CMSC 131A, Midterm 2
SOLUTION

Fall 2018

NAME:_____

UID: _____

Question	Points
1	10
2	15
3	15
4	20
5	20
Total:	80

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 (10 points). Here is a data definition for representing definitions in a dictionary:

```
;; A Defn is a (make-defn String String)
(define-struct defn (word meaning))
;; Interp: a word and its definition
;; Example:
(make-defn "pair"
           "a set of two things used together or regarded as a unit")
```

Design the following function for determining whether one definition should come before another in a dictionary. (Hint: you can use the function `string<?` to help.)

```
;; defn<? : Defn Defn -> Boolean
;; Does the word defined in d1 come before d2, alphabetically?
(define (defn<? d1 d2) ...)
```

SOLUTION:

```
(check-expect (defn<? (make-defn "a" "first") (make-defn "b" "second")) #true)
(check-expect (defn<? (make-defn "b" "second") (make-defn "a" "first")) #false)
(define (defn<? d1 d2)
  (string<? (defn-word d1) (defn-word d2)))
```

Problem 2 (15 points). A dictionary can be represented using a structure similar to a binary search tree as follows (where `Defn` is defined in problem 1):

```
;; A BSD (binary search dictionary) is one of:
;; - (make-leaf)
;; - (make-node d l r), where d is a Defn, l is a BSD, r is a BSD,
;;      and every definition in l alphabetically comes before d
;;      and d comes alphabetically before every definition in r
;; ASSUME: words are defined at most once in a BSD
(define-struct leaf ())
(define-struct node (val left right))
```

Design a function `all-words` that consumes a `BSD` and produces a list of all the words (just the words, not their meanings) defined in the dictionary, in alphabetic order.

`SOLUTION:`

```
(define a (make-defn "a" "first"))
(define b (make-defn "b" "second"))
(define c (make-defn "c" "third"))
(define l (make-leaf))

;; all-words : BSD -> [Listof String]
;; List all words defined in given dictionary in alphabetic order
(check-expect (all-words l) '())
(check-expect (all-words (make-node b (make-node a l l) (make-node c l l)))
              (list "a" "b" "c"))
(define (all-words bsd)
  (cond [(leaf? bsd) '()]
        [(node? bsd)
         (append (all-words (node-left bsd))
                 (list (defn-word (node-val bsd)))
                 (all-words (node-right bsd)))]))
```

Problem 3 (15 points). For each of the following functions, correctly reformulate the function in terms of an existing abstraction function. You may use either lambda-expressions or local definitions if needed.

```
;; any-zero? : [Listof Number] -> Boolean
;; Are any of the numbers in given list equal to zero?
(check-expect (any-zero? '(1 2 3)) #false)
(check-expect (any-zero? '(1 0 3)) #true)
(define (any-zero? xs)
  (cond [(empty? xs) #false]
        [(cons? xs)
         (or (zero? (first xs))
             (any-zero? (rest xs)))]))
```

SOLUTION:

```
(define (any-zero? xs) (ormap zero? xs))
```

```
(define DOT (circle 10 "solid" "black"))
(define MT (empty-scene 200 200))

;; draw-dots : [Listof Posn] -> Image
;; Draw dots for all given positions on an empty scene.
(check-expect (draw-dots '()) MT)
(check-expect (draw-dots (list (make-posn 10 20) (make-posn 20 30)))
              (place-image DOT 10 20 (place-image DOT 20 30 MT)))
(define (draw-dots ps)
  (cond [(empty? ps) MT]
        [(cons? ps)
         (place-image DOT
                      (posn-x (first ps))
                      (posn-y (first ps))
                      (draw-dots (rest ps)))]))
```

SOLUTION:

```
(define (draw-dots ps)
  (foldr (lambda (p img) (place-image DOT (posn-x p) (posn-y p) img))
         MT
         ps))
```

Problem 3 (cont).

```
;; remove-short-words : Number [Listof Defn] -> [Listof Defn]
;; Remove all definitions for words shorter than given length
(check-expect (remove-short-words 2 (list (make-defn "a" "first")
                                          (make-defn "ab" "rest")))
              (list (make-defn "ab" "rest")))
(define (remove-short-words len ds)
  (cond [(empty? ds) '()]
        [(cons? ds)
         (if (< (string-length (defn-word (first ds))) len)
             (remove-short-words len (rest ds))
             (cons (first ds)
                   (remove-short-words len (rest ds))))]))
```

SOLUTION:

```
(define (remove-short-words len ds)
  (filter (lambda (d) (not (< (string-length (defn-word d)) len))) ds))
```

Problem 4 (20 points). Design an abstraction of the following two functions and re-create the original functions in terms of your abstraction.

```
;; sqr-pos : [Listof Number] -> [Listof Number]
;; Square each number in the list that is positive, remove the rest
(check-expect (sqr-pos '(2 -3 4)) '(4 16))
(define (sqr-pos xs)
  (cond [(empty? xs) '()]
        [(cons? xs)
         (if (positive? (first xs))
             (cons (sqr (first xs))
                   (sqr-pos (rest xs)))
             (sqr-pos (rest xs)))]))


;; lowercase-lengths : [Listof String] -> [Listof Number]
;; Produce a list of lengths for all lowercase strings in given list
(check-expect (lowercase-lengths '("Hey" "you" "guys")) '(3 4))
(define (lowercase-lengths xs)
  (cond [(empty? xs) '()]
        [(cons? xs)
         (if (string-lower-case? (first xs))
             (cons (string-length (first xs))
                   (lowercase-lengths (rest xs)))
             (lowercase-lengths (rest xs)))]))
```

SOLUTION:

```
;; filter&map : [Listof X] [X -> Boolean] [X -> Y] -> [Listof Y]
(define (filter&map xs p f)
  (cond [(empty? xs) '()]
        [(cons? xs)
         (if (p (first xs))
             (cons (f (first xs))
                   (filter&map (rest xs) p f))
             (filter&map (rest xs) p f))]))

(define (sqr-pos xs)
  (filter&map xs positive? sqr))

(define (lowercase-lengths xs)
  (filter&map xs string-lower-case? string-length))
```

[Space for problem 4.]

Problem 5 (20 points). Consider the following data definition (and examples) for representing sentences:

```
;; A Sentence is a [NEListof String]

(define sent1 (list "a" "dog" "barked"))
(define sent2 (list "a" "cat" "meowed"))
```

The `[NEListof X]` data definition is used to represent non-empty lists of elements:

```
;; A [NEListof X] is one of:
;; - (cons X '())
;; - (cons X [NEListof X])
```

Given a `Sentence`, a *bigram* is a list of two consecutive elements of the sentence. For example, `sent1` has exactly two bigrams:

```
(list "a" "dog")
(list "dog" "barked")
```

Using the following data definition,

```
;; a Bigram is a (list String String)
```

design a function `bigrams` which takes a `Sentence` and returns a list of `Bigrams`.

`SOLUTION:`

```
;; bigrams : Sentence -> [Listof Bigram]
;; List of all adjacent words in a sentence
(check-expect (bigrams (list "one")) '())
(check-expect (bigrams sent1)
              (list (list "a" "dog") (list "dog" "barked")))
(define (bigrams sent)
  (cond [(empty? (rest sent)) '()]
        [else
         (cons (list (first sent) (second sent))
               (bigrams (rest sent)))]))
```

[Space for problem 5.]