CIS 120 Midterm I February 14, 2020

SOLUTIONS

1. **List Recursion** (20 points)

Consider the following mystery function `f`, defined using the help of an equally mysterious function `g`.

```
let rec g (x : int list) : int =
  begin match x with
    | [] -> failwith "must provide a nonempty list"
    | [hd] -> if hd > 0 then hd else 0
    | hd :: tl -> if hd > 0 then hd + g tl else 0
  end

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

Determine the value of the following expressions. Check *one* option below.

(a) `f []`

- ☒ `0`
- ☐ `1`
- ☐ `[]`
- ☐ `[0]`
- ☐ `runtime error: "must provide a nonempty list"`
- ☐ `infinite loop`

(b) `g [-3;3]`

- ☒ `0`
- ☐ `-3`
- ☐ `3`
- ☐ `1`
- ☐ `runtime error: "must provide a nonempty list"`
- ☐ `infinite loop`

(c) `f [-3;2;3;-2;3]`

- ☐ `0`
- ☒ `5`
- ☐ `8`
- ☐ `[2;3]`
- ☐ `[2;3;3]`
- ☐ `runtime error: "must provide a nonempty list"`
- ☐ `infinite loop`

Consider the mystery function `h`, shown below.

```
let rec h (x : int list) : int list =
  begin match x with
    | [] -> []
    | [y] -> [y]
    | hd :: tl -> hd :: 0 :: h tl
  end
```

Determine the value of the following expressions. Check *one* option below.

(d) `h []`

- ☐ `0`
- ☐ `5`
- ☒ `[]`
- ☐ `[0]`
- ☐ `[0;0]`
- ☐ `infinite loop`

(e) `h [0;1;2]`

- ☐ `0`
- ☐ `3`
- ☐ `[0]`
- ☐ `[0;1;0;2]`
- ☐ `[0;1;0;2;0]`
- ☒ `[0;0;1;0;2]`
- ☐ `[0;0;1;0;2;0]`
- ☐ `infinite loop`

2. **Types (18 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`, to the constructors of the type `'a tree`, defined in Appendix A, or to the operations of the `SET` interface, defined in Appendix B. Furthermore, you may also assume that an implementation of the `SET` interface is available as below, and has been opened.

```
module BSTSet : SET = struct
...
end
;; open BSTSet
```

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

```
let z : _______(int list * int) set _________ =
  add ([1], 26) empty



let a : ill-typed =
  begin match z with  (* z is defined above *)
  | Empty -> ([], 0)
  | Node(lt, y, rt) -> y
  end

(* Note: must include two different type variables. *)
let b : ('a tree * 'b set) =
  (Empty, empty)

let c  : ill-typed =
  remove 26 z

let d : int set -> int set=
  add 4

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

let f : int -> int list =
  fun (x:int) -> [x+1;x-1]

let g : int list -> int list =
  let m (x:bool) (y:int list) : int list =
    if x then y else [1] in
  m true

let h : 'a list -> 'a list list =
  fun x -> [x]
```

```
let j : bool list -> int =
  fun x ->
    begin match x with
      | [] -> 1
      | (y::_) -> if y then 2 else 3
    end
```

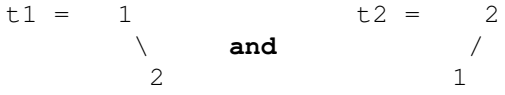
3. **Trees and Binary Search Trees** (11 points)

The next part refers to the following tree datatype.

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

Consider the possible binary search trees containing a given set of elements. For reference, the binary search tree invariant appears in Appendix A.

For example, there are only **two** different binary search trees containing the values `1` and `2`. We can draw them like this (as usual, omitting the `Empty` subtrees).



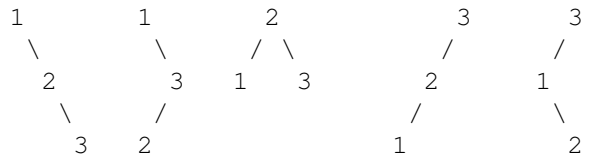
(a) Write code that constructs these two trees in OCaml.

```
let t1 : int tree = Node (Empty, 1, Node (Empty,2,Empty))
let t2 : int tree = Node (Node (Empty, 1, Empty), 2, Empty)
```

Alternative solutions with `insert` *and* `empty` *are also acceptable.*

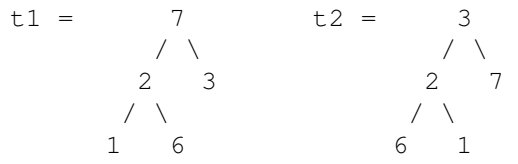
(b) Draw *all* binary search trees containing the values `1`, `2`, and `3`.

Answer:



4. **Tree Shapes** (24 points total)

We say that two trees (not necessarily binary search trees) have the *same shape* if their structure is identical even when the values at corresponding nodes differ. For example, the following two trees, called `t1` and `t2`, have the same shape:



On the next page, you will implement the `same_shape` function such that, with the two trees above, `same_shape t1 t2` evaluates to **`true`**. However, first, answer the following questions.

(a) (3 points) Is it possible to implement `same_shape` so that it can compare two trees that contain different types of elements, such as an `int tree` and a `string tree`?

☒ Yes ☐ No

(b) (3 points) Is it possible to implement `same_shape` so that it can take advantage of the *binary search tree invariant* to produce an answer more efficiently when its inputs are binary search trees?

☐ Yes ☒ No

(c) (3 points) Suppose `t` is a *binary search tree* and `x` is an element that appears in the tree. Is it always the case that `same_shape t (insert (delete t x) x)` evaluates to **`true`**?

☐ Yes ☒ No

(d) (3 points) Suppose `t` is a *binary search tree* and `x` is an element that does not appear in the tree. Is it always the case that `same_shape t (delete (insert t x) x)` evaluates to **`true`**?

☒ Yes ☐ No

(e) (3 points) Suppose `t1` and `t2` are *binary search trees* that contain the same elements and that `same_shape t1 t2` evaluates to **`true`**. Is it always the case that `t1 = t2` evaluates to **`true`**?

☒ Yes ☐ No

(f) (12 points) Write a function that determines whether two different trees (not necessarily binary search trees) have the same shape.

```
let rec same_shape (t1 : 'a tree) (t2: 'b tree) : bool =
  begin match t1 , t2 with
    | Empty, Empty -> true
    | Node (l1, _, r1), Node (l2, _ , r2) ->
      same_shape l1 l2 && same_shape r1 r2
    | _ , _ -> false
  end
```

5. **Programming with Higher-order Functions, Lists and Tuples** (24 points)

Suppose we would like to implement a data structure representing *nonempty lists*. The interface and beginning of an implementation of this data structure is shown in Appendix C. Read the code in the appendix **now** carefully including the examples; your job on this problem is to complete the implementation.

Note this line the implementation:

```
type 'a ne_list = 'a * 'a list
```

This code defines a nonempty list as a tuple of a single element (to store the first element of the nonempty list) and a regular list (to store any remaining elements of the list). All of the code that you will write for this problem may use the fact that nonempty lists are represented as tuples.

> For **all** parts of this problem, you **may not** use recursion, or the example functions `to_list` and `from_list`. Instead, your solutions **may** only use the functions listed in Appendix D, including `fst`, `snd`, `@` (or `append`), `max`, `transform` and `fold`. Constructors, such as `::` and `[]`, are also fine.

(a) Implement `ne_head` so that it passes the following test case.

```
let test () : bool =
  ne_head (from_list [-2;-1;-3]) = -2
```

Answer:

```
(* Access the first element of the non-empty list *)
let ne_head (x : 'a ne_list) : 'a =
  begin match x with
    | (hd,_) -> hd
  end
```

(b) Implement `ne_append` so that it passes the following test case.

```
let test () : bool =
   let ne1 : int ne_list = from_list [1;2]    in
   let ne2 : int ne_list = from_list [4;6;5]  in
   to_list (ne_append ne1 ne2) = [1;2;4;6;5]
```

Answer:

```
(* Append two non-empty lists into a larger non-empty list. *)
let ne_append (x1 : 'a ne_list) (x2 : 'a ne_list) : 'a ne_list =
  begin match x1 , x2 with
    |  (hd1, tl1),  (hd2, tl2) ->  (hd1 , tl1 @ hd2 :: tl2)
  end
```

(c) Use a higher-order function to implement `ne_transform` so that it passes the following test case.

```
let test () : bool =
    let ne1 : int ne_list = from_list [1;2;3] in
    to_list (ne_transform (fun x -> x + 1) ne1) = [2;3;4]
```

Answer:

```
(* Transform each element of a non-empty list *)
let ne_transform (f: 'a -> 'b) (x : 'a ne_list) : 'b ne_list =
  begin match x with
    | (hd, tl) ->   (f hd, transform f tl)
  end
```

(d) Use a higher-order function to implement `ne_maximum` so that it passes the following test case.

```
let test () : bool =
  ne_maximum (from_list [-2;-1;-3]) = -1
```

Answer:

```
(* Find the largest element of a non-empty list *)
let ne_maximum (x : 'a ne_list)  : 'a =
  begin match x with
    |  (hd, tl) -> fold max hd tl
  end
```

(NOTE: reread the box on page 9 to make sure that you have followed the instructions for all parts of this problem.)

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 you may use the back of one of the pages. However, if you do so, you must put a clear note in the exam so that we can find your answer.*

A Binary Search Trees

Tree datatype definition

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

The Binary Search Tree Invariant

- `Empty` is a binary search tree.
- A tree `Node(lt,x,rt)` is a binary search tree if `lt` and `rt` are both binary search trees, and every label of `lt` is less than `x` and every label of `rt` is greater than `x`.

BST operations

```
(* 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
(* end delete *)
```

B SET interface

```
module type SET = sig

  (* The "abstract type" 'a set.

     A set is an _unordered_ collection of _distinct_ elements.

     In mathematics, sets are typically written with curly braces;
     e.g. the set containing four, five, and six would be written {4,
     5, 6} (but note that this is NOT valid OCaml syntax for a set).

     By unordered, we mean that {4, 5, 6} and {5, 6, 4} are equivalent
     mathematically. By distinct, we mean that a set contains any
     particular element only either 0 or 1 times (no duplicates are
     allowed). *)
  type 'a set

  (* The empty set, with no elements. *)
  val empty : 'a set

  (* `is_empty s` is true exactly when s has no elements. *)
  val is_empty : 'a set -> bool

  (* `list_of_set s` returns a list of all the elements of s. This
     list should be sorted in ascending order and may not contain
     duplicate elements. *)
  val list_of_set : 'a set -> 'a list

  (* `add x s` returns a set just like s, except x is now also an
     element. If x is already an element of s, then it just returns
     s. For example:
       add 3 {4, 5, 6} = {3, 4, 5, 6}
       add "a" {"b", "a", "c"} = {"b", "a", "c"} *)
  val add : 'a -> 'a set -> 'a set

  (* `remove x s` returns a set just like s, except x is not an
     element. If x already wasn't an element, then it just returns
     s. *)
  val remove : 'a -> 'a set -> 'a set

  (* `member x s` returns true exactly when x is a member of s. *)
  val member : 'a -> 'a set -> bool

  (* `size s` returns the "cardinality" (number of elements) of s. *)
  val size : 'a set -> int


end
```

C Non-empty list interface and implementation

```
module type NELIST = sig

  type 'a ne_list

  (* (example) Convert a regular list to a non-empty list *)
  val from_list : 'a list -> 'a ne_list

  (* (example) Convert a non-empty list to a regular list *)
  val to_list : 'a ne_list -> 'a list

  (* (a) Access the first element of the non-empty list *)
  val ne_head : 'a ne_list -> 'a

  (* (b) Append two non-empty lists into a larger non-empty list. *)
  val ne_append : 'a ne_list -> 'a ne_list -> 'a ne_list

  (* (c) Transform each element of a non-empty list *)
  val ne_transform : ('a -> 'b) -> 'a ne_list -> 'b ne_list

  (* (d) Find the largest element of a non-empty list *)
  val ne_maximum : 'a ne_list -> 'a
end

module Nelist : NELIST = struct

  (* A non-empty list is a tuple of a single element (to store
     the first element of the nonempty list) and a regular list
     (to store any remaining elements of the list). *)
  type 'a ne_list = 'a * 'a list

  (* (example) Convert a regular list into a non-empty list by
     pulling out its first element.
     This function will fail for empty lists. *)
  let from_list (x : 'a list) : 'a ne_list =
    begin match x with
      | hd :: tl -> (hd, tl)
      | [] -> failwith "Input list must be non-empty"
    end

  (* (example) Convert a non-empty list to a regular list by
     'cons'-ing the first element back on the rest of the list. *)
  let to_list (x : 'a ne_list) : 'a list =
    begin match x with
      | (hd, tl) -> hd :: tl
    end

  (* You need to complete the rest  ... *)
```

D List Processing Functions

Library function declarations

```
(* Concatenate two lists, same as operator @ *)
val append : 'a list -> 'a list -> 'a list

(* Return larger of the two arguments *)
val max : 'a -> 'a -> 'a

(* Return first component of pair *)
val fst : ('a * 'b) -> 'a

(* Return second component of pair *)
val snd : ('a * 'b) -> 'b
```

Higher-order functions

```
let transform (f: 'a -> 'b) (l: 'a list): 'b list =
  fold
    (fun (h:'a) (acc:'b list) -> f h :: acc)
    []
    l

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
```