Chapitre 9 - Fonctionnelles

1. Introduction, rappels

Syntaxe allégée pour l'écriture d'une fonction Au passage, il existe une syntaxe allégée pour définir les fonctions, que nous pourrons utiliser à partir de maintenant :

```
let f x = x+2 ; ;
(* f  : int -> int *)

#f 3 ; ;
(*-  : int = 5 *)
```

Le polymorphisme

Exemple 1

La fonction

```
let rec longueur = fun
| x  : : q -> 1 + longueur(q)
| _ -> 0  ; ;
```

a pour type : `'a list -> int`. Le type `'a list` est un type **polymorphe**, et cette fonction est, elle-aussi, qualifiée de polymorphe. Cela signifie que l'on peut l'utiliser sur plusieurs types : `int list, bool list, string list...`
`'a, 'b, 'c...` sont appelés des variables polymorphes.

Lorsque l'on souhaite éviter le polymorphisme d'une fonction, il est possible de contraindre le type d'une expression à l'aide de la syntaxe :

```
(expr  : type)
```

Exemple 2

La fonction `egal`

```
let egal = fun (x,y)->x=y ; ;
(* egal  : 'a * 'a -> bool *)
```

ainsi écrite, est polymorphe. Pour l'éviter, nous pouvons ainsi annoter la fonction :

```
let egal = fun ((x  : int),(y  : int)) -> x=y ; ;
(*egal  : int * int -> bool *)
```

Plutôt que de *deviner* le type de votre fonction (nous appelons cela **l'inférence de type**), Caml va *vérifier* que le corps de votre fonction est cohérent avec les annotations de types

(nous appelons cela **la vérification de type**).

Remarque : L'annotation de type peut également s'avérer très utile, lorsqu'on écrit des fonctions complexes à plusieurs paramètres, pour se rappeler qui est qui. Cf semestre 6...
L'objet de ce chapitre Une des caractéristiques des langages fonctionnels est que les fonctions y sont des expressions à part entière (et non une super-structure comme dans les langages impératifs), cela signifie qu'elles peuvent apparaître dans des listes, des n-uplets, et bien sûr comme paramètre ou résultat d'une fonction !

Pour bien comprendre ce chapitre, il est important de **bien regarder les types** !!

2. Fonctionnelles

2.1. Définition et premiers exemples

Définition 1

Nous appelons **fonctionnelle (ou fonction d'ordre supérieur), une fonction dont les arguments et/ou le résultat sont eux-mêmes des fonctions.**

Remarque : Ces fonctions peuvent être polymorphes.

Exemple 3

La fonction `fois_n` fabrique la fonction qui multiplie par n :

```
let fois_n = fun n -> (fun m -> n*m) ;;
(* fois_n  : int -> int -> int *)
```

Comprenez vous le type retourné par Caml ?
`fois_n` est bien une fonction : appliquée à un argument, le résultat est une fonction.

```
let double = fois_n (2) ;;
(* double  : int -> int *)
```

Cette fonction doit à son tour être appliquée à un argument pour obtenir une valeur entière :

```
double (3) ;;
-  : int = 6

fois_n(2)(3) ;;
-  : int = 6
```

Attention ! Source d'erreur :

- Le constructeur de types `->` est prioritaire à droite : `int -> int -> int` équivaut à `int -> (int -> int)`.
- En revanche, l'opérateur d'application des fonctions est prioritaire à gauche : `fois_n 2 3` équivaut à `(fois_n 2) 3`.

Exemple 4

Ecrivons une fonction qui étant donnée une fonction f passée en paramètre, construit la fonction qui à x non nul, renvoie $\frac{f(x)}{x}$:

```
#let divx= fun f-> (fun 0.-> failwith "x nul"
|x->f(x)/.x) ; ;
(* divx  : (float -> float) -> float -> float *)
```

Comprenez-vous le type de `divx` ??
Voici des exemples d'utilisation :

```
#divx(sin) ; ;
-  : float -> float = <fun>

#divx(sin)(0.1) ; ;
-  : float = 0.998334166468

#let divsin=divx(sin) ; ;
divsin  : float -> float = <fun>

#divsin(0.1) ; ;
-  : float = 0.998334166468
```

Exemple 5

Autre exemple avec des chaînes de caractères :

```
#let star = fun f-> fun ch->"**"^f(ch)^"**" ; ;
star  : ('a -> string) -> 'a -> string = <fun>

#let duplique = fun ch->ch^ch ; ;
#let dupliqueEtStar = star (duplique) ; ;


#dupliqueEtStar("bonjour") ; ;
-  : string = "**bonjourbonjour**"


#let convertirEtStar = star(string_of_int) ; ;
#convertirEtStar (1234) ; ;
-  : string = "**1234**"
```

2.2. Définition d'une fonction à plusieurs paramètres

Une fonctionnelle étant une fonction comme une autre, il est toujours possible d'utiliser la syntaxe classique :

```
fun a_1 -> fun a_2 -> ... fun a_n -> Corps
```

où `Corps` est le corps de la fonction de type `t` non fonctionnel.
Il est possible d'utiliser également la syntaxe allégée :

```
fun a_1 a_ 2 ... a_n -> Corps
```

Exemple 6

Observer la différence des types :

```
let foisN1 = fun (n,a) -> n*a  ; ;
(* foisN1  : int*int -> int *)
(* Il s'agit d'une fonction à un seul paramètre, qui est un couple *)

let foisN2 = fun n -> fun a -> n*a  ; ;
(* foisN2  : int -> int -> int *)
(* Il s'agit cette fois, d'une fonctionnelle *)
```

Nous pouvons écrire également :

```
let foisN2 = fun n a -> n * a  ; ;
```

ou encore :

```
let foisN2 n a = n*a  ; ;
```

Attention toutefois, ces deux dernières syntaxes ne permettent pas le filtrage.

2.3. Application totale ou partielle

Définition 2

L'application totale d'une fonction à n paramètres nécessite n arguments. Le résultat n'est plus une fonction mais une valeur de type `t`.

Exemple 7

```
let divise = fun n -> fun
0 -> failwith "division par 0"
|p -> n / p  ; ;
(* divise  : int -> int-> int *)
```

L'appel de fonction étant prioritaire, le parenthésage est inutile :

```
divise 6 3  ; ;
-  : int = 2
```

Définition 3

Tout l'intérêt d'une fonctionnelle réside dans la possibilité d'utiliser des applications partielles. On ne fournit alors pas tous les arguments à la fonction et le résultat est une fonction.

Exemple 8

```
#let div10 = divise 10 ; ;
val div10  : int -> int = <fun>

#div10 3 ; ;
-  : int = 3
```

Important : Il est donc fondamental de choisir avec soin l'ordre des paramètres pour pouvoir définir les applications partielles intéressantes.

2.4. Curry de Caml

Une fonction à plusieurs arguments peut donc prendre deux formes :

```
let f = fun (x,y) -> 2*x + 3*y  ; ;
(* f  : int * int -> int *)
```

Cette forme est dite **non curryfiée**.

```
let f = fun x y -> 2*x + 3*y  ; ;
(* f  : int -> int -> int *)
```

S'il est intéressante d'écrire des applications partielles, alors cette deuxième forme est préférable. Elle est dite **curryfiée**, en hommage au mathématicien et logicien américain Haskell Curry (1900 - 1982).
Remarque : Il est toujours possible de passer d'une forme à l'autre :

```
let curry = fun f x y -> f(x,y)  ; ;
let ecurry = fun f (x,y) -> f x y  ; ;
```

Exercice . Exercice TP 1

1. Déterminer le type de la fonction suivante et tester :

   ```
   let fzz = fun f a -> f(a+1) = 0  ; ;
   ```

2. Pour chaque requête suivante, donner la réplique Caml et commenter

   ```
   fzz(3)  ; ;
   let g = fzz(fun n -> 3)  ; ;
   g 4  ; ;
   ```

3. Construire un exemple de fonction de chaque type suivant :
   (a) `int -> bool -> float`
   (b) `(int -> bool) -> float`

3. Fonctionnelles et récursivité

Exercice . Exercice TP 2

1. Écrire une fonction `sigma : (int -> int) -> int -> int` récursive permettant de calculer :

$$\sum_{i=0}^{n} f(i)$$

pour une fonction f et un entier n quelconques.

```
# sigma (fun i-> i*i) 10 ; ;
- : int = 385
```

Cette version possède l'inconvénient que la récursivité porte sur n et sur f, alors que f ne change pas. L'application partielle est donc recalculée à chaque appel récursif.

2. Redéfinir la fonction `sigma`, non récursive, mais possédant une fonction récursive définie localement et ne portant que sur le paramètre n.

Un petit évaluateur Caml

Exercice . TP 3

Nous avons étudié les modes d'évaluation des différentes expressions Caml. Le but de cet exercice est d'écrire un petit interpréteur Caml reconnaissant quelques expressions arithmétiques en nombre entier et les définitions globales et locales. Nous proposons d'écrire un évaluateur pour cette version très allégée de Caml que nous appellerons *Caml O'Chocolat*.

Les expressions Caml O'Chocolat En Caml O'Chocolat, une expression peut être une constante entière, un identificateur (réduit à un seul caractère), la somme de deux expressions, le produit de deux expressions ou une expression à une certaine puissance entière. Nous définissons donc le type expression comme suit :

```
type Expression = Const of int | Var of char
| Add of Expression*Expression
| Mult of Expression*Expression
| Puiss of Expression*int ; ;
```

1. Définir deux expressions e_1 et e_2 représentant respectivement $1+2x^3$ et $1+a^2$.

   Pour évaluer nos expressions, nous devons disposer d'un environnement courant. On définit une liaison entre un identificateur et la valeur d'une expression par le type produit suivant :

   ```
   type liaison = {id  : char ; valeur  : int} ; ;
   ```

   On peut alors définir un environnement comme une liste de liaisons. On pourra supposer qu'initialement, notre environnement courant, que nous noterons `envC` est :

   ```
   let envC = [{id='a' ;valeur=3} ;{id='b' ;valeur=4}] ; ;
   ```

2. Écrire une fonction `evalVar : char * liaison list -> int` qui, à un caractère *c* et à un environnement *env* associe la valeur entière liée à *c* dans l'environnement *env* si cette liaison existe et un message d'erreur sinon.

```
evalVar ('a',envC) ; ;
-  : int = 3
evalVar ('b',envC) ; ;
-  : int = 4
evalVar ('x',envC) ; ;
Exception non rattrapée : Failure " identificateur inconnu"
```

3. Écrire une fonction `puissance : int * int -> int` qui calcule (naïvement) la puissance entière puissance entière positive d'un nombre entier. On affichera une erreur si la puissance est strictement négative.
4. Écrire une fonctionnelle curryfiée `evalExp :liaison list -> Expression -> int` qui, à un environnement et une expression donnés associe la valeur de cette expression dans l'environnement ou un message d'erreur si l'expression contient un identificateur non lié.
   On définira une fonction récursive auxiliaire locale ne portant que sur l'expression à évaluer et on pourra bien sûr utiliser `evalVar` et puissance.

Définitions globales et locales : Une définition crée la liaison entre un identificateur et la valeur d'une expression. Nous représentons une définition par le type produit suivant :

```
type Definition={ident :char  ; exp : Expression } ; ;
```

1. Écrire une fonction `ajoute : liaison list -> Definition -> liaison list` qui, à un environnement *env* et à une définition *d* associe le nouvel environnement obtenu en ajoutant à *env* la liaison entre l'identificateur et la valeur de l'expression dans l'environnement *env*.

```
ajoute envC {ident='x' ;exp = Add(Var 'a',Const 3)} ; ;
-  : liaison list =
[{id = 'x' ; valeur = 6} ; {id = 'a' ; valeur = 3} ; {id = 'b' ; valeur = 4}]
```

Programmes Caml O'Chocolat Un programme Caml O'Chocolat peut être soit une expression, soit une définition globale, soit une définition locale permettant la réalisation d'une définition le temps de l'évaluation d'un programme. On définit donc le type récursif suivant :

```
type Programme =
Elementaire of Expression
| DefGlob of Definition
| DefLocale of Definition*Programme ; ;
```

Le petit programme Caml O'Chocolat suivant :

```
Soit x=7 dans
Soit y = x + 3 dans x + 3*y  ; ;
```

sera alors représenté par l'objet de type `Programme` suivant :

```
let p  = DefLocale ({ident = ‘x‘ ; exp = Const 7},
DefLocale ({ident = ‘y‘ ; exp = Add (Var ‘x‘, Const 3)},
Elementaire (Add (Var ‘x‘, Mult (Const 3, Var ‘y‘))))) ; ;
```

1. Écrire une fonction `evalProg : Programme * liaison list -> int * liaison list` qui, à un programme et un environnement *env* donnés, associe la valeur de ce programme dans l'environnement *env* (la valeur affichée par la réplique Caml) et le nouvel environnement.

```
Pour le programme  : 1+a2  ; ;
#evalProg( Elementaire e2,envC) ; ;
-  : int * liaison list = 10, [{id = ’a’ ; valeur = 3} ; {id = ’b’ ; valeur = 4}]


Pour le programme  : soit x = 1+a2  ; ;
#evalProg( DefGlob {ident=’x’  ; exp=e2 },envC) ; ;
-  : int * liaison list =
10, [{id = ’x’ ; valeur = 10} ; {id = ’a’ ; valeur = 3} ; {id = ’b’ ; valeur = 4}]


Pour le programme  : soit x = 7 dans soit y = x+3 dans x+3*y  ; ;
#evalProg ( p ,envC) ; ;
-  : int * liaison list = 37, [{id = ’a’ ; valeur = 3} ; {id = ’b’ ; valeur = 4}]
```

4. Bonus : Les exceptions

Il arrive souvent que l'on veuille écrire des fonctions qui ne sont pas définies partout. Dans de tels cas, comme toute expression Caml doit avoir une valeur, il nous faut un moyen pour signaler qu'une fonction ne peut pas être calculée : on appelle cela des exceptions.

4.1. Exceptions prédéfinies de Caml

```
1/0  ; ;
Exception non rattrapée  : Division_by_zero

hd[] ; ;
Exception non rattrapée  : Failure "hd"
```

Nous disons alors qu'une exception a été levée. L'évaluation normale de l'expression en cours est stoppée, et l'exécution s'arrête en affichant l'exception.
La fonction prédéfinie (que nous connaissons déjà !)

`failwith`

permet de lever l'exception `Failure`, accompagnée de la chaîne de caractères passée en paramètre.

```
let f = fun
0 -> failwith "pas d’argument nul"
```

```
| x -> 1./.x  ; ;

f(0.)  ; ;
Exception non rattrapée  : Failure "pas d'argument nul"
```

4.2. Exceptions déclarées par le programmeur

Il est également possible de déclarer de nouvelles exceptions, grâce à la syntaxe :

```
exception NomException  ; ;
```

Attention En OCaml, les noms d'exceptions devront commencer par une majuscule !
Ensuite, une fonction peut lever une telle exception grâce à la syntaxe :

```
raise NomException  ; ;
```

Exemple 9

```
#exception NombreNegatif ; ;

#let rec fact = fun
0->1
| n-> if n<0 then raise NombreNegatif
else n*fact(n-1) ; ;

(* fact  : int -> int = <fun> *)

#fact(-3) ; ;
Exception non rattrapée  : NombreNegatif
```

Exercice . TP 4

Écrire une fonction `associe : 'a -> ('a * 'b) list -> 'b` qui, à un élément a et une liste de couples $liste$ associe b si le couple (a,b) appartient à $liste$ et lève une exception "NonTrouvé" sinon.

4.3. Capturer une exception

Nous avons vu que la levée d'une exception interrompait l'évaluation normale. Ce comportement est parfois gênant : on voudrait pouvoir parfois appeler des fonctions qui risquent de lever des exceptions, et dans le cas où une exception est levée, faire un traitement particulier. Ceci peut se faire à l'aide de la construction de capture d'une ou plusieurs exceptions, qui a la forme suivante (analogue au filtrage) :

```
try expression with
exception_1 -> expression_1
|exception_2 -> expression_2
...
|exception_n -> expression_n
```

et qui s'évalue en `expression_i` si l'évaluation de *expression* lève l'exception `exception_i`. Toutes ces expressions doivent avoir le même type.

Exemple 10

```
fact(-3) ;;
Exception non rattrapée : NombreNegatif

try fact(-3) with
NombreNegatif -> 0 ;;
- : int = 0
```

Exercice

Nous définissons :

```
let dico= [("a","un") ; ("called","appelé") ;
("cat","chat") ; ("hand","main") ;
("is","est") ; ("language","langage") ;
("my","mon") ; ("us","nous") ;
("wonderful","magnifique")] ;;
```

1. Définir une exception `PasTrouve`.
2. Ecrire une fonction `trouve : 'a -> ('a * 'b) list -> 'b` qui cherche la traduction d'un mot à l'aide du dictionnaire, l'exception `PasTrouve` sera levée en cas d'échec.

   ```
   #trouve "cat" dico ;;
   - : string = "chat"

   #trouve "dog" dico ;;
   Exception non rattrapée : PasTrouve
   ```

3. Ecrire une fonction `traduire : string -> string` permettant de traduire un mot si on le trouve dans le dictionnaire, et rattrapant l'exception `PasTrouvé` en laissant le mot en anglais.

5. Bonus : Les types polymorphes

[A faire quand tout le reste a été fait (dont les sujets d'annales disponibles sur Moodle)]

Exemple 11

La fonction

```
let rec longueur = fun
    | x :: q -> 1 + longueur(q)
    | _ -> 0 ;;
```

a pour type : `'a list -> int`. Le type `'a list` est un type **polymorphe**, et cette fonction est, elle-aussi, qualifiée de polymorphe. Cela signifie que l'on peut l'utiliser sur plusieurs types : `int list`, `bool list`, `string list`...
`'a`, `'b`, `'c`... sont appelés des variables polymorphes.

Lorsque l'on souhaite éviter le polymorphisme d'une fonction, il est possible de contraindre le

type d'une expression à l'aide de la syntaxe :

```
(expr  : type)
```

Exemple 12

La fonction `egal`

```
let egal = fun (x,y)->x=y ; ;
(* egal  : 'a * 'a -> bool *)
```

ainsi écrite, est polymorphe. Pour l'éviter, nous pouvons ainsi annoter la fonction :

```
let egal = fun ((x  : int),(y  : int)) -> x=y ; ;
(*egal  : int * int -> bool *)
```

Plutôt que de *deviner* le type de votre fonction (nous appelons cela **l'inférence de type**), Caml va *vérifier* que le corps de votre fonction est cohérent avec les annotations de types (nous appelons cela **la vérification de type**).

Remarque : L'annotation de type peut également s'avérer très utile, lorsqu'on écrit des fonctions complexes à plusieurs paramètres, pour se rappeler qui est qui. Cf semestre 6...

ATTENTION, la suite n'est valable qu'en OCaml

Il est parfois utile de définir soi-même des types polymorphes.

Exemple 13

Nous pouvons par exemple, redéfinir le type liste :

```
type 'a list =
     Liste_vide
     | Cons of 'a * 'a liste  ; ;
```

Nous pouvons alors écrire :

```
Cons (1,Liste_vide) ; ;
-  : int liste = Cons (1, Liste_vide)

Cons("a",Cons("b",Liste_vide)) ; ;
-  : string liste = Cons ("a", Cons ("b", Liste_vide))
```

TP sur OCaml : Les multi-ensembles finis

Objectifs :

- Commencer à se familiariser avec OCaml en vue du semestre 6 (partie 1, des listes, de la récursivité)
- Manipuler un type polymorphe défini par l'utilisateur. (partie 2)

Exercice

Les multi-ensembles sont une généralisation de la notion d'ensemble.

Soit D un ensemble. Un multi-ensemble fini M d'éléments de D est un ensemble (abus de langage !) qui peut contenir plusieurs occurrences d'un élément, on dit aussi que chaque élément a un ordre de multiplicité.

Exemple : $M = \{0;0;2;2;2;3\}$ est un multi-ensemble d'entiers. L'ordre de multiplicité de 2 dans M est 3, puisque le chiffre 2 apparaît trois fois dans le multi-ensemble M. On note alors $M(2) = 3$.

Attention, l'ordre des éléments n'a pas d'importance !

Exemple : Nous avons aussi $M = \{0;2;0;2;3;2\}$.

⊛ Expliquer la différence entre les notions d'ensemble et de multi-ensemble, et les notions de liste de multi-ensemble.

Par des listes

Dans cette partie, nous représentons les multi-ensembles par des listes.

1. Écrire une fonction `multiplicite` qui calcule l'ordre de multiplicité de l'élément x dans le multi-ensemble m.

2. Écrire une fonction `enleve` qui retire une occurrence de l'élément x d'une liste m. Si x n'apparaît pas dans m, la liste sera retournée inchangée.

   On peut définir 4 opérations sur les multi-ensembles :
   - **La somme** $M +_m N$: l'ordre de multiplicité d'un élément x de $(M +_m N)$ est la somme de son ordre de multiplicité dans M et dans N (c'est à dire $\forall x, (M +_m N)(x) = M(x) + N(x)$).

     Exemple : $\{0;0;1;2\} +_m \{0;2;2;2\}$ est égal à $\{0;0;1;2;0;2;2;2\}$.

   - **L'union** $M \cup_m N$: l'ordre de multiplicité d'un élément x de $(M \cup_m N)$ est le maximum des deux ordres de multiplicité de x dns M et N (c'est à dire $\forall x, (M \cup_m N)(x) = \max(M(x), N(x))$).

     Exemple : $\{0;0;1;2\} \cup_m \{0;2;2;2\}$ est égal à $\{0;0;1;2;2;2\}$.

   - **L'intersection** $M \cap_m N$: l'ordre de multiplicité d'un élément x de $(M \cap_m N)$ est le minimum des deux ordres de multiplicité de x dans M et N (c'est à dire $\forall x, (M \cap_m N)(x) = \min(M(x), N(x))$).

     Exemple : $\{0;0;1;2\} \cap_m \{0;2;2;2\}$ est égal à $\{0,2\}$.

   - **La différence** $M \setminus_m N$: l'ordre de multiplicité d'un élément x de $M \setminus_m N$ est égal à $\max(0, M(x) - N(x))$.

     Exemple : $\{0;0;1;2\} \setminus_m \{0;2;2;2\}$ est égal à $\{0;1\}$.

3. Écrire les trois fonctions `union_l`, `intersection_l`, `difference_l` implémentant les opérations décrites ci-dessus.

   À présent, on souhaite tester l'égalité de deux multi-ensembles, rappelons que $\{0;0;1;2\} = \{0;2;1;0\}$. Une façon de procéder est de trier les listes représentant les multi-ensembles :

4. Écrire une fonction `insert` qui a pour arguments un élément x et une liste m supposée déjà triée et insère x dans la liste à la bonne place.

5. En déduire une fonction `sort` telle que `sort list` renvoie la liste `list` triée.

6. En déduire une fonction testant l'égalité de deux multi-ensembles.

7. Discuter des avantages et des inconvénients de cette représentation.

Par une application

Une définition plus formelle des multi-ensembles est la suivante : un multi-ensemble fini M d'éléments de D est une application de D dans l'ensemble des entiers naturels $\mathbb{N}$. À chaque élément, on associe sa multiplicité. En Caml, nous utiliserons le type suivant :

```
type 'a multiset = 'a -> int
```

Exemple : Le multi-ensemble $M = \{0;0;2;2;2;3\}$ s'écrit alors :

```
let (m1  : int multiset) = function
| 0 -> 2
| 1 -> 0
| 2 -> 3
| 3 -> 1
| _ -> 0  ; ;
```

1. Définir le multi-ensemble vide `empty_m`.
2. Écrire une fonction `ajoute_m` qui ajoute un élément à un multiensemble.
3. Déduire des deux questions précédentes, une fonction `multi_of_list` qui convertit une liste en multi-ensemble.
4. Écrire une fonction `enleve_un_m` qui supprime une occurrence d'un élément donné x dans un mult-ensemble m.
5. Écrire une fonction `enleve_tous_m` qui supprime toutes les occurrences d'un élément donné x dans un multi-ensemble m.
6. Écrire des fonctions `somme_m`, `intersection_m`, `union_m` et `difference_m` qui réalisent les opérations décrites dans la première partie.
7. Est-il possible d'écrire une fonction qui convertit un multi-ensemble de type `'a multiset` en un multi-ensemble de type `'a list` ? Avec ce type `'a multiset`, est-il possible d'écrire une fonction qui teste l'égalité de deux multi-ensembles ? Discuter des avantages et des inconvénients de ces deux représentations.

Par un couple

Dans cette partie, nous nous intéressons uniquement aux multi-ensembles finis sur des entiers naturels, et nous proposons de les représenter par le type suivant :

```
type multiset_paire = int * (int -> int)
```

Un multi-ensemble est maintenant représenté comme une paire, la première composante est un majorant des éléments du multi-ensemble, et la deuxième composante est un multi-ensemble au sens de la partie précédente.

Exemple : Le multi-ensemble $M = \{0;0;2;2;2;3\}$ peut s'écrire `(3,m1)`, mais il peut aussi s'écrire `(4,m1)`, puisque 3 et 4 sont tous les deux des majorants des éléments de M.

1. Adapter les fonctions de la partie précédente à ce nouveau type, en particulier `empty_mp` et `ajoute_mp`.
2. Écrire une fonction `liste_to_multipaire` qui convertit une liste en multi-ensemble et qui s'appuie sur `empty_mp` et `ajoute_mp`.
3. Écrire une fonction `multipaire_to_liste` qui convertit un multi-ensemble en liste (en imposant un ordre des éléments selon votre choix). Concevez, si nécessaire, des fonctions auxiliaires.
4. Écrire une fonction `egal_2` qui teste l'égalité de deux multi-ensembles.