MP 1 – Pattern Matching and Recursion

CS 421 – Fall 2017

Revision 1.0

Assigned Thursday, September 7, 2017
Due Thursday, September 14, 2017 22:00pm
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:

- pattern matching
- 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, or patterns, 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 only use library functions from the Pervasives module (the one that is loaded by default.) In particular, you must not use any functions from List**.

4 Problems

1. (2 pts) Write `closer_to_origin : float * float -> float * float -> int` that takes two 2-dimensional float points and determines which is closer to the origin by Euclidean distance. If the first point is closer, it should evaluate to -1, if the second is closer, it should evaluate to 1, and if the points are equidistant from the origin, it should evaluate to 0.

```
# closer_to_origin p1 p2 = ...
val closer_to_origin : float * float -> float * float -> int = <fun>
# closer_to_origin (2., 0.) (0., -1.);;
- : int = 1
```

2. (2 pts) Write `swap_eq : 'a * 'b -> 'b * 'a -> bool` that takes two pairs and determines whether the second is the left-right swap of the first.

```
# let swap_eq p1 p2 = ...
val swap_eq : 'a * 'b -&gt 'b * 'a -&gt bool = &lt;fun&gt;
# swap_eq (1., 0.) (0., 1.);;
- : bool = true
```

3. (2 pts) Write a function `two_funs : ('a -> 'b) * ('c -> 'd) -> 'a * 'c -> 'b * 'd` which takes a pair of unary functions and a pair of inputs and returns the pair of the first function applied to the first input and the second function applied to the second input.

```
# let two_funs fns ins = ...
val two_funs : ('a -> 'b) * ('c -> 'd) -> 'a * 'c -> 'b * 'd = <fun>
# two_funs (not, abs) (true, -5);;
- : bool * int = (false, 5)
```

4. (3 pts) The Ackermann function $A :\ \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is a recursive function defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{otherwise} \end{cases}$$

Write an OCaml function `ackermann : int -> int -> int` that takes the numbers m and n and returns the value of $A(m,n)$. You can assume m and n will be non-negative.

```
# let rec ackermann m n = ...
val ackermann : int -> int -> int = <fun>
# ackermann 3 4;;
- : int = 125
```

5. (3 pts) The Collatz sequence for a positive integer n starts at n and repeats the following: if the number is even, divide by 2 to get the next number in the sequence. If it is odd, multiply by 3 and add 1. It is conjectured that the Collatz sequence reaches 1 for any positive integer n. Write a function `collatz : int -> int` which, given an integer n returns the number of steps its Collatz sequence takes to reach 1 (it takes 0 steps for 1 to reach itself.) You can assume n will be positive.

```
# let rec collatz n = ...
val collatz : int -> int = <fun>
# collatz 27;;
- : int = 111
```

6. (3 pts) The Delannoy number $d_{m,n}$ counts the number of paths on a gridded $m \times n$ rectangle from the origin $(0,0)$ (at the South-West corner of the rectangle) to the point (m,n) (at the North-East corner) where only North, East, and North-East steps are allowed. Note: the number of paths from $(0,0)$ to $(0,0)$ is 1. Write a function `delannoy : int * int -> int` that takes the point (m,n) and returns $d_{m,n}$. You can assume m and n will be non-negative.

```
# let rec delannoy (m, n) = ...
val delannoy : int * int -> int = <fun>
# delannoy (1, 2);;
- : int = 5
```

7. (2 pts) Write a function `product : float list -> float` to find the product of a list of floats. The product of an empty list is 1.0.

```
# let rec product l = ...
val product : float list -> float = <fun>
# product [2.; 3.; 4.];;
- : float = 24.
```

8. (2 pts) Write a function `double_all : float list -> float list` that takes a list of floats and returns back the list with all of the elements doubled.

```
# let rec double_all l = ...
val double_all : float list -> float list = <fun>
# double_all [1.5; -3.0; 0.; 2.2];;
- : float list = [3.; -6.; 0.; 4.4]
```

9. (3 pts) Write a function `upto : int -> int list` that takes an integer n and returns the list of natural numbers (starting from 0) up to, and including, n.

```
# let rec upto n = ...
val upto : int -> int list = <fun>
# upto 8;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8]
```

10. (4 pts) Write a function `upuntil : (int -> bool) -> int list` that takes a function f from integers to booleans, and returns the list of natural numbers (starting from 0) up until the first number for which f evaluates to true. Do not include this number in the list. If f evaluates to false for the first 100 natural numbers, return the list of the first 100 natural numbers instead.

```
# let rec upuntil f = ...
val upuntil : (int -> bool) -> int list = <fun>
# upuntil (fun n -> n * n > 200);;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14]
```

11. (3 pts) Write a function `pair_with_all : 'a -> 'b list -> ('a * 'b) list` that takes a value and a list, and creates a list of pairs where the given value is first in every pair, and the elements of the list are second in the pairs, in the same order as the original list.

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

12. (4 pts) Write a function `insert_by : ('a -> 'a -> int) -> 'a -> 'a list -> 'a list` that takes a comparison function `comp`, an element x and a list. It returns a list with the element inserted immediately before the first element that is "greater" than x. If no such position exists, insert x at the end of the list. Here "greater" is determined by `comp`: it takes two elements and returns 1 if the first is "greater", -1 if the second is "greater", and 0 if the two are considered equal. Note, if the list is sorted according to the order defined by `comp`, the order is preserved by the insertion.

```
# let rec insert_by comp x l = ...
val insert_by : ('a -> 'a -> int) -> 'a -> 'a list -> 'a list = <fun>
# insert_by compare 3 [1; 2; 4];;
- : int list = [1; 2; 3; 4]
# insert_by closer_to_origin (2., 0.) [(0., -1.); (-3., 0.)];;
- : (float * float) list = [(0., -1.); (2., 0.); (-3., 0.)]
```

13. (5 pts) For two lists L_1 and L_2, L_2 is called a *sub-list* of L_1 if: (a) all the elements of L_2 occur in L_1, and (b) their order in L_1 is *exactly* the same as their order in L_2. Write a function `sub_list : 'a list -> 'a list -> bool` that takes two lists as input and determines whether the second list is a sub-list of the first one. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions.

```
# let rec sub_list l1 l2 = ... ;;
val sub_list : 'a list -> 'a list -> bool = <fun>
# sub_list [1;1;2;1;1;4;1] [1;2;1;1;1];;
- : bool = true
```

5 Extra Credit

14. (5 pts) Write a function `collect_adjacent : ('a * 'b) list -> ('a * ('b list)) list` that takes a list of (key, value) pairs and returns a list of (key, value list) pairs, where every maximal sublist whose pairs have the same key is collected into a single pair, pairing the shared key to the list of all associated values, in the order in which they appeared in the input list.

```
# let rec collect_adjacent l = ...
val collect_adjacent : ('a * 'b) list -> ('a * 'b list) list = <fun>
# collect_adjacent [(1, "a"); (1, "d"); (1, "b"); (0, "b"); (0, "z"); (1, "a");
(1, "z"); (3, "t")];;
- : (int * string list) list =
[(1, ["a"; "d"; "b"]); (0, ["b"; "z"]); (1, ["a"; "z"]); (3, ["t"])]
```