MP 2 – Pattern Matching and Recursion

CS 342 – Spring 2024
Revision 1.0

Assigned Tuesday, Jan 23, 2024
Due Feb 2, 2024 – Feb 6, 2024
Extension 96 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Objectives and Background

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

- pattern matching
- higher-order functions
- 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 and structure 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, or even a different number of arguments from the ones given in the example execution.

The functions you write must have the types prescribed in the problems. If you fail to write functions of the right type, your code will not compile with the code for the grade and you submission will receive no points.

We will sometimes use `let rec` to begin the definition of a function that may involve recursion. You are not required to start your code with `let rec`, and you may use `let rec` when we do not. However, you are expected to know when the problem requires you to write a recursive solutions, and how to do that. For all these problems, you are allowed to write your own auxiliary functions and local declarations.

In this assignment, you may not use the functions `fst` and `snd`. This restriction will not usually apply, and will be stated in the assessment when it does apply.

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.

```
# let 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 -> 'b * 'a -> bool = <fun>
# swap_eq (1., 0.) (0., 1.);;
- : bool = true
```

3. (2 pts) Write `twist : ('a * 'b) * ('c * 'd) -> ('d * 'a) * ('c * 'b)` that takes a pair of pairs and returns a new pair of pairs where the first pair in the returned value is the second element of the second pair of the input paired with the first element of the first pair of the input and the second pair in the returned value is the first element of the second pair in the input, paired with the paired with the second element of the first pair.

```
# let twist pp = ...
val twist : ('a * 'b) * ('c * 'd) -> ('d * 'a) * ('c * 'b) = <fun>
# twist (("hi",true),(2, 17.3));;
- : (float * string) * (int * bool) = ((17.3, "hi"), (2,true))
```

4. (2 pts) Write `triple_pairs : 'a -> 'b * 'c * 'd -> ('a * 'b) * ('a * 'c) * ('a * 'd)` that takes an element and a triple, and returns a triple of pairs where each pair has the first element is the first input argument and the second element is from the corresponding position in the input triple.

```
# let triple_pairs x trp = ...
val triple_pairs : 'a -> 'b * 'c * 'd -> ('a * 'b) * ('a * 'c) * ('a * 'd) =
  <fun>
# triple_pairs 2 (false, 3, true);;
- : (int * bool) * (int * int) * (int * bool) =
((2, false), (2, 3), (2, true))
```

5. (2 pts) Write `triple_xprod : 'a * 'b * 'c -> 'd * 'e -> (('a * 'd) * ('b * 'd) * ('c * 'd)) * (('a * 'e) * ('b * 'e) * ('c * 'e))` that takes a triple and then a pair, and returns a pair of triples where the pairs for the first triple have as their first element the corresponding element from the input triple and the second element is the first element of the input pair, and where the pairs of the second triple have as their first element the corresponding element from the input triple and the second element is the second element of the input pair.

```
# let triple_xprod trp pr = ...
val triple_xprod :
  'a * 'b * 'c ->
  'd * 'e ->
  (('a * 'd) * ('b * 'd) * ('c * 'd)) * (('a * 'e) * ('b * 'e) * ('c * 'e)) =
  <fun>
# triple_xprod (1,true,33.5) ("hi",17);;
- : ((int * string) * (bool * string) * (float * string)) *
    ((int * int) * (bool * int) * (float * int)) =
(((1, "hi"), (true, "hi"), (33.5, "hi")), ((1, 17), (true, 17), (33.5, 17)))
```

6. (2 pts) Write a function `two_funs : ('a -> 'b) * ('c -> 'd) -> 'a * 'c -> 'b * 'd` that takes a pair of 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)
```

7. (2 pts) Write a function `triple_app : ('a -> 'b) * ('c -> 'a) * ('d -> 'c) -> 'd -> 'b` that takes a triple of functions and then an argument as inputs and returns the result of applying the first function in the triple to the result of applying the second function in the triple to the result of applying the third function in the triple to the follow-on argument input.

```
# let triple_app (f,g,h) x = ...
val tripple_app : ('a -> 'b) * ('c -> 'a) * ('d -> 'c) -> 'd -> 'b = <fun>
# triple_app (print_string, string_of_int, (fun n -> n + 4)) 21;;
25- : unit = ()
```

8. (2 pts) Write a function `same_arg_twice : ('a -> 'a -> 'b) -> 'a -> 'b` that takes a function and then an argument as inputs and returns the result of applying the function to the argument and applying the resultant function to the argument a second time.

```
# let same_arg_twice f x = ...
val same_arg_twice : ('a -> 'a -> 'b) -> 'a -> 'b = <fun>
# same_arg_twice (fun s1 -> fun s2 -> string_of_int (s1 + s2 + 12)) 3;;
- : string = "18"
```

9. (2 pts) Write a function `rev_app : 'a -> ('a -> 'b) -> 'b` that takes an argument and then a function as inputs and returns the result of applying the function to the argument.

```
# let rev_app x f = ...
val rev_app : 'a -> ('a -> 'b) -> 'b = <fun>
# rev_app true not;;
- : bool = false
```

10. (2 pts) Write a function `map_triple : ('a -> 'b) -> 'a * 'a * 'a -> 'b * 'b * 'b` that takes a function and then a triple of inputs and returns the triple where each element id the result of applying the function to each corresponding element in the input triple.

```
# let map_triple f (a,b,c) = ...
val map_triple : ('a -> 'b) -> 'a * 'a * 'a -> 'b * 'b * 'b = <fun>
# map_triple float_of_int (1,5,10);;
- : float * float * float = (1., 5., 10.)
```

11. (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
```

12. (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` that, 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
```

13. (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
```

14. (3 pts)The Fibonacci number for any integer less than or equal to 1 is 1 and after that, the n^{th} Fibonacci number is the sum of the two preceding Fibonacci numbers in the sequence, for $n > 1$. Write a function `naive_fibonacci : int -> int` that, when applied to an integer n returns the n^{th} Fibonacci number. Do not worry about be efficient at this time; we will discuss topics related to the efficiency of recursive functions in a the next one to two weeks.

```
# let rec naive_fibonacci n = ...
val naive_fibonacci : int -> int = <fun>
# naive_fibonacci 7;;
- : int = 21
```

15. (3 pts)Write a function `sum_evens_less_eq : int -> int` that returns the sum of all positive even numbers less than or equal to the input argument, and returns 0 of their are no even positive numbers less than or equal to the input argument.

```
# let rec sum_evens_less_eq n = ...
val sum_evens_less_eq : int -> int = <fun>
# sum_evens_less_eq 17;;
- : int = 72
```