FONCTIONS d'ORDRE SUPERIEUR

ENSIIE - 1ère année - IAP1

C. Dubois

Une fonction est une valeur comme les autres :

- elle peut être argument ou résultat d'une autre fonction
- elle peut être la composante d'une autre donnée

  Exemple : liste de fonctions

```
let vals = [succ ; pred; (function x -> x/2)];;
vals : (int -> int) list = [<fun>; <fun>; <fun>]


(dernier vals) 138;;
- : int = 69
```

- elle peut être utilisée comme structure de données : représentation d'une table par une fonction par exemple

On appelle *fonction d'ordre supérieur* ou *fonctionnelle* une fonction qui admet en argument au moins une fonction.

1. Fonction comme argument et/ou résultat d'une autre fonction

Utilisations fréquentes : en calcul numérique, pour paramétrer une fonction de tri ...

Exemple 1 : Calcul de la pente d'une fonction en 0

Soit f une fonction à valeurs réelles dérivable en 0, calculons $f'(0)$, la valeur de la pente de f en 0. On approxime $f'(0)$ par $\frac{f(h)-f(0)}{h}$ avec h petit, égal à 0.001 par exemple.

```
let pente_en_0 f = let h = 0.001 in (f(h)-.f(0.))/.h;;
pente_en_0 : (float -> float) -> float = <fun>
```

Exemple 2 (généralisation) : dérivée de toute fonction

Soit f une fonction dérivable. On approxime la fonction f' dérivée de f par la fonction qui à tout x associe $\frac{f(x+h)-f(x)}{h}$ avec h petit

```
let dérivée f = let h =  0.001 in
                function x -> (f(x+.h)-.f(x))/.h;;
dérivée : (float -> float) -> float -> float = <fun>
          (float -> float) -> (float -> float)

let square' = dérivée (function x -> x*.x);;
 square' : float -> float = <fun>

square' 1.;;
- : float = 2.001

(dérivée cos) 0.0;;
- : float = -0.000499999958326
```

Exemple 3 : composition de fonctions

```
let o (f,g) = function x -> g (f x);;
```

calcule g o f

```
let o_succ f = o(f,succ);;
```

calcule succ o f

```
(o_succ pred) 5;; = o(pred,succ) 5
                  = (function x -> succ (pred x)) 5
                  = succ (pred 5)
```

Exemple 4 : paramétrer un algorithme par un ordre

Retour sur l'insertion d'un élément dans une liste triée dans l'ordre croissant

```
#let rec insérer (e, liste) =
  match liste with
  | [] -> [e]
  | x::l -> if e < x then e::liste else x::(insérer (e, l));;
val insérer : 'a * 'a list -> 'a list = <fun>

# insérer (1, [2;3;4]);;
- : int list = [1; 2; 3; 4]
# insérer ("eet", ["abc"; "def"; "klm"]);;
- : string list = ["abc"; "def"; "eet"; "klm"]
# insérer (true, [false; false]);;
- : bool list = [false; false; true]
# insérer ((3,1), [(1,2);(2,3); (2,4); (3, 2)]);;
- : (int * int) list = [(1, 2); (2, 3); (2, 4); (3, 1); (3, 2)]
```

Et dans une liste triée dans l'ordre décroissant :

```
#let rec insérer_décroissant (e, liste) =
  match liste with
  | [] -> [e]
  | x::l -> if e > x then e::liste
            else x::(insérer_décroissant (e, l));;
val insérer_décroissant : 'a * 'a list -> 'a list = <fun>
```

$\rightarrow$ 2 fonctions d'insertion polymorphes car $<$ est polymorphe.

int, float, string, char : OK pas de problème

bool : false $<$ true

pour les couples : ordre lexicographique

pour le reste ??????

```
#type t = A of int | B of float;;
# (A 1) < (A 1);;
- : bool = false
# (A 1) < (A 3);;
- : bool = true
# (B 3.0) < (B 2.99);;
- : bool = false
# (A 1) < (B 1.3);;
- : bool = true
```

```
#type t = B of float | A of int;;
# (A 1) < (A 1);;
- : bool = false
#  (A 1) < (A 3);;
- : bool = true
# (B 3.0) < (B 2.99);;
- : bool = false
#  (A 1) < (B 1.3);;
- : bool = false
```

PRUDENCE !

Attention : ne pas utiliser les fonctions $<$, $>$ et $\leq$ et $\geq$ avec n'importe quel type.

$\Rightarrow$ Introduire la fonction de comparaison en tant que paramètre de la fonction d'insertion :

```
let rec insérer_gen (priorité, e, liste) =
  match liste with
  | [] -> [e]
  | x::l -> if priorité (x, e) then x::(insérer_gen (priorité, e, l))
            else e::liste;;

let insérer (e, l) = insérer_gen (inf_int, e, l);;

let insérer_ordre_décroissant (e, l) = insérer_gen (sup_int, e, l);;

let insérer_tarot (c, main) = insérer_gen (plus_forte, c, main);;
```

De la même façon on peut paramétrer des tris

2. Fonction currifiée

Comment définir une fonction qui accepte plusieurs données en entrée (*à plusieurs arguments* avec un abus de langage) ?

$\Rightarrow$ en utilisant un n-uplet d'arguments

```
let diff (x,y) = if x < y then x-y else y-x;;
diff : int * int -> int = <fun>
```

fonction d'un seul argument (un couple d'entiers)

$\Rightarrow$ en définissant une fonction currifiée

```
let diff x y = if x > y then x-y else y-x;;
diff : int -> int -> int = <fun>

let diff = function x ->
                (function y -> if x > y then x-y else y-x);;

let diff x = function y -> if x > y then x-y else y-x;;
       int -> (int -> int)
```

fonction d'un seul argument entier (x) qui calcule une fonction des entiers vers les entiers

```
let diff5 = diff 5;;                    diff5 : int -> int = <fun>
diff5 8;;                               - : int = 3
(diff 5) 8;;                            - : int = 3
diff 5 8;;                              - : int = 3
```

A apprendre par coeur :-)

Une fonction f à deux arguments est aussi une fonction à un argument, dont le résultat est une fonction

On peut utiliser f avec un seul argument, le résultat sera une fonction.

• Le type $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4$ est syntaxiquement équivalent à
$t_1 \rightarrow (t_2 \rightarrow (t_3 \rightarrow t_4))$

Exemple : $(int \rightarrow int) \rightarrow int \rightarrow int$ équivalent à
$(int \rightarrow int) \rightarrow (int \rightarrow int)$

Attention : les parenthèses autour du type du 1er argument ne peuvent être enlevées (c'est une fonction)

• $e_1\ e_2\ e_3 \ldots e_n$ syntaxiquement équivalent à $(...(e_1(e_2))e_3) \ldots)e_n$

• Si e a le type $t_1 \rightarrow t_2 \ldots \rightarrow t_n \rightarrow t_{n+1}$
alors $e\ e_1 \ldots e_i$ a le type $t_{i+1}.... \rightarrow t_n \rightarrow t_{n+1}$ si $i \leq n$

On parle d'application *partielle* si $i < n$

Une fonctionnelle qui permet de currifier une fonction :

```
let curry f = function x -> function y -> f(x,y);;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
curry : ('a * 'b -> 'c) -> ('a -> 'b -> 'c) = <fun>
```

ou encore `let curry f x y = f(x,y);;`

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

curry min 2 4;;
- : int = 2

let min_0 = curry min 0 in min_0 (-4);;
- : int = -4
```

ici `min_0 : int -> int`

Retour sur l'insertion dans une liste triée : fonction currifiée

```
let rec insérer_gen priorité e liste =
  match liste with
  | [] -> [e]
  | x::l -> if priorité x e then x::(insérer_gen priorité e l)
            else e::liste;;
val : insérer_gen : ('a -> 'a -> bool) -> 'a -> 'a list = <fun>

let insérer e l = insérer_gen (<) e l;;

let insérer_ordre_décroissant e l = insérer_gen (>) e l;;

let insérer_date d li = insérer_gen chrono d li;;
(* chrono : date -> date -> bool *)
```

`(<) : int -> int -> bool` : fonction correspondant à l'opérateur + C'est la fonction définie par `function x -> function y -> x < y`

3. Quelques fonctionelles classiques sur les listes

$\Rightarrow$ *Appliquer un même traitement sur chaque élément d'une liste*

Soit f une fonction et l une liste $[a_1; a_2 ...; a_n]$
On veut calculer $[(f\ a_1); (f\ a_2) ...; (f\ a_n)]$

```
let rec map f li = match li with
  [] -> []
| x::l -> (f x)::(map f l);;

map : ('a -> 'b) -> 'a list -> 'b list
```

Par exemple pour élever au carré tous les éléments d'une liste `l` :

```
map (function x -> x*x) l
```

La fonctionnelle `map` existe en Ocaml dans le module List de la librairie standard :

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

$\Rightarrow$ *Sélectionner les éléments d'une liste qui vérifient un certain critère*

Soit p un prédicat (une fonction à résultat booléen) et l une liste $[a_1; a_2 ...; a_n]$
On veut calculer $[x | x \in l \wedge p(x) = true]$

```
let rec filter p li = match li with
  [] -> []
| x::l -> if (p x) then x::(filter p l) else (filter p l);;

filter : ('a -> bool) -> 'a list -> 'a list
```

Exemples :

- une fonction qui extrait les entiers pairs d'une liste

```
let termes_pairs l= filter (function x -> (x mod 2 = 0)) l
```

ou encore `let termes_pairs = filter (function x -> (x mod 2 = 0))`

- La liste des bouts d'une main de tarot

```
let bouts = filter
              (function c -> match c with Excuse -> true
                                        | Atout n -> n=1 || n=21
                                        | _ -> false);;
val bouts : carte list -> carte list
```

- La sous-liste des étudiants de 1A nés avant 1988 (promotion1A)

```
let extrait =
   filter (function e -> e.date_de_naissance.année<1988) promotion1A
val extrait : etudiant list = ....
```

La fonctionnelle `filter` existe en Ocaml : `List.filter`

Extension : séparer une liste en 2 sous-listes suivant un critère (fonction `partition` voir TD)

Application : tri `quicksort` dont l'idée est :

- choisir un élément dans la liste à trier : *pivot*
- partitionner la liste par comparaison avec le pivot
- trier les 2 sous-listes obtenues obtenues (appel récursif)
- concaténer les sous-listes obtenues en plaçant le pivot entre les 2

$\Rightarrow$ *Combiner les éléments d'une liste entre eux à l'aide d'une opération binaire*

```
let rec somme l = match l with [] -> 0
                        | a::r -> a + (somme r);;
```

$somme\ [a_1; a_2; ...; a_{n-1}; a_n] = +\ a_1\ (+\ a_2\ (\ldots(+\ a_{n-1}\ (+\ a_n\ 0))\ldots))$

```
let rec produit l = match l with [] -> 1
                           | a::r -> a * (produit r);;
```

$produit\ [a_1; a_2; ...; a_{n-1}; a_n] = *\ a_1\ (*\ a_2\ (\ldots(*\ a_{n-1}\ (*\ a_n\ 1))\ldots))$

```
let rec concat  l = match l with [] -> ""
                           | a::r -> a ^ (concat r);;
```

$concat\ [a_1; a_2; ...; a_{n-1}; a_n] = \hat{}\ a_1\ (\hat{}\ \ a_2\ (\ldots(\hat{}\ a_{n-1}\ (\hat{}\ \ a_n\ \texttt{""}))\ldots))$

Généralisation : soit f une fonction à 2 arguments, e une valeur quelconque et l une liste $[a_1; a_2 ...; a_{n-1}; a_n]$
On veut calculer $f\ a_1\ (f\ a_2\ (\ldots (f\ a_{n-1}\ (f\ a_n\ e))\ldots))$

```
let rec fold_right f l e = match l with
  [] -> e
| a::r -> f a (fold_right f r e);;

fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Exemple : calculer la somme des éléments d'une liste

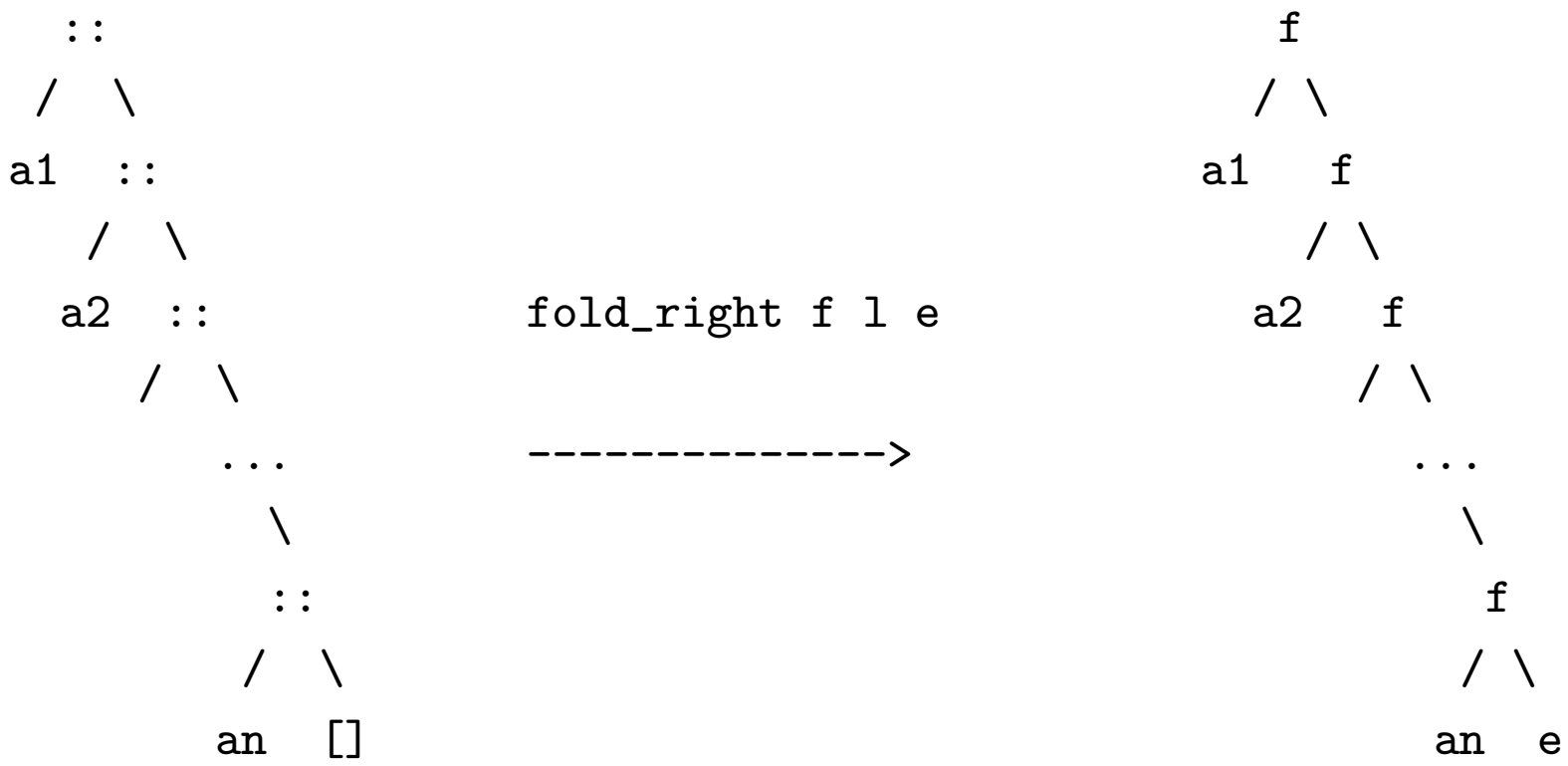
```
let somme l = fold_right (+) l 0
```

de type `int list -> int`

Graphiquement :

On part de $a_1{::}(a_2{::}(\ldots(a_{n-1}{::}(a_n{::}[]))\ldots))$

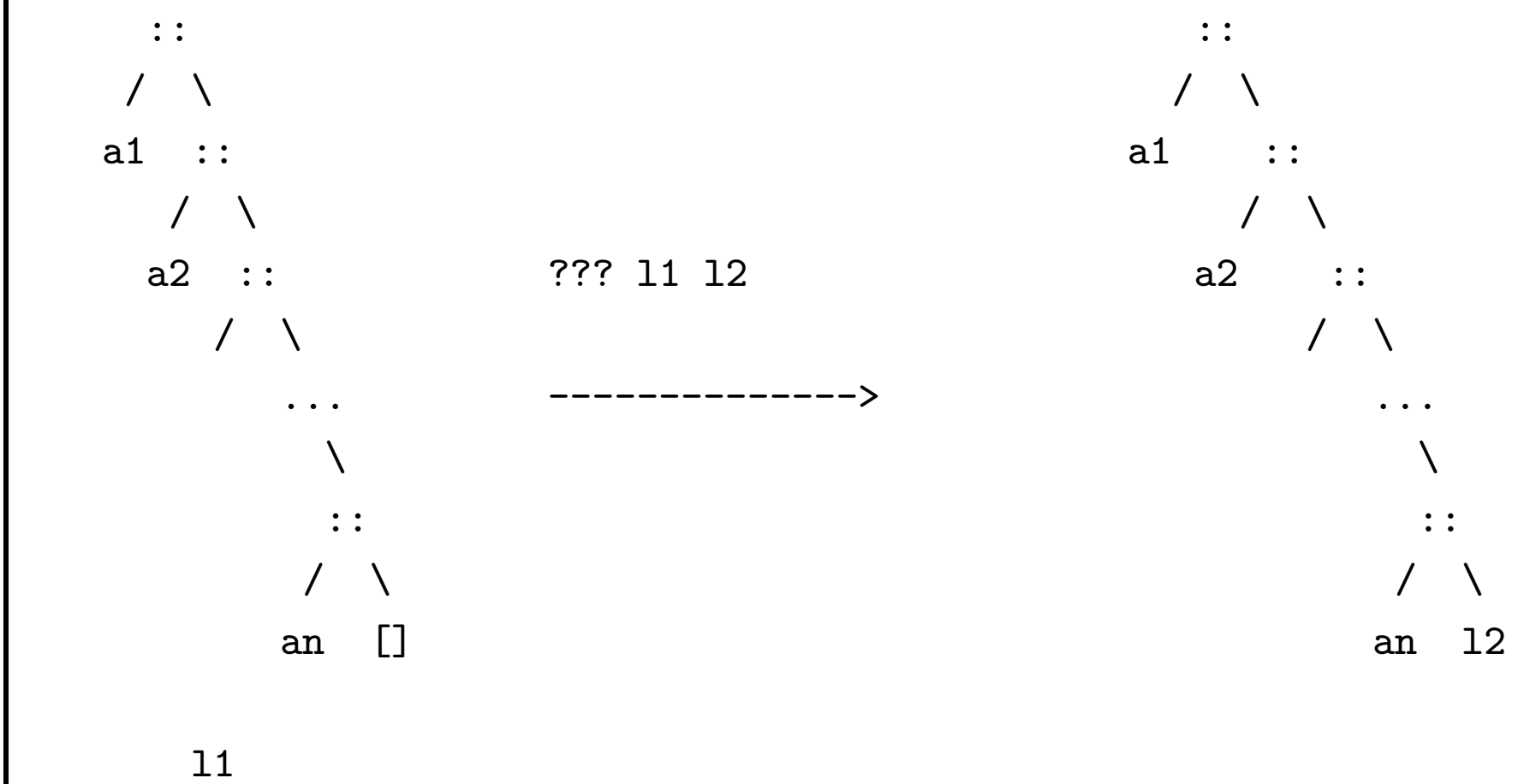
On veut calculer $f\ a_1\ (f\ a_2\ (\ldots(f\ a_{n-1}\ (f\ a_n\ e))\ldots))$



f prend son premier argument dans la liste

Devinette :

```
let ???  l1 l2 = fold_right (function x -> function y -> x::y) l1
l2
```



Et si on inversait les calculs ...

On veut calculer $f\ (\ldots(f\ (f\ e\ a_1)\ a_2)\ldots a_{n-1})a_n)$

f prend son deuxième argument dans la liste, son premier argument est le calcul déjà effectué

```
#let rec fold_left f e l = match l with
      [] -> e
    | a::r -> fold_left f (f e a) r;;

fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

#let et l = fold_left (&&) true l ;;
val et : bool list -> bool = <fun>

ou let et = fold_left (&&) true ;;

#et [true; true;false];;
- : bool = false
```

```
fold_left f e l = fold_right (function x -> function y -> (f y x)) l e

fold_right f l e = fold_left (function x -> function y -> (f y x)) e l
```

```
fold_left f e l = fold_right f l e
```

si `f` *est associative et* `e` *élément neutre pour* `f`

démonstration par induction structurelle sur l de la propriété

$\forall\ l\ \forall\ x.\ f\ x\ (fold\_left\ f\ e\ l)\ =\ fold\_left\ f\ x\ l$

$\Rightarrow$ Selon l'opération f que l'on utilise, choisir `fold_right` ou `fold_left`.

Pour retenir : `fold_left` : l'élément de base est à gauche de la liste

`fold_right` : l'élément de base est à droite de la liste

Les fonctions existent dans le module `List` de la bibliothèque standard.

Reprenons l'exemple du système arborescent de fichiers

Un fichier est :

- soit un fichier texte
- soit un répertoire contenant des fichiers

```
type fichier = Texte of string
             | Répertoire of (fichier list);;

let rec nb_texte f = match f with
  Texte _ -> 1
| Répertoire l ->
   fold_right (function fi -> function y -> (nb_texte fi) + y) l 0;;
nb_texte : fichier -> int = <fun>
```

4. Quelques fonctionelles pour itérer

itérer = répéter un certain nombre de fois l'application d'une fonction

⇒ *Itération bornée*

Appliquer n fois la fonction f à la valeur a ou calculer le n-ième terme de la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_0 = a,\ u_{n+1} = f(u_n)$

```
let rec iter n f a =
  if n = 0 then a else f (iter (n-1) f a);;
val iter : int -> ('a -> 'a) -> 'a -> 'a = <fun>

 Appliquer n fois la fonction f à la valeur a c'est aussi
  appliquer n-1 fois la fonction f à la valeur f(a)

let rec iter n f a =                        récursivité terminale
  if n = 0 then a else  iter (n-1) f (f a);;

let puissance p n = iter n (function x -> p*x) 1;;
```

```
let fib n = fst (iter n (function (x,y) -> (x+y,x)) (1,0));;

fib 5 = fst (iter 5 (function (x,y) -> (x+y,x)) (1,0))
      = fst (iter 4 (function (x,y) -> (x+y,x)) (1,1))
      = fst (iter 3 (function (x,y) -> (x+y,x)) (2,1))
      = fst (iter 2 (function (x,y) -> (x+y,x)) (3,2))
      = fst (iter 1 (function (x,y) -> (x+y,x)) (5,3))
      = fst (iter 0 (function (x,y) -> (x+y,x)) (8,5))
      = fst (8,5) = 8
```

Même algorithme que la version récursive terminale vue en TD :

```
let rec fibacc (n,c,p)= if n=1 then c else fibacc(n-1,p+c,c);;
let fib n = if n=0 then 1 else fibacc(n,1,1);;

fib 5 = fibacc(5,1,1)
      = fibacc(4,2,1)
      = fibacc(3,3,2)
      = fibacc(2,5,3)
      = fibacc(1,8,5) = 8
```

$\Rightarrow$ *Itération non bornée*

Le nombre d'applications de f dépend d'une condition (prédicat)
Le dernier terme calculé de la suite $(u_n)_{n\in\mathbb{N}}$ est le premier terme à vérifier la condition

```
let rec loop p f a =
  if p a then a else loop p f (f a);;
val loop : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a = <fun>
```

`loop` est plus générale que `iter` :

```
let iter n f a = snd (loop (function (p,x) -> p=n)
                           (function (p,x) -> (p+1, f x))
                           (0,a));;
```

Exemple 1 : la plus petite puissance de p supérieure à x

```
# let ppp p x =
  loop (function y -> y > x) (function y -> p*y) 1;;
val ppp : int -> int -> int = <fun>
# ppp 2 10;;
- : int = 16
# ppp 5 100;;
- : int = 125
```

l'exposant maintenant nous intéresse

```
# let ppe p x =
snd (loop (function (y,z) -> y>x) (function (y,z) -> (y*p,z+1)) (1,0));;
val ppe : int -> int -> int = <fun>
# ppe 2 10;;
- : int = 4
# ppe 5 100;;
- : int = 3
```

Exemple 2 : calcul du zéro d'une fonction f sur $[a, b]$ à ϵ près.

(hypothèses : $f(a)$ et $f(b)$ sont de signes différents et f est monotone)

Méthode dichotomique : on divise l'intervalle en 2, on itère le processus sur le demi-intervalle qui contient le zéro (bornes de signes opposés)

2 implantations différentes :

- définition récursive

```
let rec zero f a b eps =
if  abs_float(b -. a) < eps then (a,b)
else let m = (a +. b) /. 2. in
     if f(a) *. f(m) < 0. then zero f a m eps else zero f m b eps;;

zero (function x -> x*.x -. 2.) 1.0 2.0 (1e-10);;
 : float * float = (1.4142135623260401 2, 1.4142135623842478)
```

un encadrement de $\sqrt{2}$

- avec `loop`

```
let zero f a b eps =
  let arret (a',b') = abs_float(b' -. a') < eps and
      trait (a',b') = (let m = (a' +. b') /. 2. in
                          if f(a') *. f(m) < 0. then (a',m)
                                                else (m ,b'))
  in loop arret trait (a,b);;
```