1. (6 pts) Programming Language Features
   a. (3 pts) Describe type inference
      Type inference is the ability to *automatically determine the type of a value* in a programming language by its usage (without requiring type declarations).
   
   b. (3 pts) Describe why closures are needed in OCaml
      Closures are needed to *preserve the environment* (i.e., bindings of variables) for some function f, if f returns a function g that can access the environment for f.

2. (16 pts) OCaml Types and Type Inference
   a. (2 pts each) Give the type of the following OCaml expressions
      i. [ (“1”, 2); (“3”, 4) ]  
         Type = ( string * int ) list
      ii. let f a = [a; a+1]  
          Type = int -> int list

   b. (3 pts each) Write an OCaml expression with the following type
      i. int * int list  
         Code = ( 2 , [2] )
      ii. int list -> (int -> int)  
          Code = let f a b = match a with (h::t) -> h+b

   c. (3 pts each) Give the value of the following OCaml expressions. If an error exists, describe the error.
      i. [1;2];::[3]
         Error = 3 has type int but is used with type int list
         OR [3] has type int list but is used with type int list list
         OR can only add int list [1;2] to int list list
         OR trying to add int list [1;2] to int list [3]
      ii. let x y = y 3 in x (fun z -> z – 1)
         Value = 2
3. (18 pts) OCaml Programming

Solve the following OCaml programming problems. You are allowed to use List.rev (reverses a list) and the following (curried) map and fold functions, but no other OCaml library functions. Your solution must run in O(n) time for input lists of length n.

a. (9 pts) Write a function makeLists which when applied to a list lst, creates a new list for every element of lst, returning the results in a single list. You may use map or fold if you wish, but it is not required.

Example: makeLists [1;2;4] = [[1];[2];[4]]

Some possible answers:

let rec makeList x = match x with
  [] -> []
| (h::t) -> [h]::(makeList t) ;;

let makeList x = map ( fun y -> [y] ) x ;;

let makeList x = map ( fun y -> y::[ ] ) x ;;

let makeList x = List.rev (fold ( fun a y -> [y]::a ) [ ] x );;

b. (9 pts) Using either map or fold and an anonymous function, write a function over20 which when applied to a list of ints lst, returns a list of all elements of lst that are 21 or over (preserving their relative order in lst).

Example: over20 [33;18;21;19] = [33;21]

let over20 x = List.rev (fold (fun a y -> if (y > 20) then y::a else a) [ ] x );;

Partial credit:

let rec over20 x = match x with [] -> []
  | (h::t) -> if (h > 20) then (h::(over20 t)) else (over20 t) ;;