MP 8 – Higher-order Functions

CS 421 – Spring 2011

Revision 1.0

Assigned April 4, 2011
Due April 11, 2011 11:59pm
Extension 48 hours (20% penalty)
Total Points 50

1 Change Log

1.0 Initial Release.

1.1 Corrected specification of "setdiff" function in problem 6.

2 Objectives and Background

The purpose of this MP is to help the student master higher-order functions.

3 Collaboration

Collaboration is NOT allowed in this assignment.

4 Instructions

The problems below have sample executions that suggest how to write answers. You must use the same function names, but are free to choose different names for the arguments. You may also use `let rec` where we use `let`, and vice versa.

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. 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 the parameters must be the same as the parameters shown in sample execution.
- Students must comply with any special restrictions for each problem.

5 Included Helper Functions

Here are some functions used in this assignement; they are included in mp8common.ml so you do not have to write them yourself; you can use them as desired. You may want to look at mp8common.ml to get a feel for what is available.

```
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);;
```

6 Problems

1. (2 pts) Define `apply_n` such that `apply_n n f x` returns the result of applying `f` to `x` `n` times. That is, `apply_n 2 f x` should apply `f` to `x` twice, `apply_n 3 f x` should apply `f` three times, and so on. `apply_n 0 f x` should simply return `x`. You may assume that `n` is at least 0.

```
# let rec apply_n n f x =
val apply_n : int -> ('a -> 'a) -> 'a -> 'a = <fun>
# apply_n 3 (fun x -> x + 2) 6;;
- : int = 12
```

2. (3 pts) Using `map` from the List module, and no explicit recursion, define the function `negate` on lists of booleans such that `negate lis` returns a list that has `true` where `lis` has `false`, and vice versa.

```
# let negate = map ...;;
val negate : bool list -> bool list = <fun>
# negate [true; true; false; true; false];;
- : bool list = [false; false; true; false; true]
```

3. (6 pts) Define `every_other` such that `every_other f lis` returns the result of applying `f` to every other element of `lis`, starting with the first. In other words, it should return a list containing `f` of the first element of `lis`, `f` of the third element of `lis`, and so on.

```
# let rec every_other f lis =
val every_other : ('a -> 'b) -> 'a list -> 'b list = <fun>
# every_other (fun x -> x > 0) [1; 5; 0; -4; -3; 2];;
- : bool list = [true; false; false]
```

4. (6 pts) Define `pos_vals` such that `pos_vals f lis` returns a list containing every element `x` of `lis` for which `f x` is positive (i.e., greater than zero). Then use `pos_vals` to define `greater_than` such that `greater_than n lis` returns all the elements of `lis` that are strictly greater than `n`.

```
# let rec pos_vals f lis = ...
val pos_vals : ('a -> int) -> 'a list -> 'a list = <fun>
# let greater_than n = pos_vals ...
val greater_than : int -> int list -> int list = <fun>
```

(It is okay if the type of `greater_than` is:

```
val greater_than : 'a -> 'a list -> 'a list = <fun>)
```

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

5. (6 pts) For this problem, you must use `fold_right` from the List module, and no explicit recursion. Define a function `double`, which takes a list and returns the list containing two copies of each element. You should do this by defining value `double_base` and function `double_recur`, and calling `fold_right double_recur lis double_base`. The function `double_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 double_base = ...;;
val double_base : 'a list = ...
# let double_recur h x = ...;;
val double_recur : 'a -> 'a list -> 'a list = <fun>
# let double lis = fold_right double_recur lis double_base;;
val double : 'a list -> 'a list = <fun>
# double [1;2;3;4;5];;
- : int list = [1; 1; 2; 2; 3; 3; 4; 4; 5; 5]
```

6. (15 pts) Sets of integers are represented by functions in lecture 20 (starting on slide 36):

```
type intset = int -> bool;;
let member x s = s x;;
```

Define these additional functions:

1. `filter :  (int -> bool) -> intset -> intset.`
   Remove the elements of `s` that do not satisfy the predicate `f`.
2. `fromList :  int list -> intset.`
   Construct a set from the given list.
3. `setdiff :  intset -> intset -> intset.`
   Returns the "set difference" of the two arguments, i.e. those elements of the first set that are not in the second.

When you implement all the operations, every equality check below should evaluate to true. The code below is available in `examples.ml`.

```
let set = fromList [1;2;3;4;7];;
let fil = filter (fun x -> x>2) set1;;
let dif = setdiff set1 set2;;

member 3 set = true;;
member 5 set = false;;
member 1 fil = false;;
member 4 fil = true;;
member 2 dif = false;;
member 3 dif = true;;
```

7. (15 pts) Show the steps in simplifying the following expressions. Give your solutions on two separate pieces of paper.

1.
   ```
   let double = fun f -> fun x -> f (f x)
   in let incr = fun n -> n+1
      in (double incr) 5
   ```

2.
   ```
   let rec len = fun lis -> if lis=[] then 0 else 1 + len (tl lis)
   in len [1;2]
   ```