Homework 08

● Graded

Student
Giancarlos Marte

Total Points
100 / 100 pts

Autograder Score
100.0 / 100.0

Passed Tests
Exercise 1. eval-exp apply (1) (6/6)
Exercise 1. eval-exp apply (2) (6/6)
Exercise 1. eval-exp apply+lambda (6/6)
Exercise 1. eval-exp lambdas (12/12)
Exercise 1. eval-exp values (10/10)
Exercise 1. eval-exp variables (12/12)
Exercise 2. eval-term all (26/26)
Exercise 2. eval-term define (10/10)
Exercise 2. eval-term seq + define (6/6)
Exercise 2. eval-term seq + value (6/6)

Autograder Results

Exercise 1. eval-exp apply (1) (6/6)

Exercise 1. eval-exp apply (2) (6/6)

Exercise 1. eval-exp apply+lambda (6/6)

Exercise 1. eval-exp lambdas (12/12)

Exercise 1. eval-exp values (10/10)

Exercise 1. eval-exp variables (12/12)

Exercise 2. eval-term all (26/26)

Exercise 2. eval-term define (10/10)

Exercise 2. eval-term seq + define (6/6)

Exercise 2. eval-term seq + value (6/6)

Submitted Files

hw8.rkt Download

```
#lang typed/racket
#|
   ===> PLEASE DO NOT DISTRIBUTE THE SOLUTIONS PUBLICLY <===

  We ask that solutions be distributed only locally -- on paper, on a
  password-protected webpage, etc.

  Students are required to adhere to the University Policy on Academic
  Standards and Cheating, to the University Statement on Plagiarism and the
  Documentation of Written Work, and to the Code of Student Conduct as
  delineated in the catalog of Undergraduate Programs. The Code is available
  online at:

  https://www.umb.edu/life_on_campus/dean_of_students/student_conduct

|#

(require "hw8-util.rkt")
(provide (all-defined-out))

(: env-put (handle d:variable d:value -> (eff-op memory d:void)))
(define (env-put env var val)
  (lambda ([mem : memory]) : (eff memory d:void)
    (define new-state (environ-put mem env var val))
    (define result (d:void))
    (eff new-state result)))


(: env-push (handle d:variable d:value -> (eff-op memory handle)))
(define (env-push env var val)
  (lambda ([mem : memory]) : (eff memory handle)
    (environ-push mem env var val)))

(: env-get (handle d:variable -> (eff-op memory d:value)))
(define (env-get env var)
  (lambda ([mem : memory]) : (eff memory d:value)
    (define val (environ-get mem env var))
    (eff mem val)))

(: eval-exp (handle d:expression -> (eff-op memory d:value)))
(define (eval-exp env exp)
  (match exp
    [(? d:value?)
     ; Return: v
     (eff-pure exp)]
    [(? d:variable?)
```

```
    ; Return: E(x)
    (env-get env exp)]
   [(d:lambda x e)
    ; Return: {E, λx.t}
    (define close (d:closure env x e))
    (eff-pure close)]
   [(d:apply ef ea)
    (do
      ;; ef ⇓E {Ef, λx.tb}
      vf : d:value <- (eval-exp env ef)
      (match vf
        [(d:closure Ef x tb)
         (do
           ;; ea ⇓E va
           va : d:value <- (eval-exp env ea)
           ;; Eb ← Ef + [x := va]
           Eb : handle <- (env-push Ef x va)
           ;; tb ⇓Eb vb
           ; ...
           (eval-term Eb tb))]))]))

(: eval-term (handle d:term -> (eff-op memory d:value)))
(define (eval-term env term)
  (match term
    [(d:define x e)
     (do
       ;; e ⇓E v
       v : d:value <- (eval-term env e)
       ;; E ← [x := v]
       (env-put env x v))]
    [(d:seq t1 t2)
     (do
       ;; t1 ⇓E v1
       v1 : d:value <- (eval-term env t1)
       ;; t2 ⇓E v2
       (eval-term env t2))]
    [(? d:expression?)
     (eval-exp env term)]))

```