CS 251 in-class exercise: Implementing Pattern-Matching[1]

We will not finish this in a day. It will be tied to a future assignment.

List Helpers

The first two functions you will write will be useful in later problems.

1. 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.

2. 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 `@` (the infix list-append function). Note `all_answers f []` should evaluate to `SOME []`.

Pattern-Matching

The following type definitions are inspired by the type definitions an ML implementation 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 does not matter. The rules for matching should be unsurprising:

- `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.

[1] Credit to Dan Grossman for this exercise.

3. (This problem uses the `pattern` datatype but is not really about pattern-matching.) A function `g` has been provided to you.

   (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). No code required.

   (b) Use `g` to define a function `count_wildcards` that takes a pattern and returns how many `Wildcard` 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.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.

4. 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 15 lines. These are hints: We are not requiring `foldl` and `List.exists` here, but they make it easier.

5. 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 []`. 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.

6. 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.