Le paradigme Fonctionnel

Formation « enseignants de NSI »

Table des matières

I. Quelques rappels

I.A) Notion de paradigme

Extrait de wikipedia :

Un **paradigme** est — en épistémologie et dans les sciences humaines et sociales — une représentation du monde, une manière de voir les choses, un modèle cohérent du monde qui repose sur un fondement défini (matrice disciplinaire, modèle théorique, courant de pensée).

En informatique, on parle de paradigme pour caractériser le modèle sous-jacent à un langage. Il existe de nombreux types de paradigme : impératif, logique, fonctionnel, objet, réactif, … Certains langages sont mono-paradigmes : C est purement impératif, Haskell est purement fonctionnel, tandis que Prolog est purement logique. Mais souvent, les langages autorisent plusieurs paradigmes, même si un prédomine : Caml est essentiellement fonctionnel, même s'il possède des structures impératives ; Java est un langage impératif et objet, mais il propose des aspects fonctionnels.

Remarque : les paradigmes logiques et fonctionnels sont souvent regroupés dans une famille plus large, celle des paradigmes déclaratifs.

Le paradigme objet est un parradigme que je qualifie de « paradigme secondaire » : il « altère » un paradigme primaire : il existe ainsi des langages impératifs objets (Java, Python) et des langages fonctionnels objets (OCaml, Scala).

La taxonomie des paradigmes de programmation n'est pas figée, et la plupart des langages sont à cheval sur plusieurs cases.

I.B) Le paradigme impératif

Un **langage impératif** est un langage dont les **programmes** consistent principalement en une séquence d'instructions.

Une **instruction** est un ordre donné à l'ordinateur.

Une **variable** permet de stocker une valeur.

L'**affectation** est l'opération qui consiste à associer une information à une variable. La notion d'affectation est une notion que l'on trouve dans tous les langages impératifs. Le symbole peut varier d'un langage à l'autre (signe « = » en Python, Java ou C par exemple, signe «:= » en Pascal, etc.).

On appelle **état** d'un programme l'ensemble de la valeur de ses variables à un instant donné.

Une instruction permet de passer d'un état e_1 à un état e_2. Certaines instructions ne modifient pas la valeur de l'état, mais présentent des **effets de bord**, c'est-à-dire des modifications de l'environnement du programme (affichage sur un écran, écriture dans un fichier, envoi d'informations sur un réseau…).

On appelle **trace d'exécution** la succession des états du programme au cours de son exécution.

Exemples de langages informatiques impératifs déjà vus :

- Processing
- Python
- Shell
- Assembleur

Activité :

Illustration de la notion d'état et d'effet de bord (print…) avec Python Tutor

I.C) Rappels sur la notion de donnée

Les **données** sont l'objet même de l'existence de l'informatique (étymologiquement, informatique vient de la contraction de « information » et de « automatique » ; il s'agit donc de pouvoir traiter automatiquement des informations). Une donnée peut être vue comme une information structurée.

Les données sont **typées**. Les types classiques vus jusqu'à présents représentent des informations comme les nombres, les booléens, les textes. Les **fonctions** sont des éléments d'un autre ordre qui permettent d'appliquer des traitements sur les données.

Activité :

Illustrer des exemples de données et de fonctions dans un langage impératif

II. Le paradigme fonctionnel

II.A) Origine

Origine : travail d'Alonzo Church, en 1932, définissant le λ-calcul.

Idée première : les fonctions sont des données comme les autres que l'on doit pouvoir manipuler, passer en paramètre à d'autres fonctions, etc.

Motivations :

- les traitements à effectuer ne sont pas toujours connus quand on écrit le code (exemple : bibliothèque Tkinter : les concepteurs de l'objet « bouton » ne peuvent pas savoir ce qu'il faudra faire au moment de cliquer sur le bouton
- il peut être souhaitable de vouloir appliquer des traitements sur des traitements ; c'est le cas de l'opérateur o en mathématiques : il s'agit d'une fonction prenant 2 fonctions en paramètres et renvoyant une nouvelle fonction: h = f o g pourrait être noté h = o(f, g).

Conséquence : il faut un moyen de définir une fonction. Tout comme une donnée standard, une fonction n'a pas de nom en soi. Le « nom », c'est en fait le nom de la variable dans laquelle la fonction est stockée.

Exemple en math : f : x ↦ x + 1

- f : nom de la fonction
- x ↦ x + 1 : la fonction en elle-même

Notations introduites par Church :

- Définition d'une fonction : λ x.(x+1)
- Application d'une fonction : (λ x.(x+1))(2)

Exemples d'utilisation :

- Passage d'une fonction en paramètre : λ(f,x).(f(x+1))
- Définition d'une fonction renvoyant une fonction : λ x. (λ y .(x + y))
  - (λ x. (λ y .(x + y)))(1) = λ y .(1 + y)

Idée seconde : un programme est une composition de fonctions. Il n'y a plus de notion de séquentialité, et donc plus de notion d'état.

Corollaire 1 : il n'y a plus de « variable » au sens informatique classique du terme, c'est-à-dire un contenant dont le contenu (la valeur varie). Les données sont des objets immuables ; on ne fait que construire de nouvelles données à partir d'autres, comme en mathématiques.

- *Avantage* : plus de soucis de « partage de référence »
- *Inconvénient* : risque de perte de place/de temps si implantation naïve en cas de modification d'une donnée de grande taille

Corollaire 2 : il n'y a pas de boucle. On ne peut répéter que par appel de fonction… → récursivité

Corollaire 3 : il n'y a pas de « if then else » classique. Le « if then else » des langages fonctionnels et un « if then else » à valeur d'expression, comme l'opérateur ternaire de C, Java ou Python par exemple.

II.B) Intérêt du paradigme fonctionnel

Pouvoir passer des traitements en paramètres :

Illustration du traitement à associer au clic sur un bouton dans une IHM

Programme python avec un bouton pour dire bonjour :

```
from tkinter import *
from tkinter.messagebox import *

fenetre = Tk()
def saluer():
  showwarning("Salutation", "Bonjour tout le monde !")

bonjour = Button(fenetre, text="bonjour", command=saluer)
bonjour.pack()

fenetre.mainloop()
```

Même chose avec une méthode :

```
from tkinter import *
from tkinter.messagebox import *

fenetre = Tk()

quitter = Button(fenetre, text="quitter", command=fenetre.quit)
quitter.pack()

fenetre.mainloop()
```

Ou avec une lambda-expression :

```
from tkinter import *
from tkinter.messagebox import *

fenetre = Tk()

aurevoir = Button(fenetre, text="au revoir",
                  command=lambda: showwarning("Bye", "Au revoir"))
aurevoir.pack()

fenetre.mainloop()
```

Des programmes plus sûrs :

illustrer le problème du changement de valeur d'une donnée gérée par référence dans un langage impératif

II.C) Les langages fonctionnels

Suite aux différents intérêts du paradigme fonctionnel, des langages permettant de le mettre en œuvre ont rapidement vu leur apparition :

- **LISP** (LISt Processing), 1958 : Le premier, très longtemps utilisé, notamment en IA. À la base lent, mais des processeurs dédiés ont été développés ; premier langage ayant introduit la notion de machine virtuelle. Inconvénient souvent décrié : trop de parenthèses rendant le code peu agréable à lire ;
- **SML** (Standard Meta Language), 1983 : Le descendant le plus connu de **ML** (1970)
- **CAML** (1985) puis **OCAML**(1996) : descendant français (Inria) de ML
- **Haskell** (1990), nommé en hommage à Haskell Curry : langage fonctionnel particulièrement efficace et sûr

Caractéristiques communes :

- Les fonctions sont des données comme les autres, qu'il est possible de définir via un opérateur lambda (la syntaxe varie suivant les langages) ;
- Il est possible de ne définir des programmes que comme composition de fonctions (même si certains langages comme CAML, permettent d'inclure des concepts impératifs) ;
- Les données sont immuables ;
- Les langages sont fortement typés. Cependant, les types ne sont pas déclarés mais inférés ;

- Ces langages utilisent l'**évaluation paresseuse** : une donnée n'est calculée que lorsque c'est nécessaire, ce qui permet souvent de réduire fortement le nombre de calculs à effectuer.
- Les listes sont des structures de données fondamentales dans les langages fonctionnels. On peut leur donner de nombreux sens différents : collection de valeurs, indéterminisme par exemple.

II.D) Et au-delà ?

Certains aspects des langages fonctionnels sont très pratiques (fonction = donnée notamment). Mais le principe général (programme = composition de fonctions) ne convient pas forcément à tout le monde dans tous les cas

→ Hybridations de paradigmes :

- Des langages fonctionnels proposent des constructions impératives (CAML/OCAML notamment)
- Des langages impératifs proposent des aspects fonctionnels (Python, Java, Javascript, etc.)

De nos jours, une proportion importante de développement est faite en utilisant des aspects fonctionnels des langages impératifs.

III. Premiers pas dans un langage fonctionnel

III.A)Petite intro pratique

III.A.1) Haskell

Interpréteur :

- si `ghci` (l'interpréteur haskell) est installé, le lancer depuis un shell

Utilisation : mode REPL (Read-Evaluate-Process Loop) :
Soit on tape tout dans l'interpréteur (notamment pour les premiers exemples)
Soit on tape tout dans un fichier suffixé par « .hs », puis, en étant dans le répertoire du fichier, on lance l'interpréteur (`ghci`) puis on charge le fichier avec un `:l nomFichier` (sans le suffixe « .hs »).[1]

- en ligne :
  - Solution 1 : https://www.tutorialspoint.com/compile_haskell_online.php

utilisation de compile_haskell_online dans la suite :
pour ce qui doit être tapé dans l'interpréteur (style REPL) :
lancer l'interpréteur dans la moitié droite de la fenêtre en y tapant : « ghci »
bizarrement… il faut maintenant souvent faire précéder la ligne tapée d'un espace. Si une erreur mentionnant « bash » est affichée, il faut relancer l'interpréteur
quitter avec « :quit » (ne pas oublier le « : »)
pour ce qui doit être tapé dans un fichier :
en mode connecté
taper le code à gauche
sauver le projet

1 On peut aussi lancer directement l'interpréteur avec le nom du fichier en paramètre

lancer ghci à droite
charger le fichier : « :l main »
en mode anonyme
à gauche, faire précéder le code de la déclaration de la fonction affiche :
affiche x = putStr (show x)
faire suivre le code de la déclaration de la fonction main :
`main = putStr (affiche expressionAAfficher)`

- Solution 2 : https://replit.com/~

utilisation de repl.it :
nécessite de se créer un compte
cliquer sur « `+ Create Repl` », puis choisir « `Haskell` »
Pour le mode REPL :
à gauche, cliquer sur « shell », puis lancer « ghci »
Pour le mode « fichier » :
taper le code à gauche, puis cliquer sur le bouton « ▶ `Run` »

Remarque : ne pas hésiter à expliquer ce qu'est un éditeur de texte

Activité :

1. Illustrer l'inférence de type
    1. Illustrer l'inférence de type sur des types simples :

```
:t True
True :: Bool
```

```
:t 'b'
'b' :: Char
```

    2. Illustrer l'inférence de type sur des types génériques :

       En Haskell, une liste ne peut contenir que des éléments d'un seul type. Il s'agit d'un **type générique**, c'est-à-dire d'un type *paramétré* par un autre type.

```
[True, False]
[True, False]:: [Bool]
```

```
:t ['h', 'e', 'l', 'l', 'o']
['h', 'e', 'l', 'l', 'o'] :: [Char]
:t "hello"
"hello" :: [Char]
```

    3. Illustrer l'inférence de type sur des « classes de type » et expliquer informellement :

```
:t 12
12 :: Num p => p
```

12 pourra être considéré comme étant de différents types, tant qu'il s'agit s'agit d'un type numérique. Il pourra donc, au besoin, être considéré comme un entier ou comme un flottant.

```
:t 3.14

3.14 :: Fractional p => p
```

3.14 pourra être considéré comme étant de différents types, tant qu'il s'agit s'agit d'un type numérique non entier.

2. Un nom ne peut changer de valeur

   Montrer que ce qui est possible dans l'interpréteur « par facilité » (a = 2, puis a = 4) ne l'est pas dans un fichier

3. Appliquer une fonction

Fonction à 1 paramètre

```
head [4, 5, 6]

4
```

```
succ 4

5
```

Fonctions à 2 paramètres ou plus :

```
mod 5 2

1
```

Place des parenthèses dans la composition de fonctions :

```
mod 5 (succ 2)
```

autre notation :

```
mod 5 $ succ 2
```

Les opérateurs sont des fonctions utilisables en notation préfixe :

```
2 + 4
(+) 2 4
```

Typage des fonctions

```
:t not

not :: Bool -> Bool
```

```
:t (&&)
```

```
(&&) : Bool -> Bool -> Bool
```

```
:t head
[a] -> a
```

```
:t succ
succ :: Enum a => a -> a
```

☞ : successeur peut être appliqué à tout type « énumérable »

```
:t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

☞ : « . », c'est la composition de fonctions

Exercices :

1. calculer la somme du successeur de 1 et du successeur de 2 en utilisant l'opérateur '+' :

   en notation infixe

   en notation préfixe

2. calculer la somme du successeur de 1, du successeur de 2 et du successeur de 3 en utilisant l'opérateur '+' :

   en notation infixe

   en notation préfixe

III.A.2) OCaml

☞ Un programme Ocaml peut être interprété ou compilé

Tester OCaml online : https://try.ocamlpro.com/

Pour des tests directs :

zone de saisie en bas à droite, puis ↵

Pour des choses plus longues :

chaque définition doit se terminer par « ;; »

éditeur à gauche, puis `ctrl` – `e`.

Le type est affiché dès qu'une expression est saisie :

```
true
: bool = true
```

☞ En OCaml, les constantes booléennes commencent par une minuscule

```
[1;2;3]
```

```
: int list = [1; 2; 3]
```

```
"hello"
```

```
: string = "hello"
```

En Ocaml, les chaînes de caractères ne sont pas des listes de caractères !

Fonction « tête » (fonction `hd`, définie dans le module `List`) :

```
List.hd
```

```
: 'a list -> 'a = <fun>
```

Exemple d'utilisation :

```
List.hd [4;7;8]
```

```
:int = 4
```

Fonction « queue » :

```
List.tl
```

```
: 'a list -> 'a list = <fun>
```

Opérateur modulo :

```
(mod)
```

```
: int -> int -> int = <fun>
```

☞ : comme en Haskell, les opérateurs peuvent être utilisés en mode « préfixe » en les mettant entre parenthèses

En OCaml, il est possible de définir plusieurs variables de même nom, y compris avec des types différents. La dernière « masque » les précédentes dans la suite (mais pas dans ce qui précède)

En OCaml, les opérateurs sur les entiers et les réels diffèrent :

```
(+)
```

```
: int -> int -> int = <fun>
```

```
(+.)
```

```
: float -> float -> float = <fun>
```

III.B)Les bases d'Haskell

III.B.1) Définition de fonction à 1 paramètre

Donner la fonction `doubler` et vérifier son type

```
doubler x
```

Montrer que l'on peut déclarer le type d'une fonction, ce qui permet éventuellement de le contraindre :

```
doubler :: Int -> Int
doubler x = x + x
```

Exercice : Faire écrire la fonction quadrupler (en interdisant la multiplication) :

a. de zéro

```
quadrupler x = x + x + x + x
```

b. en utilisant la composition de fonction avec la fonction doubler

```
quadrupler x = doubler (doubler x)
```

III.B.2) Fonction à plusieurs paramètres

Exemple :

```
somme x y = x + y
```

Exercice :

faire écrire la fonction produit

```
produit x y = x * y
```

III.B.3) Le `if` en Haskell

`if` = fonction

`if a then b else c` doit avoir un type, à savoir celui de `b` et de `c`. Donc `b` et `c` doivent toujours être définis (`else` obligatoire) et de même type.

Exemple :

```
parite n = if mod n 2 == 0 then "pair" else "impair"
```

Exercice :

faire écrire la fonction maxi :

```
maxi x y = if x < y then y else x
```

En Python :

```
def parite(n) :
  return "pair" if mod n 2 == 0 else "impair"
```

```
def maxi(x, y) :
  return y if x < y else x
```

III.B.4) Définition d'une fonction par filtrage

Préambule : définir une fonction par filtrage n'est pas possible (facilement[2]) directement dans l'interpréteur ; obligation de passer par un fichier.

Exemple :

```
zero 0 = True
zero n = False
```

En python ≥ 3.10 :

```
def zero(n) :
  match n :
    case 0 : return True
    case _ : return False
```

L'instruction `match` en Python :
correspond à une sorte de « `if` » multi-branche. Les branches, introduites par le mot-clef `case`, sont examinées les unes après les autres. C'est le code de la première branche dont le *motif* suivant le `case` est *compatible* avec l'*expression* suivant le `match` qui sera exécuté.
Interprétation de « compatible » :
- si le motif est une valeur, comprendre « égalité ;
- si le motif est une variable « fraiche » (non déclarée préalablement) : fonctionne systématiquement, et la variable prend la valeur de l'expression (« _ » est une variable comme une autre, mais avec la signification, par convention de « la valeur importe peu)
- si le motif est un tuple, ou liste contenant des variables fraiches, donc la structure correspond à celle de l'expression, permet de faire une affectation terme à terme

III.B.5) Récursivité

Suite définie par récurrence

Exemples :

version 1:

```
u 0 = 1
u n = 2 * u (n - 1) + 3
```

N.B. 1 : c'est une définition par filtrage, donc dans un fichier forcément

N.B. 2 : insister sur le lien direct avec la définition mathématique :

$$\begin{cases} u_0 = 1 \\ u_n = 2 \cdot u_{n-1} + 3, \forall\, n \geq 1 \end{cases}$$

version 2 :

```
u n = if n == 0 then 1 else 2 * u (n - 1) + 3
```

En Python, version 1 :

2 En fait, c'est possible en tapant une première ligne : «:{ », puis une dernière ligne : «:} »

```
def u(n) :
  match n  :
    case 0 : return 1
    case _ : return 2 * u(n - 1) + 3
```

En Python, version 2 :

```
def u(n) :
  return 1 if n == 0 else 2 * u(n - 1) + 3
```

Exercice :

donner les 2 versions pour la suite définie par récurrence suivante :

$$\begin{cases} v_0 = 2 \\ v_n = 3 \cdot u_{n-1} - 1, \forall n \geq 1 \end{cases}$$

III.B.6) Récursivité pour répéter

Rappel : langage fonctionnel = pas d'état, donc pas de boucle

répéter → récursivité

Exemple : calculer la somme des nombres de 0 à n

Exemple avec explication :

somme 4 = 0 + 1 + 2 + 3 + 4 = (0 + 1 + 2 + 3) + 4 = somme 3 + 4

donc somme n = somme (n-1) + n mais… pour n > 0

et donc rajouter cas de base :

```
            somme 0 = 0
            somme n = n + somme (n-1)
```

En python :

```
def somme(n):
  match n:
    case 0: return 0
    case _: return n + somme(n - 1)
```

Exercice 1 : produit des nombres de 1 à n

Attention aux 2 pièges :

cas de base pour 1 et pas 0

valeur pour 1 : 1 et pas 0

Exercice 2 : Fibonacci

Rappeler la définition :

$$\begin{cases} fibo_0 = 0 \\ fibo_1 = 1 \\ fibo_n = fibo_{n-2} + fibo_{n-1}, \forall n \geq 2 \end{cases}$$

Montrer l'inefficacité avec fibo 34 et suivants, et expliquer ce qu'il se passe

Expliquer qu'on peut faire efficace, mais pas l'objet ici

Exercice 3 : Les entiers de Peano en Haskell

Axiomatique : on ne dispose que de 0, de `pred` et de `succ`

Montrer comment on construit l'addition :

x + (y + 1) = (x + y) + 1

x + succ(y) = succ (x + y)

Posons Y = succ(y). On obtient :

x + Y = succ (x + pred(y))

Avec le cas de base, en Haskell :

```
plus x 0 = x
plus x y = succ (plus x (pred y))
```

En Python :

```
def succ(n):
  return n + 1

def pred(n):
  assert n > 0
  return n - 1

def plus(x, y):
  match y :
    case 0: return x
    case n: return succ (plus(x, pred(y)))
```

Faire construire le produit :

x * (y +1) = x * y + x

x * succ(y) = x * y + x

x * Y = x * pred(Y) + x

Avec le cas de base, en Haskell :

```
fois x 0 = 0
fois x y = plus (fois x (pred y)) x
```

En Python :

```
def fois(x, y):
  match y:
    case 0: return 0
    case n: return plus(fois(x, pred(y)), x)
```

Faire construire la puissance :

x^(y+1) = x^y * x

x^succ(y) = x^y * x

x^Y = x^pred(Y) * x

Avec le cas de base, en Haskell :

```
     puiss x 0 = 1
     puiss x y = fois (puiss x (pred y)) x
```

En Python :

```
def puiss(x, y):
  match y:
    case 0: return 1
    case n: return fois(puiss(x, pred(y)), x)
```

III.B.7) Les listes en Haskell

Liste vide, liste définie en extension, éventuellement liste définie en compréhension

`element:liste` : Adjonction en tête

++ : concaténation de liste

élément à l'indice n avec !!

`head` : renvoie le premier élément d'une liste

`tail` : renvoie les n-1 derniers éléments d'une liste de taille n

`length` : renvoie la longueur de la liste

traitements récursifs sur les listes[3] (somme des éléments d'une liste, ajouter 1, doubler les éléments d'une liste).

Définition de head et tail en python (utile pour la suite) :

```
def head(liste) :
  return liste[0]

def tail(liste) :
  return liste[1:]
```

Exemple : ajouter 1 à tous les éléments de la liste

```
     incrementerListe [] = []
     incrementerListe liste = (head liste + 1) : incrementer (tail
liste)
```

En Python :

```
def incrementer_liste(liste):
  match liste:
    case []: return []
    case autre: return [head(autre)+1]+incrementer_liste(tail(autre))
```

Exercice : doubler tous les éléments de la liste

```
     doublerListe [] = []
     doublerListe liste = (head liste * 2) : doublerListe (tail
liste)
```

3 Je n'utilise pas le filtrage sur les listes pour donner l'habitude de la composition de fonctions

En Python :

```
def doubler_liste(liste):
  match liste:
    case []: return []
    case autre: return [head(liste) * 2] + doubler_liste(tail(liste))
```

Exercice : les tours de Hanoi

```
hanoi 1 depart arrivee intermediaire = [(depart, arrivee)]
```

```
hanoi n depart arrivee intermediaire = hanoi (n-1) depart intermediaire arrivee ++ hanoi 1 depart arrivee intermediaire ++ hanoi (n-1) intermediaire arrivee depart
```

En Python :

```
def hanoi(n, depart, arrivee, intermediaire):
  match n:
    case 1: return [(depart, arrivee)]
    case autre: return hanoi(n-1, depart, intermediaire, arrivee) +
hanoi(1, depart, arrivee, intermediaire) + hanoi(n-1, intermediaire,
arrivee, depart)
```

III.C) Les bases de OCaml

III.C.1) Définir une fonction

```
let doubler x = x + x ;;

val doubler: int -> int = <fun>

let somme x y = x + y ;;

val somme: int -> int -> int = <fun>

let parite x = if x mod 2 == 0 then "pair" else "impair"

val parite: int -> string = <fun>
```

III.C.2) Fonction définie par filtrage

```
let zero n =
  match n with
  | 0 -> true
  | _ -> false;;

val zero: int -> bool = <fun>
```

ou encore :

```
let zero = function
  | 0 -> true
  | _ -> false;;
```

III.C.3) Fonctions récursives en OCaml

Il faut obligatoirement rajouter le mot-clef `rec` dans la définition de la fonction :

```
let rec u n = if n = 0 then 1 else 2 * u (n - 1) + 3;;
```

Ou :

```
let rec u n =
  match n with
  | 0 -> 1
  | _ -> 2 * (u (n - 1)) + 3;;
```

III.C.4) Peano en OCaml

☞ : Les fonctions `pred` et `succ` existent en OCaml

```
let rec plus x y =
  match y with
  | 0 -> x
  | _ -> succ (plus x (pred y));;

let rec fois x y =
  match y with
  | 0 -> 0
  | _ -> plus x (fois x (pred y));;

let rec puiss x y =
  match y with
  | 0 -> 1
  | _ -> fois x (puiss x (pred y));;
```

III.C.5) Listes en OCaml

Déclaration en extension : séparateur = « ; »

```
[1;2]
```

Ajout en tête : opérateur = « :: »

```
1::[2]
```

La plupart des fonctions sur les listes sont définies dans le module List

Récupérer l'élément à l'indice n :

```
List.nth [4;5;6] 1
```

Fonctions récursives sur les listes :

```
let rec incrementer_liste  = function
  | [] -> []
  | tete::queue ->  (tete + 1) :: (incrementer_liste queue);;
```

III.C.6) Hanoi en OCaml

☞ : La concaténation de listes se note avec l'opérateur @

```
let rec hanoi n depart arrivee intermediaire =
  match n with
  | 1 -> [(depart, arrivee)]
  | _ -> hanoi (n - 1) depart intermediaire arrivee @
         hanoi 1 depart arrivee intermediaire @
         hanoi (n-1) intermediaire arrivee depart;;
```

III.D)Fonctions d'ordre supérieur

III.D.1) Pour aller plus loin sur les listes

- `last` : renvoie le denier élément d'une liste
- `init`: renvoie les *n-1* premiers éléments d'une liste de taille *n*
- `take` : renvoie les *n* premiers éléments d'une liste
- `drop` : « supprime » les *n* premiers éléments d'une liste

Donner des exemples

Exercices : faire implanter récursivement

`last` :

```
     dernier [x] = x
     dernier liste = dernier (tail liste)
```

En Python :

```
def dernier(liste):
  match liste:
    case [x]: return x
    case autre: return dernier(tail(liste))
```

`init` :

```
     debut [x] = []
     debut liste = (head liste):debut (tail liste)
```

En Python :

```
def init(liste):
  match liste:
    case [x]: return []
    case autre: return [head liste] + init(tail(liste))
```

`take`, version « simple » :

```
           prendre 0 liste = []
           prendre n liste = (head liste): prendre (n-1) (tail liste)
```

prendre, version autorisant un nombre supérieur à la longueur de la liste (comme `take`) :

```
           prendre 0 liste = []
           prendre n [] = []
           prendre n liste = (head liste): prendre (n-1) (tail liste)
```

En python :

```
def prendre(n, liste):
  match n:
    case 0: return []
    case n:
      match liste:
        case []: return []
        case autre: return [head(liste)] + prendre(n-1, tail(liste))
```

III.D.2) Les fonctions d'ordre supérieur en Haskell

Introduction :

Commencer par montrer que les traitements récursifs vus précédemment (ajouter 1 à tous les éléments, doubler les éléments) sur les listes sont similaires, à l'opération unitaire près, qui peut donc devenir un paramètre d'une fonction plus générale, et créer d'abord son propre « map » avant de montrer celui défini de base. Passer d'abord par des fonctions nommées avant d'introduire les lambda-expressions.

```
      ajouter1 x = x + 1

      doubler x = 2 * x

      appliquer traitement [] = []
      appliquer traitement liste = traitement (head liste):appliquer
traitement (tail liste)

      appliquer ajouter1 [4, 5, 8]
      appliquer doubler [4, 5, 8]
```

En Python :

```
def ajouter_1(x): return x + 1

def doubler(x): return 2 * x

def appliquer(traitement, liste):
  match liste:
    case []: return []
    case autre: return [traitement(head(liste))] +
appliquer(traitement, tail(liste))

appliquer(ajouter_1, [4, 5, 9])
appliquer(doubler, [4, 5, 9])
```

Définition d'une lambda-expression en Haskell :

```
      ajouter2 = \x -> x+2
      appliquer ajouter2 [4, 5, 8]
```

Puis :

```
      appliquer (\x -> x+2) [4, 5, 8]
```

En Python :

```
ajouter_2 = lambda x: x + 2
appliquer(ajouter_2, [4, 5, 8])

appliquer(lambda x: x + 2, [4, 5, 8])
```

Et enfin :

```
      map (\x -> x+2) [4, 5, 8]
```

En Python :

```
map(lambda x: x + 2, [4, 5, 8])
```

Sortie :

```
<map object at 0x7fcd6c644c10>
```

Explication :

Python met en œuvre le mécanisme d'**évaluation paresseuse** : l'application de la fonction n'est en fait pas réalisée ; elle ne sera effectivement réalisée que lorsque cela sera nécessaire, ce qui permettra peut-être de réduire le travail. `map` est en fait une classe, et ici, on fait juste appel au constructeur de la classe en question.

Avoir dans un premier temps une sortie « intéressante » :

```
list(map(lambda x: x + 2, [4, 5, 8]))
```

Sortie :

```
[6, 7, 10]
```

Exemple d'intérêt de l'évaluation paresseuse :

```
liste = map(lambda x: x  + 2, range(1000000000000000))
```

La réponse est instantanée : pas de calcul de fait, pas de place prise en mémoire (range peut être vu comme une définition paresseuse d'une séquence)

Si on redéfinit « `prendre` » de façon paresseuse :

```
class prendre:
    def __init__(self, nombre, iterable):
        self.nombre = nombre
        self.iterable = iterable
        self.pris = 0
        self.iter = iter(iterable)

    def __iter__(self):
        return self

    def __next__(self):
        self.pris += 1
        if self.pris > self.nombre:
            raise StopIteration
        return next(self.iter)
```

On peut alors sans problème effectuer le calcul suivant :

```
list(prendre(10, map(lambda x: x + 2, range(1000000000000000))))
```

Le résultat est instantané ; seuls les premiers éléments ont effectivement été calculés

Filter :

renvoyer la liste des éléments d'une liste vérifiant une certaine propriété

Détailler le typage

Exemple : garder les éléments pairs d'une liste

```
filter (\x -> mod x 2 == 0) [3, 4, 7, 8, 12]
```

En Python :

```
filter(lambda x: x % 2 == 0, [3, 4, 7, 8, 12])
```

Sortie :

```
<filter object at 0x7f256bdd4970>
```

filter fonctionne comme map !

Exercices :

garder les nombres supérieurs à 10

implanter son propre filter

III.D.3) Fonctions d'ordre supérieur et λ-expressions en OCaml

Définition d'une lambda-expression avec le mot-clef fun :

```
let f = fun x y -> x + y
```

Les fonctions `map` et `filter` sont définies dans le module `List` :

```
List.map
```

```
('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
List.filter
```

```
('a -> bool) -> 'a list -> 'a list = <fun>
```

III.D.4) Forme currifyiée ; interprétation [Pour aller plus loin]

Partir de la fonction :

```
somme x y = x + y
```

revenir sur son type : `Num a => a -> a -> a` (en expliquant qu'on peut simplifier, pour l'interprétation en `Num -> Num -> Num`)

Définir la variable myst ainsi :

```
myst = somme 2
```

La faire tester, analyser son type (`Num -> Num` pour simplifier)

Expliquer alors que le type de somme est à interpréter : `Num -> (Num -> Num)`, que `myst` est ce qu'on appelle une **application partielle**, et qu'on aurait aussi pu définir la fonction somme ainsi :

```
somme = \x -> (\y -> x + y)
```

En Python :

```
def somme(x) :
  return lambda y : x + y
```

la variable « x » dans la lambda-expression correspond à la valeur du paramètre de somme lors de son appel ; c'est ce qu'on appelle la « fermeture » (closure). On peut ainsi définir :

```
somme3 = somme(3)
somme5 = somme(5)
```

```
somme3(2)
```

```
5
```

```
somme5(2)
```

```
7
```

chaque fonction renvoyée par somme a son propre x

III.D.5) Pliage : foldl, foldl1

Exercices introductifs :

Somme des éléments d'une liste :

```
somme [] = 0
somme liste = head liste + somme (tail liste)
```

Ou encore :

```
somme [] = 0
somme liste = (+) (head liste) (somme (tail liste))
```

En Python :

```
def somme(x, y): return x + y

def somme_liste(liste):
  match liste:
    case []: return 0
    case autre: return somme(head(liste), somme_liste (tail(liste)))
```

Produit de éléments d'une liste :

```
produit [] = 1
produit liste = head liste * produit (tail liste)
```

Ou encore :

```
produit [] = 1
produit liste = (*) (head liste) (produit (tail liste))
```

Remarque : on peut discuter du produit par défaut à 1 sur la liste vide, mais on peut l'admettre.

En Python :

```
def produit(x, y): return x * y

def produit_liste(liste):
  match liste:
    case []: return 1
    case autre: return produit(head(liste), produit_liste
(tail(liste)))
```

Concaténation des chaînes d'une liste :

```
           concat [] = []
           concat liste = head liste ++ concat (tail liste)
```

Ou encore :

```
           concat [] = []
           concat liste = (++) (head liste) ++ (concat (tail list)
```

En Python :

```
def joindre (liste1, liste2) : return liste1 + liste2

def concat(liste):
  match liste:
    case []: return []
    case autre: return joindre(head(liste), concat (tail(liste)))
```

Somme des longueurs des mots d'une liste :

```
           longueur [] = 0
           longueur liste = length (head liste) + longueur (tail
liste)
```

Ou encore :

```
           longueur [] = 0
           longueur liste = (+) (length (head liste))  (longueur
(tail liste))
```

En Python :

```
def plus(x, y) : return x + y

def longueur(liste) :
  match liste :
    case [] : return 0
    case autre : return plus(len(head(liste)), longueur(tail(liste)))
```

Forme générale :

`fonction liste_vide = `*`init`*

`fonction liste_non_vide = `*`fusionneur`* `tête_de_liste (fonction queue_liste)`

Proposition d'implantation :

```
     pliage fusionneur [] init = init
     pliage fusionneur liste init = fusionneur (head liste) (pliage
fusionneur (tail liste) init)
```

En Python :

```
def pliage(fusionneur, liste,init):
  match liste:
    case []: return init
    case autre: return fusionneur(head(liste), pliage(fusionneur,
tail(liste), init))
```

Exemple d'utilisation pour la somme :

```
     pliage (\x -> \y -> (x+y)) 0 [4, 6, 10]
```

En Python :

```
pliage(lambda x, y: x + y, [4, 6, 10], 0)
```

Exercices :

utilisation pour le produit :

```
     pliage (\x -> \y -> (x*y)) 1 [4, 6, 10]
```

utilisation pour la concaténation :

```
     pliage (\mot1 -> \mot2 -> (mot1 ++ mot2)) "" ["bonjour", "le",
"monde"]
```

utilisation pour la somme des longueurs des mots :

```
     pliage (\mot -> \longueur -> (length mot + longueur)) 0
["bonjour", "le", "monde"]
```

L'équivalent de la fonction `pliage` existe en Haskell, c'est `foldl`, mais l'ordre des paramètres n'est pas exactement le même :

```
:t foldl

foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

Interprétation :

`foldl` est plus générale : pas que sur les listes, mais sur les « `Foldable` ». Et les listes en sont.

`a` : le type des éléments de la liste

`b` : le type du résultat. Souvent, c'est le même (somme, produit, concat), mais pas toujours (somme des longueurs)

En Python :

L'équivalent de la fonction `pliage` existe en Python, c'est la fonction `reduce` de la bibliothèque `functools` :

```
from functools import reduce
reduce(lambda x, y: x + y, [4, 6, 10], 0)
```

En OCaml :

L'équivalent de la fonction `pliage` existe en OCaml, c'est la fonction fold_left du module List :

```
List.fold_left

('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Exemple :

```
List.fold_left (fun x y -> x + y) 0 [4; 6; 10];;
```

Exercice :

reprendre les exemples précédents avec `foldl`.

```
     foldl (\x -> \y -> x + y) 0 [4, 6, 10]
     foldl (\x -> \y -> x * y) 1 [4, 6, 10]
     foldl (\x -> \y -> x ++ y) "" ["Bonjour", "le", "monde"]
     foldl (\lg -> \mot -> lg + length mot) 0 ["Bonjour", "le",
"monde"]
```

Cours :

Dans beaucoup de cas, quand le type du résultat est le même que le type des éléments, on peut utiliser directement le premier élément de la liste comme valeur de départ → pliage1 :

```
     pliage1 fusionneur [x] = x
     pliage1 fusionneur liste = fusionneur (head liste) (pliage1
fusionneur (tail liste))
```

Exercice :

ré-implanter les 3 premiers exemples précédents avec `pliage1`.

```
     pliage1 (\x -> \y -> x + y) [4, 6, 10]
     pliage1 (\x -> \y -> x * y) [4, 6, 10]
     pliage1 (\x -> \y -> x ++ y) ["Bonjour", "le", "monde"]
```

Rmq : le dernier exemple ne peut pas être implanté ainsi puisque le type du résultat (`int`) est différent du type des éléments de la liste (chaîne de caractères).

Cours : l'équivalent de pliage1 existe en Haskell, c'est `foldl1`.

Exercice :

même chose que précédemment avec `foldl1`.

```
     foldl1 (\x -> \y -> x + y) [4, 6, 10]
     foldl1 (\x -> \y -> x * y) [4, 6, 10]
     foldl1 (\x -> \y -> x ++ y) ["Bonjour", "le", "monde"]
```

En Python :

le dernier paramètre de reduce est optionnel ; sans lui, on a le comportement de `foldl1`.

```
reduce(lambda x, y: x+y, [4, 6, 10])
reduce(lambda x, y: x*y, [4, 6, 10])
reduce(lambda x, y: x+y, ["Bonjour", "le", "monde"])
```

Annexes : pour aller plus loin

IV. Le type Maybe

Motivation: exemple de la division entière

Comment traiter de manière sûre et élégante le fait qu'une division puisse mener à une erreur (division par zéro) ?

Une valeur par défaut ?

On risque de ne pas se rendre compte qu'il y a eu erreur

Des « if » à n'en plus finir ?

Code particulièrement illisible

Des exceptions ?

Incompatible avec un modèle fonctionnel

Solution : le type `Maybe` : une valeur complètement spécifique pour les cas d'erreur

Constructeurs: `Just` et `Nothing`

Exemple d'utilisation :

```
diviser x 0 = Nothing
diviser x y = Just (x / y)
```

Et par la suite, encore des « ifs » ?

non : on applique des traitements à un « `Maybe` ». Si le « `Maybe` » vaut `Nothing`, renvoie `Nothing`.

Exemples :

```
fmap (\x -> x + 1) (diviser 4 5)
fmap (\x -> x + 1) (diviser 4 0)
```

V. Retour sur la notion de classe de type

Principe :

- Une classe de type spécifie un ensemble de fonctions paramétrées par un type.
- Pour qu'un type implante une classe de type, il faut :
  - Qu'il le déclare
  - Qu'une implantation de chaque fonction de la classe de type soit proposée

Exemple : la classe de type Eq :

taper `:i Eq` pour avoir les infos suivantes :

```
class Eq a where
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
[...]
```

Explications :

- un type `a` qui déclare implanter la classe `Eq` doit fournir des fonctions pour `(==)` et `(/=)`.
- si `a` est un type implantant la classe `Eq`, alors il doit être possible d'appliquer la fonction `(==)` sur 2 données de type `a`. Cela donnera un résultat de type `Bool`.

Implantation d'une classe de type :

En poursuivant l'affichage de `:i Eq`, on obtient notamment :

```
instance Eq a => Eq [a] -- Defined in 'GHC.Classes'
instance Eq Int -- Defined in 'GHC.Classes'
instance Eq Float -- Defined in 'GHC.Classes'
```

Cela signifie, par exemple (ligne en rouge) que le type `Int` implante `Eq`. Donc qu'il existe une version de la fonction `(==)` typée :

```
Int -> Int -> Bool
```

Illustration :

```
fonc = (==) (2::Int)
:t fonc
Int -> Bool
```

Remarque complémentaire :

la ligne en bleu indique que le type liste implante l'interface Eq, et que l'on peut donc utiliser les opérateurs d'égalité et de différence sur les listes

Sous-typage entre classes de type :

Il est possible de définir une classe de type `f` comme un sous-type d'une autre `m`. Cela signifie qu'un type implantant la classe de type `f` sera considéré comme implantant la classe de type `m`, et que toutes les méthodes de `m` devront être définies pour ce type.

Exemple :

```
:i Ord
class Eq a => Ord a where
compare :: a -> a -> Ordering
```

```
(<) :: a -> a -> Bool

(<=) :: a -> a -> Bool

(>) :: a -> a -> Bool

(>=) :: a -> a -> Bool

max :: a -> a -> a

min :: a -> a -> a
```

La ligne en rouge indique que tout type implantant la classe de type `Ord` implante également la classe de type `Eq`, et donc qu'il doit exister pour ce type les fonctions `(==)` et `(/=)` vues précédemment.

VI. La notion de foncteur

Concept :

Un foncteur est un type `f` encapsulant des données d'un autre type et permettant d'appliquer un traitement aux données encapsulées, et renvoyant les résultats du traitement encapsulé dans une nouvelle donnée de type `f`.

Le type `List` est un foncteur :

Une liste encapsule des données d'un certain type `a`. En appliquant une fonction de `a` dans `b` à tous les éléments d'une liste de `a`, on obtient une liste de `b`. C'est ce que fait la fonction `map` déjà étudiée.

Le type `Maybe` est un foncteur :

Un `Maybe a` encapsule des données d'un certain type `a`. En appliquant une fonction de `a` dans `b` à l'élément encapsulé dans un `Maybe a`, on obtient un `Maybe b`. C'est ce que fait la fonction `fmap` déjà étudiée.

Formalisation :

il existe une classe de type `Functor`, implantée par `List` et `Maybe` :

```
:i Functor

class Functor (f :: * -> *) where

fmap :: (a -> b) -> f a -> f b

 (<$) :: a -> f b -> f a

instance Functor [] -- Defined in ‘GHC.Base’

instance Functor Maybe -- Defined in ‘GHC.Base’
```

Interprétation :

1. un foncteur est un type paramétré par un autre type (`* -> *`) proposant la fonction `fmap` déjà vue sur les `Maybe`, et implanté par `map` sur les listes (mais on peut utiliser `fmap` sur les listes si on veut).
2. (**Exercice** : les laisser découvrir ce que fait l'opérateur `<$`) Les foncteurs proposent également une façon de construire un foncteur encapsulant une donnée (1er paramètre) grâce à l'opérateur (`<$`), suivant un contexte (2ème paramètre), que l'on peut interpréter ainsi :
   1. si le contexte correspond à une erreur (liste vide ou `Nothing`), on propage l'erreur :

      ```
      4 <$ []

      []

      4 <$ Nothing

      Nothing
      ```

   2. sinon, on renvoie la donnée encapsulée :

      ```
      4 <$ [1, 2, 3]

      [4, 4, 4]

      4 <$ Just 3

      Just 4
      ```

VII. Foncteurs applicatifs et monades

VII.A) Foncteurs applicatifs

Idée de base : Si on peut encapsuler des données dans des foncteurs, pourquoi ne pas encapsuler des traitements ?

→ Foncteur applicatif

Infos : :i Applicative

```
class Functor f => Applicative (f :: * -> *) where
 pure :: a -> f a
 (<*>) :: f (a -> b) -> f a -> f b
 GHC.Base.liftA2 :: (a -> b -> c) -> f a -> f b -> f c
 (*>) :: f a -> f b -> f b
 (<*) :: f a -> f b -> f a
instance Applicative [] -- Defined in ‘GHC.Base’
instance Applicative Maybe -- Defined in ‘GHC.Base’
```

On ne s'intéresse qu'à ce qui est en bleu :

Les foncteurs applicatifs sont un sous-type de foncteurs (c-à-d les foncteurs applicatifs sont des foncteurs)

Les types `List` et `Maybe` sont des foncteurs applicatifs

`pure` est la fonction générale d'encapsulation d'une donnée (par exemple, le `Just` du type `Maybe`)

Remarques :

sans contexte, pas moyen de savoir ce que construit `pure` :

```
:t pure 4
pure 4 :: (Applicative f, Num a) => f a
```

Mais avec un contexte, plus le choix :

```
length (pure 5)
1
(pure 5) :: Maybe Int
Just 5
(pure 5) :: [Int]
[5]
```

`<*>` permet d'appliquée une fonction encapsulé à une donnée encapsulée

Exemples :

```
Just (\x -> x + 1) <*> Just 4
Just 5
[\x -> x + 1, \x -> x + 2] <*> [10,20]
[11,21,12,22]
```

`*>` et `<*` permettent de propager un contexte

Laisser découvrir comment ça fonctionne

Exercice :

Énoncé

Une liste représente les marches où peut potentiellement se trouver une grenouille.

À chaque étape, la grenouille peut avant d'une ou 2 marches.

Utiliser la notion de foncteur applicatif pour calculer les positions possibles de la grenouille à l'étape suivante.

Solution

```
[\x -> x+1, \x -> x+2] <*> [positions à l'étape n]
```

Problème : Que faire si le calcul renvoie déjà un foncteur applicatif ?

Exemple 1 :

```
[\x -> [x+1, x+2]] <*> [positions à l'étape n]
```

→ liste de listes, donc non chaînable :

```
[\x -> [x+1, x+2]] <*> [\x -> [x+1, x+2]] <*> [positions à l'étape n]
```

→ erreur

Exemple 2 :

```
diviser 5 (diviser 4 2)
```

→ erreur

VII.B) Les Monades

Reprenons le cas précédent :

```
\x -> [x+1, x+2]    : Int -> [Int]
                    : Int -> List[Int]
fmap : Functor f => (a -> b) -> f a -> f b
```

Dans le cas des listees :

```
fmap : (a -> b) -> List a -> List b
```

Et donc si `b` est de la forme `List c`, on a :

```
fmap : (a -> [c]) -> [a] -> [[c]]
```

On voudrait donc :

```
nouvelle-fonction : (a -> [c]) -> [a] -> [c]
```

Ou de façon plus générale :

```
(a -> m c) -> m a -> m c
```

C'est le **concept de Monade**, où la nouvelle fonction évoquée ci-dessus correspond à l'opérateur >>=, à l'ordre des paramètres près :

`:m Control.Monad` (pour importer le module qui définit la notion de monade)

```
:i Monad
class Applicative m => Monad (m :: * -> *) where
 (>>=) :: m a -> (a -> m b) -> m b
```

```
 (>>) :: m a -> m b -> m b
 return :: a -> m a
 fail :: String -> m a
instance Monad [] -- Defined in ‘GHC.Base’
instance Monad Maybe -- Defined in ‘GHC.Base’
```

Explications :

Une monade est d’abord un foncteur applicatif

L’opérateur >>= permet de faire ce que l’on voulait

L’opérateur >> permet de « propager un contexte »

La fonction `return` est celle qui permet d’encapsuler une donner dans une monade (le `pure` des foncteurs applicatifs)

Le type `List` et le type `Maybe` sont des monades

Exemples d’utilisation :

```
[2, 4, 7] >>= \x -> [x+1, x+2]
[3,4,5,6,8,9]


moitie x = if mod x 2 == 0 then Just (div x 2) else Nothing
tiers x = if mod x 3 == 0 then Just (div x 3) else Nothing
Just 12 >>= moitie
Just 6
Just 9 >>= moitie
Nothing
Just 12 >>= moitie >>= tiers
Just 2
Just 9 >>= moitie >>= tiers
Nothing
```

Exercice :

incrémenter et décrémenter des naturels sur l’intervalle [0,5] :

```
incrementer 5 = Nothing
incrementer n = Just (n+1)
decrementer 0 = Nothing
```

```
decrementer n = Just (n-1)
return 4 >>= incrementer >>= incrementer >>= decrementer
Nothing
```

VIII. Monoïdes

En mathématiques, un monoïde est un semi-groupe (ensemble avec une loi de composition interne - *lci* - associative) avec un élément neutre.

On retrouve cette notion dans les classes de type Semigroup et Monoid :

Semi-groupe :

```
:i Semigroup
class Semigroup a where
  (<>) :: a -> a -> a
```

<> est la loi de composition interne (on admet qu'elle est associative)

Monoïde :

```
:i Monoid
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
instance Monoid [a] -- Defined in ‘GHC.Base’
```

Un monoïde est un semi-groupe

Il a un élément neutre (`mempty`)

La loi associative est renommé `mappend`

puisqu'on peut traiter plein de donnée avec la loi de composition interne de façon associative, on peut appliquer cette loi à une liste de monoïdes pour obtenir le monoïde résultat avec `mconcat`.

Les listes sont des monoïdes (lci : concaténation)

: le type `Maybe` n'est pas un monoïde

Exemples :

```
mempty :: [Int]
[]
```

→ la liste vide est l'élément neutre pour la concaténation

```
mappend [4, 5, 6] [10, 11, 12]
[4, 5, 6, 10, 11, 12]
```

→ la lci correspond bien à la concaténation

```
mconcat [[1, 2], [7, 8, 9], [20]]
[1, 2, 7, 8, 9, 20]
```

→ on peut concaténer une liste de listes. Dans cet exemple, ne pas confondre la liste externe (type list imposé) et les listes internes (un monoïde parmi tant d'autres).

Intérêt des monoïdes :

On peut traiter en parallèles des données, puis fusionner le résultat par la lci.

Plus complexe :

Exemple :

avec un `:i Monoid`, on obtient entre autres :

```
instance (Monoid a, Monoid b) => Monoid (a, b)
```

Autrement dit, un couple de monoïde est un monoïde

Exercice : analyser ce que donne mappend et mconcat dans ce cas-là

```
mappend ([1,2], [3, 4]) ([5], [6, 7])
([1,2,5],[3,4,6,7])
mconcat [([1,2], [3, 4]), ([5], [6, 7]), ([10, 11, 12], [20, 30])]
([1,2,5,10,11,12],[3,4,6,7,20,30])
```

On observe une concaténation composante par composante