MP 2 – Pattern Matching and Recursion

CS 421 – Summer 2013

Revision 1.0

Assigned May 31, 2013
Due June 7, 2013, 11:59 PM
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Fixed the name of `polar_prod` in problem 1.

2 Objectives and Background

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

- pattern matching
- higher-order functions
- recursion

3 Instructions

The problems below have sample executions that suggest how to write answers. You must use the same function name, but the name of the parameters that follow the function name need not be duplicated. That is, you are free to choose different names for the arguments to the functions from the ones given in the example execution. We will sometimes 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`, and you may use `let rec` when we do not.

For 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 in several problems. In this assignment, **you may not use any library functions other than** `@` except where explicitly noted.

4 Problems

1. (3 pts) Complex numbers can be represented in polar form as a pair of floating-point numbers, the distance from the origin ("magnitude") and the angle from the positive x-axis ("argument"). We write a complex number with magnitude r and argument θ as $r \text{ cis } \theta$. It can be shown that the product of two complex numbers is given by $(r_1 \text{ cis } \theta_1)(r_2 \text{ cis } \theta_2) = r_1 r_2 \text{ cis } (\theta_1 + \theta_2)$. Write a function `polar_prod` that takes a pair of complex numbers in polar form, each represented as a pair (r, θ) of floating-point numbers, and outputs their product.

```
# let polar_prod r = ...;;
val polar_prod : (float * float) * (float * float) -> float * float = <fun>
# polar_prod ((4., 0.), (2., 3.));;
- : float * float = (8., 3.)
```

2. (3 pts) Consider the following mathematical definition of a sequence s_n:

$$s_n = \begin{cases} 0 & \text{if } n = 0 \\ 2s_{n-1} + n & \text{if } n > 0 \text{ and } n \text{ is odd} \\ 3s_{n-2} + 2n & \text{if } n > 0 \text{ and } n \text{ is even} \end{cases}$$

Write a function `s` that implements the sequence s_n. For $n \leq 0$, you should return 0.

```
# let rec s n = ...;;
val s : int -> int = <fun>
# s 5;;
- : int = 45
```

3. (3 pts) Write a function `filter : ('a -> bool) -> 'a list -> 'a list` that takes a boolean function and a list `xs` and returns the list of all the elements of `xs` for which the boolean function is true.

```
# let rec filter p xs = ...;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
# filter (fun x -> x < 0) [1; -1; 0; 4; -2; 5];;
- : int list = [-1; -2]
```

4. (3 pts) Write a function `has_two` that takes a boolean function and a list, and returns true if the predicate is true on two or more elements of the list and false otherwise.

```
# let rec has_two p xs = ...
val has_two : ('a -> bool) -> 'a list -> bool = <fun>
# has_two (fun x -> x > 30) [1; 34; 42; 6];;
- : bool = true
```

5. (5 pts) Write a function `sum` that takes two lists of numbers `xs` and `ys` and returns a list in which each element is equal to the corresponding element of `xs` plus the corresponding element of `ys`. If one list is longer than the other, treat this as if the shorter list contained 0's up to the length of the longer list.

```
# let rec sum xs ys = ...;;
val sum : int list -> int list -> int list = <fun>
# sum [1; 2; 5] [3; 4];;
- : int list = [4; 6; 5]
```

6. (5 pts) Write a function `split` that takes a list `xs` and outputs a pair of lists, in which the first list contains all the non-negative (0 and positive) elements of `xs` and the second list contains all the negative elements of `xs`, in order. Do not use the `filter` function in your solution.

```
# let rec split xs = ...;;
val split : int list -> int list * int list = <fun>
# split [1; -3; 2; -4; 5];;
- : int list * int list = ([1; 2; 5], [-3; -4])
```

7. (7 pts) The *mean* of a set of data points is the average computed by adding the data together and dividing by the number of data points. Write a function `diff_from_mean` that takes a list of floating-point numbers and subtracts the mean of the list from each element in the list. You may use `List.length` and `float_of_int` in your answer, but you may not use any other library functions. Your function should return the empty list when given the empty list, rather than crashing.

   Hint: You may want to write one or more auxiliary functions.

```
# let rec diff_from_mean xs = ...;;
val diff_from_mean : float list -> float list = <fun>
# diff_from_mean [3.1; 6.1; 9.1];;
- : float list = [-3.0, 0.0, 3.0]
```

   (Your numbers may be inexact due to rounding error, but they should still match those given by the grader. If you think you're being marked wrong due to rounding error, let the TA or the instructor know.)

8. (7 pts) We can represent a matrix (two-dimensional array) as a list of lists, where each sublist gives the elements in one row of the matrix. Write a function `check_ijth` that takes a list of lists, a boolean function, and an index (i, j) into the matrix, and returns true if the boolean function is true of the element in the ith row and the jth column, and false otherwise. If there is no element (i, j) (for instance, if i is less than 0, or there are not j elements in the ith row), `check_ijth` should return false. Keep in mind that both rows and columns are counted starting from 0, so the first element in the first row is at index $(0, 0)$ and so on.

```
# let rec check_ijth matrix p (i,j) = ...
val check_ijth : 'a list list -> ('a -> bool) -> int * int -> bool = <fun>
# check_ijth [[1;2;3;4];[3;0;4;5];[1;4;3;5];[2;1;1;2]] (fun x -> x > 3) (0,3);;
- : bool = true
```

4.1 Extra Credit

9. (5 pts)

   A matrix is *diagonal* if all its elements outside the diagonal ($(1, 1)$, $(2, 2)$, etc.) are 0. Write a function `is_diagonal` that takes a list of lists and returns true if and only if the list is actually a matrix (i.e., it has the same number of elements in every row) and is diagonal. You may write any number of helper functions, but may not use any library functions.

```
# let rec is_diagonal matrix = ...
val is_diagonal : int list list -> bool = <fun>
# is_diagonal [[1; 0]; [0; 1]];;
- : bool = true
```