CSSE 304 Exam #1 Part Winter 2022-2023

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).

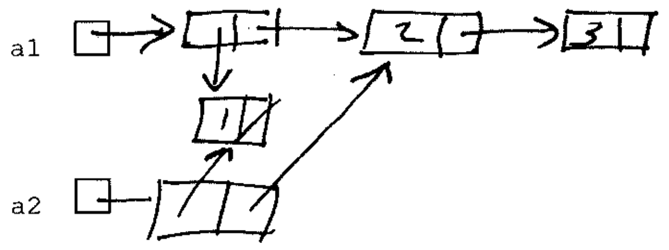
Do not start this exam before instructed to do so. Do write your name on both pages as soon as you get the exam.

1. (6 points) Consider the execution of the code below. Draw the box-and-pointer diagrams that represent the results of the `defines`. Be careful! "Almost correct" answers will usually receive no partial credit.

1a (2 points).

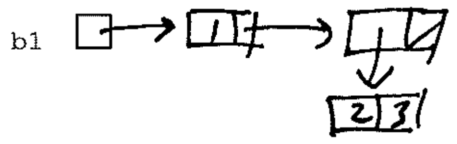
```
(define a1 '((1) 2 3))

(define a2 (cons (car a1) (cdr a1)))
```



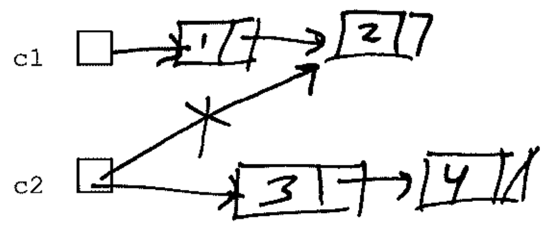
1b (2 points).

```
  (define b1 '(1 (2 . 3)))
```
* be careful this is not the same as '(1 2 . 3)



1c (2 points).

```
  (define c1 '(1 2))
  (define c2 (cdr c1))
  (set! c2 (list 3 4))
```



Name____________________

2. **(6 points)** Write a function 2unary->1binary that takes two unary (i.e. one-parameter) functions and returns a two parameter function that returns a 2 element list. The first element of the result should be the first unary function applied to the first parameter. The second element of the result should be the second unary function applied to the second parameter.

```
(define addsub1 (2unary->1binary add1 sub1))

(addsub1 10 20) ;; yields (11 19)
((2unary->1binary car cdr) '(a b) '(c d)) ;; yields (a (d))
```

```
(define 2unary->1binary
    (lambda (v1 v2)
        (lambda (p1 p2)
            (list (v1 p1) (v2 p2)))))
```

3. **(6 points)** Consider a list of lists of numbers. We want a function that returns the total length of all the sublists.

```
(total-length '((6 7) (99 100) (1000)) ;; yields 5
```

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

```
(define total-length
    (lambda (ls)
        (apply + (map length ls))))
```

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

```
(define foo
  (let ((x 10))
    (lambda ()
      (let ((y 0))
        (lambda (z)
          (set! x (- x z))
          (set! y (+ y z))
          (display (list x y)))))))

(define one (foo))
(define two (foo))

(one 2)
(two 5)
(one 2)
```

(8 2)
(3 5)
(1 4)

5. **(2 points)**

Consider the lambda calculus expression

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

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

6. **(4 points)**

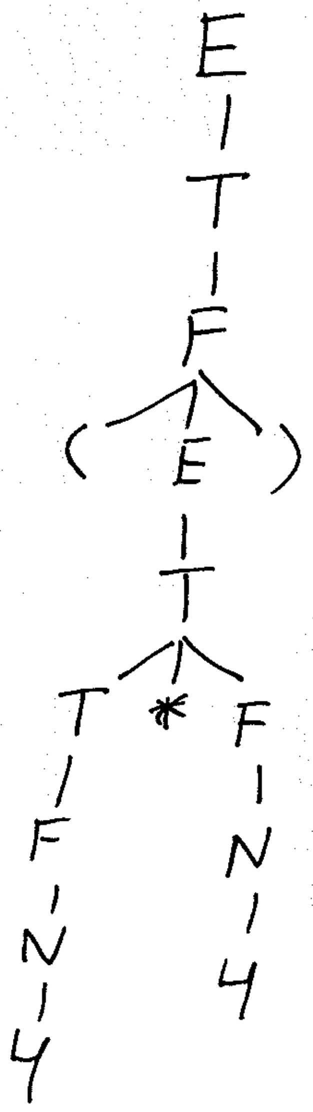
Consider a simplified version of the equation grammar we discussed in class.

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

Draw the derivation tree for the expression:

```
(3 * 4)
```

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.


E
T
F
(
E
)
T
T
*
F
F
N
N
4
4