CS421 Fall 2011 Midterm 1

Tuesday, October 12, 2011

Name:	
NetID:	

- You have **75 minutes** to complete this exam.
- This is a **closed-book** exam. You are allowed one 3inch by 5inch card of notes prepared by yourself. This card is **not to be shared**. All other materials, besides pens, pencils and erasers, are to be away.
- Do not share anything with other students. Do not talk to other students. Do not look at another student's 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 9 sheets, 17 pages to the exam, including one blank page for workspace. The exam is printed double sided. Please verify that you have all 17 pages.
- Please write your name and NetID in the spaces above, and also at the top of every sheet.

Problems	**Possible Points**	**Points Earned**
1	6	
2	8	
3	11	
4	11	
5	11	
6	10	
7	11	
8	12	
9	20	
PreTotal	100	
Extra Credit	10	
PostTotal	110	

CS 421 Midterm 1 **Name:**_________________________________

1. (6 pts total) Suppose that the following code is input one line at a time into OCaml:

```
let z = 32.0;;
let r = 1.8;;
let trans x z = (x *. r) +. z;;
let z = 4.0;;
let r = 5.0;;
let a = trans 2.0 1.0;;
let b = trans 1.0;;
let c = trans (1.0, 1.0);;
```

For each of **a**, **b**, and **c**, either give what result is returned, or give the reason why nothing is returned.

a. (2 pt) Tell what is returned, if anything, for **a,** or why not:

Solution: 4.6

b. (2 pt) Tell what is returned, if anything, for **b**, or why not:

Solution: a function from floats to floats, adding 1.8 to its input

c. (2 pt) Tell what is returned, if anything, for **c**, or why not:

Solution: Attempting to evaluate c causes a type error, because trans take as a argument an int, but it is being applied to an (int * int)

2. (8 pts total) For each of the following programs, indicate what is printed, and value is returned in each case:

a. (4pts) let x = (print_string "a"; 5)
in (fun y -> (print_string "b"; (y *x) + y)) (print_string "c"; 3);;

Solution: prints: acb returns: 18

b. (4 pts) (fun z -> (print_string "a"; ((z() * 5) + z()))) (fun () -> (print_string "b"; 3));;

Solution: prints: abb returns: 18

CS 421 Midterm 1 **Name:___________________________________**

3. (11 pts total) Consider the following OCaml code

```
let x = 5;;
let a = x + 4;;
let f y z = let h x = x + a in y + h z;;
let a = 20;;
let x = f a 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. The update operator (+) and abbreviations should not be used.

Solution:
{f -> <y, fun z -> let h x = x + a in y + h z, {x -> 5; a -> 9}>, a -> 20, x -> 32}

4. (11 pts) Write a function **partial_sums : float list -> float list** that takes a list of floats [x_0;…; x_n] and returns a list whose i^{th} element is the sum of the elements in positions i to n, $\sum_{j=i}^{n} x_j$, for each $i = 0, \ldots n$. A sample execution is as follows:

 # let rec partial_sums lst = …
val partial_sums : float list -> float list = <fun>
 # partial_sums [4.0; 2.2; 7.5];;
- : float list = [13.7; 9.7; 7.5]

You are allowed to start your code with **let rec**, but are not required to. You are allowed to use any auxiliary functions you write yourself here, but you are **not** allowed to use any library functions. You may use the append function **@.**

Solution:

```
let rec partial_sums lst =
    match lst with [] -> []
    | x::xs ->
     (match partial_sums xs with [] -> [x]
      | y::ys  -> (x +. y) :: (y :: ys))
```

CS 421 Midterm 1 **Name:___________________________________**

5. (11 pts total)
   a. (5 pts) Write a function **count_if : ('a -> bool) -> 'a list -> int** such that **count_if p lst** returns the number of elements in **lst** for which the given predicate **p** gives **true**. The function is required to use only tail recursion (no other form of recursion). You may use auxiliary functions, but they must also use no other form of recursion than tail recursion, You may **not** use any library functions. Executing your code should give the following behavior:

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

Solution:

```
let rec count_if p lst =
    let rec count_if_aux p lst count =
      (match lst with [] -> count
        | (x::xs) -> count_if_aux p xs (if p x then 1+ count else count))
   in count_if_aux p lst 0
```

   b. (6 pts) Rewrite **count_if** as described above, using

List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

but **no other** library functions and **no** explicit recursion.

Solution:

```
let count_if p lst =
List.fold_left (fun count -> fun x -> if p x then 1 + count else count) 0 lst
```

6. (10 pts) Consider the following code:

```
let rec map f l =
  match l with [] -> [] | (x :: xs) -> (f x) ::  map f xs ;;
```

   a. (2 pts) Write **consk,** the Continuation Passing Style version of **::**. Your function should have the following type:

      consk : ‘a -> ‘a list -> (‘a list -> ‘b) -> ‘b

      Solution: let consk x xs k = k(x :: xs)

   b. (8 pts) Write a function **mapk** that is a complete Continuation Passing Style transformation of **map**. Your function should have the following type:
      mapk : (‘a -> (‘b -> ‘c) -> ‘c) -> ‘a list -> (‘b list -> ‘c) -> ‘c

Solution:

```
let rec mapk fk l k =
    match  l with [] -> k []
     | x:: xs -> fk x (fun r1 -> mapk fk xs (fun r2 -> consk r1 r2 k))
```

Also acceptable:

```
let rec mapk fk l k =
    match l with [] -> k []
    | x :: xs -> mapk fk xs (fun r1 -> fk x (fun r2 -> consk r2 r1 k))
```

7. (11 pts). Create an OCaml recursive data type to describe propositional formulae made from true, false, propositional variables (named by strings), negation (not A), conjunctions (A and B), and disjunctions (A or B). Your data type should be able to model every propositional formulae described above, and nothing else.

Solution:

```
type prop = True | False | PropVar of string | Not of prop | And of prop * prop | Or of prop * prop
```

8. (12 pts total) Consider the following Ocaml recursive data types:

```
type const = Int of int | Bool of bool
type  exp = VarExp of string
         | ConExp of const
         | IfExp of exp * exp * exp
         | AppExp of exp * exp
         | FunExp of  string * exp
```

Write a function **num_of_consts : exp -> int** that counts the number of occurrences for the constructor **ConExp** in an **exp.** You may use recursion, auxiliary functions, and library functions from OCaml freely. A sample execution is as follows:

```
# let rec num_of_consts exp = …
val num_of_consts : exp -> int = <fun>
# num_of_consts(IfExp (ConExp (Bool true), FunExp ("x", ConExp (Int 5)), VarExp "y"));;
- : int = 2
```

Solution:

```
let rec num_of_consts exp =
    match exp
    with VarExp _ -> 0
    | ConExp c -> 1
    | IfExp (e1,e2,e3) -> (num_of_consts e1) + (num_of_consts e2) + (num_of_consts e3)
    | AppExp (e1,e2) -> (num_of_consts e1) + (num_of_consts e2)
    | FunExp (x, e) -> (num_of_consts e)
```

CS 421 Midterm 1 **Name:____________________________________**

Worksheet (If extra space is needed).

9. (20 pts total) Give a type derivation for the following type judgment:

{}|- let x = 5 >3 in ((if x then (fun x -> x + 2) else (fun y -> y)) 7) : int

You may use the attached sheet of typing rules. Label every use of a rule with the rule used. You may abbreviate, but you must define your abbreviations. 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:

```
                    Var ------------------  Con--------------------
                          {x:int} |- x:int       {x:int} |- 2:int
              ArithOp ---------------------------------------  Var --------------------------
                                      {x:int} |- x + 2 : int          {y:int, x:bool} |- y:int
                                 Fun ------------------------    Fun ---------------------------
                   Var  ------------       {x:bool}|-                       {x:bool}|-
                        {x:bool}|-          fun x -> x + 2 :                  fun y -> y :
                          x:bool             int -> int                        int -> int
                       If----------------------------------------------------------------
Con ---------  Con-------      {x:bool}|-(if x then (fun x -> x + 2)               Con--------------------
     {}|-5:int     {}|-3:int                  else (fun y -> y)):int -> int                {x:bool}|- 7 :int
Rel -----------------------  App-------------------------------------------------------------------------------
           {}|- 5>3:bool          {x:bool}|- ((if x then (fun x -> x + 2) else (fun y -> y)) 7) : int
LetRule-----------------------------------------------------------------------------------------------------
                  {}|- let x = 5 >3 in ((if x then (fun x -> x + 2) else (fun y -> y)) 7) : int
```

CS 421 Midterm 1 **Name:___________________________________**

Worksheet (If extra space is needed).

10. Extra Credit: (10 pts) Write the clause for the function

gather_ty_substitution : judgment -> (proof * substitution) option

for **AppExp** (corresponding to application of two expressions), that implements the following rule:

$$\frac{\Gamma \mid\!- e_1 : \tau_1 \to \tau \mid \sigma_1 \qquad \sigma_1(\Gamma) \mid\!- e_2 : \sigma_1(\tau_1) \mid \sigma_2}{\Gamma \mid\!- e_1\, e_2 : \tau \mid \sigma_2 \circ \sigma_1}$$

The following types are used:

type ‘a option = Some of ‘a | None
type constTy = {name : string; arity : int}
type typeVar = int
type monoTy = TyVar of typeVar | TyConst of (constTy * monoTy list)
type env = …
type exp = … | AppExp of exp * exp | …
type judgement = {gamma : env; exp : exp; monoTy : monoTy}
type proof = {antecedents : proof list; conclusion : judgment}
type substitution = (typeVar * monoTy) list

In addition, we have the following functions using these types:

val fresh : unit -> monoTy
for creating type variables with names not previously used
val mk_fun_ty : monoTy -> monoTy -> monoTy
for creating the function space type between two types
val subst_compose : substitution -> substitution-> substitution
for creating the composition of two substitions, and
val monoTy_lift_subst : substitution -> monoTy -> monoTy
val env_lift_subst : substitution -> env -> env
for applying a substitution to each of a **monoTy** and an **env**

You may assume your code begins

```
let rec gather_ty_substitution judgment =
   let {gamma = gamma, exp = exp,  monoTy = tau) = judgment
   match exp with …
```

Solution:

```
| AppExp (e1,e2) ->
  let tau1 = fresh() in
  match gather_ty_substitution {gamma = gamma, exp = e1, monoTy = mk_fun_ty tau1 tau}
  with None -> None
   | Some (e1proof, sigma1) ->
     (match gather_ty_substitution {gamma = env_lift_subst sigma1 gamma, exp = e2,
                                    monoTy = monoTy_lift_subst sigma1 tau1}
       with None -> None
        | Some (e2proof, sigma2) -> Some({anecedents = [e1proof; e2proof]; conclusion = judgment},
                                         subst_compose  sigma2 sigma1))
```

CS 421 Midterm 1 **Name:___________________________________**

Worksheet (If extra space is needed).

CS 421 Midterm 1 **Name:______________________________**

Rules for type derivations:

Constants:

$$\frac{}{\Gamma \vdash n : \text{int}}$$ (assuming n is an integer constant)

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

Variables:

$$\frac{}{\Gamma \vdash x : \sigma} \quad \text{if } \Gamma(x) = \sigma$$

Primitive operators ($\oplus \in \{ +, -, *, \text{mod}, \ldots\}$):

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \oplus e_2 : \text{int}}$$

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

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \sim e_2 : \text{bool}}$$

Connectives :

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \;\&\&\; e_2 : \text{bool}} \qquad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \,||\, e_2 : \text{bool}}$$

If_then_else rule:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

Application rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

Function rule:

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

Let rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad [x : \tau_1] + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

Let Rec rule:

$$\frac{[x: \tau_1] + \Gamma \vdash e_1 : \tau_1 \quad [x: \tau_1] + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}$$