Name: ____________________

ID : ____________________

CSE 130, Winter 2013: Midterm Examination

Feb 12th, 2012

- Do **not** start the exam until you are told to.
- This is a open-book, open-notes exam, but with no computational devices allowed (such as calculators/cellphones/laptops).
- Do **not** look at anyone else's exam. Do **not** talk to anyone but an exam proctor during the exam.
- Write your answers in the space provided.
- Wherever it gives a line limit for your answer, write no more than the specified number of lines. *The rest will be ignored.*
- Work out your solution in blank space or scratch paper, and only put your answer in the answer blank given.
- The points for each problem are a rough indicator of the difficulty of the problem.
- Good luck!

1.	25 Points	
2.	20 Points	
3.	10 Points	
TOTAL	55 Points	

1. [**25 points**]

a. [**18 points**] Consider the following datatype:

```
type 'a maybe =
  | None
  | Some of 'a
```

In this problem you are going to write `first : ('a -> bool) -> 'a list -> 'a maybe`. Given a "tester" function `f` and a list `l`, `(first f l)` returns the first element of the list `l` for which `f` returns true. If there is no element in the list for which `f` returns true, then `first` returns `None`. If there are multiple elements in the list for which `f` returns true, then the first such element is returned. For example:

```
# let even x = (x mod 2 = 0);;
val even : int -> bool = <fun>
# first even [1;3;4;5;7;9;11];;
- : int maybe = Some 4
# first even [1;2;3;4;5;7;9;10;11];;
- : int maybe = Some 2
# first even [1;3;5;7;9;11];;
- : int maybe = None
```

To implement `first`, you will use `fold_left`, whose type is given below:

```
fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Fill in the implementation of `first` below:

```
let first f l =


  let base = _______________________________________ in


  let fold_fn acc elmt =


      __________________________________________________________________________


      __________________________________________________________________________


      __________________________________________________________________________


      __________________________________________________________________________


      __________________________________________________________________________


  in List.fold_left fold_fn base l
```

b. [**7 points**] Now, consider the following code:

```
# let f a b c = (a = b + c);;
val f : int -> int -> int -> bool = <fun>
```

For each expression below, write down what it evaluates to:

```
first ((=) 4) [1;2;3;4;5]      ______________________________________

first ((=) 10) [1;2;3;4;5]     ______________________________________

first ((<) 3) [1;2;3;4;5]      ______________________________________

first ((<) 10) [1;2;3;4;5]     ______________________________________

first ((<) 0) [1;2;3;4;5]      ______________________________________

first (f 1 2) [1;2;3;4;5]      ______________________________________

first (f 5 2) [1;2;3;4;5]      ______________________________________
```

2. [**20 points**]

a. [**10 points**] You are going to write a function `zip : 'a list -> 'b list -> ('a * 'b) list`. Given two lists `l1` and `l2`, `(zip l1 l2)` returns a list containing pairs of corresponding elements from `l1` and `l2`. If lists `l1` and `l2` have different lengths, the returned list has the same length as the shorted of the two lists. For example:

```
# zip [1;2;3] [5;6;7];;
- : (int * int) list = [(1, 5); (2, 6); (3, 7)]
# zip ['a';'b';'c'] [1;2;3];;
- : (char * int) list = [('a', 1); ('b', 2); ('c', 3)]
# zip ['a'] [1;2;3];;
- : (char * int) list = [('a', 1)]
# zip ['a';'b';'c'] [1;2];;
- : (char * int) list = [('a', 1); ('b', 2)]
```

Fill in the implementation of `zip` below:

```
let rec zip l1 l2 =

  match (l1,l2) with

    ------------------------------------------------------------------------------

    ------------------------------------------------------------------------------

    ------------------------------------------------------------------------------

    ------------------------------------------------------------------------------

    ------------------------------------------------------------------------------

    ------------------------------------------------------------------------------
```

b. [**5 points**] Recall the `map` function, which has type `('a -> 'b) -> 'a list -> 'b list`. You will now write a function `map2:('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`. Given a function `f` and two lists `l1` and `l2`, `(map2 f l1 l2)` returns a list in which each element is produced by calling `f` on the corresponding elements of `l1` and `l2`. For example:

```
# (+);;
- : int -> int -> int = <fun>
# map2 (+) [1;2;3] [4;6;8];;
- : int list = [5; 8; 11]
# map2 (-) [1;2;3] [4;6;8];;
- : int list = [-3; -4; -5]
# map2 (/) [10;9;4] [2;3;4];;
- : int list = [5; 3; 1]
# map2 (+) [1;2] [4;6;8];;
- : int list = [5; 8]
```

Using `map` and `zip`, write the code for `map2` below. Note that `map2` is not declared as `rec` so it cannot call itself. Be careful with currying to make sure that everything typechecks properly.

```
let map2 f l1 l2 =

     ----------------------------------------------------------------------------

     ----------------------------------------------------------------------------
```

c. [**5 points**] You will now write a function:

`map3:('a -> 'b -> 'c -> 'd) -> 'a list -> 'b list -> 'c list -> 'd list`

This function works similarly to `map2` but with three lists instead of two. For example:

```
# let add a b c = a + b + c;;
val add : int -> int -> int -> int = <fun>
# map3 add [1;2] [3;4] [5;6];;
- : int list = [9; 12]
# map3 add [1] [3;4] [5;6];;
- : int list = [9]
```

Using `map` and `zip`, write the code for `map3` below. Note again that `map3` is not declared as `rec` so it cannot call itself. Be careful with currying to make sure that everything typechecks properly.

```
let map3 f l1 l2 l3 =

     ----------------------------------------------------------------------------

     ----------------------------------------------------------------------------

     ----------------------------------------------------------------------------
```

3. [**10 points**]

You will now write a function `unzip: ('a * 'b) list -> 'a list * 'b list`. Given a list `l` of pairs, `(unzip l)` returns two lists `l1` and `l2` where `l1` contains the first element of each pair in `l` and `l2` contains the second element of each pair in `l`. For example:

```
# unzip [(1,2); (3,4); (5,6)];;
- : int list * int list = ([1; 3; 5], [2; 4; 6])
# unzip [('a',1); ('b',2)];;
- : char list * int list = (['a'; 'b'], [1; 2])
```

Fill in the implementation of `unzip` below:

```
let rec unzip l =

   match l with

      ----------------------------------------------------------------------------

      ----------------------------------------------------------------------------

      ----------------------------------------------------------------------------

      ----------------------------------------------------------------------------

      ----------------------------------------------------------------------------

      ----------------------------------------------------------------------------
```