CS2500 Exam 2 — Fall 2011

Name: ________________________

Student Id (last 4 digits): ________________________

Section (morning, honors or afternoon): ________________________

- Write down the answers in the space provided.
- You may use the usual primitives and expression forms, including those suggested in hints; for everything else, define it.
- You may write `c` → `e` in place of `(check-expect c e)` to save time writing. You may also write the Greek letter λ instead of `lambda`, to save writing.
- Some basic test taking advice: (1) 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. (2) Write your name at the top of every page... just in case the staples come out of your completed test. (This happens.)

Problem	Points	/out of
1		/ 15
2		/ 10
3		/ 12
4		/ 10
5		/ 20
6		/ 9
7		/ 14
Total		/ 90

Good luck!

Problem 1 Suppose we are processing a collection of rectangles, where rectangles are given with the following data definition: 15 POINTS

```
;;; A Rectangle is a (make-rect Number Number)
(define-struct rect (width height))
```

We'd like to search a list of rectangles to select out all the squares in the list (that is, the rectangles that are as wide as they are high).

- Design a function, `all-squares`, that takes a list of rectangles, and produces a list of all the square rectangles in the lists. Do not use loop functions.
- Now rewrite the function using loop functions.

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

Problem 2 Wait, we're not done with lists of rectangles yet. We need a function, `flip-rectangles`, that will take a list of rectangles and "flip" each one, swapping its width with its height. For example, the rectangle `(make-rect 3 8)` in the input list would become the rectangle `(make-rect 8 3)` in the output list. 10 POINTS

Design this function using loop functions.

Problem 3 The function `number-winners` takes a list and a test function, and returns the number of “winners” in the list—that is, the number of items in the input list that cause the test function to return true. Some examples: 12 POINTS

```
(number-winners even?   (list 2 5 2 7 9 1 4))   ; => 3
(number-winners string? (list 0 "a" 3 1 "b" 7)) ; => 2
```

Design this function using any loop function you like. . . except `filter`.

Problem 4 Recall from class[1] our representation of *numeric sets* with lists: 10 POINTS

```
;;; An NSet (set of numbers) is a [Listof Number]
;;; - Order of elements is unimportant, of course.
;;; - No repeats allowed: a number may appear in the list
;;;   at most once.
```

Set subtraction, written $A - B$, is the set of all elements in set A that are not in set B. For example,

$$\{1, 2, 3, 4, 5, 6\} - \{2, 4, 6\} = \{1, 3, 5\}$$

Design the set-subtraction function `set-` using a loop function. You may assume the function `contains?` has already been written (after all, we did it in class[2]). So you can use `contains?` in your solution without having to write it yourself:

```
;;; NSet Number -> Boolean
;;; Does the set contain the number?
(define (contains? set num) ...) ; Already written for you.
```

[1] Assuming you've been attending classes, that is.
[2] Assuming you've been attending classes, that is.

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

Problem 5 You have a summer job developing code at a company that uses a cheap, cut-rate Scheme system, PowerSkeem![3] The bad news is that PowerSkeem!, among many other problems, doesn't have the `map` or `filter` functions. The partial good news is that it *does* have a `foldr` function. 20 POINTS

It's going to take you some time to convince your boss to switch over to Dr. Racket. In the meantime, you'd like to program using `map` and `filter`. So you need to define them first. Fortunately, you're sophisticated enough to realize that you can write both `map` and `filter` rather compactly using `foldr`.

Your task: Design `map` and `filter` using `foldr`. You may use `lambda` or `local`, if needed.

[3] Written by the CEO's nephew. Welcome to the working world.

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

Problem 6 The natural numbers (integers greater than or equal to zero) can be described by a recursive-union data definition, much like lists: 9 POINTS

```
;; A Natural is one of:
;; - 0
;; - (add1 Natural)
```

Just as we can design "loop" functions for lists, we can do the same for natural numbers. Consider the following "foldr" analog for natural numbers:

```
(define (nat-foldr op base nat)
  (cond [(zero? n) base]
        [else (op nat
                  (nat-foldr op base (sub1 nat)))]))
```

Now consider the following recursive function on natural numbers:

```
;; factorial : Natural -> Natural
;; Compute n! = 1 * 2 * ... * n-1 * n.
;; (As a special case, 0! = 1.)
(define (factorial n)
  (cond [(zero? n) 1]
        [else (* n
                 (factorial (sub1 n)))]))
(check-expect (factorial 0)  1)
(check-expect (factorial 5)  120)
(check-expect (factorial 20) 2432902008176640000)
```

- What is the contract for `nat-foldr`?
- Rewrite `factorial` using `nat-foldr`. (No need for contract, purpose statement or tests — just rewrite the code.)

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

Problem 7 Consider the following data definition: 14 POINTS

```
;; An Exp (arithmetic expression) is one of:
;; - Number
;; - (make-op Exp [Number Number -> Number] Exp)
(define-struct op (left fun right))
```

We can use Exps to represent arithmetic expressions made up of numbers and two-argument operators such as addition, subtraction, division, *etc.*

- Translate the following Intermediate Student Language expressions into Exps:

```
(+ 1 2)
(+ (* 2 3) 4)
(* (* 3 2) (+ 2 3))
```

- Design a function, `evaluate`, that will take an Exp and carry out the arithmetic computation it describes.

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