++ IPF3

...

Phoko

(et Ghorhahm pour corriger les conneries que je pourrais dire)

17 décembre 2018

Plan

- Rappels :
    - Rappels de base sur OCaml
    - Modules et foncteurs
    - Structures Set et Map
    - Fonctions map et fold

- Corrigé du partiel 2017-2018

Rappels de base sur OCaml

OCaml = langage typé

```
OCaml type

int
float
bool
char
string
unit
```

Ocaml “devine” les types :

```
# 1 + 2;;
- : int = 3
```

```
# 1.0 +. 2.0;;
- : float = 3.
```

```
# 1.0 + 2.0;;
File "", line 1, characters 0-3:
Error: This expression has type float but an expression was expected of type
         int
```

Fonctions utiles :

```
  float_of_int;;
- : int -> float = <fun>

  int_of_float;;
- : float -> int = <fun>

  string_of_int;;
- : int -> string = <fun>

  int_of_string;;
- : string -> int = <fun>
```

“Variables”

let

```
# let a = 1;;
val a : int = 1
# a;;
- : int = 1
```

let … in

```
let a = 2 in
  a + 3
;;

(* - : int = 5 *)
```

Listes et tuple

OCaml name	Example type definition	Example usage
list	int list	[1; 2; 3]
tuple	int * string	(3, "hello")

Concaténation :

```
let my_list = [] ;;
(* val my_list : 'a list = [] *)

let my_list = 3::my_list ;;
(* val my_list : int list = [3] *)

let my_list = 2::my_list ;;
(* val my_list : int list = [2; 3] *)

let my_list = my_list@[5] ;;
(* val my_list : int list = [2; 3; 5] *)
```

Fonctions utiles :

(disponibles en annexe du sujet de partiel)

```
let l = [1; 2; 3];;
(* val l : int list = [1; 2; 3] *)

List.length l;;
(* - : int = 3 *)

let l2 = List.rev l;;
(* val l2 : int list = [3; 2; 1] *)

List.mem 2 l;;
(* - : bool = true *)

List.mem 4 l;;
(* - : bool = false *)
```

Tuples :

```
# let x = (3, "salut");;
val x : int * string = (3, "salut")
```

Récupérer les valeurs d'un tuple :

```
# let (my_int, my_string) = x;;
val my_int : int = 3
val my_string : string = "salut"
```

ou

```
# fst x;;
- : int = 3
# snd x;;
- : string = "salut"
```

Définir un type

```
type foo =
  | Nothing
  | Int of int
  | Pair of int * int
  | String of string
;;

(* type foo = Nothing | Int of int | Pair of int * int | String of string *)
```

```
type forme =
  | Point
  | Cercle of float
  | Rectangle of float*float
;;

type surface = S of float
;;
```



```
# let x = Cercle(3.0);;
val x : forme = Cercle 3.
```

```
# let x = Rectangle(3.0, 3.0);;
val x : forme = Rectangle (3., 3.)
```

Définir une fonction

Définir une fonction :

```
let average a b =
  (a +. b) /. 2.0
;;

(* val average : float -> float -> float = <fun> *)
```

Appeler une fonction :

```
# let x = average 1.0 4.0;;
val x : float = 2.5
```

Mot-clé “fun” :

```
let f = fun x -> x + 1;;
(* val f : int -> int = <fun> *)

let _ = f 2;;
(* - : int = 3 *)
```

Fonctions récursives

```
(* ajoute l'élément `elt` `n` fois à la fin de la liste `l`.
   @requires: `elt` l'élément à ajouter.
   @requires: `n` l'entier qui indique combien de fois on veut ajouter notre élément.
   @requires: `l` la liste à laquelle on veut ajouter notre élément.
   val ajout : 'a -> int -> 'a list -> 'a list = <fun>
*)

let rec ajout elt n l =
  if (n = 0) then l else ajout elt (n-1) (l@[elt])
;;
```

Faire un match

Exemple sur les listes :

```
let is_empty l = match l with
  | [] -> true
  | _ -> false
;;
(* val is_empty : 'a list -> bool = <fun> *)
```

```
# is_empty [];;
- : bool = true
# is_empty [2];;
- : bool = false
```

Exemple sur un parameterized type :

```
type 'a binary_tree =
  | Leaf of 'a
  | Tree of 'a binary_tree * 'a binary_tree
;;
(* type 'a binary_tree = Leaf of 'a | Tree of 'a binary_tree * 'a binary_tree *)
```

```
let rec somme_tree t = match t with
  | Leaf(valeur) -> valeur
  | Tree(left, right) -> (somme_tree left) + (somme_tree right)
;;
(* val somme_tree : int binary_tree -> int = <fun> *)
```

```
# let my_tree = Tree(Leaf 3, Leaf 4);;
val my_tree : int binary_tree = Tree (Leaf 3, Leaf 4)
```



```
# somme_tree my_tree;;
- : int = 7
```

Exceptions

Créer une exception :

```
exception Mot_vide;;
```

Lever une exception :

```
match mot with
| [] -> raise Mot_vide
```

Écrire du code qui gère les cas où une exception peut être levée :

```
let remove_one_edge src dst g =
  try
    let old_set = succs src g in
    let new_set = MySet.filter (fun (clef, poids) -> if clef = dst then false else true) old_set in
    let new_g = MyMap.add src new_set g in
    new_g
  with Not_found -> g
;;
```

par exemple, cette fonction peut lever l'exception Not_found

Modules et foncteurs

Modules

TODO

Structures Set et Map

Structure Set

https://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.S.html

Un Set = ensemble d'éléments, sans doublon.

Pour créer un Set :

```
module OrderedInt =
struct
    type t = int
    let compare = compare
end
;;

module MySetInt = Set.Make(OrderedInt)
;;
```

Set d'int

```
module OrderedString =
struct
    type t = string
    let compare = compare
end
;;

module MySetInt = Set.Make(OrderedString)
;;
```

Set de string

Structure Set

Fonctions utiles des Set : données en annexe du partiel.

Exemple : is_empty, mem, add, remove, fold, map, cardinal, ...

```
let s = MySetInt.empty;;
let s = MySetInt.add 1 s;;
let s = MySetInt.add 2 s;;
let s = MySetInt.add 3 s;;
MySetInt.elements s;;
(* - : MySetInt.elt list = [1; 2; 3] *)

let s = MySetInt.remove 3 s;;
MySetInt.elements s;;
(* - : MySetInt.elt list = [1; 2] *)
```

Structure Map

https://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.S.html

Une Map = des associations key → value
(un “dictionnaire”)

```
type key
    The type of the map keys.

type +'a t
    The type of maps from type key to type 'a.
```

Créer une Map :

```
module OrderedInt =
struct
    type t = int
    let compare = compare
end
;;

module MyMap = Map.Make(OrderedInt)
;;
```

```
let m = MyMap.empty;;
(* val m : 'a MyMap.t = <abstr> *)

let m = MyMap.add 0 "bonjour" m;;
(* val m : string MyMap.t = <abstr> *)

let m = MyMap.add 1 3 m;;
(* Error *)
```

Structure Map

Fonctions utiles des Map : données en annexe du partiel.

Exemple : is_empty, mem, add, remove, fold, find, ...

```
let m = MyMap.add 2 "test" m;;
let m = MyMap.add 2 "XD" m;;

MyMap.mem 2 m;;
(* - : bool = true *)

MyMap.mem 3 m;;
(* - : bool = false *)

MyMap.find 2 m;;
(* - : string = "XD" *)

MyMap.find 3 m;;
(* Exception: Not_found. *)
```

Fonctions map et fold

Fonction map

List

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

`List.map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.

```
let l = 1 :: 2 :: 3 :: [];;
(* val l : int list = [1; 2; 3] *)

let _ = List.map (fun x -> x - 1) l;;
(* - : int list = [0; 1; 2] *)

let _ = List.map (fun x -> [x]) l;;
(* - : int list list = [[1]; [2]; [3]] *)
```

Set

"Since 4.04.0"

Map

```
val map : ('a -> 'b) -> 'a t -> 'b t
```

`map f m` returns a map with same domain as `m`, where the associated value `a` of all bindings of `m` has been replaced by the result of the application of `f` to `a`. The bindings are passed to `f` in increasing order with respect to the ordering over the type of the keys.

```
let m = MyMap.empty;;
let m = MyMap.add 0 "cc" m;;
let m = MyMap.add 1 "bonjour" m;;
let m = MyMap.add 2 "yo" m;;

let m = MyMap.map (fun value ->
    value ^ "!") m
;;

let _ = MyMap.find 1 m;;
(* - : string = "bonjour!" *)
```

Fonction fold : pour les List

```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
    List.fold_left f a [b1; ...; bn] is f (... (f (f a b1) b2) ...) bn.

val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
    List.fold_right f [a1; ...; an] b is f a1 (f a2 (... (f an b) ...)). Not tail-recursive.
```

Quelle différence ?

⇒ *“fold_left is tail recursive whereas fold_right is not.*
So if you need to use fold_right on a very lengthy list,
you may instead want to reverse the list first then use fold_left”

```
let l = [1; 2; 3];;

let _ = List.fold_left (fun acc x ->
    acc + x) 0 l
;;
(* - : int = 6 *)
```

Fonction “somme”

```
let l = [23; 54; 3; 65];;

let _ = List.fold_left (fun acc x ->
    if x < acc then x else acc) (List.hd l) l
;;
(* - : int = 3 *)
```

Fonction “min”

```
let l = [12; 24; 36; 48; 60];;

let _ = List.fold_left (fun acc elt ->
    if (elt < 42)
    then elt :: acc (* on l'ajoute *)
    else acc (* on ne l'ajoute pas *)
) [] l
;;
(* - : int list = [36; 24; 12] *)
```

Renvoie la liste contenant uniquement les éléments de l qui sont < 42

Fonction fold : pour les Set

```
val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
```

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`, in increasing order.

```
let s = MySet.empty;;
let s = MySet.add 1 s;;
let s = MySet.add 2 s;;
let s = MySet.add 3 s;;
let s = MySet.add 4 s;;

let _ = MySet.fold (fun elt acc ->
    elt + acc) s 0
;;
(* - : int = 10 *)
```

Fonction “somme”

```
let s = MySet.empty;;
let s = MySet.add 1 s;;
let s = MySet.add 2 s;;
let s = MySet.add 3 s;;
let s = MySet.add 4 s;;

let _ = MySet.fold (fun elt acc ->
    elt * acc) s 1
;;
(* - : int = 24 *)
```

Fonction “produit”

```
let s = MySet.empty;;
let s = MySet.add 12 s;;
let s = MySet.add 24 s;;
let s = MySet.add 36 s;;
let s = MySet.add 48 s;;
let s = MySet.add 60 s;;

let s2 = MySet.fold (fun elt acc ->
    if elt < 42
    then MySet.add elt acc (* on l'ajoute *)
    else acc (* on ne l'ajoute pas *)
) s MySet.empty
;;
(* val s2 : MySet.t = <abstr> *)

let _ = MySet.elements s2;;
(* - : MySet.elt list = [12; 24; 36] *)
```

Renvoie l’ensemble des éléments < 42

Fonction fold : pour les Map

```
val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

`fold f m a` computes `(f kN dN ... (f k1 d1 a)...)`, where `k1 ... kN` are the keys of all bindings in `m` (in increasing order), and `d1 ... dN` are the associated data.

```
let m = MyMap.empty;;
let m = MyMap.add 1 10 m;;
let m = MyMap.add 2 20 m;;
let m = MyMap.add 3 30 m;;

let _ = MyMap.fold (fun key value acc ->
    let (acc_key, acc_value) = acc in
    (key + acc_key, value + acc_value)
) m (0, 0)
;;
(* - : int * int = (6, 60) *)
```

Renvoyer le couple (somme keys, somme values)

Renvoyer la map contenant uniquement les couples key/value dont la value est < 42 :

```
let m = MyMap.empty;;
let m = MyMap.add "n1" 12 m;;
let m = MyMap.add "n2" 24 m;;
let m = MyMap.add "n3" 36 m;;
let m = MyMap.add "n4" 48 m;;
let m = MyMap.add "n5" 60 m;;
(*
"n1" -> 12
"n2" -> 24
"n3" -> 36
"n4" -> 48
"n5" -> 60
*)

let m2 = MyMap.fold (fun key value acc ->
    if value < 42
    then MyMap.add key value acc (* on l'ajoute *)
    else acc (* on ne l'ajoute pas*)
) m MyMap.empty
;;
(*
"n1" -> 12
"n2" -> 24
"n3" -> 36
*)
```