```
package Types;

public abstract class Type {
        public Type actual() {return this;}
}

subclasses of Type:
public INT();
public STRING();
public RECORD(Symbol.Symbol fieldName, Type fieldType, RECORD tail);
public ARRAY(Type element);
public NIL();
public VOID();
public NAME(Symbol.Symbol name);
```

PROGRAM 5.7. Package Types.

```
public class Table {
  public Table();
  public Table put(Symbol key, Object value);
  public Object get(Symbol key);
  public java.util.Enumeration keys();
}
```

The `put` function would return a new table without modifying the old one. We wouldn't need `beginScope` and `endScope`, because we could keep an old version of the table even as we use the new version.

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.Type`. 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 fields.

Arrays work just like records: the `ARRAY` constructor carries the type of the array elements.

For array and record types, there is another implicit piece of information carried by the `ARRAY` or `RECORD` object: the address of the object itself. That is, every Tiger-language "record type expression" creates a new (and different) record type, even if the fields are similar. We can encode this in our compiler by using `==` to compare record types to see if they are the same.

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 `VOID`.

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 `NAME` class has a `bind` method to fill in the place-holder when the definition is known. Then the `actual()` method (which for an ordinary type t simply returns t) for `NAME` types returns the filled-in binding:

```
package Types;

public class NAME extends Type {
   public Symbol.Symbol name;
   private Type binding;
   public NAME(Symbol.Symbol n) {name=n;}
   public Type actual() {return binding.actual();}
   public void bind(Type t) {binding = t;}
}
```

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.Type` – that is, a `Symbol.Table` whose `get` function always returns `Types.Type` objects. As shown in Figure 5.8, the `Env` class contains the table `tenv`, which is initialized to the "base" or "predefined" type environment. This maps the symbol `int` to `Types.INT` and `string` to `Types.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.

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.


```
package Semant;

class Env {
   Table venv;     // value environment
   Table tenv;     // type environment
   ErrorMsg.ErrorMsg errorMsg;
   Env(ErrorMsg.ErrorMsg err) {
        errorMsg=err;
        initialize venv and tenv with predefined identifiers
   }
}

abstract class Entry {}
class VarEntry extends Entry {
   Types.Type ty;
   VarEntry(Types.Type t) {ty=t;}
}
class FunEntry extends Entry {
   Types.RECORD formals;
   Types.Type result;
   public FunEntry(Types.RECORD f, Types.Type r) {formals=f; result=r;}
}
```

FIGURE 5.8. Environments for type-checking.

The `Env` class constructor initializes the `venv` environment by putting 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 class `Semant.Semant` performs semantic analysis – including type-checking – of abstract syntax. It contains a class variable `env` for accessing environments and printing error messages:

package Semant contains classes for type-checking.
- **class Semant** the only public class in this package; the main type-checking module.
- **abstract class Entry** for bindings in value environments.
  - **class VarEntry** for variable bindings.
  - **class FunEntry** for function bindings.
- **class OneFunc** helps in processing function declarations.
- **class OneType** helps in processing type declarations.
- **class ExpTy** holds the result of translating and type-checking an expression.
- **class Env** holds a value environment, type environment, and error-message printer; and is responsible for initializing the environments with predefined identifiers.

package Types describes Tiger-language types.

package Symbol handles symbols and environment tables.
- **class Symbol** makes strings into unique Symbol objects.
- **class Table** does environments with Scopes.

TABLE 5.9. Organization of packages for semantic analysis.

```
public class Semant {
 Env env;
 public Semant(ErrorMsg.ErrorMsg err) {this(new Env(err));}
 Semant(Env e) {env=e;}

 ExpTy transVar(Absyn.Var e) { ··· }
 ExpTy transExp(Absyn.Exp e) { ··· }
 Exp   transDec(Absyn.Dec e) { ··· }
 Ty    transTy (Absyn.Ty  e) { ··· }
}
```

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:

```
import Translate.Exp;
class ExpTy { Exp exp; Type ty;
   ExpTy(Exp e, Type t) {exp=e; ty=t;}
}
```

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:

```
package Translate;
abstract class Exp {}
```

and use `null` 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:

```
ExpTy transExp(Absyn.OpExp e) {
  ExpTy left = transExp(e.left);
  ExpTy right = transExp(e.right);
  if (e.oper == Absyn.OpExp.PLUS) {
       if (! (left.ty instanceof Types.INT))
            error(e.left.pos, "integer required");
       if (! (right.ty instanceof Types.INT))
            error(e.right.pos, "integer required");
       return new ExpTy(null, new Types.INT());
  }
}
```

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, it won't work. You can fill in the other cases yourself (see page 127).

It's also a bit clumsy. The case of checking for an integer type is common enough to warrant a function definition, `checkInt`. A cleaned-up version of `transExp` looks like:

```
Exp checkInt(ExpTy et, int pos) { ... ;  return et.exp; }

Types.Type INT = new Types.INT();

ExpTy transExp(Absyn.OpExp e) {
   switch (e.oper) {
        case Absyn.OpExp.PLUS:
            checkInt(transExp(e.left),e.left.pos);
            checkInt(transExp(e.right),e.right.pos);
            return new ExpTy(null, INT);
   }
}


ExpTy transExp(Absyn.Exp e) {
   if (e instanceof Absyn.VarExp)
             return transExp((Absyn.VarExp)e);
   if (e instanceof Absyn.IntExp)
             return transExp((Absyn.IntExp)e);
   if (e instanceof Absyn.CallExp)
             return transExp((Absyn.CallExp)e);
                    :
   if (e instanceof Absyn.ArrayExp)
             return transExp((Absyn.ArrayExp)e);
   throw new Error("transExp");
}
```

TYPE-CHECKING VARIABLES, SUBSCRIPTS, AND FIELDS

Each (overloaded) version of `transExp` operates on a different subclass of `Absyn.Exp`. Then a "dispatch" function `transExp(Absyn.Exp e)`, admittedly rather tedious to write, chooses from among the various overloaded versions of `transExp`. Similar dispatch functions will be necessary for the `transVar` and `transDec` methods as well.

```
ExpTy transVar(Absyn.SimpleVar v) {
   Entry x = (Entry)env.venv.get(v.name);
   if (x instanceof VarEntry) {
      VarEntry ent = (VarEntry)x;
      return new ExpTy(null, ent.ty);
   }
   else {
     error(v.pos, "undefined variable");
     return new ExpTy(null, INT);    // anything will do!
   }
}
```


The clause of `transVar` 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 new method in the `Types.Type` class, perhaps called `actual()`, 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:

```
ExpTy transExp(Absyn.LetExp e) {
     env.venv.beginScope();
     env.tenv.beginScope();
     for (Absyn.DecList p=e.decs; p!=null; p=p.tail)
          transDec(p.head);
     ExpTy et = transExp(e.body);
     env.venv.endScope();
     env.tenv.endScope();
     return new ExpTy(null, et.ty);
}
```

Here `transExp` marks the current "state" of the environments by calling `beginScope()`; calls `transDec` to augment the environments (`venv`, tenv) with new declarations; translates the body expression; then reverts to the original state of the environments using 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*.

```
Exp transDec(Absyn.VarDec d) {
    env.venv.put(d.name, new VarEntry(transExp(d.init).ty));
    return null;
}
```

What could be simpler? In practice, if d.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:

```
Exp transDec(Absyn.TypeDec d) {
  env.tenv.put(d.name, transTy(d.ty));
  return null;
}
```

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.Type). This translation is done by recurring over the structure of an Absyn.Type, turning Absyn.RecordTy into Types.RECORD, etc. While translating, transTy just looks up any symbols it finds in the type environment tenv.

The program fragment shown 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:

```
Exp transDec(Absyn.FunctionDec d) {
        Types.Type result = transTy(d.result);
        Types.RECORD formals=transTypeFields(d.params);
        env.venv.put(d.name, new FunEntry(formals,result));
        env.venv.beginScope();
        for(p=dec.params; p!=null; p=p.tail)
           env.venv.put(p.name, new VarEntry(
                        (Types.Type)env.tenv.get(p.typ)));
        transExp(d.body);
        env.venv.endScope();
        return null;
}
```

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 transTypeFields on each formal parameter; this yields a "record type," (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.

Next, the formal parameters are entered (as VarEntrys) into the value environment; this environment is used to process the *body* (with the transExp function). Finally, endScope() discards the formal-parameters (but not the FunEntry) from the environment; the resulting environment is 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 :

```
env.tenv.put(name, new Types.NAME(name));
```

Now, we can call `transTy` on the "body" of the type declaration, that is, on the record expression `{first: int, rest: list}`.

It's important that `transTy` stop as soon as it gets to any `NAME` type. If, for example, `transTy` behaved like `Types.Type.actual()` and tried to look "through" the `NAME` type bound to the identifier `list`, all it would find (in this case) would be `null` – which it is certainly not prepared for. This `null` 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 a private field within the `NAME` object, using the `bind()` method. Now we have a fully complete type environment, on which `actual()` will not have a problem.

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 $a \rightarrow b \rightarrow d \rightarrow a$. 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 `VarEntry`s in the value environment.

PROGRAM TYPE-CHECKING

Write a type-checking phase for your compiler, a class with the following interface:

```
package Semant;

public class Semant {
  public Semant(ErrorMsg.ErrorMsg err);
  public void transProg(Absyn.Exp exp);
}
```

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` class 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 4.11, 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/` Describes data types of the Tiger language.

and other files as before. Modify the makefile 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.