"library" functions are declared (as described at the end of Section 5.2) at this outermost level, which does not contain a frame or formal parameter list.

The function `transDec` will make a new level for each Tiger function declaration. But `Level(···)` must be told the enclosing function's `level`. This means that `transDec` must know, while processing each declaration, the current static nesting level.

This is easy: `transDec` will now get an additional argument (in addition to the type and value environments) that is the current `level` as given by the appropriate call to `newLevel`. And `transExp` will also require this argument, so that `transDec` can pass a `level` to `transExp`, which passes it in turn to `transDec` to process declarations of nested functions. For similar reasons, `transVar` will also need a `level` argument.

Instead of passing an extra level argument and a new "break scope" to every function call, it's simpler to make a `level` field and a `breakScope` field of the `Semant` class. Then, at each function definition, `Semant` can create a new version of itself by calling

```
Semant newsem = new Semant(env,translate,new_lev,new_brk);
```

and then use `newsem` for processing the function body.

PROGRAM FRAMES

Augment `Semant/*.java` to allocate locations for local variables, and to keep track of the nesting level. To keep things simple, assume every variable escapes.

Implement the `Translate` module as `Translate/*`.

If you are compiling for the Sparc, implement the `Sparc` package containing `Sparc/SparcFrame.java`. If compiling for the MIPS, implement the `Mips` package, and so on.

Try to keep *all* the machine-specific details in your machine-dependent `Frame` module, not in `Semant` or `Translate`.

To keep things simple, handle *only* escaping parameters. That is, when implementing `newFrame`, handle only the case where all "escape" indicators are `true`.

If you are working on a RISC machine (such as MIPS or Sparc) that passes the first k parameters in registers and the rest in memory, keep things simple by handling *only* the case where there are k or fewer parameters.

Optional: Implement `FindEscape`, the module that sets the `escape` field of every variable in the abstract syntax. Modify your `transDec` function to allocate nonescaping variables and formal parameters in registers.

Optional: Handle functions with more than k formal parameters.

Supporting files available in `$TIGER/chap6` include:

`Temp/*`	The module supporting temporaries and labels.
`Util/BoolList.java`	The class for lists of booleans.

FURTHER READING

The use of a single contiguous stack to hold variables and return addresses dates from Lisp [McCarthy 1960] and Algol [Naur et al. 1963]. Block structure (the nesting of functions) and the use of static links are also from Algol.

Computers and compilers of the 1960s and '70s kept most program variables in memory, so that there was less need to worry about which variables escaped (needed addresses). The VAX, built in 1978, had a procedure-call instruction that assumed all arguments were pushed on the stack, and itself pushed program counter, frame pointer, argument pointer, argument count, and callee-save register mask on the stack [Leonard 1987].

With the RISC revolution [Patterson 1985] came the idea that procedure calling can be done with much less memory traffic. Local variables should be kept in registers by default; storing and fetching should be done *as needed*, driven by "spilling" in the register allocator [Chaitin 1982].

Most procedures don't have more than five arguments and five local variables [Tanenbaum 1978]. To take advantage of this, Chow et al. [1986] and Hopkins [1986] designed calling conventions optimized for the common case: the first four arguments are passed in registers, with the (rare) extra arguments passed in memory; compilers use both caller- and callee-save registers for local variables; leaf procedures don't even stack frames of their own if they can operate within the caller-save registers; and even the return address need not always be pushed on the stack.

so that when it comes across a function call it can pass the called function's `level` back to `Translate`. The `FunEntry` also needs the `label` of the function's machine-code entry point:

```
/* new versions of VarEntry and FunEntry */
class VarEntry extends Entry {
   Translate.Access access;
   Types.Type ty;
   VarEntry(Translate.Access a, Types.Type t) {···}
}

class FunEntry extends Entry {
  public Translate.Level level;
  public Temp.Label label;
  public Types.RECORD formals;
  public Types.Type result;
  public FunEntry(Translate.Level v, Temp.Label l,
                  Types.RECORD f, Types.Type r) {
    level=v; label=l; formals=f; result=r;
  }
}
```

When `Semant` processes a local variable declaration at level `lev`, it calls `lev.allocLocal(esc)` to create the variable in this level; the argument `esc` specifies whether the variable escapes. The result is a `Translate.Access`, which is an abstract data type (not the same as `Frame.Access`, since it must know about static links). Later, when the variable is used in an expression, `Semant` can hand this `access` back to `Translate` in order to generate the machine code to access the variable. Meanwhile, `Semant` records the access in each `VarEntry` in the value-environment.

The abstract data type `Translate.Access` can be implemented as a pair consisting of the variable's `level` and its `Frame.access`:

```
package Translate;
public class Access {
  Level home;
  Frame.Access acc;
  Access(Level h, Frame.Access a) {home=h; acc=a;}
}
```

so that `Level.allocLocal` calls `Frame.allocLocal`, and also remembers what level the variable lives in. The level information will be necessary later for calculating static links, when the variable is accessed from a (possibly) different level.

MANAGING STATIC LINKS

The `Frame` module should be independent of the specific source language being compiled. Many source languages do not have nested function declarations; thus, `Frame` should not know anything about static links. Instead, this is the responsibility of `Translate`.

`Translate` knows that each frame contains a static link. The static link is passed to a function in a register and stored into the frame. Since the static link behaves so much like a formal parameter, we will treat it as one (as much as possible). For a function with k "ordinary" parameters, let l be the list of booleans signifying whether the parameters escape. Then

l' `= new BoolList(true,` l`)`

is a new list; the extra `true` at the front signifies that the static link "extra parameter" does escape. Then `newFrame(`*label*, l'`)` makes the frame whose formal parameter list includes the "extra" parameter.

Suppose, for example, function $f(x, y)$ is nested inside function g, and the `level` (previously created) for g is called $\texttt{level}_g$. Then `Semant.transDec` can call

```
new Translate.Level(level_g, f, new BoolList(false,
                                new BoolList(false, null)))
```

assuming that neither x nor y escapes. Then `Level(parent,name,fmls)` adds an extra element to the formal-parameter list (for the static link), and calls

```
parent.frame.newFrame(name,new BoolList(true, fmls))
```

What comes back is a `Frame`. In this frame are three frame-offset values, accessible as `frame.formals`. The first of these is the static-link offset; the other two are the offsets for x and y. When `Semant` looks at `Level.formals`, it will get these two offsets, suitably converted into `access` values.

KEEPING TRACK OF LEVELS

With every call to `new Level(···)`, `Semant` must pass the enclosing `level` value. When creating the level for the "main" Tiger program (one not within any Tiger function), `Semant` should pass a special "outermost" level value, created as `new Translate.Level(frame)`. This is not the level of the Tiger main program, it is the level within which that program is nested. All

For a language where addresses of variables can be taken explicitly by the programmer, or where there are call-by-reference parameters, a similar FindEscape can find variables that escape in those ways.

TEMPORARIES AND LABELS

The compiler's semantic analysis phase will want to choose registers for parameters and local variables, and choose machine-code addresses for procedure bodies. But it is too early to determine exactly which registers are available, or exactly where a procedure body will be located. We use the word *temporary* to mean a value that is temporarily held in a register, and the word *label* to mean some machine-language location whose exact address is yet to be determined – just like a label in assembly language.

Temps are abstract names for local variables; labels are abstract names for static memory addresses. The Temp module manages these two distinct sets of names.

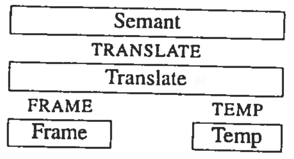
```
package Temp;
public class Temp {
  public String toString();
  public Temp();
}
public class Label {
  public String toString();
  public Label();
  public Label(String s);
  public Label(Symbol s);
}
public class TempList  {···}
public class LabelList {···}
```

new Temp.Temp() returns a new temporary from an infinite set of temps. new Temp.Label() returns a new label from an infinite set of labels. And new Temp.Label(*string*) returns a new label whose assembly-language name is *string*.

When processing the declaration function f(···), a label for the address of f's machine code can be produced by new Temp.Label(). It's tempting to call new Temp.Label("f") instead – the assembly-language program will be easier to debug if it uses the label f instead of L213 – but unfortunately there could be two different functions named f in different scopes.

TWO LAYERS OF ABSTRACTION

Our Tiger compiler will have two layers of abstraction between semantic analysis and frame-layout details:



The Frame and Temp packages provide machine-independent views of memory-resident and register-resident variables. The Translate module augments this by handling the notion of nested scopes (via static links), providing the interface TRANSLATE to the Semant module.

It is essential to have an abstraction layer at FRAME, to separate the source-language semantics from the machine-dependent frame layout. Separating Semant from Translate at the TRANSLATE interface is not absolutely necessary: we do it to avoid a huge, unwieldy module that does both type-checking and semantic translation.

In Chapter 7, we will see how Translate provides Java functions that are useful in producing intermediate representation from abstract syntax. Here, we need to know how Translate manages local variables and static function nesting for Semant.

```
package Translate;

public class Access { ... }
public class AccessList { ... }
public class Level {
  Frame.Frame frame;    // not public!
  public AccessList formals;
  public Level(Level parent, Symbol name, BoolList fmls);
  public Level(Frame.Frame f);
  public Access allocLocal(boolean escape);
}
```

In the semantic analysis phase of the Tiger compiler, transDec creates a new "nesting level" for each function by constructing

```
new Level(parent,name,formals);
```

the Level constructor in turn calls Frame.newFrame to make a new frame. Semant keeps this level in its FunEntry data structure for the function, The boolean argument to `allocLocal` specifies whether the new variable escapes and needs to go in the frame; if it is false, then the variable can be allocated in a register. Thus, `f.allocLocal(false)` might create `InReg`(t_{481}).

The calls to `allocLocal` need not come immediately after the frame is created. In a language such as Tiger or C, there may be variable-declaration blocks nested inside the body of a function. For example,

```
function f() =                    void f()
let var v := 6                    {int v=6;
 in print(v);                      print(v);
    let var v := 7                 {int v=7;
     in print (v)                   print(v);
    end;                           }
    print(v);                      print(v);
    let var v := 8                 {int v=8;
     in print (v)                   print(v);
    end;                           }
    print(v)                       print(v);
end                               }
```

In each of these cases, there are three different variables v. Either program will print the sequence 6 7 6 8 6. As each variable declaration is encountered in processing the Tiger program, `allocLocal` will be called to allocate a temporary or new space in the frame, associated with the name v. As each `end` (or closing brace) is encountered, the association with v will be forgotten – but the space is still reserved in the frame. Thus, there will be a distinct temporary or frame slot for every variable declared within the entire function.

The register allocator will use as few registers as possible to represent the temporaries. In this example, the second and third v variables (initialized to 7 and 8) could be held in the same temporary. A clever compiler might also optimize the size of the frame by noticing when two frame-resident variables could be allocated to the same slot.

CALCULATING ESCAPES

Local variables that do not escape can be allocated in a register; escaping variables must be allocated in the frame. A `FindEscape` function can look for escaping variables and record this information in the `escape` fields of the abstract syntax. The simplest way is to traverse the entire abstract syntax tree, looking for escaping uses of every variable. This phase must occur before semantic analysis begins, since `Semant` needs to know whether a variable

escapes *immediately* upon seeing that variable for the first time.

The traversal function for `FindEscape` will be a mutual recursion on abstract syntax `exp`'s and `var`'s, just like the type-checker. And, just like the type-checker, it will use environments that map variables to bindings. But in this case the binding will be very simple: it will be the boolean flag that is to be set if the particular variable escapes:

```
package FindEscape;

class Escape {
   int depth;
   abstract void setEscape();
}
class FormalEscape extends Escape {
  Absyn.FieldList fl;
  FormalEscape(int d, Abyn.FieldList f) {
          depth=d; fl=f; fl.escape=false;    }
  void setEscape() {fl.escape=true;}
}
class VarEscape extends Escape {
  Absyn.VarDec vd;
  VarEscape(int d; Abyn.VarDec v) {
          depth=d; vd=v; vd.escape=false;
  }
  void setEscape() {vd.escape=true;}
}

public class FindEscape {
  Symbol.Table escEnv = new Symbol.Table();
                    // escEnv maps Symbol to Escape
  void traverseVar(int depth, Absyn.Var v) { ··· }
  void traverseExp(int depth, Absyn.Exp e) { ··· }
  void traverseDec(int depth, Absyn.Dec d) { ··· }
  public FindEscape(Absyn.Exp e) traverseExp(0,e);
}
```

Whenever a variable or formal-parameter declaration is found at static function-nesting depth d, such as

`x = VarDec`{name=symbol("a"), escape=r,...}

then a new `VarEscape(d,x)` is entered into the environment, and `x.escape` is set to `false`.

This new environment is used in processing expressions within the scope of the variable; whenever a is used at depth $> d$, then `setEscape()` is called, which sets the `escape` field of `x` back to `true`.

`InFrame(`X`)` indicates a memory location at offset X from the frame pointer; `InReg(`t_{84}`)` indicates that it will be held in "register" t_{84}. `Frame.Access` is an abstract data type, so outside of the module the `InFrame` and `InReg` constructors are not visible. Other modules manipulate accesses using interface functions to be described in the next chapter.

The `formals` field is a list of k "accesses" denoting the locations where the formal parameters will be kept at run time, as seen from inside the callee. Parameters may be seen differently by the caller and the callee. For example, if parameters are passed on the stack, the caller may put a parameter at offset 4 from the stack pointer, but the callee sees it at offset 4 from the frame pointer. Or the caller may put a parameter into register 6, but the callee may want to move it out of the way and always access it from register 13. On the Sparc architecture, with register windows, the caller puts a parameter into register `o1`, but the `save` instruction shifts register windows so the callee sees this parameter in register `i1`.

Because this "shift of view" depends on the calling conventions of the target machine, it must be handled by the `Frame` module, starting with `newFrame`. For each formal parameter, `newFrame` must calculate two things:

- How the parameter will be seen from inside the function (in a register, or in a frame location);
- What instructions must be produced to implement the "view shift."

For example, a frame-resident parameter will be seen as "memory at offset X from the frame pointer," and the view shift will be implemented by copying the stack pointer to the frame pointer on entry to the procedure.

REPRESENTATION OF FRAME DESCRIPTIONS

The implementation module `Frame` is supposed to keep the representation of `Frame` objects secret from any clients of the `Frame` module. But really it's an object holding:

- the locations of all the formals,
- instructions required to implement the "view shift,"
- the number of locals allocated so far,
- and the `label` at which the function's machine code is to begin (see page 146).

Table 6.4 shows the formals of the three-argument function g (see page 141) as `newFrame` would allocate them on three different architectures: the Pentium, MIPS, and Sparc. The first parameter escapes, so it needs to be `InFrame`

		Pentium	MIPS	Sparc
Formals	1	`InFrame(8)`	`InFrame(0)`	`InFrame(68)`
	2	`InFrame(12)`	`InReg(`t_{157}`)`	`InReg(`t_{157}`)`
	3	`InFrame(16)`	`InReg(`t_{158}`)`	`InReg(`t_{158}`)`
View Shift		$M[\text{sp}+0] \leftarrow fp$ $\text{fp} \leftarrow \text{sp}$ $\text{sp} \leftarrow \text{sp} - K$	$\text{sp} \leftarrow \text{sp} - K$ $M[\text{sp}+K+0] \leftarrow$ `r2` $t_{157} \leftarrow$ `r4` $t_{158} \leftarrow$ `r5`	`save %sp,-K,%sp` $M[\text{fp}+68] \leftarrow$ `i0` $t_{157} \leftarrow$ `i1` $t_{158} \leftarrow$ `i2`

TABLE 6.4. Formal parameters for $g(x_1, x_2, x_3)$ where x_1 escapes.

on all three machines. The remaining parameters are `InFrame` on the Pentium, but `InReg` on the other machines.

The freshly created temporaries t_{157} and t_{158}, and the *move* instructions that copy `r4` and `r5` into them (or on the Sparc, `i1` and `i2`) may seem superfluous. Why shouldn't the body of g just access these formals directly from the registers in which they arrive? To see why not, consider

```
function m(x:int, y:int) =  (h(y,y); h(x,x))
```

If x stays in "parameter register 1" throughout m, and y is passed to h in parameter register 1, then there is a problem.

The register allocator will eventually choose which machine register should hold t_{157}. If there is no interference of the type shown in function `m`, then (on the MIPS) the allocator will take care to choose register `r4` to hold t_{157} and `r5` to hold t_{158}. Then the *move* instructions will be unnecessary and will be deleted at that time.

See also pages 176 and 271 for more discussion of the view shift.

LOCAL VARIABLES

Some local variables are kept in the frame; others are kept in registers. To allocate a new local variable in a frame f, the semantic analysis phase calls

```
f.allocLocal(true)
```

This returns an `InFrame` access with an offset from the frame pointer. For example, to allocate two local variables on the Sparc, `allocLocal` would be called twice, returning successively `InFrame(-4)` and `InFrame(-8)`, which are standard Sparc frame-pointer offsets for local variables.

10 show stores its static link (the address of prettyprint's frame) into its own frame.
15 show calls indent, passing its own frame pointer as indent's static link.
17 show calls show, passing its own static link (not its own frame pointer) as the static link.
12 indent uses the value *n* from show's frame. To do so, it fetches at an appropriate offset from indent's static link (which points at the frame of show).
13 indent calls write. It must pass the frame pointer of prettyprint as the static link. To obtain this, it first fetches at an offset from its own static link (from show's frame), the static link that had been passed to show.
14 indent uses the variable output from prettyprint's frame. To do so it starts with its own static link, then fetches show's, then fetches output.[4]

So on each procedure call or variable access, a chain of zero or more fetches is required; the length of the chain is just the *difference* in static nesting depth between the two functions involved.

6.2 FRAMES IN THE Tiger COMPILER

What sort of stack frames should the Tiger compiler use? Here we face the fact that every target machine architecture will have a different standard stack frame layout. If we want Tiger functions to be able to call C functions, we should use the standard layout. But we don't want the specifics of any particular machine intruding on the implementation of the semantic analysis module of the Tiger compiler.

Thus we must use *abstraction*. Just as the Symbol module provides a clean interface, and hides the internal representation of Symbol.Table from its clients, we must use an abstract representation for frames.

The frame interface will look something like this:

```
package Frame;
import Temp.Temp; import Temp.Label;

public abstract class Access { ··· }
public abstract class AccessList {···head;···tail;··· }
```

[4]This program would be cleaner if show called write here instead of manipulating output directly, but it would not be as instructive.

```
public abstract class Frame {
  abstract public Frame newFrame(Label name,
                                     Util.BoolList formals);
             public Label name;
             public AccessList formals;
  abstract public Access allocLocal(boolean escape);
   /* ... other stuff, eventually ... */
}
```

The abstract class Frame is implemented by a module specific to the target machine. For example, if compiling to the MIPS architecture, there would be

```
package Mips;
class Frame extends Frame.Frame { ··· }
```

In general, we may assume that the machine-independent parts of the compiler have access to this implementation of Frame; for example,

```
// in class Main.Main:
Frame.Frame frame = new Mips.Frame(···);
```

In this way the rest of the compiler may access frame without knowing the identity of the target machine (except an occurrence of the word Mips here and there).

The class Frame holds information about formal parameters and local variables allocated in this frame. To make a new frame for a function *f* with *k* formal parameters, call newFrame(*f*, *l*), where *l* is a list of *k* booleans: true for each parameter that escapes and false for each parameter that does not. The result will be a Frame object. For example, consider a three-argument function named *g* whose first argument escapes (needs to be kept in memory). Then

```
    frame.newFrame(g,new BoolList(true,
                          new BoolList(false,
                           new BoolList(false, null))))
```

returns a new frame object.

The Access class describes formals and locals that may be in the frame or in registers. This is an *abstract data type*, so its implementation as a pair of subclasses is visible only inside the Frame module:

```
package Mips;
class InFrame extends Frame.Access {int offset; ··· }
class InReg   extends Frame.Access {Temp temp; ··· }
```