
MP 2 – Recursion and Higher-order Functions

CS 421 – Fall 2006

Revision 1.0

Assigned September 5, 2006

Due September 12, 2006 23:59

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:

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

3 Problems

For problems 1 through 4, you may **not** use library functions. You must use recursion instead.

The opposite applies to problems 5 through 10: 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.

3.1 Recursion

For problems 1 through 4, you may **not** use library functions.

1. Write `range inc a b` that takes an increment function `inc`, a lower bound `a`, and an upper bound `b`. `range` returns a list that is empty if `a` is greater than `b`, and otherwise has `a` in head position. Furthermore, if element `x` is at position `i` and `inc x` is less than or equal to `b`, then `inc x` is at position `i + 1`, for arbitrary position `i` in the returned list.

```
# let rec range inc a b = ...;;
val range : ('a -> 'a) -> 'a -> 'a -> 'a list = <fun>
# range ((+) 1) 1 10;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
# range ((^) "a") "a" "aaaaa";;
- : string list = ["a"; "aa"; "aaa"; "aaaa"; "aaaaa"]
```

2. Write `zip_app flst lst` that returns a list of size `min(size(flst), size(lst))`, in which the i_{th} element is equal to the i_{th} element of `flst` applied to the i_{th} element of `lst`, for arbitrary position i in the returned list.

```
# let rec zip_app flst lst = ...;;
val zip_app : ('a -> 'b) list -> 'a list -> 'b list = <fun>
# zip_app [((+) 2); (( * ) 2); ((-) 2)] [1;2;3;4];;
- : int list = [3; 4; -1]
# zip_app [(fun x -> "Hello, " ^ x)] ["George"; "Fred"];;
- : string list = ["Hello, George"]
```

3. Write `count_true f lst` that returns the number of elements `x` of `lst` for which `f x` is true. `count_true` must be **tail recursive** (you will want to define an inner recursive function that takes an extra argument).

```
# let count_true f lst = ...;;
val count_true : ('a -> bool) -> 'a list -> int = <fun>
# count_true (fun x -> x > 4) [1;2;3;4;5;6];;
- : int = 2
```

4. Write `rev_map f lst`, which is functionally equivalent to `List.reverse (List.map f lst)`, but does **not** use any library functions. `rev_map` must be **tail recursive** (you will want to define an inner recursive function that takes an extra argument). `rev_map` must perform the list reversal and mapping all at once, **not** in separate stages.

```
# let rev_map f lst = ...;;
val rev_map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# rev_map ((+) 1) [1;2;3;4;5];;
- : int list = [6; 5; 4; 3; 2]
```

Stop: go back and make sure you used no library functions for problems 1 through 4.

3.2 Using HOFs

For problems 5 through 10, you may use `List.map`, `List.fold_left` and `List.fold_right`, and may **not** use recursion (except that which exists in previously defined functions).

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.

5. Write `range_int a b` that produces a list of integers representing the range $[a..b]$, where each element is one plus the previous. **Hint:** use a previously defined function.

```
# let range_int a b = ...;
val range_int : int -> int -> int list = <fun>
# range_int 2 5;;
- : int list = [2; 3; 4; 5]
```

6. Write `reverse lst` that returns the reverse of `lst`.

```
# let reverse lst = ...;;
val reverse : 'a list -> 'a list = <fun>
# reverse [1;2;3];;
- : int list = [3; 2; 1]
```

7. Write `graph_arg f x` that takes a function `f` and an item `x`, such that `List.map (graph_arg f) dom` returns a list with the same size as `dom`, for which the i_{th} element is equal to (x, fx) where x is the i_{th} element in `dom`.

Note that the resulting graph is a list and not a set, so it is not **really** the graph of `f` (a list differs from a set in being finite, ordered, and allowing repeat elements.)

```
# let graph_arg f x = ...;;
val graph_arg : ('a -> 'b) -> 'a -> 'a * 'b = <fun>
# let fn = (( * ) 10);;
val fn : int -> int = <fun>
# let gph = List.map (graph_arg fn) (range_int 1 5);;
val gph : (int * int) list = [(1, 10); (2, 20); (3, 30); (4, 40); (5, 50)]
```

8. Write `inverse_arg x` such that `List.map inverse_arg gph` returns the graph `gph` with each pair (x, y) replaced with (y, x) .

Note: in the following example code, `gph` is as defined in the example code for the previous problem.

```
# let inverse_arg (x, y) = ...;;
val inverse_arg : 'a * 'b -> 'b * 'a = <fun>
# List.map inverse_arg gph;;
- : (int * int) list = [(10, 1); (20, 2); (30, 3); (40, 4); (50, 5)]
```

9. Write `catlist lst` that uses the concatenation operator `^` (caret) to concatenate a list `lst` of strings. **Note:** You **must** use `List.fold_left`.

```
# let catlist lst = ...;;
val catlist : string list -> string = <fun>
# catlist ["how"; " are"; " you?"];;
- : string = "how are you?"
# catlist [];;
- : string = ""
```

10. Write `compose lst` that takes a list of functions `lst` and returns the composition of these functions, such that $[f;g;h] \rightarrow fgh$. **Note:** You **must** use `List.fold_right`.

```
# let compose lst = ...;;
val compose : ('a -> 'a) list -> 'a -> 'a = <fun>
# compose [(+) 1; (( * ) 2); ((+) 1)] 7;;
- : int = 17
```

Stop: go back and make sure that you used **no** explicit recursion for problems 5 through 10. Make sure that all functions have at least one argument.

3.3 Extra Credit

11. Write `find_min lst` that takes an `int list` called `lst` and returns `(false, -1, [])` if `lst` is empty, and otherwise `(true, m, ms)` where `m` is the minimum element in `lst` and `ms` is equal to `lst` with the left-most occurrence of `m` dropped.

Then write `sort lst` that returns `lst` sorted from least to greatest.

```
# let rec find_min lst = ...
  let rec sort lst = ...;;
val find_min : int list -> bool * int * int list = <fun>
val sort : int list -> int list = <fun>
# find_min [3;5;2;6;2];;
- : bool * int * int list = (true, 2, [3; 5; 6; 2])
# find_min [];;
  : bool * int * int list = (false, -1, [])
# sort [5;3;6;2;4];;
- : int list = [2; 3; 4; 5; 6]
```