

---

# MP 1 – Pattern Matching and Recursion

CS 421 – Spring 2010

Revision 1.0

**Assigned** January 19, 2010

**Due** January 25, 2010 10:00pm

**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 Collaboration

Collaboration is allowed in this assignment.

## 4 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).

Some functions in this assignment have polymorphic types. These functions can be tested on non-integer inputs as well. Please be careful about your function types.

## 5 Problems

### 5.1 Pattern Matching

1. (3 pts) Write `pair_to_list : 'a * 'a -> 'a list` that takes a pair of two 'a type elements and returns a list of two elements in the reversed order.

**Note:** input can be a non-integer pair. Your function should be polymorphic.

```
# let pair_to_list ... = ...;;
val pair_to_list : 'a * 'a -> 'a list = <fun>
# pair_to_list (5, 9);;
- : int list = [9; 5]
```

2. (4 pts) Write `sum_largest : int * int -> int * int` that takes a pair of integers, and returns a pair of the sum of the integers and the largest among them.

```
# let sum_largest ... = ...;;
val sum_largest : int * int -> int * int = <fun>
# sum_largest (7, 2);;
- : int * int = (9, 7)
```

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

**Note:** input can be a non-integer list. Your function should be polymorphic.

```
# let sort_first_two ... = ...;;
val sort_first_two : 'a list -> 'a list = <fun>
# sort_first_two [8; 2; 5];;
- : int list = [2; 8; 5]
# sort_first_two [3; 7; 4];;
- : int list = [3; 7; 4]
```

## 5.2 Recursion

4. (6 pts) Write `concat_odd : string list -> string` that concatenates the elements in odd positions of the input list, returning "" on the empty string.

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

5. (7 pts) Write `largest : int list -> int` that takes a list of integers and returns the largest integer. largest should return 0 on the empty list.

```
# let rec largest l = ...;;
val largest : int list -> int = <fun>
# largest [4;9;3;2;7];;
- : int = 9
```

6. (7 pts) Write a function `bitmap_nonneg : int list -> int list` that returns a list where position  $i$  is 1 if  $i$ th element of the input is nonnegative and 0 otherwise. `bitmap_nonneg` should return an empty list on the empty input.

```
# let rec bitmap_nonneg l = ...;;
val bitmap_nonneg : int list -> int list = <fun>
# bitmap_nonneg [5; -2; 0; 4; -3];;
- : int list = [1; 0; 1; 1; 0]
```

7. (8 pts) Write `flatten : 'a list list -> 'a list` that returns the list consisting of the concatenation of the lists in the argument. Do **not** use the built-in `@` operator. Also you don't need to use any auxiliary function.

**Note:** input can be a list of non-integer lists. Your function should be polymorphic.

```
# let rec flatten ... = ...;;
val flatten : 'a list list -> 'a list = <fun>
# flatten [[1;2;3]; [4;5]; [8;2;3;4]];
- : int list = [1;2;3;4;5;8;2;3;4]
```

8. (6 pts) Write `is_sorted : 'a list -> bool` that returns true if the input list is sorted ascendingly, false otherwise.

**Note:** input can be a non-integer list. Your function should be polymorphic.

```
# let rec is_sorted l = ...;;
val is_sorted : 'a list -> bool = <fun>
# is_sorted [1;2;3;4;5;8;9;11];;
- : bool = true
# is_sorted [2;3;4;5;7;6;8];;
- : bool = false
```

9. (8 pts) Write `merge : 'a list -> 'a list -> 'a list` whose arguments are lists in ascending order, and whose result is the merging of the two lists, also in ascending order.

**Note:** input can be non-integer lists. Your function should be polymorphic.

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

### 5.3 Extra Credit

(8 pts)

10. Write `simple_poker : (int * int) list -> int * int` that takes a list of integer pairs and return the highest ranking. A pair of the same integers is always higher than a pair of different integers. A pair containing higher integer is higher than others if it does not violate the first rule. `simple_poker` returns (0, -1) on the empty input.

```
# let rec simple_poker ... = ...;;  
val simple_poker : (int * int) list -> int * int = <fun>  
# simple_poker [(2, 3); (5, 9); (3, 3); (9, 8); (6, 6)];;  
- : int * int = (6, 6)  
# simple_poker [(3, 2); (8, 2); (7, 6)];;  
- : int * int = (8, 2)
```