MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1996

Problem Set 10

Register Machines and Compilation

Issued: Tuesday, April 23, 1996

Written *solutions due*: Wednesday, May 8 for EVEN numbered sections, Friday May 10 for ODD numbered sections.

Reading:

- Complete chapter 5.
- Look over the attached code files `compiler.scm` `eceval.scm`.
- Auxiliary files `regsim.scm` `assemble.scm` `evdata.scm` `syntax.scm` `compdata.scm` are also attached for reference.

****FINAL EXAM**:** The final will be held on Wednesday May 22nd in the Athletic Center at 9:00am. It will cover the whole course with extra emphasis on the material not covered in the previous two quizzes, namely, Problem Sets 9 and 10 and chapters 4–5 of the Notes.

Register machines provide a means of customizing code for particular processes. In principle, customization leads to more efficient code, since one can avoid the overhead that comes from a compiler's obligation to handle more general computations[1]. In this problem set you will handcraft two simple machines and compare them to the compiler in Chapter 5 of the notes, to the naive code generator described in lecture on April 30 and May 2, and to the explicit control evaluator of Chapter 5.

The register machines you define should use only the following few primitives:

```
+ - * / inc dec = < > zero?  not true false nil
cons car cdr pair? null? list eq? symbol? write-line
```

Even though code generated by the compilers uses more complex primitives such as

```
lookup-variable-value extend-environment ...,
```

your hand-crafted code should turn out to run more efficiently.

[1] For example, the compiler in Chapter 5 generates code in which arguments of procedures are maintained as a list in the `argl` register. On the other hand, a register machine customized for a procedure of, say, three arguments might usefully keep the arguments in three separate registers.

Remember that register machine instructions are only of the following types:

```
test branch assign goto save restore perform
```

In this problem set, you should not need to use `perform`. Remember also that the only values that can be assigned to registers or tested in branches are constants, fetches from registers, or primitive operations applied to fetches from registers. No nested operations are permitted[2].

Tutorial Preparation for the week of May 6

Tutorial Exercise 1 Define, in Scheme, two different procedures that compute the length of a list. You should make sure you define everything in terms of the available primitives. For each of your procedures, say what order of growth in time (total machine operations) and space (stack depth) you expect them to use. Try to pick your two procedures to have different orders of growth in space.

Tutorial exercise 2 Implement each of your procedures from exercise 1 as a register machine. You should show the controllers for both machines, but you need show the data paths for only one of them.

When implementing register machines, you should observe the following caveats:

- Every instruction is one of the following types: `assign`, `branch`, `goto`, `save`, `restore`. You should not need to use `perform`.
- The only values that can be assigned to registers or tested in branches are constants, fetches from registers, or primitive operations applied to fetches from registers. No nested operations are permitted.

To do in lab

In lab, you will use the register machine simulator to test the register machines you designed in the tutorial exercises. To use the simulator, load the code for problem set 10 and type in your machine definitions:

```
(define my-machine
  (make-machine
   '(x 1 val ...)
   standard-primitives          ; + - * / inc, etc.
   '((test ...)              ; add your machine controller code here
     ⋮
     )))
```

[2]For example,

```
(assign val (op inc) ((op *) (reg a) (reg b)))
```

is *not* permitted, since a call to `*` is nested inside a call to `inc`.

You'll find it convenient to define test procedures that load an input into a machine, run the machine, print some statistics, and return the result computed by the machine. Here is an example that works for a machine that has an input register `lst` and returns its result in register `result`. Depending on how you design your machine, you may need to make your own version:

```
(define (test-machine machine arg)
  (set-register-contents! machine 'lst arg)
  (newline)
  (display ";Resetting... ignore")
  (machine 'initialize-stack)
  (machine 'initialize-ops-counter)
  (newline)
  (display ";Running on arg: ") (display arg)
  (start machine)
  (newline)
  (display ";Run complete:")
  (machine 'print-stack-statistics)
  (machine 'print-ops)
  (get-register-contents machine 'result))
```

In addition to routines that gather statistics for stack usage and total number of operations, there are some procedures to help you debug your machines. `trace-reg-on` will show all assignments to a specified machine register as they occur. Evaluating:

```
(trace-reg-on my-machine 'l)
```

before running your test procedure will show you all the changes to the `l` register. To see even more stuff, try:

```
(trace-on my-machine)
```

which will print each machine instruction as it is executed. To get rid of these traces, use `trace-reg-off` and `trace-off`.

Lab exercise 1A: Debug your machines, run them on some representative inputs, and make a table that records the total number of machine operations, total number of stack pushes, and maximum stack depth as a function of the length of the list.

PostLab exercise 1B: Try to derive formulas for the total number of machine operations, total number of pushes, and maximum stack depth used by your machines, as functions of the length of the list. In most cases, the functions will turn out to be polynomials in the list length, in which case you should be able to exhibit exact formulas, not just orders of growth.

Running the Compiler

There are two ways to run the compiler. First, you may simply compile an expression and obtain the list of machine instructions as a result, so that you can study it. For instance,

```
(define test-expression '(define (f x y) (* (+ x y) (- x y))))
(define result (compile test-expression 'val 'return))
(pp result)
```

A way to look just at the produced code even more easily is:

```
(compile-and-display test-expression)
```

which does exactly the same call to `compile` as above.

The second way to run the compiler is to apply the procedure `compile-and-go` to the expression. This compiles the expression and executes it in the environment of the explicit control evaluator machine `eceval`. When evaluation is complete, you are left in the read-eval-print loop talking to the explicit control evaluator. Then you can experiment with the compiled expression by evaluating further expressions.

Running the Evaluator

The evaluator for this problem set is the explicit control evaluator of section 5.4. For your convenience, we have extended it to handle `cond` and `let`.

To evaluate an expression in the `eceval` read-eval-print loop, type the expression after the prompt, followed by `ctrl-X ctrl-E`. After each evaluation, the simulator will print the number of stack and machine operations required to execute the code.[3]

Here is an example:

```
(compile-and-go
 '(define (fact n) (if (= n 0) 1 (* n (fact (- n 1))))))
(total-pushes = 0 maximum-depth = 0)
(machine-ops = 9)
;;; EC-Eval value:  (the-unspecified-value)
;;; EC-Eval input: (fact 4)          <== you type this and ctrl-X ctrl-E
(total-pushes = 31 maximum-depth = 14)
(machine-ops = 278)
;;; EC-Eval value: 24
;;; EC-Eval input: (fact (fact 3))   <== you type this
(total-pushes = 68 maximum-depth = 20)
(machine-ops = 594)
;;; EC-Eval value: 720
;;; EC-Eval input: fact
(total-pushes = 0 maximum-depth = 0)
(machine-ops = 13)
;;; EC-Eval value: <compiled-procedure>
```

[3] These counts may include a few extra operations needed to run the driver loop itself. This is a small constant overhead that you can ignore when you collect statistics.

```
;;; EC-Eval input:
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1))))) ;<== fact gets redefined
(total-pushes = 3 maximum-depth = 3)
(machine-ops = 45)
;;; EC-Eval value: (the-unspecified-value)
;;; EC-Eval input: fact
(total-pushes = 0 maximum-depth = 0)
(machine-ops = 13)
;;; EC-Eval value:
;;;  (compound-procedure (n)
;;;    ((if (= n 0) 1 (* n (fact (- n 1))))) <procedure-env>)
;;; EC-Eval input:
(fact 4)                ;<== redefined fact gets interpreted -- slower!
(total-pushes = 144 maximum-depth = 20)
(machine-ops = 1572)
;;; EC-Eval value: 24
```

To exit back to regular Scheme type `ctrl-C ctrl-C`. To re-enter the evaluator with the previous global environment, you may do another `compile-and-go`, or you may simply evaluate `(eval-loop)` in Scheme. To start the evaluator with a reinitialized global environment, evaluate `(start-eceval)`.

Lab exercise 2A: Compile and run the (Scheme) definitions of your two `length` procedures, and make tables to record statistics.

Lab exercise 2B: Now redefine them within the `eceval` read-eval-print loop and record corresponding statistics for the interpreted definitions.

PostLab exercise 2C: Derive formulas for the total number of machine operations, total number of pushes, and maximum stack depth required, as functions of the length of the list, for the compiled and interpreted procedures.

Post Lab exercise 3: We'll consider the time used for a computation to be the total number of machine operations, and the space used to be the maximum stack depth. For each of your procedures, determine the limiting ratio, as the list length becomes large, of the time and space requirements for your hand-coded machines, versus the time and space requirements for the compiled

Lab exercise 4A: Make listings of the code generated by the compiler from the Notes for the definitions of your two procedures.

PostLab exercise 4B: Compare the listings with your hand-coded versions to see why the compiler's code is less efficient than yours. Suggest one improvement to the compiler that could lead it to do a better job. Write one or two clear paragraphs indicating how you might go about implementing your improvement. You needn't actually carry out the the implementation, but your description should be reasonably precise. For example, you should say what new information the compiler should keep track of, what new data structures may be required to maintain this information, and how the information should be used in generating the new, improved code.

Collaboration and Pedagogy

Indicate the names of your collaborators, if any, on this assignment. Indicate the source and nature of any other help you may have received in the problem solutions, including students not in 6.001 and any portions of the subject archives you may have referenced. (Remember, we encourage *open* collaboration among 6.001 students.)

Your Comments

This is it—the last 6.001 problem set this term! We hope you found it absorbing, educational, and fun. If not, we're sorry. Either way, we would very much appreciate your comments on this problem set, and any other aspects of the course you care to mention.

Good luck on the final.

Regards,
Jim and Louis