Principes de Programmation
Partiel 2025 corrigé

19 juin 2025

Cette partie du partiel dure 1 heures 30. Seule une fiche recto est autorisée.

Les exercices sont indépendants. Il est autorisé de sauter des questions, mais il faut **indiquer à quelle question vous répondez**.

Exercice 1 (Équations, estimation 45mn)**.**
On rappel la définition de la classe monoid

```
class Monoid m where
   mempty :: m
   (<>) :: m -> m -> m
--  À cette définition est associés 3 équations implicites :
--    mempty <> x = x
--    x <> mempty = x
--    (x<> y)<> z = x <> (y<>z)
```

De même la classe monade est défini sur des types paramétrés :

```
class Monad t where
   return :: t a
   (>>=) :: t a -> (a -> t b) -> t b
--  Avec trois équations dont :
--    u >>= return        =   u
--    (u >>= f) >>= g     =    u >>= (\x -> (f x) >>= g)
```

1. Rappelez l'équation de monade manquante.
   Correction: `(return x) >= f` $\simeq$ `fx`
2. Vérifiez que la dernière équation de monade fait sens en donnant le type associé à x, u, f et g.
   Correction:

   ```
   x :: a
   u :: ta
   f :: a -> tb
   g :: b -> tc
   ```

On utilise la structure suivante [1] qui prétend que si `m` est un monoid, alors le types de couples `(m, a)`, comme type paramétré par `a`, est une monade.

```
(Monoid m) => instance Monad (m,_) where
    return x       =   (mempty , x)
    (e,x) >>= f    =   (e <> (fst (f x)) , snd (f x))
```

1. Pas tout à fait correcte en Haskell où on devra utiliser un wraper supplémentaire.

où on utilise les fonctions

```
fst (x,y) = x
snd (x,y) = y
```

3. Vérifier la première loi des monades.
   Correction:

$$\begin{aligned}
\texttt{(e,x) >>= return} &\simeq \texttt{(e<> (fst (return x)), snd (return x))} \\
&\simeq \texttt{(e<> (fst (mempty,x)), snd (mempty,x))} \\
&\simeq \texttt{(e<> mempty, x)} \\
&\simeq \texttt{(e, x)}
\end{aligned}$$

4. Trouvez un exemple d'application.
   Correction: En prenant `m = string`, on obtient une monad qui enregistre des logs.

Exercice 2 (Programmation, estimation 45mn)**.** Les fonctions demandées doivent être écrites en style monadique (avec des binds ou une do-construction) lorsque c'est possible et pertinent. Vous pouvez toujours écrire des fonctions annexes.

1. `maybeToList :: Maybe a -> [a]`

   qui transforme un Nothing en la liste vide et un Just en un singleton.

   Correction:

```
maybeToList Nothing = []
maybeToList (Just x) = [x]
```

2. `updateList  :: (a -> Maybe b)   -> [a] -> [b]`
   `updateList2 :: (a -> Maybe [b]) -> [a] -> [b]`

   qui font un map et un bind, mais en enlevant des éléments à la volée (ceux pour lesquels la fonction donnée renvoit Nothing).

   Correction: Quelques codes possibles

```
updateList  f =  (>>= (maybeToList . f))
updateList2 f =  (>>= (maybeToList2 . f))
    where
    maybeToList2 Nothing = []
    maybeToList2 (Just x) = x

updateList  f l =  l >>= (\x-> maybeToList (f x))
updateList2 f l =  l >>= (\x-> maybeToList (f x)) >>= (\x->x)

updateList f l =  do -- in the list monad
    x <- l
    maybeToList(f x)
updateList2 f l =  do -- in the list monad
    x <- l
    r <- maybeToList(f x)
    r
```

3. `ajouter :: Maybe a -> Maybe [a] -> Maybe [a]`
   `retirer :: (Eq a) => Maybe a -> Maybe [a] -> Maybe [a]`

   Qui ajoutent ou retirent un élément à une liste, mais en iddentifiant la liste vide à Nothing, comme suit :

```
retirer (Just 2) (Just [2,7]) --> Just [7]
retirer (Just 2) (Just [2,2]) --> Nothing
retirer (Just 2) Nothing      --> Nothing
ajouter (Just 2) (Just [7])   --> Just [2,7]
ajouter (Just 2) Nothing      --> [2]
```

   Correction:

```
ajouter  mx ml =  do -- in the maybe monad
    x <- mx
    l  <- ml
    return (x:l)
rmEmpty :: [a] -> Maybe [a]
rmEmpty [] = Nothing
rmEmpty l  = Just l
```

```
retirer mx ml = do
    x <- mx
    l <- ml
    rmEmpty [y | y <- l, x /= y]
```

4. `bijection :: (Eq b) => [a] -> [b] -> [[(a,b)]]`

   qui, en supposant que ses arguments, p et q, soient sans doublons et de même taille n, fournit toutes les bijections $b : [(a, b)]$ entre p et q sous la forme de listes d'association, qui sont des lists de taille n de couples parcourants p et q :

$$\forall b \in (\texttt{bijection}\ p\ q), (\forall a \in p, \exists! b \in q, (a, b) \in b), (\forall b \in q, \exists! a \in p, (a, b) \in b)$$

   Correction:

```
bijection [] [] = [[]]
bijection [] _ = []
bijection (x:l1) l2 = do -- in the list monad
     y   <- l2
     bij <- bijection l1 [z | z<-l2, y /= z]
     return ((x,y):bij)
```