Feuille d'exercices n°3

Cette feuille d'exercice porte sur les listes et leur filtrage, ainsi que quelques fonctions récursives simples pour construire ou explorer des listes.

Exercice I : Filtrages sur les listes

Q1 – Définir la fonction de signature

```
len_comp_3 (xs:'a list) : int
```

qui donne

1	si `xs` contient au moins 3 éléments
0	si `xs` contient exactement 3 éléments
-1	sinon.

Q2 – Définir la fonction de signature

```
swap_hd_snd (xs:'a list) : 'a list
```

qui donne la liste obtenue en inversant l'ordre des deux premiers éléments de `xs`. Par exemple: `(swap_hd_snd [1;2;3;4])` donne la liste `[2;1;3;4]`.

La fonction `swap_hd_snd` renvoie `xs` inchangée si elle contient moins de 2 éléments.

Q3 – Définir la fonction de signature

```
swap_hd_fst (xs:('a*'a) list) : ('a*'a) list
```

qui donne la liste obtenue en inversant l'ordre des valeurs dans le premier couple de `xs`. On aura que `(swap_hd_fst [])` = `[]`.

Exemple:
`(swap_hd_fst [ (1,2); (3,4); (5,5) ])` donne la liste `[ (2,1); (3,4); (5,5) ]`.

Exercice II : Constructions de liste avec récursion sur entiers

Q1 – Définir la fonction de signature

```
repeat (n:int) (x:'a) : 'a list
```

qui donne la liste contenant `n` fois `x`.

Exemples:
`(repeat 7 true)` donne `[true; true; true; true; true; true; true]`

`(repeat 0 "Hello")` donne `[]`
`(repeat (-42) [1;2;3])` donne `[]`

Q2 – Définir la fonction

```
range_i (i:int) (j:int) : (int list)
```

qui donne la liste `[i; i+1; ..; j]`.
Remarque: si `i` > `j` alors `(range i j)` donne la liste vide.

Exercice III : Manipulation de listes avec récursion sur listes

Q1 – Définir la fonction de signature

```
intercale (z:'a) (xs:'a list) : 'a list
```

qui intercale `z` entre les éléments de `xs`. Le premier et le dernier éléments de `(intercale z xs)` sont le premier et le dernier éléments de `xs`.

Exemples:
`(intercale 1 [])` donne `[]`
`(intercale 0 [5])` donne `[5]`
`(intrecale 0 [1;2;3])` donne `[1;0;2;0;3]`

Q2 – Définir la fonction de signature

```
begaie (xs:'a list) : ('a list)
```

qui double la taille d'une liste en dupliquant chacun de ses éléments. Exemples:
`(begaie [])` donne `[]`
`(begaie [1;2;3])` donne `[1;1;2;2;3;3]`

Q3 – Définir la fonction de signature

```
add_list (xs:int list) (ys:int list) : int list
```

qui donne la liste obtenue en additionnant terme à terme les éléments de `xs` et de `ys`. Lorsque `xs` et `ys` ne sont pas de même longueur, on conserve tels quels les éléments en plus.

Exemples:
`(add_list [] [1;2;3])` vaut `[1;2;3]`
`(add_list [1;2;3] [])` vaut `[1;2;3]`
`(add_list [1;2;3] [4;5;6])` vaut `[5;7;9]`
`(add_list [1;2;3] [4;5])` vaut `[5;7;3]`
`(add_list [1;2] [4;5;6])` vaut `[5;7;6]`

Exercice IV : Récréation - À faire à la maison: suite de listes

On souhaite définir une fonction qui construit, étant donné un entier $k > 0$, la liste L_k définie comme suit :

$$L_1 = [1] \quad L_2 = [1;1] \quad L_3 = [2;1] \quad L_4 = [1;2;1;1] \quad L_5 = [1;1;1;2;2;1] \quad L_6 = [3;1;2;2;1;1] \quad \cdots$$

La liste L_k s'obtient en «lisant» le contenu de la liste L_{k-1}. Par exemple, L_5 s'obtient en lisant la liste L_4 : 1 (occurrence du chiffre) `1` suivi de 1 (occurrence du chiffre) `2` suivi de 2 (occurrences du chiffre) `1`. Ce qui donne bien la liste `[1; 1; 1; 2; 2; 1]`

Q1 – Définir la fonction de signature

```
lg_prefix (xs:'a list) : int
```

telle que `(lg_prefix xs)` donne le nombres d'élements identiques en début de la liste `xs`.

Exemples:

`(lg_prefix [])` vaut 0

`(lg_prefix [4])` vaut 1

`(lg_prefix [4; 5; 4])` vaut 1

`(lg_prefix [4; 4; 5; 4])` vaut 2

Q2 – Définir la fonction de signature

```
rm_pref (xs:'a list) (n:int) : 'a list
```

qui donne la liste obtenue en «supprimant» les `n` premiers éléments de `xs`. Si `n` est plus grand que la longueur de la liste, le résultat est la liste vide. Si `n` est négatif ou nul, le résultat est la liste `xs`.

Exemples:

`(rm_pref 0 [1;2;3;4;5])` vaut `[1;2;3;4;5]`

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

Q3 – Définir une fonction `next_list` qui construit L_k à partir de L_{k-1}. On supposera, pour simplifier, que tous les entiers de la liste L_k sont compris entre 0 et 9.

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

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

Q4 – Définir une fonction `make_list_k` qui étant donné un entier k strictement positif construit la liste L_k. On suppose k strictement positif.

```
# (make_list_k 1);;
- : int list = [1]
# (make_list_k 2);;
- : int list = [1; 1]
# (make_list_k 6);;
- : int list = [3; 1; 2; 2; 1; 1]
```