CIS 1200 Midterm I February 16, 2024

SOLUTIONS

1. **Types** (16 points total)

   For each OCaml value below, fill in the missing type annotations or else write "ill typed" if there is no way to fill in the annotation that does not cause a type error.

   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 types and functions defined in the appendix.

   We've done the first one for you.

```
let ans: ___________int list___________ = [3; 1]
```

(a)
```
let ans: bool -> bool =
    fun x -> x || x
```

(b)
```
let ans: int = 6 + if 3 < 5 then 4 else 1
```

(c)
```
let ans: ill-typed = "hello"::"world"::[]::[]
```

(d)
```
let ans: bool list * int = ([true; false], 5)
```

(e)
```
let ans: ill-typed = begin match [2; 4; 5] with
   | [] -> "empty"
   | hd::tl -> hd
   end
```

(f)
```
let ans: int -> bool =
    (fun (x : int) -> fun (y : int) -> x > y) 3000
```

(g)
```
let ans: int list list -> int list =
     fold (fun (x : int list) (acc : int list) -> x @ acc) []
```

(h)
```
let ans: ('a * 'a list) list -> 'a list list =
    fun z -> transform (fun (x, y) -> x::y) z
```

2. **Binary tree representation and recursion** (18 points total)

   This problem asks you to compare two different definitions of trees, related to those you have seen in the homework assignments.

   The first tree type is a generic version of `labeled_tree`, a data type used in the second homework assignment.

```
(* The labelled tree type from HW2, made generic *)
type 'a labeled_tree =
  | LLeaf of 'a
  | LNode of 'a labeled_tree * 'a * 'a labeled_tree
```

   (a) (4 points) Which of these trees can be represented using the `labeled_tree` type?

```
      i.              ii.         iii.          iv.

       2               3            4             4
      / \             /            / \           / \
     1   3           1            1   6         1   6
                                     / \           /
                                    5   7         5
```

True ☒	False ☐	`i.`
True ☐	False ☒	`ii.`
True ☒	False ☐	`iii.`
True ☐	False ☒	`iv.`

   (b) (4 points) What do we know about all `labeled_tree`s?

True ☐	False ☒	The tree is a binary search tree.
True ☒	False ☐	The tree contains at least one value.
True ☐	False ☒	The tree contains an even number of values.
True ☐	False ☒	~~All nodes in the tree have either two empty subtrees or two nonempty subtrees.~~ *Question removed from the exam.*

(c) (10 points) Now compare the `labeled_tree` type to the generic binary tree type that you used in the third homework assignment, repeated in the appendix on page 14.

The following function translates a `tree` to a `labeled_tree` of the same structure.

```
let rec to_labeled_tree (t : 'a tree): 'a labeled_tree =
  begin match t with
  | Empty -> failwith "No empty labeled_tree"
  | Node (Empty, x, Empty) -> LLeaf x
  | Node (l , x, r) -> LNode (to_labeled_tree l, x, to_labeled_tree r)
  end
```

However, note that this function fails on some inputs (using `failwith`).

Implement a generic function, called `is_labeled_tree`, that returns **`true`** exactly when this function succeeds. In other words, if `is_labeled_tree` function returns **`true`** for some `tree`, then `to_labeled_tree` should return a `labeled_tree` for that input. Conversely, if `is_labeled_tree` returns **`false`**, then `to_labeled_tree` should fail on that input.

```
(* Can t be represented as a labeled_tree? *)
let rec is_labeled_tree (t : 'a tree) : bool =
  begin match t with
  | Empty -> false
  | Node (Empty, x, Empty) -> true
  | Node (l, x, r) -> is_labeled_tree l && is_labeled_tree r
  end
```

3. **Binary Search Trees** (32 points total)

This problem concerns a variation of binary search trees called *sized binary search trees*, or *SBST*s for short.

A *sized tree* contains an extra `int` at each node. We represent a sized tree in OCaml using the following datatype definition.

```
type 'a sized_tree =
  | SEmpty
  | SNode of 'a sized_tree * ('a * int) * 'a sized_tree
```

A tree satisfies the **size invariant** if the integer stored at each node is one more than the sum of the sizes of its left and right subtrees. The size of a tree is defined as follows.

```
(* access the size of the tree *)
let size (t:'a sized_tree) : int =
  begin match t with
  | SEmpty -> 0
  | SNode (_,(_,sz),_) -> sz
  end
```

We can check whether a tree satisfies the *size invariant* using the following function.

```
(* make sure that the size values are correct in the tree  *)
let rec is_sized_tree (t : 'a sized_tree) : bool  =
  begin match t with
  | SEmpty -> true
  | SNode (lt, (v, sz), rt) ->
     sz = size lt + 1 + size rt &&
     is_sized_tree lt && is_sized_tree rt
  end
```

(a) (4 points)

If a tree satisfies the *size invariant*, then the `size` function will always return the number of values stored in the tree.

[✓ True ☒] False ☐

If a tree does not satisfy the *size invariant*, then the `size` function always will return a number that is different from the number of values stored in the tree.

True ☐ [✓ False ☒]

(b) (8 points) A *SBST* is a tree that satisfies both the *size* and *binary search tree* invariants.

To avoid confusing values and sizes in this problem, we will only work with trees that contain string values. Recall that in OCaml, strings can be compared using the < operator, and that `"a" < "b"` evaluates to **`true`**. If a tree containing strings satisfies the BST invariant, then the strings will be stored in alphabetical (dictionary) order.

We draw `sized_tree`s by including both the value and size at each node, separated by a comma. For example, one SBST is

```
s1 =        ("d", 3)
          /        \
     ("b",1)     ("e",1)
```

and can be expressed in OCaml as

```
let s1 = SNode (SNode (SEmpty, ("b",1), SEmpty),
                ("d", 3),
                SNode (SEmpty, ("e",1), SEmpty))
```

Which of the following trees are SBSTs?

☐ violates the BST invariant
☒ violates the size invariant
☐ is a SBST

```
s2 =        ("d", 4)
          /        \
     ("b",1)     ("e",1)
```

☒ violates the BST invariant
☐ violates the size invariant
☐ is a SBST

```
s3 =        ("d", 4)
          /        \
     ("b",2)     ("e",1)
          \
        ("f",1)
```

☐ violates the BST invariant
☒ violates the size invariant
☐ is a SBST

```
s4 =        ("d", 3)
          /        \
     ("b",1)     ("e",1)
                    \
                  ("f", 1)
```

☐ violates the BST invariant
☐ violates the size invariant
☒ is a SBST

```
s5 =        ("d", 4)
          /        \
     ("b",1)     ("e",2)
                     \
                   ("f", 1)
```

(c) (8 points) Recall from HW3 that a tree is *perfect* when

- every leaf is the same distance from the root
- every node has either 0 or 2 children.

If a tree satisfies the size invariant, there is a simple way to determine if it is a perfect tree, that does not require calculating the distance of each leaf to the root. Complete the following function, that does so. Your answer must use the `size` function defined above and may assume that the input tree satisfies the size invariant.

```
let rec is_perfect (t : 'a sized_tree) : bool =
  begin match t with
  | SEmpty -> true
  | SNode (lt, (_, _), rt) ->
     size lt = size rt && is_perfect lt && is_perfect rt
  end
```

(d) (12 points) The insertion function for SBSTs must maintain both the size invariant and the BST invariant. However, the following definition of insert is **incorrect**.

```
let rec bad_insert (t:'a sized_tree) (n:'a) : 'a sized_tree =
  begin match t with
    | SEmpty -> SNode(SEmpty, (n, 1), SEmpty)
    | SNode(lt, (x, sz), rt) ->
      if x = n then t
      else if n < x then
         SNode (bad_insert lt n, (x, sz + 1), rt)
      else
         SNode(lt, (x, sz + 1), bad_insert rt n)
  end
```

First, complete a test case that demonstrates when this function returns an incorrect tree compared to `good_insert`, the *correct* version of the function (not provided). The first blank should be a string that produces different results for `bad_insert` and `good_insert`. The second and third blanks should be the trees that result from these insertions, and one of (`s1`)-(`s5`), defined in part (b) and repeated in the appendix.

```
;; run_test "bad_insert fails" (fun () ->
     let str  : string = "e" in
     let bad  = bad_insert s1 str in
     let good = good_insert s1 str in
     not (bad = good) &&
     bad = s2  && good = s1 )
```

Now, complete a test case that shows that `bad_insert` sometimes works. Fill in the string to insert and the resulting tree, which should be one of the trees (`s1`)-(`s5`).

```
;; run_test "bad_insert works" (fun () ->
     let str : string = "d" in
     let bad = bad_insert s1 str in
     let good = good_insert s1 str in
     bad = good && bad = s1 )

;; run_test "bad_insert works 2" (fun () ->
     let str : string = "f" in
     let bad = bad_insert s1 str in
     let good = good_insert s1 str in
     bad = good && bad = s5 )
```

4. **Abstract Data Types and Higher-Order Functions** (34 points total)

   At ACME, there are employees whose job it is to shop for items that have been ordered online. For each *order*, these shoppers need to know the name of the *customer*, the *items* that the customer wants, and the *priority* of the order (some orders are rush jobs!). We'd like to keep track of this information for the shoppers in a *todo list* and support the following operations (among others).

   - There is an `empty` todo list.
   - Orders can be added to the todo list via the `add_order` function.
   - The shopper can access the next items to shop for and remove them from the todo list via `next_items`.

   For clarity in the code, we will define the following type abbreviations, and use a tuple of a *priority*, *customers*, and their *item list.* to represent an *order*.

```
type priority = int       (* higher is better *)
type customer = string    (* name of the customer *)
type item = string        (* name of a grocery item *)
```

   For example, we can create orders for Maddie and Julia as below.

```
  let order1 = (1, "Julia", ["Apple"; "Banana"; "Pear"])

  let order2 = (2, "Maddie", ["Turkey"; "Chicken"; "Beans"])
```

   We can then construct a todo list using the operations `add_order` and `empty` described above.

```
  let list1 = add_order order1 (add_order order2 empty)
```

   Finally, we can define a test case that demonstrates that because Maddie's order has higher priority than Julia's, her items should be shopped for next.

   the behavior of `next_items`.

```
  ;; run_test "next_items" (fun () ->
    let (items, _) = next_items list1 in
    items = ["Turkey"; "Chicken"; "Beans"])
```

   (There is nothing to do on this page.)

(a) We plan to represent a *todo list* using a list of orders, with the following representation invariant:

- *The orders are sorted by priority in the todo list, with the highest priority first.*
- *A customer may have multiple orders in the todo list, but no two of their orders can have the same priority.*

There may be multiple orders in the list with the same priority, as long as they are for different customers.

To safely maintain these invariants, we will use an *abstract data type*. This question asks you about various options for the interface of this abstract type. We can characterize these possible designs as:

- **Unusable**: lacking functionality: no *client* code could usefully call functions of the interface to achieve a non-trivial result
- **Unsafe**: usable, but that doesn't ensure implementation invariants are preserved: the client can provide inputs that break implementation invariants
- **Good**: usable and able to enforce invariants

For each of the following signatures, mark the box next to the characterization that best describes it. Additionally, if it is *not* "Good", briefly describe why you chose that choice. For example, if a signature is "Unsafe" explain how a client could break the implementation invariant. ***Use each characterization exactly once!***

You may assume that the types `priority`, `customer` and `item` have been defined as on page 9.

(4 points)

```
module type GROCERYORDERS = sig
  type order = priority * customer * item list
  type todo_list
  val empty : todo_list
  val add_order : order -> todo_list -> todo_list
  val next_items: todo_list -> item list * todo_list
end
```

☐ Unusable ☐ Unsafe [✓ Good]

Explanation: *This one is good, so no explanation is required.*

(4 points)

```
module type GROCERYORDERS = sig
  type order = priority * customer * item list
  type todo_list = order list
  val empty : todo_list
  val add_order: order -> todo_list -> todo_list
  val next_items: todo_list -> item list * todo_list
  end
```

☐ Unusable   [✓ ☒ Unsafe]   ☐ Good

Explanation: *Not safe: because the* `todo_list` *is NOT abstract, any list could be passed to* `next_items`*, even those that do not satisfy the representation invariant.*

(4 points)

```
module type GROCERYORDERS = sig
  type order
  type todo_list
  val empty : todo_list
  val add_order: order -> todo_list -> todo_list
  val next_items: todo_list -> item list * todo_list
  end
```

[✓ ☒ Unusable]   ☐ Unsafe   ☐ Good

Explanation: *Unusable: the order type is abstract and there is no way to create an order. So the* `add_order` *function cannot be called.*

Now consider the implementations of operations inside a module that starts with the following type definitions, in addition to the ones shown on page 9.

```
type order = priority * customer * item list
type todo_list = order list
```

(b) (12 points) Complete the following implementation of a function that adds a new order to a todo list. Your function may assume that the todo list satifies the invariants shown on page 10. To maintain these invariants, if a customer already has an order with the same priority in the list, the new items should be appended after their original items. Furthermore, any orders by other customers in the todo list with the same priority should come before the newly added order.

```
let rec add_order (new_order : order) (lst: order list): order list =
  begin match lst with
    | [] -> [ new_order ]
    | (p1, c1, i1)::tl ->
      let (p2, c2, i2) = new_order in
      if (p1 = p2 && c1 = c2) then (p1, c1, i1 @ i2) :: tl
      else if (p1 < p2) then (p2, c2, i2) :: (p1, c1, i1) :: tl
      else (p1, c1, i1)::(add_order new_order tl)
  end
```

Implement the next function using either `transform` or `fold`. Your answer may not be recursive, nor may it call other functions that use list recursion except for the `@` operator.

(c) (10 points) Get all items for a given customer from *all* of their orders. The highest priority items should appear first in the output list. If there is no order for the customer, return an empty list. You may assume that the input todo list satisfies the invariants.

```
let get_items (c1 : customer) (l : todo_list) : item list =

  fold (fun (p, c2, items) out ->
        (if c1 = c2 then items @ out else out)) [] l
```

A Generic Binary Search Trees

```
(* Generic binary trees, from HW 3 *)
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree

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

(* 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
```

B Higher-Order List Processing Functions

Here are the higher-order list processing functions:

```
let rec transform (f : 'a -> 'b) (p : 'a list) : 'b list =
  begin match p with
    | (entry::rest) -> f entry :: transform f rest
    | [] -> []
  end

let rec fold
    (combine: 'b -> 'a -> 'a)
    (base:'a)
    (l : 'b list) :  'a =
  begin match l with
    | [] -> base
    | h :: tl -> combine h (fold combine base tl)
  end
```

C Sized Binary Search Trees

```
type 'a sized_tree =
  | SEmpty
  | SNode of 'a sized_tree * ('a * int) * 'a sized_tree

(* access the size of the tree *)
let size (t:'a sized_tree) : int =
  begin match t with
  | SEmpty -> 0
  | SNode (_,(_,sz),_) -> sz
  end

(* make sure that the size values are correct in the tree  *)
let rec is_sized_tree (t : 'a sized_tree) : bool  =
  begin match t with
  | SEmpty -> true
  | SNode (lt, (v, sz), rt) ->
     sz = size lt + 1 + size rt &&
     is_sized_tree lt && is_sized_tree rt
  end
```

Example sized binary trees, used in question 3.

```
s1 =       ("d", 3)
          /       \
       ("b",1)    ("e",1)


s2 =       ("d", 4)
          /       \
       ("b",1)    ("e",1)


s3 =       ("d", 4)
          /       \
       ("b",2)    ("e",1)
          \
           ("f",1)


s4 =       ("d", 3)
          /       \
       ("b",1)    ("e",1)
                    \
                    ("f", 1)


s5 =       ("d", 4)
          /       \
       ("b",1)    ("e",2)
                     \
                     ("f", 1)
```