CSE 341, Winter 2017, Homework 3 – Pattern Matching

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.

Download `hw3-patterns.sml` from the course website.

The first two problems involve writing functions over lists that 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 `@`. 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.

3. 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.

4. 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.

5. 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.

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

```
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
```

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

(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]])`.