MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

**Problem Set 12**

**Register Machines and Compilation**

- Issued: Tuesday, April 28
- Tutorial preparation for: Week of May 4
- Written solutions due: Friday, May 8 in recitation
- Reading:
  - Complete Chapter 5: you do not need to know the details of section 5.2.
  - Look over the attached code files `compiler.scm`, `eceval-compiler.scm`, and `eceval-support.scm`.
  - Auxiliary files `regsim.scm`, and `syntax.scm` are available on-line for reference.

****FINAL EXAM**:** The final will be held on Monday, May 18, from 1:30 to 4:30 in Johnson Athletic Center (W34-100). It will cover the whole course with extra emphasis on the material not covered in the previous two quizzes, namely, Problem Sets 10, 11, and 12 and chapters 4–5 of the book.

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 hand-craft two simple machines and compare them to the compiler and 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, (e.g. `lookup-variable-value extend-environment ...`,) your hand-crafted code should turn out to run more efficiently.

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

[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.

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

The following exercises should be prepared for discussion in tutorial.

**Tutorial Exercise 1** Define, in Scheme, two different procedures that compute the minimum of the absolute values of a non-empty list of numbers. For example, `(minabs (list 5 2 -3))` evaluates to 2; `(minabs (list -1 0 -3 -7))` evaluates to 0. 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 (number of machine operations) and space (stack depth) you expect them to use. Pick your two procedures to have different orders of growth in space (i.e. iterative vs. recursive).

**Tutorial exercise 2** Implement each of your procedures from exercise 1 as a register machine, and show both the data paths and controllers for each machine. For the data paths, a diagram is needed; for the controllers, a textual description in the manner of Chapter 5 is adequate.

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 12 and type in your machine definitions. For example you might use something like:

```
(define my-machine
 (make-machine
  '(x l val ...)      ; list the registers you will use here
  standard-primitives ; + - * / inc, etc.
  '((test ...) ; add your machine controller code here
    \vdots
    )))
```

[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 will 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")
  (let ((the-stack (machine 'stack)))
    (the-stack 'initialize)
    (newline)
    (display ";Running on arg: ") (display arg)
    (start machine)
    (newline)
    (display ";Run complete:")
    (the-stack 'print-statistics))
  (get-register-contents machine 'result))
```

Notice that we use the approximation that the number of operations performed is proportional to the number of save operations performed. This is a pretty good approximation for code from our compiler, but it isn't good for hand written code.

**Computer exercise 1A:** Debug your machines, run them on some representative inputs, and make a table that records the number of stack pushes (our approximation for the number of operations) and maximum stack depth (our approximation for the amount of space required) as a function of the length of the list.

**Written exercise 1B:** Try to derive formulas for the 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.

Having built hand-crafted register machines for our problem, we now want to compare their performance with code generated by the compiler and code generated by the evaluator.

### 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)
```

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. **Warning:** the explict control evaluator and the compiler do *not* handle the special forms `cond` or `let`, so be sure any Scheme code you intend to use doesn't use them!

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]

Note that you can start up the evaluator by invoking `(start-eceval)` or you can use `compile-and-go` to enter the evaluator. Here is an example:

```
(compile-and-go
 '(define (fact n) (if (= n 0) 1 (* n (fact (- n 1))))))
(total-pushes = 0 maximum-depth = 0)
;;; EC-Eval value:
ok


;;; EC-Eval input:
(fact 4)          <== you type this, then C-X C-E
(total-pushes = 31 maximum-depth = 14)
;;; EC-Eval value:
24


;;; EC-Eval input:
(fact (fact 3))   <== you type this
(total-pushes = 68 maximum-depth = 20)
;;; EC-Eval value:
720


;;; EC-Eval input:
fact
(total-pushes = 0 maximum-depth = 0)
;;; EC-Eval value:
<compiled-procedure>


;;; EC-Eval input:
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1))))) ;<== fact gets redefined
(total-pushes = 3 maximum-depth = 3)
;;; EC-Eval value:
ok
```

[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:
fact
(total-pushes = 0 maximum-depth = 0)
;;; 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)
;;; 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`. To start the evaluator with a reinitialized global environment, evaluate `(start-eceval)`. To restart the read-eval-print loop *without* clearing the global environment, do `(continue-eceval)`.

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

**Computer exercise 2B:** Now redefine your two `minabs` procedures within the `eceval` read-eval-print loop and record corresponding statistics for the interpreted definitions.

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

**Written exercise 2D:** We'll consider the time used for a computation to be the total number of stack pushes, 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 and interpreted code.

**Computer exercise 3A:** Make listings of the code generated by the compiler for the definitions of your two procedures. Annotate these listings to indicate what various portions of the generated register code corresponds to, e.g. procedure definition, construction of argument lists, procedure application, etc.

**Written exercise 3B:** 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.

**Computer exercise 4:** To gain more understanding of the compiler (as described in Chapter 5 of the book), you will next make a small change to the language and modify the compiler accordingly. In particular, we wish to change variable assignment to actually return a useful value, in this case the new value of the variable. For example

```
(define x 1)
(set! x 2)
; Value: 2

(* (set! x (+ x 1)) 10)
; Value: 30
```

To do this problem, you will need to think carefully about how the compiler generates code and "preserves" registers by using the stack. You should not add any new registers to your machine. You will also have to consider (at least when you are testing your code) what order arguments are evaluated in – we've said "in any order" but the compiler and the interpreter use a particular order (and is it the same?).

Turn in listings of your modifications to the compiler, together with test cases that show both the compiled code generated (using `compile-and-display`) and the results returned for your test cases (using `compile-and-go`). Remember, compiled code for your `set!` expressions will have this new behavior, but `eceval` itself will not, so keep this in mind when you are debugging.

Some test cases you should consider include:

```
(set! x 1)
(set! x (+ 1 2))
(begin (set! x (+ 1 2)) 3)
(+ x (set! x 10))
```

That's all folks – the last problem set! We hope you have found them entertaining, engrossing, and educational.