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 `Trees` 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 136) is partly done implicitly by procedure prologues and epilogues that are not represented as `Trees`. 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

```
signature FRAME =
sig
  type register = string
  val RV: Temp.temp       (see p. 168)
  val FP: Temp.temp       (p. 155)
  val registers: register list
  val tempMap: register Temp.Table.table
  val wordSize: int       (p. 155)
  val externalCall: string * Tree.exp list -> Tree.exp    (p. 165)
  type frame    (p. 134)
  val newFrame: {name: Temp.label, formals: int} -> frame * int list (p. 135)
  val formals: frame -> access list     (p. 135)
  val name: frame -> Temp.label         (p. 134)
  val allocLocal: frame -> int          (p. 137)
  val string: Tree.label * string -> string   (p. 262)

  val procEntryExit1: frame * Tree.stm -> Tree.stm     (p. 261)
  val procEntryExit2: frame * Assem.instr list -> Assem.instr list  (p. 208)
  val procEntryExit3: frame * Assem.instr list ->      (p. 261)
               {prolog: string, body: Assem.instr list, epilog: string}

   datatype frag = PROC of  body: Tree.stm, frame: frame   (p. 169)
                 | STRING of Temp.label * string
end
```

PROGRAM 12.1. Signature FRAME.

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 the `FRAME` signature. 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 `Tree.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 `Tree.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. 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 `STRING` fragment, must eventually be translated into machine-dependent assembly language that reserves and initializes a block of memory. The `Frame.string` function returns a string containing the assembly-language instructions required to define and initialize a string literal. For example,

```
Frame.string(Temp.namedLabel("L3"),"hello")
```

would yield "`L3:  .ascii  "hello"\n`" in a typical assembly language. The `Translate` module might make a `STRING(L3,hello)` fragment (page 169); the `Main` module (see below) would process this fragment by calling `Frame.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` 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.