Structure and Interpretation of Computer Programs
October 2000

Problem Set 10
Evaluation, Garbage Collection, Register Machines

Issued: Friday, 27 October 2000
Due: Sunday, 29 October 2000
Reading: Text (SICP 2nd Edition by Abelson & Sussman): Sections 4.1, 5.1, 5.3 (optional)

Exercise 1 For the first series of problems you will compare the relative speeds of the metacircular evaluator and the analyzing evaluator. As discussed in lecture and recitation, by separating the syntactic analysis (and translation into an underlying representation) from the expression evaluation, we hope to achieve large gains when executing code that repeatedly evaluates a given expression.

From the last problem set you have experience running the metacircular evaluator. Now, walk through starting up the analyzing evaluator. You can tell that you are typing at the analyzing evaluator because the prompt will be `A-EVAL=>` rather than `MC-EVAL=>`. As you did for the metacircular evaluator, type a few simple expressions and definitions. Recall that you will be typing these expressions in ADU Scheme where all special form names and primitive procedures are prefixed with `adu:`.

To start the metacircular evaluator, evaluate `(start-mc-eval)`. As before, if at any point an error causes you to exit to Scheme, the evaluator can be restarted with `(driver-loop)`, leaving `the-global-environment` undisturbed. Re-evaluating `(start-mc-eval)` will replace `the-global-environment` with a newly created environment, overwriting anything that you had added.

Similarly, to start the analyzing evaluator, evaluate `(start-analyze)`. And again, if an error causes you to exit to scheme, you can restart the analyze evaluator by evaluating `(driver-loop)` to resume with `the-global-environment` that existed before the error.

Here's an example of what you should see:

```
(start-mc-eval)
;;; MC-Eval input: (adu:define (adu:fact n)
                       (adu:if (adu:= n 0)
                               1
                               (adu:* n (adu:fact (adu:- n 1)))))
;;; MC-Eval value: ok
;;; MC-Eval input: (adu:fact 3)
;;; MC-Eval value: 6
;;; MC-Eval input: ⟨C-c C-c⟩
;Quit!

(start-analyze)
;;; A-Eval input: (adu:define (adu:fact n)
                       (adu:if (adu:= n 0)
                               1
                               (adu:* n (adu:fact (adu:- n 1)))))
;;; A-Eval value: #[unspecified-return-value]
;;; A-Eval input: (adu:fact 3)
;;; A-Eval value: 6
```

There is nothing to turn in for this problem.

Exercise 2 Ben Bitdiddle thinks it is silly to reinitialize the environment every time he switches from one evaluator to the other. So, after having run the metacircular evaluator, he revises the `start-analyze` procedure so it does not reinitialize the global environment:

```
(define (start-analyze)
  (set! current-evaluator (lambda (exp env) ((analyze exp) env)))
  (set! current-prompt "A-EVAL=> ")
  (set! current-value-label ";;A-value: ")
;  (init-env)              ; Ben comments-out this line
  (eval-loop))
```

He reasons that now, if he types `(start-analyze)`, the definitions he already evaluated in the metacircular evaluator will be available to the analyzing evaluator. Eva Lu Ator agrees that the definitions of the metacircular evaluator will indeed be available to the analyzing evaluator, but that this will usually crash the system rather then being helpful.

Try Ben's suggestion and observe what Eva Lu Ator meant. Briefly, but clearly, explain what goes wrong with Ben's suggestion. Are there any definitions it would be safe to preserve when switching between evaluators?

Exercise 3 This next problem asks you to demonstrate the improved efficiency of the analyzing evaluator over the metacircular one.

To time things, you can use the procedure `show-time`. Thus, for example, you can find out how long it takes the evaluator to evaluate `(adu:fib 10)` by quitting out of the evaluator and evaluating

```
(show-time (lambda() (current-evaluator '(adu:fib 10) the-global-environment)))
```

in Scheme. Be careful to define the procedure you are timing, *e.g.*, `adu:fib`, in the evaluator, not in ordinary Scheme! Otherwise, you'll end up timing the underlying Scheme interpreter. This is because of the way we've linked the evaluator into Scheme: if you define `adu:fib` in Scheme, and not in the evaluator's global

environment, `lookup-variable-value` will find the Scheme procedure `adu:fib` and `m-apply` will treat this as a primitive.

Design and carry out an experiment to compare the speeds of the metacircular and analyzing evaluators on some procedures. Use tests that run for a reasonable amount of time (say 10 or 20 seconds). It may be helpful to use test procedures for which small changes in the input cause large changes in running time, so you can rapidly increase the time by increasing the input.

Recall that you will need to re-initialize `the-global-environment` when switching between the evaluators, and that definitions in one evaluator will not work for the other (see the exercise above). To keep them separate, you might call the two functions `adu:mc-fib` and `adu:a-fib`.

Summarize what you observe about the relative speeds of the two evaluators on your test programs.

Exercise 4 (Optional) See if you can find an example where both of the interpreters run for at least five seconds, and analyzing evaluator runs *no more* than 1.5 times the speed of the metacircular evaluator.

Exercise 5 We now switch gears and start looking at garbage collection, specifically the operation of the stop-and-copy garbage collector in the text. (This problem is adapted from the *SICP Instructor's Manual*)

Although Ben Bitdiddle has done his best to explain the stop-and-copy garbage collector algorithm to Cy D. Fect, Cy cannot understand what use the broken hearts have. He simplifies the garbage collector by removing all instructions that deal with broken hearts, so that the code at the entry `pair` is as follows:

```
pair
  (assign  oldcr (op vector-ref) (reg the-cars) (reg old))
  (assign  new   (reg free))
  (assign  free  (op +) (reg free) (const 1))
  (perform (op vector-set!) (reg new-cars) (reg new) (reg oldcr))
  (assign  oldcr (op vector-ref) (reg the-cdrs) (reg old))
  (perform (op vector-set!) (reg new-cdrs) (reg new) (reg oldcr))
  (goto    (reg relocate-continue))
```

A To test his algorithm, Cy simulates a garbage collection with the working memory as shown in Figure 5.14 of the text (page 536) and `p1` in the `root`. Show the contents of the new memory and the `root` after Cy's garbage collection. Draw a box-and-pointer diagram of the structure pointed to by the `root` after garbage collection. Is it the same as the list structure we stared with?

B Cy is satisfied that his garbage collector works, so he installs it in the system. Not long afterwards, the computer starts giving strange error messages. It turns out that shortly before the troubles started, someone evaluated

```
(define x (cons 1 2))
(set-cdr! x x)
```

Draw the box-and-pointer representation and the memory-vector representation (as in Figure 5.14) of the list structure `x`. What pointer represents `x`'s value? How did this list structure break Cy's garbage collector? (Hint: Simulate a garbage collection starting with a pointer to `x`—*i.e.*, the pointer that represents `x`'s value—in `root`.)

Exercise 6 Design a register machine to compute factorials using the iterative algorithm specified by the following procedure. Draw data-path and controller diagrams for this machine. (This is problem 5.1 from the text.) A description of valid register machine instructions on pages 512–513 may prove helpful.

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1)))))
```

Exercise 7 Use the register-machine language to describe the iterative factorial machine of the previous exercise. (This is problem 5.2 from the text.)

Exercise 8 (Optional) Examine the following register transfer language code fragment and try to determine the originating Scheme code (this process is called *decompilation)*. (Hint: Make educated guesses as to the exact compilation mechanism and keep the environment model in mind.)

```
 (assign  proc (op make-compiled-procedure) (label entry1) (reg env))
 (goto    after-lambda0)

ENTRY1
 (assign  env  (op compiled-procedure-env) (reg proc))
 (assign  env  (op extend-binding-environment) (const (a b)) (reg argl) (reg env))
 (assign  proc (op lookup-variable-value) (const /) (reg env))
 (save    proc)
 (assign  proc (op lookup-variable-value) (const +) (reg env))
 (assign  val  (op lookup-variable-value) (const A) (reg env))
 (assign  argl (op cons) (reg val) (const ()))
 (assign  val  (op lookup-variable-value) (const B) (reg env))
 (assign  argl (op cons) (reg val) (reg argl))
 (assign  continue after-call-2)
 (save    continue)
 (goto    apply-dispatch)

AFTER-CALL2
 (assign  argl (op cons) (reg val) (const ()))
 (assign  val  (const 2))
 (assign  argl (op cons) (reg val) (reg argl))
 (restore proc)
 (goto    apply-dispatch)

AFTER-LAMBDA0
 (assign  val  (const 1))
 (assign  argl (op cons) (reg val) (const ()))
 (assign  val  (const 2))
 (assign  argl (op cons) (reg val) (reg argl))
 (goto    apply-dispatch))
```

Problem Set Evaluation To help us evaluate how well the course is going, and how good a job we are doing, please tear off this page, answer the following questions, and attach it to your problem set when you hand it in. While we do not require you to fill in this survey, doing so will help correct any problems and improve the course not only for future students, but for you as well.

Number of hours spent on the reading assignment: _____

Number of total hours spent working directly on the problem set _____

Number of collaborative hours spent working directly on the problem set _____

How hard was this problem set (1–10: 1, piece of cake; 5, just right; 10, far too hard) _____

Hardest problem _____

Fraction of time spent on hardest problem, multiplied by 10 (1–10: 1, only a small fraction; 5, half of the time; 10, every single minute) _____

Coherence between problem set contents and lecture/recitation contents (1–10: 1, seemingly random; 5, good correspondence; 10, perfect coherence) _____

Coherence between problem set contents and assigned reading contents (using the same scale) _____

How much did you enjoy this problem set (1–10: 1, nearly walked out of the program; 5, neutral; 10, recruited other students because it was so good) _____

How much did you enjoy the lectures and recitations for this problem set (using the same scale) _____

You may also submit fully anonymous comments via the electronic means described in class.