École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 1 / 27 septembre 2012

présentation du cours

- **cours** le jeudi, 14h–17h en salle R
  - polycopié = ensemble des transparents
- **TD** le mercredi, 8h45–10h45 en salle Info 4 NIR
  - avec Louis Mandel (`Louis.Mandel@lri.fr`)
  - à partir du 3 octobre

toutes les infos sur le site web du cours

`http://www.lri.fr/~filliatr/ens/compil/`

questions ⇒ `Jean-Christophe.Filliatre@lri.fr`

évaluation

- un **examen**, en janvier
  - anciens sujets+corrigés sur le site
- un **projet** = un mini compilateur
  - réalisé en dehors des TD, en binômes
  - rendu en deux fois (30 novembre, 11 janvier)

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

objectif du cours

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

programmation

ici on programme

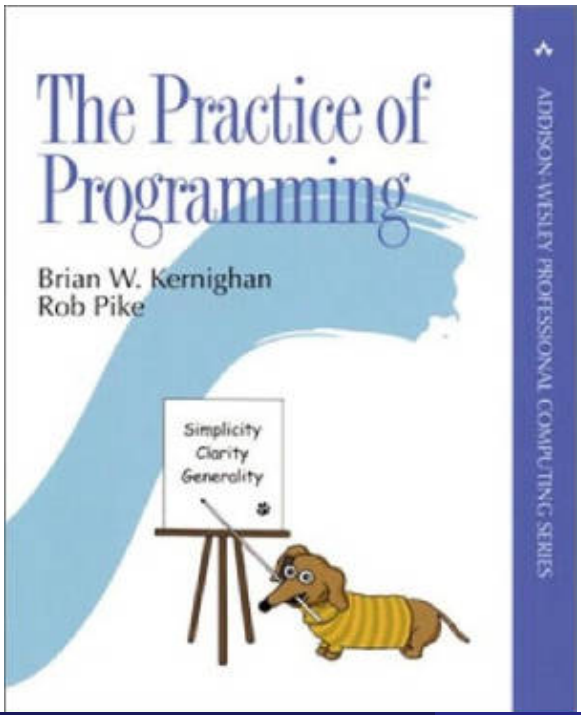
- en cours
- en TD
- pour réaliser le projet
- à l'examen

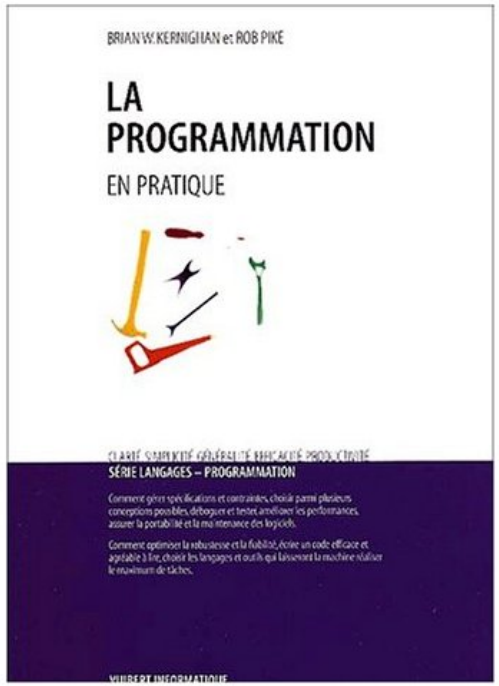
on programme en **OCaml**

message

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

présentation

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

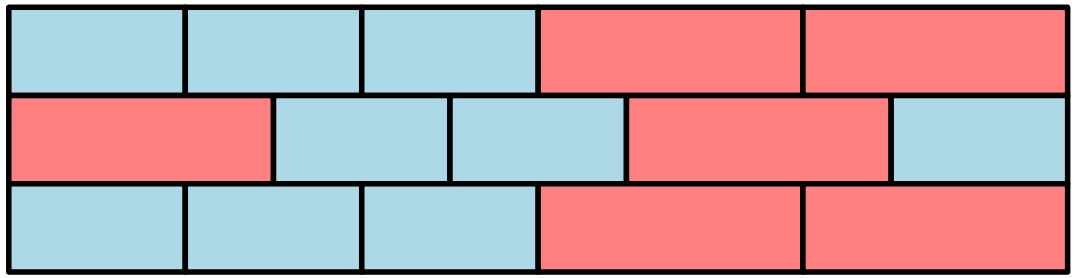
http://caml.inria.fr/

premiers pas en OCaml

un peu de maçonnerie

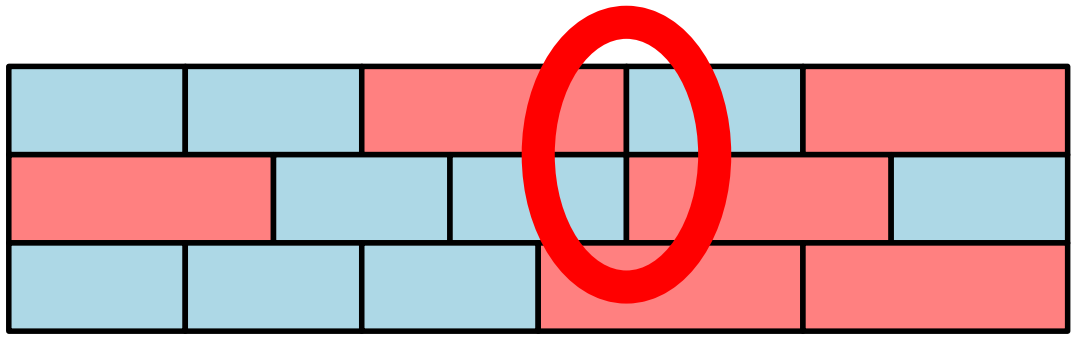
on souhaite construire un mur avec des briques de longueur 2 ( ) et de longueur 3 ( ), dont on dispose en quantités respectives infinies

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



un peu de maçonnerie sérieuse

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



question

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

première idée

on va calculer **récursivement** 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

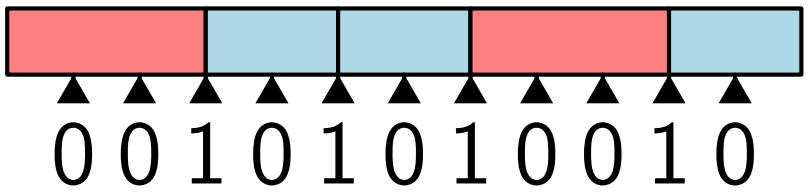
$$W(r, 1) = 1$$

- sinon

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

deuxième idée

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

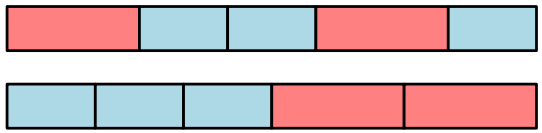


ainsi cette rangée est représentée par l'entier 338 ($= 00101010010_2$)

quel intérêt ?

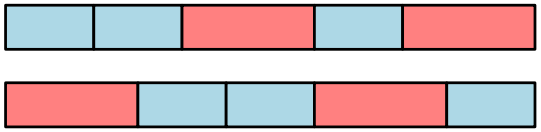
il est alors aisé de vérifier que deux rangées sont compatibles, par une simple opération de ET logique (`land` en OCaml)

ainsi



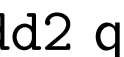
$$\begin{array}{rl} & 00101010010_2 \\ \texttt{land} & 01010100100_2 \\ = & 00000000000_2 = 0 \end{array}$$

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

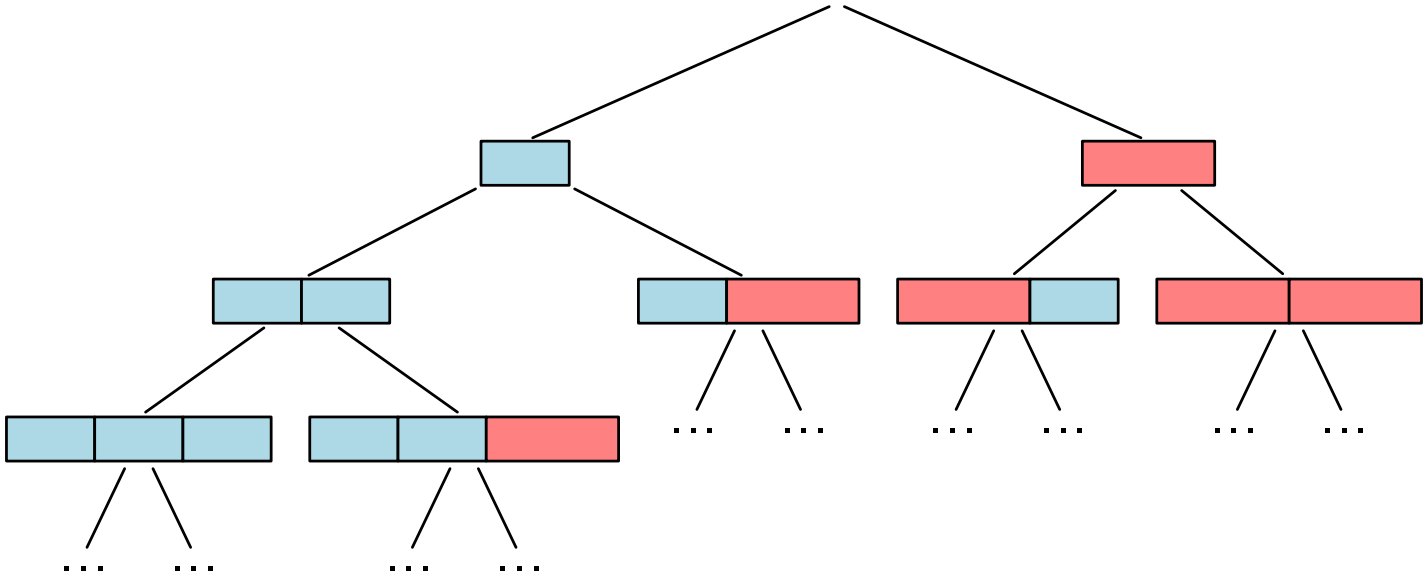
on va construire la **liste** de toutes les rangées de briques possibles de longueur 32

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 `fill` qui parcourt cet arbre



jusqu'à trouver des rangées de la bonne longueur

somme sur une liste

pour écrire la fonction récursive W, 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
```

décompte récursif

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

déception

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

une troisième idée

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 $\rightarrow$ 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**

table de hachage

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

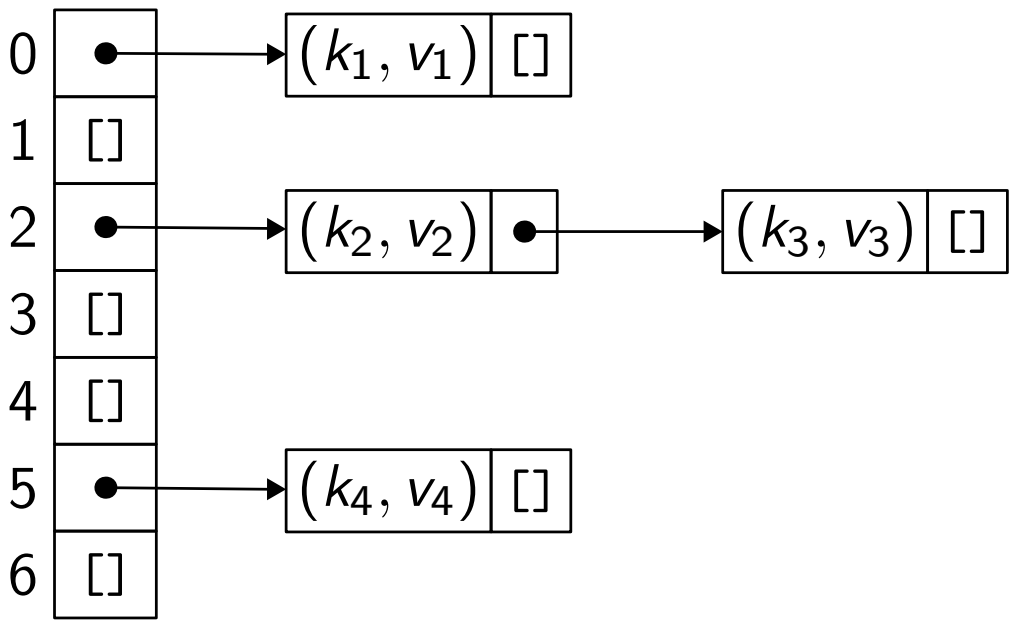
$$hash : clé \rightarrow \texttt{int}$$

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 $hash(k) \mod n$

collisions

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



mémoïsation

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

une dernière petite chose

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`

c'est gagné

on obtient finalement le résultat

```
% ocamlopt wall.ml -o wall
% time ./wall
80684323190414

real 0m6.759s
```

si vous avez aimé ce problème...

http ://projecteuler.net/

récapitulation

déclarations

programme = suite de déclarations et d'expressions à évaluer

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

variable

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`

expressions et instructions

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

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

polymorphisme

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

exemple :

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

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

exemple

dans la bibliothèque standard, on trouve

```
val List.fold_left : (α → β → α) → α → β list → α
  (** 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
```

autre exemple : mémoïsation

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
```

autre exemple : mémoïsation

```
val memo : (α → β) → α → β
```

utilisation

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

autre exemple : mémoïsation

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`

solution

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
```

autre exemple

```
val memo: ((α → β) → α → β) → α → β
```

c'est un opérateur de point fixe

utilisation

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

allocation mémoire

GC

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

Intérêts :

- récupération automatique
- allocation efficace

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

tableaux

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
```

retour sur les références

référence = enregistrement du type suivant

```
type α ref = { mutable contents : α }
```

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

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

listes

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

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>
```

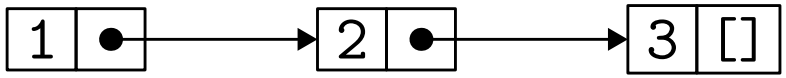
notation plus compacte pour une fonction filtrant son argument

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

représentation mémoire

listes OCaml = 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 α list = [] | :: of α × α list
```

filtrage

le filtrage se généralise

```
# let rec evalue = function
    | Vrai → true
    | Faux → false
    | Conjonction (f1, f2) → evalue f1 && evalue f2;;
```

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

filtrage

les motifs peuvent être **imbriqués** :

```
let rec evalue = function
  | Vrai → true
  | Faux → false
  | Conjonction (Faux, f2) → false
  | Conjonction (f1, Faux) → false
  | Conjonction (f1, f2) → evalue f1 && evalue f2;;
```

filtrage

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;;
```

filtrage

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)

récapitulation

- 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

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 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

génie logiciel

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**

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
```

encapsulation

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**

compilation séparée

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

$\Rightarrow$ **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
```

récapitulation

- 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**

foncteurs

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

première solution

passer les fonctions en argument :

```
type (α, β) t

val create : int → (α, β) t

val add : (α → int) → (α, β) t → α → β → unit

val find : (α → int) → (α → α → bool) →
  (α, β) t → α → β
```

problème : il faut être cohérent dans l'utilisation

deuxième solution

passer les fonctions à la création :

```
type (α, β) t

val create: (α → int) → (α → α → bool) →
  int → (α, β) t

val add: (α, β) t → α → β → unit

val find: (α, β) t → α → β
```

deuxième solution

```
type (α, β) t = {
  hash    : α → int;
  eq      : α → α → bool;
  buckets: (α × β) 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))
```

deuxième solution

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

la bonne 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 α t = (K.key × α) 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
```

foncteur : l'interface

```
module HashTable(K: KEY) : sig
  type α t
  val create : int → α t
  val add : α t → K.key → α → unit
  val find : α t → K.key → α
end
```

foncteur : utilisation

```
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)

foncteur : utilisation

```
# 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>
```

foncteur : utilisation

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

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

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

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

foncteurs : applications

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

algorithme générique

```
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
```

exercice

écrire un foncteur pour calculer modulo `m`

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

du bon usage des foncteurs

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

structures de données immuables

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

dit autrement :

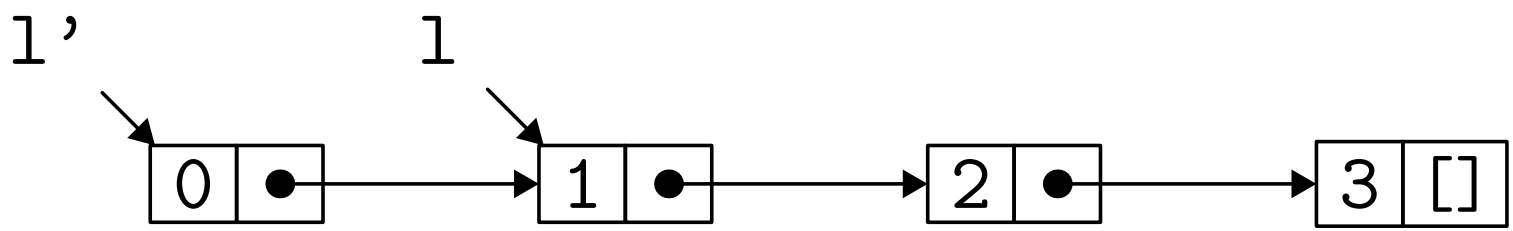
- une valeur n'est pas affectée par une opération,
- mais une **nouvelle** valeur est renvoyée

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]
```

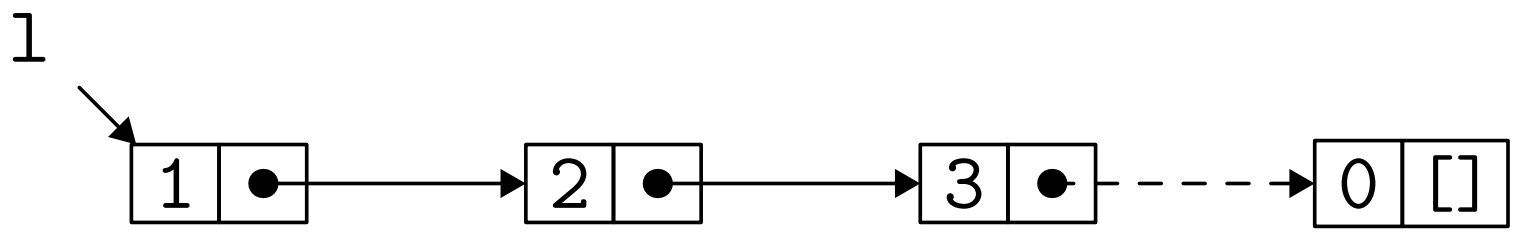
```
let l' = 0 :: l
```



pas de copie, mais **partage**

contrepartie

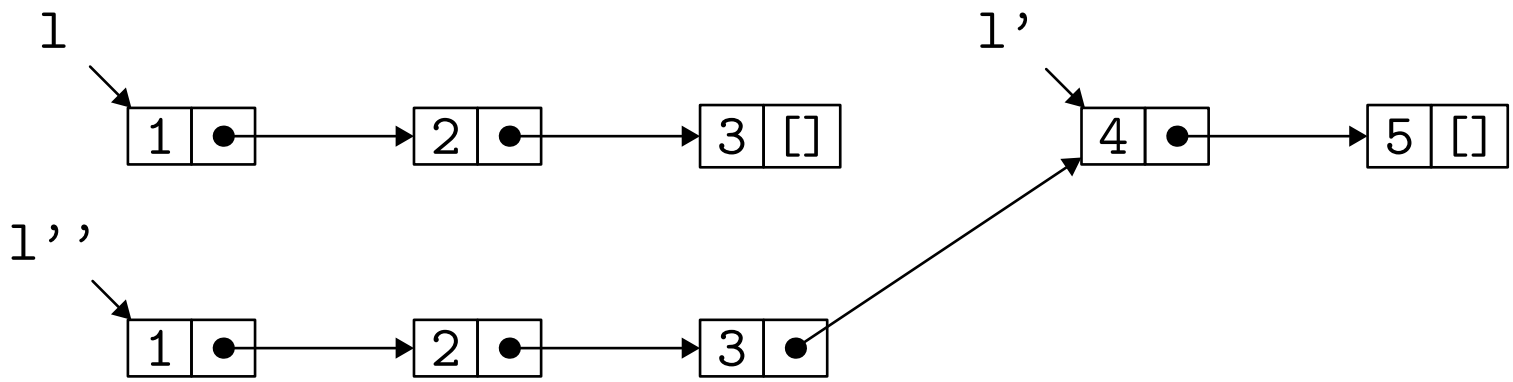
un ajout en queue de liste n'est pas aussi simple :



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**

listes chaînées modifiables en place

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

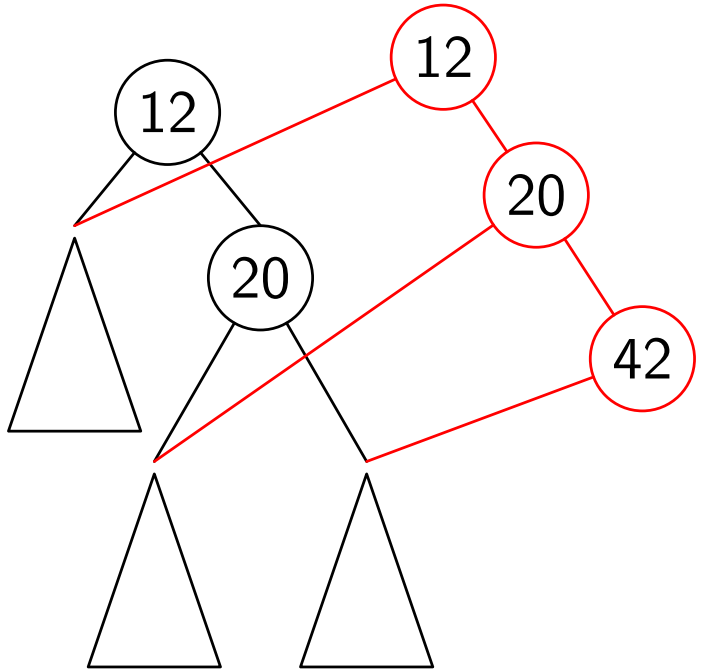
```
type α liste = Vide | Element of α element
and α element = { valeur : α; mutable suivant : α liste }
```

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**

intérêts pratiques de la persistance

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

persistance 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
```

sans persistance

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*)

persistance 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
```

persistance 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

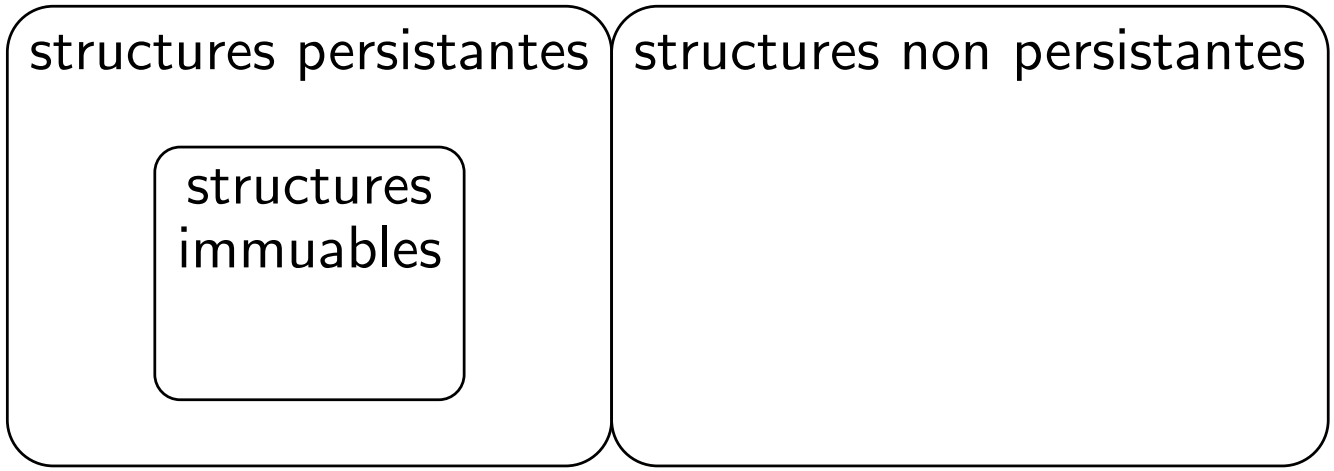
persistant ne signifie pas sans effet de bord

$$persistant = observationnellement\ immuable$$

on a seulement l'implication dans un sens :

$$immuable \Rightarrow persistant$$

la réciproque est fausse

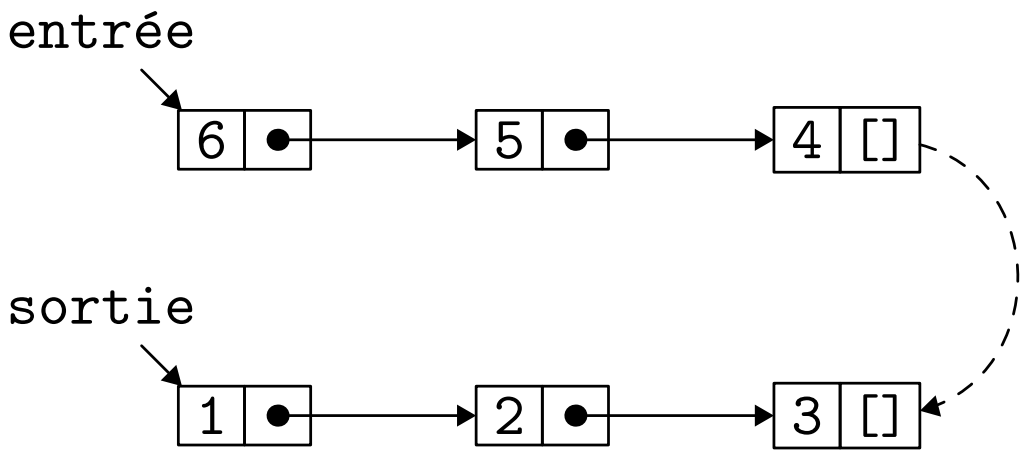


exemple : files persistantes

```
type α t
val create : unit → α t
val push : α → α t → α t
exception Empty
val pop : α t → α × α 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 α t = α list × α 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
```

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 α t = (α list × α 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
```

récapitulation

- 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

la semaine prochaine

- TD de mercredi
  - exercices de mise à niveau OCaml
  - 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));}
```