Name: ________________________

CIS 120e Midterm I
October 15, 2010

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/18</td>
</tr>
<tr>
<td>2</td>
<td>/12</td>
</tr>
<tr>
<td>3</td>
<td>/12</td>
</tr>
<tr>
<td>4</td>
<td>/16</td>
</tr>
<tr>
<td>5</td>
<td>/12</td>
</tr>
<tr>
<td>6</td>
<td>/08</td>
</tr>
<tr>
<td>7</td>
<td>/22</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>/100</td>
</tr>
</tbody>
</table>

- Do not begin the exam until you are told to do so.
- You have 50 minutes to complete the exam.
- There are 10 pages in this exam.
- Make sure your name is on the top of this page.
1. Program Design (18 points total)

Use the four-step design methodology to implement a program that, given a list of integers, calculates the list of differences between adjacent elements of the list. For example: given the input list [1; 3; 5; 10; 17] the output should be [2; 2; 5; 7] because 3-1 = 2 and 5-3 = 2 and 10-5 = 5, etc.

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

(6 points) Step 2 is formalizing the interface. Complete the following function declaration template according to the problem description:

```ml
let rec diffs ________________ = ...
```

(6 points) Step 3 is writing test cases. Complete the following three assertions with examples of the expected behavior. Note that some test cases are better than others, and partial credit will be assigned accordingly: make sure your tests cover a sufficiently broad range of “interesting” inputs.

```ml
;; assert_eq "" ________________ ________________;;
;; assert_eq "" ________________ ________________;;
;; assert_eq "" ________________ ________________;;
```

(6 points) Step 4 is implementing the program. Fill in the body of the `diffs` function to complete the design:

```ml
let rec diffs (* your interface from above *) =
```
2. List Recursion (12 points)

For each of the following programs, circle the final value computed for r (choose each answer from the given bullet lists):

a. let l = [1;2;3]
   let rec foo (l:int list) : int = begin match l with
   | [] -> 0
   | h::t -> (h * h) + (foo t)
   end
   let r = foo l
   • 6
   • 14
   • 24
   • 36

b. let l = [1;2;3]
   let rec baz (l:int list) : int = begin match l with
   | [] -> failwith "baz doesn’t work on []"
   | h::t -> let x = baz t in
         if h > x then x else h
   end
   let r = baz l
   • Exception Failure: "baz doesn’t work on []"
   • 1
   • 2
   • 3

c. let l = [1;2;3]
   let rec moo (l:int list) : int list = begin match l with
   | [] -> []
   | h::t -> moo t @ [h] @ moo t
   end
   let r = moo l
   • [1;2;3;1;2;3]
   • [1;2;1;3;1;2;1]
   • [3;2;3;1;3;2;3]
   • [1;2;3;1;3;2;1]
3. Types (12 points)

a. int 
   f. int ref 
   k. unit 

b. int list 
   g. int list ref 
   l. unit ref 

c. 'a list 
   h. int ref list 

d. int -> int 
   i. int option 

e. int list -> int 
   j. 'a option 

For each OCaml expression below, write down its type or write “ill typed” if there is a type error. If more than one type fits, choose the most generic one.

a. ________________ ref [1;2;3] 

b. ________________ (ref 0) + (ref 5) 

c. ________________ [] @ [] 

d. ________________ let y = 4 in fun (x:int) -> y 

e. ________________ let x = ref 0 in x := 1 

f. ________________ let f (x:int list) = 3 in f []
4. Binary Search Trees (16 points)

Recall the definition of generic binary trees:

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

a. (8 points) State the binary search tree invariant. Briefly explain one program-design strategy for ensuring that clients of the BST operations cannot break that invariant.

b. (8 points) Complete the definition of the `insert` function for binary search trees. You may assume that the input tree `t` satisfies the BST invariant as shown in class; the output tree must also satisfy the BST invariant.

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

5. First-class Functions (12 points)

Recall the definition of fold over lists:

```ml
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
```

For each of the following functions, choose the combination of base and comb arguments that should be given to fold to implement the desired functionality.

a. flatten takes a list of lists and returns their concatenation.
   let flatten(l:'a list list) : 'a list = fold comb base l
   base should be:   comb should be:
   • []  • fun (x:'a list) (r:'a list) -> x @ r
   • [[]]  • fun (x:'a) (r:'a list) -> x :: r
   • [x]  • fun (x:'a list) (r:'a list list) -> [x]::r
   • None  • fun (x:'a) (r:'a list) -> [x]@r
b. maximum returns None if the list is empty and Some $n$ otherwise, where $n$ is the maximum element of the list.

\[
\text{let maximum (l:'a list) : 'a option = fold comb base l}
\]

base should be:  comb should be:

- $\emptyset$  
  \[
  \text{fun (h:'a) (r:'a) -> if h > r then h else r}
  \]

- $x$  
  \[
  \text{fun (h:'a) (r:'a option) -> begin match r with}
  
  | None -> Some h
  
  | Some x -> if h > x then Some h else None end}
  \]

- Some $x$  
  \[
  \text{fun (h:'a) (r:'a option) -> begin match r with}
  
  | None -> Some h
  
  | Some x -> if h > x then Some h else Some x end}
  \]

- None  
  \[
  \text{fun (h:'a) (r:'a option) -> begin match r with}
  
  | None -> Some h
  
  | Some x -> if h > x then Some x else Some h end}
  \]
6. Generic Programming and First-class Functions (8 points)

Implement a function \texttt{comp} that has the following type.

\[
\texttt{comp}: \ ('b \rightarrow \ 'c) \rightarrow (\ 'a \rightarrow \ 'b) \rightarrow \ 'a \rightarrow \ 'c
\]

Complete the definition of \texttt{comp} below by adding suitably typed arguments, a return type, and function body:

\[
\texttt{let comp}
\]
7. Abstract Stack Machines (22 points total)

For each of the following programs, draw the state of the abstract stack machine (its workspace, stack and heap) at the point just after a binding for x has been pushed to the stack.

Assume that the program has access to the usual generic tree datatype:

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

Make sure your drawings are legible! Use scratch paper to work out the answer and then transcribe a clean version of the final state to this page for grading.

a. (10 points)

```plaintext
let y = Empty in
let x = Node(y, 3, y) in
x
```
b. (12 points)

```ocaml
let y = ref 0 in
let z = y in
let _ = z := 3 in
let w = !y in
let _ = z := 4 in
let x = !y

x
```