CSU2510H Exam 1 – Spring 2011

Name: _____

Student Id (last 4 digits): _____

- You may use the usual primitives and expression forms of any of the `class` languages; for everything else, define it.
- You may write `c` $\rightarrow$ `e` for `(check-expect c e)` and λ 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.
- We expect data *and* interface definitions (although interfaces may be defined in comments rather than with `define-interface`).
- 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.

Problem	Points	/out of
1		/ 17
2		/ 18
3		/ 12
4		/ 21
5		/ 15
Total		/ 83

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.

[Here is some more space for the previous problem.]

[Here is some more space for the previous 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.

[Here is some more space for the previous problem.]

[Here is some more space for the previous problem.]

Problem 3 One perspective that unifies the paradigms of programming with functions and programming with objects is to view a "function" as an object that understands a method called `apply`. With that in mind, we can define an interface for functions-as-objects:

12 POINTS

```
;; A [IFun X Y] implements:
;; apply : X -> Y
;; Apply this function to the given input.
```

Here we are representing a `X -> Y` function as an object that has an `apply` method that consumes an `X` and produces a `Y`.

1. Design a class that wraps a real function with contract `(X -> Y)` to implement `[IFun X Y]`.

2. Using your wrapper class, construct the objects representing the "add 1" function and "sub 1" function.

3. Another useful operation on functions is composition. Here is the interface for a `compose` method that composes two functions represented as objects:

   ```
   ;; [IFun X Y] also implements:
   ;; compose : [IFun Y Z] -> [IFun X Z]
   ;; Produce a function that applies this function to its input,
   ;; then applies the given function to that result.
   ```

   For example, if `addone` and `subone` refer to the objects you constructed in part 2, the following check should succeed:

   ```
   (check-expect ((addone . compose subone) . apply 5) 5)
   (check-expect ((addone . compose addone) . apply 5) 7)
   ```

   Implement the `compose` method for your wrapper class.

[Here is some more space for the previous problem.]

[Here is some more space for the previous problem.]

Problem 4 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.

[Here is some more space for the previous problem.]

[Here is some more space for the previous problem.]

Problem 5 Here is a simple interface and implementation for dictionaries that associate natural numbers to strings. 15 POINTS

```
;; A Dict is one of:
;;  - (ld-empty%)
;;  - (ld-cons% Nat String Dict)
;; implements:
;;
;; has-key? : Nat -> Boolean
;; Does this dict. have the given key?
;;
;; lookup : Nat -> String
;; Produce the value associated with the given key in this dict.
;; Assume: the key exists in this dict.
;;
;; set : Nat String -> Dict
;; Associate the given key to the given value in this dict.

(define-class ld-empty%
  (define/public (has-key? k) false)

  (define/public (set k v)
    (ld-cons% k v this)))

(define-class ld-cons%
  (fields key val rest)

  (define/public (has-key? k)
    (cond [(= (field key) k) true]
          [else ((field rest) . has-key? k)]))

  (define/public (lookup k)
    (cond [(= (field key) k) (field val)]
          [else ((field rest) . lookup k)]))

  (define/public (set k v)
    (ld-cons% k v this)))
```

The dictionary data structure has some important properties, one of which is that if we associate a value with a key and then look up that key, we expect to get back the value we associated. For example, for any given dictionary, if we associate `1` with `"Fred"`, then look up `1`, we should get `"Fred"`. More generally, we expect this to be true for *any* key-value pair, not just `1` and `"Fred"`.

1. Codify the general "set/lookup" property, described above, as a predicate. Write a successful test in terms of that predicate.

Now consider a generalization of the "set/lookup" property which says that if we set *two* keys to *two* values then look each of them up, we should get back the original values. For example, for any given dictionary, if we associate `1` with `"Fred"` and `2` with `"Wilma"`; then looking up `1` will produce `"Fred"` and looking up `2` will produce `"Wilma¨`. More generally, we can state this as a property for any two key-value pairs.

2. Codify the general "double-set/lookup" property as a predicate. Write a successful test in terms of that predicate.

3. Does this property hold for all dictionaries and keys-value pairs? If so, why? If not, write a counter-example, i.e. write a failing test in terms of the predicate you gave in part 2.

[Here is some more space for the previous problem.]

[Here is some more space for the previous problem.]