CSE 341, Winter 2008, Assignment 3
Due: Wednesday 6 February, 8:00AM

Last updated: January 28

You will write 11 SML functions (not counting local helper functions) relating to "contacts" (as in homework 2) and pattern-matching (somewhat like in ML). Your solutions should use pattern matching and higher-order functions well. You may use functions in ML's library (particularly the `List` and `ListPair` structures). The sample solution is about 100 lines, *including* all the type definitions given to you. Despite being shorter, this assignment is probably more difficult than homework 2.

Problems 1–6 use these type definitions. They are all from homework 2 except the last one, which defines a type for lists of contacts where the list always has at least one contact.

```
datatype age_difference = Older | Younger
datatype contact_category = LocalFriend
                          | DistantFriend
                          | Relative of age_difference
                          | Coworker
type name = string
type contact = contact_category * name
datatype my_contacts_list = One of contact | More of contact * my_contacts_list
```

1. Write a function that takes 4 curried arguments like this:

   ```
   fun contacts_processor combine base keep contacts = ...
   ```

   This function is like a combination of "filter" and "fold" for `my_contacts_list` values. In particular:

   - `contacts` is the `my_contacts_list` that is processed.
   - `keep` has type `contact_category -> bool`. Any elements of `contacts` that have a category for which `keep` returns false are irrelevant to the output.
   - `base` is the overall answer if `keep` returns false for every contact. It can have any type `'a`.
   - `combine` has type `name * 'a -> 'a`. It produces an answer from a name and a recursive answer.

   Overall, `contacts_processor` returns `combine(name1,combine(name2,...(combine(namen,base))))` where `name1`, `name2`, ..., `namen` are the names of contacts with categories for which `keep` returns `true`. You may process the names in any order (i.e., you may assume `combine` is commutative). Sample solution is 12 lines.

2. Write a function `is_informal` that takes a `contact_category` and evaluates to `true` if you can talk informally with contacts in this category. As in homework 2, such categories are younger relatives and all friends.

3. Write a function `names_informal` that takes a `my_contacts_list` and returns a list of the names of all the informal (in the sense of the previous problem) contacts. Use `contacts_processor` and `is_informal` to produce a 1-line answer. Note the result type is `string list`, not `my_contacts_list`.

4. Write a function `contact_counter` that takes a function `f` of type `contact_category -> bool` and returns a function of type `my_contacts_list -> int` that returns the number of contacts in the list with categories for which `f` returns `true`. Use `contacts_processor` to produce a 1-line answer.

5. Write a function `num_informal` of type `my_contacts_list -> int` that returns how many contacts in the list can be talked to informally. Use answers to previous problems to produce a 1-line answer.

6. Write a function `num_formal` of type `my_contacts_list -> int` that returns how many contacts in the list *cannot* be talked to informally. Use answers to previous problems and a predefined function or two to produce a 1-line answer.

Problems 7–8 involve writing functions over lists that will be useful in later problems. The only non-function binding you need is:

```
exception NoAnswer
```

7. 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 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 (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 []`.

Problems 9–11 (and 12, a challenge problem) use these type definitions, which are similar to ML-style pattern matching:

```
datatype pattern = Wildcard
                 | Variable of string
                 | TupleP of pattern list
                 | ConstructorP of string * pattern
datatype valu = Const of int
              | Tuple of valu list
              | Constructor of string * valu
```

Given `valu` v and `pattern` p, either p *matches* v or not. If it does, the matches 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.
- `Variable s` matches any value `v` and produces the one-element list holding `(s,v)`.
- `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 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 produced is the list from the nested pattern match.
- Nothing else matches.

9. 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). Note the choice of strings for constructors does not matter. 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 `List.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` is useful. Sample solution is 15 lines.

10. 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. Hints: Sample solution has one case expression with 5 branches. The branch for tuples uses `all_answers` and `ListPair.zip`. Sample solution is 11 lines.

11. 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 mathces. Hints: Sample solution is 3 lines and uses `first_answer` and a `handle`-expression.

12. **(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 = Anything (* any type of value is okay *)
             | IntT (* type for integers *)
             | TupleT of typ list (* tuple types *)
             | Datatype 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`. You may 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, if the patterns are `TupleP[Variable("x"),Variable("y")]` and `TupleP[Wildcard,Wildcard]`, you must return `TupleT[Anything,Anything]` even though they could both have type `TupleT[IntT,IntT]`. As another example, if the only patterns are `TupleP[Wildcard,Wildcard]` and `TupleP[Wildcard,TupleP[Wildcard,Wildcard]]`, you must return `TupleT[Anything,TupleT[Anything,Anything]]`.

Warning: The sample solution does not include the challenge problem.

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

```
val contacts_processor =
  fn : (name * 'a -> 'a) -> 'a -> (contact_category -> bool) -> my_contacts_list -> 'a
val is_informal = fn : contact_category -> bool
val names_informal = fn : my_contacts_list -> name list
val contact_counter = fn : (contact_category -> bool) -> my_contacts_list -> int
val num_informal = fn : my_contacts_list -> int
val num_formal = fn : my_contacts_list -> 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
```

Assessment: Your solutions should be correct, in good style (including indentation and line breaks), and using features we have used in class.

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw3.sml`, where `lastname` is replaced with your last name.
- The first line of your `.sml` file should be an ML comment with your name and the phrase `homework 3`.
- Go to `https://catalysttools.washington.edu/collectit/dropbox/djg7/1359` (link available from the course website), follow the "Homework 3" link, and upload your file.