Structure and Interpretation of Computer Programs
Second Edition
Sample Problem Set

Compilation

Register Machines and Compilation

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.

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

```
+ - * / inc dec = < > zero?  not true false
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.

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].

Here are the definitions of two Scheme procedures for deleting all occurrences of an element, `x`, from a list, `l`:

```
(define (delq1 x l)
  (cond ((null? l) '())
        ((eq? x (car l)) (delq1 x (cdr l)))
        (else (cons (car l) (delq1 x (cdr l))))))

(define (delq2 x l)
  (define (delete-reverse maybe-xs no-xs)
    (cond ((null? maybe-xs) no-xs)
          ((eq? x (car maybe-xs)) (delete-reverse (cdr maybe-xs) no-xs))
          (else (delete-reverse (cdr maybe-xs) (cons (car maybe-xs) no-xs)))))
  (delete-reverse (delete-reverse l '()) '()))
```

PreLab exercise 1A: For each of these procedures, say what order of growth in time (total machine operations) and space (maximum stack depth) you expect them to use.

PreLab exercise 1B: Implement both of these procedures as register machines. You should show the controllers for all the machines, but you need show the data paths for only one of them.

---

[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.

[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`.

To do in lab

In lab, you will use the register machine simulator to test the register machines you designed in exercise 1B. To use the simulator, load the code for problem set 10 and type in your machine definitions:

```
(define my-machine
  (make-machine
   '(x l val ...)
   standard-primitives          ; + - * / inc, etc.
   '((test ...)
     ⋮
     )))
```

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. For example:

```
(define (test-machine x l)
  (set-register-contents! my-machine 'x x)
  (set-register-contents! my-machine 'l l)
  (my-machine 'initialize-stack)
  (my-machine 'initialize-ops-counter)
  (start my-machine)
  (my-machine 'print-stack-statistics)
  (my-machine 'print-ops)
  (get-register-contents my-machine 'val))
```

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 2A: 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 from which an element is being deleted.

PostLab exercise 2B: 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 = 12)
;;; 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>

;;; 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)
```

[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)
(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 reenter 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 3A: Compile and run the (Scheme) definitions of the `delq1` and `delq2` procedures, and make tables to record statistics.

Lab exercise 3B: Now redefine them within the `eceval` read-eval-print loop and record corresponding statistics for the interpreted definitions.

PostLab exercise 3C: 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 delete procedures.

The file `naivecom.scm` contains the simple code generator described in lecture which omits the stack optimizations carried out in the Notes. Because its procedures satisfy a simplified "contract" slightly different from that in the Notes, the naive compiler is not compatible with the evaluator. We display its code using `naive-compile-and-display`, and run naive code using `naive-compile-and-go`. For example:

```
(define (fibexp n)
 `(begin
    (define (fib m)
      (if (< m 2)
          m
          (+ (fib (- m 1)) (fib (- m 2)))))
    (fib ,n)))
(naive-compile-and-go (fibexp 8))
(total-pushes = 1469 maximum-depth = 27)
(machine-ops = 6544)
;value of expression: 21
;Value: #[useless-value]
```

Compare this to:

```
(compile-and-go (fibexp 8))
(total-pushes = 332 maximum-depth = 23)
(machine-ops = 2772 )
;;; EC-Eval value: 21
```

So in this case, the optimized code executes about 40% as many instructions and uses 85% of the stack space as the code generated by the naive compiler.

Lab exercise 4: Repeat exercise 3 for the naively compiled delete procedures.

Post Lab exercise 5: 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 list-deletion 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, naive-compiled, and interpreted code.

Lab exercise 6A: Make listings of the code generated by the compiler from the Notes for the definitions of `delq1` and `delq2`.

PostLab exercise 6B: 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.