CIS 120 Midterm I February 15, 2019

SOLUTIONS

1. **Types** (21 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 12, or to the constructors of the type `ourtree`, which defined as:

```
type 'a ourtree =
 | Leaf of 'a * int
 | Node of 'a ourtree * 'a ourtree
```

   We have done the first one for you.

```
let z : string ourtree =
  Leaf("z", 26)

let a : string ourtree =
  Node(z, Node(Leaf("z", 26), z))

(*
let b : ill typed =
  Node(("a", 1), ("b",2))
*)

let c : (string ourtree * bool ourtree) list =
  [(Leaf("3", 3), Leaf(true, 4));
   (Node(z, z), Leaf(false, 5))]

let d : int -> int =
  (fun x -> fun y -> x - 2 * y) 120

(*
let e : ill typed =
  if 3 > 0 then true else "false"
*)

let f : 'a list -> 'a list -> 'a list =
  fun (v : 'a list) ->
    fold (fun x y -> x :: y) v

let g : 'a list -> bool list =
  transform (fun x -> true)
```

2. **List Processing and Higher-Order Functions (44 points)**

(a) The `dedup` function takes a list and returns a list from which duplicated elements have been removed—i.e., where any number of adjacent copies of a single value are replaced by just one copy. For example, `dedup [1;1;2;2;2;2;2;2;1;1;3]` yields `[1;2;1;3]`.

Fill in the blanks to complete the definition of `dedup`:

```
let rec dedup (l: 'a list) : 'a list =
 begin match l with
 | x1::x2::tl -> if x1=x2 then dedup (x2::tl) else x1 :: dedup (x2::tl)
 | _ -> l
 end
```

(b) The function `sorted` checks whether a list is *sorted* — i.e., whether every pair of adjacent elements is correctly ordered according to the <= relation. For example, the lists `[]`, `[1]`, `[1;2;3]`, and `[1;2;2;3]`, are sorted, while `[3;2;1]` is not.

Complete the definition of `sorted`:

```
let rec sorted (l: 'a list) : bool =
 begin match l with
 | x::y::t -> x <= y && sorted (y::t)
 | _ -> true
 end
```

(c) The `diffs` function takes two *sorted* lists and returns a list of their *differences*—i.e., the elements that appear in one list but not the other. For example:

`diffs [1;2;3] [1;3]`	yields	`[2]`
`diffs [1;2;3] [1;3;6;7]`	yields	`[2;6;7]`
`diffs [1;2;3] [1;2;3]`	yields	`[]`
`diffs [1;1;1;2] [1;2;2]`	yields	`[1;1;2]`

You should assume that both arguments to `diffs` are sorted.

Complete the definition of `diffs`:

```
let rec diffs (l1: 'a list) (l2: 'a list) : 'a list =
 begin match l1, l2 with
 | [], _ -> l2
 | _, [] -> l1
 | h1::t1, h2::t2 ->
   if h1 = h2 then diffs t1 t2
   else if h1 < h2 then h1 :: (diffs t1 l2)
   else h2 :: (diffs l1 t2)
 end

;; assert_eq "diffs1" (diffs [1;2;3] [1;3;6;7]) [2;6;7]
;; assert_eq "diffs2" (diffs [] [1]) [1]
;; assert_eq "diffs3" (diffs [1;1;1;2] [1;2;2]) [1;1;2]
```

(d) The function `take_while` takes two arguments — a boolean testing function `f` and a list `l`. It returns a list containing all the elements from the beginning of `l` for which `f` returns `true`, up to (but not including) the first element for which `f` returns `false`, if any. For example, `take_while (fun x -> x > 0) [1;2;-1;-3;4])` yields `[1;2]`.

Show how to write `take_while` nonrecursively, as an instance of `fold`. Note that `fold` takes three arguments, and we've given you three blanks; please use a separate blank for each argument.

```
let take_while (f: 'a -> bool) (l: 'a list): 'a list =
  fold
    (fun (h:'a) (acc:'a list) -> if f h then h :: acc else [])
    []
    l
```

(e) Here is a recursive definition of a function `apply_all`, which takes a list of functions and a single argument and returns a list containing the results of applying each of the functions to this argument.

```
let rec apply_all (l: ('a->'b) list) (x: 'a) : 'b list =
  begin match l with
  | [] -> []
  | f::t -> f x :: apply_all t x
  end
```

Complete the following alternative definition of `apply_all` as an instance of `transform`.

The `transform` function takes two arguments, and we've given you two blanks; please use a separate blank for each argument.

```
let apply_all (l: ('a->'b) list) (v: 'a) : 'b list =
  transform (fun (f:'a->'b) -> f v) l
```

(f) The `subseq` function checks whether its first argument, `sub`, is a *subsequence* of its second argument, `super`, meaning that all the elements of `sub` appear (in the same order, but not necessarily side-by-side) within `super`. For example, `[1;2]` is a subsequence of `[1;3;2]` and `[1;3;1;2;4]`, but not of `[2;1]`.

(Note that the arguments to `subseq` are arbitrary lists—not necessarily sorted.)

Complete the definition of `subseq`:

```
let rec subseq (sub: 'a list) (super: 'a list) : bool =
  begin match sub, super with
    [], _ -> true
  | _, [] -> false
  | hsub::tsub, hsuper::tsuper ->
     if hsub = hsuper then subseq tsub tsuper else subseq sub tsuper
  end
```

3. **Modules and Abstract types (20 points)**

Consider the following module definition

```
module M : MSIG = struct
  type t = int
  let zero : t = 0
  let incr (x : t) : t = x + 1
  let to_int (x: t) : int = x
  let from_int (x : int) : t = x
end
```

and the following invariant that the module designer would like to maintain:

A value of type `M.t` *is never negative.*

Each of the following questions asks you to evaluate whether a proposed signature `MSIG` for `M` is both

- *safe* in the sense that the `to_int` function cannot return a negative number and
- *useful* in the sense that it's possible (after enough calls to other functions in the interface) for a call to `to_int` to return any non-negatative number.

(a)
```
module type MSIG = sig
  type t
  val zero   : t
  val incr   : t -> t
  val to_int : t -> int
  val from_int : bool -> t
end
```

Choose one of the following (and, if you choose any but the first, write a short explanation):

☐ This interface is safe and useful

☐ This interface is safe but not useful
Why is it not useful?

☐ This interface is not safe
Why is it not safe?

☒ This interface doesn't match `M` (it would cause a compilation error)
What error? `from_int` *has the wrong type*

(b)
```
module type MSIG = sig
  type t
  val zero : t
  val incr : t -> t
  val to_int : t -> int
end
```

Choose one of the following (and, if you choose any but the first, write a short explanation):

☒ This interface is safe and useful

☐ This interface is safe but not useful
Why is it not useful?

☐ This interface is not safe
Why is it not safe?

☐ This interface doesn't match `M` (it would cause a compilation error)
What error?

(c)
```
module type MSIG = sig
  type t
  val zero  : t
  val incr  : t -> t
  val to_int : t -> int
  val from_int : int -> t
end
```

Choose one of the following (and, if you choose any but the first, write a short explanation):

☐ This interface is safe and useful

☐ This interface is safe but not useful
Why is it not useful?

☒ This interface is not safe
Why is it not safe? `from_int` *allows negative numbers to be turned into values of type* `M.t`

☐ This interface doesn't match `M` (it would cause a compilation error)
What error?

(d)
```
module type MSIG = sig
  type t
  val incr  : t -> t
  val to_int : t -> int
end
```

Choose one of the following (and, if you choose any but the first, write a short explanation):

☐ This interface is safe and useful

☒ This interface is safe but not useful
Why is it not useful? *There is no way to create values of type* `M.t`

☐ This interface is not safe
Why is it not safe?

☐ This interface doesn't match `M` (it would cause a compilation error)
What error?

(e)
```
module type MSIG = sig
  type t
  val zero  : t
  val incr  : t -> t
end
```

Choose one of the following (and, if you choose any but the first, write a short explanation):

- ☐ This interface is safe and useful
- ☒ This interface is safe but not useful
  Why is it not useful? *There is no way to use a value of type* `M.t` *for anything*
- ☐ This interface is not safe
  Why is it not safe?
- ☐ This interface doesn't match `M` (it would cause a compilation error)
  What error?

4. **Binary Search Trees and Testing (15 points)**

The following function tests whether a given `tree` satisfies the BST property. (The values `min_int` and `max_int` are the smallest and largest integers that can be represented using OCaml's `int` type. Intuitively, we're defining `max_label Empty` to be "negative infinity." This leads to short and simple definitions of `min_label` and `max_label`.)

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

let rec max_label (t: 'a tree) : 'a =
  begin match t with
  | Node(left, x, right) -> max (max (max_label left) x) (max_label right)
  | Empty -> min_int
  end

let rec min_label (t: 'a tree) : 'a =
  begin match t with
  | Node(left, x, right) -> min (min (min_label left) x) (min_label right)
  | Empty -> max_int
  end

let rec is_bst (t: 'a tree) : bool =
  begin match t with
  | Node(left, x, right) ->
     max_label left < x &&
     x < min_label right &&
     is_bst left &&
     is_bst right
  | Empty ->
     true
  end
```

For example, if the trees `good` and `bad` look like this

```
good  =   7                bad  =   7
         / \                       / \
        5   9                     5   6
       /   / \                   /     \
      2   8   10                2       10
```

(where, as usual, we omit `Empty` notes to reduce clutter) then applying `is_bst` to `good` will return `true`, and applying it to `bad` will return `false`.

(a) Suppose that we had made a mistake in `is_bst` and written it like this (the commented-out line is the only difference):

```
let rec is_bst (t: 'a tree) : bool =
  begin match t with
  | Node(left, x, right) ->
    max_label left < x &&
    (* x < min_label right && *)
    is_bst left &&
    is_bst right
  | Empty ->
    true
  end
```

Check the box next to each tree that does *not* satisfy the BST property but on which this variant of `is_bst` will (incorrectly) return `true`.

```
☐      1            ☒      2
      /                     \
     2                       1

☐      5            ☐      5
      / \                 / \
     2   7               2   8
    / \                 / \
   3   4               1   6

☐      1            ☒      7
      / \                 / \
     4   6               5   9
    / \   \             /   / \
   2   3   10          2   8   10
                              /
                             6
```

(b) Suppose that we had made a different mistake in `is_bst` and written it like this (again, the commented-out line is the only difference):

```
let rec is_bst (t: 'a tree) : bool =
  begin match t with
  | Node(left, x, right) ->
     max_label left < x &&
     x < min_label right &&
     (* is_bst left && *)
     is_bst right
  | Empty ->
     true
  end
```

Check the box next to each tree that does *not* satisfy the BST property but on which this variant of `is_bst` will (incorrectly) return `true`. The trees are the same as on the previous page.

```
☐      1            ☐      2
      /                     \
     2                       1

☒      5            ☐      5
      / \                 / \
     2   7               2   8
    / \                 / \
   3   4               1   6

☐      1            ☐      7
      / \                 / \
     4   6               5   9
    / \   \             /   / \
   2   3   10          2   8   10
                               /
                              6
```

(c) Suppose, instead, that we had tried to write `is_bst` without using `max_label` and `min_label`, like this:

```
let rec is_bst (t: 'a tree) : bool =
 begin match t with
  | Node(Node(ll,xl,rl), x, Node(lr,xr,rr)) ->
    xl < x && (* label of left subtree is less than x *)
    x < xr && (* x is less than label of right subtree *)
    is_bst (Node(ll,xl,rl)) &&
    is_bst (Node(lr,xr,rr))
  | Node(Node(ll,xl,rl), x, Empty) ->
    xl < x &&
    is_bst (Node(ll,xl,rl))
  | Node(Empty, x, Node(lr,xr,rr)) ->
    x < xr &&
    is_bst (Node(lr,xr,rr))
  | Node(Empty, x, Empty) ->
    true
  | Empty ->
    true
 end
```

Check the box next to each tree that does *not* satisfy the BST property but on which this variant of `is_bst` will (incorrectly) return `true`. The trees are the same as on the previous page.

```
☐       1                ☐       2
       /                          \
      2                            1

☐       5                ☒       5
       / \                      / \
      2   7                    2   8
     / \                      / \
    3   4                    1   6

☐       1                ☒       7
       / \                      / \
      4   6                    5   9
     / \   \                  /   / \
    2   3   10               2   8   10
                                     /
                                    6
```

Appendix: 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
```