CIS 120 Midterm I   October 2, 2015

Name (printed): ____________________________

Username (login id): ____________________________

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: ______________________________________   Date: _________________________

1	/12
2	/28
3	/16
4	/20
5	/24
Total	/100

- Do not begin the exam until you are told to do so.
- You have 50 minutes to complete the exam.
- There are 100 total points.
- Make sure your name and username (a.k.a. PennKey, e.g. `stevez`) is on the top of this page.
- Be sure to allow enough time for all the problems—skim the entire exam first to get a sense of what there is to do.

1. **Reasoning about Program Behavior** (12 points)

Multiple choice: For each of the following (well-typed) programs, check the box for the value computed for `ans`, or mark "infinite loop" if the program loops.

a.
```
let x : int = 3
let f (y:int) : int =
  let x = y + y in x
let ans : int = f x
```

ans = ☐ 3 ☐ 6

☐ 12 ☐ infinite loop

b.
```
let rec f (l:int list) : (int * int) list =
  begin match l with
    | []   -> [(0,0)]
    | x::xs -> (x,x)::(f l)
  end
let ans : int list = f [1;2]
```

ans = ☐ `[(0,0)]` ☐ `[(1,1); (2,2); (0,0)]`

☐ `[(0,0); (1,1); (2,2)]` ☐ infinite loop

c.
```
let rec f (l: (int -> int) list) : int -> int =
  begin match l with
    | [] -> fun x -> x
    | g::gs -> fun x -> g (f gs x)
  end
let ans : int = f [(fun x -> x + 1); (fun x -> x * 2)] 3
```

ans = ☐ 6 ☐ 8

☐ 7 ☐ infinite loop

2. **Program Design** (28 points)

In this problem, we will use the design process to implement an abstract type of *cycles*, which act like a kind of infinitely long lists. Intuitively, a cycle is some finite sequence of elements that is repeated forever. We can create a cycle from a (non-empty) list using the `cycle_of_list` operation:

```
let cyc123 : int cycle = cycle_of_list [1;2;3]
```

Here, we intend for `cyc123` to represent the infinite repeating sequence 1 2 3 1 2 3 1 2 3 ....

We can get the first element and the rest of a cycle using the `hd_and_rest` operation, for example:

```
let (hd, rest) : int * int cycle = hd_and_rest cyc123
```

After this declaration, `hd = 1` (the first element of `cyc123`) and `rest` would represent the remaining infinite cycle 2 3 1 2 3 1 2 3 1 2 3 1 .... Note that this remainder is still a cycle generated from the list `[2;3;1]`.

Finally, we can test two cycles for equality using `equals`. Note that two cycles can be equal even if they are created from different lists. For example, the following expression evaluates to **`true`**:

```
equals (cycle_of_list [1;2]) (cycle_of_list [1;2;1;2])
```

whereas the one below evaluates to **`false`** (because the head elements differ):

```
equals (cycle_of_list [1;2] [2;1])
```

One snag is that there is no good way to create a cycle from an empty list. We therefore expect `cycle_of_list` to be undefined in that case. For the purposes of this problem we will simply have `cycle_of_list` *fail* if it is called on an empty list.

(0 points) Step 1 is *understanding the problem*. You don't have to do anything for this part—your answers below will demonstrate whether or not you succeed in Step 1.

(6 points) Step 2 is *formalizing the interface*. Complete the following interface definition, by filling in appropriate types for the missing blanks:

```
module type CYCLE = sig
  type 'a cycle

  val cycle_of_list : ____________________________ -> 'a cycle

  val hd_and_rest   : 'a cycle -> ______________________________________

  val equals        : ________________________________________________
end
```

(10 points) Step 3 is *writing test cases*. Given the interface, we can now write some test cases that will help our understanding of the problem and aid in debugging. The problem description above implicitly describes several such tests, which are partially specified below. Complete the code so that it matches the problem description. We have done the first one for you (be sure you understand it!). For test `(c)`, you need to complete the test and the name; it should not be redundant. For good measure, we have added an additional test `(d)`, not described above—you should be able to complete it too.

```
let cyc123 : int cycle = cycle_of_list [1;2;3]

let test () : bool =
  cycle_of_list [] = cycle_of_list []
;; run_failing_test "no empty cycle" test

(* (a) *)
let test () : bool =
  let (hd, _) = hd_and_rest cyc123 in


  ______________________________

;; run_test "correct hd for cyc123" test


(* (b) *)
let test () : bool =
  let (_, rest) = hd_and_rest cyc123 in


  ____________________________________________________________

;; run_test "correct rest for cyc123" test


(* (c) *)
let test () : bool =


  ____________________________________________________________


;; run_test "____________________________________________________" test


(* (d) *)
let test () : bool =
  let cyc : bool cycle = cycle_of_list ______________________________ in
  let (hd, rest) = hd_and_rest cyc in
  hd && equals cyc rest

;; run_test "cyc equals rest" test
```

(12 points) Step 4 is *implementing the program*. We can implement the `CYCLE` interface in a module, using an ordinary list as the concrete representation. For example, 1 2 3 1 2 3 1 2 3 ... can be represented by either the list `[1;2;3]` or the list `[1;2;3;1;2;3]`. There is a simple invariant, justified by the lack of an "empty" cycle: the list is not `[]`.

Complete the implementation below so that all of the tests pass, matching the behavior described in the problem statement. Note that we have marked some of the type annotations with `??` so as not to give away the answers to Step 2.

- You will need to use `failwith` in *two* places: once to mark a situation that is impossible given that the invariant holds, and once to establish the invariant. Call `failwith` on the strings `"IMPOSSIBLE"` and `"ESTABLISHING INVARIANT"` to mark them accordingly.
- You may use the operation `l1 @ l2`, which appends the two lists `l1` and `l2`.
- Note that the `helper` function in `equals` can mention `c1` and `c2`, if needed.

```
module Cycle : CYCLE = struct
  (* INVARIANT: the list is not [] *)
  type 'a cycle = 'a list

  let cycle_of_list (l:??) : 'a cycle =
    begin match l with

     | [] -> ______________________________________________

     | x::tl -> ____________________________________________
    end

  let hd_and_rest (l : 'a cycle) : ??
    match l with
      | [] -> ____________________________________________

      | x::tl -> _________________________________________
    end

  let equals (c1:??) (c2:??) : ?? =
    let rec helper (l1:'a list) (l2:'a list) : bool =
      begin match (l1, l2) with

        | ([], []) -> ___________________________________

        | (_ , []) -> ___________________________________

        | ([], _ ) -> ___________________________________

        | (x::xs, y::ys) -> _____________________________
      end
    in
    helper c1 c2
end
```

3. **Types** (16 points)

For each OCaml value below, fill in the blank with the appropriate type annotation or write "ill typed" if there is a type error on that line. Your answer should be the most specific type possible, i.e. `int list` instead of `'a list`. We have done the first one for you.

Some of the definitions refer to the `MyMap` module, which satisfies the following interface:

```
module type MAP = sig
  type ('k,'v) map

  val empty    : ('k,'v) map
  val add      : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val remove   : 'k        -> ('k,'v) map -> ('k,'v) map
  val mem      : 'k -> ('k,'v) map -> bool
  val get      : 'k -> ('k,'v) map -> 'v option
  val entries  : ('k,'v) map -> ('k * 'v) list
  val equals   : ('k,'v) map -> ('k,'v) map -> bool
end
module MyMap : MAP = struct ... end
```

```
;; open MyMap

let x : ________ (int, string) map ________ = add 120 "is fun" empty


let a : ______________________________________ = ([true], [3])

let b : ______________________________________ = [1;2;3]::[4;5;6]

let c : ______________________________________ = entries [(1, "uno"); (2, "dos")]

let d : ______________________________________ = get 3 (add 1 "uno" empty)

let e : ______________________________________ = fun (g:int -> int) -> g 3

let f : ______________________________________ = fun (x:'v) ->
                                                      entries (add 3 x empty)

let g : ______________________________________ = if get 3 empty then 3 else 4

let h : ______________________________________ = [add 1 2; remove 3]
```

4. Binary Trees (20 points)

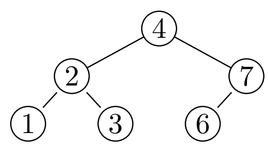
Below is the code for our standard definition of the type of generic binary trees, along with a new function called `tree_transform`, which transforms a given tree in the same way that the list transform function we saw in lecture and HW3 transforms a list.

```
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)

let rec tree_transform (f:'a -> 'b) (t:'a tree) : 'b tree =
  begin match t with
    | Empty -> Empty
    | Node(lt, x, rt) -> Node(tree_transform f lt,
                              f x,
                              tree_transform f rt)
  end
```

Consider the tree `t`, shown below (note that, as usual, the picture omits the `Empty` parts).

`let t : int tree = ...` *(* definition omitted *)*



For each of the four programs `(a)`–`(d)` below, ***draw the tree*** `ans` that is obtained by applying the given function to the tree `t` pictured above.

```
(a) let f1 : int tree -> int tree =
      tree_transform (fun x -> x + 1)

    let ans : int tree = f1 t
```

```
(b) let f2 : int tree -> bool tree =
      tree_transform (fun x -> x > 2)

    let ans : bool tree = f2 t
```

(c)
```
let f3 (t : 'a tree) : 'a tree =
  begin match t with
  | Empty                          -> Empty
  | Node(Empty, x, rt)             -> Node(Empty, x, rt)
  | Node(Node(llt, y, lrt), x, rt) -> Node(llt, y, Node(lrt, x, rt))
  end

let ans : int tree = f3 t
```

(d)
```
let rec f4 (t : 'a tree) : 'a tree =
  begin match t with
  | Empty -> Empty
  | Node(left, x, right) -> Node(f4 right, x, f4 left)
  end

let ans : int tree = f4 t
```

(4 points) Which of the functions `f1` through `f4` *preserve* the binary search tree invariant? (For *all* inputs, not just the examples shown). That is, assuming that the input is a BST, the output is guaranteed to be a BST. Circle each such function.

`f1` `f2` `f3` `f4`

5. List Processing and Higher-order Functions (24 points)

Recall the higher-order list processing functions as defined below:

```
let rec transform (f: 'a -> 'b) (l: 'a list): 'b list =
  begin match l with
    | [] -> []
    | h :: t -> (f h) :: (transform f t)
  end

let rec fold (combine: 'a -> 'b -> 'b) (base:'b) (l : 'a list) : 'b =
  begin match l with
    | [] -> base
    | h :: t -> combine h (fold combine base t)
  end

let rec filter (pred: 'a -> bool) (l: 'a list) : 'a list =
  begin match l with
  | [] -> []
  | hd :: tl -> if pred hd then hd :: (filter pred tl) else filter pred tl
  end
```

a. Use one of `transform`, `fold`, or `filter`, along with suitable anonymous function(s), to implement a function that retains only those pairs of a list whose first element is greater than its second. For example, the call `largest_first [(1,2); (4,3); (5;5); (6;0)]` evaluates to the list `[(4,3); (6;0)]`.

```
let largest_first (l: (int * int) list) : (int * int) list =
```

b. Use one of `transform`, `fold`, or `filter`, along with suitable anonymous function(s), to implement the list reverse function. Recall that `reverse [1;2;3]` evaluates to `[3;2;1]`. You may use the operation `l1 @ l2`, which appends the two lists `l1` and `l2`.

```
let reverse (l:'a list) : 'a list =
```

c. The somewhat clunky code below implements a function called `suffixes` using `fold`. This function computes a list of all the suffixes of a given list. Recall that a suffix of `l` is a contiguous sub-list starting from the *end* of `l`. For example, `suffixes [1;2;3]` evaluates to `[[1;2;3]; [2;3]; [3]; []]`.

```
let suffixes (l:'a list) : 'a list list =
  fold (fun (x:'a) (acc:'a list) ->
      begin match acc with
      | ls::rest -> (x::ls)::acc
      | _ -> failwith "impossible"
      end) [[]] l
```

Fill in the two cases below to re-implement `suffixes` *without* using `fold`. Your code should be much simpler than that above. (This example illustrates why just because it is possible to use `fold` it is not always a good idea.)

```
let rec suffixes (l:'a list) : 'a list list =
  begin match l with

  | []     -> ______________________

  | x::tl -> ______________________________________________________
  end
```

d. Having implemented `reverse` and `suffixes`, we can now use them to conveniently implement `prefixes`, which computes the list of all prefixes of a given list. Recall that a prefix of `l` is a contiguous sub-list starting from the *beginning* of `l`. For example, `prefixes [1;2;3]` evaluates to `[[1;2;3]; [1;2]; [1]; []]`.

Complete the implementation of `prefixes` below. To get full credit, you *may not* use recursion, pattern matching, or anonymous functions. Instead, simply call (some of) `transform`, `fold`, `filter`, `reverse`, and `suffixes` on appropriate arguments.

```
let prefixes (l:'a list) : 'a list list =
```