CIS 120 Midterm I   February 9, 2018

SOLUTIONS

1. **Program design**

For this problem, you will use the program design process to implement a function called `shortest`. This function should find and return the shortest string in a nonempty list of strings. If multiple strings have the same shortest length, this function should return the first one that appears in the list. If this function is given an empty list, then it should fail (using `failwith`).

a. (2 points)

First, write the type of the `shortest` function, as you might see in a signature or `mli` file.

val shortest : ______string list -> string_____

b. (8 points)

Which of these tests agree with your understanding of the `shortest` function from the problem description? Check YES for tests that capture this specification and NO for tests that are buggy. Recall that `run_failing_test` succeeds when the provided test fails.

i.
```
let test () : bool =
  shortest [] = "Fly, Eagles, Fly"
;; run_failing_test "shortest [] fails" test
```

[✓ YES] ☐ NO

ii.
```
let test () : bool =
  shortest [] = ""
;; run_test "shortest []" test
```

☐ YES [✓ NO]

iii.
```
let test () : bool =
  shortest [""] = "Fly, Eagles, Fly"
;; run_failing_test "shortest empty string fails" test
```

☐ YES [✓ NO]

In this problem, `run_failing_test` *succeeds whenever the testing function fails (with* `failwith`*). However, in a correct implementation of* `shortest`*, the testing function should return* **`false`**. *The comment also incorrectly indicates that a failure is expected.*

iv.
```
let test () : bool =
  shortest ["120"; "a"; "b"] = "a"
;; run_test "shortest is first occurring" test
```

[✓ YES] ☐ NO

c. (16 points) The implementation of `shortest` is a single line that defers to a helper function, called `shortest_aux`, that does all the work. It will be your job to implement `shortest_aux`.

```
let rec shortest ss = fst (shortest_aux ss)
```

Note, in the implementation above we have deliberately left off the type annotations. Furthermore, the `fst` function accesses the first component of a pair. For example, `fst (1,2) = 1`.

Let's restart the program design process for `shortest_aux`. The interface of this function indicates that it returns a pair of results.

```
val shortest_aux : string list -> string * int
```

The test cases for this function, such as the one shown below, show that the second component of the result should be the length of the shortest string.

```
let test () : bool =
  shortest_aux ["CIS"; "is"; "cool"] = ("is", 2)
;; run_test "shortest_aux returns shortest string and its length" test
```

The implementation of `shortest_aux` is on the next page.

(There is nothing to answer on this page.)

Fill in the blanks to complete the implementation of `shortest_aux`.

- You can use the OCaml library function `String.length` to calculate the length of a `string`. (We have done this for you in one of the cases.)
- Your implementation should calculate the length of each string in the list *only once* as it iterates through the list. We will take the efficiency of your solution into account when grading this problem.
- If you wish, you may use `fst` and `snd` to access the components of a pair.
- You may **not** use any other auxiliary functions in your implementation.
- The blanks below are a hint to the size of our solution. If you need more space you may use the scratch page, but be sure to mark the location of your code clearly.

```
let rec shortest_aux (ss : string list) : string * int =
 begin match ss with
   | [] -> failwith "need at least one"
   | __[hd]__ -> ___ (hd, String.length hd) ___
   | (hd :: tl) ->
     let len = String.length hd in
     __let p = shortest_aux tl in __________
     __if (len <= snd p) ______________________________
     __then (hd, len) ________________________________
     __else p     ________________________________
 end
```

Note, there are several answers that pass the tests but do not satisfy the efficiency requirements. For example, this version calls `String.length` twice as much as necessary.

```
let rec shortest_aux (ss : string list) : string * int =
 begin match ss with
   | [] -> failwith "need at least one"
   | __[hd]__ -> ___ (hd, String.length hd) ___
   | (hd :: tl) ->
     let len = String.length hd in
     match tl with
     | h2 :: tl2 ->
     __if (String.length hd <= String.length h2)
     __then shortest_aux (hd :: tl)
     __else shortest_aux (hd2 :: tl)
 end
```

Even worse, this one recalculates the result of the recursive call twice, making this function particularly inefficient.

```
let rec shortest_aux (ss : string list) : string * int =
 begin match ss with
   | [] -> failwith "need at least one"
   | __[hd]__ -> ___ (hd, String.length hd) ___
```

```
  | (hd :: tl) ->
    let len = String.length hd in
    __if (len <= snd (shortest_aux tl)) ____________
    __then (hd, len) ______________________________
    __else (shortest_aux tl) ______________________
end
```

2. List recursion and Higher-order Functions (18 points)

The `clean` function processes a list of data readings in preparation for further analysis. In particular, this function should, when given a list of numbers, filter out the negative numbers and terminate the output list at the first occurrence of `-999` (if present).

For example, `clean [1;-1;3;-999;43;92]` should return `[1;3]`.

a. Define `clean` using a recursive function. In your answer below, you may *not* use any library or helper functions such as `@` or `fold`. Constructors (such as `::` and `[]`) and operators (such as `<` and `=`) are fine.

```
let rec clean (l : int list) : int list =
  begin match l with
  | [] -> []
  | x :: tl -> if x = -999 then []
            else if x < 0 then clean tl
            else x :: clean tl
  end
```

b. Now recall the higher-order list processing function, `fold`:

```
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
```

Redefine the `clean` function using `fold`. Your answer may **not** be recursive. You also may not use any other library functions such as `@` or `transform`. As before, constructors (such as `::` and `[]`) and operators (such as `<` and `=`) are fine.

```
let clean (l : int list) : int list =
  fold (fun x acc ->
        if x = -999 then []
        else if x < 0 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 -> int set________ = add 3


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


let f : _______________int set set _______________ = set_of_list [add 3 empty]


let g : ______________int set list ______________ = [add 3 empty; add 4 empty]


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

4. Generic operations (10 points)

In the homework, you have seen that the `<` operator can be used to order `int`s and `string`s. However, in OCaml, this operator is generic — it works for two arguments of *any* type, as long as the two arguments have the same type.

This problem requires you to understand the behavior of the `<` operator on *pairs* from the description given below. OCaml uses *lexicographic* ordering for pairs, which means that `<` compares the components of the pairs left-to-right, in the same way as we might sort words in a dictionary.

Definition (Lexicographic ordering): the pair `(x1,x2)` *is less than the pair* `(y1,y2)` *when* `x1 < y1` *or* `x1 = y1 && x2 < y2`

Check whether each expression returns **`true`** or **`false`**, or is *ill-typed*.

a. `(1,0) < (0,1)`

☐ **`true`** [✓ ☒ **`false`**] ☐ *ill-typed*

b. `(0,1) < (0,2)`

[✓ ☒ **`true`**] ☐ **`false`** ☐ *ill-typed*

c. `(3,5) < (3,1)`

☐ **`true`** [✓ ☒ **`false`**] ☐ *ill-typed*

d. `(3,3) < (3,(3,1))`

☐ **`true`** ☐ **`false`** [✓ ☒ *ill-typed*]

e. `((1,5),4) < ((1,6),4)`

[✓ ☒ **`true`**] ☐ **`false`** ☐ *ill-typed*

Binary Search Trees and Invariants

Now consider an implementation of the `SYMREL` that uses *binary search trees* to store pairs of associated elements. For reference, a generic implementation of binary search trees appears in Appendix A.

```
type 'a symrel = ('a * 'a) tree
```

To make sure that the relation is symmetric, when we `add` associations into the tree, we will insert both orderings of the pair.

```
let add (x:'a) (y:'a) (r:'a symrel) : 'a symrel =
   insert (insert r (x,y)) (y,x)
```

For example, we can draw the relation `r1` = `add 1 3 empty` as the following tree

```
(1,3)
    \
   (3,1)
```

Furthermore, we can draw the BST corresponding to

```
r3 = add 2 3 (add 1 0 (add 1 2 empty))
```

as

```
       (1,2)
       /   \
    (1,0) (2,1)
     /       \
  (0,1)     (2,3)
               \
              (3,2)
```

(Note, there is nothing to answer on this page.)

5. (10 points)

In our BST-based implementation, we choose to maintain the following invariant.

INVARIANT: The tree must satisfy the binary search tree invariant. Furthermore, if a pair `(x,y)` *is stored in the tree, then the pair* `(y,x)` *must also be present.*

Do the following `int symrel` values satisfy the invariant? Check YES or NO. You should use the lexicographic ordering from Problem 4 to order the pairs in the tree.

a.

```
    (1,2)
   /    \
(1,0)  (2,1)
```

☐ YES ☒ NO

b.

```
    (1,2)
    /   \
(1,1)  (2,1)
```

☒ YES ☐ NO

c.

```
    (1,2)
    /
(1,2)
```

☐ YES ☒ NO

d.

```
    (1,2)
    /
(2,1)
```

☐ YES ☒ NO

e.

```
   (1,3)
   /   \
(1,2)  (3,1)
       /   \
    (1,4)  (4,1)
       \
      (2,1)
```

☒ YES ☐ NO

6. (20 points total)

Appendix D contains an incomplete implementation of the `SymRel` module using BSTs.

Below, select the correct code to complete blanks `(a)` - `(e)`. Take your time. Although more than one answer may pass the tests shown in Appendix C, you should select the *best* answer. In particular, be sure to exploit and preserve the representation invariant. The code below uses functions defined in Appendix A as well as the append operator for lists (`@`).

a. (3 points)

☒ `Empty` ☐ `(Empty,Empty)` ☐ `[]` ☐ `([],[])`

Only the first answer is of the correct type (`('a * 'a) tree`).

b. (4 points)

☐ `x = y`
☒ `lookup r (x,y)`
☐ `lookup r (x,y) || lookup r (y,x)`
☐ `lookup r (x,y) && lookup r (y,x)`

The instructions indicate that the code should take advantage of the invariant on the previous page. This invariant states that if `(x,y)` is present then `(y,x)` will also be present. Therefore, the correct answer need only use `lookup` with one of these pairs.

c. (3 points)

☐ `Empty` ☐ `(Empty,Empty)` ☒ `[]` ☐ `([],[])`

Only the first answer is of the correct type (`'a list`).

d. (5 points)

☐ `[y]`
☐ `get_all x0 lt @ get_all x0 rt`
☐ `[y] @ get_all x0 lt`
☐ `[y] @ get_all x0 lt @ get_all x0 rt`
☒ `get_all x0 lt @ [y] @ get_all x0 rt`

Because of the invariant, the correct solution needs to gather the second component of all pairs where the first component is equal to `x`. In this case, we have found one such pair. However, other pairs may occur in the left and right subtrees.

e. (5 points)

☒
```
if x0 < x then get_all x0 lt
else get_all x0 rt
```

☐
```
if (x0 = y) then [x]
else get_all x0 lt @ get_all x0 rt
```

☐ `get_all x0 lt @ get_all x0 rt`

☐ `get_all x0 lt @ [x0] @ get_all x0 rt`

Again, the correct solution should use the invariant to reduce the search for pairs where *the first component matches* `x`*. Because of the BST invariant and the lexicographic ordering of pairs, we can direct the search into either the left or right subtree.*

Appendix A: (Binary Search) Trees

```
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) ->
     if x = n then true
     else 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

(* An inorder traversal of the tree *)
let rec inorder (t: 'a tree) : 'a list =
  begin match t with
  | Empty -> []
  | Node (lt, x, rt) -> inorder lt @ [x] @ inorder rt
  end
```

Appendix B: `SET` Module Signature

The signature below defines a simplified version of 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
```

Appendix C: `SYMREL` Abstract Type

A *symmetric relation* is a data structure that keeps track of associations between pairs of data values using the following interface.

```
module type SYMREL = sig

  (* A type recording relations between values of some generic type *)
  type 'a symrel

  (* an empty relation *)
  val empty : 'a symrel

  (* add a new association to the relation *)
  val add : 'a -> 'a -> 'a symrel -> 'a symrel

  (* are two elements associated by the relation ? *)
  val related : 'a -> 'a -> 'a symrel -> bool

  (* return all values related to the given value, in sorted order *)
  val get_all : 'a -> 'a symrel -> 'a list

end
```

To help you understand the the specification of this abstract type, we have stated some its properties and have defined test cases that demonstrate the expected behavior.

Property: If a pair of values has been added, the `related` *operation returns true.*

```
;; open SymRel
let r1 : int symrel = add 1 3 empty

let test () : bool = related 1 3 r1
;; run_test "added values are related" test
```

Property: A symmetric relation associates values symmetrically. i.e. the order of the arguments does not matter when they are added or queried in the relation.

```
;; open SymRel
let r2 : int symrel = add 3 1 empty

let test () : bool = related 1 3 r2
;; run_test "relation is symmetric" test
```

Property: Only associated values are related.

```
;; open SymRel
let r3 : int symrel = add 2 4 (add 1 3 empty)

let test () : bool = not (related 1 4 r3)
;; run_test "added values are related" test
```

Property: The `get_all` *operation returns all elements related to a specified element, in sorted order.*

```
;; open SymRel
let r4 : int symrel = add 2 3 (add 1 1 (add 1 2 empty))

let test () : bool =
  get_all 1 r4 = [1;2]
;; run_test "1 is related to itself and 2" test

let test () : bool =
  get_all 2 r4 = [1;3]
;; run_test "2 is related to 1 and 3, result is sorted" test

let test () : bool =
  get_all 4 r4 = []
;; run_Test "unrelated elements return empty list" test
```

Appendix D: Partial `SymRel` implementation using BSTs

This partial implementation accompanies problem 6. The letters in the blanks below correspond to parts (a) - (e) in the problem.

You may use this page for scratch space, but we will not read or grade any of your notes here.

```
;; open BST (* Definitions from Appendix A *)

module SymRel : SYMREL = struct

  (* A type recording relations between values of some generic type *)
  type 'a symrel = ('a * 'a) tree


  (* an empty relation *)
  let empty = ___________(a)________________

  (* add a new association to the relation *)
  let add (x:'a) (y:'a) (r: 'a symrel) : 'a symrel =
      insert (insert r (x,y)) (y,x)

  (* are two elements associated by the relation ? *)
  let related (x:'a) (y:'a) (r :'a symrel) : bool =


      ___________________(b)__________________________


  (* return all values related to the given value, in sorted order *)
  let rec get_all (x0:'a) (t:'a symrel) : 'a list =

    begin match t with

     | Empty -> ______________(c)_______________

     | Node (lt, (x,y), rt) ->

       if (x0 = x) then

         ___________(d)__________________________

       else

         ___________(e)__________________________

   end
```