CS 421 Summer 2010 Midterm I

June 28, 2010

The total time for this exam is 70 minutes.

Print your name and netid below. Also write your netid at the top of each subsequent page.

Name:

Netid:

- This is a **closed-notes** exam. You are allowed only the materials in the exam packet. All other materials, besides pens, pencils, and erasers, are to be put away.
- Do not share anything with other students. Do not talk to other students. Do not look at another student's exam. Also, be careful not to expose your exam to easy viewing by other students. Violation of any of these rules may constitute cheating.
- If you believe there is an error, or an ambiguous question, you must document your assumptions about what the question means. The proctors are not allowed to answer questions about the exam other than the meaning and usage of English words and phrases.
- Including this cover sheet, the reference sheet at the end, and scratch pages, there are 8 pages to the exam. Please verify that you have all 8 pages.

Question	Total points	Score	Grader
1	6		
2	10		
3	5		
4	23		
5	12		
6	14		
TOTAL	70		

1. **(6 points)** Fill the blanks with the type of these functions.

```
let rec sumpoly p1 p2 = match p1,p2 with
 [],[] -> []
 | [], (y::ys) -> ys
 | (x::xs), [] -> xs
 | (x::xs,y::ys) -> (x+.y)::sumpoly xs ys;;

val sumpoly : ______________ -> ______________ -> ______________ = <fun>
```

Solution:

```
 val sumpoly :  float list -> float list -> float list = <fun>
```

2. **(10 points)** Write each step of evaluating this function.

```
# let app fs x =
   let rec app_aux fl acc=
       match fl with [] -> acc
       | (f :: rem_fs) -> app_aux rem_fs
                                  (fun z -> acc (f z))
   in app_aux fs (fun y -> y) x;;

# app [(fun x -> x + 1); (fun d -> d+2)] 2;;
- : int = 5
```

If you want to use any shorthand to denote an expression, explicitly mention it.
Here are the first few steps:

```
app [(fun x -> x + 1); (fun d -> d+2)] 2;;
= app_aux [f;g] (fun y -> y) 2;;  [We assume f = (fun x -> x + 1), g = (fun d -> d+2)]
= ...
```

Solution:

```
app [(fun x -> x + 1); (fun d -> d+2)] 2;;
= app_aux [f;g] (fun y -> y) 2;;  [We assume f = (fun x -> x + 1), g = (fun d -> d+2)]
= app_aux [g] (fun y -> i (f y)) 2;; [Assume i = (fun y -> y)]
= app_aux [] (fun y -> h (g y)) 2;; [Assume h = (fun y -> i (f y))]
= (fun y -> h (g y)) 2;;
= h (g 2) = h 4 = (fun y -> i (f y)) 4 = i (f 4) = i 5 = (fun y -> y) 5 = 5
```

3. **(1+1+3 points)** Write the following OCaml function `converge` as described below:
`converge` is a function of type `('a -> 'a) -> 'a -> int -> 'a`.
`converge f n i` applies `f` to `n` repeatedly until a fixed point `x` where `f x = x` is reached. At each step it checks for the equality of `(f x)` to `x`: in the terminating case `x` should be returned, otherwise the new value is stored in `x` for the next iteration. This function will fail to terminate if no fixed point can be reached by this method. In order to terminate, the function will only recurse `i` times. If `f` does not converge in `i` number of recursions, then a `Timeout` exception will be raised. If a user mistakenly supplies a negative number for `i` then a `NegativeCount` exception will be raised.

   Note that the value of `x` changes as follows, in order

   $$\texttt{x=f(n)},\ \texttt{x=f(f(n))},\ \texttt{x=f(f(f(n)))},\ \ldots$$

   and if it can't find a fixed point after the following value is assigned

   $$\texttt{x=}\overbrace{\texttt{f(f(...f(}}^{\texttt{i}\text{ times}}\texttt{n)...))}$$

   it will give up by raising an exception.

   (a) Define an exception `Timeout`.

Solution:

```
# exception Timeout;;
```

   (b) Define an exception `NegativeCount`.

Solution:

```
# exception NegativeCount;;
```

   (c) Write the function itself. First have a look at these sample runs:

```
# converge (fun x-> (x +. 2.0 /. x) /.2.0) 1.0 10;;
- : float = 1.41421356237309492

#  converge (fun x-> (x +. 2.0 /. x) /.2.0) 1.0 1;;
Exception: Timeout.

#  converge (fun x-> (x +. 2.0 /. x) /.2.0) 1.0 (-5);;
Exception: NegativeCount.
```

   and now write the function:

```
let rec converge f n i = ________________
```

Solution:

```
let rec converge f n i =
 match i with
 |0 -> raise Timeout
 |t -> if (t<0) then raise NegativeCount
       else (let res = (f n) in if n=res then n else converge f res (t-1));;
```

4. **(6+1+8+8 points)**

   (a) Write a function `split_list` in tail recursion form to split an integer list into two parts. Each part will be a list, the first one containing integers greater than or equal to zero; and the second one containing a list of negative integers. The integers will be in the same order as they are in the original list. You can use the `@` operator to concatenate lists. Here is the function with some sample runs:

```
# let split_list lst = ...
val split_list : int list -> int list * int list = <fun>
# split_list [1;3;0;-2;-5;6;0;-7];;
- : int list * int list = ([1; 3; 0; 6; 0], [-2; -5; -7])
# split_list [];;
- : int list * int list = ([], [])
```

   Now write the function (Hint: You have to keep two accumulators):

```
let split_list lst =
   let rec aux lst (acc1,acc2) = ___________
```

Solution:

```
let split_list lst =
   let rec aux lst (acc1,acc2) =
       match lst with
         [] -> (acc1,acc2)
       | x::xs -> if (x>=0)
                  then (aux xs ((acc1@[x]),acc2))
                  else (aux xs (acc1,(acc2@[x])))
   in aux lst ([],[]);;
```

   (b) If you want to rewrite the previous part function with higher order functions, which one will you use: `List.fold_left` or `List.fold_right` ?

Solution:

```
List.fold_left
```

(c) Rewrite the function with a higher order funcion: `List.fold_left` or `List.fold_right` whichever is appropriate.
(Hint: Remember you had two accumulators.)

Solution:

```
let split_list lst =  List.fold_left
                      (fun (acc1, acc2) -> fun x ->
                        if (x>=0) then (acc1@[x],acc2) else (acc1,acc2@[x]))
                      ([],[])
                      lst;;
```

(d) Suppose you have a report function that takes a tuple of lists and prints them. That is, the result of the `split_list` function written in the first part of this problem could be passed to it.

You might skip this paragraph; this paragraph specifies the report function.

```
# let rec print_list lst = match lst with
 [] -> print_string " "
 | x::xs -> print_int x; print_string " " ; print_list xs;;
 val print_list : int list -> unit = <fun>
# let report (lst1,lst2) = print_list lst1;print_list lst2;;
val report : int list * int list -> unit = <fun>
```

Write the function described in the first part of this problem in continuation passing style.

```
# let split_listk lst k =
   let rec aux lst (acc1,acc2) k = _____________
```

Solution:

```
let split_listk lst k =
   let rec aux lst (acc1,acc2) k=
       match lst with
         [] -> (k (acc1,acc2))
       | x::xs -> if (x>=0)
                  then (aux xs ((acc1@[x]),acc2) k)
                  else (aux xs (acc1,(acc2@[x])) k)
   in aux lst ([],[]) k;;
```

5. **(4+8 points)** Suppose you have a binary tree defined with the following data type.

```
# type 'a t = Empty| Node of 'a t * 'a * 'a t;;
type 'a t = Empty | Node of 'a t * 'a * 'a t
```

Here is a sample tree.

```
# let tree =
  Node(Node(Node(Empty,4,Empty),5,Node(Empty,6,Empty)),10,Node(Empty,12,Empty));;
val tree : int t =
 Node (Node (Node (Empty, 4, Empty), 5, Node (Empty, 6, Empty)), 10,
  Node (Empty, 12, Empty))
```

(a) Draw the tree.

Solution:

```
         10
      /      \
    5         12
 /     \
 4      6
```

(b) Write a function for *inorder traversal* of the tree. For a tree starting at its root, an inorder traversal would traverse the left subtree first, then the root node and then the right subtree, printing the values of the nodes on the console as visited. Note that visiting left and right subtrees themselves should be performed by an *inorder traversal* recursively.

Here is the function signature:

```
# let rec inorder tree = ...
   val inorder : int t -> unit = <fun>
```

Here is a sample run:

```
# inorder tree;;
 4 5 6 10 12 - : unit = ()
```

Now write the function:

```
# let rec inorder tree = ___________
```

Solution:

```
let rec inorder tree =
 match tree with
 |Empty -> print_string " "
 |Node (left, a, right) -> (inorder left); print_int a; (inorder right);;
   val inorder : int t -> unit = <fun>
```

6. **(14 points)** Assume we have this environment:

$$\Gamma = \texttt{[x:int, f:int -> float -> int]}.$$

Derive the type of the following statement, that is prove that:

$$\Gamma \vdash \texttt{(f x 2.1, x)} \;:\; \texttt{int * int}$$

Each time write the name of the rule that you want to apply. A list of all rules is at the end of this problem.

Solution:

(1): $\Gamma \vdash \texttt{2.1} \;:\; \texttt{float}$ (FLOAT)
(2): $\Gamma(\texttt{x}) = \texttt{int} \Rightarrow \Gamma \vdash \texttt{x} \;:\; \texttt{int}$ (VAR)
(3): $(2)\&\Gamma(\texttt{f}) = \texttt{int->float->int} \Rightarrow \Gamma \vdash \texttt{f x} \;:\; \texttt{float -> int}$ (APP)
(4): $(3)\&(1) \Rightarrow \Gamma \vdash \texttt{f x 2.1} \;:\; \texttt{int}$ (APP)
(5): $(2)\&(4) \Rightarrow \Gamma \vdash \texttt{(f x 2.1, x)} \;:\; \texttt{int * int}$ (PAIR)

(UNIT) $$\frac{}{\Gamma \vdash () \;:\; \mathsf{unit}} \qquad \frac{}{\Gamma \vdash [\,] \;:\; \tau\ \mathsf{list}}$$ (NIL)

(BOOL) $$\frac{}{\Gamma \vdash b \;:\; \mathsf{bool}} \quad \text{where } b = \mathsf{true} \text{ or } \mathsf{false}$$

(INT) $$\frac{}{\Gamma \vdash n \;:\; \mathsf{int}} \quad \text{where } n \text{ is an integer}$$

(FLOAT) $$\frac{}{\Gamma \vdash n \;:\; \mathsf{float}} \quad \text{where } n \text{ is a floating point number}$$

(VAR) $$\frac{}{\Gamma \vdash x \;:\; \tau} \quad \text{where } \Gamma(x) = \tau$$

(PAIR) $$\frac{\Gamma \vdash e_1 \;:\; \tau_1 \qquad \Gamma \vdash e_2 \;:\; \tau_2}{\Gamma \vdash (e_1,\, e_2) \;:\; (\tau_1, \tau_2)}$$

(LIST) $$\frac{\Gamma \vdash e_h \;:\; \tau \qquad \Gamma \vdash e_t \;:\; \tau\ \mathsf{list}}{\Gamma \vdash e_h{::}e_t \;:\; \tau\ \mathsf{list}}$$

(FUN) $$\frac{\Gamma \cup [x \;:\; \tau_x] \vdash e \;:\; \tau_e}{\Gamma \vdash \mathsf{fun}\ x \mathrel{\text{->}} e \;:\; \tau_x \to \tau_e}$$

(APP) $$\frac{\Gamma \vdash e_f \;:\; \tau_a \to \tau_r \qquad \Gamma \vdash e_a \;:\; \tau_a}{\Gamma \vdash e_f\ e_a \;:\; \tau_r}$$

(LET) $$\frac{\Gamma \vdash e_x \;:\; \tau_x \qquad \Gamma \cup [x \;:\; \tau_x] \vdash e \;:\; \tau}{\Gamma \vdash \mathsf{let}\ x = e_x\ \mathsf{in}\ e \;:\; \tau}$$

(LETREC) $$\frac{\Gamma \cup [x \;:\; \tau_x] \vdash e_x \;:\; \tau_x \qquad \Gamma \cup [x \;:\; \tau_x] \vdash e \;:\; \tau}{\Gamma \vdash \mathsf{let\ rec}\ x = e_x\ \mathsf{in}\ e \;:\; \tau}$$

scratch paper

scratch paper