CIS 120 Midterm I September 27, 2019

SOLUTIONS

1. **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 variable `z` (which is defined in the example at the top), to the functions `transform` and `fold` (whose definitions can be found on page 13), or to the constructors of the type `'a tree`, which is defined as:

```
type 'a tree =
  | Leaf of 'a list * int
  | Node of 'a tree * int * 'a tree
```

We have done the first one (`z`) for you. (2 points each)

```
let z : ______________int tree _____________ =
  Leaf([1], 26)


let a : int list list =
  [1;2]::[3;4]::[]

let b : ('a list * 'b list) =
  ([], [])

let c : ill-typed =
  begin match z with  (* z is defined above *)
  | [] -> 0
  | x::xs -> 1
  end

let d : int -> int tree =
  fun (x:int) -> Leaf([x], x)

let e : int list =
  transform (fun x -> x + 1) [1;2;3]

let f : int list -> (int -> int) list =
  transform (fun x y -> x + y)

let g : bool list -> int =
  fold (fun x acc -> if x then acc else acc + 1) 0

let h : ill-typed =
  Node (z, 4, Leaf([true], 3))
```

2. **List Processing and Higher-Order Functions (24 points total)**

   Recall the higher-order list processing functions shown in Appendix A.

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

   (a) (6 points) Use `transform` or `fold`, along with suitable anonymous function(s), to implement a function `flatten` that takes in a `'a list list` and returns a `'a list`. It should remove one "nesting" level for the lists. For example, the call `flatten [[1; 2]; [3; 4]]` evaluates to the list `[1; 2; 3; 4]` and the call `flatten [[[1]; [2]]; [[]]]` evaluates to the list `[[1]; [2]; []]`.

```
let flatten (l : 'a list list) : 'a list =
  fold (fun (x: 'a list) (acc: 'a list) -> x @ acc) [] l
```

   (b) (6 points) Consider the following recursive function:

```
let rec f (x: 'a) (g: 'a -> 'a -> int) (l: 'a list) : int =
  begin match l with
  | [] -> 0
  | h :: t -> g h x + f x g t
  end
```

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

```
let f (x: 'a) (g: 'a -> 'a -> int) (l: 'a list) : int =
  fold (fun h acc -> g h x + acc) 0 l
```

(c) (12 points) Consider a function that squares all the integers in a given list. If the input to that function is the list `x` shown below, the result should be the list `y`.

```
let x : int list = [1; 2; 3; 4]

let y : int list = [1; 4; 9; 16]
```

Which of the following functions will typecheck and produce the correct answer? (Mark all that apply.)

☒
```
let rec squares (l : int list) : int list =
  begin match l with
  | [] -> []
  | hd::tl -> (hd * hd) :: (squares tl)
  end
```

☒
```
let rec squares (l : int list) : int list =
  transform (fun (x : int) -> x * x) l
```

☐
```
let rec squares (l : int list) : int list =
  transform (fun (x : int) -> [x * x]) l
```

☐
```
let rec squares (l : int list) : int list =
  fold (fun (x : int) (acc : int list) -> x * x) [] l
```

☐
```
let rec squares (l : int list) : int list =
  fold (fun (x : int) (acc : int list) -> [x * x]) [] l
```

☒
```
let rec squares (l : int list) : int list =
  fold (fun (x : int) (acc : int list) -> x * x :: acc) [] l
```

3. **Modules and Abstract Types (40 points total)**

Step 1: Understand the Problem The standard list operations like `length`, `append`, and `nth` take time proportional to the size of the (first) list argument. As a reminder, `nth lst n` finds the n^{th} element of the list `lst` by counting from the head (starting at 0) towards the tail one element at a time. For instance, `nth [0;1;2;3] 0` evaluates to `0` and `nth [0;1;2;3] 2` evaluates to `2`. If `nth` is given an index greater than (or equal to) the length of the list, it fails.

For your reference, Appendix B gives the usual implementations of these operations, found in the `List` module. Sometimes these are too slow for the task at hand. In this problem we consider how to combine trees and lists to more efficiently implement them.

Step 2: Design the Interface The signature below defines an abstract type `'a rope` and operations on it. A `rope`, like a `list`, stores a sequence of data elements.

```
module type ROPE = sig
  type 'a rope
  val from_list : 'a list -> 'a rope
  val to_list   : 'a rope -> 'a list
  val append    : 'a rope -> 'a rope -> 'a rope
  val length    : 'a rope -> int
  val nth       : 'a rope -> int -> 'a
end
```

The *properties* of the `ROPE` interface are the same as those for the corresponding list operations—in that regard, a rope is "just" a different implementation of the list abstract type. This means that a functionally correct implementation of this interface is:

```
module ListRope : ROPE = struct
  type 'a rope = 'a list

  let from_list (l : 'a list) : 'a rope = l
  let to_list   (r : 'a rope) : 'a list = r
  let length    (r : 'a rope) : int = List.length r
  let append (lr : 'a rope) (rr : 'a rope) : 'a rope = List.append lr rr
  let nth (r : 'a rope) (n : int) : 'a = List.nth r n
end
```

(a) (5 points) Which of the following properties hold of `ListRope`? Assume we have done `;; open ListRope` to import the definitions above, that `r`, `r1`, and `r2` refer to arbitrary values of type `'a rope`, and `lst` is a `'a list`. (Mark all that apply.)

- ☒ `length r = List.length (to_list r)`
- ☒ If `to_list r = lst` then `nth r n = List.nth lst n`
- ☒ `length (append r1 r2) = (length r1) + (length r2)`
- ☒ If `(n < length r1)` then `nth (append r1 r2) n = nth r1 n`
- ☐ If `(n >= length r1)` then `nth (append r1 r2) n = nth r2 n`

Step 3: Define Test Cases (8 points) Our more efficient rope implementation, called `TreeRope`, should satisfy the same properties as `ListRope`. Complete each of the test cases below by filling in the blanks with identifiers `r0`, `r1`, `r2`, `r3`, or `r4` so that each test succeeds.

```
;; open TreeRope

let r0 = from_list [0;1;2]
let r1 = from_list [3;4]
let r2 = from_list [5;6;7;8]
let r3 = append r0 (append r1 r2)
let r4 = append (append r1 r1) r1
```

(a)
```
let test () =
  to_list r3 = [0;1;2;3;4;5;6;7;8]
;; run_test "test1" test
```

(b)
```
let test () =
  nth r2 2 = 7
;; run_test "test2" test
```

(c)
```
let test () =
  nth r1 2 = 0
;; run_failing_test "test3" test
```

(d)
```
let test () =
  nth (append r0 r1) 4 = nth r1 (4 - length r0)
;; run_test "test4" test
```

Step 4: Implement the Code To implement these list operations more efficiently, we choose a different representation based on binary trees, encapsulated in a module named `TreeRope`. The module declaration and tree type are shown below.

```
module TreeRope : ROPE = struct

  type 'a tree =
    | Leaf of 'a list * int
    | Node of 'a tree * int * 'a tree
  type 'a rope = 'a tree
```

We first make the `append` operation faster. If we already have a list, we can treat it as a rope by storing it directly in `Leaf`. If we want to append two ropes, we simply join them with a `Node` constructor, which, unlike `List.append` doesn't require traversing either list. The main idea is that **each leaf of the tree contains only part of the complete sequence of data stored in the rope—**to convert a tree into the corresponding list, we append all the lists at the leaves using in order traversal. This code is shown below:

```
let rec to_list (r : 'a tree) : 'a list =
  begin match r with
    | Leaf (l,_) -> l
    | Node (lt, _, rt) ->
      List.append (to_list lt) (to_list rt)
  end
```

To accelerate the `length` and `nth` operations, we store extra information in the tree. Each leaf, in addition to the (partial) list data, also stores the length of that piece; the length is computed just once when the leaf is created, so repeatedly asking for length information about the leaf data doesn't require repeated traversals of the list at the leaf. Moreover, the total length of the lists in its left child are stored at the node. Finally, there is no point in storing lots of leaves that contain the empty list, so we require that every left subtree have size strictly greater than 0 (which means its leaves can't contain just empty lists). Stated as invariants, we have:

Rope Invariants

A value `r : 'a rope` satisfies the rope invariants if:

- `r` is `Leaf(lst, n)` and `List.length lst = n`, or
- `r` is `Node(lt, n, rt)` and
  - `n > 0` and `n` is the total length of all the lists stored at the leaves in `lt`
  - `lt` and `rt` both recursively satisfy the rope invariants

(Nothing to do on this page.)

(a) (3 points) Given the invariants above, which of the following is a correct implementation for the `length` operation on ropes?

☐
```
let rec length (t : 'a tree) : int =
  begin match t with
    | Leaf (l,_) -> 0
    | Node (_, _, rt) -> 1 + length rt
  end
```

☐
```
let rec length (t : 'a tree) : int =
  begin match t with
    | Leaf (l,x) -> x
    | Node (lt, x, _) -> x + length lt
  end
```

☒
```
let rec length (t : 'a tree) : int =
  begin match t with
    | Leaf (l,x) -> x
    | Node (_, x, rt) -> x + length rt
  end
```

(b) (2 points) Given the invariants above, there is a unique value `r : int rope` such that `to_list r = []`.

☒ True ☐ False

(c) (2 points) Given the invariants above, there is a unique value `r : int rope` such that `to_list r = [2]`.

☐ True ☒ False

(d) (2 points) Given the invariants above, there is a unique value `r : int rope` such that `to_list r = [2;3;4]`.

☐ True ☒ False

Complete the code for each of the following operations that build rope trees. In each case, ensure that the resulting tree satisfies the rope invariants. You may use `List.length` to refer to the list version of length and just `length` to refer to the rope version defined above. Do *not* use `List.append` (or `@`) in this implementation. Note that neither operation below is recursive!

(e) (4 points)

```
let from_list (l : 'a list) : 'a tree =
  Leaf (l, List.length l)
```

(f) (6 points)

```
let append (lt : 'a tree) (rt : 'a tree) : 'a tree =
  let x = length lt in
  if x = 0 then rt else
    Node (lt, x, rt)
```

Complete the code for the rope version of the `nth` operation. Your implementation should exploit the rope invariants as much as possible. You may use `List.nth` to refer the list version of nth. Note that this function is recursive!

(g) (8 points)

```
let rec nth (t : 'a tree) (n : int) : 'a =
  begin match t with
    | Leaf (l,_) -> List.nth l n
    | Node (lt, x, rt) ->
      if n < x then nth lt n else nth rt (n - x)
  end
```

4. **Binary Search Trees (20 points total)**

This problem concerns *buggy* implementations of the `lookup` and `insert` functions for binary search trees, the correct versions of which are shown in Appendix C. Note that this problem refers to the `'a tree` type defined there.

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   5   10
```

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

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

(b) (9 points)

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

```
Compile Error on line _3_ : _The expression has type tree instead of bool_

Fix For Compile Error: ______replace with false________________
```

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 search the wrong part of the tree . It will work correctly only for root nodes and for nodes that are not in the tree.

```
;; run_test "bad_lookup_works_correctly" (fun () ->
      let x = _____7______ in
      bad_lookup t x  =  lookup t x)

;; run_test "bad_lookup_computes_wrong_answer" (fun () ->
      let x = _____1______ in
      not (bad_lookup t x = lookup t x))
```

(c) (9 points)

```
1   let rec bad_insert (t: 'a tree) (n: 'a) : 'a tree =
2    begin match t with
3    | Empty -> Empty
4    | Node(lt, x, rt) ->
5       if x = n then t
6       else if n < x then Node (bad_insert lt n, x, rt)
7       else Node(lt, x, bad_insert rt n)
8    end
```

Compile Error on line ______ : ___No Error________

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

```
Works Correctly:
n : int = 7
t1 : int tree =    7

Works Incorrectly
n : int = 5
t2 : int tree =         7
                       / \
                      4   9
```

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

      bad_insert t1 n  =  insert t1 n)

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

      not (bad_insert t2 n = insert t2 n))
```

ANSWER: This `insert` *function never actually inserts an element into the tree. So it will work correctly only if the element is already present in the tree.*

Scratch Space

Use this page for work that you do not want us to grade. If you run out of space elsewhere in the exam and you ***do*** *want to put something here that we should grade, make sure to put a clear note on the page for the problem in question.*

Appendix A: Higher-Order List Processing 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
```

Appendix B: List Operations

```
(* Relevant part of the list library *)
module List = struct
  (* ... other operations elided ... *)

  let rec length (l : 'a list) : int =
    begin match l with
      | [] -> 0
      | _::xs -> 1 + length xs
    end

  let rec append (l1 : 'a list) (l2 : 'a list) : 'a list =
    begin match l1 with
      | [] -> l2
      | x::xs -> x::(append xs l2)
    end

  let rec nth (l : 'a list) (n:int) : 'a =
    begin match l with
      | [] -> failwith "not found"
      | x::xs -> if n = 0 then x else nth xs (n-1)
    end
end
```

Appendix C: (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
```