CMSC 131A, Midterm 2
SOLUTION

Fall 2017

NAME:_____

UID: _____

Question	Points
1	15
2	12
3	20
4	15
5	15
6	20
Total:	97

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 (15 points). Here are some data definition relevant to representing a dictionary, which associates words with the their definitions:

```
;; A Dict is one of:
;; - '()
;; - (cons (list String String) Dict)
;; Interp: a collection of definitions where each element is a
;; two-element list of a word (first) and its meaning (second).

;; A Result is one of:
;; - #false
;; - String
;; Interp: represents the result of looking up a word in the
;; dictionary; #false means not found.
```

Design the following function:

```
;; lookup : String Dict -> Result
;; Lookup the (first) definition of a word in the dictionary,
;; produces #false if not found
```

SOLUTION:

```
(check-expect (lookup "a" '()) #false)
(check-expect (lookup "a" (list (list "a" "b"))) "b")
(check-expect (lookup "a" (list (list "c" "b") (list "a" "d"))) "d")
(define (lookup w d)
  (cond [(empty? d) #false]
        [(cons? d)
         (if (string=? (first (first d)) w)
             (second (first d))
             (lookup w (rest d)))]))
```

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

```
;; method : Number -> Number

(define (method x)
  (+ (sqr x) 2))

;; raekwon : [Listof Number] -> Boolean

(define (raekwon x)
  (ormap positive? x))

;; rza : [Listof String] [Listof Number] -> [Listof Number]

(define (rza x y)
  (cond [(empty? x) y]
        [(cons? x)
         (cons (string-length (first x))
               (rza (rest x) y))]))

;; gza : (String Number -> Number) -> Number

(define (gza x)
  (foldr x 0 (list "a" "b" "c")))

;; odb : (Number -> Number) -> (Number -> Number)

(define (odb x)
  (lambda (y)
    (/ (- (x (+ y 0.001))
          (x (- y 0.001)))
       (* 2 0.001))))


;; ghostface : [X Y] [Listof X] [X -> Y] -> [Listof Y]

(define (ghostface x y)
  (cond [(empty? x) '()]
        [(cons? x)
         (cons (y (first x))
               (ghostface (rest x) y))]))
```

Problem 3 (20 points). Design a program called `w-avg` that computes the weighted average of a list of numbers and a list of weights.

For example, let's say a class grade is based on two midterms and a project where the project is worth twice as much as the midterms. A student who gets an 80 and a 70 on the midterms and a 90 on the project would have a weighted average of 82.5: $((1 \times 80) + (1 \times 70) + (2 \times 90))/4$, which can be computed with `(w-avg (list 80 70 90) (list 1 1 2))`. You should assume the two lists have the same length.

SOLUTION:

```
;; w-avg : [Listof Number] [Listof Number] -> Number
;; Compute the weighted average of a list of numbers and weights
;; Assume: lists have the same length (and non-empty)
(check-expect (w-avg (list 70 80 90) (list 1 1 2)) 82.5)
(define (w-avg lon ws)
  (/ (w-sum lon ws)
     (foldr + 0 ws)))

;; w-sum : [Listof Number] [Listof Number] -> Number
;; Sum the list of numbers according to given weights
;; Assume: lists have same length
(check-expect (w-sum (list 70 80 90) (list 1 1 2)) (+ 70 80 180))
(define (w-sum lon ws)
  (cond [(empty? lon) 0]
        [(cons? lon)
         (+ (* (first lon) (first ws))
            (w-sum (rest lon) (rest ws)))]))

;; Alt: define w-total function for denominator

;; w-total : [Listof Number] -> Number
;; Total a list of weights
(define (w-total ws)
  (cond [(empty? ws) 0]
        [(cons? ws)
         (+ (first ws)
            (w-total (rest ws)))]))
```

Problem 4 (15 points). Here is a parametric data definition for a tree of elements:

```
;; A [Tree X] is one of:
;; - (make-leaf)
;; - (make-node X [Tree X] [Tree X])
;; Interp: a binary tree that is either empty (a leaf), or non-empty (a node)
;; with an element and two sub-trees.
(define-struct leaf ())
(define-struct node (elem left right))
```

Here is a function that produces a list of all the elements in a tree:

```
;; tree-elems : [X] . [Tree X] -> [Listof X]
;; Produce a list of all elements in tree going top-down, left-to-right
(check-expect (tree-elems (make-leaf)) '())
(check-expect (tree-elems (make-node 7
                                     (make-node 9 (make-leaf) (make-leaf))
                                     (make-node 2 (make-leaf) (make-leaf))))
              (list 7 9 2))
(define (tree-elems bt)
  (cond [(leaf? bt) '()]
        [(node? bt)
         (cons (node-elem bt)
               (append (tree-elems (node-left bt))
                       (tree-elems (node-right bt))))]))
```

Here is an abstraction function for trees that is similar to `foldr` for lists, but works on trees:

```
;; tree-fold : [X Y] . [X Y Y -> Y] Y [Tree X] -> Y
;; The fundamental abstraction function for trees
(define (tree-fold f b bt)
  (cond [(leaf? bt) b]
        [(node? bt)
         (f (node-elem bt)
            (tree-fold f b (node-left bt))
            (tree-fold f b (node-right bt)))]))
```

Give an equivalent definition of `tree-elems` in terms of `tree-fold`. (Just provide the code.)

[Provide your answer on the next page.]

SOLUTION:

```
(define (tree-elems bt)
  (tree-fold (lambda (x l r) (cons x (append l r))) '() bt))
```

[Space for problem 4.]

Problem 5 (15 points). Design a program that takes a list of strings and produces the count of strings with length longer than 3 in the list. For example, if the list contains `"a"`, `"dave"`, `"abc"`, and `"fred"`, the count is 2. You may assume the `[Listof String]` data definition is defined. For full credit, use list abstraction functions. For partial credit, follow the template for `[Listof String]`.

SOLUTION:

```
;; count-3+ : [Listof String] -> Number
;; Count the number of strings with length more than 3 in given list
(check-expect (count-3+ '()) 0)
(check-expect (count-3+ (list "a" "abcd" "fred" "c")) 2)
(define (count-3+ los)
  (foldr (lambda (s c) (if (> (string-length s) 3) (add1 c) c)) 0 los))

;; Alts:
(define (count-3+ los)
  (local [;; count+ : String Number -> Number
          ;; Bump count if string has length > 3
          (define (count+ s c)
            (if (> (string-length s) 3) (add1 c) c))]
    (foldr count+ 0 los))

(define (count-3+ los)
  (local [;; 3+? : String -> Boolean
          ;; Does the string have length > 3?
          (define (3+? s)
            (> (string-length s) 3))]
    (length (filter 3+? los))))

(define (count-3+ los)
  (length (filter (lambda (s) (> (string-length s) 3)) los)))
```

Problem 6 (20 points). Here's a data definition for choose-your-own-adventure books:

```
;; A CYOA is one:
;; - String
;; - (make-choice String CYOA CYOA)
(define-struct choice (q yes no))
;; Interp: a choose-your-own-adventure book where a string
;; is a conclusion and a choice is a yes/no question and
;; the two adventures that follow as a consequnce of the answer.
```

Design a function called `ans` that consumes a `CYOA` and a `[Listof Boolean]` that represents answers to the yes/no questions: `#true` means yes, `#false` means no. The `ans` function produces the `CYOA` adventure that results after giving all of the answers in the list. If the list contains too many answers, `ans` just produces the conclusion of the adventure.

`SOLUTION:`

```
(define a0 "A0")
(define a1 (make-choice "Q1" "A1" "A2"))
(define a2 (make-choice "Q2" a0 a1))

;; ans : CYOA [Listof Boolean] -> CYOA
;; Compute the remaining adventure after using answers from list
(check-expect (adv a0 (list #true #false)) "A0")
(check-expect (adv a1 (list #true #false)) "A1")
(check-expect (adv a2 (list #false #true)) "A1")
(check-expect (adv a2 (list #false)) a1)
(define (ans adv loa)
  (cond [(string? adv) adv]
        [(empty? loa) adv]
        [(cons? loa)
         (if (first loa)
             (ans (choice-yes adv) (rest adv))
             (ans (choice-yes adv) (rest adv)))]))
```