MP 3 – Patterns of Recursion, Higher-order Functions

CS 421 – su 2010

Assigned June 20, 2011
Due June 26, 2011 23:59
Extension 48 hours (20% penalty)

1 Change Log

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 problem 11, you **may not** use recursion, and instead **must** use `List.map`.
- 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 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 `polar_to_cart : (float * float) list -> (float * float) list` such that `polar_to_cart l` returns a list with the Cartesian coordinate representation of the polar coordinates provided in `l`. The polar coordinates are represented as a pair (r, θ) and note that $x = rcos(\theta)$ and $y = rsin(\theta)$. You may use Ocaml's `cos` and `sin` functions. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec polar_to_cart l = ... ;;
val polar_to_cart : (float * float) list -> (float * float) list = <fun>
# polar_to_cart [ (0.0, 3.1415); (2.0, 0.0) ];;
- : (float * float) list = [(-0., 0.); (2., 0.)]
```

2. (5 pts) Write a function `determine_app : ('a -> bool) -> ('a -> 'b) -> ('a -> 'b) -> 'a list -> 'b list` such that `determine_app p f g l` returns a list with `f` applied to the element from `l` if the value of that same element applied to `p` is true. If the application of `p` on the element is false, then `g` should be applied to the element. The function is required to use (only) forward recursion (no other form of recursion). Recall, that in forward recursion no other function calls can be computed before the recursive call. You may not use any library functions.

```
# let rec determine_app p f g lst = ... ;;
val determine_app :
  ('a -> bool) -> ('a -> 'b) -> ('a -> 'b) -> 'a list -> 'b list = <fun>
# determine_app (fun x -> x mod 2 = 0)
    (fun y -> y*2)
    (fun z -> z+2)
    [1;2;3;4;5;6];;
- : int list = [3; 4; 5; 8; 7; 12]
```

3. (5 pts) Write a function `separate_all_from_to : int -> int -> (int -> bool) -> int * int` such that `separate_all_from_to m n p` returns a pair of integers, the first indicates the number of values between `m` and `n` inclusive that have a value of true when `p` is applied to it. The second integer in the pair is the number of values that that return false when hen `p` is applied. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec separate_all_from_to m n p = ... ;;
val separate_all_from_to : int -> int -> (int -> bool) -> int * int = <fun>
# separate_all_from_to 1 6 (fun x -> x mod 2 = 0);;
- : int * int = (3, 3)
```

4. (5 pts) Write a function `all_pred : ('a -> bool) -> 'a list -> bool` such that `all_pred p list` returns whether every element in the input list 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 all_pred p list = ... ;;
val all_pred : ('a -> bool) -> 'a list -> bool = <fun>
# all_pred (fun x -> x mod 2 = 0) [2;4;6;8];;
- : bool = true
```

5. (6 pts) Write a function `nearest : int -> int list -> int` such that `nearest n list` returns the element closest to `n` that is in the list. If the list is empty, return 0. Make sure to handle the base case appropriately so that all boundary cases are handled. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec nearest n list = ... ;;
val nearest : int -> int list -> int = <fun>
# nearest 4 [1;3;8;9] ;;
- : int = 3
```

6. (8 pts) Write a function `min_max : ('a -> int) -> 'a list -> int * int` such that `min_max f list` applies `f` to every element in the list and returns a pair such that the first element is the minimum value after the application of `f` and the second element is the maximum value after the application of `f`. If an empty list is given, return (0,0). Make sure to handle the base case appropriately so that all boundary cases are handled. You may not use Ocaml's min or max functions when writing this function. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec min_max f list = ... ;;
val min_max : ('a -> int) -> 'a list -> int * int = <fun>
# min_max (fun x -> x*x) [1; 1; 0; (-2)];;
- : int * int = (0, 4)
```

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 `determine_app_base` and function `determine_app_rec : ('a -> bool) -> ('a -> 'b) -> ('a -> 'b) -> 'a -> 'b list -> 'b list` such that `(fun list -> fun p -> fun f -> fun g -> List.fold_right (determine_app_rec p f g) list (determine_app_base))` computes the same results as `determine_app` of Problem 2. There should be no use of recursion or library functions in defining `determine_app_rec`.

```
# let determine_app_base = ... ;;
val determine_app_base : ...
# let determine_app_rec p f g n r = ... ;;
val determine_app_rec :
  ('a -> bool) -> ('a -> 'b) -> ('a -> 'b) -> 'a -> 'b list -> 'b list =
  <fun>
# (fun p f g l -> List.fold_right (determine_app_rec p f g) l (determine_app_base))
    (fun x -> x mod 2 = 0)
    (fun y -> y*2)
    (fun z -> z+2)
    [1;2;3;4;5;6];;
- : int list = [3; 4; 5; 8; 7; 12]
```

8. (7 pts) Write a value `polar_to_cart_base` and function `polar_to_cart_rec : float * float -> (float * float) list -> (float * float) list` such that `(fun list -> List.fold_right polar_to_cart_rec list polar_to_cart_base)` computes the same results as `polar_to_cart` of Problem 1. There should be no use of recursion or library functions in defining `polar_to_cart_rec`.

```
# let polar_to_cart_base = ...
val polar_to_cart_base : ...
# let polar_to_cart_rec n r = ...
val polar_to_cart_rec :
  float * float -> (float * float) list -> (float * float) list = <fun>
# (fun list -> List.fold_right polar_to_cart_rec list polar_to_cart_base)
    [(0.0, 3.1415); (2.0, 0.0)];;
- : (float * float) list = [(-0., 0.); (2., 0.)]
```

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

```
# let all_pred_base = ...
val all_pred_base : ...
# let all_pred_rec p r x = ... ;;
val all_pred_rec : ('a -> bool) -> bool -> 'a -> bool = <fun>
# (fun p -> fun list -> List.fold_left (all_pred_rec p) all_pred_base list)
    (fun x -> x mod 2 = 0)
    [2;4;6;8];;
- : bool = true
```

10. (7 pts) Write a base case and recursive input function for fold_left in order to accomplish the `nearest` calculation found in Problem 5. The base case function will be of the type `nearest_base : int list -> int` and the recursively applied function in fold_left will be of the form `nearest_rec : int -> int -> int -> int`. They should be written such that `(fun n -> fun list -> List.fold_left (nearest_rec n) (nearest_base list) list)` functions just like `nearest` of Problem 5. There should be no use of recursion or other library functions in defining either function.

```
# let nearest_base list = ...;;
val nearest_base : int list -> int = <fun>
# let nearest_rec n r x = ...;;
val nearest_rec : int -> int -> int -> int = <fun>
# (fun n -> fun list -> List.fold_left (nearest_rec n) (nearest_base list) list)
    4
    [1;3;8;9];;
- : int = 3
```

11. (8 pts) Write a function `app_all_with : ('a -> 'b -> 'c) list -> 'a -> 'b list -> 'c list list` that takes a list of functions, a single first argument, and a list of second arguments and returns a list of list of results from consecutively applying the functions to all arguments after applying the single argument to each function first. The functions should be applied in the order they appear in the list and in the order in which the arguments appear in the second list. Each list in the result corresponds to a list of applications of a function on the single argument and then the given argument from the second list. The definition `app_all_with` 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_with fs x list = ... ;;
val app_all_with : ('a -> 'b -> 'c) list -> 'a -> 'b list -> 'c list list =
  <fun>
# app_all_with [(fun x y -> x+y); (fun x y -> x*y )] 47 [1;2;3];;
- : int list list = [[48; 49; 50]; [47; 94; 141]]]
```

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`, `absk`, `plusk`, `multk`, and `is_posk` in CPS. `subk` subtracts the first integer from the second; `catk` concatenates the first string in front of the second; `absk` get the absolute value of an integer, `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 absk x k = ...;;
val absk : 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 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"
# absk (-2) report;;
Result: 2
- : unit = ()
# 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.

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 `*.` 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: 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
```

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 = ()
```

Make sure that the multiplication is done as the last operation and the additions are applied from left to right. 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.` +, −, *, >, =, `mod`, ^) do not need to take a continuation as an argument.

14. (8 pts) Write the function `polar_to_cartk : (float * float) list -> ((float * float) list -> 'a) -> 'a` such that `polar_to_cartk` is the continuation passing style version of the code you gave for `polar_to_cart` of Problem 1. You should have that `polar_to_cartk list (fun x -> x)` computes the same results as `polar_to_cart list`. Order of evaluation must be kept when using operators and functions. Any procedure call must be in continuation passing style. You may use the `cos` and `sin` function as primitive operations in this problem. **Make sure that you are converting your solution to problem 1 into continuation passing style and not just creating a continuation passing style solution.**

```
# let rec polar_to_cartk list k = ...;;
val polar_to_cartk :
  (float * float) list -> ((float * float) list -> 'a) -> 'a = <fun>
# polar_to_cartk [(0.0, 3.1415); (2.0, 0.0)] (fun x->x);;
- : (float * float) list = [(-0., 0.); (2., 0.)]
```

15. (8 pts) Write the function `all_predk : ('a -> (bool -> 'b) -> 'b) -> 'a list -> (bool -> 'b) -> 'b` such that `all_predk` is the continuation passing style version of the code you gave for `all_pred` of Problem 4. You should have that `all_predk (fun x -> fun k -> k (p x)) list (fun x -> x)` computes the same results as `all_pred p 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 4 into continuation passing style and not just creating a continuation passing style solution.**

```
# let rec all_predk pk list k = ... ;;
val all_predk : ('a -> (bool -> 'b) -> 'b) -> 'a list -> (bool -> 'b) -> 'b =
  <fun>
# all_predk (fun x k -> k (x mod 2 = 0)) [2;4;6;8] (fun x -> x);;
- : bool = true
```

4.4 Extra Credit

16. (8 pts) Write a function `nearestk : int -> int list -> (int -> 'a) -> 'a` such that `nearestk` is the continuation passing style version of `nearest` of Problem 5. You should have that `nearestk n list (fun x->x)` computes the same results as `nearest n list`. You must make use of `absk` created in problem 12. 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 rec nearestk n list k = ...;;
val nearestk : int -> int list -> (int -> 'a) -> 'a = <fun>
# nearestk 4 [1;3;8;9] report;;
Result: 3
- : unit = ()
```