MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 19 — 11/9/2007 Solutions
Analysis & Quiz II Review

Since some sections of code will be evaluated repeatedly, performance can be improved by doing some work before beginning to evaluate, such that each evaluation takes less time. The `analyze` evaluator does this by computing how to evaluate an expression and saving it, such that at evaluation time, it doesn't need to re-figure it out each time the expression is evaluated. It saves the work in a procedure by returning a lambda that takes in an environment. Some bits of analyze:

```
(define (eval exp env)
  ((analyze exp) env))

(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp) (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))
        (else
         (error "Unknown expression type -- ANALYZE" exp))))

(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))

(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env)
      (define-variable! var (vproc env) env)
      'ok)))

(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application (fproc env)
                           (map (lambda (aproc) (aproc env))
```

```
                                aprocs)))))

(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))))
        (else
         (error
          "Unknown procedure type -- EXECUTE-APPLICATION"
          proc))))
```

1. Assuming that `analyze` is extended to include `((let? exp) (analyze-let exp))`, implement `analyze-let`:

```
(define (analyze-let exp)
  (let ((val-procs (map analyze (let-values exp)))
        (body-proc (analyze-sequence (let-body exp)))
        (vars (let-variables exp)))
    (lambda (env)
      (body-proc (extend-environment
                  vars
                  (map (lambda (v) (v env)) val-procs)
                  env)))))
```

Quiz II Review

Environment Diagrams

2. Draw an environment diagram to trace through evaluating the following expressions:

```
(define (f x)
  (if (< x 0)
    (lambda (y) (- y x))
    (lambda (y) (- x y))))

(define (foo bar x y)
 (let ((g (bar y)))
   (+ (g x) (g y))))

(foo f 1 -2)
```

Random Streams

Assume that **ran** is a primitive Scheme procedure that generates random numbers in the range 0 to 1, e.g.

```
(ran)
0.486726

(ran)
0.929204

(ran)
0.08849

(ran)
0.283186
```

Assume that successive calls to RAN *never* produce the same number.

Louis Reasoner wants to define a stream whose elements consist of different random numbers, as in the sequence above. He attempts to define a stream of random numbers as follows:

```
(define random-stream
  (cons-stream (ran)
               random-stream))
```

Lem E. Tweakit isn't sure that Louis' definition will work, and he suggests the following:

```
(define (make-random-stream)
  (cons-stream (ran)
               (make-random-stream)))

(define random-stream (make-random-stream))
```

The two friends show their work to Alyssa P. Hacker who suggests that they use PRINT-STREAM to examine the first few elements of their streams. Furthermore she suggests that they run their code on two different Scheme interpreters, one that implements delayed pairs using memoization, and one that does not.

Lous and Lem take her advice, and just to be sure, they print out their streams twice. Shown below are pairs of printouts, of the sort that either Louis or Lem might have produced.

Possible Outcomes:

```
(print-stream random-stream)            ;;; OUTCOME A
0.486726 0.929204 0.008849 0.283186 ...

(print-stream random-stream)
0.486726 0.929204 0.008849 0.283186 ...


(print-stream random-stream)            ;;; OUTCOME B
0.486726 0.929204 0.008849 0.283186 ...

(print-stream random-stream)
0.486726 0.521080 0.297045 0.991644 ...


(print-stream random-stream)            ;;; OUTCOME C
0.486726 0.929204 0.008849 0.283186 ...

(print-stream random-stream)
0.365913 0.521080 0.297045 0.991644 ...


(print-stream random-stream)            ;;; OUTCOME D
0.486726 0.486726 0.486726 0.486726 ...

(print-stream random-stream)
0.486726 0.486726 0.486726 0.486726 ...


(print-stream random-stream)            ;;; OUTCOME E
0.486726 0.486726 0.486726 0.486726 ...

(print-stream random-stream)
0.591003 0.591003 0.591003 0.591003 ...
```

List all of the Possible outcomes (chosen from A,B,C,D,E) that could have been produced in each of the following cases, or indicate **none** if none of these outcomes is possible.

(a) Louis' definition; no memoization
D

(b) Louis' definition; with memoization
D

(c) Lem's definition; no memoization
D

(d) Lem's definition; with memoization
D