TD 4 - Les listes

1 Les incontournables

Exercice 1 [**Tester la présence, trouver l'indice du premier élément**].

1. Écrire une fonction `present : 'a -> 'a list -> bool` qui teste la présence d'un élément dans une liste. (▷ Cette fonction existe déjà sous le nom `mem` dans le module `List`).
2. Écrire une fonction `index : 'a -> 'a list -> int` qui renvoie la position d'un élément dans une liste et déclenche une erreur si l'élément n'est pas présent.

Exercice 2 [**Recherche d'un élément maximal**]

1. Écrire une fonction `max_liste` qui renvoie le plus grand élément d'une liste non vide.
2. L'appel `List.fold_left f a [b1; ...; bn]` renvoie `f (... (f (f a b1) b2) ...) bn`. En utilisant cette fonction écrire une fonction qui retourne le maximum d'une liste non vide.

Exercice 3 [**Tri par insertion**] - On cherche ici à reprogrammer ce tri.

1. Réaliser une fonction `insertion` qui insère un élément à la bonne place dans une liste triée.
2. Réaliser une fonction `tri` qui ordonne une liste avec l'algorithme de tri par insertion.

Exercice 4 [**Suppression**]

1. Écrire une fonction `supprime` de type `int -> 'a list -> 'a list` et qui renvoie une liste où l'élément de rang donné a été supprimé
2. Écrire une fonction `supprime_elt` qui élimine toutes les occurrences d'un élément dans une liste
3. Écrire une fonction `supprime_doublons` qui élimine tous les doublons dans une liste (on ne garde qu'un représentant) ; par exemple : `supprime_doublon [1;2;3;4;1;1;2;5;3] = [1;2;3;4;5]`

Exercice 5 [**Crible d'Eratosthène**]

1. Écrire une fonction `intervalle_entier : int -> int -> int list` telle que l'appel `intervalle a b` renvoie la liste des entiers de `a` à `b`. Par exemple :

```
#intervalle_entier 3 7;;
- : int list = [3; 4; 5; 6; 7]
#intervalle_entier 4 4;;
- : int list = [4]
#intervalle_entier 5 3;;
- : int list = []
```

2. En déduire une fonction `liste_eratosthene : int -> int list` qui permet d'obtenir la liste des nombres premiers de l'intervalle $[1, n]$ en s'appuyant sur la méthode du crible d'Eratosthène. (On pourra avoir recours à la fonction `List.filter`. Par exemple :

```
#liste_eratosthene 20;;
- : int list = [2; 3; 5; 7; 11; 13; 17; 19]
```

2 Pour s'entrainer

Exercice 6 [**Fusion de deux listes triées**] Écrire une fonction `fusion` qui prend en entrée deux listes **triées** dans l'ordre croissant et renvoie la liste fusionnée et triée elle aussi

Exercice 7 [**Filtrer**]

1. La fonction `List.exists : ('a -> bool) -> 'a list -> bool` permet de connaitre la présence ou non d'éléments vérifiant certains critères d'une liste. La reprogrammer sous le nom `existance`. Par exemple :

```
let est_pair x = (x mod 2 = 0);;
est_pair : int -> bool = <fun>
existance est_pair [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
- : true
```

2. La fonction `List.filter : ('a -> bool) -> 'a list -> 'a list` permet de sélectionner les éléments d'une liste vérifiant une certaine condition. La reprogrammer sous le nom `selection`. Par exemple :

```
selection est_pair [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
- : int list = [0; 2; 4; 6; 8; 10]
```

Exercice 8 Écrire une fonction `aplatir : 'a list list -> 'a list` qui transforme une liste de listes en une liste unique (enlève les crochets intérieurs). On souhaite obtenir par exemple :

```
#aplatir [[1;2];[3;4;5];[6;7];[];[8;9;10]];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Exercice 9 [**Ordre lexicographique**]

1. L'ordre lexicographique (large) sur des couples d'entiers relatifs est défini par :

$$(a, b) \preceq (c, d) \iff [(a < c) \text{ ou } ((a = c) \text{ et } (b \leqslant d))]$$

Écrire une fonction `inf_lex` de type `'a * 'b -> 'a * 'b -> bool` telle que par exemple :

```
#inf_lex (-1, 2) (1,-5);;
- : bool = true
#inf_lex (1, 2) (1, 1);;
- : bool = false
```

2. Généraliser à des listes, avec une fonction `inf_lex_list` de type `'a list -> 'a list -> bool`. Par exemple on souhaite :

```
#inf_lex_list [-1;2] [1;-5];;
- : bool = true
#inf_lex_list [1;2] [1;-5];;
- : bool = false
#inf_lex_list [1;2] [1;5];;
- : bool = true
#inf_lex_list [1;2;8;9] [1;-5;7];;
- : bool = false
#inf_lex_list [1] [2;0;1];;
- : bool = true
#inf_lex_list [3] [2;0;1];;
- : bool = false
#inf_lex_list [1] [1;0;1];;
Uncaught exception: Failure "comparaison␣impossible"
```

On pourra réfléchir avant à la manière de définir l'ordre lexicographique de manière récursive.

Correction 1 1. On avance, jusqu'à avoir une liste vide ou trouver l'élément :

```
let rec present x = function
        | [] -> false
        | h::t when h=x -> true
        | h::t -> present x t
;;
```

Que l'on peut écrire en une ligne, sans filtre :

```
let rec present x l =
          (l <> []) && ((List.hd l = x) || (present x (List.tl l)));;
```

2. Une solution récursive non terminale. On convient arbitrairement que le premier index est 0.

```
let rec index a = function
        | [] -> failwith "Elément␣non␣présent"
        | h::t when h=a -> 0
        | h::t -> 1 + index a t
;;
```

Ou une version terminale (noter qu'il n'est pas nécessaire de remettre `a` dans les paramètres la fonction `indexR`)

```
let index a l =
        let rec indexR pos = function
                | [] -> failwith "Elément␣non␣présent"
                | h::t when h=a -> pos
                | h::t -> indexR (pos+1) t
        in indexR 0 l
;;
```

Correction 2 1. Idée 1 : Basée sur le principe que soit la liste ne contient qu'un élément (dans ce cas, c'est le max), soit c'est le plus grand entre la tête et le max de la queue :

```
let rec max_liste = function
          [] -> failwith "La␣liste␣est␣vide␣!!!"
        | [x] -> x
        | h::t when h > max_liste t -> h
        | h::t -> max_liste t
;;
```

Cette solution à pour inconvénient que pour un appel à la fonction `max_liste`, on risque d'avoir 2 appels à `max_liste`. D'ailleurs, elle est équivalente à la version suivante :

```
let rec max_liste = function
          [] -> failwith "La␣liste␣est␣vide␣!!!"
        | [x] -> x
        | h::t -> if h > max_liste t then h else max_liste t
;;
```

Que l'on peut donc améliorer en stockant le résultat :

```
let rec max_liste = function
          [] -> failwith "La␣liste␣est␣vide␣!!!"
        | [x] -> x
        | h::t -> let m = max_liste t in if h > m then h else m
;;
```

Si vous voulez vous convaincre comparez la vitesse d'exécution avec les 2 fonctions de l'appel :

```
max_liste [0;1;2;3;4;5;6;7;8;9;0;1;2;3;4;5;6;7;8;9;0;1;2;3;4;5;6];;
```

Une autre solution (équivalente à la dernière) : (inutile de traiter le cas où la liste est vide, car ce cas est impossible d'après l'énoncé).

```
let rec max_liste = function
        | h::[] -> h
        | h::t -> max h (max_liste t)
;;
```

Enfin une solution récursive terminale (pas nécessaire ici) :

```
let max_liste l =
        let rec maxR m = function
                | [] -> m
                | h::t when h > m -> maxR h t
                | _::t -> maxR m t
        in maxR (List.hd l) (List.tl l)
;;
```

2. En utilisant la fonction présentée :

```
let max_list l = List.fold_left max (List.hd l) l;;
```

Correction 3 1. Insertion d'un élément dans une **liste triée**

```
let rec insertion x = function
        | [] -> [x]
        | h::q as l when x < h -> x::l
        | h::q -> h::insertion x q
;;
```

2. Tri par insertion :

```
let rec tri = function
        | [] -> []
        | h::t -> insertion h (tri t)
;;
```

Correction 4

1. Une solution récessive non terminale :

```
(* Version récursive *)
let rec supprime n l =
        if n=0
        then List.tl l
        else List.hd l::supprime (n-1) (List.tl l)
;;

(* Plus légère *)
let rec supprime n (h::t) =
        if n=0
        then t
        else h::supprime (n-1) t
;;
```

Une solution terminale :

```
(* Version récursive terminale *)
let supprime n l =
        let rec supprimeR acc (t::q) = function
                | 0 -> (acc @ q)
                | n -> supprimeR (acc@[t]) q (n-1)
        in supprimeR [] l n;;
;;
```

2. Suppression de tous les éléments par valeur

```
let rec supprime_elt x = function
        | [] -> []
        | h::q when h=x -> supprime_elt x q
        | h::q -> h::supprime_elt x q
;;
```

3. En utilisant la fonction précédente :

```
let rec supprime_doublon = function
        | [] -> []
        | h::q -> h::supprime_doublon (supprime_elt h q)
;;
```

Correction 5

1. La fonction qui génère l'intervalle d'entiers :

```
let rec intervalle_entier a = function
        | b when b < a -> []
        | b -> a::intervalle_entier (a+1) b
;;
```

2. Dans la documentation page 540, on peut lire (en anglais) que la fonction `List.filter` est de type `('a -> bool) -> 'a list -> 'a list` : elle prend en paramètre une fonction f et une liste ℓ est renvoie la liste extraire de ℓ ne contenant que les éléments x pour lesquels $f(x)$ est vraie. On peut aussi utiliser la fonction réalisée plus loin en exercice 7.

   Pour répondre à la question, on peut donc créer en plusieurs étapes :

```
let non_multiple n x = (x mod n != 0) ;;

let rec eraR = function
        | [] -> []
        | h::t -> h::eraR(List.filter (non_multiple h) t)
;;

let liste_eratosthene n = eraR (intervalle_entier 2 n);;

liste_eratosthene 30;;
```

   Ainsi à chaque appel à `eraR` :
   - On garde la tête,
   - On supprime tous les multiples de la tête dans la queue,
   - On rappelle `eraR` avec la queue

Par exemple sur l'intervalle $[\![2;20]\!]$

eraR $[2 \,\|\, 3;4;5;6;7;8;9;10;11;12;13;14;15;16;17;18;19;20]$

$= [2] ::$ eraR $[3;\not4;5;\not6;7;\not8;9;\not{10};11;\not{12};13;\not{14};15;\not{16};17;\not{18};19;\not{20}]$

$= [2] ::$ eraR $[3 \,\|\, 5;7;9;11;13;15;17;19;]$

$= [2;3] ::$ eraR $[5;7;\not9;11;13;\not{15};17;19]$

$= [2;3] ::$ eraR $[5 \,\|\, 7;11;13;17;19;]$

$= [2;3;5] ::$ eraR $[7 \,\|\, 11;13;17;19;]$

$= [2;3;5;7] ::$ eraR $[11 \,\|\, 13;17;19;]$

$= [2;3;5;7;11] ::$ eraR $[13 \,\|\, 17;19;]$

$= [2;3;5;7;11;13] ::$ eraR $[17 \,\|\, 19;]$

$= [2;3;5;7;11;13;17] ::$ eraR $[19 \,\|\,]$

$= [2;3;5;7;11;13;17;19] ::$ eraR $[\,]$

$= [2;3;5;7;11;13;17;19]$

En fin,on peut aussi tout placer dans une unique fonction :

```
let liste_eratosthene n =
        let rec non_multiple n x = (x mod n != 0) and
                 eraR = function
                        | [] -> []
                        | h::t -> h::eraR(List.filter (non_multiple h) t)
        in eraR (intervalle_entier 2 n)
;;
```

Correction 6 Une solution :

```
let rec fusion l1 l2 =
  match l1, l2 with
  | [],l2 -> l2
  | l1,[] -> l1
  | h1::t1, (h2::t2 as l2) when h1 < h2 -> h1::fusion t1 l2
  | l1, h2::t2 -> h2::fusion l1 t2
;;
```

Noter la présence du mot clé `as` qui permet de nommer la liste pour ne pas avoir à réécrire `h2::q2`

Correction 7

1. Cela ressemble à la fonction `present`

```
let rec existance f = function
        | [] -> false
        | h::t when f(h) -> true
        | h::t -> existance f t
;;
```

2. Une solution :

```
let rec selection f = function
        | [] -> []
        | h::q when f h -> h::selection f q
        | h::q -> selection f q
;;
```

Correction 8 Première solution :

```
let rec aplatir = function
        | [] -> []
        | []::t -> aplatir t
        | h::t -> (List.hd h)::aplatir ((List.tl h)::t)
;;
```

ou en utilisant l'opérateur @ :

```
let rec aplatir = function
        | [] -> []
        | h::t -> h @ aplatir t
;;
```

Correction 9 Les 2 questions :

```
let inf_lex (a,b) (c,d) = a < c || (a = c && b <= d);;

let rec inf_lex_list l1 l2 =
    match (l1,l2) with
        | [],[] -> true
        | [],_ -> failwith "Comparaison␣impossible"
        | _ ,[] -> failwith "Comparaison␣impossible"
        | (h1::t1), (h2::t2) -> h1 < h2 || (h1 = h2 && (inf_lex_list t1 t2))
;;
```