• 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!

<table>
<thead>
<tr>
<th></th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>TOTAL</td>
<td>55</td>
</tr>
</tbody>
</table>

---

CSE 130, Winter 2013: Midterm Examination
Feb 12th, 2012
1. [25 points]
   a. [18 points] Consider the following datatype:
   
   ```ml
   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:
   
   ```ml
   # 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:
   
   ```ml
   fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
   ```

   Fill in the implementation of `first` below:

   ```ml
   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:

```ml
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 \texttt{zip} : \texttt{\textquotesingle a} list \to \texttt{\textquotesingle b} list \to \texttt{(\textquotesingle a \times \textquotesingle b)} list.

Given two lists \texttt{l1} and \texttt{l2}, \texttt{(zip l1 l2)} returns a list containing pairs of corresponding elements from \texttt{l1} and \texttt{l2}. If lists \texttt{l1} and \texttt{l2} have different lengths, the returned list has the same length as the shorted of the two lists. For example:

\begin{verbatim}
# 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)]
\end{verbatim}

Fill in the implementation of \texttt{zip} below:

\begin{verbatim}
let rec zip l1 l2 =

match (l1,l2) with

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

\begin{verbatim}
# (+);
- : 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]
\end{verbatim}

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

\begin{verbatim}
let map2 f l1 l2 =
|                          |
|                          |
|                          |

\end{verbatim}

\begin{verbatim}

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

\texttt{map3:('a -> 'b -> 'c -> 'd) -> 'a list -> 'b list -> 'c list -> 'd list}

This function works similarly to \texttt{map2} but with three lists instead of two. For example:

\begin{verbatim}
# 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]
\end{verbatim}

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

\begin{verbatim}
let map3 f l1 l2 l3 =
|                          |
|                          |
|                          |

\end{verbatim}
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:

```ocaml
# 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:

```ocaml
let rec unzip l =

match l with
```

```ocaml

```

```ocaml
```