CS421 Summer 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 11 pages to the exam, including one blank page for workspace. Please verify that you have all 11 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	Score
1	12	
2	6	
3	5	
4	13	
5	21	
6	20	
7	23	
Total:	100	

Problem 1. (12 points)

For each of the following functions, mark **ALL** classifications that apply.

(a) (4 points)

```
let rec foo l = match l with
                | [] -> []
                | x::[] -> []
                | x::y::xs -> y::(foo xs)
```

[✓ **Forward Recursive**] ○ Tail Recursive ○ Primitive Recursive ○ Not Recursive

(b) (4 points)

```
let rec partial_sums l = match l with
                         | [] -> []
                         | x::xs -> let p = partial_sums xs
                                    in match p with
                                       | [] -> [x]
                                       | (s::ss) -> (x+.s) :: p
```

[✓ **Forward Recursive**] ○ Tail Recursive [✓ **Primitive Recursive**] ○ Not Recursive

(c) (4 points)

```
let rec count p l c = match l with
                      | [] -> c
                      | x::xs -> count p xs (if p x then c+1 else c)
```

○ Forward Recursive [✓ **Tail Recursive**] [✓ **Primitive Recursive**] ○ Not Recursive

Problem 2. (6 points)

In the following question, for each polymorphic type, list the free type variables of that type.

(a) (2 points) α `list` $\rightarrow$ `float`

(a) α

(b) (2 points) $\forall \alpha . \beta \rightarrow$ `int list` $\rightarrow \alpha$ `list` $\rightarrow \delta$

(b) $\{\beta, \delta\}$

(c) (2 points) $\forall \alpha\ \beta\ \gamma . \alpha * \beta \rightarrow$ `int` $* (\beta \rightarrow \alpha)$

(c) $\emptyset$

Problem 3. (5 points)

What is the output of the following OCaml program when executed?

```
let rec f = print_int 1; fun x -> (print_int 5; 2*x)
in print_int (f (f (print_int 3; 2)));;
```

3. **13558**

Problem 4. (13 points)

In the follow OCaml program give the environment that would be in use at the specified points.

```
let a = 4;;
let b = a*9;;
let f = 7;;
(* 1 *)
let f x = x+a in
(* 2 *)
          a * f 5;;
let g y = f;;
(* 3 *)
```

Solution:

$$\rho_1 = \{\mathtt{a} \mapsto 4, \mathtt{b} \mapsto 36, \mathtt{f} \mapsto 7\}$$
$$\rho_2 = \{\mathtt{f} \mapsto \langle \mathtt{x} \mapsto \mathtt{x+a}, \rho_1\rangle\} + \rho_1$$
$$\rho_3 = \{\mathtt{g} \mapsto \langle \mathtt{y} \mapsto \mathtt{f}, \rho_1\rangle\} + \rho_1$$

Problem 5. (21 points)

(a) (9 points) Write an OCaml datatype that can represent expressions composed of polynomials and infinite summations that use polynomials for their inner expressions. Your datatype should be polymorphic over the type of variables and assume that coefficients and exponents are integers. Your answer should be able to represent expressions like the following:

$$x + \sum_{i=0}^{\infty} 5i^6 \qquad a^2 + 4b + 4 \qquad 10x^6 \qquad \sum_{k=2}^{\infty}(2x + k)$$

Solution:

```
type 'a poly = Sum of ('a * int * 'a poly)
             | Plus of ('a poly * 'a poly)
             | Coeff of (int *'a * int)
```

(b) (9 points) Write an OCaml function that takes a polynomial described by your datatype and counts the number of variable occurrences in it.

Solution:

```
let rec count p = match p with
                  | Sum (_,_,p’) -> 1 + count p’
                  | Plus (p1,p2) -> count p1 + count p2
                  | Coeff _ -> 1
```

(c) (3 points) Write the type of your counting function.

Solution:

```
count : ’a poly -> int
```

Problem 6. (20 points)

Give a polymorphic type derivation for the following expression. You should label each rule with the rule that was applied in some clear fashion (e.g., next to the large horizontal line).

$$\{\} \vdash \mathtt{let\ rec\ f\ =\ fun\ x\ ->\ x\ in\ f\ f} : \mathtt{int} \to \mathtt{int}$$

Solution:

$$\dfrac{\dfrac{\dfrac{}{\{\mathtt{f} : \alpha \to \alpha, \mathtt{x} : \alpha\} \vdash \mathtt{x} : \alpha}\,\text{Var}}{\{\mathtt{f} : \alpha \to \alpha\} \vdash \mathtt{fun\ x\ ->\ x} : \alpha \to \alpha}\,\text{Fun} \qquad \dagger}{\{\} \vdash \mathtt{let\ rec\ f\ =\ fun\ x\ ->\ x\ in\ f\ f} : \mathtt{int} \to \mathtt{int}}\,\text{Rec}$$

where † is the following:

$$\dfrac{\dfrac{}{\{\mathtt{f} : \forall\alpha.\alpha \to \alpha\} \vdash \mathtt{f} : (\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})}\,\text{Var} \qquad \dfrac{}{\{\mathtt{f} : \forall\alpha.\alpha \to \alpha\} \vdash \mathtt{f} : \mathtt{int} \to \mathtt{int}}\,\text{Var}}{\{\mathtt{f} : \forall\alpha.\alpha \to \alpha\} \vdash \mathtt{f\ f} : \mathtt{int} \to \mathtt{int}}\,\text{App}$$

Problem 7. (23 points)

(a) (7 points) Write a function `replace_if : ('a -> bool) -> 'a -> 'a list -> 'a list` such that `replace_if p v lst` replaces every element in `lst` for which `p` returns true with `v`. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec replace_if p v lst = ... ;;
val replace_if : ('a -> bool) -> 'a -> 'a list -> 'a list = <fun>
# replace_if (fun x -> x > 3) 62 [1;2;3;4;5];;
- : int list = [1; 2; 3; 62; 62]
```

Solution:

```
let rec replace_if p v lst =
    match lst with [] -> []
        | x::xs -> if p x then v::(replace_if p v xs)
                          else x::(replace_if p v xs)
```

(b) (7 points) Write a value `replace_if_base` and function `replace_if_rec : ('a -> bool) -> 'a -> 'a -> 'a list -> 'a list` such that

```
(fun p -> fun v -> fun lst -> List.fold_right (replace_if_rec p v) lst
                              replace_if_base)
```

computes the same results as `replace_if`. There should be no use of recursion or library functions in defining `replace_if_rec`.

```
# let replace_if_base = ... ;;
val replace_if_base : ...
# let replace_if_rec p v x r =  ... ;;
val replace_if_rec : ('a -> bool) -> 'a -> 'a -> 'a list -> 'a list = <fun>
# let replace_if' p v lst = List.fold_right
                             (replace_if_rec p v) lst replace_if_base ;;
val replace_if' : ('a -> bool) -> 'a -> 'a list -> 'a list = <fun>
# replace_if' (fun x -> x > 3) 62 [1;2;3;4;5];;
- : int list = [1; 2; 3; 62; 62]
```

Solution:

```
let replace_if_base = []
let replace_if_rec p v x r = (if p x then v else x)::r
```

(c) (9 points) Write the function `replace_ifk :('a -> (bool -> 'b) -> 'b) -> 'a -> 'a list -> ('a list -> 'b) -> 'b` such that `replace_ifk` is the continuation passing style version of the code you gave for `replace_if`. You should have that

`replace_ifk (fun x -> fun k -> k (p x)) v lst (fun x -> x)`

computes the same results as `replace_if p v lst`. Order of evaluation must be kept when using operators and functions. Any procedure call in the function must also be in continuation passing style. You may assume that CPS versions of the standard functions are available (e.g., addk and consk). **Make sure that you are converting your solution into continuation passing style and not just creating a continuation passing style function that happens to produce the same result.**

Solution:

```
let rec replace_ifk p v lst k =
    match lst with [] -> k []
        | x::xs -> p x (fun b ->
                    if b then replace_ifk p v xs (fun r -> consk v r k)
                         else replace_ifk p v xs (fun r -> consk x r k))
```

A Polymoprhic Type Rules

$$\frac{}{\Gamma \vdash c : \phi(\tau)}\ \textsc{Const}$$ where $\phi(\tau)$ is an instantiation of the polymorphic type of c

$$\frac{}{\Gamma \vdash x : \phi(\tau)}\ \textsc{Var}$$ where $\phi(\tau)$ is an instantiation $\Gamma(x)$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{int}}\ \textsc{Arith}$$ where $\oplus \in \{+, -, *\}$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{bool}}\ \textsc{Rel}$$ where $\oplus \in \{<, >, =, \leq, \geq\}$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{bool}}\ \textsc{Bool}$$ where $\oplus \in \{||, \&\&\}$

$$\frac{[x : \tau_1] + \Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau_1 \to \tau_2}\ \textsc{Fun} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

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

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

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

B Scratch Space