MP 3 – Patterns of Recursion, Higher-order Functions

CS 421 – Fall 2014
Revision 1.2

Assigned September 9, 2014
Due September 16, 2012 23:59
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Fixed type errors in Problems 9, 13, and 14.

1.2 Added hint to Problem 8. (Make sure to download the version 1.1 tarball for associated solution fix.)

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 8 – 9 you **must** use ONLY forward recursion.
- In problems 10 – 11 you **must** use ONLY tail recursion.
- In problems 12 – 14, 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.

1. (3 pts) Write a function `get_nth : 'a list -> int -> 'a list` that returns the n^{th} element of a list as a list with a single element or an empty list if the index is out of the list bounds. We count from the left at 0, incrementing to the right.

```
# let rec get_nth l n = ... ;;
val get_nth : 'a list -> int -> 'a list = <fun>
# get_nth ["a"; "hello"; "x"] 1;;
- : string list = ["hello"]
```

2. (3 pts) Write a recursive function `count_change: int -> int list -> int` that counts how many different ways you can make change for an amount, given a list of coin denominations. For example, there are 3 ways to give change for 4 if you have coins with denomination 1 and 2: 1+1+1+1, 1+1+2, 2+2.

```
# let rec count_change money coins = ... ;;
val count_change : int -> int list -> int = <fun>
# count_change 4 [1; 2];;
- : int = 3
```

3. (2 pts) Write a function `elems_in_interval: 'a list -> ('a * 'a) -> int` that, given a list of elements and an interval: (lo, hi) - a pair of elements, returns how many list elements belong to the interval: an element el belongs to the interval iff $lo \leq x \leq hi$.

```
# let rec elems_in_interval l i = ... ;;
val elems_in_interval : 'a list -> 'a * 'a -> int = <fun>
# elems_in_interval ["java"; "ocaml"; "great"; "weird"] ("g", "o");;
- : int = 2
```

4. (2 pts) Write a function `check_if_all_pos : int list -> bool` that checks if all numbers in the list are positive integers.

```
# let rec check_if_all_pos l = ... ;;
# check_if_all_pos [2; 4; 5; 7];;
- : bool = true
```

5. (3 pts) Write a function `sum_quads: (int * int * int * int) list -> int list` that, given a list of quads (4-tuples), produces a list of sums of those quads. Hint: a local function that adds up quad component leads to an elegant solution.

```
# let rec sum_quads qs = ...;;
val sum_quads :  (int * int * int * int) list -> int list = <fun>
# sum_quads [(1, 1, 1, 1); (0, 0, 0, 1); (2, 2, 2, 2)];;
- : int list = [4; 1; 8]
```

6. (4 pts) Write a function `evens_odds : int list -> int * int` that, given a list of integers, returns the pair where the first element denotes the number of odd list elements and the second element denotes the number of even list elements. As in Problem 6, we start counting list positions from 0 starting on the left. Hint: use auxiliary local function.

```
# let rec evens_odds l = ... ;;
val evens_odds : int list -> int * int = <fun>
# evens_odds [0; 1; 2; 3; 4; 5; 6];;
- : int * int = (3, 4)
```

7. (4 pts) Write a function `inc_evens_dec_odds: int list -> int list` that increments all elements at even positions in the list and decrements all elements at odd positions. As above, assume that the first list element is at position zero. Hint: use auxiliary local functions.

```
# let rec inc_evens_dec_odds l = ;;
val inc_evens_dec_odds : int list -> int list = <fun>
# inc_evens_dec_odds [0; 1; 2; 3; 4; 5];;
- : int list = [1; 0; 3; 2; 5; 4]
```

4.1 Patterns of Recursion

8. (4 pts) Write a function `odd_sum : int list -> int` such that it returns the sum of odd 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. BEWARE: When `x` is negative and `y` is positive, `x mod y` returns 0 or a negative value. (Run some tests in OCaml to check your understanding of how this works.)

```
# let rec odd_sum l = ...;;
val odd_sum : int list -> int = <fun>
# odd_sum [1;2;3];;
- : int = 4
```

9. (3 pts) Write a function `pair_sums : (int * int) list -> int list` that takes a list of pairs of integers and returns a list of the sums of those pairs in the same order. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
let rec pair_sums l = ...;
val pair_sums : (int * int) list -> int list = <fun>
# pair_sums [(1,6);(3,1);(3,2)];;
- : int list = [7;4;5]
```

10. (4 pts) Write a function `count_element : 'a list -> 'a -> int` such that `count_element l m` returns the number of elements in the input list `l` which are equal to `m`. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec count_element l m = ... ;;
val count_element : 'a list -> 'a -> int = <fun>
# count_element [0;1;2;4;2;5;4;2] 2;;
- : int = 3
```

11. (6 pts) Write a function `merge_pairs : int list -> int list` such that `merge_pairs [x0; x1; x2; x3; ...]` returns a list `[x0 + x1; x2 + x3; ...]`. If the original list is odd in length, the last element will be appended at the end of the new list. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions. HINT: This will probably require you to write (at least) two auxiliary functions.

```
# let rec merge_pairs l = ...;;
val merge_pairs : int list -> int list = <fun>
# merge_pairs [7;3;5;2;4];;
- : int list = [10;7;4]
```

4.2 Higher Order Functions

For problems 12 through 14, 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.

12. (5 pts) Write a value `odd_sum_base : int` and function `odd_sum_rec : (int -> int -> int` such that `(fun l -> List.fold_right odd_sum_rec l odd_sum_base)` computes the same solution as `odd_sum` defined in Problem 8. There should be no use of recursion or library functions in the solution to this problem.

```
# let odd_sum_base = ...;;
val odd_sum_base : int = ...
# let odd_sum_rec = ...;;
val odd_sum_rec : int -> int -> int = <fun>
# let odd_sum l = List.fold_right odd_sum_rec l odd_sum_base;;
val odd_sum : int list -> int = <fun>
# odd_sum [1;2;3];;
- : int = 4
```

13. (5 pts) Write a value `count_element_base : int` and function `count_element_rec : 'a -> int -> 'a -> int` such that `(fun l -> fun m -> List.fold_left (count_element_rec m) count_element_base l)` computes the same results as `count_element` defined in Problem 10. There should be no use of recursion or library functions in the solution to this problem.

```
# let count_element_base = ...;;
val count_element_base : int = ...
# let count_element_rec = ...;;
val count_element_rec : 'a -> int -> 'a -> int = <fun>
# let count_element l m = List.fold_left (count_element_rec m) count_element_base l;;
```

```
val count_element : 'a list -> 'a -> int = <fun>
# count_element [0;1;3;2;1;1;3] 3;;
- : int = 2
```

14. (4 pts) Write a function `pair_sums_map : (int * int) list -> int list` that computes the same results as defined in Problem 9. There should be no use of recursion or library functions except `List.map` in the solution to this problem

```
let pair_sums_map l = ...;
val pair_sums_map : (int * int) list -> int list = <fun>
# pair_sums_map [(1,6);(3,1);(3,2)];;
- : int list = [7;4;5]
```

4.3 Extra Credit

15. (3 pts) Write a function `apply_even_odd : 'a list -> ('a -> 'b) -> ('a -> 'b) -> 'b list` such that `apply_even_odd [x0; x1; x2; x3; ...] f g` returns a list `[f x0; g x1; f x2; g x3; ...]`. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec apply_even_odd l f g = ...;;
val apply_even_odd : 'a list -> ('a -> 'b) -> ('a -> 'b) -> 'b list = <fun>
# apply_even_odd [1;2;3] (fun x -> x+1) (fun x -> x - 1);;
- : int list = [2; 1; 4];;
```