MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1995

Anatomy of Compiled Code

Consider the following compiled code:

```
1.   (assign val (op make-compiled-procedure) (label entry17) (reg env))
2.   (goto (label after-lambda16))
3.  entry17
4.   (assign env (op compiled-procedure-env) (reg proc)))
5.   (assign env (op extend-binding-environment) (const (x y)) (reg argl) (reg env))
6.   (assign proc (op lookup-variable-value) (const *) (reg env))
7.   (assign val (op lookup-variable-value) (const y) (reg env))
8.   (assign argl (op list) (reg val))
9.   (assign val (op lookup-variable-value) (const x) (reg env))
10.  (assign argl (op cons) (reg val) (reg argl))
11.  (assign val (const 2))
12.  (assign argl (op cons) (reg val) (reg argl))
13.   <apply-dispatch>
14. after-lambda16
15.  (perform (op define-variable!) (const f) (reg val) (reg env))
16.  (assign val (const (the-unspecified-value)))
17.  (goto (reg continue))
```

Compilation is like translation: Given some Scheme code, how do we say the same thing in register machine language? The compiler is simply a box that translates from Scheme to register machine language; it doesn't actually run either code. The result of the compiler is a translation of the Scheme code: a register machine program which, *if* run, should produce the same behavior as running the original Scheme program.

Compiled Scheme code can be read line by line, but this is *extremely* difficult. It is much easier to group register machine code and understand the entire program as a sequence of code blocks, each of which can be understood more easily.

The goal of an evaluation is to get some appropriate value into the `val` register. Much of the register machine code you will see is simply trying to generate that "appropriate" value.

```
1.  (assign val (op make-compiled-procedure) (label entry17) (reg env))
```

This line should make you think `lambda`. It is a register-machine command to create the *procedure object* containing the wrapped-up code (`entry17`) and the environment (`env`). The machine then jumps to `after-lambda16` to bind the name of the procedure to the procedure object in `val`.

Together with line 1, the compiler needs to generate the translation of the lambda's body. This is because the body can't just be stored as Scheme code—if it were, then when some register machine tried to apply the lambda, it would have to run Scheme code. So the compiler has to translate the lambda body, too. For example, the compilation for `(lambda (x) (1+ x))` *generates* code for `(1+ x)`, but executing the resulting register machine code for `(lambda (x) (1+ x))` does *not* execute the code for `(1+ x)`.

Again, when we see a line like 1, we should write `lambda`. To find out `lambda` *of what*, look at the code at the `entry`.

```
3. entry17
4.  (assign env (op compiled-procedure-env) (reg proc))
5.  (assign env (op extend-binding-environment) (const (x y)) (reg argl) (reg env))
```

This code gets executed when, and only when, the procedure object is *applied*. That is, not when evaluating `(lambda (x) (1+ x))`, but when applying `(lambda (x) (1+ x))` to something (say, when evaluating `((lambda (x) (1+ x)) 6)`, which yields 7). The `entry17` code is called by someone else once it is discovered that the procedure object to be applied (the value in the `val` register returned in `after-lambda16`) is a compiled procedure. An invariant that is true when we enter `entry17` is that the procedure object (the value returned by `(op make-compiled-procedure)` ⟨*entry*⟩ ⟨*env*⟩) is in `proc`, and the list of already-evaluated arguments to the procedure are in `argl`.

Lines 4 and 5 extend the Scheme procedure's environment by building a new frame in which the procedure's formal parameters are bound to the arguments. They always occur together; the first line is always *identical* to line 4; the second is identical to 5 *except* that the list of formal parameters varies from one procedure to the next. The formal parameters to the `lambda` can be read off the `extend-binding-environment` command on line 5. In this case, the formal parameters are `x` and `y`.

The remainder of this code block (in this case, up to and including line 13) constitutes the compilation of the body of the procedure. Since this can be arbitrary Scheme code, its compilation can look like the compilation of any legitimate Scheme code. In this example, we need to make a call (lines 13.1 to 13.7) to an internally used procedure `*`.

```
13.1  (test (op primitive-procedure?) (reg proc))
13.2  (branch (label prim-branch11))
13.3  (assign val (op compiled-procedure-entry) (reg proc))
13.4  (goto (reg val))
13.5 prim-branch11
13.6  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
13.7  (goto (reg continue))
```

Note that we have not preserved the state of any registers before this call. In general, we would want to `save` needed registers before the call and `restore` them after the call returns, if those registers are needed for further computation.

One thing that makes compilation easier than interpretation is that we don't have to handle arbitrary code. At any time, you're only compiling one specific piece of Scheme code. Compiler optimization is the art of taking advantage of that knowledge.

The numbers after compiler-generated labels are meaningless except for identification purposes. They may provide hints, or they may be misleading! The compiler has to generate many distinct labels; it uses numbers to ensure that no two labels have the same name.