CS421 Fall 2012 Midterm 1

Name:	
NetID:	

- You have **75 minutes** to complete this exam.
- This is a **closed-book** exam. You are allowed one 3×5 inch (or smaller) card of notes (both sides may be used). This card is **not shared**. All other materials (e.g., calculators), except writing utensils are prohibited.
- Do not share anything with other students. Do not talk to other students. Do not look at another students exam. Do not expose your exam to easy viewing by other students. Violation of any of these rules will count as cheating.
- If you believe there is an error, or an ambiguous question, you may seek clarification from myself or one of the TAs. You must use a whisper, or write your question out. Speaking out aloud is not allowed.
- Including this cover sheet and rules at the end, there are 14 pages to the exam, including one blank page for workspace. Please verify that you have all 14 pages.
- Please write your name and NetID in the spaces above, and also in the provided space at the top of every sheet.

Question	Points	Bonus Points	Score
1	6	0	
2	10	0	
3	10	0	
4	13	0	
5	11	0	
6	12	0	
7	18	0	
8	20	0	
9	0	10	
Total:	100	10	

Problem 1. (6 points)

Suppose that the following code is input one line at a time into OCaml:

```
let x = "X";;
let y = 3;;
let shift x z = x + y + z;;
let x = 2;;
let y = 17;;
let a = shift x;;
let b = shift y 4;;
let c = shift (3, 5);;
```

(a) (2 points) Tell what, if anything, is returned for `a`. If no value is returned, explain why not.

Solution: `a` is bound to a function of one integer argument that will return the result of adding 5 to it that argument.

(b) (2 points) Tell what, if anything, is returned for `b`. If no value is returned, explain why not.

Solution: 24

(c) (2 points) Tell what, if anything, is returned for `c`. If no value is returned, explain why not.

Solution: No value is returned because there is a type error. The first argument to `f` must be an `int`, but here it is applied to an `(int * int)`.

Problem 2. (10 points)

Write a function `interleave : 'a list -> 'a list -> 'a list` that takes two lists and returns a list. The first element of the new list should be the first element of the first list and the second element of the new list should be the first element of the second list; then, the third element of the new list will be the second element of the first list and the fourth element of the new list will be the second element of the second list, and so on. If one list is longer than the other, put the extra elements on the end of the new list. Also, if either list is empty, `interleave` returns the other list.

```
# let rec interleave xs ys = ...;;
val interleave : 'a list -> 'a list -> 'a list = <fun>
# interleave [1;2;5] [3;4];;
- : int list = [1; 3; 2; 4; 5]
```

Solution:

```
let rec interleave xs ys =
    match xs with [] -> ys
    | (x::more_xs) ->
       (match ys with [] -> xs
        | (y::more_ys) -> x :: y :: (interleave more_xs more_ys))
```

Problem 3. (10 points)

What is printed by the following code?

```
let f = fun x ->
        let a = (print_string "A"; 5)
        in fun y -> (print_string "B\n"; a + x + y)
in let h = f 1 in f (h 3) (h 2)
```

You do not have to compute the integer value returned; just the sequence of characters printed.

Solution:

```
AB
B
AB
```

Problem 4. (13 points)

Consider the following OCaml code:

```
let a = 3;;
let b = 2;;
let f =
  let b = 7
  in fun x -> a + b;;
let x = f a;;
```

Describe the final environment that results from the execution of the above code if execution is begun in an empty environment. Your answer should be written as a set of bindings of variables to values, with only those bindings visible at the end of the execution present. Your answer should be a precise mathematical answer, with a precise description of values involved in the environment. You may name your environments and closures, and use their names in describing other environments, but all applications of the update operator (+) should be expanded out, and not appear in your final answer.

Solution:

$$\{\texttt{a} \mapsto 3;\ \texttt{b} \mapsto 2;\ \texttt{f} \mapsto \langle \texttt{x} \to \texttt{a + b}, \{\texttt{a} \mapsto 3;\ \texttt{b} \mapsto 7\}\rangle;\ \texttt{x} \mapsto 10\}$$

Problem 5. (11 points)

Write a function `split: ('a -> bool) -> 'a list -> 'a list * 'a list`, that, when applied to a test function `f`, and a list `lst`, returns a pair of lists. The first list of the pair should contain every element `x` of `lst` for which `(f x)` is true; and the second list contains every element for which `(f x)` is false. The order of the elements in the returned lists should be the same as in the original list.

(a) (5 points) Write the `split` using only forward recursion, and no other form of recursion, directly or indirect. You may not use any library functions.

Solution:

```
let rec split test list =
    match list with [] -> ([],[])
    | (first::rest) ->
      let (true_list, false_list) = split test rest
      in if test first then (first :: true_list, false_list)
                       else (true_list, first:: false_list)
```

(b) (6 points) Write a base value `split_base:'a list * 'b list` and a step function `split_step:('a -> bool) -> 'a -> 'a list * 'a list -> 'a list * 'a list` such that `List.fold_right (split_step f) lst split_base` behaves the same as the `split f lst` as described in part a.

Solution:

```
let split_base = ([],[])
let split_step test first (true_list, false_list) =
    if test first then (first ::true_list, false_list)
                  else (true_list, first :: false_list)
```

Problem 6. (12 points)

Consider the following OCaml function:

```
let rec filter test list =
    match list with [] -> []
    | (first :: rest) ->
      let rest_result = filter test rest
      in if test first then (test :: rest_result) else rest_result
```

Write the function
`filterk : ('a -> (bool -> 'b) -> 'b) -> 'a list -> ('a list -> 'b) -> 'b`
that is the CPS transformation of the above code. You may assume you have the following CPS transformation of cons:

```
let consk x y k = k (x :: y);;
val consk : 'a -> 'a list -> ('a list -> 'b) -> 'b = <fun>
```

Be careful to take note of the type of the function `filterk`, and all its arguments.

Solution:

```
let consk x y k = k (x :: y)
let rec filterk testk list k =
    match list with [] -> k []
    | (first :: rest) ->
      filterk testk list
      (fun rest_result ->
        testk first
         (fun b -> if b then consk first rest_result k
                        else k rest_result))
```

Problem 7. (18 points)

(a) (8 points) Give an OCaml data type to represent trees built from three kinds of nodes: **Black**, **Red**, and **Blue**. Associated with each node is an integer weight. In addition, each **Red** node has one subtree associated with it, and each **Blue** node has two subtrees associated with it, a left subtree and a right subtree.

Your representation should be exact: every tree should have a unique representation using your data type, and every thing that could be represented by your type should be a tree as described here.

Solution:

```
type tree = Black of int
          | Red of (int * tree)
          | Blue of (tree * int * tree)
```

(b) (10 points) Write a function `blueWeight:tree -> int` that adds up the weights of all the `Blue` nodes in the tree.

Solution:

```
let ref blueWeight tree =
    match tree
    with Black n -> 0
       | Red (n,tr) -> blueWeight tr
       | Blue (left_tree, n, right_tree) ->
         (blueWeight left_tree) + n + (blueWeight right_tree)
```

Workspace

Problem 8. (20 points)
Give a type derivation for the following type judgment:

```
{ } |- let x = 7 in (fun x -> if x then 1 else 0) (x + 2 > 8) : int
```

You may use the attached sheet of typing rules. Label every use of a rule with the rule used. You may abbreviate, provided it must be totally clear which rule is meant by which abbreviation. You may find it useful to break your derivation into pieces. If you do, give names to your pieces, which you may then use in describing the whole. Your environments should be mathematical mappings here, and NOT implementations as you might find in a program.

Solution: Let *FunTree* =

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{x:bool}\} \vdash \texttt{x:bool}}\,\textsc{Var} \quad \dfrac{}{\{\texttt{x:bool}\} \vdash \texttt{1:int}}\,\textsc{Const} \quad \dfrac{}{\{\texttt{x:bool}\} \vdash \texttt{0:int}}\,\textsc{Const}}{\{\texttt{x:bool}\} \vdash \texttt{(if x then 1 else 0):int}}\,\textsc{If}}{\{\texttt{x:int}\} \vdash \texttt{(fun x -> if x then 1 else 0):bool -> int}}\,\textsc{Fun}$$

Then the main type derivation is:

$$\dfrac{\dfrac{}{\{\,\} \vdash \texttt{7:int}}\,\textsc{Const} \quad \dfrac{\mathit{FunTree} \quad \dfrac{\dfrac{\dfrac{}{\{\texttt{x:int}\} \vdash \texttt{x:int}}\,\textsc{Var} \quad \dfrac{}{\{\texttt{x:int}\} \vdash \texttt{2:int}}\,\textsc{Const}}{\{\texttt{x:int}\} \vdash \texttt{(x + 2):int}}\,\textsc{Arith} \quad \dfrac{}{\{\texttt{x:int}\} \vdash \texttt{8:int}}\,\textsc{Const}}{\{\texttt{x:int}\} \vdash \texttt{(x + 2 > 8):bool}}\,\textsc{Rel}}{\begin{array}{l}\{\texttt{x:int}\} \vdash \texttt{(fun x -> if x then 1 else 0)} \\ \texttt{(x + 2 > 8) : int}\end{array}}\,\textsc{App}}{\{\,\} \vdash \texttt{let x = 7 in (fun x -> if x then 1 else 0) (x + 2 > 8):int}}\,\textsc{Let}$$

Workspace

9. (10 points (bonus)) Give code implementing OCaml's
`fold_left : : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
from the `List` library, and give the CPS transformation of your code.

Solution:

```
let rec fold_left f acc list =
    match list with [] -> acc
    | (x :: xs) -> fold_left f (f acc x) xs

let rec fold_leftk fk acc list k =
    match list with [] -> k acc
    | (x :: xs) -> fk acc x (fun new_acc -> fold_leftk fk new_acc xs k)
```

A Monomoprhic Typing Rules

Constants:

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \textsc{Const}$$ where n is an integer constant

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textsc{Const} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \textsc{Const}$$

Variables:

$$\frac{}{\Gamma \vdash x : \tau}\ \textsc{Var}$$ where $\tau = \Gamma(x)$

Primitive Operators $\oplus \in \{+, -, *, \texttt{mod}, \ldots\}$:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{int}}\ \textsc{Arith}$$

Relations ($\sim \in \{<, >, =, \leq, \geq\}$):

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \sim e_2 : \texttt{bool}}\ \textsc{Rel}$$

Connectives:

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\ \&\&\ e_2 : \texttt{bool}}\ \textsc{Conn} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \mid\mid e_2 : \texttt{bool}}\ \textsc{Conn}$$

If_then_else rule:

$$\frac{\Gamma \vdash e_c : \texttt{bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_e : \tau}{\Gamma \vdash \texttt{if}\ e_c\ \texttt{then}\ e_t\ \texttt{else}\ e_e : \tau}\ \textsc{If}$$

Application rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

Function rule:

$$\frac{[x : \tau_1] + \Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau_1 \rightarrow \tau_2}\ \textsc{Fun}$$

Let rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad [x : \tau_1] + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2}\ \textsc{Let}$$

Let Rec rule:

$$\frac{[x : \tau_1] + \Gamma \vdash e_1 : \tau_1 \qquad [x : \tau_1] + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2}\ \textsc{Rec}$$

B Scratch Space