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).