Arbres et Listes Paresseuses

Alan Schmitt

3 octobre 2017

Les Arbres

Arbres d'entiers

```
type arbin =
| Feuille of int
| Noeud of arbin * arbin
```

```
type arbin = Feuille of int | Noeud of arbin * arbin
```

Les objets de type arbin sont :

- soit réduits à une feuille contenant un entier
- soit un noeud qui regroupe deux valeurs de type arbin

Arbres d'entiers

```
let _ = Feuille 2
```

```
- : arbin = Feuille 2
```

```
let _ = Noeud
```

```
Characters 8-13:
  let _ = Noeud;;
          ^^^^^
Error: The constructor Noeud expects 2 argument(s),
       but is applied here to 0 argument(s)
```

```
let _ = Noeud(Noeud(Feuille 2,Feuille 7),Feuille 4)
```

```
- : arbin = Noeud (Noeud (Feuille 2, Feuille 7), Feuille 4)
```

Les arbres binaires (1)

Exercices :

- Définir la fonction qui construit le doublet (valinf,valsup) contenant la plus petite et la plus grande valeur contenue dans un arbin. Vous pouvez utiliser les fonctions suivantes

```
min                    - : 'a -> 'a -> 'a = <fun>
```

```
max                    - : 'a -> 'a -> 'a = <fun>
```

- Définir la fonction qui construit un arbin symétrique d'un arbin donné

Les arbres binaires (2)

```
let rec interv a = match a with
| Feuille x  -> (x,x)
| Noeud(g,d) -> let ig=interv g
                and id=interv d in
                min (fst ig)(fst id),
                max (snd ig)(snd id)
```

```
val interv : arbin -> int * int = <fun>
```

```
let rec interv a = match a with
| Feuille x   -> (x,x)
| Noeud (g,d) -> let (mig,mag)= interv g
                 and (mid,mad)= interv d in
                 min mig mid, max mag mad
```

```
val interv : arbin -> int * int = <fun>
```

Les arbres binaires (3)

Construction arbre symétrique

```
let rec sym a = match a with
| Feuille x    -> Feuille x
| Noeud (g,d) -> Noeud (sym d,sym g)
```

```
val sym : arbin -> arbin = <fun>
```

Arbre binaire polymorphe

Comme pour les listes, on peut définir des arbres polymorphes.

```
type 'a arbre = Feuille of 'a | Noeud of 'a arbre * 'a arbre
```

```
type 'a arbre = Feuille of 'a | Noeud of 'a arbre * 'a arbre
```

```
let _ = Noeud (Feuille 2., Feuille 4.)
```

```
- : float arbre = Noeud (Feuille 2., Feuille 4.)
```

```
let _ = Noeud(Feuille (Feuille 2),
              Feuille (Noeud(Feuille 4,
                             Feuille 17)))
```

```
- : int arbre arbre =
Noeud (Feuille (Feuille 2), Feuille (Noeud (Feuille 4, Feuille 17)))
```

Arbre avec étiquettes aux nœuds

Un type peut avoir plusieurs paramètres

```
type ('a, 'b) idtype = ... ...
```

Exemple:

```
type ('a,'b) arbre2 =
| Feuille of 'a
| Noeud of 'b * ('a,'b) arbre2 * ('a,'b) arbre2
```

```
type ('a, 'b) arbre2 =
    Feuille of 'a
  | Noeud of 'b * ('a, 'b) arbre2 * ('a, 'b) arbre2
```

Exemple : l'Arbre de Huffman

L'arbre de Huffman

Le probleme

Trouver un codage des caractères de façon à réduire la taille d'un texte. Cette technique de compression de données est appelée codage de Huffman

Le Principe

Partir d'une table contenant les fréquences d'apparition des lettres dans les textes en langue française. Les lettres les plus fréquentes auront un code court, les moins fréquentes un code long.

a	10
b	01
c	110
d	111
e	001
f	0001

`000110111001` code le mot `fade`

L'arbre de Huffman : propriété

Propriété

Les codes étant de longueur différente, pour que le décodage soit possible, il faut qu'aucun code ne soit préfixe d'un autre code.

Technique

Construire un arbre à partir de la table des fréquences
Cet arbre (de Huffman) est un arbre binaire dont les feuilles sont les lettres : le parcours depuis la racine jusqu'à la feuille étiquetée x détermine le code de la lettre x, en notant 0 le choix du sous-arbre gauche, 1 le choix du sous-arbre droit
(la construction de l'arbre revient à attribuer un code « optimal » à chaque lettre)

L'arbre de Huffman : le type

```
type huff = F of char | N of huff * huff
```

```
type huff = F of char | N of huff * huff
```

```
type bit  = Z | U
```

```
type bit = Z | U
```

L'arbre de Huffman : les données

Pour construire l'arbre qui attribue les codes optimaux (ceux qui permettent le plus important compactage) il faut disposer des fréquences d'apparitions de chaque lettre.

```
let lc =
[('w',1);('k',1);('z',101);('y',344);('j',377);('x',819);('h',916);
 ('g',1384);('b',1532);('v',1614);('q',1617);('f',1774);('m',4113);
 ('p',4569);('c',4983);('d',5591);('l',7607);('u',8715);('o',8720);
 ('i',9713);('r',9824);('a',9981);('t',10377);('n',10382);
 ('s',11527);('e',24549);(' ',37312)]
```

```
val lc : (char * int) list =
  [('w', 1); ('k', 1); ('z', 101); ('y', 344); ('j', 377); ('x', 819);
   ('h', 916); ('g', 1384); ('b', 1532); ('v', 1614); ('q', 1617);
   ('f', 1774); ('m', 4113); ('p', 4569); ('c', 4983); ('d', 5591);
   ('l', 7607); ('u', 8715); ('o', 8720); ('i', 9713); ('r', 9824);
   ('a', 9981); ('t', 10377); ('n', 10382); ('s', 11527); ('e', 24549);
   (' ', 37312)]
```

L'arbre de Huffman : solution (1)

- Créer une liste d'arbres de Huffman à partir d'une liste de fréquences:

```
let creerlh l = List.map (fun (c,i) -> (F c,i)) l
```

```
val creerlh : (char * 'a) list -> (huff * 'a) list = <fun>
```

- Agglomérer deux arbres de Huffman (associés à la fréquence):

```
let agglo = fun (h1,f1) (h2,f2) -> (N (h1,h2)), f1+f2
```

```
val agglo : huff * int -> huff * int -> huff * int = <fun>
```

L’arbre de Huffman : solution (2)

- Insérer un arbre dans une liste d’arbres ordonnée selon la fréquence

```
let rec insere (h,f) l = match l with
| [] -> [(h,f)]
| ((h1,f1)::r) -> if f<f1
                    then(h,f)::(h1,f1)::r
                    else (h1,f1)::insere (h,f) r
```

```
val insere : 'a * 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

L'arbre de Huffman : solution (3)

- Créer l'arbre en partant de la liste d'arbres

```
let reduire (c1::c2::r) = insere (agglo c1 c2) r
```

```
Characters 12-48:
  let reduire (c1::c2::r) = insere (agglo c1 c2) r;;
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val reduire : (huff * int) list -> (huff * int) list = <fun>
```

```
let rec faitHuff l = match l with
| [(h,f)] -> h
| _ -> faitHuff (reduire l)
```

```
val faitHuff : (huff * int) list -> huff = <fun>
```

L'arbre de Huffman : solution (4)

- On fabrique la table

```
let h = faitHuff (creerlh lc)
```

```
val h : huff =
  N (N (F ' ', N (N (N (F 'p', F 'c'), F 'i'), N (F 'r', F 'a'))),
   N
    (N (N (F 't', F 'n'),
      N (F 's',
       N (F 'd',
        N (N (F 'b', F 'v'),
         N (F 'q',
          N (F 'x', N (F 'j', N (N (N (F 'w', F 'k'), F 'z'), F 'y')))))
     ))),
    N (F 'e',
     N (N (F 'l', N (N (F 'f', N (F 'h', F 'g')), F 'm')),
      N (F 'u', F 'o')))))
```

L'arbre de Huffman : solution (5)

On extrait de l'arbre une table de codage

```
let huff_to_code h =
  let rec aux path res a = match a with
  | F c -> (c, List.rev path) :: res
  | N (l,r) ->
      let resl = aux (Z :: path) res l in
      let resr = aux (U :: path) resl r in
      resr
  in aux [] [] h
```

```
val huff_to_code : huff -> (char * bit list) list = <fun>
```

L'arbre de Huffman : solution (6)

On conclut

```
let code =
  let path_to_string p =
    String.concat ""
      (List.map (fun c -> match c with Z -> "0" | U -> "1") p) in
  List.iter (fun (c,p) -> Printf.printf "%c: %s\n" c (path_to_string p))
    (huff_to_code h)
```

```
o: 11111
u: 11110
m: 111011
g: 11101011
h: 11101010
f: 1110100
l: 11100
e: 110
y: 1011111111
z: 10111111101
k: 101111111001
w: 101111111000
j: 101111110
x: 10111110
q: 1011110
v: 1011101
b: 1011100
d: 10110
s: 1010
n: 1001
t: 1000
a: 0111
r: 0110
i: 0101
c: 01001
p: 01000
 : 00
val code : unit = ()
```

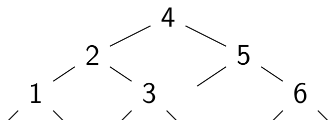
Arbres de Recherche

Arbres Binaires de Recherche

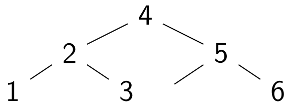
On s'intéresse à des abres binaires portant leurs valeurs aux nœuds

```
type 'a abr = Empty | Node of 'a abr * 'a * 'a abr
```

```
type 'a abr = Empty | Node of 'a abr * 'a * 'a abr
```



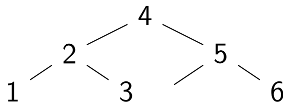
Pour faire plus court, on ne dessine pas les fils quand les deux sont vides



Arbres Binaires de Recherche

Ces arbres sont ordonnés: pour tout nœud `(l,v,r)`:

- les valeurs dans `l` sont toutes strictement plus petites que `v`;
- les valeurs dans `r` sont toutes strictement plus grandes que `v`;



Arbres Binaires de Recherche: Fonctions de base

Échauffement: calculer la hauteur d'un arbre

Arbres Binaires de Recherche: Fonctions de base

Échauffement: calculer la hauteur d'un arbre

```
let rec height a = match a with
  | Empty -> 0
  | Node (l, _, r) -> 1 + max (height l) (height r)
```

```
val height : 'a abr -> int = <fun>
```

Arbres Binaires de Recherche: Fonctions de base

Échauffement: calculer la hauteur d'un arbre

```
let rec height a = match a with
  | Empty -> 0
  | Node (l, _, r) -> 1 + max (height l) (height r)
```

```
val height : 'a abr -> int = <fun>
```

Retourner la plus petite valeur d'un arbre

Arbres Binaires de Recherche: Fonctions de base

Échauffement: calculer la hauteur d'un arbre

```
let rec height a = match a with
  | Empty -> 0
  | Node (l, _, r) -> 1 + max (height l) (height r)
```

```
val height : 'a abr -> int = <fun>
```

Retourner la plus petite valeur d'un arbre

```
let rec min_elt a = match a with
  | Empty -> assert false
  | Node (Empty, v, _) -> v
  | Node (l, _, _) -> min_elt l
```

```
val min_elt : 'a abr -> 'a = <fun>
```

Arbres Binaires de Recherche: Fonctions de base

Échauffement: calculer la hauteur d'un arbre

```
let rec height a = match a with
  | Empty -> 0
  | Node (l, _, r) -> 1 + max (height l) (height r)
```

```
val height : 'a abr -> int = <fun>
```

Retourner la plus petite valeur d'un arbre

```
let rec min_elt a = match a with
  | Empty -> assert false
  | Node (Empty, v, _) -> v
  | Node (l, _, _) -> min_elt l
```

```
val min_elt : 'a abr -> 'a = <fun>
```

Indiquer si une valeur `x` se trouve dans un arbre

Arbres Binaires de Recherche: Fonctions de base

Échauffement: calculer la hauteur d'un arbre

```
let rec height a = match a with
  | Empty -> 0
  | Node (l, _, r) -> 1 + max (height l) (height r)
```

```
val height : 'a abr -> int = <fun>
```

Retourner la plus petite valeur d'un arbre

```
let rec min_elt a = match a with
  | Empty -> assert false
  | Node (Empty, v, _) -> v
  | Node (l, _, _) -> min_elt l
```

```
val min_elt : 'a abr -> 'a = <fun>
```

Indiquer si une valeur x se trouve dans un arbre

```
let rec mem x a = match a with
  | Empty -> false
  | Node (l, v, r) ->
     x = v || if x < v then mem x l else mem x r
```

```
val mem : 'a -> 'a abr -> bool = <fun>
```

Arbres Binaires de Recherche: Ajouter un élément

Coder une fonction ajoutant un élément `x` à `t`

Arbres Binaires de Recherche: Ajouter un élément

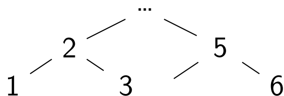
Coder une fonction ajoutant un élément `x` à `t`

```
let rec add x t =
  match t with
  | Empty -> Node (Empty, x, Empty)
  | Node (l, v, r) ->
    if x = v then t
    else if x < v then Node (add x l, v, r)
    else Node (l, v, add x r)
```

```
val add : 'a -> 'a abr -> 'a abr = <fun>
```

Arbres Binaires de Recherche: Enlever un élément

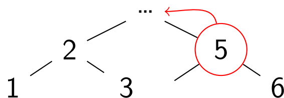
```
let rec remove x a = match a with
  | Empty -> Empty
  | Node (l, v, r) ->
      if x = v then ...
      else if x < v then Node (remove x l, v, r)
      else Node (l, v, remove x r)
```



Deux solutions:

Arbres Binaires de Recherche: Enlever un élément

```
let rec remove x a = match a with
  | Empty -> Empty
  | Node (l, v, r) ->
      if x = v then ...
      else if x < v then Node (remove x l, v, r)
      else Node (l, v, remove x r)
```

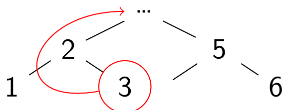


Deux solutions:

1. faire remonter le plus petit élément du sous-arbre droit

Arbres Binaires de Recherche: Enlever un élément

```
let rec remove x a = match a with
  | Empty -> Empty
  | Node (l, v, r) ->
      if x = v then ...
      else if x < v then Node (remove x l, v, r)
      else Node (l, v, remove x r)
```



Deux solutions:

1. faire remonter le plus petit élément du sous-arbre droit
2. faire remonter le plus grand élément du sous-arbre gauche

Arbres Binaires de Recherche: Enlever un élément

```
let rec remove_min_elt a = match a with
  | Empty -> Empty
  | Node (Empty, _, r) -> r
  | Node (l, v, r) -> Node (remove_min_elt l, v, r)
```

```
val remove_min_elt : 'a abr -> 'a abr = <fun>
```

Arbres Binaires de Recherche: Enlever un élément

```
let rec remove_min_elt a = match a with
  | Empty -> Empty
  | Node (Empty, _, r) -> r
  | Node (l, v, r) -> Node (remove_min_elt l, v, r)
```

```
val remove_min_elt : 'a abr -> 'a abr = <fun>
```

```
let merge t1 t2 = match t1, t2 with
  | Empty, t | t, Empty -> t
  | _ -> Node (t1, min_elt t2, remove_min_elt t2)
```

```
val merge : 'a abr -> 'a abr -> 'a abr = <fun>
```

Arbres Binaires de Recherche: Enlever un élément

```
let rec remove_min_elt a = match a with
  | Empty -> Empty
  | Node (Empty, _, r) -> r
  | Node (l, v, r) -> Node (remove_min_elt l, v, r)
```

```
val remove_min_elt : 'a abr -> 'a abr = <fun>
```

```
let merge t1 t2 = match t1, t2 with
  | Empty, t | t, Empty -> t
  | _ -> Node (t1, min_elt t2, remove_min_elt t2)
```

```
val merge : 'a abr -> 'a abr -> 'a abr = <fun>
```

```
let rec remove x a = match a with
  | Empty -> Empty
  | Node (l, v, r) ->
      if x = v then merge l r
      else if x < v then Node (remove x l, v, r)
      else Node (l, v, remove x r)
```

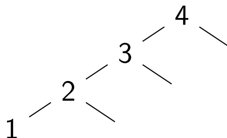
```
val remove : 'a -> 'a abr -> 'a abr = <fun>
```

Le problème du peigne

Si on ajoute les éléments dans l'ordre ou l'ordre inverse, on obtient un peigne

```
let _ = add 1 (add 2 (add 3 (add 4 Empty)))
```

```
- : int abr =
Node (Node (Node (Node (Empty, 1, Empty), 2, Empty), 3, Empty), 4,
 Empty)
```



Solution: arbre équilibré ou AVL (Adelson-Velskii et Landis)

AVL

Les arbres mentionnent leur hauteur

```
type 'a avl = Empty | Node of 'a avl * 'a * 'a avl * int
```

```
type 'a avl = Empty | Node of 'a avl * 'a * 'a avl * int
```

```
let height a = match a with
  | Empty -> 0
  | Node (_, _, _, h) -> h
```

```
val height : 'a avl -> int = <fun>
```

On se donne une fonction pour créer des nœuds plus facilement

```
let node l v r =
  Node (l, v, r, 1 + max (height l) (height r))
```

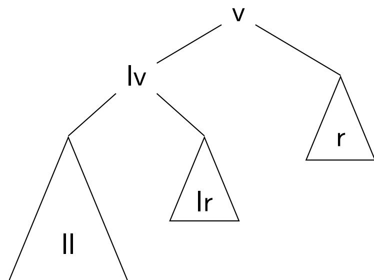
```
val node : 'a avl -> 'a -> 'a avl -> 'a avl = <fun>
```

Équilibrer un AVL

Ce que l'on veut: les tailles des branches gauches et droites ont une différence de taille d'au plus 1.

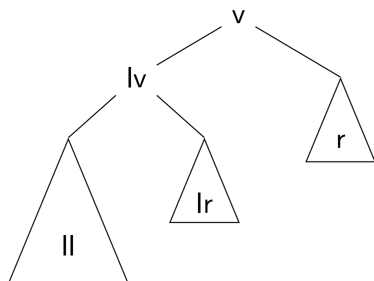
On suppose que le sous-arbre gauche est plus long. L'autre cas est symmétrique.

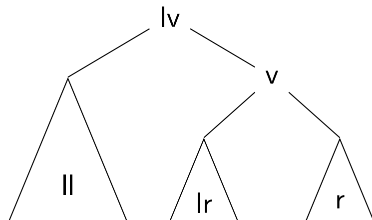
Si le sous-arbre `ll` est plus haut que le sous-arbre `lr`, on fait une rotation autour de `lv`.



Équilibrer un AVL

Ce que l'on veut: les tailles des branches gauches et droites ont une différence de taille d'au plus 1.
On suppose que le sous-arbre gauche est plus long. L'autre cas est symmétrique.
Si le sous-arbre `ll` est plus haut que le sous-arbre `lr`, on fait une rotation autour de `lv`.



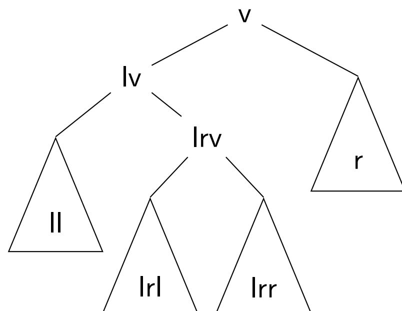


Équilibrer un AVL

```
let balance l v r =
  let hl = height l in
  let hr = height r in
  if hl > hr + 1 then begin
    match l with
    | Node (ll, lv, lr, _) ->
        if height ll >= height lr then node ll lv (node lr v r)
        else ...
    | _ -> assert false (* height l > height r + 1 *)
  end else ...
```

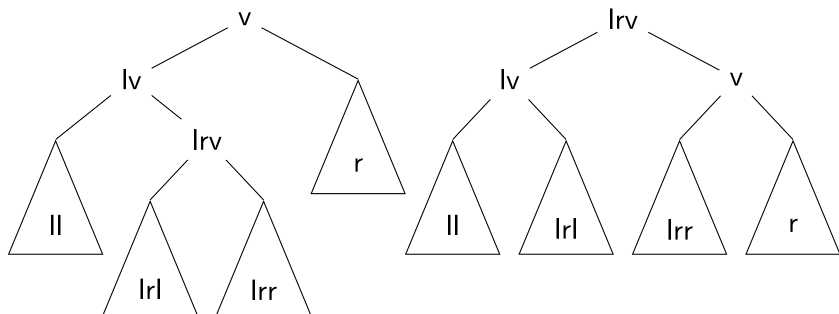
Équilibrer un AVL

Si le sous-arbre `ll` est plus court que le sous-arbre à sa droite, on fait une double rotation.



Équilibrer un AVL

Si le sous-arbre `ll` est plus court que le sous-arbre à sa droite, on fait une double rotation.



Équilibrer un AVL

```
let balance l v r =
  let hl = height l in
  let hr = height r in
  if hl > hr + 1 then begin
    match l with
    | Node (ll, lv, lr, _) ->
        if height ll >= height lr then ...
        else begin
          match lr with
          | Node (lrl, lrv, lrr, _) ->
              node (node ll lv lrl) lrv (node lrr v r)
          | _ -> assert false (* height lr > height ll *)
        end
    | _ -> assert false (* height l > height r + 1 *)
  end else ...
```

Équilibrer un AVL

```
let balance l v r = let hl = height l in let hr = height r in
  if hl > hr + 1 then begin
    match l with
    | Node (ll, lv, lr, _) ->
        if height ll >= height lr then node ll lv (node lr v r)
        else begin
          match lr with
          | Node (lrl, lrv, lrr, _) ->
              node (node ll lv lrl) lrv (node lrr v r)
          | _ -> assert false (* height lr > height ll *)
        end
    | _ -> assert false (* height l > height r + 1 *)
  end else if hr > hl + 1 then begin
    match r with
    | Node (rl, rv, rr, _) ->
        if height rr >= height rl then node (node l v rl) rv rr
        else begin
          match rl with
          | Node (rll, rlv, rlr, _) ->
              node (node l v rll) rlv (node rlr rv rr)
          | _ -> assert false (* height rl > height rr *)
        end
    | _ -> assert false (* height r > height l + 1 *)
  end else (* already balanced *) node l v r
```

```
val balance : 'a avl -> 'a -> 'a avl -> 'a avl = <fun>
```

Ajouter un élément dans un AVL

```
let rec add x t =
  match t with
  | Empty -> node Empty x Empty
  | Node (l, v, r, _) ->
    if x = v then t
    else if x < v then balance (add x l) v r
    else balance l v (add x r)
```

```
val add : 'a -> 'a avl -> 'a avl = <fun>
```

Supprimer un élément dans un AVL 1/2

```ocaml
let rec remove_min_elt a = match a with
  | Empty -> Empty
  | Node (Empty, _, r, _) -> r
  | Node (l, v, r, _) -> balance (remove_min_elt l) v r
```

```
val remove_min_elt : 'a avl -> 'a avl = <fun>
```

```ocaml
let rec min_elt a = match a with
  | Empty -> raise Not_found
  | Node (Empty, v, _, _) -> v
  | Node (l, _, _, _) -> min_elt l
```

```
val min_elt : 'a avl -> 'a = <fun>
```

Supprimer un élément dans un AVL 2/2

```
let merge t1 t2 = match t1, t2 with
  | Empty, t | t, Empty -> t
  | _ -> balance t1 (min_elt t2) (remove_min_elt t2)
```

```
val merge : 'a avl -> 'a avl -> 'a avl = <fun>
```

```
let rec remove x a = match a with
  | Empty -> Empty
  | Node (l, v, r, _) ->
      if x = v then merge l r
      else if x < v then balance (remove x l) v r
      else balance l v (remove x r)
```

```
val remove : 'a -> 'a avl -> 'a avl = <fun>
```

Les Listes Paresseuses

Motivation

Manipuler des listes produites à la demande

- listes infinies (la liste de tous les entiers)
- flots de données (flots de caractères)

Comment gérer l'infini ?

Idée fondamentale

- une fonction `fun () -> expr` est « gelée », c'est déjà une valeur
- On veut donc appeler une fonction avec `()`, qui va faire quelque chose et nous retourner une fonction gelée, que l'on pourra appeler de nouveau. Son type est donc de la forme `() -> () -> () -> …`
- c'est impossible : type récursif de la forme `'a = unit -> 'a`

```
let rec gel n =
  let _ = Printf.printf "%d\n" n in
  (fun () -> gel (n+1))
```

```
Characters 65-74:
    (fun () -> gel (n+1));;
               ^^^^^^^^^^
Error: This expression has type unit -> 'a
       but an expression was expected of type 'a
       The type variable 'a occurs inside unit -> 'a
```

Type récursif

Solution utiliser un type avec récursion

```
type 'a gel = Gel of (unit -> 'a gel)
```

```
type 'a gel = Gel of (unit -> 'a gel)
```

On peut maintenant définir

```
let rec gel n =
  let _ = Printf.printf "%d\n" n in
  Gel (fun () -> gel (n+1))

let Gel f1 = gel 1
let Gel f2 = f1 ()
let Gel f3 = f2 ()
```

```
1
2
3
val gel : int -> 'a gel = <fun>
val f1 : unit -> 'a gel = <fun>
val f2 : unit -> 'a gel = <fun>
val f3 : unit -> 'a gel = <fun>
```

Type des flots

Une liste paresseuse (un flot) est soit vide, soit elle a un premier élément et le reste de la liste. Ce reste est gelé

```
type 'a flot = Empty | Cons of 'a * (unit -> 'a flot)
```

```
type 'a flot = Empty | Cons of 'a * (unit -> 'a flot)
```

Exemple de liste paresseuse : la liste de tous les entiers

```
let rec les_entiers n = Cons (n, fun () -> les_entiers (n+1))

let v1,r1 = match les_entiers 1 with
| Cons (n, r) -> n,r
| _ -> assert false

let v2,r2 = match r1 () with
| Cons (n,r) -> n,r
| _ -> assert false
```

```
val les_entiers : int -> int flot = <fun>
val v1 : int = 1
val r1 : unit -> int flot = <fun>
val v2 : int = 2
val r2 : unit -> int flot = <fun>
```

Consommer des éléments d'un flot

Consommer les `n` premiers éléments d'un flot, et retourner la liste de ces éléments et le reste du flot

```
let rec take_n f n = match n with
  | 0 -> [], f
  | n -> begin
      match f with
      | Cons(e, r) -> let l,f' = take_n (r()) (n-1) in (e :: l),f'
      | Empty -> [],Empty
    end
```

```
val take_n : 'a flot -> int -> 'a list * 'a flot = <fun>
```

Exemple :

```
let l,r = take_n (les_entiers 1) 10
```

```
val l : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
val r : int flot = Cons (11, <fun>)
```

Filtrer un flot

```
let rec filter p l = match l with
  | Empty -> Empty
  | Cons (n,f) ->
    let f' () = filter p (f()) in
    if p n then Cons(n, f') else f'()
```

```
val filter : ('a -> bool) -> 'a flot -> 'a flot = <fun>
```

Crible d'Ératosthène

Le principe : on construit un flot de nombres premiers en éliminant tous les multiples de chaque nombre premier

```
let rec crible l = match l with
  | Empty -> Empty
  | Cons (n,f) ->
    let f' () = crible (filter (fun k -> k mod n <> 0) (f()))
    in Cons (n,f')
```

```
val crible : int flot -> int flot = <fun>
```

```
let res = take_n (crible (les_entiers 2)) 40
```

```
val res : int list * int flot =
  ([2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61;
    67; 71; 73; 79; 83; 89; 97; 101; 103; 107; 109; 113; 127; 131; 137;
    139; 149; 151; 157; 163; 167; 173],
   Cons (179, <fun>))
```