

# Programação Funcional (MCCC015-23)

## Lista de Exercícios 6

### Monads

Emilio Francesquini  
e.francesquini@ufabc.edu.br  
Universidade Federal do ABC

26 de agosto de 2024



Nesta lista de exercícios classificamos os exercícios em três categorias que refletem o esforço relativo e o XP obtido para determinação da sua nota:

- 🌟 são exercícios básicos que serão suficientes apenas para te levar a saber os rudimentos do assunto. Seu XP para determinação da sua nota final na disciplina é equivalente ao de um Charmander.
- 🌟 são exercícios intermediários que exigem um pouco mais de esforço. Resolver estes exercícios vai te levar a entender um pouco melhor os conceitos e você já começará a ser capaz de utilizar estes conceitos em situações diferentes que lhe forem apresentadas. Seu XP para determinação da sua nota final na disciplina é equivalente ao de um Charmeleon.
- 🌟 exercícios para Pokémon Masters. O nível de dificuldade elevado te fará a entender, de verdade, os conceitos por trás do assunto (ao contrário do Charmeleon) que apenas permite que você reproduza/adapte uma aplicação do conceito. Seu XP para determinação da sua nota final na disciplina é equivalente ao de um Charizard.

---

Lembrete das **Leis dos Functores**:

- Identidade: `fmap id = id`
- Composição: `fmap (f . g) == fmap f . fmap g`

Lembrete das **Leis dos Applicatives**:

- Identidade: `pure id <*> v = v`
- Homomorfismo: `pure f <*> pure x = pure (f x)`
- Intercâmbio: `u <*> pure y = pure ($ y) <*> u`
- Composição: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

Lembrete das **Leis das Mônadas**:

- Identidade à esquerda: `return a >>= h = h a`
- Identidade à direita: `m >>= return = m`
- Associatividade: `(m >>= g) >>= h = m >>= (\x -> g x >>= h)`

Suas implementações de functores, applicatives e mônadas precisam obedecer essas leis.

---

## 🦄 Exercício 1

Implemente instâncias de Functor, Applicative e Monad para o tipo Caixa.

```
newtype Caixa a = Caixa a deriving (Eq, Show)
```

### Resposta

```
instance Functor Caixa where
  fmap f (Caixa a) = Caixa $ f a

instance Applicative Caixa where
  pure = Caixa
  (Caixa f) <*> (Caixa v) = Caixa (f v)

instance Monad Caixa where
  (Caixa a) >>= f = f a
```

## 🦄 Exercício 2

Dado o tipo

```
data Expr a = Var a | Val Int | Add (Expr a) (Expr a) deriving Show
```

que contém variáveis de um tipo a, defina instâncias para esse tipo de Functor, Applicative e Monad. (Dica: você já criou as instâncias de Functor e Applicative na lista anterior).

### Resposta

```
instance Functor Expr where
  -- fmap :: (a -> b) -> f a -> f b
  -- fmap :: (a -> b) -> Expr a -> Expr b
  fmap g (Var x)    = Var (g x)
  fmap _ (Val x)    = Val x
  fmap g (Add e1 e2) = Add (fmap g e1) (fmap g e2)

instance Applicative Expr where
  -- pure :: a -> f a
  -- pure :: a -> Expr a
  pure      = Var
  -- <*> :: f (a -> b) -> f a -> f b
  -- <*> :: Expr (a -> b) -> Expr a -> Expr b
  Var g <*> e = fmap g e

instance Monad Expr where
  -- (>>=) :: f a -> (a -> f b) -> f b
  -- (>>=) :: Expr a -> (a -> Expr b) -> Expr b
  Var x    >>= g = g x
  Val x    >>= _ = Val x
  Add e1 e2 >>= g = Add (e1 >>= g) (e2 >>= g)
```

### Exercício 3

Defina instâncias de Functor, Applicative e Monad para os seguintes tipos. (Dica: você já criou as instâncias de Functor e Applicative na lista anterior).

```
newtype Identity a = Identity a
data Pair a = Pair a a
```

#### Resposta

```
newtype Identity a = Identity a

instance Functor Identity where
  -- fmap :: (a -> b) -> f a -> f b
  -- fmap :: (a -> b) -> Identity a -> Identity b
  fmap g (Identity x) = Identity (g x)

instance Applicative Identity where
  -- pure :: a -> f a
  -- pure :: a -> Id a
  pure = Identity
  -- <*> :: f (a -> b) -> f a -> f b
  -- <*> :: Identity (a -> b) -> Identity a -> Identity b
  Identity g <*> x = fmap g x

instance Monad Identity where
  return = Identity
  (Identity x) >>= g = g x

data Pair a = Pair a a deriving Show

instance Functor Pair where
  -- fmap :: (a -> b) -> Pair a -> Pair b
  fmap g (Pair x y) = Pair (g x) (g y)

instance Applicative Pair where
  -- pure :: a -> Pair a a
  pure x = Pair x x
  -- <*> :: Pair (a -> b) -> Pair a -> Pair b
  (Pair g h) <*> (Pair x y) = Pair (g x) (h y)

instance Monad Pair where
  return = pure
  -- (>>=) :: Pair a -> (a -> Pair b) -> Pair b
  (Pair x y) >>= g = g x
  -- (Pair x y) >>= g = g y --ambas estão corretas, a escolha é pelo contexto
```

### Exercício 4

**Obedecendo** as leis descritas no início da lista de exercícios, escreva instâncias de Functor, Applicative e Monad para o tipo `data Fantasma a = Fantasma`. (Dica: você já criou as instâncias de Functor e Applicative na lista anterior).

## Resposta

```
instance Functor Fantasma where
  fmap _ _ = Fantasma

instance Applicative Fantasma where
  pure _ = Fantasma
  _ <*> _ = Fantasma

instance Monad Fantasma where
  Fantasma >>= f = Fantasma
```

## Exercício 5

Obedecendo as leis descritas acima, escreva instâncias de Functor, Applicative e Monad para o tipo `data Duo a = Duo (Bool -> a)`. (Dica: você já criou as instâncias de Functor e Applicative na lista anterior).

## Resposta

```
instance Functor Duo where
  fmap f (Duo g) = Duo (f . g)

instance Applicative Duo where
  pure v = Duo $ const v
  (Duo f) <*> (Duo g) = Duo $ \b -> f b (g b)

runDuo :: Duo a -> Bool -> a
runDuo (Duo f) = f

instance Monad Duo where
  -- (>>=) :: Duo a -> (a -> Duo b) -> Duo b
  a >>= f = Duo $ \ b -> runDuo (f $ runDuo a b) b
```

## Exercício 6

Você está escrevendo uma lib que faz requisições HTTP para servidores. Você definiu o tipo:

```
data Request a = Loading | Error | Success a
```

Defina instâncias de Functor, Applicative e Monad para este tipo. Caso haja ao menos um Error envolvido nas funções (<\*>) e (>>=), o resultado deverá ser Error. Caso não haja nenhum Error e haja um Loading a resposta deverá ser Loading.

## Resposta

```
instance Functor Request where
  fmap f (Success x) = Success (f x)
  fmap _ Loading = Loading
  fmap _ Error = Error

instance Applicative Request where
  pure = Success
```

```

Error <*> _ = Error
_ <*> Error = Error
Loading <*> _ = Loading
_ <*> Loading = Loading
(Success f) <*> (Success x) = Success (f x)

```

```

instance Monad Request where
  return = pure
  (Success a) >>= amb = amb a
  Loading >>= _ = Loading
  Error >>= _ = Error

```

## Exercício 7

Dado o tipo:

```

data Bolso a = Um a | Dois a a | Tres a a a

```

- Crie uma instância de Functor que aplica a função passada em todas as "posições".
- Crie uma instância de Eq manualmente para Bolso, que compara apenas o valor mais a direita. Ou, em outras palavras: Um 5 == Dois \_ 5 == Tres \_ \_ 5 e Dois 10 5 == Tres \_ 1 5.
- Crie uma instância de Monad para Bolso, dado que, na hora de definir o (>>=), sempre o valor "mais a direita" deve ser enviado para a função. Você precisará também fazer uma instância de Applicative seguindo a mesma lógica.

## Resposta

```

instance Eq a => Eq (Bolso a) where
  Um a == Um b = a == b
  (Dois _ a) == x = Um a == x
  (Tres _ _ a) == x = Um a == x
  x == y = y == x

```

```

instance Functor Bolso where
  fmap f (Um x) = Um (f x)
  fmap f (Dois x y) = Dois (f x) (f y)
  fmap f (Tres x y z) = Tres (f x) (f y) (f z)

```

```

instance Applicative Bolso where
  pure = Um
  (Um f) <*> b = f <$> b
  (Dois _ f) <*> b = f <$> b
  (Tres _ _ f) <*> b = f <$> b

```

```

instance Monad Bolso where
  return = Um
  (Um a) >>= amb = amb a
  (Dois _ a) >>= amb = amb a
  (Tres _ _ a) >>= amb = amb a

```

## Exercício 8

Seu professor de educação física entrega uma lista de pesos e alturas e pede para você calcular o IMC de cada pessoa. Inicialmente, é dito que o dataset é uma lista de tuplas que representa uma tabela. Para cada linha há três campos: um nome [Char], uma altura Double e um peso Double. Porém, ao ter o dataset em mãos, descobre-se que algumas células estão branco.

Além de criar o próprio tipos adicionais conforme necessário, você deve criar uma função `imc` que recebe uma lista de tuplas, em que cada posição se refere a um dos campos citados anteriormente. Porém, em qualquer um deles, pode haver `Nothing`. A função deve retornar uma lista com o IMC corresponde para cada pessoa, quando isso for possível. Em outras palavras, a função precisa ter o tipo:

```
imc :: [(Maybe Nome, Maybe Peso, Maybe Altura)] -> [Maybe IMC]
```

Observações importantes:

- O IMC é calculado como  $\text{Peso}/\text{Altura}^2$ .
- Caso o nome esteja ausente, deve ser possível calcular o IMC de qualquer maneira

### Resposta

```
type Nome = String
```

```
type Peso = Double
```

```
type Altura = Double
```

```
type IMC = Double
```

```
imc :: [(Maybe Nome, Maybe Peso, Maybe Altura)] -> [Maybe IMC]
imc = fmap calculate where
  calculate (_, mp, ma) = do
    p <- mp
    a <- ma
    return $ p/(a**2)
```

## Exercício 9

Implemente a função `azul`. O seu comportamento pode ser inferido da sua assinatura dada abaixo.

```
azul :: Monad m => m (m a) -> m a
```

### Resposta

```
azul :: Monad m => m (m a) -> m a
azul m = m >>= id
```

## Exercício 10

Implemente a função `amarelo`. O seu comportamento pode ser inferido da sua assinatura dada abaixo.

```
amarelo :: Monad m => (a -> b) -> m a -> m b
```

## Resposta

```
amarelo :: Monad m => (a -> b) -> m a -> m b
amarelo = fmap
```

## Exercício 11

Implemente a função `vermelho`. O seu comportamento pode ser inferido da sua assinatura dada abaixo.

```
vermelho :: Monad m => (a -> b -> c) -> m a -> m b -> m c
```

## Resposta

```
vermelho :: Monad m => (a -> b -> c) -> m a -> m b -> m c
vermelho f ma mb = f <$> ma <*> mb
```

## Exercício 12

Implemente a função `verde`. O seu comportamento pode ser inferido da sua assinatura dada abaixo.

```
verde :: Monad m => m a -> m (a -> b) -> m b
```

## Resposta

```
verde :: Monad m => m a -> m (a -> b) -> m b
verde = flip (<*>)
```

## Exercício 13

Implemente a função `laranja`. O seu comportamento pode ser inferido da sua assinatura dada abaixo. Dica, use recursão.

```
laranja :: Monad m => [m a] -> m [a]
```

## Resposta

```
laranja :: Monad m => [m a] -> m [a]
laranja [] = return []
laranja (x:xs) = do
  x1 <- x
  xs' <- laranja xs
  return $ x1 : xs'
```

## Exercício 14

Implemente a função `roxo`. O seu comportamento pode ser inferido da sua assinatura dada abaixo. Dica, use a função `laranja` e `amarelo`.

```
roxo :: Monad m => [a] -> (a -> m b) -> m [b]
```

## Resposta

```
roxo :: Monad m => [a] -> (a -> m b) -> m [b]
roxo xs f = laranja $ amarelo f xs
```

## Exercício 15

Quase todas as funções com nomes de cores acima estão disponíveis na biblioteca padrão do Haskell. Você consegue identificá-las? (Dica: tente identificá-las apenas pela assinatura por conta própria, mas caso precise utilize o Hoogle para obter ajuda).

## Exercício 16

O tipo `MudaLista` é definido como

```
newtype MudaLista a = MudaLista { runMudaLista :: [Int] -> ([Int], a) }
```

O comportamento de uma função desse tipo é a de retornar uma nova lista, modificada ou não, e algum elemento resultante da operação. Esse tipo pode ser muito útil para implementar estruturas de dados como pilhas ou filas.

Por exemplo, eu posso ter um `MudaLista Int` chamado `desempilha` que retorna uma nova lista sem esse o elemento do topo junto do valor no topo:

```
desempilha :: MudaLista Int
desempilha = MudaLista $ \ (x:xs) -> (xs, x)
```

```
> runMudaLista desempilha [1,2,3]
([2,3], 1)
```

```
> runMudaLista desempilha [7,8,9]
([8,9], 7)
```

Como poderíamos modelar a função `empilha`? Note que precisamos receber o elemento a ser empilhado, e então modificar a lista e não retornar nada (ou seja, seria `void` em uma linguagem como C ou Java). Em Haskell, para fazer algo equivalente usamos o valor `unit` escrito como `()`. O `unit` é um valor monótono (no sentido de desinteressante, chato...) do tipo `()`, ou seja, `() :: ()`.

`()` é usado sempre que precisamos de um valor e tipo que fomos obrigados a definir (bem na linha do `void` em Java ou C, onde o método/função deve ter obrigatoriamente um tipo de retorno), mas não temos realmente interesse no valor. Este é exatamente o caso de `empilha`! Não temos interesse nenhum em um valor de retorno, apenas na lista modificada.

Assim, podemos definir o tipo de `empilha` como `a -> MudaLista ()`, e a usaremos assim:

```
> runMudaLista (empilha 42) [1,2,3]
([42,1,2,3], ())
```

```
> runMudaLista (empilha 99) []
([99], ())
```

-  Implemente a operação `empilha` conforme os exemplos fornecidos.
-  Implemente uma instância de `Functor` para `MudaLista`. Note que como esse tipo de dado armazena uma função, a instância de `Functor` deve gerar uma nova função. Exemplos no `ghci`:

```

> let cemVezesDesempilha = fmap (*100) desempilha
> runMudaLista cemVezesDesempilha [4,5,6]
([5,6], 400)

> let mostraDesempilha = fmap show desempilha
> :t mostraDesempilha
mostraDesempilha :: MudaLista String

> runMudaLista mostraDesempilha [4,5,6]
([5,6], "4")

```

c) 🐛 Implemente uma instância de Applicative para MudaLista.

```

> :t fmap (+) desempilha
fmap (+) desempilha :: MudaLista (Int -> Int)

> :t (fmap (+) desempilha) <*> desempilha
(fmap (+) desempilha) <*> desempilha :: MudaLista Int

> let somaPrimeiros = (fmap (+) desempilha) <*> desempilha
> runMudaLista somaPrimeiros [10,20,40,80]
([40,80],30)

```

d) 🐛 Implemente a função desempilhaVarios, que recebe um inteiro representando quantos elementos devem ser desempilhados da lista, e retorna-os na ordem que foram desempilhados

```

> runMudaLista (desempilhaVarios 3) [1,2,3,4,5,6,7,8,9]
([4,5,6,7,8,9],[1,2,3])

```

e) 🐛 Implemente a função empilhaVarios, que recebe uma lista de inteiros e vai empilhando-os um a um, da esquerda para a direita

```

> runMudaLista (empilhaVarios [10,20,30]) [1,2,3]
([30,20,10,1,2,3], ())

```

## Resposta

```

empilha :: Int -> MudaLista ()
empilha x = MudaLista $ \l -> (x : l, ())

desempilha :: MudaLista Int
desempilha = MudaLista $ \(x:xs) -> (xs, x)

-- exemplo
fazOperacoes :: [Int] -> [Int]
fazOperacoes l = 14
  where
    (l1, _) = runMudaLista desempilha l
    (l2, _) = runMudaLista desempilha l1
    (l3, ()) = runMudaLista (empilha 5) l2
    (l4, ()) = runMudaLista (empilha 6) l3

instance Functor MudaLista where
  fmap g (MudaLista f) = MudaLista $ \l -> fmap g (f l)

```

```

instance Applicative MudaLista where
  pure x = MudaLista $ \l -> (l, x)
  (MudaLista fx) <*> (MudaLista f) =
    MudaLista $ \l ->
      let (l1, fab) = fx l
          (l2, a) = f l1
      in (l2, fab a)

desempilhaVarios :: Int -> MudaLista [Int]
desempilhaVarios n = sequenceA (replicate n desempilha)

empilhaVarios :: [Int] -> MudaLista ()
empilhaVarios = traverse_ empilha

```