# TD - Programmation Fonctionnelle 2

**Louis Thevenet**

## 1. TD5

### 1.1. Exercice 1

```
module type FL2C = sig
  type zero
  type _ succ
  type 'a fichier

  val open_ : string -> zero fichier
  val read : 'n fichier -> char * 'n succ fichier
  val close : zero succ succ fichier -> unit
end
```

```
module type FLPair = sig
type even
type odd
  type  fichier

  val open_ : string -> (even, odd) fichier
  val read : ('a*'b) fichier -> char * ('b*'a) succ fichier
  val close : (even*odd) fichier -> unit
end
```

### 1.2. Exercice 2

```
type 'a perfect_tree = Empty | Node of 'a * ('a * 'a) perfect_tree

let rec split : 'a. ('a * 'a) perfect_tree -> 'a perfect_tree * 'a perfect_tree
    =
 fun tree ->
  match tree with
  | Empty -> (Empty, Empty)
  | Node ((l1, l2), subtree) ->
      let t1, t2 = split subtree in
      (Node (l1, t1), Node (l2, t2))
```

## 2. TD6

$$\text{fold\_right} : (\alpha \to \beta \to \beta) \to \alpha \text{ list} \to \beta \to \beta \equiv ((\alpha \times \beta) \to \beta) \to (\text{unit} \to \beta) \to \alpha \text{ list} \to \beta$$
$$\equiv ((\alpha \times \beta) \text{ option} \to \beta) \to \alpha \text{ list} \to \beta$$

$$\text{unfold} : \left( \underbrace{\beta}_{\text{type générateur}} \to \left( \alpha \times \underbrace{\beta}_{\text{pour la prochaine génération}} \right) \text{ option} \right) \to (\beta \to \alpha \to \alpha \text{ flux})$$

```
module type Iter =
sig
type 'a t
val vide : 'a t
val cons : 'a -> 'a t -> 'a t
val uncons : 'a t -> ('a * 'a t) option
val apply : ('a -> 'b) t -> ('a t -> 'b t)
val unfold : ('b -> ('a * 'b) option) -> ('b -> 'a t)
val filter : ('a -> bool) -> 'a t -> 'a t
val append : 'a t -> 'a t -> 'a t
end
```

```
let flux_nul = Flux.unfold (fun ()->Some(0, ())) ()
(* le flux qui contient tous les entiers relatifs pairs, par ordre croissant en
valeur absolue *)
let flux_pair = Iter.unfold (fun i -> Some(2*i, if i <=0 then 1-i else -i))
```

### 2.1. Exercice 1

```
let constant e = Iter.unfold (fun () -> Some(e, ())) ()
let map f fl = Flux.(apply (constant f) fl)
let map2 f fl fl' = Flux.(apply (map f fl) fl')
```