CSE 341, Spring 2020, Assignment 3
Due: ~~Thursday, April 30~~ Saturday, May 2, 11:59PM

Overview

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. 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.sml` from the course website and add your solutions to that file.

Problems

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 `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 `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 a function `longest_string_helper` that 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 stricly greater than its second), then the function returned should have the same behavior as `longest_string1`.

5. Write functions `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_string3` and `longest_string4` are both defined with `val`-bindings and partial applications of `longest_string_helper`.

6. 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.

7. 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 a `val` binding, ML's `o` operator, 2–3 library functions in the `String` module, 1 in the `Char` module, and 1–2 in the `List` module. Browse the module documentation to find the most useful functions. Note ML has strange syntax for character literals, e.g., `#"A"`.

The next two problems involve writing functions over lists that will be useful in later problems.

8. 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 5 lines and does nothing fancy.

9. 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 (order doesn't matter). 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 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.

The next four problems use the `pattern` datatype but are not really about pattern-matching. They also use the function `g` provided to you.

10. 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). Note you write no code for this subproblem.

11. Use `g` to define a function `count_wildcards` that takes a pattern and returns how many `WildcardP` patterns it contains.

12. 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.)

13. Use `g` to define a function `count_a_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.

In the last three functions, you will implement the simplified version of ML pattern matching described above using the `pattern` and `valu` datatypes.

14. 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 `@` 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 15 lines. These are hints: We are not requiring `foldl` and `List.exists` here, but they make it easier.

15. 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.

16. 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. Hints: Sample solution is 3 lines.

---

17. **(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`.

Continued on next page...

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]])`.

Summary

Evaluating a correct homework solution should generate these bindings, in addition to the bindings from the code provided to you.

```
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 caps_no_X_string = fn : string -> string
val first_answer = fn : ('a -> 'b option) -> 'a list -> 'b
val all_answers = fn : ('a -> 'b list option) -> 'a list -> 'b list option
val count_wildcards = fn : pattern -> int
val count_wild_and_variable_lengths = fn : pattern -> int
val count_a_var = fn : string * pattern -> int
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
```

Of course, generating these bindings does not guarantee that your solutions are correct.

Testing

In addition to implementing the functions described above, you must write a suite of tests to verify that your functions work correctly. Tests should be written using the approach shown in section, and should be comprehensive enough to fully verify that your functions work as indicated. Be sure to consider edge cases and unusual (but valid) inputs. Truly exceptional test suites may receive a small amount of extra credit.

Assessment

To receive full credit, your solutions should be:

- Functionally correct
- Written in good style according to the style guide, including indentation and line breaks
- Written using only features discussed in class through Monday, April 27 (Lecture 9).
- Written **without** using `null`, `hd`, `tl`, `isSome`, `valOf`, or any function named with the `#` character. (You may use `#` for character literals.)

Turn-in Instructions

- Add all your solutions to the main problems and challenge problems (if you worked on them) to the file `hw3.sml`.
- Put all your tests in another file called `hw3tests.sml`.

- Follow the link on the course website to submit your files to Gradescope.
- The Gradescope autograder will confirm that you have submitted the correct file and that your code compiles. **Submissions that do not compile will receive a 15% penalty!**