
MP 2 – Higher Order Functions

CS 421 – Summer 2008

Revision 1.1

Assigned June 12, 2008

Due June 20, 2008 23:59

Extension 48 hours. 20% penalty

1 Change Log

1.1 Fixed multifold example, clarified some questions.

1.0 Initial Release.

2 Objectives and Background

This MP is meant to give you practice writing and using higher order functions. The only functions you may use from the `List` module for this MP are `List.map`, `List.fold_right`, `List.fold_left`, `List.hd`, `List.tl`, and `List.rev`. You may, however, write helper functions.

Upon completion of this MP, you should have the following skills:

- Know how to use folding and mapping style higher order functions.
- Know how to apply HOFs in complex situations.
- Be able to write HOFs for user-defined types.

3 Getting Started

Start with the file `mp2.ml`, provided in the `.tgz` grader package. Stubs are provided for each function mentioned below. You can add helper functions, either outside of these function definitions or inside as local definitions. Note that the stub contains the following at the top:

```
open Mp2common;;
```

This will include certain data types used in this MP for you.

In order to `#use "mp2.ml"` in an interactive OCaml session, you must first download and compile `mp2common.ml` with

```
| ocamlc -c mp2common.ml
```

Make sure that `mp2.ml`, `mp2common.cmi`, and `mp2common.cmo` are in the same directory (you can also open `Mp2common` directly in a OCaml interactive session, once you've compiled it). Your solution file must have the line `open Mp2common;;` before your function definitions. Please don't remove it!

The problems are designed to be done sequentially, so you can always use the solution to an earlier problem in a later problem.

4 Problems

4.1 Writing HOFs

Write the following higher order functions. The first three are memory tests – you can of course look them up, but you should be able to write them from memory without doing so.

1. Write `map`, using only recursion (i.e., you cannot use `List.map` or a `fold`).

```
# let rec map f lst = ... ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map (fun n -> n * 2) [1;2;3;4;5]
- : int list = [2; 4; 6; 8; 10]
```

2. Write `fold_right`, using only recursion (i.e., you cannot use `List.fold_right`).

```
# let rec fold_right f lst i = ... ;;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# fold_right (+) [1;2;3;4;5] 0 ;;
- : int = 15
```

3. Write `fold_left`, using only recursion (i.e., you cannot use `List.fold_left`).

```
# let rec fold_left f i lst = ... ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_left (-) 0 [1;2;3;4;5] ;;
- : int = -15
```

4. Write `exists p lst`, which checks that at least one element in a list, `lst`, satisfies a given predicate `p`. (We call this argument `p`, because *predicates* are functions that return `true` or `false`.)

```
# let rec exists p lst = ... ;;
val exists : ('a -> bool) -> 'a list -> bool = <fun>
# exists (fun x -> x <= 8) [1;1;3;5;8];;
- : bool = true
# exists (fun x -> x <= 4) [5;7;15;5;8];;
- : bool = false
```

5. Write `exists_fold p lst` to yield the same results as `exists`, but using `fold` (i.e., without making `exists_fold` explicitly recursive).

```
# let exists_fold p lst = ... ;;
val exists_fold : ('a -> bool) -> 'a list -> bool = <fun>
# exists_fold (fun x -> x <= 8) [1;1;3;5;8];;
- : bool = true
# exists_fold (fun x -> x <= 4) [5;7;15;5;8];;
- : bool = false
```

6. Write `app_list fl x`, which applies each function in a list of functions, `fl`, to a given argument `x`. The result list should have the same length as `fl`.

```
# let rec app_list fl x = ... ;;
val app_list : ('a -> 'b) list -> 'a -> 'b list = <fun>
# app_list [(fun x -> (x * 3) + 1); ((+) 2)] 17;;
- : int list = [52; 19]
```

4.2 Using HOFs

You may not use recursion for this part of the MP, except of course that which exists in the higher order functions from part 1, and the previously mentioned functions in the `List` module. In other words, none of the following should be defined using `let rec!`

7. Write a function `bin2Int lst` which takes a list of numbers representing bits and generates the appropriate integer value. Assume that all numbers in the list are either 0 or 1 and that the integer value represented by the bit string is positive (we don't need to determine if the binary value represents a negative number). The bits go from the most significant at the head of the list to the least significant at the end. You do not need to worry about numeric overflow (i.e. we don't expect you to handle numbers larger than what fits in the OCaml int type).

```
# let bin2Int lst = ...
val bin2Int : int list -> int = <fun>
# bin2Int [];;
- : int = 0
# bin2Int [0];;
- : int = 0
# bin2Int [1;0;0;1;1];;
- : int = 19
# bin2Int [1;1;0;1;1;0;1;1;1;1;1;1;1;1];;
- : int = 14079
```

8. Write a function `multimap flst xlst` which takes a list of functions `flst` and maps them, one at a time, from right to left, to the list `xlst`.

```
# let multimap flst xlst = ...
val multimap : ('a -> 'a) list -> 'a list -> 'a list = <fun>
# multimap [(+) 1; ( *) 2; (+) 20] [2;3;4;5];;
- : int list = [45; 47; 49; 51]
```

9. Write a function `multifold filst xlst` which takes a list of pairs `filst`, with each pair containing a function and an identity, and `fold_right` each function, with the identity, over the list `xlst`, returning a list made up of the result of each `fold_right`. The resulting list should have the same length as `filst`.

```
# let multifold filst xlst = ... ;;
val multifold : (('a -> 'b -> 'b) * 'b) list -> 'a list -> 'b list = <fun>
# multifold [ ( (+), 0 ) ; ( ( * ), 1 ) ; ( (fun x y -> x * y), 1 ) ] [ 1;2;3;4;5 ] ;;
- : int list = [15; 120; 120]
```

10. Write a function `rsum lst` which computes the running sum of the list `lst`. A running sum of a list is defined as:

$$[x_1; x_2; x_3; \dots; x_n] = [0; x_1; x_1 + x_2; x_1 + x_2 + x_3; \dots; x_1 + x_2 + x_3 + \dots + x_n]$$

You may use `List.rev` for this problem if you wish.

```
# let rsum lst = ...
val rsum : int list -> int list = <fun>
# rsum [1;2;3;4;5];;
- : int list = [0; 1; 3; 6; 10; 15]
```

4.3 Writing HOFs for Tree Data Structures

Recall from the lecture that we can write functions like `map` and `fold` can work over other datatypes, such as binary trees. `Mp2common` **already defines** this type for you:

```
type 'a btree = Node of 'a btree * 'a btree | Leaf of 'a | Empty
```

11. Write a recursive function `mapbtree f t` that maps a function `f` over a tree `t`.

```
# let rec mapbtree f t = ... ;;
val mapbtree : ('a -> 'b) -> 'a btree -> 'b btree = <fun>

# let st1 = Node(Leaf 10, Leaf 20);;
# let st2 = Node(Leaf 40, Leaf 50);;
# let st3 = Node(Leaf 30, st2);;

# let tree1 = Node(st1, st3);;
# let tree2 = Node(st3, st1);;

# tree1;;
- : int btree =
Node (Node (Leaf 10, Leaf 20), Node (Leaf 30, Node (Leaf 40, Leaf 50)))
# mapbtree ( (+ ) 1 ) tree1;;
- : int btree =
Node (Node (Leaf 11, Leaf 21), Node (Leaf 31, Node (Leaf 41, Leaf 51)))
```

12. Write a recursive function `foldbtree f g t z` that folds over a tree `t`. Given a tree of the form `Empty`, `foldbtree` returns the base case value `z`. Given a tree of the form `Leaf(x)`, `foldbtree` returns the result of `g` applied to `x`. Given a tree of the form `Node(left, right)`, `foldbtree` first computes the value of `foldbtree` on the left and right subtrees and applies these two results to `f`.

```
# let rec foldbtree f g t z = ... ;;
val foldbtree : ('a -> 'a -> 'a) -> ('b -> 'a) -> 'b btree -> 'a -> 'a = <fun>
# foldbtree (+) (fun x -> x) tree1 0;;
- : int = 150
```

4.4 Using HOFs for Tree Data Structures

For this part, you may not use recursion, except that which is contained in the higher order functions you wrote above. All the solutions to these (except `isSearch`) are one line long. (Yours can be longer than that.)

13. Write a function `depth t` which returns the depth of a tree. Let the depth of a leaf be defined as one. Let the depth of a node be the maximum of the depths of the two subtrees, plus one. The depth of an empty tree is zero.

```
# let depth t = ... ;;
val depth : 'a btree -> int = <fun>
# depth (Leaf 4);;
- : int = 1
# depth (tree1);;
- : int = 4
#
```

14. Write a function `isBinary t` that will return `true` if the tree is a full binary tree. A full binary tree is one where every element is either a node or a leaf.

```
# let isBinary t = ... ;;
val isBinary : 'a btree -> bool = <fun>
# isBinary tree1;;
- : bool = true
# isBinary (Node (tree1, Empty));;
- : bool = false
```

15. Write a function `flip t` that reverses a tree.

```
# let flip t = ... ;;
val flip : 'a btree -> 'a btree = <fun>
# flip tree1;;
- : int btree =
Node (Node (Node (Leaf 50, Leaf 40), Leaf 30), Node (Leaf 20, Leaf 10))
```

16. Write a function `isSearch` that will return `true` if the tree is a search tree. A tree is a search tree if it is a leaf, it is empty, or if it is a node in which every element of the left subtree is less than every element of the right subtree (assume no duplicate elements), and if both subtrees are themselves search trees. Hint: make your own `option`-like type with 3 constructors, to represent an empty tree, non-search trees, and a search tree over a range. The solution to this one is **not** one line long.

```
# let isSearch t = ... ;;
val isSearch : int btree -> bool = <fun>
# isSearch tree1;;
- : bool = true
# isSearch tree2;;
- : bool = false
```

5 Extra Credit

Recall from lecture the datatype 'a tree:

```
type 'a tree = TreeLeaf of 'a
             | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree
               | More of ('a tree * 'a treeList);;
```

This type may be found in Mp2common.

17. Write a recursive function `mapTree f t` that applies `f` over a tree `t`, applying `f` to the leaves of the tree. You may need to add more than one function to handle the mutually recursive types.

```
val mapTree : ('a -> 'b) -> 'a tree -> 'b tree = <fun>
# let tree =
  TreeNode
    (More (TreeLeaf 5,
           (More (TreeNode
                  (More (TreeLeaf 3,
                        Last (TreeLeaf 2))),
                    Last (TreeLeaf 7))))));
val tree : int tree =
  TreeNode
    (More
     (TreeLeaf 5,
      More
       (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))), Last (TreeLeaf 7))))
# mapTree ((+)3) tree;;
- : int tree =
  TreeNode
    (More
     (TreeLeaf 8,
      More
       (TreeNode (More (TreeLeaf 6, Last (TreeLeaf 5))), Last (TreeLeaf 10))))
```

18. Write a function `foldTree leafop nodeop lastop moreop t` that folds over a tree `t`. The function `leafop` is used to return the result when the tree is a `TreeLeaf`; `nodeop` is used to return the result when the tree is a `TreeNode`. The functions `lastop` and `moreop` are used for the `Last` and `More` cases for the `treeLists` within the `TreeNode` case.

```
let rec foldTree leafop nodeop lastop moreop t = ... ;;
val foldTree :
  ('a -> 'b) -> ('c -> 'b) -> ('b -> 'c) -> ('b -> 'c -> 'c) -> 'a tree -> 'b =
  <fun>
```

19. Using the function `foldTree`, without using recursion (except as is present in `foldTree`), write a function `new_fringe` that has the same type and computes the same results as the function `fringe` given in class.

```
val new_fringe : 'a tree -> 'a list = <fun>
# fringe tree;;
- : int list = [5; 3; 2; 7]
```

Recall from lecture the datatype 'a labeled_tree:

```
type 'a labeled_tree = LTreeNode of ('a * 'a labeled_tree list);;
```

This type may be found in Mp2common.

20. Write a recursive function `foldLabTree nodeFun nilCase consFun t` folds over a labeled tree `t`.

Recall that, since this is a nested recursive definition, you should use a mutually recursive solution.

```
let rec foldLabTree nodeFun nilCase consFun t= ... ;;
val foldLabTree :
  ('a -> 'b -> 'c) ->
  'b -> ('c -> 'b -> 'b) -> 'a Mp2common.labeled_tree -> 'c = <fun>
val foldLabList :
  ('a -> 'b -> 'c) ->
  'b -> ('c -> 'b -> 'b) -> 'a Mp2common.labeled_tree list -> 'b = <fun>
```

21. Using the function `foldLabTree`, but no other recursion, write the function `mapLabTree f t` that maps a function `f` over every value in every node for the tree `t`.

```
# let ltree =
  LTreeNode (5,
    [LTreeNode (3, []);
     LTreeNode (2, [LTreeNode (1, []);
                    LTreeNode (7, [])]);
     LTreeNode (5, [])]);;
val ltree : int labeled_tree =
  LTreeNode
  (5,
    [LTreeNode (3, []); LTreeNode (2, [LTreeNode (1, []); LTreeNode (7, [])]);
     LTreeNode (5, [])])
# mapLabTree (fun x -> (x, "a")) ltree;;
val ltree : (int * string) labeled_tree =
  LTreeNode
  ((5, "a"),
   [LTreeNode ((3, "a"), []);
    LTreeNode ((2, "a"), [LTreeNode ((1, "a"), []); LTreeNode ((7, "a"), [])]);
    LTreeNode ((5, "a"), [])])
```

6 Handing In

Use the `handin` program on `dclnx?.ews.uiuc.edu` (? being some machine number, which should not matter) as you did for MP1. See the instructions in the FAQ section of the webpage.