**Name:** _____

**ID :** _____

# CSE 130, Spring 2013: Midterm Examination

May 9th, 2013

---

- 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.
- In all exercises, you are allowed to use the "@" operator.
- Good luck!

| | | |
|---|---|---|
| 1. | 20 Points | |
| 2. | 30 Points | |
| 3. | 15 Points | |
| TOTAL | 65 Points | |

**1.** [ **20 points** ] Let's warm up with two small folds.

**a.** [ **10 points** ] Use `fold_left` to implement `length : 'a list -> int`, which takes a list and returns its length. For your reference, the type of `fold_left` is given below:

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

Fill in the implementation of `length` below using `fold_left`:

```
let length l =
```

---

---

---

**b.** [ **10 points** ]
Use `fold_left` to implement `remove : 'a list -> 'a -> 'a list`, which takes a list and an element, and returns a new list in which all occurrences of the given element have been removed. The elements in the returned list should be in the same order as in the original list. Do not use `List.rev`.
Fill in the implementation of `remove` below using `fold_left`:

```
let remove l x =
```

---

---

---

**2.** [ **30 points** ] In this question you will write several functions that will allow us to use OCaml lists to represent arrays that support indexing and functional updates. Each time an index is passed in, you can assume the index is greater or equal to 0.

**a.** [ **10 points** ] **Indexing.** First, you will write a function `ith : 'a list -> int -> 'a -> 'a`, which returns the $i^{th}$ element of a list. In particular, given a list `l`, an integer index `i` greater or equal to 0, and a "default" value `d`, then `(ith l i d)` returns the `i`$^{th}$ element of the list `l`, or `d` if this element does not exists. For example:

```
# ith ["a";"b";"c";"d"] 0 "";;
- : string = "a"

# ith ["a";"b";"c";"d"] 1 "";;
- : string = "b"

# ith ["a";"b";"c";"d"] 2 "";;
- : string = "c"

# ith ["a";"b";"c";"d"] 3 "";;
- : string = "d"

# ith ["a";"b";"c";"d"] 4 "";;
- : string = ""
```

Fill in the implementation of `ith` below:

```
let rec ith l i d =

   match l with


      | [] -> ____________________________________________________________


      | h::t -> ___________________________________________________________


               ____________________________________________________________


               ____________________________________________________________


               ____________________________________________________________
```

**b.** [ **10 points** ] **Update.** Next write a function `update : 'a list -> int -> 'a -> 'a list`, which updates the $i^{th}$ element of a list. Since we are doing functional programming, the `update` function does not actually update the list, it returns a new list in which the $i^{th}$ element has been replaced. More specifically, given a list `l`, an integer index `i` greater or equal to 0, and a new value `n`, then `(update l i n)` returns a new list which is equal to `l`, except that the `i`$^{th}$ element of the returned list is equal to `n`. If the `i`$^{th}$ element does not exist, `update` returns a list equal to `l`. For example:

```
# update ["a";"b";"c";"d"] 0 "ZZZ";;
- : string list = ["ZZZ"; "b"; "c"; "d"]

# update ["a";"b";"c";"d"] 1 "ZZZ";;
- : string list = ["a"; "ZZZ"; "c"; "d"]

# update ["a";"b";"c";"d"] 2 "ZZZ";;
- : string list = ["a"; "b"; "ZZZ"; "d"]

# update ["a";"b";"c";"d"] 3 "ZZZ";;
- : string list = ["a"; "b"; "c"; "ZZZ"]

# update ["a";"b";"c";"d"] 4 "ZZZ";;
- : string list = ["a"; "b"; "c"; "d"]
```

Fill in the implementation of `update` below:

```
let rec update l i n =

   match l with


      | [] -> ________________________________________


      | h::t -> ____________________________________________________________


                ____________________________________________________________


                ____________________________________________________________


                ____________________________________________________________
```

**c.** [ **10 points** ] **Functional Update, Revisited.** Next you will write a new version of `update`, namely a function `update2 : 'a list -> int -> 'a -> 'a -> 'a list`, which updates the $i^{th}$ element of a list, but also extends the list with a number of default values if the $i^{th}$ element does not exist. In particular, given a list `l`, an integer index `i` greater or equal to 0, a new value `n`, and a "default" value `d`, then `(update l i n d)` returns a new list which is equal to `l`, except that the $\texttt{i}^{th}$ element of the returned list is equal to `n`. If the $\texttt{i}^{th}$ element does not exist, `update2` returns a list in which enough default values `d` are added to the returned list so that the $\texttt{i}^{th}$ element exists. For example:

```
# update2 ["a";"b";"c";"d"] 0 "ZZZ" "";;
- : string list = ["ZZZ"; "b"; "c"; "d"]

# update2 ["a";"b";"c";"d"] 1 "ZZZ" "";;
- : string list = ["a"; "ZZZ"; "c"; "d"]

# update2 ["a";"b";"c";"d"] 2 "ZZZ" "";;
- : string list = ["a"; "b"; "ZZZ"; "d"]

# update2 ["a";"b";"c";"d"] 3 "ZZZ" "";;
- : string list = ["a"; "b"; "c"; "ZZZ"]

# update2 ["a";"b";"c";"d"] 4 "ZZZ" "";;
- : string list = ["a"; "b"; "c"; "d"; "ZZZ"]

# update2 ["a";"b";"c";"d"] 5 "ZZZ" "";;
- : string list = ["a"; "b"; "c"; "d"; ""; "ZZZ"]

# update2 ["a";"b";"c";"d"] 6 "ZZZ" "";;
- : string list = ["a"; "b"; "c"; "d"; ""; ""; "ZZZ"]

# update2 ["a";"b";"c";"d"] 7 "ZZZ" "";;
- : string list = ["a"; "b"; "c"; "d"; ""; ""; ""; "ZZZ"]
```

Fill in the implementation of `update2` below (**hint:** In my solution the `h::t` case of `update` and `update2` are exactly the same. The only difference between `update` and `update2` is in the base case):

```
let rec update2 l i n d =

   match l with

      | [] -> ____________________________________________________________

      | h::t -> ____________________________________________________________

                ____________________________________________________________

                ____________________________________________________________

                ____________________________________________________________
```

**3.** [ **15 points** ] In this problem you will write a function `categorize : ('a -> int) -> 'a list -> 'a list list` , which will categorize the elements of a list into different bins. Bins are numbered starting at 0. The first parameter to `categorize` is a function `f`, which given an element returns what bin to place that element in. The second parameter to `categorize` is the list `l` of elements to categorize. Then `(categorize f l)` returns a list `r` such that the $i^{th}$ element of `r` is a list of all elements from `l` for which `f` returned $i$. The length of the list `r` is one more than the maximum value returned by `f` when called on the elements of `l`. For example:

```
# let f i =  if i < 0 then 0
             else (if i < 10 then 1
                   else (if i < 20 then 2 else 3));;
val f : int -> int = <fun>

# categorize f [1;2;-3;15;7;30;-1;22;33;14;105];;
- : int list list = [[-3; -1]; [1; 2; 7]; [15; 14]; [30; 22; 33; 105]]

# categorize f [-3;12;14];;
- : int list list = [[-3]; []; [12; 14]]

# categorize f [];;
- : int list list = []
```

Note a few things:

- the elements in each bin appear in the same order as they did in the original list
- in the first call to `categorize` four bins are returned, numbered 0, 1, 2 and 3, because in this call `f` returns a maximum of 3.
- in the second call to `categorize` three bins are returned, because in this call `f` returns a maximum of 2

Fill in the implementation of `categorize` below (**hint:** you may want to use some of the functions you wrote in the previous problem):

```
let categorize f l =

   let base = ________________________________________ in

   let fold_fn acc elmt =

        ____________________________________________________________________________

        ____________________________________________________________________________

        ____________________________________________________________________________

        ____________________________________________________________________________

        ____________________________________________________________________________

   in List.fold_left fold_fn base l
```