
MP 9 – Higher Order Functions

CS 421 – Spring 2008

Revision 1.0

Assigned Thursday, March 27, 2008

Due Tuesday, April 1, at 23:59pm

Extension 48 hours (20% penalty)

1 Included Helper Functions

Here are some functions used in this assignment; they are included in `mp9common.ml` so you do not have to write them yourself; you can just use them as desired. You may want to look at `mp9common.ml` to get a feel for what is available.

In some cases, they are used just for examples; in others, they are used in our solutions, so you might find them helpful in yours.

```
let incr x = x+1;;
let decr x = x-1;
let funmod f x y = fun z -> if x=z then y else f z;
let compose f g = fun x -> f (g x);
```

2 Problems

1. (5 pts) Define `split` such that `split f lis` returns a pair of lists (`lis1`, `lis2`) where `lis1` contains all the elements of `lis` for which `f` is true, and `lis2` contains all the others (in the same order as they appear in `lis`).

```
let rec split f lis = ...
val split : ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>

let posneg = split (fun x -> x < 0);;
val posneg : int list -> int list * int list = <fun>

posneg [1; 5; 0; -4; -3; 2];;
- : int list * int list = ([-4; -3], [1; 5; 0; 2])
```

2. (5 pts) Define `filter` such that `filter f lis` returns a list containing the elements of `lis` for which `f` is true. Then use `filter` to define `filtergt` such that `filtergt n lis` returns all the elements of `lis` that are less than or equal to `n`.

```
let rec filter f lis = ...
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>

let filtergt n = filter ...
val filtergt : 'a -> 'a list -> 'a list = <fun>

filtergt 10 [4; 25; 12; 8; 10; 7; 11];;
- : int list = [4; 8; 10; 7]
```

3. (5 pts) For this problem, you must use `fold_right` from the `List` module, and no explicit recursion. Define `length`, which gives the length of a list, by defining value `length_base` and function `length_recur`, and calling `fold_right length_recur lis length_base`. The function `length_recur` takes as arguments the head of the list and the result of the recursive call on the tail of the list, and gives the result for this list.

```
let length_base = ...;;
val length_base : int = ...
let length_recur h x = ...;;
val length_recur : 'a -> int -> int = <fun>
let length lis = fold_right length_recur lis length_base;;
val length : 'a list -> int = <fun>
length [1;2;3;4;5];
- : int = 5
```

4. (5 pts) For this problem, you must again use `fold_right` and no explicit recursion. Define `split_base` and function `split_recur` such that `fold_right (split_recur f) lis split_base` is the same function as `split f lis` defined in problem 1.

```
let split_base = ...;;
val split_base : 'a list * 'b list = ([], [])
let split_recur f h x = ...;;
val split_recur :
  ('a -> bool) -> 'a -> 'a list * 'a list -> 'a list * 'a list = <fun>
let split2 f lis = fold_right (split_recur f) lis split_base;;
val split2 : ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>
```

5. (5 pts) Using `map` from the `List` module, and no explicit recursion, define the function `case_map` such that `case_map f g h lis` returns a list with the following values: wherever `lis` has a value `x` for which `f` is true, it has `g x`, and everywhere else it has `h x`.

```
let case_map f g h = map ...;;
val case_map : ('a -> bool) -> ('a -> 'b) -> ('a -> 'b) -> 'a list -> 'b list = <fun>

case_map (fun x -> x>2) incr decr [0;1;2;3];;
- : int list = [-1; 0; 1; 4]
```

6. (10 pts) For this problem, you'll have to define several functions. We want to represent finite sets by pairs giving two values: the size of the set and the elements. The elements are represented by a function from values to booleans. Thus, in `mp9common.ml`, you will find:

```
type 'a fset = int * ('a -> bool);;
```

For example, the empty set, also defined in `mp9common.ml`, is represented by a pair containing zero and the function that always returns false:

```
let empty = (0, fun x -> false);;
```

You are to define the functions `member`, `size`, `add`, and `remove` (with their obvious meanings):

```
let member x s = ...
val member : 'a -> 'b * ('a -> 'c) -> 'c = <fun>

let size s = ...
val size : 'a * 'b -> 'a = <fun>

let add x s = ...
val add : 'a -> int * ('a -> bool) -> int * ('a -> bool) = <fun>

let remove x s = ...
val remove : 'a -> int * ('a -> bool) -> int * ('a -> bool) = <fun>

let set1 = add 1 (add 2 empty);;
member 1 set1;;
- : bool = true;;
member 3 set1;;
- : bool = false;;
size set1;;
- : int = 2;;
let set2 = remove 3 set1;;
size set2;;
- : int = 2;;
let set3 = remove 1 set1;;
size set3;;
- : int = 1;;
```

For `add` and `remove`, remember to check whether the element is already in the set. (You may find `funmod` to be useful here.)

7. (10 pts) Define exponentiation on Church numerals:

```
let exp m n = ...
val exp : 'a -> ('a -> 'b) -> 'b = <fun>

(* The following, reproduced from mp9common.ml, are Church numerals for 2 and 3 *)
let two = fun f -> compose f f;;
let three = fun f -> compose f (compose f f);;
(* show prints a Church numeral as a regular integer *)
let show nbar = print_int (nbar succ 0);;

show (exp two three);;
8;;
show (exp three two);;
9;
```

Note that `two`, `three`, and `show` are defined in `mp9common.ml`.

3 Extra Credit Problems

8. (4 pts) In this problem you will define functions for a slightly different set representation than the one in problem 6. This time, the sets may be infinite, so there is no size field. In fact, sets, as defined in mp9common.ml, are just represented by functions:

```
type 'a set = 'a -> bool;;
```

Define functions `member2`, `add2`, `remove2`, `compl`, `inter`, and `union`. `member2`, `add2`, and `remove2` are very much as above. `compl`, `inter`, and `union` give the complement of a set, and the intersection and union of two sets. (The complement of a finite set is infinite, so that is how we get infinite sets.)

```
let empty_inf = fun x -> false;;
val empty_inf : 'a -> bool = <fun>

let member2 x s = ...
val member2 : 'a -> ('a -> 'b) -> 'b = <fun>

let add2 x s = ...
val add2 : 'a -> ('a -> bool) -> 'a -> bool = <fun>

let remove2 x s = ...
val remove2 : 'a -> ('a -> bool) -> 'a -> bool = <fun>

let compl s = ...
val compl : ('a -> bool) -> 'a -> bool = <fun>

let inter s1 s2 = ...
val inter : ('a -> bool) -> ('a -> bool) -> 'a -> bool = <fun>

let union s1 s2 = ...
val union : ('a -> bool) -> ('a -> bool) -> 'a -> bool = <fun>

let set1 = add2 1 (add2 2 empty_inf);;
member2 1 set1;;
- : bool = true;;
let set2 = compl set1;;
member2 100 set2;;
- : bool = true;;
let set3 = inter set1 set2;;
member2 100 set3;;
- : bool = false;;
let set4 = union set1 set2;;
member2 50 set4;;
- : bool = true;;
```

9. (3 pts) Suppose `n` is a Church numeral. Define the function `nth` such that `nth n lis` returns the `n`(th) element of `lis`, indexing from zero. You may assume `lis` has at least `n+1` elements. You may not use recursion, but can only use the argument `n` to apply some function `n` times; it is your job to define that function; you may also do some more computation (but no recursion) after you have applied that function.

```
let nth n lis = ...
val nth : ('a list -> 'a list) -> 'b -> 'c list -> 'b -> 'c = <fun>

nth three [1;2;3;4;5];;
4;
```

10. (4 pts) Suppose n is a Church numeral. Define `sum n` that returns the sum of the numbers from 1 to n (as a regular integer, not a Church numeral), without using recursion.

```
let sum n = ...

sum three;;
- : int = 6;
```

Here is how to approach this problem: Suppose you had to solve it in an imperative language, but the only kind of loop you could use was one that executed its body exactly n times, where n was given at the start of the loop. You could solve the problem using two variables, one to hold the running sum, and one to hold the next integer. Thus, the loop would be:

```
loop n {
  s = s+b
  b = b+1
}
```

Thus, the two values (s, b) are transformed to two values $(s + b, b + 1)$ on each iteration; after n iterations, s has the desired value. As in the previous problem, you need to define a function that is to be applied n times; in this case, the function you want is just the function that takes pair (s, b) to pair $(s + b, b + 1)$. (You may also supply an initial pair, and you may do some fixed amount of work after doing the n iterations (but no recursion).