MP 3 – Patterns of Recursion, Higher-order Functions

CS 421 – Fall 2011
Revision 1.1

Assigned September 6, 2011
Due September 13, 2011 23:59
Extension 48 hours (20% penalty)

1 Change Log

1.1 Changed to clarify when `let` may be changed to `let rec` and *vice versa*.

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 some of the functions that are allowed to use recursion. You are not required to start your code with `let rec`. Similarly, if you are not prohibited from using explicit recursion in a given problem, you may change any function defintion from starting with just `let` to starting 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

- In problems 1 through 3 you **must** use forward recursion.
- In problems 4 through 6 you **must** use tail recursion.
- In problem 11, you **may not** use recursion, and instead **must** use `List.map`.
- 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 problem 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 `replace_if : ('a -> bool) -> 'a -> 'a list -> 'a list` such that `replace_if p v lst` replaces every element in `lst` for which `p` returns true with `v`. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

   ```
   # let rec replace_if p v lst = ... ;;
   val replace_if : ('a -> bool) -> 'a -> 'a list -> 'a list = <fun>
   # replace_if (fun x -> x > 3) 62 [1;2;3;4;5];;
   - : int list = [1; 2; 3; 62; 62]
   ```

2. (5 pts) Write a function `partial_sums : float list -> float list` such that `partial_sums l`, where `l` $= [x_0; \ldots; x_n]$ returns a list whose i^{th} element is the sum of the partial list from i to n ($\Sigma_{j=i}^{n}$), for each $i = 0, \ldots, n$.

   ```
   # let rec partial_sums l = ... ;;
   val partial_sums : float list -> float list = <fun>
   # partial_sums [4.0; 2.2; 7.5];;
   - : float list = [13.7; 9.7; 7.5]
   ```

3. (5 pts)

   Write a function `not_same_next : 'a list -> 'a list` such that `not_same_next lst` returns a list with the same elements in `lst`, in the same order except reduced such that no two adjacent elements have the same value. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

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

4. (5 pts) Write a function `exists_pred : ('a -> bool) -> 'a list -> bool` such that `exists_pred p list` returns whether there exists an element in the input list that is true for the given predicate. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec exists_pred p list = ... ;;
val exists_pred : ('a -> bool) -> 'a list -> bool = <fun>
# exists_pred (fun x -> x mod 2 = 0) [1;2;3;4];;
- : bool = true
```

5. (6 pts) Write a function `count_if : ('a -> bool) -> 'a list -> int` such that `count_if p lst` returns the number of elements in `lst` for which the given predicate `p` is true. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

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

6. (8 pts)

   Write a function `min_element : 'a list -> int` such that `min_element lst` returns the index of the smallest element in `lst`. When the smallest value occurs multiple times, return the smallest index. Indices will be counted from the head of the list, with index 0, to the tail of the list, with last index one less than the length of the list. If an empty list is given, return -1. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let min_element lst = ... ;;
val min_element : 'a list -> int = <fun>
# min_element [1;1;2;3;4];;
- : int = 0
```

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 10, 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 `replace_if_base` and function `replace_if_rec : ('a -> bool) -> 'a -> 'a -> 'a list -> 'a list` such that `(fun p -> fun v -> fun lst -> List.fold_right (replace_if_rec p v) lst replace_if_base)` computes the same results as `replace_if` of Problem 1. There should be no use of recursion or library functions in defining `replace_if_rec`.

```
# let replace_if_base = ... ;;
val replace_if_base : ...
# let replace_if_rec p v x r =  ... ;;
val replace_if_rec : ('a -> bool) -> 'a -> 'a -> 'a list -> 'a list = <fun>
# let replace_if' p v lst = List.fold_right
                            (replace_if_rec p v) lst replace_if_base ;;
val replace_if' : ('a -> bool) -> 'a -> 'a list -> 'a list = <fun>
# replace_if' (fun x -> x > 3) 62 [1;2;3;4;5];;
- : int list = [1; 2; 3; 62; 62]
```

8. (7 pts) Write a value `not_same_next_base` and function `not_same_next_rec : 'a -> 'a list -> 'a list` such that `(fun lst -> List.fold_right not_same_next_rec lst not_same_next_base)` computes the same results as `not_same_next` of Problem 3. There should be no use of recursion or library functions in defining `not_same_next_rec`.

```
# let not_same_next_base = ... ;;
val not_same_next_base : ...
# let not_same_next_rec x r = ... ;;
val not_same_next_rec : 'a -> 'a list -> 'a list = <fun>
# let not_same_next' lst = List.fold_right not_same_next_rec lst not_same_next_base ;
val not_same_next' : 'a list -> 'a list = <fun>
# not_same_next' [1;2;2;3;2;1];;
- : int list = [1; 2; 3; 2; 1]
```

9. (7 pts) Write a value `exists_pred_base` and function `exists_pred_rec : ('a -> bool) -> bool -> 'a -> bool` such that `(fun p -> fun list -> List.fold_left (exists_pred_rec p) exists_pred_base list)` returns the same results as `exists_pred` of Problem 4. There should be no use of recursion or other library functions in defining `exists_pred_rec`.

```
# let exists_pred_base = ...
val exists_pred_base : ...
# let exists_pred_rec p r x = ... ;;
val exists_pred_rec : ('a -> bool) -> bool -> 'a -> bool = <fun>
# (fun p -> fun list -> List.fold_left (exists_pred_rec p) exists_pred_base list)
    (fun x -> x mod 2 = 0)
    [1;2;3;4];;
- : bool = true
```

10. (7 pts) Write a base value `count_if_base` and function `count_if_rec : ('a -> bool) -> int -> 'a -> int` such that `(fun p -> fun lst -> List.fold_left (count_if_rec p) count_if_base lst)` computes the same results as `count_if` of Problem 5. There should be no use of recursion or library functions in defining `count_if_rec`.

```
# let count_if_base = ... ;;
val count_if_base : ...
# let count_if_rec p r x = ... ;;
val count_if_rec : ('a -> bool) -> int -> 'a -> int = <fun>
# let count_if' p lst = List.fold_left (count_if_rec p) count_if_base lst ;;
val count_if' : ('a -> bool) -> 'a list -> int = <fun>
# count_if' (fun x -> x > 3) [1;2;3;4;5];;
- : int = 2
```

11. (8 pts) Write a function `cross_prod : 'a list -> 'b list -> ('a * 'b) list list = <fun>` that takes two lists and forms a list of lists where each inner list of the result of pairing a fixed element from the first list with each element form the second, in order, and the inner lists appear in the same as the corresponding elements in the first list. The definition `cross_prod` 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 cross_prod xs ys = ...;;
val cross_prod : 'a list -> 'b list -> ('a * 'b) list list = <fun>
```

```
# cross_prod [1;2] ["hello"; "world"];;
- : (int * string) list list =
[[(1, "hello"); (1, "world")]; [(2, "hello"); (2, "world")]]
```

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`, `consk`, `addk`, `timesk`, `plusk`, `multk`, `lessk`, and `eqk` in CPS. The function `subk` subtracts the second integer from the first; `consk` makes a new list by consing an element onto the front of an existing list, `addk` adds two integers, `timesk` multiplies two integers, `plusk` adds two floats, `multk` multiplies two floats, `lessk` determines if one value is strictly less than another, and `eqk` determines whether one value is equal to another.

```
# let subk n m k = ...;;
val subk : int -> int -> (int -> 'a) -> 'a = <fun>
# let consk h t k = ...;;
val consk : 'a -> 'a list -> ('a list -> 'b) -> 'b = <fun>
# let addk x y k = ...;;
val addk : int -> int -> (int -> 'a) -> 'a = <fun>
# let timesk a b k = ...;;
val timesk : int -> int -> (int -> '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 lessk n m k = ...;;
```

```
val lessk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
# let eqk x y k = ..;;
val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>

# subk 10 5 report;;
Result: 5
- : unit = ()
# addk 3 4
  (fun x -> timesk x x
   (fun y -> (print_string "Result: ";print_int y; print_newline())));;
Result: 49.
- : unit = ()
# lessk 2 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.

Also, note that when using continuation passing style you are specifying the order of evaluations. For example, the following code will add c and b before adding that result to a.

```
# let radd3k a b c k = addk c b (fun cb -> addk cb a k);;
val radd3k : int -> int -> int -> (int -> 'a) -> 'a = <fun>
# radd3k 1 2 3 report;;
Result: 6
- : unit = ()
```

In the following problems, keep in mind the order of operations and evaluation when using continuation style.

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" $(a + b) * (c + d)$. Perform the operations such that the addition on c and d is done first, then the addition on a and b, and then finally the multiplication. You may not use the normal `*.` 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: 45
- : 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
```

We can rewrite this making each step of computation explicit as follows:

```
# let rec factoriale n =
      let b = n = 0 in
          if b then 1
          else let s = n - 1 in
               let m = factoriale s in
               n * m;;
val factoriale : int -> int = <fun>
# factoriale 5;;
- : int = 120
```

To put the function into full 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. In addition, each intermediate computation must be converted so that it also takes a continuation. Thus the code becomes:

```
# let rec factorialk n k =
      eqk n 0
      (fun b -> if b then k 1
                else subk n 1
                     (fun s -> factorialk s
                                (fun m  -> timesk n m k)));;
# factorialk 5 report;;
Result: 120
- : unit = ()
```

Notice that to make a recursive call, we needed to build an intermediate continuation capturing all the work that must be done after the recursive call returns and before we can return the final result. If `m` is the result of the recursive call in direct style (without continuations), then we need to build a continuation to:

- take the recursive value: `m`
- build it to the final result: `n * m`
- pass it to the final continuation `k`

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. All uses of functions need to take a continuation as an argument. All uses of primitive operations (*e.g.* +, −, ⋆, >, =) should use the corresponding functions defined in Problem 12. If you need to make use of primitive operations not covered in Problem 12, you should include a definition of the corresponding version that takes a continuation as an additional argument, as in Problem 12.

14. (8 pts) Write the function `partial_sumsk : float list -> (float list -> 'a) -> 'a` such that `partial_sumsk` is the continuation passing style version of the code you gave for `partial_sums` of Problem 2. You should have that `partial_sumsk lst (fun x -> x)` computes the same results as `partial_sums lst`. Order of evaluation must be kept when using operators and functions. Any procedure call must be in continuation passing style. **Make sure that you are converting your solution to Problem 2 into continuation passing style and not just creating a continuation passing style solution.**

```
# let rec partial_sumsk lst k = ... ;;
val partial_sumsk : float list -> (float list -> 'a) -> 'a = <fun>
# partial_sumsk [4.0; 2.2; 7.5] (fun x -> x);;
- : float list = [13.7; 9.7; 7.5]
```

15. (8 pts) Write the function `count_ifk : ('a -> (bool -> 'b) -> 'b) -> 'a list -> (int -> 'b) -> 'b` such that `count_ifk` is the continuation passing style version of the code you gave for `count_if` of Problem 5. You should have that `count_ifk (fun x -> fun k -> k (p x)) lst (fun x -> x)` computes the same results as `count_if p lst`. Order of evaluation must be kept when using operators and functions. Any procedure call in the function must also be in continuation passing style. **Make sure that you are converting your solution to Problem 5 into continuation passing style and not just creating a continuation passing style solution.**

```
let count_ifk pk lst k = ... ;;
- : ('a -> (bool -> 'b) -> 'b) -> 'a list -> (int -> 'b) -> 'b = <fun>
# count_ifk (fun x k -> k (x > 3)) [1;2;3;4;5] (fun x -> x);;
- : int = 2
```

4.4 Extra Credit

16. (8 pts) Write the function `min_elementk : 'a list -> (int -> 'b) -> 'b` such that `min_elementk` is the continuation passing style version of the code you gave for `min_element` of Problem 6. You should have that `min_elementk lst (fun x -> x)` computes the same results as `min_element list`. Order of evaluation must be kept when using operators and functions. Any procedure call in the function must also be in continuation passing style. **Make sure that you are converting your solution to Problem 6 into continuation passing style and not just creating a continuation passing style solution.**

```
# let min_elementk lst k = ... ;;
val min_elementk : 'a list -> (int -> 'b) -> 'b = <fun>
# min_elementk [] (fun x -> x);;
- : int = -1
```