MP 1 – Pattern Matching and Recursion

CS 421 – Spring 2013
Revision 1.0

Assigned January 17, 2013
Due January 22, 2013, 9:30am
Extension 48 hours (20% penalty)
Total points 40

1 Change Log

1.0 Initial Release.

2 Objectives and Background

The purpose of this MP is to help you master pairs, lists, pattern matching (on pairs and lists), and recursion (on lists).

3 Instructions

Define the functions listed below. You must use the same function names (but may choose different parameter names if you wish). We will sometimes use `let` and sometimes use `let rec` to begin the definition of a function; you can take this as a hint, although it is not a requirement; the only requirement is that the function work correctly. (In particular, you may find that it is easier to define an *auxiliary* function recursively and then define the required function just by calling the auxiliary one, so that the required function is not itself defined recursively. We note, however, that we did not find that necessary in writing the solutions.)

You can use any functions already defined in the OCaml modules, including pervasives (which you do not have to open explicitly) and `List` (which you do).

Some functions in this assignment have polymorphic types. These functions can be tested on non-integer inputs as well. Please be careful about your function types.

4 Problems

4.1 Pattern Matching

1. (2 pts) Write `pair_to_list : 'a * 'a -> 'a list` that takes a *pair* of two `'a` type elements and returns a *list* of two elements in the reversed order.

   Note: input can be a non-integer pair. Your function should be polymorphic.

   ```
   # let pair_to_list ... = ...;;
   val pair_to_list : 'a * 'a -> 'a list = <fun>
   # pair_to_list (5, 9);;
   - : int list = [9; 5]
   ```

2. (3 pts) Write `dist : (float * float) * (float * float) -> float` that takes two points (pairs of floats), and calculates the distance between them:

   Note: you may use `sqrt` function from the Pervasives module.

```
# let dist ... = ...;;
val dist : (float * float) * (float * float) -> float = <fun>
# dist ((1.0, 4.0), (0.0, 4.0));;
- : float = 1.
```

3. (4 pts) Write `sort_first_two : 'a list -> 'a list` that reverses the first two elements of a list if the first element is larger than the second element, or does nothing to a one- or zero-element list.

   Note: input can be a non-integer list. Your function should be polymorphic. This will actually be automatic, since you will use a comparison operation such as <, which is polymorphic (unlike, for example, +, which only takes integer arguments). The same comment applies to several of the problems that follow.

```
# let sort_first_two ... = ...;;
val sort_first_two : 'a list -> 'a list = <fun>
# sort_first_two [8; 2; 5];;
- : int list = [2; 8; 5]
# sort_first_two [3; 7; 4];;
- : int list = [3; 7; 4]
```

4.2 Recursion

4. (5 pts) Write `concat_odd : string list -> string` that concatenates the elements in odd positions of the input list, returning `""` on the empty input.

```
# let rec concat_odd l = ...;;
val concat_odd : string list -> string = <fun>
# concat_odd ["How "; "hey"; "are "; "things"; "you?"];;
- : string = "How are you?"
```

5. (5 pts) Write `is_sorted : 'a list -> bool` that returns true if the input list is in non-descending order, false otherwise.

   Note: input can be a non-integer list. Your function should be polymorphic.

```
# let rec is_sorted l = ...;;
val is_sorted : 'a list -> bool = <fun>
# is_sorted [1; 2; 3; 4; 4; 5; 8; 9; 11; 11];;
- : bool = true
# is_sorted [2; 3; 4; 5; 7; 6; 8];;
- : bool = false
```

6. (5 pts) Write `group_ascending: 'a list -> 'a list list` that puts all strictly increasing sequences into separate sublists.

   Note: input can be a non-integer list. Your function should be polymorphic.

```
# let rec group_ascending l = ...;;
val group_ascending : 'a list -> 'a list list = <fun>
# group_ascending [1; 2; 3; 3; 1; 4; 5; 4; 9];;
- : int list list = [[1; 2; 3]; [3]; [1; 4; 5]; [4; 9]]
```

7. (5 pts) Write `split_list: 'a list -> 'a list * 'a list` that divides a list into two lists of alternate elements.

```
# let rec split_list l = ...;;
val split_list : 'a list -> 'a list * 'a list = <fun>
#  split_list [1; 2; 3; 4; 5];;
- : int list * int list = ([1; 3; 5], [2; 4])
```

8. (5 pts) Write `merge : 'a list -> 'a list -> 'a list` whose arguments are two lists sorted in non-descending order, and whose result is the merging of the two lists, also in non-descending order.

   Note: input can be non-integer lists. Your function should be polymorphic.

```
# let rec merge ... = ...;;
val merge : 'a list -> 'a list -> 'a list = <fun>
# merge [1; 3; 5] [4; 5; 6];;
- : int list = [1; 3; 4; 5; 5; 6]
```

9. (6 pts) Write `mergesort : 'a list -> 'a list`. It should use functions `split_list` and `merge` defined above.

```
let rec mergesort lis = ...
val mergesort : 'a list -> 'a list = <fun>
# mergesort [1; 2; 3; 2; 1; 4; 5; 4; 9];;
- : int list = [1; 1; 2; 2; 3; 4; 4; 5; 9]
```