

---

# MP 3 – Patterns of Recursion, Higher-order Functions

CS 421 – Fall 2009

Revision 1.1

**Assigned** September 8, 2009

**Due** September 15, 2009 23:59

**Extension** 48 hours (20% penalty)

---

## 1 Change Log

1.1 Corrected error in type of `all_from_tok` in written description of problem 16.

1.0 Initial Release.

## 2 Objectives and Background

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

1. forward recursion and tail recursion
2. higher-order functions
3. continuation passing style

## 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 functions. 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 or continuation passing style, while others ask students to use higher-order functions in place of recursion.

## 4 Problems

- For problems 1 through 9 and 12 through 16, you **may not** use library functions or @.
- In problems 10 and 11, you **may not** use recursion, and instead **must** use the library functions specified.
- In problems 1 through 3 you **must** use forward recursion.
- In problems 4 through 6 you **must** use tail recursion.
- Problems 12 through 16 **must** be in continuation passing style.

**Note:** All library functions are off limits for all problems on this assignment, except those that are specifically required (in problems 10 and 11.) For purposes of this assignment @ is treated as a library function and is not to be used..

### 4.1 Patterns of Recursion

For problems 1 through 6, you may **not** use library functions.

1. (3 pts) Write a function `remove_all : int list -> int -> int list` such that `remove_all list m` returns a list in the same order as the input list, but with all the numbers equal to `m` removed. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec remove_all list m = ... ;;
val remove_all : 'a list -> 'a -> 'a list = <fun>
# remove_all [2; 4; 3; 7; 2; 8; 2] 2;;
- : int list = [4; 3; 7; 8]
```

2. (5 pts) Write a function `all_from_to : int -> int -> (int -> bool) -> int` such that `all_from_to m n p` tells the number of integers greater than or equal to `m` and also less than or equal to `n` which satisfy a given predicate `p : int -> bool`. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec all_from_to m n p = ... ;;
val all_from_to : int -> int -> (int -> bool) -> int = <fun>
# all_from_to (-5) 7 ((<) 0);;
- : int = 7
# all_from_to 3 7 (fun x -> x mod 2 = 0);;
- : int = 2
```

3. (5 pts) Write a function `separate : ('a -> bool) -> 'a list -> int * int` such that `separate p l` returns a pair of integers, the first indicates the number of elements of `l` for which `p` returns true, and the second indicates the number of elements for which `p` returns false. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec separate p l = ... ;;
val separate : ('a -> bool) -> 'a list -> int * int = <fun>
# separate (fun x -> x mod 2 = 0) [-3; 5; 2; -6];;
- : int * int = (2, 2)
```

4. (5 pts) Write a function `all_even : int list -> bool` that returns whether every element in the input list is even. The function is required to use (only) tail recursion (no other form of recursion). You may use `mod` for testing whether an integer is even. You may not use any library functions.

```
# let rec all_even list = ... ;;
val all_even : int list -> bool = <fun>
# all_even [4; 2; 12; 5; 6];;
- : bool = false
```

5. (6 pts) Write a function `sum_square : int -> int -> int` such that `sum_square m n` calculates the sum of the squares of the elements strictly greater than `m` and strictly less than `n` if there are any, and 0 otherwise. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec sum_square m n = ... ;;
val sum_square : int -> int -> int = <fun>
# sum_square 3 9;;
- : int = 190
```

6. (8 pts) Write a function `concat : string -> string list -> string` such that `concat s l` creates a string consisting of the strings in the list `l` concatenated together, with a single space inserted between consecutive elements. Also all strings equal to `s` should be excluded. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec concat s list = ... ;;
val concat : string -> string list -> string = <fun>
# concat "hi" ["How"; "are"; "hi"; "you?"];;
- : string = "How are you?"
```

**Stop:** go back and make sure you used no library functions for problems 1 through 6.

## 4.2 Higher-Order Functions

For problems 7 through 9, you will be supplying arguments to the higher-order functions `List.fold_right` and `List.fold_left`. You should not need to use explicit recursion for any of 7 through 11.

7. (7 pts) Write a value `remove_all_base` and function `remove_all_rec : int -> int -> int list -> int list` such that `(fun list -> List.fold_right (remove_all_rec m) list remove_all_base)` computes the same results as `remove_all` of Problem 1. There should be no use of recursion or library functions in defining `remove_all_rec`.

```
# let remove_all_base = ... ;;
val remove_all_base : ...
# let remove_all_rec m n r = ... ;;
val remove_all_rec : 'a -> 'a -> 'a list -> 'a list = <fun>
# (fun list -> List.fold_right (remove_all_rec 2) list remove_all_base)
  [2; 4; 3; 7; 2; 8; 2];;
- : int list = [4; 3; 7; 8]
```

8. (7 pts) Write a value `separate_base` and function `separate_rec` : `('a -> bool) -> 'a -> int * int -> int * int` such that `(fun p -> fun list -> List.fold_right (separate_rec p) list separate_base)` computes the same results as `separate` of Problem 3. There should be no use of recursion or library functions in defining `separate_rec`.

```
# let separate_base = ...
val separate_base : ...
# let separate_rec p x (tl, fl) = ...
val separate_rec : ('a -> bool) -> 'a -> int * int -> int * int =
  <fun>
# (fun p -> fun list -> List.fold_right (separate_rec p) list separate_base)
  (fun x -> x mod 2 = 0)
  [-3; 5; 2; -6];;
- : int * int = (2, 2)
```

9. (7 pts) Write a value `all_even_base` and function `all_even_rec` : `bool -> int -> bool` such that `List.fold_left all_even_rec all_even_base` computes the same results as `all_even` of Problem 4. You may use `mod` for testing whether an integer is even. There should be no use of recursion or other library functions in defining `all_even_rec`.

```
# let all_even_base = ...
# let all_even_rec r x = ... ;;
val all_even_rec : bool -> int -> bool = <fun>
# List.fold_left all_even_rec all_even_base [4; 2; 12; 5; 6];;
- : bool = false
```

10. (7 pts) Write a function `concat2` : `string -> string list -> string` that computes the same results as `concat` of Problem 6. The definition of `concat2` may use `List.fold_left` : `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` but no direct use of recursion, and no other library functions.

```
# let concat2 i list = ...;;
val concat2 : string -> string list -> string = <fun>
# concat2 "hi" ["How"; "are"; "hi"; "you?"];;
- : string = "How are you?"
```

11. (8 pts) Write a function `app_all` : `('a -> 'b) list -> 'a list -> 'b list list` that takes a list of functions, and a list of arguments for those functions, and returns the list of list of results from consecutively applying the functions to all arguments, in the order in which the functions occur in the list and in the order in which the arguments occur in the list. Each list in the result list corresponds to a list of applications of each function to the given arguments. The definition of `app_all` may use the library function `List.map` : `('a -> 'b) -> 'a list -> 'b list` but no direct use of recursion, and no other library functions.

```
# let app_all fs list = ... ;;
val app_all : ('a -> 'b) list -> 'a list -> 'b list list = <fun>
# app_all [(fun x -> x > 0); (fun y -> y mod 2 = 0);
  (fun x -> x * x = x)] [1; 3; 6];;
- : bool list list =
[[true; true; true]; [false; false; true]; [true; false; false]]
```

### 4.3 Continuation Passing Style

These exercises are designed to give you a feel for continuation passing style. A function that is written in continuation passing style does not return once it has finished its computation. Instead, it calls another function (the continuation) with the result. Here is a small example:

```
# let report x =
  print_string "Result: ";
  print_int x;
  print_newline();;
val report : int -> unit = <fun>

# let inck i k = k (i+1)
val inck : int -> (int -> 'a) -> 'a = <fun>
```

The `inck` function takes an integer and a continuation. After adding 1 to the integer, it passes the result to its continuation.

```
# inck 3 report;;
Result: 4
- : unit = ()
# inck 3 inck report;;
Result: 5
- : unit = ()
```

In line 1, `inck` increments 3 to be 4, and then passes the 4 to `report`. In line 4, the first `inck` adds 1 to 3, and passes the resulting 4 to the second `inck`, which then adds 1 to 4, and passes the resulting 5 to `report`.

#### Simple Continuations

12. (8 pts) Write the functions `subk`, `catk`, `doublek`, `plusk`, `multk`, and `is_posk` in CPS. `subk` subtracts one integer from another; `catk` concatenates two strings; `doublek` concatenates a string with itself, `plusk` adds two floats, `multk` multiplies two floats, and `is_posk` determines if an integer is greater than 0.

```
# let subk n m k = ...;;
val subk : int -> int -> (int -> 'a) -> 'a = <fun>
# let catk a b k = ...;;
val catk : string -> string -> (string -> 'a) -> 'a = <fun>
# let doublek a k = ...;;
val doublek : string -> (string -> 'a) -> 'a = <fun>
# let plusk x y k = ...;;
val plusk : float -> float -> (float -> 'a) -> 'a = <fun>
# let multk x y k = ...;;
val multk : float -> float -> (float -> 'a) -> 'a = <fun>
# let is_posk n k = ...;;
val is_posk : int -> (bool -> 'a) -> 'a = <fun>
# subk 10 5 report;;
Result: 5
- : unit = ()
# catk "hi " "there" (fun x -> x);;
- : string = "hi there"
# doublek "pom " (fun x -> x);;
- : string = "pom pom "
# plusk 3.0 4.0
```

```

    (fun x -> multk x x
      (fun y -> (print_string "Result: "; print_float y; print_newline())));;
    Result: 49.
- : unit = ()
# is_posk 7 (fun b -> (report (if b then 3 else 4)));;
Result: 3
- : unit = ()

```

**Nesting Continuations** One common technique used in CPS is that of nesting continuations. For example, consider the following code:

```

# let add3k a b c k =
  addk a b (fun ab -> addk ab c k);;
val add3k : int -> int -> int -> (int -> 'a) -> 'a = <fun>
# add3k 1 2 3 report;;
Result: 6
- : unit = ()

```

We needed to add three numbers together, but `addk` itself only adds two numbers. On line 2, we give the first call to `addk` a function that saves the sum of `a` and `b` in the variable `ab`. Then this function adds `ab` to `c` and passes its result to the continuation `k`.

13. (5 pts) Using `multk` and `plusk` as helper functions, write a function `abcdk`, which takes four float arguments `a b c d` and “returns”  $((a + b) * c) + d$ . You may not use the normal `*` operator or `+` operators; you must instead use `multk` and `plusk`.

```

# let abcdk a b c d k = ...
val abcdk : float -> float -> float -> float -> (float -> 'a) -> 'a = <fun>
# abcdk 2.0 3.0 4.0 5.0 (fun y -> report (int_of_float y));;
Result: 25
- : unit = ()

```

**Recursion and Continuation** How do we write recursive programs in CPS? Consider the following recursive function:

```

# let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120

```

To put the function into CPS, we must make `factorial` take an additional argument, a continuation, to which the result of the factorial function should be passed. When the recursive call is made to `factorial`, instead of it returning a result to build the next higher factorial, it needs to take a continuation for building that next value from its result. Thus the code becomes:

```

# let rec factorialk n k =
  if n = 0 then k 1 else factorialk (n - 1) (fun m -> k (n * m));;
val factorialk : int -> (int -> 'a) -> 'a = <fun>
# factorialk 5 report;;
120
- : unit = ()

```

To make a recursive call, we must build an intermediate continuation to:

- take the recursive value:  $m$
- build it to the final result:  $n * m$

And pass it to the final continuation:  $k (n * m)$ . Notice that this is an extension of the "nested continuation" method.

In problems 14 through 16 all functions are to be written in continuation passing style. Only the application of primitive operations (*e.g.*  $+$ ,  $-$ ,  $*$ ,  $>$ ,  $=$ ,  $\text{mod}$ ,  $\wedge$ ) do not need to take a continuation as an argument.

14. (8 pts) Write function `remove_allk : int list -> int -> (int list -> 'a) -> 'a` such that `remove_allk` is the continuation passing style version of the code you gave for `remove_all` of Problem 1. You should have that `remove_allk list m (fun x -> x)` computes the same results as `remove_all list`.

```
# let rec remove_allk list m k = ...;;
val remove_allk : 'a list -> 'a -> ('a list -> 'b) -> 'b = <fun>
# remove_allk [2; 4; 3; 7; 2; 8; 2] 2 (fun x -> x);;
- : int list = [4; 3; 7; 8]
```

15. (8 pts) Write function `all_evenk : int list -> (bool -> 'a) -> 'a` such that `all_evenk` is the continuation passing style version of the code you gave for `all_even` of Problem 4. You should have that `all_evenk list (fun x -> x)` computes the same results as `all_even list`.

```
# let rec all_evenk list k = ... ;;
val all_evenk : int list -> (bool -> 'a) -> 'a = <fun>
# all_evenk [4; 2; 12; 5; 6] (fun x -> x);;
- : bool = false
```

#### 4.4 Extra Credit

16. (9 pts) Write a function `all_from_tok : int -> int -> (int -> (bool -> 'a) -> 'a) -> (int -> 'a) -> 'a` such that `all_from_tok` is the continuation passing style version of the code you gave for `all_from_to` of Problem 2. You should have that `all_from_tok m n (fun i -> fun k -> k(p i)) (fun x -> x)` computes the same results as `all_from_to m n p`.

```
# let rec all_from_tok m n p k = ...;;
val all_from_tok :
  int -> int -> (int -> (bool -> 'a) -> 'a) -> (int -> 'a) -> 'a = <fun>
# all_from_tok 3 7 (fun i -> fun k -> k (i mod 2 = 0)) (fun x -> x);;
- : int = 2
```