CIS 1200 Midterm I   September 30, 2022

SOLUTIONS

1. **List Recursion** (4 points)

   Fill in the blanks in the following function so that it will pass the tests below:

   Answer:

```
let rec upto_helper (n: int) (limit: int) : int list =
  if n > limit then []
  else n :: upto_helper (n+1) limit


let upto (n: int) : int list = upto_helper 0 n

;; run_test "upto 0" (fun () -> upto 0 = [0])
;; run_test "upto 5" (fun () -> upto 5 = [0;1;2;3;4;5])
```

2. **Compression** (12 points total)

   Suppose we want a data structure for representing sequences of readings taken at one-second intervals from a temperature sensor, where some of the values may be missing (because the sensor only produces readings every few seconds).

   One way to represent such data—let's call it the *expanded* representation—is as a simple list of values, one for each one-second interval, filling in `0` for intervals when the sensor does not actually produce a reading.

   For example, if the sensor produces `100` at the very first time interval and `101` a few seconds later, we might represent this as:

   ```
   [ 100; 0; 0; 0; 0; 0; 101 ]
   ```

   (We'll assume, for simplicity, that the sensor never produces the value `0` as a valid reading.)

   Alternatively, we could represent the same readings more compactly by giving a list of just the valid readings together with the times at which they occurred:

   ```
   [ (100,0); (101,6) ]
   ```

   (Note that we're calling the first interval "interval `0`," counting from zero like good computer scientists!) We call this the *compressed* representation of our time-series data.

   Your job in this problem will be to write functions that convert back and forth between the expanded and compressed representations.

Continued on next page...

First, let's define a couple of type abbreviations:

```
type expanded = int list

type compressed = (int * int) list
```

(a) (4 points) Now let's write the compression function.

Fill in the blanks in the following helper so that the main function `compress` will pass the tests below:

Answer:

```
let rec compress_helper (e: expanded) (time: int) =
  begin match e with
    | [] -> []
    | x::tail -> if not (x = 0) then [(x,time)] @ compress_helper tail (time+1)
                 else compress_helper tail (time+1)
  end

let compress (e: expanded) : compressed = compress_helper e 0

;; run_test "compress empty"
      (fun () -> compress [] = [])
;; run_test "compress singleton"
      (fun () -> compress [100] = [(100,0)])
;; run_test "compress example from above"
      (fun () -> compress [ 100; 0; 0; 0; 0; 0; 101 ] = [(100,0); (101,6)])
;; run_test "compress with initial and final zeros"
      (fun () -> compress [0;100;99;0] = [(100,1);(99,2)])
```

(b) (5 points) Next let's write the *un*compression function.

Fill in the blanks in the following helper so that the main function `expand` will pass the tests below:

Answer:

```
let rec expand_helper (c: compressed) (time: int) : expanded =
  begin match c with
    | [] -> []
    | (x,m) :: rest ->
        if m=time then x :: (expand_helper rest (time+1))
        else           0 :: (expand_helper c    (time+1))
  end


let expand (c: compressed) : expanded =
  expand_helper c 0

;; run_test "expand empty"
      (fun () -> expand [] = [])
;; run_test "expand with initial zeros"
      (fun () -> expand [(100,2)] = [0;0;100])
;; run_test "expand example from earlier"
      (fun () -> expand [(100,0); (101,6)] = [ 100; 0; 0; 0; 0; 0; 101 ])
```

(c) (3 points)

i. "For any `e : expanded`, it is always the case that `expand (compress e) = e`."

True ☐ False ☒

If you choose False, give a counterexample:

```
e  = [ 0 ]
```

ii. "For any `c : compressed`, it is always the case that `compress (expand c) = c`."

True ☒ False ☐

If you choose False, give a counterexample:

3. **ADTs** (8 points total)

Suppose we define the following module to collect together some functions operating on time-series data.

```
module COMPRESSED (* : I *) =
  struct
    type t = compressed

    let import (m: expanded) : t = compress m
    let export (m: t) : expanded = expand m

    (* Check whether any element of a list satisfies a boolean
       predicate (helper function for extend) *)
    let rec exists (test: int*int -> bool) (m: compressed) : bool =
       begin match m with
       | [] -> false
       | hd::tl -> test hd || exists test tl
       end

    (* Add a new reading at the end of a compressed list of readings *)
    let extend (m: t) (newreading: int) (newtime: int) : t =
      if exists (fun (_, oldtime) -> oldtime >= newtime) m then
        failwith "time is less than that of an existing measurement"
      else
        m @ [(newreading, newtime)]
  end
```

The `import` and `extend` functions maintain the invariant that the times of the measurements are sorted in increasing order.

For each of the following possible definitions of the interface `I`, check the box next to the phrase that best describes it:

- *Ill typed* (the interface does not match the module and the code will not compile)
- (Well typed but) *Unusable* (because there is no way to call any of the functions)
- (Well typed and usable but) *does not preserve the invariant* ☹
- *Good* (well typed, usable, and maintains the invariant)

Continued on next page...

(2 points each)

(a)

```
module type I =
  sig
    type t
  end
```

☐ Ill typed ☒ Unusable ☐ Invariant ☹ ☐ Good

(b)

```
module type I =
  sig
    type t
    val import : expanded -> t
    val export : t -> expanded
    val extend : t -> int -> int -> t
  end
```

☐ Ill typed ☐ Unusable ☐ Invariant ☹ ☒ Good

(c)

```
module type I =
  sig
    val import : expanded -> compressed
    val export : compressed -> expanded
    val extend : compressed -> int -> int -> compressed
  end
```

☐ Ill typed ☐ Unusable ☒ Invariant ☹ ☐ Good

(d)

```
module type I =
  sig
    type t
    val export : t -> expanded
    val extend : t -> int -> int -> t
  end
```

☐ Ill typed ☒ Unusable ☐ Invariant ☹ ☐ Good

4. **Binary Search Trees** (12 points total)

This problem concerns *buggy* implementations of the `lookup` and `insert` functions for binary search trees, the correct versions of which are shown in Appendix B. Note that this problem refers to the `'a tree` type defined there.

Consider the following tree `t` (as usual, `Empty` constructors are not shown, to avoid clutter).

```
let t : int tree =     9
                      / \
                     4   10
                    / \    \
                   1   5   15
```

(a) (1 point) "Tree `t` satisfies the BST invariants." ☒ True ☐ False

(b) (5 points) Consider this incorrect definition of `lookup`. (The correct version can be found in Appendix B.)

```
1   let rec bad_lookup (t: int tree) (n: int) : bool =
2     begin match t with
3       | Empty -> n
4       | Node(lt, x, rt) ->
5         if n = x then false
6         else if n <= x then bad_lookup lt n
7         else bad_lookup rt n
8     end
```

i. "The code above compiles." ☐ True ☒ False

ii. If you chose `False`, specify which line number the error will be reported for, what error the typechecker will print (don't worry about the exact wording), and one possible fix that makes the program compile.

```
Compile Error on line ____3____

Error message: ___The expression has type int instead of bool__

Fix For Compile Error: ______replace n with false________________
```

iii. Even after the compile-time error (if any) is fixed, the code will still be buggy: for some inputs it will produce the correct answer, but for others it will not.

Complete each of the test cases below with an `int` value for `x` so that the test passes, demonstrating that this implementation (including your fix for the compilation error, if any) sometimes produces the correct answers and sometimes does not.

Both of your test cases must use the tree `t` shown above.

ANSWER: This lookup function will always return false—i.e., it will return the correct answer only for nodes that are not in the tree.

```
;; run_test "bad_lookup works correctly" (fun () ->
     let x = _____3______ in
     bad_lookup t x  =  lookup t x)

;; run_test "bad_lookup computes wrong answer" (fun () ->
     let x = _____9______ in
     not (bad_lookup t x = lookup t x))
```

(c) (6 points) Consider this incorrect definition of `insert`. (The correct version can be found in Appendix B.)

```
1    let rec bad_insert (t: 'a tree) (n: 'a) : 'a tree =
2     begin match t with
3      | Empty -> Node(Empty, n, Empty)
4      | Node(lt, x, rt) ->
5        if x = n then t
6        else if n < x then Node(lt, x, bad_insert rt n)
7        else Node(bad_insert lt n, x, rt)
8     end
```

i. "The code above compiles." ☒ True ☐ False

ii. If you chose `False`, specify which line number the error will be reported for, what error the typechecker will print (don't worry about the exact wording), and one possible fix that makes the program compile..

```
Compile error on line: ___No Error________

Error message        : ______________________

Fix for Compile Error: ______________________
```

iii. Draw two pictures of Binary Search Trees corresponding to the test cases below. For the first, `bad_insert` (after your correction from part (ii), if any) should behave correctly, and for the second it should behave incorrectly, demonstrating that this implementation sometimes produces the correct answers and sometimes does not.

```
;; run_test "bad_insert works correctly" (fun () ->
     bad_insert t1 5 = insert t1 5)

;; run_test "bad_insert works incorrectly" (fun () ->
     not (bad_insert t2 5 = insert t2 5))
```

```
Works Correctly:
t1 : int tree =    5

Works Incorrectly
t2 : int tree =         7
                       / \
                      4   9
```

ANSWER: This `insert` *function inserts an element into the opposite side of the tree. If a value should be on the left, it will go to the right and vice versa. It will work correctly only if the element is already present in the tree or if you start with an* `Empty` *tree.*

5. **Generic Types, Higher-Order Functions, Testing** (12 points total)

Recall the `transform` and `fold` 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
```

In the first two parts, you are asked to write a test case (involving a list with at least three elements). For example, one possible test case for the `fold` function above could be:

```
;; run_test "fold plus" (fun () -> fold (fun x y -> x+y) 0 [1;2;3] = 6)
```

Your solutions in this problem *must not* use any list library functions. The list constructors `::` and `[]` are fine.

(a) (3 points) Write a test case, involving a list with at least three elements, for the variant of `transform` shown below.

```
let rec squish (f: 'a -> 'a -> 'b) (l: 'a list) : 'b list =
  begin match l with
  | [] -> []
  | [x] -> []
  | x::y::rest -> (f x y) :: (squish f (y::rest))
  end
```

Answer:

```
;; run_test "squish multiply" (fun () ->
  squish (fun x y -> x * y) [1; 2; 3; 4] = [2; 6; 12])
```

(b) (4 points) Write a test case, involving a list with at least three elements, for the variant of `zip` shown below:

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

Answer:

```
;; run_test "zip_with linear combination" (fun () ->
  zip_with (fun x y -> 2 * x + y) [1; 2; 3; 4] [5; 6; 7; 8] = [7; 10; 13; 16])
```

(c) (5 points) Use `transform` or `fold`, along with suitable anonymous function(s), to implement the `chunk` function below.

This function should partition the input list into a tuple of lists `(list1, list2)` based on whether the elements satisfy the given predicate `p`. Elements that satisfy the predicate should be included in `list1` and elements that do not satisfy the predicate should be included in `list2`. For example, the call

```
chunk (fun x -> x mod 2 = 0) [1; 2; 3; 4; 5]
```

should evaluate to

```
([2; 4], [1; 3; 5])
```

since `2` and `4` satisfy the predicate, while `1`, `3`, and `5` do not.

Answer:

```
let chunk (p: 'a -> bool) (l: 'a list) : ('a list * 'a list) =
  fold (fun x (acc_yes, acc_no) ->
    if p x then (x :: acc_yes, acc_no) else (acc_yes, x :: acc_no))
    ([], []) l
```

6. **Types** (12 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 and Appendix B.

(2 points each)

(a)

```
let a : (int * string) list list =    (* note the brackets below... *)
  [[(1200, "STIT B6"); (1600, "Towne 100")]; [(0100, "MEYH B1")]]
```

(b)

```
let b : ill typed =
  let z : int list list = [[1; 3]; [2]] in
  begin match z with
  | [] -> 0
  | _ -> z
  end
```

(c)

```
let c (value: (int * int) list) : int list =
  transform (fun (x, y) -> x) value
```

(d)

```
let d : int list -> int =
  fun (l: 'a list) -> fold (fun x acc -> x + acc) 0 l
```

(e)

```
let e (base: 'a list): 'a list -> 'a list =
  fold (fun x acc -> [x] @ acc) base
```

(f)

```
let f : ('a -> 'b) -> 'a -> 'b =
  fun f x -> f x
```

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: 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
```