Sample Questions for Midterm 1 (CS 421 Fall 2008)

On the actual midterm, you will have plenty of space to put your answers.
Some of these questions may be reused for the exam.

1. Given the following OCAML code:

   ```
   let x = 3;;
   let f y = x + y;;
   let x = 5;;
   let z = f 2;;
   let x = "hi";;
   ```

   What value will **z** have? Will the last declaration (**let x = "hi";**) cause a type error?
   What is the value of **x** after this code has been executed?

Solution:
   z is bound to 5
   let x = "hi" will not cause a type error
   x is bound to "hi"

2. What environment is in effect after each declaration in the code in Problem 1?
Solution:

   ```
   let x = 3;;
   {x → 3}
   let f y = x + y;;
   {f → <y → x+y, {x → 3}>,  x → 3}
   let x = 5;;
   {x → 5} + {f → <y → x+y, {x → 3}>,  x → 3} =
    {x → 5, f → <y → x+y, {x → 3}>}
   let z = f 2;;
   {z → 5, x → 5, f → <y → x+y, {x → 3}>}
   let x = "hi";;
   {x → "hi"} + {z → 5, x → 5, f → <y → x+y, {x → 3}>} =
    {x → "hi", z → 5,  f → <y → x+y, {x → 3}>}
   ```

3. Consider the following two OCaml functions, **loop1** and **loop2**:

   ```
   let rec loop1 () = loop1(); ()
   let rec loop2 () = loop2();;
   val loop1 : unit -> unit = <fun>
   val loop2 : unit -> 'a = <fun>
   ```

   Suppose you were to run **loop1();;** and **loop2();;** in OCaml, (pressing CTRL + C after at
   least a minute to terminate infinite loops when necessary).
   a. For each program, what behavior would you expect to see?
   b. What is the difference between loop1 and loop2?
   c. For each program state if it is:
      i. recursive,
      ii. forward recursive,

iii. tail-recursive.

Solution:

a. The first program generates a stack overflow, while the second program runs indefinitely.

b. Because **loop1** is not tail-recursive, each new recursive call must push a new activation record onto the stack, hence the stack overflow, but since **loop2** is tail-recursive, each new activation record may overwrite the previous call, and thus the stack does not grow. It should also be observed that **loop2** has a more general type (**:unit -> 'a**) than that of **loop1:unit -> unit**, and hence may be used in places where other return types beside unit are required. (Of course, it had better never actually be applied.)

c. Both programs are recursive, and in fact forward recursive, but **loop2** is tail-recursive while **loop1** is not

4. Write an OCAML function **pair_up** that takes first a function, then an input list and returns a list of pairs of an element from input list (the second argument), paired with the result of applying the first argument to that element. What is the OCAML type of **pair_up**? What is the result of the following expressions:

a. **pair_up (fun x -> x + 3) [6;4;1];;**
b. **pair_up ((fun x -> "Hi, "^x), ["John"; "Mary";"Dana"]);;**
c. **pair_up (fun x -> x *. 2.0);;**

Solution:

let rec pair_up f l =
 (match l with [] -> []
  | x :: xs -> (x, f x)::pair_up f xs)
alternately let pair_up f = map (fun x -> (x, f x))

pair_up : ('a -> 'b) -> 'a list -> ('a * 'b) list
a. [(6, 9); (4,7); (1,4)];;
b. type error
c. A function of type float list -> (float * float) list that returns a list of pairs of an element from the input list paired with twice itself.

5. Write an Ocaml function **palindrome :string list -> unit** that first prints the strings in the list from left to right, followed by printing them right to left, recursing over the list only once. (Potential extra credit problem: Do this using each of **List.fold_right** and **List.fold_left** but no explicit use of **let rec**.)

Solution:

let rec palindrome l =
    match l with [] -> ()
      | s::ss -> (print_string s; palindrome ss; print_string s);;

let rec palindrome l =
   List.fold_right
   (fun s -> fun print_middle -> (fun () -> (print_string s; print_middle (); print_string s)))

```
    l
    (fun () -> ())
    ();;

let palindrome l =
    List.fold_left
    (fun print_middle -> fun s ->  (print_string s; fun () -> (print_middle (print_string s))))
    (fun () -> ())
    l
    ();;
```

6. Using **fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b**, but without using explicit recursion, write a function **concat** : ' a list list -> 'a list that appends all the lists in the input list of lists, preserving the order of elements. You may use the append function **@.**
Solution: let concat lst = List.fold_right (@) lst [];;

7. Write an Ocaml function **list_print : string list -> unit** that prints all the strings in a list from left to right:
    a. using tail recursion, but no higher order functions,
    b. using **fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a** but no explicit recursion.
Solution:
    a. let rec list_print lst = match lst with [ ] -> ( ) | s::ss -> (print_string s; list_print ss);;
    b. let list_print lst = List.fold_left (fun ( ) -> fun s -> print_string s) () lst;;

8. Put the following function in full continuation passing style:
    let rec sum_odd n = if n <= 0 then 0 else ((2 * n) – 1) + sum_odd (n – 1);;
    Primitive operations (+, - , *, <=) do not have to be converted to CPS, but all procedure calls must be.
Solution:
```
    let rec  sum_odd_k n k =
            if n <= 0 then k 0
            else sum_odd_k (n – 1) (fun r -> k (((2 * n) – 1) + r));;
```

9. Write the definition of an OCAML variant type **reg_exp** to express abstract syntax trees for regular expressions over a base character set of booleans. Thus, a boolean is a **reg_exp,** epsilon is a **reg_exp**, the concatenation of two **reg_exp**'s is a **reg_exp,** the "choice" of two **reg_exp**'s is a **reg_exp**, and the Kleene star of a **reg_exp** is a **reg_exp**.
Solution:
```
type reg_exp =
   Char of bool
 | Epsilon
 | Concat of (reg_exp * reg_exp)
 | Choice of (reg_exp * reg_exp)
 | Kleene_star of reg_exp
 | Paren of reg_exp     (* I would accept it with this case missing *)
```

10. Given the following OCAML datatype:

type int_seq = Null | Snoc of (int_seq * int)

write a tail-recursive function in OCAML **all_pos : int_seq -> bool** that returns **true** if every integer in the input **int_seq** to which **all_pos** is applied is strictly greater than 0 and **false** otherwise. Thus **all_pos (Snoc(Snoc(Snoc(Null,3),5),7))** should returns **true,** but) **all_pos (Snoc(Null,~1))** and **all_pos (Snoc(Snoc(Null, 3),0))** should both return **false.**

Solution:

```
let rec all_pos s =
    (match s with Null -> true
       | Snoc(seq, x) -> if x <= 0 then false else all_pos seq);;
```

11. What is **type checking**? What do the terms **static** and **dynamic** mean when referring to type checking? Given an example of a property that can be type checked statically, and an example of a property that can only be checked dynamically.

Solution: Type checking means checking that for every application **proc(a₁,…,aₙ),** the number and types for the input arguments is the same as the number and type arguments specified..
Static type checking means type checking done at compile time, before object language code is produced. **Dynamic type checking** refers to perform type checking as a part of the execution of procedures at run time. That the argument to a function is a 32-bit integer can be checked **statically**. Divide by zero errors must be checked **dynamically.**