Fonctionnalité et Modularité

Cours 3

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-fm

Plan

- listes
- filtrage (*pattern matiching*)
- types de données
- files d'attente

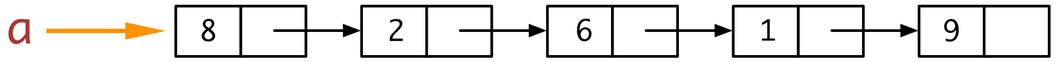
télécharger Ocaml en `http://www.ocaml.org`

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

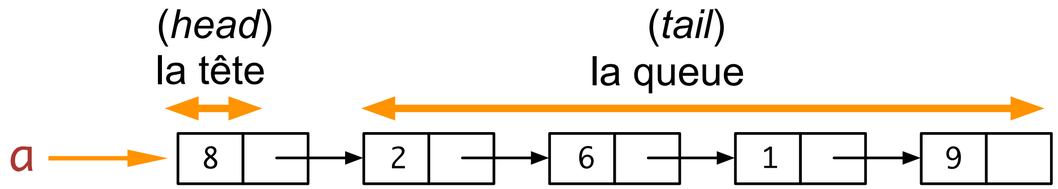
- une liste est :
  - soit la liste vide [ ]
  - soit cons d'un élément et d'une liste de même type

- 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)

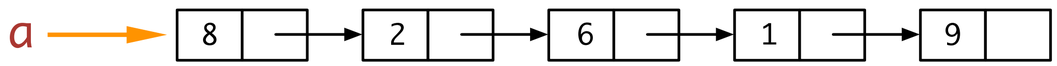
- quelques exemples:

```
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

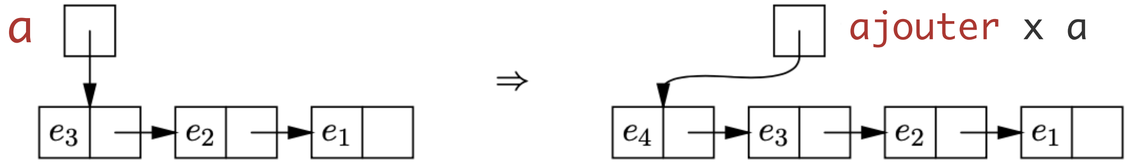
```
List.map f a
```



Listes chaînées

Exercice Ajouter un élément dans une liste

```
let ajouter x a = x :: a ;;
```



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

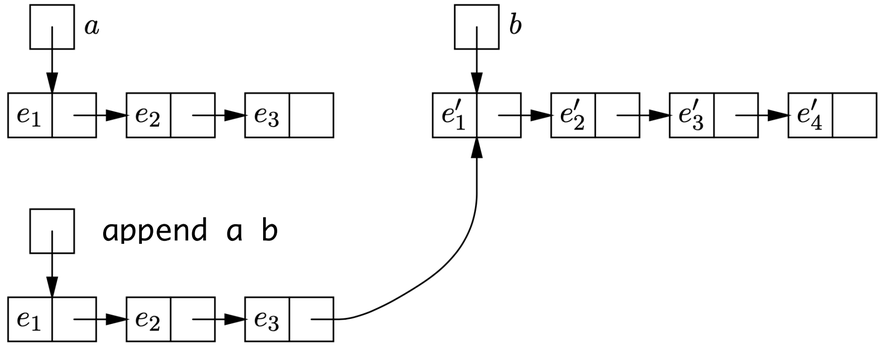
Listes chaînées

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

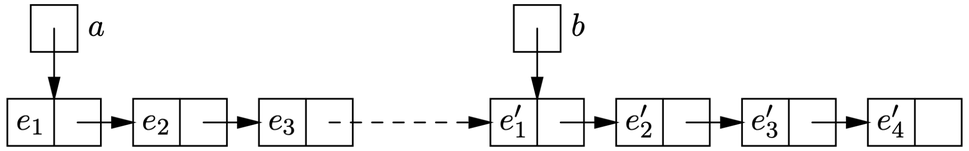
`a @ b`

- 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



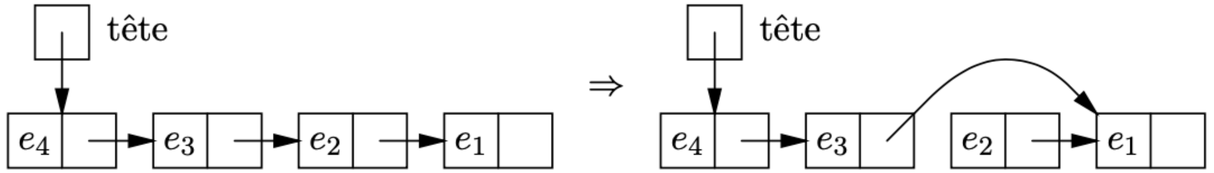
Listes chaînées

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 [ ] ;;
```

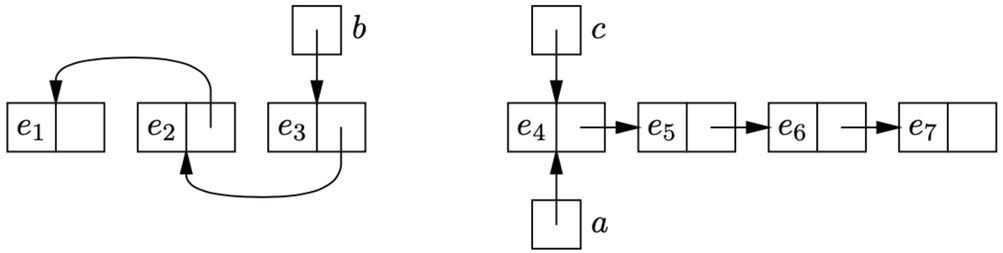
```
List.rev a
```

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



Types (*datatypes*)

- définition du type des listes d'entiers

```
type list_int = Nil | Cons of int * list_int ;;
```

- définition du type des listes polymorphes

```
type 'a list = Nil | Cons of 'a * 'a list ;;
```

- définition alternative du type des listes d'entiers

```
type list_int = int list ;;
```

- en fait, les listes de Ocaml utilisent `[ ]` et `::` (infixe) pour constructeurs

```
type list_int =
  | Nil | Cons of int * list_int ;;
```

autre écriture possible (plus cohérente avec match)

```
type 'a list =
| [ ]
| (::) of 'a * 'a list
```

les listes polymorphes sont prédéfinies dans la librairie standard de Ocaml

Types (*datatypes*)

- exemple: tri d'une liste d'entiers

```
let random_list n p =
  Random.self_init() ;
  List.init n (fun _ -> Random.int p) ;;
```

- tri par insertion

```
let rec sort s =
  match s with
  | [ ] -> [ ]
  | x :: s' -> insert x (sort s')
and insert elt s =
  match s with
  | [ ] -> [elt]
  | x :: s' -> if elt <= x then elt :: s else x :: insert elt s'
;;
```

```
let print_int_list = List.iter (Printf.printf "%d ") ;;

let test f s =
  print_int_list s ;
  print_newline() ;
  print_int_list (f s) ;
  print_newline() ;;
```

```
let s = random_list 10 100 ;;
test sort s ;;
```

Types (*datatypes*)

- autre exemple de type

```
type 'a option = None | Some of 'a ;;
```

- permet de traiter les cas exceptionnels

```
(* val nth : 'a list -> int -> 'a option = <fun> *)

let rec nth a i = match a with
| [ ] -> None
| x :: a' -> if i = 0 then Some x else nth a' (i - 1) ;;
```

- et avec impression

```
match nth [1; 2; 4; 5] 100 with
  | None -> Printf.printf "liste trop courte"
  | Some x -> Printf.printf "%d\n" x ;;
```

```
type 'a option =
  | None
  | Some of 'a
```

le type option est
prédéfini dans la librairie standard
de Ocaml

Files d'attente

- une file d'attente (*FIFO — First In First Out*)

- comment ajouter un élément ?

- les variables sont des constantes en Ocaml

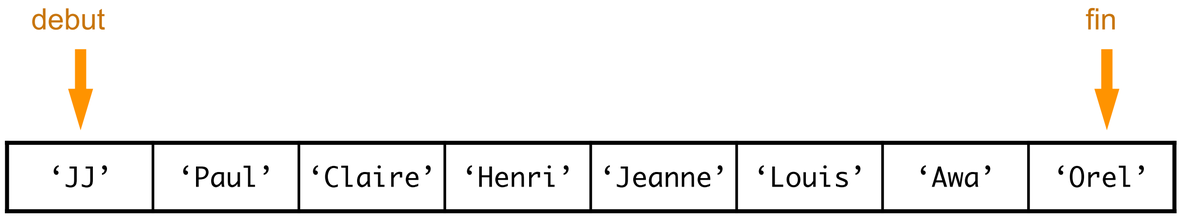
- on peut recopier toute la liste en ajoutant le dernier élément (mais très coûteux)

```
let ajouter x a = a @ [x] ;;
let retirer = List.tl ;;
```

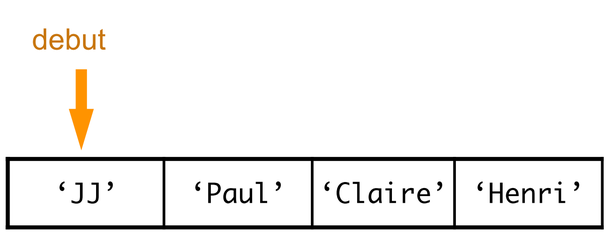
les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c7a.py

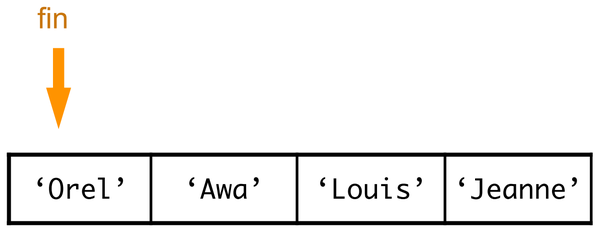
Files d'attente

- avec 2 listes (meilleure implémentation)




```
let ajouter x (debut, fin) = (debut, x :: fin) ;;

let rec retirer = function
  | [ ], [ ] -> raise Error
  | [ ], b -> retirer (List.rev b, [ ])
  | (x :: a), b -> (a, b) ;;
```





les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c7a.py

Conclusion

VU:

- récursivité
- listes
- filtrage
- exemple de structures fonctionnelles

TODO list

- arbres
- références
- données modifiables
- enregistrements