**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 |  |

Y**ou 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).

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)))) \_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
  
> (apply apply (list list (list 3 4 5))) \_\_\_\_\_\_\_\_\_\_\_\_\_\_

> (procedure? and)  \_\_\_\_\_\_\_\_\_\_\_\_\_\_

**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:**

**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 : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Diagram:

**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.

**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.

Use the space below if needed.

(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

(define function?

**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

(define count-my-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

(define compose

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? '())

**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) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#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

**(define my-list?**