COMP 210, Fall 2000
Third Exam
Due Wednesday, December 6, 2000 at 5:00 PM
Hand the exam in at DH 2065

You will need to supply your own paper.

This exam is conducted under the Rice Honor Code. It is a closed-notes, closed-book exam. You have **three hours** to take the exam. You may take one break of up to fifteen minutes during the exam. The break does not count against your three hours. Of course, you are not allowed to discuss the exam with other people during the break, or to consult your notes, book, or DrScheme.

You may use a computer to type your answers to the various questions. You may not use DrScheme during the exam [or any other Scheme interpreter or compiler].

Please

- Print your name legibly on the outside of the blue book. If you use more than one blue book, print your name on the outside of each blue book.
- Write the pledge *correctly* and sign it. (No points this time.)
- Record your starting and stopping times.

The exam has four questions. The relative weights (points) are marked on each question. If a question uses the results from an earlier question, you should assume that you have the perfect solution to that earlier question. For example, if question one has you create a program **foo** , you can assume, in question two that **foo** exists and that it works as specified, *even if your answer to question one is incomplete or incorrect.*

You may use the usual Scheme constructs and constants in your programs.

I have tried to give you enough information to answer each question on the exam. In the event that you need additional information, or a clarification, make a reasonable assumption and document it. (Write down, explicitly, any assumptions that you make.)

1. ***Using an Accumulator*** (25 points)
Scheme's abstract function **foldl** is best implemented using an accumulator. Given the definition of **foldl**,

;; foldl: (α β → β) β list-of-α → β
;; Purpose: given a function *f*, an initial value *base*, and a list
;; (list x_0 x_1 x_2 **…** x_n), foldl computes
;; ($f\ x_n$ **…** ($f\ x_2$ ($f\ x_1$ ($f\ x_0$ *base*))) **…**)
(define (foldl f base a-list) …)

show the code that you would write to create **foldl** if it were not present in your implementation of Scheme.

You may not use ***set!*** *(or any of its variants, like* ***set-structure!****) in your code for this problem. You need not show templates, test data, or any evaluation.*

2. ***Naming and Scoping*** (15 points)
Write a program **counter** that takes no arguments and returns a number that indicates the total number of times that **counter** has been called. Hide any state that **counter** requires inside a **local**.

You need not show data analysis, templates, or test data (obviously). Be sure to include all the comments required by the design methodology, though.

3. ***Putting It All Together*** (35 points)
As a COMP 210 alum, you have been asked to write the password subsystem for your company's new Scheme-based operating system. It needs the following functions:

   a. **create-account –** consumes two symbols, *name* and *pwd*, and returns a boolean. The symbol *name* is taken as a user name, and *pwd* is the initial password for that user name. **create-account** should set up the internal data structures that allow **check-password** and **change-password** to recognize *name* & *pwd* as valid. If name already exists as an account, **create-account** should return false. If, for any other reason, it cannot create the name, it should return false. If the account is successfully created, it should return **true**.

   b. **check-password –** consumes two symbols *aname* and *apwd*, and returns a boolean. If it determines that, (1) *aname* is a valid user name created with the program **create-account**, and (2) that *pwd* is the password registered for that user name (either by **create-account** or by **change-password**), then **check-password** returns **true**. If either of those conditions is not true (the user name is not registered or the password is incorrect), then **check-password** returns **false.**

   c. **change-password –** consumes four symbols, *thename, oldpwd, newpwd1*, and *newpwd2,* and performs a complex operation. First, it verifies that *thename* is registered with current password *oldpwd*. If this is not the case, it returns **false**. Next, it confirms that *newpwd1*and *newpwd2* are identical. If this is not the case, it returns **false**. If the inputs pass all these checks, then **change-password** modifies the internal data structures so that *newpwd1* is the password associated with the user name *thename*, and it should return **true**.

Develop these three programs, along with all the data structures that they need. Hide the programs and their data structures inside an interface function named **security**. The contract for security is a little unusual

```
;; security:  symbol → password-service
;; Purpose:  Given ‘create, it returns create-account,
;;                 given ‘check, it returns check-account, and
;;                 given ‘change, it returns change-account
(define (security a-symbol) …)
```

Include all of the comments required by the design methodology for all of the programs that you write, including the ones hidden inside ***security****. You do not need to show examples, templates, or test data.*

4. ***Using Vectors*** (25 points)

   a. Write a program **vector-add** that consumes a vector of number and returns the sum of all the elements in the vector.

   b. Write a program that consumes two vectors of numbers, ***a*** and ***b***, and produces a vector of numbers, ***c***, according to the following rules.

      - ***a*** and ***b*** need not be the same length.
      - ***c*** has the same length as ***b.***
      - If ***a*** is $\langle a_0\ a_1\ a_2 \ldots a_n \rangle$ and b is $\langle b_0\ b_1\ b_2 \ldots b_m \rangle$, then ***c*** is computed as

$$c_0 = a_0b_0 + a_1b_0 + a_2b_0 + \ldots a_nb_0$$

$$c_1 = a_0b_1 + a_1b_1 + a_2b_1 + \ldots a_nb_1$$

$$c_2 = a_0b_2 + a_1b_2 + a_2b_2 + \ldots a_nb_2$$

$$\ldots$$

$$c_m = a_0b_m + a_1b_m + a_2b_m + \ldots a_nb_m$$

Note that each row is a separate element of ***c***

Show <u>*all*</u> *the steps in the development of your programs.*

Extra Credit (10 points)
We can approximate the logarithm of a number by iteration, as long as we are not overly concerned with accuracy. The functional version of the code might look like this:

```
;; log-approx: num num → void
;; Purpose: prints an approximation of the logarithm of "start" by
;;                    repeatedly dividing the base into the start value
(define (log-approx start base)
   (local [(define (log-help start base count)
                 (cond [(<= start 1) count]
                          [else (log-help (/ start base) base  (add1 count))]
                          ))]
             (printf "~s raised ~s times is ~s-n" base (log-help start base 0) start)
    ))
```

Feeding this program some test cases produces the following results

(log-approx 100 5) ==> 5 raised 3 times is 100
(log-approx 256 2) ==> 2 raised 8 times is 256
(log-approx 1024 2) ==> 2 raised 10 times is 1024

Write an iterative version of this program using the built-in function **loop-until.**

As a reminder, the contract for **loop-until** is

;; loop-until: α $(\alpha \rightarrow \text{boolean})$ $(\alpha \rightarrow \alpha)$ $(\alpha \rightarrow \beta)$ $\rightarrow$ void

Simply show your final code.