CS 135 Fall 2019
Midterm 2 Help Session
Reminders

- The midterm will be held on **Monday, November 4 at 7:00 PM**.
- Check “odyssey.uwaterloo.ca” for seating arrangements.
- The midterm will cover up till **module 10** (slide 27).
- There will be **NO** assignment due Tuesday, November 5.
Clicker Question 1 - Traversing Lists

What will the following expression produce?

\[\text{(define test-list ')((achieved comedy) (0 1 1 2 3 5) ()))}\]
\[\text{(length (rest (rest (second test-list))))}\]

A 1  
B 2  
C 4  
D 5  
E error
Review: Structure Template

Access all of the fields for a structure in its template function.

(define-struct songinfo (performer title genre length))

;; An SongInfo is a (make-songinfo Str Str Sym Nat)

;; songinfo-template: SongInfo → Any
(define (songinfo-template info)
  (... (songinfo-performer info) ...)
  ... (songinfo-title info) ...
  ... (songinfo-genre info) ...
  ... (songinfo-length info) ...))
Review: List Template

Cover all cases in the data definition of list.

;; listof-str-template: (listof Str) → Any
(define (listof-str-template los)
  (cond [(empty? los) . . . ]
    [else . . . (first los) . . .
      . . . (listof-str-template (rest los)) . . . )]

  . . . (listof-str-template (rest los)) . . . )]]))
Problem 1: Shape
Write a template function for a Shape and another template function for a list of Shapes given the following data definitions:

(define-struct rectangle (length width))
;; A Rectangle is a (make-rectangle Num Num)
;; requires: length, width > 0

(define-struct triangle (base height))
;; A Triangle is a (make-triangle Num Num)
;; requires: base, height > 0

;; A Shape is (anyof Rectangle Triangle)
Problem 2: Sort Shapes

Using your template functions as a guide, write a function called `sort-shapes` that uses insertion sort to sort a list of Shapes in non-decreasing order of area. If two shapes have the same area, they should appear in the same order as in the original list. The ordering of a rectangle vs a triangle does not matter.
Problem 2: Shape Area Design Recipe

;; (area-shape shape) Consumes a shape and produces the
;; area of that shape

;; area-shape: Shape → Num

;; Example:

(check-expect (area-shape (make-triangle 5 5)) 12.5)

(define (area-shape shape) ...)


Problem 2: Insert Design Recipe

;; (insert-shape shape sorted-shapes) Inserts
;; a shape into a list of sorted-shapes by area.
;; insert-shape: Shape (listof Shape) → (listof Shape)
;; requires: sorted-shapes is sorted by non-decreasing area
;; Example:
(check-expect (insert-shape (make-rectangle 2 2) (list (make-rectangle 1 1)))
            (list (make-rectangle 1 1) (make-rectangle 2 2))))

(define (insert-shape shape sorted-shapes) . . . )
Problem 2: Sort Design Recipe

;; (sort-shapes shape-list) sorts a shape-list in non-decreasing order.
;; sort-shapes: (listof Shape) → (listof Shape)
;; Example:

(check-expect (sort-shapes
    (list (make-rectangle 10 10) (make-triangle 5 5)))
  (list (make-triangle 5 5) (make-rectangle 10 10)))

(define (sort-shapes shape-list) . . . )

;; Test:

(check-expect (sort-shapes empty) empty)
Problem 3: Stepping Structures

(define-struct painting (price colours))
;; A Painting is a (make-painting Num (listof Sym))

(define starry-night-colours '(blue yellow))
(define starry-night (make-painting 14 starry-night-colours))

(define (vandalize art)
  (make-painting (/ (painting-price art) 4)
    (cons 'red (painting-colours art))))

Step through: (vandalize starry-night)
Problem 3: Steps Solution

⇒ (vandalize starry-night)
Problem 4: Sublist

Write a function called `sublist` which consumes a list, `lst`, and two natural numbers, `start` and `end`. `sublist` should produce the elements in `lst` indexed from `start` up to but not including `end`. If `start` is equal to `end`, produce `empty`. If the list doesn’t have sufficient elements at any point then any contents within the range so far are returned. Note that the first element of a list is indexed at 0.

(sublist '(a b c d e f) 2 5) → '(c d e)
(sublist '(a b c d e f) 4 8) → '(e f)
Problem 4: Sublist Design Recipe

;; (sublist lst start end) Produces elements
;; from index start, to index end of lst.
;; sublist: (listof Any) Nat Nat → (listof Any)
;; requires: start ≤ end
;; Example:
(check-expect (sublist '(a b c) 1 2) '(b))

(define (sublist lst start end) ...)

;; Tests:
(check-expect (sublist empty 1 2) empty)
Problem 4: Get Table Chunk

;;; A Table is a (listof (listof Any))
;;; requires: all the sublists have the same length

Using sublist, write a function called get-table-chunk which consumes a Table and four natural numbers, col-start, col-end, row-start and row-end. get-table-chunk should produce the table with only rows from row-start up to but not including row-end with their columns indexed from col-start up to but not including col-end. You may assume the input is valid.

Note that columns and rows’ indices start at 0.
Problem 4: Table Example 1

(get-table-chunk ’( (1 2 3 4 5)
                      (a b c d e)
                      (3 6 9 7 5)
                      (f g h i j)) 1 3 1 3)

⇒ ’( (b c)
      (6 9))
Problem 4: Table Example 2

\[(\text{get-table-chunk '}(1\ 2\ 3\ 4\ 5\ 7)\n\text{\quad (a\ b\ c\ d\ e\ 8)}\n\text{\quad (3\ 6\ 9\ 7\ 5\ 8))\ \text{1\ 4\ 0\ 3)}}\]

\[\Rightarrow \text{'}(2\ 3\ 4)\n\text{\quad (b\ c\ d)}\n\text{\quad (6\ 9\ 7))}\]
Problem 4: Get Table Chunk Design Recipe

;;; get-table-chunk: Table Nat Nat Nat Nat Nat → Table
;;; requires: col-start ≤ col-end, row-start ≤ row-end
;;; col-end ≤ the length of table columns
;;; Example:
(check-expect (get-table-chunk '((a b) (c d)) 0 1 0 1) '((a)))

(define (get-table-chunk table col-start col-end row-start row-end) . . . )

;;; Test:
(check-expect (get-table-chunk empty 0 0 0 0) empty)
Problem 5: Add and Multiply
First, define a **Simple Recursive** function called *my-add* that adds two **Natural Numbers** without using the “+” function.

Then, define a **Simple Recursive** function called *my-multiply* that multiplies two **Natural Numbers** using your *my-add* function.

You may **not** use any helper functions, however you may use the built-in functions *add1* and *sub1*. 
Problem 6: Factor

Define an **Accumulative Recursive** function called `find-factors` that takes in one **Positive Integers** `n` and produces a list of its factors in descending order.

Because we require you to use accumulative recursion, you will **need** a wrapper function.

```
(find-factors 6) ⇒ (list 6 3 2 1)
(find-factors 7) ⇒ (list 7 1)
```
Problem 7: Hangman
In the game of hangman, one player decides on a secret word and the other player tries to guess the word one letter at a time.
Write a function called hangman that consumes a string called secret-word and another string called current-state, as well as a single character guess. current-state is the same string as secret-word except all the letters that have not been guessed yet are replaced by "∗". hangman should produce a new string such that if guess is in secret-word, all the corresponding blanks in current-state are replaced by guess. Otherwise, current-state is produced.
Problem 7: Hangman Examples

(hangman "joker" "*oker" #\j) ⇒ "joker"
(hangman "spook" "*****" #\o) ⇒ "**oo*"
(hangman "team" "tea*" #\i) ⇒ "tea*"
Problem 7: Helper Design Recipe

;; (hangman/chars secret-word current-state guess) Plays hangman
;; given the secret-word, current-state, and a next guess.
;; hangman/chars: (listof Char) (listof Char) Char → (listof Char)
;; Example:
(check-expect (hangman/chars (#\y #\u #\h) (#\* #\u #\h) #\y)
              (#\y #\u #\h))

(define (hangman/chars secret-word current-state guess)...)

Problem 7: Hangman Design Recipe

;;; (hangman secret-word current-state guess) Plays hangman
;;; given the secret-word, current-state, and a next guess.
;;; hangman: Str Str Char → Str
;;; Example:
(check-expect (hangman "joker" "*oker" #\j) "joker")

(define (hangman secret-word current-state guess) . . .)

;;; Test:
(check-expect (hangman "team" "tea*" #\i) "tea*")
Problem 8: Compute Average

;; A GradeList is one of:
;; * empty
;; * (cons (list Str Num) GradeList)
;; requires: GradeList is sorted using string<?
;; numbers are between 0 and 100, inclusive.

Write a function **compute-average** that takes in two GradeLists and produces one GradeList combining students from both GradeLists. If a student appears in both GradeLists, their new grade is the average of their grades from both of their classes.
Problem 8: Compute Average Examples

Here are a few examples:

\[
\text{(compute-average (list (list "Ben" 95) (list "Klarence" 69))}
\]
\[
\quad \text{(list (list "Ben" 87) (list "Joe" 90)))}
\]
\[
\Rightarrow \text{(list (list "Ben" 91) (list "Joe" 90) (list "Klarence" 69))}
\]

\[
\text{(compute-average (list (list "Ben" 99) (list "Klarence" 69))}
\]
\[
\quad \text{(list (list "Joe" 100) (list "Klarence" 69)))}
\]
\[
\Rightarrow \text{(list (list "Ben" 99) (list "Joe" 100) (list "Klarence" 69))}
\]