THE UNIVERSITY OF BRITISH COLUMBIA
CPSC 110: FINAL EXAMINATION
DECEMBER 16TH, 2010

Name: _______________________    Student #: _________________    CS Dept. ID #: ________

Signature: _____________________    Lab Section: _____     Lecture Section:_____

Important notes about this examination
1. You have 2 hours and 15 minutes to write this examination.
2. You may use pencil or pen.
3. No questions will be answered.
4. No one may leave the room for the first or last 30 minutes of the exam.
5. Follow the design recipes!

Rules Governing Formal Examinations
1. Each candidate must be prepared to produce, upon request, a UBC card for identification.
2. Candidates are not permitted to ask questions of the invigilators, except in cases of supposed errors or ambiguities in examination questions.
3. No candidate shall be permitted to enter the examination room after the expiration of one-half hour from the scheduled starting time, or to leave during the first half hour of the examination.
4. Candidates suspected of any of the following, or similar, dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action:
   ○ having at the place of writing any books, papers or memoranda, calculators, computers, sound or image players/recorders/transmitters (including telephones), or other memory aid devices, other than those authorized by the examiners;
   ○ speaking or communicating with other candidates; and
   ○ purposely exposing written papers to the view of other candidates or imaging devices. The plea of accident or forgetfulness shall not be received.
5. Candidates must not destroy or mutilate any examination material; must hand in all examination papers; and must not take any examination material from the examination room without permission of the invigilator.
6. Candidates must follow any additional examination rules or directions communicated by the instructor or invigilator.

Please do not write in this space:

<table>
<thead>
<tr>
<th>Question</th>
<th>Mark</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Extra Credit</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
Questions 1, 2 and 3 are all based on the following types comment and corresponding function template:

```
(define-struct foo (fn ln subs))
;; Foo is (make-foo String String ListOfFoo)
;;
;; ListOfFoo is one of:
;;  - empty
;;  - (cons Foo ListOfFoo)
;;
#
(define (fn-for-foo f0)
  (local [(define (fn-for-foo f)
            (... (foo-fn f)
                 (foo-ln f)
                 (fn-for-lof (foo-subs f))))
          
          (define (fn-for-lof lof)
            (cond [(empty? lof) (...)]
                  [else
                   (... (fn-for-foo (first lof))
                         (fn-for-lof (rest lof)))])))))

(fn-for-foo f0))
```

**Problem 1 - (15 points)**

Read this entire problem carefully before making any marks on the types comment or template above. Then make your marks neatly.

(A) Annotate the TYPES COMMENT (not the template) with arrows to indicate reference, self-reference and mutual reference. Label each arrow with R, SR or MR; and number each arrow 1, 2 etc.

(B) Annotate the TEMPLATE (not the types comment) as follows: for each function call that corresponds directly to an arrow on the types comment, mark the function call with the number of the corresponding arrow.
Problem 2 - (18 points)

NOTE: For this question do not make any marks on the types comment or function template above.

Now consider the following potential changes to the above type comments:

- Introduce a new type called Name as follows:

    ;; Name is String

- Replace all occurrences of String with Name.

- Add a new field of type Integer to Foo.

(A) How many more (or fewer) arrows of each of the following kinds would need to be added to update the arrows annotations from problem 1? Do not add arrows to the types comments above. Just fill in below how many more (or fewer) arrows there would be:

    Reference:  __________

    Self-Reference:  __________

    Mutual reference:  __________

(B) Write the revised version of the locally defined fn-for-foo in the space below.
Problem 3 - (5 points)

Which of the following data structures we have studied is Foo (together with ListOfFoo) in problem 1 most like? (Circle one.)

- directed acyclic graph
- arbitrary-arity tree
- binary tree
Problem 4 - (10 points)

Given this data definition for binary trees:

```
(define-struct node (k v l r))
;; BTree is one of:
;;  - false
;;  - (make-node Number String BTree BTree)
;; interp. A binary tree where each node has a key and value
```

```
(define BT1 (make-node 1 "1" false false))
(define BT2 (make-node 2 "2" false false))
(define BT3 (make-node 3 "3" BT1 BT2))
(define BT4 (make-node 4 "4" BT1 BT3))
```

Consider the following initial design of the function `bt=?`, which has been designed following the recipe for the general case of a function operating on two complex data.

```
;; BTree BTree -> Boolean
;; produce true if t1 and t2 have same shape and
;; same key and value at each node
(check-expect (bt=? false false) true)
(check-expect (bt=? false BT1) false)
(check-expect (bt=? BT1 false) false)
(check-expect (bt=? BT4 BT3) false)
(check-expect (bt=? BT4 BT4) true)
```

```
(define (bt=? t1 t2)
  (cond [(and (false? t1) (false? t2)) true]
        [(and (false? t1) (node? t2)) false]
        [(and (node? t1) (false? t2)) false]
        [(and (node? t1) (node? t2))
         (and (= (node-k t1) (node-k t2))
              (string=? (node-v t1) (node-v t2))
              (bt=? (node-l t1) (node-l t2))
              (bt=? (node-r t1) (node-r t2))))]
  )
```

Now follow a systematic process to simplify this function. Full credit will be awarded to solutions that:
- show explicit intermediate steps of the simplification
- explain the simplification done at each step
- end up with the simplest possible code

NOTE: To save space, you do not need to rewrite the entire body of the function to explain a simplification step. You only need to write enough to make it completely clear what simplification you are doing. Do your work on the next page (or two).
(if needed continue your solution to problem 4 here)
Problem 5 - (15 points)

Ron loves the snowflakes program and the balloons program so much that he would like to have balloons floating around in the snow. He combines the two to get a program that works properly. His program includes the following two functions:

;; (listof Balloon/Flake) Image -> Image
;; render a list of balloons/snowflakes onto image
;;
;; <<<<tests removed to save space>>>>

(define (render-balloons lob img)
  (cond [(empty? lob) img]
        [else
         (render-balloons (rest lob)
                          (place-balloon (first lob) img))]))

(define (render-flakes lof img)
  (cond [(empty? lof) img]
        [else
         (render-flakes (rest lof)
                        (place-flake (first lof) img))]))

Hermione takes one look and says "are you a wizard or not? Look how repetitive those functions are! You can do better than that." (She is always pretty hard on Ron.)

Use function abstraction to produce an abstract function that abstracts the commonality between render-balloons and render-flakes. Be sure to develop a complete design for the abstract function, including signature, purpose and tests. Also be sure to rewrite render-balloons and render-flakes in terms of the abstract function. You do not need to include signature, purpose or tests for your revised render-balloons and render-flakes.

NOTE: Even if you believe that one of the built-in abstract functions can be used you must do the complete function abstraction recipe to produce a new abstract function.

(use next page to continue your solution to problem 5)
(continue your solution to problem 5 here)
Problem 6 - (12 points)

In the arcade game Skee Ball players roll a ball and try to get it into one of several rings. At the end of the game the score can be computed by awarding 1 point for each ball in the first ring, 2 points for each ball in the second ring and so on. The following data definition describes how to represent the balls at the end of the game:

;;; Balls is (listof Number)
;;; interp. number of balls in each ring at the end of the game
;;; (list 3 1 2) means 3 balls in the 1 point ring
;;; 1 ball in the 2 point ring
;;; 2 balls in the 3 point ring
(define BALLS1 (list 3 1 2))
(define BALLS2 (list 2 1))

Design a function called score that consumes Balls and produces the score corresponding to that arrangement of balls. Do not assume any fixed number of rings. For example:

(score (list 3 1 2)) ; produces 3*1 + 1*2 + 2*3 = 11
(score (list 2 1))   ; produces 2*1 + 1*2       =  4

You may design either a non-tail recursive function with one accumulator, or a tail recursive function with two accumulators. As usual, when we say “design” we mean we want all the parts of a proper design.

(use next page to continue your solution to problem 6)
(continue your solution to problem 6 here)
Problem 7 - (15 points)

The balloons program we looked at in class would look more realistic if balloons bounced off each other as well as off of the sides of the box. Seamus says he knows how to figure out how a balloon's direction should change when it hits other balloons, but he needs to know all the balloons involved when a balloon hits others. To help Seamus you need to design a function called all-touching that consumes two arguments, a single balloon and a list of balloons, and produces a list of all the balloons in the list that are touching the single balloon. You are free to use any of the built-in abstract list functions listed below. As usual, when we say “design” we mean we want all the parts of a proper design.

You should assume the existence of a function called touch? as follows:

```plaintext
;; Balloon Balloon -> Boolean  ;; produce true if the two balloons are touching, false otherwise

In your examples you may make reasonable assumptions about how close balloons have to be in order to be touching.

```plaintext
;; N (N -> X) -> (listof X)  ;; to construct (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

;; (X -> boolean) (listof X) -> (listof X)  ;; to construct a list from all items on alox for which p holds
(define (filter p alox) ...)

;; (X -> Y) (listof X) -> (listof Y)  ;; to construct a list by applying f to each item on alox
;; (map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))
(define (map f lox) ...)

;; (X -> boolean) (listof X) -> boolean  ;; to determine whether p holds for every item on alox
;; (andmap p (list x-1 ... x-n)) = (and (p x-1) (and ... (p x-n)))
(define (andmap p lox) ...)

;; (X -> boolean) (listof X) -> boolean  ;; to determine whether p holds for every item on alox
;; (ormap p (list x-1 ... x-n)) = (or (p x-1) (or ... (p x-n)))
(define (ormap p lox) ...)
```

(USE THE NEXT PAGE TO WRITE YOUR SOLUTION TO PROBLEM 7)
Problem 8 - (10 points)

In the Google personnel system each employee has a first name, last name and a salary. (It is more complex than that of course, but stick with this simple version.)

In the space below, do TWO things:
(A) Design a data definition to represent employees.
(B) Design a function that consumes Employee, produces void and mutates the employee to give them a 10% raise.

As usual, when we say “design” we mean we want all the parts of a proper design.
(continue your solution to problem 8 here)
Extra Credit - (6 points)

(Since this is an extra credit problem, partial credit will be awarded very sparingly. A correct or nearly correct solution will be required to receive points.)

Recall the data definition for a binary search tree:

(define-struct node (k v l r))
;; A BST (Binary Search Tree) is one of:
;; - false
;; - (make-node Integer String BST BST)
;; interp. A binary search tree; each node has a key and value
;;
;; Invariants:
;; - every key in the left branch of a node is smaller
;;   the node's own key
;; - every key in the right branch of a node is larger
;;   the node's own key

(define BST1 false)
(define BST2 (make-node 2 "two" false false))
(define BST4 (make-node 4 "four" false false))
(define BST3 (make-node 3 "three" BST2 BST4))

In other words, a binary search tree is a binary tree in which the two invariants above are satisfied.

Design the function check-bst? that consumes a binary tree and produces true if both invariants hold and false otherwise. You may assume that the tree does not contain the same key more than once.

NOTE: DrRacket provide two special numbers, positive and negative infinity, which are written as +inf.0 and -inf.0. +inf.0 is greater than any other number; -inf.0 is less than any other number. So, for example:

\[
\begin{align*}
& (< 49538750375 +inf.0) \quad \text{; is true} \\
& (< -inf.0 -23059570845734) \quad \text{; is true}
\end{align*}
\]

HINT: You will want to use 2 accumulators.

Be sure to follow the recipe. Start by writing examples that should produce true and false. Follow the accumulator design recipe and write examples for the inner function as well.

( use the next page to write your solution to the extra credit problem )
(write your solution to the extra credit problem here)
(overflow space for other problems. If you write anything here CLEARLY MARK WHICH
PROBLEM IT IS PART OF THE SOLUTION TO, AND CLEARLY MARK AT THE
ORIGINAL PROBLEM THAT PART OF YOUR SOLUTION IS AT THE END OF THE
EXAM.)
(overflow space for other problems. If you write anything here CLEARLY MARK WHICH PROBLEM IT IS PART OF THE SOLUTION TO, AND CLEARLY MARK AT THE ORIGINAL PROBLEM THAT PART OF YOUR SOLUTION IS AT THE END OF THE EXAM.)
(overflow space for other problems. If you write anything here CLEARLY MARK WHICH
PROBLEM IT IS PART OF THE SOLUTION TO, AND CLEARLY MARK AT THE
ORIGINAL PROBLEM THAT PART OF YOUR SOLUTION IS AT THE END OF THE
EXAM.)