6 Activation Records

stack: an orderly pile or heap

Webster's Dictionary

In almost any modern programming language, a function may have *local* variables that are created upon entry to the function. Several invocations of the function may exist at the same time, and each invocation has its own *instantiations* of local variables.

In the Tiger function,

```
function f(x: int) : int =
  let var y := x+x
   in if y < 10
          then f(y)
          else y-1
  end
```

a new instantiation of `x` is created (and initialized by `f`'s caller) each time that `f` is called. Because there are recursive calls, many of these `x`'s exist simultaneously. Similarly, a new instantiation of `y` is created each time the body of `f` is entered.

In many languages (including C, Pascal, and Tiger), local variables are destroyed when a function returns. Since a function returns only after all the functions it has called have returned, we say that function calls behave in last-in-first-out (LIFO) fashion. If local variables are created on function entry and destroyed on function exit, then we can use a LIFO data structure – a stack – to hold them.

```
fun f(x) =
  let fun g(y) = x+y
   in g
  end

val h = f(3)
val j = f(4)

val z = h(5)
val w = j(7)
```

(a) Written in ML

```
int (*)() f(int x) {
  int g(int y) {return x+y;}
  return g;
}

int (*h)() = f(3);
int (*j)() = f(4);

int z = h(5);
int w = j(7);
```

(b) Written in pseudo-C

PROGRAM 6.1. An example of higher-order functions.

HIGHER-ORDER FUNCTIONS

But in languages supporting both nested functions *and* function-valued variables, it may be necessary to keep local variables after a function has returned! Consider Program 6.1: This is legal in ML, but of course in C one cannot really nest the function g inside the function f.

When $f(3)$ is executed, a new local variable x is created for the activation of function f. Then g is returned as the result of $f(x)$; but g has not yet been called, so y is not yet created.

At this point f has returned, but it is too early to destroy x, because when $h(5)$ is eventually executed it will need the value $x = 3$. Meanwhile, $f(4)$ is entered, creating a *different* instance of x, and it returns a *different* instance of g in which $x = 4$.

It is the combination of *nested functions* (where inner functions may use variables defined in the outer functions) and *functions returned as results* (or stored into variables) that causes local variables to need lifetimes longer than their enclosing function invocations.

Pascal (and Tiger) have nested functions, but they do not have functions as returnable values. C has functions as returnable values, but not nested functions. So these languages can use stacks to hold local variables.

ML, Scheme, and several other languages have both nested functions and functions as returnable values (this combination is called *higher-order functions*). So they cannot use stacks to hold all local variables. This complicates the implementation of ML and Scheme – but the added expressive power of higher-order functions justifies the extra implementation effort.

For the remainder of this chapter we will consider languages with stackable local variables and postpone discussion of higher-order functions to Chapter 15.

6.1 STACK FRAMES

The simplest notion of a *stack* is a data structure that supports two operations, *push* and *pop*. However, it turns out that local variables are pushed in large batches (on entry to functions) and popped in large batches (on exit). Furthermore, when local variables are created they are not always initialized right away. Finally, after many variables have been pushed, we want to continue accessing variables deep within the stack. So the abstract *push* and *pop* model is just not suitable.

Instead, we treat the stack as a big array, with a special register – the *stack pointer* – that points at some location. All locations beyond the stack pointer are considered to be garbage, and all locations before the stack pointer are considered to be allocated. The stack usually grows only at the entry to a function, by an increment large enough to hold all the local variables for that function, and, just before the exit from the function, shrinks by the same amount. The area on the stack devoted to the local variables, parameters, return address, and other temporaries for a function is called the function's *activation record* or *stack frame*. For historical reasons, run-time stacks usually start at a high memory address and grow toward smaller addresses. This can be rather confusing: stacks grow downward and shrink upward, like icicles.

The design of a frame layout takes into account the particular features of an instruction set architecture and the programming language being compiled. However, the manufacturer of a computer often prescribes a "standard" frame layout to be used on that architecture, where possible, by all compilers for all programming languages. Sometimes this layout is not the most convenient one for a particular programming language or compiler. But by using the "standard" layout, we gain the considerable benefit that functions written in one language can call functions written in another language.

Figure 6.2 shows a typical stack frame layout. The frame has a set of *incoming arguments* (technically these are part of the previous frame but they are at a known offset from the frame pointer) passed by the caller. The *return address* is created by the CALL instruction and tells where (within the calling function) control should return upon completion of the current function. Some *local variables* are in this frame; other local variables are kept in

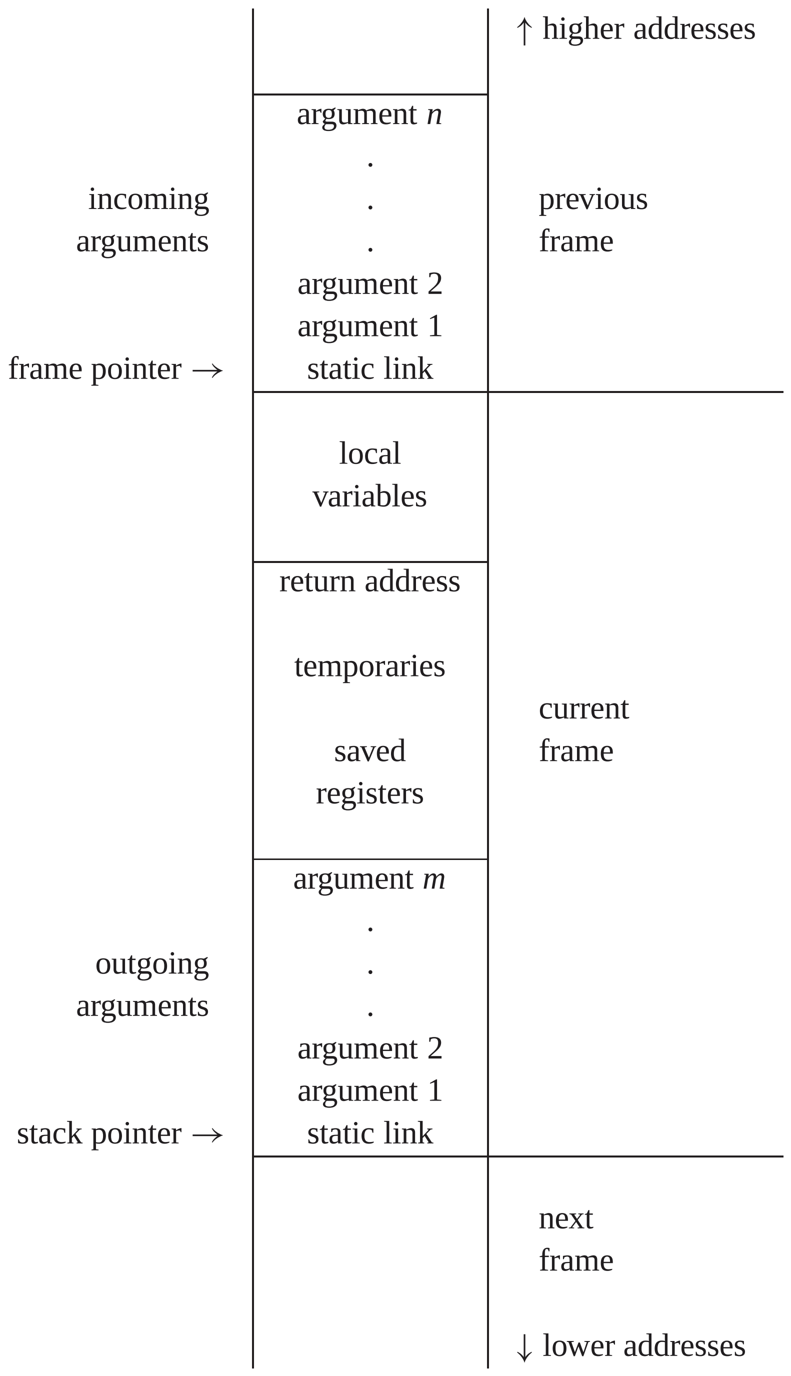


FIGURE 6.2. A stack frame.

machine registers. Sometimes a local variable kept in a register needs to be *saved* into the frame to make room for other uses of the register; there is an area in the frame for this purpose. Finally, when the current function calls other functions, it can use the *outgoing argument* space to pass parameters.

THE FRAME POINTER

Suppose a function $g(\ldots)$ calls the function $f(a_1, \ldots, a_n)$. We say g is the *caller* and f is the *callee*. On entry to f, the stack pointer points to the first argument that g passes to f. On entry, f allocates a frame by simply subtracting the frame size from the stack pointer SP.

The old SP becomes the current *frame pointer* FP. In some frame layouts, FP is a separate register; the old value of FP is saved in memory (in the frame) and the new FP becomes the old SP. When f exits, it just copies FP back to SP and fetches back the saved FP. This arrangement is useful if f's frame size can vary, or if frames are not always contiguous on the stack. But if the frame size is fixed, then for each function f the FP will always differ from SP by a known constant, and it is not necessary to use a register for FP at all – FP is a "fictional" register whose value is always SP+*framesize*.

Why talk about a frame pointer at all? Why not just refer to all variables, parameters, etc. by their offset from SP, if the frame size is constant? The frame size is not known until quite late in the compilation process, when the number of memory-resident temporaries and saved registers is determined. But it is useful to know the offsets of formal parameters and local variables much earlier. So, for convenience, we still talk about a frame pointer. And we put the formals and locals right near the frame pointer at offsets that are known *early*; the temporaries and saved registers go farther away, at offsets that are known *later*.

REGISTERS

A modern machine has a large set of *registers* (typically 32 of them). To make compiled programs run fast, it's useful to keep local variables, intermediate results of expressions, and other values in registers instead of in the stack frame. Registers can be directly accessed by arithmetic instructions; on most machines, accessing memory requires separate *load* and *store* instructions. Even on machines whose arithmetic instructions can access memory, it is faster to access registers.

A machine (usually) has only one set of registers, but many different procedures and functions need to use registers. Suppose a function f is using register r to hold a local variable and calls procedure g, which also uses r for its own calculations. Then r must be saved (stored into a stack frame) before g uses it and restored (fetched back from the frame) after g is finished using it. But is it f's responsibility to save and restore the register, or g's? We say that r is a *caller-save* register if the caller (in this case, f) must save and restore the register, and r is *callee-save* if it is the responsibility of the callee (in this case, g).

On most machine architectures, the notion of caller-save or callee-save register is not something built into the hardware, but is a convention described in the machine's reference manual. On the MIPS computer, for example, registers 16–23 are preserved across procedure calls (callee-save), and all other registers are not preserved across procedure calls (caller-save).

Sometimes the saves and restores are unnecessary. If f knows that the value of some variable x will not be needed after the call, it may put x in a caller-save register *and not save it* when calling g. Conversely, if f has a local variable i that is needed before and after several function calls, it may put i in some callee-save register r_i and, save r_i just once (upon entry to f) and fetch it back just once (before returning from f). Thus, the wise selection of a caller-save or callee-save register for each local variable and temporary can reduce the number of stores and fetches a program executes. We will rely on our register allocator to choose the appropriate kind of register for each local variable and temporary value.

PARAMETER PASSING

On most machines whose calling conventions were designed in the 1970s, function arguments were passed on the stack.[1] But this causes needless memory traffic. Studies of actual programs have shown that very few functions have more than four arguments, and almost none have more than six. Therefore, parameter-passing conventions for modern machines specify that the first k arguments (for $k = 4$ or $k = 6$, typically) of a function are passed in registers r_p, ..., r_{p+k-1}, and the rest of the arguments are passed in memory.

Now, suppose $f(a_1, \ldots, a_n)$ (which received its parameters in $r_1, \ldots, r_n$, for example) calls $h(z)$. It must pass the argument z in r_1; so f saves the old contents of r_1 (the value a_1) in its stack frame before calling h. But there is the memory traffic that was supposedly avoided by passing arguments in registers! How has the use of registers saved any time?

[1]Before about 1960, they were passed not on the stack but in statically allocated blocks of memory, which precluded the use of recursive functions.

There are four answers, any or all of which can be used at the same time:

1. Some procedures don't call other procedures – these are called *leaf* procedures. What proportion of procedures are leaves? Well, if we make the (optimistic) assumption that the average procedure calls either no other procedures or calls at least two others, then we can describe a "tree" of procedure calls in which there are more leaves than internal nodes. This means that *most* procedures called are leaf procedures.

   Leaf procedures need not write their incoming arguments to memory. In fact, often they don't need to allocate a stack frame at all. This is an important savings.

2. Some optimizing compilers use *interprocedural register allocation*, analyzing all the functions in an entire program at once. Then they assign different procedures different registers in which to receive parameters and hold local variables. Thus $f(x)$ might receive x in r_1, but call $h(z)$ with z in r_7.

3. Even if f is not a leaf procedure, it might be finished with all its use of the argument x by the time it calls h (technically, x is a dead variable at the point where h is called). Then f can overwrite r_1 without saving it.

4. Some architectures have *register windows*, so that each function invocation can allocate a fresh set of registers without memory traffic.

If f needs to write an incoming parameter into the frame, where in the frame should it go? Ideally, f's frame layout should matter only in the implementation of f. A straightforward approach would be for the caller to pass arguments $a_1, \ldots, a_k$ in registers and $a_{k+1}, \ldots, a_n$ at the end of its own frame – the place marked *outgoing arguments* in Figure 6.2 – which become the *incoming arguments* of the callee. If the callee needed to write any of these arguments to memory, it would write them to the area marked *local variables*.

The C programming language actually allows you to take the address of a formal parameter and guarantees that all the formal parameters of a function are at consecutive addresses! This is the `varargs` feature that `printf` uses. Allowing programmers to take the address of a parameter can lead to a *dangling reference* if the address outlives the frame – as the address of `x` will in `int *f(int x){return &x;}` – and even when it does not lead to bugs, the consecutive-address rule for parameters constrains the compiler and makes stack-frame layout more complicated. To resolve the contradiction that parameters are passed in registers, but have addresses too, the first k parameters are passed in registers; but any parameter whose address is taken must be written to a memory location on entry to the function. To satisfy `printf`, the memory locations into which register arguments are written must all be consecutive with the memory locations in which arguments $k + 1$, $k + 2$, etc. are written. Therefore, C programs can't have some of the arguments saved in one place and some saved in another – they must all be saved contiguously.

So in the standard calling convention of many modern machines the *calling* function reserves space for the register arguments in its own frame, next to the place where it writes argument $k + 1$. But the calling function does not actually write anything there; that space is written into *by the called function*, and only if the called function needs to write arguments into memory for any reason.

A more dignified way to take the address of a local variable is to use *call-by-reference*. With call-by-reference, the programmer does not explicitly manipulate the address of a variable x. Instead, if x is passed as the argument to $f(y)$ where y is a "by-reference" parameter, the compiler generates code to pass the address of x instead of the contents of x. At any use of y within the function, the compiler generates an extra pointer dereference. With call-by-reference, there can be no "dangling reference," since y must disappear when f returns, and f returns before x's scope ends.

RETURN ADDRESSES

When function g calls function f, eventually f must return. It needs to know where to go back to. If the *call* instruction within g is at address a, then (usually) the right place to return to is $a + 1$, the next instruction in g. This is called the *return address*.

On 1970s machines, the return address was pushed on the stack by the *call* instruction. Modern science has shown that it is faster and more flexible to pass the return address in a register, avoiding memory traffic and also avoiding the need to build any particular stack discipline into the hardware.

On modern machines, the *call* instruction merely puts the return address (the address of the instruction after the call) in a designated register. A non-leaf procedure will then have to write it to the stack (unless interprocedural register allocation is used), but a leaf procedure will not.

FRAME-RESIDENT VARIABLES

So a modern procedure-call convention will pass function parameters in registers, pass the return address in a register, and return the function result in a register. Many of the local variables will be allocated to registers, as will the intermediate results of expression evaluation. Values are written to memory (in the stack frame) only when necessary for one of these reasons:

- the variable will be passed by reference, so it must have a memory address (or, in the C language the & operator is anywhere applied to the variable);
- the variable is accessed by a procedure nested inside the current one;[2]
- the value is too big to fit into a single register;[3]
- the variable is an array, for which address arithmetic is necessary to extract components;
- the register holding the variable is needed for a specific purpose, such as parameter passing (as described above), though a compiler may move such values to other registers instead of storing them in memory;
- or there are so many local variables and temporary values that they won't all fit in registers, in which case some of them are "spilled" into the frame.

We will say that a variable *escapes* if it is passed by reference, its address is taken (using C's & operator), or it is accessed from a nested function.

When a formal parameter or local variable is declared, it's convenient to assign it a location – either in registers or in the stack frame – right at that point in processing the program. Then, when occurrences of that variable are found in expressions, they can be translated into machine code that refers to the right location. Unfortunately, the conditions in our list don't manifest themselves early enough. When the compiler first encounters the declaration of a variable, it doesn't yet know whether the variable will ever be passed by reference, accessed in a nested procedure, or have its address taken; and doesn't know how many registers the calculation of expressions will require (it might be desirable to put some local variables in the frame instead of in registers). An industrial-strength compiler must assign provisional locations to all formals and locals, and decide later which of them should really go in registers.

STATIC LINKS

In languages that allow nested function declarations (such as Pascal, ML, and Tiger), the inner functions may use variables declared in outer functions. This language feature is called *block structure*.

For example, in Program 6.3, `write` refers to the outer variable `output`, and `indent` refers to outer variables `n` and `output`. To make this work, the function `indent` must have access not only to its own frame (for variables `i` and `s`) but also to the frames of `show` (for variable `n`) and `prettyprint` (for variable `output`).

[2]However, with register allocation across function boundaries, local variables accessed by inner functions can sometimes go in registers, as long as the inner function knows where to look.

[3]However, some compilers spread out a large value into several registers for efficiency.

```
 1   type tree = {key: string, left: tree, right: tree}
 2
 3   function prettyprint(tree: tree) : string =
 4    let
 5        var output := ""
 6
 7        function write(s: string) =
 8            output := concat(output,s)
 9
10        function show(n:int, t: tree) =
11           let function indent(s: string) =
12                   (for i := 1 to n
13                     do write(" ");
14                    output := concat(output,s); write("\n"))
15            in if t=nil then indent(".")
16               else (indent(t.key);
17                     show(n+1,t.left);
18                     show(n+1,t.right))
19           end
20
21     in show(0,tree); output
22    end
```

PROGRAM 6.3. Nested functions.

There are several methods to accomplish this:

- Whenever a function f is called, it can be passed a pointer to the frame of the function statically enclosing f; this pointer is the *static link*.
- A global array can be maintained, containing – in position i – a pointer to the frame of the most recently entered procedure whose *static nesting depth* is i. This array is called a *display*.
- When g calls f, each variable of g that is actually accessed by f (or by any function nested inside f) is passed to f as an extra argument. This is called *lambda lifting*.

I will describe in detail only the method of static links. Which method should be used in practice? See Exercise 6.7.

Whenever a function f is called, it is passed a pointer to the stack frame of the "current" (most recently entered) activation of the function g that *immediately encloses* f in the text of the program.

For example, in Program 6.3:

Line #

21 `prettyprint` calls `show`, passing `prettyprint`'s own frame pointer as `show`'s static link.

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 `S_table` from its clients, we must use an abstract representation for frames.

The frame interface will look something like this:

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

```
/* frame.h */

typedef struct F_frame_ *F_frame;
typedef struct F_access_ *F_access;

typedef struct F_accessList_ *F_accessList;
struct F_accessList_ {F_access head; F_accessList tail;};

F_frame F_newFrame(Temp_label name, U_boolList formals);
Temp_label F_name(F_frame f);
F_accessList F_formals(F_frame f);
F_access F_allocLocal(F_frame f, bool escape);
  ⋮
```

The abstract interface `frame.h` is implemented by a module specific to the target machine. For example, if compiling to the MIPS architecture, there would be a file `mipsframe.c` containing

```
#include "frame.h"
  ⋮
```

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

```
 /* in translate.c */
#include "frame.h"
  ⋮
F_frame frame = F_newFrame(···);
```

In this way the rest of the compiler may access the `Frame` module without knowing the identity of the target machine.

The type `F_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 `F_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 `F_frame` object. For example, consider a three-argument function named g whose first argument escapes (needs to be kept in memory). Then

```
F_newFrame(g,U_BoolList(TRUE,
                  U_BoolList(FALSE,
                   U_BoolList(FALSE, NULL))))
```

returns a new frame object.

The `F_access` type describes formals and locals that may be in the frame or in registers. This is an *abstract data type*, so the contents of `struct F_access_` are visible only inside the `Frame` module:

```
/* mipsframe.c */
#include "frame.h"

struct F_access_
       {enum {inFrame, inReg} kind;
        union {
          int offset;       /* InFrame */
          Temp_temp reg;    /* InReg */
        } u;
      };
static F_access InFrame(int offset);
static F_access InReg(Temp_temp reg);
```

`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}. `F_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 `F_formals` interface function extracts 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

		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		$M[\text{sp}+0] \leftarrow fp$	$\text{sp} \leftarrow \text{sp} - K$	`save %sp,-K,%sp`
		$\text{fp} \leftarrow \text{sp}$	$M[\text{sp}+K+0] \leftarrow$ `r2`	$M[\text{fp}+68] \leftarrow$ `i0`
		$\text{sp} \leftarrow \text{sp} - K$	$t_{157} \leftarrow$ `r4`	$t_{157} \leftarrow$ `i1`
			$t_{158} \leftarrow$ `r5`	$t_{158} \leftarrow$ `i2`

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

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 the `F_frame` type secret from any clients of the `Frame` module. But really it's a data structure 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 141).

Table 6.4 shows the formals of the three-argument function g (see page 136) as `newFrame` would allocate them on three different architectures: the Pentium, MIPS, and Sparc. The first parameter escapes, so it needs to be `InFrame` 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 172 and 267 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(f,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.

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(`f`,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() =
let var v := 6
 in print(v);
    let var v := 7
     in print (v)
    end;
    print(v);
    let var v := 8
     in print (v)
    end;
    print(v)
end
```

```
void f()
{int v=6;
 print(v);
 {int v=7;
  print(v);
 }
 print(v);
 {int v=8;
  print(v);
 }
 print(v);
}
```

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:

```
/* escape.h */
void Esc_findEscape(A_exp exp);
```

```
/* escape.c */
static void traverseExp(S_table env, int depth, A_exp e);
static void traverseDec(S_table env, int depth, A_dec d);
static void traverseVar(S_table env, int depth, A_var v);
```

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

`A_VarDec`{name=symbol(“`a`”), escape=r,...}

then `EscapeEntry(d,&(x->escape))` 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 the `escape` field of `x` is set to `TRUE`.

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.

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

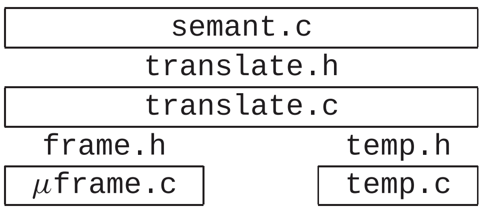
```
 /* temp.h */

typedef struct Temp_temp_ *Temp_temp;
Temp_temp Temp_newtemp(void);

typedef S_symbol Temp_label;
Temp_label Temp_newlabel(void);
Temp_label Temp_namedlabel(string name);
string Temp_labelstring(Temp_label s);

typedef struct Temp_tempList_ *Temp_tempList;
struct Temp_tempList_ {Temp_temp head; Temp_tempList tail;}
Temp_tempList Temp_TempList(Temp_temp head,
                            Temp_tempList tail);
typedef struct Temp_labelList_ *Temp_labelList;
struct Temp_labelList_{Temp_label head; Temp_labelList tail;}
Temp_labelList Temp_LabelList(Temp_label head,
                              Temp_labelList tail);
   ...
  /* Temp˙map type, and operations on it, described on page 207 */
```

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

When processing the declaration `function f(`$\cdots$`)`, a label for the address of `f`'s machine code can be produced by `Temp_newlabel()`. It's tempting to call `Temp_namedlabel("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.h` and `temp.h` interfaces 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.h` to the `Semant` module.

It is essential to have an abstraction layer at `frame.h`, to separate the source-language semantics from the machine-dependent frame layout (μ stands for a target machine such as `mips`, `sparc`, `pentium`). Separating `Semant` from `Translate` at the `translate.h` 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 C 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`.

```
/* translate.h */

typedef struct Tr_access_ *Tr_access;

typedef ··· Tr_accessList ···
Tr_accessList Tr_AccessList(Tr_access head,
                            Tr_accessList tail);

Tr_level Tr_outermost(void);
Tr_level Tr_newLevel(Tr_level parent, Temp_label name,
                     U_boolList formals);
Tr_accessList Tr_formals(Tr_level level);
Tr_access Tr_allocLocal(Tr_level level, bool escape);
```

In the semantic analysis phase of the Tiger compiler, `transDec` creates a new "nesting level" for each function by calling `Tr_newLevel`. That function in turn calls `F_newFrame` to make a new frame. `Semant` keeps this `level` in its `FunEntry` data structure for the function, 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 */
struct E_enventry_ {
      enum {E_varEntry, E_funEntry} kind;
      union {struct {Tr_access access; Ty_ty ty;} var;
             struct {Tr_level level; Temp_label label;
                  Ty_tyList formals; Ty_ty result;} fun;
             } u;
      };

E_enventry E_VarEntry(Tr_access access, Ty_ty ty);
E_enventry E_FunEntry(Tr_level level, Temp_label label,
                      Ty_tyList formals, Ty_ty result);
```

When `Semant` processes a local variable declaration at level `lev`, it calls `Tr_allocLocal(lev,esc)` to create the variable in this level; the argument `esc` specifies whether the variable escapes. The result is a `Tr_access`, which is an abstract data type (not the same as `F_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 `Tr_access` can be implemented as a pair consisting of the variable's `level` and its `F_access`:

```
 /* inside translate.c */
struct Tr_access_ {Tr_level level; F_access access;};
```

so that `Tr_allocLocal` calls `F_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' `= U_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 `level`$_g$. Then `transDec` (inside `semant.c`) can call

```
Tr_newLevel(level_g, f,
            U_BoolList(FALSE, U_BoolList(FALSE, NULL)))
```

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

```
F_newFrame(label,U_BoolList(TRUE, fmls))
```

What comes back is a `F_frame`. In this frame are three frame-offset values, accessible by calling `F_formals(frame)`. The first of these is the static-link offset; the other two are the offsets for x and y. When `Semant` calls `Tr_formals(level)`, it will get these two offsets, suitably converted into `access` values.

KEEPING TRACK OF LEVELS

With every call to `Tr_newLevel`, `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 level value, obtained by calling `Tr_outermost()`. This is not the level of the Tiger main program, it is the level within which that program is nested. All "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. `Tr_outermost()` returns the same level every time it is called; it is a function just because initializing global variables to heap-allocated values is difficult in C.

The function `transDec` will make a new level for each Tiger function declaration. But `Tr_newLevel` 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.

PROGRAM FRAMES

Augment `semant.c` 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.c`.

If you are compiling for the Sparc, implement the `SparcFrame` module (matching `frame.h`) as `sparcframe.c`. If compiling for the MIPS, implement `MipsFrame`, 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.h, temp.c` The module supporting temporaries and labels.

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.

EXERCISES

6.1 Using the C compiler of your choice (or a compiler for another language), compile some small test functions into assembly language. On Unix this is usually done by `cc -S`. Turn on all possible compiler optimizations. Then evaluate the compiled programs by these criteria:

a. Are local variables kept in registers?

b. If local variable *b* is live across more than one procedure call, is it kept in a callee-save register? Explain how doing this would speed up the following program:

```
int f(int a) {int b; b=a+1; g(); h(b); return b+2;}
```

c. If local variable x is never live across a procedure call, is it properly kept in a caller-save register? Explain how doing this would speed up the following program:

```
void h(int y) {int x; x=y+1; f(y); f(2);}
```

6.2 If you have a C compiler that passes parameters in registers, make it generate assembly language for this function:

```
extern void h(int, int);
void m(int x, int y) {h(y,y); h(x,x);}
```

Clearly, if arguments to $m(x, y)$ arrive in registers r_{arg1} and r_{arg2}, and arguments to h must be passed in r_{arg1} and r_{arg2}, then x cannot stay in r_{arg1} during the marshalling of arguments to $h(y, y)$. Explain when and how your C compiler moves x out of the r_{arg1} register so as to call $h(y, y)$.

6.3 For each of the variables a, b, c, d, e in this C program, say whether the variable should be kept in memory or a register, and why.

```
int f(int a, int b)
{ int c[3], d, e;
  d=a+1;
  e=g(c, &b);
  return e+c[1]+b;
}
```

***6.4** How much memory should this program use?

```
int f(int i) {int j,k; j=i*i; k=i?f(i-1):0; return k+j;}
void main() {f(100000);}
```

a. Imagine a compiler that passes parameters in registers, wastes no space providing "backup storage" for parameters passed in registers, does not use static links, and in general makes stack frames as small as possible. How big should each stack frame for f be, in words?

b. What is the maximum memory use of this program, with such a compiler?

c. Using your favorite C compiler, compile this program to assembly language and report the size of f's stack frame.

d. Calculate the total memory use of this program with the real C compiler.

e. Quantitatively and comprehensively explain the discrepancy between (a) and (c).

f. Comment on the likelihood that the designers of this C compiler considered deeply recursive functions important in real programs.

***6.5** Some Tiger functions do not need static links, because they do not make use of a particular feature of the Tiger language.

a. Characterize precisely those functions that do not need a static link passed to them.

b. Give an algorithm (perhaps similar to `FindEscape`) that marks all such functions.

***6.6** Instead of (or in addition to) using static links, there are other ways of getting access to nonlocal variables. One way is just to leave the variable in a register!

```
function f() : int =
  let var a := 5
      function g() : int =
         (a+1)
   in g()+g()
  end
```

If a is left in register r_7 (for example) while g is called, then g can just access it from there.

What properties must a local variable, the function in which it is defined, and the functions in which it is used, have for this trick to work?

***6.7** A *display* is a data structure that may be used as an alternative to static links for maintaining access to nonlocal variables. It is an array of frame pointers, indexed by static nesting depth. Element D_i of the display always points to the most recently called function whose static nesting depth is i.

The bookkeeping performed by a function f, whose static nesting depth is i, looks like:

```
Copy $D_i$ to *save location* in stack frame
Copy frame pointer to $D_i$
   ··· body of $f$ ···
Copy *save location* back to $D_i$
```

In Program 6.3, function `prettyprint` is at depth 1, `write` and `show` are at depth 2, and so on.

a. Show the sequence of machine instructions required to fetch the variable `output` into a register at line 14 of Program 6.3, using static links.

b. Show the machine instructions required if a display were used instead.

c. When variable x is declared at depth d_1 and accessed at depth d_2, how many instructions does the static-link method require to fetch x?

d. How many does the display method require?

e. How many instructions does static-link maintenance require for a procedure entry and exit (combined)?

f. How many instructions does display maintenance require for procedure entry and exit?

Should we use displays instead of static links? Perhaps; but the issue is more complicated. For languages such as Pascal and Tiger with block structure but no function variables, displays work well.

But the full expressive power of block structure is obtained when functions can be returned as results of other functions, as in Scheme and ML. For such languages, there are more issues to consider than just variable-access time and procedure entry-exit cost: there is closure-building cost, and the problem of avoiding useless data kept live in closures. Chapter 15 explains some of the issues.

7 Translation to Intermediate Code

trans-late: to turn into one's own or another language

Webster's Dictionary

The semantic analysis phase of a compiler must translate abstract syntax into abstract machine code. It can do this after type-checking, or at the same time.

Though it is possible to translate directly to real machine code, this hinders portability and modularity. Suppose we want compilers for N different source languages, targeted to M different machines. In principle this is $N \cdot M$ compilers (Figure 7.1a), a large implementation task.

An *intermediate representation* (IR) is a kind of abstract machine language that can express the target-machine operations without committing to too much machine-specific detail. But it is also independent of the details of the source language. The *front end* of the compiler does lexical analysis, parsing, semantic analysis, and translation to intermediate representation. The *back end* does optimization of the intermediate representation and translation to machine language.

A portable compiler translates the source language into IR and then translates the IR into machine language, as illustrated in Figure 7.1b. Now only N front ends and M back ends are required. Such an implementation task is more reasonable.

Even when only one front end and one back end are being built, a good IR can modularize the task, so that the front end is not complicated with machine-specific details, and the back end is not bothered with information specific to one source language. Many different kinds of IR are used in compilers; for this compiler I have chosen simple expression trees.

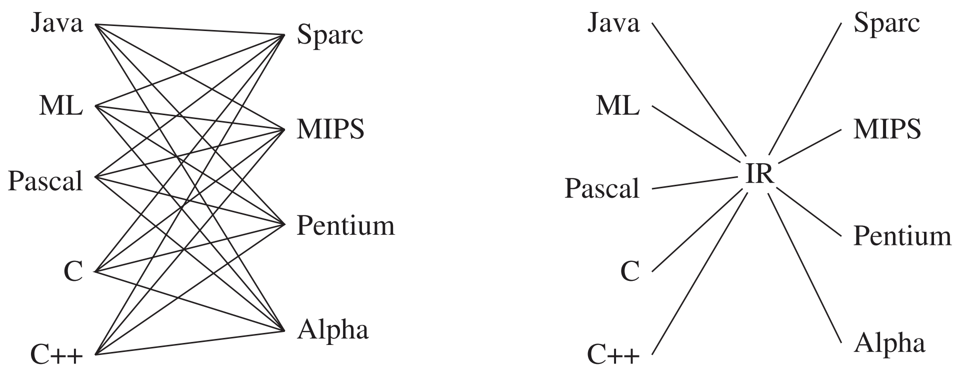


FIGURE 7.1. Compilers for five languages and four target machines: (left) without an IR, (right) with an IR.

7.1 INTERMEDIATE REPRESENTATION TREES

The intermediate representation tree language is defined by the interface `tree.h`, as shown in Figure 7.2.

A good intermediate representation has several qualities:

- It must be convenient for the semantic analysis phase to produce.
- It must be convenient to translate into real machine language, for all the desired target machines.
- Each construct must have a clear and simple meaning, so that optimizing transformations that rewrite the intermediate representation can easily be specified and implemented.

Individual pieces of abstract syntax can be complicated things, such as array subscripts, procedure calls, and so on. And individual "real machine" instructions can also have a complicated effect (though this is less true of modern RISC machines than of earlier architectures). Unfortunately, it is not always the case that complex pieces of the abstract syntax correspond exactly to the complex instructions that a machine can execute.

Therefore, the intermediate representation should have individual components that describe only extremely simple things: a single fetch, store, add, move, or jump. Then any "chunky" piece of abstract syntax can be translated into just the right set of abstract machine instructions; and groups of abstract machine instructions can be clumped together (perhaps in quite different clumps) to form "real" machine instructions.

Here is a description of the meaning of each tree operator. First, the ex-

```
/* tree.h */

typedef struct T_stm_ *T_stm;
struct T_stm_ {enum {T_SEQ, T_LABEL, T_JUMP, ..., T_EXP} kind;
               union {struct {T_stm left, right;} SEQ;
                        ⋮
                    } u; };
T_stm T_Seq(T_stm left, T_stm right);
T_stm T_Label(Temp_label);
T_stm T_Jump(T_exp exp, Temp_labelList labels);
T_stm T_Cjump(T_relOp op, T_exp left, T_exp right,
                Temp_label true, Temp_label false);
T_stm T_Move(T_exp, T_exp);
T_stm T_Exp(T_exp);

typedef struct T_exp_ *T_exp;
struct T_exp_ {enum {T_BINOP, T_MEM, T_TEMP, ..., T_CALL} kind;
               union {struct {T_binOp op; T_exp left, right;} BINOP;
                        ⋮
                    } u; };
T_exp T_Binop(T_binOp, T_exp, T_exp);
T_exp T_Mem(T_exp);
T_exp T_Temp(Temp_temp);
T_exp T_Eseq(T_stm, T_exp);
T_exp T_Name(Temp_label);
T_exp T_Const(int);
T_exp T_Call(T_exp, T_expList);

typedef struct T_expList_ *T_expList;
struct T_expList_ {T_exp head; T_expList tail;};
T_expList T_ExpList (T_exp head, T_expList tail);

typedef struct T_stmList_ *T_stmList;
struct T_stmList_ {T_stm head; T_stmList tail;};
T_stmList T_StmList (T_stm head, T_stmList tail);

typedef enum {T_plus, T_minus, T_mul, T_div, T_and, T_or,
              T_lshift, T_rshift, T_arshift, T_xor} T_binOp ;
typedef enum  {T_eq, T_ne, T_lt, T_gt, T_le, T_ge,
                T_ult, T_ule, T_ugt, T_uge} T_relOp;
```

FIGURE 7.2. Intermediate representation trees.

pressions (`T_exp`), which stand for the computation of some value (possibly with side effects):

CONST(i) The integer constant i. Written in C as `T_Const(i)`.

NAME(n) The symbolic constant n (corresponding to an assembly language label). Written in C as `T_Name(n)`.

TEMP(t) Temporary t. A temporary in the abstract machine is similar to a register in a real machine. However, the abstract machine has an infinite number of temporaries.

BINOP(o, e_1, e_2) The application of binary operator o to operands e_1, e_2. Subexpression e_1 is evaluated before e_2. The integer arithmetic operators are PLUS, MINUS, MUL, DIV; the integer bitwise logical operators are AND, OR, XOR; the integer logical shift operators are LSHIFT, RSHIFT; the integer arithmetic right-shift is ARSHIFT. The Tiger language has no logical operators, but the intermediate language is meant to be independent of any source language; also, the logical operators might be used in implementing other features of Tiger.

MEM(e) The contents of *wordSize* bytes of memory starting at address e (where *wordSize* is defined in the `Frame` module). Note that when MEM is used as the left child of a MOVE, it means “store,” but anywhere else it means “fetch.”

CALL(f, l) A procedure call: the application of function f to argument list l. The subexpression f is evaluated before the arguments which are evaluated left to right.

ESEQ(s, e) The statement s is evaluated for side effects, then e is evaluated for a result.

The statements (`T_stm`) of the tree language perform side effects and control flow:

MOVE(TEMP t, e) Evaluate e and move it into temporary t.

MOVE(MEM(e_1), e_2) Evaluate e_1, yielding address a. Then evaluate e_2, and store the result into *wordSize* bytes of memory starting at a.

EXP(e) Evaluate e and discard the result.

JUMP($e, labs$) Transfer control (jump) to address e. The destination e may be a literal label, as in NAME(*lab*), or it may be an address calculated by any other kind of expression. For example, a C-language `switch(i)` statement may be implemented by doing arithmetic on i. The list of labels `labs` specifies all the possible locations that the expression e can evaluate to; this is necessary for dataflow analysis later. The common case of jumping to a known label l is written as

```
T_Jump(l,Temp_LabelList(l,NULL));
```

CJUMP(o, e_1, e_2, t, f) Evaluate e_1, e_2 in that order, yielding values a, b. Then compare a, b using the relational operator o. If the result is `true`, jump to

t; otherwise jump to f. The relational operators are EQ and NE for integer equality and nonequality (signed or unsigned); signed integer inequalities LT, GT, LE, GE; and unsigned integer inequalities ULT, ULE, UGT, UGE.

SEQ(s_1, s_2) The statement s_1 followed by s_2.

LABEL(n) Define the constant value of name n to be the current machine code address. This is like a label definition in assembly language. The value NAME(n) may be the target of jumps, calls, etc.

It is almost possible to give a formal semantics to the `Tree` language. However, there is no provision in this language for procedure and function definitions – we can specify only the body of each function. The procedure entry and exit sequences will be added later as special "glue" that is different for each target machine.

7.2 TRANSLATION INTO TREES

Translation of abstract syntax expressions into intermediate trees is reasonably straightforward; but there are many cases to handle.

KINDS OF EXPRESSIONS

What should the representation of an abstract syntax expression `A_exp` be in the `Tree` language? At first it seems obvious that it should be `T_exp`. However, this is true only for certain kinds of expressions, the ones that compute a value. Expressions that return no value (such as some procedure calls, or while expressions in the Tiger language) are more naturally represented by `T_stm`. And expressions with Boolean values, such as $a > b$, might best be represented as a conditional jump – a combination of `T_stm` and a pair of destinations represented by `Temp_label`s.

Therefore, we will make a union type (with `kind` tag, as usual) in the `Translate` module, to model these three kinds of expressions:

```
/* in translate.h */
typedef struct Tr_exp_ *Tr_exp;
```

```
 /* in translate.c */
struct Cx {patchList trues; patchList falses; T_stm stm;};

struct Tr_exp_
      {enum {Tr_ex, Tr_nx, Tr_cx} kind;
        union {T_exp ex; T_stm nx; struct Cx cx; } u;
      };


static Tr_exp Tr_Ex(T_exp ex);
static Tr_exp Tr_Nx(T_stm nx);
static Tr_exp Tr_Cx(patchList trues, patchList falses,
                    T_stm stm);
```

`Ex` stands for an “expression,” represented as a `Tr_exp`.
`Nx` stands for “no result,” represented as a `Tree` statement.
`Cx` stands for “conditional,” represented as a statement that may jump to a true-label or false-label, but these labels have yet to be filled in. If you fill in a true-destination and a false-destination, the resulting statement evaluates some conditionals and then jumps to one of the destinations (the statement will never “fall through”).

For example, the Tiger expression `a>b|c<d` might translate to the conditional:

```
  Temp_label z = Temp_newlabel();
  T_stm s1 = T_Seq(T_Cjump(T_gt,a,b,[NULL]_t,z),
                T_Seq(T_Label(z),
                   T_Cjump(T_lt,c,d,[NULL]_t,[NULL]_f)));
```

The problem here is that *t* and *f* are not known yet, so the statement is full of `NULL`s. We won’t know the true-destination and false-destination until much later. We need to make a list of where all the `NULL`s are that need to be filled in with `t` when it is known, and another list of all the places that need to be filled in with `f`.

To represent “a list of places where a label must be filled in” we use a `patchList`:

```
typedef struct patchList_ *patchList;
struct patchList_ {Temp_label *head; patchList tail;};
static patchList PatchList(Temp_label *head, patchList tail);
```

Therefore, we can complete the translation of `a>b|c<d` into a `Tr_exp` as follows:

```
patchList trues = PatchList(&s1->u.SEQ.left->u.CJUMP.true,
                   PatchList(&s1->u.SEQ.right->u.SEQ.right->
                                                u.CJUMP.true,
                    NULL));
patchList falses= PatchList(&s1->u.SEQ.right->u.SEQ.right->
                                                u.CJUMP.false,
                   NULL);
Tr_exp e1 = Tr_Cx(trues, falses, s1);
```

Sometimes we will have an expression of one kind and we will need to convert it to an equivalent expression of another kind. For example, the Tiger statement

```
  flag := (a>b | c<d)
```

requires the conversion of a `Cx` into an `Ex` so that a 1 (for true) or 0 (for false) can be stored into `flag`.

It is helpful to have three conversion functions:

```
static   T_exp   unEx(Tr_exp e);
static   T_stm   unNx(Tr_exp e);
static struct Cx unCx(Tr_exp e);
```

Each of these behaves as if it were simply stripping off the corresponding constructor (`Ex`, `Nx`, or `Cx`), but the catch is that each conversion function must work no matter what constructor has been used!

Suppose e is the representation of `a>b|c<d`, so

```
e = Tr_Cx(trues,falses,stm)
```

Then the assignment statement can be implemented as

MOVE(TEMP$_{flag}$, `unEx`(e)).

We have "stripped off the `Ex` constructor" even though `Cx` was really there instead.

Program 7.3 is the implementation of `unEx`. To convert a "conditional" into a "value expression," we invent a new temporary r and new labels t and f. Then we make a `T_stm` that moves the value 1 into r, and follow it with the statement `e->u.cs.stm`; it will jump to t if true, and f if false. If the condition is false, then 0 is moved into r; if true, then execution proceeds at t and the second move is skipped. The result of the whole thing is just the temporary r containing zero or one.

```
static T_exp unEx(Tr_exp e) {
switch (e->kind) {
 case Tr_ex:
   return e->u.ex;
 case Tr_cx: {
   Temp_temp r = Temp_newtemp();
   Temp_label t = Temp_newlabel(), f = Temp_newlabel();
   doPatch(e->u.cx.trues, t);
   doPatch(e->u.cx.falses, f);
   return T_Eseq(T_Move(T_Temp(r), T_Const(1)),
           T_Eseq(e->u.cx.stm,
            T_Eseq(T_Label(f),
             T_Eseq(T_Move(T_Temp(r), T_Const(0)),
              T_Eseq(T_Label(t),
                      T_Temp(r))))));
 }
 case Tr_nx:
   return T_Eseq(e->u.nx, T_Const(0));
 }
 assert(0);   /* can't get here */
}
```

PROGRAM 7.3. The conversion function `unEx`.

By calling `doPatch(e->u.cx.trues,t)` we fill in all the label-holes in the `trues` patch list with the label `t`, and similarly for the `falses` list with `doPatch(e->u.cx.trues,f)`. The `doPatch` function is one of two useful utility functions on patch lists:

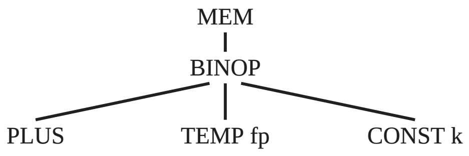
```
void doPatch(patchList tList, Temp_label label) {
   for (; tList; tList=tList->tail)
     *(tList->head) = label;
}

patchList joinPatch(patchList first,patchList second) {
  if (!first) return second;
  for (; first->tail; first=first->tail);  /* go to end of list */
  first->tail = second;
  return first;
}
```

The functions `unCx` and `unNx` are left as an exercise. It's helpful to have `unCx` treat the cases of CONST 0 and CONST 1 specially, since they have particularly simple and efficient translations. Also, `unCx` should never expect to see a `Tr_exp` with a `kind` of `Tr_nx` – such a case should never occur in compiling a well typed Tiger program.

SIMPLE VARIABLES

The semantic analysis phase has a function that type-checks a variable in the context of a type environment `tenv` and a value environment `venv`. This function `transVar` returns a `struct expty` containing a `Tr_exp` and a `Ty_ty`. In Chapter 5 the `exp` was merely a place-holder, but now `Semant` must be modified so that each `exp` holds the intermediate-representation translation of each Tiger expression.

For a simple variable v declared in the current procedure's stack frame, we translate it as



MEM(BINOP(PLUS, TEMP `fp`, CONST k))

where k is the offset of v within the frame and TEMP `fp` is the frame pointer register. For the Tiger compiler we make the simplifying assumption that all variables are the same size – the natural word size of the machine.

Interface between Translate and Semant. The type `Tr_exp` is an abstract data type, whose `Ex` and `Nx` constructors are visible only within `Translate`.

The manipulation of MEM nodes should all be done in the `Translate` module, not in `Semant`. Doing it in `Semant` would clutter up the readability of that module and would make `Semant` dependent on the `Tree` representation.

We add a function

```
Tr_Exp Tr_simpleVar(Tr_Access, Tr_Level);
```

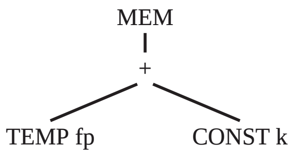
to the `Translate` interface. Now `Semant` can pass the `access` of x (obtained from `Tr_allocLocal`) and the `level` of the function in which x is used and get back a `Tr_exp`.

With this interface, `Semant` never gets its hands dirty with a `T_exp` at all. In fact, this is a good rule of thumb in determining the interface between `Semant` and `Translate`: the `Semant` module should not contain any direct reference to the `Tree` or `Frame` module. Any manipulation of IR trees should be done by `Translate`.

The `Frame` module holds all machine-dependent definitions; here we add to it a frame-pointer register `FP` and a constant whose value is the machine's natural word size:

```
/* frame.h */
  ⋮
Temp_temp F_FP(void);
extern const int F_wordSize;
T_exp F_Exp(F_access acc, T_exp framePtr);
```

In this and later chapters, I will abbreviate BINOP(PLUS, e_1, e_2) as $+(e_1, e_2)$, so the tree above would be shown as



+(TEMP `fp`, CONST k)

The function `F_Exp` is used by `Translate` to turn an `F_access` into the `Tree` expression. The `T_exp` argument is the address of the stack frame that the `F_access` lives in. Thus, for an access a such as `InFrame`(k), we have

`F_Exp(`a`, T_Temp(F_FP()))` *returns* MEM(BINOP(PLUS, TEMP(FP), CONST(k)))

Why bother to pass the tree expression `T_Temp(F_FP())` as an argument? The answer is that the address of the frame is the same as the current frame pointer *only* when accessing the variable from its own level. When accessing a from an inner-nested function, the frame address must be calculated using static links, and the result of this calculation will be the `T_exp` argument to `F_Exp`.

If a is a register access such as `InReg`(t_{832}), then the frame-address argument to `F_Exp` will be discarded, and the result will be simply TEMP t_{832}.

An l-value such as v or $a[i]$ or $p.next$ can appear either on the left side or the right side of an assignment statement – l stands for *left*, to distinguish from r-values that can appear only on the right side of an assignment. Fortunately, only MEM and TEMP nodes can appear on the left of a MOVE node.

FOLLOWING STATIC LINKS

When a variable x is declared at an outer level of static scope, static links must be used. The general form is

$$\text{MEM}(+(\text{CONST } k_n,\ \text{MEM}(+(\text{CONST } k_{n-1}, \ldots \text{MEM}(+(\text{CONST } k_1,\ \text{TEMP FP}))\ldots))))$$

where the $k_1, \ldots, k_{n-1}$ are the various static link offsets in nested functions, and k_n is the offset of x in its own frame.

To construct this expression, we need the `level` l_f of the function f in which x is used, and the `level` l_g of the function g in which x is declared. As we strip levels from l_f, we use the static link offsets $k_1, k_2, \ldots$ from these levels to construct the tree. Eventually we reach l_g, and we can stop.

`Tr_simpleVar` must produce a chain of MEM and + nodes to fetch static links for all frames between the level of use (the `level` passed to `simpleVar`) and the level of definition (the `level` within the variable's `access`).

ARRAY VARIABLES

For the rest of this chapter I will not specify all the interface functions of `Translate`, as I have done for `simpleVar`. But the rule of thumb just given applies to all of them; there should be a `Translate` function to handle array subscripts, one for record fields, one for each kind of expression, and so on.

Different programming languages treat array-valued variables differently.

In Pascal, an array variable stands for the contents of the array – in this case all 12 integers. The Pascal program

```
var a,b : array[1..12] of integer
begin
      a := b
end;
```

copies the contents of array *a* into array *b*.

In C, there is no such thing as an array variable. There are pointer variables; arrays are like "pointer constants." Thus, this is illegal:

```
{int a[12], b[12];
 a = b;
}
```

but this is quite legal:

```
{int a[12], *b;
 b = a;
}
```

The statement `b = a` does not copy the elements of *a*; instead, it means that *b* now points to the beginning of the array *a*.

In Tiger (as in Java and ML), array variables behave like pointers. Tiger has no named array constants as in C, however. Instead, new array values are created (and initialized) by the construct $t_a[n]$ `of` i, where t_a is the name of an array type, *n* is the number of elements, and *i* is the initial value of each element. In the program

```
let
 type intArray = array of int
 var a := intArray[12] of 0
 var b := intArray[12] of 7
in a := b
end
```

the array variable *a* ends up pointing to the same 12 sevens as the variable *b*; the original 12 zeros allocated for *a* are discarded.

Tiger record values are also pointers. Record assigment, like array assignment, is pointer assigment and does not copy all the fields. This is also true of modern object-oriented and functional programming languages, which try to blur the syntactic distinction between pointers and objects. In C or Pascal, however, a record value is "big," and record assigment means copying all the fields.

STRUCTURED *L*-VALUES

An *l*-value is the result of an expression that can occur on the *left* of an assignment statement, such as `x` or `p.y` or `a[i+2]`. An *r*-value is one that can only appear on the *right* of an assignment, such as `a+3` or `f(x)`. That is, an *l*-value denotes a *location* that can be assigned to, and an *r*-value does not.

Of course, an *l*-value can occur on the right of an assignment statement; in this case the *contents* of the location are implicitly taken.

We say that an integer or pointer value is a "scalar," since it has only one component. Such a value occupies just one word of memory and can fit in a register. All the variables and *l*-values in Tiger are scalar. Even a Tiger array or record variable is really a pointer (a kind of scalar); the *Tiger Language Reference Manual* does not say so explicitly, because it is talking about Tiger semantics instead of Tiger implementation.

In C or Pascal there are structured *l*-values – structs in C, arrays and records in Pascal – that are not scalar. To implement a language with "large" variables such as the arrays and records in C or Pascal requires a bit of extra work. In a C compiler, the `access` type would require information about the size of the variable. Then, the MEM operator of the `TREE` intermediate language would need to be extended with a notion of size:

```
T_exp T_Mem(T_exp, int size);
```

The translation of a local variable into an IR tree would look like

$\text{MEM}(+(\text{TEMP}\ \mathtt{fp},\ \text{CONST}\ k_n),\ S)$

where the S indicates the size of the object to be fetched or stored (depending on whether this tree appears on the left or right of a MOVE).

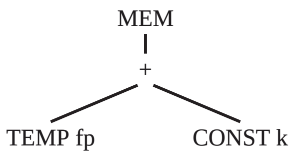
Leaving out the size on MEM nodes makes the Tiger compiler easier to implement, but limits the generality of its intermediate representation.

SUBSCRIPTING AND FIELD SELECTION

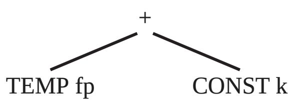
To subscript an array in Pascal or C (to compute $a[i]$), just calculate the address of the ith element of a: $(i - l) \times s + a$, where l is the lower bound of the index range, s is the size (in bytes) of each array element, and a is the base address of the array elements. If a is global, with a compile-time constant address, then the subtraction $a - s \times l$ can be done at compile time.

Similarly, to select field f of a record *l*-value a (to calculate $a.f$), simply add the constant field offset of f to the address a.

An array variable a is an *l*-value; so is an array subscript expression $a[i]$, even if i is not an *l*-value. To calculate the *l*-value $a[i]$ from a, we do arithmetic on the address of a. Thus, in a Pascal compiler, the translation of an *l*-value (particularly a structured *l*-value) should *not* be something like



but should instead be the `Tree` expression representing the base address of the array:

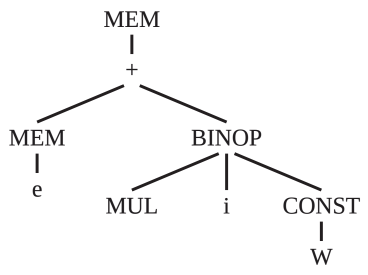


What could happen to this l-value?

- A particular element might be subscripted, yielding a (smaller) l-value. A "+" node would add the index times the element size to the l-value for the base of the array.
- The l-value (representing the entire array) might be used in a context where an r-value is required (e.g., passed as a by-value parameter, or assigned to another array variable). Then the l-value is *coerced* into an r-value by applying the MEM operator to it.

In the Tiger language, there are no structured, or "large," l-values. This is because all record and array values are really pointers to record and array structures. The "base address" of the array is really the contents of a pointer variable, so MEM is required to fetch this base address.

Thus, if *a* is a memory-resident array variable represented as MEM(e), then the contents of address e will be a one-word pointer value p. The contents of addresses p, $p + W$, $p + 2W$, ... (where W is the word size) will be the elements of the array (all elements are one word long). Thus, $a[i]$ is just



MEM(+(MEM(e), BINOP(MUL, i, CONST W)))

***L*-values and MEM nodes.** Technically, an l-value (or *assignable variable*) should be represented as an *address* (without the top MEM node in the diagram above). Converting an l-value to an r-value (when it is used in an expression) means *fetching* from that address; assigning to an l-value means *storing* to that address. We are attaching the MEM node to the l-value before knowing whether it is to be fetched or stored; this works only because in the `Tree` intermediate representation, MEM means both *store* (when used as the left child of a MOVE) and *fetch* (when used elsewhere).

A SERMON ON SAFETY

Life is too short to spend time chasing down irreproducible bugs, and money is too valuable to waste on the purchase of flaky software. When a program has a bug, it should detect that fact as soon as possible and announce that fact (or take corrective action) before the bug causes any harm.

Some bugs are very subtle. But it should not take a genius to detect an out-of-bounds array subscript: if the array bounds are $L..H$, and the subscript is i, then $i < L$ or $i > H$ is an array bounds error. Furthermore, computers are well-equipped with hardware able to compute the condition $i > H$. For several decades now, we have known that compilers can automatically emit the code to test this condition. There is no excuse for a compiler that is unable to emit code for checking array bounds. Optimizing compilers can often *safely* remove the checks by compile-time analysis; see Section 18.4.

One might say, by way of excuse, "but the language in which I program has the kind of address arithmetic that makes it impossible to know the bounds of an array." Yes, and the man who shot his mother and father threw himself upon the mercy of the court because he was an orphan.

In some rare circumstances, a portion of a program demands blinding speed, and the timing budget does not allow for bounds checking. In such a case, it would be best if the optimizing compiler could analyze the subscript expressions and prove that the index will always be within bounds, so that an explicit bounds check is not necessary. If that is not possible, perhaps it is reasonable in these rare cases to allow the programmer to explicitly specify an unchecked subscript operation. But this does not excuse the compiler from checking all the other subscript expressions in the program.

Needless to say, the compiler should check pointers for `nil` before dereferencing them, too.[1]

ARITHMETIC

Integer arithmetic is easy to translate: each `Absyn` arithmetic operator corresponds to a `Tree` operator.

The `Tree` language has no unary arithmetic operators. Unary negation of integers can be implemented as subtraction from zero; unary complement can be implemented as XOR with all ones.

Unary floating-point negation cannot be implemented as subtraction from zero, because many floating-point representations allow a *negative zero*. The

[1]A different way of checking for `nil` is to unmap page 0 in the virtual-memory page tables, so that attempting to fetch/store fields of a `nil` record results in a page fault.

negation of negative zero is positive zero, and vice versa. Some numerical programs rely on identities such as $-0 < 0$. Thus, the `Tree` language does not support unary negation very well.

Fortunately, the Tiger language doesn't support floating-point numbers; but in a real compiler, a new operator would have to be added for floating negation.

CONDITIONALS

The result of a comparison operator will be a `Cx` expression: a statement *s* that will jump to any true-destination and false-destination you specify.

Making "simple" `Cx` expressions from `Absyn` comparison operators is easy with the CJUMP operator. However, the whole point of the `Cx` representation is that conditional expressions can be combined easily with the Tiger operators `&` and `|`. Therefore, an expression such as `x<5` will be translated as a `Cx` with

$$\begin{aligned} \texttt{trues} &= \{t\} \\ \texttt{falses} &= \{f\} \\ \texttt{stm} &= \text{CJUMP}(\text{LT}, x, \text{CONST}(5), \square_t, \square_f). \end{aligned}$$

The `&` and `|` operators of the Tiger language, which combine conditionals with short-circuit conjunction and disjunction (*and* and *or*) respectively, have already been translated into `if`-expressions in the abstract syntax.

The most straightforward thing to do with an `if`-expression

if e_1 then e_2 else e_3

is to treat e_1 as a `Cx` expression, and e_2 and e_3 as `Ex` expressions. That is, apply `unCx` to e_1 and `unEx` to e_2 and e_3. Make two labels t and f to which the conditional will branch. Allocate a temporary r, and after label t, move e_2 to r; after label f, move e_3 to r. Both branches should finish by jumping to a newly created "join" label.

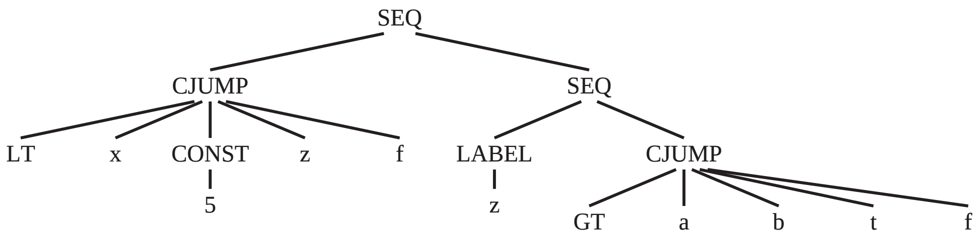
This will produce perfectly correct results. However, the translated code may not be very efficient at all. If e_2 and e_3 are both "statements" (expressions that return no value), then their representation is likely to be `Nx`, not `Ex`. Applying `unEx` to them will work – a coercion will automatically be applied – but it might be better to recognize this case specially.

Even worse, if e_2 or e_3 is a `Cx` expression, then applying the `unEx` coercion to it will yield a horrible tangle of jumps and labels. It is much better to recognize this case specially.

For example, consider

if $x < 5$ then $a > b$ else 0

As shown above, $x < 5$ translates into `Cx`(s_1); similarly, $a > b$ will be translated as `Cx`(s_2) for some s_2. The whole `if`-statement should come out approximately as



$\text{SEQ}(s_1(z,\ f),\ \text{SEQ}(\text{LABEL}\ z, s_2(t,\ f)))$

for some new label z. The shorthand $s_1(z,\ f)$ means the `Cx` statement s_1 with its `trues` labels filled in with z and its `falses` labels filled in with f.

String comparison. Because the string equality operator is complicated (it must loop through the bytes checking byte-for-byte equality), the compiler should call a runtime-system function `stringEqual` that implements it. This function returns a 0 or 1 value (false or true), so the CALL tree is naturally contained within an `Ex` expression. String not-equals can be implemented by generating `Tree` code that complements the result of the function call.

STRINGS

A string literal in the Tiger (or C) language is the constant address of a segment of memory initialized to the proper characters. In assembly language a label is used to refer to this address from the middle of some sequence of instructions. At some other place in the assembly-language program, the *definition* of that label appears, followed by the assembly-language pseudo-instruction to reserve and initialize a block of memory to the appropriate characters.

For each string literal `lit`, the `Translate` module makes a new label `lab`, and returns the tree `T_NAME(lab)`. It also puts the assembly-lan-guage fragment `F_string(lab,lit)` onto a global list of such fragments to be handed to the code emitter. "Fragments" are discussed further on page 172; translation of string fragments to assembly language, on page 269.

All string operations are performed in functions provided by the runtime system; these functions heap-allocate space for their results, and return pointers. Thus, the compiler (almost) doesn't need to know what the representation is, as long as it knows that each string pointer is exactly one word long. I say "almost" because string literals must be represented.

But how are strings represented in Tiger? In Pascal, they are fixed-length arrays of characters; literals are padded with blanks to make them fit. This is not very useful. In C, strings are pointers to variable-length, zero-terminated sequences. This is much more useful, though a string containing a zero byte cannot be represented.

Tiger strings should be able to contain arbitrary 8-bit codes (including zero). A simple representation that serves well is to have a string pointer point to a one-word integer containing the length (number of characters), followed immediately by the characters themselves. Then the `string` function in the machine-specific `Frame` module (`mipsframe.c`, `sparcframe.c`, `pentiumframe.c`, etc.) can make a string with a label definition, an assembly-language pseudo-instruction to make a word containing the integer length, and a pseudo-instruction to emit character data.

RECORD AND ARRAY CREATION

The Tiger language construct $a\{f_1 = e_1,\ f_2 = e_2, \ldots,\ f_n = e_n\}$ creates an n-element record initialized to the values of expressions e_i. Such a record may outlive the procedure activation that creates it, so it cannot be allocated on the stack. Instead, it must be allocated on the *heap*. There is no provision for freeing records (or strings); industrial-strength Tiger systems should have a *garbage collector* to reclaim unreachable records (see Chapter 13).

The simplest way to create a record is to call an external memory-allocation function that returns a pointer to an n-word area into a new temporary r. Then a series of MOVE trees can initialize offsets $0, 1W, 2W, \ldots, (n-1)W$ from r with the translations of expressions e_i. Finally the result of the whole expression is $\text{TEMP}(r)$, as shown in Figure 7.4.

In an industrial compiler, calling `malloc` (or its equivalent) on every record creation might be too slow; see Section 13.7.

Array creation is very much like record creation, except that all the fields are initialized to the same value. The external `initArray` function can take the array length and the initializing value as arguments.

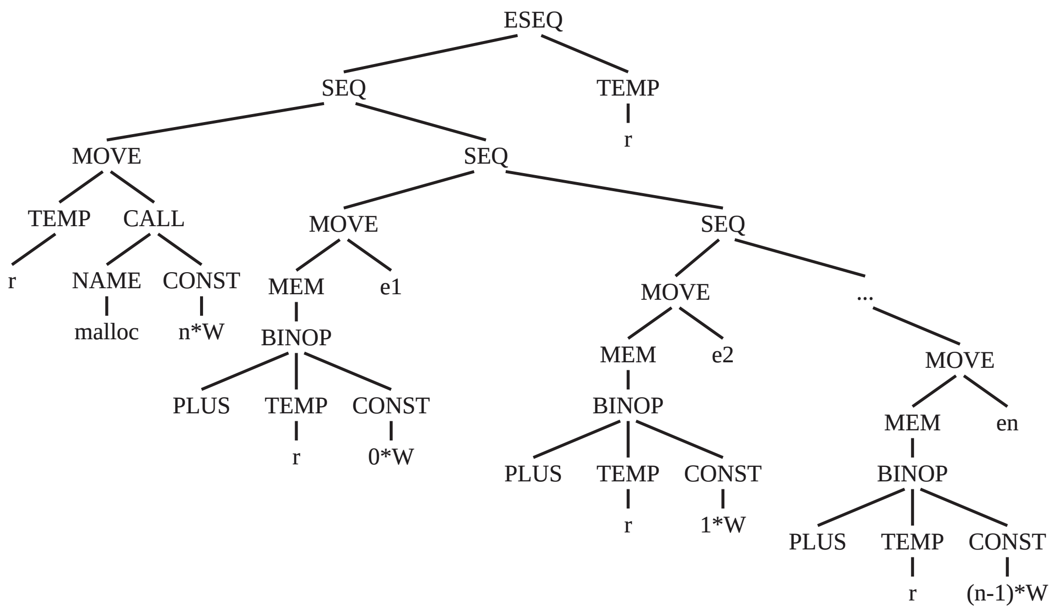


FIGURE 7.4. Record allocation.

Calling runtime-system functions. To call an external function named `init-Array` with arguments *a*, *b*, simply generate a CALL such as

CALL(NAME(`Temp_namedlabel("initArray")`),
  `T_ExpList`(*a*, `T_ExpList`(*b*, `NULL`)))

This refers to an external function `initArray` which is written in a language such as C or assembly language – it cannot be written in Tiger because Tiger has no mechanism for manipulating raw memory.

But on some operating systems, the C compiler puts an underscore at the beginning of each label; and the calling conventions for C functions may differ from those of Tiger functions; and C functions don't expect to receive a static link, and so on. All these target-machine-specific details should be encapsulated into a function provided by the `Frame` structure:

```
/* frame.h */
    ⋮
T_exp F_externalCall(string s,T_expList args);
```

where `F_externalCall` takes the name of the external procedure and the arguments to be passed.

The implementation of `externalCall` depends on the relationship between Tiger's procedure-call convention and that of the external function. The simplest possible implementation looks like

```
T_exp F_externalCall(string s, T_expList args) {
  return T_Call(T_Name(Temp_namedlabel(s)), args);
}
```

but may have to be adjusted for static links, or underscores in labels, and so on.

WHILE LOOPS

The general layout of a while loop is

```
test:
        if not(condition) goto done
        body
        goto test
done:
```

If a `break` statement occurs within the *body* (and not nested within any interior while statements), the translation is simply a JUMP to *done*.

So that `transExp` can translate `break` statements, it will have a new formal parameter `break` that is the *done* label of the nearest enclosing loop. In translating a while loop, `transExp` is called upon *body* with the *done* label passed as the `break` parameter. When `transExp` is recursively calling itself in nonloop contexts, it can simply pass down the same `break` parameter that was passed to it.

The `break` argument must also be added to the `transDec` function.

FOR LOOPS

A for statement can be expressed using other kinds of statements:

```
                        let var i := lo
                             var limit := hi
for i := lo to hi       in while i <= limit
  do body                      do (body; i := i+1)
                        end
```

A very straightforward approach to the translation of for statements is to rewrite the *abstract syntax* into the abstract syntax of the let/while expression shown, and then call `transExp` on the result.

This is almost right, but consider the case where *limit*=*maxint*. Then $i + 1$ will overflow; either a hardware exception will be raised, or $i \leq limit$ will always be true! The solution is to put the test at the *bottom* of the loop, where $i < limit$ can be tested *before* the increment. Then an extra test will be needed before entering the loop to check $lo \leq hi$.

FUNCTION CALL

Translating a function call $f(a_1, ...a_n)$ is simple, except that the static link must be added as an implicit extra argument:

$\text{CALL}(\text{NAME}\ l_f, [sl, e_1, e_2, ..., e_n])$

Here l_f is the label for f, and sl is the static link, computed as described in Chapter 6. To do this computation, both the `level` of f and the `level` of the function calling f are required. A chain of (zero or more) offsets found in successive `level` descriptors is fetched, starting with the frame pointer TEMP(FP) defined by the `Frame` module.

7.3 DECLARATIONS

The clause to type-check let expressions was shown on page 118. It is not hard to augment this clause to translate into `Tree` expressions. `TransExp` and `transDec` now take more arguments than before (as described elsewhere in this chapter), and `transDec` must now return an extra result – the `Tr_exp` resulting from the evaluation of the declaration (this will be explained below).

The call to `transDec` will now side-effect the `frame` data structure: for each variable declaration within the declaration, additional space will be reserved in the current level's `frame`. Also, for each function declaration, a new "fragment" of `Tree` code will be kept for the function's body.

VARIABLE DEFINITION

The `transDec` function, described in Chapter 5, updates the value environment and type environment that are used in processing the body of a let expression.

However, the initialization of a variable translates into a `Tree` expression that must be put just before the body of the let. Therefore, `transDec` must also return a `Tr_exp` containing assignment expressions that accomplish these initializations.

If `transDec` is applied to function and type declarations, the result will be a "no-op" expression such as `Ex(CONST(0))`.

FUNCTION DEFINITION

Each Tiger function is translated into a segment of assembly language with a *prologue*, a *body*, and an *epilogue*. The body of a Tiger function is an expression, and the *body* of the translation is simply the translation of that expression.

The *prologue*, which precedes the body in the assembly-language version of the function, contains

1. pseudo-instructions, as needed in the particular assembly language, to announce the beginning of a function;
2. a label definition for the function name;
3. an instruction to adjust the stack pointer (to allocate a new frame);
4. instructions to save "escaping" arguments – including the static link – into the frame, and to move nonescaping arguments into fresh temporary registers;
5. store instructions to save any callee-save registers – including the return address register – used within the function.

Then comes

6. the function *body*.

The *epilogue* comes after the body and contains

7. an instruction to move the return value (result of the function) to the register reserved for that purpose;
8. load instructions to restore the callee-save registers;
9. an instruction to reset the stack pointer (to deallocate the frame);
10. a *return* instruction (JUMP to the return address);
11. pseudo-instructions, as needed, to announce the end of a function.

Some of these items (1, 3, 9, and 11) depend on exact knowledge of the frame size, which will not be known until after the register allocator determines how many local variables need to be kept in the frame because they don't fit in registers. So these instructions should be generated very late, in a `FRAME` function called `procEntryExit3` (see also page 269). Item 2 (and 10), nestled between 1 and 3 (and 9 and 11, respectively) are also handled at that time.

To implement 7, the `Translate` phase should generate a move instruction

MOVE(`RV`, `body`)

that puts the result of evaluating the body in the return value (`RV`) location specified by the machine-specific `frame` structure:

```
 /* frame.h */
    ⋮
Temp_temp F_RV(void);
```

Item 4 (moving incoming formal parameters), and 5 and 8 (the saving and restoring of callee-save registers), are part of the *view shift* described page 137. They should be done by a function in the `Frame` module:

```
 /* frame.h */
    ⋮
T_stm F_procEntryExit1(F_frame frame, T_stm stm);
```

The implementation of this function will be discussed on page 267. `Translate` should apply it to each procedure body (items 5–7) as it is translated.

FRAGMENTS

Given a Tiger function definition comprising a `level` and an already-translated `body` expression, the `Translate` phase should produce a descriptor for the function containing this necessary information:

`frame`: The frame descriptor containing machine-specific information about local variables and parameters;
`body`: The result returned from `procEntryExit1`.

Call this pair a *fragment* to be translated to assembly language. It is the second kind of fragment we have seen; the other was the assembly-language pseudo-instruction sequence for a string literal. Thus, it is useful to define (in the `Translate` interface) a `frag` datatype:

```
 /* frame.h */
    ⋮
typedef struct F_frag_ *F_frag;
struct F_frag_ { enum {F_stringFrag, F_procFrag} kind;
              union {
                struct {Temp_label label;
                        string str;} stringg;
                struct {T_stm body; F_frame frame;} proc;
              } u;
            };
F_frag F_StringFrag(Temp_label label, string str);
F_frag F_ProcFrag(T_stm body, F_frame frame);

typedef struct F_fragList_ *F_fragList;
struct F_fragList_ {F_frag head; F_fragList tail;};
F_fragList F_FragList(F_frag head, F_fragList tail);
```

```
/* translate.h */
  ⋮
void Tr_procEntryExit(Tr_level level, Tr_exp body,
                      Tr_accessList formals);
F_fragList Tr_getResult(void);
```

The semantic analysis phase calls upon `Tr_newLevel` in processing a function header. Later it calls other interface functions of `Translate` to translate the body of the Tiger function; this has the side effect of remembering `DataFrag` fragments for any string literals encountered (see pages 166 and 269). Finally the semantic analyzer calls `procEntryExit`, which has the *side effect* of remembering a `ProcFrag`.

All the remembered fragments go into a private fragment list within `Translate`; then `getResult` can be used to extract the fragment list.

PROGRAM TRANSLATION TO TREES

Design `translate.h`, implement `translate.c`, and rewrite the `Semant` structure to call upon `Translate` appropriately. The result of calling `SEM_transProg` should be a `F_fragList`.

To keep things simpler (for now), keep all local variables in the frame; do not bother with `FindEscape`, and assume that every variable escapes.

In the `Frame` module, a "dummy" implementation

```
T_stm F_procEntryExit1(F_frame frame, T_stm stm) {
    return stm;
}
```

is suitable for preliminary testing of `Translate`.

Supporting files in `$TIGER/chap7` include:

`tree.h,tree.c`	Data types for the `Tree` language.
`printtree.c`	Functions to display trees for debugging.

and other files as before.

A simpler Translate. To simplify the implementation of `Translate`, you may do without the `Ex`, `Nx`, `Cx` constructors. The entire `Translate` module can be done with ordinary value-expressions. This makes `Tr_exp` identical to `T_exp`. That is, instead of `Ex`(e), just use e. Instead of `Nx`(s), use the expression ESEQ(s, CONST 0). For conditionals, instead of a `Cx`, use an expression that just evaluates to 1 or 0.

The intermediate representation trees produced from this kind of naive translation will be bulkier and slower than a "fancy" translation. But they *will* work correctly, and in principle a fancy back-end optimizer might be able to clean up the clumsiness. In any case, a clumsy but correct `Translate` module is better than a fancy one that doesn't work.

EXERCISES

7.1 Suppose a certain compiler translates all expressions and subexpressions into T_exp trees, and does not use the `Nx` and `Cx` constructors to represent expressions in different ways. Draw a picture of the IR tree that results from each of the following expressions. Assume all variables are nonescaping unless specified otherwise.

a. `a+5`

b. `b[i+1]`

c. `p.z.x`, where `p` is a Tiger variable whose type is

```
type m = {x:int,y:int,z:m}
```

d. `write(" ")`, as it appears on line 13 of Program 6.3.

e. `a<b`, which should be implemented by making an ESEQ whose left-hand side moves a 1 or 0 into some newly defined temporary, and whose right-hand side is the temporary.

f. `if a then b else c`, where `a` is an integer variable (true if $\neq 0$); this should also be translated using an ESEQ.

g. `a := x+y`, which should be translated with an EXP node at the top.

h. `if a<b then c:=a else c:=b`, translated using the `a<b` tree from part (e) above; the whole statement will therefore be rather clumsy and inefficient.

i. `if a<b then c:=a else c:=b`, translated in a less clumsy way.

7.2 Translate each of these expressions into IR trees, but using the `Ex`, `Nx`, and `Cx` constructors as appropriate. In each case, just draw pictures of the trees; an `Ex` tree will be a Tree `exp`, an `Nx` tree will be a Tree `stm`, and a `Cx` tree will be a `stm` with holes labeled *true* and *false* into which labels can later be placed.

a. `a+5`

b. `output := concat(output,s)`, as it appears on line 8 of Program 6.3. The `concat` function is part of the standard library (see page 525), and for purposes of computing its static link, assume it is at the same level of nesting as the `prettyprint` function.

c. `b[i+1]:=0`

d. `(c:=a+1; c*c)`

e. `while a>0 do a := a-1`

f. `a<b` moves a 1 or 0 into some newly defined temporary, and whose right-hand side is the temporary.

g. `if a then b else c`, where `a` is an integer variable (true if $\neq 0$).

h. `a := x+y`

i. `if a<b then a else b`

j. `if a<b then c:=a else c:=b`

7.3 Using the C compiler of your choice (or a compiler for another language), translate some functions to assembly language. On Unix this is done with the `-S` option to the C compiler.

Then identify all the components of the calling sequence (items 1–11), and explain what each line of assembly language does (especially the pseudo-instructions that comprise items 1 and 11). Try one small function that returns without much computation (a *leaf* function), and one that calls another function before eventually returning.

7.4 The `Tree` intermediate language has no operators for floating-point variables. Show how the language would look with new binops for floating-point arithmetic, and new relops for floating-point comparisons. You may find it useful to introduce a variant of MEM nodes to describe fetching and storing floating-point values.

***7.5** The `Tree` intermediate language has no provision for data values that are not exactly one word long. The C programming language has signed and unsigned integers of several sizes, with conversion operators among the different sizes. Augment the intermediate language to accommodate several sizes of integers, with conversions among them.

Hint: Do not distinguish signed values from unsigned values in the intermediate trees, but do distinguish between signed operators and unsigned operators. See also Fraser and Hanson [1995], sections 5.5 and 9.1.

8 Basic Blocks and Traces

ca-non-i-cal: reduced to the simplest or clearest schema possible

Webster's Dictionary

The trees generated by the semantic analysis phase must be translated into assembly or machine language. The operators of the `Tree` language are chosen carefully to match the capabilities of most machines. However, there are certain aspects of the tree language that do not correspond exactly with machine languages, and some aspects of the `Tree` language interfere with compile-time optimization analyses.

For example, it's useful to be able to evaluate the subexpressions of an expression in any order. But the subexpressions of `Tree.exp` can contain side effects – ESEQ and CALL nodes that contain assignment statements and perform input/output. If tree expressions did not contain ESEQ and CALL nodes, then the order of evaluation would not matter.

Some of the mismatches between `Trees` and machine-language programs are:

- The CJUMP instruction can jump to either of two labels, but real machines' conditional jump instructions fall through to the next instruction if the condition is false.
- ESEQ nodes within expressions are inconvenient, because they make different orders of evaluating subtrees yield different results.
- CALL nodes within expressions cause the same problem.
- CALL nodes within the argument-expressions of other CALL nodes will cause problems when trying to put arguments into a fixed set of formal-parameter registers.

Why does the `Tree` language allow ESEQ and two-way CJUMP, if they are so troublesome? Because they make it much more convenient for the `Translate` (translation to intermediate code) phase of the compiler.

We can take any tree and rewrite it into an equivalent tree without any of the cases listed above. Without these cases, the only possible parent of a SEQ node is another SEQ; all the SEQ nodes will be clustered at the top of the tree. This makes the SEQs entirely uninteresting; we might as well get rid of them and make a linear list of `T_stm`s.

The transformation is done in three stages: First, a tree is rewritten into a list of *canonical trees* without SEQ or ESEQ nodes; then this list is grouped into a set of *basic blocks*, which contain no internal jumps or labels; then the basic blocks are ordered into a set of *traces* in which every CJUMP is immediately followed by its `false` label.

Thus the module `Canon` has these tree-rearrangement functions:

```
/* canon.h */

typedef struct C_stmListList_ *C_stmListList;
struct C_block { C_stmListList stmLists; Temp_label label;};
struct C_stmListList_ { T_stmList head; C_stmListList tail;};

T_stmList C_linearize(T_stm stm);
struct C_block C_basicBlocks(T_stmList stmList);
T_stmList C_traceSchedule(struct C_block b);
```

`Linearize` removes the ESEQs and moves the CALLs to top level. Then `BasicBlocks` groups statements into sequences of straight-line code. Finally `traceSchedule` orders the blocks so that every CJUMP is followed by its `false` label.

8.1 CANONICAL TREES

Let us define *canonical trees* as having these properties:

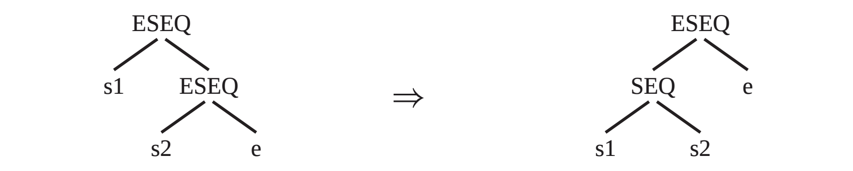
1. No SEQ or ESEQ.
2. The parent of each CALL is either EXP(. . .) or MOVE(TEMP t, . . .).

TRANSFORMATIONS ON ESEQ

How can the ESEQ nodes be eliminated? The idea is to lift them higher and higher in the tree, until they can become SEQ nodes.

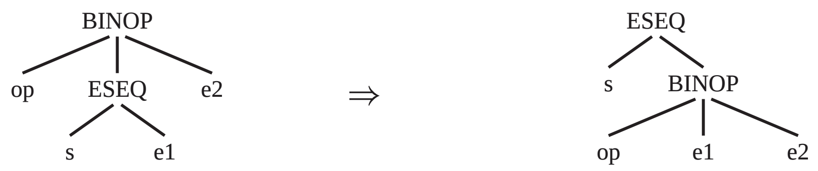
Figure 8.1 gives some useful identities on trees.

(1)



$\text{ESEQ}(s_1, \text{ESEQ}(s_2, e)) \quad = \quad \text{ESEQ}(\text{SEQ}(s_1, s_2), e)$

(2)



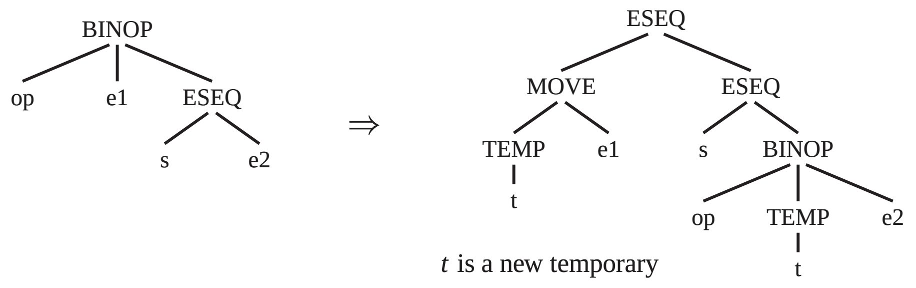
$\text{BINOP}(op, \text{ESEQ}(s, e_1), e_2) \quad = \quad \text{ESEQ}(s, \text{BINOP}(op, e_1, e_2))$

$\text{MEM}(\text{ESEQ}(s, e_1)) \quad = \quad \text{ESEQ}(s, \text{MEM}(e_1))$

$\text{JUMP}(\text{ESEQ}(s, e_1)) \quad = \quad \text{SEQ}(s, \text{JUMP}(e_1))$

$\text{CJUMP}(op, \text{ESEQ}(s, e_1), e_2, l_1, l_2) \quad = \quad \text{SEQ}(s, \text{CJUMP}(op, e_1, e_2, l_1, l_2))$

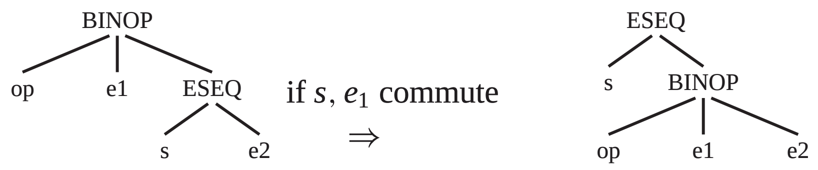
(3)



$\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2)) \quad = \quad \text{ESEQ}(\text{MOVE}(\text{TEMP } t, e_1), \text{ESEQ}(s, \text{BINOP}(op, \text{TEMP } t, e_2)))$

$\text{CJUMP}(op, e_1, \text{ESEQ}(s, e_2), l_1, l_2) \quad = \quad \text{SEQ}(\text{MOVE}(\text{TEMP } t, e_1), \text{SEQ}(s, \text{CJUMP}(op, \text{TEMP } t, e_2, l_1, l_2)))$

(4)



if s, e_1 commute

$\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2)) \quad = \quad \text{ESEQ}(s, \text{BINOP}(op, e_1, e_2))$

$\text{CJUMP}(op, e_1, \text{ESEQ}(s, e_2), l_1, l_2) \quad = \quad \text{SEQ}(s, \text{CJUMP}(op, e_1, e_2, l_1, l_2))$

FIGURE 8.1. Identities on trees (see also Exercise 8.1).

Identity (1) is obvious. So is identity (2): Statement s is to be evaluated; then e_1; then e_2; then the sum of the expressions is returned. If s has side effects that affect e_1 or e_2, then either the left-hand side or the right-hand side of the first equation will execute those side effects before the expressions are evaluated.

Identity (3) is more complicated, because of the need not to interchange the evaluations of s and e_1. For example, if s is $\text{MOVE}(\text{MEM}(x), y)$ and e_1 is $\text{BINOP}(\text{PLUS}, \text{MEM}(x), z)$, then the program will compute a different result if s is evaluated before e_1 instead of after. Our goal is simply to pull s out of the BINOP expression; but now (to preserve the order of evaluation) we must pull e_1 out of the BINOP with it. To do so, we assign e_1 into a new temporary t, and put t inside the BINOP.

It may happen that s causes no side effects that can alter the result produced by e_1. This will happen if the temporaries and memory locations assigned by s are not referenced by e_1 (and s and e_1 don't both perform external I/O). In this case, identity (4) can be used.

We cannot always tell if two expressions commute. For example, whether $\text{MOVE}(\text{MEM}(x), y)$ commutes with $\text{MEM}(z)$ depends on whether $x = z$, which we cannot always determine at compile time. So we *conservatively approximate* whether statements commute, saying either "they definitely do commute" or "perhaps they don't commute." For example, we know that any statement "definitely commutes" with the expression $\text{CONST}(n)$, so we can use identity (4) to justify special cases like

$$\text{BINOP}(op, \text{CONST}(n), \text{ESEQ}(s, e)) = \text{ESEQ}(s, \text{BINOP}(op, \text{CONST}(n), e)).$$

The `commute` function estimates (very naively) whether two expressions commute:

```
static bool isNop(T_stm x) {
  return x->kind == T_EXP && x->u.EXP->kind == T_CONST;
}
static bool commute(T_stm x, T_exp y) {
 return isNop(x) || y->kind==T_NAME || y->kind==T_CONST;
}
```

A constant commutes with any statement, and the empty statement commutes with any expression. Anything else is assumed not to commute.

GENERAL REWRITING RULES

In general, for each kind of `Tree` statement or expression we can identify the subexpressions. Then we can make rewriting rules, similar to the ones in Figure 8.1, to pull the ESEQs out of the statement or expression.

For example, in $[e_1, e_2, \text{ESEQ}(s, e_3)]$, the statement s must be pulled leftward past e_2 and e_1. If they commute, we have $(s;\ [e_1, e_2, e_3])$. But suppose e_2 does not commute with s; then we must have

$$(\text{SEQ}(\text{MOVE}(t_1, e_1), \text{SEQ}(\text{MOVE}(t_2, e_2), s));\quad [\text{TEMP}(t_1), \text{TEMP}(t_2), e_3])$$

Or if e_2 commutes with s but e_1 does not, we have

$$(\text{SEQ}(\text{MOVE}(t_1, e_1), s);\quad [\text{TEMP}(t_1),\ e_2,\ e_3])$$

The `reorder` function takes a list of expressions and returns a pair of (statement, expression-list). The statement contains all the things that must be executed before the expression-list. As shown in these examples, this includes all the statement-parts of the ESEQs, as well as any expressions to their left with which they did not commute. When there are no ESEQs at all we will use EXP(CONST 0), which does nothing, as the statement.

Algorithm. Step one is to make a "subexpression-extraction" method for each kind. Step two is to make a "subexpression-insertion" method: given an ESEQ-clean version of each subexpression, this builds a new version of the expression or statement.

```
typedef struct expRefList_ *expRefList;
struct expRefList_ {T_exp *head; expRefList tail;};

struct stmExp {T_stm s; T_exp e;};

static T_stm reorder(expRefList rlist);

static T_stm do_stm(T_stm stm);
static struct stmExp do_exp(T_exp exp);
```

The `reorder` function is supposed to pull all the ESEQs out of a list of expressions and combine the *statement-parts* of these ESEQ into one big `T_stm`. The argument to `reorder` is a list of *references* to the immediate subexpressions of that statement. Figure 8.2 illustrates the use of a pointer to a pointer. If we call `reorder`(l_2), we are saying, "please pull any ESEQs out of the children and grandchildren of this BINOP node e_2. For your convenience, the places where it points to its children are at the locations pointed to by the list l_2. For each child that is an ESEQ(s_k, e_k), you should update the child-pointer to point to e_k instead and put s_k on the big sequence of statements that you will return as a result.”

`Reorder`(l_2) calls upon an auxiliary function `do_exp` on each expression in the list l_2, that is, the expressions e_1 and e_3. `Do_exp`(e_1) returns a statement s_1 and an expression e_1', where e_1' contains no ESEQs, such that ESEQ(s, e_1') would be equivalent to the original expression e_1. In this case, since e_1 is so trivial, s_1 will be a no-op statement EXP(CONST(0)) and $e_1' = e_1$. But if expression e_3’s MEM node pointed to ESEQ$(s_x$, TEMP $a)$, then `do_exp`(e_3) will yield $s_3 = s_x$ and $e_3' =$ MEM(TEMP a).

The implementation of `do_exp` is rather simple. For any kind of expression except ESEQ, `do_exp` just makes a list of the subexpression references and calls `reorder`:

```
static struct stmExp do_exp(T_exp exp) {
switch(exp->kind) {
 case T_BINOP:
  return StmExp(reorder(ExpRefList(&exp->u.BINOP.left,
               ExpRefList(&exp->u.BINOP.right,NULL))), exp);
 case T_MEM:
  return StmExp(reorder(ExpRefList(&exp->u.MEM,NULL)), exp);
 case T_ESEQ: {
    struct stmExp x = do_exp(exp->u.ESEQ.exp);
    return StmExp(seq(do_stm(exp->u.ESEQ.stm), x.s), x.e);
   }
 case T_CALL:
    return StmExp(reorder(get_call_rlist(exp)), exp);
 default:
    return StmExp(reorder(NULL), exp);
}}
```

The function `seq`(s_1, s_2) just returns a statement equivalent to SEQ(s_1, s_2), but in the very common case that s_1 or s_2 is a no-op, we can so something simpler:

```
static T_stm seq(T_stm x, T_stm y) {
 if (isNop(x)) return y;
 if (isNop(y)) return x;
 return T_Seq(x,y);
}
```

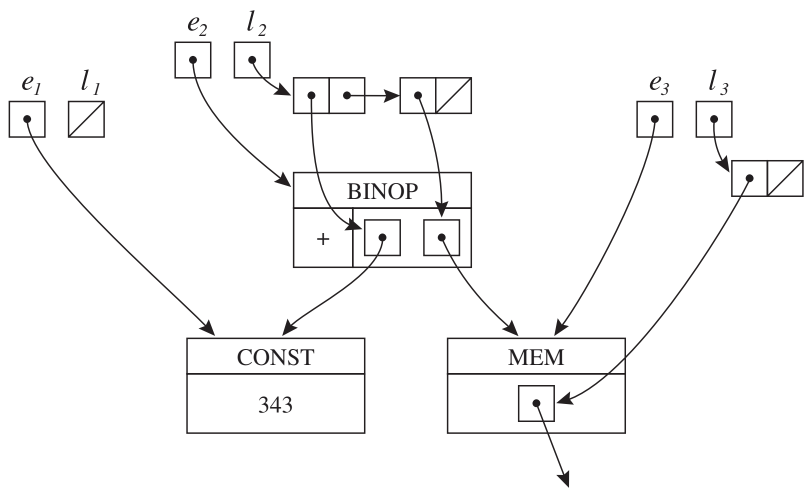


FIGURE 8.2. List-of-refs argument passed to `reorder`.

The ESEQ case of `do_exp` must call `do_stm`, which pulls all the ESEQs out of a statement. It also works by making a list of all the subexpression references and calling `reorder`:

```
static T_stm do_stm(T_stm stm) {
switch (stm->kind) {
 case T_SEQ:
  return seq(do_stm(stm->u.SEQ.left),
             do_stm(stm->u.SEQ.right));
 case T_JUMP:
  return seq(reorder(ExpRefList(&stm->u.JUMP.exp,NULL)), stm);
 case T_CJUMP:
  return seq(reorder(ExpRefList(&stm->u.CJUMP.left,
              ExpRefList(&stm->u.CJUMP.right,NULL))), stm);
 case T_MOVE:
   ⋮ see below
 case T_EXP:
  if (stm->u.EXP->kind == T_CALL)
       return seq(reorder(get_call_rlist(stm->u.EXP)), stm);
  else return seq(reorder(ExpRefList(&stm->u.EXP, NULL)),
                  stm);
 default:
  return stm;
}}
```

The left-hand operand of the MOVE statement is not considered a subexpression, because it is the *destination* of the statement – its value is not used by the statement. However, if the destination is a memory location, then the *address* acts like a source. Thus we have,

```
static T_stm do_stm(T_stm stm) {
    ⋮
  case T_MOVE:
    if (stm->u.MOVE.dst->kind == T_TEMP &&
        stm->u.MOVE.src->kind == T_CALL)
      return seq(reorder(get_call_rlist(stm->u.MOVE.src)),
                 stm);
    else if (stm->u.MOVE.dst->kind == T_TEMP)
      return seq(reorder(ExpRefList(&stm->u.MOVE.src, NULL)),
                 stm);
    else if (stm->u.MOVE.dst->kind == T_MEM)
      return seq(reorder(ExpRefList(&stm->u.MOVE.dst->u.MEM,
                      ExpRefList(&stm->u.MOVE.src, NULL))),
                 stm);
    else if (stm->u.MOVE.dst->kind == T_ESEQ) {
      T_stm s = stm->u.MOVE.dst->u.ESEQ.stm;
      stm->u.MOVE.dst = stm->u.MOVE.dst->u.ESEQ.exp;
      return do_stm(T_Seq(s, stm));
    }
    ⋮
```

With the assistance of `do_exp` and `do_stm`, the `reorder` function can pull the statement s_i out of each expression e_i on its list of references, going from right to left.

MOVING CALLS TO TOP LEVEL

The `Tree` language permits CALL nodes to be used as subexpressions. However, the actual implementation of CALL will be that each function returns its result in the same dedicated return-value register TEMP(RV). Thus, if we have

BINOP(PLUS, CALL(...), CALL(...))

the second call will overwrite the RV register before the PLUS can be executed.

We can solve this problem with a rewriting rule. The idea is to assign each return value immediately into a fresh temporary register, that is

$$\text{CALL}(\textit{fun}, \textit{args}) \quad \rightarrow \quad \text{ESEQ}(\text{MOVE}(\text{TEMP } t, \text{CALL}(\textit{fun}, \textit{args})), \text{TEMP } t)$$

Now the ESEQ-eliminator will percolate the MOVE up outside of its containing BINOP (etc.) expressions.

This technique will generate a few extra MOVE instructions, which the register allocator (Chapter 11) can clean up.

The rewriting rule is implemented as follows: `reorder` replaces any occurrence of CALL(f, *args*) by

$$\text{ESEQ}(\text{MOVE}(\text{TEMP } t_{\text{new}}, \text{CALL}(f, \mathit{args})),\ \text{TEMP } t_{\text{new}})$$

and calls itself again on the ESEQ. But `do_stm` recognizes the pattern

$$\text{MOVE}(\text{TEMP } t_{\text{new}}, \text{CALL}(f, \mathit{args})),$$

and does not call `reorder` on the CALL node in that case, but treats the f and *args* as the children of the MOVE node. Thus, `reorder` never "sees" any CALL that is already the immediate child of a MOVE. Occurrences of the pattern EXP(CALL(f, *args*)) are treated similarly.

A LINEAR LIST OF STATEMENTS

Once an entire function body s_0 is processed with `do_stm`, the result is a tree s_0' where all the SEQ nodes are near the top (never underneath any other kind of node). The `linearize` function repeatedly applies the rule

$$\text{SEQ}(\text{SEQ}(a, b), c) \ = \ \text{SEQ}(a, \mathit{seq}(b, c))$$

The result is that s_0' is linearized into an expression of the form

$$\text{SEQ}(s_1, \text{SEQ}(s_2, \ldots, \text{SEQ}(s_{n-1}, s_n) \ldots))$$

Here the SEQ nodes provide no structuring information at all, and we can just consider this to be a simple list of statements,

$$s_1, s_2, \ldots, s_{n-1}, s_n$$

where none of the s_i contain SEQ or ESEQ nodes.

These rewrite rules are implemented by `linearize`, with an auxiliary function `linear`:

```
static T_stmList linear(T_stm stm, T_stmList right) {
 if (stm->kind == T_SEQ)
    return linear(stm->u.SEQ.left,
             linear(stm->u.SEQ.right,
               right));
 else return T_StmList(stm, right);
}

T_stmList C_linearize(T_stm stm) {
    return linear(do_stm(stm), NULL);
}
```

8.2 TAMING CONDITIONAL BRANCHES

Another aspect of the `Tree` language that has no direct equivalent in most machine instruction sets is the two-way branch of the CJUMP instruction. The `Tree` language CJUMP is designed with two target labels for convenience in translating into trees and analyzing trees. On a real machine, the conditional jump either transfers control (on a `true` condition) or "falls through" to the next instruction.

To make the trees easy to translate into machine instructions, we will rearrange them so that every CJUMP$(cond, l_t, l_f)$ is immediately followed by LABEL(l_f), its "false branch." Each such CJUMP can be directly implemented on a real machine as a conditional branch to label l_t.

We will make this transformation in two stages: first, we take the list of canonical trees and form them into *basic blocks*; then we order the basic blocks into a *trace*. The next sections will define these terms.

BASIC BLOCKS

In determining where the jumps go in a program, we are analyzing the program's *control flow*. Control flow is the sequencing of instructions in a program, ignoring the data values in registers and memory, and ignoring the arithmetic calculations. Of course, not knowing the data values means we cannot know whether the conditional jumps will go to their true or false labels; so we simply say that such jumps can go either way.

In analyzing the control flow of a program, any instruction that is not a jump has an entirely uninteresting behavior. We can lump together any sequence of non-branch instructions into a basic block and analyze the control flow between basic blocks.

A *basic block* is a sequence of statements that is always entered at the beginning and exited at the end, that is:

- The first statement is a LABEL.
- The last statement is a JUMP or CJUMP.
- There are no other LABELs, JUMPs, or CJUMPs.

The algorithm for dividing a long sequence of statements into basic blocks is quite simple. The sequence is scanned from beginning to end; whenever a LABEL is found, a new block is started (and the previous block is ended); whenever a JUMP or CJUMP is found, a block is ended (and the next block is started). If this leaves any block not ending with a JUMP or CJUMP, then a JUMP to the next block's label is appended to the block. If any block has been left without a LABEL at the beginning, a new label is invented and stuck there.

We will apply this algorithm to each function-body in turn. The procedure "epilogue" (which pops the stack and returns to the caller) will not be part of this body, but is intended to follow the last statement. When the flow of program execution reaches the end of the last block, the epilogue should follow. But it is inconvenient to have a "special" block that must come last and that has no JUMP at the end. Thus, we will invent a new label `done` – intended to mean the beginning of the epilogue – and put a JUMP(NAME `done`) at the end of the last block.

In the Tiger compiler, the function `C_basicBlocks` implements this simple algorithm.

TRACES

Now the basic blocks can be arranged in any order, and the result of executing the program will be the same – every block ends with a jump to the appropriate place. We can take advantage of this to choose an ordering of the blocks satisfying the condition that each CJUMP is followed by its false label.

At the same time, we can also arrange that many of the unconditional JUMPs are immediately followed by their target label. This will allow the deletion of these jumps, which will make the compiled program run a bit faster.

A *trace* is a sequence of statements that could be consecutively executed during the execution of the program. It can include conditional branches. A program has many different, overlapping traces. For our purposes in arranging CJUMPs and false-labels, we want to make a set of traces that exactly covers the program: each block must be in exactly one trace. To minimize the number of JUMPs from one trace to another, we would like to have as few traces as possible in our covering set.

A very simple algorithm will suffice to find a covering set of traces. The idea is to start with some block – the beginning of a trace – and follow a possible execution path – the rest of the trace. Suppose block b_1 ends with a JUMP to b_4, and b_4 has a JUMP to b_6. Then we can make the trace b_1, b_4, b_6.

But suppose b_6 ends with a conditional jump CJUMP$(cond, b_7, b_3)$. We cannot know at compile time whether b_7 or b_3 will be next. But we can assume that some execution will follow b_3, so let us imagine it is that execution that we are simulating. Thus, we append b_3 to our trace and continue with the rest

```
Put all the blocks of the program into a list $Q$.
while $Q$ is not empty
    Start a new (empty) trace, call it $T$.
    Remove the head element $b$ from $Q$.
    while $b$ is not marked
        Mark $b$; append $b$ to the end of the current trace $T$.
        Examine the successors of $b$ (the blocks to which $b$ branches);
        if there is any unmarked successor $c$
          $b \leftarrow c$
    (All the successors of b are marked.)
    End the current trace $T$.
```

ALGORITHM 8.3. Generation of traces.

of the trace after b_3. The block b_7 will be in some other trace.

Algorithm 8.3 (which is similar to `C_traceSchedule`) orders the blocks into traces as follows: It starts with some block and follows a chain of jumps, marking each block and appending it to the current trace. Eventually it comes to a block whose successors are all marked, so it ends the trace and picks an unmarked block to start the next trace.

FINISHING UP

An efficient compiler will keep the statements grouped into basic blocks, because many kinds of analysis and optimization algorithms run faster on (relatively few) basic blocks than on (relatively many) individual statements. For the Tiger compiler, however, we seek simplicity in the implementation of later phases. So we will flatten the ordered list of traces back into one long list of statements.

At this point, most (but not all) CJUMPs will be followed by their true or false label. We perform some minor adjustments:

- Any CJUMP immediately followed by its false label we let alone (there will be many of these).
- For any CJUMP followed by its true label, we switch the true and false labels and negate the condition.
- For any CJUMP$(cond, a, b, l_t, l_f)$ followed by neither label, we invent a new false label l'_f and rewrite the single CJUMP statement as three statements, just to achieve the condition that the CJUMP is followed by its false label:

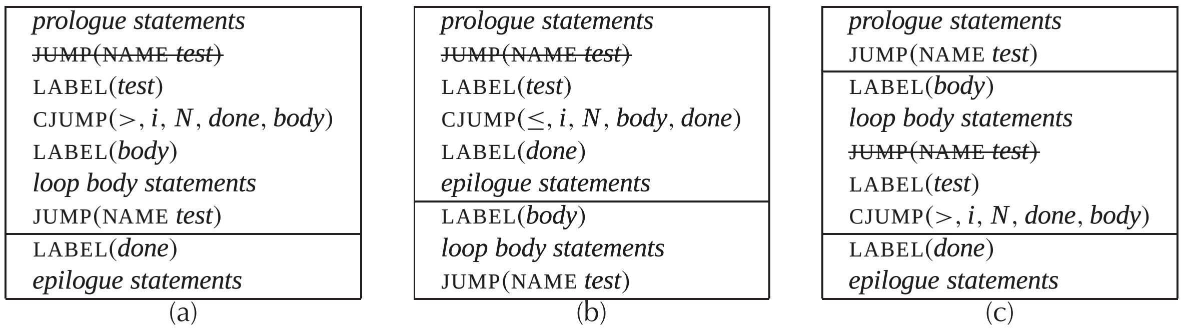


FIGURE 8.4. Different trace coverings for the same program.

CJUMP$(cond, a, b, l_t, l'_f)$
LABEL l'_f
JUMP(NAME l_f)

The trace-generating algorithm will tend to order the blocks so that many of the unconditional JUMPs are immediately followed by their target labels. We can remove such jumps.

OPTIMAL TRACES

For some applications of traces, it is important that any frequently executed sequence of instructions (such as the body of a loop) should occupy its own trace. This helps not only to minimize the number of unconditional jumps, but also may help with other kinds of optimization such as register allocation and instruction scheduling.

Figure 8.4 shows the same program organized into traces in different ways. Figure 8.4a has a CJUMP and a JUMP in every iteration of the while-loop; Figure 8.4b uses a different trace covering, also with CJUMP and a JUMP in every iteration. But 8.4c shows a better trace covering, with no JUMP in each iteration.

The Tiger compiler's `Canon` module doesn't attempt to optimize traces around loops, but it is sufficient for the purpose of cleaning up the `Tree`-statement lists for generating assembly code.

FURTHER READING

The rewrite rules of Figure 8.1 are an example of a *term rewriting system*; such systems have been much studied [Dershowitz and Jouannaud 1990].

Fisher [1981] shows how to cover a program with traces so that frequently executing paths tend to stay within the same trace. Such traces are useful for program optimization and scheduling.

EXERCISES

***8.1** The rewriting rules in Figure 8.1 are a subset of the rules necessary to eliminate all ESEQs from expressions. Show the right-hand side for each of the following incomplete rules:

a. MOVE(TEMP t, ESEQ(s, e)) $\Rightarrow$

b. MOVE(MEM(ESEQ(s, e_1)), e_2) $\Rightarrow$

c. MOVE(MEM(e_1), ESEQ(s, e_2)) $\Rightarrow$

d. EXP(ESEQ(s, e)) $\Rightarrow$

e. EXP(CALL(ESEQ(s, e), *args*)) $\Rightarrow$

f. MOVE(TEMP t, CALL(ESEQ(s, e), *args*)) $\Rightarrow$

g. EXP(CALL(e_1, [e_2, ESEQ(s, e_3), e_4])) $\Rightarrow$

In some cases, you may need two different right-hand sides depending on whether something commutes (just as parts (3) and (4) of Figure 8.1 have different right-hand sides for the same left-hand sides).

8.2 Draw each of the following expressions as a tree diagram, and then apply the rewriting rules of Figure 8.1 and Exercise 8.1, as well as the CALL rule on page 183.

a. MOVE(MEM(ESEQ(SEQ(CJUMP(LT, TEMP$_i$, CONST$_0$, L_{out}, L_{ok}), LABEL$_{ok}$),
TEMP$_i$)), CONST$_1$)

b. MOVE(MEM(MEM(NAME$_a$)), MEM(CALL(TEMP$_f$, [])))

c. BINOP(PLUS, CALL(NAME$_f$, [TEMP$_x$]),
CALL(NAME$_g$, [ESEQ(MOVE(TEMP$_x$, CONST$_0$), TEMP$_x$)]))

***8.3** The directory `$TIGER/chap8` contains an implementation of every algorithm described in this chapter. Read and understand it.

8.4 A primitive form of the `commute` test is shown on page 179. This function is conservative: if interchanging the order of evaluation of the expressions will change the result of executing the program, this function will definitely return false; but if an interchange is harmless, `commute` might return true or false.

Write a more powerful version of `commute` that returns true in more cases, but is still conservative. Document your program by drawing pictures of (pairs of) expression trees on which it will return true.

***8.5** The left-hand side of a MOVE node really represents a destination, not an expression. Consequently, the following rewrite rule is *not* a good idea:

$$\text{MOVE}(e_1, \text{ESEQ}(s, e_2)) \;\rightarrow\; \text{SEQ}(s,\ \text{MOVE}(e_1, e_2)) \qquad \text{if } s, e_1 \text{ commute}$$

Write an statement matching the left side of this rewrite rule that produces a different result when rewritten.

Hint: It is very reasonable to say that the statement $\text{MOVE}(\text{TEMP}_a, \text{TEMP}_b)$ commutes with expression TEMP_b (if a and b are not the same), since TEMP_b yields the same value whether executed before or after the MOVE.

Conclusion: The only subexpression of $\text{MOVE}(\text{TEMP}_a, e)$ is e, and the subexpressions of $\text{MOVE}(\text{MEM}(e_1), e_2)$ are $[e_1, e_2]$; we should not say that a is a subexpression of $\text{MOVE}(a, b)$.

8.6 Break this program into basic blocks.

```
1   $m \leftarrow 0$
2   $v \leftarrow 0$
3   if $v \geq n$ goto 15
4   $r \leftarrow v$
5   $s \leftarrow 0$
6   if $r < n$ goto 9
7   $v \leftarrow v + 1$
8   goto 3
9   $x \leftarrow M[r]$
10  $s \leftarrow s + x$
11  if $s \leq m$ goto 13
12  $m \leftarrow s$
13  $r \leftarrow r + 1$
14  goto 6
15  return $m$
```

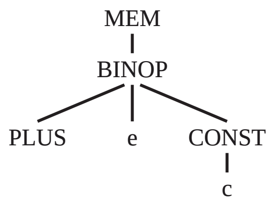
8.7 Express the basic blocks of Exercise 8.6 as statements in the `Tree` intermediate form, and use Algorithm 8.3 to generate a set of traces.

9 Instruction Selection

in-struc-tion: a code that tells a computer to perform a particular operation

Webster's Dictionary

The intermediate representation (`Tree`) language expresses only one operation in each tree node: memory fetch or store, addition or subtraction, conditional jump, and so on. A real machine instruction can often perform several of these primitive operations. For example, almost any machine can perform an add and a fetch in the same instruction, corresponding to the tree



Finding the appropriate machine instructions to implement a given intermediate representation tree is the job of the *instruction selection* phase of a compiler.

TREE PATTERNS

We can express a machine instruction as a fragment of an IR tree, called a *tree pattern*. Then instruction selection becomes the task of tiling the tree with a minimal set of tree patterns.

For purposes of illustration, we invent an instruction set: the *Jouette* architecture. The arithmetic and memory instructions of *Jouette* are shown in Figure 9.1. On this machine, register r_0 always contains zero.

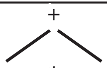
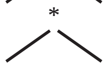
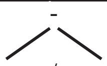
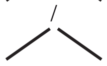
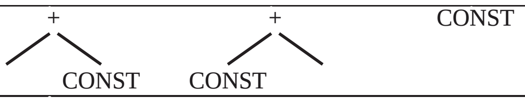
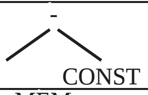
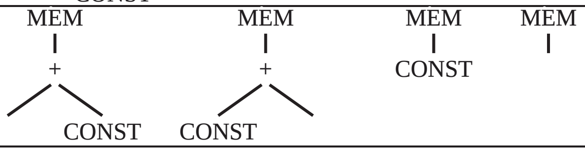
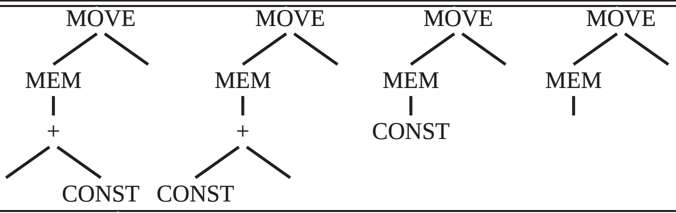
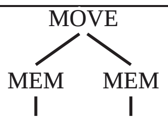
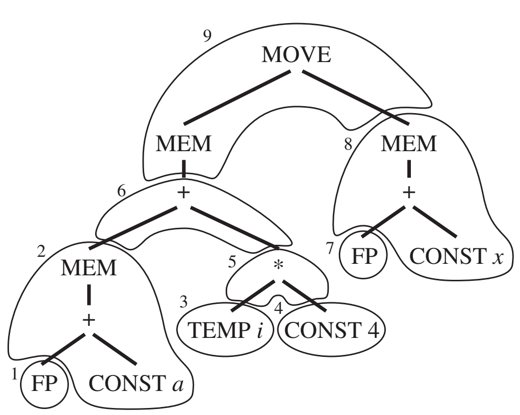
Name	*Effect*	*Trees*
	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	
MUL	$r_i \leftarrow r_j \times r_k$	
SUB	$r_i \leftarrow r_j - r_k$	
DIV	$r_i \leftarrow r_j / r_k$	
ADDI	$r_i \leftarrow r_j + c$	
SUBI	$r_i \leftarrow r_j - c$	
LOAD	$r_i \leftarrow M[r_j + c]$	 MEM + CONST MEM + CONST MEM CONST MEM
STORE	$M[r_j + c] \leftarrow r_i$	 MOVE MEM + CONST MOVE MEM + CONST MOVE MEM CONST MOVE MEM
MOVEM	$M[r_j] \leftarrow M[r_i]$	

FIGURE 9.1. Arithmetic and memory instructions. The notation $M[x]$ denotes the memory word at address x.

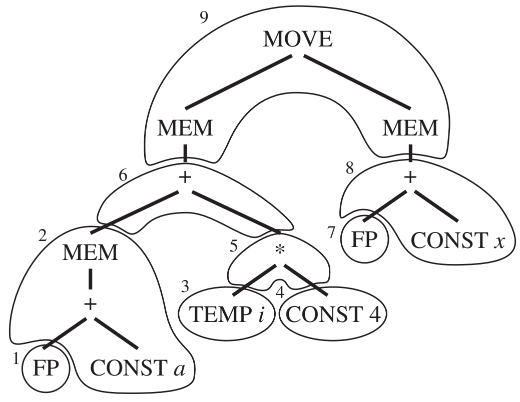
Each instruction above the double line in Figure 9.1 produces a result in a register. The very first entry is not really an instruction, but expresses the idea that a TEMP node is implemented as a register, so it can "produce a result in a register" without executing any instructions at all. The instructions below the double line do not produce results in registers, but are executed only for side effects on memory.

For each instruction, the tree-patterns it implements are shown. Some instructions correspond to more than one tree pattern; the alternate patterns are obtained for commutative operators (+ and *), and in some cases where a register or constant can be zero (LOAD and STORE). In this chapter we abbre-



2	LOAD	$r_1 \leftarrow M[\mathrm{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\mathrm{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$

(a)



2	LOAD	$r_1 \leftarrow M[\mathrm{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \mathrm{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$

(b)

FIGURE 9.2. A tree tiled in two ways.

viate the tree diagrams slightly: BINOP(PLUS, x, y) nodes will be written as $+(x, y)$, and the actual values of CONST and TEMP nodes will not always be shown.

The fundamental idea of instruction selection using a tree-based intermediate representation is *tiling* the IR tree. The *tiles* are the set of tree patterns corresponding to legal machine instructions, and the goal is to cover the tree with nonoverlapping tiles.

For example, the Tiger-language expression such as $a[i] := x$, where i is a register variable and a and x are frame-resident, results in a tree that can be tiled in many different ways. Two tilings, and the corresponding instruction sequences, are shown in Figure 9.2 (remember that a is really the frame offset of the pointer to an array). In each case, tiles 1, 3, and 7 do not correspond to any machine instructions, because they are just registers (TEMPs) already containing the right values.

Finally – assuming a "reasonable" set of tile-patterns – it is always possible to tile the tree with tiny tiles, each covering only one node. In our example, such a tiling looks like this:

ADDI	$r_1 \leftarrow r_0 + a$
ADD	$r_1 \leftarrow \mathrm{fp} + r_1$
LOAD	$r_1 \leftarrow M[r_1 + 0]$
ADDI	$r_2 \leftarrow r_0 + 4$
MUL	$r_2 \leftarrow r_i \times r_2$
ADD	$r_1 \leftarrow r_1 + r_2$
ADDI	$r_2 \leftarrow r_0 + x$
ADD	$r_2 \leftarrow \mathrm{fp} + r_2$
LOAD	$r_2 \leftarrow M[r_2 + 0]$
STORE	$M[r_1 + 0] \leftarrow r_2$

For a reasonable set of patterns, it is sufficient that each individual `Tree` node correspond to some tile. It is usually possible to arrange for this; for example, the LOAD instruction can be made to cover just a single MEM node by using a constant of 0, and so on.

OPTIMAL AND OPTIMUM TILINGS

The best tiling of a tree corresponds to an instruction sequence of least cost: the shortest sequence of instructions. Or if the instructions take different amounts of time to execute, the least-cost sequence has the lowest total time.

Suppose we could give each kind of instruction a cost. Then we could define an *optimum* tiling as the one whose tiles sum to the lowest possible value. An *optimal* tiling is one where no two adjacent tiles can be combined into a single tile of lower cost. If there is some tree pattern that can be split into several tiles of lower combined cost, then we should remove that pattern from our catalog of tiles before we begin.

Every *optimum* tiling is also *optimal*, but not vice versa. For example, suppose every instruction costs one unit, except for MOVEM which costs m units. Then either Figure 9.2a is optimum (if $m > 1$) or Figure 9.2b is optimum (if $m < 1$) or both (if $m = 1$); but both trees are optimal.

Optimum tiling is based on an idealized cost model. In reality, instructions are not self-contained with individually attributable costs; nearby instructions interact in many ways, as discussed in Chapter 20.

9.1 ALGORITHMS FOR INSTRUCTION SELECTION

There are good algorithms for finding optimum and optimal tilings, but the algorithms for optimal tilings are simpler, as you might expect.

Complex Instruction Set Computers (CISC) have instructions that accomplish several operations each. The tiles for these instructions are quite large, and the difference between optimum and optimal tilings – while never very large – is at least sometimes noticeable.

Most architectures of modern design are *Reduced Instruction Set Computers (RISC)*. Each RISC instruction accomplishes just a small number of operations (all the *Jouette* instructions except MOVEM are typical RISC instructions). Since the tiles are small and of uniform cost, there is usually no difference at all between optimum and optimal tilings. Thus, the simpler tiling algorithms suffice.

MAXIMAL MUNCH

The algorithm for optimal tiling is called *Maximal Munch*. It is quite simple. Starting at the root of the tree, find the largest tile that fits. Cover the root node – and perhaps several other nodes near the root – with this tile, leaving several subtrees. Now repeat the same algorithm for each subtree.

As each tile is placed, the instruction corresponding to that tile is generated. The Maximal Munch algorithm generates the instructions *in reverse order* – after all, the instruction at the root is the first to be generated, but it can only execute after the other instructions have produced operand values in registers.

The "largest tile" is the one with the most nodes. For example, the tile for ADD has one node, the tile for SUBI has two nodes, and the tiles for STORE and MOVEM have three nodes each.

If two tiles of equal size match at the root, then the choice between them is arbitrary. Thus, in the tree of Figure 9.2, STORE and MOVEM both match, and either can be chosen.

Maximal Munch is quite straightforward to implement in C. Simply write two recursive functions, `munchStm` for statements and `munchExp` for expressions. Each clause of `munchExp` will match one tile. The clauses are ordered in order of tile preference (biggest tiles first).

Programs 9.3 and 9.4 sketch a partial example of a *Jouette* code-generator based on the Maximal Munch algorithm. Executing this program on the tree of Figure 9.2 will match the first clause of `munchStm`; this will call `munchExp` to emit all the instructions for the operands of the STORE, followed by the STORE itself. Program 9.3 does not show how the registers are chosen and operand syntax is specified for the instructions; we are concerned here only with the pattern-matching of tiles.

```
static void munchStm(T_stm s) {
  switch(s->kind) {
   case T_MOVE: {
     T_exp dst = s->u.MOVE.dst, src = s->u.MOVE.src;
     if (dst->kind==T_MEM) {
        if (dst->u.MEM->kind==T_BINOP
              && dst->u.MEM->u.BINOP.op==T_plus
              && dst->u.MEM->u.BINOP.right->kind==T_CONST) {
              T_exp e1 = dst->u.MEM->u.BINOP.left, e2=src;
              /* MOVE(MEM(BINOP(PLUS, e1, CONST(i))), e2)   */
              munchExp(e1); munchExp(e2); emit("STORE");
            }
        else if (dst->u.MEM->kind==T_BINOP
              && dst->u.MEM->u.BINOP.op==T_plus
              && dst->u.MEM->u.BINOP.left->kind==T_CONST) {
              T_exp e1 = dst->u.MEM->u.BINOP.right, e2=src;
              /* MOVE(MEM(BINOP(PLUS, CONST(i), e1)), e2)   */
              munchExp(e1); munchExp(e2); emit("STORE");
            }
        else if (src->kind==T_MEM)  {
              T_exp e1 = dst->u.MEM, e2=src->u.MEM;
              /* MOVE(MEM(e1), MEM(e2))   */
              munchExp(e1); munchExp(e2); emit("MOVEM");
            }
        else  {
              T_exp e1 = dst->u.MEM, e2=src;
              /* MOVE(MEM(e1), e2)   */
              munchExp(e1); munchExp(e2); emit("STORE");
        }   }
     else if (dst->kind==T_TEMP) {
              T_exp e2=src;
              /* MOVE(TEMP i, e2)   */
              munchExp(e2); emit("ADD");
            }
     else assert(0);  /* destination of MOVE must be MEM or TEMP */
     break;

   case T_JUMP: ···
   case T_CUMP: ···
   case T_NAME: ···
      ⋮
```

PROGRAM 9.3. Maximal Munch in C.

If, for each node-type in the `Tree` language, there exists a single-node tile pattern, then Maximal Munch cannot get “stuck” with no tile to match some subtree.

```
static void munchStm(T_stm s)
  MEM(BINOP(PLUS, e1, CONST(i)))  ⇒ munchExp(e1); emit("LOAD");
  MEM(BINOP(PLUS, CONST(i), e1))  ⇒ munchExp(e1); emit("LOAD");
  MEM(CONST(i))  ⇒ emit("LOAD");
  MEM(e1)  ⇒ munchExp(e1); emit("LOAD");
  BINOP(PLUS, e1, CONST(i))  ⇒ munchExp(e1); emit("ADDI");
  BINOP(PLUS, CONST(i), e1)  ⇒ munchExp(e1); emit("ADDI");
  CONST(i)  ⇒ munchExp(e1); emit("ADDI");
  BINOP(PLUS, e1, CONST(i))  ⇒ munchExp(e1); emit("ADD");
  TEMP(t)  ⇒ {}
```

PROGRAM 9.4. Summary of the C function `munchStm`.

DYNAMIC PROGRAMMING

Maximal Munch always finds an optimal tiling, but not necessarily an optimum. A dynamic-programming algorithm can find the optimum. In general, dynamic programming is a technique for finding optimum solutions for a whole problem based on the optimum solution of each subproblem; here the subproblems are the tilings of the subtrees.

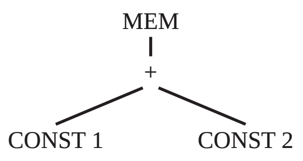
The dynamic-programming algorithm assigns a *cost* to every node in the tree. The cost is the sum of the instruction-costs of the best instruction sequence that can tile the subtree rooted at that node.

This algorithm works bottom-up, in contrast to Maximal Munch, which works top-down. First, the costs of all the children (and grandchildren, etc.) of node n are found recursively. Then, each tree-pattern (tile kind) is matched against node n.

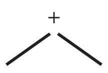
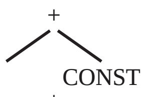
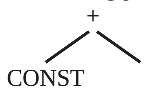
Each tile has zero or more *leaves*. In Figure 9.1 the leaves are represented as edges whose bottom ends exit the tile. The leaves of a tile are places where subtrees can be attached.

For each tile t of cost c that matches at node n, there will be zero or more subtrees s_i corresponding to the leaves of the tile. The cost c_i of each subtree has already been computed (because the algorithm works bottom-up). So the cost of matching tile t is just $c + \sum c_i$.

Of all the tiles t_j that match at node n, the one with the minimum-cost match is chosen, and the (minimum) cost of node n is thus computed. For example, consider this tree:

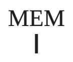
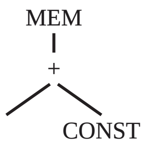
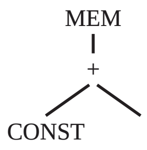


The only tile that matches CONST 1 is an ADDI instruction with cost 1. Similarly, CONST 2 has cost 1. Several tiles match the + node:

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
	ADD	1	1+1	3
	ADDI	1	1	2
	ADDI	1	1	2

The ADD tile has two leaves, but the ADDI tile has only one leaf. In matching the first ADDI pattern, we are saying "though we computed the cost of tiling CONST 2, we are not going to use that information." If we choose to use the first ADDI pattern, then CONST 2 will not be the root of any tile, and its cost will be ignored. In this case, either of the two ADDI tiles leads to the minimum cost for the + node, and the choice is arbitrary. The + node gets a cost of 2.

Now, several tiles match the MEM node:

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
	LOAD	1	2	3
	LOAD	1	1	2
	LOAD	1	1	2

Either of the last two matches will be optimum.

Once the cost of the root node (and thus the entire tree) is found, the *instruction emission* phase begins. The algorithm is as follows:

> Emission(node n): for each leaf l_i of the tile selected at node n, perform Emission(l_i). Then emit the instruction matched at node n.

Emission(n) does *not* recur on the children of node n, but on the *leaves of the tile* that matched at n. For example, after the dynamic programming algorithm finds the optimum cost of the simple tree above, the Emission phase emits

$$\begin{array}{l}\text{ADDI}\ \ r_1 \leftarrow r_0 + 1\\ \text{LOAD}\ r_1 \leftarrow M[r_1 + 2]\end{array}$$

but no instruction is emitted for any tile rooted at the + node, because this was not a leaf of the tile matched at the root.

TREE GRAMMARS

For machines with complex instruction sets and several classes of registers and addressing modes, there is a useful generalization of the dynamic programming algorithm. Suppose we make a brain-damaged version of *Jouette* with two classes of registers: *a* registers for addressing, and *d* registers for "data." The instruction set of the *Schizo-Jouette* machine (loosely based on the Motorola 68000) is shown in Figure 9.5.

The root and leaves of each tile must be marked with *a* or *d* to indicate which kind of register is implied. Now, the dynamic programming algorithm must keep track, for each node, of the min-cost match as an *a* register, *and also* the min-cost match as a *d* register.

At this point it is useful to use a context-free grammar to describe the tiles; the grammar will have nonterminals *s* (for statements), *a* (for expressions calculated into an *a* register), and *d* (for expressions calculated into a *d* register). Section 3.1 describes the use of context-free grammars for source-language syntax; here we use them for quite a different purpose.

The grammar rules for the LOAD, MOVEA, and MOVED instructions might look like this:

$$\begin{array}{l}d \rightarrow \text{MEM}(+(a, \text{CONST}))\\ d \rightarrow \text{MEM}(+(\text{CONST}, a))\\ d \rightarrow \text{MEM}(\text{CONST})\\ d \rightarrow \text{MEM}(a)\\ d \rightarrow a\\ a \rightarrow d\end{array}$$

Such a grammar is highly ambiguous: there are many different parses of the same tree (since there are many different instruction sequences implementing the same expression). For this reason, the parsing techniques described in Chapter 3 are not very useful in this application. However, a generalization of the dynamic programming algorithm works quite well: the

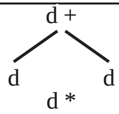
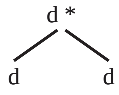
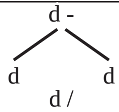
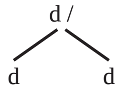
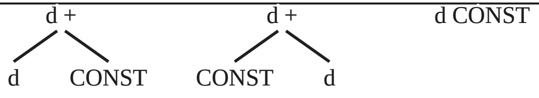
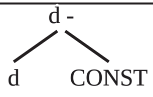
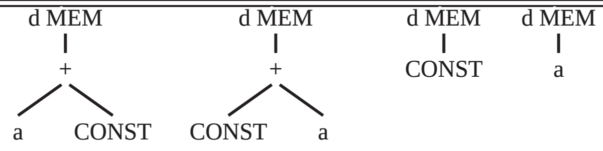
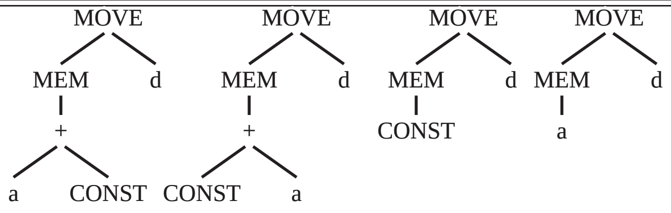
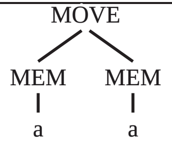
Name	*Effect*	*Trees*
	r_i	TEMP
ADD	$d_i \leftarrow d_j + d_k$	
MUL	$d_i \leftarrow d_j \times d_k$	
SUB	$d_i \leftarrow d_j - d_k$	
DIV	$d_i \leftarrow d_j / d_k$	
ADDI	$d_i \leftarrow d_j + c$	
SUBI	$d_i \leftarrow d_j - c$	
MOVEA	$d_j \leftarrow a_i$	d a
MOVED	$a_j \leftarrow d_i$	a d
LOAD	$d_i \leftarrow M[a_j + c]$	
STORE	$M[a_j + c] \leftarrow d_i$	 MOVE MEM d + a CONST MOVE MEM d + CONST a MOVE MEM d CONST MOVE MEM d a
MOVEM	$M[a_j] \leftarrow M[a_i]$	

FIGURE 9.5. The *Schizo-Jouette* architecture.

minimum-cost match at each node *for each nonterminal of the grammar* is computed.

Though the dynamic programming algorithm is conceptually simple, it becomes messy to write down directly in a general-purpose programming language such as C. Thus, several tools have been developed. These *code-generator generators* process grammars that specify machine instruction sets; for each rule of the grammar, a cost and an action are specified. The costs are used to find the optimum tiling, and then the actions of the matched rules are used in the *emission* phase.

Like Yacc and Lex, the output of a code-generator generator is usually a program in C that operates a table-driven matching engine with the action fragments (written in C) inserted at the appropriate points.

Such tools are quite convenient. Grammars can specify addressing modes of treelike CISC instructions quite well. A typical grammar for the VAX has 112 rules and 20 nonterminal symbols; and one for the Motorola 68020 has 141 rules and 35 nonterminal symbols. However, instructions that produce more than one result – such as autoincrement instructions on the VAX – are difficult to express using tree patterns.

Code-generator generators are probably overkill for RISC machines. The tiles are quite small, there aren't very many of them, and there is little need for a grammar with many nonterminal symbols.

FAST MATCHING

Maximal Munch and the dynamic programming algorithm must examine, for each node, all the tiles that match at that node. A tile matches if each nonleaf node of the tile is labeled with the same operator (MEM, CONST, etc.) as the corresponding node of the tree.

The naive algorithm for matching would be to examine each tile in turn, checking each node of the tile against the corresponding part of the tree. However, there are better approaches. To match a tile at node *n* of the tree, the label at *n* can be used in a `case` statement:

```
match(n) {
  switch (label(n)) {
    case MEM: · · ·
    case BINOP: · · ·
    case CONST: · · ·
}
```

Once the clause for one label (such as MEM) is selected, only those patterns rooted in that label remain in consideration. Another `case` statement can use the label of the child of n to begin distinguishing among those patterns.

The organization and optimization of decision trees for pattern matching is beyond the scope of this book. However, for better performance the naive sequence of clauses in function `munchExp` should be rewritten as a sequence of comparisons that never looks twice at the same tree node.

EFFICIENCY OF TILING ALGORITHMS

How expensive are Maximal Munch and dynamic programming?

Let us suppose that there are T different tiles, and that the average matching tile contains K nonleaf (labeled) nodes. Let K' be the largest number of nodes that ever need to be examined to see which tiles match at a given subtree; this is approximately the same as the size of the largest tile. And suppose that, on the average, T' different patterns (tiles) match at each tree node. For a typical RISC machine we might expect $T = 50$, $K = 2$, $K' = 4$, $T' = 5$.

Suppose there are N nodes in the input tree. Then Maximal Munch will have to consider matches at only N/K nodes because, once a "munch" is made at the root, no pattern-matching needs to take place at the nonleaf nodes of the tile.

To find all the tiles that match at one node, at most K' tree nodes must be examined; but (with a sophisticated decision tree) each of these nodes will be examined only once. Then each of the successful matches must be compared to see if its cost is minimal. Thus, the matching at each node costs $K' + T'$, for a total cost proportional to $(K' + T')N/K$.

The dynamic programming algorithm must find all the matches at *every* node, so its cost is proportional to $(K' + T')N$. However, the constant of proportionality is higher than that of Maximal Munch, since dynamic programming requires two tree-walks instead of one.

Since K, K', and T' are constant, the running time of all of these algorithms is linear. In practice, measurements show that these instruction selection algorithms run very quickly compared to the other work performed by a real compiler – even lexical analysis is likely to take more time than instruction selection.

9.2 CISC MACHINES

A typical modern RISC machine has

1. 32 registers,
2. only one class of integer/pointer registers,
3. arithmetic operations only between registers,
4. "three-address" instructions of the form $r_1 \leftarrow r_2 \oplus r_3$,
5. load and store instructions with only the M[reg+const] addressing mode,
6. every instruction exactly 32 bits long,
7. one result or effect per instruction.

Many machines designed between 1970 and 1985 are *Complex Instruction Set Computers* (CISC). Such computers have more complicated addressing modes that encode instructions in fewer bits, which was important when computer memories were smaller and more expensive. Typical features found on CISC machines include

1. few registers (16, or 8, or 6),
2. registers divided into different classes, with some operations available only on certain registers,
3. arithmetic operations can access registers or memory through "addressing modes,"
4. "two-address" instructions of the form $r_1 \leftarrow r_1 \oplus r_2$,
5. several different addressing modes,
6. variable-length instructions, formed from variable-length opcode plus variable-length addressing modes,
7. instructions with side effects such as "auto-increment" addressing modes.

Most computer architectures designed since 1990 are RISC machines, but most general-purpose computers installed since 1990 are CISC machines: the Intel 80386 and its descendants (486, Pentium).

The Pentium, in 32-bit mode, has six general-purpose registers, a stack pointer, and a frame pointer. Most instructions can operate on all six registers, but the multiply and divide instructions operate only on the `eax` register. In contrast to the "3-address" instructions found on RISC machines, Pentium arithmetic instructions are generally "2-address," meaning that the destination register must be the same as the first source register. Most instructions can have either two register operands ($r_1 \leftarrow r_1 \oplus r_2$), or one register and one memory operand, for example $M[r_1 + c] \leftarrow M[r_1 + c] \oplus r_2$ or $r_1 \leftarrow r_1 \oplus M[r_2 + c]$, but not $M[r_1 + c_1] \leftarrow M[r_1 + c_1] \oplus M[r_2 + c_2]$

We will cut through these Gordian knots as follows:

1. Few registers: we continue to generate TEMP nodes freely, and assume that the register allocator will do a good job.

2. Classes of registers: The multiply instruction on the Pentium requires that its left operand (and therefore destination) must be the `eax` register. The high-order bits of the result (useless to a Tiger program) are put into register `edx`. The solution is to move the operands and result explicitly; to implement $t_1 \leftarrow t_2 \times t_3$:

   ```
   mov eax, t2      eax ← t2
   mul  t3          eax ← eax × t3;    edx ← garbage
   mov t1, eax      t1  ← eax
   ```

   This looks very clumsy; but one job that the register allocator performs is to eliminate as many move instructions as possible. If the allocator can assign t_1 or t_3 (or both) to register `eax`, then it can delete one or both of the move instructions.
3. Two-address instructions: We solve this problem in the same way as we solve the previous one: by adding extra move instructions. To implement $t_1 \leftarrow t_2 + t_3$ we produce

   ```
   mov t1, t2      t1 ← t2
   add  t1, t3     t1 ← t1 + t3
   ```

   Then we hope that the register allocator will be able to allocate t_1 and t_2 to the same register, so that the move instruction will be deleted.
4. Arithmetic operations can address memory: The instruction selection phase turns every TEMP node into a "register" reference. Many of these "registers" will actually turn out to be memory locations. The *spill* phase of the register allocator must be made to handle this case efficiently; see Chapter 11.

   The alternative to using memory-mode operands is simply to fetch all the operands into registers before operating and store them back to memory afterwards. For example, these two sequences compute the same thing:

   ```
   mov eax, [ebp – 8]
   add  eax, ecx                    add [ebp – 8], ecx
   mov [ebp – 8], eax
   ```

   The sequence on the right is more concise (and takes less machine-code space), but *the two sequences are equally fast*. The load, register-register add, and store take 1 cycle each, and the memory-register add takes 3 cycles. On a highly pipelined machine such as the Pentium Pro, simple cycle counts are not the whole story, but the result will be the same: the processor has to perform the load, add, and store, no matter how the instructions specify them.

   The sequence on the left has one significant disadvantage: it trashes the value in register `eax`. Therefore, we should try to use the sequence on the right when possible. But the issue here turns into one of register allocation, not of instruction speed; so we defer its solution to the register allocator.

5. Several addressing modes: An addressing mode that accomplishes six things typically takes six steps to execute. Thus, these instructions are often no faster than the multi-instruction sequences they replace. They have only two advantages: they "trash" fewer registers (such as the register `eax` in the previous example), and they have a shorter instruction encoding. With some work, tree-matching instruction selection can be made to select CISC addressing modes, but programs can be just as fast using the simple RISC-like instructions.
6. Variable-length instructions: This is not really a problem for the compiler; once the instructions are selected, it is a trivial (though tedious) matter for the assembler to emit the encodings.
7. Instructions with side effects: Some machines have an "autoincrement" memory fetch instruction whose effect is

   $$r_2 \leftarrow M[r_1]; \quad r_1 \leftarrow r_1 + 4$$

   This instruction is difficult to model using tree patterns, since it produces two results. There are three solutions to this problem:
   (a) Ignore the autoincrement instructions, and hope they go away. This is an increasingly successful solution, as few modern machines have multiple-side-effect instructions.
   (b) Try to match special idioms in an ad hoc way, within the context of a tree pattern-matching code generator.
   (c) Use a different instruction algorithm entirely, one based on DAG-patterns instead of tree-patterns.

Several of these solutions depend critically on the register allocator to eliminate move instructions and to be smart about spilling; see Chapter 11.

9.3 INSTRUCTION SELECTION FOR THE Tiger COMPILER

Pattern-matching of "tiles" is simple (if tedious) in C, as shown in Program 9.3. But this figure does not show what to do with each pattern match. It is all very well to print the name of the instruction, but which registers should these instructions use?

In a tree tiled by instruction patterns, the root of each tile will correspond to some intermediate result held in a register. Register allocation is the act of assigning register-numbers to each such node.

The instruction selection phase can simultaneously do register allocation. However, many aspects of register allocation are independent of the particular target-machine instruction set, and it is a shame to duplicate the register-allocation algorithm for each target machine. Thus, register allocation should come either before or after instruction selection.

Before instruction selection, it is not even known which tree nodes will need registers to hold their results, since only the roots of tiles (and not other labeled nodes within tiles) require explicit registers. Thus, register allocation before instruction selection cannot be very accurate. But some compilers do it anyway, to avoid the need to describe machine instructions without the real registers filled in.

We will do register allocation after instruction selection. The instruction selection phase will generate instructions without quite knowing which registers the instructions use.

ABSTRACT ASSEMBLY-LANGUAGE INSTRUCTIONS

We will invent a data type for "assembly language instruction without register assignments," called `AS_instr`:

```
 /* assem.h */
typedef struct {Temp_labelList labels;} *AS_targets;
AS_targets AS_Targets(Temp_labelList labels);

typedef struct {
    enum {I_OPER, I_LABEL, I_MOVE} kind;
    union {struct {string assem; Temp_tempList dst, src;
                   AS_targets jumps;} OPER;
            struct {string assem; Temp_label label;} LABEL;
            struct {string assem;
                    Temp_tempList dst, src;} MOVE;
                  } u;
} *AS_instr;

AS_instr AS_Oper(string a, Temp_tempList d,
                 Temp_tempList s, AS_targets j);
AS_instr AS_Label(string a, Temp_label label);
AS_instr AS_Move(string a, Temp_tempList d, Temp_tempList s);

void AS_print(FILE *out, AS_instr i, Temp_map m);

  ⋮
```

An OPER holds an assembly-language instruction `assem`, a list of operand registers `src`, and a list of result registers `dst`. Either of these lists may be empty. Operations that always fall through to the next instruction have `jump=NULL`; other operations have a list of "target" labels to which they may jump (this list must explicitly include the next instruction if it is possible to fall through to it).

A LABEL is a point in a program to which jumps may go. It has an `assem` component showing how the label will look in the assembly-language program, and a `label` component identifying which label-symbol was used.

A MOVE is like an OPER, but must perform only data transfer. Then, if the `dst` and `src` temporaries are assigned to the same register, the MOVE can later be deleted.

Calling `AS_print`(f, i, m) formats an assembly instruction as a string and prints it to the file f); m is a *temp mapping* that tells the register assignment (or perhaps just the name) of every temp.

The `temp.h` interface describes functions for operating on temp mappings:

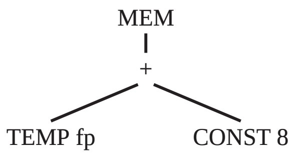
```
/* temp.h */
   ⋮
typedef struct Temp_map_ *Temp_map;
Temp_map Temp_empty(void);  /* create a new, empty map */
Temp_map Temp_layerMap(Temp_map over, Temp_map under);
void Temp_enter(Temp_map m, Temp_temp t, string s);
string Temp_look(Temp_map m, Temp_temp t);

Temp_map Temp_name(void);
```

A `Temp_map` is just a table whose keys are `Temp_temp`s and whose bindings are strings. However, one mapping can be layered over another; if $\sigma_3 =$ `layer`$(\sigma_1, \sigma 2)$, this means that `look`(σ_3, t) will first try `look`(σ_1, t), and if that fails it will try `look`(σ_2, t). Also, `enter`(σ_3, t, a) will have the effect of entering $t \mapsto a$ into σ_2.

The primary users of these `Temp_map` operations will be the register allocator, which will decide which register name to use for each temporary. But the `Frame` module makes a `Temp_map` to describe the names of all the preallocated registers (such as the frame-pointer, stack-pointer, etc.); and for debugging purposes it's useful to have a special `Temp_name` mapping that just maps each temporary (such as t_{182}) to its "name" (such as `"t182"`).

Machine-independence. The `AS_instr` type is *independent* of the chosen target-machine assembly language (though it is tuned for machines with only one class of register). If the target machine is a Sparc, then the `assem` strings will be Sparc assembly language. I will use *Jouette* assembly language for illustration.

For example, the tree



could be translated into *Jouette* assembly language as

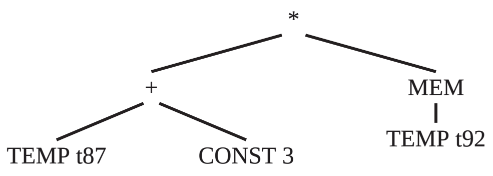
```
AS_Oper("LOAD ‘d0 <- M[‘s0+8]",
        Temp_TempList(Temp_newtemp(), NULL),
        Temp_TempList(T_Temp(F_FP()), NULL),
        NULL)
```

This instruction needs some explanation. The actual assembly language of *Jouette*, after register allocation, might be

```
LOAD r1 <- M[r27+8]
```

assuming that register r_{27} is the frame pointer `fp` and that the register allocator decided to assign the new temp to register r_1. But the `Assem` instruction does not know about register assignments; instead, it just talks of the sources and destination of each instruction. This LOAD instruction has one source register, which is referred to as `‘s0`; and one destination register, referred to as `‘d0`.

Another example will be useful. The tree



could be translated as

assem	*dst*	*src*
`ADDI ‘d0 <- ‘s0+3`	t908	t87
`LOAD ‘d0 <- M[‘s0+0]`	t909	t92
`MUL ‘d0 <- ‘s0*‘s1`	t910	t908,t909

where t908, t909, and t910 are temporaries newly chosen by the instruction selector.

After register allocation the assembly language might look like:

```
ADDI  r1 <- r12+3
LOAD  r2 <- M[r13+0]
MUL   r1 <- r1 * r2
```

The `string` of an `instr` may refer to *source registers* `‘s0`, `‘s1`, ... `‘s`$(k-1)$, and *destination* registers `‘d0`, `‘d1`, etc. Jumps are OPER instructions that refer to labels `‘j0`, `‘j1`, etc. Conditional jumps, which may branch away or fall through, typically have two labels in the `jump` list but refer to only one of them in the `assem` string.

Two-address instructions. Some machines have arithmetic instructions with two operands, where one of the operands is both a source and a destination. The instruction `add t1,t2`, which has the effect of $t_1 \leftarrow t_1 + t_2$, can be described as

assem	*dst*	*src*
`add ‘d0,‘s1`	t1	t1, t2

where `‘s0` is implicitly, but not explicitly, mentioned in the `assem` string.

PRODUCING ASSEMBLY INSTRUCTIONS

Now it is a simple matter to write the right-hand sides of the pattern-matching clauses that "munch" `Tree` expressions into `Assem` instructions. I will show some examples from the *Jouette* code generator, but the same ideas apply to code generators for real machines.

The functions `munchStm` and `munchExp` will produce `Assem` instructions, bottom-up, as side effects. `MunchExp` returns the temporary in which the result is held.

```
static Temp_temp munchExp(T_exp e);
static void munchStm(T_stm s);
```

The "actions" of the `munchExp` clauses of Program 9.3 can be written as shown in Programs 9.7 and 9.6.

The `emit` function just accumulates a list of instructions to be returned later, as shown in Program 9.8. The `assem.h` interface contains data structures and functions for lists of instructions, `AS_instrList`:

```
static Temp_temp munchExp(T_exp e) {
  switch( e )
   case MEM(BINOP(PLUS,e1,CONST(i))):  {
     Temp_temp r = Temp_newtemp();
     emit(AS_Oper("LOAD ‘d0 <- M[‘s0+" + i + "]\n",
                 L(r,NULL), L(munchExp(e1),NULL), NULL));
     return r; }
   case MEM(BINOP(PLUS,CONST(i),e1)): {
     Temp_temp r = Temp_newtemp();
     emit(AS_Oper("LOAD ‘d0 <- M[‘s0+" + i + "]\n",
                 L(r,NULL), L(munchExp(e1),NULL), NULL));
     return r; }
   case MEM(CONST(i)): {
     Temp_temp r = Temp_newtemp();
     emit(AS_Oper("LOAD ‘d0 <- M[r0+" + i + "]\n",
          L(r,NULL), NULL, NULL));
     return r; }
   case MEM(e1): {
     Temp_temp r = Temp_newtemp();
     emit(AS_Oper("LOAD ‘d0 <- M[‘s0+0]\n",
                 L(r,NULL), L(munchExp(e1),NULL), NULL));
    return r; }
   case BINOP(PLUS,e1,CONST(i)): {
     Temp_temp r = Temp_newtemp();
     emit(AS_Oper("ADDI ‘d0 <- ‘s0+" + i + "\n",
                 L(r,NULL), L(munchExp(e1),NULL), NULL));
     return r; }
   case BINOP(PLUS,CONST(i),e1): {
     Temp_temp r = Temp_newtemp();
     emit(AS_Oper("ADDI ‘d0 <- ‘s0+" + i + "\n",
                  L(r,NULL), L(munchExp(e1),NULL), NULL));
     return r; }
   case CONST(i): {
     Temp_temp r = Temp_newtemp();
     emit(AS_Oper("ADDI ‘d0 <- r0+" + i + "\n",
                 NULL, L(munchExp(e1),NULL), NULL));
     return r; }
   case BINOP(PLUS,e1,e2): {
     Temp_temp r = Temp_newtemp();
     emit(AS_Oper("ADD  ‘d0 <- ‘s0+‘s1\n",
                 L(r,NULL), L(munchExp(e1),L(munchExp(e2),NULL)), NULL));
     return r; }
   case TEMP(t):
     return t;
     ⋮
```

PROGRAM 9.6. Assem-instructions for munchExp.

```
Temp_tempList L(Temp_temp h, Temp_tempList t) {return Temp_TempList(h,t);}

static void munchStm(T_stm s) {
  switch ( s )
    case MOVE(MEM(BINOP(PLUS,e1,CONST(i))),e2):
      emit(AS_Oper("STORE M[‘s0+" + i + "] <- ‘s1\n",
                 NULL, L(munchExp(e1), L(munchExp(e2), NULL)), NULL));
    case MOVE(MEM(BINOP(PLUS,CONST(i),e1)),e2):
      emit(AS_Oper("STORE M[‘s0+" + i + "] <- ‘s1\n",
                 NULL, L(munchExp(e1), L(munchExp(e2), NULL)), NULL));
    case MOVE(MEM(e1),MEM(e2)):
      emit(AS_Oper("MOVE  M[‘s0] <- M[‘s1]\n",
                 NULL, L(munchExp(e1), L(munchExp(e2), NULL)), NULL));
    case MOVE(MEM(CONST(i)),e2):
      emit(AS_Oper("STORE M[r0+" + i + "] <- ‘s0\n",
                 NULL, L(munchExp(e2), NULL), NULL));
    case MOVE(MEM(e1),e2):
      emit(AS_Oper("STORE M[‘s0] <- ‘s1\n",
                 NULL, L(munchExp(e1), L(munchExp(e2), NULL)), NULL));
    case MOVE(TEMP(i), e2):
      emit(AS_Move("ADD   ‘d0 <- ‘s0 + r0\n",
                 i, munchExp(e2)));
    case LABEL(lab):
      emit(AS_Label(Temp_labelstring(lab) + ":\n", lab));
     ⋮
}
```

PROGRAM 9.7. Assem-instructions for munchStm.

```
/* more of assem.h */
   ⋮
typedef struct AS_instrList_ *AS_instrList;
struct AS_instrList_ { AS_instr head; AS_instrList tail;};
AS_instrList AS_InstrList(AS_instr head, AS_instrList tail);


AS_instrList AS_splice(AS_instrList a, AS_instrList b);
void AS_printInstrList (FILE *out, AS_instrList iList,
                        Temp_map m);


typedef struct {
  string prolog; AS_instrList body; string epilog;
} *AS_proc;
```

```
/* codegen.c */
   ⋮
static AS_instrList iList=NULL, last=NULL;
static void emit(AS_instr inst) {
  if (last!=NULL)
       last = last->tail = AS_InstrList(inst,NULL);
  else last = iList = AS_InstrList(inst,NULL);
}

AS_instrList F_codegen(F_frame f, T_stmList stmList)  {
  AS_instrList list;  T_stmList sl;
    ⋮   /* miscellaneous initializations as necessary */
  for (sl=stmList; sl; sl=sl->tail) munchStm(sl->head);
  list=iList; iList=last=NULL; return list;
}
```

PROGRAM 9.8. The `codegen` function.

PROCEDURE CALLS

Procedure calls are represented by EXP(CALL(f, *args*)), and function calls by MOVE(TEMP t, CALL(f, *args*)). These trees can be matched by tiles such as

```
case EXP(CALL(e,args)): {
   Temp_temp r = munchExp(e);
   Temp_tempList l = munchArgs(0,args);
   emit(AS_Oper("CALL ‘s0\n", calldefs, L(r,l), NULL));}
```

In this example, `munchArgs` generates code to move all the arguments to their correct positions, in outgoing parameter registers and/or in memory. The integer parameter to `munchArgs` is i for the ith argument; `munchArgs` will recur with $i + 1$ for the next argument, and so on.

What `munchArgs` returns is a list of all the temporaries that are to be passed to the machine's `CALL` instruction. Even though these temps are never written explicitly in assembly language, they should be listed as "sources" of the instruction, so that liveness analysis (Chapter 10) can see that their values need to be kept up to the point of call.

A `CALL` is expected to "trash" certain registers – the caller-save registers, the return-address register, and the return-value register. This list of `calldefs` should be listed as "destinations" of the `CALL`, so that the later phases of the compiler know that something happens to them here.

In general, any instruction that has the side effect of writing to another register requires this treatment. For example, the Pentium's multiply instruction writes to register `edx` with useless high-order result bits, so `edx` and `eax` are both listed as destinations of this instruction. (The high-order bits can be very useful for programs written in assembly language to do multiprecision arithmetic, but most programming languages do not support any way to access them.)

IF THERE'S NO FRAME POINTER

In a stack frame layout such as the one shown in Figure 6.2, the frame pointer points at one end of the frame and the stack pointer points at the other. At each procedure call, the stack pointer register is copied to the frame pointer register, and then the stack pointer is incremented by the size of the new frame.

Many machines' calling conventions do not use a frame pointer. Instead, the "virtual frame pointer" is always equal to stack pointer plus frame size. This saves time (no copy instruction) and space (one more register usable for other purposes). But our `Translate` phase has generated trees that refer to this fictitious frame pointer. The `codegen` function must replace any reference to FP+k with SP $+ k + fs$, where fs is the frame size. It can recognize these patterns as it munches the trees.

However, to replace them it must know the value of fs, which cannot yet be known because register allocation is not known. Assuming the function f is to be emitted at label `L14` (for example), `codegen` can just put `sp+L14_framesize` in its assembly instructions and hope that the prologue for f (generated by `F_procEntryExit3`) will include a definition of the assembly-language constant `L14_framesize`. `Codegen` is passed the `frame` argument (Program 9.8) so that it can learn the name `L14`.

Implementations that have a "real" frame pointer won't need this hack and can ignore the `frame` argument to `codegen`. But why would an implementation use a real frame pointer when it wastes time and space to do so? The answer is that this permits the frame size to grow and shrink even after it is first created; some languages have permitted dynamic allocation of arrays within the stack frame (e.g., using `alloca` in C). Calling-convention designers now tend to avoid dynamically adjustable frame sizes, however.

PROGRAM INSTRUCTION SELECTION

```
/* codegen.h */
AS_instrList F_codegen(F_frame f, T_stmList stmList);
```

Implement the translation to Assem-instructions for your favorite instruction set (let μ stand for *Sparc*, *Mips*, *Alpha*, *Pentium*, etc.) using Maximal Munch. If you would like to generate code for a RISC machine, but you have no RISC computer on which to test it, you may wish to use SPIM (a MIPS simulator implemented by James Larus), described on the Web page for this book.

First write the module μ`codegen.c` that implements the `codegen.h` interface using the "Maximal Munch" translation algorithm from IR trees to the `Assem` data structure.

Use the `Canon` module (described in Chapter 8) to simplify the trees before applying your `Codegen` module to them. Use the `AS_printInstrList` function to translate the resulting `Assem` trees to μ assembly language. Since you won't have done register assignment, just pass `Temp_name` to `AS_print` as the translation function from temporaries to strings.

This will produce "assembly" language that does not use register names at all: the instructions will use names such as `t3`, `t283`, and so on. But some of these temps are the "built-in" ones created by the `Frame` module to stand for particular machine registers (see page 159), such as `Frame.FP`. The assembly language will be easier to read if these registers appear with their natural names (e.g., `fp` instead of `t1`).

The `Frame` module must provide a mapping from the special temps to their names, and nonspecial temps to `NULL`:

```
/* frame.h */
    ⋮
Temp_map F_tempMap;
```

Then, for the purposes of displaying your assembly language prior to register allocation, use `Temp_layerMap` to make a new function that first tries `F_tempMap`, and if that returns `NULL`, resorts to `Temp_name`.

REGISTER LISTS

Make the following lists of registers; for each register, you will need a string for its assembly-language representation and a `Temp_temp` for referring to it in `Tree` and `Assem` data structures.

`specialregs` a list of μ registers used to implement "special" registers such as RV and FP and also the stack pointer SP, the return-address register RA, and (on some machines) the zero register ZERO. Some machines may have other special registers;

`argregs` a list of μ registers in which to pass outgoing arguments (including the static link);

`calleesaves` a list of μ registers that the called procedure (callee) must preserve unchanged (or save and restore);

`callersaves` a list of μ registers that the callee may trash.

The four lists of registers must not overlap, and must include any register that might show up in `Assem` instructions. These lists are not exported through the `frame.h` interface, but they are useful internally for both `Frame` and `Codegen` – for example, to implement `munchArgs` and to construct the `calldefs` list.

Implement the `F_procEntryExit2` function of the `frame.h` interface.

```
/* frame.h */
      ⋮
AS_instrList F_procEntryExit2(AS_instrList body);
```

This function appends a "sink" instruction to the function body to tell the register allocator that certain registers are live at procedure exit. In the case of the *Jouette* machine, this is simply:

```
static Temp_tempList returnSink = NULL;

AS_instrList F_procEntryExit2(AS_instrList body) {
  if (!returnSink) returnSink =
   Temp_TempList(ZERO, Temp_TempList(RA,
                        Temp_TempList(SP, calleeSaves)));
  return AS_splice(body, AS_InstrList(
             AS_Oper("", NULL, returnSink, NULL), NULL));
}
```

meaning that the temporaries *zero, return-address, stack-pointer*, and all the callee-saves registers are still live at the end of the function. Having *zero* live at the end means that it is live throughout, which will prevent the register allocator from trying to use it for some other purpose. The same trick works for any other special registers the machine might have.

Files available in `$TIGER/chap9` include:

`canon.c` Canonicalization and trace-generation.

`assem.c` The `Assem` module.
`main.c` A `Main` module that you may wish to adapt.

Your code generator will handle only the body of each procedure or function, but not the procedure entry/exit sequences. Use a “scaffold” version of `F_procEntryExit3` function:

```
AS_proc F_procEntryExit3(F_frame frame, AS_instrList body) {
  char buf[100];
  sprintf(buf,"PROCEDURE %s\n", S_name(frame->name));
  return AS_Proc(String(buf), body, "END\n");
}
```

FURTHER READING

Cattell [1980] expressed machine instructions as tree patterns, invented the Maximal Munch algorithm for instruction selection, and built a *code generator generator* to produce an instruction-selection function from a tree-pattern description of an instruction set. Glanville and Graham [1978] expressed the tree patterns as productions in LR(1) grammars, which allows the Maximal Munch algorithm to use multiple nonterminal symbols to represent different classes of registers and addressing modes. But grammars describing instruction sets are inherently ambiguous, leading to problems with the LR(1) approach; Aho et al. [1989] use dynamic programming to parse the tree grammars, which solves the ambiguity problem, and describe the *Twig* automatic code-generator generator. The dynamic programming can be done at compiler-construction time instead of code-generation time [Pelegri-Llopart and Graham 1988]; using this technique, the *BURG* tool [Fraser et al. 1992] has an interface similar to Twig’s but generates code much faster.

EXERCISES

9.1 For each of the following expressions, draw the tree and generate *Jouette*-machine instructions using Maximal Munch. Circle the tiles (as in Figure 9.2), but number them *in the order that they are munched*, and show the sequence of *Jouette* instructions that results.

a. MOVE(MEM(+(+(CONST$_{1000}$, MEM(TEMP$_x$)), TEMP$_{fp}$)), CONST$_0$)

b. BINOP(MUL, CONST$_5$, MEM(CONST$_{100}$))

***9.2** Consider a machine with the following instruction:

```
mult const1(src1), const2(src2), dst3
```

$r_3 \leftarrow M[r_1 + \text{const}_1] * M[r_2 + \text{const}_2]$

On this machine, r_0 is always 0, and $M[1]$ always contains 1.

a. Draw all the tree patterns corresponding to this instruction (and its special cases).

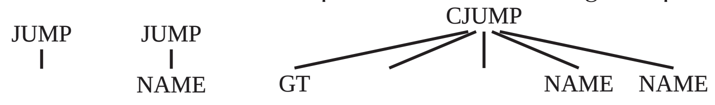
b. Pick **one** of the bigger patterns and show how to write a C if-statement to match it, with the `Tree` representation used for the Tiger compiler.

9.3 The *Jouette* machine has control-flow instructions as follows:

BRANCHGE	if $r_i \geq 0$ goto L
BRANCHLT	if $r_i < 0$ goto L
BRANCHEQ	if $r_i = 0$ goto L
BRANCHNE	if $r_i \neq 0$ goto L
JUMP	goto r_i

where the JUMP instruction goes to an address contained in a register.

Use these instructions to implement the following tree patterns:



Assume that a CJUMP is always followed by its false label. Show the best way to implement each pattern; in some cases you may need to use more than one instruction or make up a new temporary. How do you implement CJUMP(GT, . . .) without a BRANCHGT instruction?

10 Liveness Analysis

live: of continuing or current interest

Webster's Dictionary

The front end of the compiler translates programs into an intermediate language with an unbounded number of temporaries. This program must run on a machine with a bounded number of registers. Two temporaries a and b can fit into the same register, if a and b are never "in use" at the same time. Thus, many temporaries can fit in few registers; if they don't all fit, the excess temporaries can be kept in memory.

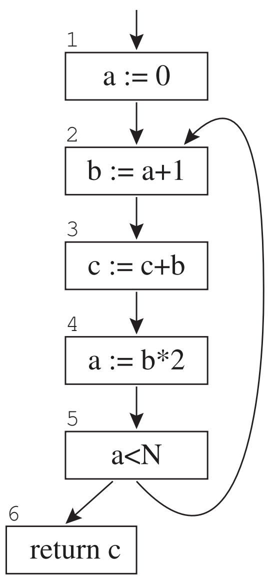
Therefore, the compiler needs to analyze the intermediate-representation program to determine which temporaries are in use at the same time. We say a variable is *live* if it holds a value that may be needed in the future, so this analysis is called *liveness* analysis.

To perform analyses on a program, it is often useful to make a *control-flow graph*. Each statement in the program is a node in the flow graph; if statement x can be followed by statement y, there is an edge from x to y. Graph 10.1 shows the flow graph for a simple loop.

Let us consider the liveness of each variable (Figure 10.2). A variable is live if its current value will be used in the future, so we analyze liveness by working from the future to the past. Variable b is used in statement 4, so b is live on the $3 \rightarrow 4$ edge. Since statement 3 does not assign into b, then b is also live on the $2 \rightarrow 3$ edge. Statement 2 assigns into b. That means that the contents of b on the $1 \rightarrow 2$ edge are not needed by anyone; b is dead on this edge. So the *live range* of b is $\{2 \rightarrow 3,\ 3 \rightarrow 4\}$.

The variable a is an interesting case. It's live from $1 \rightarrow 2$, and again from $4 \rightarrow 5 \rightarrow 2$, but not from $2 \rightarrow 3 \rightarrow 4$. Although a has a perfectly

$$\begin{aligned} & a \leftarrow 0 \\ L_1 : \; & b \leftarrow a + 1 \\ & c \leftarrow c + b \\ & a \leftarrow b * 2 \\ & \text{if } a < N \text{ goto } L_1 \\ & \text{return } c \end{aligned}$$



GRAPH 10.1. Control-flow graph of a program.

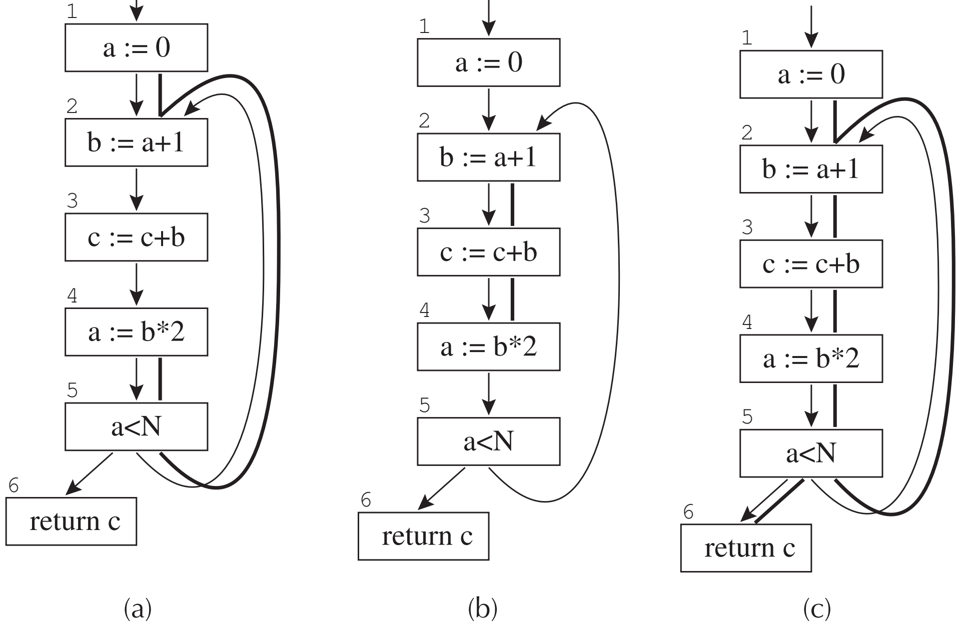


FIGURE 10.2. Liveness of variables a, b, c.

well defined value at node 3, that value will not be needed again before *a* is assigned a new value.

The variable *c* is live on entry to this program. Perhaps it is a formal parameter. If it is a local variable, then liveness analysis has detected an uninitialized variable; the compiler could print a warning message for the programmer.

Once all the live ranges are computed, we can see that only two registers are needed to hold *a*, *b*, and *c*, since *a* and *b* are never live at the same time. Register 1 can hold both *a* and *b*, and register 2 can hold *c*.

10.1 SOLUTION OF DATAFLOW EQUATIONS

Liveness of variables "flows" around the edges of the control-flow graph; determining the live range of each variable is an example of a *dataflow* problem. Chapter 17 will discuss several other kinds of dataflow problems.

Flow graph terminology. A flow-graph node has *out-edges* that lead to *successor* nodes, and *in-edges* that come from *predecessor* nodes. The set $pred[n]$ is all the predecessors of node n, and $succ[n]$ is the set of successors.

In Graph 10.1 the out-edges of node 5 are $5 \rightarrow 6$ and $5 \rightarrow 2$, and $succ[5] = \{2, 6\}$. The in-edges of 2 are $5 \rightarrow 2$ and $1 \rightarrow 2$, and $pred[2] = \{1, 5\}$.

Uses and defs. An assignment to a variable or temporary *defines* that variable. An occurrence of a variable on the right-hand side of an assignment (or in other expressions) *uses* the variable. We can speak of the *def* of a variable as the set of graph nodes that define it; or the *def* of a graph node as the set of variables that it defines; and similarly for the *use* of a variable or graph node. In Graph 10.1, $def(3) = \{c\}$, $use(3) = \{b, c\}$.

Liveness. A variable is *live* on an edge if there is a directed path from that edge to a *use* of the variable that does not go through any *def*. A variable is *live-in* at a node if it is live on any of the in-edges of that node; it is *live-out* at a node if it is live on any of the out-edges of the node.

CALCULATION OF LIVENESS

Liveness information (*live-in* and *live-out*) can be calculated from *use* and *def* as follows:

$$in[n] = use[n] \cup (out[n] - def[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

EQUATIONS 10.3. Dataflow equations for liveness analysis.

```
for each $n$
    $in[n] \leftarrow \{\};\ out[n] \leftarrow \{\}$
repeat
    for each $n$
        $in'[n] \leftarrow in[n];\ out'[n] \leftarrow out[n]$
        $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
        $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all $n$
```

ALGORITHM 10.4. Computation of liveness by iteration.

1. If a variable is in $use[n]$, then it is *live-in* at node n. That is, if a statement uses a variable, the variable is live on entry to that statement.
2. If a variable is *live-in* at a node n, then it is *live-out* at all nodes m in $pred[n]$.
3. If a variable is *live-out* at node n, and not in $def[n]$, then the variable is also *live-in* at n. That is, if someone needs the value of a at the end of statement n, and n does not provide that value, then a's value is needed even on entry to n.

These three statements can be written as Equations 10.3 on sets of variables. The live-in sets are an array $in[n]$ indexed by node, and the live-out sets are an array $out[n]$. That is, $in[n]$ is all the variables in $use[n]$, plus all the variables in $out[n]$ and not in $def[n]$. And $out[n]$ is the union of the live-in sets of all successors of n.

Algorithm 10.4 finds a solution to these equations by iteration. As usual, we initialize $in[n]$ and $out[n]$ to the the empty set $\{\}$, for all n, then repeatedly treat the equations as assignment statements until a fixed point is reached.

Table 10.5 shows the results of running the algorithm on Graph 10.1. The columns 1st, 2nd, etc. are the values of in and out on successive iterations of the repeat loop. Since the 7th column is the same as the 6th, the algorithm terminates.

We can speed the convergence of this algorithm significantly by ordering the nodes properly. Suppose there is an edge $3 \rightarrow 4$ in the graph. Since $in[4]$

			1st		2nd		3rd		4th		5th		6th		7th	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	

TABLE 10.5. Liveness calculation following forward control-flow edges.

			1st		2nd		3rd	
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

TABLE 10.6. Liveness calculation following reverse control-flow edges.

is computed from *out*[4], and *out*[3] is computed from *in*[4], and so on, we should compute the in and out sets in the order *out*[4] $\rightarrow$ *in*[4] $\rightarrow$ *out*[3] $\rightarrow$ *in*[3]. But in Table 10.5, just the opposite order is used in each iteration! We have waited as long as possible (in each iteration) to make use of information gained from the previous iteration.

Table 10.6 shows the computation, in which each for loop iterates from 6 to 1 (approximately following the *reversed* direction of the flow-graph arrows), and in each iteration the out sets are computed before the in sets. By the end of the second iteration, the fixed point has been found; the third iteration just confirms this.

When solving dataflow equations by iteration, the order of computation should follow the "flow." Since liveness flows *backward* along control-flow arrows, and from "out" to "in," so should the computation.

Ordering the nodes can be done easily by depth-first search, as shown in Section 17.4.

Basic blocks. Flow-graph nodes that have only one predecessor and one successor are not very interesting. Such nodes can be merged with their predecessors and successors; what results is a graph with many fewer nodes, where each node represents a basic block. The algorithms that operate on flow graphs, such as liveness analysis, go much faster on the smaller graphs. Chapter 17 explains how to adjust the dataflow equations to use basic blocks. In this chapter we keep things simple.

One variable at a time. Instead of doing dataflow "in parallel" using set equations, it can be just as practical to compute dataflow for one variable at a time as information about that variable is needed. For liveness, this would mean repeating the dataflow traversal once for each temporary. Starting from each *use* site of a temporary t, and tracing backward (following *predecessor* edges of the flow graph) using depth-first search, we note the liveness of t at each flow-graph node. The search stops at any definition of the temporary. Although this might seem expensive, many temporaries have very short live ranges, so the searches terminate quickly and do not traverse the entire flow graph for most variables.

REPRESENTATION OF SETS

There are at least two good ways to represent sets for dataflow equations: as arrays of bits or as sorted lists of variables.

If there are N variables in the program, the bit-array representation uses N bits for each set. Calculating the union of two sets is done by *or*-ing the corresponding bits at each position. Since computers can represent K bits per word (with $K = 32$ typical), one set-union operation takes N/K operations.

A set can also be represented as a linked list of its members, sorted by any totally ordered key (such as variable name). Calculating the union is done by merging the lists (discarding duplicates). This takes time proportional to the size of the sets being unioned.

Clearly, when the sets are sparse (fewer than N/K elements, on the average), the sorted-list representation is asymptotically faster; when the sets are dense, the bit-array representation is better.

TIME COMPLEXITY

How fast is iterative dataflow analysis?

A program of size N has at most N nodes in the flow graph, and at most N variables. Thus, each live-in set (or live-out set) has at most N elements;

			X		Y		Z	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac	cd	acd	c	ac
2	a	b	ac	bc	acd	bcd	ac	b
3	bc	c	bc	bc	bcd	bcd	b	b
4	b	a	bc	ac	bcd	acd	b	ac
5	a		ac	ac	acd	acd	ac	ac
6	c		c		c		c	

TABLE 10.7. X and Y are solutions to the liveness equations; Z is not a solution.

each set-union operation to compute live-in (or live-out) takes $O(N)$ time.

The for loop computes a constant number of set operations per flow-graph node; there are $O(N)$ nodes; thus, the for loop takes $O(N^2)$ time.

Each iteration of the repeat loop can only make each in or out set larger, never smaller. This is because the in and out sets are *monotonic* with respect to each other. That is, in the equation $in[n] = use[n] \cup (out[n] - def[n])$, a larger $out[n]$ can only make $in[n]$ larger. Similarly, in $out[n] = \bigcup_{s \in succ[n]} in[s]$, a larger $in[s]$ can only make $out[n]$ larger.

Each iteration must add something to the sets; but the sets cannot keep growing infinitely; at most every set can contain every variable. Thus, the sum of the sizes of all in and out sets is $2N^2$, which is the most that the repeat loop can iterate.

Thus, the worst-case run time of this algorithm is $O(N^4)$. Ordering the nodes using depth-first search (Algorithm 17.5, page 396) usually brings the number of repeat-loop iterations to two or three, and the live-sets are often sparse, so the algorithm runs between $O(N)$ and $O(N^2)$ in practice.

Section 17.4 discusses more sophisticated ways of solving dataflow equations quickly.

LEAST FIXED POINTS

Table 10.7 illustrates two solutions (and a nonsolution!) to the Equations 10.3; assume there is another program variable d not used in this fragment of the program.

In solution Y, the variable d is carried uselessly around the loop. But in fact, Y satisfies Equations 10.3 just as X does. What does this mean? Is d live or not?

The answer is that any solution to the dataflow equations is a *conservative approximation*. If the value of variable a will truly be needed in some execution of the program when execution reaches node n of the flow graph, then we can be assured that a is live-out at node n in any solution of the equations. But the converse is not true; we might calculate that d is live-out, but that doesn't mean that its value will really be used.

Is this acceptable? We can answer that question by asking what use will be made of the dataflow information. In the case of liveness analysis, if a variable is *thought to be live* then we will make sure to have its value in a register. A conservative approximation of liveness is one that may erroneously believe a variable is live, but will never erroneously believe it is dead. The consequence of a conservative approximation is that the compiled code might use more registers than it really needs; but it will compute the right answer.

Consider instead the live-in sets Z, which fail to satisfy the dataflow equations. Using this Z we think that b and c are never live at the same time, and we would assign them to the same register. The resulting program would use an optimal number of registers but *compute the wrong answer*.

A dataflow equation used for compiler optimization should be set up so that any solution to it provides conservative information to the optimizer; imprecise information may lead to suboptimal but never incorrect programs.

Theorem. Equations 10.3 have more than one solution.

Proof. X and Y are both solutions.

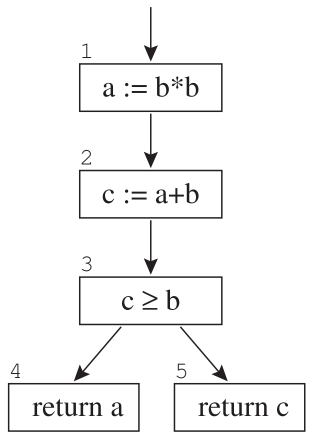
Theorem. All solutions to Equations 10.3 contain solution X. That is, if $in_X[n]$ and $in_Y[n]$ are the live-in sets for some node n in solutions X and Y, then $in_X[n] \subseteq in_Y[n]$.

Proof. See Exercise 10.2.

We say that X is the *least solution* to Equations 10.3. Clearly, since a bigger solution will lead to using more registers (producing suboptimal code), we want to use the least solution. Fortunately, Algorithm 10.4 always computes the least fixed point.

STATIC VS. DYNAMIC LIVENESS

A variable is live "if its value will be used in the future." In Graph 10.8, we know that $b \times b$ must be nonnegative, so that the test $c \geq b$ will be true. Thus,



GRAPH 10.8. Standard static dataflow analysis will not take advantage of the fact that node 4 can never be reached.

node 4 will never be reached, and a's value will not be used after node 2; a is not live-out of node 2.

But Equations 10.3 say that a is live-in to node 4, and therefore live-out of nodes 3 and 2. The equations are ignorant of which way the conditional branch will go. "Smarter" equations would permit a and c to be assigned the same register.

Although we can prove here that $b*b \geq 0$, and we could have the compiler look for arithmetic identities, no compiler can ever fully understand how all the control flow in every program will work. This is a fundamental mathematical theorem, derivable from the halting problem.

Theorem. There is no program H that takes as input any program P and input X and (without infinite-looping) returns true if $P(X)$ halts and false if $P(X)$ infinite-loops.

Proof. Suppose that there were such a program H; then we could arrive at a contradiction as follows. From the program H, construct the function F,

$$F(Y) = \text{if } H(Y, Y) \text{ then (while true do ()) else true}$$

By the definition of H, if $F(F)$ halts, then $H(F, F)$ is true; so the then clause is taken; so the while loop executes forever; so $F(F)$ does not halt. But if $F(F)$ loops forever, then $H(F, F)$ is false; so the else clause is taken; so $F(F)$ halts. The program $F(F)$ halts if it doesn't halt, and doesn't halt if it halts: a contradiction. Thus there can be no program H that tests whether another program halts (and always halts itself).

Corollary. No program $H'(X, L)$ can tell, for any program X and label L within X, whether the label L is ever reached on an execution of X.

Proof. From H' we could construct H. In some program that we want to test for halting, just let L be the end of the program, and replace all instances of the halt command with goto L.

Conservative approximation. This theorem does not mean that we can *never* tell if a given label is reached or not, just that there is not a general algorithm that can *always* tell. We could improve our liveness analysis with some special-case algorithms that, in some cases, calculate more information about run-time control flow. But any such algorithm will come up against many cases where it simply cannot tell exactly what will happen at run time.

Because of this inherent limitation of program analysis, no compiler can really tell if a variable's value is truly needed – whether the variable is truly live. Instead, we have to make do with a conservative approximation. We assume that any conditional branch goes both ways. Thus, we have a dynamic condition and its static approximation:

Dynamic liveness A variable a is dynamically live at node n if some execution of the program goes from n to a use of a without going through any definition of a.

Static liveness A variable a is statically live at node n if there is some path of control-flow edges from n to some use of a that does not go through a definition of a.

Clearly, if a is dynamically live it is also statically live. An optimizing compiler must allocate registers, and do other optimizations, on the basis of static liveness, because (in general) dynamic liveness cannot be computed.

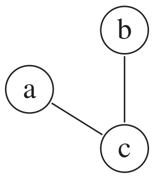
INTERFERENCE GRAPHS

Liveness information is used for several kinds of optimization in a compiler. For some optimizations, we need to know exactly which variables are live at each node in the flow graph.

One of the most important applications of liveness analysis is for register allocation: we have a set of temporaries $a, b, c, \ldots$ that must be allocated to

	a	b	c
a			x
b			x
c	x	x	

(a) Matrix



(b) Graph

FIGURE 10.9. Representations of interference.

registers $r_1, \ldots, r_k$. A condition that prevents a and b being allocated to the same register is called an *interference*.

The most common kind of interference is caused by overlapping live ranges: when a and b are both live at the same program point, then they cannot be put in the same register. But there are some other causes of interference: for example, when a must be generated by an instruction that cannot address register r_1, then a and r_1 interfere.

Interference information can be expressed as a matrix; Figure 10.9a has an x marking interferences of the variables in Graph 10.1. The interference matrix can also be expressed as an undirected graph (Figure 10.9b), with a node for each variable, and edges connecting variables that interfere.

Special treatment of MOVE instructions. In static liveness analysis, we can give MOVE instructions special consideration. It is important not to create artifical interferences between the source and destination of a move. Consider the program:

$$\begin{array}{ll}
t \leftarrow s & (\textit{copy}) \\
\quad\vdots & \\
x \leftarrow \ldots\ s\ \ldots & (\textit{use of } s) \\
\quad\vdots & \\
y \leftarrow \ldots\ t\ \ldots & (\textit{use of } t)
\end{array}$$

After the copy instruction both s and t are live, and normally we would make an interference edge (s, t) since t is being defined at a point where s is live. But we do not need separate registers for s and t, since they contain the same value. The solution is just not to add an interference edge (t, s) in this case. Of course, if there is a later (nonmove) definition of t while s is still live, that will create the interference edge (t, s).

Therefore, the way to add interference edges for each new definition is:

1. At any nonmove instruction that *defines* a variable a, where the *live-out* variables are $b_1, \ldots, b_j$, add interference edges $(a, b_1), \ldots, (a, b_j)$.
2. At a move instruction $a \leftarrow c$, where variables $b_1, \ldots, b_j$ are *live-out*, add interference edges $(a, b_1), \ldots, (a, b_j)$ for any b_i that is *not* the same as c.

10.2 LIVENESS IN THE Tiger COMPILER

The flow analysis for the Tiger compiler is done in two stages: first, the control flow of the Assem program is analyzed, producing a control-flow graph; then, the liveness of variables in the control-flow graph is analyzed, producing an interference graph.

GRAPHS

To represent both kinds of graphs, let's make a Graph abstract data type (Program 10.10).

The function G_Graph() creates an empty directed graph; G_Node(g,x) makes a new node within a graph g, where x is any extra information that the caller wants to keep "attached" to the new node. A directed edge from n to m is created by G_addEdge(n,m); after that, m will be found in the list g_succ(n) and n will be in G_pred(m). When working with undirected graphs, the function adj is useful: G_adj(m) = G_succ(m) ∪ G_pred(m).

To delete an edge, use G_rmEdge. To test whether m and n are the same node, use m==n.

When using a graph in an algorithm, we want each node to represent something (an instruction in a program, for example). To make mappings from nodes to the things they are supposed to represent, we use a G_table. The following idiom associates information x with node n in a mapping mytable.

```
G_enter(mytable,n,x);
```

Instead of keeping a separate table mapping $n \mapsto x$, we can put x directly inside n. Executing n=G_Node(g,x) creates a new node n with "associated information" x. Calling G_nodeInfo(n) retrieves x.

CONTROL-FLOW GRAPHS

The Flowgraph module manages control-flow graphs. Each instruction (or basic block) is represented by a node in the flow graph. If instruction m can be followed by instruction n (either by a jump or by falling through), then there will be an edge (m, n) in the graph.

```
/* graph.h */

typedef struct G_graph_ *G_graph;                     The graph type
typedef struct G_node_ *G_node;                       The node type

typedef struct G_nodeList_ *G_nodeList;               List of nodes
struct G_nodeList_ { G_node head; G_nodeList tail;};

G_graph G_Graph(void);                                Make a new graph
G_node G_Node(G_graph g, void *info);                 Make new node in g
G_nodeList G_NodeList(G_node head, G_nodeList tail);
G_nodeList G_nodes(G_graph g);                        Get the list of g's nodes
bool G_inNodeList(G_node a, G_nodeList l);            Is a in l?
void G_addEdge(G_node from, G_node to);               Make a new edge
void G_rmEdge(G_node from, G_node to);                Delete an edge
void G_show(FILE *out, G_nodeList p, void showInfo(void *));
G_nodeList G_succ(G_node n);                          Get n's successors
G_nodeList G_pred(G_node n);                          Get n's predecessors
G_nodeList G_adj(G_node n);                           G_ succ(n) ∪ G_ pred(n)
bool G_goesTo(G_node a, G_node b);                    Is there an edge from a to b?
int G_degree(G_node n);                               How many edges to/from n?
void *G_nodeInfo(G_node n);                           Get n's info

typedef struct TAB_table_  *G_table;                  Mapping nodes to whatever
G_table G_empty(void);                                Make a new table
void G_enter(G_table t, G_node n, void *v);           Make n↦ v in t
void *G_look(G_table t, G_node n);                    Find n ↦ v, report v
```

PROGRAM 10.10. The `Graph` abstract data type.

```
/* flowgraph.h */
Temp_tempList FG_def(G_node n);
Temp_tempList FG_use(G_node n);
bool FG_isMove(G_node n);

G_graph FG_AssemFlowGraph(AS_instrList il);
```

A *flow graph* is just a `G_graph` with some extra (hidden) information in each node. From this information it is possible to learn three things about each node n:

`FG_def`(n) a list of the temporaries defined at node n (destination registers of the instruction);

`FG_use`(n) a list of the temporaries used at n (source registers of the instruction);

`FG_isMove`(n) tells whether n represents a MOVE instruction, that could be deleted if the `def` and `use` are identical.

The `Flowgraph` module is an abstract data type; clients cannot see the information inside the nodes. The implementation – `flowgraph.h` – contains a function `FG_AssemFlowGraph` that takes a list of instructions and returns a flow graph in which the `info` of each `G_node` is actually a pointer to an `AS_instr`. In making the flow graph, the `jump` fields of the `instrs` are used in creating control-flow edges, and the `use` and `def` information is obtained from the `src` and `dst` fields of the instructions. Clients of `Flowgraph` should never call `G_nodeInfo` directly, but go through the operations provided in `flowgraph.h`.

Information associated with the nodes. For a flow graph, we want to associate some *use* and *def* information with each node in the graph. Then the liveness-analysis algorithm will also want to remember *live-in* and *live-out* information at each node. We have made room in the `G_node` struct to store all of this information – this is the "associated information" accessed by `G_nodeInfo()`. This works well and is quite efficient. However, it may not be very modular. Eventually we may want to do other analyses on flow graphs, which remember other kinds of information about each node. We may not want to modify the data structure (which is a widely used interface) for each new analysis.

Instead of storing the information *in* the nodes, a more modular approach is to say that a graph is a graph, and that a flow graph is a graph along with separately packaged auxiliary information (tables, or functions mapping nodes to whatever). Similarly, a dataflow algorithm on a graph does not need to modify dataflow information *in* the nodes, but modifies its own privately held mappings.

There may be a trade-off here between efficiency and modularity, since it may be faster to keep the information *in* the nodes, accessibly by a simple pointer-traversal instead of a hash-table or search-tree lookup.

We will compromise by putting the "permanent" information about a node – that is, the `Assem` instruction it represents – into the `info`, and making up a `G_table` for each dataflow analysis that needs to keep additional information about the nodes.

LIVENESS ANALYSIS

The `Liveness` module takes a flow graph and produces two things: an interference graph, and a list of node-pairs (representing MOVE instructions) that should be assigned the same register if possible (so that the MOVE can be deleted).

/ liveness.h */*

```
typedef struct Live_moveList_ *Live_moveList;
struct Live_moveList_ {G_node src, dst;
                       Live_moveList tail;};
Live_moveList Live_MoveList(G_node src, G_node dst,
                            Live_moveList tail);

struct Live_graph { G_graph graph; Live_moveList moves; };
Temp_temp Live_gtemp(G_node n);

struct Live_graph Live_liveness(G_graph flow);
```

For an interference-graph node n, `Live_gtemp` tells what temporary variable is represented by n. It is implemented, of course, by making the `info` field of each graph node point to a `Temp_temp`.

In the implementation of the `Liveness` module, it is useful to maintain a data structure that remembers what is live at the exit of each flow-graph node:

```
static void enterLiveMap(G_table t, G_node flowNode,
                         Temp_tempList temps) {
  G_enter(t, flowNode, temps);
}
static Temp_tempList lookupLiveMap(G_table t,
                                   G_node flownode) {
  return (Temp_tempList)G_look(t, flownode);
}
```

Given a flow-graph node n, the set of live temporaries at that node can be looked up in a global `liveMap`.

Having calculated a complete `liveMap`, we can now construct an interference graph. At each flow node n where there is a newly defined temporary $d \in \mathit{def}(n)$, and where temporaries $\{t_1, t_2, \ldots\}$ are in the `liveMap`, we just add interference edges $(d, t_1), (d, t_2), \ldots$. For MOVEs, these edges will be safe but suboptimal; pages 228–229 describe a better treatment.

What if a newly defined temporary is not live just after its definition? This would be the case if a variable is defined but never used. It would seem that there's no need to put it in a register at all; thus it would not interfere with any other temporaries. But if the defining instruction is going to execute (perhaps it is necessary for some other side effect of the instruction), then it *will* write to some register, and that register had better not contain any other live variable. Thus, zero-length live ranges *do* interfere with any live ranges that overlap them.

PROGRAM CONSTRUCTING FLOW GRAPHS

Implement `flowgraph.c` that turns a list of `Assem` instructions into a flow graph. Use the interfaces `graph.h` and `flowgraph.h`, and the implementation of `graph.c`, provided in `$TIGER/chap10`.

PROGRAM LIVENESS

Implement the `Liveness` module. Use either the set-equation algorithm with the array-of-boolean or sorted-list-of-temporaries representation of sets, or the one-variable-at-a-time method.

EXERCISES

10.1 Perform flow analysis on the program of Exercise 8.6:

a. Draw the control-flow graph.

b. Calculate live-in and live-out at each statement.

c. Construct the register interference graph.

****10.2** Prove that Equations 10.3 have a least fixed point and that Algorithm 10.4 always computes it.

Hint: We know the algorithm refuses to terminate until it has a fixed point. The questions are whether (a) it must eventually terminate, and (b) the fixed point it computes is smaller than all other fixed points. For (a) show that the sets can only get bigger. For (b) show by induction that at any time, the *in* and *out* sets are subsets of those in any possible fixed point. This is clearly true initially, when *in* and *out* are both empty; show that each step of the algorithm preserves the invariant.

***10.3** Analyze the asymptotic complexity of the one-variable-at-a-time method of computing dataflow information.

***10.4** Analyze the worst-case asymptotic complexity of making an interference graph, for a program of size N (with at most N variables and at most N control-flow nodes). Assume the dataflow analysis is already done and that *use*, *def*, and *live-out* information for each node can be queried in constant time. What representation of graph adjacency matrices should be used for efficiency?

10.5 The DEC Alpha architecture places the following restrictions on floating-point instructions, for programs that wish to recover from arithmetic exceptions:

1. Within a basic block (actually, in any sequence of instructions not separated by a *trap-barrier* instruction), no two instructions should write to the same destination register.
2. A source register of an instruction cannot be the same as the destination register of that instruction or any later instruction in the basic block.

$r_1 + r_5 \rightarrow r_4$	$r_1 + r_5 \rightarrow r_4$	$r_1 + r_5 \rightarrow r_3$	$r_1 + r_5 \rightarrow r_4$
$r_3 \times r_2 \rightarrow r_4$	$r_4 \times r_2 \rightarrow r_1$	$r_4 \times r_2 \rightarrow r_4$	$r_4 \times r_2 \rightarrow r_6$
violates rule 1.	*violates rule 2.*	*violates rule 2.*	*OK*

Show how to express these restrictions in the register interference graph.

11 Register Allocation

reg-is-ter: a device for storing small amounts of data
al-lo-cate: to apportion for a specific purpose

Webster's Dictionary

The `Translate`, `Canon`, and `Codegen` phases of the compiler assume that there are an infinite number of registers to hold temporary values and that MOVE instructions cost nothing. The job of the register allocator is to assign the many temporaries to a small number of machine registers, and, where possible, to assign the source and destination of a MOVE to the same register so that the MOVE can be deleted.

From an examination of the control and dataflow graph, we derive an *interference graph*. Each node in the inteference graph represents a temporary value; each edge (t_1, t_2) indicates a pair of temporaries that cannot be assigned to the same register. The most common reason for an interference edge is that t_1 and t_2 are live at the same time. Interference edges can also express other constraints; for example, if a certain instruction $a \leftarrow b \oplus c$ cannot produce results in register r_{12} on our machine, we can make a interfere with r_{12}.

Next we *color* the interference graph. We want to use as few colors as possible, but no pair of nodes connected by an edge may be assigned the same color. Graph coloring problems derive from the old mapmakers' rule that adjacent countries on a map should be colored with different colors. Our "colors" correspond to registers: if our target machine has K registers, and we can K-color the graph (color the graph with K colors), then the coloring is a valid register assignment for the inteference graph. If there is no K-coloring, we will have to keep some of our variables and temporaries in memory instead of registers; this is called *spilling*.

11.1 COLORING BY SIMPLIFICATION

Register allocation is an NP-complete problem (except in special cases, such as expression trees); graph coloring is also NP-complete. Fortunately there is a linear-time approximation algorithm that gives good results; its principal phases are Build, Simplify, Spill, and Select.

Build: Construct the interference graph. We use dataflow analysis to compute the set of temporaries that are simultaneously live at each program point, and we add an edge to the graph for each pair of temporaries in the set. We repeat this for all program points.

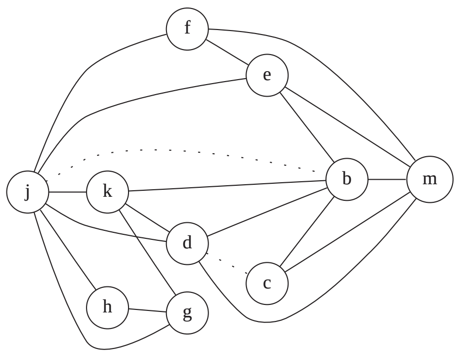
Simplify: We color the graph using a simple heuristic. Suppose the graph G contains a node m with fewer than K neighbors, where K is the number of registers on the machine. Let G' be the graph $G - \{m\}$ obtained by removing m. If G' can be colored, then so can G, for when m is added to the colored graph G', the neighbors of m have at most $K - 1$ colors among them so a free color can always be found for m. This leads naturally to a stack-based (or recursive) algorithm for coloring: we repeatedly remove (and push on a stack) nodes of degree less than K. Each such simplification will decrease the degrees of other nodes, leading to more opportunity for simplification.

Spill: Suppose at some point during simplification the graph G has nodes only of *significant degree*, that is, nodes of degree $\geq K$. Then the *simplify* heuristic fails, and we mark some node for spilling. That is, we choose some node in the graph (standing for a temporary variable in the program) and decide to represent it in memory, not registers, during program execution. An optimistic approximation to the effect of spilling is that the spilled node does not interfere with any of the other nodes remaining in the graph. It can therefore be removed and pushed on the stack, and the simplify process continued.

Select: We assign colors to nodes in the graph. Starting with the empty graph, we rebuild the original graph by repeatedly adding a node from the top of the stack. When we add a node to the graph, there must be a color for it, as the premise for removing it in the simplify phase was that it could always be assigned a color provided the remaining nodes in the graph could be successfully colored.

When *potential spill* node n that was pushed using the *Spill* heuristic is

```
live-in:  k j
      g := mem[j+12]
      h := k - 1
      f := g * h
      e := mem[j+8]
      m := mem[j+16]
      b := mem[f]
      c := e + 8
      d := c
      k := m + 4
      j := b
live-out:  d k j
```



GRAPH 11.1. Interference graph for a program. Dotted lines are not interference edges but indicate move instructions.

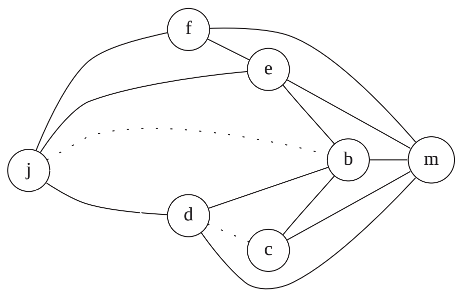
popped, there is no guarantee that it will be colorable: its neighbors in the graph may be colored with K different colors already. In this case, we have an *actual spill*. We do not assign any color, but we continue the *Select* phase to identify other actual spills.

But perhaps some of the neighbors are the same color, so that among them there are fewer than K colors. Then we can color n, and it does not become an actual spill. This technique is known as *optimistic coloring*.

Start over: If the Select phase is unable to find a color for some node(s), then the program must be rewritten to fetch them from memory just before each use, and store them back after each definition. Thus, a spilled temporary will turn into several new temporaries with tiny live ranges. These will interfere with other temporaries in the graph. So the algorithm is repeated on this rewritten program. This process iterates until *simplify* succeeds with no spills; in practice, one or two iterations almost always suffice.

EXAMPLE

Graph 11.1 shows the interferences for a simple program. The nodes are labeled with the temporaries they represent, and there is an edge between two nodes if they are simultaneously live. For example, nodes `d`, `k`, and `j` are all connected since they are live simultaneously at the end of the block. Assuming that there are four registers available on the machine, then the simplify phase can start with the nodes `g`, `h`, `c`, and `f` in its working set, since they



GRAPH 11.2. After removal of h, g, k.

m	1
c	3
b	2
f	2
e	4
j	3
d	4
k	1
h	2
g	4
(a) stack	(b) assignment

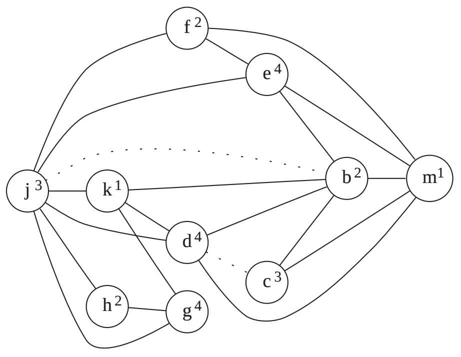


FIGURE 11.3. Simplification stack, and a possible coloring.

have less than four neighbors each. A color can always be found for them if the remaining graph can be successfully colored. If the algorithm starts by removing h and g and all their edges, then node k becomes a candidate for removal and can be added to the work-list. Graph 11.2 remains after nodes g, h, and k have been removed. Continuing in this fashion a possible order in which nodes are removed is represented by the stack shown in Figure 11.3a, where the stack grows upward.

The nodes are now popped off the stack and the original graph reconstructed and colored simultaneously. Starting with m, a color is chosen arbitrarily since the graph at this point consists of a singleton node. The next node to be put into the graph is c. The only constraint is that it be given a color different from m, since there is an edge from m to c. A possible assignment of colors for the reconstructed original graph is shown in Figure 11.3b.

11.2 COALESCING

It is easy to eliminate redundant move instructions with an interference graph. If there is no edge in the interference graph between the source and destination of a move instruction, then the move can be eliminated. The source and destination nodes are *coalesced* into a new node whose edges are the union of those of the nodes being replaced.

In principle, any pair of nodes not connected by an interference edge could be coalesced. This aggressive form of copy propagation is very successful at eliminating move instructions. Unfortunately, the node being introduced is more constrained than those being removed, as it contains a union of edges. Thus, it is quite possible that a graph, colorable with K colors before coalescing, may no longer be K-colorable after reckless coalescing. We wish to coalesce only where it is *safe* to do so, that is, where the coalescing will not render the graph uncolorable. Both of the following strategies are safe:

Briggs: Nodes a and b can be coalesced if the resulting node ab will have fewer than K neighbors of significant degree (i.e., having $\geq K$ edges). The coalescing is guaranteed not to turn a K-colorable graph into a non-K-colorable graph, because after the simplify phase has removed all the insignificant-degree nodes from the graph, the coalesced node will be adjacent only to those neighbors that were of significant degree. Since there are fewer than K of these, *simplify* can then remove the coalesced node from the graph. Thus if the original graph was colorable, the conservative coalescing strategy does not alter the colorability of the graph.

George: Nodes a and b can be coalesced if, for every neighbor t of a, either t already interferes with b or t is of insignificant degree. This coalescing is safe, by the following reasoning. Let S be the set of insignificant-degree neighbors of a in the original graph. If the coalescing were not done, *simplify* could remove all the nodes in S, leaving a reduced graph G_1. If the coalescing is done, then *simplify* can remove all the nodes in S, leaving a graph G_2. But G_2 is a subgraph of G_1 (the node ab in G_2 corresponds to the node b in G_1), and thus must be at least as easy to color.

These strategies are *conservative*, because there are still safe situations in which they will fail to coalesce. This means that the program may perform some unnecessary MOVE instructions – but this is better than spilling!

Interleaving simplification steps with conservative coalescing eliminates most move instructions, while still guaranteeing not to introduce spills. The coalesce, simplify, and spill procedures should be alternated until the graph is empty, as shown in Figure 11.4.

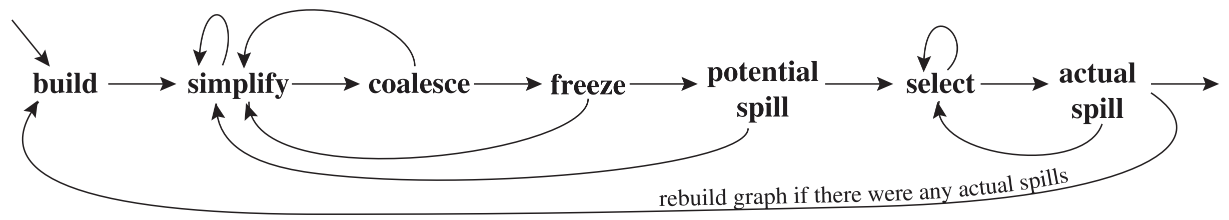


FIGURE 11.4. Graph coloring with coalescing.

These are the phases of a register allocator with coalescing:

Build: Construct the interference graph, and categorize each node as either *move-related* or *non-move-related*. A move-related node is one that is either the source or destination of a move instruction.

Simplify: One at a time, remove non-move-related nodes of low ($< K$) degree from the graph.

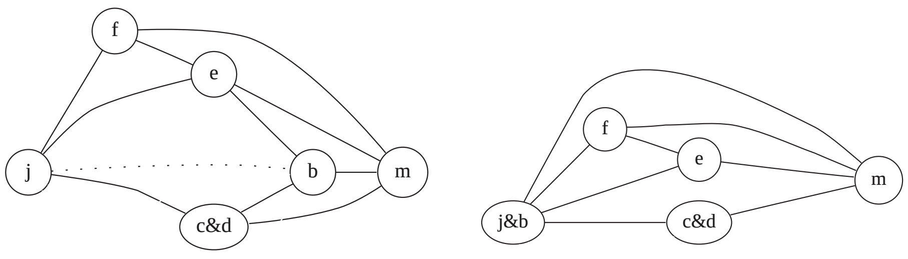
Coalesce: Perform conservative coalescing on the reduced graph obtained in the simplification phase. Since the degrees of many nodes have been reduced by *simplify*, the conservative strategy is likely to find many more moves to coalesce than it would have in the initial interference graph. After two nodes have been coalesced (and the move instruction deleted), if the resulting node is no longer move-related it will be available for the next round of simplification. *Simplify* and *coalesce* are repeated until only significant-degree or move-related nodes remain.

Freeze: If neither *simplify* nor *coalesce* applies, we look for a move-related node of low degree. We *freeze* the moves in which this node is involved: that is, we give up hope of coalescing those moves. This causes the node (and perhaps other nodes related to the frozen moves) to be considered non-move-related, which should enable more simplification. Now, *simplify* and *coalesce* are resumed.

Spill: If there are no low-degree nodes, we select a significant-degree node for potential spilling and push it on the stack.

Select: Pop the entire stack, assigning colors.

Consider Graph 11.1; nodes `b`, `c`, `d`, and `j` are the only move-related nodes. The initial work-list used in the simplify phase must contain only non-move-



GRAPH 11.5. (a) after coalescing `c` and `d`; (b) after coalescing `b` and `j`.

stack	coloring
e	1
m	2
f	3
j&b	4
c&d	1
k	2
h	2
g	1

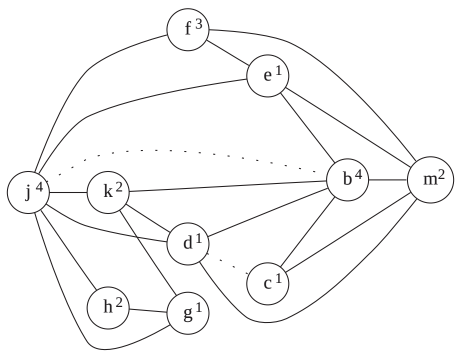


FIGURE 11.6. A coloring, with coalescing, for Graph 11.1.

related nodes and consists of nodes `g`, `h`, and `f`. Once again, after removal of `g`, `h`, and `k` we obtain Graph 11.2.

We could continue the simplification phase further; however, if we invoke a round of coalescing at this point, we discover that `c` and `d` are indeed coalesceable as the coalesced node has only two neighbors of significant degree: `m` and `b`. The resulting graph is shown in Graph 11.5a, with the coalesced node labeled as `c&d`.

From Graph 11.5a we see that it is possible to coalesce `b` and `j` as well. Nodes `b` and `j` are adjacent to two neighbors of significant degree, namely `m` and `e`. The result of coalescing `b` and `j` is shown in Graph 11.5b.

After coalescing these two moves, there are no more move-related nodes, and therefore no more coalescing is possible. The simplify phase can be invoked one more time to remove all the remaining nodes. A possible assignment of colors is shown in Figure 11.6.

Some moves are neither coalesced nor frozen. Instead, they are *constrained*. Consider the graph x, y, z, where (x, z) is the only interference edge and there are two moves $x \leftarrow y$ and $y \leftarrow z$. Either move is a candidate for coalescing. But after x and y are coalesced, the remaining move $xy \leftarrow z$ cannot be coalesced because of the interference edge (xy, z). We say this move is *constrained*, and we remove it from further consideration: it no longer causes nodes to be treated as move-related.

SPILLING

If spilling is necessary, *build* and *simplify* must be repeated on the whole program. The simplest version of the algorithm discards any coalescences found if *build* must be repeated. Then it is easy to see that coalescing does not increase the number of spills in any future round of *build*. A more efficient algorithm preserves any coalescences done *before the first potential spill was discovered*, but discards (uncoalesces) any coalescences done after that point.

Coalescing of spills. On a machine with many registers (> 20), there will usually be few spilled nodes. But on a six-register machine (such as the Intel Pentium), there will be many spills. The front end may have generated many temporaries, and transformations such as SSA (described in Chapter 19) may split them into many more temporaries. If each spilled temporary lives in its own stack-frame location, then the frame may be quite large.

Even worse, there may be many move instructions involving pairs of spilled nodes. But to implement $a \leftarrow b$ when a and b are both spilled temporaries requires a fetch-store sequence, $t \leftarrow M[a_{\text{loc}}];\ M[b_{\text{loc}}] \leftarrow t$. This is expensive, and also defines a temporary t that itself may cause other nodes to spill.

But many of the spill pairs are never live simultaneously. Thus, they may be graph-colored, with coalescing! In fact, because there is no fixed limit to the number of stack-frame locations, we can coalesce aggressively, without worrying about how many high-degree neighbors the spill-nodes have. The algorithm is thus:

1. Use liveness information to construct the interference graph for spilled nodes.
2. While there is any pair of non-interfering spilled nodes connected by a move instruction, coalesce them.
3. Use *simplify* and *select* to color the graph. There is no (further) spilling in this coloring; instead, *simplify* just picks the lowest-degree node, and *select* picks the first available color, without any predetermined limit on the number of colors.
4. The colors correspond to activation-record locations for the spilled variables.

This should be done *before* generating the spill instructions and regenerating the register-temporary interference graph, so as to avoid creating fetch-store sequences for coalesced moves of spilled nodes.

11.3 PRECOLORED NODES

Some temporaries are *precolored* – they represent machine registers. The front end generates these when interfacing to standard calling conventions across module boundaries, for example. For each actual register that is used for some specific purpose, such as the frame pointer, standard-argument-1-register, standard-argument-2-register, and so on, the `Codegen` or `Frame` module should use the particular temporary that is permanently bound to that register (see also page 267). For any given color (that is, for any given machine register) there should be only one precolored node of that color.

The *select* and *coalesce* operations can give an ordinary temporary the same color as a precolored register, as long as they don't interfere, and in fact this is quite common. Thus, a standard calling-convention register can be reused inside a procedure as a temporary variable. Precolored nodes may be coalesced with other (non-precolored) nodes using conservative coalescing.

For a K-register machine, there will be K precolored nodes that all interfere with each other. Those of the precolored nodes that are not used explicitly (in a parameter-passing convention, for example) will not interfere with any ordinary (non-precolored) nodes; but a machine register used explicitly will have a live range that interferes with any other variables that happen to be live at the same time.

We cannot *simplify* a precolored node – this would mean pulling it from the graph in the hope that we can assign it a color later, but in fact we have no freedom about what color to assign it. And we should not spill precolored nodes to memory, because the machine registers are by definition *registers*. Thus, we should treat them as having "infinite" degree.

TEMPORARY COPIES OF MACHINE REGISTERS

The coloring algorithm works by calling *simplify, coalesce,* and *spill* until only the precolored nodes remain, and then the *select* phase can start adding the other nodes (and coloring them).

Because precolored nodes do not spill, the front end must be careful to keep their live ranges short. It can do this by generating MOVE instructions to move values to and from precolored nodes. For example, suppose r_7 is a callee-save register; it is "defined" at procedure entry and "used" at procedure exit. Instead of being kept in a precolored register throughout the procedure (Figure 11.7a), it can be moved into a fresh temporary and then moved back

enter:	def(r_7)	enter:	def(r_7)
			$t_{231} \leftarrow r_7$
	$\vdots$		$\vdots$
			$r_7 \leftarrow t_{231}$
exit:	use(r_7)	exit:	use(r_7)

FIGURE 11.7. Moving a callee-save register to a fresh temporary.

(Figure 11.7b). If there is *register pressure* (a high demand for registers) in this function, t_{231} will spill; otherwise t_{231} will be coalesced with r_7 and the MOVE instructions will be eliminated.

CALLER-SAVE AND CALLEE-SAVE REGISTERS

The most basic of spill heuristics can achieve the effect of allocating variables live across calls to callee-save registers. A local variable or compiler temporary that is not live across any procedure call should usually be allocated to a caller-save register, because in this case no saving and restoring of the register will be necessary at all. On the other hand, any variable that is live across several procedure calls should be kept in a callee-save register, since then only one save/restore will be necessary (on entry/exit from the calling procedure).

The register allocator should allocate variables to registers using this criterion. Fortunately, a graph-coloring allocator with spilling can do this very easily. The CALL instructions in the `Assem` language have been annotated to *define* (interfere with) all the caller-save registers. If a variable is not live across a procedure call, it will tend to be allocated to a caller-save register.

If a variable x is live across a procedure call, then it interferes with all the caller-save (precolored) registers, *and* it interferes with all the new temporaries (such as t_{231} in Figure 11.7) created for callee-save registers. Thus, a spill will occur. Using the common spill-cost heuristic that spills a node with high degree but few uses, the node chosen for spilling will not be x but t_{231}. Since t_{231} is spilled, r_7 will be available for coloring x (or some other variable).

EXAMPLE WITH PRECOLORED NODES

A worked example will illustrate the issues of register allocation with precolored nodes, callee-save registers, and spilling.

A C compiler is compiling Program 11.8a for a target machine with three

```
int f(int a, int b) {
   int d=0;
   int e=a;
   do {d = d+b;
       e = e-1;
   } while (e>0);
   return d;
}
```

(a)

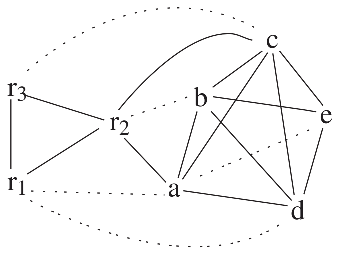
```
enter:  $c \leftarrow r_3$
        $a \leftarrow r_1$
        $b \leftarrow r_2$
        $d \leftarrow 0$
        $e \leftarrow a$
loop:   $d \leftarrow d + b$
        $e \leftarrow e - 1$
        if $e > 0$ goto loop
        $r_1 \leftarrow d$
        $r_3 \leftarrow c$
        return          ($r_1, r_3$ live out)
```

(b)

PROGRAM 11.8. A C function and its translation into instructions

registers; r_1 and r_2 are caller-save, and r_3 is callee-save. The code generator has therefore made arrangements to preserve the value of r_3 explicitly, by copying it into the temporary c and back again.

The instruction-selection phase has produced the instruction-list of Program 11.8b. The interference graph for this function is shown at right.

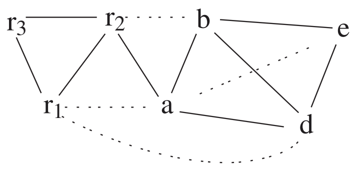


The register allocation proceeds as follows (with $K = 3$):

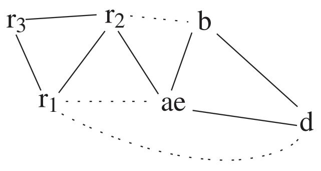
1. In this graph, there is no opportunity for *simplify* or *freeze* (because all the non-precolored nodes have degree $\geq K$). Any attempt to *coalesce* would produce a coalesced node adjacent to K or more significant-degree nodes. Therefore we must *spill* some node. We calculate spill priorities as follows:

Node		Uses+Defs outside loop		Uses+Defs within loop			Degree		Spill priority
a	(	2	$+ 10 \times$	0	)	/	4	=	0.50
b	(	1	$+ 10 \times$	1	)	/	4	=	2.75
c	(	2	$+ 10 \times$	0	)	/	6	=	0.33
d	(	2	$+ 10 \times$	2	)	/	4	=	5.50
e	(	1	$+ 10 \times$	3	)	/	3	=	10.33

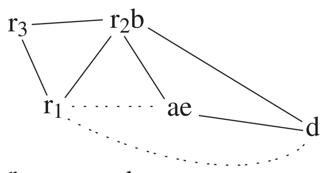
Node c has the lowest priority – it interferes with many other temporaries but is rarely used – so it should be spilled first. Spilling c, we obtain the graph at right.



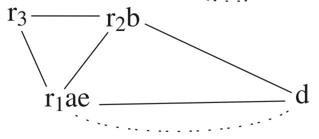
2. We can now coalesce a and e, since the resulting node will be adjacent to fewer than K significant-degree nodes (after coalescing, node d will be low-degree, though it is significant-degree right now). No other *simplify* or *coalesce* is possible now.



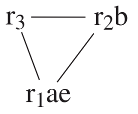
3. Now we could coalesce ae&r_1 or coalesce b&r_2. Let us do the latter.



4. We can now coalesce either ae&r_1 or coalesce d&r_1. Let us do the former.



5. We cannot now coalesce r_1ae&d because the move is *constrained:* the nodes r_1ae and d interfere. We must *simplify* d.

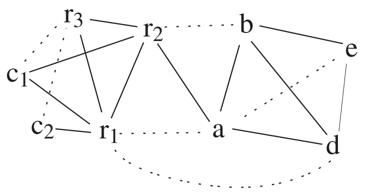


6. Now we have reached a graph with only precolored nodes, so we pop nodes from the stack and assign colors to them. First we pick d, which can be assigned color r_3. Nodes a, b, e have already been assigned colors by coalescing. But node c, which was a *potential spill,* turns into an *actual spill* when it is popped from the stack, since no color can be found for it.

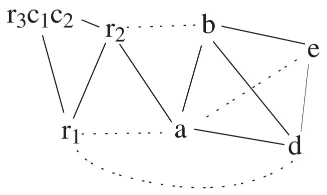
7. Since there was spilling in this round, we must rewrite the program to include spill instructions. For each use (or definition) of c, we make up a new temporary, and fetch (or store) it immediately afterward (or beforehand).

```
enter:  $c_1 \leftarrow r_3$
        $M[c_{\text{loc}}] \leftarrow c_1$
        $a \leftarrow r_1$
        $b \leftarrow r_2$
        $d \leftarrow 0$
        $e \leftarrow a$
loop:   $d \leftarrow d + b$
        $e \leftarrow e - 1$
        if $e > 0$ goto loop
        $r_1 \leftarrow d$
        $c_2 \leftarrow M[c_{\text{loc}}]$
        $r_3 \leftarrow c_2$
        return
```

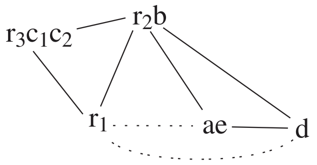
8. Now we build a new inteference graph:



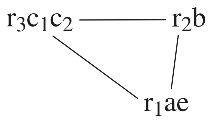
9. Graph coloring proceeds as follows. We can immediately coalesce c_1&r_3 and then c_2&r_3.



10. Then, as before, we can coalesce a&e and then b&r_2.



11. As before, we can coalesce ae&r_1 and then simplify d.



12. Now we start popping from the stack: we select color r_3 for d, and this was the only node on the stack – all other nodes were coalesced or precolored. The coloring is shown at right.

Node	Color
a	r_1
b	r_2
c	r_3
d	r_3
e	r_1

13. Now we can rewrite the program using the register assignment.

```
enter:  $r_3 \leftarrow r_3$
        $M[c_{\text{loc}}] \leftarrow r_3$
        $r_1 \leftarrow r_1$
        $r_2 \leftarrow r_2$
        $r_3 \leftarrow 0$
        $r_1 \leftarrow r_1$
loop:   $r_3 \leftarrow r_3 + r_2$
        $r_1 \leftarrow r_1 - 1$
        if $r_1 > 0$ goto loop
        $r_1 \leftarrow r_3$
        $r_3 \leftarrow M[c_{\text{loc}}]$
        $r_3 \leftarrow r_3$
        return
```

14. Finally, we can delete any move instruction whose source and destination are the same; these are the result of coalescing.

```
enter:  $M[c_{loc}] \leftarrow r_3$
        $r_3 \leftarrow 0$
loop:   $r_3 \leftarrow r_3 + r_2$
        $r_1 \leftarrow r_1 - 1$
        if $r_1 > 0$ goto loop
        $r_1 \leftarrow r_3$
        $r_3 \leftarrow M[c_{loc}]$
        return
```

The final program has only one uncoalesced move instruction.

11.4 GRAPH COLORING IMPLEMENTATION

The graph coloring algorithm needs to query the inteference-graph data structure frequently. There are two kinds of queries:

1. Get all the nodes adjacent to node X; and
2. Tell if X and Y are adjacent.

An adjacency list (per node) can answer query 1 quickly, but not query 2 if the lists are long. A two-dimensional bit matrix indexed by node numbers can answer query 2 quickly, but not query 1. Therefore, we need both data structures to (redundantly) represent the interference graph. If the graph is very sparse, a hash table of integer pairs may be better than a bit matrix.

The adjacency lists of machine registers (precolored nodes) can be very large; because they're used in standard calling conventions they interfere with any temporaries that happen to be live near *any* of the procedure-calls in the program. But we don't need to represent the adjacency list for a precolored node, because adjacency lists are used only in the *select* phase (which does not apply to precolored nodes) and in the Briggs coalescing test. To save space and time, we do not explicitly represent the adjacency lists of the machine registers. We coalesce an ordinary node a with a machine register r using the George coalescing test, which needs the adjacency list of a but not of r.

To test whether two ordinary (non-precolored) nodes can be coalesced, the algorithm shown here uses the Briggs coalescing test.

Associated with each move-related node is a count of the moves it is involved in. This count is easy to maintain and is used to test if a node is no longer move-related. Associated with all nodes is a count of the number of neighbors currently in the graph. This is used to determine whether a node is of significant degree during coalescing, and whether a node can be removed from the graph during simplification.

It is important to be able to quickly perform each *simplify* step (removing a low-degree non-move-related node), each *coalesce* step, and each *freeze* step. To do this, we maintain four work-lists:

- Low-degree non-move-related nodes *(simplifyWorklist)*;
- Move instructions that might be coalesceable *(worklistMoves)*;
- Low-degree move-related nodes *(freezeWorklist)*;
- High-degree nodes *(spillWorklist)*.

Using these work-lists, we avoid quadratic time blowup in finding coalesceable nodes.

MOVE-WORK-LIST MANAGEMENT

When a node x changes from significant to low degree, the moves associated with its neighbors must be added to the move work-list. Moves that were blocked with too many significant neighbors might now be enabled for coalescing. Moves are added to the move work-list in only a few places:

- During simplify the degree of a node x might make the transition as a result of removing another node. Moves associated with neighbors of x are added to the *worklistMoves*.
- When u and v are coalesced, there may be a node x that interferes with both u and v. The degree of x is decremented as it now interferes with the single coalesced node. Moves associated with neighbors of x are added. If x is move-related, then moves associated with x itself are also added as both u and v may have been significant degree nodes.
- When u is coalesced into v, moves associated with u are added to the move work-list. This will catch other moves from u to v.

DATA STRUCTURES

The algorithm maintains these data structures to keep track of graph nodes and move edges:

Node work-lists, sets, and stacks. The following lists and sets are always *mutually disjoint* and every node is always in exactly one of the sets or lists.

precolored: machine registers, preassigned a color.
initial: temporary registers, not precolored and not yet processed.
simplifyWorklist: list of low-degree non-move-related nodes.
freezeWorklist: low-degree move-related nodes.

spillWorklist: high-degree nodes.
spilledNodes: nodes marked for spilling during this round; initially empty.
coalescedNodes: registers that have been coalesced; when $u \leftarrow v$ is coalesced, v is added to this set and u put back on some work-list (or vice versa).
coloredNodes: nodes successfully colored.
selectStack: stack containing temporaries removed from the graph.

Since membership in these sets is often tested, the representation of each node should contain an enumeration value telling which set it is in. Since nodes must frequently be added to and removed from these sets, each set can be represented by a doubly linked list of nodes. Initially (on entry to Main), and on exiting RewriteProgram, only the sets *precolored* and *initial* are nonempty.

Move sets. There are five sets of move instructions, and every move is in exactly one of these sets (after Build through the end of Main).

coalescedMoves: moves that have been coalesced.
constrainedMoves: moves whose source and target interfere.
frozenMoves: moves that will no longer be considered for coalescing.
worklistMoves: moves enabled for possible coalescing.
activeMoves: moves not yet ready for coalescing.

Like the node work-lists, the move sets should be implemented as doubly linked lists, with each move containing an enumeration value identifying which set it belongs to.

Other data structures.

adjSet: the set of interference edges (u, v) in the graph. If $(u, v) \in$ adjSet then $(v, u) \in$ adjSet.
adjList: adjacency list representation of the graph; for each non-precolored temporary u, adjList[u] is the set of nodes that interfere with u.
degree: an array containing the current degree of each node.
moveList: a mapping from a node to the list of moves it is associated with.
alias: when a move (u, v) has been coalesced, and v put in coalescedNodes, then alias$(v) = u$.
color: the color chosen by the algorithm for a node; for precolored nodes this is initialized to the given color.

Invariants. After Build, the following invariants always hold:

Degree invariant.

$$
\begin{aligned}
&(u \in \text{simplifyWorklist} \cup \text{freezeWorklist} \cup \text{spillWorklist}) \quad \Rightarrow \\
&\quad \text{degree}(u) = |\text{adjList}(u) \cap (\text{precolored} \cup \text{simplifyWorklist} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \cup \text{freezeWorklist} \cup \text{spillWorklist})|
\end{aligned}
$$

Simplify worklist invariant.

$$
\begin{aligned}
&(u \in \text{simplifyWorklist}) \quad \Rightarrow \\
&\quad \text{degree}(u) < K \wedge \ \text{moveList}[u] \cap (\text{activeMoves} \cup \text{worklistMoves}) = \{\}
\end{aligned}
$$

Freeze worklist invariant.

$$
\begin{aligned}
&(u \in \text{freezeWorklist}) \quad \Rightarrow \\
&\quad \text{degree}(u) < K \wedge \ \text{moveList}[u] \cap (\text{activeMoves} \cup \text{worklistMoves}) \neq \{\}
\end{aligned}
$$

Spill worklist invariant.

$$(u \in \text{spillWorklist}) \quad \Rightarrow \quad \text{degree}(u) \geq K$$

PROGRAM CODE

The algorithm is invoked using the procedure `Main`, which loops (via tail recursion) until no spills are generated.

```
procedure Main()
    LivenessAnalysis()
    Build()
    MakeWorklist()
    repeat
        if simplifyWorklist $\neq$ {} then Simplify()
        else if worklistMoves $\neq$ {} then Coalesce()
        else if freezeWorklist $\neq$ {} then Freeze()
        else if spillWorklist $\neq$ {} then SelectSpill()
    until simplifyWorklist = {} $\wedge$ worklistMoves = {}
         $\wedge$ freezeWorklist = {} $\wedge$ spillWorklist = {}
    AssignColors()
    if spilledNodes $\neq$ {} then
        RewriteProgram(spilledNodes)
        Main()
```

If `AssignColors` spills, then `RewriteProgram` allocates memory locations for the spilled temporaries and inserts store and fetch instructions to access them. These stores and fetches are to newly created temporaries (with tiny live ranges), so the main loop must be performed on the altered graph.

```
procedure Build ()
    forall $b \in$ blocks in program
        let live = liveOut($b$)
        forall $I \in$ instructions($b$) in reverse order
            if isMoveInstruction($I$) then
                live $\leftarrow$ live$\backslash$use($I$)
                forall $n \in$ def($I$) $\cup$ use($I$)
                    moveList[$n$] $\leftarrow$ moveList[$n$] $\cup \{I\}$
                worklistMoves $\leftarrow$ worklistMoves $\cup \{I\}$
            live $\leftarrow$ live $\cup$ def($I$)
            forall $d \in$ def($I$)
                forall $l \in$ live
                    AddEdge($l, d$)
            live $\leftarrow$ use($I$) $\cup$ (live$\backslash$def($I$))
```

Procedure `Build` constructs the interference graph (and bit matrix) using the results of static liveness analysis, and also initializes the `worklistMoves` to contain all the moves in the program.

```
procedure AddEdge($u, v$)
    if (($u, v$) $\notin$ adjSet) $\wedge$ ($u \neq v$) then
        adjSet $\leftarrow$ adjSet $\cup \{(u, v), (v, u)\}$
        if $u \notin$ precolored then
            adjList[$u$] $\leftarrow$ adjList[$u$] $\cup \{v\}$
            degree[$u$] $\leftarrow$ degree[$u$] + 1
        if $v \notin$ precolored then
            adjList[$v$] $\leftarrow$ adjList[$v$] $\cup \{u\}$
            degree[$v$] $\leftarrow$ degree[$v$] + 1

procedure MakeWorklist()
    forall $n \in$ initial
        initial $\leftarrow$ initial $\backslash \{n\}$
        if degree[$n$] $\geq K$ then
            spillWorklist $\leftarrow$ spillWorklist $\cup$ {n}
        else if MoveRelated($n$) then
            freezeWorklist $\leftarrow$ freezeWorklist $\cup$ {n}
        else
            simplifyWorklist $\leftarrow$ simplifyWorklist $\cup$ {n}

function Adjacent($n$)
    adjList[$n$] $\backslash$ (selectStack $\cup$ coalescedNodes)
```

```
function NodeMoves ($n$)
    moveList[$n$] $\cap$ (activeMoves $\cup$ worklistMoves)

function MoveRelated($n$)
    NodeMoves($n$) $\neq$ {}

procedure Simplify()
    let $n \in$ simplifyWorklist
    simplifyWorklist $\leftarrow$ simplifyWorklist \ {$n$}
    push($n$, selectStack)
    forall m $\in$ Adjacent($n$)
        DecrementDegree($m$)
```

Removing a node from the graph involves decrementing the degree of its *current* neighbors. If the `degree` of a neighbor is already less than $K-1$ then the neighbor must be move-related, and is not added to the `simplifyWork-list`. When the degree of a neighbor transitions from K to $K-1$, moves associated with *its* neighbors may be enabled.

```
procedure DecrementDegree($m$)
    let $d$ = degree[$m$]
    degree[$m$] $\leftarrow$ d-1
    if $d = K$ then
        EnableMoves({$m$} $\cup$ Adjacent($m$))
        spillWorklist $\leftarrow$ spillWorklist \ {$m$}
        if MoveRelated($m$) then
            freezeWorklist $\leftarrow$ freezeWorklist $\cup$ {$m$}
        else
            simplifyWorklist $\leftarrow$ simplifyWorklist $\cup$ {$m$}

procedure EnableMoves(nodes)
    forall $n \in$ nodes
        forall $m \in$ NodeMoves($n$)
            if $m \in$ activeMoves then
                activeMoves $\leftarrow$ activeMoves \ {$m$}
                worklistMoves $\leftarrow$ worklistMoves $\cup$ {$m$}
```

Only moves in the `worklistMoves` are considered in the coalesce phase. When a move is coalesced, it may no longer be move-related and can be added to the simplify work-list by the procedure `AddWorkList`. `OK` implements the heuristic used for coalescing a precolored register. `Conservative` implements the conservative coalescing heuristic.

```
procedure Coalesce()
    let $m_{(=copy(x,y))} \in$ worklistMoves
    $x \leftarrow$ GetAlias($x$)
    $y \leftarrow$ GetAlias($y$)
    if $y \in$ precolored then
        let $(u, v) = (y, x)$
    else
        let $(u, v) = (x, y)$
    worklistMoves $\leftarrow$ worklistMoves $\setminus \{m\}$
    if ($u = v$) then
        coalescedMoves $\leftarrow$ coalescedMoves $\cup \{m\}$
        AddWorkList($u$)
    else if $v \in$ precolored $\vee$ $(u, v) \in$ adjSet then
        constrainedMoves $\leftarrow$ constrainedMoves $\cup \{m\}$
        AddWorkList($u$)
        AddWorkList($v$)
    else if $u \in$ precolored $\wedge$ ($\forall t \in$ Adjacent($v$), OK($t$, $u$))
                $\vee$    $u \notin$ precolored $\wedge$
                          Conservative(Adjacent($u$) $\cup$ Adjacent($v$)) then
        coalescedMoves $\leftarrow$ coalescedMoves $\cup \{m\}$
        Combine($u$,$v$)
        AddWorkList($u$)
    else
        activeMoves $\leftarrow$ activeMoves $\cup \{m\}$

procedure AddWorkList(u)
    if ($u \notin$ precolored $\wedge$ not(MoveRelated($u$)) $\wedge$ degree[$u$] $<$ K) then
        freezeWorklist $\leftarrow$ freezeWorklist $\setminus \{u\}$
        simplifyWorklist $\leftarrow$ simplifyWorklist $\cup \{u\}$

function OK($t$,$r$)
    degree[$t$] $< K \vee t \in$ precolored $\vee$ $(t, r) \in$ adjSet

function Conservative(nodes)
    let $k = 0$
    forall $n \in$ nodes
        if degree[$n$] $\geq K$ then $k \leftarrow k + 1$
    return ($k < K$)

function GetAlias ($n$)
    if $n \in$ coalescedNodes then
        GetAlias(alias[$n$])
    else $n$
```

```
procedure Combine(u,v)
    if $v \in$ freezeWorklist then
        freezeWorklist $\leftarrow$ freezeWorklist $\setminus \{v\}$
    else
        spillWorklist $\leftarrow$ spillWorklist $\setminus \{v\}$
    coalescedNodes $\leftarrow$ coalescedNodes $\cup \{v\}$
    alias[$v$] $\leftarrow u$
    nodeMoves[$u$] $\leftarrow$ nodeMoves[$u$] $\cup$ nodeMoves[$v$]
    forall $t \in$ Adjacent($v$)
        AddEdge($t$,$u$)
        DecrementDegree($t$)
    if degree[$u$] $\geq K \wedge u \in$ freezeWorkList
        freezeWorkList $\leftarrow$ freezeWorkList $\setminus \{u\}$
        spillWorkList $\leftarrow$ spillWorkList $\cup \{u\}$

procedure Freeze()
    let $u \in$ freezeWorklist
    freezeWorklist $\leftarrow$ freezeWorklist $\setminus \{u\}$
    simplifyWorklist $\leftarrow$ simplifyWorklist $\cup \{u\}$
    FreezeMoves($u$)

procedure FreezeMoves($u$)
    forall $m_{(=copy(x,y))} \in$ NodeMoves($u$)
        if GetAlias($y$)=GetAlias($u$) then
            $v \leftarrow$ GetAlias($x$)
        else
            $v \leftarrow$ GetAlias($y$)
        activeMoves $\leftarrow$ activeMoves $\setminus \{m\}$
        frozenMoves $\leftarrow$ frozenMoves $\cup \{m\}$
        if NodeMoves($v$) $= \{\} \ \wedge$ degree[$v$] $< K$ then
            freezeWorklist $\leftarrow$ freezeWorklist $\setminus \{v\}$
            simplifyWorklist $\leftarrow$ simplifyWorklist $\cup \{v\}$

procedure SelectSpill()
    let $m \in$ spillWorklist *selected using favorite heuristic*
        *Note: avoid choosing nodes that are the tiny live ranges*
        *resulting from the fetches of previously spilled registers*
    spillWorklist $\leftarrow$ spillWorklist $\setminus \{m\}$
    simplifyWorklist $\leftarrow$ simplifyWorklist $\cup \{m\}$
    FreezeMoves($m$)
```

```
procedure AssignColors()
    while SelectStack not empty
        let $n$ = pop(SelectStack)
        okColors $\leftarrow \{0, \ldots, \text{K-1}\}$
        forall $w \in$ adjList[$n$]
            if GetAlias($w$) $\in$ (coloredNodes $\cup$ precolored) then
                okColors $\leftarrow$ okColors $\setminus$ {color[GetAlias($w$)]}
        if okColors = {}  then
            spilledNodes $\leftarrow$ spilledNodes $\cup \{n\}$
        else
            coloredNodes $\leftarrow$ coloredNodes $\cup \{n\}$
            let $c \in$ okColors
            color[$n$] $\leftarrow c$
    forall $n \in$ coalescedNodes
        color[$n$] $\leftarrow$ color[GetAlias($n$)]

procedure RewriteProgram()
    Allocate memory locations for each $v \in$ spilledNodes,
    Create a new temporary $v_i$ for each definition and each use,
    In the program (instructions), insert a store after each
    definition of a $v_i$, a fetch before each use of a $v_i$.
    Put all the $v_i$ into a set newTemps.
    spilledNodes $\leftarrow$ {}
    initial $\leftarrow$ coloredNodes $\cup$ coalescedNodes $\cup$ newTemps
    coloredNodes $\leftarrow$ {}
    coalescedNodes $\leftarrow$ {}
```

I show a variant of the algorithm in which all coalesces are discarded if the program must be rewritten to incorporate spill fetches and stores. For a faster algorithm, keep all the coalesces found before the first call to `Select-Spill` and rewrite the program to eliminate the coalesced move instructions and temporaries.

In principle, a heuristic could be used to select the freeze node; the `Freeze` shown above picks an arbitrary node from the freeze work-list. But freezes are not common, and a selection heuristic is unlikely to make a significant difference.

```
function SimpleAlloc($t$)
    for each nontrivial tile $u$ that is a child of $t$
        SimpleAlloc($u$)
    for each nontrivial tile $u$ that is a child of $t$
        $n \leftarrow n - 1$
    $n \leftarrow n + 1$
    assign $r_n$ to hold the value at the root of $t$
```

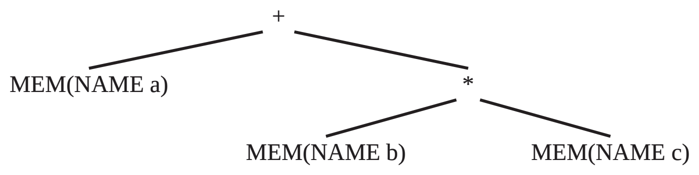
ALGORITHM 11.9. Simple register allocation on trees.

11.5 REGISTER ALLOCATION FOR TREES

Register allocation for expression trees is much simpler than for arbitrary flow graphs. We do not need global dataflow analysis or interference graphs. Suppose we have a tiled tree such as in Figure 9.2a. This tree has two *trivial* tiles, the TEMP nodes *fp* and *i*, which we assume are already in registers r_{fp} and r_i. We wish to label the roots of the nontrivial tiles (the ones corresponding to instructions, i.e., 2, 4, 5, 6, 8) with registers from the list $r_1, r_2, \ldots, r_k$.

Algorithm 11.9 traverses the tree in postorder, assigning a register to the root of each tile. With n initialized to zero, this algorithm applied to the root (tile 9) produces the allocation {tile2 $\mapsto r_1$, tile4 $\mapsto r_2$, tile5 $\mapsto r_2$, tile6 $\mapsto r_1$, tile8 $\mapsto r_2$, tile9 $\mapsto r_1$}. The algorithm can be combined with Maximal Munch, since both algorithms are doing the same bottom-up traversal.

But this algorithm will not always lead to an optimal allocation. Consider the following tree, where each tile is shown as a single node:



The SimpleAlloc function will use three registers for this expression (as shown at left below), but by reordering the instructions we can do the computation

```
function Label($t$)
  for each tile $u$ that is a child of $t$
     Label($u$)
  if $t$ is trivial
     then $need[t] \leftarrow 0$
  else if $t$ has two children, $u_{\text{left}}$ and $u_{\text{right}}$
     then if $need[u_{\text{left}}] = need[u_{\text{right}}]$
             then $need[t] \leftarrow 1 + need[u_{\text{left}}]$
             else  $need[t] \leftarrow \max(1, need[u_{\text{left}}], need[u_{\text{right}}])$
  else if $t$ has one child, $u$
     then $need[t] \leftarrow \max(1, need[u])$
  else if $t$ has no children
     then $need[t] \leftarrow 1$
```

ALGORITHM 11.10. Sethi-Ullman labeling algorithm.

using only two registers (as shown at right):

$r_1 \leftarrow M[a]$	$r_1 \leftarrow M[b]$
$r_2 \leftarrow M[b]$	$r_2 \leftarrow M[c]$
$r_3 \leftarrow M[c]$	$r_1 \leftarrow r_1 \times r_2$
$r_2 \leftarrow r_2 \times r_3$	$r_2 \leftarrow M[a]$
$r_1 \leftarrow r_1 + r_2$	$r_1 \leftarrow r_2 + r_1$

Using dynamic programming, we can find the optimal ordering for the instructions. The idea is to label each tile with the number of registers it needs during its evaluation. Suppose a tile t has two nontrivial children u_{left} and u_{right} that require n and m registers, respectively, for their evaluation. If we evaluate u_{left} first, and hold its result in one register while we evaluate u_{right}, then we have needed $\max(n, 1+m)$ registers for the whole expression rooted at t. Conversely, if we evaluate u_{right} first, then we need $\max(1+n, m)$ registers. Clearly, if $n > m$ we should evaluate u_{left} first, and if $n < m$ we should evaluate u_{right} first. If $n = m$ we will need $n + 1$ registers no matter which subexpression is evaluated first.

Algorithm 11.10 labels each tile t with $need[t]$, the number of registers needed to evaluate the subtree rooted at t. It can be generalized to handle tiles with more than two children. Maximal Munch should identify – but not emit

```
function SethiUllman($t$)
  if $t$ has two children, $u_{\text{left}}$ and $u_{\text{right}}$
    if $need[u_{\text{left}}] \geq K \;\wedge need[u_{\text{right}}] \geq K$
        SethiUllman($t_{\text{right}}$)
        $n \leftarrow n - 1$
        spill: emit instruction to store $reg[t_{\text{right}}]$
        SethiUllman($t_{\text{left}}$)
        unspill: $reg[t_{\text{right}}] \leftarrow$ “$r_{n+1}$”; emit instruction to fetch $reg[t_{\text{right}}]$
    else if $need[u_{\text{left}}] \geq need[u_{\text{right}}]$
        SethiUllman($t_{\text{left}}$)
        SethiUllman($t_{\text{right}}$)
        $n \leftarrow n - 1$
    else  $need[u_{\text{left}}] < need[u_{\text{right}}]$
        SethiUllman($t_{\text{right}}$)
        SethiUllman($t_{\text{left}}$)
        $n \leftarrow n - 1$
    $reg[t] \leftarrow$ “$r_n$”
    emit OPER($instruction[t]$,  $reg[t]$, [ $reg[t_{\text{left}}]$,  $reg[t_{\text{right}}]$])
  else if $t$ has one child, $u$
    SethiUllman($u$)
    $reg[t] \leftarrow$ “$r_n$”
    emit OPER($instruction[t]$,  $reg[t]$,  [$reg[u]$])
  else if $t$ is nontrivial but has no children
    $n \leftarrow n + 1$
    $reg[t] \leftarrow$ “$r_n$”
    emit OPER($instruction[t]$,  $reg[t]$,  [ ])
  else if $t$ is a trivial node TEMP($r_i$)
    $reg[t] \leftarrow$ “$r_i$”
```

ALGORITHM 11.11. Sethi-Ullman register allocation for trees.

– the tiles, simultaneously with the labeling of Algorithm 11.10. The next pass emits `Assem` instructions for the tiles; wherever a tile has more than one child, the subtrees must be emitted in decreasing order of register *need*.

Algorithm 11.10 can profitably be used in a compiler that uses graph-coloring register allocation. Emitting the subtrees in decreasing order of *need* will minimize the number of simultaneously live temporaries and reduce the number of spills.

In a compiler without graph-coloring register allocation, Algorithm 11.10 is used as a pre-pass to Algorithm 11.11, which assigns registers as the trees are emitted and also handles spilling cleanly. This takes care of register allocation for the internal nodes of expression trees; allocating registers for explicit TEMPs of the Tree language would have to be done in some other way. In general, such a compiler would keep almost all program variables in the stack frame, so there would not be many of these explicit TEMPs to allocate.

PROGRAM GRAPH COLORING

Implement graph coloring register allocation as two modules: Color, which does just the graph coloring itself, and RegAlloc, which manages spilling and calls upon Color as a subroutine. To keep things simple, do not implement spilling or coalescing; this simplifies the algorithm considerably.

```
/* color.h */
struct COL_result{Temp_map coloring; Temp_tempList spills;};
struct COL_result COL_color(G_graph ig,
                            Temp_map initial,
                            Temp_tempList registers);


/* regalloc.h */
struct RA_result {Temp_map coloring; AS_instrList il;};
struct RA_result RA_regAlloc(F_frame f, AS_instrList il);
```

Given an interference graph, an initial allocation (precoloring) of some temporaries imposed by calling conventions, and a list of colors (registers), color produces an extension of the initial allocation. The resulting allocation assigns all temps used in the flow graph, making use of registers from the registers list.

The initial allocation is the F_tempMap provided by the Frame structure; the registers argument is just the list of all machine registers, F_registers() (see page 267). The registers in the initial allocation can also appear in the registers argument to COL_color(), since it's OK to use them to color other nodes as well.

The result of COL_color is a Temp_map describing the register allocation, along with a list of spills. The result of RegAlloc – if there were no spills – is an identical Temp_map, which can be used in final assembly-code emission as an argument to AS_print.

A better `COL_color` interface would have a `spillCost` argument that specifies the spilling cost of each temporary. This can be just the number of uses and defs, or better yet, uses and defs weighted by occurrence in loops and nested loops. A naive `spillCost` that just returns 1 for every temporary will also work.

A simple implementation of the coloring algorithm without coalescing requires only one work-list: the `simplifyWorklist`, which contains all non-precolored, nonsimplified nodes of degree less than K. Obviously, no `freezeWorklist` is necessary. No `spillWorklist` is necessary either, if we are willing to look through all the nodes in the original graph for a spill candidate every time the `simplifyWorklist` becomes empty.

With only a `simplifyWorklist`, the doubly linked representation is not necessary: this work-list can be implemented as a singly linked list or a stack, since it is never accessed "in the middle."

ADVANCED PROJECT: SPILLING

Implement spilling, so that no matter how many parameters and locals a Tiger program has, you can still compile it.

ADVANCED PROJECT: COALESCING

Implement coalescing, to eliminate practically all the MOVE instructions from the program.

FURTHER READING

Kempe [1879] invented the simplification algorithm that colors graphs by removing vertices of degree $< K$. Chaitin [1982] formulated register allocation as a graph-coloring problem – using Kempe's algorithm to color the graph – and performed copy propagation by (nonconservatively) coalescing non-interfering move-related nodes before coloring the graph. Briggs et al. [1994] improved the algorithm with the idea of optimistic spilling, and also avoided introducing spills by using the conservative coalescing heuristic before coloring the graph. George and Appel [1996] found that there are more opportunities for coalescing if conservative coalescing is done during simplification instead of beforehand, and developed the work-list algorithm presented in this chapter.

Ershov [1958] developed the algorithm for optimal register allocation on expression trees; Sethi and Ullman [1970] generalized this algorithm and showed how it should handle spills.

EXERCISES

11.1 The following program has been compiled for a machine with three registers r_1, r_2, r_3; r_1 and r_2 are (caller-save) argument registers and r_3 is a callee-save register. Construct the interference graph and show the steps of the register allocation process in detail, as on pages 244–248. When you coalesce two nodes, say whether you are using the Briggs or George criterion.

Hint: When two nodes are connected by an interference edge *and* a move edge, you may delete the move edge; this is called *constrain* and is accomplished by the first **else if** clause of procedure *Coalesce*.

```
f :   $c \leftarrow r_3$
      $p \leftarrow r_1$
      if $p = 0$ goto $L_1$
      $r_1 \leftarrow M[p]$
      call $f$              (uses $r_1$, defines $r_1, r_2$)
      $s \leftarrow r_1$
      $r_1 \leftarrow M[p+4]$
      call $f$              (uses $r_1$, defines $r_1, r_2$)
      $t \leftarrow r_1$
      $u \leftarrow s + t$
      goto $L_2$
$L_1$ :  $u \leftarrow 1$
$L_2$ :  $r_1 \leftarrow u$
      $r_3 \leftarrow c$
      return                (uses $r_1, r_3$)
```

11.2 The table below represents a register-interference graph. Nodes 1–6 are precolored (with colors 1–6), and nodes A–H are ordinary (non-precolored). Every pair of precolored nodes interferes, and each ordinary node interferes with nodes where there is an x in the table.

	1	2	3	4	5	6	A	B	C	D	E	F	G	H
A	x	x	x	x	x	x								
B	x		x	x	x	x								
C			x	x	x	x				x	x	x	x	x
D	x		x	x	x				x		x	x	x	x
E	x		x		x	x			x	x		x	x	x
F	x		x	x		x			x	x	x		x	x
G									x	x	x	x		
H	x			x	x	x			x	x	x	x		

The following pairs of nodes are related by MOVE instructions:

$(A, 3)$ $(H, 3)$ $(G, 3)$ $(B, 2)$ $(C, 1)$ $(D, 6)$ $(E, 4)$ $(F, 5)$

Assume that register allocation must be done for an 8-register machine.

a. Ignoring the MOVE instructions, and without using the *coalesce* heuristic, color this graph using *simplify* and *spill*. Record the sequence (stack) of *simplify* and *potential-spill* decisions, show which potential spills become actual spills, and show the coloring that results.

b. Color this graph using coalescing. Record the sequence of *simplify*, *coalesce*, *freeze*, and *spill* decisions. Identify each *coalesce* as Briggs- or George-style. Show how many MOVE instructions remain.

*c. Another coalescing heuristic is *biased coloring*. Instead of using a *conservative coalescing* heuristic during simplification, run the *simplify-spill* part of the algorithm as in part (a), but in the *select* part of the algorithm,

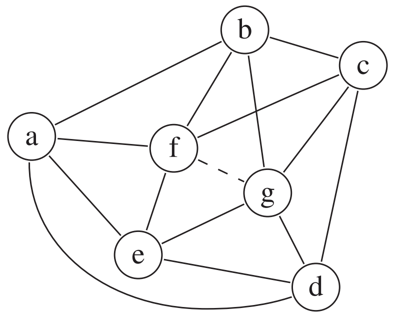
  i. When selecting a color for node X that is move-related to node Y, when a color for Y has already been selected, use the same color if possible (to eliminate the MOVE).

  ii. When selecting a color for node X that is move-related to node Y, when a color for Y has not yet been selected, use a color that is *not* the same as the color of any of Y's neighbors (to increase the chance of heuristic (i) working when Y is colored).

  Conservative coalescing (in the *simplify* phase) has been found more effective than biased coloring, in general; but it might not be on this particular graph. Since the two coalescing algorithms are used in different phases, they can both be used in the same register allocator.

*d. Use both conservative coalescing and biased coloring in allocating registers. Show where biased coloring helps make the right decisions.

11.3 *Conservative coalescing* is so called because it will not introduce any (potential) spills. But can it avoid spills? Consider this graph, where the solid edges represent interferences and the dashed edge represents a MOVE:



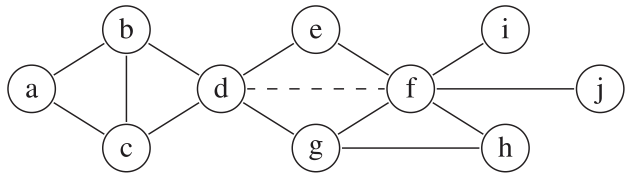
a. 4-color the graph without coalescing. Show the *select*-stack, indicating the order in which you removed nodes. Is there a potential spill? Is there an actual spill?

b. 4-color the graph with conservative coalescing. Did you use the Briggs or George criterion? Is there a potential spill? Is there an actual spill?

11.4 It has been proposed that the conservative coalescing heuristic could be simplified. In testing whether MOVE(a, b) can be coalesced, instead of asking whether the combined node ***ab*** is adjacent to $< K$ nodes of significant degree, we could simply test whether ***ab*** is adjacent to $< K$ nodes of any degree. The theory is that if ***ab*** is adjacent to many low-degree nodes, they will be removed by simplification anyway.

a. Show that this kind of coalescing cannot create any new potential spills.

b. Demonstrate the algorithm on this graph (with $K = 3$):



*c. Show that this test is less effective than standard conservative coalescing. **Hint:** Use the graph of Exercise 11.3, with $K = 4$.

12 Putting It All Together

de-bug: to eliminate errors in or malfunctions of

Webster's Dictionary

Chapters 2–11 have described the fundamental components of a good compiler: a *front end*, which does lexical analysis, parsing, construction of abstract syntax, type-checking, and translation to intermediate code; and a *back end*, which does instruction selection, dataflow analysis, and register allocation.

What lessons have we learned? I hope that the reader has learned about the algorithms used in different components of a compiler and the interfaces used to connect the components. But the author has also learned quite a bit from the exercise.

My goal was to describe a good compiler that is, to use Einstein's phrase, "as simple as possible – but no simpler." I will now discuss the thorny issues that arose in designing Tiger and its compiler.

Nested functions. Tiger has nested functions, requiring some mechanism (such as static links) for implementing access to nonlocal variables. But many programming languages in widespread use – C, C++, Java – do not have nested functions or static links. The Tiger compiler would become simpler without nested functions, for then variables would not escape, and the `FindEscape` phase would be unnecessary. But there are two reasons for explaining how to compile nonlocal variables. First, there are programming languages where nested functions are extremely useful – these are the *functional* languages described in Chapter 15. And second, escaping variables and the mechanisms necessary to handle them are also found in languages where addresses can be taken (such as C) or with call-by-reference (such as C++).

Structured l-values. Tiger has no record or array variables, as C, C++, and Pascal do. Instead, all record and array values are really just pointers to heap-allocated data. This omission is really just to keep the compiler simple; implementing structured l-values requires some care but not too many new insights.

Tree intermediate representation. The `Tree` language has a fundamental flaw: it does not describe procedure entry and exit. These are handled by opaque procedures inside the `Frame` module that generate `Tree` code. This means that a program translated to `Tree`s using, for example, the `PentiumFrame` version of `Frame` will be different from the same program translated using `SparcFrame` – the `Tree` representation is not completely machine independent.

Also, there is not enough information in the trees themselves to simulate the execution of an entire program, since the *view shift* (page 137) is partly done implicitly by procedure prologues and epilogues that are not represented as `Tree`s. Consequently, there is not enough information to do whole-program optimization (across function boundaries).

The `Tree` representation is useful as a *low-level* intermediate representation, useful for instruction selection and intraprocedural optimization. A *high-level* intermediate representation would preserve more of the source-program semantics, including the notions of nested functions, nonlocal variables, record creation (as distinguished from an opaque external function call), and so on. Such a representation would be more tied to a particular family of source languages than the general-purpose `Tree` language is.

Register allocation. Graph-coloring register allocation is widely used in real compilers, but does it belong in a compiler that is supposed to be "as simple as possible"? After all, it requires the use of global dataflow (liveness) analysis, construction of interference graphs, and so on. This makes the back end of the compiler significantly bigger.

It is instructive to consider what the Tiger compiler would be like without it. We could keep all local variables in the stack frame (as we do now for variables that escape), fetching them into temporaries only when they are used as operands of instructions. The redundant loads within a single basic block can be eliminated by a simple intrablock liveness analysis. Internal nodes of `Tree` expressions could be assigned registers using Algorithms 11.10 and 11.9. But other parts of the compiler would become much uglier: The TEMPs introduced in canonicalizing the trees (eliminating ESEQs) would have to be dealt with in an ad hoc way, by augmenting the `Tree` language with an operator that provides explicit scope for temporary variables; the `Frame` interface, which mentions registers in many places, would now have to deal with them in more complicated ways. To be able to create arbitrarily many `temps` and `moves`, and rely on the register allocator to clean them up, greatly simplifies procedure calling sequences and code generation.

PROGRAM PROCEDURE ENTRY/EXIT

Implement the rest of the `Frame` module, which contains all the machine-dependent parts of the compiler: register sets, calling sequences, activation record (frame) layout.

Program 12.1 shows `frame.h`. Most of this interface has been described elsewhere. What remains is:

registers A list of all the register names on the machine, which can be used as "colors" for register allocation.

tempMap For each machine register, the `Frame` module maintains a particular `Temp_temp` that serves as the "precolored temporary" that stands for the register. These temps appear in the `Assem` instructions generated from CALL nodes, in procedure entry sequences generated by `procEntryExit1`, and so on. The `tempMap` tells the "color" of each of these precolored temps.

procEntryExit1 For each incoming register parameter, move it to the place from which it is seen from within the function. This could be a frame location (for escaping parameters) or a fresh temporary. One good way to handle this is for `newFrame` to create a sequence of `T_MOVE` statements as it creates all the formal parameter "accesses." `newFrame` can put this into the `frame` data structure, and `procEntryExit1` can just concatenate it onto the procedure `body`.

Also concatenated to the body are statements for saving and restoring of callee-save registers (including the return-address register). If your register allocator does not implement spilling, all the callee-save (and return-address) registers should be written to the frame at the beginning of the procedure body and fetched back afterward. Therefore, `procEntryExit1` should call `allocLocal` for each register to be saved, and generate `T_MOVE` instructions to save and restore the registers. With luck, saving and restoring the callee-save registers will give the register allocator enough headroom to work with, so that some nontrivial programs can be compiled. Of course, some programs just cannot be compiled without spilling.

If your register allocator implements spilling, then the callee-save registers should not always be written to the frame. Instead, if the register allocator needs the space, it may choose to spill only some of the callee-save registers.

```
/* frame.h */
typedef struct F_frame_ *F_frame;
typedef struct F_access_ *F_access;
typedef struct F_accessList_ *F_accessList;
struct F_accessList_ {F_access head; F_accessList tail;};
F_accessList F_AccessList(F_access head, F_accessList tail);

typedef struct F_frag_ *F_frag;
struct F_frag_ {
    enum {F_stringFrag, F_procFrag} kind;
    union {struct {Temp_label label; string str;} stringg;
            struct {T_stm body; F_frame frame;} proc;
    } u;
};
F_frag F_StringFrag(Temp_label label, string str);
F_frag F_ProcFrag(T_stm body, F_frame frame);

typedef struct F_fragList_ *F_fragList;
struct F_fragList_ {F_frag head; F_fragList tail;};
F_fragList F_FragList(F_frag head, F_fragList tail);

Temp_map F_tempMap;
Temp_tempList F_registers(void);
string F_getlabel(F_frame frame);
T_exp F_Exp(F_access acc, T_exp framePtr);
F_access F_allocLocal(F_frame f, bool escape);   (see p. 139)
F_accessList F_formals(F_frame f);    (p. 137)
Temp_label F_name(F_frame f);         (p. 136)
extern const int F_wordSize;   (p. 159)
Temp_temp F_FP(void);          (p. 159)
Temp_temp F_SP(void);
Temp_temp F_ZERO(void);
Temp_temp F_RA(void);
Temp_temp F_RV(void);          (p. 172)
F_frame F_newFrame(Temp_label name, U_boolList formals);   (p. 136)
T_exp F_externalCall(string s, T_expList args);     (p. 168)
F_frag F_string (Temp_label lab, string str);       (p. 269)
F_frag F_newProcFrag(T_stm body, F_frame frame);
T_stm F_procEntryExit1(F_frame frame, T_stm stm);   (p. 267)
AS_instrList F_procEntryExit2(AS_instrList body);   (p. 215)
AS_proc F_procEntryExit3(F_frame frame, AS_instrList body);

/* codegen.h */
AS_instrList F_codegen(F_frame f, T_stmList stmList);   (p. 212)
```

PROGRAM 12.1. Interface `frame.h`.

But "precolored" temporaries are never spilled; so `procEntryExit1` should make up new temporaries for each callee-save (and return-address) register. On entry, it should move all these registers to their new temporary locations, and on exit, it should move them back. Of course, these moves (for nonspilled registers) will be eliminated by register coalescing, so they cost nothing.

procEntryExit3 Creates the procedure prologue and epilogue assembly language. First (for some machines) it calculates the size of the *outgoing parameter space* in the frame. This is equal to the maximum number of outgoing parameters of any CALL instruction in the procedure body. Unfortunately, after conversion to `Assem` trees the procedure calls have been separated from their arguments, so the outgoing parameters are not obvious. Either `procEntryExit2` should scan the body and record this information in some new component of the `frame` type, or `procEntryExit3` should use the maximum legal value.

Once this is known, the assembly language for procedure entry, stack-pointer adjustment, and procedure exit can be put together; these are the `prologue` and `epilogue`.

string A string literal in Tiger, translated into a `F_StringFrag` fragment, must eventually be translated into machine-dependent assembly language that reserves and initializes a block of memory. The `F_string` function returns a string containing the assembly-language instructions required to define and initialize a string literal. For example,

```
F_string(Temp_namedlabel("L3"),"hello")
```

would yield "`L3: .ascii "hello"\n`" in a typical assembly language. The `Translate` module might make a `F_StringFrag(L3,hello)` (page 172); the `Main` module (see below) would process this fragment by calling `F_string`.

PROGRAM MAKING IT WORK

Make your compiler generate working code that runs.

The file `$TIGER/chap12/runtime.c` is a C-language file containing several external functions useful to your Tiger program. These are generally reached by `externalCall` from code generated by your compiler. You may modify this as necessary.

Write a module `main.c` that calls on all the other modules to produce an assembly language file `prog.s` for each input program `prog.tig`. This assembly language program should be assembled (producing `prog.o`) and linked with `runtime.o` to produce an executable file.

Programming projects

After your Tiger compiler is done, here are some ideas for further work:

12.1 Write a garbage collector (in C) for your Tiger compiler. You will need to make some modifications to the compiler itself to add descriptors to records and stack frames (see Chapter 13).

12.2 Implement first-class function values in Tiger, so that functions can be passed as arguments and returned as results (see Chapter 15).

12.3 Make the Tiger language object-oriented, so that instead of records there are objects with methods. Make a compiler for this object-oriented Tiger (see Chapter 14).

12.4 Implement dataflow analyses such as *reaching definitions* and *available expressions* and use them to implement some of the optimizations discussed in Chapter 17.

12.5 Figure out other approaches to improving the assembly-language generated by your compiler. Discuss; perhaps implement.

12.6 Implement instruction scheduling to fill branch-delay and load-delay slots in the assembly language. Or discuss how such a module could be integrated into the existing compiler; what interfaces would have to change, and in what ways?

12.7 Implement "software pipelining" (instruction scheduling around loop iterations) in your compiler.

12.8 Analyze how adequate the Tiger language itself would be for writing a compiler. What are the smallest possible additions/changes that would make it a much more useful language?

12.9 In the Tiger language, some record types are recursive and *must* be implemented as pointers; others are not recursive and could be implemented without pointers. Modify your compiler to take advantage of this by keeping nonrecursive, nonescaping records in the stack frame instead of on the heap.

12.10 Similarly, some arrays have bounds known at compile time, are not recursive, and are not assigned to other array variables. Modify your compiler so that these arrays are implemented right in the stack frame.

12.11 Implement inline expansion of functions (see Section 15.4).

12.12 Suppose an ordinary Tiger program were to run on a parallel machine (a multiprocessor)? How could the compiler automatically make a parallel program out of the original sequential one? Research the approaches.