CSSE 304 Exam #1 Part Fall 2021-2022 Name KEY Section 02 (11:00) 03 (3:00)

Problem		Possible	Earned
1		8	
2		6	
3		6	
4		4	
5		6	
Total		30	

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.
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, filter
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).

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. **(8 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.

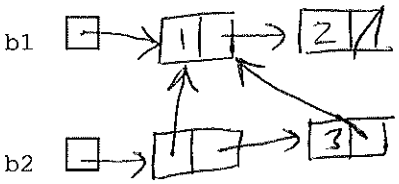
1a (2 points).

```
(define a '(() ()))
```

a 

1b (4 points).

```
(define b1 '(1 2))
(define b2 (cons b1 (cons 3 b1)))
```



1c (2 points). Given the defines above, what would be the output of `(display b2)`. If it would produce some kind of error, write ERROR.

((1 2) 3 1 2)

Name KEY

2. **(6 points)** So although scheme does not have static typing, it does have predicates like `number?` and `symbol?` which allow you to check the types of things. `list?` allows you to check that something is a list, but if you want to have a list of numbers it's a little tricky to do that check. Let's write a procedure that allows us to build new type checking procedures. It'll be called `list-of` and it will be used like this:

```
(define list-of-numbers? (list-of number?))
(define list-of-symbols? (list-of symbol?))

(list-of-numbers? '(1 2 a)) ;; yields #f
((list-of string?) '("hello" "world")) ;; yields #t
```

(define list-of

```
(lambda (pred)
  (lambda (list)
    (and map pred list)))
```

3. **(6 points)** Imagine I keep of scheme list representing the age of leftovers in my fridge as a list of numbers. I want to write a function age-leftovers that I run when a day passes. However, when leftovers become more than 6 days old, I throw them out (and therefore they ought to be removed from the result list).

```
(age-leftovers '(1 2 6 4 6)) ;; yields (2 3 5)
```

Write an implementation of `age-leftovers` using `map` and `filter`. Do not use any looping or recursion constructs. If you are unable to do that, you can get 3/5 credit for a version that only uses `map` and replaces too old leftovers with the symbol `'yuck` rather than removing them from the list.

(define age-leftovers

```
(lambda (list)
  (map (lambda (x) (+ 1 x))
       (filter (lambda (x) (< x 6)) list)))
```

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

```
(define make-thingy
  (let ((a 'apple))
    (lambda ()
      (let ((b 'banana))
        (lambda (a2 b2 c2)
          (let ((c 'cherry))
            (display (list a b c))
            (newline)
            (set! a a2)
            (set! b b2)
            (set! c c2)))))))

(let ([thingy1 (make-thingy)] [thingy2 (make-
thingy)])
  (thingy1 'atlanta 'boston 'chicago)
  (thingy2 'ant 'bear 'cat)
  (thingy1 'aaa 'bbb 'ccc))
```

(apple banana cherry)
(atlanta banana cherry)
(ant boston cherry)

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

5. (6 points) Our original grammar for lambda-calculus expressions:

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

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

(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>`.