

Programação Funcional (MCCC015-23)

Lista de Exercícios 5

Monoid, Foldable, Functor, Applicative

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

14 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)`

Suas implementações de funtores e aplicativos precisam obedecer essas leis.

👤 Exercício 1

Considere o tipo `data Resultado = Pontuacao Int | Cola`, que representa o resultado das atividades entregues por um aluno. No final do quadrimestre, deseja-se somar a pontuação de todas as atividades entregues. No entanto, no caso de `Cola`, toda a pontuação obtida até o momento deve ser descartada, pois implica reprovação automática. Implemente uma instância de `Monoid` para `Resultado` que modele esse comportamento.

Resposta

```
instance Semigroup Resultado where
  (Pontuacao x) <> (Pontuacao y) = Pontuacao (x + y)
  _ <> _                          = Cola
```

```
instance Monoid Resultado where
  mempty = Pontuacao 0
```

👤 Exercício 2

Considere o tipo `data Set a = Set [a] deriving Eq`, que deve representar um conjunto arbitrário de qualquer `a`. **Invariante:** a lista armazenada pelo construtor `Set` deve sempre conter elementos únicos e ordenados.

1. Implemente uma instância de `Show` para o `set`, que mostre-o conforme o seguinte exemplo: `show Set [1,2,4] → "{1,2,4}"` (considere a função `intercalate`¹ do `Data.List`).
2. Implemente uma função `fromList :: Ord a => [a] -> Set a` que gera um conjunto a partir de uma lista.
3. Implemente uma função `member :: Ord a => a -> Set a -> Bool` que retorna se um elemento pertence àquele conjunto
4. Implemente uma função `insert :: Ord a => a -> Set a -> Set a` que adiciona o elemento passado por parâmetro no conjunto passado por parâmetro
5. Implemente uma função `delete :: Ord a => a -> Set a -> Set a` que faz o inverso da função acima

¹<https://hackage.haskell.org/package/base-4.20.0.1/docs/Data-List.html#v:intercalate>

Resposta

```
instance Show a => Show (Set a) where
  show (Set xs) = '{ ' : (L.intercalate ", " $ show <$> xs) <> "}"

fromList :: Ord a => [a] -> Set a
fromList = Set . L.sort . L.nub

member :: Ord a => a -> Set a -> Bool
member x (Set xs) = x `elem` xs

insert :: Ord a => a -> Set a -> Set a
insert x (Set xs) = fromList $ (x : xs)

delete :: Ord a => a -> Set a -> Set a
delete x (Set xs) = Set $ L.delete x xs
```

Exercício 3

Implemente uma instância de Monoid para Set a, dado que a seja Ord, utilizando a operação de união de conjuntos

Resposta

```
instance Ord a => Semigroup (Set a) where
  (Set as) <> (Set bs) = fromList $ as <> bs

instance Ord a => Monoid (Set a) where
  mempty = Set []
```

Exercício 4

Você está abrindo uma lanchonete diferente, pois não existe um cardápio fixo. Você apenas fornece uma lista de ingredientes possíveis, e os clientes podem combiná-los como bem entenderem. Considere os tipos `data Dieta = Vegano | Vegetariano | Tradicional`, e `data Lanche = Lanche (Set String) Int`. Dessa forma, um Lanche é composto por um conjunto de ingredientes, um preço em centavos e qual a Dieta adequada para aquele Lanche. Implemente uma instância de monóide para Dieta, considerando o seguinte que duas dietas são combinadas usando "denominador comum", ou seja, duas dietas diferentes resultam na menos restritiva:

- Se você colocar queijo (alimento vegetariano, mas não vegano) em um lanche vegano, ele deixa de ser vegano
- Mas colocar queijo em um lanche tradicional não faz com que ele deixe de ser tradicional

Resposta

```
instance Semigroup Dieta where
  Tradicional <> _      = Tradicional
  _ <> Tradicional     = Tradicional
  Vegetariano <> _     = Vegetariano
  Vegano <> Vegetariano = Vegetariano
  Vegano <> Vegano     = Vegano
```

```
instance Monoid Dieta where
  mempty = Vegano
```

Exercício 5

Implemente uma instância de monoide para o `Lanche` conforme as seguintes regras para combinar dois `Lanche` s:

- A lista de ingredientes deve ser combinada usando união de conjuntos (pode usar sua implementação de `<>`)
- O preço deve ser simplesmente somado
- A `Dieta` deve seguir o `<>` implementado anteriormente

Resposta

```
instance Semigroup Lanche where
  (Lanche is1 p1 d1) <> (Lanche is2 p2 d2) =
    Lanche (is1 <> is2) (p1 + p2) (d1 <> d2)
```

```
instance Monoid Lanche where
  mempty = Lanche mempty 0 mempty
```

Exercício 6

Defina a instância de `Functor` o seguinte tipo de árvores binárias:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a) deriving Show
```

Resposta

```
instance Functor Tree where
  --fmap :: (a -> b) -> Tree a -> Tree b
  fmap g (Leaf x) = Leaf $ g x
  fmap g (Node l x r) = Node (fmap g l) (g x) (fmap g r)
```

Exercício 7

Defina a função `arvorePossui` a que retorna `True` caso `a` seja um valor presente na árvore e `False` caso contrário.

Resposta

```
arvorePossui :: Eq a => Tree a -> a -> Bool
arvorePossui (Leaf v) x = v == x
arvorePossui (Node l v r) x =
  x == v ||
  arvorePossui l x ||
  arvorePossui r x
```

Exercício 8

Defina a função `contaLetras :: Tree String -> Tree Int` que recebe uma árvore onde cada nó contém uma string e devolve uma nova árvore onde cada nó contém o comprimento das strings que continham na árvore dada como entrada.

Resposta

```
contaLetras :: Tree String -> Tree Int
contaLetras = fmap length
```

Exercício 9

Defina uma instância de `Foldable` para `Tree`. Dica! Para definir a instância de `Foldable`, basta implementar qualquer um dentre os seguintes métodos `foldMap` ou `foldr`.

Resposta

```
instance Foldable Tree where
  foldMap f (Leaf v) = f v
  foldMap f (No l v r) = foldMap f l <> f v <> foldMap f r
```

Exercício 10

Implemente a função `convertString2Int :: String -> Maybe Int`, que converte, se possível, uma string para inteiro. Você pode usar as funções em `Text.Read`.

Resposta

```
convertString2Int :: String -> Maybe Int
convertString2Int = readMaybe
```

Exercício 11

Implemente a função `nothingToZero :: Maybe Int -> Int` que dado um `Maybe Int` devolve o próprio inteiro caso presente ou 0 caso contrário.

Resposta

```
nothingToZero :: Num a => Maybe a -> a
nothingToZero Nothing = 0
nothingToZero (Just x) = x
```

Exercício 12

Usando a instância de `Foldable` (cuja instância você definiu nos exercícios anteriores) e o monoide `Sum` (veja `Data.Monoid` para mais informações) defina a função `frutasDaArvore :: Tree String -> Int` que recebe uma árvore de strings e devolve o número total de frutas que a árvore tem. Cada nó da árvore recebida como parâmetro possui uma string que informa a quantidade de frutas que aquele nó possui. Caso a string não corresponda a um número inteiro, ignore o nó e continue a contagem.

Resposta

```
frutasDaArvore :: Tree String -> Int
frutasDaArvore = getSum . foldMap (Sum . nothingToZero . convertString2Int)
```

Exercício 13

Escreva as instâncias de `Functor` e `Applicative` para o tipo `ZipList`, no qual a função pura faz uma lista infinita de cópias do argumento, e o operador `<*>` aplica cada função argumento no valor correspondente na mesma posição.

```
newtype ZipList a = Z [a] deriving Show

instance Functor ZipList where
  -- fmap :: (a -> b) -> ZipList a -> ZipList b
  fmap g (Z xs) = ..

instance Applicative ZipList where
  -- pure :: a -> ZipList a
  pure x = ..
```

Resposta

```
newtype ZipList a = Z [a] deriving Show

instance Functor ZipList where
  -- fmap :: (a -> b) -> ZipList a -> ZipList b
  fmap g (Z xs) = Z (fmap g xs)

instance Applicative ZipList where
  -- pure :: a -> ZipList a
  pure x = Z (repeat x)
  -- <*> :: f (a -> b) -> f a -> f b
```

```
-- <*> :: Ziplist (a -> b) -> Ziplist a -> Ziplist b
(Z gs) <*> (Z xs) = Z [g x | (g, x) <- zip gs xs]
```

Exercício 14

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.

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
```

Exercício 15

Defina instâncias de Functor, Applicative para os seguintes tipos:

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

Resposta

```
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 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)

```

Exercício 16

Run-length encoding, ou RLE para os íntimos, é um método de compactação sem perdas que é muito utilizado para comprimir dados com muitas sequências repetitivas valores. Talvez o seu uso mais comum tenha sido para compressão de arquivos de imagem simples como ícones. Considere o ADT `data RLE a = Repeat Int a (RLE a) | End deriving (Eq, Show)`. Caso se trate de uma sequência vazia utilizamos o construtor `End` e caso se trate de uma sequência não vazia, utilizamos o construtor `Repeat` com o inteiro contendo o número de elementos repetidos do tipo `a` seguido de outro `Repeat` ou um `End` para indicar o termino da sequência. Implemente a função `rleCons :: Eq a => a -> RLE a -> RLE a` cujo comportamento é análogo à função `cons` para listas `(:)`.

Resposta

```

rleCons :: Eq a => a -> RLE a -> RLE a
rleCons x End = Repeat 1 x End
rleCons x r@(Repeat i y ys)
  | x == y = Repeat (i + 1) y ys
  | otherwise = Repeat 1 x r

```

Exercício 17

Implemente uma instância de `Foldable` para RLE.

Resposta

```

instance Foldable RLE where
  foldMap _ End = mempty
  foldMap f (Repeat i x xs) = foldMap f (replicate i x) <> foldMap f xs

```

Exercício 18

Implemente a função `encode :: Eq a => [a] -> RLE a` que faz as conversões entre listas comuns e RLE. Use a instância de `foldable` definida no exercício anterior. Exemplos:


```

> encode [1,1,2,2,2]
  Repeat 2 1 (Repeat 3 2 End)

> encode [1,2,3,3,3,2,2]
  Repeat 1 1 (Repeat 1 2 (Repeat 3 3 (Repeat 2 2 End)))

> encode ['a','x','x','z','x']
  Repeat 1 'a' (Repeat 2 'x' (Repeat 1 'z' (Repeat 1 'x' End)))

```

Resposta

```

encode :: Eq a => [a] -> RLE a
encode = foldr rleCons End

```

Exercício 19

Implemente a função `decode :: RLE a -> [a]` que faz o processo inverso de `encode`. Use a instância de `foldable` definida para o tipo `RLE`.

```

decode :: RLE a -> [a]
decode = foldr (:) []

```

Exercício 20

Obedecendo as leis descritas no início da lista de exercícios, escreva instâncias de `Functor` e `Applicative` para o tipo `data Fantasma a = Fantasma`.

Resposta

```

instance Functor Fantasma where
  fmap _ _ = Fantasma

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

```

Exercício 21

Obedecendo as leis descritas acima, escreva instâncias de `Functor` e `Applicative` para o tipo `data Duo a = Duo (Bool -> a)`.

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)

```

Exercício 22

Suponha que você está cursando a disciplina de Arquitetura de Computadores e precisa implementar uma hierarquia de memória para o seu emulador MIPS. Você desenvolveu a seguinte estrutura de dados:

```
data Memory a = UnifiedCache a (Memory a) | SplitCache a a (Memory a) | RAM a
```

Essa estrutura tem a propriedade interessante de que **sempre** vai terminar na memória RAM, e podemos ter quantos níveis de cache quisermos. Por exemplo, se quisermos armazenar os acessos a cada um dos níveis de memória, podemos fazer: `SplitCache 5 2 (UnifiedCache 10 (RAM 5))`.

Imagine que queremos calcular o total de acessos feitos a todos os níveis de memória usando:

```
foldMap Sum $ SplitCache 5 2 (UnifiedCache 10 (RAM 5))
```

Implemente instâncias de Functor, Foldable e Traversable para a estrutura Memory.

Resposta

```
instance Functor Memory where
```

```
  fmap f (UnifiedCache x m) = UnifiedCache (f x) (fmap f m)
  fmap f (SplitCache x y m) = SplitCache (f x) (f y) (fmap f m)
  fmap f (RAM x)             = RAM (f x)
```

```
instance Foldable Memory where
```

```
  foldMap toMonoid (UnifiedCache x m) = (toMonoid x) <> (foldMap toMonoid m)
  foldMap toMonoid (SplitCache x y m) =
    (toMonoid x) <> (toMonoid y) <> (foldMap toMonoid m)
  foldMap toMonoid (RAM x) = toMonoid x
```

```
instance Traversable Memory where
```

```
  traverse fx (UnifiedCache x m) = UnifiedCache <$> (fx x) <*> (traverse fx m)
  traverse fx (SplitCache x y m) =
    SplitCache <$> (fx x) <*> (fx y) <*> (traverse fx m)
  traverse fx (RAM x) = RAM <$> fx x
```