CS181, Winter 2022, Assignment 3
Due: Monday February 7, 11:00PM

You will define several SML functions. Many will be very short because they will use other higher-order functions. You may use functions in ML's library; the problems point you toward the useful functions and often ***require*** that you use them. (You can see SML's standard libarary document at: `https://smlfamily.github.io/Basis/manpages.html`.) The sample solution less than 75 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.sml` from the course website.

---

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 an lowercase letter. Assume all strings have at least 1 character. Use the standard library functions `List.filter`, `Char.isLower`, and `String.sub` 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.foldl`, `String.size`, and no recursion in your code (the recursion inside `List.foldl` is fine).

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 `foldl` and `String.size`.

4. Write three 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 called with a function that behaves like the greater-than operator `>` (so it returns `true` exactly when its first argument is strictly greater than its second), it returns a function with same behavior as `longest_string1`.
   - `longest_string3` and `longest_string4` are defined with `val`-bindings and partial applications of `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 an lowercase letter, or `""` if there are no such strings. Assume all strings have at least 1 character. Use a `val`-binding and the ML library's `o` operator for composing functions. Resolve ties like in problem 2. **Hint:** you should just be able to compose a couple of the functions you have already defined to solve this problem.

The next problem involves writing a function over lists that will be useful in a later problem.

6. Write a function `all_answers` of type `('a -> 'b list option) -> 'a list -> 'b list option` (notice the 2 arguments are curried). The first function argument `f` should be applied to elements of the second argument `l` (you can use different names if you like). If `f` returns `NONE` for any element in `l`, then the result for `all_answers` is `NONE`. Otherwise, the calls to `f` on the elements of `l` will have produced `SOME lst1`, `SOME lst2`, ... `SOME lstn`. The result of `all_answers` is `SOME lst`, where `lst` consists of `lst1`, `lst2`, ..., `lstn` appended together (order doesn't matter). Hints: The sample solution is 8 lines. It uses a helper function with an accumulator and uses `@`. Also note that `all_answers f []` should evaluate to `SOME []`.

The remaining problem uses these type definitions, which are inspired by the type definitions an ML implementation would use to implement pattern matching:

```
datatype pattern = WildcardP | VariableP of string | UnitP | ConstantP of int
                 | ConstructorP of string * pattern | TupleP of pattern list
datatype 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.

7. Write a function `match` that takes a `valu * pattern` and returns a `(string * valu) list option`, namely `NONE` if the pattern does not match and `SOME lst` where `lst` is the list of bindings if it does. Note that if the value matches but the pattern has no patterns of the form `VariableP s`, then the result is `SOME []`. Hints: Sample solution has one case expression with 7 branches. The branch for tuples uses `all_answers` and `ListPair.zip`. Sample solution is 13 lines. Remember to look above for the rules for what patterns match what values, and what bindings they produce. These are hints: We are not requiring `all_answers` and `ListPair.zip` here, but they make it easier.

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

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

`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 `Datatype "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 datatype 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]`, 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 must return `SOME (TupleT [AnythingT, TupleT[AnythingT, AnythingT]])`.

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

```
val only_lowercase = fn : string list -> string list
val longest_string1 = fn : string list -> string
val longest_string2 = fn : string list -> string
val longest_string_helper = fn : (int * int -> bool) -> string list -> string
val longest_string3 = fn : string list -> string
val longest_string4 = fn : string list -> string
val longest_lowercase = fn : string list -> string
val all_answers = fn : ('a -> 'b list option) -> 'a list -> 'b list option
val match = fn : valu * pattern -> (string * valu) list option
```

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 `null`, `hd`, `tl` and features like `#1`.

Turn-in Instructions

- Put all your solutions in one file, `hw3.sml`. Put tests you wrote in `hw3tests.sml`.
- Follow the link on the course website (homework section) for the web-form for submitting your files.