CSSE 304 Exam #1 Part 1 Sep 24, 2019 (day 12) Name_____ Section 01 (9:00) 02 (10:00) 03 (11:00)

Problem	Possible	Earned
1	6	
2	2	
3	4	
4	4	
5	6	
6	5	
7	6	
8	7	
Total	40	

You must turn in Part 1 before you use your computer for anything. During the entire exam you may not use email, IM, phone, tablet, headphones, ear buds, or any other communication device or software. Except where specified, efficiency and elegance will not affect your scores, provided that I can understand your code.

On both parts, assume that all input arguments will be of the correct types for any procedure you are asked to write; you do not need to check for illegal input data.
Except where specified, mutation is not allowed in code that you write for this exam.

Part 1, written. Allowed resources: Writing implement. **Sign the statement on the last page.**
Suggestion: Spend no more than 50 minutes on this part, so that you have a lot of time for the computer part.

Procedures & syntax that are sufficient for paper part of this exam:

Procedures:
Arithmetic: +, - , *, /, modulo, max, min, =, <, ≤, >, ≥
Predicates and logic: not, eq?, equal?, null?, zero?, procedure? positive?, negative?, pair?, list?, even?, odd?, number?, symbol?, integer?, member
Lists: cons, list, append, length, reverse, set-car!, set-cdr!, car, cdr, cadr, cddr, etc.
Functional: map, apply, andmap, ormap
Homework: Any procedure that was assigned for A01-A08.

Syntax:
lambda, including (lambda x …) and
(lambda (x y . z) …),
define, if, cond, and, or, let, let*, letrec, named let, begin, set! (You may not use mutation in your code unless a specific problem says you can)

No partial credit for the first two. Mystery checks for palindromes.
Example 2: there is a list of two things: the list procedure and a list of the three numbers. First call to apply results in the equivalent of
(apply list '(3 4 5)), which then becomes
(list 3 4 5).

1. (6 points) Show the output from each of these scheme inputs

```
> (let ([mystery?
         (lambda (ls)
           (andmap eq? ls (reverse ls)))])
    (list (mystery? '(a b c)) (mystery? '(a b a))))
```
____ **(#f #t)** ____

```
> (apply apply (list list (list 3 4 5)))
```
_____ **(3 4 5)** _____

```
> (procedure? and)
```
One point for #f

_____**Exception:** ____

***and* is syntax, not a variable whose value can be looked up.**

2. (**2 points**) Suppose that the code at the right has been executed. Write a Scheme expression involving the symbol `wonder` and the numbers 3 and 5 (and nothing else except parentheses) such that the value of the expression is 2.

```
(define wonder
  (lambda (x)
    (lambda (y)
      (- y x))))
```

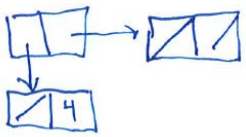
Answer: **((wonder 3) 5) One point for ((wonder 5) 3) , otherwise no partial credit.**

3. (4 points) Show the output, and draw the box-and-pointer diagram when the following Scheme code is executed:

```
> (cons (cons '() 4) (cons '() '()))
```

Output : _____ **((() . 4) ())** _____ **one point for something that's close**

Diagram: **one point for something that's close**



The rightmost `cons` makes the pair on the right. The middle `cons` makes the bottom pair. The first `cons` makes the top left pair.

Problems 4-8 have short answers. The total code in my solutions (including the *lambda* lines) is 13 lines. You can use procedures from the homework exercises by name without defining them again here.

4. (4 points) In Assignment 1, you defined a `dot-product` procedure for vectors in three dimensions. It can be defined for any number of dimensions, as the sum of the products of the corresponding vector components.

Examples:

```
(dot-product  '(2) '(3)) ➔ 6
(dot-product  '(2 4 1 3) '(3 0 7 2)) ➔ 19
```

Write `dot-product` in a very simple way by using `map` and `apply`.

```
(define dot-product   ; assume that v1 and v2 ate lists of numbers that have the same length.
  (lambda (v1 v2)
    (apply + (map * v1 v2))))
```

I can't see any other way to write this, so I can't think of how to give partial credit. If you find student code that may deserve partial credit, bring it to me.

5. (6 points) **From Assignment 3:** A *relation* is defined in mathematics to be a set of ordered pairs. The set of all items that appear as the first member of one of the ordered pairs is called the *domain* of the relation. The set of all items that appear as the second member of one of the ordered pairs is called the *range* of the relation. In Scheme, we can represent a relation as a list of 2-lists (a 2-list is a list of length 2). For example `((2 3) (3 4) (-1 3))` represents a relation with domain `(2 3 –1)` and range `(3 4)`.

A relation is a *function* if and only if there are no duplicates in the first ; elements of the 2-lists. See the examples below. You are to write the Scheme procedure `function?`. You may assume that the argument really is a relation; your code does not have to check for that.

```
(function? '()) ➔ #t
(function? '((4 5) (4 6))) ➔ #f
(function? '((3 5) (6 2) (4 7) (5 2))) ➔ #t
(function? '((1 2) (4 5) (4 6))) ➔ #f
(function? '((3 4) (4 5) (5 6) (6 3))) ➔ #t
(function? '((3 2) (2 3) (3 4))) ➔ #f
```

set? is a procedure from the homework. Students may try to do it the hard way, buy writing some helper procedures. All I can say is to try to figure out how close to correct they are and assign partial credit accordingly. Ask me if you are not sure. If you see the same thing done by several students, write it on this paper along with how much partial credit you gave.

```
(define function?
  (lambda (rel)
    (set? (map car rel))))
```

`(map car rel)` gets all of the first elements of the pairs. Calling `set?` checks for duplicates.

6. (5 points) Write a procedure `count-my-args` that counts the number of arguments that it is given.

```
> (count-my-args 3 2 7 8)
4
> (count-my-args)
0
```

This should be the easiest problem on the exam, since it is so simple, and it was a class example (in the day 9 slides, I think). I can't think of any code that merits partial credit except as noted below. If you (the grader) find a case that you think should receive partial credit, bring it to me. A few students basically wrote their own version of length and included it.

```
(define count-my-args
     (lambda args (length args)))
```

only give 1 point if student puts parens around args.

7. (6 points) Write the function *compose* which takes any number of procedures of one argument and returns a procedure that is the composition (in the order given) of those procedures. This is the procedure that was presented in class and used in HW7. We discussed an efficient version and a less-efficient version. You do not have to be concerned about efficiency here.

```
> ((compose add1 - sub1) 6)
-4
> ((compose cadr cadr) '(3 (4 5 6) (7 8)))
5
> ((compose zero?) 7)
#f
> ((compose) 5)
5
```

There are many other correct ways of doing this. See for example, the Day 9 slides. In order to receive more than 0 points, code must
(a) have a way of taking different numbers of arguments.
(b) Have at least one lambda in the body.
-3 if code fails to use apply with the recursive call to compose.
-2 if code does not actually apply procedure to x or whatever they call the variable.

```
(define compose
  (lambda fun-list
    (cond [(null? fun-list) (lambda (x) x)]
          [else (let ([composed-cdr (apply compose (cdr fun-list))])
                  (lambda (x) ((car fun-list) (composed-cdr x))))])))
```

A few students tried to use `list-recur` for this problem. I had not thought of doing that. I don't think anyone got it entirely correct, but some were close. I decided to try it. Here is my version:

```
(define compose
  (lambda procs ; the list of procedures to compose.
    ((list-recur (lambda (x) x)  ; base-val for list-recur, which is the identity function in this case.
                 (lambda (x y)   ; list-fun for list-recur. It's basically compose2.
                    (lambda (z)
                      (x (y z)))))  ; list-recur makes the recursive compose procedure.
     procs)))                       ; now apply it to the list of procedures.
```

8 . (**7 points**) `list?` does not have to be a built-in procedure in Scheme. We could write it ourselves. You will write a procedure that does the same thing that `list?` does (returns #t if and only if its argument is a proper list, #f otherwise). Using `pair?` as one of the helper procedures,, write `my-list?`. You may not use the built-in `list?` procedure in your code. This is not a trick question. The solution is straightforward and short.

```
> (my-list? 3)
#f
> (my-list? '())
#t
> (my-list? '#(4 5))
#f
> (my-list? '(3 (4 . 5) 6))
#t
> (my-list? (cons 'a 'b))
#f
> (my-list? "(1 2 3)")
#f
```

Another problem that is so simple that it's hard to figure out how to give partial credit. Of course students can use nested *if*s or *cond* instead of *and* and *or*. This problem is mainly a way of getting at "do you understand the pair datatype and what constitutes a proper list?

This problem is basically asking, "do you know what a list is?" The solution shows that while `pair?` is constant time, `list?` is not.

You must sign the following statement (unless it is not true):

I did not receive help on this exam, and I will not reveal any aspect of its content to anyone other than the instructor before 10:00 PM on August 24, 2019.

(signature) _____

```
(define my-list?
  (lambda (obj)
    (or (null? obj)
        (and (pair? obj)
             (my-list? (cdr obj))))))
```

```
(define string-index  ; in the style of list-index
  (lambda (ch str)
    (let ([len (string-length str)])
      (let loop ([pos 0])
        (cond [(>= pos len) -1]
              [(eqv? ch (string-ref str pos)) pos]
              [else (loop (add1 pos))])))))

; This is the basic merge algorithm form Weiss section 8.5,
; but it is much simpler with linked lists than with arrays.
(define merge-2-sorted-lists
  (lambda (lon1 lon2)
    (cond [(null? lon1) lon2]
          [(null? lon2) lon1]
          [(< (car lon1) (car lon2))
           (cons (car lon1)
                 (merge-2-sorted-lists (cdr lon1) lon2))]
          [else
           (cons (car lon2)
                 (merge-2-sorted-lists lon1 (cdr lon2)))])))

(define slist-equal? ; follow the grammar, and the style of day 8.
  (lambda (s1 s2)
    (cond [(null? s1) (null? s2)]
          [(symbol? (car s1))
           (and (not (null? s2))
                (symbol? (car s2))
                (eq? (car s1) (car s2)); both symbols, so const. time.
                (slist-equal? (cdr s1) (cdr s2)))]
          [else ; in this case, car is an slist.
           (or (and (null? (car s1)) (null? (car s2)))
               (and (pair? (car s2))
                    (slist-equal? (car s1) (car s2))
                    (slist-equal? (cdr s1) (cdr s2))))])))

(define (make-queue) ; represent as linked list; car is front of the queue
  (let ([first '()] [last '()]) ; references to first and last list elements
    (lambda (msg . args)
      (case msg
        [(empty?) (null? first)]
        [(enqueue) (if (null? first)
                       ; must set first and last to same list elt.
                       (begin (set! first args) ; args is list of 1 thing.
                              (set! last first))
                       (begin (set-cdr! last args) ; add to end of list.
                              (set! last (cdr last))))]
        [(dequeue) (let ([item (car first)])
                     (set! first (cdr first))
                     (if (null? first)     ; probably not really
                         (set! last '())) ; necessary to do this.
                     item)]))))
```