

# LIFPF – Programmation fonctionnelle

CM3

Structures inductives

Types paramétrés

Licence informatique UCBL – Printemps 2025

<https://forge.univ-lyon1.fr/programmation-fonctionnelle/lifpf/-/blob/main/README.md>

# Ensembles inductifs

## Definition (Définition d'un ensemble inductif)

### Règles de construction

- Prennent 0, 1 ou plusieurs éléments
- fabriquent un élément

L'ensemble défini inductivement par ces règles est le plus petit ensemble stable par ces règles,

c.-à-d. le plus petit ensemble tel que ces règles ne font que fabriquer des éléments déjà dans cet ensemble.

## Exemple d'ensemble inductif : nombres impairs

- Règles :
  - 1 est un nombre impair (règle qui prend 0 élément)
  - si  $n$  est impair alors  $n + 2$  est impair (règle qui prend 1 élément :  $n$ )
- On peut constater que l'ensemble des nombres impairs est bien stable par ces règles.
- On peut aussi remarquer que l'ensemble des entiers naturels  $\mathcal{N}$  est lui aussi stable par ces règles
  - mais c'est n'est pas le plus petit

# Décomposition

Dans un ensemble inductif, tout est élément est obtenu par application d'une règle

- attention cette condition ne garantit pas qu'on a le plus petit ensemble
- par contre, on peut raisonner/calculer en s'appuyant sur la règle utilisée pour construire l'élément.

Exemple : si  $n$  est impair, alors

- soit  $n = 1$  (première règle)
- soit il existe  $n'$  tel que  $n = n' + 2$  (deuxième règle)

## Types inductifs en OCaml

- Type faisant référence à lui-même dans sa définition.
- Souvent utilisé avec un type somme
  - Chaque **constructeur** correspond à une **règle de construction** de l'ensemble inductif.
- La décomposition se fait grâce au *pattern matching*.

### Exemple : listes d'int

```
type liste_ints =  
    L Vide (* règle a 0 valeur *)  
  | L Cons of int * liste_ints  
          (* règle a 1 valeur: la liste_int *)  
;;
```

## Récursivité + *pattern matching* : des outils naturels pour les types inductifs

Pour une valeur  $v$  d'un type inductif :

- Le *pattern matching* : permet de savoir comment (quelle règle / constructeur) et à partir de quelles valeurs  $v$  a été fabriquée
- La récursivité : permet de supposer qu'on sait faire le calcul pour les valeurs ayant servi à fabriquer  $v$ .

## Une fonction simple : longueur

```
let rec longueur (l: liste_ints): int =  
  match l with  
  | LVide (* premiere regle, pas de valeur *)  
    -> 0  
  | LCons (_,l')  
    (* deuxieme regle, 1 valeur l' *)  
    -> longueur l'  
    (* on sait faire sur l' *)  
    + 1  
  
;;
```

## Une fonction simple : somme

```
let rec somme (l: liste_ints): int =  
  match l with  
  | LVide (* premiere regle, pas de valeur *)  
    -> 0  
  | LCons (n,l')  
    (* deuxieme regle, 1 valeur l' *)  
    -> somme l'  
      (* on sait faire sur l' *)  
      + n  
  
;;
```



## Une fonction simple : contient

```
let rec contient (k: int) (l: liste_ints): bool =  
  match l with  
  | LVide (* premiere regle, pas de valeur *)  
    -> false  
  | LCons (n,l')  
    (* deuxieme regle, 1 valeur l' *)  
    -> n = k  
      || contient k l'  
      (* on sait faire sur l' *)  
;;
```

## Arbres binaires d'int

```
type arbre_bin =  
  AVide (* regle a 0 valeur *)  
  | ANoeud of int * arbre_bin * arbre_bin  
    (* regle a 2 valeurs (+ 1 int) *)  
;;
```

# Taille

```
let rec taille (a: arbre_bin): int =  
  match a with  
  | AVide -> 0  
  | ANoeud (_, fg, fd) ->  
    1  
    + taille fg  
      (* on sait faire sur fg *)  
    + taille fd  
      (* on sait faire sur fd *)  
;;
```

# Hauteur

```
let rec hauteur (a: arbre_bin): int =  
  match a with  
  | AVide -> 0  
  | ANoeud (_,fg,fd) ->  
    1  
    + max  
      (hauteur fg)  
      (* on sait faire sur fg *)  
      (hauteur fd)  
      (* on sait faire sur fd *)  
;;
```

## Exemple : expressions

```
type expression =  
  Const of int  
  | Plus of expression * expression  
  | Moins of expression * expression  
  | Mult of expression * expression  
;;  
  
let rec eval (e: expression): int =  
  match e with  
  | Const n -> n  
  | Plus (e1,e2) -> eval e1 + eval e2  
  | Moins (e1,e2) -> eval e1 - eval e2  
  | Mult (e1,e2) -> eval e1 * eval e2  
;;
```

## Récursion mutuelle

Des fonctions mutuellement récursives sont des fonctions qui s'appellent l'une et l'autre.

### Exemple pair et impair

```
let rec pair n =  
  if n = 0  
  then true  
  else not (impair (n-1))  
and impair n =  
  if n = 0  
  then false  
  else not (pair (n-1));;
```

Remarque : pour vraiment coder pair mieux vaut faire :

```
let pair n = (n mod 2 = 0);;
```

## Références mutuelles de types

Des types peuvent également se référencer mutuellement.

Exemple types mutuellement référencés : arbres n-aires et listes d'arbres

```
type arbre =  
  ANVide  
  | ANNoeud of int * liste_arbres  
and liste_arbres =  
  LAVide  
  | LACons of arbre * liste_arbres  
;;
```

# Taille

```
let rec taille_a (a: arbre): int =  
  match a with  
  | ANVide -> 0  
  | ANNoeud (_,la) ->  
    1  
    + taille_l la      (* appel sur liste *)  
  
and taille_l (la: liste_arbres): int =  
  match la with  
  | LAVide -> 0  
  | LACons (a,la') ->  
    taille_a a          (* appel sur arbre *)  
    + taille_l la'      (* appel sur liste *)  
  
;;
```



# Types paramétrés

- Constat : structure de certains types indépendante de ce qu'on y met
  - Exemple : structure d'une liste indépendante du type des éléments
- Objectif : avoir une définition pour la structure commune tous les types d'éléments
  - Une seule définition du type liste valable pour tous les types d'éléments
  - Tout en gardant l'homogénéité des éléments (ex. que des `int` ou que des `string`)
- Moyen : ajouter un (ou plusieurs) paramètre au type
  - le(s) paramètre(s) correspond(ent) au(x) type(s) des éléments
  - fixer le paramètre revient à fixer le type des éléments

## En OCaml : paramètre 'a

Syntaxe : ajouter le paramètre (avec une apostrophe ') avant le type : 'a mon\_type

### Exemple : liste générique

```
type 'a liste_g =  
    LGVide  
  | LGCons of 'a * 'a liste_g  
;;
```

Si plusieurs paramètres : ('a,'b) mon\_type

## Fonctions génériques simples

Les types dans les fonctions peuvent aussi être des paramètres

### Exemple

```
let rec longueur l =  
  match l with  
  | LGVide -> 0  
  | LGCons (_,l') -> 1 + longueur l'  
;;  
val longueur : 'a liste_g -> int = <fun>
```

# Comparateurs

Le type des comparateurs est paramétré

Exemples : < et =

```
(<) ; ;
```

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

```
(=) ; ;
```

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

## Fonction générique simple : contient

```
let rec contient_g e l =  
  match l with  
  | LGVide -> false  
  | LGCons (e', l') ->  
    e = e'  
    || contient_g e l'  
;;  
val contient_g : 'a -> 'a liste_g -> bool = <fun>
```

## Structure générique mais fonction sur des éléments particuliers

```
let rec somme l =  
  match l with  
  | LGVide -> 0  
  | LGCons (n,l') -> n + somme l'  
;;  
val somme : int liste_g -> int = <fun>
```

## Arbres binaires génériques

```
type 'a arbre_bin_g =  
    ABGVide  
  | ABGNoeud of 'a * 'a arbre_bin_g * 'a arbre_  
;;
```

```
let rec taille_abg a =  
  match a with  
  | ABGVide -> 0  
  | ABGNoeud (_,fg,fd) ->  
      1 + taille_abg fg + taille_abg fd  
;;  
val taille_abg : 'a arbre_bin_g -> int = <fun>
```

## Quelques type génériques prédéfinis

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

```
type 'a option = None | Some of 'a
```

```
type ('a, 'b) result = Ok of 'a | Error of 'b
```



## Contenu d'arbres binaires génériques

```
let contenu =  
  let rec contenu_aux a acc =  
    match a with  
    | ABGVide -> acc  
    | ABGNoeud (x, fg, fd) ->  
      x :: contenu_aux fg (contenu_aux fd a)  
  in  
  fun a -> contenu_aux a []  
;;  
val contenu : 'a arbre_bin_g -> 'a list = <fun>
```

Remarque : ce code n'utilise pas @ (concaténation)