Programmation Fontionnelle Avancée

Correction de l'examen du Mercredi 10 Janvier 2018, réalisée par un étudiant le soir de l'examen. Des erreurs ont pu s'y glisser.

Exercice 1

```
let l = [[1; 2; 3]; [4]; [5]; [6]; [7]; [8; 9]]

let printexpr l =
  String.concat "+" (List.map (fun l' ->
                         String.concat "*" (List.map string_of_int l')) l)
let x = printexpr l
(* val x : string = "1*2*3+4+5+6+7+8*9" *)

let evalexpr l =
  List.fold_left (fun a l -> a + List.fold_left (fun a i -> a * i) 1 l) 0 l

let x = evalexpr l
(* val x : int = 100 *)

let splits l =
  let rec aux acc last = function
    | [] -> [last :: acc]
    | h :: t -> (aux (last::acc) [h] t) @ (aux acc (h::last) t)
  in
  let l =
    match l with
    | [] -> [[[]]]
    | h :: t -> aux [] [h] t
  in
  List.map (fun l -> List.rev (List.map List.rev l)) l

let x = splits [1; 2; 3]
(* val x : int list list list =
 *   [[[1]; [2]; [3]]; [[1]; [2; 3]]; [[1; 2]; [3]]; [[1; 2; 3]]] *)

let expr = splits [1; 2; 3; 4; 5; 6; 7; 8; 9]
let expr100 = List.filter (fun l -> 100 = evalexpr l) expr
(* val expr100 : int list list list =
 *   [[[1]; [2]; [3]; [4]; [5]; [6]; [7]; [8; 9]];
 *    [[1; 2; 3]; [4]; [5]; [6]; [7]; [8; 9]];
 *    [[1; 2; 3; 4]; [5]; [6]; [7; 8]; [9]]] *)
```

Exercice 2

```
type t1 = [`A | `B]
type t2 = [`A | `B | `C]

let f (x : t2) = function
  | `A -> `B
  | `B -> `C
  | `C -> `A
(* val f : t2 -> [< `A | `B | `C ] -> [> `A | `B | `C ] = <fun> *)

let h x = [ f x; `D]
(*               ^^
 * Error: This expression has type [> `D ]
 *        but an expression was expected of type
 *          [< `A | `B | `C ] -> [> `A | `B | `C ] *)
```

```
let ff x = f (f x)
(*           ^^^^^
 * Error: This expression has type [< `A | `B | `C ] -> [> `A | `B | `C ]
 *        but an expression was expected of type t2 *)

let g (x : t1) = f x
(*                 ^
 * Error: This expression has type t1 but an expression was expected of type t2
 *        The first variant type does not allow tag(s) `C *)

let g (x : t1) = f (x :> t2)
(* val g : t1 -> [< `A | `B | `C ] -> [> `A | `B | `C ] = <fun> *)

let g (x : t1) = ((f x) :> t2)
(*                    ^
 * Error: This expression has type t1 but an expression was expected of type t2
 *        The first variant type does not allow tag(s) `C *)
```

Exercice 3

```
type 'a t =
  | MkB : bool -> bool t
  | MkI : int -> int t

let extract = function
  | MkB b -> b
  | MkI i -> i
(*   ^^^^^
 * Error: This pattern matches values of type int t
 *        but a pattern was expected which matches values of type bool t
 *        Type int is not compatible with type bool *)

 let f i = MkB (MkI i = MkI i)
(* val f : int -> bool t = <fun> *)

let f i = if i < 0 then MkB true else MkI i
(*                                     ^^^^^
 * Error: This expression has type int t but an expression was expected of type
 *          bool t
 *        Type int is not compatible with type bool *)
```

Exercice 4

```
type digit = Zero | One
type real = cell Lazy.t
and cell = Cons of digit * real

let rec infty d = lazy (Cons (d, infty d))
let one = infty One
let zero = infty Zero
let alt =
  let rec alter d = lazy (Cons (d, alter (if d = Zero then One else Zero)))
  in alter One

let rec is_digit d n = function
  | lazy (Cons (d', f)) when d = d' ->
     if n = 1 then true else is_digit d (n-1) f
  | _ -> false
```

```
let is_zero = is_digit Zero
let is_one = is_digit One

let max f1 f2 =
  let rec aux f1' f2' =
    match f1', f2' with
    | lazy (Cons (d1, f1')), lazy (Cons (d2, f2')) ->
       if d1 = d2 then max f1' f2'
       else if d1 = One && d2 = Zero then f1 else f2
  in
  aux f1 f2

let rec syntactic_equal n f1 f2 =
  match f1, f2 with
  | lazy (Cons (d1, f1)), lazy (Cons (d2, f2)) when d1 = d2 ->
     if n = 1 then true else syntactic_equal (n-1) f1 f2
  | _ -> false

(* Série géométrique
 * 1/2^x == sum from i = x+1 to infty of 1/2^i *)
let real_equal n f1 f2 =
  let rec aux n f1 f2 b =
    match f1, f2 with
    | lazy (Cons (d1, f1)), lazy (Cons (d2, f2)) ->
       if b && d1 = Zero && d2 = One then
           if n = 1 then false else aux (n-1) f1 f2 true
       else if d1 = One && d2 = Zero then
         if n = 1 then true else aux (n-1) f1 f2 false
       else false
  in
  match f1, f2 with
  | lazy (Cons (d1, f1)), lazy (Cons (d2, f2)) ->
     if d1 = d2 then syntactic_equal (n-1) f1 f2
     else if d1 = Zero && d2 = One then
       aux (n-1) f1 f2 true
     else aux (n-1) f2 f1 true

let x = real_equal 10
          (lazy (Cons (Zero, one)))
          (lazy (Cons (One, zero)))
        = true
let x = real_equal 10
          (lazy (Cons (Zero, lazy (Cons (Zero, one)))))
          (lazy (Cons (One,  lazy (Cons (One, zero)))))
        = true
let x = real_equal 10
          (lazy (Cons (Zero, lazy (Cons (One, lazy (Cons (Zero, one)))))))
          (lazy (Cons (One, lazy (Cons (Zero, lazy (Cons (One, zero)))))))
        = false
```