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.

```
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`.

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. 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`.

```
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.

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.

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).