Sommaire PF

TD1 : Liste

1. Structure de données : liste

Exercice 1

```
(* ---Contrat---
deuxieme : renvoie le 2ème elt d'une liste
pre : taille(list) >= 2
paramètres :
list : 'a list - (liste pour laquelle on souhaite récupérer le 2ème elt)

---Tests---
test : l1 = []
l2 = [1]
l3 = [1;2]
l4 = [1;2;3;4;5]
l5 = ['a', 'b', 'c']

deuxieme l1 -> "la liste est vide!"
deuxieme l2 -> "la liste contient un seul élement!"
deuxieme l3 -> 2
deuxieme l4 -> 2
deuxieme l5 -> 'b'
*)

let deuxieme list = match list with
| [] -> failwith "la liste est vide!"
| [_] -> failwith "la liste contient un seul élement!"
| _::s::t -> s
```

```
(* ---Contrat---
n_a_zero : construit la liste des n à zero entier
pre : n >= 0
paramètres :
n : int

---Tests---
test :
n_a_zero 0 -> [0]
n_a_zero 1 -> [1;0]
n_a_zero 4 -> [4;3;2;1;0]
*)

let rec n_a_zero n = match n with
| 0 -> [0]
| _ -> n::(n_a_zero (n-1))

let zero_a_n n =
let rec aux p list
if p < 0 then list
else aux (p-1) p::list
in aux n []
```

```
(* ---Contrat---
indice : renvoie la liste des indices d'un elt e dans une liste l
pre : e du même type que les elt de la liste
paramètres :
e : 'a (elt à chercher)
l : 'a list (liste dans laquelle on effectue la recherche)
résultat : une liste d'entiers (int list) correspondant aux indices de l'élément e

---Tests---
test :
indice 5 [] -> []
indice 5 [5;1;2] -> [0]
indice 5 [1;2;3;4] -> []
indice 5 [5;2;5;3] -> [0;2]
indice 'a' ['b';'z';'a';'a'] -> [2;3]
*)

let indice e l =
        let rec aux e l i = match l with
        | [] -> []
        | h::t -> if h = e then i::(aux e l i) else aux e t (i+1)
        in aux e l 0
```

Exercice 2

```
(*1*)
let map f l = List.fold_right (fun h map_t -> (f h)::map_t) l []

(*2*)
let flatten l = List.fold_right (fun sl fl -> sl@fl) l []

(*3*)
let fsts = List.map fst

(*4*)
let split l = List.fold_right (fun (a,b) (l1,l2) -> (a::l1,b::l2)) l []

(*5*)
let remove l = List.fold_right (fun e sort_t -> if (List.mem e sort_t)
then sort_t else e::sort_t) l []
```

---

2. Modules, application aux files

Exercice 3

Exercice 4

TD2 : Algorithmes combinatoires

1. Parties d'un ensemble

Exercice 1

E = {}
P_E = {{}} -> 2^0

E = {...}
P_E = {E_1,...,E_n} -> 2^n

E + {e} = E'
$P_{E'}$ = {E_1, {e} U E_1,..., E_n, {e} U E_n} -> 2^{n+1}

Exercice 2

```
let ajout e l = List.fold_right(fun h ft = h::((e::h)::ft)) l []

let rec parties l = match l with
| [] -> [[]]
| h::t -> ajout h (parties t)

let parties l = List.fold_right ajout l [[]]
```

---

2. Permutation d'une liste

Exercice 4

```
let rec insertions e l = match l with
| [] -> [[e]]
| h::t -> (e::l)::List.map (fun x -> t::x) (insertions e t)

let permutations l = List.fold_right
        (fun h perm_t -> List.flatten(List.map (insertions h perm_t))) l [[]]
```

---

3. Combinaisons

Exercice 5

C(n,0) = 1
C(0, k+1) = 0
C(n+1, k+1) = C(n, k+1) + C(n, k)

TD3 : Arbres lexicographiques

1. Structure arborescente n-aire

Exercice 1 (Définition des types)

Node {- Liste de branche | - Bool}
Edge {- symbole 'a | - 'a Node}

```
type 'a arbre = Node of bool * ('a Edge list)
        and 'a Edge = 'a * 'a arbre
```

Exercice 2 (Test d'appartenance)

```
val appartient = 'a list -> 'a arbre -> bool

let rec appartient l (Node(value, edges)) = match l with
| [] -> value
| h::t -> match edges with
        | [] -> false
        | (ce,ae)::te ->
                if h = ce then appartient ae
                else if h < ce then false
                else appartient l (Node(value, te))

(* Mellieur version *)
let rec recherche ch lc =
        match lc with
                | [] -> None
                | (ce,ae)::te ->
                        if ch = ce then Some ae
                        else if ch < ce then None
                        else recherche ch te

let rec appartient l (Node(value, edges)) = match l with
| [] -> value
| c::t -> match (recherche c edges) with
        | None -> false
        | Some a -> appartient t a
```

Exercice 3 (Ajout)

```
val ajout = 'a list -> 'a arbre -> 'a arbre

let rec ajout l (Node(true, edges)) = match l with
| [] -> Node(true, edges)
| h::t -> match (recherche h edges) with
        | None -> ajout (Node(value, insert h edges (Node(false,[])))) 
        | Some a -> ajout t a

val insert 'a -> [('a * 'a arbre)] -> 'a arbre -> [('a * 'a arbre)]

let rec insert ch edges new_e = match edges with
| [] -> [(ch,new_e)]
| (ce,ae)::t -> if ch > ce then (ce,ae)::(insert ch t new_e)
        else (ch,new_e)::edges
```

---

2. Arbres lexicographiques

voir TD

TD4 : Les modules

2. Utilisation des modules

Exercice 1

```
module type Collection =
sig
        type 'a t
        execption CollectionVide
        val empty: 'a t
        val isEmpty: 'a t -> bool
        val add: 'a -> 'a t -> 'a t
        val pop: 'a t -> ('a * 'a t)
end

module Pile: Collection =
struct
        type 'a t = 'a list
        execption CollectionVide
        let empty = []
        let isEmpty p = (p = empty)
        let add e p = e::p          (*f@[e] pour les files*)
        let pop e p = match p with
        | empty -> raise CollectionVide
        | h::t -> (h, t)
end
```

Exercice 2

```
module type Fold =
sig
        type a
        type b
        val cas_de_base: b
        val traitre_et_combine: a -> b -> b
end

module CreerListe: Fold with type a = int and type b = int list =
struct
        type a = int
        type b = int list
        let cas_de_base = []
        let traite_et_combine elt lt = elt::lt
end

module TrouvePair: Fold with type a = int and type b = int Option =
struct
        type a = int
        type b = int Option
        let cas_de_base = None
        let traite_et_combine elt t  = if ((elt mod 2) = 0)
                then Some elt
                else t
end
```

---

3. Les foncteurs

Exercice 3

```
module FoldList (F: Fold) =
struct
        let rec fold_right l = match l with
        | [] -> F.cas_de_base
        | h::t -> F.traite_et_combine h (foldright t)
end

module FoldCollection (C: Collection) (F: Fold)
struct
        let rec fold col =
                if C.isEmpty col then F.cas_de_base
                else let (h, t) = pop col in
                        F.traite_et_combine h (fold t)
end

module FoldPile = FoldCollection (Pile)
module CreerListePile = FoldPile (CreerListe)

let%test _ = CreerListePile(Pile.(add 1)(add 2 (add 3))) = [1;2;3]
```

TD5 : Typage Avancé

1. Types Fantôme

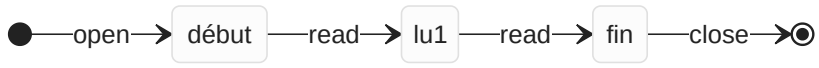
ex :

```
type 'a list = [] of 'a * 'a list

type 'a truc = t of int
```

```
let F = open_ "abc" in
let (c, F') = read F in
(* let (c', F') = read F' in
=> renvoie une erreur car F' est de type fin fichier*)
let close F
```

`in_channel` mets en lecture les fichiers dans la librairie std
`open_in` ouvre un fichier dans la librairie std

Exercice 1




```
type debut
type lu1
type fin
val open_ : string -> debut Fichier
val read1 : debut Fichier -> char * lu1 Fichier
val read2 : lu1 Fichier -> char * fin Fichier
val close : fin Fichier -> unit


let read1 = (input.char F, F)
let read2 = ...
```

Arithmétique de Peano :

$$0 \in \mathbb{N}$$
$$\forall n,\ s(n) \in \mathbb{N}$$

```
type zero
type _ succ
val open_ : string -> zero Fichier
val read : 'n Fichier -> char * 'n succ Fichier
val close : zero succ succ Fichier -> unit
```

```
type even
type odd
type _Fichier
val open_ : string -> (even * odd) Fichier
val read : ('a, 'b) Fichier -> char * ('b * 'a) Fichier
vla close : (even * odd) Fichier -> unit
```

2. Type non uniforme

ici `place` remplace `perfect_tree`.

Exercice 2

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

let t = Node(1,((2,3),Node(((4,5),(6,7), Empty)))) (*exemple*)

val split : ('a * 'a) place -> ('a place * 'a place)

let rec split : type a . a place -> (a place * a place) =
fun t -> match t with
        | Empty -> (Empty, Empty)
        | Node((t1,t2),q) -> let (q1,q2) = split q
                in (Node(t1,q2),Node(t2,q2))
```

Exercice 3

```
val merge : 'a place -> 'a place -> ('a * 'a) place

(* Node(3,Node((2,5),Empty)) => int place
Node(6,Empty) => int place
place non fusionable
*)

type zero
type _ succ
type ('a, _) place =
        | Empty : ('a * zero) place
        | Node : 'a * (('a * 'a), 'p) place -> ('a, 'p succ) place

let rec split : type a p . ((a * a), p) place -> ((a, p) place * (a, p) place) =
function
| Empty -> (Empty, Empty)
| Node((t1,t2),q) -> let (q1,q2) = split q in (Node(t1,q1),Node(t2,q2))

let rec merge : type a p . (a, p) place -> (a, p) place -> (a * a, p) place =
fun t1 t2 -> match (t1, t2) with
        | Empty,Empty -> Empty
        | Node(a1,q1),Node(a2,q2) -> let q = merge q1 q2 in Node((a1,a2),q)
```

3. GADT

#TODO

```
type _ repr =
| Int : int -> int repr
| Add : (int -> int -> int) repr

let eval : type a. a repr -> a =
        function
        | Int i -> i
        | Add -> (fun a b -> a+b)

eval Add (eval (Int 3) eval(Int 5))
```

TD6 : Les flux

1 Type Abstrait : les flux

les listes on les déconstruit
les flux on les construit

`uncons` récupère la tête et la queue d'un flux et renvoie une option

unfold est le dual de fold :
`Fold::('a -> 'b -> 'b) -> 'b -> 'a list -> 'b`
`unFold::('b -> ('a * 'b) option) -> 'b -> 'a t`

```
let flux_nul = Flux.unfold (fun c -> Some(c,c)) 0
```

Exercice 1

```
val constant : 'a -> 'a flux

let constant c = Flux.unfold (fun a -> Some(a,a)) c
let constant' c = Flux.unfold (fun () -> Some(a,())) c

val apply : ('a -> 'b) -> 'a t -> 'b t

let map f flux = apply (constant f) flux
let map2 f flux1 flux2 = apply (map f flux1) flux2
```

Un flux est une valeur *récursive* (`x = 0::x`)
`Tick` -> machine à friandises

Les constructeurs en Ocaml ne sont pas des fonctions et interviennent autre part dans la compilation.

```
type 'a t = Tick of (('a * 'a t) option) Lazy.t

let vide = Tick (lazy None)

let cons t q = Tick (lazy Some(t,q))

(* lazy 5 -> int Lazy.t
 * lazy Some(3, lazy None)
 *)

let uncons (Tick fp) = Lazy.force fp
(* fait fondre les glacon est force le calcul *)

let rec unfold f e =
        Tick (lazy (
        match f e with
        | None -> None
        | Some(t, e') -> Some(t, unfold f e')
        ))
```

On *considère* `lazy` comme un constructeur

---

2 Applications

Exercice 2

La suite de *Fibonacci* est défini de la manière suivante :

$$(f_n)_{n\in\mathbb{N}} \begin{cases} f_0 = f_1 = 0 \\ f_{n+2} = f_{n+1} + f_n \end{cases}$$

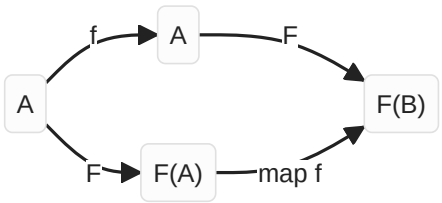
```
let f01 = cons 0 (cons 1 vide)

let fib = unfold (fun (fn, fn_plus_un)
        -> Some(fn (fn_plus_un, fn + fn_plus_un)))
        (0,1)

(* Idée
 * << f0, f1, f2, ...
 * << f1, f2, f3, ...
 * ------------------
 * (+) << f0+f1, f1+f2, f2+f3
 *)
let tail f =
        match uncons f with
        | None -> failwith "flux vide :("
        | Some(_,q) -> q

let rec fib =
        Tick (lazy (Some(0, Tick (lazy (Some(1, map2 (+) fib (tail fib)))))))

(* fib est une valeur récursive *)
```

Exercice 3

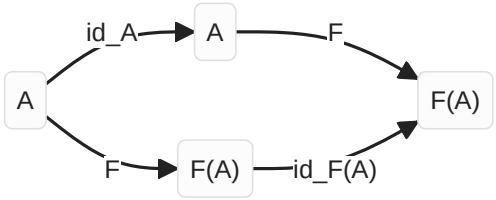
#TODO

TD7 : Structures monadiques

le type `FONCTEUR` ressemble à une fonction sur les types (*cf. la théorie des catégories*)



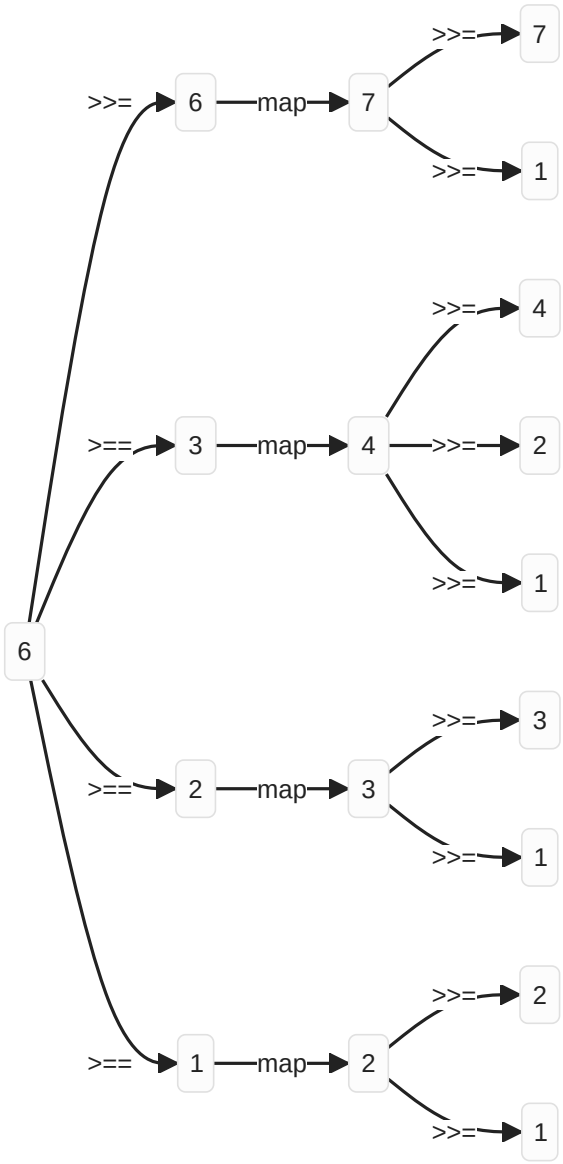
exemple : +1 sur un entier => +1 sur une liste d'entier



Le `FONCTEUR` préserve la structure mais est irréversible

Monade ≈ Monoïde

`return` : wrap
`bind` : séquence de calculs



pour une monade
`map f S =` $\{s' \mid s' = f(s),\ s \in S\}$
`return a =` $\{a\}$
`S >>= f =` $\bigcup\{f(a) \mid a \in S\}$

Exercice 1

```
type 'a t = Iter of 'a node Lazy.t
and 'a node = ('a * 'a t) option

let uncons (Iter i) =Lazy.force i

let return x = Iter(lazy(
        Some(x, Iter(lazy None)
))

let rec map f a = Iter(lazy(
        match (uncons a) with
        | None -> None
        | Some(t,q) -> Some(f t,map f q)
))

let zero = Iter(lazy None)

let rec (++) l1 l2 = Iter(lazy(
        match (uncons l1) with
        | None -> uncons l2
        | Some(t,q) -> Some(t,q ++ l2)
))

let rec (>>=) a f = Iter(lazy(
        match (uncons a) with
        | None -> None
        | Some(t,q) -> uncons((f t)++(q>>+f))
))
```

Exercice 2

```
module type WT = sig type t end

module WRITER (W : WT) (*: MONADE*) =
        struct
                type 'a t = 'a * W.t Flux.t

                (*tell : W.t -> unit t*)
                let tell m = ((),Flux.(cons m vide))

                let map f (v,t) = (f v, t)
                let return x = (x, Flux.vide)
                let (>>=) (a,t) f = let (b,w) = f a
                        in (b,Flux.append t w)

                let run (v,m) = (v,m)
        end
```

Exercice 4

```
type 'a t = s -> ('a * s)

let map f x = fun s ->
        let (a,s') = (x,s) in (f a,s)

let return x = fun s -> (x,s)

let (>>=) a f = fun s0 ->
        let (va,s1) = a s0 in (f va) s1
```

TD9 : Les continuations

3. Continuations natives

```
let append l1 l2 =
        let rec app_X_l2 l1 =
                match l1 with
                | [] -> l2 (* shift (fun k -> k l2) *)
                | t1::q1 -> let app_q1_l2 = app_X_l2 q1
                        in t1::app_X_l2
         in app_X_l2 l1 (* in reset(app_X_l2 l1) *)
```

`shift (fun k -> k l2)` est une fermeture

Ex : `let plus n = fun x -> x + n`
si on appelle `plus 3` on obtient la fonction `fun x -> x + 3`, la valeur 3 est sauvegardée, c'est une fermeture.

`shift` va de paire avec `reset`

---

4. Application : optimisation de code

Exercice 1

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

let rec prod_int_list' l =
        match l with
        | [] -> 1
        | 0::q -> 0
        | t::q -> t * (prod_int_list q)

let rec prod_int_list_cc l =
        match l with
        | [] -> 1
        | 0::q -> shift prompt0 (fun _ -> 0)
        | t::q -> t * prod_int_list_cc q

let prod_int_list_rap l =
        push.prompt0                        (* reset s'appelle push dans Delimcc *)
                (fun () -> prod_int_list_cc l)
```

---

5. Application : Resumable Exceptions

Exercice 2

```
let leccture_cas_nominal nom =
        let f = open_in (if Sys.file_exists nom then nom
                else shift prompt0 (fun k -> Request k)) in
        let l = input_line f in close_in f; Done l

let lecture_cas_erreur nom k =
        printf "gngngn %s \n" nom;
        let nom' = read_line () in
                k nom'

let main nom =
        match push_prompt prompt0 (fun () -> lecture_cas_nominal nom) with
        | Done l -> l
        | Request k -> match lecture_cas_erreur nom k with
                | Done l -> l
                | Request k -> assert false
```

---

6. Application : programmation concurrente et Green Threads

Exercice 3

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

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

Exercice 4

```
let scheduler () =
        let rec loop file =
                match file with
                | [] -> ()
                | proc::q ->
                        let yield = push_prompt p proc in
                        match yield with
                        | Done () -> loop q
                        | Request k -> loop (q @ [k])
        in loop (ping,pong)
```

Application : coroutine et *yield*

Exercice 5

TD10 : Les parseurs

1. Reconnaissance de langage

"a b" -> |a, b| -> { Λ }
"a b c" -> |a, b| -> {"c"}
"c d" -> |a, b| -> ϕ
{"a b c", "a b d", "a e"} -> |a, b| -> {"c", "d"}

"ax" -> |a + b| -> {"x"}
"bx" -> |a + b| -> {"x"}
{"axy", "bz"} -> |a + b| -> {"xy", "z"}

```
let F x = ...
let F = fun x -> ...
```

-> p1 -> p2 ->

```
let psequence p1 p2 = fun flux ->
        Solution.(p1 flux >>= p2)

let pchoix p1 p2 = fun flux ->
        Solution.(p1 flux ++ p2 flux)
```

2. Les parseur comme dénotations de langages

```
let rec eval lang =
        match lang with
        | Nothing -> perreur
        | Empty -> pvide
        | Letter a -> ptest ((=) a)
        | Sequence (l1, l2) -> psequence (eval l1) (eval l2)
        | Choice (l1, l2) -> pchoix (eval l1) (eval l2)
        | Repeat l -> eval (Choice (Empty, Sequence(l, Repeat(l))))

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

3. Parsing plus général

'a Flux.t -> |('a, 'b)parser| -> ('b * 'a Flux.t) Solution.t

```
let perreur = fun f -> zero f

let pnul = fun f -> (return ()) f

let pvide = fun f ->
        match Flux.uncons f with
        | None -> Solution.return ((),f)
        | Some _ -> Solution.zero

let ptest (p, 's -> bool) = fun f ->
        match Flux.uncons f with
        | None -> Solution.zero
        | Some(t,q) -> if p t then Solution.return(t,q) else Solution.zero

let pchoix p1 p2 = (p1 ++ p2)

let psequence p1 p2 =
        p1 >>= (fun ra -> p2 >>= (fun rb -> return (ra,rb)))
```

```
type ast = Plus of ast *bast | Var of char
```

Expr -> Var
Expr -> '(' Expr '+' Expr ')'
Var -> char

```
let paro = ptest ((=) '(')
let parf = ptest ((=) ')')
let plus = ptest ((=) '+')
let pchar = ptest (fun x -> 'a' <= x && x <= 'z')

let var = pchar >>= fun c -> return (Var(c))
let rec expr flux =
        (var ++ (paro >>= fun _ -> expr >>= fun e1 -> plus >> fun _ ->
                expr >>= fun e2 -> parf >>= fun _ -> return (Plus(e1,e2))
        )) flux
```