CIS 120 Midterm I February 16, 2016

SOLUTIONS

1. Types (14 points)

For each OCaml value below, fill in the blank where the type annotation could go or write "ill typed" if there is a type error on that line. Your answer should be the most specific type possible, i.e. `int list` instead of `'a list`.

Some of these expressions refer to values specified by the `SET` interface from homework 3. An abbreviated version this interface appears on page **??** in the reference appendix. You may assume that all of the definitions below appear *outside* of a module that implements this interface, such as `OLSet`, and that this module has already been opened.

We have done the first one for you.

```
;; open OLSet

let z : _______ int list __________ = [1]


let a : __ ill typed __ = [add 3 empty; add true empty]


let b : ___ int set list _____ = [add 3 empty; empty]


let c : _____int list set ___ = add [3] empty


let d : ______ill typed____ = if (3 = []) then 1 else 2


let e : _____ill typed____ = add 3 []


let f : ___int set -> int set ___ = add 3


let g : ___int set set_____ = set_of_list [ add 3 empty ]
```

Grading Scheme: 2 points each.

2. Higher-order Functions

Recall 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
```

Multiple choice: For each of the following (well-typed) programs, check the box for the value computed for `ans`, or mark "infinite loop" if the program loops. Note that we have deliberately replaced some types with `??` in the code below.

Grading: 4 points each, no partial credit.

a. (4 points)

```
let f (x:bool) (y:int) : ?? =
  if x then 3 else y in
let ans = f true
```

ans =

☐ `1`

☐ **`true`**

☒ **`fun`** `(y:int) ->` **`if true then`** `3` **`else`** `y`

☐ **`false`**

☐ **`fun`** `(x:bool) ->` **`if`** `x` **`then`** `3` **`else`** `y`

☐ infinite loop

b. (4 points)

```
let rec f (x:int) (y:int) : int list -> int list =
  transform (fun h -> if h = x then y else h)
let ans = f 0 2 [1;0;3;0]
```

ans =

☐ `[0;2;0;2]`

☒ `[1;2;3;2]`

☐ `[1;0;3;0]`

☐ `[2;0;2;0]`

☐ `[1;2;3;4]`

☐ infinite loop

c. (4 points)

```
let rec g (x : 'a list) : ?? =
  transform (fun h -> [h]) x
let ans = g [1;2;3]
```

ans =

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

d. (4 points)

```
let rec n (x : int list) : ?? =
  begin match x with
  | [] -> n x
  | h :: t -> fold (fun y a -> if y > a then y else a) h t
  end
let ans = n [1;2;3;5]
```

ans =

- ☐ `0`
- ☐ `1`
- ☐ `4`
- ☒ `5`
- ☐ `11`
- ☐ infinite loop

3. Binary Search Trees

The next problem concerns a *buggy* implementation of binary search trees. For reference, the correct implementation (presented in class) appears in the Appendix on page **??**.

Although the implementations below are incorrect, there are still some inputs that work. Complete each of the test cases below to demonstrate that these implementations sometimes produce the correct answer and sometimes do not. Your answer will be always be an integer. These test cases all use the tree shown pictorially as:

```
t =   5
     / \
    1   7
   / \
  0   3
```

a. (6 points)

```
let rec bad_lookup (t:tree) (n:int) : bool =
 begin match t with
   | Empty -> false
   | Node(lt, x, rt) ->
    if n < x then bad_lookup lt n
    else bad_lookup rt n
 end


;; run_test "bad_lookup_works" (fun () ->


    let x = _____________ in
    bad_lookup t x = lookup t x)

;; run_test "bad_lookup_fails" (fun () ->


    let x = _____________ in
    not (bad_lookup t x = lookup t x))
```

ANSWER: This lookup function always returns false. Therefore, answers to the first test include any number not present in the tree, and for the second, any number in the tree.

b. (6 points)

```
let rec bad_insert (t:tree) (n:int) : tree =
 begin match t with
   | Empty -> Node(Empty, n, Empty)
   | Node(lt, x, rt) ->
    if x = n then t
    else if n < x then bad_insert lt n
    else bad_insert rt n
 end

;; run_test "bad_insert_works" (fun () ->


    let x = _____________ in
    bad_insert t x = insert t x)

;; run_test "bad_insert_fails" (fun () ->


    let x = _____________ in
    not (bad_insert t x = insert t x))
```

ANSWER: This insert function only works for 5 (which is already in the tree).

4. Abstract types and invariants (28 points total)

In this problem, you will think about invariants in the implementation of an abstract type of *interval sets of integers*, defined by the signature `IVALSET` below.

```
module type IVALSET = sig
  type ivalset

  (* the empty interval set *)
  val empty    : ivalset

  (* test whether an interval set contains a specific number *)
  val contains : int -> ivalset -> bool

  (* test if two sets contain the same elements *)
  val equals   : ivalset -> ivalset -> bool

  (* "add_interval lo hi s" constructs a set that extends s with all elements
     greater or equal to lo and strictly less than hi. *)
  val add_interval : int -> int -> ivalset -> ivalset
end
```

This data structure is like a standard set—for example, there is an empty set `empty`, we can test whether a set `contains` a particular element, and we can compare two sets for equality with `equals`.

The main novelty of this abstract type is that it optimizes the storage of dense ranges of numbers. For example, suppose that you want to create a set containing the numbers `1`, `2`, `3`, `4`, `5`, and `6`. Using the sets from HW3, you must add each of these numbers individually. However, this structure allows the bulk operation of adding an *interval* of numbers to create the set.

```
let set1to6 : ivalset = add_interval 1 7 empty
```

An *interval* is defined by two elements, `lo` and `hi`, representing all `x` such that `lo` $\leq$ `x` $<$ `hi`. (Note that if `lo` $\geq$ `hi`, then the interval is empty.) As an example, the following expressions evaluate to **`true`**, indicating that `4` is contained in the interval specified by `1` and `7`

```
contains 4 (add_interval 1 7 empty)
```

but `7` is not

```
not (contains 7 (add_interval 1 7 empty))
```

and adding an empty interval does not change a set.

```
equals (add_interval 3 3 empty) empty
```

Your job is to implement this interface. We have started this implementation for you using an ordered list of pairs as the concrete representation. For example, the set $\{1\ 2\ 3\ 4\ 5\ 6\}$ is represented by a list containing just the interval `[(1,7)]`. Similarly, the set $\{1\ 2\ 4\ 5\ 6\}$ is represented by the two intervals `[(1,3);(4,7)]`.

```
module Ivalset : IVALSET = struct

  type ivalset = (int*int) list

  let empty : ivalset = []

  let rec equals (s1:ivalset) (s2:ivalset) : bool = (s1 = s2)

  let rec contains (i:int) (x:ivalset) : bool =

    (* ... implementation of contains (part b) ... *)

  let rec add_interval (lo : int) (hi: int) (x:ivalset) : ivalset =

    (* ... implementation of add_interval (part c) ... *)

end
```

a. (6 points) We have already implemented the `equals` function above. Two sets are equal when `contains` returns the same answer for every input. For example, the following expression should evaluate to **`true`**

```
equals (add_interval 1 3 (add_interval 3 7 empty))
       (add_interval 1 7 empty)
```

However, our implementation of `equals` relies on the following *invariant* about `ivalset`s.

- The intervals must be nonempty: Every `(lo,hi)` in the list must have `lo` $<$ `hi`.
- The intervals must be disjoint: Each `(lo1,hi1)` followed by `(lo2,hi2)` in the list must have `hi1` $<$ `lo2`.

Note that this invariant implies that the intervals in the list are sorted.

Mark below whether each list satisfies or does not satisfy this invariant.

`[]`	☒	satisfies	☐	does not satisfy the invariant.
`[(1,7)]`	☒	satisfies	☐	does not satisfy the invariant.
`[(1,1);(2,3)]`	☐	satisfies	☒	does not satisfy the invariant.
`[(1,4);(2,5)]`	☐	satisfies	☒	does not satisfy the invariant.
`[(1,2);(2,3)]`	☐	satisfies	☒	does not satisfy the invariant.
`[(1,2);(3,4)]`	☒	satisfies	☐	does not satisfy the invariant.

Grading Scheme: 1 point each.

b. (10 points) Now implement the `contains` function. Your implementation should take advantage of the invariant from part (a) to do less work.

```
let rec contains (i:int) (x:ivalset) : bool =
  begin match x with
    | [] -> false
    | (lo, hi) :: t ->
      if i < hi then i >= lo
      else contains i t
  end
```

Grading Scheme: 2 points base case, 2 points testing between lo and hi, 4 points for correct behavior when $i < hi$, (including skipping the recursive call), 2 points for recursive case when $i \geq hi$.

c. (12 points) Finally, consider several potential implementations of the `add_interval` function on the next few pages. As above, `add_interval` should assume that the invariant holds for `x`, but this time it also must ensure that it holds for *any* output for this function.

For each potential implementation, circle whether it correctly preserves the invariant or give an example where the output of the function violates the invariant.

Grading Scheme: 4 points each.

```
let rec add_interval (lo:int) (hi:int) (x:ivalset) : ivalset =
  (lo,hi) :: x
```

Circle one:

correct violates invariant **when**

lo = ___2_____

hi = ___3_____

x = ___[(1,2)]_____

and produces the following result

_____[(2,3); (1,2)]_____

Grading Scheme: This function violates the invariant when the new interval includes a number larger than any already contained in the ivalset, or when the new interval is empty.

```
let rec add_interval (lo:int) (hi:int) (x:ivalset) : ivalset =
  if hi <= lo then x
  else begin match x with
    | [] -> [(lo,hi)]
    | (lo2,hi2) :: t ->
      if hi < lo2
       then (lo,hi) :: x
      else if lo > hi2
       then (lo2,hi2) :: add_interval lo hi t
      else
       let nlo = min lo lo2 in
       let nhi = max hi hi2 in
       (nlo, nhi) :: t
  end
```

Circle one:

correct violates invariant **when**

lo = ____2_____

hi = ____6_____

x = _____[(1,3);(5,7)]_____

and produces the following result

_[(1,6);(5,7)]_____

Grading Scheme: This function violates the invariant when the new interval overlaps two intervals already in the list.

```
let rec add_interval (lo:int) (hi:int) (x:ivalset) : ivalset =
  if hi <= lo then x
  else begin match x with
    | [] -> [(lo,hi)]
    | (lo2,hi2) :: t ->
      if hi < lo2
       then (lo,hi) :: x
      else if lo > hi2
       then (lo2, hi2) :: add_interval lo hi t
      else
       let nlo = min lo lo2 in
       let nhi = max hi hi2 in
       add_interval nlo nhi t
  end
```

Circle one:

correct violates invariant **when**

lo = _____

hi = _____

x = _____

and produces the following result

_____

Grading Scheme: This version is correct.

5. List Recursion and Program Design (30 points total)

For this problem, you will use the program design process to implement a function called `sublist`. This function takes two lists and determines whether the second list is fully contained within the first. The second list may appear anywhere within the first list (not just at the beginning) and all of the elements of the second list must appear contiguously and in the same order. The `sublist` function should have a generic type.

For example, the following four test cases demonstrate the behavior of `sublist`:

```
;; run_test "[] is always a sublist" (fun () ->
  sublist [true;false] [])

;; run_test "sublist need not be at the start" (fun () ->
  sublist [1;1;1;2] [1;1;2])

;; run_test "all elements of second list must appear in first" (fun () ->
  not (sublist ['a';'b'] ['a';'b';'c']))

;; run_test "elements must occur contiguously" (fun () ->
  not (sublist [1;2;3] [1;3]))
```

You must use a helper function in your implementation of `sublist`, in part (e) below. In parts (a)-(d) of this problem, you will use the program design process to develop that helper function.

a. (0 points) The first step is to understand how your helper function should relate to `sublist`. You may want to look ahead to part (e) and think about how you will implement `sublist` using your helper function before completing the next steps.

In the space below, write a short description of what your helper function does. Although this part is worth no points, it will help us to understand your thinking as we grade your answers in parts (b)-(e).

Answer: Our helper function takes two lists and determines whether the second list is a prefix of the first list. Grading Scheme: In general, we used this answer to understand the general approach and not penalize students to took a different tactic too harshly.

b. (2 points) Next, define the interface of your helper function. This interface must be consistent with how you use your helper function in `sublist`. Write the type of your helper function as you might see it in a signature or `.mli` file.

```
val helper: _______ 'a list -> 'a list -> bool _____________________
```

Grading Scheme: One point deduction for non-generic types, no deduction for syntax errors or answers that are "almost" correct. If the type does not match the above, but it is consistent with general approach taken, then no deduction.

c. (6 points) Now, write three *different* tests for **your helper function**. Put some thought into your answers; we will be grading your answers not just on correctness, but on how well your tests specify its behavior. Don't forget to give each case a descriptive name.

Grading Scheme: We are looking for test cases that correctly specify the behavior of the helper function and exhaustively test it. A good test suite includes tests along these lines:

- *at least one test that returns true and one test that returns false*
- *at least one test that returns a different answer from* `sublist`
- *at least one test where one of the inputs is nil.*

Points were deducted for test cases that did not specify the helper function well or were otherwise redundant.

The tests must also be correct. Here correctness means that they be consistent with the description of the helper function in part (a) and the definition of the helper function in part (c). Some students wrote tests for sublist instead of their helper function and received a three point deduction (if the tests were otherwise correct).

d. (8 points) Now, *implement* your helper function in the space provided below. Don't forget to include the type declarations for the arguments to your helper function, which should be consistent with your answer for part (b).

```
let rec helper (x : 'a list) (y : 'a list) : bool =
 begin match x, y with
 | _, []        -> true
 | [], _        -> false
 | h :: t, h' :: t' -> h = h' && helper t t'
 end
```

Grading Scheme: 2 points for first nil case, 2 points for second nil case, four points for last case. The implementation must be consistent with the description of the helper function in part (a) and the tests for the helper function in part (b). No deduction here for implementing a different helper function from the one above as long as it is correctly written and consistent with the description and the test.

e. (14 points) Finally, complete the definition of `sublist`

```
let rec sublist (x : 'a list) (y : 'a list) : bool =
 begin match x with
 | h :: t -> helper x y || sublist t y
 | []   -> y = []
 else
```

Grading Scheme: 3 point deduction for errors in the base case. 4 point deduction for not including a recursive call when the helper function fails. 8 point deduction for trying to implement `sublist` *without a suitable recursive helper function (by trying to do too much in the implementation here or trying to do too much in the helper function itself).These versions either accept too few inputs (they reject* `sublist [1;1;2] [1;2]`*) or too many (they accept* `sublist [1;2;3] [1;3]`*). Other errors at discretion.*

Appendix: SET interface

The interface for the set abstract type.

```
module type SET = sig

  type 'a set

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

  ...
end
```

Appendix: Binary Search Tree implementation

```
type tree =
    | Empty
    | Node of tree * int * tree

let rec lookup (t:tree) (n:int) : 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

let rec insert (t:tree) (n:int) : 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

let rec tree_max (t:tree) : int =
 begin match t with
   | Empty -> failwith "tree_max called on empty tree"
   | Node(_,x,Empty) -> x
   | Node(_,_,rt) -> tree_max rt
 end

let rec delete (t:tree) (n:int) : 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
```