CS421 Spring 2014 Midterm 1

Name:	
NetID:	

- You have **75 minutes** to complete this exam.
- This is a **closed-book** exam. All other materials (e.g., calculators, telephones, books, card sheets), 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 and you will be reported as such.
- If you believe there is an error, or an ambiguous question, you may seek clarification from myself or one of the TAs. Raise your hand. 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 19 pages to the exam, including one blank page for workspace. Please verify that you have all 19 pages.
- Please write your name and NetID in the spaces above, and also in the provided space at the top of every sheet.
- We are going to check IDs after the exam starts; when you see us next to your seat, please show us your ID or put it on the desk.

Question	Points	Bonus Points	Score
1	8	0	
2	6	0	
3	13	0	
4	16	0	
5	10	0	
6	17	0	
7	13	0	
8	17	0	
9	0	10	
Total:	100	10	

Problem 1. (8 points)

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

```
let m = 20;;
let n = 30;;
let f m p = m + p - 50;;
let a = f n;;
let b = f (70,20);;
```

(a) (2 points) What is the type of `f`?

Solution:
`int -> int -> int`

(b) (3 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 subtracting 20 from that argument.

(c) (3 points) Tell what, if anything, is returned for `b`. 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. (6 points)

Write a function `insert : 'a -> 'a list -> 'a list` that takes in a element `n` and a list. It returns a list with the element inserted immediately before the first element that is greater than `n`. If there is no such position exists, insert `n` at the end of the list. Note, if the list is sorted in ascending order, the order is preserved by the insertion.

```
# let rec insert n l = ...
val insert : 'a -> 'a list -> 'a list = <fun>
# insert 2 [1;3;5];;
- : int list = [1; 2; 3; 5]
```

Solution:

```
let rec insert n l =
    match l with [] -> [n]
    | x::xs -> if x > n then n::l else x::insert n xs
```

Problem 3. (13 points)

Consider the following OCaml code:

```
let m = 1;;
let n = 2;;
let f x y = (m * x) + (n * y);;
let y = f 3;;
```

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:

$$\rho_1 = \{\ \texttt{m} \mapsto 1;\ \texttt{n} \mapsto 2\ \}$$

$$c_f = \langle \texttt{x} \to \texttt{y} \to \texttt{(m * x) + (n * y)},\ \rho_1 \rangle$$

$$\rho_2 = \{\ \texttt{m} \mapsto 1;\ \texttt{n} \mapsto 2;\ \texttt{f} \mapsto c_f;\ \texttt{x} \mapsto 3\ \}$$

$$c_y = \langle \texttt{y} \to \texttt{(m * x) + (n * y)},\ \rho_2 \rangle$$

$$\rho_3 = \{\ \texttt{m} \mapsto 1;\ \texttt{n} \mapsto 2;\ \texttt{f} \mapsto c_f;\ \texttt{y} \mapsto c_y\ \}$$

ρ_3 is the final environment.

Problem 4. (16 points)

(a) (7 points) Write a function `split_sum : int list -> (int -> bool) -> int * int` that takes a list of integers and returns a pair of integers. The first integer in the pair is the sum of all numbers in the input list `l` where the input function `f` returns true. The second is the sum of all remaining numbers that `f` return false. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec split_sum l f = ...;;
val split_sum : int list -> (int -> bool) -> int * int = <fun>
# split_sum [1;2;3] (fun x -> x>1);;
- : int * int = (5, 1)
```

Solution:

```
let rec split_sum l f =
    match l with [] -> (0,0)
    | x::xs -> (match split_sum xs f
                with (m,n) ->
                if f x then (m+x,n)
                else (m,n+x))
```

Problem 4 (cont.)

(b) (9 points) Write a value `split_sum_base : int * int` and function `split_sum_rec : ((int -> bool) -> int -> int * int -> int * int` such that `(fun l -> fun f -> List.fold_right (split_sum_rec f) l split_sum_base)` computes the same solution as `split_sum` defined in (a). There should be no use of recursion or library functions in the solution to this problem. The type of `List.fold_right` is `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`.

```
# let split_sum_base = ...;;
val split_sum_base : int * int = ...
# let split_sum_rec = ...;;
val split_sum_rec : (int -> bool) -> int -> int * int -> int * int = <fun>
# let split_sum l f = List.fold_right (split_sum_rec f) l split_sum_base;;
val split_sum : int list -> (int -> bool) -> int * int = <fun>
# split_sum [1;2;3] (fun x -> x>1);;
- : int * int = (5, 1)
```

Solution:

```
let split_sum_base = (0,0)

let split_sum_rec =
  fun f -> fun x -> fun (r1,r2) ->
    if f x then (r1+x,r2)
    else (r1,r2+x)
```

Problem 5. (10 points)

Consider the following OCaml function:

```
let apply_to_two g = g 2;;
val apply_to_two : (int -> 'a) -> 'a = <fun>
```

Write the function
`apply_to_two_k : (int -> 'a -> 'b) -> 'a -> 'b`
that is the CPS transformation of the above code. Be careful to take note of the type of the function `apply_to_two_k`, and all its arguments.

Solution:

```
let apply_to_two_k gk k = gk 2 k;;
```

Workspace

Problem 6. (17 points)

The Abstract Syntax Trees for MicroML expressions are given by the following OCaml (redacted) type:

```
type exp =
   | VarExp of string                    (* variables *)
   | ConstExp of const                   (* constants *)
. . .
   | AppExp of exp * exp                 (* exp1 exp2 *)
. . .
```

In addition to having abstract syntax trees for the expressions of MicroML, we need to have abstract syntax trees for the type of continuations and expressions in CPS.

```
type cps_cont =
. . .
 | ContCPS of string * exp_cps                    (* FUN x -> exp_cps *)

and exp_cps =
   VarCPS of cps_cont * string                    (* k x *)
 | ConstCPS of cps_cont * const                   (* k c *)
. . .
 | AppCPS of cps_cont * string * string           (* x y k *)
. . .
```

The augmentation of the constructors with a place for a continuation, and the replacment of general expression arguments by variable arguments are the changes necessary to guarantee that terms built in this type represent expressions in CPS.

When transforming a function into CPS, it is necessary to expand the arguments to the function to include one that is for passing the continuation to it. We represent this variable by an integer rather than a string. It really is a different type of variable because it is always internally generated and it is to supply a continuation and not an expression. When transforming an expression, we will take in and hand back an integer giving the next integer available to be used for a continuation variable.

Mathematically we represent CPS transformation by the functions $[[e]]_\kappa$, which calculates the CPS form of an expression e when passed the continuation κ. κ does not represent a programming language variable, but rather a complex expression describing the current continuation for e.

In MicroML, we will uniformly use left-to-right evaluation. Therefore, to evaluate an application, first evaluate the function, e_1, to a closure, then evaluate e_2 to a value to which that closure is applied. We create a new continuation that takes the result of e_1 and binds it to v_1, then evaluates e_2 and binds it to v_2. Finally, v_1 is applied to v_2 and,

since the CPS transformation makes all functions take a continuation, it is also applied to the current continuation κ. Implement this rule.

$$[[e_1\ e_2]]\kappa = [[e_1]]_{\texttt{fun}\ v_1\ \texttt{->}\ [[e_2]]_{\texttt{fun}\ v_2\ \texttt{->}\ v_1\ v_2\ \kappa}} \quad \text{Where} \begin{array}{l} v_1 \text{ is fresh for } e_2 \text{ and } \kappa \\ v_2 \text{ is fresh for } v_1 \text{ and } \kappa \end{array}$$

```
# string_of_exp_cps (fst (cps_exp (AppExp (VarExp "f", VarExp "x"))
                                  (ContVarCPS 0) 1));;
- : string = "(FUN a -> (FUN b -> a b _k0)) x) f"
```

By v being fresh for an expression e, we mean that v needs to be some variable that is NOT free in e. You may use the function `freshFor : string list -> string` that, when given a list of names, will generate a name that is not in the list. You may use `freeVarsInExp : exp -> string list` and `freeVarsInContCPS : cps_cont -> string list` and `freeVarsInExpCPS : exp_cps -> string list` for calculating the free variables in an expression, a continuation and a CPS-transformed expression respectively.

Write the OCaml code for the function application case
`cps_exp: exp -> cps_cont -> int -> exp_cps * int`

```
let rec cps_exp e k kx =
   match e with
```

Solution:

```
(*[[e1 e2]]k = [[e1]]_fun v1 -> [[e2]]_fun v2 -> v1 v2 k*)
   AppExp (e1,e2) ->
      (* make sure v1 does not appear in e2 or k *)
      let v1 = freshFor (freeVarsInContCPS k @ freeVarsInExp e2) in
      (* make sure v2 is not v1 and does not appear in k *)
      let v2 = freshFor (v1 :: freeVarsInContCPS k) in
      (* v1 v2 k *)
      let app_cps = AppCPS (k, v1, v2) in
      (* fun v2 -> v1 v2 k *)
      let e2cont = ContCPS (v2, app_cps) in
      (* [[e2]]_fun v2 -> v1 v2 k *)
      let (e2cps, ky) = cps_exp e2 e2cont kx in
      (* fun v1 -> [[e2]]_fun v2 -> v1 v2 k *)
      let e1cont = ContCPS (v1, e2cps) in
      (* [[e1]]_fun v1 -> [[e2]]_fun v2 -> v1 v2 k *)
      cps_exp e1 e1cont ky
```

Problem 7. (13 points)

(a) (6 points) Give an OCaml data type to represent non-empty lists. You may not use the existing type of lists in OCaml.

Your representation should be exact: every non-empty list should have a unique representation using your data type, and every thing that could be represented by your type should be a non-empty list.

Solution:

```
type 'a nonemptylist = First of 'a
          | More of ('a * 'a nonemptylist)
```

(b) (7 points) Write a function `prod : int nonemptylist -> int` that multiplies all the integers in an `int nonemptylist`.

Solution:

```
let rec prod nelist =
    match nelist with First n -> n
    | More (n, nel) -> n * prod nel
```

Workspace

Problem 8. (17 points)

Give a type derivation for the following type judgment:

```
{ } |- let x = true in (fun f -> (f x) * 7)  : (bool -> int) -> 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:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\texttt{f:bool -> int; x:bool}\} \vdash \texttt{f:bool -> int}}\textsc{Var} \quad \dfrac{}{\{\texttt{f:bool -> int; x:bool}\} \vdash \texttt{x:bool}}\textsc{Var}}{\{\texttt{f:bool -> int; x:bool}\} \vdash \texttt{f x:int}}\textsc{App} \quad \dfrac{}{\{\texttt{f:bool -> int; x:bool}\} \vdash \texttt{7:int}}\textsc{Const}}{\{\texttt{f:bool -> int; x:bool}\} \vdash \texttt{(f x) * 7:int}}\textsc{Arith}}{\{\texttt{x:bool}\} \vdash \texttt{(fun f -> (f x) * 7) : (bool -> int) -> int}}\textsc{Fun} \quad \dfrac{}{\{\,\} \vdash \texttt{true:bool}}\textsc{Const}}{\{\,\} \vdash \texttt{let x = true in (fun f -> (f x) * 7) : (bool -> int) -> int}}\textsc{Let}$$

Workspace

9. (10 points (bonus)) Consider the following OCaml code extended from Problem 3:

```
let m = 1;;
let n = 2;;
let f x y = (m * x) + (n * y);;
let y = f 3;;
y 2;;
```

Starting from the environment you gave in Problem 3, show step by step how the application of `y 2` would be evaluated. You may use the update operator here.

Solution: From Problem 3, we have

$$\begin{aligned}
\rho_1 &= \{\ \texttt{m} \mapsto 1;\ \texttt{n} \mapsto 2\ \} \\
c_f &= \langle \texttt{x} \to \texttt{y} \to \texttt{(m * x) + (n * y)},\ \rho_1 \rangle \\
\rho_2 &= \{\ \texttt{m} \mapsto 1;\ \texttt{n} \mapsto 2;\ \texttt{f} \mapsto c_f;\ \texttt{x} \mapsto 3\ \} \\
c_y &= \langle \texttt{y} \to \texttt{(m * x) + (n * y)},\ \rho_2 \rangle \\
\rho_3 &= \{\ \texttt{m} \mapsto 1;\ \texttt{n} \mapsto 2;\ \texttt{f} \mapsto c_f;\ \texttt{y} \mapsto c_y\ \}
\end{aligned}$$

Let

$$\begin{aligned}
\rho_4 &= \{\ \texttt{y} \mapsto 2\ \} + \rho_2 \\
&= \{\ \texttt{m} \mapsto 1;\ \texttt{n} \mapsto 2;\ \texttt{f} \mapsto c_f;\ \texttt{x} \mapsto 3;\ \texttt{y} \mapsto 2\ \}
\end{aligned}$$

Then we have

$$\begin{aligned}
\mathsf{Eval}(\texttt{y 2}, \rho_3) &= \mathsf{Eval}(\mathsf{App}(\langle \texttt{y} \to \texttt{(m * x) + (n * y)},\ \rho_2 \rangle, \texttt{2}), \rho_3) \\
&= \mathsf{Eval}(\texttt{(m * x) + (n * y)}, \{\ \texttt{y} \mapsto 2\ \} + \rho_2) \\
&= \mathsf{Eval}(\texttt{(m * x) + (n * y)}, \rho_4) \\
&= \mathsf{Eval}(\texttt{(1 * 3) + (2 * 2)}, \rho_4) \\
&= \mathsf{Eval}(7, \rho_4) \\
&= 7
\end{aligned}$$

Scratch Space

Workspace

A Monomorphic Typing Rules

Constants:

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

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

Variables:

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

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:

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

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:

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

Function rule:

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

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:

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