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

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

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.

```
struct bucket {string key; void *binding; struct bucket *next;};

#define SIZE 109

struct bucket *table[SIZE];

unsigned int hash(char *s0)
{unsigned int h=0; char *s;
 for(s=s0; *s; s++)
       h = h*65599 + *s;
 return h;
}

struct bucket *Bucket(string key, void *binding, struct bucket *next) {
  struct bucket *b = checked_malloc(sizeof(*b));
  b->key=key; b->binding=binding; b->next=next;
  return b;
}

void insert(string key, void *binding) {
  int index = hash(key) % SIZE;
  table[index] = Bucket(key, binding, table[index]);
}

void *lookup(string key) {
  int index = hash(key) % SIZE;
  struct bucket *b;
  for(b=table[index]; b; b=b->next)
    if (0==strcmp(b->key,key)) return b->binding;
  return NULL;
}

void pop(string key) {
  int index = hash(key) % SIZE;
  table[index] = table[index]->next;
}
```

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`.

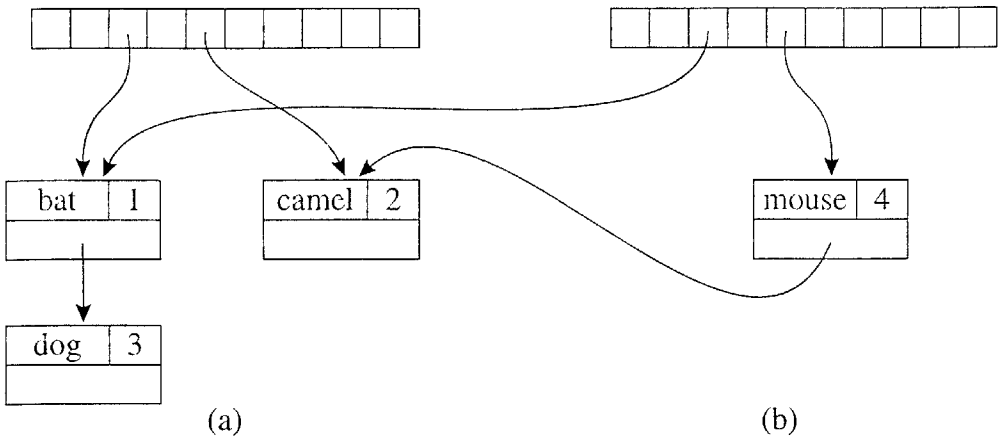

FIGURE 5.3. Hash tables.

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. 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,

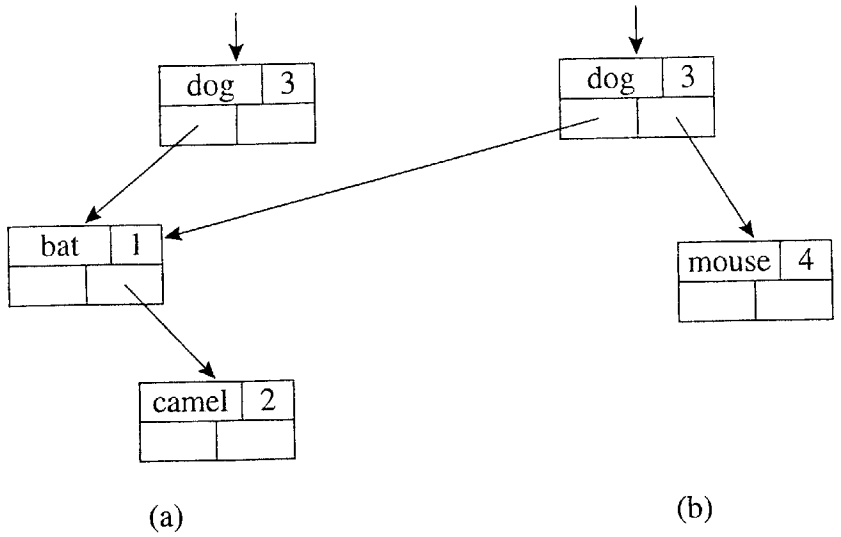

FIGURE 5.4. Binary search trees.

which represents the mapping

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

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 292).

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).

```
/* symbol.h */
typedef struct S_symbol_ *S_symbol;
S_symbol S_Symbol(string);
string S_name(S_symbol);

typedef struct TAB_table_ *S_table;
S_table S_empty(void);
void S_enter(S_table t, S_symbol sym, void *value);
void *S_look(S_table t, S_symbol sym);
void S_beginScope(S_table t);
void S_endScope(S_table t);
```

PROGRAM 5.5. `symbol.h`, the interface for symbols and symbol tables.

- Extracting an integer hash-key is very fast (in case we want to make hash table mapping symbols to something else). We will use the Symbol-pointer itself as the integer hash-key.
- Comparing two symbols for "greater-than" (in some arbitrary ordering) is very fast (in case we want to make binary search trees).

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 `symbol.c` as `S_Tables` mapping `S_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 the bindings be `void*`, though in any given table every binding should be a type binding, or every binding should be a value binding, and so on.

To implement `S_Symbol` (Program 5.6), we use hashing much as in Program 5.2.

For the Tiger compiler in C we choose to use destructive-update environments. The `S_empty()` function of the `symbol` module makes a new `S_Table`.

To handle the "undo" requirements of destructive update, the interface function `S_beginScope` remembers the current state of the table, and `S_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

```
#include <stdio.h>
#include <string.h>
#include "util.h"
#include "symbol.h"

struct S_symbol_ {string name; S_symbol next;};

static S_symbol mksymbol(string name, S_symbol next) {
  S_symbol s=checked_malloc(sizeof(*s));
  s->name=name; s->next=next;
  return s;
}

#define SIZE 109
static S_symbol hashtable[SIZE];

static unsigned int hash(char *s0) {... as in Program 5.2}

S_symbol S_Symbol(string name)  {
  int index= hash(name) % SIZE;
  S_symbol syms = hashtable[index], sym;
  for(sym=syms; sym; sym=sym->next)
    if (0==strcmp(sym->name,name)) return sym;
  sym = mksymbol(name,syms);
  hashtable[index]=sym;
  return sym;
}

string S_name(S_symbol sym) {
  return sym->name;
}

S_table S_empty(void) {  return TAB_empty(); }
void S_enter(S_table t, S_symbol sym, void *value){TAB_enter(t,sym,value);}
void *S_look(S_table t, S_symbol sym) {return TAB_look(t,sym);}

static struct S_symbol_ marksym = "<mark>",0;
void S_beginScope(S_table t) { S_enter(t,&marksym,NULL); }
void S_endScope(S_table t) {
  S_symbol s;
  do s=TAB_pop(t); while (s != &marksym);
}
    ⋮
```

PROGRAM 5.6. Symbol table (`symbol.c`) implementation.

```
/* table.h - generic hash table */

typedef struct TAB_table_ *TAB_table;

TAB_table TAB_empty(void);                          /* Make a new table */

void TAB_enter(TAB_table t, void *key, void *value);
    /* Enter key→ value into table t, shadowing any previous binding for key. */

void *TAB_look(TAB_table t, void *key);     /* Look up key in t */

void *TAB_pop(TAB_table t);
  /* Pop the most recent binding and return its key. This may expose another binding for the same key. */
```

PROGRAM 5.7. The `table.h` interface.

$x \mapsto b$ is entered (`S_enter(table,x,b)`), x is hashed into an index i, and a `Binder` object $x \mapsto b$ 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`).

The key x is not a character string, but is the `S_symbol` pointer itself. Module `table` implements generic pointer hash tables (`TAB_table`), mapping a key type (`void*`) to a binding type (also `void*`). Program 5.7 shows the `table.h` interface. Since using so many `void*`'s can easily lead to programming mistakes, the `symbol` module will encapsulate these operations with functions `S_empty`, `S_enter`, and so on, where the key type is `S_symbol` instead of `void*`.

There must also be an auxiliary stack, showing in what order the symbols were "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.

The auxiliary stack can be integrated into the `Binder` by having a global variable `top` showing the most recent `Symbol` bound in the table. Then "pushing" is accomplished by copying `top` into the `prevtop` field of the `Binder`. Thus, the "stack" is threaded through the binders.

FUNCTIONAL-STYLE SYMBOL TABLES

If we wanted to use functional-style symbol tables in the Tiger compiler, the `S_table` interface might look like this:

```
typedef struct TAB_table_ *S_table;
S_table S_empty(void);
S_table S_enter(S_table t, S_symbol sym, void *value);
void *S_look(S_table t, S_symbol sym);
```

The `S_enter` 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 `Ty_ty`. The `Types` module describes the structure of types, as shown in Program 5.8.

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 `Ty_array` constructor carries the type of the array elements.

For array and record types, there is another implicit piece of information carried by the `Ty_array` or `Ty_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
```

```
/* types.h */
typedef struct Ty_ty_ *Ty_ty;
typedef struct Ty_tyList_ *Ty_tyList;
typedef struct Ty_field_ *Ty_field;
typedef struct Ty_fieldList_ *Ty_fieldList;

struct Ty_ty_ {enum {Ty_record, Ty_nil, Ty_int, Ty_string,
                     Ty_array, Ty_name, Ty_void} kind;
               union {Ty_fieldList record;
                      Ty_ty array;
                      struct {S_symbol sym; Ty_ty ty;} name;
                     } u;
              };

Ty_ty Ty_Nil(void);
Ty_ty Ty_Int(void);
Ty_ty Ty_String(void);
Ty_ty Ty_Void(void);

Ty_ty Ty_Record(Ty_fieldList fields);
Ty_ty Ty_Array(Ty_ty ty);
Ty_ty Ty_Name(S_symbol sym, Ty_ty ty);

struct Ty_tyList_ {Ty_ty head; Ty_tyList tail;};
Ty_tyList Ty_TyList(Ty_ty head, Ty_tyList tail);

struct Ty_field_ {S_symbol name; Ty_ty ty;};
Ty_field Ty_Field(S_symbol name, Ty_ty ty);

struct Ty_fieldList_ {Ty_field head; Ty_fieldList tail;};
Ty_fieldList Ty_FieldList(Ty_field head, Ty_fieldList tail);
```

PROGRAM 5.8. Module `Types`.

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 `Ty_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. We can create a `Ty_Name(sym,NULL)` as a place-holder for the type-name `sym` and later on fill in the `ty` field of the `Ty_Name` object with the type that `sym` is supposed to stand for.

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 `Ty_ty` – that is, a `S_table` whose `S_lookup` function always returns `Ty_ty` pointers. As shown in Figure 5.9, 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

```
typedef struct E_enventry_ *E_enventry;

struct E_enventry_ {enum {E_varEntry, E_funEntry} kind;
                    union {struct {Ty_ty ty;} var;
                           struct {Ty_tyList formals; Ty_ty result;} fun;
                          } u;
                   };

E_enventry E_VarEntry(Ty_ty ty);
E_enventry E_FunEntry(Ty_tyList formals, Ty_ty result);

S_table E_base_tenv(void);   /* Ty_ ty environment */
S_table E_base_venv(void);   /* E_ enventry environment */
```

FIGURE 5.9. `env.h`: Environments for type-checking.

shown in Figure 5.9; 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).

5.3 TYPE-CHECKING EXPRESSIONS

The `Semant` module (`semant.h`, `semant.c`) performs semantic analysis – including type-checking – of abstract syntax. It contains four functions that recur over syntax trees:

```
struct expty transVar(S_table venv, S_table tenv, A_var v);
struct expty transExp(S_table venv, S_table tenv, A_exp a);
void         transDec(S_table venv, S_table tenv, A_dec d);
struct Ty_ty transTy (              S_table tenv, A_ty a);
```

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:

```
struct expty {Tr_exp exp; Ty_ty ty;};

struct expty expTy(Tr_exp exp, Ty_ty ty) {
  struct expty e; e.exp=exp; e.ty=ty; return e;
}
```

where `Tr_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:

```
typedef void *Tr_exp;
```

and use `NULL` for every value. We will flesh out the `Tr_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:

```
struct expty transExp(S_table venv, S_table tenv, A_exp a) {
  switch(a->kind) {
      :
    case A_opExp: {
      A_oper oper = a->u.op.oper;
      struct expty left =transExp(venv,tenv,a->u.op.left);
      struct expty right=transExp(venv,tenv,a->u.op.right);
      if (oper==A_plusOp) {
         if (left.ty->kind!=Ty_int)
            EM_error(a->u.op.left->pos, "integer required");
         if (right.ty->kind!=Ty_int)
            EM_error(a->u.op.right->pos,"integer required");
         return expTy(NULL,Ty_Int());
      }
      :
    }
  }
  assert(0); /* should have returned from some clause of the switch */
}
```

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, an assertion will fail. You can fill in the other cases yourself (see page 122).

TYPE-CHECKING VARIABLES, SUBSCRIPTS, AND FIELDS

The function `transVar` recurs over `A_var` expressions just as `transExp` recurs over `A_exp`.

```
struct expty transVar(S_table venv, S_table tenv, A_var v ) {
   switch(v->kind) {
     case A_simpleVar: {
        E_enventry x = S_look(venv,v->u.simple);
         if (x && x->kind==E_varEntry)
           return expTy(NULL, actual_ty(x->u.var.ty));
         else {EM_error(v->pos,"undefined variable %s",
                        S_name(v->u.simple));
               return expTy(NULL,Ty_Int());}
     }
 case A_fieldVar:
      :
}
```

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 5.9).

The type in the `VarEntry` will sometimes be a "`Name` type" (Program 5.8), 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 `Names`. The result will be a `Ty_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:

```
struct expty transExp(S_table venv, S_table tenv, A_exp a) {
   switch(a->kind) {
       ⋮
     case A_letExp: {
           struct expty exp;
           A_decList d;
           S_beginScope(venv);
           S_beginScope(tenv);
           for (d = a->u.let.decs; d; d=d->tail)
                transDec(venv,tenv,d->head);
           exp = transExp(venv,tenv,a->u.let.body);
           S_endScope(tenv);
           S_endScope(venv);
           return exp;
     }
       ⋮
```

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*.

```
void transDec(S_table venv, S_table tenv, A_dec d) {
   switch(d->kind) {
     case A_varDec: {
       struct expty e = transExp(venv,tenv,d->u.var.init);
       S_enter(venv, d->u.var.var, E_VarEntry(e.ty));
     }
      ⋮
}
```

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 `Ty_Nil` must be constrained by a `Ty_Record` type.

TYPE DECLARATIONS

Nonrecursive type declarations are not too hard:

```
void transDec(S_table venv, S_table tenv, A_dec d) {
       ⋮
     case A_typeDec: {
       struct expty e = transExp(venv,tenv,d->u.var.init);
       S_enter(tenv, d->u.type->head->name,
               transTy(d->u.type->head->ty));
     }
```

The `transTy` function translates type expressions as found in the abstract syntax (`A_ty`) to the digested type descriptions that we will put into environments (`Ty_ty`). This translation is done by recurring over the structure of an `A_ty`, turning `A_recordTy` into `Ty_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:

```
void transDec(S_table venv, S_table tenv, A_dec d) {
 switch(d->kind) {
   ⋮
  case A_functionDec: {
   A_fundec f = d->u.function->head;
   Ty_ty resultTy = tylook(tenv,f->result,f->pos);
   Ty_tyList formalTys = makeFormalTyList(tenv,f->params);
   S_enter(venv,f->name,E_FunEntry(formalTys,resultTy));
   S_beginScope(venv);
   {A_fieldList l; Ty_tyList t;
    for(l=f->params, t=formalTys; l; l=l->tail, t=t->tail)
       S_enter(venv,l->head->name,E_VarEntry(t->head));
   }
   transExp(venv, tenv, d->u.function->body);
   S_endScope(venv);
   break;
  }
   ⋮
```

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 `makeFormalTyList`, which traverses

the list of formal parameters and returns a list of their types (by looking each parameter type-id in the `tenv`). 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 `VarEntry`s) 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) $t_1, ..., t_n$ 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 `Ty_Name` type with an empty binding :

```
S_enter(tenv, name, Ty_Name(name,NULL));
```

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 `Ty_Name` type. If, for example, `transTy` behaved like `actual_ty` and tried to look "through" the `Ty_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 the `ty` field within the `Ty_Name` struct. Now we have a fully complete type environment, on which `actual_ty` 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 $a \to b \to d \to a$. 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 `VarEntry`s in the value environment.

PROGRAM TYPE-CHECKING

Write a type-checking phase for your compiler, a module `semant.c` matching the following header file:

```
/* semant.h */
void SEM_transProg(A_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` module described in this chapter. Make a module `Main` that calls the parser, yielding an `A_exp`, and then calls `SEM_transProg` on this expression.

You must use precisely the `Absyn` interface described in Figure 4.7, 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.h, types.c` 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.

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 a pointer to a dynamically allocated 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).

***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 prob-