CSSE 304 Exam #1 Part 1 Jan 8, 2020 (day 14.5) Name_____ Section 01 (10:00) 02 (11:00) 03 (12:00)

Problem	Possible	Earned
1	10	
2	5	
3	5	
4	4	
5	6	
Total	30	

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

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
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. **(10 points, 2 for each part)** Consider the execution of the code below. Draw the box-and-pointer diagrams that represent the results of the two `define`s. 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 three parts can be done even if you cannot draw the two diagrams correctly.

```
> (define a '(((())))
> (define b (list (car a) a (cdr a)))
```

a ☐

b ☐

```
> b
```

```
> (procedure? ((lambda (x) (lambda (y) (+ x y))) 5))
```

```
> (procedure? and)
```

2. (5 points) In assignment A6, you were asked to write `curry2`. The `curry2` procedure takes as its argument a procedure `f` that expects two parameters. Then `(curry2 f)` returns a procedure that takes the two parameters one at a time. My solution:

```
(define curry2
  (lambda (f)
    (lambda (x)
      (lambda (y)
        (f x y)))))
```

For this problem, you must write an inverse for `curry2`. `uncurry2` takes a curried procedure and produces a corresponding procedure of two arguments. If `g` is any procedure for which `((g a) b)` makes sense, then the return value of `((uncurry2 g) a b)` is equal to the return value of `((g a) b)`. Another way of saying the same thing is that if **`f`** is a procedure for which `(f a b)` makes sense, then `((uncurry2 (curry2 f)) a b)` is equal to `(f a b)`.

Examples:

```
(let ([f (lambda (x)
           (lambda (y)
             (list y x)))])
  ((uncurry2 f) 6 7))                          ➔ (7 6)

((uncurry2 (curry2 cons)) 4 '())               ➔ (4)
```

You are to fill in the rest of the definition of `uncurry2`. Be *very careful* about parentheses placement.
Incorrect parens will probably result in zero points for this problem.

```
(define uncurry2      ; The correct answer is almost entirely about where the parentheses go.
```

3. (5 points) Given the definitions in the box at the right:

What is the value of `(mystery 4 3)`?

```
(define atoi ; not related to the C function that has the same name
  (lambda (n)
    (if (zero? n)
        '()
        (cons (- n 1) (atoi (- n 1))))))
(define mystery
  (lambda (m  n)
    (map (lambda (x) (+ m x))
         (atoi n))))
```

4. (4 points) Here is the stack "object" code from Assignment 9:
We discussed the following questions in class.

```
(define make-stack
 (lambda ()
  (let ([stk '()])
   (lambda (msg  . args )
    (case msg
      [(empty?) (null? stk)]
      [(push)   (set! stk (cons (car args) stk))]
      [(pop)    (let ([top (car stk)])
                  (set! stk (cdr stk))
                   top)]
      [else (errorf 'stack
                   "illegal message")])))))
```

How will stacks behave differently if we
(a) move the `let` so it comes **before** both `lambda`s (and leave the rest of the code unchanged)?

(b) move the `let` so it comes **after** both `lambda`s (and leave the rest of the code unchanged)?

```
<LcExpr> ::=
 <identifier> |
 (lambda (<identifier>) <LcExpr>) |
 ( <LcExpr> <LcExpr> )
```

5. (6 points) Here is our original grammar for lambda-calculus expressions:

Consider the expression `(lambda (x) ((lambda (y) (y z)) z))`

(a) **(2 points)** In that expression, which variables occur bound? _____ occur free? _____

(b) (4 points) Draw the derivation tree for that expression. To make it easier to draw this (as I did on the whiteboard in the Day 11 class), you are allowed to write **L** in place of `<LcExpr>`, **λ** in place of `lambda`, and **id** in place of `<identifier>`.