CMSC 131A, Final Exam
SOLUTION

Fall 2017

NAME:____

UID: ____

Question	Points
1	10
2	10
3	15
4	15
5	15
6	15
7	15
Total:	95

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 S 1)
(define (q z)
  (cond [(< 4 z) (+ z S)]
        [else 9]))
(q 8)
```

```
SOLUTION:

(q 8)                                --[plug]-->
^^^^^
(cond [(< 4 8) (+ 8 S)] [else 9])    --[arith]-->
       ^^^^^^^
(cond [#true (+ 8 S)] [else 9])      --[cond]-->
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
(+ 8 S)                              --[const]-->
     ^
(+ 8 1)                              --[arith]-->
^^^^^^^
9
```

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 qt (nw ne se sw))
```

```
SOLUTION:

- make-qt  : Constructor
- qt-nw    : Accessor
- qt-ne    : Accessor
- qt-se    : Accessor
- qt-sw    : Accessor
- qt?      : Predicate
```

Problem 3 (15 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"))
```

Recall the following data definition of `[NEListof X]`:

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

Problem 4 (15 points). Here is a program that, given a sentence (see problem 3), computes a two element list consisting of the minimal and maximal words according to alphabetic ordering (`string<?`).

```
;; min-and-max : Sentence -> (list String String)
;; Produces the min and max words in the sentence.
(check-expect (min-and-max (list "the" "zoo" "has" "aardvarks"))
              (list "aardvarks" "zoo"))
(define (min-and-max s)
  (cond [(empty? (rest s)) (list (first s) (first s))]
        [else
         (local [(define w (first s))
                 (define m+m (min-and-max (rest s)))]
            (cond [(string<? w (first m+m))
                   (list w (second m+m))]
                  [(string<? (second m+m) w)
                   (list (first m+m) w)]
                  [else m+m]))]))
```

Redesign this program to use an accumulator-based helper function with two accumulators: the mininmal and maximal elements of the sentence seen so far.

```
SOLUTION:

(define (min-and-max s)
  (min-and-max/a (rest s) (first s) (first s)))

;; min-and-max/a : [Listof String] String String -> (list String String)
;; Accumulator: min and max element seen so far
(define (min-and-max/a s min max)
  (cond [(empty? s) (list min max)]
        [(cons? s)
         (local [(define w (first s))]
           (cond [(string<? w min)
                  (min-and-max/a (rest s) w max)]
                 [(string<? max w)
                  (min-and-max/a (rest s) min w)]
                 [else
                  (min-and-max/a (rest s) min max)]))]))
```

[Space for problem 4.]

Problem 5 (15 points). Design the following program for determining if two sentences (see problem 3) are equal, meaning they are lists of the same length with the same elements:

```
;; sent=? : Sentence Sentence -> Boolean
;; Are the two sentences the same?
```

(You may not use `equal?` in the design of this program.)

```
SOLUTION:

;; sent=? : Sentence Sentence -> Boolean
;; Are the two sentences equal?
(check-expect (sent=? (list "one") (list "one")) #true)
(check-expect (sent=? (list "one") (list "two")) #false)
(check-expect (sent=? (list "one") (list "one" "two")) #false)
(check-expect (sent=? (list "one" "two") (list "one" "two")) #true)
(define (sent=? s1 s2)
  (cond [(empty? (rest s1))
         (and (empty? (rest s2))
              (string=? (first s1) (first s2)))]
        [else
         (and (not (empty? (rest s2)))
              (string=? (first s1) (first s2))
              (sent=? (rest s1) (rest s2)))]))
```

Problem 6 (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 computing the *reverse complement* of DNA strands. A DNA strand is an arbitrarily long sequence of bases, which are either: C, G, A, or T. A and T are complements of each other, as are C and G. The reverse complement of a DNA strand is formed by reversing the strand and taking the complement of each symbol. For example, the reverse complement of AAAACCCGGT is ACCGGGTTTT. Design a program that, given a DNA strand, computes its reverse complement.

```
SOLUTION:

;; A DNA is a [Listof Base]
;; A Base is one of: "A", "C", "G", "T"
;; Interp: DNA is a sequence of nucleotide bases

;; rev-comp : DNA -> DNA
;; Compute the reverse complement of the given strand
(check-expect (rev-comp (list "A" "A" "C" "G" "T"))
              (list "A" "C" "G" "T" "T"))
(define (rev-comp dna)
  (reverse (map comp dna)))

;; comp : Base -> Base
;; Compute the complement of the given base
(check-expect (comp "A") "T")
(check-expect (comp "T") "A")
(check-expect (comp "C") "G")
(check-expect (comp "G") "C")
(define (comp b)
  (cond [(string=? b "A") "T"]
        [(string=? b "T") "A"]
        [(string=? b "C") "G"]
        [(string=? b "G") "C"]))
```

Problem 7 (15 points). Design a program that takes two strings and counts how many times the first string occurs within the second. For example,

- `"ho"` occurs three times in `"ho ho ho"`, while
- `"ho ho"` occurs once in `"ho ho ho"` (notice that occurences cannot overlap).

You may assume the first string is non-empty. It may be helpful to use the `substring` function. Recall: `(substring "hello world" 1 5)` produces `"ello"` and `(substring "hello world" 4)` produces `"o world"`.

SOLUTION:

```
;; occurs : String String -> Natural
;; Count (non-overlapping) occurences of first string in second
;; Assume: first string is non-empty
;; Termination: because s1 is non-empty, each recursive call is on a strictly
;; smaller s2, therefore eventually reaching first cond clause (trivial case).
(check-expect (occurs "ho" "ho ho ho") 3)
(check-expect (occurs "ho ho" "ho ho ho") 1)
(define (occurs s1 s2)
  (local [(define s1-len (string-length s1))]
    (cond [(< (string-length s2) s1-len) 0]
          [(string=? s1 (substring s2 0 s1-len))
           (add1 (occurs s1 (substring s2 s1-len)))]
          [else
           (occurs s1 (substring s2 1))])))
```