UBC CPSC 110 Midterm 2, November 4th 2009

Name: ____

UBC ID #: ____

Please do not open this exam until we ask you to do so. But please do read this entire first page now.

Please write your answers neatly and only on the front side of each page.

Some of the problems on this midterm ask for written answers, rather than code, numbers or letters. The best answers to such questions are short, concrete and to-the-point. Sometimes a small example program can help to make the answer clear. Do not attempt to write 'cover all the bases' answers to such questions – overly long answers will receive less credit.

This midterm is designed so that to the greatest degree possible it does not require you to remember details like the name or exact order of arguments to a four argument big-bang handler function. If you find an exception to this don't worry about it, use a reasonable name and order of arguments. Points will not be deducted for details like that.

Q1 -		/ 10
Q2 -		/ 40
Q3 -		/ 20
Q4 -		/ 15
Q5 -		/ 15
EXTRA CREDIT		
TOTAL		

(1) (10 points)

Consider the following program which consists of two definitions followed by an expression.

```
(define x 0)
(define y 1)

(+ x
   (local [(define x 2)
           (define y 3)
           (define (foo z)
             (* x y z))]
     (foo x)))
```

(1a) What is the value of the expression?

(1b) Show the first step of evaluating the local (which is the 2nd step of evaluating the expression). You should show any lifted definitions, as well as showing the renamed body of the local in the expression in which it appears.
SUGGESTION: Check your answer by making sure it will evaluate to the same answer you gave in part a.

(2) (40 points)

In this question you will be working with the following structure and data definition:

```
(define-struct node (value l r))
;;
;; A BTree is one of:
;;   - false
;;   -(make-node v l r) where
;;       v is a Number
;;       l is a BTree
;;       r is a BTree
;;
```

(2a) Show two examples of BTrees. We are not asking you for test cases here, just two expressions that produce valid BTrees.

(2b) The BTree data definition is (circle all that apply):

an enumeration

an itemization

an interval

a data definition involving multiple cases

a self-referential data definition

mutually-referential data definitions

(2c) IF you decided that the BTree data definition was self- or mutually-referential, draw an arrow on the data definition from each such reference back to the data definition. Label your arrows (a), (b), etc.

(2d) Write the template for a function operating on BTrees. If you drew arrows on your data definition then mark the natural recursions in your template with the corresponding letter (a), (b), etc.

(2e) Design the function bt-sum which consumes a BTree and returns the sum of all the node-values in the tree.

(3) (20 points)

In this problem you will be working with the following structure and data definitions.

```
(define-struct node (left right))
;; Tree is one of:
;;  - Number
;;  - (make-node Tree Tree)
;;
;; Path is one of:
;;  - empty
;;  - (cons "L" Path)
;;  - (cons "R" Path)
```

(3a) Give two examples of a Tree. We are not asking you for test cases here, just two expressions that produce valid Trees.

(3b) Give two examples of a Path.

(3c) Show the template for a function that consumes a Tree and a Path.

(this page is intentionally left blank)

(4) (15 points)

Consider the following only-as and above functions:

```
;; only-as: (listof String) -> (listof String)
;; keep only the "a"s from los
(check-expect (only-as empty) empty)
(check-expect (only-as '("a" "b" "a")) '("a" "a"))

(define (only-as los)
  (cond [(empty? los) empty]
        [else (cond [(string=? (first los) "a")
                     (cons (first los)
                           (only-as (rest los)))]
                    [else (only-as (rest los))])]))

;; above: Number (listof Number) -> (listof Number)
;; keep only those numbers in lon above n
(check-expect (above 2 empty) empty)
(check-expect (above 2 '(1 2 3 4)) '(3 4))

(define (above n lon)
  (cond [(empty? lon) empty]
        [else (cond [(> (first lon) n)
                     (cons (first lon)
                           (above n (rest lon)))]
                    [else (above n (rest lon))])]))
```

(4a) Use functional abstraction to write the definition of an abstraction of these two functions. Also include new definitions of only-as and above that use the new function. Extra credit if your abstract function accepts 2 arguments instead of 3. But, a SUGGESTION, write the 3 argument version first and then the 2 argument version. (We have given you an extra blank page to do so.)

(this page is intentionally left blank)

(4b) What is the contract of your abstract function?
If you can make it more abstract do so.

(4c) What is the usual name of the 2 argument version of this abstract function in ISL and Scheme?

(5) (15 points)

Consider the following structures and data definitions.

```
(define-struct foo (a))
(define-struct bar (b))
;; A Snarfle is one of:
;;   - a Number
;;   - a (make-foo w) where w is a Whatzzle
;;
;; A Whatzzle is one of:
;;   - a String
;;   - a (make-bar s) where s is a Snarfle
```

(5a) Show THREE examples of a Snarfle:

(5b) Show the template for function(s) operating on a Snarfle:

EXTRA CREDIT

Define the function(s) snarfle-contents that consumes a Snarfle and returns (listof Number/ String) that contains the numbers/strings reachable from the Snarfle.

What strikes you as strange about these data definitions? (Aside from their names.)