CMSC 131A, Midterm 2 (Practice)
SOLUTION

Fall 2017

NAME:_____

UID: _____

Question	Points
1	12
2	12
3	15
4	15
5	15
6	10
7	15
Total:	94

This test is open-book, open-notes, but you may not use any computing device other than your brain and may not communicate with anyone. You have 50 minutes to complete the test.

The phrase "design a program" or "design a function" means follow the steps of the design recipe. Unless specifically asked for, you do not need to provide intermediate products like templates or stubs, though they may be useful to help you construct correct solutions.

You may use any of the data definitions given to you within this exam and do not need to repeat their definitions.

Unless specifically instructed otherwise, you may use any built-in ISL+ functions or data types.

When writing tests, you may use a shorthand for writing check-expects by drawing an arrow between two expressions to mean you expect the first to evaluate to same result as the second. For example, you may write `(add1 3)` $\rightarrow$ 4 instead of `(check-expect (add1 3) 4)`.

Problem 1 (12 points). Each of the following signatures describes a class of functions:

```
Boolean String -> String
[Boolean -> String] -> String
[Listof Boolean] [Listof Number] -> Number
[X] . [Number -> X] -> X
[X Y] . [X -> Y] [Listof X] -> [Listof Y]
[X Y Z] . [X -> Y] [Y -> Z] X -> Z
```

Define one example of a function in each class. (You only need to provide the code, not a full design.)

SOLUTION:

```
(define (f b s) s)
(define (g fs) (fs #false))
(define (h bs ns) 2)
(define (i f) (f 2))
(define (j f xs) (map f xs))
(define (k f g x) (g (f x)))
```

Problem 2 (12 points). For each of the following functions, provide the most general signature that correctly describes the function:

```
;; f : Number -> Number

(define (f x)
  (+ (sqr x) 2))

;; g : [Listof Number] -> [Listof Number]

(define (g x)
  (map add1 x))

;; h : [Listof Number] [Listof Number] -> [Listof Number]

(define (h x y)
  (cond [(empty? x) y]
        [(cons? x)
         (cons (sqr (first x))
               (h (rest x) y))]))

;; i : (Number -> Number) -> Number

(define (i x)
  (sqr (x 5)))

;; j : Number -> (Number -> Number)

(define (j x)
  (lambda (y)
    (+ x y)))

;; k : [X] (Natural -> X) Natural -> [Listof X]

(define (k x y)
  (cond [(= y 0) (list (x 0))]
        [else
         (cons (x y) (k x (sub1 y)))]))
```

3

Problem 3 (15 points). Design a program called avg that computes the average of a list of numbers.

SOLUTION:

```
;; avg : [Listof Number] -> Number
;; Compute the average of a list of numbers
;; Assume: the list is non-empty
(check-expect (avg (list 3 4 5)) 4)
(check-expect (avg (list 10 20)) 15)
(define (avg lon)
  (/ (sum lon)
     (length lon)))

;; sum : [Listof Number] -> Number
;; Sum the list of numbers
(check-expect (sum '()) 0)
(check-expect (sum (list 3 4 5)) 12)
(define (sum lon)
  (foldr + 0 lon))

;; Alt: follow [Listof Number] template
```

Problem 4 (15 points). Here is a parametric data definition for a non-empty list of items:

```
;; A [NEListof X] is one of:
;; - (cons X '())
;; - (cons X [NEListof X])
;; Interp: an arbitrarily long but non-empty sequence of items.
```

Here are two similar programs that operate over non-empty lists:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; smallest : [NEListof Number] -> Number
;; Produce the smallest number in the list
(check-expect (smallest (list 6 3 2 4)) 2)
(define (smallest xs)
  (cond [(empty? (rest xs)) (first xs)]
        [else
          (min (first xs)
               (smallest (rest xs)))]))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; longest : [NEListof String] -> String
;; Produce the first longest string in the list
(check-expect (longest (list "hi" "there")) "there")
(check-expect (longest (list "same" "size")) "same")
(define (longest xs)
  (cond [(empty? (rest xs)) (first xs)]
        [else
          (longer (first xs)
                  (longest (rest xs)))]))

;; longer : String String -> String
;; Select the longer of the two strings (s1 if tied)
(check-expect (longer "hi" "there") "there")
(check-expect (longer "same" "size") "same")
(define (longer s1 s2)
  (cond [(< (string-length s1) (string-length s2)) s2]
        [else s1]))
```

Design an abstraction of the `smallest` and `longest` functions and redefine `smallest` and `longest` in terms of it. Be sure to give the most general parametric signature for your abstraction function.

[Space for problem 4.]
SOLUTION:

```
;; mostest : [X] [NEListof X] [X X -> X] -> X
;; Produce the first most-est element in the list
(define (mostest xs most)
  (cond [(empty? (rest xs)) (first xs)]
        [else
          (most (first xs)
                (mostest (rest xs) most))]))

(define (longest los)
  (mostest los longer))

(define (smallest lon)
  (mostest lon min))
```

Problem 5 (15 points). Design a program that takes a list of numbers and produces the count of positive numbers in the list. For example, if the list contains 3, 4, -1, 0, -2, and 5, the count of positive numbers is 3. You may assume the [Listof Number] data definition is defined. For full credit, use list abstraction functions. For partial credit, follow the template for [Listof Number].

SOLUTION:

```
;; count-pos : [Listof Number] -> Number
;; Count the number of non-negative numbers in given list
(check-expect (count-pos '()) 0)
(check-expect (count-pos (list 3 -1 0 2)) 2)
(define (count-pos lon)
  (foldr (lambda (n c) (if (positive? n) (add1 c) c)) 0 lon))

;; Alt:
(define (count-pos lon)
  (length (filter positive? lon)))
```

Problem 6 (10 points). It's been said that `foldr` is *the* fundamental list abstraction function because any function that follows the list template can be expressed in terms of `foldr`. Put that theory to the test by providing an equivalent definition for the following function in terms of `foldr`:

```
;; map : [X Y] . [X -> Y] [Listof X] -> [Listof Y]
;; Apply f to each element of list, collect a list of results
(define (map f ls)
  (cond [(empty? ls) '()]
        [(cons? ls)
         (cons (f (first ls))
               (map f (rest ls)))]]))
```

(Just provide code, not a full design.)

SOLUTION:

```
(define (map f xs)
  (foldr (lambda (x ys) (cons (f x) ys)) '() xs))
```

Problem 7 (15 points). A sports tournament in which teams pair off against eachother and the winner advances to the next level in the tournament can be modelled with the following data definition:

```
;; A Tournament is one of:
;; - String
;; - (make-bracket Tournament Tournament)
;; Interp: a sports tournament where a String is a team name
;; that is still in the tournament and a bracket represents
;; a match between the winner of two tournaments.
(define-struct bracket (t1 t2))
```

Design a function `most-games` that computes the maximum number of games any team in the tournament would have to play in order to win the tournament.

SOLUTION:

```
;; most-games : Tournament -> Number
;; Compute most games any team has to win to win tournament
(check-expect (most-games "NY") 0)
(check-expect (most-games (make-bracket "LA" "HOU")) 1)
(check-expect (most-games (make-bracket (make-bracket "LA" "DC") "HOU")) 2)
(define (most-games t)
  (cond [(string? t) 0]
        [(bracket? t)
         (add1 (max (most-games (bracket-t1 t))
                    (most-games (bracket-t2 t))))]))
```