CIS 120 Midterm I   October 12, 2018

SOLUTIONS

1. **Binary Search Trees (16 points total)**

This problem concerns *buggy* implementations of the `lookup` and `tree_max` functions for binary search trees, the correct versions of which are shown in Appendix A.

First: At most one of the lines of code contains a *compile-time (i.e., typechecking)* error. If there is a compile-time error, explain what the error is and one way to fix it. If there is no compile-time error, say "No Error".

Second: even after the compile-time error (if any) is fixed, the code is still buggy—for some inputs the function works correctly and produces the correct answer, and for other inputs, the function produces an incorrect answer.

```
let t : int tree =     7
                      / \
                     4   9
                    /   / \
                   1   8   10
```

where, as usual, `Empty` constructors are not shown, to avoid clutter.

a. (2 points) Tree `t` satisfies the BST invariants: ☒ True ☐ False

b. (7 points)

```
1   let rec bad_lookup (t: int tree) (n: int) : bool =
2    begin match t with
3      | Empty(_, x, _) -> false
4      | Node(lt, x, rt) ->
5        if n < x then bad_lookup lt n
6        else bad_lookup rt n
7    end
```

```
Compile Error on line _3_ : _Empty constructor doesn't take any arguments_

Fix For Error: ______replace with Empty______________________
```

Complete each of the test cases with an `int` value for `x` so that the test passes, demonstrating that this implementation sometimes produces the correct answers and sometimes does not. Both of the test cases must use the tree `t` shown pictorially above.

ANSWER: This lookup function will always return false. It will work correctly for nodes that are not in the tree.

```
;; run_test "bad_lookup_works_correctly" (fun () ->
     let x = _____15_____ in
     bad_lookup t x = lookup t x)

;; run_test "bad_lookup_computes_wrong_answer" (fun () ->
     let x = _____7______ in
     not (bad_lookup t x = lookup t x))
```

c. (7 points)

```
1  let rec bad_tree_max (t: 'a tree) : 'a =
2    begin match t with
3    | Empty -> failwith "bad_tree_max called on empty tree"
4    | Node(Empty, x, _) -> x
5    | Node(lt, _, _) -> bad_tree_max lt
6    end
```

Compile Error on line ______ : ___No Error_______

For the test cases below, draw pictures of Binary Search Trees `t1` and `t2` where `bad_tree_max` works correctly and incorrectly respectively, demonstrating that this implementation sometimes produces the correct answers and sometimes does not.

```
t1 : int tree =    7

t2 : int tree =          7
                        / \
                       4   9
                      /   / \
                     1   8   10
```

```
;; run_test "bad_tree_max_works_correctly" (fun () ->

     bad_tree_max t1 x = tree_max t1 x)

;; run_test "bad_tree_max_computes_wrong_answer" (fun () ->

     not (bad_tree_max t2 x = tree_max t2 x))
```

ANSWER: This `tree_max` *function actually finds the min of the tree. It will only work correctly when the min and the max are the same, i.e., there is only 1 int in the tree.*

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

Recall the higher-order list processing functions:

```
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
```

For these problems *do not* use any list library functions other than `@`. Constructors, such as `::` and `[]`, are fine.

a. Use `transform` or `fold`, along with suitable anonymous function(s), to implement a function `partition` that returns a pair of lists `(list1, list2)`, where `list1` is the list of all the elements of the input list that satisfy the given predicate `p`, and `list2` is the list of all the elements of the input list that do not satisfy the given predicate `p`. For example, the call `partition (fun x -> x < 4) [6; 5; 2; 3; 4]` evaluates to the pair of lists `([2; 3], [6; 5; 4])`.

```
let partition (p: 'a -> bool) (l: 'a list) : ('a list * 'a list) =
  fold (fun x (acc1, acc2) ->
    if p x then (x::acc1, acc2) else (acc1, x::acc2)) ([], []) l
```

b. Consider the following recursive function:

```
let rec g (x: int) (l: int list) : bool =
  begin match l with
  | [] -> false
  | h :: t -> h = x || g x t
  end
```

Rewrite the above function using `transform` or `fold`.

```
let g (x: int) (l: int list) : bool =
      fold (fun h acc -> h = x || acc) false l
```

c. Consider a modification to the transform function that now takes in two input lists.

```
val transform2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

where,

`transform2 f [a1; ...; an] [b1; ...; bn] is [f a1 b1; ...; f an bn].`

Use `transform2` along with suitable anonymous function(s), to implement a function that creates a `zip` of two lists. For example, the call `zip [1; 2; 3] [``uno''; ``due''; ``tre'']` evaluates to the list `[(1, ``uno''); (2, ``due''); (3, ``tre'')]`. You can assume that the inputs to both `zip` and `transform2` will be lists of the same length.

```
let zip (a: 'a list) (b: 'b list) : ('a * 'b) list =
  transform2 (fun x y -> (x, y)) a b
```

3. **Types** (16 points)

For each OCaml value below, fill in the blank for the type annotation or else write "ill typed" if there is a type error on that line. Your answer should be the *most generic* type that OCaml would infer for the value–i.e., if `int list` and `bool list` are both possible types of an expression, you should write `'a list`.

Some of these expressions refer to the module `Q`, which implements the `Quadrant` interface. The `Quadrant` interface is shown in Appendix B. Note that all of the code appears after the module Q has been opened. The last expression refers to the `Node` that's defined for a `'a tree` in Appendix A.

We have done the first one for you.

```
;; open Q

let z : ______________ 'a list list _____________ = [[]]


let a : ____________quadrant______________________ =
  create (1.0, 2.0) (5.0, 4.0)

let b : ______(point -> quadrant) list__________ =
  [create (1.0, 2.0); create (5.0, 4.0)]

let c : ______________(int -> 'a) -> 'a__________ =
  fun g -> g 3

let d : _____int list * int list list___________ =
  (1::[2], [3]::[[4]])

let e : ______________quadrant list______________ =
  split (enclosing_quad [])

let f : _______________ill typed_________________ =
  fun x y -> begin match x with
          | [] -> split y
          | _ -> inside_quad y
          end

let g : ____'a -> ('a option * 'a) list_________ =
  fun x -> (None, x)::[(Some x, x)]

let h : _______________ill typed_________________ =
  Node("a", "b", Node(Empty, Empty, "c"))
```

4. Abstract Types, Invariants, and Modularity (29 points total)

In this problem we will implement a new abstract type called a `Quadrant`. Quadrants are used to represent spatial data (maps). The `Quadrant` interface is shown in Appendix B.

As usual, the behavior of the `quadrant` abstract type is specified by defining the properties of its operations. For each of the following properties, define a corresponding test case. Assume that the `quadrant` module is opened and that `q` is defined as shown. We have done an example test case for you below.

Example:

Property: A quadrant is defined by its bottom left and top right points. A point is defined by its x and y coordinates.

```
;; open Quadrant

let botLeft : point = (1.0, 2.0)
let topRight : point = (5.0, 4.0)
let q : quadrant = create botLeft topRight

let test () : bool =
  bounds q = ((1.0, 2.0), (5.0, 4.0))

;; run_test "create q1" test
```

a. (4 points) Property: When a quadrant is split, the sub quadrants are returned in the order displayed below.

```
if q =  ____________tr   split q =  ____________tr
        |           |               |(1)  |  (2) |
        |           |               |_____|______|
        |           |               |(3)  |  (4) |
      bl____________|             bl_____|______|
```

bl: bottom left point

tr: top right point

`split q` will return

`[quadrant (1); quadrant (2); quadrant (3); quadrant (4)]`

```
let test () =

  split q1 = [create (1.0, 3.0) (3.0, 4.0);
              create (3.0, 3.0) topRight;
              create botLeft (3.0, 3.0);
              create (3.0, 2.0) (5.0, 3.0)]
;; run_test "list of quadrants returned after split" test
```

b. (4 points) Property: When `enclosing_quad` is called with one point, the quadrant is a square of size 1 (top right x - bottom left x = 1 and top right y - bottom left y = 1) and the point is the bottom left bound.

```
let test () =
(* bounds (enclosing_quad [(x, y)]) = ((x, y), (x+1, y+1)) *)
bounds (enclosing_quad [(1.0, 2.0)]) = ((1.0, 2.0), (2.0, 3.0))
;; run_test "enclosing_quad q1 1 point" test
```

c. (4 points) Property: `enclosing_quad` returns the smallest quadrant containing all the points.

```
let test () =
 bounds (enclosing_quad [(2.0, 3.0); (1.0, 2.0);
 (5.0, 4.0); (1.0, 1.5)]) = ((1.0, 1.5), (5.0, 4.0))
;; run_test "enclosing_quad " test
```

d. (4 points) Property: `make_quads` returns only one quadrant when only one point is in the list and the number of points n = 1. The quadrant is created using the property listed in question 4.b.

```
let test () =
 make_quads [(1.0, 2.0)] 1 = [create (1.0, 2.0) (2.0, 3.0)]
;; run_test "make_quad 1 point " test
```

e. (13 points) We can implement the `Quadrant` interface in many ways, but in this problem we use as the representation type a tuple of points that are the bottom left and top right points respectively. Complete the following implementation of the quadrant `enclosing_quad` operation. Note the following:

- We'll define a `Quadrant` as follows: `type quadrant = point * point`
- When `enclosing_quad` is called with an empty list, the quadrant is of size 0 (bottom left and top right points are the same).
- When `enclosing_quad` is called with one point, the quadrant is a square of size 1 and the point is the bottom left bound.

```
module Q : Quadrant = struct

  type quadrant = point * point

  let rec enclosing_quad (l: point list) : quadrant =
   begin match l with
    | [] -> ((0.0, 0.0), (0.0, 0.0))
    | [h] -> begin match h with
              | (x, y) -> ((x, y), (x +. 1.0, y +. 1.0))
              end
    | h :: y -> begin match y with
                | [b] -> let (x, y) = h in
                         let (x0, y0) = b in
                         let left_X = min x x0 in
                         let left_Y = min y y0 in
                         let right_X = max x x0 in
                         let right_Y = max y y0 in
                           ((left_X, left_Y), (right_X, right_Y))
                | _ ->
                  let ((x0,y0), (x1, y1)) = bounds (enclosing_quad y) in
                  let (x, y) = h in
                  let left_X = min x x0 in
                  let left_Y = min y y0 in
                  let right_X = max x x1 in
                  let right_Y = max y y1 in
                    ((left_X, left_Y), (right_X, right_Y))
                end
   end
```

5. Recursion and Trees (15 points)

Recall the type of a generic Binary Search Tree:

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

Implement a (partial) function called `scs`, short for smallest containing subtree. This function should, when given two values that may appear in a binary search tree, return the smallest subtree that contains both of those values, if possible.

For example, given the tree the smallest containing subtree of 1 and 4 is

```
t1 =     5
       /   \
      3     8
     / \   / \
    1   4 7   10
```

```
t2 =      3
         / \
        1   4
```

Likewise, the smallest subtree of `t1` containing 1 and 3 is also `t2`. On the other hand, the smallest subtree of t1 that contains both 1 and 5 is the whole tree.

You should assume that the input tree is a binary search tree, and that the first argument is smaller than the second. Your solution does not need to detect whether any of these assumptions are violated. Your implementation must take advantage of the binary search tree invariant and must work for generic binary search trees. If there is no such tree, e.g., if the values don't appear in the tree, return `None`.

The definition of a BST along with the `insert`, `delete`, and `lookup` functions are provided in Appendix A. You're welcome to use any of these in your code if needed.

```
(* Assume that x < y and t is a BST *)
(* Hint: The BST invariants will be helpful here! *)

(* Solution 1 *)
let rec scs (x: 'a) (y: 'a) (t:'a tree) : 'a tree option =
  let rec loop (x: 'a) (y: 'a) (t: 'a tree) : 'a tree option =
      begin match t with
      | Empty -> None
      | Node(lt, z, rt) ->
        if x > z then loop x y rt
        else if y < z then loop x y lt
        else Some t
      end
    in
    if (lookup t x && lookup t y) then loop x y t
    else None
```

```
(* Solution 2 *)
let rec scs (x: 'a) (y: 'a) (t:'a tree) : 'a tree option =
   begin match t with
   | Empty -> None
   | Node(lt, z, rt) ->
     if x > z then scs x y rt
     else if y < z then scs x y lt
     else if lookup t x && lookup t y then Some t
     else None
   end

(* This solution also correctly returned the smallest common subtree,
   but did not take advantage of the BST invariants and called lookup
   at every recursive step (lookup only had to be called twice) *)
let rec scs (x: 'a) (y: 'a) (t:'a tree) : 'a tree option =
   begin match t with
   | Empty -> None
   | Node(lt, z, rt) ->
     if lookup lt x && lookup lt y then scs x y t
     else if lookup rt x && lookup rt y then scs x y t
     else if lookup t x && lookup t y then Some t
     else None
   end
```

Appendix A: (Binary Search) Trees

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

(* checks if n is in the BST t *)
let rec lookup (t:'a tree) (n:'a) : bool =
  begin match t with
  | Empty -> false
  | Node(lt, x, rt) ->
    if x = n then true
    else if n < x then lookup lt n
    else lookup rt n
  end

(* returns the maximum integer in a *NONEMPTY* BST t *)
let rec tree_max (t: 'a tree) : 'a =
  begin match t with
  | Empty -> failwith "tree_max called on empty tree"
  | Node(_, x, Empty) -> x
  | Node(_, _, rt) -> tree_max rt
  end

(* Inserts n into the BST t *)
let rec insert (t: 'a tree) (n: 'a) : 'a tree =
  begin match t with
  | Empty -> Node(Empty, n, Empty)
  | Node(lt, x, rt) ->
    if x = n then t
    else if n < x then Node (insert lt n, x, rt)
    else Node(lt, x, insert rt n)
  end

(* returns a BST that has the same set of nodes as t except with n removed (if it's there) *)
let rec delete (t: 'a tree) (n: 'a) : 'a tree =
  begin match t with
  | Empty -> Empty
  | Node(lt, x, rt) ->
    if x = n then
      begin match (lt, rt) with
      | (Empty, Empty) -> Empty
      | (Empty, _) -> rt
      | (_, Empty) -> lt
      | (_,_)      -> let y = tree_max lt in Node (delete lt y, y, rt)
      end
    else if n < x then Node(delete lt n, x, rt)
    else Node(lt, x, delete rt n)
  end
```

Appendix B: `Quadrant` Module Signature

The signature below defines the `Quadrant` interface and the module `Q` (whose code is not shown), which implements that interface.

```
(* a point is defined by its x and y coordinates *)
type point = float * float

module type Quadrant = sig

  type quadrant

  (* create a new quadrant *)
  val create : point -> point -> quadrant

  (* return the bottom left and top right points of the quadrant *)
  val bounds : quadrant -> point * point

  (* divide the quadrant --vertically and horizontally -- in 4 equal size sub quadrants *)
  val split : quadrant -> quadrant list

  (* return the smallest quadrant containing all the points *)
  val enclosing_quad : point list -> quadrant

  (* return only the points contained in the quadrant *)
  val inside_quad : quadrant -> point list -> point list

  (* return a list of quadrants. Each quadrant containing at most n points *)
  val make_quads : point list -> int -> quadrant list

end


module Q : Quadrant = struct

  type quadrant = point * point

  (* ... rest of the code not shown ... *)

end
```

Appendix C: List Processing Higher Order Functions

Here are the higher-order list processing functions:

```
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
```