CIS 120 Midterm I     October 3, 2014

# SOLUTIONS

## Problem 1: Binary Search Trees (10 points)

Recall the definition of generic binary trees and the BST `insert` and `lookup` functions:

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

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
```

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

(a) Circle the trees that satisfy the *binary search tree invariant*. (Note that we have omitted the `Empty` nodes from these pictures, to reduce clutter.)

```
(a)          (b)         (c)        (d)        (e)

  1            4            4         7         4
 / \          / \          / \       / \       / \
0   2        3   6        1   5     4   \     5   6
            / \            \       / \   \         \
           1   3            2     0   5   9         7
                           /             /           \
                          3             8             8
```

ANSWER: `(a)` and `(d)` are BSTs. One point per tree.

(b) Suppose you create a BST called `big_tree` by inserting a list of one million distinct integers (called `a_million_ints`) into the `Empty` tree, like this:

```
let rec insert_list (l: 'a list) (t: 'a tree) : 'a tree =
  begin match l with
  | [] -> t
  | hd::tl -> insert_list tl (insert hd t)
  end
let big_tree : int tree = insert_list a_million_ints Empty
```

In general, would you expect `(lookup x big_tree)` to run faster if `a_million_ints` was a *sorted* list or a *randomly-ordered* list? In one sentence, explain why.

ANSWER: A randomly-ordered list would perform better because a sorted list creates a skewed, list-like tree, so the BST search would not be able to prune the search space.

*Grading Scheme: 2 points for correctly identifying "random", 3 points for the explanation, 2 or 3 points for "almost" correct other explanations*

## Program Design: Abstract Vectors

***Note: there are no questions on this page, but the concepts are used in the remainder of the exam.***

Recall that in Homework 4, the N-body simulation used a representation of *vectors* that was fixed to be the tuple type `float * float`. This representation is sufficient for representing points, like `(1.0, 2.0)`, that live in a 2-dimensional space, but it turns out that many modern algorithms (particularly those in web search and machine learning) represent data as very high-dimensional vectors, often with thousands or tens-of-thousands of coordinates. In the next few problems, we will develop (parts of) a library for an abstract type, `vector`, suitable for working with high dimensional floating-point data.

**Understanding the Problem** Instead of just two coordinates, an abstract vector's coordinates are numbered by the non-negative `int` values: `0`, `1`, `2`, `3`, .... A vector `v` assigns a floating-point value to *every* such coordinate, so, conceptually, we can think of an abstract vector as an "infinitely wide" tuple.[1]

To (informally) write down example vectors, we'll use a tuple-like notation, but also require them to end with "*0.0 . . .*", which stands for an infinitely repeating sequence of `0.0` values. For example, the notation *(1.0, 2.0, 0.0 . . .)* means the vector with coordinate `0` equal to `1.0`, coordinate `1` equal to `2.0`, and all larger coordinates equal to `0.0`.

**Exploiting Sparsity** In many applications of very high-dimensional vectors, the vectors are *sparse*, which means that almost all of the coordinates are `0.0`—there are just a few non-zero values. For instance, a typical sparse vector might have just coordinate `120` set to the value `2.0`, coordinate `341` set to the value `1.0`, and all other coordinates set to `0.0`. Pictorially:

$$( \overbrace{0.0, 0.0, ..., 0.0}^{\text{coordinates 0–119}}, 2.0, \overbrace{0.0, 0.0, ..., 0.0}^{\text{coordinates 121–340}}, 1.0, 0.0, \ldots )$$

Sparsity suggests that we represent these vectors as lists of coordinate/value pairs, where all of the `0.0` values are omitted. Concretely, the example above could be represented by the OCaml value:

```
let v : (int * float) list = [(120, 2.0); (341, 1.0)]
```

The first component of each pair in the list is the index into the vector; the second component is the non-zero value at that index. It is also helpful for computations over vectors to require that the lists be sorted by coordinates. This leads to the datatype definition and invariant shown below in Figure 1.

```
(* INVARIANT: A sparse vector is a list of int * float pairs
 *  [ (i1, x1) ; (i2, x2) ; ... ; (iN, xN) ]
 *  such that
 *  (1)  the i's are sorted:  i1 < i2 < ... < iN
 *  (2)  none of x1, ..., xN are 0.0                *)
type vector = (int * float) list
```

Figure 1: Sparse vector representation invariant

---

[1] In practice, the width is limited by the largest `int` value, but for the purposes of this problem, we'll assume there's no such upper limit.

### Define the interface

We first need a way to create and access vectors, which can be done using these three operations:

| | |
|---|---|
| `zero` | the vector all of whose coordinates are `0.0`, namely: *(0.0 . . .)* |
| `set v i x` | returns a vector that is the same as `v` except at coordinate `i`, where it has value `x` |
| `get v i` | returns the value at the `i`$^{th}$ coordinate of `v` |

Their types are given by:

```
let zero : vector = ...
let set (v: vector) (i: int) (x: float) : vector = ...
let get (v: vector) (i: int) : float = ...
```

### Problem 2: Write Test Cases (10 points)

Complete these test cases by filling in the concrete value that should make the test case pass according to the sparse-vector representation invariant. We have done the first one for you:

(a)
```
let test () : bool =
  (set (set zero 120 2.0) 341 1.0) = ______[(120, 2.0); (341, 1.0)]________
;; run_test "given example" test
```

(b)
```
let test () : bool =

  zero = ______________[]____________________
;; run_test "zero representation" test
```

(c)
```
let test () : bool =
  (set (set zero 341 1.0) 120 2.0) = ______[(120, 2.0); (341, 1.0)]________
;; run_test "given example, other order" test
```

(d)
```
let test () : bool =
 let v : vector = set (set zero 1 1.0) 2 2.0 in

 set v 1 0.0 = ___________[(2, 2.0)]______________
;; run_test "set non-zero to zero" test
```

(e)
```
let test () : bool =

 get zero 27 = _______________0.0_________________
;; run_test "get zero 27" test
```

(f)
```
let test () : bool =
 let v : vector = set (set zero 2 1.0) 2 2.0 in

 get v 2 = _______________2.0_________________
;; run_test "get v 2" test
```

*Grading Scheme: 2 points each*

## Problem 3: Implement the Behavior (20 points)

The code below implements the `get` operation for sparse vectors. Note how it uses the representation invariant, which requires that the list be sorted by coordinate, to determine whether to return the (implicit) `0.0` value:

```
(* Returns the value of the i'th coordinate of v *)
let rec get (v: vector) (i: int) : float =
  begin match v with
    | [] -> 0.0
    | (j,y)::tl ->
      if i < j then 0.0          (* Coordinate i must be 0.0 since it's not in the list *)
      else if i = j then y       (* Found a non-zero value *)
      else get tl i              (* Note: i > j *)
  end
```

Using the code for `get` as a model, complete the implementation of the `set` operation. Remember that it must maintain the invariant—the list must be sorted by coordinate and may not contain any `0.0` values.

```
(* Returns a vector that is the same as v except at coordinate i, where it has value x *)
let rec set (v: vector) (i: int) (x: float) : vector =
  begin match v with
    | [] -> if x = 0.0 then [] else [(i, x)]
    | (j,y)::tl ->
      if    i < j then (if x = 0.0 then v else (i,x)::v)
      else if i = j then (if x = 0.0 then tl else (i,x)::tl)
      else (j,y)::(set tl i x) (* Note: i > j *)
  end
```

*Grading Scheme:*

- *3 pts. proper checks for* `x = 0.0`
- *base case: 1 pt.*
- `i < j` *case: 4 pts.*
- `i = j` *case: 4 pts.*
- `i > j` *case: 4 pts. properly using recursion, another 4 pts. for maintaining the list*
- *flipped comparison: -4 pts.*
- *failing rather than maintaining the invariant 1 out of the 3 pts.*
- *small syntax errors -0.5 or -1 pts.*

## Problem 4: Vector Dot Product (20 points)

One very common operation on vectors is called the *dot product*. (Don't worry if you haven't heard of this before, the idea is pretty simple.) This operation takes two vectors, multiplies the values at corresponding coordinates, and sums the results. For example, if

`v` is the vector *(2.0, 3.0, 0.0 . . .)* sparsely represented as: `[(0, 2.0); (1, 3.0)]`
and `u` is the vector *(0.0, 4.0, 5.0, 0.0 . . .)* sparsely represented as: `[(1, 4.0); (2, 5.0)]`

then their dot product is given by:

```
    (2.0 *. 0.0)  +. (3.0 * 4.0)  +. (0.0 *. 5.0)  +. (0.0 *. 0.0) ...
  = 0.0           +. 12.0         +. 0.0           +. 0.0...
  = 12.0
```

Sparse vectors are especially good for implementing dot product because multiplying by `0.0` doesn't contribute to the sum, and the algorithm has a naturally recursive structure. Since the sparse representation doesn't even contain the `0.0` values, we can exploit the invariant to efficiently carry out the computation, but we have to be careful to multiply the values at the *same* coordinates from each vector. In the example, this means identifying that both `v` and `u` have non-zero values at index `1` and adding their product (`3.0 *. 4.0`) to the running total.

Your task is to complete the implementation of vector `dot_prod`. We have given you the **`match`** expression to get you started. How to complete the case analysis is up to you. Note that using the `get` operation repeatedly could be very inefficient, so *do not use it* .

```
let rec dot_prod (v: vector) (u: vector) : float =
  begin match (v, u) with
    | ([], _) -> 0.0
    | (_, []) -> 0.0
    | ((i,x)::vtl, (j,y)::utl) ->
      if i < j then dot_prod vtl u else
      if i = j then (x *. y) +. (dot_prod vtl utl)
      else dot_prod v utl
  end
```

*Grading Scheme:*
*Each base case: 1 pt. correct pattern, 2 pt. correct answer (total: 6)*
*Non-base case: 1 pt. correct pattern match, 2 pts. for each correct condition (there are three cases), 3 pts. for correct answer in* `i < j` *and* `i > j` *cases, 1 pt. for correct* `=` *case. (total: 14)*
*Other deductions:*

- *-1 globally for* `int` *instead of* `float`
- *only* `([], [])` *base case: 2 pts. instead of 6 pts.*
- *-10 if there is only the* `=` *case in the logic*
- *swapping* `i < j` *and* `i > j`*: 2 out of 6 pts. for the conditions*
- *checking equality on tuple rather than index: -1 for each condition*
- *miscellaneous other deductions for incorrect code*

## Problem 5: Types and Abstraction (16 points)

Continuing to develop the sparse vector library, we can package it into a module as shown below, where we have added several other vector operations whose implementations are not shown.

```
module type VECTOR = sig
  type vector

  val zero : vector
  val set : vector -> int -> float -> vector
  val get : vector -> int -> float
  val add : vector -> vector -> vector
  val scale : float -> vector -> vector
  val dot_prod : vector -> vector -> float
  val equals : vector -> vector -> bool
end

module SparseVector : VECTOR = struct
  type vector = (int * float) list
  ... (* Implementation of the vector operations omitted *)
end
```

Figure 2: `VECTOR` interface and the `SparseVector` implementation of it.

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. These declarations take place in a top-level module called `client.ml` *after* the command `;; open SparseVector`. We have done the first one for you.

```
(* client .ml *)
;; open SparseVector

let z : ____________________ float ______________________ = get zero 0

let a : __________________ ill typed ____________________ = get [(1, 0.0)] 1

let b : __________________ vector * float ______________ = (zero, 0.0)

let c : ________ (vector -> vector) list ______________ = [(add zero); (scale 1.0)]

let d : ___________________ ill typed __________________ = zero::[[]]

let e : ___________________ ill typed __________________ = [add; scale; equals]

let f : __________ int list -> int list _______________ = (fun x -> fun y -> x :: y) 3

let g : _______________ vector -> bool ________________ = (fun x -> equals x x)

let h : ___________________ int option _________________ = Some 0
```

*Grading scheme: 2 points per answer. Half credit for omitting necessary parentheses (such as int * bool list vs. (int * bool) list or* `int -> int list` *vs.* `(int -> int) list` *). Half credit for types that are too generic. Half credit for other "almost correct" answers.*

## Problem 6: Higher-order function patterns (24 points)

Recall the functions `transform` and `fold` discussed in lecture and used in HW04:

```
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 fold (combine: 'a -> 'b -> 'b) (base:'b) (x : 'a list) : 'b =
 begin match x with
 | [] -> base
 | h :: t -> combine h (fold combine base t)
 end
```

Because the representation type of sparse vectors is the list type (shown below):

```
type vector = (int * float) list
```

we can use higher-order functions to implement operations of the vector library. In each case below, rewrite the given function to make use of either `transform` or `fold` as appropriate. You can introduce a helper function (such as `xyz_combine`), or use an anonymous function.

(a)
```
(* Calculates the number of non-zero entries in a sparse vector. *)
 let rec num_nonzeros (v: vector) : int =
  begin match v with
    | [] -> 0
    | _::tl -> 1 + (num_nonzeros tl) (* Each list element represents a non-zero coordinate *)
  end
```

```
let num_nonzeros (v: vector) : int =
  fold (fun (x:(int * float)) (acc:int) -> 1 + acc) 0 v
```

*Grading Scheme: 6 pts.:*

- *Not using* `fold`*: 0 pts.*
- *1 pt. for* `fold` *and passing in list* `v`.
- *base case: 1 pt. for correct* `int`*, 0.5 pt for* `float`
- *1 pt. for each arguments: type and syntax flexible*
- *2 pts. computation of the body*

(b)
```
(* Scale a vector by multiplying each coordinate's value by c.
   Maintains the no-zeros invariant by returning [] when scaling by 0.0. *)
let scale (c: float) (v: vector) : vector =
  let rec multiply_all (v: vector) : vector =
    begin match v with
      | [] -> []
      | (i,x)::tl -> (i, c *. x)::(multiply_all tl)
    end
  in
  if c = 0.0 then [] else multiply_all v


let scale (c: float) (v: vector) : vector =
  if c = 0.0 then [] else
  transform (fun (i, x) -> (i, c *. x)) v

or

let scale (c: float) (v: vector) : vector =
  let m (p:(int * float)) : (int * float) =
   match p with
    | (i, x) -> (i, c *. x)
  end
in if c = 0.0 then [] else transform m v
```

*Grading Scheme: 8 pts.:*

- *1 pt* `transform` *and* `v`.
- *2 pt. doing the invariant check*
- *1 pt. correct parameter*
- *1 pt. for deconstructing the tuple (matching or using* `fst` *or* `snd`*)*
- *1 pt. for doing* `c *. x`
- *2 pt. for returning a tuple (half credit for wrong order)*

(c)
```
(* Returns the value of the i'th coordinate of v *)
let rec get (v: vector) (i: int) : float =
  begin match v with
    | [] -> 0.0
    | (j,y)::tl ->
      if i < j then 0.0          (* Coordinate i must be 0.0 since it's not in the list *)
      else if i = j then y       (* Found a non-zero value *)
      else get tl i              (* Note: i > j *)
  end


let get (v: vector) (i: int) : float =
  fold (fun (j,y) acc -> if i < j then 0.0 else if i = j then y else acc) 0.0 v

or

let get (v: vector) (i: int) : float =
```

```
let get_combine (i:int) : (int * float) -> float -> float =
  fun (j, y) -> fun (acc: float) ->
    if i < j then 0.0
    else if i = j then y
    else acc
in
fold (get_combine i) 0.0 v
```

*Grading Scheme: 10 pts.:*

- *1 pt* `fold` *and* `v`*.*
- *2 pts. correct parameters*
- *Base case: 1 pt. (0.5 for* `int` *instead of* `float`*)*
- *6 pts. for body logic: There are either three cases:*
  - `i = j` *case: return val (2 pts.)*
  - `i < j` *case: return* `0.0` *(2 pts.)*
  - `i > j` *case: return accumulator (2 pts.)*

  *or there are two cases*
  - `i = j` *case: return val (3 pts.)*
  - `i > j` *case: return accumulator (3 pts.)*