

École Normale Supérieure

# Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 1 / 29 septembre 2011

- **cours** le jeudi, 16h45–19h00 en salle R
  - photocopié = ensemble des transparents
- **TD** le mercredi, 16h30–18h30 en salle Info 4 NIR
  - avec Louis Mandel (Louis.Mandel@lri.fr)
  - à partir du 5 octobre
- **évaluation**
  - un examen
  - un projet : réalisé en dehors des TD, en binômes

toutes les infos sur le site web du cours

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

questions ⇒ Jean-Christophe.Filliatre@lri.fr

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 **Objective Caml**

aujourd'hui :

## Mise à niveau Caml

il y a un polycopié spécifique pour ce cours

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

Objective Caml 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

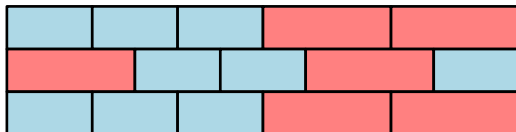
# premiers pas en Caml

---

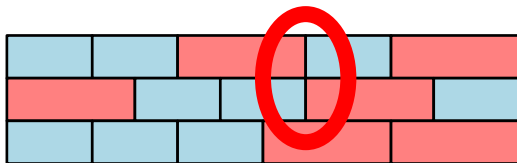
## un peu de maçonnerie

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

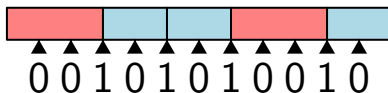
- cas de base

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

- 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

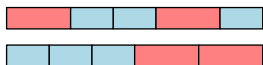


ainsi cette rangée est représentée par l'entier 338 ( $= 00101010010_2$ )

# quel intérêt ?

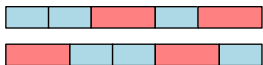
il est alors aisé de vérifier que deux rangées sont compatibles, par une simple opération de ET logique (`land` en OCaml)

ainsi




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


mais



$$\begin{aligned} & 01010010100_2 \\ \text{land} & 00101010010_2 \\ = & 00000010000_2 \neq 0 \end{aligned}$$

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`



# énumérer toutes les rangées de briques

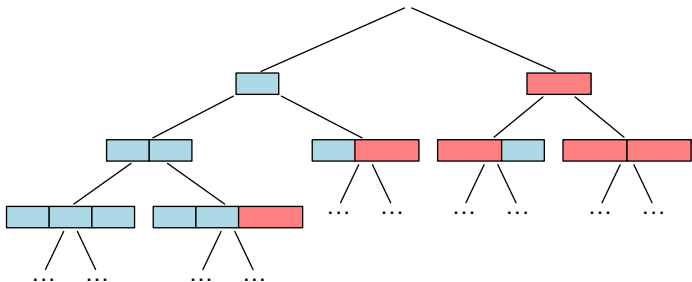
on va construire la **liste** de toutes les rangées de briques possibles de longueur 32

en Caml, 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**

# quel genre de table ?

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

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

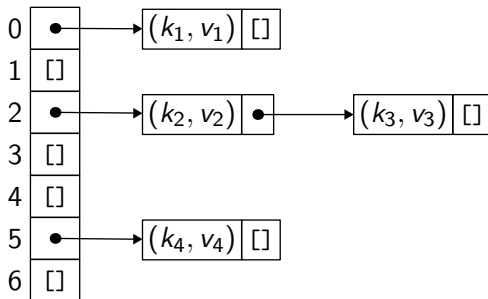
**arbitraire** et un tableau de taille  $n$

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

note : plusieurs clés peuvent se retrouver dans la même case  $\Rightarrow$  chaque case est une liste

# table de hachage

ainsi si  $n = 7$ ,  $\text{hash}(k_1) = 0 \pmod{7}$ ,  $\text{hash}(k_2) = \text{hash}(k_3) = 2 \pmod{7}$  et  $\text{hash}(k_4) = 5 \pmod{7}$ , alors



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

on va écrire deux fonctions  $w$  et  $\text{memo\_}w$  **mutuellement récursives**

- $w$  effectue le calcul, en appelant  $\text{memo\_}w$  récursivement
- $\text{memo\_}w$  consulte la table, et si besoin appelle  $w$  pour la remplir



le calcul ne prend plus que quelques secondes

mais il provoque un dépassement de capacité des entiers 32 bits (non signalé)  $\Rightarrow$  il faut calculer avec des **entiers 64 bits**

c'est-à-dire

- remplacer 0 par 0L et 1 par 1L
- remplacer + par Int64.add



on obtient finalement le résultat

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

real 0m6.759s
```

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 Caml :

- 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

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

exemple :

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

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

# 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
```

# enregistrements

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
```

## champs modifiables en place

```
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  $\alpha$  ref = { mutable contents :  $\alpha$  }
```

**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,  $\alpha$  `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;;
```

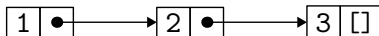
```
val somme : int list -> int = <fun>
```

notation plus compacte pour une fonction filtrant son argument

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

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

la liste `[1; 2; 3]` correspond à



# types construits

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  $\alpha$  list = [] | :: of  $\alpha \times \alpha$  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 evalue = function
  | Vrai → true
  | Faux → false
  | Conjonction (Faux, _) | Conjonction (_, Faux) → false
  | Conjonction (f1, f2) → evalue f1 && evalue 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 des valeurs construites efficace
- 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
```

## exceptions (exemple 1)

exception utilisée pour un résultat exceptionnel

```
try Hashtbl.find table clé  
with Not_found → ...
```

## exceptions (exemple 2)

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 Caml : fonctionnalités apportées par les **modules**

# fichiers et 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
```

## encapsulation (suite)

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

# langage de modules

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
```

# langage de modules

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 à la fonction de hachage et la fonction d'égalité

passer les fonctions en argument :

```
type  $\alpha$  t
```

```
val create : int  $\rightarrow$   $\alpha$  t
```

```
val add : ( $\alpha \rightarrow$  int)  $\rightarrow$   $\alpha$  t  $\rightarrow$   $\alpha \rightarrow$  unit
```

```
val mem : ( $\alpha \rightarrow$  int)  $\rightarrow$  ( $\alpha \rightarrow$   $\alpha \rightarrow$  bool)  $\rightarrow$   
   $\alpha$  t  $\rightarrow$   $\alpha \rightarrow$  bool
```

## deuxième solution

passer les fonctions à la création :

```
type  $\alpha$  t
```

```
val create : ( $\alpha \rightarrow$  int)  $\rightarrow$  ( $\alpha \rightarrow \alpha \rightarrow$  bool)  $\rightarrow$  int  $\rightarrow \alpha$  t
```

```
val add :  $\alpha$  t  $\rightarrow \alpha \rightarrow$  unit
```

```
val mem :  $\alpha$  t  $\rightarrow \alpha \rightarrow$  bool
```

```
type  $\alpha$  t = { hash :  $\alpha \rightarrow$  int;  
            eq :  $\alpha \rightarrow \alpha \rightarrow$  bool;  
            data :  $\alpha$  list array }
```

```
let create h eq n =  
  { hash = h; eq = eq; data = Array.create n [] }
```

```
...
```

# la bonne solution : un foncteur

```
module F(X : S) = struct ... end
```

avec

```
module type S = sig
  type elt
  val hash : elt → int
  val eq : elt → elt → bool
end
```

```
module F(X : S) = struct
  type t = X.elt list array
  let create n = Array.create n []
  let add t x =
    let i = (X.hash x) mod (Array.length t) in
    t.(i) ← x :: t.(i)
  let mem t x =
    let i = (X.hash x) mod (Array.length t) in
    List.exists (X.eq x) t.(i)
end
```

## foncteur : l'interface

```
module F(X : S) : sig
  type t
  val create : int → t
  val add : t → X.elts → unit
  val mem : t → X.elts → bool
end
```

# foncteur : utilisation

```
module Entiers = struct
  type elt = int
  let hash x = abs x
  let eq x y = x=y
end
```

```
module Hentiers = F(Entiers)
```

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

```
val t : Hentiers.t = <abstr>
```

```
# Hentiers.add t 13;;
```

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

```
# Hentiers.add t 173;;
```

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

## 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 Dijkstra de recherche de plus court chemin écrit indépendamment de la structure de graphes

# algorithme de Dijkstra « générique »

```
module Dijkstra
  (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
```

# persistance

---

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

dit autrement :

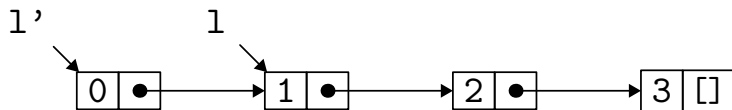
- une valeur n'est pas affectée par une opération,
- mais une **nouvelle** valeur est renvoyée

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

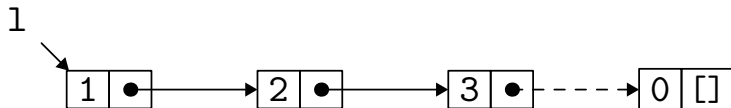
```
let l = [1; 2; 3]
```

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



**pas de copie**, mais **partage**

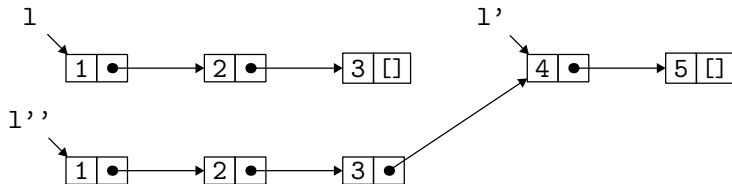
un ajout en queue de liste n'est pas aussi simple :



# 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

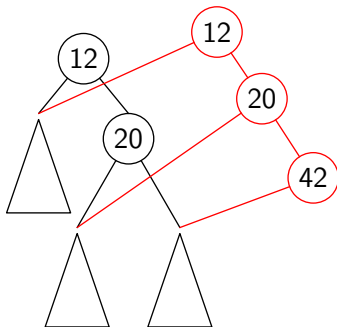
```
type  $\alpha$  liste = Vide | Element of  $\alpha$  element  
and  $\alpha$  element = { valeur :  $\alpha$ ; mutable suivant :  $\alpha$  liste }
```

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

## Autre exemple : les arbres

```
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

# persistence et branchement (1)

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*)

## persistence et branchement (2)

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;
```

## persistance et branchement (2)

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
```

## persistence et branchement (2)

```
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
```

## persistance et branchement (3)

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 ...
```

## persistance et branchement (3)

avec une structure persistante

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

# interface et persistance

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
...
```

# persistance et effets de bords

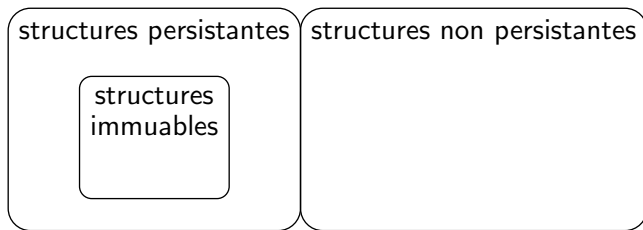
persistant ne signifie pas sans effet de bord

*persistant = observationnellement immuable*

on a seulement l'implication dans un sens :

*immuable*  $\Rightarrow$  *persistant*

la réciproque est fausse

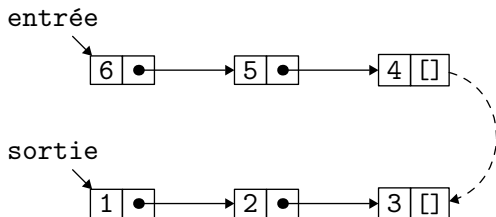


## exemple : files persistantes

```
type  $\alpha$  t
val create : unit  $\rightarrow$   $\alpha$  t
val push :  $\alpha \rightarrow \alpha$  t  $\rightarrow$   $\alpha$  t
exception Empty
val pop :  $\alpha$  t  $\rightarrow$   $\alpha \times \alpha$  t
```

## exemple : files persistantes

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



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

## exemple : files persistantes

```
type  $\alpha$  t =  $\alpha$  list  $\times$   $\alpha$  list

let create () = [], []

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

exception Empty

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

## exemple : files persistantes

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  $\alpha$  t = ( $\alpha$  list  $\times$   $\alpha$  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)

## exemple : files persistantes

```
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

- TD de mercredi
  - exercices de mise à niveau Caml
  - il y en aura pour tous les niveaux
- Cours de jeudi
  - Principes de la compilation
  - Assembleur MIPS
  - 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));}
```