MP 1 – Lists, Structural Recursion and Higher-Order Functions

CS 421 – Fall 2019
Revision 1.0

Assigned Saturday, September 14, 2019
Due Friday, September 20, 2019 22:00pm
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:

- pattern matching over lists
- recursion over lists
- forward and tail redcursion
- use of highe-order operators for recursion

3 Instructions

The problems below have sample executions that suggest how to write answers. You have to use the same function name, but the name of the parameters that follow the function name need not be duplicated. That is, you are free to choose different names, or patterns, for the arguments to the functions from the ones given in the example execution. We will sometimes 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`, and you may use `let rec` when we do not.

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. **In this assignment, you may only use library functions from the Pervasives module (the one that is loaded by default), expect where directly told otherwise. In particular, you must not use any functions from List. Also you may not use the infixed `@` function.**

Turn-in is via PrairieLearn. Ther eis no need to go the the CBTF for this.

4 Problems

4.1 Patern-Matching on Lists and Recursion

1. (2 pts) Write a function `product : float list -> float` to find the product of a list of floats. The product of an empty list is 1.0.

```
# let rec product l = ...
val product : float list -> float = <fun>
# product [2.; 3.; 4.];;
- : float = 24.
```

2. (2 pts) Write a function `double_all : float list -> float list` that takes a list of floats and returns back the list with all of the elements doubled.

```
# let rec double_all l = ...
val double_all : float list -> float list = <fun>
# double_all [1.5; -3.0; 0.; 2.2];;
- : float list = [3.; -6.; 0.; 4.4]
```

3. (3 pts) Write a function `pair_with_all : 'a -> 'b list -> ('a * 'b) list` that takes a value and a list, and creates a list of pairs where the given value is first in every pair, and the elements of the list are second in the pairs, in the same order as the original list.

```
# let rec pair_with_all x l = ...
val pair_with_all : 'a -> 'b list -> ('a * 'b) list = <fun>
# pair_with_all 1 ["a"; "b"; "c"];;
- : (int * string) list = [(1, "a"); (1, "b"); (1, "c")]
```

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

5. (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` 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
```

Patterns of Recursion

Forward Recursion

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

6. (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. You may use the infix function `mod : int -> int -> int`.

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

7. (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]
```

8. (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]
```

9. (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])
```

10. (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];;
```

Tail Recursion

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

11. (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
```

12. (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
```

13. (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
```

14. (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)
```

15. (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.

16. (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 6. 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
```

17. (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_sums` defined in Problem 7. 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]
```

18. (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 8. 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]
```

19. (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 11. 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 acc_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
```

20. (5 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 15. 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)
```