CSSE 304 Exam 1 Part 2 Dec 18, 2018 (day 14.5) Name_____ Section: 01 (9:00) 02 (9:55) 03 (10:50)

Problem	Possible	Earned	Comments
C1	8		
C2	8		
C3	8		
C4	8		
C5	8		
Total	40		

During the entire exam you may not use email, IM, cell phone, PDA, MP3 player, headphones, ear buds, or any other communication device or software. Efficiency and elegance will not affect your score, provided that we can understand your code.

You may assume that all of your procedures' input arguments have the correct types and values; your code does not need to check for illegal input data. **Mutation is not allowed in the code that you write, except for the iterator problem.**

Part 2, programming. For this part, you may use any printed or written notes, plus a Scheme programming environment, the three textbooks from the course, The *Chez* Scheme Users' Guide, plus the PLC grading program and any materials that I provided online for the course. You may do searches for built-in Scheme procedures, but not for the particular problems that you are solving. You may not use any other web or network resources, or programs written by other (past or present) RHIT students. You are allowed to look at and use any Scheme code that *you* have previously written.

Test your Part 2 code off-line before you submit to the grading server. The test cases are based on the examples given in the problem description, so there should be no surprises. If the grading program gives all of the points for a problem, that will be your score, unless you tailored your code for those specific cases. If you do not get all of the points from the server, we will look at your code, and we may give you different partial credit (either more or less) than the grading program gives you, based on how much understanding your code indicates.

Caution! It is possible to get so caught up in getting all of the points for one problem and spend so much time on it that you do not get to the other problems.

Getting the off-line test code: Code will be in a location that your instructor will specify. Download it and use it to test your code before submitting it to the PLC server.

The statements of the problems begin on the next page.

C1. (8 points) `(symmetric? matrix)` A symmetric matrix is a square matrix M such that for every row i and column j, M[i,j]=M[j,i]. The format for a matrix is the same as described in problem 1 of HW 4. You may assume that each matrix given as an argument to `symmetric?` is a square matrix (you do not have to test for this).

```
> (symmetric? '((1 2) (2 3)))
#t
> (symmetric? '((1 2) (3 3)))
#f
> (symmetric? '((1)))
#t
> (symmetric? '((1 2 3) (2 3 4) (3 4 7)))
#t
> (symmetric? '((1 2 3) (2 3 4) (3 2 7)))
#f
```

C2. (8 points) `(sum-of-depths slist)`. The *depth* of a symbol is defined as in `notate-depth` from the day 8 live coding. `sum-of-depths` finds the sum of the depths of all of the symbols in `slist`. For full credit, your code must traverse `slist` only once. You may assume that `slist` is a valid s-list.

```
> (sum-of-depths '())
0
> (sum-of-depths '(()))
0
> (sum-of-depths '(a))
1
> (sum-of-depths '((a)))
2
> (sum-of-depths '(a b))
2
> (sum-of-depths '((a () b)))
4
> (sum-of-depths '(a () ((b c))))
7
> (sum-of-depths '((() (a (b)) c ((d) () e ((f))))))
21
```

C3. (8 points) `(un-notate ls)`. `ls` a list that is returned by calling `notate-depth` on an s-list. `un-notate` returns the original s-list.

```
> (un-notate '())
()
> (un-notate '(()))
(())
> (un-notate '((a 1)))
(a)
> (un-notate '(((a 2))))
((a))
> (un-notate '((a 1) (b 1)))
(a b)
> (un-notate '(((a 2) () (b 2))))
((a () b))
> (un-notate '((a 1) () (((b 3) (c 3)))))
(a () ((b c)))
> (un-notate '((() ((a 3) ((b 4))) (c 2) (((d 4)) () (e 3) (((f 5)))))))
((() (a (b)) c ((d) () e ((f)))))
```

C4. (8 points) In the homework, you were asked to write `path-to`. To refresh your memory, I have provided the `path-to` problem description and a solution in the boxes below. The solution is also in the Exam 1 starting code that I will give you. You are to write a procedure `(find-by-path path-list slist)` that that finds the symbol obtained by following the path through `slist` that is prescribed by `path-list`. You may assume that there *is* a symbol in that location; your code does not have to test for this. To make the test-cases simpler for me to create and easier for you to check, I build each `path-list` by calling `path-to`.

Examples:

```
> (let ([slist '(a b)]
        [sym 'a])
    (find-by-path (path-to slist sym) slist))
a
> (let ([slist '(c a b)]
        [sym 'a])
    (find-by-path (path-to slist sym) slist))
a
> (let ([slist '(c a b)]
        [sym 'b])
    (find-by-path (path-to slist sym) slist))
b
> (let ([slist '(c () ((a b)))]
        [sym 'a])
    (find-by-path (path-to slist sym) slist))
a
> (let ([slist '((d (f ((b a)) g)))]
        [sym 'a])
    (find-by-path (path-to slist sym) slist))
a
> (let ([slist '((d (f ((b a)) g)))]
        [sym 'g])
    (find-by-path (path-to slist sym) slist))
g
```

Starting code for problem 4. You should not change it.

```
(define path-to
 (lambda (slist sym)
   (let pathto ([slist slist] [path-so-far '()])
     (cond [(null? slist) #f]
           [(eq? (car slist) sym)
            (reverse (cons 'car path-so-far))]
           [(symbol? (car slist))
             (pathto (cdr slist) (cons 'cdr path-so-far))]
           [else
             (or (pathto (car slist)
                         (cons 'car path-so-far))
                 (pathto (cdr slist)
                         (cons 'cdr path-so-far)))]))))
```

`(path-to slist sym)` Produces a list of `car`s and `cdr`s that (when read left-to-right) take us to the position of the leftmost occurrence of `sym` in the s-list `slist`. Notice that the returned list contains the symbols `'car` and `'cdr`, not the *car* and *cdr* procedures. Return `#f` if `sym` is not in `slist`. Only traverse as much of `slist` as is necessary to find `sym` if it is there.

```
> (path-to '(a b) 'a)
(car)
> (path-to '(c a b) 'a)
(cdr car)
> (path-to '(c () ((a b))) 'a)
(cdr cdr car car car)
> (path-to '((d (f ((b a)) g))) 'a)
(car cdr car cdr car car cdr car)
> (path-to '((d (f ((b a)) g))) 'c)
#f
```

C5. (8 points) `(make-vect-iterator v)` takes a vector `v` and returns an iterator for that vector. If `vi` is a vector iterator returned by this function, then the iterator's initial position is 0. The four methods of the iterator object are:

- `(vi 'val)` returns the vector element that is in the current position.
- `(vi 'set-val! obj)` replaces the vector element that is in the current position by `obj`.
- `(vi 'next)` adds one to the current position and does not return anything. It is illegal to call this when the current position is the last index of the vector. Your code does not have to test for illegal calls.
- `(vi 'prev)` subtracts one from the current position and does not return anything. It is illegal to call this if the current position is the first index of the vector. Your code does not have to test for illegal calls.

```
> (let* ([v '#(a b c)]
       [vi (make-vec-iterator v)]
       [x (vi 'val)]
       [y (begin (vi 'next)
                 (vi 'val))])
    (list y x))
(b a)
> (let* ([v '#(a b c)]
         [vi (make-vec-iterator v)]
         [x (vi 'val)]
         [y (begin (vi 'next)
                   (vi 'prev)
                   (vi 'val))])
   (list y x))
(a a)
> (let* ([v '#(a b c)]
         [vi (make-vec-iterator v)]
         [x (vi 'val)]
         [y (begin (vi 'next)
                   (vi 'next)
                   (vi 'set-val! 'd)
                   (vi 'val))])
  (list y x))
(d a)
> (let* ([v '#(a b c)]
       [vi (make-vec-iterator v)]
       [x (begin (vi 'next)
                 (vi 'val))]
       [y (begin (vi 'next)
                 (vi 'set-val! 'd)
                 (vi 'val))])
   (list x y))
(b d)
> (let* ([v '#(a b c)]
         [vi (make-vec-iterator v)]
         [x (begin (vi 'next)
                   (vi 'val))]
         [y (begin (vi 'next)
            (vi 'set-val! 'd)
                 (vi 'prev)
            (vi 'set-val! 'e)
            (vi 'next)
            (vi 'val))]
         [z (begin (vi 'prev)
                   (vi 'val))])
  (list x y z))
(b d e)
```

```
> (let* ([v '#(a b c)]
         [vi (make-vec-iterator v)]
         [x (begin (vi 'next)
                   (vi 'val))]
         [y (begin (vi 'next)
                   (vi 'set-val! 'd)
                   (vi 'prev)
                   (vi 'set-val! 'e)
                   (vi 'next)
                   (vi 'val))]
         [v2 (make-vec-iterator v)]
         [x2 (begin (v2 'next)
                    (v2 'val))]
         [y2 (begin (v2 'set-val! (vi 'val))
                    (v2 'val))])
    (list x y x2 y2))
(b d e d)
> (let* ([v '#(a b c)]
         [vi (make-vec-iterator v)]
         [x (begin (vi 'next)
                   (vi 'val))]
         [y (begin (vi 'next)
                  (vi 'set-val! 'd)
                  (vi 'prev)
                  (vi 'set-val! 'e)
                  (vi 'next)
                  (vi 'val))]
         [v2 (make-vec-iterator v)]
         [x2 (begin (v2 'next)
                    (v2 'val))]
         [y2 (begin (vi 'prev)
                    (vi 'prev)
                   (v2 'set-val! (vi 'val))
                   (v2 'val))])
  (list x y x2 y2))
(b d e a)
```