Computing Science and Mathematics
CSCU9Y4 Practical 10 Functional Programming in OCaml!

As before, the following online OCaml environments are available for use.

a) https://try.ocamlpro.com
b) https://ocsigen.org/js_of_ocaml/2.7/files/toplevel/index.html
c) https://www.tutorialspoint.com/compile_ocaml_online.php

The confusion over '::' is because **it is not an operator**. It is, instead, a constructor that adds an element to a list. Though it is often called the ":: operator," its true name is "cons." So, the list

```
        # [1;2;3];;
        - : int list = [1; 2; 3]
```

can equivalently be constructed by any of the following:

```
        # 1 :: (2 :: (3 :: [])) ;;
        - : int list = [1; 2; 3]

        # 1 :: 2 :: 3 :: [] ;;
        - : int list = [1; 2; 3]

        # let my_lst = [2;3];;
        - : val my_lst = [2; 3]
        # 1 :: [2;3];;
        - : int list = [1; 2; 3]
```

Both constructions are possible because :: is right-associative, so nested parens to the right are unnecessary. Also keep in mind that [] is the empty list. It is polymorphic, and so is also used to terminate lists. Thus, the structure containing the above list is

```
        +---+---+   +---+---+   +---+---+
        | 1 | *---->| 2 | *---->| 3 | *---->[]
        +---+---+   +---+---+   +---+---+
```

(* On matching *)
Pattern matching in Ocaml can be thought of as similar to the imperative `switch` statements. However, OCaml's type system means that we can match, not only against exact values, but also predicates and other type constructors! For example,

```
        # let is_zero x =
             match x with
             | 0 -> true
             | _ -> false  (* The "_" pattern means "anything else". *)
          ;;
(* Also, match predicates, eg. return the positive of any negative value *)
        # let abs x =
             match x with
             | x when x < 0 -> -x
             | _ -> x
          ;;
```

(* TASK 1 *)
Re-write the following recursive function using pattern matching.

```
        # let rec sum_to x =
             | if x = 0 then 0
             | else x + sum_to (x -1)
          ;;
```

In your new version, it is recommended that all appearances of **x** in the prediates be written as **x'** (single quotes after characters are permitted in identifier names). Why? This is just convention to indicate that the parameter and predicate vars are different.

Note the difference with a *prepended* single quote, eg. **'x** , that means "any type."

(* TASK 2 *)
Imperative switch statements only work when they embed all possible "cases." The elegance of OCaml's inference engine is that it can tell when a case is missing! Consider the function,

```
        (* Remove contiguous duplicates from a list. *)
        # let rec destutter list =
            match list with
            | [] -> []
            | hd1 :: hd2 :: tl ->
               if hd1 = hd2 then destutter (hd2 :: tl)
               else hd1 :: destutter (hd2 :: tl)
          ;;
```

that returns a "not exhaustive" error. Enumerate all possible cases; fix the function!

Checkpoint

Show and explain your solutions to a demonstrator.

(* Higher-order Functions! *)
Just as regular functions take data types as parameters, a higher-order function takes a *function* as an argument. Sometimes, functions can also be returned. Wow!

Yes, this can be done in imperative languages, eg. C's function pointers, though they are complex to write. By contrast, functional languages make their construction elegant. Here is a program with higher-order function that takes two parameters: one **function**, and one list.

```
open Printf;;
print_string "Hello world!\n";;

(* filter returns a new list with f applied to each list element. *)
let rec filter f lst =
    match lst with
    | [] -> []
    | hd :: tl -> (f hd) :: (filter f tl)
    ;;

(* Now call filter with an anonymous function; what is the outcome? *)
filter (fun x -> x * 2) [1; 2; 3; 4; 5];;

(* To see the output, one could type as follows. *)
List.iter (printf "%d ") (filter (fun x -> x * 2) [1; 2; 3; 4; 5]);;
```

It is at this stage that functional power should start to become clear. The function `filter` can be used to apply *any* sequence of operations on each element in a list. WOW!

(* TASK 3 – Bonus/optional *)
Using `filter` as a template, write your own higher-order function `my_filter` that takes

- a boolean function, i.e. it returns true or false), say `f`
- a list, say `lst`

and returns a new list that consists of elements of `lst` such that `f` returns true on that element. In other words,

$$\forall e \in lst, e \in lst' \text{ if } f(e) \text{ returns } true.$$

Use your new function to return only even integers in a list. HINT: The easiest implementation consists of an if/else *inside* of a match.