[TP 6] Récursivité et listes.

Objectifs : fonctions récursives sur les listes.

1 Préliminaires

Dans le dossier BPF, créez un répertoire TP6. Les exercices de ce TP sont à réaliser dans ce répertoire.

Dans la suite, il est demandé de développer des fonctions sur des paramètres de type liste. Ces fonctions devront être récursives terminales ou faire appel à des fonctions qui le sont. Vous pouvez utiliser toute fonction du module `List` lorsque cela est justifié. Vous pouvez également utiliser des fonctions des exercices précédents pour répondre à une question.

N'oubliez pas de tester les fonctions au fur et à mesure : pour cela, utilisez (par exemple) la fonction suivante permettant de générer simplement des listes d'entiers :

```
let (--) x y =
  if x <= y
  then List.init (y - x + 1) ((+) x)
  else List.init (x - y + 1) ((-) x);;
```

Par exemple :

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

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

2 Exercices

Exercice 1 (Préfixes et suffixes)
Considérons les exemples suivants :

```
# is_prefix (1 -- 10) (1 -- 15);;
- : bool = true

# is_prefix (1 -- 10) (-10 -- 10);;
- : bool = false

# is_suffix (1 -- 10) (1 -- 15);;
- : bool = false

# is_suffix (1 -- 10) (-10 -- 10);;
- : bool = true

# strip_prefix (1 -- 10) (1 -- 15);;
- : int list = [11; 12; 13; 14; 15]

# strip_prefix (1 -- 10) (-10 -- 10);;
Exception: Failure "xs must be a prefix of ys".

# strip_prefix_opt (1 -- 10) (1 -- 15);;
- : int list option = Some [11; 12; 13; 14; 15]

# strip_prefix_opt (1 -- 10) (-10 -- 10);;
- : int list option = None
```

Écrivez les fonctions suivantes :

1. `is_prefix`, de type `'a list -> 'a list -> bool`, où `is_prefix xs ys` renvoie vrai si et seulement si `xs` est préfixe de `ys` ;
2. `is_suffix`, de type `'a list -> 'a list -> bool`, où `is_suffix xs ys` renvoie vrai si et seulement si `xs` est suffixe de `ys` ;
3. `strip_prefix` de type `'a list -> 'a list -> 'a list`, où `strip_prefix xs ys` renvoie `zs` si `ys = xs @ zs`, une exception sinon ;;
4. `strip_prefix_opt` de type `'a list -> 'a list -> 'a list option`, où `strip_prefix_opt xs ys` renvoie `Some zs` si `ys = xs @ zs`, `None` sinon.

Exercice 2 (Recherches)
Considérons les exemples suivants :

```
# let even x = x mod 2 = 0;;
val even : int -> bool = <fun>

# find_index even (1 -- 10);;
- : int = 1

# find_index even (1 -- 1);;
Exception: Failure "element not found".

# find_index_opt even (1 -- 10);;
- : int option = Some 1

# find_index_opt even (1 -- 1);;
- : int option = None

# find_indices even (1 -- 10);;
- : int list = [1; 3; 5; 7; 9]

# elem_index 10 (5 -- 15);;
- : int = 5

# elem_index 10 (1 -- 1);;
Exception: Failure "element not found".

# elem_index_opt 10 (5 -- 15);;
- : int option = Some 5

# elem_index_opt 10 (1 -- 1);;
- : int option = None

# elem_indices 10 (1 -- 10 @ 10 -- 1);;
- : int list = [9; 10]
```

Écrivez les fonctions suivantes :

1. `find_index` de type `('a -> bool) -> 'a list -> int`, où `find_index f xs` renvoie l'indice du premier élément `x` de la liste `xs` tel que `f x` soit vrai si un tel élément existe, une exception sinon ;
2. `find_index_opt` de type `('a -> bool) -> 'a list -> int option`, où `find_index_opt f xs` renvoie `Some i` où `i` est l'indice du premier élément `x` de la liste `xs` tel que `f x` soit vrai si un tel élément existe, `None` sinon ;
3. `find_indices` de type `('a -> bool) -> 'a list -> int list`, où `find_indices f xs` renvoie la liste des indices des éléments `x` de la liste `xs` tel que `f x` soit vrai.

Utilisez ensuite ces fonctions pour rechercher le ou les indices d'un élément donné dans une liste dans des fonctions `elem_{index, index_opt, indices}`.

Exercice 3 (Sous-listes)
Considérons les exemples suivants :

```
# drop 5 (1 -- 10);;
- : int list = [6; 7; 8; 9; 10]

# drop 5 (1 -- 3);;
- : int list = []

# tails (1 -- 5);;
- : int list list =
[[1; 2; 3; 4; 5]; [2; 3; 4; 5]; [3; 4; 5]; [4; 5]; [5]; []]

# inits (1 -- 5);;
- : int list list =
[[]; [1]; [1; 2]; [1; 2; 3]; [1; 2; 3; 4]; [1; 2; 3; 4; 5]]

# is_infix (5 -- 10) (1 -- 15);;
- : bool = true
```

Écrivez les fonctions suivantes :

1. `drop` de type `int -> 'a list -> 'a list`, où `drop n xs` renvoie la liste `xs` privée de ses `n` premiers éléments ;
2. `tails` de type `'a list -> 'a list list`, où `tails xs` renvoie la liste des listes suffixes de `xs` ;
3. `inits` de type `'a list -> 'a list list`, où `inits xs` renvoie la liste des listes préfixes de `xs`.

Utilisez ces fonctions pour définir la fonction `is_infix` où `is_infix xs ys` renvoie vrai si et seulement si `xs` apparait dans `ys`.

Exercice 4 (Suppression)
Considérons les exemples suivants :

```
# let equiv_mod_3 x y = x mod 3 = y mod 3;;
val equiv_mod_3 : int -> int -> bool = <fun>

# delete_by equiv_mod_3 6 (1 -- 10);;
- : int list = [1; 2; 4; 5; 6; 7; 8; 9; 10]

# delete 6 (1 -- 10);;
- : int list = [1; 2; 3; 4; 5; 7; 8; 9; 10]

# diff_by equiv_mod_3 (1 -- 10) (5 -- 15 @ 1 -- 10);;
- : int list = [14; 15; 2; 3; 4; 5; 6; 7; 8; 9; 10]

# diff (1 -- 10) (5 -- 15 @ 1 -- 10);;
- : int list = [11; 12; 13; 14; 15; 5; 6; 7; 8; 9; 10]
```

Écrivez les fonctions suivantes :

1. `delete_by` de type `('a -> 'b -> bool) -> 'a -> 'b list -> 'b list`, où `delete_by cp x ys` renvoie la liste `xs` privée du premier élément `y` tel que `cp x y` soit vrai ;
2. `delete` de type `'a -> 'a list -> 'a list`, où `delete x xs` renvoie la liste `xs` privée de la première occurrence de `x` ;
3. `diff_by` de type `('a -> 'b -> bool) -> 'a list -> 'b list -> 'b list` telle que `diff_by cp [x1; ...; xn] ys` retire successivement la première occurrence des éléments `(z1, ..., zn)` satisfaisant `cp xi zi` dans `ys` ;
4. `diff` de type `'a list -> 'a list -> 'a list` telle que `diff [x1; ...; xn] ys` retire successivement la première occurrence des éléments `(x1, ..., xn)` dans `ys`.

Exercice 5 (Union et intersection)
Considérons les exemples suivants :

```
# let equiv_mod_5 x y = x mod 5 = y mod 5;;
val equiv_mod_5 : int -> int -> bool = <fun>

# nub_by equiv_mod_5 (1 -- 100);;
- : int list = [1; 2; 3; 4; 5]

# nub (1 -- 10 @ 5 -- 15);;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15]

# union_by equiv_mod_5 (1 -- 10) (5 -- 15);;
- : int list = [1; 2; 3; 4; 5]

# union (1 -- 10) (5 -- 15);;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15]

# intersect_by equiv_mod_5 (1 -- 10) (5 -- 15);;
- : int list = [1; 2; 3; 4; 5]

# intersect (1 -- 10) (5 -- 15);;
- : int list = [5; 6; 7; 8; 9; 10]
```

Écrivez les fonctions suivantes :

1. `nub_by` de type `('a -> 'a -> bool) -> 'a list -> 'a list` telle que `nub_by cp xs` ne conserve que les éléments de `xs` tous différents (selon la fonction `cp`), c'est-à-dire que la liste obtenue ne contient pas deux éléments `x1` et `x2` tels que `cp x1 x2` soit vrai ;
2. `nub` de type `'a list -> 'a list` telle que `nub xs` ne conserve que les éléments de `xs` tous différents (selon la fonction `compare`) ;
3. `union_by` (resp. `intersection_by`) de type `('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list` telle que `union_by cp xs ys` (resp. `union_by cp xs ys`) renvoie l'union (resp. l'intersection) des listes `xs` et `ys`, c'est-à-dire contenant les éléments de `xs` ou (resp. et) `ys` différents au sens de `cp` ;
4. `union` (resp. `intersection`) de type `'a list -> 'a list -> 'a list` telle que `union xs ys` (`intersection xs ys`) renvoie l'union (resp. l'intersection) des listes `xs` et `ys`.

Exercice 6 (Intercalations)
Considérons les exemples suivants :

```
# prepend_to_all 0 (1 -- 5);;
- : int list = [0; 1; 0; 2; 0; 3; 0; 4; 0; 5]

# intersperse 0 (1 -- 5);;
- : int list = [1; 0; 2; 0; 3; 0; 4; 0; 5]

# intercalate [0; 0] [1 -- 2; -1 -- -2; 1 -- 3];;
- : int list = [1; 2; 0; 0; -1; -2; 0; 0; 1; 2; 3]
```

Écrivez les fonctions suivantes :

1. `prepend_to_all` de type `'a -> 'a list -> 'a list` telle que `prepend_to_all x xs` renvoie la liste `xs` en ajoutant l'élément `x` devant chacun de ses éléments ;
2. `intersperse` de type `'a -> 'a list -> 'a list` telle que `intersperse x xs` renvoie la liste `xs` en ajoutant l'élément `x` devant chacun de ses éléments, sauf le premier ;
3. `intercalate` de type `'a list -> 'a list list -> 'a list` où `intercalate xs xss` intercale la liste `xs` devant toutes les listes de `xss` (sauf la première) et concatène le résultat.

Exercice 7 (Tris)
Considérons les exemples suivants :

```
# let even_before x y =
  match x mod 2, y mod 2 with
    | 0, 0
    | 1, 1 -> compare x y
    | 0, _ -> -1
    | _, _ -> 1;;
val even_before : int -> int -> int = <fun>

# let inv = ( * ) (-1);;
val inv : int -> int = <fun>

# sort_on_by even_before inv (-1 -- -10);;
- : int list = [-2; -4; -6; -8; -10; -1; -3; -5; -7; -9]

# sort_on inv (-1 -- -10);;
- : int list = [-1; -2; -3; -4; -5; -6; -7; -8; -9; -10]

# merge_two_by_two_by even_before [[2;4];[3;5];[5;7];[2;11]];;
- : int list list = [[2; 4; 3; 5]; [2; 5; 7; 11]]

# merge_sort_by even_before (10 -- 1);;
- : int list = [2; 4; 6; 8; 10; 1; 3; 5; 7; 9]

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

Écrivez les fonctions suivantes :

1. `sort_on_by` de type `('a -> 'a -> int) -> ('b -> 'a) -> 'b list -> 'b list` où `sort_on_by cp f xs` trie la liste `xs` en fonction de l'ordre défini par `cp` sur les images des éléments de `xs` par `f`;
2. `sort_on` de type `('b -> 'a) -> 'b list -> 'b list` où `sort_on f xs` trie la liste `xs` en fonction des images de ses éléments par `f`;
3. `merge_two_by_two_by` de type `('a -> 'a -> int) -> 'a list list -> 'a list list` où `merge_two_by_two_by cp xss` fusionne les éléments de `xss` deux par deux; c'est-à-dire que `merge_two_by_two_by cp [x1; x2; x3; x4; x5] = [y1; y2; y3]` où `y1` est le résultat de la fusion des listes `x1` et `x2`, `y2` est le résultat de la fusion des listes `x3` et `x4`, et `y3 = x5`;
4. `merge_sort_by` de type `('a -> 'a -> int) -> 'a list -> 'a list` où `merge_sort_by xp xs` trie `xs` en fonction de l'ordre défini par `cp`;
5. `merge_sort` de type `'a list -> 'a list` où `merge_sort xs` trie `xs`.

Exercice 8 (Sous-séquences et permutations)
Considérons les exemples suivants :

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

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

Écrivez les fonctions suivantes :

1. `subsequences` de type `'a list -> 'a list list` renvoyant la liste des sous-séquences d'une liste donnée ;
2. `permutations` de type `'a list -> 'a list list` renvoyant la liste des permutations d'une liste donnée.

Exercice 9 (Regroupement et reproduction)
Considérons les exemples suivants :

```
# let equiv_mod_2 x y = x mod 2 = y mod 2;;
val equiv_mod_2 : int -> int -> bool = <fun>

# group_by equiv_mod_2 [1; 2; 2; 4; 5; 5; 5; 7; 8];;
- : int list list = [[1]; [4; 2; 2]; [7; 5; 5; 5]; [8]]

# group [1; 2; 2; 4; 5; 5; 5; 7; 8];;
- : int list list = [[1]; [2; 2]; [4]; [5; 5; 5]; [7]; [8]]

# replicate 5 2;;
- : int list = [2; 2; 2; 2; 2]
```

Écrivez les fonctions suivantes :

1. `group_by` de type `('a -> 'a -> bool) -> 'a list -> 'a list list` où `group_by cp xs` est la liste des listes d'éléments consécutifs égaux selon `cp` de `xs` ; par exemple `group_by (fun x y -> x mod 3 = y mod 3) [1;2;5;8;1;4;0;3] = [[1]; [8; 5; 2]; [4; 1]; [3; 0]];`
2. `group` de type `'a list -> 'a list list` où `group xs` est la liste des listes d'éléments consécutifs égaux de `xs` ;
3. `replicate` de type `int -> 'a -> 'a list` où `replicate n x` est la liste contenant `n` fois la valeur `x`.

Exercice 10 (Run Length Encoding)
Utilisez les fonctions de l'exercice précédent pour réaliser l'encodage et le décodage de liste selon la méthode *RLE*, conformément à l'exemple suivant :

```
# rle_encode ['a'; 'a'; 'a'; 'c'; 'c'; 'b'];;
- : (int * char) list = [(3, 'a'); (2, 'c'); (1, 'b')]

# rle_decode [(5, true); (4, false)];;
- : bool list = [true; true; true; true; true; false; false; false; false]
```

Pour cela, réalisez les deux fonctions suivantes :

1. `rle_decode` de type `(int * 'a) list -> 'a list` transformant une liste de couples *(nombre d'occurrences, élément)* en la liste correspondante *développée* ;
2. `rle_encode` de type `'a list -> (int * 'a) list` réalisant l'inverse.

Exercice 11 (Produit matriciel)
Considérons les exemples suivants :

```
# (* m1 =   *)
(*  1 2 3 *)
(*  4 5 6 *)
(*  5 8 9 *)
let m1 = [[1; 2; 3];[4; 5; 6];[7; 8; 9]];;
val m1 : int list list = [[1; 2; 3]; [4; 5; 6]; [7; 8; 9]]

# (* m2 =          *)
(*  1  2  3  4 *)
(*  5  6  7  8 *)
(*  9 10 11 12 *)
let m2 = [[1; 2; 3; 4];[5; 6; 7; 8];[9; 10; 11; 12]];;
val m2 : int list list = [[1; 2; 3; 4]; [5; 6; 7; 8]; [9; 10; 11; 12]]

# (* m3 =        *)
(*   1  2  3 *)
(*   4  5  6 *)
(*   7  8  9 *)
(*  10 11 12 *)
let m3 = [[1; 2; 3];[4; 5; 6];[7; 8; 9];[10; 11; 12]];;;
val m3 : int list list = [[1; 2; 3]; [4; 5; 6]; [7; 8; 9]; [10; 11; 12]]

# transpose m1;;
- : int list list = [[1; 4; 7]; [2; 5; 8]; [3; 6; 9]]

# transpose m2;;
- : int list list = [[1; 5; 9]; [2; 6; 10]; [3; 7; 11]; [4; 8; 12]]

# transpose m3;;
- : int list list = [[1; 4; 7; 10]; [2; 5; 8; 11]; [3; 6; 9; 12]]

# prod_mat m1 m1;;
- : int list list = [[30; 36; 42]; [66; 81; 96]; [102; 126; 150]]

# prod_mat m2 m3;;
- : int list list = [[70; 80; 90]; [158; 184; 210]; [246; 288; 330]]

# prod_mat m3 m2;;
- : int list list =
[[38; 44; 50; 56]; [83; 98; 113; 128];
 [128; 152; 176; 200]; [173; 206; 239; 272]]

# prod_mat m1 m3;;
Exception: Invalid_argument "List.combine".
```

Écrivez la fonction `transpose` de type `'a list list -> 'a list list` transposant une matrice donnée comme une liste de liste.
Utilisez la fonction `List.combine` pour réaliser une fonction calculant le produit de deux matrices à coefficients entiers et de dimensions compatibles.