CIS 120 Midterm I February 16, 2015

Name (printed): ___________________________

Username (login id): ___________________________

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: ___________________________ Date: ___________________________

1	/18
2	/16
3	/14
4	/6
5	/18
6	/18
7	/10
Total	/100

- Do not begin the exam until you are told to do so.
- You have 50 minutes to complete the exam.
- There are 100 total points.
- Make sure your name and username (a.k.a. PennKey, e.g. `sweirich`) is on the top of this page.
- Be sure to allow enough time for all the problems—skim the entire exam first to get a sense of what there is to do.

1. **Substitution semantics** (18 points)

Circle the final result of simplifying the following OCaml expressions, or "Infinite loop" if there is no final answer. Note: the definition of the `transform` function appears on page **??**. Also recall that the `@` operator appends lists together.

a.
```
let rec f (x :int) (m : int list) : bool =
  begin match m with
  | [] -> false
  | y :: t -> if x < y then f x t else false
  end
in
  f 3 [1;4;3]
```

- `true`
- `false`
- `[1;4;3]`
- `[true; false; false]`
- Infinite loop

b.
```
let rec g (x :int) (m : int list) : bool =
  begin match m with
  | [] -> false
  | y :: t -> if x <= y then true else g x (x :: t)
  end
in
  g 3 [6;4;3]
```

- `true`
- `false`
- `[6;4;3]`
- `[false; false; false]`
- Infinite loop

c.
```
let j (x : int) : int -> int list =
   fun (y:int) -> [x;y;x;y]
in
  j 3
```

- `3`
- `[3;3]`
- `[3;3;3;3]`
- `fun (y:int) -> [3;y;3;y]`
- Infinite loop

d.
```
let m (x : int) : (int * bool) =
    (x - 5, x > 5)
in
 transform m [6;4;3]
```

- `(1,true)`
- `[(1,true); (-1, false); (-2,false)]`
- `[1;-1;-2]`
- `[true; false; false]`
- Infinite loop

e.
```
let rec k (m : int -> int list) (x : int list) : int list =
 begin match x with
 | [] -> []
 | h :: t -> m h @ k m t
 end
in
 k (fun (x:int) -> [x;x]) [1;2;3]
```

- `[1;2;3]`
- `[1;1;2;2;3;3]`
- `[[1;1];[2;2];[3;3]]`
- `[[1];[2];[3]]`
- Infinite loop

f.
```
let rec h (m : int list list) : int list =
 begin match m with
 | [] -> []
 | [] :: u -> 0 :: h u
 | (x :: t) :: u -> 1 :: h (t :: u)
 end
in
 h [[1];[];[1;2]]
```

- `[1;0;0;1;1;0]`
- `[1;1;2]`
- `[1;0;1;1]`
- `[[1];[];[1;1]]`
- Infinite loop

2. **Types** (16 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 `ULSet`, and that this module has already been opened.

We have done the first one for you.

```
;; open ULSet

let z : _______ int list __________ = [1]


let a : ______________________________ = 1 :: 2 :: 3 :: []


let b : ______________________________ = 0::[[]]


let c : ______________________________ = [1] :: [2] :: [3] :: []


let d : ______________________________ = (fun x -> x :: []) 3


let e : ______________________________ = (fun x -> fun y -> x :: y) 3


let f : ______________________________ = add 3


let g : ______________________________ = [add 1 empty]


let h : ______________________________ = add 3 [1]
```

3. Binary Search Trees (14 points)

Page **??** shows an implementation of the `SET` interface using BSTs. This implementation preserves the Binary Search Tree invariant.

a. What is the result of the `is_bst` function (shown on page **??**) applied to the following trees? Note that we have omitted the `Empty` nodes from these pictures, to reduce clutter. Circle either **`true`** or **`false`**.

i.

```
        5
       / \
      2   7              true        false
     / \
    1   6
```

ii.

```
        3
       / \
      2   7              true        false
     /
    2
```

iii.

```
      2
       \
        3                true        false
         \
          4
```

iv.

```
      4
     / \
    3   7                true        false
       / \
      6   8
```

b. Suppose we change the `member` function (see page **??**) to the following:

```
let rec member (n:'a) (t:'a tree) : bool =
  begin match t with
    | Empty -> false
    | Node (lt, x, rt) ->
      if x = n then true
      else if member n lt then true
      else member n rt
  end
```

This implementation calculates the *same* answer as the original definition of `member`, but the old version is better. Briefly explain why, noting not just how this version is different but also why this difference is important.

c. Now suppose we change the `member` function (see page **??**) to the following:

```
let rec member (n:'a) (t:'a tree) : bool =
  if not (is_bst t)
  then failwith "input is not a binary search tree"
  else begin match t with
    | Empty -> false
    | Node (lt, x, rt) ->
      if x = n then true
      else if n < x then member n lt
      else member n rt
  end
```

This implementation calculates the *same* answer as the original definition of `member`, but the old version is better. Briefly explain why, noting not just how this version is different but also why this difference is important.

For the last four problems you will extend the `SET` interface (shown on page **??**) with the following new function:

```
  (* Return a new set containing all elements of the
   * given set that are strictly less than the specified
   * element. *)

 val prefix : 'a -> 'a set -> 'a set
```

For example, the prefix of the set $\{4, 1, 3, 2\}$ with respect to the element 3 is the set $\{1, 2\}$. Note that the specified element may or may not be contained within the set.

4. Test-driven development (6 points)

Using the set interface above, write two test cases that specify the behavior of `prefix`. You may assume that a module implementing this interface has been opened and all of the functions from that module are in scope. Make sure that you provide informative names for the tests too!

For example, one test case that you might write is:

```
let test () : bool =
  equals (prefix 3 empty) empty
;; run_test "prefix of an empty set is empty" test
```

Write your two tests below:

```
let test () : bool =






;; run_test "________________________________________" test

let test () : bool =






;; run_test "________________________________________" test
```

5. List recursion and invariants (18 points)

Recall the `OLSet` implementation of the `SET` interface from homework 3, which represents sets using ordered lists that do not contain duplicates. For reference, part of this implementation appears on page **??**.

Implement the `prefix` function for this module below. Your solution *must* take advantage of the representation invariant to avoid extra work, and must return a list that satisfies the representation invariant for this module. Your solution **must be recursive** and **cannot call any helper functions**. In particular, you cannot call helper functions that you write yourself or a functions from the `OLSet` module in your implementation.

```
let rec prefix (x : 'a) (l : 'a list) : 'a list =
```

6. Tree recursion and invariants (18 points)

Recall the `BSTSet` implementation of the `SET` interface, which represents sets using Binary Search Trees. This implementation maintains the BST invariant. For reference, part of this implementation appears in on page **??**.

Implement the prefix function for this module below. Your solution *must* take advantage of the representation invariant to avoid extra work, and must return a tree that satisfies the BST invariant. Your solution **must be recursive** and **cannot call any helper functions**. In particular, you cannot call helper functions that you write yourself or a functions from the `BSTSet` module in your implementation.

```
let rec prefix (x: 'a) (t : 'a tree) : 'a tree =
```

7. Higher-order functions (10 points)

Recall the `ULSet` implementation of the `SET` interface, which also represents sets using lists. However, this implementation does not maintain any invariant. For reference, part of this implementation appears on page **??**.

Implement the prefix function for this module below. In this case, your answer **cannot be recursive** and **must use one of the higher-order functions** shown on page **??**. You may define your *own* helper function in this problem, but you may not use any others, such as from the `ULSet` module.

```
let prefix (x : 'a) (l : 'a list) : 'a list =
```

Appendix: Higher-order functions

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

  let rec for_all (pred: 'a -> bool) (l: 'a list): bool =
   begin match l with
   | [] -> true
   | h :: t -> pred h && for_all pred t
   end

  let rec filter (f: 'a -> bool) (l: 'a list) : 'a list =
   begin match l with
   | [] -> []
   | h :: t -> if f h then h :: filter f t else filter f t
  end
```

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: unordered-list SET implementation

Sets implemented via lists. This implementation does not maintain any invariants about its representation. Only part of this implementation is shown.

```
module ULSet : SET = struct

  type 'a set = 'a list

  let empty : 'a set = []

  let add (x: 'a) (s: 'a list) : 'a list =
    x :: s

  ...
end
```

Appendix: ordered-list SET implementation

Ordered lists sets. This implementation maintains the invariant that all elements are stored in a list, without duplicates, in ascending order. Only part of this implementation is shown.

```
module OLSet : SET = struct

  type 'a set = 'a list

  let empty : 'a set = []

  let rec add (x: 'a) (s: 'a list) : 'a list =
    begin match s with
    | [] -> [x]
    | y :: ys ->
        if x = y then s
        else if x < y then x :: s
        else y :: add x ys
    end

  ...
end
```

Appendix: Binary Search Tree SET implementation

Sets based on Binary Search Trees. This implementation maintains the invariant that all trees satisfy the BST invariant. Only part of this implementation is shown.

```
module BSTSet : SET = struct

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

  type 'a set = 'a tree

  let empty : 'a tree = Empty

  let rec member (n:'a) (t: 'a tree) : bool =
    begin match t with
    | Empty -> false
    | Node (lt, x, rt) ->
       if x = n then true
       else if n < x then member n lt
       else member n rt
    end

  let rec tree_lt (t: 'a tree) (max: 'a) : bool =
    begin match t with
    | Empty -> true
    | Node (lt, v, rt) -> v < max && tree_lt lt max && tree_lt rt max
    end

  let rec tree_gt (t: 'a tree) (min: 'a) : bool =
    begin match t with
    | Empty -> true
    | Node (lt, v, rt) -> min < v && tree_gt lt min && tree_gt rt min
    end

  let rec is_bst (t: 'a tree) : bool =
    begin match t with
    | Empty -> true
    | Node (lt, v, rt) -> is_bst lt && is_bst rt && tree_lt lt v && tree_gt rt v
    end

   ...

end
```