CIS 120 Midterm I   February 16, 2015

SOLUTIONS

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 11. 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

Grading Scheme: 3 points each

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 11 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 : __ int list ________________ = 1 :: 2 :: 3 :: []

let b : __ ill typed _______________ = 0::[[]]

let c : __ int list list __________ = [1] :: [2] :: [3] :: []

let d : __ int list ________________ = (fun x -> x :: []) 3

let e : __ int list -> int list __ = (fun x -> fun y -> x :: y) 3

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

let g : __ int set list ___________ = [add 1 empty]

let h : ___ ill typed ______________ = add 3 [1]
```

Grading Scheme: 2 points each.

3. Binary Search Trees (14 points)

Page 13 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 13) 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
```

`false`

ii.

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

`false`

iii.

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

`true`

iv.

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

`true`

Grading Scheme: 2 points each

b. Suppose we change the `member` function (see page 13) 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.

Answer: This implementation doesn't take advantage of the BST invariant that is guaranteed by the module. As a result, it may need to search the entire tree instead of just one branch.

Grading Scheme: 3 points: observes that (a) this version does not take advantage of the BST invariant and (b) therefore must search the entire tree instead of just one path from the root to the leaves. 2 points: answers that make sense but don't mention both points. 1 point: some attempt at an answer that doesn't say anything wrong. 0 points: blank, or makes a claim that is incorrect: says that this version is better, or says that this version calculates the wrong answer.

c. Now suppose we change the `member` function (see page 13) 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.

Answer: The checks to see if the tree is a bst is both redundant (as the tree can be assumed to be a BST because that is the module invariant) and computationally very expensive (as the check is done with each recursive call, the subtrees will be rechecked multiple times).

Grading Scheme: 3 points: observes that (a) this version does not need to check the BST invariant because it is preserved by the module interface and (b) this check is really expensive because it is done at each recursive call. 2 points: answers that make sense but don't mention both points (i.e. notes redundancy but don't observe how expensive the check is). 1 point: some attempt at an answer that doesn't say anything wrong. 0 points: blank, or makes a claim that is incorrect: i.e. says that this version is better, or says that this version calculates the wrong answer.

For the last four problems you will extend the `SET` interface (shown on page 11) 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 =
 equals (prefix 1 (set_of_list [1;2])) empty
;; run_test "all elements removed" test
let test () : bool =
 equals (prefix 3 (set_of_list [1;2])) (set_of_list [1;2])
;; run_test "all elements retained" test
```

   Grading Scheme: 2 points - correct test, 1 point - interesting test and descriptive string.

   Good tests include coding the provided informal test from the problem description, a case where all elements are bigger, a case where all elements are smaller. The test cases must use `equals` *to compare sets, not* = *to be correct. The test case should not assume a particular implementation of sets, such as lists.*

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 12.

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 =
  begin match l with
  | [] -> []
  | h :: t -> if x <= h then [] else h :: prefix x t
  end
```

Grading Scheme: Note that the use of other functions, such as `add`*, was specifically disallowed by the instructions.*

- *2 pattern match l*
- *2 case for []*
- *2 compare x and h*
- *2 correct behavior when x = h (perhaps included in below)*
- *4 correct behavior when x < h (deduct 3 points if there is a recursive call to prefix in this case)*
- *6 correct behavior when x > h cons h, recursive call exists, correct args to recursive call*
- *No deduction for minor syntax errors*

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 13.

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 =
 begin match t with
  | Empty -> Empty
  | Node (lt,h,rt) ->
    if x < h then prefix x lt
        else if x > h then Node (lt, h, prefix x rt)
        else lt
```

Grading Scheme:

Note that the correct answer uses the BST invariant in several ways: when the argument `x` *is less than the value at the node* `h`*, the entire right subtree can be pruned entirely, when the argument is greater, then there is no need to call prefix recursively on the left subtree (all elements are known to be smaller) and when the argument* `x` *is equal then the left tree can just be returned (again as all elements are known to be smaller).*

- *2 pattern match on t and case for* `Empty`
- *2 compare* `x` *and* `h`
- *4 correct code when* `x = h`
- *4 correct recursive call when* `x < h` *(recursive call exists, correct args to prefix,* `rt discarded`*)*
- *6 correct case when* `x > h` *recursive call exists, correct args to prefix, constructs node with result*
- *No deduction for minor syntax errors as long as they are unambiguous*

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 12.

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 11. 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 =
  filter (fun (h : 'a) -> h < x) l
```

or

```
let prefix (x : 'a) (l : 'a list) : 'a list =
  let helper (h : 'a) : bool = h < x in
  filter helper l
```

or

```
let helper (x : 'a) (h : 'a) : bool = h < x in
let prefix (x : 'a) (l : 'a list) : 'a list =
  filter (helper x) l
```

Grading Scheme:

- *2 points for picking the right HOF (filter)*
- *2 points for defining any helper function (does not need to be an anonymous function, could be defined inside or outside* `prefix`*).*
- *2 points for the helper function having the right type*
- *2 points for the helper function calculating the correct result*
- *2 points for providing the* `l` *argument to* `filter`

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
```