

Topic: Lazy evaluator, Analyzing evaluator, Nondeterministic evaluator

Lectures: Monday August 4, Tuesday August 5

Reading: Abelson & Sussman, Section 4.1.7–4.3.2 (Pages 393–426) skim the parsing stuff

This assignment is an evaluator potpourri, giving you practice with the lazy and nondeterministic evaluators mostly “above the line.”

- `~cs61a/lib/lazy.scm` – Lazy evaluator
- `~cs61a/lib/vambeval.scm` – Nondeterministic evaluator

Please put your solutions into a file called `hw7-1.scm` and submit it online as usual. Include only the code you wrote and test cases. The assignment is due at **8 PM on Sunday, August 7.**

Question 1. In the lazy evaluator `actual-value` is called in four places: to evaluate the arguments to a primitive procedure, to evaluate the operator in a procedure application, to print the results in the REPL and to evaluate the predicate in a conditional. This question investigates what happens when we replace `actual-value` with `mc-eval` in two of these. For each of the following two scenarios, describe what goes wrong and include a brief session with the lazy evaluator that demonstrates the problem.

A. Suppose we change the application clause to use `mc-eval`, like this:

```
((application? exp)
 (mc-apply (mc-eval (operator exp) env)      ;; was actual-value
            (operands exp)
            env))
```

B. Suppose we change `eval-if` to use `mc-eval`, like this:

```
(define (eval-if exp env)
  (if (true? (mc-eval (if-predicate exp) env))      ;; was actual-value
      (mc-eval (if-consequent exp) env)
      (mc-eval (if-alternative exp) env)))
```

The adventure continues on the next page.

Question 2. We'd like to write a nondeterministic program to crack a combination lock. Since there is only a finite number of combinations, all it takes is time! We will represent locks as message-passing objects created with the following procedure:

```
(define (make-lock combination)
  (lambda (message combo)
    (cond ((eq? message 'try) (if (equal? combo combination) 'open 'nice-try))
          (else (error "I don't understand " message))))))
```

As you can see, it's not a very sophisticated lock; it only knows the message `try`, which comes with one argument taken to be a test combination. If the test combination matches the real combination, the lock says `open`; otherwise it says `nice-try`.

- A.** Your task is to write a nondeterministic program `code-breaker` that takes a lock and returns the combination that opens it. Assume that a combination is a list of three elements

```
((left n) (right n) (left n))
```

where n is between 0 and 20, inclusive, and the directions are exactly as shown: left, right, left. Here is the desired behavior:

```
;;; Amb-Eval input:
(define lock1 (make-lock '((left 10) (right 14) (left 3))))

;;; Starting a new problem
;;; Amb-Eval value:
ok

;;; Amb-Eval input:
(code-breaker lock1)

;;; Starting a new problem
;;; Amb-Eval value:
((left 10) (right 14) (left 3))
```

- B.** Now let's remove the left-right-left requirement. Combinations are still three-element lists, but the directions can be in any order. Each of the following are valid combinations:

```
((left 3) (left 4) (left 5))
((right 17) (left 4) (left 15))
((right 20) (right 20) (right 20))
```

Modify your program from Part A to crack these locks.