for all scopes. Recalling the discussion of Section 8.2.2, an alternative approach segregates symbols by scope, as shown in Figure 8.8. Modify the code of Figure 8.6 to manage a stack of symbol tables, one for each active scope. Retain the symbol table interface as defined in Section 8.1.2.