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 `VarEntry`s) 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, ..., 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 $a \rightarrow b \rightarrow d \rightarrow a$. 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 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 $\sigma' = \sigma + \{a \longmapsto \tau\}$, 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,

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

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.