MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1999

Recitation – Friday, April 30

1. Compilers

Here are some important ideas to understand about compilers.

1. What is a compiler and how does it work?

   A compiler takes in a program in one language and produces a program in another language that does the same thing the original code did.

   One (simple but inefficient) way it can work is to just paste together code from an evaluator for the input language that is written in the output language.

2. Why are some processes iterative and some recursive?

   If all procedures restore all the registers they save before making a call, it will lead to an iterative process.

   If a procedure calls itself while it has saved but not restored some registers, then it will lead to a recursive process.

3. How does a compiler "optimize" the output program in ways that an interpreter cannot?

   The interpreter must both figure out what to do and also do it. This means the interpreter uses instructions to do the figuring out that the compiler does before the program runs.

   Also, the compiler can analyze the code sequences before combining them, so it can "look into the future" in a way that an interpreter cannot. (For example, it can figure out what registers do and don't need saved.)

4. Given some compiled code, figure out what the original scheme expression was.

   We'll do an example of this later on.

2. Optimizing Saves and Restores

The explicit control evaluator always saves and restores the `env` register around the evaluation of the operator and each of the operands. Why?

For each of the following combinations, determine which of the save and restore operations are superfluous and thus could be eliminated by the compiler's "preserving" mechanism:

```
(f 'x 'y)
```

```
((f) 'x 'y)
```

```
(f (g 'x) y)
```

3. Compiling IFs

Here is the code from the explicit control evaluator for dealing with `if`s.

```
ev-if
  (assign unev (reg exp))
  (save unev)
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg unev))
  (goto (label eval-dispatch))
ev-if-decide
  (restore continue)
  (restore env)
  (restore unev)
  (test (op false?) (reg val))
  (branch (label ev-if-consequent))
ev-if-alternative
  (assign exp (op if-alternative) (reg unev))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg unev))
  (goto (label eval-dispatch))
```

A naive compiler would simply collect the used instructions. Compiling `(if a b c)` would yield:

```
1.    (save env)
2.    (save continue)
3.    (assign continue (label after-pred28))
4.    (assign val (op lookup-variable-value) (const a) (reg env))
5.    (goto (reg continue))
6.  after-pred29
7.    (restore continue)
8.    (restore env)
9.    (test (op false?) (reg val))
10.   (branch (label false-branch27))
11. true-branch28
12.   (assign val (op lookup-variable-value) (const b) (reg env))
13.   (goto (reg continue))
14. false-branch27
15.   (assign val (op lookup-variable-value) (const c) (reg env))
16.   (goto (reg continue))
```

By using stack discipline and targetting, we can create much more efficient registration machine code. Basic idea is to keep track of what registers are needed by an expression and what registers are changed by an expression, and combine these to decide what stack manipulations are actually necessary. What lines can be removed from the above compilation to more efficiently handle `if`s?

Compiling `(if a b c)`:

4. Decompiling Code - Short Examples

The following code results from the compilation of `(define x 3)`.

```
  (assign val (const 3))
  (perform (op define-variable!) (const x) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```

The following code results from the compilation of `(- x 2)`. For the evaluation of a combination, look for the procedure to be put into the `proc` register and the `cons`ing up of the argument list in the `argl` register. **Remember that the arguments are evaluated right to left!**

```
  (assign proc (op lookup-variable-value) (const -) (reg env))
  (assign val (const 2))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const x) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch14))
compiled-branch13
  (assign continue (label after-call12))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch14
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call12
  (goto (reg continue))
```

The following code results from the compilation of `(if a b c)`. When you see labels like `true-branch` and `false-branch`, you should think `if`.

```
  (assign val (op lookup-variable-value) (const a) (reg env))
  (test (op false?) (reg val))
  (branch (label false-branch16))
true-branch17
  (assign val (op lookup-variable-value) (const b) (reg env))
  (goto (label after-if15))
false-branch16
  (assign val (op lookup-variable-value) (const c) (reg env))
after-if15
  (goto (reg continue))
```

The following code results from the compilation of `(lambda (x) x)`. Note that the body of the lambda (lines 3-7) are not evaluated. The label is just saved along with the current environment to make the procedure object.

```
1.   (assign val (op make-compiled-procedure) (label entry19) (reg env))
2.   (goto (label after-lambda18))
3. entry19
4.   (assign env (op compiled-procedure-env) (reg proc))
5.   (assign env (op extend-environment) (const (x)) (reg argl) (reg env))
6.   (assign val (op lookup-variable-value) (const x) (reg env))
7.   (goto (reg continue))
8. after-lambda18
9.   (goto (reg continue))
```

5. More Decompiling Code

Below is a listing of code produced by the compiler. For each of the following sections of compiled code, determine the Scheme expression that produced it.

```
 1.   (assign val (op make-compiled-procedure) (label entry2) (reg env))
 2.   (goto (label after-lambda1))
 3. entry2
 4.   (assign env (op compiled-procedure-env) (reg proc))
 5.   (assign env (op extend-environment) (const (a)) (reg argl) (reg env))
 6.   (save continue)
 7.   (save env)
 8.   (assign proc (op lookup-variable-value) (const >) (reg env))
 9.   (assign val (const 0))
10.   (assign argl (op list) (reg val))
11.   (assign val (op lookup-variable-value) (const a) (reg env))
12.   (assign argl (op cons) (reg val) (reg argl))
13.   (test (op primitive-procedure?) (reg proc))
14.   (branch (label primitive-branch11))
15. compiled-branch10
16.   (assign continue (label after-call9))
17.   (assign val (op compiled-procedure-entry) (reg proc))
18.   (goto (reg val))
19. primitive-branch11
20.   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
21. after-call9
22.   (restore env)
23.   (restore continue)
24.   (test (op false?) (reg val))
25.   (branch (label false-branch4))
26. true-branch5
27.   (assign val (op lookup-variable-value) (const a) (reg env))
28.   (goto (reg continue))
29. false-branch4
30.   (assign proc (op lookup-variable-value) (const -) (reg env))
31.   (assign val (op lookup-variable-value) (const a) (reg env))
32.   (assign argl (op list) (reg val))
33.   (test (op primitive-procedure?) (reg proc))
34.   (branch (label primitive-branch8))
35. compiled-branch7
36.   (assign val (op compiled-procedure-entry) (reg proc))
37.   (goto (reg val))
38. primitive-branch8
39.   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
40.   (goto (reg continue))
41. after-call6
42. after-if3
43. after-lambda1
44.   (perform (op define-variable!) (const f) (reg val) (reg env))
45.   (assign val (const ok)))
46.   (goto (reg continue))
```

Lines 8–20:

Lines 30–40:

Lines 8–40:

Lines 1–46: