CSSE 304 Exam 2 Jan 22, 2019 (day 26.5) Name_____ Section 01(9:00) 02(9:55) 03(10:50)

Part 2, programming (non-interpreter). For this part, you may use the resources from Part 1, plus TSPL, EoPL, 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 have previously written.

Whenever you finish either of Part 2 or part 3, please turn in the paper, so we can begin grading it.

Submit your code to the E2-part2-201920 **assignment on the PLC server.**

Mutation is allowed (in fact, required) for both of these problems. Efficiency and elegance will not affect your score.

Problem	Possible	Earned	Comments
C1	10		
C2	12		
Total	24		

1. (10 points) When trying to make your code more efficient, it is often useful to discover where it is spending its time. One way to do this is to count how many times various procedures are called. It would be too painful to have to rewrite all of the individual procedures to include a counter in each one; it would be better to have a generic "counter-maker" procedure that can create a counting "wrapper" procedure around any existing procedure, that counts the number of external calls to that procedure (Can you see why counting recursive calls is impossible)

You are to define a procedure called `counter-maker` that takes a single argument, which must also be a procedure. `(counter-maker f)` returns a procedure that

- if it is called with the single argument `'count`, returns a count of how many times it has called f. It does not call f .
- if it is called with any other number of arguments or any other single argument, it simply increments the counter and calls f, passing the given arguments to f.

```
> (define counted-member (counter-maker member))
> (define counted-cons (counter-maker cons))
> (counted-member 'count (counted-cons 'I
                          (counted-cons 'can
                            (counted-cons 'count (counted-cons 'this! '()))))))
(count this!)
> (counted-member 'count)
1
> (counted-cons (counted-member 'count) (counted-cons 'count))
(1 . 4)
> (counted-cons 'count)
5
> (counted-cons 'count)
5
```

Be careful. There are at least two dangers to avoid.
Danger A: Creating a single counter that is shared by all counted procedures.
Danger B: Creating a new counter (with count zero) every time you call a counted procedure.

More examples (and problem 2) on the back of the page

```
> (begin
  (define counted-member (counter-maker member))
  (define counted-cons (counter-maker cons))
  (counted-member 'count
                  (counted-cons 'I
                   (counted-cons 'can
                    (counted-cons 'count
                       (counted-cons 'this! '()))))))
  (counted-member 'count))
1


> (begin
  (define counted-member (counter-maker member))
  (define counted-cons (counter-maker cons))
  (counted-member 'count
                  (counted-cons 'I
                   (counted-cons 'can
                    (counted-cons 'count
                      (counted-cons 'this! '()))))))
  (counted-member 'count)
  (counted-cons (counted-member 'count) (counted-cons 'count))
)
(1 . 4)
> (begin
  (define counted-member (counter-maker member))
  (define counted-cons (counter-maker cons))
  (counted-member 'count
                  (counted-cons 'I
                    (counted-cons 'can
                     (counted-cons 'count
                       (counted-cons 'this! '()))))))
  (counted-member 'count)
  (counted-cons (counted-member 'count) (counted-cons 'count))
  (counted-cons 'count))
5
> (begin
  (define counted-member (counter-maker member))
  (define counted-cons (counter-maker cons))
  (counted-member 'count
                  (counted-cons 'I
                   (counted-cons 'can
                     (counted-cons 'count
                       (counted-cons 'this! '()))))))
  (counted-member 'count)
  (counted-cons (counted-member 'count) (counted-cons 'count))
  (counted-cons 'count)
  (counted-cons 'count))
5
> (begin
   (define fact
    (lambda (n)
      (if (zero? n)
          1
          (* n (fact (- n 1))))))
  (define counted-fact (counter-maker fact))
  (define fact-list (map counted-fact '(1 2 3 4 5 6)))
  (list fact-list (counted-fact 'count)))
((1 2 6 24 120 720) 6)
```

2. (12points) This is the same problem as #3 (stack implementation) on the written part. Here are the differences:

a) You have to actually get your code working in order to get credit.

b) At the beginning of the test-code file, I give you a major hint that was not available when you did the "on paper" problem.

The code in the test-case file should be a sufficient reminder of what the five operations are supposed to do.