Solutions for Sample Questions for Midterm 1 (CS 421 Fall 2019)

On the actual midterm, you will have plenty of space to put your answers. The actual midterm will have fewer main questions, but one extra credit question. In addition to questions of the kind asked below, you should expect to see questions from your MPs and WA1 on the exams. Some of these questions may be reused for the exam.

1. Given the following OCAML code:
   ```ocaml
   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?
   What is the step-by-step formal evaluation of f 2 in the fourth declaration, starting from the environment after the third declaration?

   **Solution:**
   ```ocaml
   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;;
   Eval(f 2, {x → 5, f → <y → x+y, {x → 3}>}) =
   App(Eval(f, {x → 5, f → <y → x+y, {x → 3}>}),
       Eval(2, {x → 5, f → <y → x+y, {x → 3}>})) =
   App(Eval(f, {x → 5, f → <y → x+y, {x → 3}>}), 2) =
   App(<y → x+y, {x → 3}>, 2) =
   Eval(x+y, {y→2} + {x → 3}) = Eval(x+y, {y→2, x → 3}) =
   Eval(x, {y→2, x → 3}) + Eval(y, {y→2, x → 3}) =
   Eval(x, {y→2, x → 3}) + 2 = 3 + 2 = 5
   ```
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. What is the effect of each of the following pieces of code?
   a. (fun x -&gt; (print_string "a"; x + 2)) (print_string "b"; 4);;
      It prints ba and returns 6
   b. let f = (print_string "a"; fun x -&gt; x + 2) in f (print_string "b"; 4);;
      It prints ab and returns 6
   c. let f = fun g -&gt; (print_string "a"; g 2) in f (fun x -&gt; print_string "b"; 4 + x);;
      It prints ab and returns 6

4. Consider the following two OCaml functions, loop1 and loop2:
   let rec loop1 () = loop1(); ();
   let rec loop2 () = loop2();;
   val loop1 : 'a -&gt; unit = <fun>
   val loop2 : unit -&gt; '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 -&gt; 'a) than that of loop1:unit -&gt; 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

5. 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 each of the following expressions:
   a. pair_up (fun x -&gt; x + 3) [6;4;1];;
b. `pair_up ((fun x -> "Hi, " ^ x), ["John"; "Mary"; "Dana"]);

c. `pair_up (fun x -> x * 2.0);

Solution:

```ocaml
let rec pair_up f l =
  (match l with [] -> []
   | x :: xs -> (x, f x)::pair_up f xs)

alternately let pair_up f = List.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.
```

6. 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:

```ocaml
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))))
  palindrome
  (fun () -> ());

7. Using `List.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 [];

8. 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 \texttt{List.fold_left : ('a -> 'b -> 'a) -> 'a list -> 'a} but no explicit recursion.

\textbf{Solution:}

\begin{itemize}
  \item a. let rec list_print lst = match lst with [ ] -> ( ) | s::ss -> (print_string s; list_print ss);
  \item b. let list_print lst = List.fold_left (fun ( ) -> fun s -> print_string s) () lst;;
\end{itemize}

9. Put the following function in full continuation passing style:
\begin{verbatim}
let rec sum_odd n = if n <= 0 then 0 else ((2 * n) – 1) + sum_odd (n – 1);;
\end{verbatim}
Use \texttt{addk, subk, mulk, leqk,} for the CPS forms of the primitive operations \((+, -, *, <=)\). All other procedure calls and constructs must be put in CPS.

\textbf{Solution:}

\begin{verbatim}
let add_k a b k = k(a + b)
let minus_k a b k = k(a - b)
let times_k a b k = k(a * b)
let leq_k a b k = k(a <= b)

let rec sum_odd_k n k =
  leq_k n 0 (fun b -> if b then k 0
  else minus_k n 1
  (fun d -> sum_odd_k d
  (fun r -> times_k 2 n
  (fun t -> minus_k t 1
  (fun m -> add_k m r k ))))))
\end{verbatim}