**CSSE 304 Final Exam Part 2 (interpreter part)** Monday evening, Feb 24, 2020 **Name\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

For this part, you may use the resources allowed for Part 1, plus TSPL and EoPL, your notes, and a Scheme programming environment, plus the PLC grading program and any materials that I provided online for the course. You may not use any other web or network resources. You are allowed to look at and use any Scheme code that **you** (and in the case of interpreter code, your interpreter partner)have previously written.

**Please test your code offline before submitting to the PLC server, since the server is responding so slowly these days**.

**Caution!** It is possible to get so caught up in getting all of the points for one problem that you do not get to the other problems on either computer part. Don’t do that! If you get stuck on one problem for a long time, try another problem.

Efficiency and elegance will not affect your score. Do not write code that is specific to tonight’s test-cases.

Submit to the PLC server: As usual, submit either a single file with solutions to both problems, or a zip file that is such that loading main.ss will load solutions to all three problems.

There are two server assignments, **F202020-int1** and **F202020-int2**.

Just in case something goes wrong with the server after the exam ends: email me your solutions to all computer problems before you leave the room.

If you finish these problems before you finish the non-interpreter computer part, please go ahead and turn in this paper, so that we can begin grading your work.

**Instructor use:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Problem** | **Max score** | **Your score** | **Comments** |
| local defines revisited | 1 | 15 |  |  |
| escaper | 2 | 10 |  |  |
|  | **Total** | **25** |  |  |

**1. (15 points)**

This is another chance for you to work on local defines, in case you ran out of time on exam 2 or my previous test cases did not catch some of the errors or omissions in your previous code. If your previous code is correct and it works with my new test cases, you should not have to change anything for this exam.

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.

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))

(eval-one-exp '

(begin

(define a 3)

(set! a (let ([t 5])

(\* 5 (define a 7) (+ a 6))))

a))

Some valid test cases:

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

**(begin**

**(define foo 4)**

**(define bar**

**(let ([x 5])**

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

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

**(foo (+ x 3))))**

**(list foo bar)))**

(4 45)

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

**(begin**

**(define fib 3)**

**(set! fib**

**(+ fib**

**(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))))))**

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

**(+ fib 6)))**

6227020864

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

**(begin**

**(define f 6)**

**(define g 3)**

**(+ f**

**(let ([f (lambda (x) (\* 5 x))])**

**(let ([g (lambda (x) (+ 2 (f x)))])**

**(define f (lambda (x) (g (+ x 4))))**

**(f 3)))**

**g)))**

46

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

**(begin**

**(define fact 1860)**

**(define a**

**(begin (set! fact**

**(+ fact (let ([fact (let ([n 4])**

**(define fact**

**(lambda (x)**

**(if (zero? x)**

**1**

**(\* x (fact (- x 1))))))**

**(fact n))])**

**(set! fact (+ 6 fact))**

**fact)))**

**(+ 8 fact)))**

**(+ a fact)))**

3788

>

**2. (10 points)** Recall that ((escaper f) x y z) returns the same value that (f x y z) returns, but it returns that value as the final answer for the entire expression being evaluated. Add escaper to the language of your A18 interpreter.

(eval-one-exp '(+ 5 ((escaper \*) 6 4))) 🡺 24

(eval-one-exp '(let ([escape-\* (escaper \*)])

(- 6 (escape-\* 7 8)))) 🡺 56

(eval-one-exp '(procedure? escaper)) 🡺 #t

(eval-one-exp '(procedure? (escaper +))) 🡺 #t

(eval-one-exp '(procedure? ((escaper +) 3 5))) 🡺 8

(eval-one-exp '(let ([escape-\* (escaper \*)])

(- 6 (escape-\* 4 (escape-\* 7 8)))))) 🡺 56

(eval-one-exp '(let ([e-list (map escaper (list + \*))])

(+ 5 ((cadr e-list) 6 7)))) 🡺 42

Submit to the PLC server: There are two server assignments, **F202020-int1** and **F202020-int2**.

Just in case something goes wrong with the server after the exam ends: email me your solutions to all computer problems before you leave the room.

If you finish these problems before you finish the non-interpreter computer part, please go ahead and turn in this paper, so that we can begin grading your work.