CS 3, MIDTERM #2
Professor Dan Garcia
Fall Semester 2002
Test given 17 October 2002.

Exam topic: Recursion

NOTE:

- There were two parts to this exam.
- Part 1, worth 40 points, was done by hand.
- Part 2, worth 10 points, was done with a computer.

CS 3, MIDTERM #2
Professor Dan Garcia
Fall Semester 2002
Test given 17 October 2002.

Exam topic: Recursion

THIS IS PART 1 (ONE) OF THE EXAM.

Question 1: What a difference a day makes… (13 points)

Your friend tells you about a fun thing to try with a sequence of numbers. She asks you to write as many numbers as you wish in a row, like this:

3 5 0 -3 10

- Then, *for all adjacent pairs of numbers, find the absolute value of their difference* and write that below each pair.
- When you are done you will have a new sequence one element shorter than the one you started with.
- In the example below, $|3 - 5| = 2$, so 2 was the first number in the second row. Similarly, $|5 - 0| = 5$, $|0 - -3| = 3$, etc.
- She says you then to continue to do this until you are left with a signle number, called the *last-difference.* E.g.,

Row 1: 3 5 0 -3 10 ;original sequence
Row 2: 2 5 3 13 ;abs value of ROW1 differences
Row 3: 3 2 10 ;abs-value of ROW2 differences
Row 4: 1 8 ;abs-value of ROW3 differences
Row 5: 7 ;abs-value of ROW4 differneces

- You are to write a program to do this, but we'll help you with the steps.
- Your partner writes a helper function called *abs-differences* that takes a sentence of numbers and returns the sentence of the absolute values of the differences of successive elements. That is, if the inputs to *abs-differences* were one of the rows above, it would return the row below it (*as a sentence)*

a. Fill in the blanks to complete the function *last-difference* which computes the *last difference* of a sentence of numbers. **You MAY NOT define any helper functions, <u>but assume *abs-differences* is available (5 points.)</u>**

;;INPUTS: A sentence of numbers.
;;REQUIRES: The sentence be non-empty.
;;SIDE-EFFECTS: None
;;RETURNS: The last difference *number* of the sentence, i.e., the results of taking the abs value of adjacent differences over and over until one number is left.
;;EXAMPLE: (last-difference '(3 5 0 –3 10)) → 7

```
(define (last-difference s)
        (if _____________________________________________________)

            _____________________________________________________

            _____________________________________________________))
```

b. What pattern is *last-differnece?* (Circle one- 1 point.)

MAPPING FINDING COUNTIN FILTERING TESTING COMBINING

c. Your partner then disappears to Hawaii and takes *abs-differences* with her. Fill in the blanks to complete *abs-differences* that takes a sentence of numbers and returns the absolute value of the differences of successive elements.
   - Feel free to use *second,* which returns the *second* item from a word or sentence (similar to *first.*)
   - Feel free to use *abs,* which returns the absolute value of a number.
   - **You MAY NOT define any helper functions** (5 points.)

```
;;INPUTS: A sentence of numbers.
;;REQUIRES: The sentence has at least 2 elements.
;;SIDE-EFFECTS: None
;;RETURNS: A sentence containing the abs. value of the differences of successive
elements.
;;EXAMPLE: (last-difference '(3 5 0 –3 10)) → (2 5 3 13)

(define (abs-difference s)
        (if _____

            _____

            _____
```

d. As you can guess, sometimes two different sentences produce the same *last-difference.* Write down two sentences that have the same last-difference of 1. What makes this challenging is that **each** *of the six blanks below must be filled with a* ***different*** *single-digit number from 0 through 9.* There may be multiple solutions (2 points.)

   - (last-difference '(___ ___ ___)) → 1
   - (last-difference '(___ ___ ___)) → 1

Take a deep breath, you're about one-third of the way done!

Question 2: Solving a fun mystery… (13 points)
Your CS3 friend loves solving mysteries and writes the following function (for all questions on this page, assume function arguments are evaluated left-to-right):

```
(define (mystery n ans)
        (if (= n 0)
                ans
                (se n (mystery (- n 1) (se ans n))) ))
```

a. Is *mystery* TAIL or EMBEDDED recursion? Circle one (1 point.)
    a. TAIL
    b. EMBEDDED
b. What does (mystery 1 'cal) return? If this is an error, say what the error is. If it is an infinite loop, write "it never returns" (1 point).

    _____

c. What does (mystery 2 'cal) return? If this is an error, say what the error is. If it is an infinite loop, write "it never returns" (1 point).

    _____

d. What does the following expression return? If this is an error, say what the error is. If it is an infinite loop, write "it never returns" (6 points).

    _____

e. Provide values for the blanks which cause *mystery* to return the values listed. If it is impossible, circle the word IMPOSSIBLE on the far right (4 points).
    > (count (mystery ___ ___)) → 6 IMPOSSIBLE
    > (mystery ___ 'cal) → *goes into an infinite loop.* IMPOSSIBLE
    > (/ 5 0) → *ERROR! Division by zero.*
    > (mystery ___ (/ 5 0)) → *goes into an infinite loop.* IMPOSSIBLE

(turn the page for question 3)

Question 3: The students +ed and +ed like rabbits… (13 points)
We are going to look at the function *multiply-then-add,* a procedure that takes three arguments a, b, and cand multiples the first two and then adds the third. The catch is that **YOU MAY NOT <u>USE * OR / from scheme!</u>** Here are the specs:

;;INPUTS: Numbers a, b, and c
;;REQUIRES: a, b, and c are not negative.
;;SIDE-EFFECTS: none
;;RETURNS: The same thing that (+ (* a b) c) does
;;EXAMPLE: (multiply-then-add 3 4 100) → 112. (3 * 4) + 100 = 112.

a. Write a version of *multiply-then-add* using **EMBEDDED recursion,** without using * or /. **You MAY NOT define any helper functions.** (6 points)

```
(define (multiply-then-add a b c)
```

b. Write a version of *multiply-then-add* using **TAIL recursion, without using * or /.** You **MAY NOT define any helper functions (6 points).**

```
(define (multiple-then-add a b c)
```

c. Take a look at your **EMBEDDED** solution in (A) above. It might be that the restrictions we placed on the inputs in the REQUIRES section were too strict. IF you think the earlier REQUIRES section was fine, **<u>circle</u> it above and leave the box below blank.** Otherwise, **<u>cross it off</u>** and **write a new REQUIRES section <u>based on your (A) solution</u> below.**

;;REQUIRES:

CS 3, MIDTERM #2
Professor Dan Garcia
Fall Semester 2002
PART II given 22 October 2002.

Exam topic: Recursion

THIS IS PART 2 (TWO) OF THE EXAM, THE COMPUTER PART

For this exam, pretend you know nothing about higher-order functions or lambda. You are to use only recursion here.

THE SETUP Log in to the computer in front of you and type: *quiz.* Emacs will open up with some buggy code in oen buffer and *stk* in another. You are welcome to use this to help answer these questions; you may not run other programs, send mail, etc. *You may find it faster to finish it all on paper first, then use stk to check your work.*

THE STORY (there is a fair bit of reading to set up the question, but don't despair- the questions on the other side of this page don't require much writing to answer)

Throughout the semester you have been asked to choose different partners for your homework and labs. As you can imagine, it's hard for the TAs to keep track of all these partnerships. It would be very handy to have a function that takes a sentence of the first names of everyone in section and returns a sentence of all the possible student partnerships that could ever result (by concatenating the partner names together with a dash in-between.) For example, if *ana* an *dan* were chosen to be partners, we would represent that partnership as *ana-dan*. You may assume that you can never be partners with yourself and that all student names in the class are unique.

Sometimes, however, there are some limitations as to who an be paired with whom. For example, the TA might only want partnerships from students who live in separate dorms, or from students who were the same age, or who had vastly different heights, etc. If two students can be paired together, we'll cal them "good" partners.

To help with this process, we are going to write a function called *all-good-partners*, which takes two arguments:

- *good-partners?* –a predicate which takes two student names and returns #t if and only if the students are good partners according to the partner restrictions.
- *students*- a sentence with the first names of all the students in the section.

all-good-partners should return a sentence (whose order doesn't matter) of the good partners' names concatenated together. Each potential partnership should be listed once and only once. I.e. it doesn't make sense to list both *ana-dan* and *dan-ana*.

For example, let's say all students in the section were named *ana, bo, che, dan,* and *ed,* and the TA mandates that all students partnerships be between students whose names were the same length. Furthermore, let's say someone wrote the predicate *same-name-length?* which returns #t if and only if the two names are the same length (#f otherwise.) Then (all-good-partners same-name-length? '(ana bo che dan ed)) would return (ana-che ana-dan che-dan bo-ed) in any order. Note that the other potential partnerships, *ana-bo, ana-ed, bo-che, bo-dan, che-ed* and *dan-ed* are not 'good' and would not be included because their names have different lengths.

THE QUESTIONS

We're going to write *all-good-partners* by first debugging its helper function, called *pair-with-others* (below we have changed A SINGLE LINE) from a perfect, working version of the function). *pair-with-others* takes four arguments:

- *good-partners?* (the predicate mentioned earlier.)
- *person* (a particular student's name as a word)
- *others* (the sentence of remaining potential student partners), and
- *answer* (the current answer so far)

and returns a sentence of *all* the good partnerships involving *person.* For example:

> (pair-with-others same-name-length? 'bo '(che dan ed) '()) → (bo-ed)

```
(define (make-partners student-a student-b) ;;helper function
        (word student-a '- 'student-b))

(define (pair-with-others good-partners? person others answer)
        (cond ((empty? others) answers);; line1
                ((good-partners? person (first others));; line 2
                        (pair-with-others good-partners? ;; line 3
                                                person ;; line 4
                                                (bf others) ;; line 5
                                                se (make-partners person (first others))))) ;;line 6
                (else
                        (pair-with-others good-partners? ;; line 7
                                                person ;; line 8
                                                (bf others) ;; line 9
                                                answer)))) ;; line 10
```

a. Why is *pair-others* buggy? **Fill in the blanks for the bug that applies** and provide the **simplest yourname** input that makes the statement correct and triggers **only that bug.** Then change only one line to fix the error. (*Simplest* = fewest students in the 'others' input sentence; *yourname* = you must supply your own names for the input students. You **may not use** *ana*, *bo*, *che*, *dan*, or *ed*. Be creative.) You might find it handy to write *same-name-length?* for online testing. (4 points)

It rejects good partnerships (when it shouldn't.) E.g., (pair-with-others same-name-length? _____ _____ '()) returns _____ when it should return _____

It accepts bad partnerships (when it shouldn't.) E.g., (pair-with-others same-name-length? _____ _____ '()) returns _____ when it should return _____

It can go into an infinite loop (when it shouldn't. E.g., (pair-with-others same-name-length? _____ _____ '()) returns _____ when it should return _____

"Replacing line _____ with _____
will fix the bug so that *pair-with-others* works correctly on all input".

```
(define (all-good-partners good-partners? students)
        (if (empty? students) ;;line 1
                '() ;;line 2
                (se (pair-with-others good partners? (first students) students '()) ;;line 3
                        (all-good-partners good-partners? (bf students))))) ;;line 4
```

b. Above you see *all-good-partners.* As before, we have changed *only a single line* from the *perfect working* version to introduce a bug. Complete the statement below with the **simplest surname** input that triggers the bug (*simplest* = fewest people in the 'students' input sentence, *yourname*= you must supply your own name for the students. You **may not use** *ana, bo, che, dan,* or *ed*. Be creative.) If you wish, experiment with your online version of *all-good-partners* (4 points).

(all-good-partners same-name-length? _____)
returns _____ when it *should return*
_____. Replacing line _____ with
_____ will fix the bug so that *all-good-partners* works correctly on all input.

c. Let's say your TA *doesn't care* about the silly rules which restrict partnerships. Using *all-good-partners,* write *all-partners* which returns all possible partners without restrictions. For example:

> (all-partners '(ana bo che dan ed)) →
(ana-bo ana-che ana-dan ana-ed bo-che bo-dan bo-ed che-dan che-ed dan-ed)

The partners can be in any order in the sentence. You may write a one-line helper function if you need it. (2 points)

```
(define (all-partners students)
        _____)

;;Below you may write a one-line helper function if you need…

(define (_____) _____)
```