ML2 – Patterns of Recursion, Higher-order Functions

CS 421 – Fall 2017
Revision 1.1

Assigned September 14, 2017
Due September 20, 2017 – September 22, 2017
Extension None past the allowed lab sign-up time

1 Change Log

1.2 In Problem 17, `even_count_tr_step` had its arguments in the wrong order.

1.1 Problem 20 uses `fold_left`, nit `fold_right` as originally stated. Also I took away the answer to Problem 20, `sift_rec : ('a -> bool) -> 'a -> 'a list * 'a list -> 'a list * 'a list` since you will not have this in the CBTF.

1.0 Initial Release.

2 Objectives and Background

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

- forward recursion and tail recursion
- higher-order functions

Another purpose of MPs and MLs in general is to provide a framework to study for the exam. Several of the questions on the exam will appear similar to the MP and ML problems. By the time of the exam, your goal is to be able to solve any of the following problems with pen and paper in less than 2 minutes.

3 Done in Computer-Based Testing Facility

You are asked to sign up for a time next week to go to the CBTF, where you will be given a random portion (less than 25%) of this assignment to complete for your grade for the assignment. We recommend that you do the whole assignment beforehand to be comfortable with the problems, so you will be able to do the problems efficiently when you go in. To help yourself in that, please read the *Guide for Doing MPs* at the bottom of the `mps` webpage for the course.

Be aware that if we have difficulties with the CBTF, this assignment will revert to an MP and will be turned in, in full, in the same manner as MPs.

When working in the CBTF, or on EWS, you will need to use the current version of OCaml, version 4.05. To do so, you will probably need to execute

```
module load ocaml
ocaml -version
```

to make sure you are using the correct version.

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 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 supply non-recursive arguments to higher-order functions in place of recursion, or to directly use higher-order functions in place of 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.

5 Problems

Patterns of Recursion

Forward Recursion

For the problems in this section, you **must** use **forward recursion**.

1. (3 pts) Write a function `even_count_fr : 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 or problems later in this set.

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

2. (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]
```

3. (3 pts) Write a function `remove_even : int list -> int list` that returns a list in the same order as the input list, but with all the even numbers removed. The function is required to use (only) forward recursion (no other form of recursion). You may use `mod` for testing whether an integer is even. You may not use any library functions.

```
# let rec remove_even list = ... ;;
val remove_even : int list -> int list = <fun>
# remove_even [1; 4; 3; 7; 2; 8];;
- : int list = [1; 3; 7]
```

4. (3 pts) Write a function `sift : ('a -> bool) -> 'a list -> 'a list * 'a list` such that `sift p l` returns a pair of lists, the first containing all the elements of `l` for which `p` returns `true`, and the second containing all those for which `p` returns false. The lists should be in the same order as 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 sift p l = ... ;;
val sift : ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>
# sift (fun x -> x mod 2 = 0) [-3; 5; 2; -6];;
- : int list * int list = ([2; -6], [-3; 5])
```

5. (5 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];;
```

6. (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 consisting 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 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)]
```

Tail Recursion

For the problems in this section, you **must** use **tail recursion**.

7. (3 pts) Write a function `even_count_tr : int list -> int` such that it returns the number of even integers found in the input list. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions or earlier problems in this set. You may use `mod`.

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

8. (3 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` that 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
```

9. (3 pts) Write a function `all_nonneg : int list -> bool` that returns whether every element in the input list is greater than or equal to 0. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec all_nonneg list = ... ;;
val all_nonneg : int list -> bool = <fun>
# all_nonneg [4; 7; -3; 5];;
- : bool = false
```

10. (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 elements in the input list `l` where the input function `f` returns true. The second is the sum of all remaining elements for which `f` returns false. The function is required to use (only) tail 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)
```

11. (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 a 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]
```

12. (5 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"
```

Higher-order Functions

For the problems in this section, you **must not** use recursion.

13. (3 pts) Write a value `even_count_fr_base : int` and a function `even_count_fr_rec : int -> int -> int` such that `(fun l -> List.fold_right even_count_fr_rec l even_count_fr_base)` computes the same function as `even_count_fr` in Problem 1. There should be no use of recursion in the solution to this problem.

```
# let even_count_fr_base = ... ;;
val even_count_fr_base : int = ...
# let even_count_fr_rec x rec_val = ... ;;
val even_count_fr_rec : int -> int -> int = <fun>
# (fun l -> List.fold_right even_count_fr_rec l even_count_fr_base)
  [1; 2; 3];;
- : int = 1
```

14. (3 pts) Write a function `pair_sums_map_arg : (int * int) -> int` such that `List.map pair_sums_map_arg` computes the same results as `pair_map` defined in Problem 2. There should be no use of recursion in the solution to this problem.

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

15. (3 pts) Write a value `remove_even_base` and function `remove_even_rec : int -> int list -> int list` such that `(fun list -> List.fold_right remove_even_rec list remove_even_base)` computes the same results as `remove_even` of Problem 3. There should be no use of recursion or library functions in defining `remove_even_rec`.

```
# let remove_even_base = ... ;;
val remove_even_base : ...
# let remove_even_rec n r = ... ;;
val remove_even_rec : int -> int list -> int list = <fun>
# (fun list -> List.fold_right remove_even_rec list remove_even_base)
 [1; 4; 3; 7; 2; 8];;
# - : int list = [1; 3; 7]
```

16. (3 pts) Write a value `sift_base` and function `sift_rec : ('a -> bool) -> 'a -> 'a list * 'a list -> 'a list * 'a list` such that `(fun p -> fun list -> List.fold_right (sift_rec p) list sift_base)` computes the same results as `sift` of Problem 4. There should be no use of recursion or library functions in defining `sift_rec`.

```
# let sift_base = ... ;;
val sift_base : ...
# let sift_rec p x (tl, fl) = ... ;;
val sift_rec : ('a -> bool) -> 'a -> 'a list * 'a list -> 'a list * 'a list =
  <fun>
# (fun p -> fun list -> List.fold_right (sift_rec p) list sift_base)
   (fun x -> x mod 2 = 0)
   [-3; 5; 2; -6];;
- : int list * int list = ([2; -6], [-3; 5])
```

17. (3 pts) Write a value `even_count_tr_start : int` and a function `even_count_tr_step : int -> int -> int` such that `(List.fold_left even_count_tr_step even_count_tr_start)` computes the same function as `even_count_tr` in Problem 7. There should be no use of recursion in the solution to this problem.

```
# let even_count_tr_start = ... ;;
val even_count_tr_start : int = ...
# let even_count_tr_step rec_val x = ... ;;
val even_count_tr_step : int -> int -> int = <fun>
# List.fold_left even_count_tr_step even_count_tr_start [1; 2; 3];;
- : int = 1
```

18. (3 pts) Write a value `count_element_start : int` and function `count_element_step : 'a -> int -> 'a -> int` such that `(fun l -> fun m -> List.fold_left (count_element_step m) count_element_start l)` computes the same results as `count_element` defined in Problem 8. There should be no use of recursion or library functions in the solution to this problem.

```
# let count_element_start = ...;;
val count_element_start : int = ...
# let count_element_step = ...;;
val count_element_step : 'a -> int -> 'a -> int = <fun>
# (fun l -> fun m -> List.fold_left
                     (count_element_step m)
                     count_element_start l)
  [0;1;2;4;2;5;4;2] 2;;
- : int = 3
```

19. (3 pts) Write a value `all_nonneg_start` and function `all_nonneg_step : bool -> int -> bool` such that `List.fold_left all_nonneg_step all_nonneg_start` computes the same results as `all_nonneg` of Problem 9. There should be no use of recursion or library functions in defining `all_nonneg_step`.

```
# let all_nonneg_start = ...
# let all_nonneg_step r x = ... ;;
```

```
val all_nonneg_step : bool -> int -> bool = <fun>
# List.fold_left all_nonneg_step all_nonneg_start [4; 7; -3; 5];;
- : bool = false
```

20. (3 pts) Write a value `split_sum_start : int * int` and function `split_sum_step : (int -> bool) -> int * int -> int -> int * int` such that `(fun l -> fun f -> List.fold_left (split_sum_step f) split_sum_start l)` computes the same solution as `split_sum` defined in Problem 10. There should be no use of recursion or library functions in the solution to this problem.

```
# let split_sum_start = ...;;
val split_sum_start : int * int = ...
# let split_sub_step = ...;;
val split_sum_step : (int -> bool) -> int * int -> int -> int * int = <fun>
# (fun l -> fun f -> List.fold_left (split_sum_step f) split_sum_start l)
 [1;2;3] (fun x -> x>1);;
- : int * int = (5, 1)
```

21. (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]]
```

22. (5 pts) Write a value `exists_between_start : bool` and a function `exists_between_step: int -> int -> bool -> int -> bool` such that `(fun m -> fun n -> fun l -> List.fold_left (exists_between_step m n) exists_between_start l) m n l` returns true if there is an element `x` of `l` such the $m \leq x \leq n$. There should be no use of recursion or library functions in the solution to this problem.

```
# let exists_between_start = ...;;
val exists_between_start : bool = ...
# let exists_between_step m n b x = ...;;
val exists_between_step : 'a -> 'a -> bool -> 'a -> bool = <fun>
# (fun m -> fun n -> fun l -> List.fold_left
  (exists_between_step m n) exists_between_start l)
  5 10 [1; 20; 7; 9];;
- : bool = true
```

Extra Credit

23. (5 pts) Write functions `rev_append_base : 'a -> 'a` and `rev_append_rec : 'a -> ('a list -> 'b) -> 'a list -> 'b` such that applying `(fun fl -> List.fold_right rev_append_rec fl rev_append_base)` to two lists reverses the first list onto the front of the second. The definitions of `rev_append_base` and `rev_append_rec` may not use library functions or recursion.

```
# let rev_append_base = ... ;;
val rev_append_base : 'a -> 'a = <fun>
# let rev_append_rec = ... ;;
val rev_append_rec : 'a -> ('a list -> 'b) -> 'a list -> 'b = <fun>
# (fun fl -> List.fold_right rev_append_rec fl rev_append_base) [1; 6; 9] [7; 2];;
- : int list = [9; 6; 1; 7; 2]
```