# CSU2510H Exam 1 – Practice Problems

Name: ____________________

Student Id (last 4 digits): ____________________

- You may use the usual primitives and expression forms of any of the `class` languages, and the `List` interface on the next page; for everything else, define it.
- You may write *`c`* $\rightarrow$ *`e`* for `(check-expect` *`c e`*`)` and $\lambda$ for `lambda` to save time writing.
- To add a method to an existing class definition, you may write just the method and indicate the appropriate class name rather than re-write the entire class definition.
- If an interface is given to you, you do not need to repeat the contract and purpose statements in your implementations. Likewise, you do not need to repeat any test cases given to you, but you should add tests wherever appropriate.
- Some basic test taking advice: Before you start answering any problems, read *every* problem, so your brain can be thinking about the harder problems in background while you knock off the easy ones.

*Good luck!*

Here is the standard `List` interface that we have been using in class:

```
;; A [List T] is one of
;;   (new cons% T [List T])
;;   (new mt%)
;; A [List T] implements
;; -- fold [U]: (U, T -> U) U -> U
;;      folds the provided function over this list and the provided
;;      initial value
;; -- map [U]: (T -> U) -> [List U]
;;      maps the provided function over this list and returns a new
;;      list of the mapped results
;; -- append: [List T] -> [List T]
;;      returns a new list containing all the elements of this list
;;      followed by all the elements of the provided list
```

Also, you may use

```
;; string-append : String ... -> String
;;   concatenates all the provided strings into a single string
```

As one combined example:

```
(define ex1
  (new cons% "Feb 10" (new cons% "2014" (new cons% "Exam 1" (new mt%)))))
(ex1 . fold (λ(s acc) (string-append s ", " acc)) "good luck!")
  ⟹
  "Feb 10, 2014, Exam 1, good luck!"
```

**Problem 1** A *range* represents a set of numbers between two endpoints. To start with, you only need to consider ranges that *include* the smaller endpoint and *exclude* the larger endpoint—such ranges are called *half-open*. For example, the range $[3, 4.7)$ includes all of the numbers between 3 and $4.7$, including 3 but *not* including $4.7$. So 4 and $3.0000001$ are both in the range, but 5 is not. In the notation used here, the [ means include, and the ) means exclude. 17 POINTS

1. Design a representation for ranges and implement the `in-range?` method, which determines if a number is in the range. For example, the range $[3, 7.2)$ includes the numbers 3 and $5.0137$, but not the numbers $-17$ or $7.2$.

2. Extend the data definition and implementation of ranges to represent ranges that *exclude* the low end of the range and *include* the high end, written $(lo, hi]$.

3. Add a `union` method to the interface for ranges and implement it in all range classes. This method should consume a range and produces a new range that includes all the numbers in this range *and* all the numbers in the given range.

   You may extend your data definition for ranges to support this method.

   You may apply the abstraction recipe, but you are not required to do so; you will not be marked down for duplicated code on this problem.

**Problem 2** Here are data, class, and interface definitions for Shapes: 18 POINTS

```
;; A Shape is one of:
;; - (rect% Number Number)
;; - (circ% Number)
;; implements:
;; bba : -> Number     (short for "bounding-box-area")
;; Compute the area of the smallest bounding box for this shape.

(define-class rect%
  (fields width height))

(define-class circ%
  (fields radius))
```

Here are some examples of how `bounding-box-area` should work:

```
(check-expect ((rect% 3 4) . bba) 12)
(check-expect ((circ% 1.5) . bba)  9)
```

1. Design the `bba` method for the `rect%` and `circ%` class.
2. Design a super class of `rect%` and `circ%` and lift the `bba` method to the super class. Extend the shape interface as needed, but implement any methods you add.
3. Design a new variant of a Shape, Square, which should support all of the methods of the interface.

**Problem 3** The Northeastern University Registrar is instituting a new course registration system, in which each student will wait in a "Virtual Line" until every student ahead of them has registered. A simple way to represent a line (also known as a *queue*) is by using a list. But this representation makes it slow to add somebody to the end of the line (or to take somebody off the front of the line, depending on whether the front of the list represents the front or rear of the line). 21 POINTS

In order to provide maximal waiting efficiency, you have been tasked with implementing a representation that uses *two* lists! The key idea of this fancy representation is that one list will represent some portion of the front of the line, while the other will represent the remainder of the line *in reverse order*. So if you're the first element of the first list, you are at the head of the line. On the other hand, if you're the first element of the second list, you are the very last person in line. Here is the interface for queues:

```
;; A [IQ X] implements:
;;
;; head : -> X
;; Produce the element at the head of this queue.
;; Assume: this queue is not empty.
;;
;; deq : -> [IQ X]      (Short for "dequeue")
;; Produces a new queue like this queue, but without
;; this queue's first element.
;; Assume: this queue is not empty.
;;
;; enq : X -> [IQ X]    (Short for "enqueue")
;; Produce new queue with given element added to the
;; END of this queue.
;;
;; emp? : -> Boolean
;; Is this queue empty?
```

The `head` and `deq` operations require that the queue be non-empty when they are used, but this can be assumed and these operations do not need to check for an empty queue.

Further, the Registrar's office has just learned about *invariants*, and insists on maintaining the following invariant about all of their queues:

*if the front of the queue is empty, the whole queue must also be empty.*

The Registrar's office has given you three tasks to prepare their Virtual Line for its launch later this semester:

1. Design an implementation of the queue data structure to the Registrar's specifications. You must maintain the invariant stated above, and you should take advantage of the invariant when implementing the operations.

2. Unfortunately, when testing the queue, the Registrar has discovered that some queues with the same elements in the same order can be represented in multiple ways. Give an example

of two different representations of the same queue. Implement a `to-list` operation which produces a list of elements going in order from the front to the rear of the queue. In your tests, you should show how this addresses the problem.

3. The Registrar has a problem with careless data entry. Design and implement a constructor for queues which, given two input lists of elements, ensures that the invariant is maintained.

**Problem 4** Here are data and class definitions for representing the files and directories of a computer: 15 POINTS

```
;; An Elem is one of
;; - (new file% String Number)
;; - (new dir% String LoElem)
(define-class file% (fields name size))
(define-class dir% (fields name elems))

;; A LoElem is one of
;; - (new mt%)
;; - (new cons% Elem LoElem)
(define-class mt%)
(define-class cons% (fields first rest))
```

Here is an example of a directory tree and its representation as an `Elem`:

```
#|
 User
   +-- A
   |   +-- a.txt
   |   +-- B
   |       +-- b.rkt
   +-- C
       +-- c.c
|#

(define mt (new mt%))
(define a (new file% "a.txt" 5))
(define b (new file% "b.rkt" 10))
(define c (new file% "c.c" 1000))

(define C/ (new dir% "C" (new cons% c mt)))
(define B/ (new dir% "B" (new cons% b mt)))
(define A/ (new dir% "A" (new cons% a (new cons% B/ mt))))
(define User/ (new dir% "/" (new cons% A/ (new cons% C/ mt))))
```

Your job is to implement one of the main tasks of the common Unix utility `find`, which gives a list of all the files within an element. To do so, add a `list-files` method of `Elem` which produces a list of files. So for example, we expect:

```
(check-expect (User/ . list-files)
              (new cons% a (new cons% b (new cons% c mt))))
```

*Hint*: it may come in handy to be able to concatenate two `LoElem` together.