5 Semantic Analysis

se-man-tic: of or relating to meaning in language

Webster's Dictionary

The *semantic analysis* phase of a compiler connects variable definitions to their uses, checks that each expression has a correct type, and translates the abstract syntax into a simpler representation suitable for generating machine code.

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 $\sigma_0 + \{\texttt{a} \mapsto \texttt{int}, \texttt{b} \mapsto \texttt{int}, \texttt{c} \mapsto \texttt{int}\}$, 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 $\sigma_2 = \sigma_1 + \{\texttt{j} \mapsto \texttt{int}\}$ is created; and at line 4, $\sigma_3 = \sigma_2 + \{\texttt{a} \mapsto \texttt{string}\}$ 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 $\{\texttt{a} \mapsto \texttt{string}\}$ 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 $\{\texttt{a} \mapsto \texttt{string}\}$ 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 $\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_1, \sigma_0$ 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.

```
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.

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

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

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 $\{M \mapsto \sigma_7\}$.

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 $\{M \mapsto \sigma_7\}$.

EFFICIENT IMPERATIVE SYMBOL TABLES

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

```
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.

Imperative-style environments are usually implemented using hash tables, which are very efficient. The operation $\sigma' = \sigma + \{a \mapsto \tau\}$ 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 $\{a \mapsto \tau\}$ to recover σ at the end of the scope of a).

Program 5.2 implements a simple hash table. The ith 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.

Consider $\sigma + \{a \mapsto \tau_2\}$ when σ contains $a \mapsto \tau_1$ already. The `insert` function leaves $a \mapsto \tau_1$ in the bucket and puts $a \mapsto \tau_2$ earlier in the list.

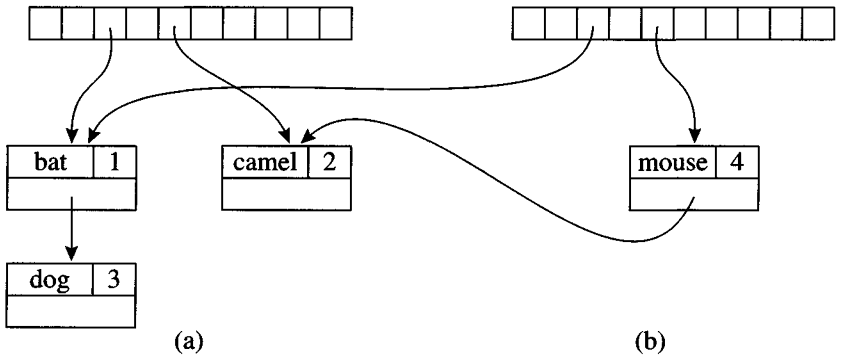

FIGURE 5.3. Hash tables.

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 $\sigma' = \sigma + \{a \mapsto \tau\}$ 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.

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

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

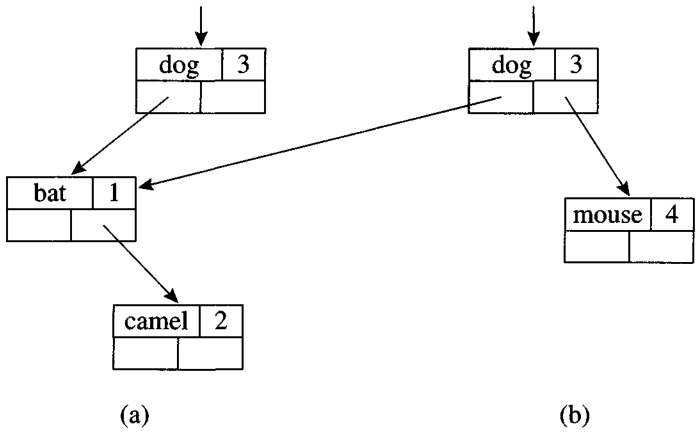

FIGURE 5.4. Binary search trees.

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).

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).

```
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`.

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.

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 α `table` is a mapping from `symbol` to α, whether α 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 α `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).

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

```
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 `IntBinaryMap` come from the *Standard ML of New Jersey Library*.

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 $x \mapsto b$ is entered, x is hashed into an index i and $x \mapsto b$ is placed at the head of the linked list for the ith bucket. If the table had already contained a binding $x \mapsto b'$, that would still be in the bucket, hidden by $x \mapsto b$. 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.

5.2 BINDINGS FOR THE Tiger COMPILER

With what should a symbol table be filled – that is, what is a binding? Tiger has two separate name spaces, one for types and the other for functions and variables. A type identifier will be associated with a Types.ty. The Types module describes the structure of types, as shown in Program 5.7.

The primitive types in Tiger are int and string; all types are either primitive types or constructed using records and arrays from other (primitive, record, or array) types.

Record types carry additional information: the names and types of the

```
structure Types =
struct
  type unique = unit ref

  datatype ty = INT
              | STRING
              | RECORD of (Symbol.symbol * ty) list * unique
              | ARRAY of ty * unique
              | NIL
              | UNIT
              | NAME of Symbol.symbol * ty option ref
end
```

PROGRAM 5.7. Structure `Types`.

fields. Furthermore, since every "record type expression" creates a new (and different) record type, even if the fields are similar, we have a "unique" value to distinguish it. (The only interesting thing you can do with a `unit ref` is to test it for equality with another one; each `ref` is unique.)

Arrays work just like records: the `ARRAY` constructor carries the type of the array elements, and also a "unique" value to distinguish this array type from all others.

If we were compiling some other language, we might have the following as a legal program:

```
let type a = {x: int, y: int}
    type b = {x: int, y: int}
    var i : a := ···
    var j : b := ···
in i := j
end
```

This is illegal in Tiger, but would be legal in a language where structurally equivalent types are interchangeable. To test type equality in a compiler for such a language, we would need to examine record types field by field, recursively.

However, the following Tiger program is legal, since type `c` is the same as type `a`:

```
let type a = {x: int, y: int}
    type c = a
    var i : a := ···
    var j : c := ···
in i := j
end
```

It is not the type *declaration* that causes a new and distinct type to be made, but the type *expression* `{x:int,y:int}`.

In Tiger, the expression `nil` belongs to any record type. We handle this exceptional case by inventing a special "nil" type. There are also expressions that return "no value," so we invent a type `unit`.

When processing mutually recursive types, we will need a place-holder for types whose name we know but whose definition we have not yet seen. The type NAME(sym, ref(SOME(t))) is equivalent to type t; but NAME(sym, ref(NONE)) is just the place-holder.

ENVIRONMENTS

The `table` type of the `Symbol` module provides mappings from symbols to bindings. Thus, we will have a *type environment* and a *value environment*. The following Tiger program demonstrates that one environment will not suffice:

```
let type a = int
    var a : a := 5
    var b : a := a
 in b+a
end
```

The symbol `a` denotes the type "a" in syntactic contexts where type identifiers are expected, and the variable "a" in syntactic contexts where variables are expected.

For a type identifier, we need to remember only the type that it stands for. Thus a type environment is a mapping from symbol to `Types.ty` – that is, a `Types.ty Symbol.table`. As shown in Figure 5.8, the `Env` module will contain a `base_tenv` value – the "base" or "predefined" type environment. This maps the symbol `int` to `Ty.INT` and `string` to `Ty.STRING`.

We need to know, for each value identifier, whether it is a variable or a function; if a variable, what is its type; if a function, what are its parameter and result types, and so on. The type `enventry` holds all this information, as shown in Figure 5.8; and a value environment is a mapping from symbol to environment-entry.

```
signature ENV =
sig
  type access
  type ty
  datatype enventry = VarEntry of {ty: ty}
                    | FunEntry of {formals: ty list, result: ty}
  val base_tenv : ty Symbol.table        (* predefined types *)
  val base_venv : enventry Symbol.table  (* predefined functions *)
end
```

FIGURE 5.8. Environments for type-checking.

A variable will map to a `VarEntry` telling its type. When we look up a function we will obtain a `FunEntry` containing:

`formals` The types of the formal parameters.
`result` The type of result returned by the function (or `UNIT`).

For type-checking, only `formals` and `result` are needed; we will add other fields later for translation into intermediate representation.

The `base_venv` environment contains bindings for predefined functions `flush`, `ord`, `chr`, `size`, and so on, described in Appendix A.

Environments are used during the type-checking phase.

As types, variables, and functions are declared, the type-checker augments the environments; they are consulted for each identifier that is found during processing of expressions (type-checking, intermediate code generation).

5.3 TYPE-CHECKING EXPRESSIONS

The structure `Semant` performs semantic analysis – including type-checking – of abstract syntax. It contains four functions that recur over syntax trees:

```
type venv = Env.enventry Symbol.table
type tenv = ty Symbol.table

transVar: venv * tenv * Absyn.var -> expty
transExp: venv * tenv * Absyn.exp -> expty
transDec: venv * tenv * Absyn.dec -> {venv: venv, tenv: tenv}
transTy:         tenv * Absyn.ty  -> Types.ty
```

The type-checker is a recursive function of the abstract syntax tree. I will call it `transExp` because we will later augment this function not only to type-check but also to translate the expressions into intermediate code. The arguments of `transExp` are a value environment `venv`, a type environment `tenv`, and an expression. The result will be an `expty`, containing a translated expression and its Tiger-language type:

```
type expty = {exp: Translate.exp, ty: Types.ty}
```

where `Translate.exp` is the translation of the expression into intermediate code, and `ty` is the type of the expression.

To avoid a discussion of intermediate code at this point, let us define a dummy `Translate` module:

```
structure Translate = struct type exp = unit end
```

and use `()` for every `exp` value. We will flesh out the `Translate.Exp` type in Chapter 7.

Let's take a very simple case: an addition expression $e_1 + e_2$. In Tiger, both operands must be integers (the type-checker must check this) and the result will be an integer (the type-checker will return this type).

In most languages, addition is *overloaded*: the + operator stands for either integer addition or real addition. If the operands are both integers, the result is integer; if the operands are both real, the result is real. And in many languages if one operand is an integer and the other is real, the integer is implicitly converted into a real, and the result is real. Of course, the compiler will have to make this conversion explicit in the machine code it generates.

Tiger's nonoverloaded type-checking is easy to implement:

```
fun transExp(venv,tenv,
             Absyn.OpExp{left,oper=Absyn.PlusOp,right,pos})=
    let val {exp=_, ty=tyleft} = transExp(venv,tenv,left)
        val {exp=_, ty=tyright} = transExp(venv,tenv,right)
     in case tyleft of Types.INT => ()
                     | _ => error pos "integer required";
        case tyright of Types.INT => ()
                      | _ => error pos "integer required";
        {exp=(), ty=Types.INT}
    end
```

This works well enough, although we have not yet written the cases for other kinds of expressions (and operators other than +), so when the recursive calls on `left` and `right` are executed, a `Match` exception will be raised. You can fill in the other cases yourself (see page 121).

It's also a bit clumsy. Most of the recursive calls to `transExp` will pass the exact same `venv` and `tenv`, so we can factor them out using nested functions. The case of checking for an integer type is common enough to warrant a function definition, `checkInt`. We can use a local structure definition to abbreviate a frequently used structure name such as `Absyn`. A cleaned-up version of `transExp` looks like:

```
structure A = Absyn

fun checkInt ({exp,ty},pos) = (...)

fun transExp(venv,tenv) =
  let fun trexp (A.OpExp{left,oper=A.PlusOp,right,pos}) =
                (checkInt(trexp left, pos);
                 checkInt(trexp right, pos);
                 {exp=(),ty=Types.INT})
        | trexp (A.RecordExp ...)  ...
```

TYPE-CHECKING VARIABLES, SUBSCRIPTS, AND FIELDS

The function `trexp` recurs over `Absyn.exp`, and `trvar` recurs over `Absyn.var`; both these functions are nested within `transExp` and access `venv` and `tenv` from `transExp`'s formal parameters. In the rare cases where `trexp` wants to change the `venv`, it must call `transExp` instead of just `trexp`.

```
      and trvar (A.SimpleVar(id,pos)) =
            (case Symbol.look(venv,id)
              of SOME(E.VarEntry{ty}) =>
                  {exp=(), ty=actual_ty ty}
               | NONE => (error pos ("undefined variable "
                                       ^ S.name id);
                         exp=(), ty=Types.INT))
        | trvar (A.FieldVar(v,id,pos)) = ...
   in trexp
  end
```

The clause of `trvar` that type-checks a `SimpleVar` illustrates the use of environments to look up a variable binding. If the identifer is present in the environment *and* is bound to a `VarEntry` (not a `FunEntry`), then its type is the one given in the `VarEntry` (Figure 5.8).

The type in the `VarEntry` will sometimes be a "`NAME` type" (Program 5.7), and all the types returned from `transExp` should be "actual" types (with the names traced through to their underlying definitions). It is therefore useful to have a function, perhaps called `actual_ty`, to skip past all the `NAME`s. The result will be a `Types.ty` that is not a `NAME`, though if it is a record or array type it might contain `NAME` types to describe its components.

For function calls, it is necessary to look up the function identifier in the environment, yielding a `FunEntry` containing a list of parameter types. These types must then be matched against the arguments in the function-call expression. The `FunEntry` also gives the result type of the function, which becomes the type of the function call as a whole.

Every kind of expression has its own type-checking rules, but in all the cases I have not already described the rules can be derived by reference to the *Tiger Language Reference Manual* (Appendix A).

5.4 TYPE-CHECKING DECLARATIONS

Environments are constructed and augmented by declarations. In Tiger, declarations appear only in a `let` expression. Type-checking a `let` is easy enough, using `transDec` to translate declarations:

```
| trexp(A.LetExp{decs,body,pos}) =
    let val {venv=venv',tenv=tenv'} =
                transDecs(venv,tenv,decs)
     in transExp(venv',tenv') body
    end
```

Here `transExp` augments the environments `(venv,tenv)` to produce new environments `venv'`, `tenv'` which are then used to translate the body expression. Then the new environments are discarded.

If we had been using the imperative style of environments, `transDecs` would add new bindings to a global environment, altering it by side effect. First `beginScope()` would mark the environments, and then after the recursive call to `transExp` these bindings would be removed from the global environment by `endScope()`.

VARIABLE DECLARATIONS

In principle, processing a declaration is quite simple: a declaration augments an environment by a new binding, and the augmented environment is used in the processing of subsequent declarations and expressions.

The only problem is with (mutually) recursive type and function declarations. So we will begin with the special case of nonrecursive declarations.

For example, it is quite simple to process a variable declaration without a type constraint, such as `var x := ` *exp*.

```
fun transDec (venv,tenv,A.VarDec{name,typ=NONE,init,...}) =
    let val {exp,ty} = transExp(venv,tenv,init)
     in {tenv=tenv,
         venv=S.enter(venv,name,E.VarEntry{ty=ty})}
    end
```

What could be simpler? In practice, if `typ` is present, as in

```
var x : type-id := exp
```

it will be necessary to check that the constraint and the initializing expression are compatible. Also, initializing expressions of type `NIL` must be constrained by a `RECORD` type.

TYPE DECLARATIONS

Nonrecursive type declarations are not too hard:

```
  | transDec (venv,tenv,A.TypeDec[{name,ty}]) =
      {venv=venv,
       tenv=S.enter(tenv,name,transTy(tenv,ty))}
```

The `transTy` function translates type expressions as found in the abstract syntax (`Absyn.ty`) to the digested type descriptions that we will put into environments (`Types.ty`). This translation is done by recurring over the structure of an `Absyn.ty`, turning `Absyn.RecordTy` into `Types.RECORD`, etc. While translating, `transTy` just looks up any symbols it finds in the type environment `tenv`.

The pattern `[{name,ty}]` is not very general, since it handles only a type-declaration list of length 1, that is, a singleton list of mutually recursive type declarations. The reader is invited to generalize this to lists of arbitrary length.

FUNCTION DECLARATIONS

Function declarations are a bit more tedious:

```
| transDec(venv,tenv,
          A.FunctionDec[{name,params,body,pos,
                        result=SOME(rt,pos)}]) =
    let val SOME(result_ty) = S.look(tenv,rt)
        fun transparam{name,typ,pos} =
                         case S.look(tenv,typ)
                          of SOME t => {name=name,ty=t}
        val params' = map transparam params
        val venv' = S.enter(venv,name,
                    E.FunEntry{formals= map #ty params',
                               result=result_ty})
        fun enterparam ({name,ty},venv) =
                   S.enter(venv,name,
                          E.VarEntry{access=(),ty=ty})
        val venv'' = fold enterparam params' venv'
     in transExp(venv'',tenv) body;
        {venv=venv',tenv=tenv}
    end
```

This is a very stripped-down implementation: it handles only the case of a single function; it does not handle recursive functions; it handles only a function with a result (a function, not a procedure); it doesn't handle program errors such as undeclared type identifiers, etc; and it doesn't check that the type of the body expression matches the declared result type.

So what does it do? Consider the Tiger declaration

function f(a: ta, b: tb) : rt = *body*.

First, `transDec` looks up the result-type identifier `rt` in the type environment. Then it calls the local function `transparam` on each formal parameter; this yields a list of pairs, (a, t_a), (b, t_b) where t_a is the `NAME` type found by looking up `ta` in the type environment. Now `transDec` has enough information to construct the `FunEntry` for this function and enter it in the value environment, yielding a new environment `venv'`.

Next, the formal parameters are entered (as `VarEntrys`) into `venv'`, yielding `venv''`; this environment is used to process the *body* (with the `transExp` function). Finally, `venv''` is discarded, and `{venv',tenv}` is the result: this environment will be used for processing expressions that are allowed to call the function `f`.

RECURSIVE DECLARATIONS

The implementations above will not work on recursive type or function declarations, because they will encounter undefined type or function identifiers (in `transTy` for recursive record types or `transExp(body)` for recursive functions).

The solution for a set of mutually recursive things (types or functions) $t_1, \ldots, t_n$ is to put all the "headers" in the environment first, resulting in an environment e_1. Then process all the "bodies" in the environment e_1. During processing of the bodies it will be necessary to look up some of the newly defined names, but they will in fact be there – though some of them may be empty headers without bodies.

What is a header? For a type declaration such as

```
type list = {first: int, rest: list}
```

the header is approximately `type list =`.

To enter this header into an environment `tenv` we can use a `NAME` type with an empty binding (`ty option`):

```
tenv' = S.enter(tenv,name,Types.NAME(name,ref NONE))
```

Now, we can call `transTy` on the "body" of the type declaration, that is, on the record expression `{first: int, rest: list}`. The environment we give to `transTy` will be `tenv'`.

It's important that `transTy` stop as soon as it gets to any `NAME` type. If, for example, `transTy` behaved like `actual_ty` and tried to look "through" the `NAME` type bound to the identifier `list`, all it would find (in this case) would be `NONE` – which it is certainly not prepared for. This `NONE` can be replaced only by a valid type after the entire `{first:int, rest:list}` is translated.

The type that `transTy` returns can then be assigned into the reference variable within the `NAME` constructor. Now we have a fully complete type environment, on which `actual_ty` will not have a problem.

The assignments to `ref` variables (in the `NAME` type descriptors) mean that `Translate` is not a "purely functional" program. Any use of side effects adds to the difficulty of writing and understanding a program, but in this case a limited use of side effects seems reasonable.

Every cycle in a set of mutually recursive type declarations must pass through a record or array declaration; the declaration

```
type a = b
type b = d
type c = a
type d = a
```

contains an illegal cycle $a \to b \to d \to a$. Illegal cycles should be detected by the type-checker.

Mutually recursive functions are handled similarly. The first pass gathers information about the *header* of each function (function name, formal parameter list, return type) but leaves the bodies of the functions untouched. In this pass, the *types* of the formal parameters are needed, but not their names (which cannot be seen from outside the function).

The second pass processes the bodies of all functions in the mutually recursive declaration, taking advantage of the environment augmented with all the function headers. For each body, the formal parameter list is processed again, this time entering the parameters as `VarEntrys` in the value environment.

PROGRAM TYPE-CHECKING

Write a type-checking phase for your compiler, a module `Semant` containing a function

```
transProg : Absyn.exp → unit
```

that type-checks an abstract syntax tree and produces any appropriate error messages about mismatching types or undeclared identifiers.

Also provide the implementation of the `Env` module described in this chapter. Make a module `Main` that calls the parser, yielding an `Absyn.exp`, and then calls `transProg` on this expression.

You must use precisely the `Absyn` interface described in Figure 4.8, but you are free to follow or ignore any advice given in this chapter about the internal organization of the `Semant` module.

You'll need your parser that produces abstract syntax trees. In addition, supporting files available in `$TIGER/chap5` include:

`types.sml` Describes data types of the Tiger language.

and other files as before. Modify the `sources.cm` file from the previous exercise as necessary.

Part a. Implement a simple type-checker and declaration processor that does not handle recursive functions or recursive data types (forward references to functions or types need not be handled). Also don't bother to check that each **break** statement is within a **for** or **while** statement.

Part b. Augment your simple type-checker to handle recursive (and mutually recursive) functions; (mutually) recursive type declarations; and correct nesting of **break** statements.

EXERCISES

5.1 Improve the hash table implementation of Program 5.2:

a. Double the size of the array when the average bucket length grows larger than 2 (so `table` is now `ref(array)`). To double an array, allocate a bigger one and rehash the contents of the old array; then discard the old array.

b. Allow for more than one table to be in use by making the table a parameter to `insert` and `lookup`.

c. Hide the representation of the `table` type inside an abstraction module, so that clients are not tempted to manipulate the data structure directly (only through the `insert`, `lookup`, and `pop` operations).

d. Use a faster `hash` function that does not compute an expensive `mod` for each character.

e. Instead of a fixed `binding` type, allow tables of "anything" by changing the `table` type to `'a table` (and change `bucket` to `'a bucket`).

*****5.2** In many applications, we want a + operator for environments that does more than add one new binding; instead of $\sigma' = \sigma + \{a \mapsto \tau\}$, we want $\sigma' = \sigma_1 + \sigma_2$, where σ_1 and σ_2 are arbitrary environments (perhaps overlapping, in which case bindings in σ_2 take precedence).

We want an efficient algorithm and data structure for environment "adding." Balanced trees can implement $\sigma + \{a \mapsto \tau\}$ efficiently (in $\log(N)$ time, where N is the size of σ), but take $O(N)$ to compute $\sigma_1 + \sigma_2$, if σ_1 and σ_2 are both about size N.

To abstract the problem, solve the general nondisjoint integer-set union problem. The input is a set of commands of the form,

$$
\begin{array}{lll}
s_1 = \{4\} & & \textit{(define singleton set)} \\
s_2 = \{7\} & & \\
s_3 = s_1 \cup s_2 & & \textit{(nondestructive union)} \\
6 \overset{?}{\in} s_3 & & \textit{(membership test)} \\
s_4 = s_1 \cup s_3 & & \\
s_5 = \{9\} & & \\
s_6 = s_4 \cup s_5 & & \\
7 \overset{?}{\in} s_2 & &
\end{array}
$$

An efficient algorithm is one that can process an input of N commands, answering all membership queries, in less than $o(N^2)$ time.

*a. Implement an algorithm that is efficient when a typical set union $a \leftarrow b \cup c$ has b much smaller than c [Brown and Tarjan 1979].

***b. Design an algorithm that is efficient even in the worst case, or prove that this can't be done (see Lipton et al. [1997] for a lower bound in a restricted model).

***5.3** The Tiger language definition states that every cycle of type definitions must go through a record or array. But if the compiler forgets to check for this error, nothing terrible will happen. Explain why.