Programmation fonctionnelle et Parallélisme

Cours 5

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-fp

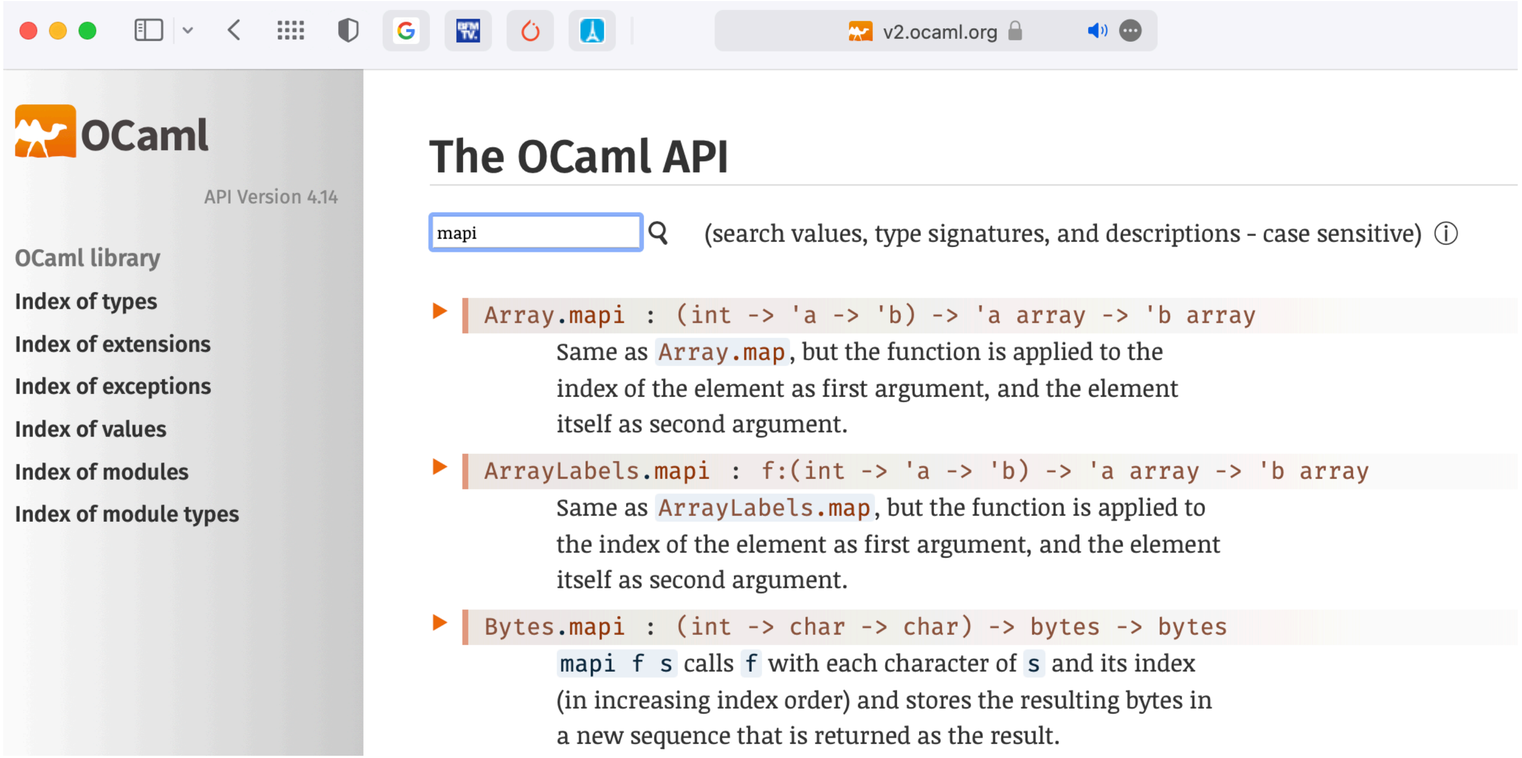
Plan

- 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`



Rappels et exercices

VU:

- int, float, char, strings, array
- itérateurs sur tableaux et chaînes
- tableaux multidimensionnels
- fonctions anonymes
- types polymorphes
- exceptions

Exercice 1 Trouver l'indice du maximum dans un tableau d'entiers

Exercice 2 Trouver l'indice du premier nombre négatif dans un tableau d'entiers

Exercice 3 Trouver l'indice du dernier nombre négatif dans un tableau d'entiers

Exercice 4 Trouver l'indice du premier caractère différent dans 2 chaînes `s` et `s'` (-1 si les mêmes chaînes)

[indication: utiliser la fonction `String.iteri`]

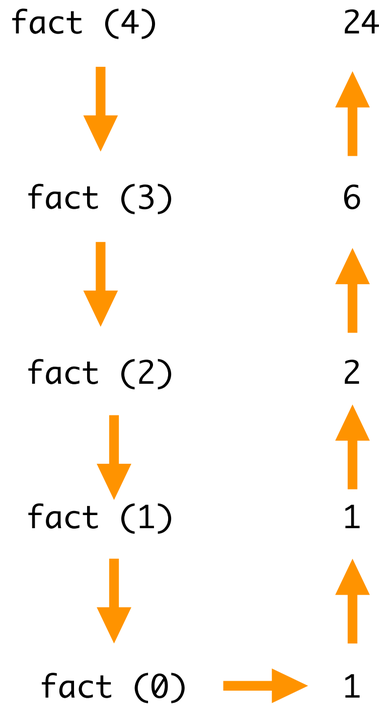
Fonctions récursives

- une fonction qui se rappelle avec un argument plus petit

```
let rec fact =
  if n = 0 then 1 else n * fact (n-1) ;;

fact 3 ;;

fact 10 ;;
```

factorielle

appels récursifs



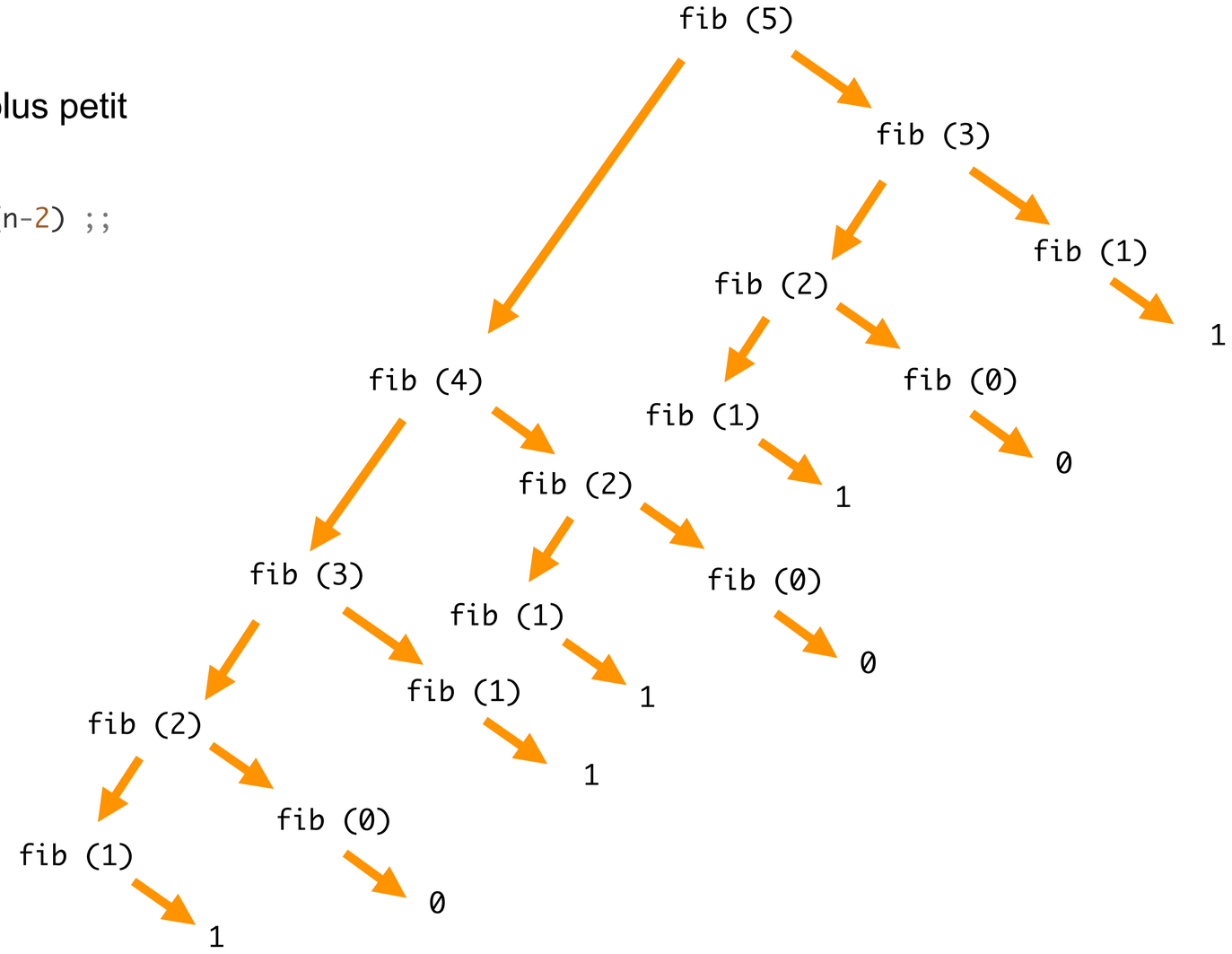
les sources sont en http://jeanjacqueslevy.net/prog-fp/progs/fp5a.ml

Fonctions récursives

- une fonction qui se rappelle avec un argument plus petit

```
let rec fib n =
  if n = 0 || n = 1 then n else fib (n-1) + fib (n-2) ;;

fib 10 ;;

fib 35 ;;
```

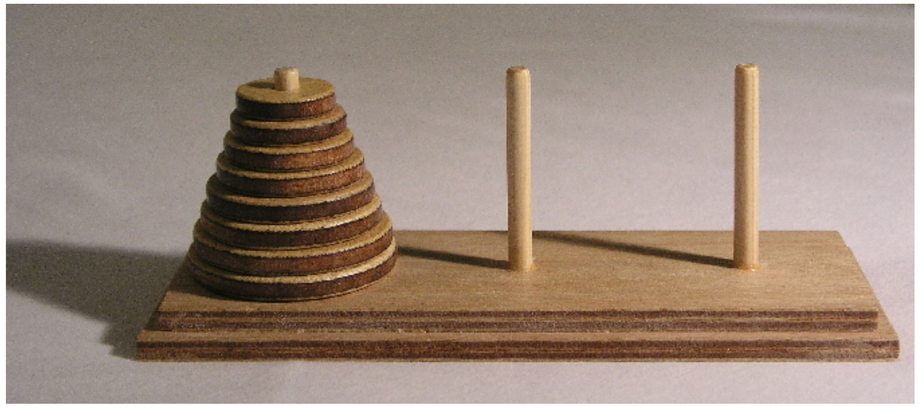
fibonacci



- écrire une fonction qui calcule `fibonacci` plus rapidement

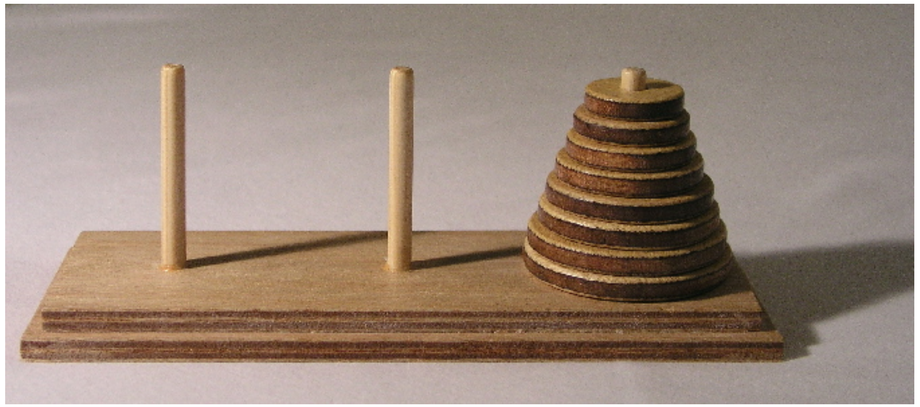
Les tours de Hanoi

- on a 3 piles et n rondelles sur la pile 1
- jamais une rondelle grosse au-dessus d'une rondelle petite



pile 1 pile 2 pile 3

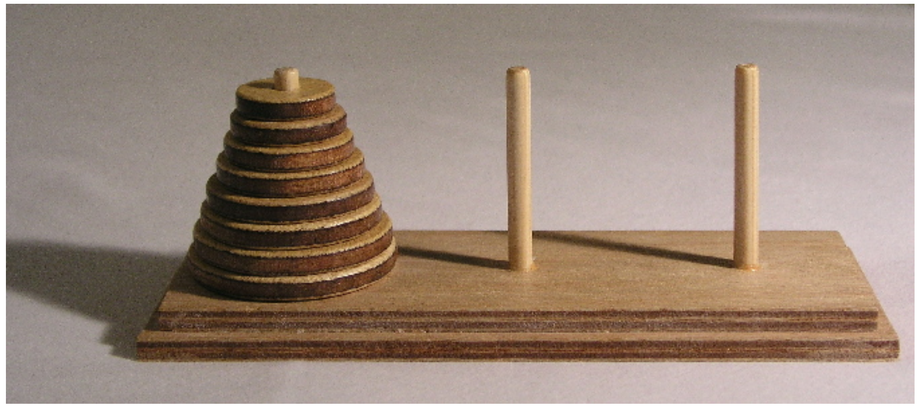
- il faut amener les n rondelles sur la pile 3
- on ne déplace qu'une seulle rondelle à la fois
- et on ne met jamais une rondelle au-dessus d'une plus petite



pile 1 pile 2 pile 3

Les tours de Hanoi

- on a 3 piles et n rondelles sur la pile 1
- jamais une rondelle grosse au-dessus d'une rondelle petite



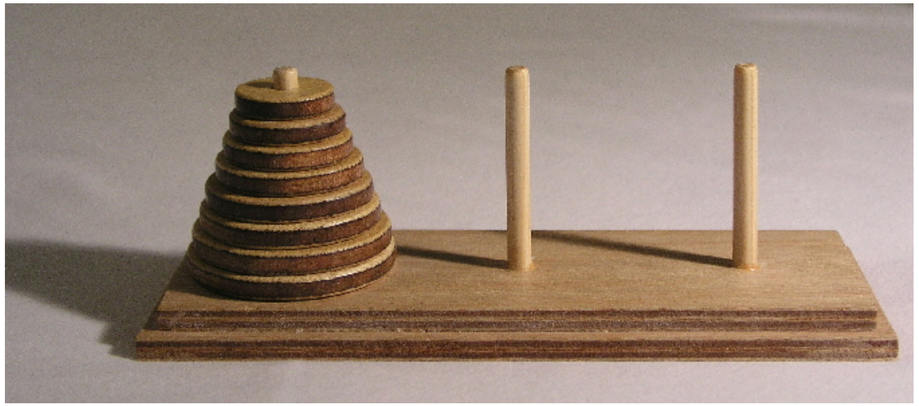
pile 1 **pile 2** **pile 3**

- il faut amener les n rondelles sur la pile 3
- on ne déplace qu'une seule rondelle à la fois
- et on ne met jamais une rondelle au-dessus d'une plus petite

Les tours de Hanoi

- on généralise le problème pour aller de la pile i à la pile j
  où $1 \leq i \leq 3$ et $1 \leq j \leq 3$

  la troisième pile est alors $6 - i - j$



pile i pile $6 - i - j$ pile j

- supposons le problème résolu pour n-1 rondelles entre pile i et pile j

- j'amène les n-1 rondelles du dessus de la pile i sur la troisième pile
- j'amène la grosse rondelle de la pile i vers la pile j
- j'amène les n-1 rondelles de la troisième pile vers la pile j

```
let rec hanoi n i j =
  if n > 0 then begin
    hanoi (n-1) i (6 - i -j) ;
    printf "%d --> %d\n" i j ;
    hanoi (n-1) (6 -i -j) j
  end ;;
```

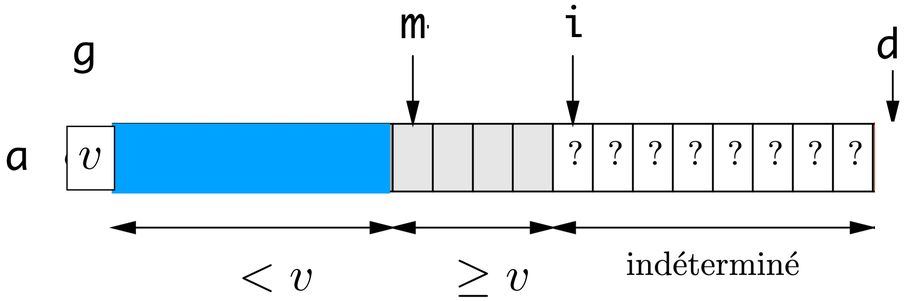
Tri récursif

- tri rapide (*Quicksort*)

```
let rec tri_rapide1 a g d =
  if g < d - 1 then
    let v = a.(g) in
    let m = ref (g + 1) in
    for i = g + 1 to d - 1 do
      if a.(i) < v then begin
        xchange a !m i;
        incr m
      end;
    done;
    xchange a g (!m-1);
    tri_rapide1 a g (!m-1);
    tri_rapide1 a !m d ;;
```

```
let tri_rapide a =
  let n = Array.length a in
  tri_rapide1 a 0 n ;;
```

- bonne méthode de tri en moyenne



a) on met `a[g]` à sa place dans `a` trié

b) et on recommence sur les parties gauche et droite

```
let xchange a i j =
  let tmp = a.(i) in
  a.(i) <- a.(j);
  a.(j) <- tmp ;;
```

échanger les valeurs de `a.(i)` et `a.(j)`

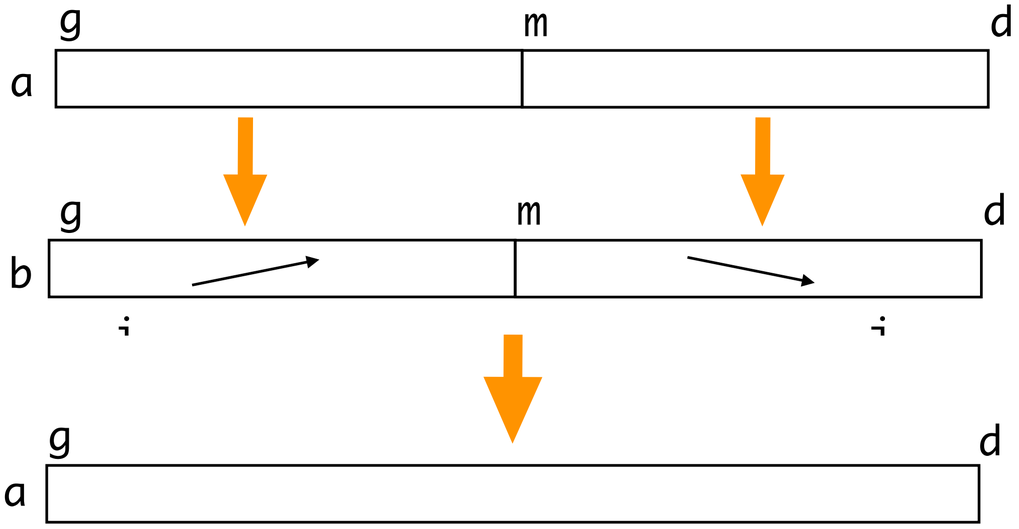
Tri récursif

• tri fusion (*merge sort*)

```
let rec tri_fusion1 a g d b =
  if g < d - 1 then
    let m = (g + d) / 2 in
    tri_fusion1 a g m b;
    tri_fusion1 a m d b;
    for i = 0 to m-1 do b.(i) <- a.(i) done;
    for j = m to d-1 do b.(m + d-1 - j) <- a.(j) done;
    let i = ref g and j = ref (d-1) in
    for k = g to (d-1) do
      if b.(!i) < b.(!j) then begin
          a.(k) <- b.(!i); incr i; end
      else begin
          a.(k) <- b.(!j); decr j; end
    done ;;
```

```
let tri_fusion a =
  let n = Array.length a in
  if n > 0 then
    let b = Array.make n a.(0) in
    tri_fusion1 a 0 n b ;;
```

• très bonne méthode de tri



a) on coupe a en 2

b) on trie les moitiés gauche et droite

c) on copie les résultats dans un tableau annexe b

d) on fusionne les 2 moitiés dans le tableau a

Quelques remarques

```
# selection_sort ;;
- : 'a array -> unit = <fun>
# insertion_sort ;;
- : 'a array -> unit = <fun>
# bubble_sort ;;
- : 'a array -> unit = <fun>
```

- ces fonctions de tris ont des types polymorphes

```
let a = [| 44; 127; 24; 15; 60; 149; 147; 72; 36; 34 |] ;;
let b = [| 2.3; 2.0; 4.6 |] ;;
let c = [| "camille"; "jean-jacques"; "paul"; "axel" |];;

tri_fusion a        →    [|15; 24; 34; 36; 44; 60; 72; 127; 147; 149]|
tri_fusion b        →    [|2.; 2.3; 4.6|]
tri_fusion c        →    [|"axel"; "camille"; "jean-jacques"; "paul"|] ;;
```

- et on voit les erreurs de types avant l'exécution

```
let d = [| "camille"; 28; "paul"; "axel"|] ;;
Error: This expression has type int but an expression was expected of type
         string

let e = [| 2.3; 2; 4.6 |] ;;
Error: This expression has type int but an expression was expected of type
         float
  Hint: Did you mean `2.'?
```

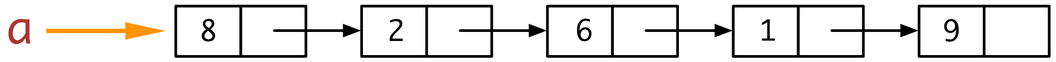
- en Ocaml, les **types sont statiques**

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

- la définition inductive des listes est : $\text{liste}(\alpha) = [\,] \oplus \alpha :: \text{liste}(\alpha)$

cons

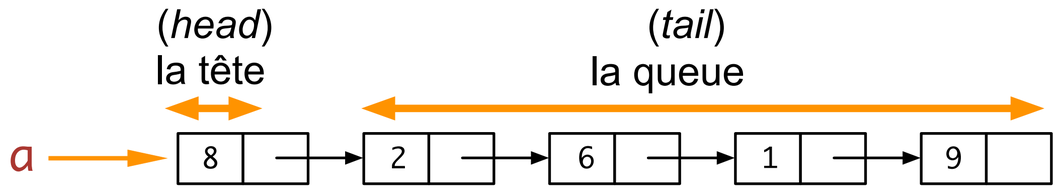
- une liste est :
  - soit la liste vide [ ]
  - soit cons d'un élément et d'une liste de même type

- 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 :: [ ]
```

```
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)

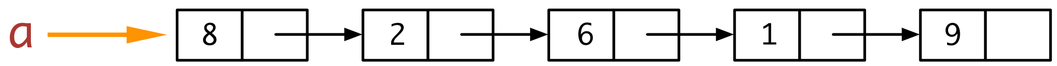
- quelques exemples:

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

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

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

```
List.length a
```

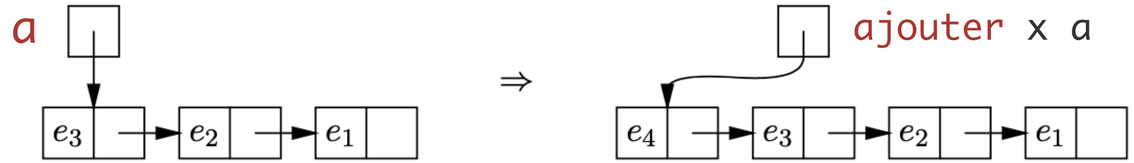
librairie standard de Ocaml

```
List.map f a
```



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

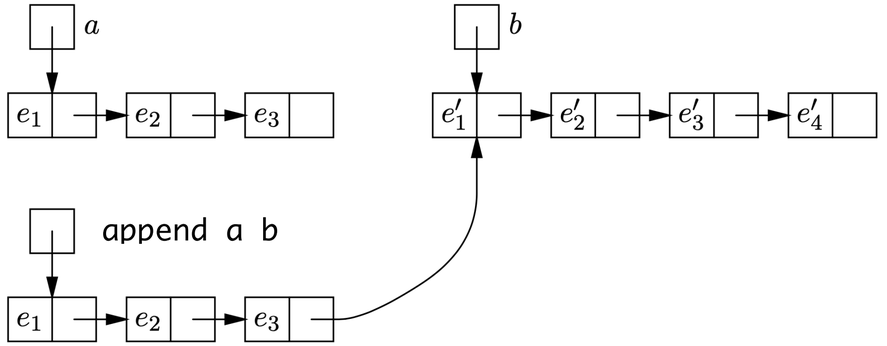
Listes chaînées

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

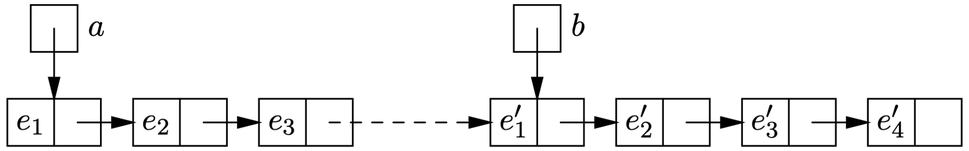
`a @ b`

- 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



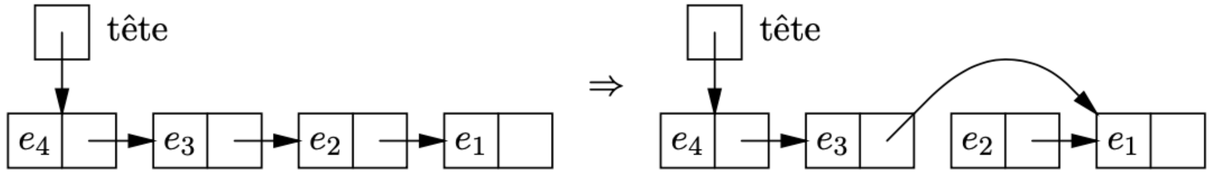
Listes chaînées

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' ;;
```



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 [ ] ;;
```

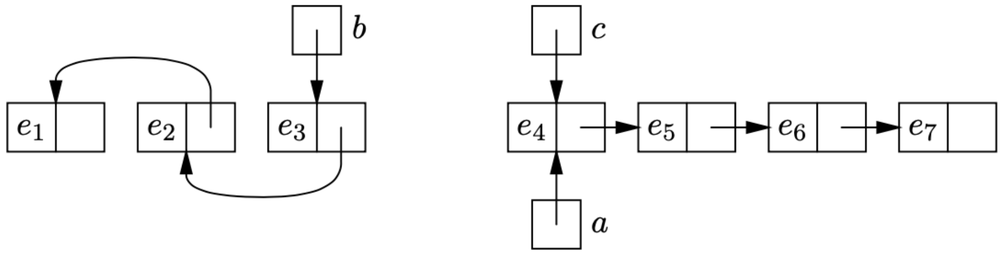
```
List.rev a
```

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



Conclusion

VU:

- récursivité
- listes
- filtrage

TODO list

- listes (suite)
- arbres
- types de données structurées