

---

# MP 2 – Pattern Matching and Recursion

CS 421 – Fall 2008

Revision 1.1

**Assigned** September 2, 2008

**Due** September 9, 2008 23:59

**Extension** 48 hours (20% penalty)

---

## 1 Change Log

1.1 Corrected base case for 6 (accumulate) to return [] instead of 0.

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. 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 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_pair : 'a * 'b -> 'b * 'a` that reverses the elements of the given pair.

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

2. (4 pts) Write `add_times : (int * int) * (int * int) -> int` that takes a pair of integer pairs, and returns the sum of the products of the elements of the integer pairs.

```
# let add_times ... = ...;;
val add_times : (int * int) * (int * int) -> int = <fun>
# add_times ((3, 4), (5, 7));;
- : int = 47
```

3. (5 pts) Write `rev_first_two : 'a list -> 'a list` that reverses the first two elements of a list, or does nothing to a one- or zero-element list.

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

## 4.2 Recursion

4. (6 pts) Write `concat : string list -> string` that concatenates the elements of its argument, returning "" on the empty string.

```
# let rec concat l = ...;;
val concat : string list -> string = <fun>
# concat ["How "; "are "; "you?"];;
- : string = "How are you?"
```

5. (7 pts) Write `add_sub : int list -> int` that calculates the sum of the elements in odd positions of the input list, minus the sum of elements in even positions of the input list. `add_sub` should return 0 on the empty list.

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

6. (8 pts) Write `accumulate : int list -> int list` that returns a list where position  $i$  is the sum of the first  $i$  elements of the input list. `accumulate` should return [] on the empty list.

```
# let accumulate l = ...;;
val accumulate : int list -> int list = <fun>
# accumulate [1;2;3;4;5];;
- : int list = [1; 3; 6; 10; 15]
```

7. (8 pts) Write `fixpoint : ('a -> 'a) -> 'a -> 'a` such that `fixpoint f n` applies `f` to `n` repeatedly until a fixed point `x`, where `f x = x`, is reached. `x` should be returned.

**NOTE:** this function will fail to terminate if no fixed point can be reached by this method. We will not test you on such inputs.

```
# let fixpoint f n = ...;;
val fixpoint : ('a -> 'a) -> 'a -> 'a = <fun>
# fixpoint (fun x -> (x +. 2.0 /. x) /. 2.0) 1.0;;
- : float = 1.41421356237309492
```

8. (6 pts) Write `count : 'a list -> 'a -> int` that such that `count l a` returns the number of times that `a` occurs in `l`.

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

9. (6 pts) Write `all : ('a -> bool) -> 'a list -> bool` that returns true on all `p l` iff `p` holds of all elements in `l`.

```
# let rec all p l = ...;;
val all : ('a -> bool) -> 'a list -> bool = <fun>
# all (fun x -> x > 0) [1;2;3;4;5];;
- : bool = true
```

10. (6 pts) Write `filter : ('a -> bool) -> 'a list -> 'a list`, such that `filter p l` returns, in their original order, the elements of `l` satisfying `p`.

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

### 4.3 Extra Credit

(5 pts)

11. Write `riffle : 'a list -> 'a list -> 'a list` such that `riffle [x1; x2; ...; xn] [y1; y2; ...]` returns `[x1; y1; x2; y2; ...; xn]`, cycling through the second list as many times as necessary, and discarding any elements remaining in the second list at the end of the process (in particular, if the first list is empty, the second list is entirely discarded.)

If the second list is empty, then the first list should be returned unchanged.

```
# let riffle l1 l2 = ...;;
val riffle : 'a list -> 'a list -> 'a list = <fun>
# riffle ["how"; "are"; "you?"] [" "];;
- : string list = ["how"; " "; "are"; " "; "you?"]
# riffle [1;2;3;4;5;6] [-1;-2];;
- : int list = [1; -1; 2; -2; 3; -1; 4; -2; 5; -1; 6]
```