CMSC 131A, Final Exam (Practice)
SOLUTION

Fall 2017

NAME:_____

UID: _____

Question	Points
1	10
2	10
3	15
4	12
5	15
6	15
7	15
8	20
Total:	112

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 120 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). For the following program, write out each step of computation. At each step, underline the expression being simplified. Label each step as being "arithmetic" (meaning any built-in operation), "conditional", "plug" (for plugging in an argument for a function parameter), or "constant" for replacing a constant with its value.

```
(define T 7)
(define (q z) (sqr z))
(cond [(> T 3) (q 4)]
      [else 9])
```

SOLUTION:

```
(cond [(> T 3) (q 4)] [else 9])          -->[const]
          ^
(cond [(> 7 3) (q 4)] [else 9])          -->[arith]
       ^^^^^^^
(cond [#true (q 4)] [else 9])            -->[cond]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^
(q 4)                                    -->[plug]
^^^^^
(sqr 4)                                  -->[arith]
^^^^^^^
16
```

Problem 2 (10 points). For the following structure definition, list the names of every function it creates. For each function, classify it as being either a constructor, accessor, or predicate.

```
(define-struct zip (pad msg))
```

SOLUTION:

- make-zip : Constructor
- zip-pad : Accessor
- zip-msg : Accessor
- zip?   : Predicate

Problem 3 (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).
```

Design the following function:

```
;; update : String String Dict -> Dict
;; Update entry for given word and meaning, if it exists
;; Add the entry if it does not
```

SOLUTION:

```
(check-expect (update "a" "b" '()) (list (list "a" "b")))
(check-expect (update "a" "c" (list (list "a" "b")))
              (list (list "a" "c")))
(check-expect (update "a" "e" (list (list "c" "b") (list "a" "d")))
              (list (list "c" "b") (list "a" "e")))
(define (update w m d)
  (cond [(empty? d) (list (list w m))]
        [(cons? d)
         (if (string=? (first (first d)) w)
             (cons (list w m) (rest d))
             (cons (first d) (update w m (rest d))))]))
```

Problem 4 (12 points). For each of the following functions, provide the most general **type** signature in Typed Racket that correctly describes the function:

```
(: method : Number -> Number                                          )

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

(: raekwon : [Listof Real] -> 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 : (All [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 5 (15 points). Here is the design of a program `w-avg` that computes the weighted average of a list of numbers and a list of weights.

Design an accumulator variant of `w-avg` that maintains two accumulators: the weighted sum of scores and the sum of weights. Redefine `w-avg` in terms of your accumulator design.

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))`.

In the accumulator version, after all elements of the list have been seen, the first accumulator would be $330 = 80 + 70 + 2 \times 90$ and the second accumulator would be $4 = 1 + 1 + 2$.

```
;; 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)))]))
```

Your solution should not need a helper function like `w-sum`.

[Space for problem 5.]

SOLUTION:

```
(define (w-avg xs ws) (w-avg/a xs ws 0 0))

;; w-avg/a : [Listof Number] [Listof Number] Number Number -> Number
;; Compute weighted average with accumulators:
;;  - sum-wxs: sum of weighted elements seen so far
;;  - sum-ws: sum of weights
(define (w-avg/a xs ws sum-wxs sum-ws)
  (cond [(empty? xs) (/ sum-wxs sum-ws)]
        [(cons? xs)
         (w-avg/a (rest xs)
                  (rest ws)
                  (+ (* (first xs) (first ws)) sum-wxs)
                  (+ (first ws) sum-ws))]))
```

Problem 6 (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 counts the elements in a tree:

```
;; tree-count : [X] . [Tree X] -> Natural
;; Count all the elements in a tree
(check-expect (tree-count (make-leaf)) 0)
(check-expect (tree-count (make-node 7
                                     (make-node 9 (make-leaf) (make-leaf))
                                     (make-node 2 (make-leaf) (make-leaf))))
              3)
(define (tree-count bt)
  (cond [(leaf? bt) 0]
        [(node? bt)
         (add1 (+ (tree-count (node-left bt))
                  (tree-count (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-count` in terms of `tree-fold`. (Just provide the code.)

[Provide your answer on the next page.]

[Space for problem 6.]

SOLUTION:

```
(define (tree-count bt)
  (tree-fold (lambda (x l r) (add1 (+ l r))) 0 bt))
```

Problem 7 (15 points). During your internship at the University of Maryland's Center for Bioinformatics and Computational Biology (CBCB), you're asked to design a program for transcribing DNA strands into RNA strands. A DNA strand is an arbitrarily long sequence of bases, which are either: C, G, A, or T; an RNA strand is an arbitrarily long sequences of either: C, G, A, or U. The *transcription* of DNA to RNA replaces each occurrence of T with U.

Design a program that, given a DNA strand, computes its transcription to RNA.

SOLUTION:

```
;; A DNA is a [Listof DNA-Base]
;; A RNA is a [Listof RNA-Base]
;; A DNA-Base is one of: "C", "G", "A", "T"
;; A RNA-Base is one of: "C", "G", "A", "U"

;; trans : DNA -> RNA
;; Transcribe DNA into RNA
(check-expect (trans '()) '())
(check-expect (trans (list "T" "A" "C" "T" "G"))
              (list "U" "A" "C" "U" "G"))
(define (trans dna)
  (map (lambda (db) (if (string=? db "T") "U" db)) dna))
```

Problem 8 (20 points). Here is the data definition for graphs we studied in class:

```
;; A Graph is [Listof (cons String [Listof String])]
;; Interp: a graph is a list of nodes and neighbors reachable from the node

;; A Path is a [Listof String]
;; Interp: series of nodes neighboring each other that connect a source
;; to destination in some graph.
```

Design the following function:

```
;; all-paths : Graph String String -> [Listof Path]
;; Produce all non-cyclic paths from src to dst in g.
;; Assume: both nodes exist in graph.
(define (all-paths g src dst) '())

(define G
  (list (list "a" (list "b" "c"))
        (list "b" (list "a" "d"))
        (list "c" (list "d"))))

(check-expect (all-paths G "a" "a")
              (list (list "a")))
(check-expect (all-paths G "a" "d")
              (list (list "a" "b" "d")
                    (list "a" "c" "d")))
(check-expect (all-paths G "a" "d")
              (list (list "a" "b" "d")
                    (list "a" "c" "d")))
```

You may assume the following function is already defined and works correctly:

```
;; neighbors : Graph String -> [Listof String]
;; Produce the list of immediate neighbors of given node name.
;; Assume: node exists in graph.
```

Note: this function is tricky to test since the tests shouldn't specify the order of the paths in the list. You may ignore this subtly and just write tests as though a `check-expect` will pass if the checked and expected expressions produce lists with the same elements in any order, e.g. `(check-expect (list 1 2) (list 2 1))` would pass. (Tests are provided above, you don't need to add more unless it's useful to you.)

[Space for problem 8.]

SOLUTION:

```
(define (all-paths g src dst)
  (all-paths/seen g src dst '()))

;; all-paths/seen : Graph String String [Listof String] -> [Listof Path]
;; Accumulator: list of nodes visited so far on path
;; Termination: each recursive call adds an item to seen list not previously
;; seen.  Since there are finite nodes, we get closer to trivial case:
;; src is in seen.
(define (all-paths/seen g src dst seen)
  (cond [(string=? src dst) (list (list src))]
        [(member? src seen) '()]
        [else
         (local [;; all-paths-from : String -> [Listof Path]
                 ;; All paths from given node to dst
                 (define (all-paths-from n)
                   (map (lambda (p) (cons n p))
                        (all-paths/seen g n dst (cons src seen))))]

         (map-append all-paths-from (neighbors g src)))]))

;; map-append : [X Y] . [X -> [Listof Y]] [Listof X] -> [Listof Y]
;; map a function and append all the list of results
(define (map-append f xs) (foldr append '() (map f xs)))
;; NB: could be more efficient
```