CSCI 2041: Advanced Programming Principles
Fall 2017, University of Minnesota
Homework 4

Posted: Oct 18, 2017
Due: Oct 27, 2017 by midnight

Submission Protocol

Some requirements that you must adhere to in submitting your work for grade for this homework:

- All of your work must be submitted in a file named *hw4.ml*. This file must be pushed to your personal git repository in the organization at the UMN github for this course, i.e. in a repository that has the name *umn-csci-2041F17d/repo-abcde001* where *abcde001* is your username. This file should be placed at the top level in this directory.
- Include in the file a comment of the form (* Solution for Problem X *), where X is the number of the problem, before the solution to each problem.
- The programming problems are specific about the names of the types, functions and the types of the functions you must define. To get credit for the work in each case, you **must** stick to the requirements spelled out.
- In the first problem, you are required to complete the missing parts if a definition. You **must** define the function that is needed by doing this; simply writing your own code for the function will not get you any credit.

Note that general requirements for homework submissions are described on the homeworks page and have also been discussed in class. This is a course on *programming principles* so you should keep in mind all the requirements for good programming that you would have learnt in preceding programming courses.

You should ensure that the code you submit actually compiles using ocaml. To check you could use the following command on a cselabs machine in a directory containing the file:

```
ocamlbuild hw4.byte || echo "\n\n---INCORRECT SUBMISSION---\n\n"
```

If you see:

```
---INCORRECT SUBMISSION---
```

it means that your file does not compile and hence is not yet ready to be submitted.

Problem 1 (4 points for addOnePair, 3 points for each of the other functions)

This problem is modelled on an application for the map-reduce paradigm that is described here. The task is to calculate all the friends in common between all pairs of (Facebook) friends.

The general structure of the problem is the following. For each person we have given to us a list of friends. For example, if `a`, `b`, `c`, `d` and `e` are the people in question, the corresponding list of friends may be the following:

```
a -> b c d
b -> a c d e
c -> a b d e
d -> a b c e
e -> b c d
```

What we want to compute eventually is an list of common friends for each pair of friends. For example, from the information given above, we want to compute the following information:

```
(a b) -> (c d)
(a c) -> (b d)
(a d) -> (b c)
(b c) -> (a d e)
(b d) -> (a c e)
(b e) -> (c d)
(c d) -> (a b e)
(c e) -> (b d)
(d e) -> (b c)
```

This problem structures this computation as a map-reduce one. Your task is to understand this structure and to fill in the missing pieces in some function definitions to complete the solution.

At the very beginning, we have to decide how to represent "persons." We will do this using their names, written as strings. Thus, the person written as `a` above will be represented by the string `"a"`. We will then represent the input in the problem by a list of pairs, the first item of each pair being a person and the second a list of his/her friends. Thus, the input in the example above will be encoded as follows:

```
    [ ("a", ["b"; "c"; "d"]);
      ("b", ["a"; "c"; "d"; "e"]);
      ("c", ["a"; "b"; "d"; "e"]);
      ("d", ["a"; "b"; "c"; "e"]);
      ("e", ["b"; "c"; "d"]) ]
```

In the rest of the description below, we will assume that the identifier `friendsList` has been bound to this list through a declaration of the form

```
let friendsList =
   [ ("a", ["b"; "c"; "d"]);
     ("b", ["a"; "c"; "d"; "e"]);
     ("c", ["a"; "b"; "d"; "e"]);
     ("d", ["a"; "b"; "c"; "e"]);
     ("e", ["b"; "c"; "d"]) ]
```

Not surprisingly, the solution to a problem of the kind described will involve a *mapping* and a *reducing* phase.

1. In the mapping phase, we will transform the input into a list of lists, where each list associates the friends of a person with pairs consisting of that person and each of his/her friends. For example, given `friendsList` as input, we want to transform this into the list

   ```
   [[(("a", "b"), ["b"; "c"; "d"]); (("a", "c"), ["b"; "c"; "d"]);
     (("a", "d"), ["b"; "c"; "d"])];
    [(("a", "b"), ["a"; "c"; "d"; "e"]); (("b", "c"), ["a"; "c"; "d"; "e"]);
     (("b", "d"), ["a"; "c"; "d"; "e"]); (("b", "e"), ["a"; "c"; "d"; "e"])];
    [(("a", "c"), ["a"; "b"; "d"; "e"]); (("b", "c"), ["a"; "b"; "d"; "e"]);
     (("c", "d"), ["a"; "b"; "d"; "e"]); (("c", "e"), ["a"; "b"; "d"; "e"])];
    [(("a", "d"), ["a"; "b"; "c"; "e"]); (("b", "d"), ["a"; "b"; "c"; "e"]);
     (("c", "d"), ["a"; "b"; "c"; "e"]); (("d", "e"), ["a"; "b"; "c"; "e"])];
    [(("b", "e"), ["b"; "c"; "d"]); (("c", "e"), ["b"; "c"; "d"]);
     (("d", "e"), ["b"; "c"; "d"])]]
   ```

   One thing to note here: when we pair two people together, we will represent them by using their names in alphabetical order in the pair. For example, the pair representing `"a"` and `"b"` will be written as `("a","b")`, whereas the pair representing `"c"` and `"b"` will be written as `("b","c")`.

   We will implement the mapping transformation in two steps:

   a. Given a person and a list of his/her friends, define a function `makePairLists` that will associate the list with each pair of the person and a friend. Here is an example interaction that shows the behaviour we expect of the function:

      ```
      # makePairLists "b" ["a"; "c"; "d"; "e"];;
      - : ((string * string) * string list) list =
      [(("a", "b"), ["a"; "c"; "d"; "e"]); (("b", "c"), ["a"; "c"; "d"; "e"]);
       (("b", "d"), ["a"; "c"; "d"; "e"]); (("b", "e"), ["a"; "c"; "d"; "e"])]
      #
      ```

      We can define `makePairLists` in many ways but in this problem we want you to define it via a mapping operation. In particular, fill in the blank in the partial definition below to complete it:

      ```
       let makePairLists p fl = List.map ___ fl
      ```

      Compare the behaviour of your function with that shown above.

   b. Now define the function `makeAllPairLists` that will take the input to the problem and will transform it into the form we want as output of the mapping phase. An interaction that shows the behaviour of this function is the following:

      ```
       makeAllPairLists friendsList;;
      - : ((string * string) * string list) list list =
      [[(("a", "b"), ["b"; "c"; "d"]); (("a", "c"), ["b"; "c"; "d"]);
        (("a", "d"), ["b"; "c"; "d"])];
       [(("a", "b"), ["a"; "c"; "d"; "e"]); (("b", "c"), ["a"; "c"; "d"; "e"]);
        (("b", "d"), ["a"; "c"; "d"; "e"]); (("b", "e"), ["a"; "c"; "d"; "e"])];
       [(("a", "c"), ["a"; "b"; "d"; "e"]); (("b", "c"), ["a"; "b"; "d"; "e"]);
        (("c", "d"), ["a"; "b"; "d"; "e"]); (("c", "e"), ["a"; "b"; "d"; "e"])];
       [(("a", "d"), ["a"; "b"; "c"; "e"]); (("b", "d"), ["a"; "b"; "c"; "e"]);
        (("c", "d"), ["a"; "b"; "c"; "e"]); (("d", "e"), ["a"; "b"; "c"; "e"])];
       [(("b", "e"), ["b"; "c"; "d"]); (("c", "e"), ["b"; "c"; "d"]);
        (("d", "e"), ["b"; "c"; "d"])]]
      ```

      We can again define this function in many ways, but what we want to do in this problem is realize it explicitly as a mapping operation. In particular, fill in the blanks below to complete its definition:

      ```
       let makeAllPairLists l =  List.map ___ l
      ```

Hint: think of using the function `makePairLists` defined in the first subpart above in constructing the mapping function.

2. The second phase will "reduce" the list constructed in the first phase to produce the output desired in this problem. The key out here would be to define the aggregation function. This function would implement the following idea. One of its inputs will be a list that associates the friends of a person with each pair of that person and his/her friends such as, for example,

```
[(("a", "c"), ["a"; "b"; "d"; "e"]); (("b", "c"), ["a"; "b"; "d"; "e"]);
  (("c", "d"), ["a"; "b"; "d"; "e"]); (("c", "e"), ["a"; "b"; "d"; "e"])]
```

Another of its inputs will be a partially constructed output, obtained by aggregating the later (or earlier, depending on the kind of folding being done) lists of this kind. This could, for example, be the following:

```
[(("b", "e"), ["b"; "c"; "d"]); (("c", "e"), ["b"; "c"; "d"]);
 (("d", "e"), ["b"; "c"]); (("a", "d"), ["a"; "b"; "c"; "e"]);
 (("b", "d"), ["a"; "b"; "c"; "e"]); (("c", "d"), ["a"; "b"; "c"; "e"])]
```

These two lists represent the possible shared friends from two different perspectives. For example, the first input indicates that the shared friends for `"c"` and `"e"` are limited to `["a"; "b"; "d"; "e"]` because these are the only friends `"c"` has. Relatedly the second input indicates that the shared friends for `"c"` and `"e"` are limited to `["b"; "c"; "d"]` because these are the only friends `"e"` has. We want to combine these perspectives to conclude that the common friends for `"c"` and `"e"` are `["b"; "d"]`, the intersection of their two friends lists. More completely, given the above two inputs, we want our aggregation function to compute the new partial output

```
[(("b", "e"), ["b"; "c"; "d"]); (("c", "e"), ["b"; "d"]);
 (("d", "e"), ["b"; "c"]); (("a", "d"), ["a"; "b"; "c"; "e"]);
 (("b", "d"), ["a"; "b"; "c"; "e"]); (("c", "d"), ["a"; "b"; "e"]);
 (("a", "c"), ["a"; "b"; "d"; "e"]); (("b", "c"), ["a"; "b"; "d"; "e"])]
```

We will construct the aggregation function in the following steps.

a. One operation that we will need is that of constructing the intersection of two sets of friends. As in Lab 7, we represent sets as lists in with a particular item appears at most once. As we did for union in Lab 7, we will define intersection using an aggregation function. In particular, complete its definition by filling the blanks in the following partial definition:

```
let intersect l1 l2 = List.fold_right ___ l1 ___
```

Test that it works correctly by using examples such as that of combining the potential list of common friends of `"c"` and `"e"` into one list of actual common friends as discussed above.

b. Now define the function `addOnePair` that takes one pair of friends and a list of potential common friends and assimilates this information into a previously computed, partial list of common friends. Some example interactions that show the expected behaviour of this function:

```
addOnePair
  (("a", "c"), ["a"; "b"; "d"; "e"])
  [(("b", "e"), ["b"; "c"; "d"]); (("c", "e"), ["b"; "c"; "d"]);
   (("d", "e"), ["b"; "c"]); (("a", "d"), ["a"; "b"; "c"; "e"]);
   (("b", "d"), ["a"; "b"; "c"; "e"]); (("c", "d"), ["a"; "b"; "c"; "e"])];;
    - : ((string * string) * string list) list =
[(("b", "e"), ["b"; "c"; "d"]); (("c", "e"), ["b"; "c"; "d"]);
 (("d", "e"), ["b"; "c"]); (("a", "d"), ["a"; "b"; "c"; "e"]);
 (("b", "d"), ["a"; "b"; "c"; "e"]); (("c", "d"), ["a"; "b"; "c"; "e"]);
 (("a", "c"), ["a"; "b"; "d"; "e"])]
# addOnePair
  (("c", "e"), ["a"; "b"; "d"; "e"])
```

```
      [(("b", "e"), ["b"; "c"; "d"]); (("c", "e"), ["b"; "c"; "d"]);
       (("d", "e"), ["b"; "c"]); (("a", "d"), ["a"; "b"; "c"; "e"]);
       (("b", "d"), ["a"; "b"; "c"; "e"]); (("c", "d"), ["a"; "b"; "c"; "e"])];;
            - : ((string * string) * string list) list =
   [(("b", "e"), ["b"; "c"; "d"]); (("c", "e"), ["b"; "d"]);
    (("d", "e"), ["b"; "c"]); (("a", "d"), ["a"; "b"; "c"; "e"]);
    (("b", "d"), ["a"; "b"; "c"; "e"]); (("c", "d"), ["a"; "b"; "c"; "e"])]
   #
```

Note that in the first case the pair gets added as is to the partial list of common friends because there isn't another previous perspective of common friends for `"a"` and `"c"` whereas in the second case, two different perspectives of the common friends of `"c"` and `"e"` have to be combined. Note also that I had to define this function without using the aggregation or mapping functions, I could not see a way to do it otherwise.

c. Finally, define the aggregation function called `addAllPairs` that will take a list that associates the friends of a person with each pair of that person and his/her friends and a partially aggregated output to construct a more accurate aggregated output. An example interaction to indicate the expected behaviour for `addAllPairs`:

```
# addAllPairs
  [(("a", "c"), ["a"; "b"; "d"; "e"]); (("b", "c"), ["a"; "b"; "d"; "e"]);
   (("c", "d"), ["a"; "b"; "d"; "e"]); (("c", "e"), ["a"; "b"; "d"; "e"])]
  [(("b", "e"), ["b"; "c"; "d"]); (("c", "e"), ["b"; "c"; "d"]);
   (("d", "e"), ["b"; "c"]); (("a", "d"), ["a"; "b"; "c"; "e"]);
   (("b", "d"), ["a"; "b"; "c"; "e"]); (("c", "d"), ["a"; "b"; "c"; "e"])];;
    - : ((string * string) * string list) list =
[(("b", "e"), ["b"; "c"; "d"]); (("c", "e"), ["b"; "d"]);
 (("d", "e"), ["b"; "c"]); (("a", "d"), ["a"; "b"; "c"; "e"]);
 (("b", "d"), ["a"; "b"; "c"; "e"]); (("c", "d"), ["a"; "b"; "e"]);
 (("a", "c"), ["a"; "b"; "d"; "e"]); (("b", "c"), ["a"; "b"; "d"; "e"])]
#
```

This function can be defined by aggregation using the previously defined `addOnePair` function. In particular, fill in the blanks below to complete its definition:

```
let addAllPairs ppls l = List.fold_left ___ l ppls
```

3. The only remaining thing is to put the mapping and the aggregation operations together in a map-reduce combination to get a solution to the problem. Use the following code to do this

```
let commonFriends l = List.fold_right addAllPairs (makeAllPairLists l) []
```

If you have defined the functions used in implementing `commonFriends` correctly, you should get behaviour of the following sort:

```
# commonFriends friendsList;;
- : ((string * string) * string list) list =
[(("b", "e"), ["c"; "d"]); (("c", "e"), ["b"; "d"]);
 (("d", "e"), ["b"; "c"]); (("a", "d"), ["b"; "c"]);
 (("b", "d"), ["a"; "c"; "e"]); (("c", "d"), ["a"; "b"; "e"]);
 (("a", "c"), ["b"; "d"]); (("b", "c"), ["a"; "d"; "e"]);
 (("a", "b"), ["c"; "d"])]
#
```

Problem 2 (3 points for each of the part (1), (3), (4) and (5), 2 points each for the other parts)

An ordered list is a list in which an element that appears later in the list is never smaller than an element that appears earlier in the list, where the notion of "smaller than" is given by a specific chosen relation. For example, the list

```
[1; 5; 7; 12; 13]
```

is an ordered list relative to the usual < relation on numbers, the list

```
[17; 14; 9; 6; 2]
```

is an ordered list relative to the > relation on numbers but not relative to the < order and the list

```
[17; 14; 9; 13; 2]
```

is an ordered list relative to neither the usual < nor the usual > relation on numbers.

1. Clearly, whether or not a list is ordered is dependent also on the ordering relation in use. Define a type constructor `olist` that is like the `list` type constructor except that values of this type carry enough information to determine whether or not they are ordered lists relative to the intended ordering relation.
2. Define a function

   ```
   initOList : ('a -> 'a -> bool) -> 'a olist
   ```

   that takes an ordering relation over a given type and returns an empty `olist` of that type.
3. Using let declarations, bind `list1` and `list2` to two different values of `int olist` type that use different ordering relations and that satisfy the invariant based on that ordering relation. Then bind `list3` to a value of `int olist` that does not satisfy the necessary invariant.
4. Define a function

   ```
   isOrderedList : ('a olist) -> bool
   ```

   that takes an ordered list and determines whether or not it satisfied the required ordering invariant based on the intended ordering relation. Test this function using the identifiers `list1`, `list2` and `list3`.
5. Define a function

   ```
   insertOList : 'a -> ('a olist) -> ('a olist)
   ```

   that takes an element and an ordered list and returns an ordered list that has all the original elements plus the new one and that continues to satisfy the invariant based on the relevant ordering property.
6. Define a function

   ```
   olistToList : 'a olist -> 'a list
   ```

   that takes an ordered list and converts it into an ordinary list with the same elements and in the same order as in the given olist.

Problem 3 (4 + 6 points)

Transform the functions shown below into a tail-recursive form by using continuations in the manner (to be) discussed in class. Use the names indicated for the transformed versions of the functions; we need you to do this for our automated testing tools to work.

1. The append function

   ```
   let rec append l1 l2 =
     match l1 with
     | [] -> l2
     | (h::t) -> h::append t l2
   ```

   The function you should define in this case is the following:

   ```
   cont_append : 'a list -> 'a list -> ('a list -> 'b) -> 'b
   ```

2. Suppose that we have defined the `btree` type constructor as follows:

   ```
   type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
   ```

   The function to transform is then the following:

   ```
   let rec sumTree t =
     match t with
     | Empty -> 0
     | Node (i,l,r) -> i + sumTree l + sumTree r
   ```

   The function you should define is the following:

   ```
   cont_sumTree : int btree -> (int -> 'a) -> 'a
   ```
