CSSE 304 Exam 2 Oct29, 2019 (day 30.5) Name_____ Section 01(9:00) 02(10:00) 03(11:00)

Part 2, programming (non-interpreter). 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 have previously written.

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

Problem	Possible	Earned	Comments
C1	10		
C2	20		
Total	30		

1. (10 points) Here is the code for `snlist-filter`, which takes a predicate and an n-list (like an s-list, but with numbers instead of symbols). It returns an n-list with the same structure as its input n-list, but it leaves out all numbers that do not satisfy the predicate.

```
(define nlist-filter
  (lambda (pred? s)
    (cond [(null? s) '()]
          [(number? (car s))
           (if (pred? (car s))
               (cons (car s)
                     (nlist-filter pred? (cdr s)))
               (nlist-filter pred? (cdr s)))]
          [else (cons (nlist-filter pred? (car s))
                      (nlist-filter pred? (cdr s)))])))
```

```
> (nlist-filter even? '(((1 2 (2 3 4)) () (5 7))))
(((2 (2 4)) () ()))
```

You are to write `nlist-filter-cps`, which must be in proper tail form. You may treat all other procedures (including `pred?`) as primitives. At the beginning of the offline test code, I have provided the code for `make-k` and `apply-k`, which you should call when appropriate.

```
> (nlist-filter-cps even?
                   '(((1 2 (2 3 4)) () (5 7)))
                     (make-k list->vector))
#(((2 (2 4)) () ()))
```

Submit this problem to the E2-202010-cps **assignment on the PLC grading server.**

2. (20 points) This is basically problem 6 (add for-loop to interpreted languager) from the written part. Here are the differences:

a) You have to actually get your code working in order to get credit.
b) Most of the credit is for test cases like those on the written exam document. But there are some new tests that bring out features that are more subtle. For example,
   a. For efficiency and correctness, the expression that provides the loop variable's ending value should only be executed once.
   b. for loops can be nested. After the sample test cases (on the back of this page), I provide some code that may help you with this.

You can start with your team's A13, A14, or A16 interpreter. When you add code to one of the major procedures that has several cases, please add your new case near the top so it is easy for us to find. In most cases 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. On the written part, partial credit was given for good ideas. Here, where you have the opportunity to get feedback on your ideas from Scheme, the credit is for passing the tests.

Submit this problem to the E2-202010-interpreter **assignment on the PLC server.**
If you submit multiple files, please include chez-init.ss **in your ZIP archive. This will make grading easier.**

```
> (eval-one-exp
 '(let ([sum (list 0)])
  (for i from 2 to 5
       (set-car! sum (+ i (car sum))))
  (car sum)))
14
>
(eval-one-exp
 '(let ([sum (list 0)]
        [i 8])
    (for i from 2 to 5
       (set-car! sum (+ i (car sum))))
    (list i (car sum))))
(8 14)
>
(eval-one-exp
 '(let ([sum (list 0)])
  (for i from 2 to 7
       (if (even? i)
           (set-car! sum (+ i (car sum)))))
  (car sum)))
12
>
(eval-one-exp
 '(let ([sum (list 0)])
  (for i from 2 to 1
       (set-car! sum (+ i (car sum))))
  (car sum)))
0
>
(eval-one-exp
 '(let ([f (lambda (n)
           (let ([sum (list 0)])
             (for i from n to (* 2 n)
                  (set-car! sum (+ i (car sum))))
             (car sum)))])
  (f 3)))
18
>
(eval-one-exp
 '(let ([double-sum (lambda (n)
                      (let ([sum (list 0)])
                        (for i from 1 to n
                             (for j from 1 to i
                                  (set-car! sum (+ j (car sum)))))
                        (car sum)))])
  (double-sum 3)))
10
>
(eval-one-exp
 '(let ([double-sum (lambda (n)
                      (let ([sum (list 0)])
                        (for i from 1 to n
                             (for j from 1 to i
                                  (set-car! sum (+ j (car sum)))))
                        (car sum)))])
    (map double-sum '(1 2 3 4 5 6))))
(1 4 10 20 35 56)
>
(eval-one-exp
 '(let ([sum (list 0)] [a (list 1)])
  (for i from 2 to (begin (set-car! a (+ 2 (car a))) 4)
           (set-car! sum (+ i (car a) (car sum))))
  (car sum)))
18
```

Code that may help with nested loops:

```
(define last-used-var-number 0)
(define-syntax ++
  (syntax-rules ()
    [(_ name) (begin (set! name (add1 name)) name)]))
(define (new-var)
  (string->symbol (string-append "temp-var-" (number->string (++ last-used-var-number)))))
```