# CIS 120 Midterm I February 10, 2017

Name (printed): ___________________________________

PennKey (penn 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: ________________________

- Do not begin the exam until you are told it is time to do so.
- Make sure that your username (a.k.a. PennKey, e.g. `stevez`) is written clearly at the bottom of *every page*.
- There are 100 total points. The exam lasts 50 minutes. Do not spend too much time on any one question.
- Be sure to recheck all of your answers.
- The last page of the exam can be used as scratch space. By default, we will ignore anything you write on this page. If you write something that you want us to grade, make sure you mark it clearly as an answer to a problem *and* write a clear note on the page with that problem telling us to look at the scratch page.
- Good luck!

## 1. Binary Search Trees

This problem concerns a *buggy* implementation of the `lookup` and `insert` functions for binary search trees.

Although the implementations below are incorrect, there are still some inputs for which they do work correctly. 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
```

where `Empty` nodes are not shown, to avoid clutter.

**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))
```

**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))
```

## 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
```

Each part of this problem below begins with a sample function written using simple recursion over lists, followed by several alternatives written using `transform` or `fold`. In each part, mark *all* of the alternatives that implement the same behavior as the recursive sample. *There may be zero, one, or more than one such function.* Some of the alternatives may not typecheck—do not mark these.

**a.** (6 points)

```
let rec strings_of_ints (lst: int list) : string list =
  begin match lst with
  | [] -> []
  | hd :: tl -> string_of_int hd :: strings_of_ints tl
  end
```

☐
```
let strings_of_ints (lst: int list) : string list =
  transform string_of_int lst
```

☐
```
let strings_of_ints (lst: int list) : string list =
  transform (fun s -> string_of_int s) lst
```

☐
```
let strings_of_ints (lst: int list) : string list =
  fold (fun s acc -> string_of_int s :: acc) [] lst
```

**b.** (6 points)

```
let rec dupl (lst: 'a list) : 'a list =
 begin match lst with
  | [] -> []
  | x::xs -> x :: x :: dupl xs
 end
```

☐
```
let dupl (lst: 'a list) : 'a list =
  fold (fun x acc -> [x;x] @ acc) [] lst
```

☐
```
let dupl (lst: 'a list) : 'a list =
  fold (fun x acc -> x :: x :: acc) [] lst
```

☐
```
let dupl (lst: 'a list) : 'a list =
  transform (fun x -> (x,x)) lst
```

**c.** (6 points)

```
let rec remove_all (n : 'a) (lst : 'a list) : 'a list =
  begin match lst with
   | [] -> []
   | h :: tl -> let rest = remove_all n tl in
              if h = n then rest else h :: rest
  end
```

☐
```
let remove_all (n: 'a) (lst: 'a list) : 'a list =
   fold (fun x acc -> if x = n then acc else x :: acc) [] lst
```

☐
```
let remove_all (n: 'a) (lst: 'a list) : 'a list =
   transform (fun x -> if x = n then [] else x) lst
```

☐
```
let remove_all (n: 'a) (lst: 'a list) : 'a list =
   fold (fun x acc -> if x = n then acc else lst) [] lst
```

☐
```
let remove_all (n: 'a) (lst: 'a list) : 'a list =
   fold (fun x acc -> if x = n then acc else x :: acc) 0 lst
```

**3. Types** (16 points)

For each OCaml value below, fill in the blank for the type annotation or else write "ill typed" if there is a type error on that line. Your answer should be the *most generic* type for the value—i.e. if `int list` and `bool list` are both possible types of an expression, you should write `'a list`.

We have done the first one for you.

```
let z : __________ 'a list list _________ = [[]]
```

```
let b : ____________________________________ = [] :: [] :: []
```

```
let c : ____________________________________ = begin match [] with
                                                 | [] -> "foo"
                                                 | h::t -> h+3
                                                 end
```

```
let d : ____________________________________ = transform (fun x -> x>2)
```

```
let e : ____________________________________ = fold (fun x y -> x::y)
```

```
let f : ____________________________________ = (fun x -> fun y -> x) 42
```

```
let g : ____________________________________ = let f x = [x] in f (f 5)
```

```
let h : ____________________________________ = let sum x y = x+y in
                                                  transform (transform sum)
```

**4. Abstract Types, Invariants, and Modularity**

A *priority queue* is a data structure whose job it is to maintain a collection of elements, each associated with a numeric *priority* telling how urgent it is. For example, the items in the queue could represent patients waiting to be seen by an emergency-room doctor and the priorities could indicate which patients require the most urgent attention. In this simplified presentation, priority queues come with just four operations:

- `empty` is a constant representing an empty priority queue;
- `add` takes a priority queue, a priority, and a new item, and adds the item and its priority to the queue;
- `largest` takes a queue and returns the item with the largest ( = most urgent) priority;
- `droplargest` takes a queue and returns a new queue in which the item with the largest priority has been removed.

Here is a module giving a simple implementation of priority queues (the signature `PRIQUEUE` will be defined below):

```
module PriQueue : PRIQUEUE = struct
  type 'a priqueue = (int*'a) list

  let empty : 'a priqueue = []

  let rec add (pri: int) (item: 'a) (q: 'a priqueue) : 'a priqueue =
    begin match q with
    | [] -> [(pri,item)]
    | (pri',item')::q' ->
       if pri >= pri' then (pri,item) :: q
       else (pri',item') :: (add pri item q')
    end

  let largest (q: 'a priqueue) : 'a =
    begin match q with
    | [] -> failwith "largest called on empty priority queue"
    | (pri,item) :: q' -> item
    end

  let droplargest (q: 'a priqueue) : 'a priqueue =
    begin match q with
    | [] -> failwith "droplargest called on empty priority queue"
    | (pri,item) :: q' -> q'
    end
end
```

*Problem continues on following pages...*

**a.** (4 points) The above implementation relies on an invariant to make sure that `largest` and `droplargest` can be answered very quickly. Briefly state this invariant in English:

**b.** Here are several alternative versions of the `PRIQUEUE` signature mentioned in the `PriQueue` implementation above. For each one, please indicate whether it is a good signature for this module or, if not, in what way it is not good.

**i.** (2 points)

```
module type PRIQUEUE = sig
  type 'a priqueue
  val empty : 'a priqueue
  val add : int -> 'a -> 'a priqueue -> 'a priqueue
  val remove : 'a -> 'a priqueue -> 'a priqueue
  val largest : 'a priqueue -> 'a
  val droplargest : 'a priqueue -> 'a priqueue
end
```

Circle one:

**A.** Good (i.e., it is a reasonable interface for the module)

**B.** Not useful (i.e., with this interface, clients cannot use the module to do anything nontrivial)

**C.** Not safe (i.e., it allows clients to break the module's invariant)

**D.** Wrong (i.e., the `PriQueue` module will not compile with this signature)

**ii.** (2 points)

```
module type PRIQUEUE = sig
  type 'a priqueue
  val add : int -> 'a -> 'a priqueue -> 'a priqueue
  val largest : 'a priqueue -> 'a
  val droplargest : 'a priqueue -> 'a priqueue
end
```

Circle one:

**A.** Good (i.e., it is a reasonable interface for the module)

**B.** Not useful (i.e., with this interface, clients cannot use the module to do anything nontrivial)

**C.** Not safe (i.e., it allows clients to break the module's invariant)

**D.** Wrong (i.e., the `PriQueue` module will not compile with this signature)

**iii.** (2 points)

```
module type PRIQUEUE = sig
  type 'a priqueue
  val empty : 'a priqueue
  val add : int -> 'a -> (int * 'a) list -> 'a priqueue
  val largest : 'a priqueue -> 'a
  val droplargest : 'a priqueue -> 'a priqueue
end
```

Circle one:

**A.** Good (i.e., it is a reasonable interface for the module)

**B.** Not useful (i.e., with this interface, clients cannot use the module to do anything nontrivial)

**C.** Not safe (i.e., it allows clients to break the module's invariant)

**D.** Wrong (i.e., the `PriQueue` module will not compile with this signature)

**iv.** (2 points)

```
module type PRIQUEUE = sig
  type 'a priqueue
  val empty : 'a priqueue
  val add : int -> 'a -> 'a priqueue -> 'a priqueue
  val largest : 'a priqueue -> 'a
  val droplargest : 'a priqueue -> 'a priqueue
end
```

Circle one:

**A.** Good (i.e., it is a reasonable interface for the module)

**B.** Not useful (i.e., with this interface, clients cannot use the module to do anything nontrivial)

**C.** Not safe (i.e., it allows clients to break the module's invariant)

**D.** Wrong (i.e., the `PriQueue` module will not compile with this signature)

**c.** (12 points) Another useful operation on priority queues is `merge`, which takes two priority queues and yields a new queue containing all the items (and their priorities) from both. A reasonable signaure for `merge` is:

```
merge : 'a priqueue -> 'a priqueue -> 'a priqueue
```

Here is a skeleton for an implementation of `merge`. Please fill in what's missing.

```
let rec merge (q1: 'a priqueue) (q2: 'a priqueue) : 'a priqueue =
  begin match q1, q2 with

  | [], _ ->

  | _, [] ->

  | (pri1,item1)::q1', (pri2,item2)::q2' ->









  end
```

## 5. List Recursion and Program Design

For this problem, you will use the program design process to implement a function called `remove_n`. This function takes a list `lst`, an element `item` that may be in that list, perhaps multiple times, and a count `n` , and returns a list which is identical to the initial list, except that the first `n` instances of `item` have been removed. If there are less than `n` instances of `item` in `lst`, the function removes all instances from the list. If `item` isn't in the list, the returned list should be identical with the original list. The function should work on lists of any type.

**a.** (4 points) First, define the interface of your function. Write the type of your function as you might see it in a signature or `.mli` file.

```
val remove_n: ________________________________________________
```

**b.** (10 points) Now, write three *different* tests for `remove_n`. Put some thought into your answers; we will be grading your answers not just on correctness, but on how well your tests cover different aspects of its behavior. Don't forget to give each case a descriptive name. We've filled in one test for you.

```
;; run_test "remove_n from singleton list" (fun () ->
    remove_n "a" 1 ["a"] = [])
```

```
;; run_test "____________________________________" (fun () ->

    ____________________________________________________)
```

```
;; run_test "____________________________________" (fun () ->

    ____________________________________________________)
```

```
;; run_test "____________________________________" (fun () ->

    ____________________________________________________)
```

**c.** (16 points) Now, *implement* your function in the space provided below.

## 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.*