Chapter 3
Scopes and Symbol Tables

The scope of thrift is limitless.

Thomas Edison (1847–1931)

An important concept in programming languages is the ability to *name* items such as variables, functions and types. Each such named item will have a *declaration*, where the name is defined as a synonym for the item and where properties of the item are described. This is called *binding*. Each name will also have a number of *uses*, where the name is used as a reference to the item to which it is bound.

Often, the declaration of a name has a limited *scope*: a portion of the program where the name will be visible. Such declarations are called *local declarations*, whereas a declaration that makes the declared name visible in the entire program is called *global*. It may happen that the same name is declared in several nested scopes. In this case, it is normal that the declaration closest to a use of the name will be the one that defines that particular use. In this context *closest* is related to the syntax tree of the program: The scope of a declaration will be a sub-tree of the syntax tree, and nested declarations will give rise to scopes that are nested sub-trees. The closest declaration of a name is, hence, the declaration corresponding to the smallest sub-tree that encloses the use of the name. As an example, look at this C statement block:

```
{
  int x = 1;
  int y = 2;
  {
    double x = 3.14159265358979;
    y += (int)x;
  }
  y += x;
}
```


T. Æ. Mogensen, *Introduction to Compiler Design*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-46460-7_3

The two lines immediately after the first opening brace declare integer variables `x` and `y` with scope until the closing brace in the last line. A new scope is started by the second opening brace, and a floating-point variable `x` with an initial value close to π is declared. This will have scope until the first closing brace, so the original `x` variable is not visible until the inner scope ends. The assignment `y += (int)x;` will add 3 to `y`, so its new value is 5. In the next assignment `y += x;`, we have exited the inner scope, so the original `x` is restored. The assignment will, hence, add 1 to `y`, which will have the final value 6.

Scoping based on the structure of the syntax tree, as shown in the example, is called *static* or *lexical* binding and is the most common scoping rule in modern programming languages. We will in the rest of this chapter (indeed, the rest of this book) assume that static binding is used. A few languages have *dynamic* binding, where the declaration that was most recently encountered during execution of the program defines the current use of the name. By its nature, dynamic binding can not be resolved at compile-time, so the techniques that in the rest of this chapter are described as being used in a compiler will have to be used at run-time if the language uses dynamic binding.

A compiler will need to keep track of names and the items these are bound to, so that any use of a name will be attributed correctly to its declaration. This is typically done using a *symbol table* (or *environment*, as it is sometimes called).

3.1 Symbol Tables

A symbol table is a table that binds names to properties of the items the names are bound to. We need a number of operations on symbol tables to accomplish this:

- We need an *empty* symbol table, in which no name is defined.
- We need to be able to *bind* a name to the properties of an item. In case the name is already defined in the symbol table, the new binding takes precedence over the old.
- We need to be able to *look up* a name in a symbol table to find the properties of the item to which the name is bound. If the name is not defined in the symbol table, we need to be told that.
- We need to be able to *enter* a new scope.
- We need to be able to *exit* a scope, reestablishing the symbol table to what it was before the scope was entered.

3.1.1 Implementation of Symbol Tables

There are many ways to implement symbol tables, but the most important distinction between these is how scopes are handled. This may be done using a *persistent* (or

functional) data structure, or it may be done using an *imperative* (or destructively-updated) data structure.

A persistent data structure has the property that no operation on the structure will destroy it. Conceptually, a new modified copy is made of the data structure whenever an operation updates it, hence preserving the old structure unchanged. This means that it is trivial to reestablish the old symbol table when exiting a scope, as it has been preserved by the persistent nature of the data structure. In practice, only a small portion of the data structure representing a persistent symbol table is copied when a modified symbol table is created, most of the structure is shared with the previous version.

In the imperative approach, only one copy of the symbol table exists, so explicit actions are required to store the information needed to restore the symbol table to a previous state. This can be done by using an auxiliary stack. When an update is made, the old binding of a name that is overwritten is recorded (pushed) on the auxiliary stack. When a new scope is entered, a marker is pushed on the auxiliary stack. When the scope is exited, the bindings on the auxiliary stack (down to the marker) are used to reestablish the old symbol table. The bindings and the marker are popped off the auxiliary stack in the process, returning the auxiliary stack to the state it was in before the scope was entered.

Below, we will look at simple implementations of both approaches and discuss more advanced approaches that are more efficient than the simple approaches.

3.1.2 Simple Persistent Symbol Tables

In functional languages like Standard ML, F#, Scheme or Haskell, persistent data structures are the norm rather than the exception (which is why persistent data structures are sometimes called *functional* data structures). For example, when a new element is added to the front of a list or an element is taken off the front of the list, the old list still exists and can be used elsewhere. A list is a natural way to implement a symbol table in a functional language: A binding is a pair of a name and its associated properties, and a symbol table is a list of such pairs. The operations are implemented in the following way:

empty: An empty symbol table is an empty list.

bind: A new binding (name/properties pair) is added (consed) to the front of the list.

lookup: The list is searched until a pair with a matching name is found. The properties paired with the name is then returned. If the end of the list is reached, an indication that this happened is returned instead. This indication can be made by raising an exception or by letting the lookup function return a special value representing “not found”. This requires a type that can hold both normal property information and this special value, i.e., an option type (Standard ML, F#) or a Maybe type (Haskell).

enter: The old list is remembered, i.e., a reference is made to it.
exit: The old list is recalled, i.e., the above reference is used.

The latter two operations are not really explicit operations, as the variable used to hold the symbol table before entering a new scope will still hold the same symbol table after the scope is exited. So all that is needed is a variable to hold (a reference to) the symbol table.

As new bindings are added to the front of the list and the list is searched from the front to the back, bindings in inner scopes will automatically take precedence over bindings in outer scopes.

Another functional approach to symbol tables is using functions: A symbol table is quite naturally seen as a function from names to information. The operations are:

empty: An empty symbol table is a function that returns an error indication (or raises an exception) no matter what its argument is.
bind: Adding a binding of the name `n` to the properties `p` in a symbol table t is done by defining a new symbol-table function t' in terms t and the new binding. When t' is called with a name `n1` as argument, it compares `n1` to `n`. If they are equal, t' returns the properties `p`. Otherwise, t' calls t with `n1` as argument and returns the result that this call yields. In Standard ML, we can define a binding function this way:

```
fun bind (n,p,t)
  = fn n1 => if n1=n then p else t n1
```

lookup: The symbol-table function is called with the name as argument.
enter: The old function is remembered (referenced).
exit: The old function is recalled (by using a reference).

Again, the latter two operations are mostly implicit.

3.1.3 A Simple Imperative Symbol Table

Imperative symbol tables are natural to use if the compiler is written in an imperative language. A simple imperative symbol table can be implemented as a stack, which works in a way similar to the list-based functional implementation:

empty: An empty symbol table is an empty stack.
bind: A new binding (name/properties pair) is pushed on top of the stack.
lookup: The stack is searched top-to-bottom until a matching name is found. The properties paired with the name is then returned. If the bottom of the stack is reached, we instead return an error-indication.
enter: We push a marker on the top of the stack.

exit: We pop bindings from the stack until a marker is found. This is also popped from the stack.

Note that since the symbol table is itself a stack, we don't need the auxiliary stack mentioned in Sect. 3.1.1.

This is not quite a persistent data structure, as leaving a scope will destroy its symbol table. For simple languages, this won't matter, as a scope isn't needed again after it is exited. But language features such as classes, modules and lexical closures can require symbol tables to persist after their scope is exited. In these cases, a real persistent symbol table must be used, or the needed parts of the symbol table must be copied and stored for later retrieval before exiting a scope.

3.1.4 Efficiency Issues

While all of the above implementations are simple, they all share the same efficiency problem: Lookup is done by linear search, so the worst-case time for lookup is proportional to the size of the symbol table. This is mostly a problem in relation to libraries: It is quite common for a program to use libraries that define literally hundreds of names.

A common solution to this problem is *hashing*: Names are hashed (processed) into integers, which are used to index an array. Each array element is then a linear list of the bindings of names that share the same hash code. Given a large enough hash table, these lists will typically be very short, so lookup time is basically constant.

Using hash tables complicates entering and exiting scopes somewhat. While each element of the hash table is a list that can be handled like in the simple cases, doing this for *all* the array-elements at every entry and exit imposes a major overhead. Instead, it is typical for imperative implementations to use a single auxiliary stack (as described in Sect. 3.1.1) to record all updates to the table so they can be undone in time proportional to the number of updates that were done in the local scope. Functional implementations typically use persistent hash-tables or persistent binary search trees, which eliminates the problem.

3.1.5 Shared or Separate Name Spaces

In some languages (like Pascal) a variable and a function in the same scope may have the same name, as the context of use will make it clear whether a variable or a function is used. We say that functions and variables have *separate name spaces*, which means that defining a name in one space doesn't affect the same name in the other space, even if the names occur in the same scope. In other languages (e.g. C, F#, or Standard ML) the context can not (easily) distinguish variables from functions. Hence, declaring a local variable will hide a function with the same name declared in

an outer scope or vice versa. These languages have a *shared name space* for variables and functions.

Name spaces may be shared or separate for all the kinds of names that can appear in a program, e.g., variables, functions, types, exceptions, constructors, classes, field selectors, etc. Which name spaces are shared is language-dependent.

Separate name spaces are easily implemented using one symbol table per name space, whereas shared name spaces naturally share a single symbol table. However, it is sometimes convenient to use a single symbol table even if there are separate name spaces. This can be done fairly easily by adding name-space indicators to the names. A name-space indicator can be a textual prefix to the name or it may be a tag that is paired with the name. In either case, a lookup in the symbol table must match both name and name-space indicator of the symbol that is looked up with both name and name-space indicator of the entry in the table.

Suggested exercises: 3.1.

3.2 Further Reading

Most algorithms-and-data-structures textbooks include descriptions of methods for hashing strings and implementing hash tables. A description of efficient persistent data structures for functional languages can be found in [1].

3.3 Exercises

Exercise 3.1 Pick some programming language that you know well and determine which of the following items share name spaces: Variables, functions, procedures and types. If there are more kinds of named items (labels, data constructors, modules, etc.) in the language, include these in the investigation.

Exercise 3.2 Implement, in a programming language of your choice, data structures and operations (empty, bind, lookup, enter and exit) for simple symbol tables.

Exercise 3.3 In some programming languages, identifiers are case-insensitive so, e.g., `size` and `SiZe` refer to the same identifier. Describe how symbol tables can be made case-insensitive.

Reference

1. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)