Atividade X

Problema 5

A classe Monad permite a modelagem de computações que têm uma estrutura sequencial e pode encapsular efeitos colaterais, como estado mutável, I/O, exceções, entre outros. A definição básica da classe Monad em Haskell é:

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

operações:

return:

- Pega um valor de qualquer tipo a e retorna uma monad de tipo a. É o meio de introduzir um valor em um contexto monádico.

(>>=) (também conhecido como "bind"):

- É a operação fundamental da classe Monad. Ele pega um valor monádico e uma função que retorna um valor monádico, e "liga" os dois juntos. A intuição é que ele pega um valor "embrulhado", aplica uma função a ele e retorna um novo valor "embrulhado".

(>>):

- Semelhante ao bind (>>=), mas descarta qualquer valor produzido pelo monad à esquerda. É frequentemente usado quando se quer sequenciar duas ações monádicas, mas não se preocupa com o resultado da primeira.

fail:

- Fornece um mecanismo para falhas monádicas com uma mensagem de erro. No entanto, o uso de fail não é encorajado em design monádico moderno em Haskell, e muitas vezes é melhor usar monads como Either ou Maybe para modelar falhas explicitamente.

Diferença entre >> e >>=:

- (>>): É usado para sequenciar duas ações monádicas, onde não se está interessado no resultado da primeira ação, que é descartado.

```
-     do
-         putStrLn "Qual é o seu nome?"
-         name <- getLine
-         putStrLn ("Olá, " ++ name ++ "!")
-
```

- Em vez de capturar o resultado de putStrLn "Qual é o seu nome?" (que é ()), nós apenas sequenciamos para a próxima ação usando (>>).

- (>>=): É usado para "extrair" o valor de um monad e passá-lo para a próxima função monádica.
    - getLine >>= \name -> putStrLn ("Olá, " ++ name ++ "!")
    - -- o resultado de getLine é passado para a função lambda que imprime a saudação.

Problema 6

As classes Functor e Applicative são duas classes de tipos centrais em Haskell que definem operações para trabalhar com tipos "conteinerizados" ou "embrulhados". Ambas são prelúdios para a classe Monad, fornecendo níveis crescentes de complexidade e capacidade.

1. Classe Functor:
    - A classe Functor define a operação fmap, que permite que você aplique uma função a um valor "embrulhado" dentro de algum contexto.
    - Definição básica:
        - class Functor f where
        - fmap :: (a -> b) -> f a -> f b
    - Exemplo com Maybe:
        - data Maybe a = Nothing | Just a
        -
        - instance Functor Maybe where
        - fmap _ Nothing = Nothing
        - fmap f (Just x) = Just (f x)
    - Agora é possível fazer algo como:
        - fmap (+1) (Just 3) -- Resulta em Just 4
        - fmap (+1) Nothing -- Resulta em Nothing
2. Classe Applicative:
    - A classe Applicative estende a ideia de Functor para permitir a aplicação de funções "embrulhadas" a valores "embrulhados".
    - Definição básica:
        - class Functor f => Applicative f where
        - pure :: a -> f a
        - (<*>) :: f (a -> b) -> f a -> f b
    - pure pega um valor e o coloca em um contexto.
    - <*> (pronuncia-se "aplicar") pega uma função embrulhada e um valor embrulhado e aplica a função ao valor.
    - Exemplo com Maybe:
        - instance Applicative Maybe where
        - pure = Just
        - Nothing <*> _ = Nothing
        - _ <*> Nothing = Nothing
        - Just f <*> Just x = Just (f x)
    - Agora é possível fazer algo como:
        - Just (+1) <*> Just 3 -- Resulta em Just 4

- Just (+1) <*> Nothing -- Resulta em Nothing

Problema 9

O monad Either é uma extensão natural do tipo de dados Maybe que não apenas permite que um cálculo falhe, mas também fornece informações adicionais sobre o motivo da falha.

O tipo Either é definido da seguinte forma:

```
data Either a b = Left a | Right b
```

Aqui, Left e Right são os construtores para o tipo Either. Convenções comuns em Haskell são:

Left: Representa um erro ou falha, e o valor a dentro dele frequentemente carrega informações sobre o erro.
Right: Representa um valor bem-sucedido de tipo b.
Quando usado como um monad, Either permite que você encadeie cálculos que podem falhar, interrompendo a cadeia no primeiro erro.

Exemplo:

```
checkPositive :: Int -> Either String Int
checkPositive x
    | x > 0     = Right x
    | otherwise = Left "Número não é positivo."

half :: Int -> Either String Int
half x
    | even x    = Right (x `div` 2)
    | otherwise = Left "Número não é par."

-- Agora, usando o monad Either, pode-se encadear essas operações:
compute :: Int -> Either String Int
compute x = do
    positive <- checkPositive x
    half positive

compute 4  -- Resulta em Right 2
compute 3  -- Resulta em Left "Número não é par."
compute -4 -- Resulta em Left "Número não é positivo."
```