CIS 1200 Midterm I February 10th, 2023

SOLUTIONS

1. **OCaml Programming and List Recursion** (20 points total)

   (12 points) For each program below, fill in the blank with the value computed for `ans` by running the following expressions (all of which are well-typed).

(a)
```
let ans : int =
  let x = 3 + 5 in
  let x = 4 + x in x
```

   `ans` = 12

(b)
```
let rec foo (l:int list) : int =
  begin match l with
    | [] -> 0
    | x::xs -> 1 + x + (foo xs)
  end
let ans : int = foo [1;2;3]
```

   `ans` = 9

(c)
```
let rec foo (l:int list) : int list =
  begin match l with
    | [] -> [0]
    | x::xs -> 0::foo xs
  end
let ans : int list = foo [1;2;3]
```

   `ans` = `[0;0;0;0]`

(d) (8 points) Implement a function that uses list recursion to count the number of occurrences of the given argument `n` that appear in the list `l`. We have given you some test cases to demonstrate the desired behavior; a correct implementation will pass them all.

Answer:

```
let rec count_of (n:int) (l:int list) : int =
  begin match l with
    | [] -> 0
    | x::xs -> (if x = n then 1 else 0) + (count_of n xs)
  end

let test () = count_of 3 [] = 0
;; run_test "count_of 3 [] = 0" test

let test () = count_of 3 [3] = 1
;; run_test "count_of 3 [3] = 1" test

let test () = count_of 3 [1;2;3;3;2;3] = 3
;; run_test "count_of 3 [1;2;3;3;2;3] = 3" test
```

2. **Higher-order Functions and Type Checking** (15 points)

   For your reference, Appendix A contains the code for the standard `transform` and `fold` higher order functions from lecture, which you should consider to be in scope for this problem. Consider the following definitions also to be in scope:

```
let sum (x:int) (y:int) = x + y
let mul (x:int) (y:int) = x * y
let max (x:int) (y:int) = if x > y then x else y
let min (x:int) (y:int) = if x > y then y else x

let fs = [sum; mul; max; min]

let rec apply_list (fs:('a -> 'b) list) (l:'a list) : 'b list =
  begin match fs,l with
    | g::gs,x::xs -> (g x)::(apply_list gs xs)
    | _,_ -> []
  end
```

   (3 points each) Indicate the *type* of each expression below (they are all well typed).

   (a) `[1; 2; 3]`

   ☐ `int` [✓ `int list`] ☐ `(int -> int) list`

   ☐ `(int -> int -> int) list` ☐ `((int -> int) -> int) list`

   (b) `fs`

   ☐ `int` ☐ `int list` ☐ `(int -> int) list`

   [✓ `(int -> int -> int) list`] ☐ `((int -> int) -> int) list`

   (c) `fold max 0 [2;4;1;3]`

   [✓ `int`] ☐ `int list` ☐ `(int -> int) list`

   ☐ `(int -> int -> int) list` ☐ `((int -> int) -> int) list`

   (d) `transform sum [1;2;3;4]`

   ☐ `int` ☐ `int list` [✓ `(int -> int) list`]

   ☐ `(int -> int -> int) list` ☐ `((int -> int) -> int) list`

   (e) `apply_list (apply_list fs [1;2;3;4]) [5;6;7;8]`

   ☐ `int` [✓ `int list`] ☐ `(int -> int) list`

   ☐ `(int -> int -> int) list` ☐ `((int -> int) -> int) list`

3. **Higher-order Functions** (16 points)

(4 points each) For each of the following list-processing functions indicate how it might be implemented using `fold` or `transform`. In each case, choose one option.

(a)
```
let rec list_max (l:int list) : int =
  begin match l with
    | [] -> failwith "no max"
    | x::[] -> x
    | x::xs -> max x (list_max xs)
  end
```

- ☐ must be implemented using `fold`
- ☐ can be implemented using `transform` (and so also `fold`)
- [✓ ☒] cannot be implemented using either `transform` or `fold`

(b)
```
let rec list_str (l:int list) : string list =
  begin match l with
    | [] -> []
    | x::xs -> (string_of_int x)::(list_str xs)
  end
```

- ☐ must be implemented using `fold`
- [✓ ☒] can be implemented using `transform` (and so also `fold`)
- ☐ cannot be implemented using either `transform` or `fold`

(c)
```
let rec list_mul (l:int list) : int =
  begin match l with
    | [] -> 1
    | x::xs -> x * (list_mul xs)
  end
```

- [✓ ☒] must be implemented using `fold`
- ☐ can be implemented using `transform` (and so also `fold`)
- ☐ cannot be implemented using either `transform` or `fold`

(d)
```
let rec zip (l1 : 'a list) (l2 : 'b list) : ('a * 'b) list =
  begin match (l1, l2) with
    | (x::xs, y::ys) -> (x,y)::zip xs ys
    | (_, _)  -> []
  end
```

- [✓ ☒] must be implemented using `fold`
- ☐ can be implemented using `transform` (and so also `fold`)
- ☐ cannot be implemented using either `transform` or `fold`

4. **Abstract Types: Ordered Multisets**

In this series of questions we use the design process to implement an abstract type called an *ordered multiset* (`OMSET`). An ordered multiset is a collection of data elements, such as strings or integers. Unlike the `SET` abstract type from Homework 3, an `OMSET` may contain multiple occurrences of the same element and the elements are sorted (ascending) sequentially.

Step 1: Understand the problem As an example, suppose we wanted to do a statistical analysis of the scores for CIS 1200 exams. A small part of that data set might be given by the list: `[72; 85; 85; 85; 93; 93; 99]`

While a list is one way to represent exam scores, it may not be the most efficient for some purposes. For instance, when working with such data, it might be convenient to get the `count` of a given score, which is the number of times it occurs in the collection. So in the data above, the `count` of `85` is `3`, the `count` of `99` is `1`, and the count of any value, like `17`, not in the list, is `0`. Calculating the `count` using a list representation can take time proportional to the length of the list, so we would like to do better.

We may also want to efficiently determine the `size` of the collection as a whole. In the list representation, the `size` is just the length of the list, but computing that also takes time proportional to the length.

We also want to efficiently compute the `nth` element of the data set (indexed from `0`). The element at index `0` in the data above is `72`; index `3` is `85`, while the `99` has index `6`. With a list representation, `nth i` takes time proportional to the index `i`, but we can do better when we take into account repeats. Note that `nth i` fails when `i` is greater than or equal to the size of the data set.

Finally, we need to be able to construct such data sets. There is an `empty` ordered multiset and, we can add several occurrences of a value using a "bulk" `add` operation. `add x amt m` increases the `count` of element `x` by some integer `amt` (where we assume `amt > 0`). This would let us add a whole bunch of (duplicate) exam scores simultaneously.

Step 2: Design the interface

These considerations lead us to the following module signature (a.k.a. interface) of operations for the abstract type `OMSET`.

```
module type OMSET = sig
  type 'a omset

  val empty : 'a omset
  val size : 'a omset -> int
  val count : 'a -> 'a omset -> int
  val add : 'a -> int -> 'a omset -> 'a omset
  val nth : 'a omset -> int -> 'a
end
```

There is nothing to do for Steps 1 and 2. You will demonstrate your understanding in the following parts.

Step 3: Write Test Cases (22 points total)

a. (8 points) Which of the following would create an `int omset` value suitable for representing the example data set: `[72; 85; 85; 85; 93; 93; 99]`? (mark all that apply)

☐

```
let m = [72; 85; 85; 85; 93; 93; 99]
```

☒

```
let m = add 72 1 (add 85 3 (add 93 2 (add 99 1 empty)))
```

☒

```
let m = fold (fun x acc -> add x 1 acc) empty [72;85;85;85;93;93;99]
```

☐

```
let m = let m = empty in
  nth m 0 = 72 && nth m 1 = 85 && nth m 2 = 85 && nth m 3 = 85 &&
  nth m 4 = 93 && nth m 5 = 93 && nth m 6 = 99
```

Recall that for abstract types, we write *property-based* tests.

b. (10 points) Let `m` be an `'a omset`, `x` and `y` be values of type `'a`, and `amt` be an `int` amount greater than `0`. Which of the following properties characterize the type `OMSET` as described above? (mark all that apply)

☒ `count x empty = 0`

☒ `size (add x amt m) = amt + size m`

☒ `count x (add x amt m) = amt + (count x m)`

☐ if `count x m = count y m` then `x = y`

☒ if `x <> y` (i.e., they are not equal), then `count x (add y amt m) = count x m`

c. (4 points) As mentioned above, the `nth m i` operation should *fail* if the index `i` is larger than or equal to the `m`'s `size`. Which of the following test cases would confirm that behavior? (choose one)

☐

```
let test () =
  let m = add 1 3 empty in
  not (nth m (size m) = 17)
;; run_test "nth fails" test
```

☒

```
let test () =
  let m = add 1 3 empty in
  nth m (size m) = 17
;; run_failing_test "nth fails" test
```

☐

```
let test () =
  let m = add 1 3 empty in
  let i = (size m) + 1 in
   i >= (size m)
;; run_test "nth fails" test
```

☐

```
let test () =
  let m = add 1 3 empty in
  let i = (size m) + 1 in
   i >= (size m)
;; run_failing_test "nth fails" test
```

Step 4: Implement the Code (39 points total)

To implement an `OMSET`, we need to fulfill the requirements of its interface. Similar to HW 3, we will use a variant of *binary search trees* (BSTs) for that purpose. We cannot use BSTs directly, though, because they cannot store multiple copies of the same element, which is required by the `OMSET` abstraction. We therefore use trees with a different structure and invariant more suited to this application. We call this a "BST + Size" or BST+S, for short.

Unlike an ordinary BST, whose nodes carry just data values sorted in a particular way, a BST+S tree node carries a pair of a *value* and the *size* of the collection rooted at that subtree. The value parts of the tree follow the usual BST invariants, but the size is maintained separately. For example, recall the list of elements `[72; 85; 85; 85; 93; 93; 99]` from earlier. One way to represent that same information using a BST+S is shown on the left below. For comparison, a BST without size information but with the same data values is shown to the right. (As usual, we omit the `Empty` constructors from these pictures; they are also shown in Appendix B.)

```
     example  BST+S                BST (without size)
        (85, 7)                           85
       /       \                         /  \
(72, 1)       (99, 3)                  72    99
              /                              /
         (93, 2)                           93
```

Each node of a BST+S has size information. For the leaf nodes such as `(72, 1)` and `(93, 2)`, that size information is just the count: `72` occurs once and `93` occurs twice. For interior nodes, the size is the *total* of the size of the subtrees (which is just the size at their roots) plus the count of the data at that node. For example, because `85` occurs three times in the data, the size at its node is 7: 3 (count of 85) + 1 (size of the left subtrees) + 3 (size of the right subtree)—this accounts for all 7 elements of the data. Following this invariant, the size at the root node is the number of elements in the whole collection. We can thus implement the `size` operation required for an `OMSET` like this:

```
let size (t: ('a * int) tree) : int =
  begin match t with
    | Empty -> 0
    | Node(_, (_, s), _) -> s
  end
```

Putting all of that together, we end up with the BST+S Invariant, as spelled out in Appendix B (which also repeats the definition of `size` for your reference).

a. (6 points) Which of the following are correct BST+S trees that represent the *same* data set as illustrated by the `example` tree? (mark all that apply)

```
☒                                ☒                                ☐
     (85, 7)                          (93, 7)                          (85, 7)
    /       \                        /       \                        /       \
(72, 1)     (93, 3)              (85, 4)     (99, 1)              (72, 3)       (99, 1)
               \                 /                                   \
               (99, 1)      (72, 1)                                  (93, 2)
```

b. (4 points) Suppose we were to add the value `100` with a count of `5` to the `example` BST+S. If our implementation of `add` follows the usual strategy for BST `insert` with respect to the `value` (see Appendix C) but also maintains the BST+S invariants, which of the following will be the resulting BST+S? (choose one)

☐
```
       (85, 7)
      /       \
 (72, 1)      (99, 3)
              /      \
       (93, 2)       (100, 5)
```

☐
```
       (85, 12)
      /        \
 (72, 1)       (100, 8)
               /
          (99, 3)
          /
     (93, 2)
```

[✓]
```
       (85, 12)
      /        \
 (72, 1)        (99, 8)
                /      \
         (93, 2)       (100, 5)
```

☐
```
       (85, 7)
      /       \
 (72, 1)      (100, 5)
              /
         (99, 3)
         /
    (93, 2)
```

c. (15 points) The code for the `add` operation for a BST+S follows a pattern similar to the usual BST `insert` (shown in Appendix C), but must additionally maintain the size information part of the invariant. Fill in the blanks below to complete this implementation. Assume that `t` satisfies the BST+S invariants and that `count > 0`.

Answer:

```
let rec add (n:'a) (count:int) (t:('a * int) tree) : ('a * int) tree =
  begin match t with
    | Empty -> Node(Empty, (n,count), Empty)
    | Node(lt, (x, s), rt) ->
      if n = x then
        Node(lt, (x, s+count), rt)
      else if n < x then
        Node(add n count lt, (x, s+count), rt)
      else
        Node(lt, (x, s+count), add n count rt)
  end
```

d. (8 points) To get the `count` of a value stored in a BST+S, we need to do a bit of computation. For instance, in our `example` tree, the `count 85 example` is `3`, but we arrive at that answer by calculating it from the size information stored in the children: `7 - (1 + 3)`, as depicted (with suggestive formatting) below:

```
              example  BST+S        count 85 example = 3 =
                (85, 7)                         7
               /       \                        -
          (72, 1)      (99, 3)              (1  +   3)
                       /
                 (93, 2)
```

That leads us to the following code, in which `value_count` is a helper that computes the count of value at the root of a BST+S tree.

```
let value_count (t:('a * int) tree) : int =
  begin match t with
    | Empty -> 0
    | Node(lt, (_, s), rt) -> s - ((size lt) + (size rt))
  end

let rec count (n:'a) (t:('a * int) tree)  : int =
  begin match t with
    | Empty -> 0
    | Node(lt, (x, _), rt) ->
      if x = n then value_count t
      else if n < x then count n lt
      else count n rt
  end
```

Which of the following are true statements about the code above? (mark all that apply)

(a) True ☐ [✓ False ☒]
If `t` satisfies the BST+S invariants, then `count n t` will visit exactly `size t` nodes (where `size t` is defined earlier).

(b) True ☐ [✓ False ☒]
If `t` *does not* satisfy the BST+S invariants, then `count n t` will visit all of the nodes in the tree before returning an answer.

(c) True ☐ [✓ False ☒]
Running `value_count t` takes time proportional to the number of nodes in the *height* of the tree `t`.

(d) True ☐ [✓ False ☒]
If we change the type annotation on the argument `t` of `value_count` from `('a * int) tree` to instead be `int tree`, the program would still typecheck.

e. (6 points) Finally, we can implement the `nth` operation, which uses the `size` information stored at each node to efficiently index into a BST+S structure.

To see how it works, recall that in a BST+S the count of the value `x` stored at a node `Node(lt, (x, s), rt)` is equal to `s - ((size lt) + (size rt))`. (This is the definition of the `value_count` function above.) That means that if we want to find the element at index `i` we can decide whether it is in `lt`, one of the copies of `x` or in `rt` by doing arithmetic and comparing with `i`. For example, if `(size lt) <= i` then the `i`th element we are looking for must not be in the left subtree.

The code for `nth` below has holes marked by `(A)`, `(B)` and `(C)`.

```
let rec nth (t:('a * int) tree) (i:int) : 'a =
  begin match t with
    | Empty -> failwith "no such element"
    | Node(lt, (x, s), rt) ->
      let lt_size = size lt in
      let rt_size = size rt in

      if i < ___(A)_____ then nth lt i

      else if i < ____(B)____ then x

      else nth rt ____(C)____
  end
```

Match each hole with the correct OCaml expression such to complete `nth`. Choose your answers from among the options 1 through 5. Two of the options will not be used. (Note that there may be more than one way to write the correct arithmetic expression, but only version is listed below—these options are all different.)

____(A)____ = [✓ 1] 2 ☐ 3 ☐ 4 ☐ 5 ☐

____(B)____ = 1 ☐ 2 ☐ [✓ 3] 4 ☐ 5 ☐

____(C)____ = 1 ☐ 2 ☐ 3 ☐ [✓ 4] 5 ☐

OPTIONS:

1. `lt_size`
2. `(rt_size + lt_size)`
3. `(s - rt_size)`
4. `(i - (s - rt_size))`
5. `(rt_size - i)`

Step 5: Modularity and Abstraction - Using the OMSET (8 points)

Suppose we package the code developed above into a module implementing the `OMSET` interface as follows (where we repeat the definition of `OMSET` from before and omit the code definitions developed previously—you can assume they are implemented correctly following the BST+S invariants.)

```
;; open Trees

module type OMSET = sig
  type 'a omset

  val empty : 'a omset
  val size : 'a omset -> int
  val count : 'a -> 'a omset -> int
  val add : 'a -> int -> 'a omset -> 'a omset
  val nth : 'a omset -> int -> 'a
end

module BSTSOmset : OMSET = struct
  type 'a omset = ('a * int) tree

  let empty = Empty
  let size (t : ('a * int) tree) : int =  (* omitted *)
  let value_count (t:('a * int) tree) : int = (* omitted *)
  let count (n:'a) (t:('a * int) tree) : int = (* omitted *)
  let add (n:'a) (count:int) (t:('a * int) tree) : ('a * int) tree = (* omitted *)
  let nth (t:('a * int) tree) (i:int) : 'a = (* omitted *)

  (* __(A)__ *)
end

;; open BSTSOmset

(* __(B)__ *)
```

(a) [✓ True ☒] False ☐
If we place the code `let ans : int = count 3 empty` at the point marked `(A)` above, the resulting program will typecheck.

(b) [✓ True ☒] False ☐
If we place the code `let ans : int = count 3 empty` at the point marked `(B)` above, the resulting program will typecheck.

(c) [✓ True ☒] False ☐
If we place the code `let ans : int = value_count empty` at the point marked `(A)` above, the resulting program will typecheck.

(d) True ☐ [✓ False ☒]
If we place the code `let ans : int = value_count empty` at the point marked `(B)` above, the resulting program will typecheck.

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) (xs: 'a list): 'b list =
  begin match xs with
  | [] -> []
  | h::tl -> f h :: transform f tl
  end

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

Appendix B: Binary Search Tree + Size (BST+S)

BST+S Invariant A tree `t : ('a * int) tree` satisfies the BST+S invariant if:

- `t` is `Empty`, or
- `t` is `Node(lt, (x, s), rt)` and:
  - Every value in `lt` is less than `x`
  - Every value in `rt` is greater than `x`
  - `s > (size lt) + (size rt)`
  - both `lt` and `rt` (recursively) satisfy the BST+S invariant

An example BST+S for the data set `[72; 85; 85; 85; 93; 93; 99]` (left) and the BST structure of the values in the tree (right).

```
        example  BST+S                BST (without size)
           (85, 7)                          85
          /       \                        /  \
     (72, 1)       (99, 3)               72    99
                   /                            /
             (93, 2)                          93
```

Following the invariant above, the size of a BST+S tree `t` is:

```
let size (t: ('a * int) tree) : int =
  begin match t with
    | Empty -> 0
    | Node(_, (_, s), _) -> s
  end
```

Appendix C: Generic 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
```