# CIS 120 Midterm I February 15, 2013

Name (printed): ____________________________
Pennkey (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: ____________________________

| | |
|---|---|
| 1ab | /15 |
| 1c | /12 |
| 2ab | /18 |
| 2cde | /11 |
| 3 | /16 |
| 4ab | /13 |
| 4c | /15 |
| 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.
- There are 9 pages in this exam.
- Make sure your name and Pennkey (a.k.a. username) 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. **Program Design (27 points total)**

Use the four-step design methodology to implement a function called `insert` that takes an `int` and a list of ints and inserts the number into the list at the first position where it is less than or equal to the next number. If the number is greater than all others in the list, it should be added to the end. If the list is sorted before the call, the result will also be sorted.

For example, `insert 3 [1; 2; 4; 5]` should yield the list `[1; 2; 3; 4; 5]`.

(0 points) Step 1 is *understanding the problem*. You don't have to write anything for this part—your answers below will demonstrate whether or not you succeeded with Step 1.

(3 points) Step 2 is *formalizing the interface*. Write down the *type* of the `insert` function as you might find it in a `.mli` file or module interface.

```
val insert:
```

(12 points) Step 3 is *writing test cases*. Complete the following three tests with the expected behavior. We have done the first one for you, based on the problem description.

Note that some test cases are better than others, and credit will be assigned accordingly: make sure your tests cover a sufficiently broad range of "interesting" input numbers and lists. Fill in the description string of the `run_test` function with a short explanation of *why* the test case is interesting. Your description should not just restate the test case, e.g. "insert 3 [1;2;4;5]".

**i.**
```
let test () : bool =
  insert 3 [1;2;4;5] = [1;2;3;4;5]
;; run_test "insert into middle of list" test
```

**ii.**
```
let test () : bool =

  (insert __________ ______________________) = __________________

;; run_test "__________________________________________________" test
```

**iii.**
```
let test () : bool =

  (insert __________ ______________________) = __________________

;; run_test "__________________________________________________" test
```

**iv.**
```
let test () : bool =

 (insert __________ ______________________) = __________________

;; run_test "__________________________________________________" test
```

(12 points) Step 4 is *implementing the program*. Fill in the body of the `insert` function to complete the design. Do *not* use any list library functions (such as `fold`, or `@`) to solve this problem. If you would like to use a helper function in your answer, you must define it.

```
let rec insert (x:____________) (lst:_______________) : _______________ =
```

2. **List recursion, higher-order functions and generic types (29 points total)**

This problem considers the following function, called `separate`.

```
let rec separate (v:int) (lst : int list) : int list * int list =
  begin match lst with
  | [] -> ([],[])
  | hd :: tl ->
    let (xs,ys) = separate v tl in
    if hd >= v then
      (xs, hd :: ys)
    else
      (hd :: xs, ys)
  end
```

**a.** (9 points) Complete the following test cases for `separate` so that they return true.

```
let test () : bool =

  separate 5 [] = ______________________________

let test () : bool =

  separate 5 [1;3;6;7] = ________________________

let test () : bool =

  separate 5 [1;5;6] = __________________________
```

**b.** (9 points) Now consider a version of `separate`, called `ho_separate`, that takes a higher-order function as an additional argument. Here are two test cases for this version.

```
let nonnegative (x:int):bool = x >= 0
let test () : bool =
   ho_separate nonnegative [-1; 1; 0; -2] = ([-1; -2],[1;0])

let positive (x:int):bool = x > 0
let test () : bool =
  ho_separate positive [-1; 0; 2; -2] = ([-1; 0; -2],[2])
```

Fill in the blanks to complete the implementation of `ho_separate`.

```
let rec ho_separate (f : ____________________________)
                    (lst : int list) : int list * int list =
  begin match lst with
  | [] -> ([],[])
  | hd :: tl ->

    let (xs,ys) = ______________________________ in


    if _________________ then
      (xs, hd :: ys)
    else
      (hd :: xs, ys)
  end
```

**c.** (4 points) Reimplement `separate` using `ho_separate` as a helper function. You should not use recursion—just call `ho_separate` with the appropriate arguments.

```
let separate (v:int) (lst : int list) : int list * int list =
```

**d.** (3 points) Now consider a different version of `separate`, called `generic_separate`. Here are two test cases for `generic_separate`.

```
let test () : bool =
  generic_separate "c" ["a";"b";"d"] = (["a";"b"],["d"])

let test () : bool =
  generic_separate 0.5 [0.0;0.7;0.2; 0.8] = ([0.0;0.2],[0.7;0.8])
```

(Note that `generic_separate` does *not* take a higher-order function as an argument.)

What is the interface to this function? Write the type as it might appear in a `.mli` file.

```
val generic_separate:
```

**e.** (4 points) Reimplement `separate` using `generic_separate` as a helper function. You should not use recursion—just call `generic_separate` with the appropriate arguments.

```
let separate (v:int) (lst : int list) : int list * int list =
```

3. **Types (16 points)**

For each OCaml value or function definition below, fill in the blank where the type annotation could go or write "ill typed" if there is a type error. If an expression can have multiple types, give the most generic one. Recall that the `@` operator appends two lists together in OCaml. We have done the first one for you. Consider the definitions to be below the following code:

```
module type SET = sig
  type 'a set
  val fromList : 'a list -> 'a set
end

module LSet : SET = struct
  type 'a set = 'a list
  let fromList (l : 'a list) = l
end

open LSet;;

let x : ______ string ______________ = "120 " ^ "is fun"


let a : ________________________________________ = "120" ^ 120

let b : ________________________________________ = [120] :: [120]

let c : ________________________________________ = 120 :: [120]

let d : ________________________________________ = (120, 120)

let e : ________________________________________ = [(120, 120)]

let f : ________________________________________ = fromList [120]

let g : ________________________________________ = fromList ([2] @ [3])

let h : ________________________________________ = (fromList [2]) @ (fromList [3])
```

## 4. Binary Search Trees (28 points total)

Recall the definition of generic binary trees and the binary search tree `insert` function:

```
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
```

**a.** (5 points) Circle the trees that satisfy the *binary search tree invariant*. (Note that we have omitted the Empty nodes from these pictures.)

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

  4         3         2        3       7
   \       / \       / \        \       \
    5     2   6     1   6       6       6
     \       /           \     / \       \
      6     5             4   2   7       5
       \                                   \
        7                                   4
```

**b.** (8 points) For each definition below, circle the letter of the tree above that it constructs or "none of the above".

```
let t1 : int tree =
 Node(Node(Empty, 1, Empty), 2, Node(Empty, 6, (Node (Empty, 4, Empty))))
```

(a) (b) (c) (d) (e) none of the above

```
let t2 : int tree =
 insert (insert (insert (insert Empty 4) 5) 6) 7
```

(a) (b) (c) (d) (e) none of the above

```
let t3 : int tree =
 insert (insert (insert (insert Empty 2) 5) 3) 6
```

(a) (b) (c) (d) (e) none of the above

```
let t4 : int tree =
  Node(Empty, 3, Node(Node (Empty, 2, Empty), 6, (Node (Empty, 7, Empty))))
```

`(a)` `(b)` `(c)` `(d)` `(e)` none of the above

**c.** (15 points) Complete the definition of a function `bst_separate` that, when given an integer x, separates a binary search tree into two parts. The first part should contain the values less than x, the second part should contain the values greater than or equal to x.

For example, when given the binary search tree `t`

```
      5
     / \
    3   6
   / \
  0   4
```

the result of `bst_separate 5 t` is the pair of binary search trees:

```
      3               5
     / \     and       \
    0   4               6
```

Your solution must take advantage of the binary search tree invariant to avoid traversing the entire tree and should not refer to any of the bst operations such as `insert`, `remove`, and `inorder`.

(Use the next page for your implementation.)

```
let rec bst_separate (x:int) (t : int tree) : int tree * int tree =
  begin match t with
   | Empty ->




   | Node (l, y, r) ->
     if x = y then









     else if x < y then









     else
```