THE UNIVERSITY OF BRITISH COLUMBIA
CPSC 110: MIDTERM EXAMINATION
NOVEMBER 2ND, 2010

Name: _____ Student #:_____ CS Dept. ID #:_____

Signature: _____ Lab Section:_____ Lecture Section:_____

Important notes about this examination

1. You have 90 minutes to write this examination.
2. **Follow the design recipes!** You have been given a copy of the template redux page. Use it.
3. Put away your books, notebooks, laptops, cell phones... everything but pens, pencils, erasers and this exam.
4. Good luck!

Rules Governing Formal Examinations

1. Each candidate must be prepared to produce, upon request, a UBCcard 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:

Question	Mark	Max
1		5
2		10
3		30
A		
B		
C		
D		
4		25
5		20
6		10
Total		**100**

Problem 1 - (10 points)

There are 5 variable references in this short program. Neatly draw arrows from each reference to the parameter or `define` that provides the value for it. (We are looking for arrows like those that the DrRacket "check syntax" button enables, but using curved lines will probably help make your work neater.)

```
(define a 1)

(define (foo b)

  (local [(define (bar c)

            (* a b c))

          (define a 10)]

    (bar a)))

(foo a)
```

Problem 2 - (10 points)

The following program has a definition followed by an expression.

```
(define (f1 n lon)
  (local [(define (f2 m) (* n m))]
    (map f2 lon)))

(f1 3 (list 2 4 6))
```

(A) What is the value of the expression?

(B) Show, in the space above, the state of the hand execution, including lifted definitions, right before the call to map is evaluated.

Problem 3 - (30 points)

Note: this question has 4 parts, A, B, C and D. Be sure to do all four parts.

Consider the following structure definitions and type comments:

```
(define-struct kvp (k v))
;; KVP is (make-kvp Integer String)
;; interp. a key/value pair

(define-struct node (p subs))
;; Tree is (make-node KVP ListOfTree)
;; interp. arbitrary arity tree, each node has kvp and sub-nodes

;; ListOfTree is one of:
;; - empty
;; - (cons Tree ListOfTree)
```

(A) Define two examples for each of the above three types (6 examples total). Feel free to use variables you define for some examples in the definition of other examples.

(B) Neatly annotate the type comments above by drawing a line from each type reference to the the corresponding type definition. All your arrows should end at one of the names that appear right before 'is'. Neatly label each line with one of MR, SR or R, depending on whether the reference is a mutual reference, self reference or an ordinary reference.

(C) Design the function **`sum-keys`**, which consumes a Tree and produces the sum of the keys in every KVP in the tree. When designing the function you do not have to leave your template behind, but using the redux page provided you should write down, in order, the names of the redux rules you used to generate the template.

(D) Now consider the design of the function `sum-val-lengths`, which consumes a Tree and produces the sum of the string lengths of the keys. Will the design of `sum-val-lengths` include the same number of functions as `sum-keys`? Why or why not?

Problem 4 - (25 points)

In this problem you will design a function to merge two lists of numbers that are already sorted in ascending order into a single list of numbers, also sorted in ascending order. Call the function `merge`. For example:

`(merge (list 3 6) (list 1 4 8))` produces `(list 1 3 4 6 8)`

In addition to the usual signature, purpose and examples, your solution should show the step by step process going from the first version of the function, where the template has been filled in to get code that correctly passes the tests, to the final version after simplification.

(continue your solution to problem 4 here)

Problem 5 - (20 points)

Below are two functions excerpted from a version of the fireworks world program. Improve the code for these functions using the built in abstract list functions listed in the appendix to this midterm. You only need to rewrite the body of each function - the signature, purpose and examples/tests are already there for you. You only need to use one abstract list function in each of these functions.

```
;; (listof Firework) -> (listof Firework)
;; tick each firework in lofw
(check-expect (tick-fws empty) empty)
(check-expect (tick-fws (list (make-fw 10 300 "red" 20)
                              (make-fw 10 100 "blue" 0)))
              (list (make-fw 10 (- 300 SPEED) "red" 19)
                    (make-fw 10 (- 100 SPEED) "blue" -1)))

(define (tick-fws lofw)
  (cond [(empty? lofw) empty]
        [else
         (cons (tick-fw (first lofw))
               (tick-fws (rest lofw)))]))
```

```
;; (listof Firework) -> Image
;; render all fireworks in lofw onto MTS
(check-expect (render-fws empty) BG)
(check-expect (render-fws (list (make-fw 10 20 "red" 1)
                                (make-fw 30 40 "blue" 0)))
              (place-image (circle RADIUS "solid" "red")
                           10 20
                           (place-image
                             (radial-star 8 8 30 "solid" "blue")
                             30 40
                             BG)))

(define (render-fws lofw)
  (cond [(empty? lofw) BG]
        [else
         (place-fw (first lofw)
                   (render-fws (rest lofw)))]))
```

Problem 6 - (10 points)

The template for a function operating on (listof X) is

```
(define (fn-for-lox lox)
  (cond [(empty? lox) (...)]
        [else
          (... (first lox)
               (fn-for-lox (rest lox)))]))
```

The template for a function operating on Natural is

```
(define (fn-for-nat n)
  (cond [(zero? n) (...)]
        [else
          (... n
              (fn-for-nat (sub1 n)))]))
```

The abstract functions based on these templates are called `fold` and `natfun`.

Using `fold`, it is easy to write a function that consumes `(listof Number)` and produces the sum of the numbers in the list.

```
;; (listof Number) -> Number
;; produce sum of elements of lon
(check-expect (sumlon (list 3 2 1 0)) 6)

(define (sumlon lon) (fold + 0 lon))
```

Using `natfun` it is easy to write a function that consumes a `Natural n` and produces the sum of the first `n` natural numbers.

```
;; Natural -> Natural
;; produce sum of first n natural numbers
(check-expect (sumnat 3) 6)

(define (sumnat n) (natfun + 0 n))
```

Now consider designing a function called `red` that abstracts both `fold` and `natfun`. Such a function could be used to write both `sumlon` and `sumnat`.

(A) How many parameters would the abstract function `red` have?

(B) Write the definitions of `sumlon` and `sumnat` assuming the abstract function exists. You do not have to write the abstract function, but you may find it helpful to do so.