Programmation Fonctionnelle: Listes

Adrien Durier

13 septembre 2023

Listes et Filtrage

Les listes

```
let l = [4; 1; 3; 6; 2; 3]

let rec longueur l =
  match l with
  | [] -> 0
  | _ :: s -> 1 + longueur s

let rec recherche n l =
  match l with
  | [] -> false
  | x :: xs -> x = n || recherche n xs
```

Concaténation de listes

La fonction append construit une nouvelle liste en réunissant deux listes bout à bout.

```
let rec append l1 l2 =
  match l1 with
    [] -> l2
  | x::s -> x::(append s l2)
```

Concaténation de listes

La fonction append construit une nouvelle liste en réunissant deux listes bout à bout.

```
let rec append l1 l2 =
  match l1 with
    [] -> l2
  | x::s -> x::(append s l2)
```

```
append [2;5;1] [10;6;8;15]
```

- Cette fonction est prédéfinie en OCaml, il s'agit de `List.append`
- L'opérateur infixe `@` est un raccourci syntaxique pour cette fonction, on note `l1@l2` la concaténation de `l1` et `l2`

Évaluation de **append**

Évaluation de `append [1;2] [3;4]`

`append [1;2] [3;4]`

Évaluation de **append**

Évaluation de append `[1;2]` `[3;4]`

$$\texttt{[1;2]} \neq \texttt{[]}, x = 1, s = \texttt{[2]} \quad \Rightarrow \quad \begin{array}{l}\texttt{append [1;2] [3;4]} \\ \texttt{1::(append [2] [3;4])}\end{array}$$

Évaluation de **append**

Évaluation de append `[1;2]` `[3;4]`

$$
\begin{array}{lcl}
 & & \texttt{append [1;2] [3;4]} \\
\texttt{[1;2]} \neq \texttt{[]}, x = 1, s = \texttt{[2]} & \Rightarrow & \texttt{1::(append [2] [3;4])} \\
\texttt{[2]} \neq \texttt{[]}, x = 2, s = \texttt{[]} & \Rightarrow & \texttt{1::2::(append [] [3;4])}
\end{array}
$$

Évaluation de **append**

Évaluation de append `[1;2]` `[3;4]`

		`append [1;2] [3;4]`
$[1;2] \neq [], x = 1, s = [2]$	$\Rightarrow$	`1::(append [2] [3;4])`
$[2] \neq [], x = 2, s = []$	$\Rightarrow$	`1::2::(append [] [3;4])`
$[] = []$	$\Rightarrow$	`1::2::[3;4]`
	$\Rightarrow$	`1::[2;3;4]`
	$\Rightarrow$	`[1;2;3;4]`

Concaténation rapide

- La fonction append n'est pas récursive terminale.
- Si l'ordre des éléments n'a pas d'importance, on peut définir une concaténation récursive terminale qui inverse les éléments de la première liste.

Concaténation rapide

- La fonction append n'est pas récursive terminale.
- Si l'ordre des éléments n'a pas d'importance, on peut définir une concaténation récursive terminale qui inverse les éléments de la première liste.

```
let rec rev_append l1 l2 =
  match l1 with
    [] -> l2
  | x :: s -> rev_append s (x :: l2)
```

Concaténation rapide

- La fonction append n'est pas récursive terminale.
- Si l'ordre des éléments n'a pas d'importance, on peut définir une concaténation récursive terminale qui inverse les éléments de la première liste.

```
let rec rev_append l1 l2 =
  match l1 with
    [] -> l2
  | x :: s -> rev_append s (x :: l2)
```

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

Renverser une liste

La fonction `rev` pour renverser une liste `l` s'obtient facilement en concaténant la liste `l` avec la liste vide `[]`, en utilisant `rev_append`

Renverser une liste

La fonction `rev` pour renverser une liste `l` s'obtient facilement en concaténant la liste `l` avec la liste vide `[]`, en utilisant `rev_append`

```
let rev l = rev_append l []
val rev : 'a list -> 'a list = <fun>
```

Renverser une liste

La fonction `rev` pour renverser une liste `l` s'obtient facilement en concaténant la liste `l` avec la liste vide `[]`, en utilisant `rev_append`

```
let rev l = rev_append l []
val rev : 'a list -> 'a list = <fun>
```

```
rev [4;2;6;1]
- : int list = [1; 6; 2; 4]
```

Fonctions sur les listes : Map

La fonction suivante applique une fonction à chacun des éléments.

```
let rec map f l =
  match l with
  | [] -> []
  | x :: s -> f x :: map f s
```

Son type est `('a -> 'b) -> 'a list -> 'b list`

Prédéfinie sous le nom de `List.map`.

```
# map (fun x -> 2 * x) [1; 4; 7];;
- : int list = [2; 8; 14]
# map string_of_int [16; 24; 32; 6];;
- : string list = ["16"; "24"; "32"; "6"]
```

Tri de listes

Tri Rapide : principe

Soit une liste `l` à trier :

1. si `l` est vide alors elle est triée
2. sinon, choisir un élément `p` de la liste (le premier par exemple) nommé le pivot
3. partager `l` en deux listes `g` et `d` contenant les autres éléments de `l` qui sont plus petits (resp. plus grands) que la valeur du pivot `p`
4. trier récursivement `g` et `d`, on obtient deux listes `g'` et `d'` triées
5. on renvoie la liste `g'@[p]@d'` (qui est bien triée)

Insertion

```
let rec inserer x l =
match l with
   [] -> [x]
 | y::s -> if x<=y then x::l else y::(inserer x s)
```

Insertion

```
let rec inserer x l =
match l with
   [] -> [x]
 | y::s -> if x<=y then x::l else y::(inserer x s)
```

```
inserer 5 [3;7;10]
```

Insertion

```
let rec inserer x l =
match l with
   [] -> [x]
 | y::s -> if x<=y then x::l else y::(inserer x s)
```

$$[3;7;10] \neq [], y = 3, s = [7;10], 5 > 3 \quad \Rightarrow$$

```
inserer 5 [3;7;10]
3::(inserer 5 [7;10])
```

Insertion

```
let rec inserer x l =
match l with
   [] -> [x]
 | y::s -> if x<=y then x::l else y::(inserer x s)
```

$$
\begin{array}{lcl}
 & & \texttt{inserer 5 [3;7;10]} \\
[3;7;10] \neq [], y = 3, s = [7;10], 5 > 3 & \Rightarrow & \texttt{3::(inserer 5 [7;10])} \\
[7;10] \neq [], y = 7, s = [10], 5 <= 7 & \Rightarrow & \texttt{3::5::[7;10]}
\end{array}
$$

Insertion

```
let rec inserer x l =
match l with
   [] -> [x]
 | y::s -> if x<=y then x::l else y::(inserer x s)
```

$$
\begin{array}{lcl}
 & & \texttt{inserer 5 [3;7;10]} \\
[3;7;10] \neq [], y = 3, s = [7;10], 5 > 3 & \Rightarrow & \texttt{3::(inserer 5 [7;10])} \\
[7;10] \neq [], y = 7, s = [10], 5 <= 7 & \Rightarrow & \texttt{3::5::[7;10]} \\
 & \Rightarrow & \texttt{3::[5;7;10]}
\end{array}
$$

Insertion

```
let rec inserer x l =
match l with
   [] -> [x]
 | y::s -> if x<=y then x::l else y::(inserer x s)
```

$[3;7;10] \neq [], y = 3, s = [7;10], 5 > 3$

$[7;10] \neq [], y = 7, s = [10], 5 <= 7$

```
inserer 5 [3;7;10]
⇒ 3::(inserer 5 [7;10])
⇒ 3::5::[7;10]
⇒ 3::[5;7;10]
⇒ [3;5;7;10]
```

Trier une liste

```
let rec trier l =
match l with
   [] -> []
 | x::s -> inserer x (trier s)
```

```
trier [6;4;1;5]
```

Trier une liste

```
let rec trier l =
match l with
   [] -> []
 | x::s -> inserer x (trier s)
```

$x = 6, s = [4; 1; 5]$ $\Rightarrow$

```
trier [6;4;1;5]
inserer 6 (trier [4;1;5])
```

Trier une liste

```
let rec trier l =
match l with
   [] -> []
 | x::s -> inserer x (trier s)
```

$$
\begin{array}{lcl}
 & & \texttt{trier [6;4;1;5]} \\
x = 6, s = [4;1;5] & \Rightarrow & \texttt{inserer 6 (trier [4;1;5])} \\
x = 4, s = [1;5] & \Rightarrow & \texttt{inserer 6 (inserer 4 (trier [1;5]))}
\end{array}
$$

Trier une liste

```
let rec trier l =
match l with
   [] -> []
 | x::s -> inserer x (trier s)
```

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`

Trier une liste

```
let rec trier l =
match l with
   [] -> []
 | x::s -> inserer x (trier s)
```

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))))`

Trier une liste

```
let rec trier l =
match l with
   [] -> []
 | x::s -> inserer x (trier s)
```

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))))`
	$\Rightarrow$	`...(inserer 5 []))))`

Trier une liste

```
let rec trier l =
match l with
   [] -> []
 | x::s -> inserer x (trier s)
```

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))`
	$\Rightarrow$	`...(inserer 5 []))`
	$\Rightarrow$	`inserer 6 (inserer 4 (inserer 1 [5]))`

Trier une liste

```
let rec trier l =
match l with
   [] -> []
 | x::s -> inserer x (trier s)
```

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))`
	$\Rightarrow$	`...(inserer 5 []))`
	$\Rightarrow$	`inserer 6 (inserer 4 (inserer 1 [5]))`
	$\Rightarrow$	`inserer 6 (inserer 4 [1;5])`

Trier une liste

```
let rec trier l =
match l with
   [] -> []
 | x::s -> inserer x (trier s)
```

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))`
	$\Rightarrow$	`...(inserer 5 []))`
	$\Rightarrow$	`inserer 6 (inserer 4 (inserer 1 [5]))`
	$\Rightarrow$	`inserer 6 (inserer 4 [1;5])`
	$\Rightarrow$	`inserer 6 [1;4;5]`

Trier une liste

```
let rec trier l =
match l with
   [] -> []
 | x::s -> inserer x (trier s)
```

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))`
	$\Rightarrow$	`...(inserer 5 []))`
	$\Rightarrow$	`inserer 6 (inserer 4 (inserer 1 [5]))`
	$\Rightarrow$	`inserer 6 (inserer 4 [1;5])`
	$\Rightarrow$	`inserer 6 [1;4;5]`
	$\Rightarrow$	`[1;4;5;6]`

Tri Rapide : principe

Soit une liste `l` à trier :

1. si `l` est vide alors elle est triée
2. sinon, choisir un élément `p` de la liste (le premier par exemple) nommé le pivot
3. partager `l` en deux listes `g` et `d` contenant les autres éléments de `l` qui sont plus petits (resp. plus grands) que la valeur du pivot `p`
4. trier récursivement `g` et `d`, on obtient deux listes `g'` et `d'` triées
5. on renvoie la liste `g'@[p]@d'` (qui est bien triée)

Partage d'une liste

```
let rec partage p l =
match l with
  [] -> ([] , [])
| x::s -> let (g, d) = partage p s in
          if x <= p then (x::g, d) else (g, x::d)
```

Tri rapide

```
let rec tri_rapide l =
match l with
  [] -> []
| p::s -> let g , d = partage p s in
          (tri_rapide g)@[p]@(tri_rapide d)
```

Tri rapide

```
let rec tri_rapide l =
match l with
  [] -> []
| p::s -> let g , d = partage p s in
          (tri_rapide g)@[p]@(tri_rapide d)
```

```
tri_rapide [5; 1; 9; 7; 3; 2; 4]
- : int list = [1; 2; 3; 4; 5; 7; 9]
```