CSC 330 —Programming Languages
Fall 2018
Assignment No. 3

Note 1 **This assignment is to be done individually**

Note 2 You can discuss the assignment with others, but copying code is prohibited.

- Due date: as indicated in Connex.
- This assignment is worth 1% of your total course mark.
- Submit electronically via Connex the file `hw3.sml` with your solutions.

Objectives

After completing this assignment, you will have experience:

- Learn to use higher-order functions
- Learn to use `map`, `filter`, `foldl`
- Learn to use currying

Your task, should you choose to accept it

You will write 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. The sample solution is about 120 lines, including the provided code. Many solutions are 1-liners.

1. Write a function `only_capitals` that takes a `string list` and returns a `string list` that has only the strings in the argument that start with an uppercase letter. Assume all strings have at least 1 character. Use `List.filter`, `Char.isUpper`, 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 `foldl`, `String.size`, and no recursion (other than the implementation of foldl 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 `foldl` and `String.size`.

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.
- `longest_string3` and `longest_string4` are defined with val-bindings and partial applications of `longest_string_helper`.

5. Write a function `longest_capitalized` that takes a `string list` and returns the longest string in the list that begins with an uppercase letter (or `""` if there are no such strings). Use a val-binding and the ML library's `o` operator for composing functions. Resolve ties like in problem 2.

6. Write a function `rev_string` that takes a `string` and returns the `string` that is the same characters in reverse order. Use ML's `o` operator, the library function `rev` for reversing lists, and two 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` that has type `('a -> 'b option) -> 'a list -> 'b`. Notice that 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 5 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 (the order in the result list should be preserved). Hints: The sample solution is 8 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 an ML interpreter would use to implement pattern matching:

```
datatype pattern = Wildcard | Variable of string | UnitP | ConstP of int
    | TupleP of pattern list | ConstructorP of string * pattern

datatype valu = Const of int | Unit | Tuple of valu list
    | Constructor of string * valu
```

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 should be preserved. The rules for matching should be unsurprising (they are the same as in ML):

- `Wildcard` matches everything and produces the empty list of bindings.
- `Variable 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.

- `ConstP 17` matches only `Const 17` and produces the empty list of bindings (and similarly for other integers).
- `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.
- `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*.
- Nothing else matches.

9. This problem uses the pattern datatype but is not really about pattern-matching. A function `g` has been provided to you. These functions can be written in one line of code.
   (a) In an ML comment, 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). You do not write code for this sub-problem. Search for the comment: `Description of g:` and write your comment there.
   (b) Use `g` to define a function `count_wildcards` that takes a pattern and returns how many `Wildcard` patterns it contains. See the test cases to illustrate what it does.
   (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.size`. We care only about variable names; the constructor names are not relevant.
   (d) Use `g` to define a function `count_some_var` that takes a string and a pattern (as a pair) 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 `foldl` with a function that uses `append` is useful in one case. The second takes a list of strings and decides if it has repeats. `List.exists` may be useful. Sample solution is approximately 18 lines. These are hints, it is not rquired to use `foldl` and `List.exists`, but they might make it easier.
11. 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 `Variable s`, then the result is `SOME []`. **Remember to look above for the rules for what patterns match what values, and what bindings they produce**. Hints: Sample solution has one case expression with 7 branches. The branch for tuples uses `all_answers` and `ListPair.zip`. Sample solution is approximately 20 lines. These are hints: We are not requiring `all_answers` and `ListPair.zip here`, but they make it easier.

12. Write a function `first_match` that takes a value and a list of patterns and returns a `(string * valu) list option`, namely `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. Use `first_answer` and a handle-expression. Notice that the 2 arguments are curried. Hints: Sample solution is 2 lines.

Hints

1. These are the bindings your program should generate:

```
val only_capitals = 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_capitalized = fn : string list -> string
val rev_string = fn : string -> string
val g = fn : (unit -> int) -> (string -> int) -> pattern -> int
val count_wildcards = fn : pattern -> int
val count_wild_and_variable_lengths = fn : pattern -> int
val count_some_var = fn : string * pattern -> int
val first_answer = fn : ('a -> 'b option) -> 'a list -> 'b
val all_answers = fn : ('a -> 'b list option) -> 'a list -> 'b list option
val check_pat = fn : pattern -> bool
val match = fn : valu * pattern -> (string * valu) list option
val first_match = fn : valu -> pattern list -> (string * valu) list option
```

Evaluation

Solutions should be:

1. Correct. We might use more tests than the ones provided. It will be run using SML/NJ.

2. In good style, including indentation and line breaks