MP 9 – Higher-order Functions

CS 421 – Spring 2009

Revision 1.0

Assigned Mar 19, 2009
Due Apr 2, 2009 23:59
Extension 48 hours (20% penalty)
Total Points 30

1 Change Log

1.0 Initial Release.

2 Objectives and Background

The purpose of this MP is to help the student master higher-order functions

3 Collaboration

Collaboration is NOT 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 also will 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`.

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 use higher-order functions in place of recursion.

5 Included Helper Functions

Here are some functions used in this assignement; they are included in mp9common.ml so you do not have to write them yourself; you can use them as desired. You may want to look at mp9common.ml to get a feel for what is available.

In some cases, they are used just for examples; in others, they are used in our solutions, so you might find them helpful in yours.

```
let incr x = x + 1;;
let decr x = x - 1;;
let funmod f x y = fun z -> if x = z then y else f z;;
let compose f g = fun x -> f (g x);;
```

6 Problems

1. (6 pts) Define `split` such that `split f lis` returns a pair of lists `(lis1, lis2)` where `lis1` contains all the elements of `lis` for which `f` is true, and `lis2` contains all the others (in the same order as they appear in `lis`)

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

2. (6 pts) Define `filter` such that `filter f lis` returns a list containing the elements of `lis` for which `f` is true. Then use `filter` to define `filterlt` such that `filterlt n lis` returns all the elements of `lis` that are less than or equal to n.

```
# let rec filter f lis = ...
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
# let filterlt n = filter ...
val filterlt : 'a -> 'a list -> 'a list = <fun>
# filterlt 10 [4; 25; 12; 8; 10; 7; 11];;
- : int list = [4; 8; 10; 7]
```

3. (6 pts) For this problem, you must use `fold_right` from the List module, and no explicit recursion. Define a function `length`, which gives the length of a list, by defining value `length_base` and function `length_recur`, and calling `fold_right length_recur lis length_base`. The function `length_recur` takes as arguments the head of the list and the result of the recursive call on the tail of the list, and gives the result for this list.

```
# let length_base = ...;;
val length_base : int = ...
# let length_recur h x = ...;;
val length_recur : 'a -> int -> int = <fun>
# let length lis = fold_right length_recur lis length_base;;
val length : 'a list -> int = <fun>
# length [1;2;3;4;5];;
- : int = 5
```

4. (6 pts) Using `map` from the List module, and no explicit recursion, define the function `case_map` such that `case_map f g h lis` returns a list with the following values: wherever `lis` has a value x for which `f` is true, it has `g x`, and everywhere else it has `h x`.

```
# let case_map f g h = map ...;;
val case_map : ('a -> bool) -> ('a -> 'b) -> ('a -> 'b) -> 'a list -> 'b list = <fun>
# case_map (fun x -> x > 2) incr decr [0;1;2;3];;
- : int list = [-1; 0; 1; 4]
```

5. (6 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 functions.

```
# let app_all fs lis = ... ;;
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; true; true]; [false; false; true]; [true; false; false]]
```