MP 3 – Patterns of Recursion, Higher-order Functions

CS 421 – Spring 2014
Revision 1.0

Assigned February 6, 2014
Due February 16, 2014 11:59pm

1 Objectives and Background

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

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

2 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 definition 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.

3 Problems

- In problems 1 – 3 you **must** use forward recursion.
- In problems 4 – 6 you **must** use tail recursion.
- In problems 7 – 12, you **must not** use recursion.

Note: Any auxiliary functions implemented in any recursion problem **must** also follow the recursion form specified in the problem statement.
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.

3.1 Patterns of Recursion

1. (3 pts) Write a function `even_count : int list -> int` such that it returns the number of even integers found in the input list. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec even_count l = ...;;
val even_count : int list -> int = <fun>
# even_count [1;2;3];;
- : int = 1
```

2. (3 pts) Write a function `split_sum : int list -> (int -> bool) -> int * int` such that it returns a pair of integers. The first integer in the pair is the sum of all number in the input list `l` where the input function `f` return true. The second is the sum of all remaining number that `f` return false. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec split_sum l f = ...;;
val split_sum : int list -> (int -> bool) -> int * int = <fun>
# split_sum [1;2;3] (fun x -> x>1);;
- : int * int = (5, 1)
```

3. (3 pts) Write a function `merge : 'a list -> 'a list -> 'a list` such that `merge [x1; x2; ...] [y1; y2; ...]` returns a list `[x1; y1; x2; ...]`. If one list is longer than another, extra elements will be appends at the end of the merged list in the same order as they appear 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 merge l1 l2 = ...;;
val merge : 'a list -> 'a list -> 'a list = <fun>
# merge [1;3;5] [2;4];;
- : int list = [1; 2; 3; 4;5]
```

4. (4 pts) Write a function `range_count : 'a list -> 'a -> 'a -> int` such that `range_count l m n` returns the number of elements in the input list `l` which are strictly greater than `m` and strictly less than `n`. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec range_count l m n = ... ;;
val range_count : 'a list -> 'a -> 'a -> int = <fun>
# range_count [0;1;2;3] 1 3;;
- : int = 1
```

5. (5 pts) Write a function `max_index : 'a list -> int list` that given an input list returns a list of all indices of the elements with the maximum value. The function returns list of indices in descending order. Note that the index of the first element is 0. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec max_index = ...;;
val max_index : 'a list -> int list = <fun>
# max_index [1;2;1];;
- : int list = [1]
```

6. (5 pts) Write a function `unique_to_neighbor : 'a list -> 'a list` such that `unique_to_neighbor l` returns a list with the same elements and order in `l` and no two adjacent elements in the returned list have the same value. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec unique_to_neighbor = ...;;
val unique_to_neighbor : 'a list -> 'a list = <fun>
# unique_to_neighbor [1;2;2;3;2];;
- : int list = [1; 2; 3; 2]
```

3.2 Higher Order Functions

For problems 7 through 9, you will be supplying arguments to the higher-order functions `List.fold_right` and `List.fold_left`. You should not need to use explicit recursion for any of these problems.

7. (6 pts) Write a value `remove_if_base : 'a list` and a function `remove_if_rec : ('a -> bool) -> 'a -> 'a list` such that `(fun l -> fun f -> List.fold_right (remove_if_rec f) l remove_if_base)` returns a list in the same order as input list `l`, but with all element for which `f` returns true removed. There should be no use of recursion or library functions in the solution to this problem.

```
# remove_if_base = ...;;
val remove_if_base : 'a list = ...
# remove_if_rec = ...;;
val remove_if_rec : ('a -> bool) -> 'a -> 'a list -> 'a list = <fun>
# let remove_if l f = List.fold_right (remove_if_rec f) l remove_if_base;;
val remove_if : 'a list -> ('a -> bool) -> 'a list = <fun>
# remove_if [1;2;3] (fun x -> x=2);;
-: int list = [1; 3]
```

8. (5 pts) Write a value `split_sum_base : int * int` and function `split_sum_rec : ((int -> bool) -> int -> int * int -> int * int` such that `(fun l -> fun f -> List.fold_right (split_sum_rec f) l split_sum_base)` computes the same solution as `split_sum` defined in Problem 2. There should be no use of recursion or library functions in the solution to this problem.

```
# let split_sum_base = ...;;
val split_sum_base : int * int = ...
# let split_sub_rec = ...;;
val split_sum_rec : (int -> bool) -> int -> int * int -> int * int = <fun>
# let split_sum l f = List.fold_right (split_sum_rec f) l split_sum_base;;
```

```
val split_sum : int list -> (int -> bool) -> int * int = <fun>
# split_sum [1;2;3] (fun x -> x>1);;
- : int * int = (5, 1)
```

9. (5 pts) Write a value `all_positive_base : bool` and function `all_positive_rec : bool -> int -> bool` such that `(fun l -> List.fold_left all_positive_rec all_positive_base l)` returns true if all elements in the input list are positive, and false otherwise. There should be no use of recursion or library functions in the solution to this problem.

```
# let all_positive_base = ...;;
val all_positive_base : bool = ...
# let all_positive_rec = ...;;
val all_positive_rec : bool -> int -> bool = <fun>
# let all_positive l = List.fold_left all_positive_rec all_positive_base l;;
val all_positive : int list -> bool = <fun>
# all_positive [1;2];;
- : bool = true
```

10. (5 pts) Write a value `range_count_base : int` and function `range_count_rec : 'a -> 'a -> int -> 'a -> int` such that `(fun l -> fun m -> fun n -> List.fold_left (range_count m n) range_count_base l)` computes the same results as `range_count` defined in Problem 4. There should be no use of recursion or library functions in the solution to this problem.

```
# let range_count_base = ...;;
val range_count_base : int = ...
# let range_count_rec = ...;;
val range_count_rec : 'a -> 'a -> int -> 'a -> int = <fun>
# let range_count l m n = List.fold_left (range_count_rec m n) range_count_base l;;
val range_count : 'a list -> 'a -> 'a -> int = <fun>
# range_count [0;1;2;3] 1 3;;
- : int = 1
```

11. (5 pts) Write a function `app_all_with : ('a -> 'b -> 'c) list -> 'a -> 'b list -> 'c list list` that takes a list of functions, a single first argument, and a list of second arguments and returns a list of list of results from consecutively applying the functions to all arguments after applying the single argument to each function first. The functions should be applied in the order they appear in the list and in the order in which the arguments appear in the second list. Each list in the result corresponds to a list of applications of a function on the single argument and then the given argument from the second list. There should be no use of recursion or library functions except `List.map` in the solution to this problem

```
let app_all_with fs b l = ...;
val app_all_with : ('a -> 'b -> 'c) list -> 'a -> 'b list -> 'c list list = <fun>
# app_all_with [(fun x y -> x*y); (fun x y -> x+y)] 10 [-1;0;1];;
- : int list list = [[-10; 0; 10]; [9; 10; 11]]
```

3.3 Extra Credit

12. (5 pts) *Run-length encoding*(RLE) is a data compression technique in which maximal (non-empty) consecutive occurrences of a value are replaced by a 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` that takes a list and encode it using RLE technique. There should be no use of recursion or library functions except `List.fold_left` in the solution to this problem.

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