CSSE 304 Exam #1 Part Fall 2022-2023

Your Name (write clearly and dark)

Buffalo

Problem		Possible	Earned
1		6	
2		6	
3		6	
4		4	
5		2	
6		4	
Total		28	

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 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.

Mutation is not allowed in code that you write for this exam except where noted.

Part 1, written. Allowed resources: Writing implement.

Suggestion: Spend no more than 40 minutes on this part, so that you have a lot of time for the computer part. 30 minutes is ideal.

Built-in procedures & syntax that are sufficient for this 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, filter
Handy: display, newline

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).

Do not start this exam before instructed to do so. Do write your name on both pages as soon as you get the exam.

1. **(6 points)** Consider the execution of the code below. Draw the box-and-pointer diagrams that represent the results of the `defines`. Then show what Scheme would output from the execution of each of the last three expressions. Be careful! "Almost correct" answers will usually receive no partial credit. Note that the last part can be done even if you cannot draw the two diagrams correctly.

1a (2 points).

```
(define a '(1 2 . 3))
```

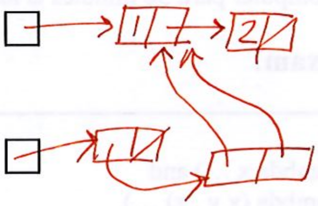
a

1b (2 points).

```
(define b1 '(1 2))
(define b2 (list (cons b1 b1)))
```

b1

b2

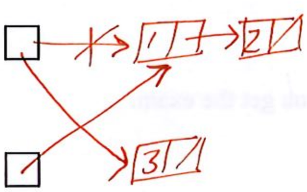


1c (2 points).

```
(define c1 '(1 2))
(define c2 c1))
(set! c1 '(3))
```

c1

c2



Name____________________

2. (6 points) Write a function listify-unary that takes a unary (i.e. one-parameter) function and returns a function that operates on lists. The returned function should apply the unary function to each element of the list. You may find it helpful to use named let or an additional helper in the solution to this problem (but there are solutions that do not require this).

```
(define add1-list (listify-unary add1))
(define neg-abs-list (listify-unary (lambda (n) (* -1 (abs n)))))

(add1-list '(1 2 10)) ;; yields (2 3 11)
(neg-abs-list '(-1 2 3)) ;; yields (-1 -2 -3)
```

(define listify-unary

```
(lambda (proc)
  (lambda (lst)
    (map proc lst)))
```

3. (6 points) Consider a list that contains numbers and lists of numbers. We want a function that removes the numbers and makes the lists "double lists" i.e. single element lists containing the original lists.

```
(i-love-lists '((1 2) 3 4 (5))) ;; yields (((1 2)) ((5)))
```

Write an implementation of `i-love-lists` using some combination of `map filter apply` (not all will be needed). Do not use any looping or recursion constructs.

(define i-love-lists

```
(lambda (lst)
  (map list (filter list? lst))))
```

4. (4 points) Here some code that uses `let` and `lambda` in an interesting way. What does this code print out when run?

```
(define triple
  (let ([c1 0])
    (lambda ()
      (let ([c2 10])
        (lambda (value)
          (let ([c3 100])
            (set! c1 (+ c1 value))
            (set! c2 (+ c2 value))
            (set! c3 (+ c3 value))
            (display (list c1 c2 c3))
            (newline)))))))

(define t1 (triple))
(define t2 (triple))
(t1 1)
(t2 2)
(t1 3)
```

(1 11 101)

(3 12 102)

(6 14 103)

5. (2 points)

Consider the lambda calculus expression

`(x (lambda (x) (lambda (y) (lambda (z) (x y)))))`

In that expression, which variables occur bound? x y occur free? x

6. (4 points)

Consider a simplified version of the equation grammar we discussed in class.

```
<exp>    ::= <exp> + <term>   |  <term>
<term>   ::= <term> * <factor> | <factor>
<factor> ::= ( <exp> )   | <number>
```

Draw the derivation tree for the expression:

`1 * 2 * 3`

Note that exp is the start symbol of this grammar. Also feel free to use E T F N rather than exp term factor number in your tree.