This supplement to Exam 2 is intended for students enrolled in the Honors section of 2500.

See the instructions on the regular exam.

Good luck!
Problem 1  Design the function `concat` that consumes a list of lists and appends each list to produce a single list.

For example:

```scheme
> (concat (list (list 1 2) (list 3) (list 4 5)))
(list 1 2 3 4 5)
```

Give `concat` its most general contract and its definition in terms of a loop function.
A sequence represents a series of values. For example, 1, 2, 3 is a sequence. The even natural numbers up to 16 are a sequence: 0, 2, 4, ..., 16. The strings "A", "B", "C" are a sequence. One thing you can do with sequences is get the \(i\)th element of the sequence (and we always start counting from 0). So the 0th element of "A", "B", "C" is "A", the 1st even natural number is 2, and so on.

Here is a data definition for representing finite sequences:

```scheme
;; A [Sequence X] is a
;; (make-seq Natural [Natural -> X]).
(define-struct seq (length i->elem))
```

Notice that this data definition uses a function to represent the sequence of elements. When the function is applied to an index, it gives back the element at that index.

Here is an example of a [Sequence Natural] : the even natural numbers up to (and including) 16:

```scheme
(make-seq 9 (\(x\) (* 2 x)))
```

Here's a convenient function for determining if a sequence is empty:

```scheme
;; seq-empty? : [Sequence X] -> Boolean
;; Is the sequence empty?
(define (seq-empty? s)
  (zero? (seq-length s)))
```

And here are another couple of convenient functions for turning a list of elements into a sequence and vice versa:

```scheme
;; list->seq : [Listof X] -> [Sequence X]
;; Turn a list into a sequence
(define (list->seq ls)
  (make-seq (length ls)
    (\(i\) (list-ref ls i))))

;; seq->list : [Sequence X] -> [Listof X]
;; Turn a sequence into a list
(define (seq->list seq)
  (build-list (seq-length seq) (seq-i->elem seq)))
```

You may use list->seq and seq->list for tests, but they should not be used otherwise.
Problem 2 Using the Sequence data definition, design the seq-ref function: 

;;; seq-ref : [Sequence X] Natural -> X
;;; Get element of sequence at given index.
;;; Assume: given index is < the sequence length.
Problem 3  Design the seq-left and seq-right functions, which are useful for splitting sequences at a given index:

;;; seq-left : [Sequence X] Natural -> [Sequence X]
;;; Sequence of elements [0,i) of given sequence.
;;; Assume: given index i is <= the sequence length.
(define (seq-left seq i) ...)

;;; seq-right : [Sequence X] Natural -> [Sequence X]
;;; Sequence of elements [i,length) of given sequence.
;;; Assume: given index i is <= the sequence length.
(define (seq-right seq i) ...)

Some examples:

(define one-to-five (make-seq 5 add1)) ; 1,2,3,4,5

(seq-left one-to-five 0) ; empty sequence
(seq-left one-to-five 5) ; 1,2,3,4,5

(seq-right one-to-five 0) ; 1,2,3,4,5
(seq-right one-to-five 5) ; empty sequence

(seq-left one-to-five 3) ; 1,2,3
(seq-right one-to-five 3) ; 4,5
[Here is some more space for the previous problem.]
Problem 4  Sorted sequences are very useful, particularly if you want to quickly find out if some value is in a given sequence. For example, given a sorted list of student names, it’s easy to find out if "Flunker, Freddy" is in the class, even if there are lots and lots of students.

Here is an example of a sequence that is sorted according to string<?:

```scheme
;; [Sequence String]
(define roster
  (make-seq 4
    (λ (i)
      (cond [(= i 0) "Ahmed, Amal"]
            [(= i 1) "Flunker, Freddy"]
            [(= i 2) "Shivers, Olin"]
            [(= i 3) "Van Horn, David"]))))
```

Design the function contains? which consumes a sorted sequence of strings, and produces true if the sequence contains the string and false otherwise.

```scheme
;; contains? : [Sequence String] String -> Boolean
;; Does the sequence contain the given string?
```

You will get partial credit for a correct, but inefficient solution that uses a structural recursion design. For full credit, you will need to use an efficient generative recursion design. [Hint: the idea behind the generative recursion design will take advantage of the fact that the sequence is sorted.]
[Here is some more space for the previous problem.]
[Here is some more space for the previous problem.]