

École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 1 / 25 septembre 2008

- **cours** le jeudi, 16h45–19h00 en salle U/V
 - pas de polycopié, mais transparents disponibles
- **TD** le lundi, 8h45–10h45 en salle INFO NIR
 - avec Gaëtan LEURENT
- **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

exception à la règle :
il y a un polycopié 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

le premier programme

```
hello.ml
```

```
print_string "hello world!\n"
```

Compilation

```
% ocamlc -o hello hello.ml
```

```
% ocamlopt -o hello hello.ml
```

Exécution

```
% ./hello
```

```
hello world!
```

programme = suite de déclarations et d'expressions à évaluer

```
let x = 1 + 2;;  
print_int x;;  
let y = x * x;;  
print_int y;;
```

`let x = e` introduit une variable globale

Différences avec la notion usuelle de variable :

- 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`

```
let x = ref 1 ;;  
print_int !x ;;  
x := !x + 1 ;;  
print_int !x ;;
```

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

constructions usuelles :

- conditionnelle

```
if i = 1 then 2 else 3
```

- boucle for

```
for i = 1 to 10 do x := !x + i done
```

- séquence

```
x := 1 ; 2 * !x
```

les expressions sans réelle valeur (affectation, boucle, ...) ont pour type **unit**

ce type a une unique valeur, notée **()**

c'est le type donné à la branche **else** lorsqu'elle est absente

correct :

```
if !x > 0 then x := 0
```

incorrect :

```
2 + (if !x > 0 then 1)
```

En C ou Java, la portée d'une variable locale est définie par le **bloc** :

```
{  
  int x = 1 ;  
  ...  
}
```

En Caml, variable locale introduite par **let in** :

```
let x = 10 in x * x
```

Comme une variable globale :

- nécessairement initialisée
- type inféré
- immuable
- mais **portée limitée à l'expression qui suit le in**

let in = expression

`let x = e1 in e2` est une expression
son type et sa valeur sont ceux de `e2`,
dans un environnement où `x` a le type et la valeur de `e1`

Exemple

```
let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
```


Java

```
{ int x = 1;  
  x = x + 1;  
  int y = x * x;  
  System.out.print(y); }
```

Caml

```
let x = ref 1 in  
x := !x + 1;  
let y = !x * !x in  
print_int y
```

- programme = suite d'expressions et de déclarations
- variables introduites par le mot clé `let` non modifiables
- pas de distinction expression / instruction

la boucle d'interaction

version **interactive** du compilateur

```
% ocaml
  Objective Caml version 3.09.1

# let x = 1 in x + 2 ;;
- : int = 3

# let y = 1 + 2 ;;
val y : int = 3

# y * y ;;
- : int = 9
```

Fonctions

```
# let f x = x * x ;;  
val f : int -> int = <fun>
```

- corps = expression (pas de **return**)
- type inféré (types de l'argument x et du résultat)

```
# f 4 ;;  
- : int = 16
```

une procédure = une fonction dont le résultat est de type `unit`

Exemple

```
# let x = ref 1 ;;
# let set v = x := v ;;

val set : int -> unit = <fun>
# set 3 ;;
- : unit = ()
# !x ;;
- : int = 3
```

fonction “sans argument”

prend un argument de type `unit`

Exemple

```
# let reset () = x := 0 ;;  
val reset : unit -> unit = <fun>  
# reset () ;;
```

fonction à plusieurs arguments

```
# let f x y z = if x > 0 then y + x else z - x;;  
val f : int -> int -> int -> int = <fun>  
# f 1 2 3;;  
- : int = 3
```


fonction locale à une expression

```
# let carre x = x * x in carre 3 + carre 4 = carre 5 ;;  
- : bool = true
```

fonction locale à une autre fonction

```
# let pythagore x y z =  
    let carre n = n * n in  
    carre x + carre y = carre z ;;  
val pythagore : int -> int -> int -> bool = <fun>
```

fonction comme valeur de première classe

fonction = expression comme une autre, introduite par **fun**

```
# fun x -> x+1
```

```
- : int -> int = <fun>
```

```
# (fun x -> x+1) 3 ;;
```

```
- : int = 4
```

En réalité

```
let f x = x+1 ;;
```

est la même chose que

```
let f = fun x -> x+1 ;;
```

application partielle

```
fun x y -> x*x + y*y
```

est la même chose que

```
fun x -> fun y -> x*x + y*y
```

on peut appliquer une fonction **partiellement**

Exemple

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

```
val f : int -> int -> int = <fun>
```

```
# let g = f 3;;
```

```
val g : int -> int = <fun>
```

```
# g 4;;
```

```
- : int = 25
```

l'application partielle est une manière de **retourner** une fonction
mais on peut aussi retourner une fonction à l'issue d'un calcul

```
# let f x = let x2 = x * x in fun y -> x2 + y * y;;
```

```
val f : int -> int -> int = <fun>
```

une application partielle de `f` ne calcule `x*x` qu'**une seule fois**

application partielle : exemple

```
# let compteur_depuis n =  
    let r = ref (n-1) in fun () -> incr r; !r;;  
val compteur_depuis : int -> unit -> int = <fun>  
# let c = compteur_depuis 0;;  
val c : unit -> int = <fun>  
# c ();;  
- : int = 0  
# c ();;  
- : int = 1
```

une fonction peut prendre des fonctions en arguments

```
# let intègre f =  
  let n = 100 in  
  let s = ref 0.0 in  
  for i = 0 to n-1 do  
    let x = float i /. float n in s := !s +. f x  
  done ;  
  !s /. float n
```

```
# intègre sin;;
```

```
- : float = 0.455486508387318301
```

```
# intègre (fun x -> x*.x) ;;
```

```
- : float = 0.32835
```

En Java on itère ainsi sur les éléments d'une structure de données

```
for (Iterator it = v.elements() ; it.hasNext() ;) {  
    ... on traite it.next() ...  
}
```

En Caml, itérateur = fonction d'ordre supérieur

Exemple : table associant des chaînes de caractères

```
val iter : (string -> string -> unit) -> table -> unit  
let n = ref 0 in iter (fun x y -> incr n) t ; !n  
iter (fun x y -> Printf.printf "%s -> %s\n" x y) t
```

“en C aussi on peut passer et retourner des fonctions par l'intermédiaire de pointeurs de fonctions”

mais les fonctions d'ocaml sont plus que des pointeurs de fonctions

```
let f x = let x2 = x * x in fun y -> x2 + y * y ;;
```


En Caml, le recours aux fonctions récursives est naturel, car

- un appel de fonction ne coûte pas cher
- la récursivité terminale est correctement compilée

Exemple :

```
let zéro f =  
  let rec cherche i = if f i = 0 then i else cherche (i+1) in  
  cherche 0
```

code récursif \Rightarrow plus lisible, plus simple à justifier

```
# let f x = x;;  
val f : 'a -> 'a = <fun>  
# f 3;;  
- : int = 3  
# f true;;  
- : bool = true  
# f print_int;;  
- : int -> unit = <fun>  
# f print_int 1;;  
1- : unit = ()
```

Cam1 infère toujours le type **le plus général possible**

Exemple :

```
# let compose f g = fun x -> f (g x) ;;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

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

Allocation mémoire

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

Intérêts :

- récupération automatique
- allocation efficace

⇒ perdre le réflexe « allouer coûte cher »

... mais continuer à se soucier de complexité !

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

```
val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

nécessairement initialisé

```
# let a = [| 1; 2; 3; 4 |] ;;
```

```
# a.(1) ;;
```

```
- : int = 2
```

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

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

```
# a;;
```

```
- : int array = [|1; 5; 3; 4|]
```

tri par insertion

```
let tri_insertion a =  
  let swap i j = let t = a.(i) in a.(i) <- a.(j); a.(j) <- t in  
  for i = 1 to Array.length a - 1 do  
    (* insérer l'élément i dans 0..i-1 *)  
    let j = ref (i - 1) in  
    while !j >= 0 && a.(!j) > a.(!j + 1) do  
      swap !j (!j + 1); decr j  
    done  
  done
```


tri par insertion

```
let tri_insertion a =  
  let swap i j = let t = a.(i) in a.(i) <- a.(j); a.(j) <- t in  
  for i = 1 to Array.length a - 1 do  
    (* insérer l'élément i dans 0..i-1 *)  
    let rec insère j =  
      if j >= 0 && a.(j) > a.(j+1) then  
        begin swap j (j+1); insère (j-1) end  
    in  
    insère (i-1)  
  done
```

comme les records de Pascal ou les structures de C
on déclare le type enregistrement

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

allocation et initialisation simultanées :

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

```
val x : complexe = {re = 1. ; im = -1.}
```

```
# x.im;;
```

```
- : float = -1.
```

champs modifiables en place

```
type personne = { nom : string; mutable age : int }  
# let p = { nom = "Martin"; age = 23 };;  
val p : personne = {nom = "Martin"; age = 23}  
# 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

seuls les tableaux et les champs déclarés **mutable** sont modifiables en place

notation usuelle

```
# (1,2,3) ;;
```

```
- : int * int * int = (1, 2, 3)
```

```
# let v = (1, true, "bonjour", 'a') ;;
```

```
val v : int * bool * string * char = (1, true, "bonjour", 'a')
```

accès aux éléments

```
# let (a,b,c,d) = v ;;
```

```
val a : int = 1
```

```
val b : bool = true
```

```
val c : string = "bonjour"
```

```
val d : char = 'a'
```

n-uplets (suite)

Exemple d'utilisation

```
# let rec division n m =  
  if n < m then (0, n)  
  else let (q,r) = division (n - m) m in (q + 1, r) ;;  
val division : int -> int -> int * int = <fun>
```

fonction prenant un *n*-uplet en argument

```
# let f (x,y) = x + y ;;  
val f : int * int -> int = <fun>  
# f (1,2) ;;  
- : int = 3
```

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]
```

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

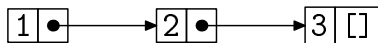
filtrage = construction par cas sur la forme d'une liste

```
# let rec somme l =  
  match l with  
    [] -> 0  
  | x :: r -> x + somme r ;;  
  
val somme : int list -> int = <fun>  
  
# somme [1;2;3] ;;  
  
- : int = 6
```

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



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 evalue = function
  | Vrai -> true
  | Faux -> false
  | Conjonction (f1, f2) -> evalue f1 && evalue f2;;

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

les motifs peuvent être **imbriqués** :

```
let rec evalue = fonction
  | Vrai -> true
  | Faux -> false
  | Conjonction (Faux, f2) -> false
  | Conjonction (f1, Faux) -> false
  | Conjonction (f1, f2) -> evalue f1 && evalue 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 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**

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

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 à la fonction de hachage et la fonction d'égalité

passer les fonctions en argument :

```
type 'a t
```

```
val create : int -> 'a t
```

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

```
val mem : ('a -> int) -> ('a -> 'a -> bool) ->  
          'a t -> 'a -> bool
```

deuxième solution

passer les fonctions à la création :

```
type 'a t
```

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

```
val add : 'a t -> 'a -> unit
```

```
val mem : 'a t -> 'a -> bool
```

```
type 'a t = { hash : 'a -> int;  
             eq : 'a -> 'a -> bool;  
             data : 'a 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
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 = ()

# Hentiers.mem t 13 ;;

- : bool = true

# Hentiers.mem t 14 ;;

- : bool = false
```

- 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 sommet
    val voisins : graph -> sommet -> (sommet * int) list
  end) :
sig
  val plus_court_chemin :
    G.graph -> G.sommet -> G.sommet -> G.sommet list * int
end
```

Persistence

structures de données immuables

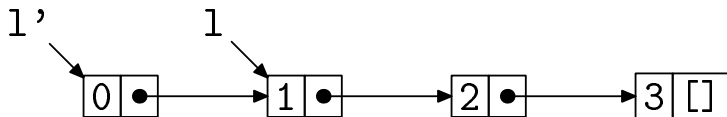
structure de données **immuables** \Rightarrow

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

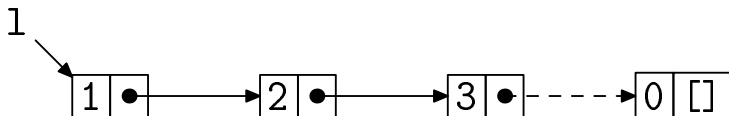
```
let l = [1 ; 2 ; 3]
```

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

pas de copie, mais **partage**



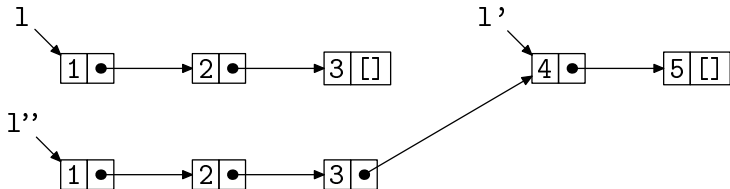
un ajout en queue de liste n'est pas aussi simple :



concaténation de deux listes

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

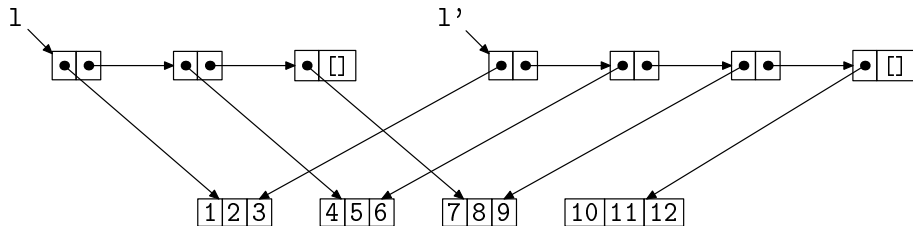
```
let l = [1; 2; 3]  
let l' = [4; 5]  
let l'' = append l l'
```



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

quid des éléments

```
let l = [(1,2,3) ; (4,5,6) ; (7,8,9)] ;;  
append l [(10,11,12)] ;;
```



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*)

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

persistence et backtracking (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 = fonction
  | [] -> 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 = fonction  
  | [] -> false  
  | d :: r -> (déplace d ; cherche ()) || (revient d ; essaye r)
```

persistence et backtracking (2)

programmes très simples, représentés par

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

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 backtracking (2)

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


persistence et backtracking (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 ...
```

persistence et backtracking (3)

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

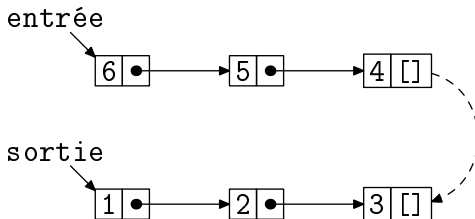
la réciproque est fausse

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
```

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 '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
```

exemple : files persistantes

maintenant avec un effet de bord

```
type 'a t =  
  { mutable entrée : 'a list; mutable sortie : 'a list }  
let create () = { entrée = []; sortie = [] }  
let push x q = { entrée = x :: q.entrée; sortie = q.sortie }  
exception Empty  
  
let pop q = match q.sortie with  
  | x :: s ->  
    x, { entrée = q.entrée; sortie = s }  
  | [] -> match List.rev q.entrée with  
    | [] ->  
      raise Empty  
    | x :: s as r ->  
      q.entrée <- []; q.sortie <- r;  
      x, { entrée = []; sortie = 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

- TD de lundi
 - 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));}
```