MP 2 – Pattern Matching and Recursion

CS 421 – Fall 2012
Revision 1.0

Assigned September 4, 2012
Due September 11, 2012 23:59
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Objectives and Background

The purpose of this MP is to help the student master:

- pattern matching
- higher-order functions
- recursion

3 Instructions

The problems below have sample executions that suggest how to write answers. You have to use the same function name, but the name of the parameters that follow the function name need not be duplicated. That is, you 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 that is allowed to use `rec`. You are not required to start your code with `let rec`, and you may use `let rec` when we do not.

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 fact, you will find it helpful to do so on several problems. In this assignment, **you may not use any library functions other than @ and mod** except where explicitly noted.

4 Problems

1. (3 pts) Complex numbers can be represented as a pair of floating point numbers, the real part and the imaginary part. Complex multiplication is defined as $(x + yi)(u + vi) = (xu - yv) + (xv + yu)i$. Write `com_mul : (float * float) * (float * float) -> (float * float)` that takes two complex numbers as a pair of pairs of floating point numbers, and output the multiplication of them.

```
# let com_mul r = ...;;
val com_mul : (float * float) * (float * float) -> float * float = <fun>
# com_mul ((4., 0.), (2., 3.));;
- : float * float = (8., 12.)
```

2. (3 pts) Consider the following mathematical definition of a sequence s_n:

$$s_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2s_{(n-2)} + 2n & \text{if } n > 0 \text{ and } n = 2m \\ 3s_{(n-2)} + 3n & \text{if } n > 1 \text{ and } n = 2m + 1 \end{cases}$$

Write a Ocaml function `s : int -> int` that implements the sequence s_n. For $n \leq 0$, you should return 0.

```
# let rec s n = ...;;
val s : int -> int = <fun>
# s 5;;
- : int = 51
```

3. (3 pts) Write a function `list_all : ('a -> bool) -> 'a list -> bool` that takes a predicate and a list and decides if all the elements of the list satisfies the predicate.

```
# let rec list_all p xs = ...;;
val list_all : ('a -> bool) -> 'a list -> bool = <fun>
# list_all (fun x -> x < 0) [1;-1;0;4;-2;5];;
- : bool = false
```

4. (3 pts) Write a function `is_less : 'a -> 'a list -> bool` that decides if the first argument is less than all the elements of the list.

```
# let rec is_less x xs = ...
val is_less : 'a -> 'a list -> bool = <fun>
# is_less 5 [1;34;42;6];;
- : bool = false
```

5. (5 pts) Write a function `interleave : 'a list -> 'a list -> 'a list` that takes two lists and returns a list. The first element of the new list should be the first element of the first list and the second element of the new list should be the first element of the second list; then, the third element of the new list will be the second element of the first list and the fourth element of the new list will be the second element of the second list, and so on. If one list is longer than the other, put the extra elements on the end of the new list. Also, if either list is empty, `interleave` returns the other list.

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

6. (5 pts) Write a function `separate : 'a list -> 'a list * 'a list` that takes a list and outputs a pair of lists with the elements of first list containing all the even positions of the original list and the elements of the second list contains all the odd positions of the original list. The counting of the list positions starts from zero.

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

7. (7 pts) Suppose that we have a record of discrete events, ordered 1 through n, occurring in a given sample space. We may record the frequency distribution of these events in that space by a list $[f_1, \ldots, fn]$ where f_i is the number of occurrences of the i^{th} event in the sample space. Then, the probability p_i of the i^{th} event occurring in the sample space is given by

$$p_i = \frac{f_i}{\Sigma_i f_i}$$

Let's have some fun here. Write a function `odds : int list -> (int * int) list` that takes a list of intergers and returns a list of pairs of intergers, where the first element of the pair is the numerator of a rational number and the second element of the pair is the denominator of the rational number, the the rational number represented by the i^{th} element is the probabilty of the i^{th} event occuring.

Hint: You may want to write one or more auxiliary functions.

```
# let rec odds xs = ...;;
val odds xs : int list -> (int * int) list = <fun>
# odds [3;7;9];;
- : (int * int) list = [(3, 19); (7, 19); (9, 19)]
```

8. (7 pts) Recall that a directed graph is a pair $G = (V, A)$ where

- V is a set of vertices or nodes,
- A is a set of ordered pairs of vertices, called arcs, directed edges, or arrows.

When we implements the directd graph, we usually use an initial interval of integers to represent V, and an adjacency matrix or list to represent the set A.

In this problem we will use an adjacency list to represent the set A. This means that we will use the integer list list to represent the adjacency list of a graph, and it has one list per node in the integer list list. The i^{th} position of the list contains all the nodes connected to the i^{th} node by an out-going edge, where the integers in each list represent the position of the corresponding node in the adjacency list. The labeling of nodes start from zero, and elements in the list are counted from zero. Write a function `check_adj : int list list -> int * int -> bool` that takes an adjacency list and a pair of integers to check if there exists a edge from the first element to the second one. Remember that this is a directed graph. In this problem, you may use any library functions you choose. Also, the input integers in the lists will always be nonnegative, and you can safely assume that we will not test on any number which is bigger than the graph size minus one. However, if the input pair of integers contains a negative number, you should return false. Note OCamls type inference may infer a more general type for your solution than the type listed above, so do not panic.

```
# let check_adj adj_list (a,b) = ...
val check_adj : 'a list list -> int * 'a -> bool = <fun>
# check_adj [[1;2;3;4];[3;0;4;5];[1;4;3;5];[2;1];[1;2];[2;3;4]] (0,3);;
- : bool = true
```

4.1 Extra Credit

9. (5 pts)

The description of and implementaion of graphs is the same as in Problem 8. Write a function `check_path : int list list -> int * int -> bool` that takes an adjacency list and a pair of integers to check if there exists a *path* from the first element to the second one. A path is the transitive closure of adjacency. That is, if b is adjacent to a, then there is a path from a to b, and if there is a path for a to some third node c and a further path from c to b, then there is a path from a to b. The node a will not have a path to itself if the graph does not specify it explicitly. Remember that this is a directed graph. As before, the input integers in the lists will always be nonnegative, and you can safely assume that we will not test on any number which is bigger than the graph size minus one. However, if the input pair of integers contains a negative number, you should return false. In this problem, you may use any library function of your choosing. Note OCaml's type inference may infer a more general type for your solution, so do not panic.

```
# let rec check_path adj_list (a,b) = ...
val check_path : int list list -> int * int -> bool = <fun>
# check_path [[1;2;3;4];[3;0;4;5];[1;4;3;5];[2;1];[1;2];[2;3;4]] (0,5);;
- : bool = true
```