Chapitre 3
Les fonctionnelles

Programmation fonctionnelle, Licence d'informatique, 2021

map

- Applique une même fonction sur tous les éléments d'une liste
- Exemples :

```
# open List;;
# map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map (fun nb -> nb + 1) [1; 2; 3; 4];;
- : int list = [2; 3; 4; 5]

# map String.capitalize_ascii ["louis"; "lea"; "rachel"; "pierre";
"marie"];;
- : string list = ["Louis"; "Lea"; "Rachel"; "Pierre"; "Marie"]
```

Exercices

- En utilisant le map, écrire les traitements suivants sur les listes :
  - mettre au carré tous les entiers d'une liste d'entiers,
  - créer la liste des initiales d'une liste de chaines de caractères (String.get s n, retourne le caractère d'indice n de la chaine s)

Correction

```
# map (fun nb -> nb * nb) [1; 2; 3; 4];;
- : int list = [1; 4; 9; 16]

# map (fun s -> String.get s 0) ["louis"; "lea"; "rachel";
"pierre"; "marie"];;
- : char list = ['l'; 'l'; 'r'; 'p'; 'm']
```

fold_left

- Applique une même fonction de cumul sur tous les éléments d'une liste de gauche à droite :

```
# fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# fold_left (+) 0 [1; 2; 3; 4];;
- : int = 10
```

- L'interpréteur calcul l'expression :

(((0 + 1) + 2) + 3) + 4

fold_left

- Exemples :

```
# fold_left (&&) true [true; true; false; true];;
- : bool = false
```

```
# fold_left (^) "" ["Ceci "; "est "; "un "; "exemple."];;
- : string = "Ceci est un exemple. »
```

```
# fold_left (fun cpt _ -> cpt + 1) 0 ["Ceci "; "est "; "un ";
"exemple."];;
- : int = 4
```

Exercices

- En utilisant fold_left, écrire les traitements suivants sur les listes :
  - calculer le produit des entiers d'une liste
  - calculer la longueur d'une liste
  - tester si une liste d'entiers contient un zéro

Correction

```
# fold_left ( * ) 1 [1; 2; 3; 2];;
- : int = 12
```

```
# fold_left (fun cpt _ -> cpt + 1) 0 [1; 2; 3; 4];;
- : int = 4
```

```
# fold_left (fun b elem -> b || (elem = 0)) false [1; 2; 0;
4];;
- : bool = true
```

Fonctionnelles sur les structures récursives

- On peut définir des fonctionnelles sur toutes les autres structures récursives

- Exemple :

```
type 'a tree = Nil | Node of ‘a * ’a tree * 'a tree;;

# let rec tree_map func the_tree =
   match the_tree with
      Nil -> Nil
    | Node(root, left_tree, right_tree) -> Node(func root,
tree_map func left_tree, tree_map func right_tree)
;;
val tree_map : ('a -> 'b) -> 'a tree -> 'b tree = <fun>
```

Fonctionnelles sur les structures récursives

```
# tree_map (fun nb -> nb + 1) (Node(1, Node(2, Nil, Nil), Node(3, Nil, Nil)));;
- : int tree = Node (2, Node (3, Nil, Nil), Node (4, Nil, Nil))

# tree_map ((+) 1) (Node(1, Node(2, Nil, Nil), Node(3, Nil, Nil)));;
- : int tree = Node (2, Node (3, Nil, Nil), Node (4, Nil, Nil))
```

Exercice

- Considèrons le type des arbres binaire suivant dont les valeurs sont uniquement sur les feuilles :

  type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;;

- Définir un intérateur tree_map, qui applique une même fonction à toutes les valeurs de l’arbre et retourne l’arbre de ces nouvelles valeurs.

Correction

```
# type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;;
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree

# let rec tree_map func the_tree =
    match the_tree with
      Leaf value -> Leaf(func value)
    | Node(left_tree, right_tree) -> Node(tree_map func left_tree,
tree_map func right_tree)
;;
val tree_map : ('a -> 'b) -> 'a tree -> 'b tree = <fun>

# tree_map ((-) 0) (Node(Node(Leaf(1), Leaf(2)), Node(Leaf(3),
Leaf(4))));;
- : int tree = Node (Node (Leaf (-1), Leaf (-2)), Node (Leaf (-3), Leaf
(-4)))
```

Fonctionnelles sur les structures récursives (suite)

- Même principe pour le fold_left

```
# let rec tree_fold_infix_left func val_init the_tree =
   match the_tree with
      Nil -> val_init
    | Node(root, left_tree, right_tree) ->
        let left_val = tree_fold_infix_left func val_init left_tree in
        let root_val = func left_val root in
        tree_fold_infix_left func root_val right_tree
;;
val tree_fold_infix_left : ('a -> 'b -> 'a) -> 'a -> 'b tree -> 'a =
<fun>
```

Fonctionnelles sur les structures récursives (suite)

```
# tree_fold_infix_left (+) 0 (Node(1, Node(2, Nil, Nil),
Node(3, Nil, Nil)));;
- : int = 6
```

- L'interpréteur calcule ((0 + 2) + 1) + 3

```
# tree_fold_infix_left (fun the_list elem -> elem ::
the_list) [] (Node(1, Node(2, Nil, Nil), Node(3, Nil,
Nil)));;
- : int list = [3; 1; 2]
```

- L'interprèteur calcule 2 :: [], 1 :: [2], puis 3 :: [1; 2]

Exercices

- Utiliser tree_fold_infix_left pour définir les fonctions suivantes :
  - Compter le nombre de noeuds d'un arbre donné.
  - Rechercher un élément donné dans un arbre donné.
  - Compter le nombre d'occurrences d'un élément donné dans un arbre donné

Corrections

```
# let nb_node the_tree = tree_fold_infix_left (fun cpt _ -> cpt + 1) 0 the_tree ;;
val nb_node : 'a tree -> int = <fun>
# nb_node (Node(1, Node(2, Nil, Nil), Node(3, Nil, Nil)));;
- : int = 3

# let is_in elem the_tree = tree_fold_infix_left (fun fund root -> fund || (elem =
root)) false the_tree;;
val is_in : 'a -> 'a tree -> bool = <fun>
# is_in 3 (Node(1, Node(2, Nil, Nil), Node(3, Nil, Nil)));;
- : bool = true

# let nb_in elem the_tree = tree_fold_infix_left (fun cpt root -> if elem = root
then cpt + 1 else cpt) 0 the_tree;;
val nb_in : 'a -> 'a tree -> int = <fun>
# nb_in 3 (Node(1, Node(3, Nil, Nil), Node(3, Nil, Nil)));;
- : int = 2
```

Correction avec application partielle

```
# let nb_node = tree_fold_infix_left (fun cpt _ -> cpt + 1) 0;;
val nb_node : 'a tree -> int = <fun>
# nb_node (Node(1, Node(2, Nil, Nil), Node(3, Nil, Nil)));;
- : int = 3

# let is_in elem = tree_fold_infix_left (fun fund root -> fund || (elem =
root)) false;;
val is_in : 'a -> 'a tree -> bool = <fun>
# is_in 3 (Node(1, Node(2, Nil, Nil), Node(3, Nil, Nil)));;
- : bool = true

# let nb_in elem = tree_fold_infix_left (fun cpt root -> if elem = root
then cpt + 1 else cpt) 0;;
val nb_in : 'a -> 'a tree -> int = <fun>
# nb_in 3 (Node(1, Node(3, Nil, Nil), Node(3, Nil, Nil)));;
- : int = 2
```