Final Examination (Tuesday morning)

Policies for the Final Examination

1. This is a closed book examination. You may not rely on notebooks, papers, textbooks, computers, colleagues, or anything similar. You may, however, refer to one 8.5 x 11 inch, double sided, hand-written, set of notes that you brought to this exam.

2. This is also a paper examination. Answer all problems in pen or pencil on blank pieces of paper and then staple them together. You may use additional blank pieces of paper as scratch, provided you turn them in at the end of the examination.

3. You have received a cover sheet for this exam with a number on it. Please write your name on the cover sheet. Please write your number (not your name) on each of the non-scratch pages you turn in.

4. There are four problems on this exam. They are not necessarily of equal difficulty.

Four correct or mostly correct solutions will earn you an A. Three correct or mostly correct solutions will earn you a B. Two correct or mostly correct solutions will earn you a C. One correct or mostly correct solution will earn you a D. Zero correct or mostly correct solutions will earn you an F. Failure to attempt the exam will earn you a 0.

Partially correct solutions may or may not earn you a higher grade, at the discretion of the grader.

5. Many of the problems ask you to write Scheme code. Although you need not write correct working Scheme code, your code should be of sufficient quality that the grader can be confident that you would be able to make it work correctly with minimal effort when given access to DrRacket.

6. Other people may be taking this exam (or a similar exam) at another time. You may not discuss this examination with anyone.

7. After you have completed the examination, please write, sign, and date the following statements on the cover sheet:

I have neither received nor given inappropriate assistance on this examination.

I am not aware of any other students who have given or received inappropriate assistance on this examination.

If you do not feel that you can sign either of these statements, please arrange to meet with me as soon as possible.
8. Please arrange the remaining sheets in order by problem number, make sure you’ve written your number and the page number on the top of each sheet, and then staple them together *without* the cover sheet.

9. Please turn in this examination, the cover sheet, the stapled pages, your notes (if used), and your scrap paper.

**Problem 1: Whatzitdo?**

*Topics: Files, file recursion, code reading, documentation*

Consider the following procedure, which was written by a student who does not take much care in naming things.

```
(define w
  (lambda (a b)
    (let ((c (open-input-file a)))
      (let d ([e 0])
        (let ([f (read-line c)])
          (cond
            [(eof-object? f) e]
            [(b f) (d (+ e 1))]
            [else (d e)]))))))
```

Figure out what the procedure does and then do the following.

a. **Rewrite the procedure using better names for** \( w, a, b, c, d, e, \) and \( f \).

b. **Document your rewritten procedure using 6P-style documentation.**

**Problem 2: A radial blend**

*Topics: Projects, GIMP Tools, Computing pixels*

Consider the following design statement for a single image in the course project.

*In our image, we will explore “radial blends”, blends that expand outwards from the center of the image. We will use black and white images to avoid the complexities of color. We will work only with square images. In exploring the contrast between mechanization and the organic, we will place our blend within a circle, but will make the blend have a more rectangular aspect. In particular, the darkness of the blend will depend on the sum of the x and y distances from the center of the image. (At the center, the image will be white. At the middle of the left and right edges, the image will be black.) Outside of the circle, the image will be white.*
They’ve even sketched out a sample image.

The designers have also worked out the following framework for building their image.

```scheme
(define project-sample
  (lambda (side)
    (let ((image (image-new side side)))
      ; Select the circle
      (image-select-ellipse! image REPLACE ___ ___ ___ ___)
      ; Fill in the blend
      (image-recompute! image (lambda (col row) ____))
      ; Clear the selection
      (image-select-nothing! image)
      ; Return the computed image
      image)))
```

Finish implementing this procedure. Your implementation should allow the image to scale appropriately, so that we always get black at the edges of the circle (as in the sample), whether the image is 200x200 or 50x50 or 1000x1000 or any other square.

Note that the framework above gives very little space for the body of the lambda expression. You will certainly need more room than is provided. You may also add additional code (such as a `let` expression), if you find it appropriate to do so.

**Problem 3: Reducing operations**

*Topics: List recursion, higher-order procedures*

As you may recall, when we first considered recursive helper procedures, our motivation was to correctly subtract lists of numbers from left to right. We wrote a procedure similar to the following:
(define difference
  (lambda (numbers)
    (let kernel ([difference-so-far (car numbers)]
                  [remaining (cdr numbers)])
      (if (null? remaining)
          difference-so-far
          (kernel (- difference-so-far (car numbers))
                   (cdr numbers))))))

We ended up defining a similar procedure for summing the values in a list.

(define sum
  (lambda (numbers)
    (let kernel ([sum-so-far (car numbers)]
                 [remaining (cdr numbers)])
      (if (null? remaining)
          sum-so-far
          (kernel (+ sum-so-far (car numbers))
                   (cdr numbers))))))

This pattern is useful for more than just computing differences and sums. It is so useful that computer scientists have generalized it to a technique called “reduction”. Given a binary procedure, proc, we create a “reducer” for the procedure that takes a list as a parameter and builds a procedure that does something similar to difference and sum above. That is, the new procedure iterates through the list, repeatedly applying proc to the current result and the next value in the list.

;;; Procedure:
;;;    reducer
;;; Parameters:
;;;    proc, a binary procedure
;;; Purpose:
;;;    Create a procedure that "reduces" a list by repeatedly applying
;;;    proc.
;;; Produces:
;;;    rproc, a unary procedure that takes a list as a value
;;; Preconditions:
;;;    proc can take its return type as an input
;;; Postconditions:
;;;    (rproc (list v1 v2 v3 v4 ... vn)) =
;;;      (proc (proc ... (proc (proc proc v1 v2) v3) v4) ...) vn

Here are some examples.

> (define difference (reducer -))
> (difference (list 4 2 1))
  1       ; (4 - 2 = 2, 2 - 1 = 1)
> (difference (list 1 2 3 4))
  -8      ; (1 - 2 = -1, -1 - 3 = -4, -4 - 4 = -8
> (define sum (reducer +))
> (sum (list 4 2 1))
  7       ; (4 + 2 = 6, 6 + 1 = 7
> (sum (list 1 2 3 4))
> (define sap (reducer string-append))
> (sap (list "a" "b" "c"))
  "abc"
Implement reducer.

**Problem 4: Selection sort**

*Topics:* Vectors, sorting, selection, efficiency, code reading.

As you may recall, the selection sort algorithm works by repeatedly finding the smallest remaining value in a list or vector and swapping it to the beginning of the list or vector.

(define selection-sort!
  (lambda (vec)
    (let ([len (vector-length vec)])
      (let kernel ([pos 0])
        (when (< pos len)
          (swap! vec pos (index-of-smallest vec pos len))
          (kernel (+ pos 1)))))))

a. In class, we studied two sorting algorithms in detail: insertion sort and merge sort. Which of these is more similar to selection sort? How is it similar?

If we apply selection sort to vectors, it’s important to find not just the smallest value, but also its position. Consider the following implementation of vector-select-smallest.

;;; Procedure:
;;; vector-select-smallest
;;; Parameters:
;;;   vec, a vector of real numbers
;;;   first, a non-negative integer
;;;   last, a non-negative integer
;;; Purpose:
;;;   Finds the index of the smallest value in the vector between the
;;;   positions first and last.
;;; Produces:
;;;   index-of-smallest, a non-negative integer
;;; Preconditions:
;;;   0 <= first <= last < (vector-length vec)
;;; Postconditions:
;;;   For all i in the range first ... last, inclusive,
;;;   (vector-ref vec index-of-smallest) <= (vector-ref vec i)
(define vector-select-smallest
  (lambda (vec first last)
    (cond [(= first last)
        first]
        [(< (vector-ref vec first)
            (vector-ref vec (vector-select-smallest vec (+ 1 first) last)))
        first]
        [else
        (vector-select-smallest vec (+ 1 first) last)])))

b. This implementation of vector-select-smallest is not very efficient. Why not?
c. Show or explain how `vector-select-smallest` can be implemented more efficiently.

Samuel A. Rebelsky, rebelsky@grinnell.edu

Copyright © 2007-2015 Charlie Curtsinger, Janet Davis, Samuel A. Rebelsky, and Jerod Weinman. (Selected materials are copyright by John David Stone or Henry Walker and are used with permission.)

This work is licensed under a Creative Commons Attribution 3.0 Unported License To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.