Programmation fonctionnelle

Fiche de TP 5

L3 Informatique 2020-2021

Fonctions d'ordre supérieur

Les fonctions d'ordre supérieur sont des fonctions paramétrées par une ou plusieurs fonctions et/ou qui renvoient une fonction. Le but de ce TP est d'utiliser et d'écrire de telles fonctions.

Exercice 1. (Premières fonctionnelles et curryfication)

Dans cet exercice, on utilisera les deux liaisons

```
# let sqr x = x * x;;
# let my_list = [3; 12; 3; 40; 6; 4; 6; 0];;
```

1. Exprimier le type de la fonction `f_sum` paramétrée par une fonction f ainsi que deux entiers a et b et qui calcule $f(a) + f(b)$.
2. Écrire la fonction `f_sum`.

```
# f_sum sqr 2 3;;
- : int = 13

# f_sum (fun x -> x + 1) 2 3;;
- : int = 7
```

3. Curryfier (explicitement) la fonction `f_sum`.

   En d'autres termes, créer une fonction `new_f_sum` à un paramètre fonction f de type `int -> int` et un paramètre a de type `int`, et renvoyant une fonction à un paramètre b calculant $f(a) + f(b)$.

```
# new_f_sum sqr 2;;
- : int -> int = <fun>

# (new_f_sum sqr 2) 3;;
- : int = 13
```

4. Lier aux noms `f1`, `f2`, `f3`, `f4` et `f5` des définitions de fonctions de sorte à vérifier les types suivants.

```
val f1 : int -> int -> int = <fun>
val f2 : (int -> int) -> int = <fun>
val f3 : (int -> int) -> int -> int = <fun>
val f4 : (int -> int) -> (int -> int) = <fun>
val f5 : ((int -> int) -> int) -> int = <fun>
```

5. Sans définir de nouvelle fonction et en utilisant la fonction `sqr` de manière appropriée, créer la liste des carrés des éléments de `my_list`. Utiliser la fonction `List.map`.

```
- : int list = [9; 144; 9; 1600; 36; 16; 36; 0]
```

6. Sans définir de nouvelle fonction et en utilisant la multiplication (attention, il y a une astuce pour transformer l'opérateur `*` en fonction) de manière appropriée, créer la liste des doubles des éléments de `my_list`.

```
- : int list = [6; 24; 6; 80; 12; 8; 12; 0]
```

7. Exprimer le type de la fonction `make_list` paramétrée par un entier n et une fonction `make` (dont l'unique paramètre est de type `unit`) et qui renvoie une liste de n éléments fabriqués avec la fonction `make`.
8. Écrire la fonction `make_list`.

```
# let f () = 0;;
val f : unit -> int = <fun>
# make_list f 8;;
- : int list = [0; 0; 0; 0; 0; 0; 0; 0]
```

9. En utilisant `make_list` et la fonction appropriée du module `Random`, créer une liste de 64 booléens générés aléatoirement.
10. En utilisant `make_list` et une fonction anonyme, créer une liste de 16 entiers (compris entre 0 et 100) générés aléatoirement.

Exercice 2. (Sur une ligne)

Dans cet exercice, on utilisera les liaisons

```
let entiers = [2; 5; 7; 3; 12; 4; 9; 2; 11];;
let animaux = ["Wombat"; "aXolotl"; "pangolin"; "suricate"; "paresseuX"; "quoKKa"; "lemurien"];;
```

Toutes les expressions demandées par les descriptions suivantes doivent être sous la forme d'appels de fonctions d'ordre supérieur du module `List` (comme `List.map`, `List.for_all`, `List.filter`, `List.fold_left`, *etc.*). Il est interdit de définir de nouvelles fonctions non anonymes.

Écrire de telles expressions pour

1. calculer la liste des longueurs des chaînes de caractères de la liste `animaux` ;

```
- : int list = [6; 7; 8; 8; 9; 6; 8]
```

2. calculer la liste de tous les animaux de la liste `animaux` écrits en majuscules ;

```
- : string list = ["WOMBAT"; "AXOLOTL"; "PANGOLIN"; "SURICATE"; ...]
```

3. calculer la liste des chaînes de la liste `animaux` qui sont écrites en minuscules ;

```
- : String.t list = ["pangolin"; "suricate"; "lemurien"]
```

4. calculer la liste des noms des `animaux` de longueurs paires ;

```
- : string list = ["Wombat"; "pangolin"; "suricate"; "quoKKa"; "lemurien"]
```

5. calculer la liste suivante, construite à partir de la liste `entiers` ;

```
- : (int * string) list = [(2, "pair"); (5, "impair"); (7, "impair");
   (3, "impair"); (12, "pair"); (4, "pair"); (9, "impair"); (2, "pair");
   (11, "impair")]
```

6. calculer une liste de listes composées de n fois la valeur de n, pour chaque entier n dans la liste `entiers` ;

```
- : int list list = [[2; 2]; [5; 5; 5; 5; 5]; [7; 7; 7; 7; 7; 7; 7]; [3; 3; 3];
   [12; 12; 12; 12; 12; 12; 12; 12; 12; 12; 12; 12]; [4; 4; 4; 4]; ... ]
```

7. tester s'il existe une élément de la liste `animaux` qui commence par le caractère `'s'` ;
8. tester si tous les éléments de la liste `animaux` sont de longueur congrue à 2 modulo 5.

Note : penser à consulter la documentation des modules `List`, `Char` *et* `String`.

Exercice 3. (Pliage)

À la manière de l'exercice 2 écrire les fonctions suivantes de sorte à avoir un corps de fonction qui tienne sur une seule ligne et utilise des fonctions d'ordre supérieur :

1. `(sum l)` qui renvoie la somme des éléments d'une liste d'entiers `l` ;

2. `(size l)` qui renvoie la longueur d'une liste `l` ;
3. `(last l)` qui renvoie le dernier élément d'une liste `l` non vide ;
4. `(nb_occ e l)` qui renvoie le nombre d'occurrences d'un élément `e` dans `l` ;
5. `(max_list l)` qui renvoie le maximum de la liste `l` non vide ;
6. `(average l)` qui renvoie la moyenne (de type `float`) des nombres entiers de la liste `l`. Il est interdit d'utiliser `size` et `sum`.

Rappel : la plupart de ces fonctions ont été implantées sans utiliser de fonctions d'ordre supérieur lors du TP 3.

Exercice 4. (Prédicats sur les listes)

Un prédicat est une fonction qui teste si une propriété est vérifiée par son paramètre (et renvoie donc un booléen). Par exemple :

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

# is_even 0, is_even 1, is_even 2;;
- : bool * bool * bool = (true, false, true)
```

1. Écrire la fonction récursive `my_for_all` paramétrée par un prédicat `p` et une liste `l` et qui teste si tous les éléments de `l` vérifient `p`.
2. Écrire la fonction `my_for_all2` qui fait la même chose que `my_for_all` mais utilise `fold_left`.
3. Si ce n'est pas déjà fait, réécrire la fonction `my_for_all2` sans utiliser `map`.
4. Écrire la fonction `my_for_all3` qui fait la même chose que `my_for_all` mais utilise `fold_right`.
5. Écrire la fonction `my_exists` paramétrée par un prédicat `p` et une liste `l` et qui teste s'il existe un élément de `l` vérifiant `p`.
6. Écrire la fonction récursive `none` paramétrée par un prédicat `p` et une liste `l` et qui teste si aucun des éléments de la liste ne vérifie `p`.
7. Écrire la fonction `not_all` paramétrée par un prédicat `p` et une liste `l` et qui teste s'il existe un élément de `l` ne vérifiant pas `p`.
8. Écrire la fonction `ordered` paramétrée par un prédicat binaire `p` et une liste $[a_1; \ldots; a_n]$ et qui teste si toutes les expressions `(p` a_i a_{i+1}`)` sont vraies pour tout $1 \leqslant i \leqslant n-1$.

```
# ordered (<) [1; 2; 3];;
- : bool = true

# ordered (<) [1; 4; 3];;
- : bool = false
```

```
# ordered (fun x y -> x + y >= 1) [1; 4; -3; 6];;
- : bool = true

# ordered (fun x y -> x + y >= 1) [1; 4; -5; 6];;
- : bool = false
```

9. Écrire la fonction `filter2` paramétrée par un prédicat binaire `p` et deux listes $[a_1;\ldots;a_n]$ et $[b_1;\ldots;b_n]$ et qui renvoie la liste des couples (a_i, b_i) tels `(p` a_i b_i`)` est vrai.

```
# filter2 (<) [2; 2; 3] [1; 4; 5];;
- : (int * int) list = [(2, 4); (3, 5)]
```

Exercice 5. ★ (Génération exhaustive de permutations)

Une *permutation* d'une liste `l` est une liste `l'` qui contient les mêmes éléments que ceux de `l` mais dans un ordre potentiellement différent.

Écrire une fonction `perm l` qui renvoie la liste de toutes les permutations de la liste `l`.

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

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

# perm [1;2;3;4];;
- : int list list =
[[1; 2; 3; 4]; [2; 1; 3; 4]; [2; 3; 1; 4]; [2; 3; 4; 1]; [1; 3; 2; 4];
 [3; 1; 2; 4]; [3; 2; 1; 4]; [3; 2; 4; 1]; [1; 3; 4; 2]; [3; 1; 4; 2];
 [3; 4; 1; 2]; [3; 4; 2; 1]; [1; 2; 4; 3]; [2; 1; 4; 3]; [2; 4; 1; 3];
 [2; 4; 3; 1]; [1; 4; 2; 3]; [4; 1; 2; 3]; [4; 2; 1; 3]; [4; 2; 3; 1];
 [1; 4; 3; 2]; [4; 1; 3; 2]; [4; 3; 1; 2]; [4; 3; 2; 1]]
```

Exercice 6. (Sur les arbres)

Dans cet exercice, nous réutiliserons les définitions de la fiche de TP 4 pour le type `bintree` servant à représenter des arbres binaires étiquetés par des entiers.

```
type bintree = Empty | Node of int * bintree * bintree;;

let example_tree =
   Node(1,
       Node(2,
           Node(4, Empty, Empty),
           Node(5, Node(7, Empty, Empty), Node(8, Empty, Empty))),
       Node(3, Empty, Node(6, Node(9, Empty, Empty), Empty)));;
```

1. Écrire la fonction `map_tree` dont le fonctionnement généralise celui de `map` pour les arbres binaires.

```
# tree_map (fun x -> x * 2) example_tree;;
- : bintree =
  Node(2,
      Node(4,
          Node(8, Empty, Empty),
          Node(10, Node(14, Empty, Empty), Node(16, Empty, Empty))),
      Node(6, Empty, Node(12, Node(18, Empty, Empty), Empty)))
```

2. Écrire la fonction `fold_tree` dont le fonctionnement généralise celui de `fold_right` pour des arbres binaires. Sur l'arbre `example_tree`, ceci revient à calculer la valeur de l'expression

```
(f 1 (f 2 (f 4 x x) (f 5 (f 7 x x) (f 8 x x))) (f 3 x (f 6 (f 9 x x) x)))
```

   où `f` est une fonction de type `int -> int -> int` et `x` une valeur de type `int`.

3. Utiliser `fold_tree` pour écrire les fonctions suivantes :
   (a) `(bintree_count_internal_nodes t)` qui renvoie le nombre de nœuds internes dans `t` ;
   (b) `(bintree_collect_internal_nodes t)` qui renvoie la liste des entiers apparaissant dans les nœuds internes de `t`.

```
# bintree_count_internal_nodes example_tree;;
- : int = 9

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