Homework Assignment #3

Due Monday, April 16, at the **start** of lecture. As always, turn in a typed hardcopy of your answers.

In all the problems on this and future ML homeworks, pattern-matching should be used in place of the accessors `null`, `hd`, `tl`, `#`*`int`* (tuple accessing), and `#`*`name`* (record accessing).

1. An association list is a list of key/value pairs, where the keys support equality testing, as in the following type synonym declaration:

   ```
   type (''k,'v) assoc_list = (''k * 'v) list
   ```

   Association lists are an implementation of a lookup table, mapping keys to corresponding values. For example, a `(string, real) assoc_list` could store a table mapping names of grocery items to their prices.

   To make use of association lists, we need four things:

   ```
   empty_assoc_list: (''k,'v) assoc_list
       (* the empty association list value *)
   store: (''k,'v) assoc_list * ''k * 'v -> (''k,'v) assoc_list
       (* a function that takes an association list alist, key k, and value v, and returns a new
          association list alist' that behaves like alist except that k maps to v. alist' should not have
          any unnecessary pairs in it (e.g. old unused bindings of k) *)
   fetch: (''k,'v) assoc_list * ''k -> 'v
       (* a function that takes an association list alist and a key k. if alist maps k to a value v, then v
          is returned, otherwise the exception NotFound is raised *)
   NotFound: an exception
       (* the exception raised by fetch for an unmapped key *)
   ```

   Along with the above type synonym declaration, implement these four things.

2. Use your association list operations to maintain the results for a sports leagues. Define a type synonym `Records` that is an association list mapping team names (strings) to team won-loss records (records of type `{wins:int, losses:int}`). Without making any direct list references, using only the association list operations, implement the following functions and exception to manipulate `Records`:

   ```
   create_league: string list -> Records
       (* a function that takes a list of team names and creates a Records association list mapping
          each team name to a (0,0) record *)
   record_game: Records * {winner:string, loser:string} -> Records
       (* a function that takes a Records association list and a record of the winning and losing
          team names, and returns a new Records association list where the winner has an extra
          win and the loser has an extra loss in their respective won-loss records, unless either the
          winner or loser team names are not in the league, in which case the NotInLeague
          exception is raised *)
   NotInLeague: an exception
       (* the exception raised by record_game for undefined team names *)
   ```

Implement these functions and exception, using the association list building blocks from problem 1. In addition, demonstrate the use of these functions to maintain the game results of your favorite real or imaginary sports league, by including a transcript of interactions with the SML interpreter. (You should not call any association list functions from problem 1 directly during this interaction, only the ones from this question.) (A transcript can be constructed by running `sml` from inside `emacs`, and then saving the `emacs` buffer containing the `sml` interaction to a file.)

3. We wish to print out a sorted standings for our sports league. We need three things:

```
better_record: {wins:int, losses:int} * {wins:int, losses:int} -> bool
```
(* a function that returns true if the first record is better than the second record, where one record is better than another if its number of wins minus its number of losses is greater than the other's wins minus losses *)

```
sort_standings: Records -> Records
```
(* a function that takes a Records association list and returns a new `Records` association list where each team in the list has a record at least as good as all teams later in the list (using `better_record` to judge). the function can manipulate the `Records` association list directly as a list, and should use the *quicksort* algorithm to sort the list. quicksort is a divide-and-conquer algorithm comprised of four steps. First, it picks a pivot element; this can be the first element of the list, or, for better performance, the middle element if the input list is already close to sorted. Second, it divides its input list (excluding the pivot) into two sublists: all those elements less than the pivot element, and all those elements greater than the pivot. Third, it recursively sorts the two sublists. Finally, it concatenates the two sorted sublists and the pivot element in the right order using the list append operator (@). *)

```
print_league: Records -> unit
```
(* a function that prints out a nicely formatted report on the teams and their records, in sorted order, using `sort_standings` and the `print` and `Int.toString` functions described in section 4.1 of the textbook. the function can access the list directly to recur through its elements *)

Implement these functions, and demonstrate how they work for your sports league.

4. Extra-credit challenge problem. One inefficiency of the ML version of the quicksort algorithm compared to the mergesort algorithm given in the textbook (section 3.4.4) is the use of append to combine the lists back together, causing one additional copy of each list "cons cell" during the sorting process. Rewrite the quicksort algorithm to avoid this extra copy. Hint: a helper function `quicksort_onto` that takes a list to sort and a second list onto the front of which to prepend the sorted list might come in handy. E.g. `quicksort_onto([3,1,5,2], [6,8,9])` → `[1,2,3,5,6,8,9]`.