MP 2 – Pattern Matching and Recursion

CS 421 – su 2011

Assigned June 13, 2011
Due June 19, 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. pattern matching
2. higher-order functions
3. recursion

3 Instructions

The problems below have sample executions that suggest how to write answers. You have to 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 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. In this assignment, **you may not use any library functions** (except `@`, which is also pervasive).

4 Problems

4.1 Pattern Matching

1. (3 pts) Write `reverse_fst_trd : 'a * 'b * 'c -> 'c * 'b * 'a` that reverses the first and the third elements in a given triple.

```
# let reverse_fst_trd ... = ...;;
val reverse_fst_trd : 'a * 'b * 'c -> 'c * 'b * 'a = <fun>
# reverse_fst_trd (3, 1.0, "hi");;
- : string * float * int = ("hi", 1.0, 3)
```

2. (4 pts) Write `square_dist : (float * float) * (float * float) -> float` that takes two points as a pair of pairs of floats, and calculates the square of the distance between them

```
# let square_dist ... = ...;;
val square_dist : (float * float) * (float * float) -> float = <fun>
# square_dist ((1.0, 2.0), (3.0, 4.0));;
- : float = 8.0
```

3. (5 pts) Write `first_two_pairs : 'a list -> 'b list -> ('a * 'b) list` that returns a list whose first element is a pair of the first elements of the input lists and the second element is a pair of the second elements of the input lists. If either of input lists has less than 2 elements, `first_two_pairs` should return an empty list.

```
# let first_two_pairs l1 l2 = ...;;
val first_two_pairs : 'a list -> 'b list -> ('a * 'b) list = <fun>
# first_two_pairs [1;2;3] [4;5;6;7];;
- : (int * int) list = [(1,4);(2,5)]
# first_two_pairs [1] [5;7]
- : (int * int) list = []
```

4.2 Recursion

4. (4 pts) Write `sum : int list -> int` that returns the sum of all elements in the list, or 0 if the list is empty.

```
# let rec sum l = ...;;
val sum : int list -> int = <fun>
# sum [0; 2; 5];;
- : int = 7
```

5. (5 pts) Consider the following procudure: given a positive integer n, if n is even, divide it by 2, if n is odd multiply it by 3 and add 1. The Collatz conjecture states that if you repeatedly apply this procedure, you will reach 1 eventually. Write a function `collatz : int -> int` that takes a positive integer n and returns the number of steps it is needed for n to reach 1.

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

6. (5 pts) Write a function `unzip : ('a * 'b) list -> 'a list * 'b list` that returns a pair of lists where the first list contains the first element in each pair and the second list contains the second element in each pair.

```
# let rec unzip l = ...;;
val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>
# unzip [(1,"hello"); (2,"world")];;
- : int list * string list = ([1; 2], ["hello"; "world"])
```

7. (6 pts) Write a function `dotproduct : int list -> int list -> int` that calculates a dot-product of 2 lists (sum of products of respective elements). If one list is greater than the other one, then excessive elements are discarded. If either list is empty, `dotproduct` should return 0.

```
# let rec dotproduct l1 l2 = ...;;
val dotproduct : int list -> int list -> int = <fun>
# dotproduct [1;2] [3;4];;
- : int = 11
```

8. (6 pts) Write a function `partition : 'a -> 'a list -> 'a list * 'a list * 'a list` that takes an integer x and an int list l, and partiotions l with respect to x, producing a tuple of three lists (a, b, c), such that:

   - a - list of elements of l that are less than x;
   - b - list of elements of l that are equal to x;
   - c - list of elements of l that are greater than x;

   Elements of a, b and c must remain in the same order as they were in l.

```
# let rec partition x l = ...
val partition : 'a -> 'a list -> 'a list * 'a list * 'a list = <fun>
# partition 2 [1;2;1;4;5;2;5;3;7;1;0];;
- : int list * int list * int list = ([1; 1; 1; 0], [2; 2], [4; 5; 5; 3; 7])
```

9. (5 pts) Write a function `remove_all : ('a -> bool) -> 'a list -> 'a list` that takes a predicate and a list and returns a new list with all elements that satisfy the predicate removed.

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

10. (7 pts) Write a function `mapidx : (int -> 'a -> 'b) -> 'a list -> 'b list` that takes a function and a list and builds a new list whose elements are the result of applying the function to each element of the list and that element's index (position in the list, starting from 0).

```
# let mapidx f l = ...
val mapidx : (int -> 'a -> 'b) -> 'a list -> 'b list = <fun>
# mapidx (fun i s -> (i,s)) ["hello"; "world"];;
- : (int * string) list = [(0, "hello"); (1, "world")]
```

4.3 Extra Credit

11. (5 pts) Write a function `group : 'a list -> 'a list list` that takes a list of elements and groups each sequence of consecutive equal elements in a list to a separate list, producing a list of lists.

```
# let rec group l = ...
val group : 'a list -> 'a list list = <fun>
# group [1;1;2;3;1;4;5;5;6];;
- : int list list = [[1; 1]; [2]; [3]; [1]; [4]; [5; 5]; [6]]
# group [1;1;2;2;1;1];;
- : int list list = [[1; 1]; [2; 2]; [1; 1]]
```