École Normale Supérieure

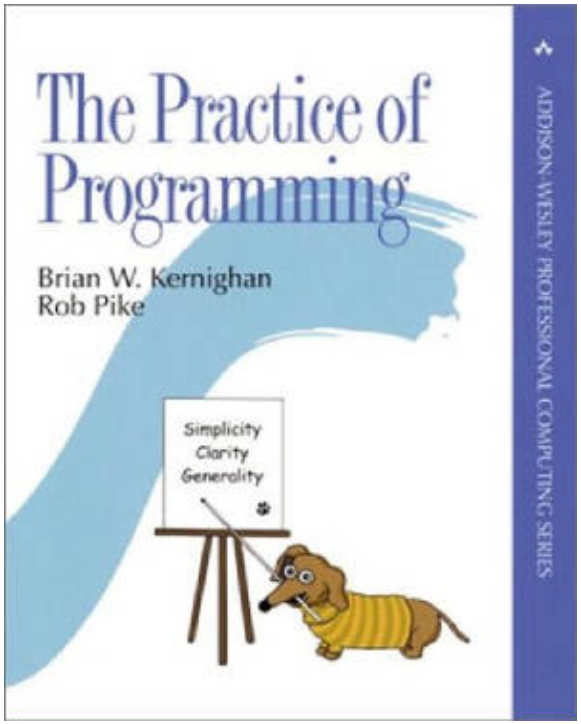
Le langage OCaml

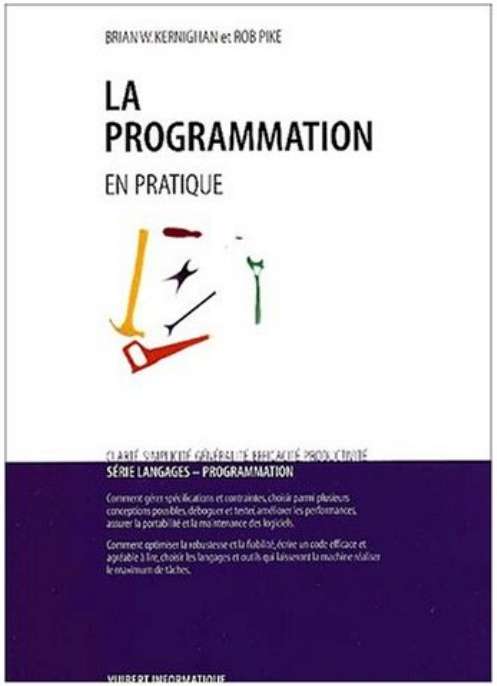
Jean-Christophe Filliâtre

Semaine *Informatique pratique*, septembre 2025

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





(disponible à la bibliothèque)

- leçon
  - 8h30–11h45 en salle E. Noether
- travaux pratiques
  - 13h15–17h en salles info 3 & 4
  - avec Samuel Vivien et moi-même
  - pour tous les niveaux

disponible sur moodle
`https://moodle.psl.eu/course/view.php?id=34902`

- ces transparents
- un polycopié de 50 pages
- comment installer un environnement OCaml
- travaux pratiques (avec corrigés)

OCaml est un langage fonctionnel, fortement typé, généraliste, de la famille ML

issu d'une longue lignée de travaux, notamment à l'Inria

- 1987 Caml
- 1990 Caml Light (Leroy, Doligez)
- 1995 Caml Special Light
- 1996 OCaml

plus de détails sur `http://ocaml.org/`

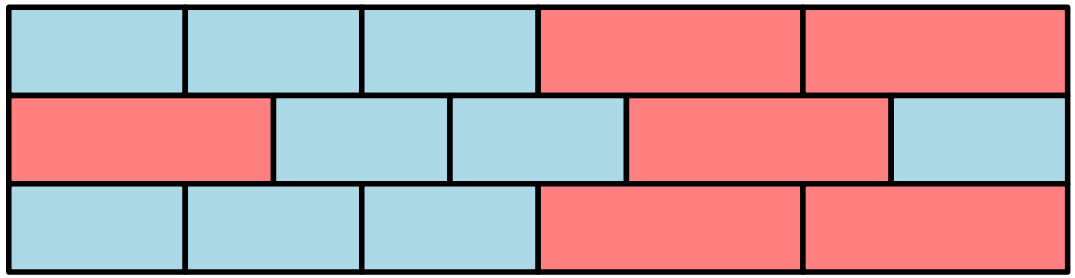
plan

1. premiers pas
2. types construits et filtrage
3. modules et foncteurs
4. persistance
5. écosystème

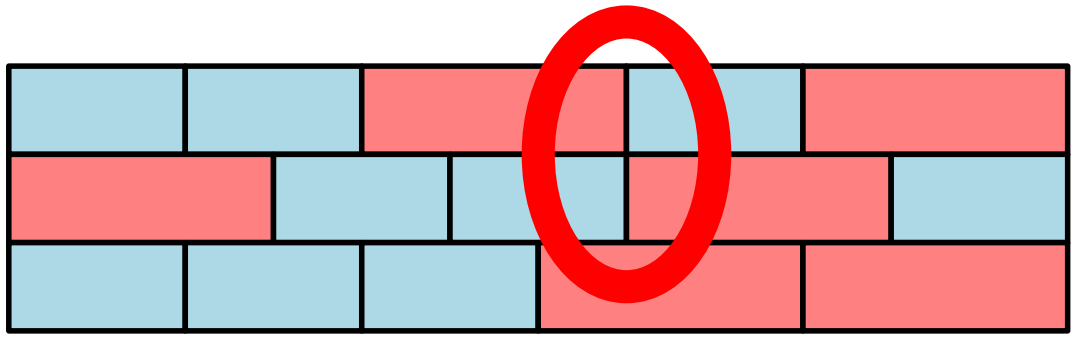
1 – premiers pas

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 **récursivement** le nombre de façons $C(r, h)$ de construire un mur de hauteur h, dont la rangée de briques la plus basse r est donnée

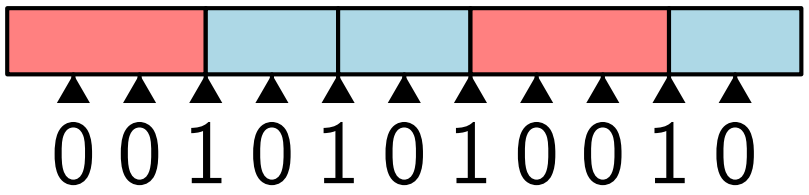
- cas de base

$$C(r, 1) = 1$$

- sinon

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

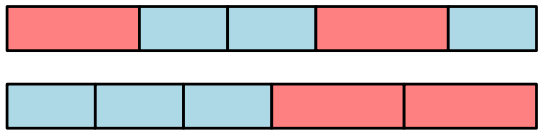
on va représenter les rangées de briques par des **entiers** en base 2 dont les chiffres 1 correspondent à la présence de jointures



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

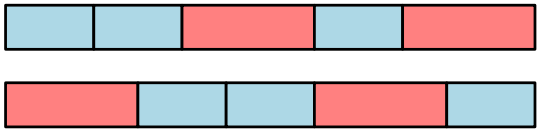
il est alors aisé de vérifier que deux rangées sont compatibles, par une simple opération de ET logique (`land` en OCaml)

ainsi



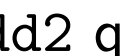
$$\begin{array}{rl} & 00101010010_2 \\ \texttt{land} & 01010100100_2 \\ = & 00000000000_2 = 0 \end{array}$$

mais



$$\begin{array}{rl} & 01010010100_2 \\ \texttt{land} & 00101010010_2 \\ = & 0000010000_2 \neq 0 \end{array}$$

rangées de briques

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

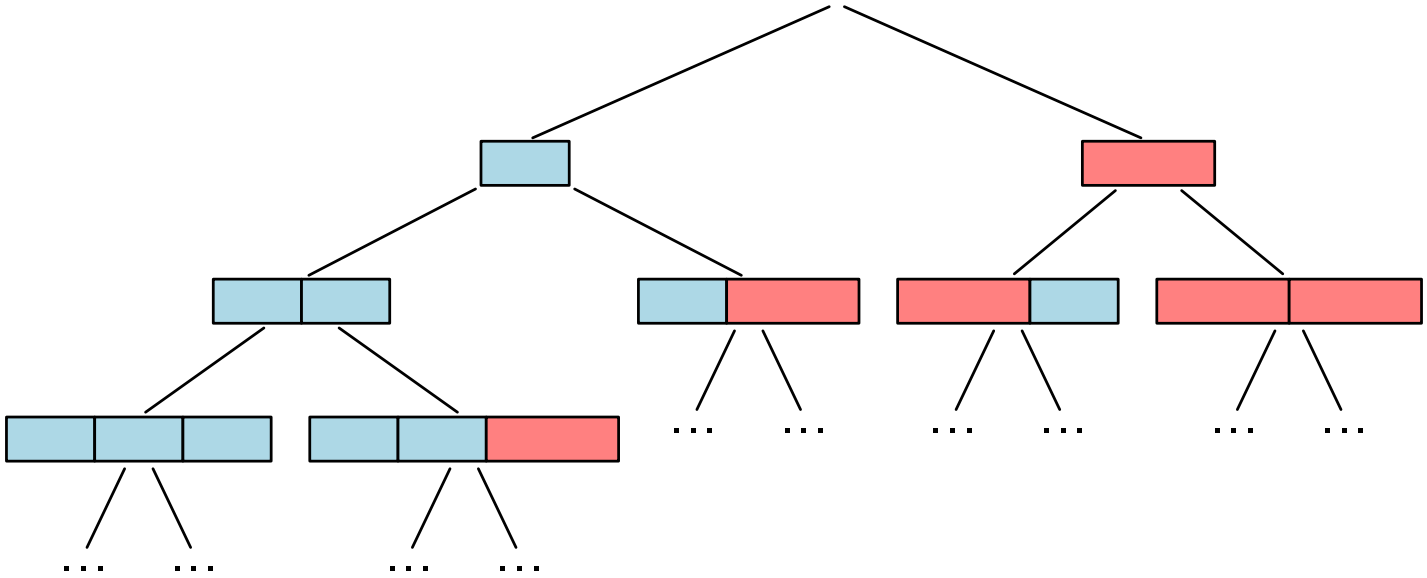
on va construire la **liste** de toutes les rangées de briques possibles de longueur 32

en 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 qui parcourt cet arbre (conceptuel)



jusqu'à trouver des rangées de la bonne longueur

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

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

c'est-à-dire

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

on écrit enfin la fonction récursive C de décompte

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

déception

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 C, et donc qu'on calcule plusieurs fois la même chose

d'où une troisième idée : stocker dans une table les résultats $C(r, h)$ déjà calculés $\rightarrow$ 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 $C(r, h)$

on va utiliser une **table de hachage**

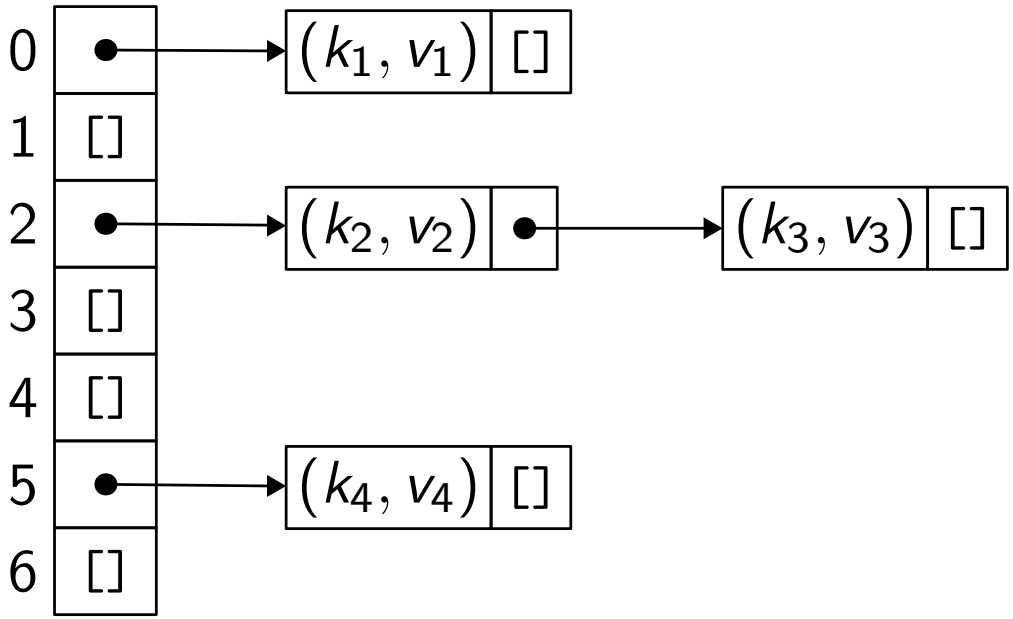
l'idée est très simple : on se donne une fonction

$$hash : clé \to \texttt{int}$$

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 $hash(k)$

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



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

on écrit deux fonctions `count` et `memo_count` **mutuellement récursives**

- `count` effectue le calcul, en appelant `memo_count` récursivement
- `memo_count` consulte la table et la remplit avec `count` si besoin

on obtient finalement le résultat

```
$ ocamlopt wall.ml -o wall
$ time ./wall
80684432319041 4

real 0m0.388s
```

si vous avez aimé ce problème...

http ://projecteuler.net/

récapitulation

programme = suite de déclarations et d'expressions à évaluer

plusieurs façons de l'exécuter

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

variable OCaml :

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

une variable peut être globale (`let`) ou locale (`let in`)

une variable modifiable s'appelle une **référence**

- introduite avec `ref`
- déréférencée avec `!`
- modifiée avec `:=`

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 `()`

dans l'expression

```
e1 ; e2
```

le point-virgule est un **opérateur binaire**

- l'expression `e1`, de type `unit`, est évaluée pour ses effets
- puis l'expression `e2` est évaluée et donne sa valeur à `e1;e2`

code C

```
{ int x = 1;
  x = x + 1;
  int y = x * x;
  printf("%d", y); }
```

code OCaml

```
let x = ref 1 in
x := !x + 1;
let y = !x * !x in
printf "%d" y
```

- égalité physique ==
  - égalité de la valeur (un entier ou une adresse)
  - comme en C/C++/Java

- égalité structurelle =
  - comparaison récursive, en profondeur
  - permet de comparer des *n*-uplets, des listes, des tableaux, etc.

( en Python, c'est `is` et == respectivement)

une fonction est une **valeur** comme une autre

en particulier, une fonction peut être

- locale

```
let find k =
  let rec lookup = ... in
  ...
```

- argument d'une autre fonction

```
let rec sum f l = ...
```

- anonyme

```
fun r h -> ...
```

une fonction qui reçoit plusieurs arguments

```
let f x y = x*x + y*y
```

est la même chose qu'une fonction qui reçoit un argument et renvoie une fonction

```
let f x = fun y -> x*x + y*y
```

et son type est

```
int -> int -> int
```

on appelle cela la **curryfication**

note : si on écrit `f 3 4`, le compilateur évite de construire la fonction intermédiaire

mais on peut appliquer `f` **partiellement**

```
let g = f (1+2)
```

et obtenir une fonction

- l'expression `1+2` est évaluée
- sa valeur est donnée à la variable `x`
- le résultat est la fonction `fun y -> x*x + y*y` en retenant la valeur de `x`, à savoir 3, dans ce qu'on appelle une **fermeture**

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 8192 in
  fun x ->
    try  Hashtbl.find h x
    with Not_found -> let y = f x in Hashtbl.add h x y; y
```

(`Hashtbl` : des tables de hachage utilisant l'égalité polymorphe et une fonction de hachage polymorphe fournie par OCaml)

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

utilisation

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

ne convient cependant pas à une fonction récursive ; le code

```
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 8192 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 count = memo (fun count (r, h) -> ...)
```

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

intérêts :

- allocation efficace
- récupération automatique

on a vu les tableaux

- allocation

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

- type `int array`
- accès

```
a.(1)
```

- affectation

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

n-uplets

on a vu les *n*-uplets

- syntaxe usuelle

```
(1, true, "hello")
```

- type produit `int * bool * string`
- immuable
- déconstruit avec

```
let (n, b, s) = ...
```

une fonction qui reçoit un *n*-uplet

```
let f (x, y) = x*x + y*y
```

est une fonction à **un seul argument**

son type est

```
int * int -> int
```

ce n'est pas la même chose que

```
int -> int -> int
```

utile pour renvoyer plusieurs valeurs

```
let rec division n m =
  if n < m then (0, n)
  else let (q,r) = division (n - m) m in (q + 1, r)
```

de type

```
int -> int -> int * int
```

on déclare le type enregistrement

```
type complex = { 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 person = { name: string; mutable age: int }
```

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

modification en place

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

référence = enregistrement du type suivant

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

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

```
let ref  v   = { contents = v }
let (!)  r   = r.contents
let (:=) r v = r.contents <- v
```

une exception est **levée** avec `raise`

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

et **rattrapée** avec `try with`

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

une exception se propage tant qu'elle n'est pas rattrapée,
et l'exécution se termine si elle n'est jamais rattrapée

les exceptions sont utilisées dans la bibliothèque OCaml pour signifier un résultat exceptionnel

exemple : `Not_found` pour signaler une valeur absente

```
try
  let v = Hashtbl.find table key in
  ...
with Not_found ->
  ...
```

ou de manière équivalente

```
match Hashtbl.find table key with
| v -> ...
| exception 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
```

on peut déclarer de nouvelles exceptions

```
exception Error
```

elles peuvent transporter des valeurs

```
exception Error of string
```

2 – types construits et filtrage

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

```
let lst = 1 :: 2 :: 3 :: []
```

ou encore

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

listes OCaml = des listes simplement chaînées

- `[]` est représenté (en interne) par un entier
- `x::l` est un pointeur vers un bloc

la déclaration `let lst = [1; 2; 3]` correspond à

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

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

notation plus compacte pour une fonction filtrant son argument

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

listes = cas particulier de **types construits**

type construit = réunion de plusieurs **constructeurs**

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

le type option

un autre type prédéfini

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

représente une valeur optionnelle

```
find_opt: ('a -> bool) -> 'a array -> 'a option
```

on filtre une telle valeur

```
match find_opt prime a with
| None   -> printf "pas de nombre premier"
| Some p -> printf "contient %d" p
```

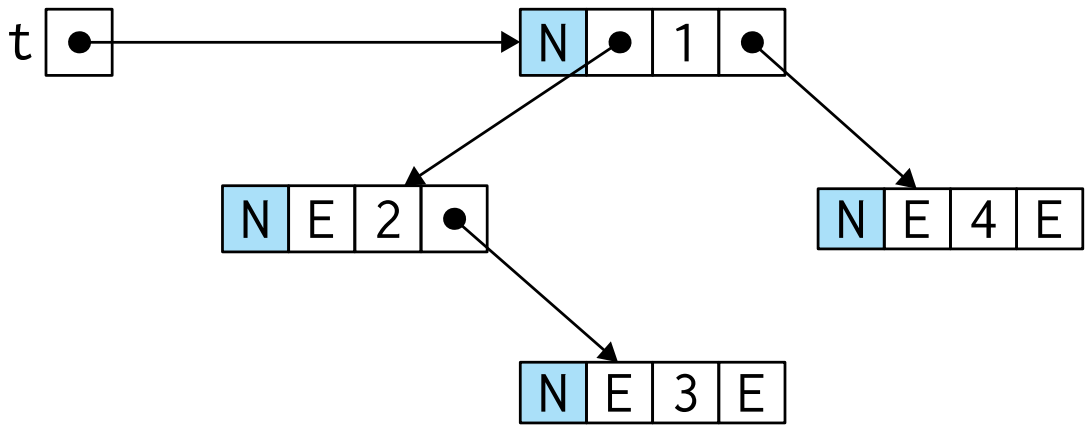
on peut introduire **nos propres** types construits,
avec leurs constructeurs, constants ou non

- un constructeur constant est (en interne) un entier
- un constructeur non constant est un bloc mémoire sur le tas

le filtrage s'applique à tout type construit

```
type 'a bintree =
  | E
  | N of 'a bintree * 'a * 'a bintree
```

```
let t =
  N (N (E,
        2,
        N (E, 3, E)),
     1,
     N (E, 4, E))
```



les parenthèses sont obligatoires (N n'est pas une fonction)

```
let rec size = function
  | E            -> 0
  | N (l, _, r) -> 1 + size l + size r
```

exemple : des formules

```
type formula = True | False | And of formula * formula
```

```
let rec eval = function
  | True -> true
  | False -> false
  | And (f1, f2) -> eval f1 && eval f2
```

les motifs peuvent être **imbriqués**

```
let rec eval = function
  | True -> true
  | False -> false
  | And (False, f2) -> false
  | And (f1, False) -> false
  | And (f1, f2) -> eval f1 && eval f2
```

les motifs peuvent être **regroupés**

```
let rec eval = function
  | True -> true
  | False -> false
  | And (False, _) | And (_, False) -> false
  | And (f1, f2) -> eval f1 && eval 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` ou encore `let () = e`)

les types construits permettent notamment de définir de la **syntaxe abstraite**, c'est-à-dire des arbres représentant des formules, des expressions, des programmes, etc.

le filtrage est alors un outil puissant pour les **manipuler**,
dans un compilateur, un démonstrateur, un logiciel de calcul formel, etc.

3 – 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

```
$ ocamlopt -c arith.ml
```

utilisation dans un autre module `main.ml` :

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

```
$ ocamlopt -c main.ml
```

```
$ ocamlopt arith.cmx main.cmx -o main
```

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

dans un fichier `arith.mli`

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

```
$ ocamlopt -c arith.mli
$ ocamlopt -c arith.ml
```

```
$ ocamlopt -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 `set.ml`

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

mais dans `set.mli`

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

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

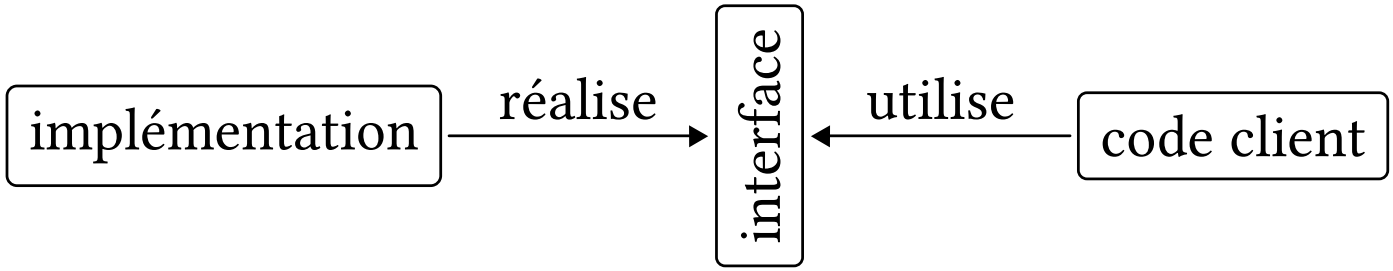
la bibliothèque d'OCaml fournit un module `Hashtbl`

- sans révéler la définition du type `Hashtbl.t`
- sans exporter les fonctions auxiliaires

la documentation nous dit qu'il s'agit de tables de hachages, on peut même lire le code par curiosité, mais le compilateur ne nous laisse pas accéder à ce qui est caché

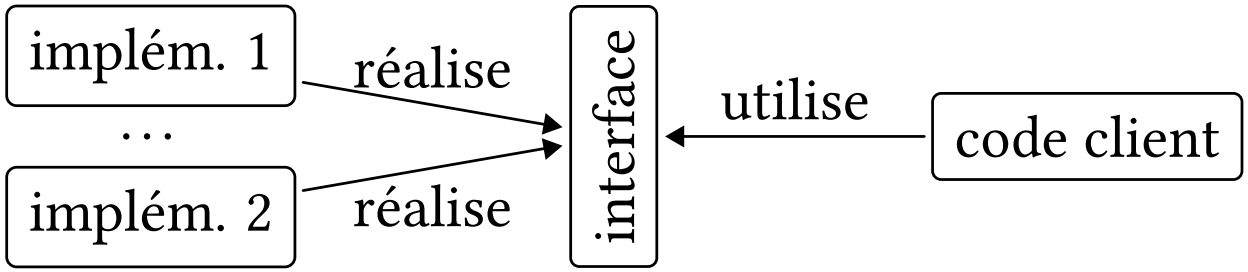
c'est un bon principe que de **cacher** tout ce que l'on peut cacher

le langage OCaml assure une vraie **barrière d'abstraction**, qu'il n'est pas possible de contourner



c'est un bon principe que de **cacher** tout ce que l'on peut cacher

le langage OCaml assure une vraie **barrière d'abstraction**, qu'il n'est pas possible de contourner



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
```

on peut rendre visible tous les éléments d'un module avec `open`

```
open Printf
...
... printf "%d" ...
...
... printf "%s" ...
...
```

nous évite d'écrire systématiquement `Printf.printf`

- modularité par découpage du code en unités appelées **modules**
- encapsulation de types et de valeurs, **types abstraits**
- 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

la solution : un foncteur

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

avec

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

foncteur : le code

```
module HashTable(K: KEY) = struct
  type 'a t = (K.key * 'a) list array
  let create n = Array.make n []
  let add t k v =
    ... K.hash k ...
  let find t k =
    ... if K.eq k' k then ...
end
```

foncteur : l'interface

```
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 x = x
  let eq x y = x=y
end
```

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

```
let table = Hint.create 16
let () = Hint.add table 1729 "Ramanujan-Hardy"
...
```

(ici `table` a le type `string Hint.t`)

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
```

structure de données paramétrée par une autre structure de donnée

- tables de hachage

```
module H = Hashtbl.Make(struct
  type t = ...
  let hash x = ...
  let equal x y = ...
end)
```

- `Set.Make` : ensembles finis immuables (arbres équilibrés)
- `Map.Make` : tables associatives immuables (arbres équilibrés)

```
module S = Set.Make(struct
  type t = ...
  let compare x y = ...
end)
```

algorithmes paramétrés par des structures de données

exemple : algorithme de recherche de plus court chemin

```
module DijkstraShortestPath
  (G: sig
        type graph
        type vertex
        val adj: graph -> vertex -> (vertex * float) list
      end) :
  sig
    val shortest_path:
      G.graph -> G.vertex -> G.vertex ->
      G.vertex list * float
  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 besoin d'au moins deux instances

4 – 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

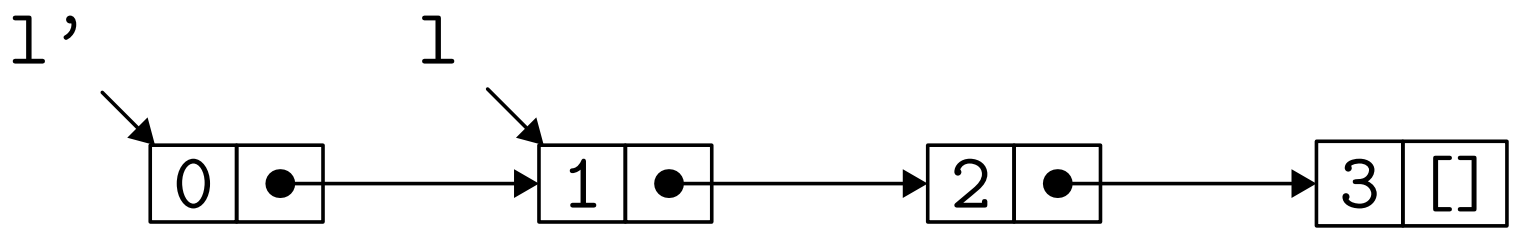
on parle alors de **structure persistante**

un style de programmation n'utilisant que des structures immuables est dit **purement applicatif** ou encore simplement **pur** (parfois aussi **purement fonctionnel**)

exemple de structure immuable : les listes

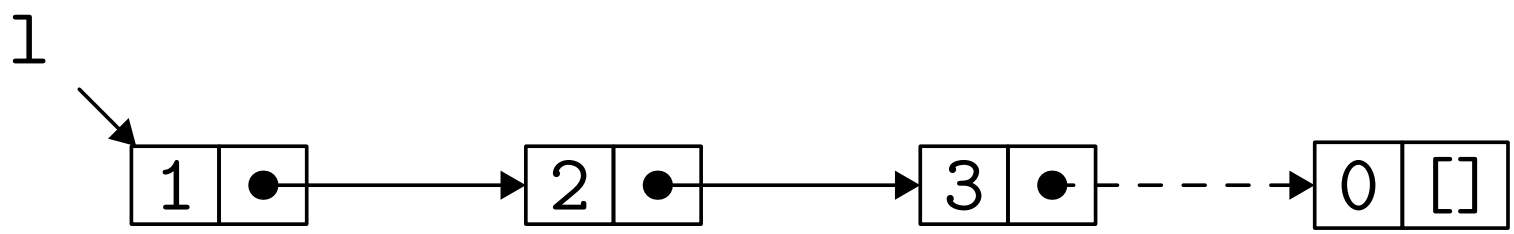
```
let l = [1; 2; 3]
```

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



pas de copie, mais **partage**

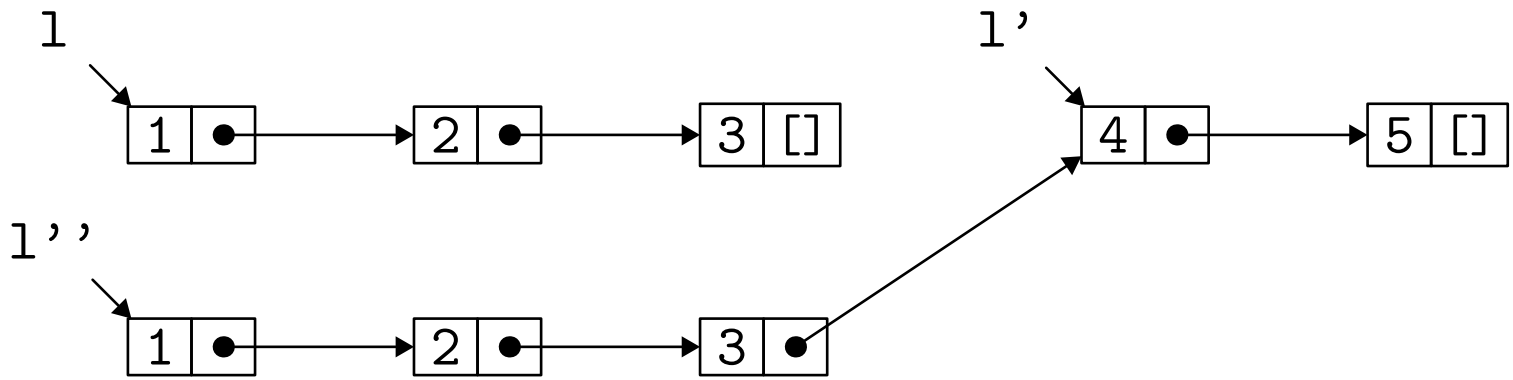
un ajout en queue de liste n'est pas aussi simple :



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 mutables, par exemple ainsi

```
type 'a mlist =
  | Nil
  | Cons of { value: 'a; mutable next: 'a mlist }
```

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

autre exemple : des arbres binaires

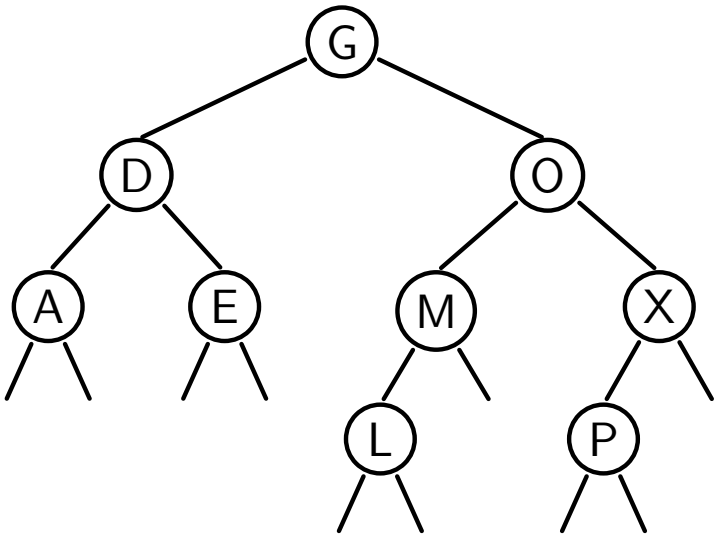
```
type 'a bintree = E | N of 'a bintree * 'a * 'a bintree
```

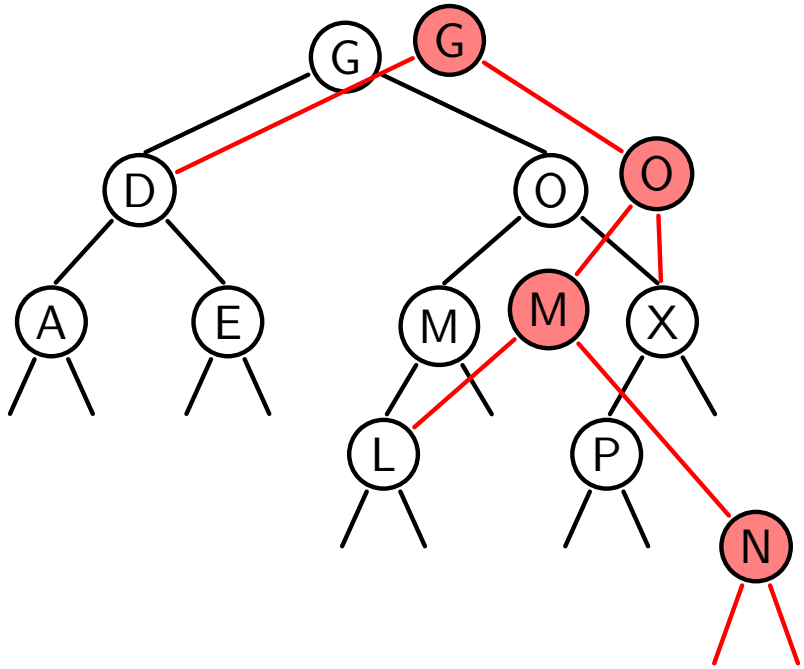
```
let rec add v = function
  | E              -> N (E, v, E)
  | N (l, x, r) -> if v < x then N (add v l, x, r)
                            else N (l, x, add v r)
```

```
val add: elt -> elt bintree -> elt bintree
```

```
let t1 = N (..., "G", ...)
```

```
let t2 = add "N" t1
```





une insertion copie des nœuds et **partage** des sous-arbres

la bibliothèque `Set` fournit des ensembles **immuables**,
réalisés par des arbres binaires de recherche équilibrés (des AVL)

exemple : des ensembles d'entiers

```
module S = Set.Make(Int)
```

les opérations `S.add`, `S.mem`, `S.remove`, etc., ont un coût $O(\log N)$ où N est le cardinal de l'ensemble, **en temps et en espace**

un tableau contenant **tous** les ensembles

$$\{\}, \{1\}, \{1, 2, \}, \ldots, \{1, 2, \ldots, n-1\}$$

```
module S = Set.Make(Int)
let n = 1_000_000
let a = Array.make n S.empty
let () = for i = 1 to n - 1 do a.(i) <- S.add i a.(i-1) done
```

fait remarquable,
sa construction a demandé **un temps et un espace** $N \log N$

intérêts pratiques de la persistance

1. **correction** des programmes
   - code plus simple
   - raisonnement mathématique possible

2. **gain** de place
   - rendu possible par le partage

3. outil puissant pour le **rebroussement**
   - algorithmes de recherche
   - rétablissement suite à une erreur

persistance et rebroussement (1)

recherche de la sortie dans un labyrinthe

```
type state
val  success: state -> bool
type move
val  moves: state -> move list
val  move: state -> move -> state
```

```
let rec find_path s =
  success s || take_first s (moves s)
and take_first s = function
  | []     -> false
  | m :: r -> find_path (move m s) || take_first s r
```

avec un état global modifié en place :

```
let rec find_path () =
  success () || take_first (moves ())
and take_first = function
  | []     -> false
  | m :: r -> (move m; find_path ())
              || (undo m; take_first r)
```

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

persistance et rebroussement (2)

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 rebroussement (2)

avec une structure persistante

```
let db = ref (... base initiale ...)
...
try
  db := (... opération de mise à jour de !db ...)
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
...
```

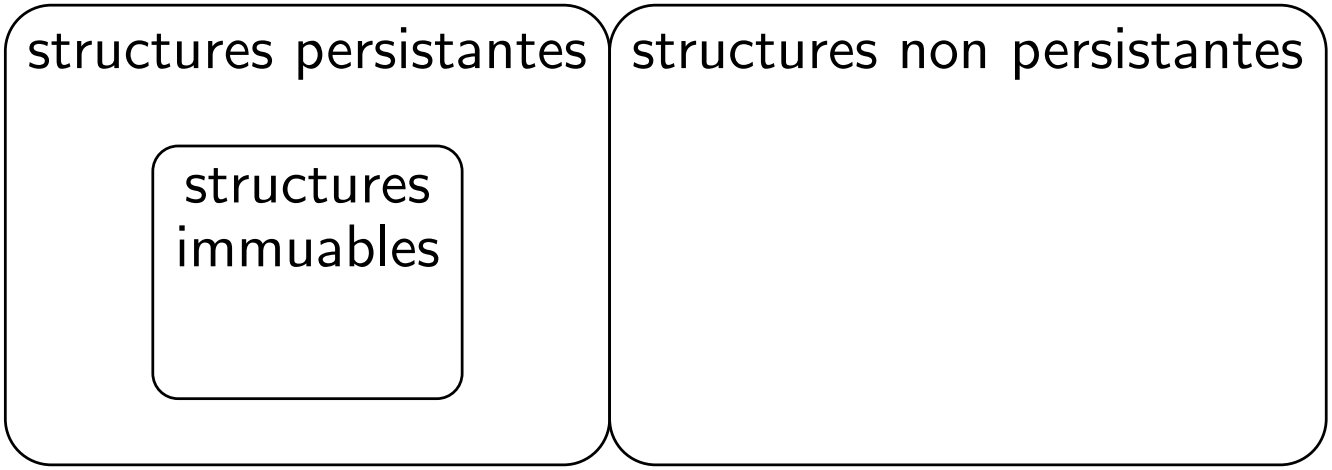
en réalité, persistant ne signifie pas sans effet de bord

$$persistant = observationnellement\ immuable$$

on a seulement l'implication dans un sens :

$$immuable \Rightarrow persistant$$

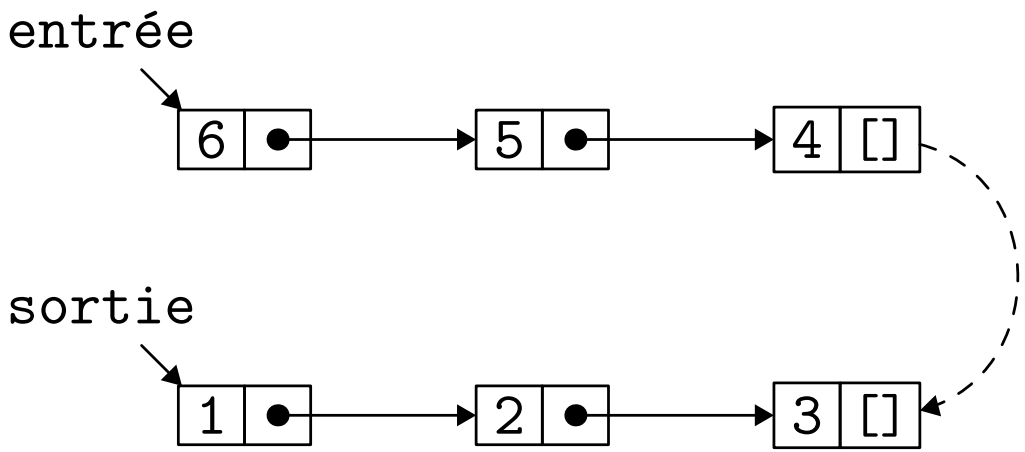
la réciproque est fausse



exemple : files persistantes

```
type '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 est vide, on calculera plusieurs fois le même `List.rev e`

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

```
type 'a t = {
  mutable  rear: 'a list; (* on ajoute ici *)
  mutable front: 'a list; (* on retire ici *)
}
```

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 () = { rear = []; front = [] }

let push x q = { rear = x :: q.rear; front = q.front }

exception Empty
```

```
let pop q = match q.front with
  | x :: s -> x, { rear = q.rear; front = s }
  | [] -> match List.rev q.rear with
          | [] -> raise Empty
          | x :: s as r ->
              q.rear <- [];
              q.front <- r;
              x, { rear = []; front = s }
```

- 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

5 – écosystème

gestionnaire de **paquets** pour OCaml

```
$ opam install graphics
$ opam search union
...
unionFind -- Implementations of the union-find data structure
$ opam show unionFind
...
$ opam install unionFind
```

permet aussi d'installer plusieurs versions d'OCaml sur sa machine
(`opam switch create 5.2.0`)

système de **compilation** pour OCaml

- dans un fichier `dune-project`

```
(lang dune 3.4)
```

- dans un fichier `dune`

```
(executable
 (name maze)
 (libraries graphics unionFind))
```

puis

```
$ dune build
```

pour construire l'exécutable

et

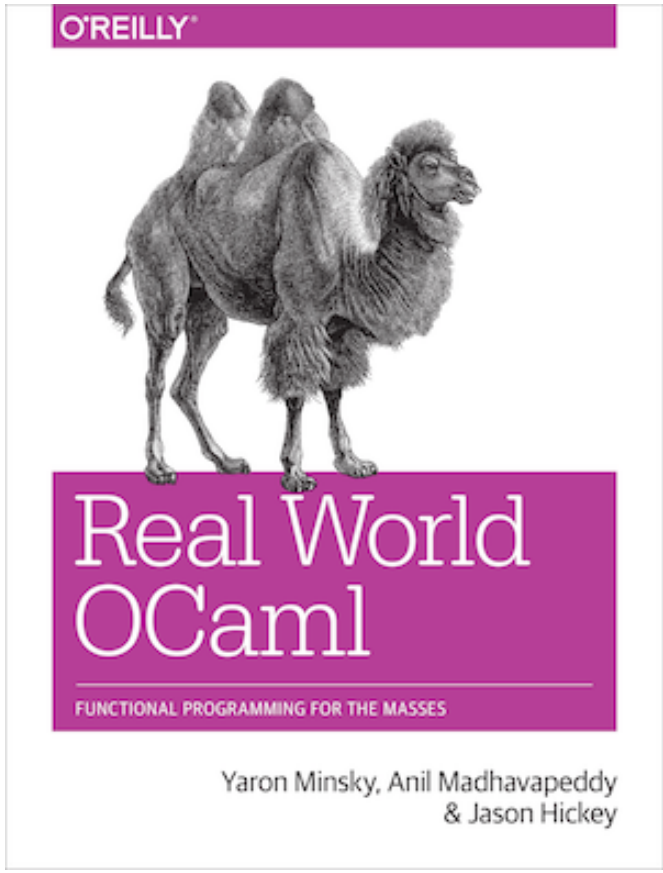
```
$ dune exec ./maze.exe 20
```

pour exécuter
(tous les fichiers produits, dont l'exécutable, sont dans `_build`)

beaucoup de ressources sur `http://ocaml.org/`

deux ouvrages

(à la bibliothèque de l'ENS)



(disponible en ligne)