CIS 120 Midterm I   October 13, 2017

SOLUTIONS

1. Binary Search Trees (16 points total)

This problem concerns *buggy* implementations of the `insert` and `delete` 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 BST, and for other inputs, the function produces an incorrect tree. Complete each of the test cases with an `int` value for `x` so that the test passes, demonstrating that these implementations sometimes produce the correct answers and sometimes do not. The test cases all use the tree `t` shown pictorially as

```
let t : int tree =     4
                      / \
                    1    7
                  / \     \
                 0    2    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_insert (t:int tree) (n:int) : int tree =
2   begin match t with
3     | Empty -> n
4     | Node(lt, x, rt) ->
5       if n < x then Node(bad_insert lt n, x, rt)
6       else Node(lt, x, bad_insert rt n)
7   end
```

```
Compile Error on line ____3_____ : _returning an int not a int tree_

Fix for Error: ________replace n with Node(Empty, n, Empty)_________
```

ANSWER: This insert function will add duplicate nodes to the right side. It will work correctly if insert is only called on nodes that don’t already exist.

```
;; run_test "bad_insert_works_correctly" (fun () ->
     let x = _____________ in
     bad_insert t x = insert t x)

;; run_test "bad_insert_computes_wrong_answer" (fun () ->
     let x = _____________ in
     not (bad_insert t x = insert t x))
```

c. (7 points)

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

Compile Error on line ________ : _____NO ERRORS_______________________

Fix for Error: ____

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


      let x = _____________ in
      bad_delete t x = delete t x)

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


      let x = _____________ in
      not (bad_delete t x = delete t x))
```

ANSWER: This deletion function only deletes nodes not in the tree (so not very helpful).

2. Higher-order Functions (21 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 such as `@`. Constructors, such as `::` and `[]`, are fine.

a. Use `tranform` or `fold`, along with suitable anonymous function(s), to implement a function that removes all elements from a list that match the criteria specified. For example, the call `reject (fun x -> x > 2) [1; 2; 3; 4]` evaluates to the list `[1; 2]`.

```
let reject (pred: 'a -> bool) (l: 'a list) : 'a list =
  fold (fun x acc -> if pred x then acc else x::acc) [] l
```

b. Is `tranform` just a `fold`? Implement `transform` using `fold` if possible. If it's not possible, explain why.

```
let transform_using_fold (f: 'a -> 'b) (l: 'a list) : 'b list =
      fold (fun x acc -> (f x) :: acc) [] l;;
```

c. Use `tranform` or `fold`, along with suitable anonymous function(s), to implement a function that removes all duplicates from a list. For example, the call `uniq [1; 2; 3; 4; 1; 2]` evaluates to the list `[1; 2; 3; 4]`. The order of elements in the returned list doesn't matter so the above call could also return `[3; 4; 1; 2]`.

```
let uniq (l: 'a list) : 'a list =
  let mem x l = fold (fun y acc -> x = y || acc) false l in
  fold (fun x acc -> if mem x acc then acc else (x::acc)) [] l
(* or *)
let uniq' (l:'a list) : 'a list =
  fold (fun x acc ->
    if (fold (fun y acc -> x = y || acc) false acc) then acc else (x::acc)) [] l
```

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 `S`, which implements the `SET` interface. The `SET` interface is shown in Appendix B. Note that all of the code appears after the module `S` has been opened.

We have done the first one for you.

```
;; open S

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


let a : ________________ill typed________________ = add 3 []


let b : __________'a set list_____________________ = [empty; empty]


let c : ________________ill typed________________ =
  begin match (add 3 empty) with | [] -> 0 | x::xs -> 1 end


let d : ______int set -> bool __________________ = equals (add 3 empty)


let e : ________'a list -> int____________________ =
  fun x -> begin match x with | [] -> 3 | _ -> 4 end


let f : _______________ill typed_________________ =
  remove "hello" (add 3 empty)


let g : _____int set set -> int set set _________ = add (add 3 empty)


let h : ___(int set -> int set) list____________ = [add 3; add 4; add 5]
```

Grading Scheme: 2 points each. 1 points partial credit for “close, but not quite”

4. Abstract Types, Invariants, and Modularity (32 points total)

In this problem, you will use the `S` set module (see Appendix B) to implement another abstract collection type, called a *multimap*. A multimap associates *keys* of type `'k` to *sets* of values of type `'v`. The interface for a multimap is the following:

```
module type MULTIMAP = sig
  type ('k, 'v) multimap
  val empty : ('k, 'v) multimap
  val add : 'k -> 'v -> ('k, 'v) multimap -> ('k, 'v) multimap
  val mem : 'k -> ('k, 'v) multimap -> bool
  val get : 'k -> ('k, 'v) multimap -> 'v S.set
  val remove : 'k -> ('k, 'v) multimap -> ('k, 'v) multimap
end
```

As usual, the behavior of the multimap abstract type is specified by defining the properties of its operations. For each of the properties on the next page, define a corresponding test case. Assume that the `MultiMap` module is `open`ed and that `m1` and `m2` are defined as shown. We have done an example test case for you below.

Example:

Property: A multimap collects together *all* of the values added with a given key, which are returned by the `get` operation as a value of type `'v S.set`.

```
;; open MultiMap
let m1 : (int, string) multimap = add 1 "a" empty
let m2 : (int, string) multimap = add 1 "b" m1

let test () =
  S.equals (get 1 m2) (S.set_of_list ["a"; "b"])
;; run_test "m2 maps key 1 to both a and b" test
```

(There are no questions on this page.)

PennKey: ____________________________________

```
;; open MultiMap
let m1 : (int, string) multimap = add 1 "a" empty
let m2 : (int, string) multimap = add 1 "b" m1
```

a. (3 points) Property: If no values have been added with a given key, calling `get` on that key returns an empty set.

```
let test () =
  S.equals (get 1 empty) (S.empty)
;; run_test "get unassociated key" test
```

b. (3 points) Property: When we `remove` a key from the multimap, *all* of the associated values are removed too.

```
let test () =
  S.equals (get 1 (remove 1 m2)) (S.empty)
;; run_test "remove removes all" test
```

c. (3 points) Property: The `mem` operation returns **`false`** for a key `k` after `k` is `remove`d.

```
let test () =
  not (mem 1 (remove 1 m2))
;; run_test "key not a member after removal " test
```

d. (4 points) Suggest a fifth property (different from the ones above) that you would expect to hold about the `MultiMap` abstraction. Write a one-sentence description of it and a test case that checks the property:

Example Properties:

- Adding the same key–value pair twice is the same as adding it once.
- The `mem` operation returns **`true`** for any key that has been added.

```
let test () =

  ______________________________________________________

;; run_test "______________________________________________________" test
```

We can implement the multimap interface in many ways, but in this problem we use as the representation type an *association list* defined in the code below. We also choose to use the following invariant:

INVARIANT: the pairs in the list are sorted (in increasing order) by key values

We have given you the definition of the `empty` multimap.

```
module MultiMap : MULTIMAP = struct
  (* Invariant: the list is sorted in strictly increasing order by keys *)
  type ('k, 'v) multimap = ('k * 'v S.set) list

  let empty : ('k, 'v) multimap = []
```

e. (15 points) Complete the following implementation of the multimap `add` operation. Be sure to exploit and preserve the representation invariant. Note that this code is within the `MultiMap` module structure, but it can refer to `S`'s set operations using the "dot" notation (e.g. `S.empty`).

```
let rec add (k:'k) (v:'v) (m:('k, 'v) multimap) : ('k, 'v) multimap =
 begin match m with
 | [] -> [(k, S.add v S.empty)]
 | (x,vs)::t ->
   if k < x then (k, S.add v S.empty)::m
   else if k = x then (x, S.add v vs)::t
   else (x,vs)::(add k v t)
 end
```

Consider the following two possible implementations of the multimap `get` operation. (Note that the definition of `fold` is found in problem 2.)

```
(* A *)
let rec get (k:'k) (m:('k, 'v) multimap) : 'v S.set =
  begin match m with
  | [] -> S.empty
  | (x,vs)::xs ->
    if k < x then S.empty
    else if k = x then vs
    else get k xs
  end

(* B *)
let get (k:'k) (m:('k, 'v) multimap) : 'v S.set =
  fold (fun (x,vs) acc -> if k = x then vs else acc) S.empty m
```

(4 points) Mark all correct answers (there may be zero or more than one):

f. Which of the implementations correctly implement the desired behavior of the multimap?

☒ A ☒ B

g. Which of the implementations use the representation invariant to improve efficiency?

☒ A ☐ B

5. Recursion and Trees (15 points)

This problem uses the `'a tree` datatype from Appendix A, but *does not assume the binary search tree invariants.*

Complete this (partial) recursive function called `deepest_leaf` that, given a tree `t` finds the value and depth of the leaf farthest from the root of `t`. If there is no such leaf (i.e. the tree is `Empty`) return `None`. If there is a tie for deepest, pick the left-most of the deepest values.

Here are several examples, and the expected outputs:

```
     1                 3                  0
    / \                 \                / \
   2   2                 1              1   1
      / \               / \            / \   \
     3   7             4   0          2   2   2
                            \            /
                             6          3

Some (3, 2)          Some(6, 3)       Some (3, 3)
```

Hint: Think about how to adapt the recursive algorithm for tree `height` to this scenario.

```
let rec deepest_leaf (t:'a tree) : ('a * int) option =
  begin match t with
  | Empty -> None
  | Node(lt, x, rt) ->
    begin match (deepest_leaf lt, deepest_leaf rt) with
    | None, None -> Some (x, 0)
    | Some (x, n), None -> Some (x, n+1)
    | None, Some (x, n) -> Some (x, n+1)
    | Some (x, n), Some (y, m) ->
      if n < m then Some (y, m+1) else Some (x, n+1)
    end
  end
```

Appendix A: (Binary Search) Trees

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

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

(* returns a binary search tree that has the same set of
   nodes as t except with n removed (if it's there)
 *)
let rec delete (t:'a tree) (n:int) : '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: `SET` Module Signature

The signature below defines a simplified version the `SET` interface used in the homework project about abstract types, and the module `S` (whose code is not shown), which implements that interface.

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add : 'a -> 'a set -> 'a set
  val remove : 'a -> 'a set -> 'a set
  val member : 'a -> 'a set -> bool
  val set_of_list : 'a list -> 'a set
  val equals : 'a set -> 'a set -> bool
end
module S : SET = struct (* ... code not shown ... *) end
```