CIS 1200 Midterm I   February 17, 2025

SOLUTIONS

1. **Types** (24 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 Appendix A.

   We've done the first one for you.

```
let example: _________________int list_________________ =
    [4; 5]
```

(a)
```
let ans: bool * bool * int =
    (true, false, 8)
```

(b)
```
let ans: int list =
    begin match [1; 2; 0] with
    | [] -> [-1]
    | hd :: tl -> tl
    end
```

(c)
```
let ans: bool list =
    transform (fun x -> x mod 2 = 0) [1; 2; 5]
```

(d)
```
let ans: ill-typed =
    3 +. 2
```

(e)
```
let ans: ill-typed =
    if true then "ocaml" else 1200
```

(f)
```
let ans : 'a list -> 'a list =
     fun x -> begin match x with
              | [] -> []
              | (h :: tl) -> (h :: h :: tl)
              end
```

(g)
```
let ans: int * string * int =
     (fun x y -> (x, "ocaml", y)) 1200 120
```

(h)
```
let ans: ('a -> int list -> int list) -> 'a list -> int list =
    fun x -> fold x [3; 5; 6]
```

2. **Tree recursion** (24 points total)

Consider a version of binary trees, called `branchy_trees` where data is stored only in leaf nodes. Like evolutionary trees from Homework 2, branches do not include labels. Unlike evolutionary trees, any subtree can be `Empty`.

```
type 'a branchy_tree =
  | Empty
  | Leaf of 'a
  | Branch of 'a branchy_tree * 'a branchy_tree
```

For clarity, when we draw branchy trees, we use the symbol `+` to indicate a branch and the symbol `*` to indicate an empty subtree. For example, this tree

```
let tree : int branchy_tree =
    Branch (Leaf 1, Branch (Empty, Leaf 2))
```

would be drawn as:

```
      +
     / \
    1   +
       / \
      *   2
```

(a) How would you represent the following tree in OCaml?

```
       +
      / \
     +   3
    / \
   1   2
```

(3 points) Check one.

☒
```
let tree_1 : int branchy_tree =
   Branch (Branch (Leaf 1, Leaf 2), Leaf 3)
```

☐
```
let tree_1 : int branchy_tree =
   Branch (Branch (Leaf 1, Leaf 2), Branch (Leaf 3, Empty))
```

☐
```
let tree_1 : int branchy_tree =
   Branch (Leaf 1, Branch (Leaf 2, Leaf 3))
```

(b) How would you represent the following tree in OCaml?

```
         +
        / \
       /   \
      +     +
     / \   / \
    *   1 +   *
         / \
        2   3
```

(3 points) Check one.

☐
```
let tree_3 : int branchy_tree =
   Branch (Leaf 1,
           Branch (Branch (Leaf 2, Leaf 3), Empty))
```

☐
```
let tree_3 : int branchy_tree =
   Branch (Branch (Leaf 1, Empty),
           Branch (Branch (Leaf 2, Empty), Leaf 3))
```

☒
```
let tree_3 : int branchy_tree =
    Branch (Branch (Empty, Leaf 1),
            Branch (Branch (Leaf 2, Leaf 3), Empty))
```

(c) (4 points) Consider the function `cons`, shown below.

```
let rec cons (x : 'a )(t : 'a branchy_tree) : 'a branchy_tree =
  begin match t with
  | Empty -> Leaf x
  | Leaf y -> Branch (Leaf x, Leaf y)
  | Branch (l, r) -> Branch (cons x l, r)
end
```

Which of the following trees corresponds to this OCaml expression:

```
let tree_4 = cons 1 (cons 2 (cons 3 Empty))
```

Circle one option below `(i)-(iv)`.

```
(i)             +         (ii)      +
               / \                 / \
              +   3               1   +
             / \                     / \
            1   2                   2   +
                                       / \
                                      3   *

(iii)      +              (iv)    +
          / \                    / \
         1   +                  +   *
            / \                / \
           2   3              +   3
                             / \
                            1   2
```

The correct tree is (i)

(d) (14 points) The `prune` function tidies up branching trees by removing as many occurrences of `Empty` as possible. In its result, no `Branch` should have an `Empty` subtree.

For example, both `tree_2` and `tree_3`, should produce the tree on the right when pruned.

```
tree_2         tree_3             pruned

   +              +                  +
  / \            / \                / \
 1   +          /   \              1   +
    / \        +     +                 /\
   2   +      / \   / \               2  3
      / \    *   1 +   *
     *   3        / \
                 2   3
```

The prune function is implemented through tree recursion.

```
let rec prune (t : 'a branchy_tree) : 'a branchy_tree =
  begin match t with
  | Empty -> Empty
  | Leaf x -> Leaf x
  | Branch (t1, t2) -> helper (prune t1) (prune t2)
end
```

Note that `prune` delegates to a helper function in the `Branch` case. Complete an appropriate definition for `helper` below.

```
(* helper function for prune *)
let helper (t1 : 'a branchy_tree) (t2 : 'a branchy_tree) : 'a branchy_tree =
  begin match (t1, t2) with
  | (Empty, _) -> t2
  | (t1, Empty) -> t1
  | (_,_) -> Branch (t1, t2)
end
```

3. **Binary Search Trees** (19 points total)

   This function concerns generic binary search trees `'a tree`, as defined in the lectures, Homework 3, Appendix B, and shown below.

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

   The `leaf` function constructs trees with no children.

```
let leaf (i:'a) : 'a tree = Node(Empty, i, Empty)
```

   (a) (8 points) The following value has type `bool tree` and satisfies the *binary search tree invariant*. (Note: we can compare boolean values in OCaml, with **`false`** `<` **`true`**.)

```
Node (leaf false, true, Empty)
```

   Below, list *all* other values of type `bool tree` that satisfy the binary search tree invariant. You may use the `leaf` function and constructors for the `tree` type in your answer.

```
Empty

leaf false

leaf true

Node (Empty, false, leaf true)
```

(b) (8 points) Consider the following mystery function that uses the binary search tree invariant to calculate some result.

```
(* helper function *)
let rec mystery_helper (t : int tree) (n : int) (k : int) : int =
  begin match t with
  | Empty -> k
  | Node (lt, v, rt) ->
      if n > v then mystery_helper rt n v
      else mystery_helper lt n k
  end

(* mystery function *)
let mystery (t:int tree) (n:int) : int = mystery_helper t n (-999)
```

Complete the following test cases, demonstrating your understanding of this function.

```
let tree_1 = Node (Node (Empty, 2, Empty), 4, Node (Empty, 6, Empty))

let test () : bool = mystery tree_1 1 = -999
;; run_test "mystery 1" test

let test () : bool = mystery tree_1 3 = 2
;; run_test "mystery 3" test

let test () : bool = mystery tree_1 4 = 2
;; run_test "mystery 4" test

let test () : bool = mystery tree_1 7 = 6
;; run_test "mystery 7" test
```

(c) (3 points) Suppose we add this function to the `SET` interface (Appendix C) with type:

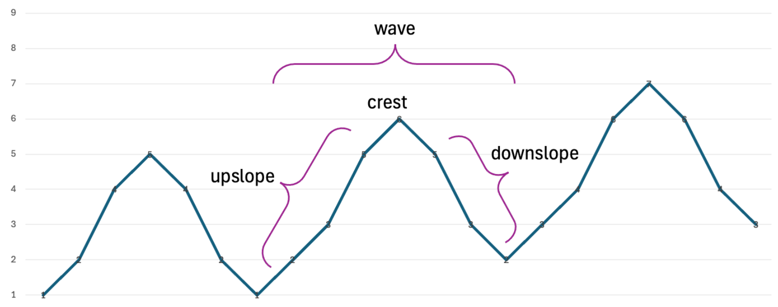
```
val mystery : int set -> int -> int
```

Give a concise description of this implementation when given arguments `s` and `x`. Your answer should be about sets and should not mention the `tree` type or its constructors. You may assume that `-999` is not an element of the set `s`.

Answer: `mystery s x` *returns the largest number in the set* `s` *that is strictly smaller than* `x`*, or* `-999` *if there is no such number.*

4. **Program design and List Recursion** (25 points total)

Some data comes in the form of periodic waves. This data is easiest to understand visually.



However, in OCaml we must work with the raw data that generates these charts. In this problem, we represent this data as a list of *waves*, where each wave is a triple containing an *upslope*, *crest value*, and *downslope*.

```
type wave = int list * int * int list
```

For example, this list contains three waves and corresponds to the chart above.

```
let data = [([1;2;4],5,[4;2;1]); ([2;3;5],6,[5;3;2]); ([3;4;6],7,[6;4;3])]
```

When working with data, the first step is validation. For each wave, we want to ensure that the list of values for the upslope is monotonically increasing and that the downslope is monotonically decreasing. To do so, we will use a higher-order function called `monotonic`, which is parameterized by a comparison function.

The following two functions determine whether their provided lists contain elements in strictly increasing or strictly decreasing order, respectively.

```
let increasing : int list -> bool = monotonic (fun x y -> x < y)
let decreasing : int list -> bool = monotonic (fun x y -> x > y)
```

(a) (3 points) Write the type of the `monotonic` function based on its usage above as you might see it in an `mli` file or module signature.

```
val monotonic : ('a -> 'a -> bool) -> 'a list -> bool
```

or

```
val monotonic : (int -> int -> bool) -> int list -> bool
```

or (technically correct)

```
val monotonic : ('a -> 'a -> bool) -> int list -> bool
```

(b) (9 points) Next, we should write some tests. A monotonically increasing list is one where each value in the list is strictly larger than the previous value, and the analogue holds for a decreasing list. For example, we can test the `decreasing` function with the following code:

```
let test (): bool = let x = [5; 4; 3; 2] in
                    decreasing x
;; run_test "decreasing list" test
```

Complete the following test cases for the `increasing` and `decreasing` functions defined above. Don't forget that `increasing` and `decreasing` are defined using the < and > operators.

```
;; run_test "increasing" (fun () ->
    let x = [1; 2; 3; 5] in
   increasing x && not (decreasing x))
;; run_test "both" (fun () ->
    let x = [] in
    increasing x && decreasing x)
;; run_test "neither" (fun () ->
    let x = [3;4;3] in
    not (increasing x) && not (decreasing x))
```

(c) (13 points) Now complete the following definition of the `monotonic` function. Your solution must be recursive and should not use `transform`, `fold`, or any other list library function. Constructors, such as `::` and `[]`, and pattern matching expressions are fine. (Type annotations have been omitted, but you don't have to fill them in.)

```
(* is the list monotonic with respect to the given comparison function? *)
let rec monotonic  cmp (l : int list) : bool =
  begin match l with
        | [] -> true
        | [x] -> true
        | x :: y :: tl -> cmp x y && monotonic cmp (y :: tl)
end
```

5. **Higher Order Functions** (16 points total)

Use the higher-order list processing functions `transform` *and* `fold` *(see Appendix A) to complete the following functions. For these problems* ***do NOT use recursion*** *or any other list library functions. Constructors, such as* `::` *and* `[]`*, are fine.*

(a) (8 points) Implement a generic function, called `all`, that determines whether all values in a list satisfy a given predicate. The test below should pass.

```
let all (pred : 'a -> bool) (l : 'a list) : bool =
  fold (fun x acc -> pred x && acc) true l

let test () : bool = all (fun x -> x < 6) [1; 2; 3; 5]
;; run_test "all" test
```

(b) (8 points) Recall the `wave` type with components for the *upslope*, *crest*, and *downslope*.

```
type wave = int list * int * int list
```

Write a function, called `summarize` that returns all of the wave crests found in a list of waves, using `transform` and/or `fold`. The test below should pass.

```
let crest (w:wave) : int =
     begin match w with
     | (_,c,_) -> c
     end
let summarize (s:wave list) : int list =
    transform crest s

let data = [([1;2;4],5,[4;2;1]); ([2;3;5],6,[5;3;2]); ([3;4;6],7,[6;4;3])]

let test () : bool = summarize data = [5;6;7]
;; run_test "summarize" test
```

6. **Abstract Data Types** (12 points total)

Recall the wave type from Problem 4.

```
type wave = int list * int * int list
```

This type comes with a *representation invariant*. A wave is *valid* if its upslope is increasing, its downslope is decreasing, and the crest is the largest value. We can test waves for validity using the function `valid_wave`, shown in Appendix D.

Furthermore, a `sequence` is a list of waves.

```
type sequence = wave list
```

This type also has a representation invariant. A sequence is *valid sequence* if every wave in the list is valid. We can test a sequence for validity using the function `valid_sequence`, shown in Appendix D.

We would like to use an *abstract data type* to safely maintain these invariants when working with wave data. In other words, we want to create a module signature that ensures that the `summarize` function in this interface can only be called with a valid sequence. (For reference, Appendix D contains the implementation of the module.)

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.

(There is nothing to do on this page.)

(a) (3 points)

```
module type DATA = sig
  type wave = int list * int * int list
  type sequence = wave list
  val empty : sequence
  val add_wave : wave -> sequence -> sequence
  val summarize : sequence -> int list
end
```

☐ Unusable [✓ Unsafe] ☐ Good

Explanation: *Not safe: because the* `wave` *type is NOT abstract, a triple could be passed to* `add_wave` *that is not a valid wave. This would produce a* `sequence` *that could be passed to* `summarize`*.*

(b) (3 points)

```
module type DATA = sig
   type wave
   type sequence = wave list
   val make_wave : int list -> int -> int list -> wave
   val empty : sequence
   val add_wave  : wave -> sequence -> sequence
   val summarize : sequence -> int list
end
```

☐ Unusable ☐ Unsafe [✓ Good]

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

(c) (3 points)

```
module type DATA = sig
  type wave
  type sequence = wave list
  val empty : sequence
  val add_wave : wave -> sequence -> sequence
  val summarize : sequence -> int list
end
```

[✓ Unusable] ☐ Unsafe ☐ Good

Explanation: *Unusable: the* `wave` *type is abstract and there is no way to create a wave. So the* `add_wave` *function cannot be called.*

(d) (3 points)

```
module type DATA = sig
  type wave
  type sequence
  val empty : sequence
  val make_wave : int list -> int -> int list -> wave
  val add_wave : wave -> sequence -> sequence
  val summarize : sequence -> int list
end
```

☐ Unusable ☐ Unsafe [✓ Good]

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

A Higher-Order List Processing Functions

The higher-order list processing functions `transform` and `fold`:

```
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
```

B Generic Binary Search Trees

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

let leaf (i:'a) : 'a tree = Node(Empty, i, Empty)

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
```

C SET interface

```
module type SET = sig
  type 'a set

  val empty : 'a set
  val add : 'a -> 'a set -> 'a set
  val member : 'a -> 'a set -> bool
  val equals : 'a set -> 'a set -> bool
  val set_of_list : 'a list -> 'a set

end
```

D Wave Module

```
module Wave : DATA = struct

  type wave = int list * int * int list

  type sequence = wave list

  let increasing : int list -> bool = monotonic (fun x y -> x < y)
  let decreasing : int list -> bool = monotonic (fun x y -> x > y)

  let valid_wave (w : wave) : bool =
    begin match w with
    | (up, crest, down) -> increasing up  && decreasing down
        && all (fun x -> crest > x) up && all (fun x -> crest > x) down
    end

  let valid_sequence (s: sequence) : bool =
    all valid_wave s

  let make_wave (up : int list) (crest : int) (down : int list) : wave =
     let w = (up, crest, down) in
     if valid_wave w then w else failwith "ERROR: invalid wave"

  let empty : sequence = []

  let add_wave (w: wave) (s : sequence) : sequence = w :: s

  let summarize (s : sequence) : int list =  ... (* implementation not shown *)

end
```