MP2 – Pattern Matching and Recursion

CS 421 – Summer 2009
Revision 1.2

Assigned June 4, 2009
Due June 12, 2009, 1:00 PM
Extension 48 hours (penalty 20% of total points possible)

1 Change Log

1.2 Fixed Problem 10 grader bug introduced in 1.1.
1.1 Extra Credit Problem 10 has been further disambiguated, mp2grader script for Problem 10 has been fixed to conform to the problem description, and its point value has been increased from 8 to 10.
1.0 Initial release.

2 Objectives and Background

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

- recursion
- pattern matching
- polymorphic functions

3 Collaboration

Collaboration is NOT 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 `tuple_to_list : 'a * 'a * 'a -> 'a list` that takes a tuple with three `'a` type elements and returns a list of three elements in the *reversed* order.

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

```
# let tuple_to_list ... = ...;;
val tuple_to_list : 'a * 'a * 'a -> 'a list = <fun>
# tuple_to_list (5, 7, 9);;
- : int list = [9; 7; 5]
```

2. (4 pts) Write `mean_max : int * int -> int * int` that takes a pair of integers, and returns a pair containing the mean (average) and maximum of the two values. Let OCaml handle the rounding, if any.

```
# let mean_max ... = ...;;
val mean_max : int * int -> int * int = <fun>
# mean_max (8, 2);;
- : int * int = (5, 8)
```

3. (5 pts) Write `sort_first_two : 'a list -> 'a list` that reverses the fist 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_even : string list -> string` that concatenates the elements in even positions of the input list, returning "" on empty and one-element lists.

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

5. (7 pts) Write `largest_of : int list -> int * int` that takes a list of integers and retuns a pair containing the largest integer and the number of elements in the list. `largest_of` should return `(0,0)` on the empty list.

```
# let rec largest_of l = ...;;
val largest_of : int list -> int * int = <fun>
# largest_of [4;9;3;2;7];;
- : int * int = (9, 5)
```

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

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

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 `prune_sort : 'a list -> 'a list` that sorts the input list by discarding out-of-order elements. The list should be sorted in ascending order.

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

```
# let rec prune_sort l = ...;;
val prune_sort : 'a list -> 'a list = <fun>
# prune_sort [1;3;2;5;4;5;11;9];;
- : int list = [1; 3; 5; 5; 11]
```

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

10. (10 pts) Write `simple_poker : (int * int) list -> int * int` that takes a list of integer pairs and returns the highest ranking. Rules:

    - A pair of the same integers is always higher than a pair of different integers.
    - A pair containing the higher integer is higher than others if it does not violate the first rule.

- If pairs have the same higher integer, the one whose lower integer is higher than others is higher if it does not violate the first rule.
- `simple_poker` returns $(0, -1)$ on the empty input.
- **New rule:** If two pairs are equivalent, the one appearing earlier in the list should be returned.

```
# 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)
# simple_poker [(2, 3); (3, 2)];;
- : int * int = (2, 3)
```