CSE 341, Winter 2022, Assignment 3
Due: Tuesday, February 1, 5:00PM Pacific Time

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 120 lines, including the provided code, but not including the challenge problem. Note that problems with 1-line answers can still be challenging, perhaps because the answers are intended to be so short.

Download `hw3.ml` and `hw3types.ml` from the course website.

Important note on function bindings:

In `hw3.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:

```
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 "regular" variable binding, but if `e3` evaluates to a function, then `baz` will be bound to that function.

When `hw3.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:

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

whereas these solutions would pass all tests but still be graded incorrect:

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

---

1. 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. 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. 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. 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`.
   - `longest_string_helper` has type `(int -> int -> bool) -> string list -> 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 `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. Write a function `longest_lowercase` that takes a `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. Write a function `caps_no_X_string` that takes a `string` and returns the `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 that will be useful in later problems.

7. Write a function `first_answer` of type `('a -> 'b  option) -> 'a list -> '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 `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. Write a function `all_answers` of type `('a -> 'b list option) -> 'a list -> '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 `Some lst1`, `Some lst2`, ... `Some lstn` and the result of `all_answers` is `Some lst` where `lst` is `lst1`, `lst2`, ..., `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 []`.

The remaining problems use these type definitions, which are inspired by the type definitions OCaml's implementation would use to implement pattern matching:

```
type pattern = WildcardP | VariableP of string | UnitP | ConstantP of int
             | ConstructorP of string * pattern | TupleP of pattern list
type valu = Constant of int | Unit | Constructor of string * valu | Tuple of valu list
```

Given `valu` v and `pattern` p, either p *matches* v or not. If it does, the match produces a list of `string * valu` pairs; order in the list does not matter. The rules for matching should be unsurprising:

- `WildcardP` matches everything and produces the empty list of bindings.
- `VariableP s` matches any value `v` and produces the one-element list holding `(s,v)`.
- `UnitP` matches only `Unit` and produces the empty list of bindings.
- `ConstantP 17` matches only `Constant 17` and produces the empty list of bindings (and similarly for other integers).
- `ConstructorP(s1,p)` matches `Constructor(s2,v)` if `s1` and `s2` are the same string (you can compare them with `=`) and `p` matches `v`. The list of bindings produced is the list from the nested pattern match. We call the strings `s1` and `s2` the *constructor name*.

- `TupleP ps` matches a value of the form `Tuple vs` if `ps` and `vs` have the same length and for all i, the i^{th} element of `ps` matches the i^{th} element of `vs`. The list of bindings produced is all the lists from the nested pattern matches appended together.
- Nothing else matches.

9. (This problem uses the `pattern` recursive variant type but is not really about pattern-matching.) A function `g` has been provided to you in `hw3types.ml`.
   (a) In an OCaml comment in your `hw3.ml` file, describe in a few English sentences the arguments that `g` takes and what `g` computes (not how `g` computes it, though you will have to understand that to determine what `g` computes). Note: you write no code for this subproblem, only a comment.
   (b) Use `g` to define a function `count_wildcards` that takes a pattern and returns how many `WildcardP` patterns it contains.
   (c) Use `g` to define a function `count_wild_and_variable_lengths` that takes a pattern and returns the number of `Wildcard` patterns it contains plus the sum of the string lengths of all the variables in the variable patterns it contains. (Use `String.length`. We care only about variable names; the constructor names are not relevant.)
   (d) Use `g` to define a function `count_a_var` that takes a string and a pattern (curried) and returns the number of times the string appears as a variable in the pattern. We care only about variable names; the constructor names are not relevant.
10. Write a function `check_pat` that takes a pattern and returns true if and only if all the variables appearing in the pattern are distinct from each other (i.e., use different strings). The constructor names are not relevant. Hints: The sample solution uses two helper functions. The first takes a pattern and returns a list of all the strings it uses for variables. Using `List.fold_left` with a function that uses `@` is useful in one case. The second takes a list of strings and decides if it has repeats. `List.exists` or `List.mem` may be useful. Sample solution is 15 lines. These are hints: We are not requiring `List.fold_left` and `List.exists`/`List.mem` here, but they make it easier.
11. Write a function `matches` of type `valu -> pattern -> (string * valu) list option`. It should take a value and a pattern, and return `Some lst` where `lst` is the list of bindings if the value matches the pattern, or `None` otherwise. Note that if the value matches but the pattern has no variables in it (i.e., no patterns of the form `VariableP s`), then the result is `Some []`. Hints: Sample solution has one `match` expression with 7 branches. The branch for tuples uses `all_answers` and `List.combine`. Sample solution is about 18 lines. Remember to look above for the rules for what patterns match what values, and what bindings they produce.
12. Write a function `first_match` of type

    `valu -> pattern list -> (string * valu) list option`.

    It returns `None` if no pattern in the list matches or `Some lst` where `lst` is the list of bindings for the first pattern in the list that matches. Hints: Use `first_answer` and a `try-with`-expression. Sample solution is about 3 lines.

(Challenge Problem) Write a function `typecheck_patterns` that "type-checks" a `pattern list`. Types for our made-up pattern language are defined by:

```
type typ = AnythingT (* any type of value is okay *)
         | UnitT (* type for Unit *)
         | IntT (* type for integers *)
         | TupleT of typ list (* tuple types *)
         | VariantT of string (* some named variant *)
```

`typecheck_patterns` should have type

`(string * string * typ) list -> pattern list -> typ option`.

The first argument contains elements that look like `("foo","bar",IntT)`, which means constructor `foo` makes a value of type `VariantT "bar"` given a value of type `IntT`. Assume list elements all have different first fields (the constructor name), but there are probably elements with the same second field (the variant name). Under the assumptions this list provides, you "type-check" the `pattern list` to see if there exists some `typ` (call it t) that *all* the patterns in the list can have. If so, return `Some t`, else return `None`.

You must return the "most lenient" type that all the patterns can have. For example, given patterns

`TupleP [VariableP "x", VariableP "y"]` and `TupleP [WildcardP, WildcardP]`,

you should return

`Some (TupleT [AnythingT, AnythingT])`

even though they could both have type `TupleT [IntT, IntT]`. As another example, if the only patterns are `TupleP [WildcardP, WildcardP]` and `TupleP [WildcardP, TupleP [WildcardP, WildcardP]]`, you should return `Some (TupleT [AnythingT, TupleT[AnythingT, AnythingT]])`.

Type Summary: Evaluating a correct homework solution should generate these bindings, in addition to the bindings for variant and exception definitions:

```
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
val count_wildcards : pattern -> int
val count_wild_and_variable_lengths : pattern -> int
val count_a_var : string -> pattern -> int
val check_pat : pattern -> bool
val matches : valu -> pattern -> (string * valu) list option
val first_match : valu -> pattern list -> (string * valu) list option
(* optional challenge problem generates this binding: *)
val typecheck_patterns : (string * string * typ) list -> pattern list -> typ option
```

Notice all functions use currying for multiple arguments. Of course, generating these bindings does not guarantee that your solutions are correct. *Test your functions: Put your testing code in a second file. We will not grade it, but you must turn it in.*

Assessment

Your solutions should be correct, in good style, and use only features we have used in class. As in Homework 2, prefer pattern matching over functions like `List.hd`, `List.tl`.

Turn-in Instructions

Upload your `hw3.ml` and `hw3test.ml` to Gradescope. Do not modify `hw3types.ml` and do not turn it in.