# MP 3 – Patterns of Recursion, Higher-order Functions

CS 421 – Fall 2012
Revision 1.0

**Assigned** September 11, 2012
**Due** September 18, 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:

1. forward recursion and tail recursion
2. higher-order functions

## 3 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 also will use `let rec` to begin the definition of some of the functions that are allowed to use recursion. You are not required to start your code with `let rec`. Similarly, if you are not prohibited from using explicit recursion in a given problem, you may change any function defintion from starting with just `let` to starting with `let rec`.

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. All helper functions must satisfy any coding restrictions (such as being in tail recursive form, or not using explicit recursion) as the main function being defined for the problem must satisfy.

Here is a list of the strict requirements for the assignment.

- The function name must be the same as the one provided.
- The type of parameters must be the same as the parameters shown in sample execution.
- Students must comply with any special restrictions for each problem. Some of the problems require that the solution should be in forward recursive form or tail-recursive form, while others ask students to use higher-order functions in place of recursion.

## 4 Problems

- In problems 1 and 2 you **must** use forward recursion.
- In problems 3 and 4 you **must** use tail recursion.

- In problems 5 – 7, you **may not** use recursion.

**Note:** All library functions are off limits for all problems in this assignment, except those that are specifically mentioned as required/allowed. For purposes of this assignment `@` is treated as a library function and is not to be used.

## 4.1 Patterns of recursion

1. (4 pts) Write a function `split`, that, when applied to a test function `f`, and a list `lst`, returns a pair of lists. The first list of the pair should contain every element `x` of `lst` for which `(f x)` is true; and the second list contains every element for which `(f x)` is false. The order of the elements in the returned lists should be the same as in the original list. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec split f lst = ...;;
val split : ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>
# split (fun x -> x > 2) [0;2;3;5;1;4];;
- : int list * int list = ([3; 5; 4], [0; 2; 1])
```

2. (5 pts) *Run-length encoding* (RLE) is a data compression technique in which maximal (non-empty) consecutive occurrences of a value are replaced by pa pair of the value and a counter showing how many times the value was repeated in that consecutive sequence. For example, RLE would encode the list `[1;1;1;2;2;2;3;1;1;1]` as: `[(1,3);(2,3);(3,1);(1;3)]`. Write a function `rle : 'a list -> ('a * int) list = <fun>` that takes a list and encodes it using the RLE technique. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

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

3. (5 pts) For two lists $L_1$ and $L_2$, $L_2$ is called a *sub-list* of $L_1$ if: (a) all the elements of $L_2$ occur in $L_1$, and (b) their order in $L_1$ is *exactly* the same as their order in $L_2$. Write a function `sub_list : 'a list -> 'a list -> bool = <fun>` that takes two lists as input and determines whether the second list is a sub-list of the first one. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec sub_list l1 l2 = ...;;
val sub_list : 'a list -> 'a list -> bool = <fun>
# sub_list [1;1;2;1;1;4;1] [1;2;1;1;1];;
- : bool = true
```

4. (7 pts) Write a function `concat : string -> string list -> string` such that `concat s l` creates a string consisting of the strings in the list `l` concatenated together, with the first string `s` inserted between. If the list is empty, you should return the empty string (`""`). If the list is a singleton, you should return just the single string in that list. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec concat s list = ... ;;
val concat : string -> string list -> string = <fun>
# concat " * " ["3"; "6"; "2"];;
- : string = "3 * 6 * 2"
```

## 4.2 Higher order functions

5. (5 pts) Write a base value `split_base` and a step function `split_step` such that `List.fold_right (split_step f) lst split_base` behaves the same as the `split f lst` as described in Problem 1.

```
# let split_base = ...
val split_base : 'a list * 'b list = ...
# let split_step f x (true_xs, false_xs) = ...
val split_step : ('a -> bool) -> 'a -> 'a list * 'a list -> 'a list * 'a list =
  <fun>
List.fold_right (split_step (fun x -> x > 2)) [0;2;3;5;1;4] split_base;;
- : int list * int list = ([3; 5; 4], [0; 2; 1])
```

6. (5 pts) Write a function `rle2:'a list -> ('a * int) list` that computes the same results as `rle` from Problem 2. The definition of `rle2` may use `List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` but no direct use of recursion, and no other library functions.

```
# let rle2 lst = ... ;;
val rle2 : 'a list -> ('a * int) list = <fun>
# rle2 [1;1;1;2;2;2;3;1;1;1];;
- : (int * int) list = [(1, 3); (2, 3); (3, 1); (1, 3)]
```

7. (7 pts) Write a function `concat2 : string -> string list -> string` that computes the same results as `concat` of Problem 4. The definition of `concat2` may use `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` but no direct use of recursion, and no other library functions.

```
# let concat2 s list = ...;;
val concat2 : string -> string list -> string = <fun>
# concat2 " * " ["3"; "6"; "2"];;
- : string = "3 * 6 * 2"
```

8. (8 pts) Write a function `app_all : ('a -> 'b) list -> 'a list -> 'b list list` that takes a list of functions, and a list of arguments for those functions, and returns the list of list of results from consecutively applying the functions to all arguments, in the order in which the functions occur in the list and in the order in which the arguments occur in the list. Each list in the result list corresponds to a list of applications of each function to the given arguments. The definition of `app_all` may use the library function `List.map : ('a -> 'b) -> 'a list -> 'b list` but no direct use of recursion, and no other library function.

```
# let app_all fs list = ... ;;
val app_all : ('a -> 'b) list -> 'a list -> 'b list list = <fun>
# app_all [(fun x -> x > 0); (fun y -> y mod 2 = 0);
  (fun x -> x * x = x)] [1; 3; 6];;
- : bool list list =
[[true; false; true]; [true; false; false]; [true; true; false]]
```

## 4.3 Extra Credit

9. (4 pts) Write a function `sub_list2` that computes the same results as `sub_list` of Problem 3. The definition of `sub_list2` may use `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` but no direct use of recursion, and no other library functions.

```
# let sub_list2 l1 l2 = ...;;
val sub_list : 'a list -> 'a list -> bool = <fun>
# sub_list2 [1;1;2;1;1;4;1] [1;2;1;1;1];;
- : bool = true
```