Programmation Fonctionnelle Avancée

IN4 TheGeeks

*Geek: **DEMO FKD** - If I need to define myself in one word that would be a tech-obsessed!!!*

§ Correction Examen 2023 §

1: Questions de cours

(1) QCM : Pour chacune des questions, donnons dans un tableau toute les réponses

Question	Réponse
1	a, d
2	e
3	c
4	b
5	b
6	b

(2) Fonction frac2double qui convertie une chaine de caractères représentant une fraction en le double correspondant

```
frac2double :: String -> Double
frac2double ns = read ns :: Double
```

2: Ecriture des fonctions

(1) Type de la fonction

```
series :: (a -> a) -> a -> [a]
```

(2) Utilisons cette fonction pour définir les listes suivantes :

- Les entiers naturels

```
listeNat = series (+1) 1
```

- Les puissances de 2

```
listePuisDeux = series (^2) 1
```

- La liste infinie de 1

```
listeInfUn = series id 1
```

(3)

a) Types des fonctions *fib* et *fibs*

```
fib :: Int -> Int
fibs :: Int -> (Int, Int)
```

b) Utilisons la fonction *series* pour definir la suite des nombres de Fibonacci

```
suite :: [Int]
suite = map fib (series (+1) 0)
```

(4) Fonction qui coupe les branches d'un arbre à une profondeur donnée

```
prune :: Int -> Rose a -> Rose a
prune 0 (Node r rs) = Node r []
prune n (Node r rs) = Node r (map (prune (n-1)) rs)
```

3: Etude de cas: Monade des traces

(1) Appliquée à la programmation fonctionnelle, la monade sera définie par :

- un constructeur de type, qui sera donc un type monadique m ;
- une fonction nommée *return*, qui construira à partir d'un objet de type a, un autre objet de type monadique $m\ a$. À ce titre, il s'agira d'une fonction monadique, de signature return :: $a \rightarrow m\ a$;
- une opération nommée *bind*, ou >>=, qui permettra de composer une fonction monadique à partir d'autres fonctions monadiques, en respectant une logique propre à chaque monade, de signature $(\gg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$.

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

(2) Les trois lois des monades sont exprimées ainsi :

- Composition neutre par return à gauche : $(return\ x) >\!>\!= f = f\ x$
- Composition neutre par return à droite : $m >\!>\!= return = m$
- Associativité : $(m >\!>\!= f) >\!>\!= g = m >\!>\!= (\ x \rightarrow (f\ x >\!>\!= g))$

(3) Complétion du code de la fonction *makeEvalM*

```
makeEvalM :: (Monad m) => (Int -> Int -> m Int) -> (Term -> m Int)
makeEvalM divM = (\t -> return t >>= evalM)
        where evalM (Const c) = return c
              evalM (Div n d) = evalM n >>= \x ->
              evalM d >>= \y ->
              divM x y
```

(4) Code permettant de faire de *Logger* une instance de la classe *Monad*

```
import Data.Monoid

data Logger o r = Logger {
                    output :: o,
                    result :: r
                  } deriving Show
```

```haskell
instance (Monoid o) => Monad (Logger o) where
    -- return :: r -> Logger o r
    return e = Logger mempty e
    -- (>>=) :: Logger o r -> (r -> Logger o s) -> Logger o s
    m >>= k = let (Logger o1 a) = m
                  (Logger o2 b) = k a
                  out = mappend o1 o2 in
              Logger out b
```

(5) Complétion du code

```haskell
evalLogger :: Term -> Logger String Int
evalLogger = makeEvalM divLogger
    where divLogger :: Int -> Int -> Logger String Int
          divLogger a b = let r = div a b
                              out = concat [
                                        "Division : ",
                                        show a, " / ", show b,
                                        " = ", show r, "\n"
                                    ]
                          in Logger out r
```

(6) Pour accéder directement au résultat de l'évaluation d'une expression il faut utiliser *result* $ *evalLogger expression*