UBC CPSC 110 Midterm 2, March 15th 2010

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.

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.

(1) Lexical Scoping and Local (15 points) __________

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

```
(define x 1)
(define y 2)

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

(1a) Draw a line from the three variable references (x, x and y) to the definition that provides their value.

(1b) Write out the renamed and lifted local definitions.

(1c) Write out the + expression, AFTER the local defines have been lifted and the call to foo has been replaced with its body, but BEFORE beginning to evaluate the body of foo.

(2) Trees (35 points) __________

Consider:

```
(define-struct unit (name subs))
;;
;; A Unit is (make-unit n subs) where
;;   n is String
;;   subs is (listof Unit)
;;
```

(2a) When we write (listof Unit), it is shorthand for a longer data definition of a kind we wrote many times earlier in the term. Write out that longer version of the data definition.

(2b) Show two examples of Unit. We are not asking you for test cases here, just two expressions that produce Units.

(2c) For each of the lines below, circle the one of the two terms that most accurately describes the given data definitions.

Unit:	binary tree	n-ary tree
Unit:	fixed size	arbitrary size
(listof Unit):	fixed size	arbitrary size
(listof Unit):	involved in mutual-reference	self-reference
Unit:	single case	multiple cases

(2d) Complete the templates below. You only need to write the question parts of any cond expressions, you do not need to write the answer parts. BUT, do put an "NR" or "MR" in any answer part that will probably contain a natural recursion or a mutual recursion in a completed function.

```
;; fun-for-unit: Unit -> ??
(define (fun-for-unit u)







)

;; fun-for-lou: (listof Unit) -> ??
(define (fun-for-lou lou)











)
```

(2e) Design the function count-units which consumes a Unit and returns a count of how many units it comprises, including itself <u>and</u> all subs.

(3) Two Complex Data (20 points) __________

Design the function intersection, which consumes two lists of numbers, and produces a list containing all numbers in the first list that also appear in the second list. Remember that "design the function" means you need to write data contract, purpose, tests and the function. You are free to use or not to use abstract functions in your answer. Hint: First decide whether this is the general case of a function operating on two complex data or one of the special cases.

(4) Abstract Functions (15 points) __________

Rewrite the functions below more concisely using one of the following abstract functions.

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

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

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

;; foldr : (X Y -> Y) Y (listof X) -> Y
;; (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
(define (foldr f base alox) ...)

;; foldl : (X Y -> Y) Y (listof X) -> Y
;; (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
(define (foldl f base alox) ...)
```

(4a)

```
;; squares: (listof Number) -> (listof Number)
(check-expect (squares (list 2 3 4)) (list 4 9 16))

(define (squares lon)
  (cond [(empty? lon) empty]
        [else
         (cons (sqr (first lon))
               (squares (rest lon)))]))
```

(4b)
```
;; powers: Number (listof Number) -> (listof Number)
(check-expect (powers 3 (list 2 3 4)) (list 8 27 64))

(define (powers n lon)
  (cond [(empty? lon) empty]
        [else
         (cons (expt (first lon) n)
               (powers n (rest lon)))]))
```

(4c)
```
;; just-within: (listof Posn) -> (listof Posn)
(check-expect (just-within (list (make-posn 1 3)
                                 (make-posn -1 3)))
              (list (make-posn 1 3)))

(define (just-within lop)
  (cond [(empty? lop) empty]
        [else
         (if (within? (first lop))
             (cons (first lop) (just-within (rest lop)))
             (just-within (rest lop)))]))
```

(4d)

```
;; digits->num: (listof Digit) -> Integer
(check-expect (digits->num empty) 0)
(check-expect (digits->num (list 2)) 2)
(check-expect (digits->num (list 3 2 4)) 423)

(define (digits->num lod)
  (cond [(empty? lod) 0]
        [else
         (+ (first lod)
            (* 10 (digits->num (rest lod))))]))
```

(5) (15 points) Function Simplification __________

The following function works with the unit data definition of problem 2, although you shouldn't need that data definition to solve this problem. Consider:

```
;; equal-units?: Unit Unit -> Boolean
(define (equal-units? u1 u2)
  (local [(define (equal-units? u1 u2)
            (and (string=? (unit-name u1) (unit-name u2))
                 (equal-lou? (unit-subs u1) (unit-subs u2))))
          (define (equal-lou? lou1 lou2)
            (cond [(and (empty? lou1) (empty? lou2)) true]
                  [(and (empty? lou1) (cons?  lou2)) false]
                  [(and (cons?  lou1) (empty? lou2)) false]
                  [else
                   (and (equal-units? (first lou1) (first lou2))
                        (equal-lou? (rest lou1)
                                    (rest lou2)))]))]
    (equal-units? u1 u2)))
```

Simplify the equal-lou? local function. To save yourself extra writing, you can just circle the part above that is amenable to simplification and then just write the simplifications for that part below. If you write just the final simplified form and it is completely correct you will receive full marks. For partial credit, write at least two simplification steps, and for each step write both the new code as well as a short plain english description of what you have done.

EXTRA CREDIT (10 Points) __________

A more natural version of digits->num would consume a (listof Digit) that was in the same order as the digits in the resulting number. In other words, a natural digits to num function would have these tests:

```
(check-expect (ndigits->num empty)          0)
(check-expect (ndigits->num     (list 2))   2)
(check-expect (ndigits->num   (list 3 2))  32)
(check-expect (ndigits->num (list 1 3 2)) 132)
```

ndigits->num can be defined succinctly using foldl (that's fold left). Show how to do this: