# Examen 1ère Session

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2022

**Durée** : 2 heures

**Documents autorisés** : tous documents papier (notes de cours, corrections de TD, etc.)

$\Longrightarrow$ **La clarté et la concision des réponses seront particulièrement appréciées** $\Longleftarrow$

## Exercice 1 : La monade (I)State indexée

Nous avons déjà manipulé la monade *State* dont voici un rappel de définition :

```
newtype State s a = State { runState :: s -> (a, s) }
```

Cette définition de type permet de manipuler des valeurs d'un type `a` arbitraire dans le contexte d'un *état* de type `s` que l'on peut lire (au sens de *reader*) ou écrire (au sens de *writer*), avec notamment les deux fonctions suivantes :

```
-- lecture de l'état
get :: State s s
get = State $ \q -> (q, q)

-- écriture de l'état
put :: s -> State s ()
put q = State $ \_ -> ((), q)
```

On rappelle également les définitions de *return* (*pure*) et *bind* pour la monade *State* :

```
instance Monad (State s) where
  return :: a -> State s a
  return x = State $ \q -> (x, q)

  (>>=) :: State s a -> (a -> State s b) -> State s b
  (State step) >>= g =
    State $ \q -> let (x, q') = step q
                  in runState (g x) q'
```

**Remarque** : cette définition suppose des instances associées pour les foncteurs et applicatifs, mais dans cet exercice nous ne nous occuperons que de monades.

Une restriction importante de la monade *State* est que lors d'un changement d'état — que l'on appelle communément une *transition* — l'état de départ et l'état d'arrivée doivent être du même type (`s` dans la définition). Une variante, que l'on appelle la monade *State* indexée (*indexed state monad*), relâche cette restriction. On se base pour cela sur la *typeclasse* suivante:

```
class IMonad m where
  ireturn :: a -> m s s a
  ibind :: m s t a -> (a -> m t u b) -> m s u b
```

On définit également des opérateurs *bind* infixes :

```
(>>>=) :: IMonad m => m s t a -> (a -> m t u b) -> m s u b
(>>>=) = ibind
```

```
infixl 1 >>>=

(>>>) :: IMonad m => m s t a -> m t u b -> m s u b
st1 >>> st2 = st1 >>>= \_ -> st2
infixl 1 >>>
```

Pour un constructeur de type `m s t` avec `m` instance de `IMonad`, le type `s` représente le type de l'état de départ d'une transition, et `t` le type d'arrivée.

**Remarque** : on ne pourra pas utiliser la notation `do` dans cet exercice, car les classes `IMonad` et `Monad` ne sont pas compatibles. En pratique il existe des extensions de langage pour lever cette restriction, , mais il s'agit d'un aspect avancé du langage.

### Question 1.1

Proposer une instance de la *typeclass* `IMonad` pour le type suivant :

```
newtype IState s t a = IState { runIState :: s -> (a, t) }
```

### Question 1.2

Définir les fonctions suivantes :

- la fonction `iget`, variante de `get`, et permettant de lire un état indicé de type entrant `s`
- la fonction `iput`, variante de `put`, et permettant d'écrire un état indicé de type sortant `t`
- la fonction `evalIState` de signature `IState s t a -> s -> a`

### Question 1.3

Voici un exemple d'utilisation de la monade State indexée :

```
data Started = Started
data Stopped = Stopped

start :: IState i Started ()
start = iput Started

stop :: IState Started Stopped ()
stop = iput Stopped

use :: (a -> a) -> a -> IState Started Started a
use g x = IState (\_ -> (g x, Started))

type SafeUse a = IState a Stopped a
```

Les définitions suivantes sont elles acceptées par le compilateur ?

- Si oui donner le type inféré selon-vous par le typeur, ainsi qu'un exemple d'utilisation.
- Sinon quel message d'erreur le typeur de Haskell indique-t-il selon vous ?

```
ex1 f = iget >>>= \x -> start >>> use f x >>>= \y -> stop >>> ireturn y

ex2 f = iget >>>= \x -> start >>> use f x >>>= \y -> ireturn y

ex3 f = iget >>>= \x -> use f x >>>= \y -> ireturn y
```

## Exercice 2 : Des graphes inductifs

Une problématique récurrente en programmation fonctionnelle concerne la modélisation et la manipulation des graphes. Les deux approches les plus courantes consistent : (1) soit à se placer dans IO et faire des graphes classiques, mutables, ou (2) modéliser les graphes avec des structures immutables, notamment des tables associatives (*maps*). Il existe des approches alternatives dont le but est de rendre la manipulation de graphe plus proches des concepts fondamentaux de la programmation fonctionnelle. Parmi ces approches, nous allons nous intéresser à la bibliothèque *fgl* (functional graph library)[1].

Voici une version simplifiée de la représentation proposée pour les graphes :

```haskell
type NodeId = Int

data Node a =
  Node { preds ::[NodeId]
       , nid :: NodeId
       , val :: a
       , succs :: [NodeId] }
  deriving (Show, Eq)

data Graph a = GEmpty | GBuild (Node a) (Graph a)
  deriving Show
```

Les nœuds (ou sommets) d'un graphe sont identifiés par des entiers de type `NodeId`. Le type `Node` représente, plutôt que juste un nœud, un *voisinage* avec en plus des identifiants du nœud et la valeur qu'il contient (d'un type `a` arbitraire), les listes de prédécesseurs et successeurs directs du nœud.

Le graphe lui même est construit de façon inductive comme :

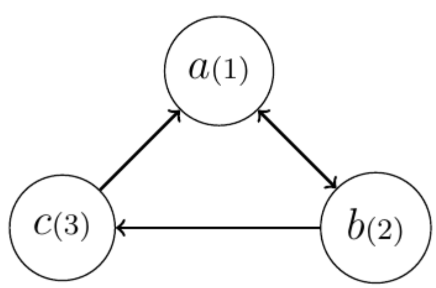
- soit le graphe vide `GEmpty`
- soit une construction d'un voisinage accolé à un graphe, avec le constructeur `GBuild`.

L'intérêt de cette représentation est qu'elle est *inductive*, de façon similaire aux listes, et est donc manipulable récursivement. L'inconvénient principal est qu'un même graphe peut-être représenté de différentes façons. Par exemple, les deux suivantes :

```haskell
exGraph1 = GBuild (Node [2, 3] 1 "a" [2]) $
           GBuild (Node [] 2 "b" [3]) $
           GBuild (Node [] 3 "c" []) $
           GEmpty

exGraph2 = GBuild (Node [2] 3 "c" [1]) $
           GBuild (Node [1] 2 "b" [1]) $
           GBuild (Node [] 1 "a" []) $
           GEmpty
```

représentent toutes deux le même graphe :




## Question 2.1

Proposer un ou plusieurs invariants permettant de garantir que la construction d'un `Graph a` est cohérente.

**Remarque** : on ne demande pas, dans cette question, d'implémenter cet invariant, puisque c'est l'objet de la dernière question de l'exercice (et il peut donc être utile de lire l'énonce de l'exercice en entier).

[1] https://hackage.haskell.org/package/fgl

### Question 2.2

Définir la fonction dont la signature est la suivante :

```
gmap :: (Node a -> Node b) -> Graph a -> Graph b
```

En déduire une instance de la *typeclasse* `Functor` rappelée ci-dessous :

```
type Functor :: (* -> *) -> Constraint
class Functor f where
   fmap :: (a -> b) -> f a -> f b
   (<$) :: a -> f b -> f a
   {-# MINIMAL fmap #-}
```

### Question 2.3

En utilisant `gmap`, proposer une définition de la fonction :

```
greverse :: Graph a -> Graph a
```

et qui permet d'inverser le sens des arêtes d'un graphe `g`.

### Question 2.4

Montrer, par raisonnement équationnel, que :

1. `gmap id` = `id`
2. `gmap f . gmap f'` = `gmap (f . f')` pour tout `f, f' :: Node a -> Node b`
3. `greverse . greverse` = `id`

### Question 2.5

Proposer une fonction de *fold* pour les graphes inductifs, avec la signature suivante :

```
gfold :: (Node a -> b -> b) -> b -> Graph a -> b
```

En utilisant `gfold` :

- implémenter la fonction `nodes :: Graph a -> [a]` qui retourne une liste des nœuds d'un graphe `g`
- réimplémenter la fonction `gmap`

### Question 2.6

Définir la fonction :

```
successors :: Graph a -> NodeId -> [NodeId]
```

qui retourne dans un graphe `g` une liste des successeurs directs d'un nœud identifié par son identifiant `nid`.

**Remarque** : on supposera que le nœud est bien présent dans le graphe

En déduire une définition de la fonction `descendants` de même signature mais qui retourne la liste de *tous* les successeurs du nœud. Cette fonction doit bien sûr ne pas boucler.

### Question 2.7

Proposer une implémentation du ou des invariants de la question 2.1