You will define several OCaml functions. Many will be very short because they will use other higher-order functions. You may use functions in OCaml's library; the problems point you toward the useful functions and often require that you use them. The sample solution is about 60 lines, including the provided code. Note that problems with 1-line answers can still be challenging, perhaps because the answers are intended to be so short.

**Important note on function bindings:**

In `hw6.ml`, for each function you will implement, the first line is given to you. *The form of this first line matters and you should not change it.* Consider:

```ocaml
let foo x = e1
let bar x y = e2
let baz = e3
```

Notice `foo` takes one argument named `x` while `bar` uses currying to take two arguments `x` and `y`. Most importantly, `baz` is a “variable-style” let binding, but if `e3` evaluates to a function, then `baz` will be bound to that function.

When `hw6.ml` provides something like `let baz = failwith "..."`, you need to replace the `failwith "..."` with an expression that evaluates to the correct function. You should *not* change the form of the binding to be like `let foo x = ...`, nor should your expression be an anonymous function. For example, suppose a problem asked for a function that takes an `int list` and produces a list containing only the positive numbers in the input. If the provided code was `let only_positive = failwith "..."`, then a correct solution is:

```ocaml
let only_positive = List.filter (fun x -> x > 0)
```

whereas these solutions would pass all tests but still be graded incorrect(!) because they do not use variable-style let bindings, or because they use unnecessary function wrapping:

```ocaml
let only_positive xs = List.filter (fun x -> x > 0) xs
let only_positive = fun xs -> List.filter (fun x -> x > 0) xs
```

Throughout the assignment, we use the notation $(m+n)$ at the beginning of a problem to indicate that the problem is worth $m$ points and that the (additional) tests you write are worth $n$ points. Usually, but not always $m = n$.

1. **(10+10)** Write a function `only_lowercase` that takes a `string list` and returns a `string list` that has only the strings in the argument that start with a lowercase letter. Assume all strings have at least 1 character. Use `List.filter`, `Char.lowercase_ascii`, and string index access (`str.[pos]`) to make a 1-2 line solution.

2. **(5+5)** Write a function `longest_string1` that takes a `string list` and returns the longest `string` in the list. If the list is empty, return `""`. In the case of a tie, return the string closest to the beginning of the list. Use `List.fold_left`, `String.length`, and no recursion (other than the fact that the implementation of `List.fold_left` is recursive).

3. **(5+5)** Write a function `longest_string2` that is exactly like `longest_string1` except in the case of ties it returns the string closest to the end of the list. Your solution should be almost an exact copy of `longest_string1`. Still use `List.fold_left` and `String.length`.

4. **(20+20)** Write functions `longest_string_helper`, `longest_string3`, and `longest_string4` such that:

- `longest_string3` has the same behavior as `longest_string1` and `longest_string4` has the same behavior as `longest_string2`.  

---

1
• **longest_string_helper** has type \((\text{int} \to \text{int} \to \text{bool}) \to \text{string list} \to \text{string}\) (notice the currying). This function will look a lot like **longest_string1** and **longest_string2** but is more general because it takes a function as an argument.

• If **longest_string_helper** is passed a function that behaves like \(>\) (so it returns \text{true} exactly when its first argument is strictly greater than its second), then the function returned has the same behavior as **longest_string1**.

• **longest_string3** and **longest_string4** are bound to the result of calls to **longest_string_helper**.

5. (10+10) Write a function **longest_lowercase** that takes a \text{string list} and returns the longest string in the list that begins with a lowercase letter, or "" if there are no such strings. Assume all strings have at least 1 character. Use the \% operator from the starter code for composing functions. Resolve ties like in problem 2.

6. (10+10) Write a function **caps_no_X_string** that takes a \text{string} and returns the \text{string} that is like the input except every letter is capitalized and every “x” or “X” is removed (e.g., “aBxXXxDdx” becomes “ABDD”). Use the \% operator and 3 library functions in the **String** module. Browse the module documentation to find the most useful functions.

The next two problems involve writing functions over lists.

7. (20+20) Write a function **first_answer** of type \((\text{'a} \to \text{'b option}) \to \text{'a list} \to \text{'b}\) (notice the 2 arguments are curried). The first argument should be applied to elements of the second argument in order until the first time it returns \text{Some} \(v\) for some \(v\) and then \(v\) is the result of the call to **first_answer**. If the first argument returns **None** for all list elements, then **first_answer** should raise the exception **NoAnswer**. Hints: Sample solution is 7 lines and does nothing fancy.

8. (15+15) Write a function **all_answers** of type \((\text{'a} \to \text{'b list option}) \to \text{'a list} \to \text{'b list option}\) (notice the 2 arguments are curried). The first argument should be applied to elements of the second argument. If it returns **None** for any element, then the result for **all_answers** is **None**. Else the calls to the first argument will have produced \text{Some} \(\text{lst1}, \text{Some} \text{lst2}, \ldots, \text{Some} \text{lstn}\) and the result of **all_answers** is **Some** \(\text{lst}\) where \(\text{lst}\) is \(\text{lst1}, \text{lst2}, \ldots, \text{lstn}\) appended together. (Your solution can return these lists appended in any order.) Hints: The sample solution is 10 lines. It uses a helper function with an accumulator and uses \@. Note **all_answers** \(f\;[]\) should evaluate to **Some** \([]\).

Further instructions appear on the next page.
**Type Summary:** Evaluating a correct homework solution should generate these bindings, in addition to the bindings for variant and exception definitions:

```ocaml
val only_lowercase : string list -> string list
val longest_string1 : string list -> string
val longest_string2 : string list -> string
val longest_string_helper : (int -> int -> bool) -> string list -> string
val longest_string3 : string list -> string
val longest_string4 : string list -> string
val longest_lowercase : string list -> string
val caps_no_X_string : string -> string
val first_answer : ('a -> 'b option) -> 'a list -> 'b
val all_answers : ('a -> 'b list option) -> 'a list -> 'b list option
```

Notice all functions use currying for multiple arguments. Of course, generating these bindings does not guarantee that your solutions are correct.

**Assessment**

Your solutions should be correct, in good style, and use only features we have used in class. As in Homework 4, prefer pattern matching over functions like `List.hd, List.tl`.

There are 190 points available from the main problems and their tests, plus 30 points for filling out `FEEDBACK.md`, for a total of 220 points, all of which are “on the syllabus”.

**Turn-in Instructions**

Put your solutions to the problems in `hw6.ml`. Put your tests in `hw6test.ml`. Make sure that running both `dune build` and `dune runtest` work. Commit and push these files to Gitlab and submit your branch on Gradescope.

Do not modify `hw6types.ml`, as our autograder will overwrite your changes during grading and your code may break.