Programmation fonctionnelle et Parallélisme

Cours 5

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-fp

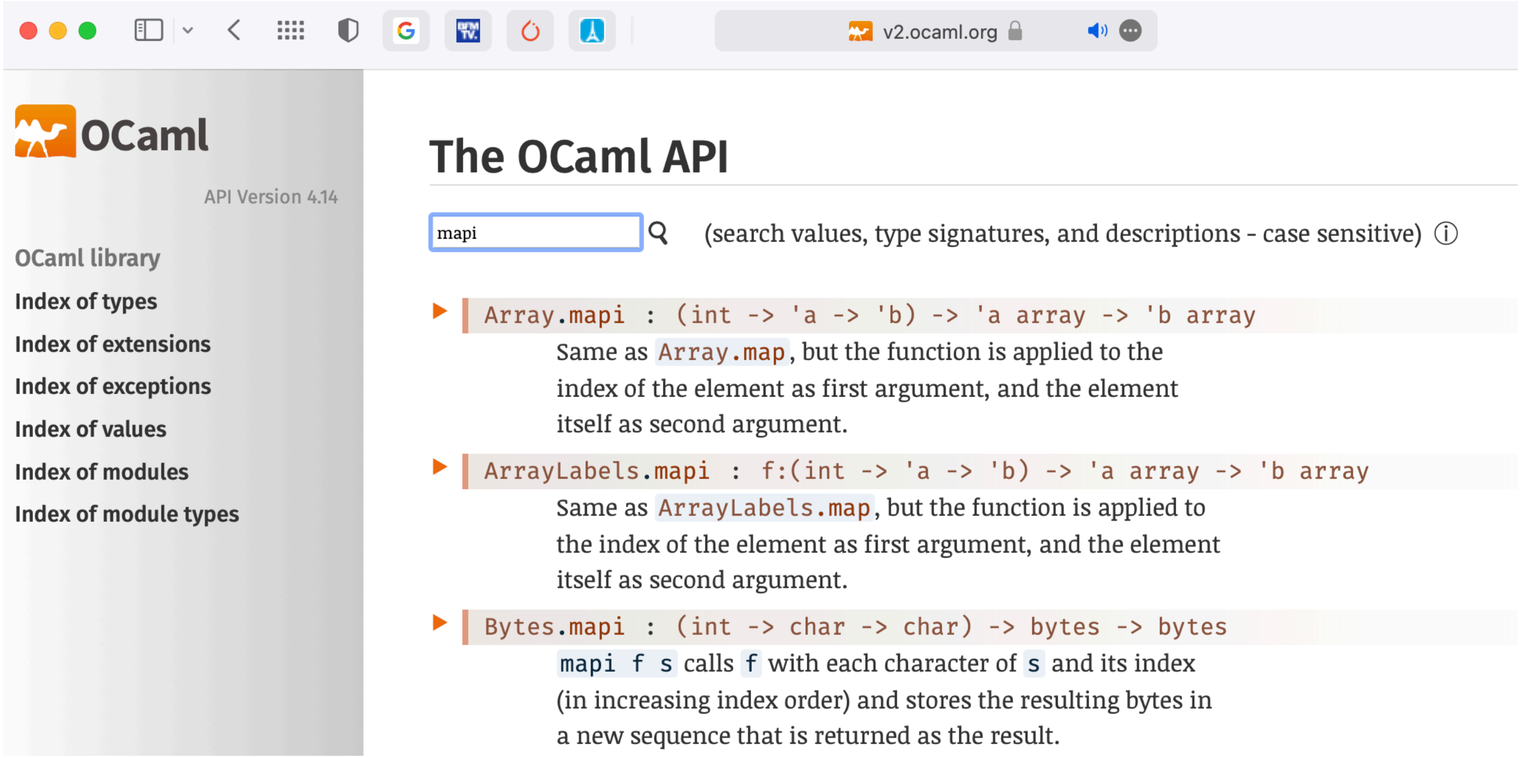
Plan

- récusivité
- tris récursifs (*Quicksort, Mergesort*)
- listes
- filtrage (*pattern matiching*)

télécharger Ocaml en `http://www.ocaml.org`

Librairie standard

- l'API de Ocaml est visible en `http://v2.ocaml.org/api`
- avec les fonctions de la librairie standard aussi en `http://v2.ocaml.org/manual/stdlib.html`

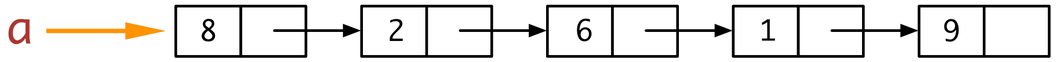


Listes chaînées

- les tableaux sont des zones mémoire contigües de taille **fixe**
- les listes chaînées ont une taille **variable**
- les listes chaînées ont un **accès séquentiel** à partir de la tête de liste
- chaque cellule de liste a une valeur et un pointeur vers la cellule suivante
- le suivant du dernier élément est `[ ]` (*nil* — la liste vide)

```
let a = [ 8 ; 2 ; 6 ; 1 ; 9]
```

```
val a : int list = [8; 2; 6; 1; 9]
```



Listes chaînées

- une liste est :
  - soit la liste vide [ ]
  - soit cons d'un élément et d'une liste de même type

$$\text{liste}(\alpha) = [\,] \oplus \alpha :: \text{liste}(\alpha)$$

liste vide — cons

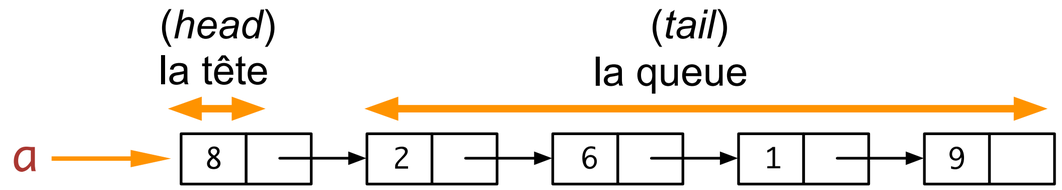
- on a donc :

```
let a = [ 8 ; 2 ; 6 ; 1 ; 9]
```

```
a = 8 :: [ 2 ; 6; 1; 9]
a = 8 :: 2 :: [ 6; 1; 9]
a = 8 :: 2 :: 6 :: [ 1; 9]
a = 8 :: 2 :: 6 :: 1 :: [9]
a = 8 :: 2 :: 6 :: 1 :: 9 :: [ ]
```

- et :

```
List.hd a        8
List.tl a        [2; 6; 1; 9]
```



Listes chaînées

- on raisonne par cas sur les listes avec le filtrage

```
match a with
  | [ ] -> ...
  | x :: a' -> ...
```

```
match a with
   [ ] -> ...
 | x :: a' -> ...
```

écriture aussi possible (déconseillé)

- quelques exemples:

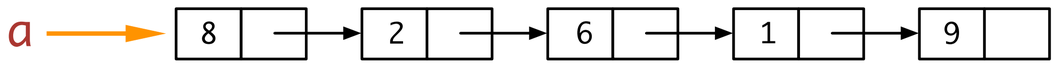
```
let rec len a = match a with
  | [ ] -> 0
  | _ :: a' -> 1 + len a' ;;

let len' = List.length ;;
let len_l a = List.fold_left (fun r _ -> r + 1) 0 a ;;
let len_r a = List.fold_right (fun _ r -> r + 1) a 0 ;;
```

```
List.length a
```

librairie standard de Ocaml

```
List.map f a
```

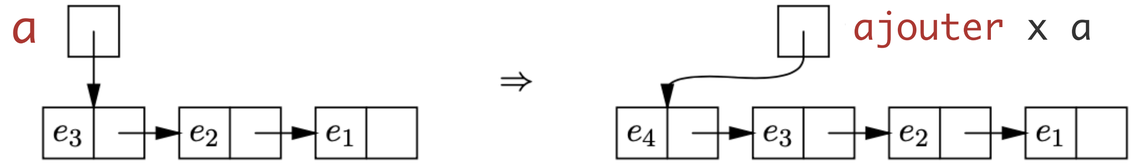
```
let rec map f a = match a with
  | [ ] -> [ ]
  | x :: a' -> (f x) :: map f a' ;;

let map' = List.map ;;

map (fun x -> x + 1) a ;;
map (fun s -> "Coucou " ^ s) ["JJ"; « Talla"; "Takatoshi"] ;;
map String.length ["JJ"; "Talla"] ;;
```



les sources sont en http://jeanjacqueslevy.net/prog-fp/progs/fp6a.ml

Listes chaînées

Exercice Ajouter un élément dans une liste




```
let ajouter x a = x
```

Exercice Trouver le i-ème élément dans une liste

```
exception Error ;;

let rec nth a i = match a with
| [ ] -> raise Error
| x :: a' -> if i = 0 then x else nth a' (i - 1) ;;
```

`List.nth a i` librairie standard de Ocaml

les sources sont en `http://jeanjacqueslevy.net/prog-fp/progs/fp6a.ml`

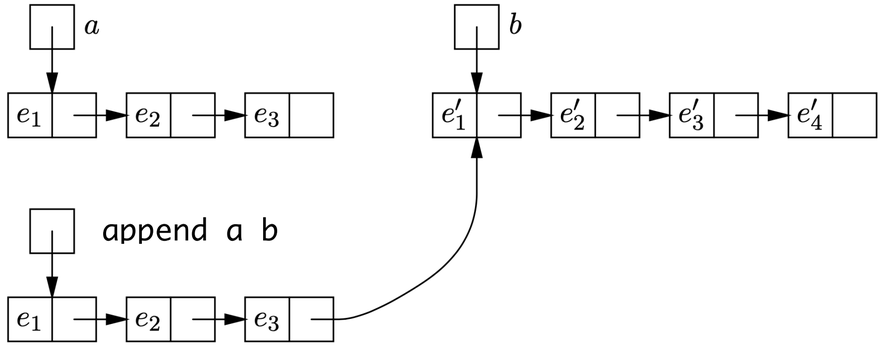
Listes chaînées

Exercice Concaténer 2 listes (en programmation fonctionnelle)

```
let rec append a b = match a with
| [ ] -> b
| x :: a' -> x :: append a' b ;;
```

ou encore

```
let append' a b = List.fold_right (fun x r -> x :: r) a b ;;

let append'' a b = a @ b ;;
```



`List.append a b`

librairie standard de Ocaml

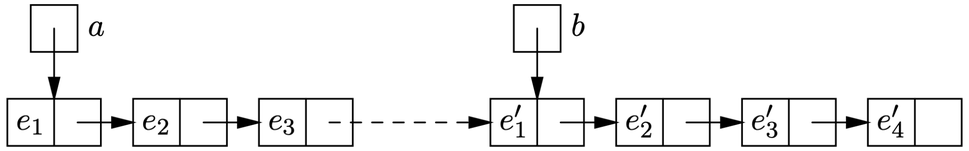
`a @ b`

- append ne modifie pas les listes. C'est donc différent de la fonction nconc qui modifie la liste a

(programmation impérative)

```
let nconc a b = . . .
```

impossible à programmer avec des listes non modifiables



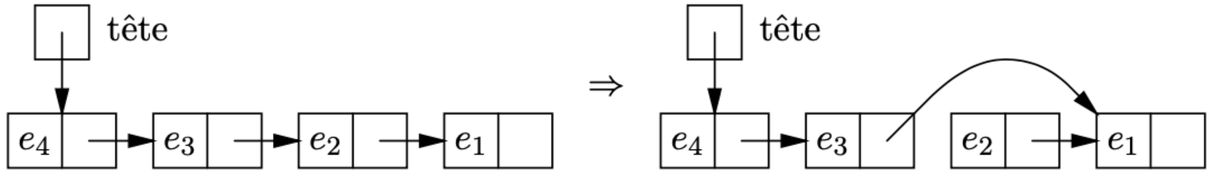
Listes chaînées

Exercice Insérer un élément avant le i-ème élément dans une liste

```
let rec insererAV x i a = match a with
| [ ] -> raise Error
| e :: a' -> if i = 0
             then x :: e :: a'
             else e :: insererAV x (i-1) a' ;;
```

Exercice Supprimer du i-ème élément dans une liste

```
let rec supprimer i a = match a with
| [ ] -> raise Error
| e :: a' -> if i = 0 then a'
               else e :: supprimer (i-1) a' ;;
```



les sources sont en http://jeanjacqueslevy.net/prog-fp/progs/fp6a.ml

Listes chaînées

Exercice Calculer l'image miroir d'une liste (en programmation fonctionnelle)

```
let rec reverse a = match a with
| [ ] -> [ ]
| x :: a' -> append (reverse a') [x] ;;
```

et une autre version plus efficace

```
let rec rev_append a b = match a with
| [ ] -> b
| x :: a' -> rev_append a' (x :: b) ;;

let reverse' a = rev_append a [ ] ;;
```

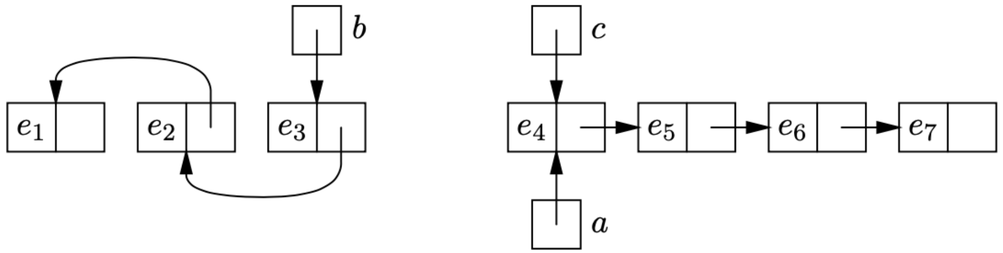
```
List.rev a
```

librairie standard de Ocaml

```
List.rev_append a b
```

et encore l'image miroir d'une liste (en programmation impérative)

impossible à programmer avec des listes non modifiables



Types de données (*datatypes*)

- mes listes avec définition inductive

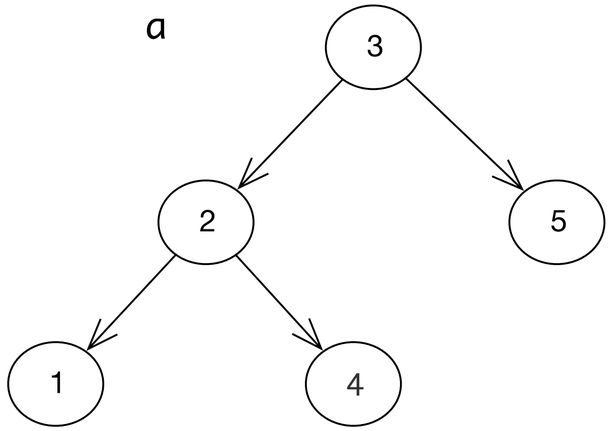
```
type 'a mylist =
| Mynil
| Mycons of 'a * 'a mylist ;;
```

- des arbres binaires avec définition inductive

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

ou en supposant les feuilles sans valeur associée

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



les sources sont en http://jeanjacqueslevy.net/prog-fp/progs/fp6a.ml

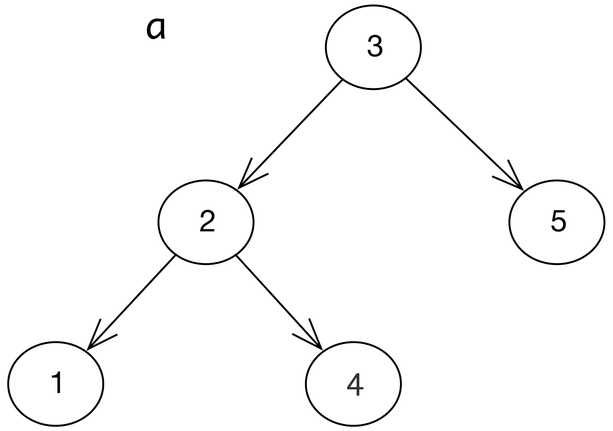
Types de données (*datatypes*)

- des arbres binaires avec définition inductive

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


let a = Node (3, Node (2, Leaf(1), Leaf(4)), Leaf(5)) ;;
```

ou en supposant les feuilles sans valeur associée

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

let a = Node (3, Node (2, Node (1, Leaf, Leaf), Node (4, Leaf, Leaf)),
                 Node (5, Leaf, Leaf)) ;;
```



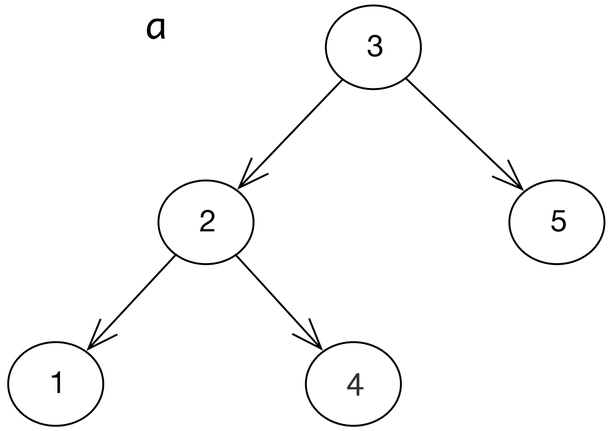
Types de données (*datatypes*)

- des arbres binaires avec définition inductive

```
let rec taille a = match a with
| Leaf _ -> 1
| Node (_, g, d) -> 1 + taille g + taille d ;;
```

```
let rec hauteur a = match a with
| Leaf _ -> 0
| Node (_, g, d) -> 1 + max (hauteur g) (hauteur d) ;;
```

```
let rec max_elt a = match a with
| Leaf x -> x
| Node (x, g, d) -> max x (max (max_elt g) (max_elt d)) ;;
```



Conclusion

VU:

- listes
- filtrage
- datatypes

TODO list

- enregistrements
- modules
- types de données modifiables
- parallélisme
- concurrence