TD - Programmation Fonctionnelle 2

Louis Thevenet

1. TD5

1.1. Exercice 1

```ocaml
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
```

```ocaml
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

```ocaml
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

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

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

```ocaml
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
```

```ocaml
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

```ocaml
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')
```

3. TD8

Parser Entrée $\to$ Ensemble des solutions possibles

```ocaml
type 'a parser: 'a Flux.t -> 'a result
type 'a result = 'a Flux.t Solution.t (* Solution.t: ensemble mais on va
                                        l'implémenter avec des flux pour
                                        avoir une évaluation paresseuse
                                      *)

(* Pour Flux.t on utilisera que uncons donc c'est facile, par contre pour
Solution.t on aura besoin de bind par exemple *)
```

3.1. Exercice 1

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

3.2. Exercice 2

```ocaml
let rec eval: 'a language -> 'a Flux.t -> 'a result = fun l flux -> match l with
Nothing -> perreur flux
| Empty -> pnul flux
| Letter(c) -> ptest ((=) a) flux
| Sequence(l,l') -> psequence (eval l) (eval l') flux
| Choice(l,l') -> pchoix (eval l) (eval l') flux
| Repeat(l) -> eval (Choice(Empty, Sequence(l, Repeat(l)))) flux

let rec belongs : 'a language -> 'a Flux.t -> bool = fun l flux ->
  Solution.uncons
    (Solution.filter (fun s -> Flux.uncons f = None))
    (eval l flux))
  <> None
```

3.3. Exercice 3

```ocaml
let perreur= Solution.zero
let pnul = return ()
let ptest p f = match Flux.uncons f with
| None -> Solution.zero
| Some(t,q) -> if p t then
                 Solution.return (t,q)
               else
                 Solution.zero

let pchoice = (++)
let (*>) p1 p2 =
  p1 >>= fun b ->
  p2 >>= fun c -> return (b,c)

type ast = Div of ast | Var of char
let rec expr flux = var >>= fun v -> return (Var v)
++
 paro *> expr *> div
      *> expr *> parf
>>= fun (((((_, e1), _), e2), _)) -> return (Div(e1,e2)) flux
)
```

4. TD9

4.1. Exercice 1

```ocaml
let rec prod_int_list l =
match l with
| [] -> 1
| t::q -> t * prod_int_list q


let prod l =
let p = Delimcc.new_prompt () in

let rec loop l = match l with
| [] -> 1
| 0::_ -> Delimcc.shift p (fun k -> 0)
| hd::tl -> hd * (loop l)

in
push_prompt p (fun () -> loop l)
```

4.2. Exercice 2

```ocaml
type res =
| Done of string
| Request of (string -> res)

let p = new_prompt ()

let cas_nominal nom =
  let f = open_in
    (if sys.file_exists nom then nom
    else shift p (fun k -> Request k))
  in
  let l = read_line f in
  close_in f;
  Done l

let redemande nom k =
  Format.printf "%s n'existe pas, entrez un nouveau nom" nom;
  let new = read_line () in
  k new

let handler nom =  match push_prompt p (fun () -> cas_nominal nom) with
  | Done l->l
  | Request k ->
    begin
      match redemande nom k with
        | Done l->l
        | Request _ -> assert false
    end
```

4.3. Exercice 3

```ocaml
type res =
| Yield of (-> res)
| Done

let ping () =
  begin
    for i = 1 to 10
      do
        print_endline "ping !";
        shift p (fun k  -> Yield k)
      done;
      Done
    end
let pong () =
  begin
    for i = 1 to 10
    do
      print_endline "pong !";
      shift p (fun k  -> Yield k)
    done
    Done
  end

let scheduler () =
  let p = new_prompt () in
  let rec loop ps =
    match ps with
    | [] -> ()
    | hd :: ps' ->
      match push_promp p (fun () -> hd ()) with
      | Done -> loop ps'
      | Yield kp -> loop ps'@[kp]
      in loop [ping; pong]
```

4.4. Exercice 3

```ocaml
type res =
| Done
| Yield of int*(()->res)

let p = new_prompt ()

let yield i = shift p (fun k -> Yield (i,k))
let foreach f iter t =
  let rec loop = function
  | Done -> ()
  | Yield (i,k) -> f i; k ()
in loop (push_prompt p (fun () -> iter t; Done))
```