Fonctionnalité et Modularité

Cours 5

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-fm

Plan

- enregistrements
- champs modifiables
- références
- programmation impérative
- listes modifiables
- arbres modifiables
- polymorphisme et contenus modifiables

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

Enregistrements

- on définit le type `point` comme un enregistrement

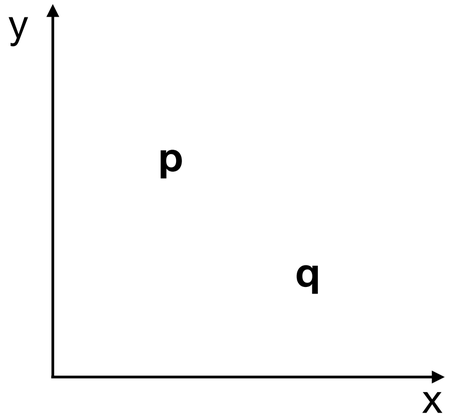
```
type point = {x: int; y: int} ;;

let p = {x = 10; y = 20} ;;
let q = {x = 20; y = 10} ;;
```

- et on obtient chaque champ avec une notation qualifiée

```
let milieu p q = {x = (p.x + q.x) / 2;
                  y = (p.y + q.y) / 2} ;;
```

- impression d'un point

```
let string_of_point p =
  Printf.sprintf "{x = %d; y = %d}" p.x p.y ;;

let print_point p =
  Printf.printf "%s\n" (string_of_point p) ;;
```



les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Enregistrement modifiable

- un type `pointM` avec champs modifiables

```
type pointM = {mutable x: int; mutable y:int} ;;

let p' = {x = 11; y = 22} ;;
```

- et on modifie les champs

```
let avancer p dx dy =
   p.x <- p.x + dx;
   p.y <- p.y + dy ;;

avancer p' 40 50 ;;
```

- on peut passer du type point au type point modifiable et vice-versa

```
let pointM_from_point (p: point) = {x = p.x; y = p.y };;
let point_from_pointM p : point = {x = p.x; y = p.y };;
let print_pointM p = print_point (point_from_pointM p) ;;
```

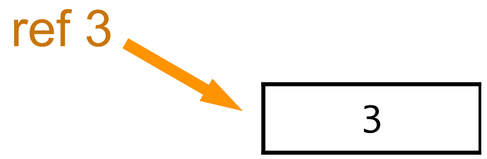
Références

- une référence est un enregistrement avec un seul champ `contents` modifiable

```
type 'a ref = { mutable contents: 'a };;
```

type prédéfini

```
let x = {contents = 3} ;;

Printf.printf "%d\n" x.contents;;

x.contents <- 30 ;;
```



- syntaxe spéciale pour les références avec des nouveaux opérateurs

```
let x = ref 3 ;;

Printf.printf "%d\n" !x;;

x := 30 ;;
```

- incrémentation

```
# ref ;;
- : 'a -> 'a ref = <fun>
# (!) ;;
- : 'a ref -> 'a = <fun>
# (:=) ;;
- : 'a ref -> 'a -> unit = <fun>
```

Données modifiables

- en Ocaml, les données **modifiables** sont:

  - les contenus des tableaux

  - les champs modifiables des enregistrements

  - les contenus de références

- les autres données sont **constantes**

- la programmation **fonctionnelle** n'utilise pas de données modifiables

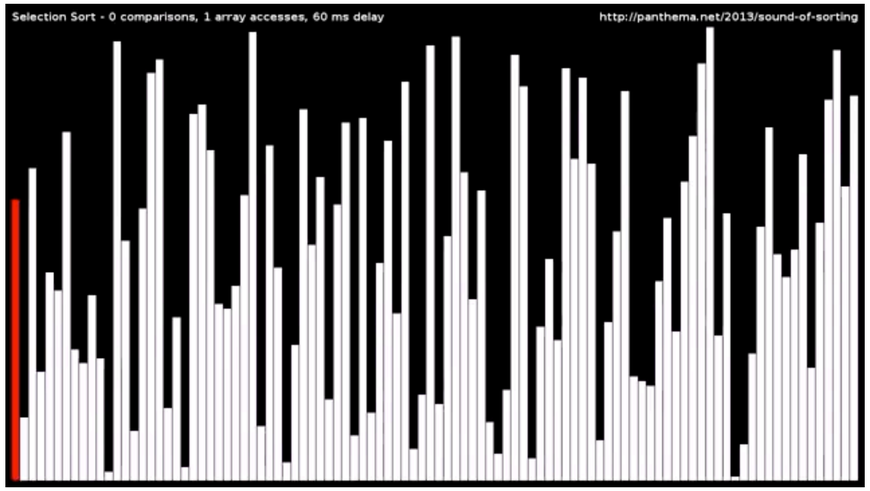
- la programmation impérative peut utiliser des données modifiables

Tri sélection

- on cherche le minimum et on le met en tête..
  et on recommence à partir du deuxième élément, etc…

```
let selection_sort a =
  let n = Array.length a in
  for i = 0 to n-2 do
    let jmin = ref i in
    for j = i+1 to n-1 do
      if a.(j) < a.(!jmin) then
          jmin := j
    done;
    xchange a i !jmin
  done;;
```

programmation impérative




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

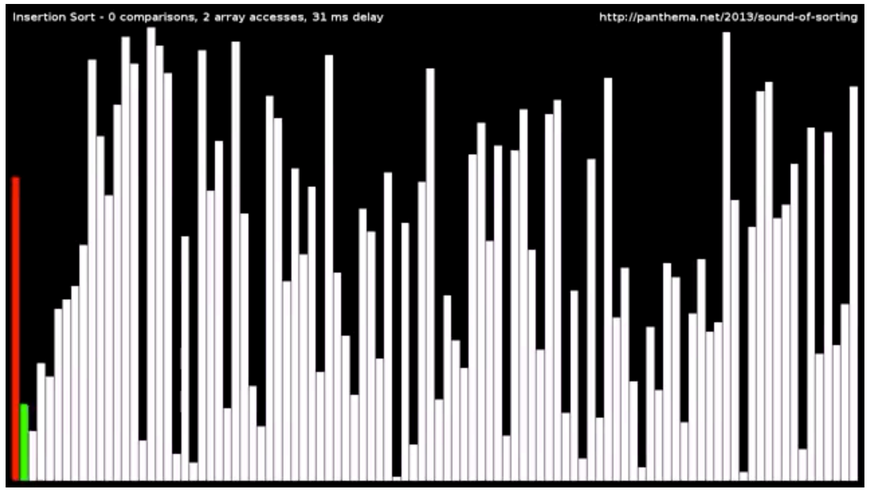
échanger les valeurs de `a.(i)` et `a.(j)`

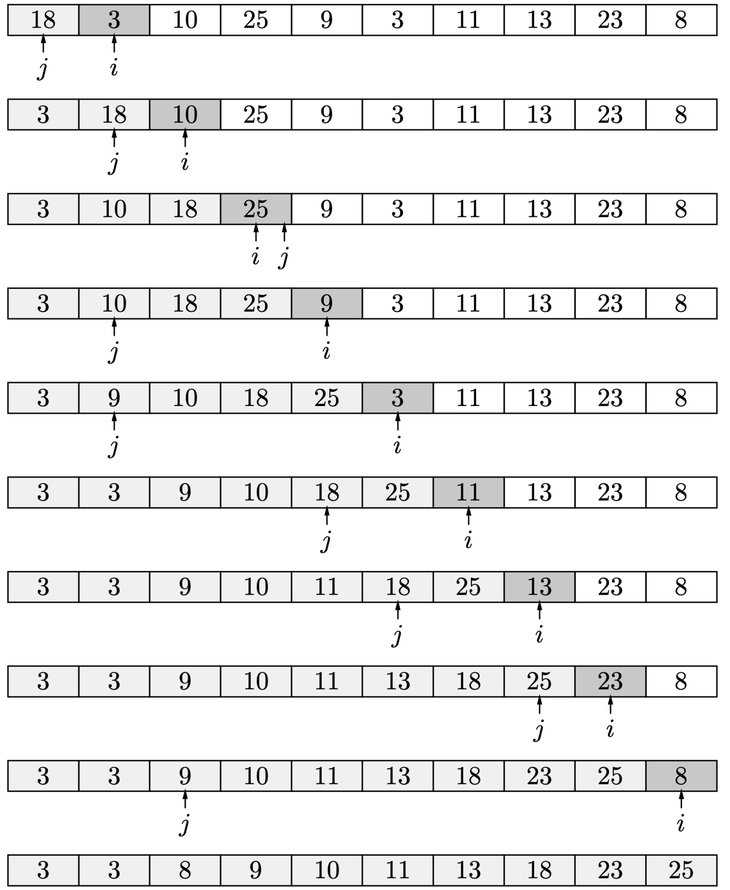
```
let print_array a =
  Array.iter (Printf.printf "%d ") a;
  print_newline() ;;

let rand_array n p =
  Random.self_init ();
  Array.init n (fun i -> Random.int p) ;;
```

Tri par insertions

- on insère les éléments dans la partie gauche déjà triée, etc…… [comme dans un jeu de cartes]

```
let insertion_sort a =
  let n = Array.length a in
  for i = 1 to n-1 do
    let v = a.(i) and j = ref i in
    while (!j > 0) && (a.(!j-1) > v) do
      a.(!j) <- a.(!j-1);
      decr j
    done ;
    a.(!j) <- v
  done ;;
```

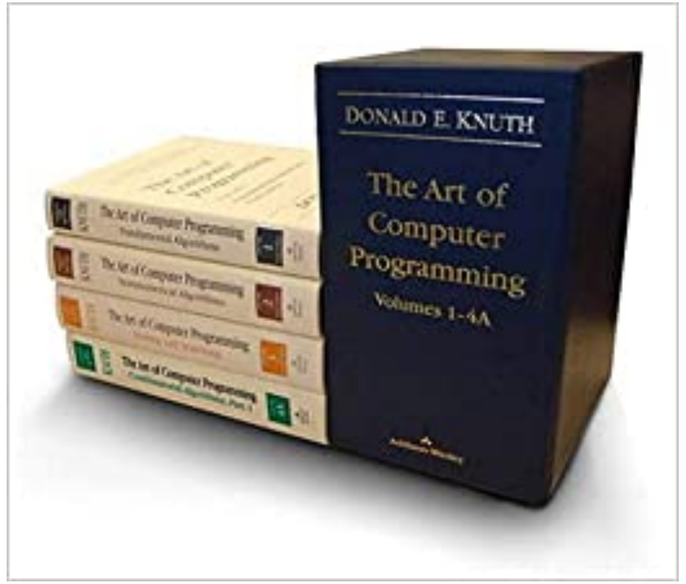


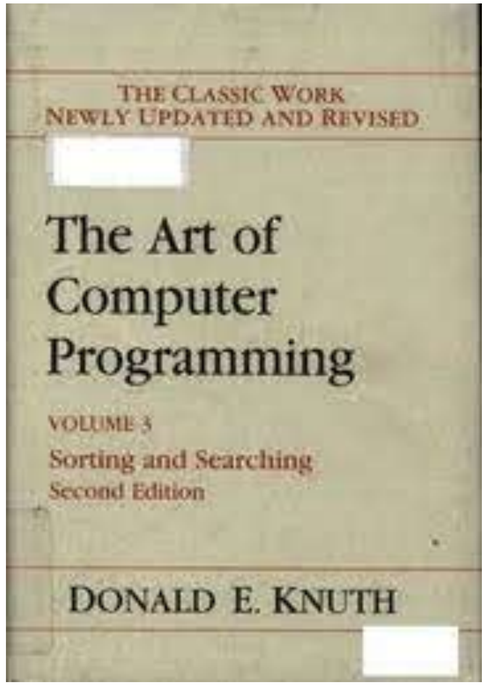


Exercices

Exercice 1 Ecrire le tri par insertions

Exercice 2 quel est le meilleur de ces 3 tris ?





Donald Knuth

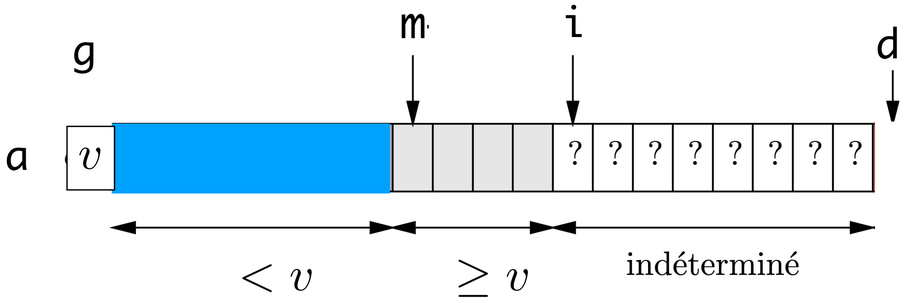
Exercice 3 compter le nombre d'opérations de ces tris en fonction de la longeur n du tableau à trier

Tri récursif

- tri rapide (*Quicksort*)

```
let quicksort a =
  let rec qsort 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);
        qsort a g (!m-1);
        qsort a !m d in
  qsort a 0 (Array.length a) ;;
```

- 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

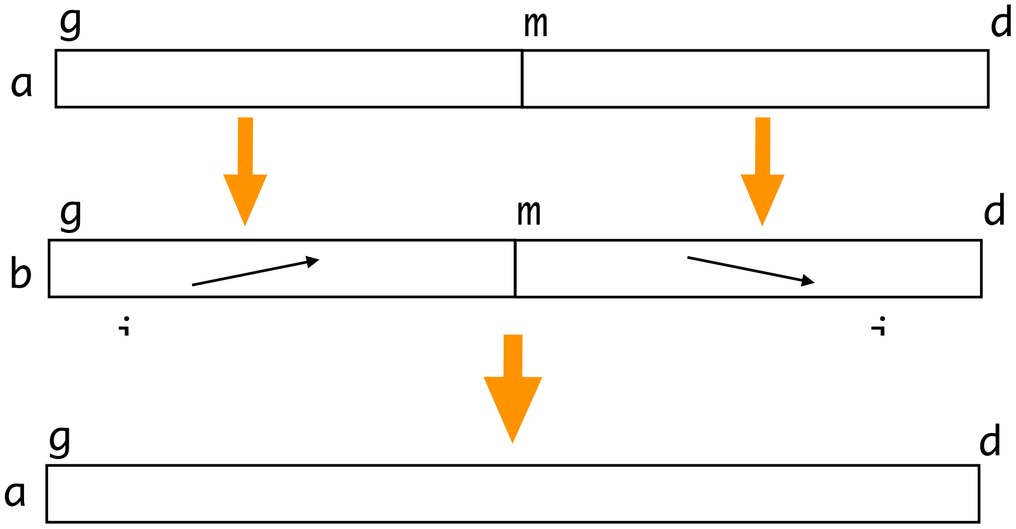
```
let xchange a i j =
  let tmp = a.(i) in
  a.(i) <- a.(j);
  a.(j) <- tmp ;;
```

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

Tri récursif

- tri fusion (*merge sort*)

```
let mergesort a =
    let rec msort a g d b =
    if g < d - 1 then
      let m = (g + d) / 2 in
      msort a g m b;
      msort 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 in
  let n = Array.length a in
  if n > 0 then
    let b = Array.make n a.(0) in
    msort a 0 n b ;;
```



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`

- très bonne méthode de tri

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" |];;

mergesort a  ->  [|15; 24; 34; 36; 44; 60; 72; 127; 147; 149]|

mergesort b  ->  [|2.; 2.3; 4.6|]

mergesort 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**

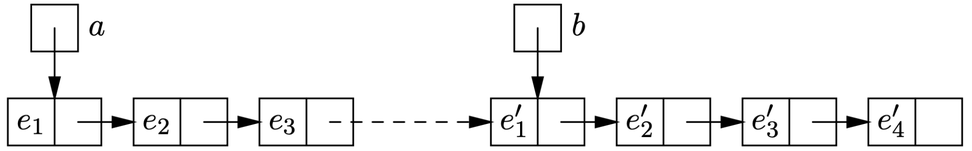
Listes modifiables

- un type `mlist` pour des listes modifiables

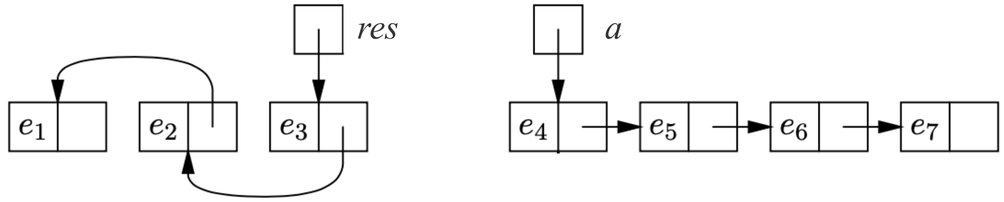
```
type 'a mlist = 'a cell option
and 'a cell = {value : 'a; mutable next : 'a mlist} ;;

let (@) x y = Some {value = x; next = y} ;;

let a = 3 @ 4 @ None ;;
let b = 5 @ 6 @ None ;;
```

- et on peut modifier ces listes

```
let rec nconc a b = match a with
  | None -> b
  | Some r -> r.next <- nconc (r.next) b; a ;;
```




```
let mirror a =
  let rec mirror1 res a = match a with
  | None -> res
  | Some r -> let a' = r.next in
      r.next <- res; mirror1 a a' in
  mirror1 None a;;
```



Arbres modifiables

- un type `arbreM` pour des arbres modifiables

```
type 'a arbreM = 'a noeud option
  and 'a noeud = {valeur : 'a; mutable gauche : 'a arbreM;
  mutable droite : 'a arbreM} ;;

let noeud x g d = Some {valeur = x; gauche = g; droite = d} ;;
```

- et on peut modifier ces arbres

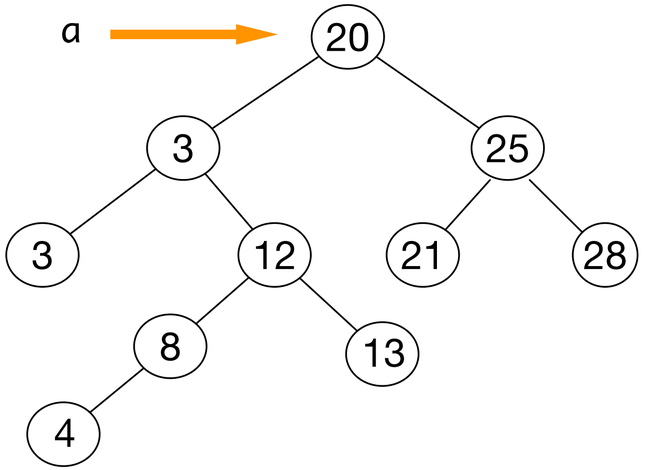
```
let rec ajouter x = function
  | None -> noeud x None None
  | Some r as a ->
      let {value = y; gauche = g; droite = d } = r in
      if x <= y then r.gauche <- ajouter x g
      else r.droite <- ajouter x d ;
      a ;;
```


```
type 'a arbreM = Feuille | Noeud of {valeur : 'a;
       mutable gauche : 'a arbre; mutable droite : 'a arbre} ;;

let noeud x g d = Noeud {valeur = x; gauche = g; droite = d} ;;


let rec ajouter x = function
| Feuille -> noeud x Feuille Feuille
| Noeud r as a ->
    if x <= r.valeur then r.gauche <- ajouter x r.gauche
    else r.droite <- ajouter x r.droite ;
    a ;;
```

Arbres modifiables

- un type `arbreM` pour des arbres modifiables

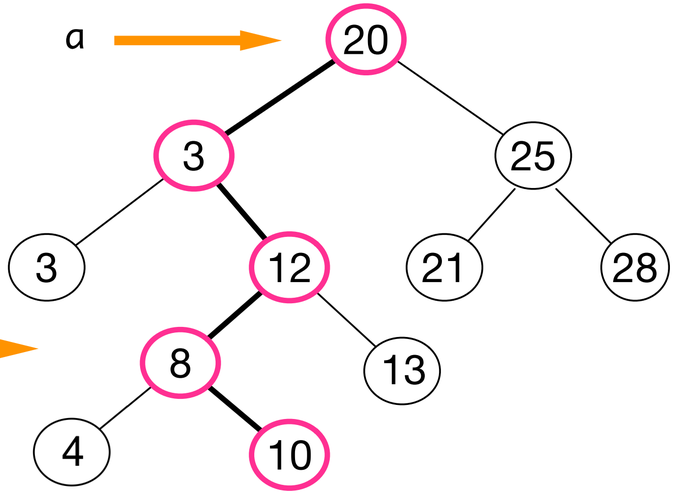
```
type 'a arbreM = 'a noeud option
  and 'a noeud = {valeur : 'a; mutable gauche : 'a arbreM;
  mutable droite : 'a arbreM} ;;

let noeud x g d = Some {valeur = x; gauche = g; droite = d} ;;
```



- et on peut modifier ces arbres

```
let rec ajouter x = function
  | None -> noeud x None None
  | Some r as a ->
      let {value = y; gauche = g; droite = d } = r in
      if x <= y then r.gauche <- ajouter x g
      else r.droite <- ajouter x d ;
      a ;;
```

on modifie l'arbre `a`, les noeuds **rouges** sont toujours les anciens



Polymorphisme des données modifiables

- un type faiblement polymorphe pour les données modifiables

```
(* val p : '_weak1 option ref = {contents = None} *)
let p = ref None;;

let plus x y = (p := Some x;
                x ) +
               (p := Some (string_of_int y) ;
                y ) ;;
(* Error: This expression has type string but an expression was expected of type
                int *)

(* val plus1 : int -> int -> int = <fun> *)
let plus1 x y =  (p := Some x; x ) + y ;;

(* - : int option ref = {contents = None} *)
p;;

let plus2 x y =  x + (p := Some (string_of_int y); y) ;;
(* Error: This expression has type string but an expression was expected of type
         int *)
```

p est une référence vers tout type de données

p devenu une référence vers un entier (optionnel)

- le type des données modifiables prend le type de leur première valeur

Conclusion

VU:

- récursivité
- listes
- filtrage
- arbres
- références
- données modifiables

TODO list

- modules
- modules