CSSE 304 Exam #1 Part Fall 2024-2025

Your Name (write clearly and dark)

Key

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 30 minutes on this part, so that you have a lot of time for the computer part. 20 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.

Name_____

1. **(6 points)** Write a function "take a list" that takes a multiparameter scheme procedure and returns a version of that procedure that takes a single argument which is a list. Hint: you'll need apply.

```
(define boring-+ (take-a-list +))

(boring-+ '(1 2 3)) ;; yields 6
(boring-+ 4 5 6) ;; errors
((take-a-list append) '((a) (b))) ;; yields (a b)
```

```
(define take-a-list
  (lambda (proc)
    (lambda (lst)
      (apply proc lst))))
```

2. **(6 points)** Consider a list of numbers. We want to find the largest (i.e. closest to zero) negative number. You can assume there will be at least one negative number in the list.

```
(largest-negative '(-5 -2 1 3)) ;; yields -2
```

Write an implementation of `largest-negative` using some combination of `map filter apply` (not all will be needed). Do not use looping or recursion.

```
(define largest-negative
  (lambda (lst)
    (apply max (filter negative? lst))))
```

5. **(4 points)**

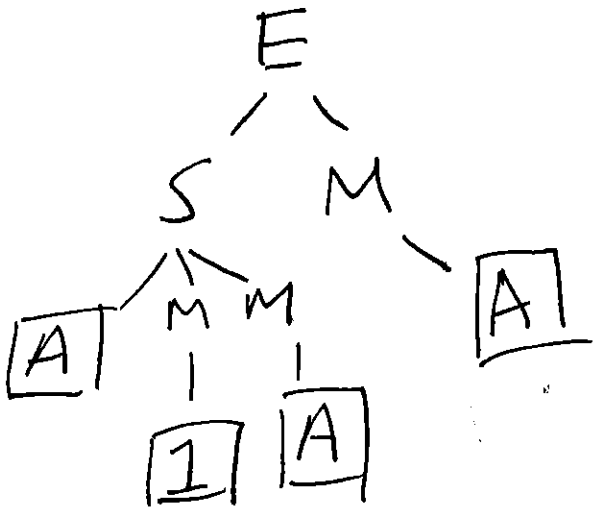
Consider the following grammar:

```
<exp>    ::= A <mod> | <sub-exp> <mod>
<sub-exp>   ::= A <mod> <mod>
<mod> ::= A | <number>
```

Draw the derivation tree for the expression:

A 1 A A

Note that exp is the start symbol of this grammar. Also feel free to use E S M rather than exp sub-exp mod in your tree.



3. (4 points) Here some code that uses `let` and `lambda` in an interesting way. What does this code print out when run?

```
(define pool
  (let ([p 10])
    (lambda ()
      (let ([a 0])
        (lambda ()
          (let ([b 0])
            (set! p (- p 2))
            (set! a (+ a 1))
            (set! b (+ b 1))
            (display (list p a b))
            (newline)))))))

(define x (pool))
(define y (pool))
(x)
(x)
(y)
(x)
```

(8 1 1)

(6 2 1)

(4 1 1)

(2 3 1)

4. (2 points)

Consider the lambda calculus expression

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

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

6. **(6 points)**

Consider the following grammar:

```
<exp>    ::= (lambda ({<iden>}*) <exp>) | <iden>
<iden>   ::= x | y
```

Write yes next to the strings that are in this grammar, no next to the ones that are not.

`(lambda (x y y) x)`	yes
`(x x)`	no
`x`	no
`(lambda (x y) x y)`	no
`(lambda (x) (lambda (y) y))`	yes
`(lambda () x)`	yes