**CSSE 304 Exam 2 Feb 5, 2019 (day 30.5) Name\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Section 01(10:00) 02(11:00) 03(12:00)**

**Part 2, programming.** For this part, you may use TSPL, EoPL, CSUG, your notes, and a Scheme programming environment, plus the PLC grading server and any materials that I provided online for the course. You may not use any other web or network resources. You may look at and use any Scheme code that you (or you and your partner) have previously written.

There are two assignments on the PLC server, one for each problem. I suggest that you have a separate solution file for problem 1.

**Mutation is allowed for the interpreter problem, but not the CPS problem.**

|  |  |  |  |
| --- | --- | --- | --- |
| Problem | Possible | Earned | Comments |
| C1 | 15 |  |  |
| C2 | 15 |  |  |
| Total | 30 |  |  |

**1. (15 points)** The starting code that I will email to you contains two cps procedures that use Scheme procedures to represent continuations. You are to rewrite both procedures so that they use the “define-datatype” representations of continuations.   
Be sure to read the comments at the top of the starting code. Here are some test-cases:

> **(flatten-cps '() (list-k))**

(())

> **(flatten-cps '(a) (init-k))**

(a)

> **(flatten-cps '(a (b)) (init-k))**

(a b)

> **(flatten-cps '((a) b) (init-k))**

(a b)

> **(flatten-cps '((a) () b) (list-k))**

((a b))

> **(flatten-cps '(() (((a b (c)) () (d) e) () f)) (init-k))**

(a b c d e f)

**Submit this problem to the** E2-202020-cps **assignment on the PLC grading server.**

**2. (15 points)** Add local defines to your interpreter’s interpreted language. For example,

(let ([x 5])

(define foo (lambda (y) (bar x y)))

(define bar (lambda (a b) (+ (\* a b) a)))

(foo (+ x 3)))

returns 45, and it is equivalent to:

(let ((x 5))

(letrec ([foo (lambda (y) (bar x y))]

[bar (lambda (a b) (+ (\* a b) a))])

(foo (+ x 3))))

While Scheme allows local defines (just like letrec) to bind the variables to something other than lambda-expressions, your interpreter’s local defines (just like letrec in your interpreter) are not required to do so; you are allowed to treat a local define whose expression is not a lambda-expression as an error. My test code will not have any of those.

(Problem description continues on the back of this page)

Local defines must be immediately inside the lambda, let, etc. in which they live, and must come before any actual bodies. Otherwise your parser should call (eopl:error **'**parse-expression ″some descriptive string″) as in assignment A11. Some example inputs that should cause parse errors include

(eval-one-exp '

(let ([a 4])

(define a (lambda (x) (\* x (+ x 2))))

(a 4)

(define b (lambda (y) (- y 7))

(b 2))))

(eval-one-exp '

(let ([a 4])

(define a (lambda (x) (\* x (+ x 2))))

(define b (lambda (y) (- y 7)))))

(eval-one-exp '

(let ([x 3])

(if (< x 2)

(define a (lambda (x) (- 2 x)))

(define a (lambda (x) (+ 3 x))))

(a 7))

Some valid test cases:

> **(eval-one-exp '**

**((lambda (x y)**

**(define a (lambda (z) (+ x y z)))**

**(define b (lambda (w) (+ w (\* x y))))**

**(list (a 4) (b 5)))**

**6 7))**

(17 47)

> **(eval-one-exp '**

**(let ([a 10] [b (list 6)])**

**(set-car! b 7)**

**(letrec ([fact (lambda (n)**

**(if (zero? n)**

**1**

**(\* n (fact (- n 1)))))])**

**(define fib (lambda (n)**

**(if (< n 2)**

**n**

**(+ (fib (- n 1)) (fib (- n 2))))))**

**(list (fib a) (fact (car b))))))**

(55 5040)

> **(eval-one-exp '**

**(begin**

**(define odd?**

**(lambda (x)**

**(define odd? (lambda (n) (if (zero? n) #f (even? (- n 1)))))**

**(define even? (lambda (m) (if (zero? m) #t (odd? (- m 1)))))**

**(odd? x)))**

**(list (odd? 3) (odd? 4))))**

(#t #f)

> **(eval-one-exp '**

**((lambda (x y)**

**(define a (lambda (z) (+ (b x) y z)))**

**(define b (lambda (w) (+ w (\* x y))))**

**(list (a 4) (b 5)))**

**6 7))**

(59 47)

All but one of these valid test cases can work if you start with your A16 interpreter. That one needs top-level define from A17a.

When you add code to one of the major procedures (such as parse-exp) that has several cases, please add your new case near the top so it is easy for us to find. Most likely, your score for this problem will be the score that you get on the server, unless it is reduced because we discover that you wrote code that is customized for some of the test cases in order to get a few server points.

[Hint: parsing may be the hardest part of this problem, so be very careful with that. If you don’t already have a parse-bodies procedure (yours may have a different name) that parses all of the bodies of a lambda, let, etc. adding it might be very helpful.]

**Submit your code for this problem to the** E2-202020-int **assignment on the PLC server.   
If you submit multiple files, please include** *chez-init.ss* **in your ZIP archive. This will make grading easier.**