CS270: LAB #4

Higher Order Functions

You may work in groups of one or two people (three is acceptable in the event of an unscheduled absence). Unless stated otherwise, the lab is due to be submitted into Gradescope at the end of the day.
In order to receive credit, follow these instructions:

[a] Every team member should be discussing simultaneously the same problem – do NOT try to divvy up the labor and assign different problems to different students since the material is cumulative.

[b] Directly edit this lab PDF using Sedja/PDFescape with your answers (extra pages can be added in the rare event you need more than the allotted space)

[c] Each lab, rotate which member has the responsibility of being the Scribe. This is the person that is typing the answers and uploading the final PDF – note that only a single copy of the filled in PDF is turned into Gradescope. Only one lab needs to be submitted for the entire team, and all members receive the same score. Make sure to use a font that your PDF editor is compatible with (otherwise you might find your answers appear as weird shapes/sizes or simply disappear entirely!)

[d] The Gradescope submission must have each answer properly tagged with the appropriate question. Moreover, every member of the team must be listed as a submitter. Although it is the Scribe which executes these actions, it is still the responsibility of the entire team to make certain this is done properly (thus it is highly recommended that the Scribe share their screen so the entire team can witness it). Answers which are improperly tagged cannot be seen by the grader and thus cannot be scored.

[e] **FOR REMOTE ONLY**: Each lab, rotate which member has the responsibility of being the Recorder. This is the person who hits the Zoom Record button (once the technical permission is granted by the TA/RCF/Professor) and ensures that everyone has their camera/microphone on. They are also the member that is responsible to make sure the DrexelStream video is marked as viewable and entered into the https://tinyurl.com/VidLinkForm webform before 11:59pm (they should also email the rest of their team as confirmation.) Note that the video file doesn't get created/processed until after the Recorder has quit Zoom.

[f] Each lab, rotate which member has the responsibility of being the Manager. This is the person that ensures that everyone is participating equally and honestly, keeps the group on task, ensures that all team members understand a solution before going on to the next question, and presses the "hand up" button in Zoom to summon a TA or the professor (but they only do so after surveying the group to make sure everyone has the same question).

Team Name (CS pioneer):	Tim Berners-Lee
Scribe name:	Brendan Hoag
Recorder name:	Cole Bardin
Manager name:	Jackson Masterson
Other team member (if any):	Jeremy Mathews

Question 1 : 12 points

Not all functions need to have names. An **anonymous function** (also called **a lambda expression**) is a function that is not given a name.

In Racket, we can create an **anonymous function** using the **lambda** command.

```
(lambda (x) (+ x 2))
```

This command returns a #<procedure> in Racket. This means the object it returns is of the type procedure. It creates a function.

The syntax of the lambda function is

```
(lambda (inputs) (function body) )
```

We can give the lambda inputs and it will execute.

```
( (lambda (x) (+ x 2)) 7) ;Returns 9
```

To see how the Lambda command executes, we will provide a proof by equational reasoning.

Prove: The Racket Expression ((lambda (x) (+ x 2)) 7) will return 9.

1.	((lambda (x) (+ x 2)) 7)	Premise
2.	(+ 7 2)	Apply Lambda Function
3.	9	Evaluate Addition

(a) (2 points) In Racket, execute ((lambda (x) (∗ x x)) 4). What value is returned?

16

(b) (2 points) In Racket, execute ((lambda (x y) (+ x y)) 3 4). What value is returned?

7

(c) (2 points) In Racket, execute ((lambda (x) (if (> x 2) 3 5)) 8). What value is returned?

3

(d) (6 points) Use equational reasoning to prove ((lambda (x) (if (> x 2) 3 5)) 1) evaluates to 5.

1.	((lambda (x) (if (> x 2) 3 5)) 1)	Premise
2.	(if (> 1 2) 3 5)	Apply lambda function
3.	(if #f 3 5)	Evaluate greater than
4.	5	Evaluate if

Question 2 : 12 points

We can create functions that take other functions as inputs.

The following function takes as input a function with 1 input and a number. It applies the function twice.

```
(define (applyTwice f x)
  (f (f x))
)
```

Prove: The Racket Expression (applyTwice (lambda (x) (∗ x x)) 2) will return 16.

1.	(applyTwice (lambda (x) (* x x)) 2)	Premise
2.	((lambda (x) (* x x)) ((lambda (x) (* x x)) 2))	Apply Def. of applyTwice
3.	((lambda (x) (* x x)) (* 2 2))	Apply Lambda Function
4.	((lambda (x) (* x x)) 4)	Evaluate Multiplication
5.	(* 4 4)	Apply Lambda Function
6.	16	Evaluate Multiplication

(a) (2 points) In Racket, execute (applyTwice (lambda (y) (+ 2 y)) 5). What value is returned?

9

(b) (2 points) In Racket, execute (applyTwice (lambda (a) (− a 2)) 5). What value is returned?

1

(c) (2 points) In Racket, execute (applyTwice (lambda (a) (− 2 a)) 5). What value is returned?

5

(d) (6 points) **Prove**: The Racket Expression (applyTwice (lambda (a) (− 2 a)) 7) will return 7.

1. (applyTwice (lambda (a) (- 2 a)) 7)	Premise
2. ((lambda (a) (- 2 a)) ((lambda (a) (- 2 a)) 7))	Apply Def. of applyTwice
3. ((lambda (a) (- 2 a)) (- 2 7))	Apply lambda function
4. ((lambda (a) (- 2 a)) -5)	Evaluate subtraction
5. (- 2 -5)	Apply lambda function
6. 7	Evaluate subtraction

Question 3 : 15 points

We have already seen some recursive patterns to use with lists.

Now that we can pass function into other functions, we can create **High Order Functions**. The applyTwice function is an example of a **High Order Function**. A **High Order Function** is a function that takes other functions as inputs or returns functions as outputs.

We want to create a function that generates a sequence of numbers from **a** to **b**.

```
(define (seq a b)
        (if (equal? a b)
                (cons a '())
                (cons a (seq (+ a 1) b))
        )
)
(seq 0 10) ;Returns '(0 1 2 3 4 5 6 7 8 9 10)
```

That works fine. What if you want to generate a sequence of even numbers? You could write a new function. Instead, we can write a version that takes as input a function that generates elements of the sequence. Then use it to solve multiple problems.

```
(define (seq2 f a b)
        (if (equal? a b)
                (cons (f a) '())
                (cons (f a) (seq2 f (+ a 1) b))
        )
)
(seq2 (lambda (x) (+ 0 x)) 0 10) ; Returns '(0 1 2 3 4 5 6 7 8 9 10)
(seq2 (lambda (x) (* 2 x)) 0 10) ; Returns '(0 2 4 6 8 10 12 14 16 18 20)
```

(a) (5 points) Give a Racket Expression using the seq2 command to generate the list '(1 3 5 7 9 11 13 15 17 19 21).

(seq2 (lambda (x) (- (* x 2) 1)) 1 11)

(b) (5 points) Give a Racket Expression using the seq2 command to generate the list '(0 −1 −2 −3 −4 −5).

(seq2 (lambda (x) (* -1 x)) 0 5)

(c) (5 points) Give a Racket Expression using the seq2 command to generate the list '(0 1 4 9 16 25).

(seq2 (lambda (x) (* x)) 0 5)

Question 4 : 9 points

We can also use **anonymous functions** to write functions that create other functions.

The function computeTax takes a tax rate as input. It returns a function that computes the price of an item with tax added.

```
(define (computeTax percent)
  (lambda (price) (+ price (* price percent)))
)
```

This function can be used to create different functions for different percentages.

```
(define sixPrecent (computeTax 0.06))
(define sevenPrecent (computeTax 0.07))
(sixPrecent 100) ; Returns 106.0
(sevenPrecent 100) ; Returns 107.0
```

Review the following **High Order Function**.

```
(define (mystery x)
        (lambda (y) (+ x y)))
```

(a) (1 point) What is the return value of ((mystery 10) 7)?

17

(b) (1 point) What is the return value of ((mystery 9) −5)?

4

(c) (3 points) What is the return value of (seq2 (mystery 5) 0 5)?

'(5 6 7 8 9 10)

(d) (4 points) Use seq2 and mystery to write a Racket Expression that evaluates to '(3 4 5 6 7).

(seq2 (mystery 1) 2 6)

Question 5 : 18 points

The function (map f L) is built into Racket. It takes two arguments:

1. f a function of a single variable
2. L a list of elements in the domain of f

If L = (x1 x2 ... xn), the output of the function is the list ((f x1) (f x2) ... (f xn)). Map creates a new list by applying the function to each value in the list.

Assume, we have defined (define (sqr x) (∗ x x)). The result of (**map** sqr ‘(1 2 3 4)) is ‘(1 4 9 16). It makes a list (1^2 2^2 3^2 4^2).

Assume the following functions are defined. (Do not actually enter them into DrRacket, just find the answer manually)

```
(define (sqr x) (∗ x x))
(define (dbl x) (+ x x))
(define (trp x) (∗ 3 x))
(define (neg x) (− 0 x))
```

(a) (2 points) What does (**map** sqr ’(1 2 3 4 5)) return?

'(1 4 9 16 25)

(b) (2 points) What does (**map** sqr ’(2 5 9)) return?

'(4 25 81)

(c) (2 points) What does (**map** dbl ’(1 2 3 4 5)) return?

'(2 4 6 8 10)

(d) (2 points) What does (**map** trp ’(1 2 3 4)) return?

'(3 6 9 12)

(e) (2 points) What does (**map** neg ’(1 2 3 4 5)) return?

'(-1 -2 -3 -4 -5)

(f) (2 points) What does (**map** neg ’(−1 1 −1 1)) return?

'(1 -1 1 -1)

(g) (6 points) Define your own version of the map function(define (myMap f X) ...) in Racket. Give the code below.

```
(define (myMap f X)
  (if (null? X) null (cons (f (first X)) (myMap f (rest X)))))
```

Question 6 : 24 points

Racket has two functions to combine the elements of a list. Both have the same three inputs.

1. f a function of two variables
2. *init* the value to be returned when L = ‘()
3. L a list of elements in the domain of f

Calling (foldr − 0 ’(1 2 3 4)) computes −2

This function computes (− 1 (− 2 (− 3 (− 4 0)))).

Calling (foldl − 0 ’(1 2 3 4)) computes 2

This function computes (− 4 (− 3 (− 2 (− 1 0)))). You may verify you answers in DrRacket, but find them manually first.

(a) (3 points) What does (foldr + 2 ’(2 4 6)) return?

14

(b) (3 points) What does (foldl + 1 ’(1 3 5)) return?

10

(c) (3 points) What does (foldr ∗ 1 ’(2 4 6)) return?

48

(d) (3 points) What does (foldl ∗ 0 ’(2 3 4)) return?

0

(e) (3 points) What does (foldr − 3 ’(2 3 4 5)) return?

1

(f) (3 points) What does (foldl − 3 ’(2 3 4 5)) return?

5

(g) (6 points) Define your own version of the function foldr (define (myFoldr f init X) ...) in Racket. Give the code below.

```
(define (myFoldr f init X)
  (cond
    [(null? X) init]
    [else (f (first X) (myFoldr f init (rest X)))]
  )
)
```

Question 7 : 10 points

(a) (2 points) Write a function (define (neg? x) ...) that returns 1 if the number is negative and 0 otherwise.

```
(define (neg? x)
  (if (< x 0) #t #f))
```

(b) (4 points) Write a racket expression that uses map to apply your neg? function to the list $(1\ -3\ -4\ 5\ 9)$.

```
(map neg? '(1 -3 -4 5 9))
```

(c) (4 points) Write a function called QtyNegs to count the number of negative numbers in a list. Do not use any helpers, and this time do not explicitly call the neg? function you just wrote. Instead, use higher higher order functions such as map, foldr, and λ. Include a simple input contract that insures the input is a list, and a simple output contract guarantee that the answer will be an integer.

```
(define/contract (QtyNegs l) (-> list? integer?)
  (foldl (λ (x y) (+ x y)) 0 (map (λ (x) (if (< x 0) 1 0)) l)))
```