Do not start the exam until you are told to.

This is an 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!

<table>
<thead>
<tr>
<th></th>
<th>1. 20 Points</th>
<th>2. 30 Points</th>
<th>3. 15 Points</th>
<th>TOTAL 65 Points</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. [20 points] Let's warm up with two small folds.
   
a. [10 points] Use `fold_left` to implement \texttt{length} : \texttt{a list -> int}, which takes a list and returns its length. For your reference, the type of \texttt{fold_left} is given below:
   
   \begin{verbatim}
   fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
   \end{verbatim}
   
   Fill in the implementation of \texttt{length} below using \texttt{fold_left}:
   
   \begin{verbatim}
   let length l =
   \end{verbatim}

   \begin{verbatim}
   \end{verbatim}

   \begin{verbatim}
   \end{verbatim}

   \begin{verbatim}
   \end{verbatim}

   \begin{verbatim}
   \end{verbatim}

   b. [10 points]
   
   Use \texttt{fold_left} to implement \texttt{remove} : \texttt{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 \texttt{List.rev}.
   
   Fill in the implementation of \texttt{remove} below using \texttt{fold_left}:
   
   \begin{verbatim}
   let remove l x =
   \end{verbatim}

   \begin{verbatim}
   \end{verbatim}

   \begin{verbatim}
   \end{verbatim}

   \begin{verbatim}
   \end{verbatim}

   \begin{verbatim}
   \end{verbatim}
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 \( \text{ith} : \text{'a list -> int -> 'a -> 'a} \), 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 \( \text{ith} \ l \ i \ d \) returns the \( i \)th element of the list \( l \), or \( d \) if this element does not exist. For example:

\[
\begin{align*}
# \text{ith} ["a";"b";"c";"d"] 0 "";; \\
- : \text{string} = "a"

# \text{ith} ["a";"b";"c";"d"] 1 "";; \\
- : \text{string} = "b"

# \text{ith} ["a";"b";"c";"d"] 2 "";; \\
- : \text{string} = "c"

# \text{ith} ["a";"b";"c";"d"] 3 "";; \\
- : \text{string} = "d"

# \text{ith} ["a";"b";"c";"d"] 4 "";; \\
- : \text{string} = ""
\end{align*}
\]

For example:

\[
\begin{align*}
# \text{ith} ["a";"b";"c";"d"] 0 "";; \\
- : \text{string} = "a"

# \text{ith} ["a";"b";"c";"d"] 1 "";; \\
- : \text{string} = "b"

# \text{ith} ["a";"b";"c";"d"] 2 "";; \\
- : \text{string} = "c"

# \text{ith} ["a";"b";"c";"d"] 3 "";; \\
- : \text{string} = "d"

# \text{ith} ["a";"b";"c";"d"] 4 "";; \\
- : \text{string} = ""
\end{align*}
\]

Fill in the implementation of \( \text{ith} \) below:

\[
\begin{align*}
\text{let rec } \text{ith} \ l \ i \ d = \\
\quad \text{match } l \ \text{with} \\
\quad | \ [] \rightarrow \text{__________________________} \\
\quad | \ h :: t \rightarrow \text{__________________________} \\
\quad \quad \text{__________________________} \\
\quad \quad \text{__________________________} \\
\quad \quad \text{__________________________}
\end{align*}
\]
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:

```ocaml
# 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:

```ocaml
let rec update l i n =
  match l with
  | [] -> ________________________________
  | h::t -> ________________________________

  _________________________________________
  _________________________________________
  _________________________________________
  _________________________________________
```

4
c. [10 points] Functional Update, Revisited. Next you will write a new version of update, namely a function update2 : 'a list -> int -> 'a -> 'a list, which updates the \textit{i}th element of a list, but also extends the list with a number of default values if the \textit{i}th element does not exist. In particular, given a list \textit{l}, an integer index \textit{i} greater or equal to 0, a new value \textit{n}, and a "default" value \textit{d}, then (update \textit{l \ i \ n \ d}) returns a new list which is equal to \textit{l}, except that the \textit{i}th element of the returned list is equal to \textit{n}. If the \textit{i}th element does not exist, update2 returns a list in which enough default values \textit{d} are added to the returned list so that the \textit{i}th element exists. For example:

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

Fill in the implementation of update2 below (\textbf{hint}: In my solution the \texttt{h::t} case of \textit{update} and \textit{update2} are exactly the same. The only difference between \textit{update} and \textit{update2} is in the base case):

\begin{verbatim}
let rec update2 l i n d =
  match l with
  | [] -> ____________________________________________________________
  | h::t -> _________________________________________________________

  ________________________________________________________________

  ________________________________________________________________

  ________________________________________________________________

  ________________________________________________________________
\end{verbatim}
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:

```ocaml
# 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):

```ocaml
let categorize f l =

  let base = ________________________________ in

  let fold_fn acc elmt =

    ____________________________________________

    ____________________________________________

    ____________________________________________

    ____________________________________________

    ____________________________________________

  in List.fold_left fold_fn base l
```