5.1 SYMBOL TABLES

This phase is characterized by the maintenance of *symbol tables* (also called *environments*) mapping identifiers to their types and locations. As the declarations of types, variables, and functions are processed, these identifiers are bound to "meanings" in the symbol tables. When *uses* (nondefining occurrences) of identifiers are found, they are looked up in the symbol tables.

Each local variable in a program has a *scope* in which it is visible. For example, in a Tiger expression `let` *D* `in` *E* `end` all the variables, types, and functions declared in *D* are visible only until the end of *E*. As the semantic analysis reaches the end of each scope, the identifier bindings local to that scope are discarded.

An environment is a set of *bindings* denoted by the $\mapsto$ arrow. For example, we could say that the environment σ_0 contains the bindings {g $\mapsto$ string, a $\mapsto$ int}; meaning that the identifier a is an integer variable and g is a string variable.

Consider a simple example in the Tiger language:

```
1        function f(a:int, b:int, c:int) =
2           (print_int(a+c);
3            let var j := a+b
4                var a := "hello"
5             in print(a); print_int(j)
6            end;
7            print_int(b)
8           )
```

Suppose we compile this program in the environment σ_0. The formal parameter declarations on line 1 give us the table σ_1 equal to σ_0 + {a $\mapsto$ int, b $\mapsto$ int, c $\mapsto$ int}, that is, σ_0 extended with new bindings for a, b, and c. The identifiers in line 2 can be looked up in σ_1. At line 3, the table $\sigma_2 = \sigma_1$ + {j $\mapsto$ int} is created; and at line 4, $\sigma_3 = \sigma_2$ + {a $\mapsto$ string} is created.

How does the + operator for tables work when the two environments being "added" contain different bindings for the same symbol? When σ_2 and {a $\mapsto$ String} map a to int and string, respectively? To make the scoping rules work the way we expect them to in real programming languages, we want {a $\mapsto$ String} to take precedence. So we say that *X* + *Y* for tables is not the same as *Y* + *X*; bindings in the right-hand table override those in the left.

Finally, in line 6 we discard σ_3 and go back to σ_1 for looking up the identifier b in line 7. And at line 8, we discard σ_1 and go back to σ_0.

How should this be implemented? There are really two choices. In a *functional* style, we make sure to keep σ_1 in pristine condition while we create σ_2 and σ_3. Then when we need σ_1 again, it's rested and ready.

In an *imperative* style, we modify σ_1 until it becomes σ_2. This *destructive update* "destroys" σ_1; while σ_2 exists, we cannot look things up in σ_1. But when we are done with σ_2, we can *undo* the modification to get σ_1 back again. Thus, there is a single global environment σ which becomes σ_0, σ_1, σ_2, σ_3,σ_1,σ_0 at different times and an "undo stack" with enough information to remove the destructive updates. When a symbol is added to the environment, it is also added to the undo stack; at the end of scope (e.g., at line 6 or 8), symbols popped from the undo stack have their latest binding removed from σ (and their previous binding restored).

Either the functional or imperative style of environment management can be used regardless of whether the language being compiled, or the implementation language of the compiler, is a "functional" or "imperative" or "object-oriented" language.

MULTIPLE SYMBOL TABLES

In some languages there can be several active environments at once: each module, or class, or record, in the program has a symbol table σ of its own.

In analyzing Figure 5.1, let σ_0 be the base environment containing predefined functions, and let

$$\sigma_1 = \{a \mapsto int\}$$
$$\sigma_2 = \{E \mapsto \sigma_1\}$$
$$\sigma_3 = \{b \mapsto int, a \mapsto int\}$$
$$\sigma_4 = \{N \mapsto \sigma_3\}$$
$$\sigma_5 = \{d \mapsto int\}$$
$$\sigma_6 = \{D \mapsto \sigma_5\}$$
$$\sigma_7 = \sigma_2 + \sigma_4 + \sigma_6$$

In ML, the *N* is compiled using environment $\sigma_0 + \sigma_2$ to look up identifiers; *D* is compiled using $\sigma_0 + \sigma_2 + \sigma_4$, and the result of the analysis is $\{M \longmapsto \sigma_7\}$.

```
structure M = struct
   structure E = struct
      val a = 5;
   end
   structure N = struct
      val b = 10
      val a = E.a + b
   end
   structure D = struct
      val d = E.a + N.a
   end
end
```

(a) An example in ML

```
package M;
class E {
    static int a = 5;
}
class N {
    static int b = 10;
    static int a = E.a + b;
}
class D {
    static int d = E.a + N.a;
}
```

(b) An example in Java

FIGURE 5.1. Several active environments at once.

In Java, forward reference is allowed (so inside *N* the expression *D.d* would be legal), so *E*, *N*, and *D* are all compiled in the environment σ_7; for this program the result is still $\{M \longmapsto \sigma_7\}$.

EFFICIENT IMPERATIVE SYMBOL TABLES

Because a large program may contain thousands of distinct identifiers, symbol tables must permit efficient lookup.

Imperative-style environments are usually implemented using hash tables, which are very efficient. The operation $\sigma' = \sigma + \{a \longmapsto \tau\}$ is implemented by inserting τ in the hash table with key *a*. A simple *hash table with external chaining* works well and supports deletion easily (we will need to delete $\{a \longmapsto \tau\}$ to recover σ at the end of the scope of *a*).

Program 5.2 implements a simple hash table. The *i*th bucket is a linked list of all the elements whose keys hash to *i* mod `SIZE`. The type `binding` could be any type; in a real program this hash-table module might be a functor, or the `table` type might be polymorphic.

```
val SIZE = 109    should be prime
type binding = ···
type bucket = (string * binding) list
type table = bucket Array.array
val t : table = Array.array(SIZE, nil)

fun hash(s: string) : int =
    CharVector.foldl (fn (c,n)=> (n*256+ord(c)) mod SIZE) 0 s

fun insert(s: string, b: binding) =
   let val i = hash(s) mod SIZE
    in Array.update(t,i,(s,b)::Array.sub(t,i))
   end

exception NotFound

fun lookup(s: string) =
   let val i = hash(s) mod SIZE
       fun search((s',b)::rest) = if s=s' then b
                                          else search rest
         | search nil = raise NotFound
    in search(Array.sub(t,i))
   end

fun pop(s: string) =
   let val i = hash(s) mod SIZE
       val (s',b)::rest = Array.sub(t,i)
    in assert(s=s');
       Array.update(t,i,rest)
   end
```

PROGRAM 5.2. Hash table with external chaining.

Consider $\sigma + \{a \rightarrowtail \tau_2\}$ when σ contains $\{a \rightarrowtail \tau_1\}$ already. The `insert` function leaves $\{a \rightarrowtail \tau_1\}$ in the bucket and puts $\{a \rightarrowtail \tau_2\}$ earlier in the list. Then, when `pop`(*a*) is done at the end of *a*'s scope, σ is restored. Of course, `pop` works only if bindings are inserted and popped in a stacklike fashion.

An industrial-strength implementation would improve on this in several ways; see Exercise 5.1.

EFFICIENT FUNCTIONAL SYMBOL TABLES

In the functional style, we wish to compute $\sigma' = \sigma + \{a \rightarrowtail \tau\}$ in such a way that we still have σ available to look up identifiers. Thus, instead of "altering" a table by adding a binding to it we create a new table by computing the "sum" of an existing table and a new binding. Similarly, when we add 7 + 8 we don't alter the 7 by adding 8 to it; we create a new value 15 – and the 7 is still available for other computations.

However, nondestructive update is not efficient for hash tables. Figure 5.3a shows a hash table implementing mapping m_1. It is fast and efficient to add *mouse* to the fifth slot; just make the *mouse* record point at the (old) head of the fifth linked list, and make the fifth slot point to the *mouse* record. But then we no longer have the mapping m_1: we have destroyed it to make m_2. The other alternative is to copy the array, but still share all the old buckets, as shown in Figure 5.3b. But this is not efficient: the array in a hash table should be quite large, proportional in size to the number of elements, and we cannot afford to copy it for each new entry in the table.

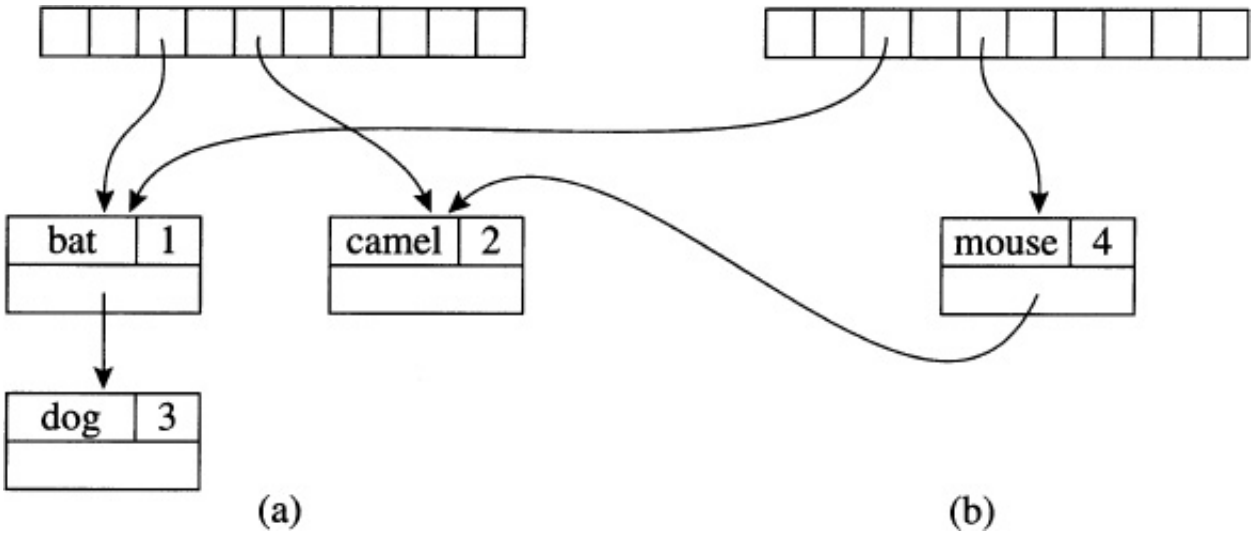

FIGURE 5.3. Hash tables.

By using binary search trees we can perform such "functional" additions to search trees efficiently. Consider, for example, the search tree in Figure 5.4, which represents the mapping

$m_1 = \{bat \mapsto 1, camel \mapsto 2, dog \mapsto 3\}$.

We can add the binding $mouse \mapsto 4$, creating the mapping m_2 without destroying the mapping m_1, as shown in Figure 5.4b. If we add a new node at depth d of the tree, we must create d new nodes – but we don't need to copy the whole tree. So creating a new tree (that shares some structure with the old one) can be done as efficiently as looking up an element: in $\log(n)$ time for a balanced tree of n nodes. This is an example of a *persistent data structure;* a persistent *red-black* tree can be kept balanced to guarantee $\log(n)$ access time (see Exercise 1.1c, and also page 286).

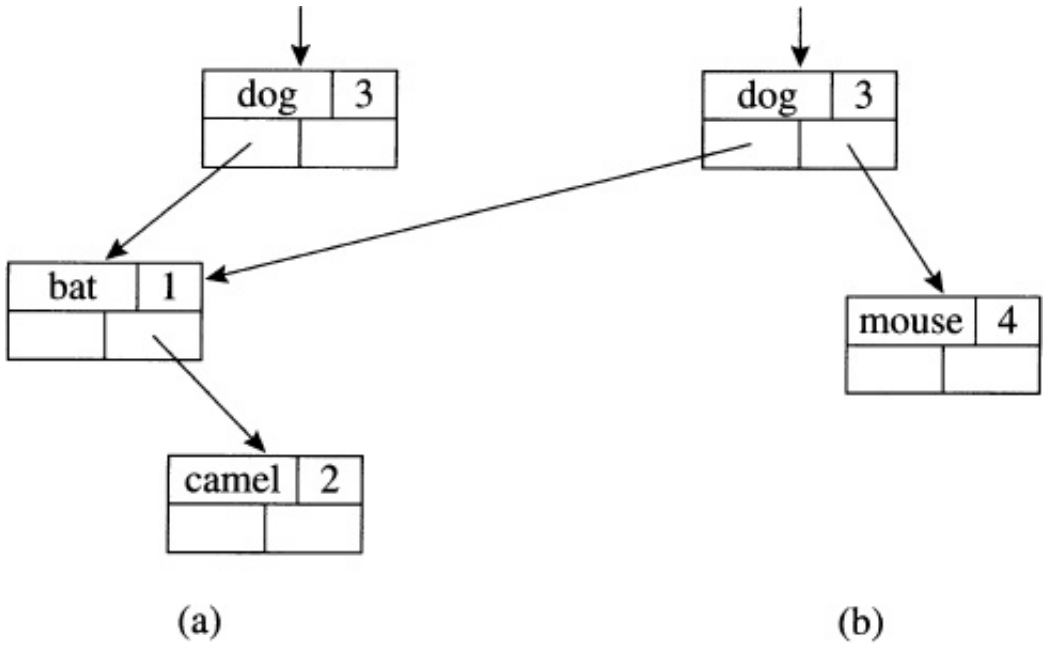

FIGURE 5.4. Binary search trees.

SYMBOLS IN THE Tiger COMPILER

The hash table of Program 5.2 must examine every character of the string s for the `hash` operation, and then again each time it compares s against a string in the ith bucket. To avoid unnecessary string comparisons, we can convert each string to a `symbol`, so that all the different occurrences of any given string convert to the same symbol object.

The `Symbol` module implements symbols and has these important properties:

- Comparing two symbols for equality is very fast (just pointer or integer comparison).
- Extracting an integer hash-key is very fast (in case we want to make hash table mapping symbols to something else).
- Comparing two symbols for "greater-than" (in some arbitrary ordering) is very fast (in case we want to make binary search trees).

Even if we intend to make functional-style environments mapping symbols to bindings, we can use a destructive-update hash table to map strings to symbols: we need this to make sure the second occurrence of "abc" maps to the same symbol as the first occurrence. Program 5.5 shows the interface of the `Symbol` module.

```
signature SYMBOL =
sig
  eqtype symbol
  val symbol : string -> symbol
  val name : symbol -> string

  type 'a table
  val empty : 'a table
  val enter : 'a table * symbol * 'a -> 'a table
  val look  : 'a table * symbol -> 'a option
end
```

PROGRAM 5.5. Signature `SYMBOL`.

Environments are implemented in the `Symbol` module as `tables` mapping `symbols` to bindings. We want different notions of `binding` for different purposes in the compiler – type bindings for types, value bindings for variables and functions – so we let `table` be a polymorphic type. That is, an *a* `table` is a mapping from `symbol` to *a*, whether *a* is a type binding, or a value binding, or any other kind of binding.

Given a table, new bindings can be added (creating a new table, without altering the old table) using `enter`. If there is a binding of the same symbol in the old table, it will be replaced by the new one.

The `look(t, s)` function finds the binding *b* of symbol *s* in table *t*, returning `SOME`(*b*). If the symbol is not bound in that table, `NONE` is returned.

The implementation of the `Symbol` abstraction (Program 5.6) has been designed to make symbol tables efficient. To make "functional mappings" such as *a* `table` efficient, we use balanced binary search trees. The search trees are implemented in the `IntBinaryMap` module of the *Standard ML of New Jersey Library* (see the Library manual for details).

```
structure Symbol :> SYMBOL =
struct
  type symbol = string * int

  exception Symbol
  val nextsym = ref 0
  val hashtable : (string,int) HashTable.hash_table =
     HashTable.mkTable(HashString.hashString, op = ) (128,Symbol)

  fun symbol name =
      case HashTable.find hashtable name
       of SOME i => (name,i)
        | NONE => let val i = !nextsym
                   in nextsym := i+1;
                      HashTable.insert hashtable (name,i);
                      (name,i)
                  end

  fun name(s,n) = s

  type 'a table= 'a IntBinaryMap.map
  val empty = IntBinaryMap.empty
  fun enter(t: 'a table, (s,n): symbol, a: 'a) = IntBinaryMap.insert(t,n,a)
  fun look(t: 'a table, (s,n): symbol) = IntBinaryMap.look(t,n)
end
```

PROGRAM 5.6. Symbol table implementation. `HashTable` and `IntBinary–Map come` from the *Standard ML of New Jersey Library*.

Binary search trees require a total ordering function ("less than") on the `symbol` type. Strings have a total ordering function, and thus could be used as symbols, but the comparison function on strings is not very fast.

Instead, symbols contain integers for use in the "less than" comparison. The ordering of symbols is totally arbitrary – it is based on the order in which different strings are seen in the file – but that doesn't matter to the search tree algorithm. The internal representation of a symbol is a pair `string` × `int`, where the `string` component is used only to implement the `name` function of the interface.

Different strings must have different integers. The `Symbol` module can keep a count of how many distinct strings it has seen so far; for each never-before-seen string it just uses the current count value as the integer for that string; and for each previously seen string it uses the same integer it used previously. A conventional hash table with destructive update can be used to look up the integer for a string, because the `Symbol` module will never need a previous version of the `string→symbol` mapping. The `Symbol` module keeps all the destructive-update side-effect nastiness hidden from its clients by the simple nature of the `symbol` interface.

IMPERATIVE-STYLE SYMBOL TABLES

If we wanted to use destructive-update tables, the "table" portion of the `SYMBOL` signature would look like

```
type 'a table
val new : unit -> 'a table
val enter : 'a table * symbol * 'a -> unit
val look  : 'a table * symbol -> 'a option

val beginScope: 'a table -> unit
val endScope : 'a table -> unit
```

To handle the "undo" requirements of destructive update, the interface function `beginScope` remembers the current state of the table, and `endScope` restores the table to where it was at the most recent `beginScope` that has not already been ended.

An imperative table is implemented using a hash table. When the binding $x \longmapsto b$ is entered, x is hashed into an index i and $x \longmapsto b$ is placed at the head of the linked list for the ith bucket. If the table had already contained a binding $x \longmapsto b'$, that would still be in the bucket, hidden by $x \longmapsto b$. This is important because it will support the implementation of *undo* (`beginScope` and `endScope`).

We also need an auxiliary stack, showing in what order the symbols are "pushed" into the symbol table. When $x \mapsto b$ is entered, then x is pushed onto this stack. A `beginScope` operation pushes a special marker onto the stack. Then, to implement `endScope`, symbols are popped off the stack down to and including the topmost marker. As each symbol is popped, the head binding in its bucket is removed.