
MP 2 – Recursion and Higher-order Functions

CS 421 – Spring 2007

Revision 1.3

Assigned January 29, 2007

Due February 7, 2007 23:59

Extension 48 hours (20% penalty)

1 Change Log

1.3 Wrong question numbers in the first paragraph of Section 4 fixed.

1.2 Type of the function `smallest` corrected as `int list -> int`.

1.1 Typos corrected:

- Problem 8: “...and x otherwise.” changed to “...and v otherwise.”
- Problem 8: The return result of the example should read 3, not 1.

1.0 Initial Release.

2 Objectives and Background

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

1. tail recursion
2. higher-order functions
3. recursion over recursive datatypes

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 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 function. 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 tail-recursive form while others ask students to use higher-order functions in place of recursion.

4 Problems

For problems 1 through 8, you may **not** use library functions. You must use recursion instead.

The opposite applies to problems 13 through 15:

you may use `List.map`, `List.fold_left` and `List.fold_right`, and may **not** use recursion (except for that which exists in previously defined functions).

Note: All library functions not listed above are off limits for all problems on this assignment.

4.1 Recursion

For problems 1 through 8, you may **not** use library functions.

1. (5 pts) Write a function `remove_first_instance` that takes a list and an value and returns a list with the first instance of the value removed from the list. If the value does not occur in the list, the original list should be returned.

```
# let rec remove_first_instance lst value =
val remove_first_instance : 'a list -> 'a -> 'a list = <fun>
# remove_first_instance [1;2;3] 3;;
- : int list = [1; 2]
# remove_first_instance [1;2;3;4;1;2] 1;;
- : int list = [2; 3; 4; 1; 2]
```

2. (5 pts) Write a function `split`, that, when applied to a test function `f`, and a list `lst`, returns a pair of lists. The first list of the pair should contain every element `x` of `lst` for which `(f x)` is true; and the second list contains every element for which `(f x)` is false. The order of the elements in the returned lists should be the same as in the original list.

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

3. (7 pts) Write a function `prod_diff` that calculates the difference between the product of all the elements in the list and each element in the list.

```
# let rec prod_diff lst = ...
val prod_diff : int list -> int list = <fun>
# prod_diff [2;4;6];;
- : int list = [46; 44; 42]
```

4. (7 pts)

Write a function `first_and_rest` that, when applied to an integer `n` and a list `lst`, returns a pair of lists where the first list contains the first `n` elements of `lst`, and the second list contains the remaining elements. The elements in the returned lists should be in the same order as the elements in `lst`. If `lst` has fewer than `n` elements, the first list should be `lst` and the second should be `[]`.

```

# let rec first_and_rest n lst = ...;;
val first_and_rest : int -> 'a list -> 'a list * 'a list = <fun>
# first_and_rest 3 [1;2;3;4];;
- : int list * int list = ([1; 2; 3], [4])
# first_and_rest 3 [1;2];;
- : int list * int list = ([1; 2], [])

```

For the next 4 problems, all recursive functions must be in tail recursive form.

5. (5 pts) Write a function `exists_false` that takes a function `f` and a list as parameters and returns true if and only if there exists some element `x` in the list for which `(f x)` is false. This function MUST be tail recursive.

```

# let rec exists_false f lst = ...;;
val exists_false : ('a -> bool) -> 'a list -> bool = <fun>
# exists_false (fun x -> x>2) [3;4;2];;
- : bool = true
# exists_false (fun x -> x<3) [0;1;2];;
- : bool = false

```

6. (6 pts) Find the smallest element in a list. Write a function `smallest` that takes a list as input and finds the smallest element. If the list is empty, you should return 0. This function MUST be tail recursive.

```

# let rec smallest lst = ... ;;
val smallest : int list -> int = <fun>
# smallest [3;4;1];;
- : int = 1
# smallest [1;3;4;-1;-2;5];;
- : int = -2

```

7. (5 pts) Write a function `mult`, which when applied to a list of integers, returns the product of the integers in the list. If the list is empty, the result should be 1. This function MUST be tail recursive.

```

# let rec mult lst = ...;;
val mult : int list -> int = <fun>
# mult [];;
- : int = 1
# mult [1;2;3];;
- : int = 6

```

8. (5 pts) Write a function `find_last`, that takes a function `f`, a default value `v`, and a list as parameters. It returns the last element in the list `x` for which `(f x)` is true, if there is one, and `v` otherwise. This function MUST be tail recursive.

```

# let rec find_last f v lst = ...;;
val find_last : ('a -> bool) -> 'a -> 'a list -> 'a = <fun>
# find_last (fun x -> x > 0) 17 [1; 5; 0; (- 2); 3; (-1)];;
- : int = 3

```

Stop: go back and make sure you used no library functions for problems 1 through 8.

4.2 Using HOFs

For problems 9 through 12, you will be supplying arguments to the higher-order functions `List.fold_right` and `List.fold_left`. You should not need to use recursion or library functions for any of problems 9 through 12.

9. (5 pts) Write a base value `length_base` and a step function `length_step` such that `List.fold_right length_step lst length_base` calculates the length of the the list `lst`.

```
# let length_base = ...
val length_base : int ...
# let length_step x n = ...
# val length_step : 'a -> int -> int = <fun>
List.fold_right length_step [1;5;2] length_base;;
- : int = 3
```

10. (6 pts) Write a base value `split_base` and a step function `split_step` such that `List.fold_right (split_step f) lst split_base` behaves the same as the `split f lst` as described in problem 2.

```
# let split_base = ...
val split_base : 'a list * 'b list = ...
# let split_step f x (true_xs, false_xs) = ...
val split_step : ('a -> bool) -> 'a -> 'a list * 'a list -> 'a list * 'a list =
  <fun>
List.fold_right (split_step (fun x -> x > 2)) [0;2;3;5;1;4] split_base =
- : int list * int list = ([3; 5; 4], [0; 2; 1])
```

11. (5 pts) Write a base value `mult_base` and a step function `mult_step` such that `List.fold_left mult_step mult_base lst` behaves the same as the `mult lst` as described in problem 7.

```
# let mult_base = ...
val mult_base : int = ...
# let mult_step p x = ...
val mult_step : int -> int -> int = <fun>
# List.fold_left mult_step mult_base [1;2;3];;
- : int = 6
```

12. (6 pts) Write a base value `exists_false_base` and a step function `exists_false_step` such that `List.fold_left (exists_false_step f) exists_false_base lst` behaves the same as the `exists_false f lst` as described in problem 5.

```
# let exists_false_base = ...
val exists_false_base : bool = ...
# let exists_false_step = ...
val exists_false_step : ('a -> bool) -> bool -> 'a -> bool = <fun>
List.fold_left (exists_false_step (fun x -> x>2)) exists_false_base [3;4;2];;
# - : bool = true
```

For problems 13 through 15, you may use `List.map`, `List.fold_left` and `List.fold_right`, and must **not** use recursion (except that which exists in the above-mentioned functions). You may not use other functions from the `List` library, including `@`.

Warning: Every function you define must take at least one argument, either explicitly (`let f x = ...`) or implicitly (`let f = fun x -> ...`). Ignoring this rule can result in “unknown” types, which differ from “polymorphic” types. This is due to **value restriction**, which is beyond the scope of this course.

13. (5 pts) Write a function `case_map` that, when applied to functions `f`, `g`, and `h`, and list `lst`, returns a new list `lst'`. The list `lst'` is of the same length as `lst`. If `x` is the `i`th element of `lst`, then the `i`th element of `lst'` is equal to `(g x)` if `(f x)` is true, `(h x)` otherwise.

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

14. (8 pts) Write a function `expand_lst` that takes a function `f` and a list `lst` as arguments. The function `f` takes a value to and returns a list of new values of possibly different type. The function `expand_lst` applies `f` to each element in the list and appends the resulting lists together in the order in which the arguments originally came.

```
# let expand_lst f lst = ...;;
val expand_lst : ('a -> 'b list) -> 'a list -> 'b list = <fun>
# expand_lst (fun x -> if (x > 2) then ([1;1]) else ([2;2])) [1;2;3;4];;
- : int list = [2; 2; 2; 2; 1; 1; 1; 1]
```

15. (5 pts) Write a function `all_pos` that returns true if applied to a list, all of whose elements are strictly greater than 0.

```
# let all_pos l = ...
val all_pos : int list -> bool = <fun>
all_pos [1;2;0;4];;
# all_pos [1;2;4];;
- : bool = true
```

Stop: go back and make sure that you used **no** explicit recursion for problems 13 through 15. Make sure that all functions have at least one argument.

4.3 Extra Credit

16. (5 pts) Use the functions in questions 1 and 6 to write a simple selection sort program on a list. The function will first iterate through the list to find the smallest element and then place it in the head of the list, remove the element from the list and recursively call selection sort on the rest of the list.

```
# let rec selection_sort lst =
val selection_sort : int list -> int list = <fun>
# selection_sort [1;3;4;2];;
- : int list = [1; 2; 3; 4]
# selection_sort [-1;-2;4;10;34;-5;3];;
- : int list = [-5; -2; -1; 3; 4; 10; 34]
```