MP 1 – Pattern Matching and Recursion

CS 421 – Spring 2011
Revision 1.0

Assigned January 18, 2011
Due January 24, 2011, 23:59
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Added "strictly" to problem 7.

2 Objectives and Background

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

3 Collaboration

Collaboration is allowed in this assignment.

4 Instructions

The problems below have sample executions that suggest how to write answers. Students have to use the same function name, but the name of the parameters that follow the function name need not be duplicated. That is, the students are free to choose different names for the arguments to the functions from the ones given in the example execution. We will sometimes use `let rec` to begin the definition of a function, indicating that we have defined our solution recursively. This is intended as a hint, but the use of recursion is up to you. 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).

For all these problems, you are allowed to write your own auxiliary functions, either internally to the function being defined or externally as separate functions. In this assignment, **you may not use any library functions** (except `@` and `sqrt`, both of which are pervasive).

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

5 Problems

5.1 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 Ocaml standard library.

```
# 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.

```
# 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]
```

5.2 Recursion

4. (5 pts) Write `sum : int list -> int` that returns the sum of all elements in the list, or 0 if the list is empty.

```
# let rec sum l = ...;;
val sum : int list -> int = <fun>
# sum [0; 2; 5];;
- : int = 7
```

5. (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?"
```

6. (6 pts) Write a function `dotproduct : int list -> int list -> int` that calculates a dot-product of two lists (sum of products of respective elements). If one list is longer than the other one, then excessive elements are discarded. If either list is empty, `dotproduct` should return 0.

```
# let rec dotproduct l1 l2 = ...;;
val dotproduct : int list -> int list -> int = <fun>
# dotproduct [1;2] [3;4];;
- : int = 11
```

7. (5 pts) Write `is_sorted : 'a list -> bool` that returns true if the input list is sorted strictly ascendingly, 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;5;8;9;11];;
- : bool = true
# is_sorted [2;3;4;5;7;6;8];;
- : bool = false
```

8. (5 pts) Write `total_dist: (float * float) list -> float` that calculates the total distance from the first point in the list to the last; if the list has fewer than two elements, then the distance is zero. You can call the `dist` function you defined above.

```
# let rec total_dist l = ...;;
val total_dist : (float * float) list -> float = <fun>
# total_dist [(1.0,4.0); (4.0,0.0); (8.0, 3.0); (5.0, 7.0)];;
- : float = 15.
```

9. (6 pts) Write `merge : 'a list -> 'a list -> 'a list` whose arguments are two lists sorted in ascending order, and whose result is the merging of the two lists, also in ascending 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]
```

10. (6 pts) Write a function `firstelts : 'a list list -> 'a list` that takes a list of lists $\ell_0, \ell_1, \ldots$ and returns a list containing the first elements of each of ℓ_0, ℓ_1, etc. If any of the contained lists is empty, then it is skipped.

```
# let rec firstelts lis = ...
val firstelts : 'a list list -> 'a list = <fun>
# firstelts  [[1;2]; [3;4;5]; []; [6;7]];;
- : int list = [1; 3; 6]
```

11. (7 pts) Write a function `group : 'a list -> 'a list list` that takes a list of elements and groups each sequence of consecutive equal elements in a list to a separate list, producing a list of lists.

```
# let rec group l = ...
val group : 'a list -> 'a list list = <fun>
# group [1;1;2;3;1;4;5;5;6];;
- : int list list = [[1; 1]; [2]; [3]; [1]; [4]; [5; 5]; [6]]
# group [1;1;2;2;1;1];;
- : int list list = [[1; 1]; [2; 2]; [1; 1]]
```