• 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!
1. [20 points] Let's warm up with two small folds. For your reference, the type of fold_left is given below:

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

a. [10 points] Use fold_left to implement count : 'a list -> 'a -> int, which returns the number of times a given element occurs in a list. For example:

```
# count [1;2;3;4;5] 10;;
- : int = 0
# count [1;2;3;4;5] 3;;
- : int = 1
# count [1;3;2;3;4;3;5] 3;;
- : int = 3
```

Fill in the implementation of count below:

```
let count l x =
```

b. [10 points] Use fold_left to implement make_palyndrome : 'a list -> 'a list, which takes a list and returns a palindrome, which is produced by adding the elements of the original list in reverse order to the beginning of the original list. For example:

```
# make_palyndrome [1;2];;
- : int list = [2; 1; 1; 2]
# make_palyndrome [1;2;3];;
- : int list = [3; 2; 1; 1; 2; 3]
# make_palyndrome [];;
- : 'a list = []
```

Fill in the implementation of make_palyndrome below using fold_left:

```
let make_palyndrome l =
```
2. [20 points] In this question you will use fold to write a slight variant of fold and then you will use this variant of fold to implement indexing into a list.
   a. [10 points] Fold2. You will first start by using fold to write another function:

   \[
   \text{fold}_2 : \left( \text{'a} \to \text{'b} \to \text{int} \to \text{'a} \right) \to \text{'a} \to \text{'b list} \to \text{'a}+
   \]

   This variant of fold works exactly like the original fold, except that the folding function gets an additional int parameter, which is the index of the element that is passed into the folding function. For example the call:

   \[
   \text{fold}_2 f "" \left[ \text{"a"}; \text{"b"};\text{"c"} \right]
   \]

   would result in

   \[
   (f (f (f "" "a" 0) "b" 1) "c" 2)
   \]

   Fill in the implementation of \text{fold}_2 below using fold:

   \[
   \text{let fold}_2 f b l =
   \]

   \[
   \text{-----------------------------}
   \]

   \[
   \text{-----------------------------}
   \]

   \[
   \text{-----------------------------}
   \]

   \[
   \text{-----------------------------}
   \]

   \[
   \text{-----------------------------}
   \]

   \[
   \text{-----------------------------}
   \]
b. [ 10 points ] Indexing. You will now use `fold_2` to 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:

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

Fill in the implementation of \(ith\) below using `fold_2`:

```ocaml
let rec ith l i d =
```

```ocaml

```
3. [30 points] Consider the following binary tree datatype:

```ocaml
type 'a fun_tree =
| Leaf of ('a -> 'a)
| Node of ('a fun_tree) * ('a fun_tree);
```

a. [10 points] ApplyAll. You will implement `apply_all : 'a fun_tree -> 'a -> 'a`, which applies all the functions in the tree, using an in-order traversal. For example, suppose we had the following:

```ocaml
# let f1 x = x + 1;;
val f1 : int -> int = <fun>

# let f2 x = x * 2;;
val f2 : int -> int = <fun>

# let f3 x = x + 3;;
val f3 : int -> int = <fun>

# let t = Node(Leaf f1, Node(Leaf f2, Leaf f3));;
val t : int fun_tree = Node (Leaf <fun>, Node (Leaf <fun>, Leaf <fun>))
```

```ocaml
# apply_all t 0;;
- : int = 5
```

In particular, `(apply_all t 0)` computes `(f3 (f2 (f1 0)))`. Now fill in the implementation of `fold_2` below using `fold`:

```ocaml
let rec apply_all t x =
```

```
```

---

5
b. [ 10 points ] For each call to apply_all below, write down the value returned by apply_all:

```ocaml
define f1 = (+) 1;;
define f2 = (-) 2;;
define f3 = (+) 3;;
define t = Node(Node(Leaf f1, Leaf f2), Leaf f3);;
apply_all t 0;;
```

```ocaml
define f1 = (^) "a";;
define f2 x = x "b";;
define f3 x = x "ab";;
define t = Node(Leaf f1, Node(Leaf f2, Leaf f3));;
apply_all t "123";;
```

```ocaml
define f1 = List.fold_left (fun x y -> (y*2)::x) [];;
define f2 = List.fold_left (fun x y -> x@[y]) [];;
define t = Node(Node(Leaf f1, Leaf f1), Node(Leaf f1, Leaf f2));;
apply_all t [1;2;3];;
```
c. [ 10 points ] Compose. You will now write a function:

    compose : 'a fun_tree -> 'a fun_tree -> 'a fun_tree

which takes two trees of the same shape and size, and returns a new tree in which the function stored at each leaf is the mathematical composition of the functions stored at the corresponding leaves in the original two trees. Recall that the mathematical composition of two functions \( f_1 \) and \( f_2 \) is a function \( f_3(x) = f_1(f_2(x)) \). For example, consider:

    let t1 = Node(Leaf f1, Leaf f2);;
    let t2 = Node(Leaf f3, Leaf f4);;
    let t3 = compose t1 t2;;

In this example, \( t3 \) would be the tree \( \text{Node(Leaf f5, Leaf f6)} \), where \( f5 \) is the mathematical composition of \( f1 \) and \( f3 \) and \( f6 \) is the mathematical composition of \( f2 \) and \( f4 \).

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

    let rec compose t1 t2 =

        match (t1,t2) with

       

    