Programação Funcional (MCCC015-23)
Lista de Exercícios 7
Monads

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

05 de setembro 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.

AVISO

Como preparação para resolver os exercícios desta lista, recomendo fortemente que você resolva o exercício 16 da Lista 6 (`MudaLista`).

Estamos simulando um sistema de comércio em um RPG, e queremos simular lojas e clientes.

Considere os tipos `Produto` e `Loja`, fornecidos abaixo:

```haskell
data Produto = Produto
    { nome  :: String
    , preco :: Int
    }
  deriving (Show, Eq)

data Loja = Loja
    { caixa   :: Int
    , estoque :: [Produto]
    }
  deriving (Show, Eq)
```

Um `Produto` contém apenas um `nome` e um `preco`, enquanto uma `Loja` possui o seu `estoque` (uma lista de `Produto`) e um `caixa`, representando quanto dinheiro aquela loja tem em caixa.

Considere ainda as funções `comprar` e `vender`, cujas implementações são dadas abaixo:

```haskell
comprar :: String -> State Loja (Maybe Produto)
comprar nomeProduto = state realizaCompra
  where
    realizaCompra :: Loja -> (Maybe Produto, Loja)
    realizaCompra lj =
      case findAndRemoveProduto $ estoque lj of
        Nothing -> (Nothing, lj)
        Just (p, novoEstoque) -> (Just p, Loja novoCaixa novoEstoque)
          where novoCaixa = caixa lj + preco p
    findAndRemoveProduto :: [Produto] -> Maybe (Produto, [Produto])
    findAndRemoveProduto ps = go ps []
      where
        go [] _ = Nothing
        go (x:xs) acc
          | nome x == nomeProduto = Just (x, reverse acc ++ xs)
          | otherwise = go xs (x : acc)

vender :: String -> Int -> State Loja Int
vender nomeProduto valor = state realizaVenda
  where
    realizaVenda :: Loja -> (Int, Loja)
    realizaVenda lj
      | valor <= caixa lj =
        ( valor
        , Loja (caixa lj - valor) ((Produto nomeProduto valor) : estoque lj))
      | otherwise = (0, lj)
```

A função `comprar` representa o interesse de um cliente comprar um `Produto`. A função recebe o nome do produto que o cliente deseja e retorna a ação da loja.

- Se o `Produto` estiver em estoque: removemos o produto do `estoque`, aumentamos o `caixa` da loja, e entregamos o produto para o cliente no formato com um valor `Just produto` (`:: Maybe Produto`)

- Caso contrário: a loja fica intacta, e retornamos `Nothing`

A função `vender` representa o interesse de um cliente em vender um `Produto`. Recebemos o nome do produto e o valor que o cliente quer por ele.

- Se a loja tiver dinheiro o suficiente para pagar pelo produto: ele é adicionado ao estoque, descontamos o dinheiro do caixa e "entregamos"o dinheiro ao cliente (representado por retornar o 'Int' com o valor que ele pediu)
- Caso contrário, a loja fica intacta, e retornamos '0' (representando o número de moedas que entregamos ao cliente)

Usando a Monad `State` [1], queremos especificar o comportamento de vários clientes, e retornar se saíram satisfeitos da loja ou não (`Bool`), potencialmente modificando a `Loja`. Em outras palavras, o `type Cliente = State Loja Bool`. Por exemplo, suponha que temos um cliente que quer simplesmente vender uma espada por 10 moedas:

```
vendeEspada :: Cliente
vendeEspada = do
    valorVendido <- vender "Espada" 10
    return $ valorVendido > 0
```

Ou um cliente que tenta comprar um escudo[2]:

```
compraEscudo :: Cliente
compraEscudo = do
    maybeProduto <- comprar "Escudo"
    return $ isJust maybeProduto
```

Poderíamos simular esses clientes com:

```
> runState vendeEspada (Loja 100 [Produto "Escudo" 15])
(True,Loja {caixa = 90, estoque = [Produto {nome = "Espada", preco = 10},
                                   Produto {nome = "Escudo", preco = 15}]})
-- cliente satisfeito, o produto entrou em estoque, e a loja reduziu
-- seu caixa em 10 moedas

> runState compraEscudo (Loja 100 [Produto "Escudo" 15])
(True,Loja {caixa = 115, estoque = []})
-- cliente satisfeito, o produto saiu do estoque, e a loja aumentou
-- seu caixa em 15 moedas

> runState vendeEspada (Loja 5 [Produto "Escudo" 15])
(False,Loja {caixa = 5, estoque = [Produto {nome = "Escudo", preco = 15}]})
-- cliente insatisfeito, pois a loja não tinha dinheiro para comprar a
-- espada

> runState compraEscudo (Loja 100 [Produto "Arco" 10])
(False,Loja {caixa = 100, estoque = [Produto {nome = "Arco", preco = 10}]})
-- cliente insatisfeito, pois a loja não vendia escudos
```

[1] Lembre-se de fazer `import Control.Monad.State` no início do seu arquivo

[2] Para utilizar a função `isJust` faça `import Data.Maybe`

Vamos agora ver um exemplo um pouco mais complexo. Considere o cliente `shepard`, que tenta vender uma `"Espada"` por 10 moedas e um `"Escudo"` por 5, e que sai satisfeito se vender qualquer um dos dois. Note que, mesmo que ele não consiga vender a `"Espada"`, ele ainda deve tentar vender o `"Escudo"`. Poderíamos implementá-lo assim:

```
-- I'm Commander Shepard, and This Is My Favorite Store on the Citadel
shepard :: Cliente
shepard = do
  valorEspada <- vender "Espada" 10
  valorEscudo <- vender "Escudo" 5
  return $ valorEspada + valorEscudo > 0
```

E a execução:

```
> runState shepard (Loja 20 [])
(True,Loja {caixa = 0, estoque = [Produto {nome = "Escudo", preco = 5},
                                  Produto {nome = "Espada", preco = 10}]})

> runState shepard (Loja 10 [])
(True,Loja {caixa = 0, estoque = [Produto {nome = "Espada", preco = 10}]})

> runState shepard (Loja 4 [])
(False,Loja {caixa = 4, estoque = []})
```

Você encontra um esqueleto com as funções iniciais necessárias para o seu programa aqui.

Exercício 1

Implemente o cliente `frisk` que tenta vender uma `"Espada"` por 10 moedas e um `"Escudo"` por 5, e só sai satisfeito se vender os dois. Note que, mesmo que ele não consiga vender a Espada, ele ainda deve tentar vender o escudo (mesmo que isso signifique que isso não mude se ele sairá insatisfeito).

Resposta

```
frisk :: Cliente
frisk = do
  valorEspada <- vender "Espada" 10
  valorEscudo <- vender "Escudo" 5
  return $ valorEspada + valorEscudo == 15
```

Exercício 2

Implemente o cliente `loneWanderer` que tenta vender uma `"Espada"` por 10 moedas. Se for bem sucedido, tenta comprar um `"Escudo"`. Ele sai satisfeito se conseguir sair de lá com o Escudo.

Resposta

```haskell
loneWanderer :: E.Cliente
loneWanderer = do
  valorEspada <- vender "Espada" 10
  if valorEspada == 0
    then return False
    else isJust <$> comprar "Escudo"
```

 Exercício 3

Implemente o cliente `dragonborn` que tenta vender o máximo de `"Queijo"` que conseguir, por 3 moedas cada. Ele sairá satisfeito de qualquer forma, independente de quantos queijos vender.

Resposta

```haskell
dragonborn :: Cliente
dragonborn = do
  valor <- vender "Queijo" 3
  if valor > 0
    then dragonborn
    else return True
```

 Exercício 4

Implemente o cliente `geralt` que tenta vender 10 `"Espada"` por 15 moedas cada. Se ele conseguir vender ao menos 6 espadas, ele deve tentar comprar um `"Escudo"`. Ele sai satisfeito se conseguir sair de lá com o Escudo.

Resposta

```haskell
geralt :: Cliente
geralt = do
  valorEspadas <- sum <$> (sequenceA $ replicate 10 $ vender "Espada" 15)
  if valorEspadas >= (15 * 6)
    then isJust <$> comprar "Escudo"
    else return False
```