

TD - Programmation Fonctionnelle 2

Louis Thevenet

1. TD5

1.1. Exercice 1

```
1 module type FL2C = sig
2   type zero
3   type _ succ
4   type 'a fichier
5
6   val open_ : string -> zero fichier
7   val read : 'n fichier -> char * 'n succ fichier
8   val close : zero succ succ fichier -> unit
9 end
```

```
1 module type FLPair = sig
2   type even
3   type odd
4   type fichier
5
6   val open_ : string -> (even, odd) fichier
7   val read : ('a*'b) fichier -> char * ('b*'a) succ fichier
8   val close : (even*odd) fichier -> unit
9 end
```

1.2. Exercice 2

```
1 type 'a perfect_tree = Empty | Node of 'a * ('a * 'a) perfect_tree
2
3 let rec split : 'a. ('a * 'a) perfect_tree -> 'a perfect_tree * 'a perfect_tree
4   =
5   fun tree ->
6     match tree with
7     | Empty -> (Empty, Empty)
8     | Node ((l1, l2), subtree) ->
9       let t1, t2 = split subtree in
10      (Node (l1, t1), Node (l2, t2))
```

2. TD6

$\text{fold_right} : (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \rightarrow \beta \equiv ((\alpha \times \beta) \rightarrow \beta) \rightarrow (\text{unit} \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta$
 $\equiv ((\alpha \times \beta) \text{ option} \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta$

$\text{unfold} : \left(\underbrace{\beta}_{\text{type g  n  rateur}} \rightarrow \left(\alpha \times \underbrace{\beta}_{\text{pour la prochaine g  n  ration}} \right) \text{ option} \right) \rightarrow (\beta \rightarrow \alpha \rightarrow \alpha \text{ flux})$

```
1 module type Iter =
2   sig
3     type 'a t
4     val vide : 'a t
5     val cons : 'a -> 'a t -> 'a t
6     val uncons : 'a t -> ('a * 'a t) option
7     val apply : ('a -> 'b) t -> ('a t -> 'b t)
8     val unfold : ('b -> ('a * 'b) option) -> ('b -> 'a t)
9     val filter : ('a -> bool) -> 'a t -> 'a t
10    val append : 'a t -> 'a t -> 'a t
11  end
```

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

2.1. Exercice 1

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

3. TD8

Parser Entr  e \rightarrow Ensemble des solutions possibles

```
1 type 'a parser: 'a Flux.t -> 'a result
2 type 'a result = 'a Flux.t Solution.t (* Solution.t: ensemble mais on va
3                                         l'impl  menter avec des flux pour
4                                         avoir une   valuation paresseuse
5                                         *)
6
7 (* Pour Flux.t on utilisera que uncons donc c'est facile, par contre pour
8    Solution.t on aura besoin de bind par exemple *)
```

3.1. Exercice 1

```
1 let psequence p1 p2 flux = (p1 flux) >=> p2
2 let pchoix p1 p2 flux = Solution.((p1 flux) ++ (p2 flux))
```

3.2. Exercice 2

```
1 let rec eval: 'a language -> 'a Flux.t -> 'a result = fun l flux -> match l with
2   Nothing -> perreur flux
3   | Empty -> pnul flux
4   | Letter(c) -> ptest ((=) a) flux
5   | Sequence(l,l') -> psequence (eval l) (eval l') flux
6   | Choice(l,l') -> pchoix (eval l) (eval l') flux
7   | Repeat(l) -> eval (Choice(Empty, Sequence(l, Repeat(l)))) flux
8
9 let rec belongs : 'a language -> 'a Flux.t -> bool = fun l flux ->
10   Solution.uncons
11     (Solution.filter (fun s -> Flux.uncons f = None))
12     (eval l flux))
13   <> None
```

3.3. Exercice 3

```
1 let perreur= Solution.zero
2 let pnul = return ()
3 let ptest p f = match Flux.uncons f with
4   | None -> Solution.zero
5   | Some(t,q) -> if p t then
6     Solution.return (t,q)
7     else
8     Solution.zero
9
10 let pchoice = (++)
11 let (*>) p1 p2 =
12   p1 >=> fun b ->
13     p2 >=> fun c -> return (b,c)
14
15 type ast = Div of ast | Var of char
16 let rec expr flux = var >=> fun v -> return (Var v)
17 ++
18   paro *> expr *> div
19   *> expr *> parf
20 >=> fun ((((_ , e1), _), e2), _) -> return (Div(e1,e2)) flux
21 )
```