MP 3 – Recursion Patterns and Higher-order Functions

CS 421 – Summer 2013
Revision 1.0

Assigned June 6, 2013
Due June 14, 2013, 11:59 PM
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Fixed problem numbers in the descriptions of allowed recursion patterns.

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 the sample execution.
- Students must comply with any special restrictions for each problem. Some of the problems require that the solution should be in forward- or 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 2 you **must** use forward recursion.
- In problems 3 through 4 you **must** use tail recursion.
- In problems 5 through 7, you **may not** use recursion, and instead **must** use the specified list library function.
- Problems 8 through 10 **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. For purposes of this assignment `@` is treated as a library function and is not to be used.

4.1 Recursion Patterns

For problems 1 through 4, you may **not** use library functions.

1. (3 pts) Write a function `concat` that takes a list of strings and returns the string formed by concatenating them all together, left to right. If the list is empty, you should return `""`. The function must use only *forward* recursion.

```
# let rec concat list = ... ;;
val concat : string list -> string = <fun>
# concat ["Hi, "; "my "; "name"; " is"];;
- : string = "Hi, my name is"
```

2. (5 pts) Write a function `part_concat` that takes a list of strings `list` and returns new list in which each element is the concatenation of that element and all following elements in `list`. The function must use only *forward* recursion.

```
# let rec part_concat list = ... ;;
val part_concat : string list -> string list = <fun>
# part_concat ["test"; "1"; "2"];;
- : string list = ["test12"; "12"; "2"]
```

3. (5 pts) Write a function `min` that takes a list of integers and returns the smallest integer in the list. If the list is empty, `min` should return -1. The function must use only *tail* recursion. You may want to use a (tail-recursive) helper function.

```
# let rec min list = ... ;;
val min : int list -> int = <fun>
# min [1; 2; 3; 4];;
- : int = 1
```

4. (6 pts) Write a function `dup` that takes a list `list` and returns a list in which each element of `list` is duplicated. You may use `@`, but use no other library functions. Your function must use only *tail* recursion. You may want to use a (tail-recursive) helper function.

```
# let dup list = ... ;;
val dup : 'a list -> 'a list = <fun>
# dup [1;2;3;4;5];;
- : int list = [1; 1; 2; 2; 3; 3; 4; 4; 5; 5]
```

Stop: go back and make sure you used no library functions for problems 1 through 3.

4.2 Higher-Order Functions

For problems 5 through 7, 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 5 through 7.

5. (3 pts) Write a value `concat_base` and function `concat_step` such that `List.fold_right concat_step list concat_base` computes the same results as `concat list` in Problem 1. There should be no use of recursion or library functions in `concat_step`.

```
# let concat_base = ... ;;
val concat_base : string = ...
# let concat_step x r =  ... ;;
val concat_step : string -> string -> string = <fun>
# List.fold_right concat_step ["Hi, "; "my "; "name"; " is"] concat_base;;
- : string = "Hi, my name is"
```

6. (7 pts) Write a function `min2` that computes the same results as `min` in Problem 3. The definition of `min2` may use `List.fold_left` but not recursion or any other library functions (though it may still use pattern matching).

```
# let min2 list = ... ;;
val min2 : int list -> int = <fun>
# min2 [1; 2; 3; 4];;
- : int = 1
```

7. (8 pts) Write a function `cross_sum` that takes two lists `xs`, `ys` and forms a matrix (i.e., list of lists) such that the (i, j)th element of the matrix is equal to the ith element of `xs` plus the jth element of `ys`. The definition `cross_sum` may use the library function `List.map`, but not recursion or any other library functions (though it may still use pattern matching).

```
# let cross_sum xs ys = ...;;
val cross_sum : int list -> int list -> int list list = <fun>
# cross_sum [1;3] [3;4];;
- : int list list = [[4; 5]; [6; 7]]
```

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 addk x y k = k (x + y);;
val addk : int -> int -> (int -> 'a) -> 'a = <fun>
```

The addk function takes two integers and a continuation, adds the two integers, and passes the result to its continuation.

```
# addk 3 4 report;;
Result: 7
- : unit = ()
# addk 1 2 addk 3 report;;
Result: 6
- : unit = ()
```

In line 1, addk adds 3 and 4, and then passes the result to report. In line 4, the first addk adds 1 to 2, and passes the resulting 3 to the second addk, which then adds 3 to it, passing the resulting 6 to report. Recall that we can also use more complex continuations to control the order of evaluation of expressions.

8. **(5 pts)** Write a function in calck in continuation passing style that takes four arguments a, b, c, d and a continuation k and returns (that is, passes k) the result of calcuating $(a + b) - (c + d)$. The function should first add c and d, then add a and b, and finally perform the subtraction. You must use the addk and subk functions instead of $+$ and $-$; these functions have been defined for you in mp3common. Note that simply passing the test cases is not sufficient to guarantee you full points here; you must perform the operations in the correct order and write your function in continuation passing style.

   ```
   # let calck a b c d k = ...
   val calck : int -> int -> int -> int -> (int -> 'a) -> 'a = <fun>
   # calck 5 4 3 2 report;;
   Result: 4
   - : unit = ()
   ```

While doing the following problems, it may help to refer to the slides on putting recursive functions into continuation passing style.

9. (8 pts) Write a function part_concatk that is the continuation passing style equivalent of part_concat from Problem 2. It should be the case that part_concatk list (fun x -> x) computes the same results as part_concat list. The functions concatk and consk, the continuation passing style equivalents of ^ and :: respectively, have been defined for you and are available for your use. Order of evaluation must be kept when using operators and functions. All procedure calls 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 part_concatk list k = ... ;;
   val part_concatk : string list -> (string list -> 'a) -> 'a = <fun>
   # part_concatk ["test"; "1"; "2"] (fun x -> x);;
   - : string list = ["test12"; "12"; "2"]
   ```

4.4 Extra Credit

10. (8 pts) Write a function dupk that is the continuation passing style equivalent of dup from Problem 4. It should be the case that dupk list (fun x -> x) computes the same results as dup list. The functions consk and appk, the continuation passing style equivalents of :: and @ respectively, have been defined for you and are available for your use. Order of evaluation must be kept when using operators and functions. All procedure calls must be in continuation passing style. **Make sure that you are converting your solution to Problem 4 into continuation passing style and not just creating a continuation passing style solution.**

```
# let dupk list k = ... ;;
val dup : 'a list -> ('a list -> 'b) -> 'b = <fun>
# dupk [1;2;3;4;5] (fun x -> x);;
- : int list = [1; 1; 2; 2; 3; 3; 4; 4; 5; 5]
```