Programmation Avancée

OCaml : modèle d'exécution et représentation des valeurs

Armaël Guéneau

(merci à J.C. Filliâtre pour une partie des transparents)

annonce

Précision sur les modalités d'évaluation pour la **seconde session** :

- Notes de projet (40%) et contrôle continu (TPs, 20%) **reportées en seconde session**
- **Oral de rattrapage** qui remplace la note d'exam (40%)

OCaml implémente un modèle d'exécution simple, élégant et efficace

- coûts et consommation mémoire **prédictibles**
- tout en écrivant du code **haut niveau** et élégant
- particulièrement adapté pour l'implémentation de structures de données avancées avec **partage**

Quizz 1

```
let l : int list = ...

let a = [| l; l |]
let b = [| l; 0 :: l |]
```

Définir `a` nécessite-t-il de copier `l` en mémoire ? Et pour `b` ?

Quizz 2

```
let l : int array list = ...
let a : int array = ...

let l' = a :: l
```

Définir `l'` nécessite-t-il de copier `l` ? et `a` ?

Quizz 3

```
type box = { mutable v : int }

let b : box = { v = 0 }

let a = [| b; b |]
a.(0).v <- 1
```

Que vaut `a.(1).v` ?

Quizz 4

```
type box = { mutable v : int }

let f (b1 : box) (b2 : box) : int * int =
  b1.v <- 42;
  (b1.v, b2.v)

let b : box = { v = 0 }
let (x, y) = f b b
```

Que valent `x` et `y` ?

comment comprendre le comportement de ces programmes ?

≪ **Modèle d'exécution** ≫ :

modèle de l'évolution de la machine lorsque le programme s'exécute

Représentation des valeurs :

comment les données du programme sont représentées en mémoire

nécessaire pour raisonner sur les performances / la consommation mémoire

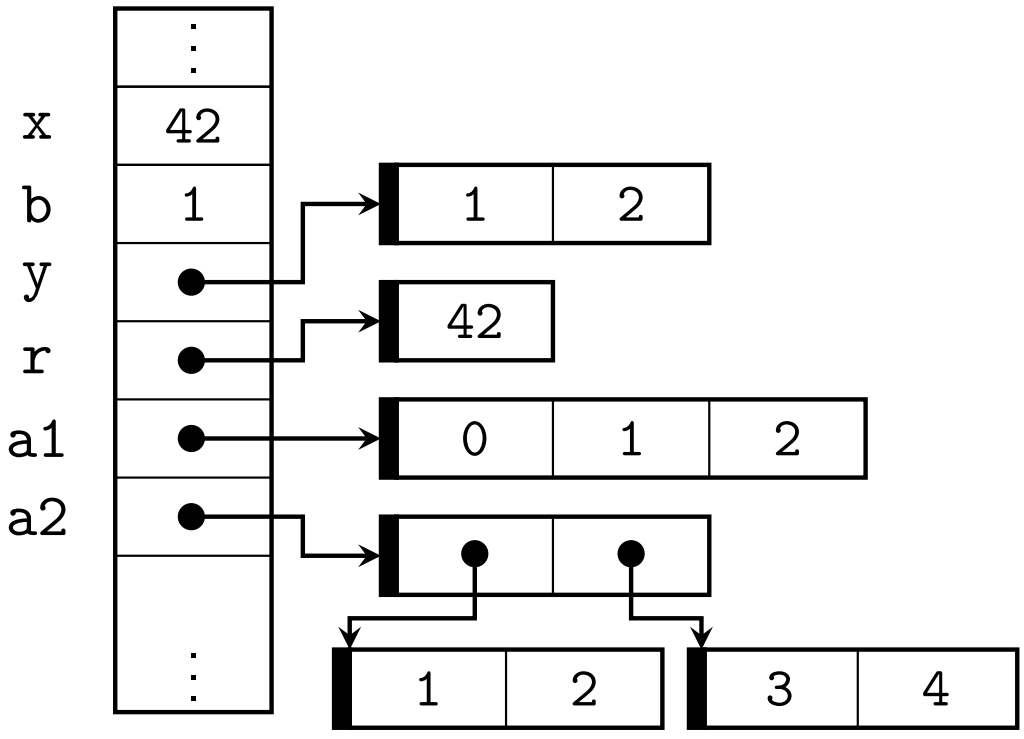
modèle d'exécution pour OCaml

OCaml implémente un modèle d'exécution basé sur une **représentation uniforme** des valeurs :

- les **valeurs sont toutes de la même taille : un mot machine** (un entier 64 bits sur x86_64)
- une valeur est **soit un entier**,
  soit un pointeur vers un bloc alloué sur le tas
- les blocs alloués sur le tas ne sont **jamais copiés** implicitement
- **les types sont effacés à l'exécution**

blocs et entiers

```
let x = 42
let b = true
let y = (1, 2)
let r = ref 42
let a1 = [| 0; 1; 2 |]
let a2 =
  [| (1, 2); (3, 4) |]
```



\+ types enregistrement (*records*) : pareil que les tuples ou tableaux

blocs et entiers : exemples

```
let x = ref (1, (2, 3))
```

blocs et entiers : exemples

```
let x = ref (1, (2, 3))

let x = ref (true, [|2, 3|])
```

blocs et entiers : exemples

```
let x = ref (1, (2, 3))

let x = ref (true, [|2, 3|])

let m1 = Array.make 3 (Array.make 3 0)
```

blocs et entiers : exemples

```
let x = ref (1, (2, 3))

let x = ref (true, [|2, 3|])

let m1 = Array.make 3 (Array.make 3 0)

let m2 = Array.init 3 (fun _ -> Array.make 3 0)
```

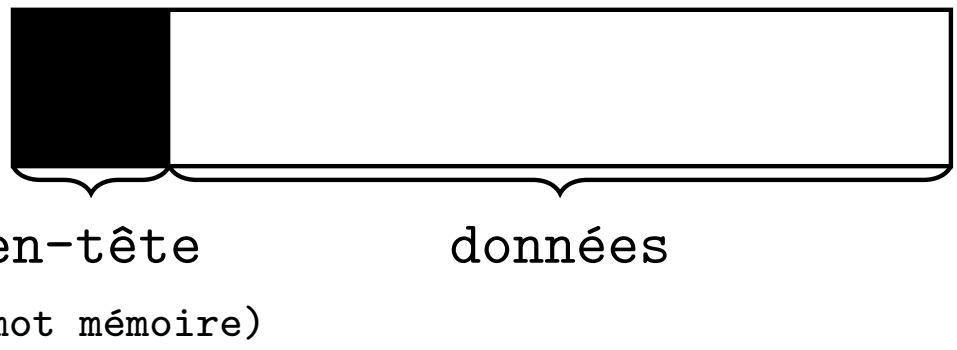
inspecter les valeurs d'un programme pour de vrai

grâce à une bibliothèque astucieuse (`memgraph`) on peut **inspecter et afficher la représentation de valeurs** directement depuis un programme OCaml

(Instructions d'installation détaillées en TP)

explore récursivement la mémoire correspondant à une valeur OCaml

anatomie d'un bloc : en-tête



l'en-tête d'un bloc contient des **metadonnées** :

la **taille** du bloc, par exemple lue par `Array.length`

le **tag**, entier qui indique comment interpréter le contenu du bloc :

- valeurs ocaml (tuples, tableaux, ...)
- octets bruts (chaines de caractères)
- pointeur de code + environnement (clotures, cf cours à venir)
- ...

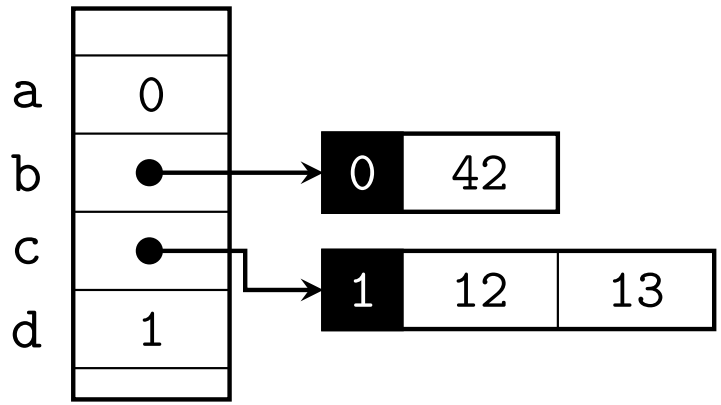
représentation des types de données algébriques

```
type t =
  | A
  | B of int
  | C of int * int
  | D

let a = A
let b = B 42
let c = C (12, 13)
let d = D
```

représentation des types de données algébriques

le `tag` intervient également dans la représentation des types algébriques

```
type t =
  | A
  | B of int
  | C of int * int
  | D


let a = A
let b = B 42
let c = C (12, 13)
let d = D
```



les constructeurs constants sont représentés par des **entiers**

les constructeurs non constants sont distingués par leur **tag**

entiers vs pointeurs

comment distinguer un **entier** d'un **pointeur sur un bloc** ?

$\rightarrow$ identifié par **un bit** dans chaque valeur OCaml

Conséquence : les entiers OCaml sont sur 63 bits !

bit utilisé par le pattern matching, mais aussi le gestionnaire de mémoire pour pouvoir explorer récursivement les valeurs OCaml vivantes

$\rightarrow$ cf. cours sur la gestion mémoire

cycle de vie d'un bloc

créé quand

- alloués par les **constructeurs** de valeurs (`(_, _)`, `[| ... |]`, etc)
  jamais de copie implicite ! (copie = prend du temps)
- lorsqu'un bloc n'est plus utilisé, la mémoire correspondante est **automatiquement libérée** et réutilisée (grâce au GC – *garbage collector*, cf cours à venir)

consommation mémoire d'un programme : application

Conséquence de ce modèle d'exécution : **pas d'allocations cachées**.

Chaque allocation est identifiable *dans le code source du programme* : seuls les constructeurs de valeurs allouent des nouveaux blocs.

combien de blocs mémoire alloue le programme suivant ? combien de mots mémoire ?

```
let a = [|1;2;3|]
let p = (a, a)
let b = Array.append a a
```

à l'exécution, que reste-t-il des types ?

à l'exécution, que reste-t-il des types ?

les types ocaml sont vérifiés **à la compilation**
ils sont ensuite **effacés** et n'existent pas à l'exécution

ces valeurs ont la même représentation à l'exécution :

à l'exécution, que reste-t-il des types ?

les types ocaml sont vérifiés **à la compilation**
ils sont ensuite **effacés** et n'existent pas à l'exécution

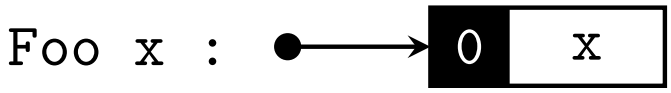
ces valeurs ont la même représentation à l'exécution :

- `(1, 2)` vs `[| 1; 2 |]`
- `Some 10` vs `ref 10`
- `false, true` vs `A, B` pour `type t = A|B`
- `A, (B 42)` vs `Foo, (Bar 42)`
  pour `type t = A | B of int`
  `type u = Foo | Bar of int`

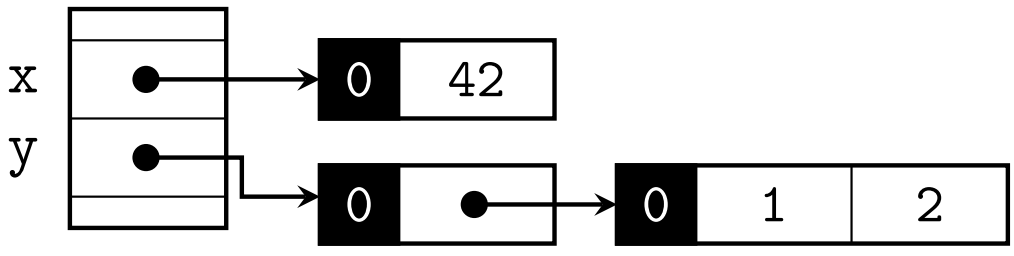
types de données polymorphes

les types de données polymorphes sont compilés en effaçant simplement les types

```
type 'a t =
  | Foo of 'a
```




```
let x = Foo 42
let y = Foo (1, 2)
```



possible grâce à la **représentation uniforme** des valeurs OCaml : toutes les valeurs (pointeur ou entier) ont la même taille.

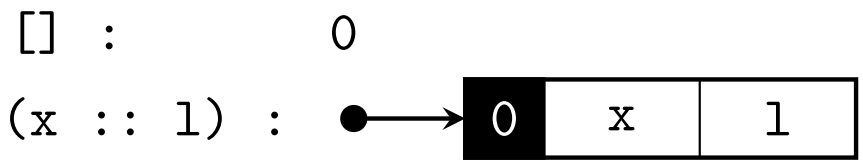
même stratégie pour la compilation de fonctions polymorphes

représentation des listes

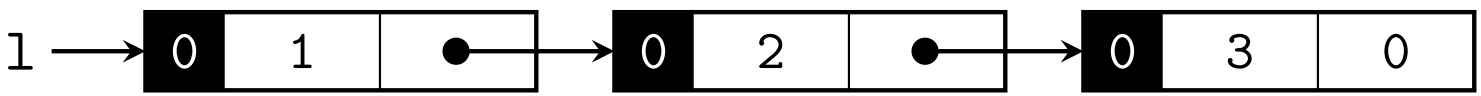
```
type 'a list =
  | []
  | (::) of 'a * 'a list
```

représentation des listes

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




```
let l = [1; 2; 3] = 1 :: (2 :: (3 :: []))
```



représentation des valeurs OCaml : pour aller plus loin

- Real World OCaml, Chapitre 23 : `dev.realworldocaml.org/runtime-memory-layout.html`
- Manuel OCaml, Chapitre 20, notamment Sections 2-4 : `ocaml.org/manual/intfc.html`

partage et persistance

à quoi bon connaitre la représentation des valeurs OCaml ?

Un autre modèle d'exécution pour OCaml : une sémantique opérationnelle « standard » basée sur le lambda-calcul.

$$
\begin{array}{llll}
\mathit{Val} & v & ::= & () \mid \mathrm{true} \mid \mathrm{false} \mid n \mid \ell \mid (v, v) \mid \ldots \\
\mathit{Expr} & e & ::= & x \mid n \mid () \mid \mathrm{if}\ e\ \mathrm{then}\ e\ \mathrm{else}\ e \mid \\
 & & & (e, e) \mid \mathrm{fst}\ e \mid \mathrm{snd}\ e \mid \lambda x.\, e \mid e\ e \mid \ldots \\
\mathit{Heap} & h & \in & \mathit{Loc} \rightharpoonup \mathit{Val}
\end{array}
$$

$$
\begin{array}{l}
(\mathrm{if}\ \mathrm{true}\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2, h) \rightsquigarrow (e_1, h) \\
(\mathrm{if}\ \mathrm{false}\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2, h) \rightsquigarrow (e_2, h) \\
(\mathrm{fst}(v_1, v_2), h) \rightsquigarrow (v_1, h) \\
\ldots
\end{array}
$$

à quoi bon connaitre la représentation des valeurs OCaml ?

connaître une représentation **en terme de pointeurs et de blocs** :

permet de raisonner plus finement sur les performances et la consommation mémoire de son code (jusqu'à un certain point)

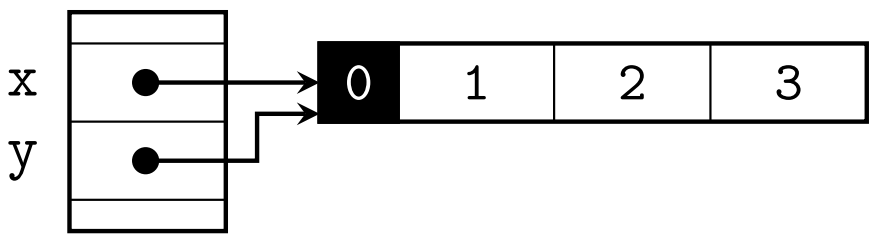
- les allocations sont identifiables depuis le code source
- jamais de copies implicites

indispensable pour comprendre et implémenter des programmes tirant parti du **partage**

partage

il n'est pas difficile de créer plusieurs pointeurs vers un même bloc :

```
let x = [|1;2;3|] in
let y = x in
x.(0) <- 42;
(* y.(0) = 42 *)
```



Piège classique :

```
   Array.make 3 (Array.make 3 0)
```

vs `Array.init 3 (fun _ -> Array.make 3 0)`

on crée du **partage** alors qu'on n'en voulait pas !

partage

- en présence de valeurs **mutables** :
  partage directement observable, **souvent piégeux**
  (mais parfois utile)

- avec des valeurs **immuables** :
  partage **très courant**, et **crucial** pour la construction efficace de structures de données **persistantes**

structures de données immuables

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

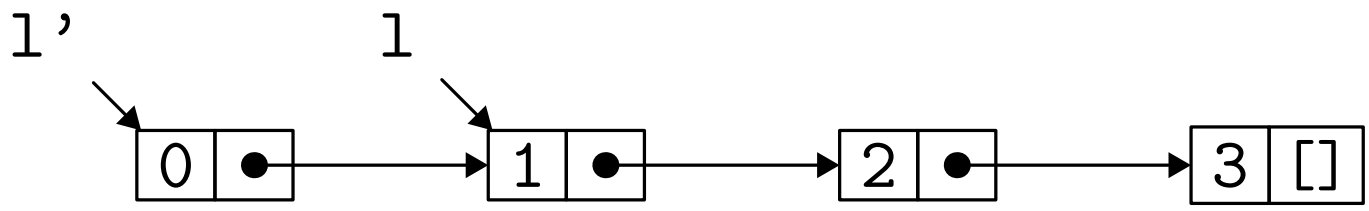
dit autrement :

- 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** ou code **purement fonctionnel**.

exemple de structure immuable : les listes

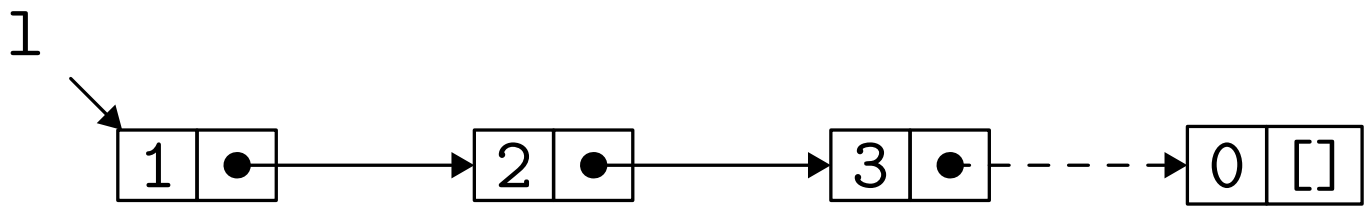
```
let l = [1; 2; 3]
let l' = 0 :: l
```



pas de copie, mais **partage**

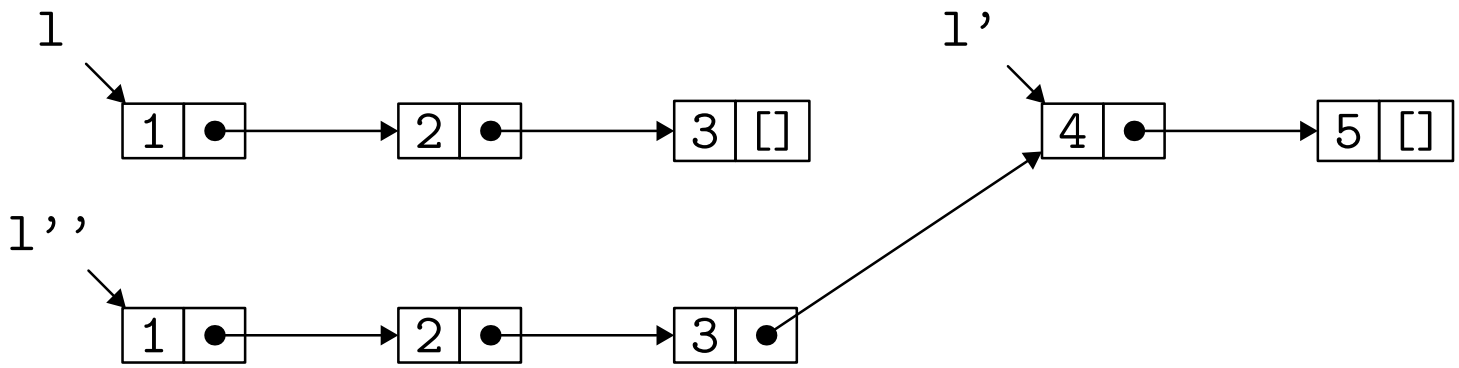
contrepartie

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'
```

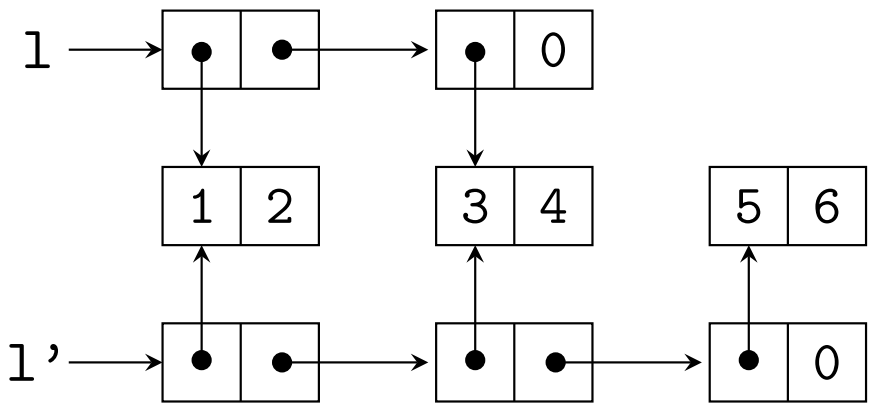


blocks de `l` **copiés**, blocs de `l'` **partagés**

concaténation de deux listes

les *éléments* de la liste sont eux-mêmes toujours **partagés** :

```
let l = [(1, 2); (3, 4)]
let l' = append l [(5, 6)]
```



listes chaînées mutables

note : on peut définir des listes chaînées mutables, par exemple ainsi :

```
type 'a mlist =
  | Empty
  | Element of { value : 'a; mutable next : 'a mlist }
```

mais il faut alors faire très attention au **partage**

intérêts pratiques de la persistance

les structures immuables sont donc **persistantes** : une opération ne modifie pas la structure mais renvoie une **nouvelle version**

Quel intérêt ?

1. **correction** des programmes
   - code plus simple
   - raisonnement mathématique possible
2. Seule possibilité dans les langages sans structures mutables ! (Haskell, Clojure, Coq, ...)
3. outil puissant pour le **rebroussement** (*backtracking*)
   - algorithmes de recherche
   - récupération suite à une erreur

persistance et rebroussement (1)

programme manipulant une base de connaissances

mise à jour complexe, nécessitant de nombreuses opérations

avec une structure modifiée **en place** :

```
let base : (_, _) Hashtbl.t = Hashtbl.create 37
... initialisation de la base ...
try
   ... effectuer l'opération de mise à jour ...
with exn -> (* oh no ! *)
   ... rétablir la base dans un état cohérent ...
   ... traiter ensuite l'erreur ...
```

persistance et rebroussement (1)

avec une structure **persistante** :

```
let base : 'a tree = ... base initiale ...
...
let nouvelle_base =
  try
    ... calcul de la base mise à jour ...
  with exn ->
    ... traiter l'erreur ...;
    base  (* valeur d'avant la tentative de mise à jour *)
```

persistance et rebroussement (2)

recherche de la sortie dans un labyrinthe

```
type etat
val sortie : etat -> bool
type deplacement
val deplacements : etat -> deplacement list
val deplace : etat -> deplacement -> etat


let rec cherche e =
  sortie e || essaye e (deplacements e)
and essaye e = function
  | []      -> false
  | d :: ds -> cherche (deplace d e) || essaye e ds
```

sans persistance

avec un état modifié en place :

```
val deplace : etat -> deplacement -> unit
val revient : deplacement -> etat -> unit


let rec cherche e =
  sortie e || essaye e (deplacements e)
and essaye e = function
  | []      -> false
  | d :: ds ->
    (deplace d e; cherche e) || (revient d e; essaye e ds)
```

il faut **annuler** l'effet de bord manuellement

persistance avec effets de bord

persistance et effets de bord

Attention : structure persistante ne signifie pas sans effet de bord !

$$\textit{persistant} = \textbf{observationellement}\ \textit{immuable}$$

on a donc :

$$\textit{immuable} \Rightarrow \textit{persistant}$$

mais la réciproque est fausse...

persistance et effets de bord

Attention : structure persistante ne signifie pas sans effet de bord !

$$\textit{persistant} = \textbf{observationellement}\ \textit{immuable}$$

on a donc :

$$\textit{immuable} \Rightarrow \textit{persistant}$$

mais la réciproque est fausse...

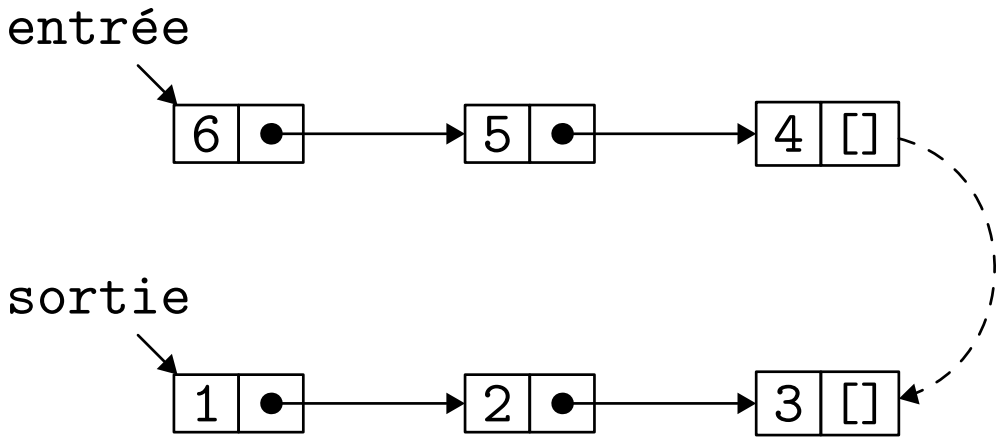
$\rightarrow$ il existe des structures **persistantes** avec **effets de bord "cachés"** !

exemple 1 : files persistantes

```
type 'a t
val empty : 'a t
val push : 'a -> 'a t -> 'a t
val pop : 'a t -> ('a * 'a t) option
```

exemple 1 : 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 1 : files persistantes

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

let empty = ([], [])

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

let pop = function
  | e, x :: s -> Some (x, (e, s))
  | e, []     ->
    match List.rev e with
    | x :: s -> Some (x, ([], s))
    | []     -> None
```

exemple 1 : files persistantes

Problème d'efficacité :

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

```
let q = push 1 (push 2 (push 3 empty)) in
let res = pop q in (* calcule List.rev [3;2;1] *)
...
let res = pop q in (* recalcule List.rev [3;2;1] *)
...
```

exemple 1 : files persistantes

Idée :

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

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

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

exemple 1 : files persistantes

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

let empty = ref ([], [])

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

let pop q =
  match !q with
  | e, x :: s -> Some (x, ref (e, s))
  | e, []     ->
    match List.rev e with
    | x :: s as r -> q := ([], r); Some (x, ref ([], s))
    | []          -> None
```

exemple 1 : files persistantes

En fait, cette idée **ne suffit pas** dans le cas général pour avoir une complexitée en $O(1)$ amorti pour `push` et `pop` dans tous les cas.

Penser au cas où on pop plusieurs fois à partir d'une file avec un élément dans la liste de sortie... $\rightarrow$ `https://gist.github.com/Armael/b0ca3dbd00eca84dd85d066eafefb5cd`

exemple 1 : files vraiment persistantes

Une solution possible : la file persistante avec `push` et `pop` en *O(1) amorti* du Chapitre §6.4.2 de *Purely Functional Data Structures* (Okasaki), utilisant une **suspension** (`lazy`)[1].

```
type 'a queue = 'a list * int * 'a list lazy * int * 'a list
```

`lazy` : une forme (restreinte) d'effet de bord !

1. avec une preuve de complexité en Coq ! `https://gitlab.inria.fr/cambium/iris-time-proofs/-/tree/master/theories/pqueue`

exemple 2 : splay trees

structure d'arbre implémentant des **tables associatives** persistantes

```
type key
type 'a t
val empty : 'a t
val add : key -> 'a -> 'a t -> 'a t
val find : key -> 'a t -> 'a
```

même interface que le module `Map` de la bibliothèque standard

persistante : `add` renvoie une **nouvelle** table

exemple 2 : splay trees

twist : les splay trees sont "**self-adjusting**"

```
type 'a t = 'a tree ref
```

```
val find : key -> 'a t -> 'a
```

`find` fait un **effet de bord caché** : rebalance l'arbre dans la référence optimise de futurs recherches pour la même clef

exemple 2 : splay trees (2)

statistiquement très intéressant dès que les accès à la structure ne sont pas complètement aléatoires

Exemples de rebalancements :
`https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf#page=6`

(*Self-Adjusting Binary Search Trees. Sleator, Tarjan. 1985*)

persistance et effets de bord : autres exemples

pour aller plus loin, autres exemples de structures persistantes utilisant des effets de bord « sous le capot » :

- structures utilisant la paresse (`lazy`) dans *Purely Functional Data Structures* de C. Okasaki
- tableaux persistants de J.C. Filliâtre (**en TP cette après-midi**)
- la bibliothèque `Sek`[2] : séquences persistantes et éphémères avec conversions efficaces entre les deux (voir aussi la formalisation en Coq de la correction *et complexité* d'un sous-ensemble de `Sek` ![3])

2. `https://gitlab.inria.fr/fpottier/sek`
3. `https://gitlab.inria.fr/amoine/cfml-sek`

pour récapituler

OCaml : représentation uniforme des valeurs

une valeur : entier ou pointeur vers un bloc mémoire

les blocs sont alloués par les constructeurs de valeurs, et désalloués automatiquement

ce qui rend les structures de données exploitant le partage très naturelles

application : structures persistantes = pas de modifications observables (en OCaml : `List`, `Set`, `Map`, ...)