

École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 1 / 28 septembre 2015

- **cours** le lundi, 13h15–15h15 en salle W
 - photocopié = ensemble des transparents
- **TD** le lundi/vendredi, 15h30–17h00 en salle Info 4 NIR
 - avec Robin Morisset (robin.morisset@normalesup.org)
 - à partir d'aujourd'hui

toutes les infos sur le site web du cours

<http://www.lri.fr/~filliatr/ens/compil/>

questions ⇒ Jean-Christophe.Filliatre@lri.fr

- un **examen**, en janvier
 - anciens sujets+corrigés sur le site
- un **projet** = un mini compilateur
 - réalisé en dehors des TD, seul ou en binôme
 - rendu en deux fois (29 novembre, 10 janvier)

$$note\ finale = \frac{examen + projet}{2}$$

maîtriser les mécanismes de la **compilation**, c'est-à-dire de la transformation d'un langage dans un autre

comprendre les différents aspects des **langages de programmation** par le biais de la compilation

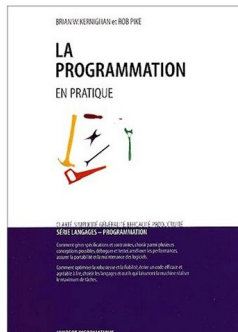
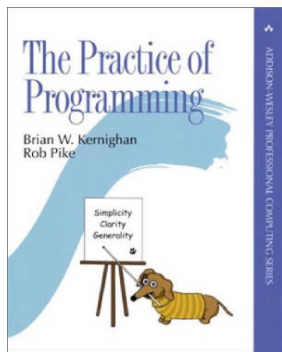
ici on programme

- en cours
- en TD
- pour réaliser le projet
- à l'examen

on programme en **OCaml**

il n'y a pas de bon langage, il n'y a que de bons programmeurs

lire **La programmation en pratique** de Brian Kernighan & Robert Pike



aujourd'hui :

Mise à niveau OCaml

il y a un polycopié spécifique pour ce cours

OCaml est un langage fonctionnel, fortement typé, généraliste



Successeur de Caml Light (lui-même successeur de « Caml lourd »)
De la famille ML

Conçu et implémenté à l'INRIA Rocquencourt par Xavier Leroy et d'autres

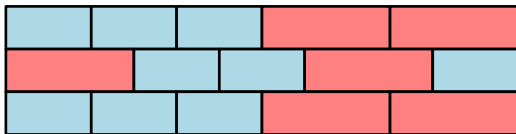
Quelques applications : calcul symbolique et langages (IBM, Intel, Dassault Systèmes), analyse statique (Microsoft, ENS), manipulation de fichiers (Unison, MLDonkey), finance (LexiFi, Jane Street Capital), enseignement

<http://ocaml.org/>

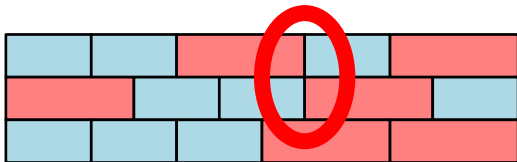
premiers pas en OCaml

on souhaite construire un mur avec des briques de longueur 2 () et de longueur 3 (), dont on dispose en quantités respectives infinies

voici par exemple un mur de longueur 12 et de hauteur 3 :



pour être solide, le mur ne doit jamais superposer deux jointures



combien y a-t-il de façons de construire un mur de longueur 32 et de hauteur 10 ?



on va calculer **récurivement** le nombre de façons $W(r, h)$ de construire un mur de hauteur h , dont la rangée de briques la plus basse r est donnée

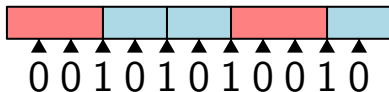
- cas de base

$$W(r, 1) = 1$$

- sinon

$$W(r, h) = \sum_{r' \text{ compatible avec } r} W(r', h - 1)$$

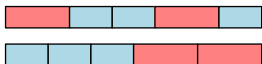
on va représenter les rangées de briques par des **entiers** en base 2 dont les chiffres 1 correspondent à la présence de jointures



ainsi cette rangée est représentée par l'entier 338 ($= 00101010010_2$)

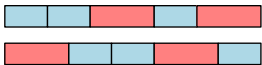
il est alors aisé de vérifier que deux rangées sont compatibles, par une simple opération de ET logique (`land` en OCaml)

ainsi




$$\begin{aligned}
 & 00101010010_2 \\
 \text{land } & 01010100100_2 \\
 = & 00000000000_2 = 0
 \end{aligned}$$

mais



$$\begin{aligned}
 & 01010010100_2 \\
 \text{land } & 00101010010_2 \\
 = & 000000\mathbf{1}0000_2 \neq 0
 \end{aligned}$$

écrivons une fonction `add2` qui ajoute une brique de longueur 2  à droite d'une rangée de briques `r`

il suffit de décaler les bits 2 fois vers la gauche et d'ajouter 10_2

de même on ajoute une brique  avec une fonction `add3`



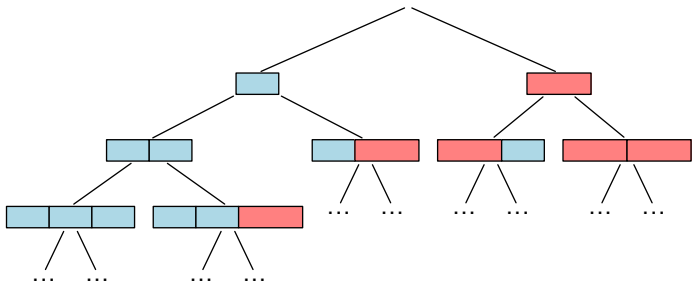
on va construire la **liste** de toutes les rangées de briques possibles de longueur 32

en OCaml, les listes sont construites à partir de

- la liste vide notée []
- l'ajout d'un élément x au début d'une liste l noté $x :: l$

énumérer toutes les rangées de briques

on va écrire une fonction récursive `fill` qui parcourt cet arbre



jusqu'à trouver des rangées de la bonne longueur



pour écrire la fonction récursive W , on commence par écrire une fonction `sum` qui calcule

$$\text{sum } f \ l = \sum_{x \in l} f(x)$$

c'est-à-dire

```
sum: (int -> int) -> int list -> int
```



on écrit enfin la fonction récursive W de décompte



et pour obtenir la solution du problème, il suffit de considérer toutes les rangées de base possibles



malheureusement, c'est beaucoup, beaucoup, beaucoup trop long. . .

le problème est qu'on retrouve très souvent les mêmes couples (r, h) en argument de la fonction W , et donc qu'on calcule plusieurs fois la même chose

d'où une troisième idée : stocker dans une table les résultats $W(r, h)$ déjà calculés → c'est ce qu'on appelle la **mémoïsation**

il nous faut donc une table d'association qui associe à certaines clés (r, h) la valeur $W(r, h)$

on va utiliser une **table de hachage**

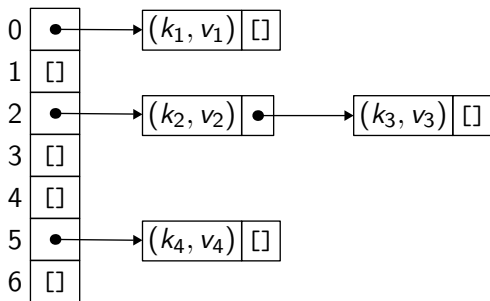
l'idée est très simple : on se donne une fonction

$$\textit{hash} : \textit{clé} \rightarrow \textit{int}$$

arbitraire à valeur dans $0..n - 1$ et un tableau de taille n

pour une clé k associée à une valeur v , on range le couple (k, v) dans la case du tableau $\textit{hash}(k)$

plusieurs clés peuvent se retrouver dans la même case
⇒ chaque case est une liste



on peut maintenant utiliser la table de hachage dans la fonction W

on va écrire deux fonctions `w` et `memo_w` **mutuellement récursives**

- `w` effectue le calcul, en appelant `memo_w` récursivement
- `memo_w` consulte la table, et si besoin appelle `w` pour la remplir



on obtient finalement le résultat

```
% ocamlpt wall.ml -o wall
% time ./wall
806844323190414

real 0m1.055s
```

si vous avez aimé ce problème...



[http ://projecteuler.net/](http://projecteuler.net/)

récapitulation

programme = suite de déclarations et d'expressions à évaluer

- interprétation, éventuellement interactive
- deux compilateurs (*bytecode* et natif)

variable OCaml :

1. nécessairement **initialisée**
2. type pas déclaré mais **inféré**
3. contenu **non modifiable**

une variable modifiable s'appelle une **référence**
elle est introduite avec `ref`

pas de distinction expression/instruction dans la syntaxe : **que des expressions**

toutes les expressions sont **typées**

les expressions sans réelle valeur (affectation, boucle, ...) ont pour type **unit** ; ce type a une unique valeur, notée **()**

- fonctions = valeurs comme les autres : locales, anonymes, arguments d'autres fonctions, etc.
- partiellement appliquées
- l'appel de fonction ne coûte pas cher
- polymorphes

OCaml infère toujours le type **le plus général possible**

exemple :

```
val sum: ('a -> int) -> 'a list -> int
```

où 'a représente une **variable de type**

dans la bibliothèque standard, on trouve

```
val List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
  (** List.fold_left f a [b1; ...; bn]
      is f (... (f (f a b1) b2) ...) bn *)
```

application

```
let sum f l = List.fold_left (fun s x -> s + f x) 0 l
```

on peut écrire le principe de mémoïsation de façon générique

```
let memo f =  
  let h = Hashtbl.create 5003 in  
  fun x ->  
    try Hashtbl.find h x  
    with Not_found -> let y = f x in Hashtbl.add h x y; y
```

```
val memo: ('a -> 'b) -> 'a -> 'b
```

utilisation

```
let is_prime n = ...  
let f = memo is_prime
```

ne convient cependant pas à une fonction récursive

```
let rec f x = ...  
let g = memo f
```

n'aura pas l'efficacité espérée

car les appels récursifs dans `f` se font sur `f`, pas `g`

il faut ajouter f en argument de la fonction qui calcule $f\ x$

```
let memo ff =  
  let h = Hashtbl.create 5003 in  
  let rec f x =  
    try Hashtbl.find h x  
    with Not_found -> let y = ff f x in Hashtbl.add h x y; y  
  in  
  f
```

```
val memo: (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

c'est un opérateur de point fixe

utilisation

```
let w = memo (fun w (r, h) -> ...)
```


allocation mémoire

allocation mémoire réalisée par un **garbage collector** (GC)

Intérêts :

- récupération automatique
- allocation efficace

⇒ perdre le réflexe « allouer dynamiquement coûte cher »
... mais continuer à se soucier de complexité !

on a déjà vu les tableaux

- allocation

```
let a = Array.create 10 0
```

- accès

```
a.(1)
```

- affectation

```
a.(1) <- 5
```

comme dans beaucoup de langages

on déclare le type enregistrement

```
type complexe = { re: float; im: float }
```

allocation et initialisation simultanées :

```
let x = { re = 1.0; im = -1.0 }
```

accès avec la notation usuelle :

```
x.im
```

```
type personne = { nom: string; mutable age: int }
```

```
# let p = { nom = "Martin"; age = 23 };;
```

```
val p : personne = {nom = "Martin"; age = 23}
```

modification en place :

```
# p.age <- p.age + 1;;
```

```
- : unit = ()
```

```
# p.age;;
```

```
- : int = 24
```

référence = enregistrement du type suivant

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

`ref`, `!` et `:=` ne sont que du sucre syntaxique

les seules données modifiables en place sont les tableaux et les champs déclarés `mutable`

type prédéfini de listes, α `list`, immuables et homogènes
construites à partir de la liste vide `[]` et de l'ajout en tête `::`

```
# let l = 1 :: 2 :: 3 :: [];;
```

```
val l : int list = [1; 2; 3]
```

ou encore

```
# let l = [1; 2; 3];;
```

filtrage = construction par cas sur la forme d'une liste

```
# let rec somme l =  
  match l with  
  | [] -> 0  
  | x :: r -> x + somme r;;
```

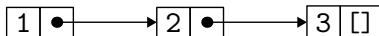
```
val somme : int list -> int = <fun>
```

notation plus compacte pour une fonction filtrant son argument

```
let rec somme = function  
  | [] -> 0  
  | x :: r -> x + somme r;;
```


listes OCaml = mêmes listes chaînées qu'en C ou Java

la liste `[1; 2; 3]` correspond à



listes = cas particulier de **types construits**

type construit = réunion de plusieurs **constructeurs**

```
type formule = Vrai | Faux | Conjonction of formule * formule
```

```
# Vrai;;
```

```
- : formule = Vrai
```

```
# Conjonction (Vrai, Faux);;
```

```
- : formule = Conjonction (Vrai, Faux)
```

listes définies par

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

le filtrage se généralise

```
# let rec evaluate = function
  | Vrai -> true
  | Faux -> false
  | Conjonction (f1, f2) -> evaluate f1 && evaluate f2;;
```

```
val evaluate : formule -> bool = <fun>
```

les motifs peuvent être **imbriqués** :

```
let rec evaluate = function
  | Vrai -> true
  | Faux -> false
  | Conjonction (Faux, f2) -> false
  | Conjonction (f1, Faux) -> false
  | Conjonction (f1, f2) -> evaluate f1 && evaluate f2;;
```

les motifs peuvent être **omis** ou **regroupés**

```
let rec evaluate = function
  | Vrai -> true
  | Faux -> false
  | Conjonction (Faux, _) | Conjonction (_, Faux) -> false
  | Conjonction (f1, f2) -> evaluate f1 && evaluate f2;;
```

le filtrage n'est pas limité aux types construits

```
let rec mult = function
  | [] -> 1
  | 0 :: _ -> 0
  | x :: l -> x * mult l
```

on peut écrire `let motif = expression` lorsqu'il y a un seul motif
(comme dans `let (a,b,c,d) = v` par exemple)

- allouer ne coûte pas cher
- libération automatique
- valeurs allouées nécessairement initialisées
- majorité des valeurs **non** modifiables en place (seuls tableaux et champs d'enregistrements mutable)
- représentation mémoire efficace des valeurs construites
- filtrage = examen par cas sur les valeurs construites

exceptions

c'est la notion usuelle

une exception peut être **levée** avec **raise**

```
let division n m =  
  if m = 0 then raise Division_by_zero else ...
```

et **rattrapée** avec **try with**

```
try division x y with Division_by_zero -> (0,0)
```

on peut déclarer de nouvelles exceptions

```
exception Error  
exception Unix_error of string
```

exception utilisée pour un résultat exceptionnel

```
try find table clé  
with Not_found -> ...
```

exception utilisée pour modifier le flot de contrôle

```
try
  while true do
    let key = read_key () in
    if key = 'q' then raise Exit;
    ...
  done
with Exit ->
  close_graph (); exit 0
```

modules et foncteurs

lorsque les programmes deviennent gros il faut

- découper en unités (**modularité**)
- occulter la représentation de certaines données (**encapsulation**)
- éviter au mieux la duplication de code

en OCaml : fonctionnalités apportées par les **modules**

chaque fichier est un module

si arith.ml contient

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

alors on le compile avec

```
% ocamlc -c arith.ml
```

utilisation dans un autre module main.ml :

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlc -c main.ml
```

```
% ocamlc arith.cmo main.cmo
```

on peut restreindre les valeurs exportées avec une **interface**

dans un fichier `arith.mli`

```
val round: float -> float
```

```
% ocamlc -c arith.mli
```

```
% ocamlc -c arith.ml
```

```
% ocamlc -c main.ml
```

```
File "main.ml", line 2, characters 33-41:
```

```
Unbound value Arith.pi
```

une interface peut restreindre la visibilité de la **définition** d'un type

dans `ensemble.ml`

```
type t = int list
let vide = []
let ajoute x l = x :: l
let appartient = List.mem
```

mais dans `ensemble.mli`

```
type t
val vide: t
val ajoute: int -> t -> t
val appartient: int -> t -> bool
```

le type `t` est un **type abstrait**

la compilation d'un fichier ne dépend **que des interfaces** des fichiers utilisés

⇒ **moins de recompilation** quand un code change mais pas son interface

modules non restreints aux fichiers

```
module M = struct
  let c = 100
  let f x = c * x
end
```

```
module A = struct
  let a = 2
  module B = struct
    let b = 3
    let f x = a * b * x
  end
  let f x = B.f (x + 1)
end
```

de même pour les signatures

```
module type S = sig
  val f: int -> int
end
```

contrainte

```
module M: S = struct
  let a = 2
  let f x = a * x
end
```

```
# M.a;;
```

```
Unbound value M.a
```

- modularité par découpage du code en unités appelées **modules**
- encapsulation de types et de valeurs, **types abstraits**
- vraie **compilation séparée**
- organisation de l'**espace de nommage**

foncteur = **module paramétré** par un ou plusieurs autres modules

exemple : table de hachage **générique**

il faut paramétrer par rapport aux fonctions de hachage et d'égalité

passer les fonctions en argument :

```
type ('a, 'b) t

val create: int -> ('a, 'b) t

val add: ('a -> int) -> ('a, 'b) t -> 'a -> 'b -> unit

val find: ('a -> int) -> ('a -> 'a -> bool) ->
  ('a, 'b) t -> 'a -> 'b
```

problème : il faut être cohérent dans l'utilisation

passer les fonctions à la création :

```
type ('a, 'b) t
```

```
val create: ('a -> int) -> ('a -> 'a -> bool) ->  
  int -> ('a, 'b) t
```

```
val add: ('a, 'b) t -> 'a -> 'b -> unit
```

```
val find: ('a, 'b) t -> 'a -> 'b
```

```
type ('a, 'b) t = {  
  hash    : 'a -> int;  
  eq      : 'a -> 'a -> bool;  
  buckets: ('a * 'b) list array }  
  
let create hash eq n =  
  { hash = hash; eq = eq; buckets = Array.create n [] }  
  
let add t k v =  
  let i = (t.hash k) mod (Array.length t.buckets) in  
  t.buckets.(i) <- (k, v) :: t.buckets.(i)  
  
let find t k =  
  let rec lookup = function  
    | [] -> raise Not_found  
    | (k', v) :: l -> if t.eq k' k then v else lookup l  
  in  
  lookup t.buckets.((t.hash k) mod (Array.length t.buckets))
```


problèmes

- on ne peut plus comparer (avec =) ou stocker sur le disque les tables de hachage
- on n'a pas toujours une structure pour y stocker les fonctions

```
module HashTable(K: KEY) = struct ... end
```

avec

```
module type KEY = sig
  type key
  val hash: key -> int
  val eq: key -> key -> bool
end
```

```
module HashTable(K: KEY) = struct
  type 'a t = (K.key * 'a) list array
  let create n = Array.create n []
  let add t k v =
    let i = (K.hash k) mod (Array.length t) in
    t.(i) <- (k, v) :: t.(i)
  let find t k =
    let rec lookup = function
      | [] -> raise Not_found
      | (k', v) :: l -> if K.eq k' k then v else lookup l
    in
    lookup t.((K.hash k) mod (Array.length t))
end
```

```
module HashTable(K: KEY): sig
  type 'a t
  val create: int -> 'a t
  val add: 'a t -> K.key -> 'a -> unit
  val find: 'a t -> K.key -> 'a
end
```

```
module Int = struct
  type key = int
  let hash = abs
  let eq = (=)
end
```

```
module Hint = HashTable(Int)
```

```
# let t = Hint.create 17;;
```

```
val t : '_a Hint.t = <abstr>
```

(on ne connaît pas encore le type des valeurs, d'où le `'_a`)

```
# Hint.add t 13 "foo";;
```

```
- : unit = ()
```

```
# Hint.add t 173 "bar";;
```

```
- : unit = ()
```

remarque : maintenant on connaît le type des valeurs

```
# t;;
```

```
- : string Hint.t = <abstr>
```

```
# Hint.find t 13;;
```

```
- : string = "foo"
```

```
# Hint.find t 14;;
```

```
- : Exception: Not_found.
```

1. structures de données paramétrées par d'autres structures de données

- `Hashtbl.Make` : tables de hachage
- `Set.Make` : ensembles finis codés par des arbres équilibrés
- `Map.Make` : tables d'association codées par des arbres équilibrés

2. algorithmes paramétrés par des structures de données

exemple : algorithme de recherche de plus court chemin


```
module DijkstraShortestPath
  (G: sig
    type graph
    type node
    val successors: graph -> node -> (node * int) list
  end) :
  sig
    val shortest_path:
      G.graph -> G.node -> G.node -> G.node list * int
  end
```

écrire un foncteur pour calculer modulo m

```
module Modular(M: sig val m: int end): sig
  type t
  val of_int: int -> t
  val add: t -> t -> t
  val sub: t -> t -> t
  val mul: t -> t -> t
  val div: t -> t -> t
  (** divise x par y, en supposant y premier avec m *)
  val to_int: t -> int
end
```

il faut résister à la tentation de tout généraliser

suggestion : écrire un foncteur si

- il s'agit d'une bibliothèque générique (dont on ne connaît pas les clients)
- on a soi-même au moins deux instances à en faire

persistance

en OCaml, la majorité des structures de données sont **immuables**
(seules exceptions : tableaux et enregistrements à champ mutable)

dit autrement :

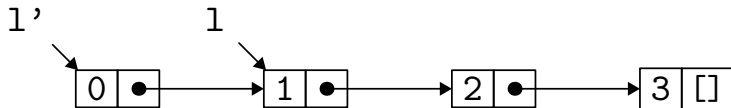
- une valeur n'est pas affectée par une opération,
- mais une **nouvelle** valeur est renvoyée

vocabulaire : on parle de code **purement applicatif** ou encore simplement de **code pur** (parfois aussi de code **purement fonctionnel**)

exemple de structure immuable : les listes

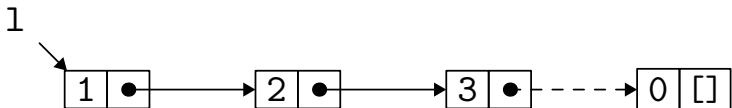
```
let l = [1; 2; 3]
```

```
let l' = 0 :: l
```



pas de copie, mais **partage**

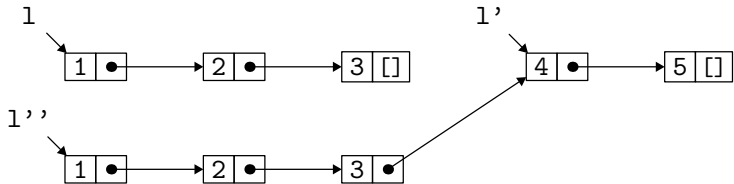
un ajout en queue de liste n'est pas aussi simple :



concaténation de deux listes

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

```
let l   = [1; 2; 3]  
let l'  = [4; 5]  
let l'' = append l l'
```



blocs de l **copiés**, blocs de l' **partagés**

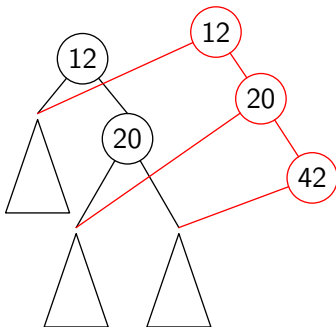
note : on peut définir des listes chaînées « traditionnelles », par exemple ainsi

```
type 'a liste = Vide | Element of 'a element
and 'a element = { valeur: 'a; mutable suivant: 'a liste }
```

mais alors il faut faire attention au **partage** (*aliasing*)

```
type tree = Empty | Node of int * tree * tree
```

```
val add: int -> tree -> tree
```



là encore, peu de copie et beaucoup de **partage**

1. **correction** des programmes
 - code plus simple
 - raisonnement mathématique possible
2. outil puissant pour le **branchement**
 - algorithmes de recherche
 - manipulations symboliques et portées
 - rétablissement suite à une erreur

recherche de la sortie dans un labyrinthe

```
type état
val sortie: état -> bool
type déplacement
val déplacements: état -> déplacement list
val déplace: état -> déplacement -> état
```

```
let rec cherche e =
  sortie e || essaye e (déplacements e)
and essaye e = function
  | [] -> false
  | d :: r -> cherche (déplace d e) || essaye e r
```

avec un état global modifié en place :

```
let rec cherche () =  
  sortie () || essaye (déplacements ())  
and essaye = function  
  | [] ->  
    false  
  | d :: r ->  
    (déplace d; cherche ()) || (revient d; essaye r)
```

i.e. il faut **annuler** l'effet de bord (*undo*)

programmes très simples, représentés par

```
type instr =  
  | Return of string  
  | Var of string * int  
  | If of string * string * instr list * instr list
```

exemple :

```
int x = 1;  
int z = 2;  
if (x == z) {  
  int y = 2;  
  if (y == z) return y; else return z;  
} else  
return x;
```

on veut vérifier que toute variable utilisée est auparavant déclarée
(dans une liste d'instructions)

```
val vérifie_instr: string list -> instr -> bool
val vérifie_prog : string list -> instr list -> bool
```

```
let rec vérifie_instr vars = function
  | Return x ->
    List.mem x vars
  | If (x, y, p1, p2) ->
    List.mem x vars && List.mem y vars &&
    vérifie_prog vars p1 && vérifie_prog vars p2
  | Var _ ->
    true
```

```
and vérifie_prog vars = function
  | [] ->
    true
  | Var (x, _) :: p ->
    vérifie_prog (x :: vars) p
  | i :: p ->
    vérifie_instr vars i && vérifie_prog vars p
```


programme manipulant **une** base de données

mise à jour complexe, nécessitant de nombreuses opérations

avec une structure modifiée en place

```
try
  ... effectuer l'opération de mise à jour ...
with e ->
  ... rétablir la base dans un état cohérent ...
  ... traiter ensuite l'erreur ...
```

avec une structure persistante

```
let bd = ref (... base initiale ...)  
...  
try  
  bd := (... opération de mise à jour de !bd ...)  
with e ->  
  ... traiter l'erreur ...
```

le caractère persistant d'un type abstrait n'est pas évident

la signature fournit l'information **implicitement**

structure modifiée en place

```
type t
val create: unit -> t
val add: int -> t -> unit
val remove: int -> t -> unit
...
```

structure persistante

```
type t
val empty: t
val add: int -> t -> t
val remove: int -> t -> t
...
```

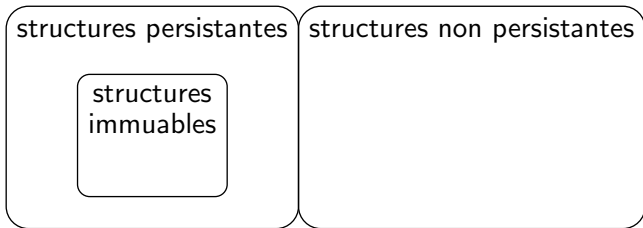
persistant ne signifie pas sans effet de bord

persistant = observationnellement immuable

on a seulement l'implication dans un sens :

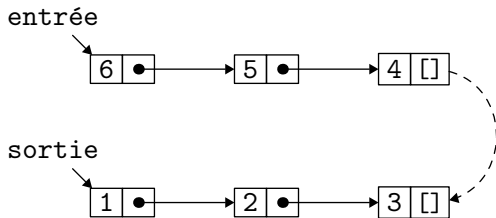
immuable \Rightarrow persistant

la réciproque est fausse



```
type 'a t
val create: unit -> 'a t
val push: 'a -> 'a t -> 'a t
exception Empty
val pop: 'a t -> 'a * 'a t
```

idée : représenter la file par une **paire de listes**,
une pour l'entrée de la file, une pour la sortie



représente la file $\rightarrow 6, 5, 4, 3, 2, 1 \rightarrow$

```
type 'a t = 'a list * 'a list

let create () = [], []

let push x (e,s) = (x :: e, s)

exception Empty

let pop = function
  | e, x :: s -> x, (e,s)
  | e, [] -> match List.rev e with
    | x :: s -> x, ([], s)
    | [] -> raise Empty
```

si on accède plusieurs fois à une même file dont la seconde liste `e` est vide, on calculera plusieurs fois le même `List.rev e`

ajoutons une référence pour pouvoir enregistrer ce retournement de liste la première fois qu'il est fait

```
type 'a t = ('a list * 'a list) ref
```

l'effet de bord sera fait « sous le capot », à l'insu de l'utilisateur, sans modifier le contenu de la file (effet caché)


```
let create () = ref ([], [])
```

```
let push x q = let e,s = !q in ref (x :: e, s)
```

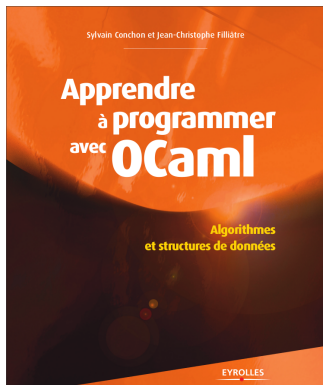
```
exception Empty
```

```
let pop q = match !q with  
  | e, x :: s -> x, ref (e,s)  
  | e, [] -> match List.rev e with  
    | x :: s as r -> q := ([], r); x, ref ([], s)  
    | [] -> raise Empty
```

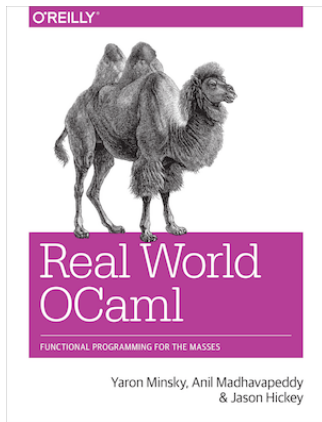
- structure persistante = pas de modification observable
 - en OCaml : `List`, `Set`, `Map`
- peut être très efficace (beaucoup de partage, voire des effets cachés, mais pas de copies)
- notion indépendante de la programmation fonctionnelle

beaucoup de ressources sur <http://ocaml.org/>

deux ouvrages



(à la bibliothèque de l'ENS)



(disponible en ligne)

- TD cette semaine
 - exercices de mise à niveau OCaml
 - il y en aura pour tous les niveaux
- Cours lundi prochain
 - Principes de la compilation
 - Assembleur X86-64
 - Exemple de compilation d'un petit programme

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```