Programação Funcional (MCCC015-23)
Lista de Exercícios 3 - ADTs

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

26 de julho 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.

Exercício 1

De modo semelhante à função `add`

```
add :: Nat -> Nat -> Nat
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

defina uma função recursiva de multiplicação

```
mult :: Nat -> Nat -> Nat
```

Para o tipo de números naturais:

```
data Nat = Zero | Succ Nat
```

Dica: utilize a função `add` para definir a multiplicação.

- **Resposta**

  ```
  mult :: Nat -> Nat -> Nat
  mult m     Zero = Zero
  mult m (Succ n) = add m (mult m n)
  ```

Exercício 2

O prelude padrão define

```
data Ordering = LT | EQ | GT
```

com a função

```
compare :: Ord a => a -> a -> Ordering
```

que decide se um valor do tipo *ordering* é menor que (LT), igual a (EQ) ou maior que (GT) outro valor. Usando essa função, redefina

```haskell
occurs :: Ord a => a -> Tree a -> Bool
```

para árvores de busca do tipo

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

- **Resposta**

```haskell
occurs :: Ord a => a -> Tree a -> Bool
occurs x (Leaf y)     = x == y
occurs x (Node l y r) = case compare x y of
                          LT -> x `occurs` l
                          EQ -> True
                          GT -> x `occurs` r
```

Exercício 3

Defina uma função

```haskell
flatten :: Tree a -> [a]
```

que retorna uma lista ordenada com os elementos da árvore percorrida em ordem.

- **Resposta**

```haskell
flatten :: Tree a -> [a]
flatten (Leaf x)     = [x]
flatten (Node l x r) = flatten l ++ [x] ++ flatten r
```

Exercício 4

Considere o tipo de árvores binárias

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Uma árvore é dita balanceada se o número de folhas das sub-árvores à esquerda e à direita em cada nó difere em, no máximo, um. As folhas estão trivialmente balanceadas. Defina a função

```haskell
balanced :: Tree a -> Bool
```

que verifica se uma árvore binária está balanceada ou não. Dica: primeiro defina uma função que retorna o número de folhas em uma árvore.

- **Resposta**

```
leaves :: Tree a -> Int
leaves (Leaf _)   = 1
leaves (Node l r) = leaves l + leaves r

balanced :: Tree a -> Bool
balanced (Leaf _)   = True
balanced (Node l r) =
  abs (leaves l - leaves r) <= 1 && balanced l && balanced r
```

Exercício 5

Defina a função

```
balance :: [a] -> Tree a
```

que converte uma lista não vazia em uma árvore balanceada. Dica: primeiro, defina uma função que divide uma lista em duas partes, cujo tamanho difere em no máximo um.

- **Resposta**

```
halve :: [a] -> ([a],[a])
halve xs = splitAt (length xs `div` 2) xs

balance :: [a] -> Tree a
balance [x] = Leaf x
balance xs  = Node (balance ls) (balance rs)
  where (ls, rs) = halve xs
```

Exercício 6

Dada a declaração de tipo

```
data Expr = Val Int | Add Expr Expr
```

defina a função de alta-ordem

```
folde :: (Int -> a) -> (a -> a -> a) -> Expr -> a
```

tal que `folde f g` substitui cada Val em uma expressão pelo retorno da função f, e cada Add pelo retorno de g.

- **Resposta**

```
folde :: (Int -> a) -> (a -> a -> a) -> Expr -> a
folde f g (Val n)   = f n
folde f g (Add x y) = g (folde f g x) (folde f g y)
```

Exercício 7

Usando `folde`, defina a função

```
eval :: Expr -> Int
```

que avalia uma expressão para um valor inteiro, e a função

```
size :: Expr -> Int
```

que calcula o número de valores em uma expressão.

- **Resposta**

```
eval :: Expr -> Int
eval = folde id (+)

size :: Expr -> Int
size = folde (n -> 1) (+)
```

- Nos exercícios que seguem, você não deve usar as listas definidas no `Prelude` do Haskell.

Considere o seguinte ADT, cujo propósito é o de capturar a noção de uma lista:

```haskell
data List a = Nil | a :- List a
infixr 5 :-
```

Eis alguns valores da variedade finita de listas:

```haskell
5 :- 3 :- 2 :- Nil :: Num a => List a
"Wadler" :- "Peyton" :- "Bird" :- Nil :: List String
```

Considere agora o seguinte ADT, cujo propósito é o de modelar um semáforo de trânsito:

```haskell
data Sem = Green | Yellow | Red
    deriving (Eq, Show)
```

Na primeira parte deste exercício, vamos lidar com valores do tipo `List Sem`, por exemplo,

```haskell
Green :- Yellow :- Red :- Red :- Green :- Nil
```

A seguinte função conta o número de semáforos com uma determinada cor em uma lista de semáforos:

```haskell
count :: Sem -> List Sem -> Int
count _ Nil = 0
count x (y :- ys) | x == y = 1 + count x ys
                  | otherwise = count x ys
```

Exercício 8

Escreva uma função

```haskell
next :: Sem -> Sem
```

Que recebe `c :: Sem` e devolve a cor que sucede `c` em um semáforo. Caso você não se lembre: `Green` → `Yellow` → `Red` → `Green`.

- **Resposta**

  ```haskell
  next :: Sem -> Sem
  next Green  = Yellow
  next Yellow = Red
  next Red    = Green
  ```

Exercício 9

Um automóvel precisa atravessar uma sequência de semáforos. Suponha que um automóvel leve uma unidade de tempo para atravessar um semáforo. Ademais, suponha que leve também uma unidade de tempo para cada semáforo mudar de cor. Neste exercício, você vai determinar quanto tempo um automóvel leva para atravessar uma sequência de semáforos. Por exemplo, se a sequência é `Green, Yellow, Red`, então após uma unidade de tempo o automóvel atravessou o semáforo `Green` e, assim, resta agora atravessar a sequência `Red, Green`. O automóvel deve esperar uma unidade de tempo para ocorrer a transição `Red` → `Green` para enfim poder atravessar, gastando mais uma unidade de tempo, o semáforo `Green`. Após estas duas unidades, resta atravessar a sequência `Red`. Logo, mais duas unidades de tempo são necessárias totalizando, assim, cinco unidades de tempo. Escreva uma função

```
timeList :: List Sem -> Int
```

que recebe `xs :: List Sem` e devove o tempo que um automóvel leva para atravessar `xs`.

- **Resposta**

```
-- Função mapl, equivalente a map para listas comuns
mapl :: (a -> b) -> List a -> List b
mapl _ Nil = Nil
mapl f (x :- xs) = f x :- mapl f xs

-- Versão sem recursão de cauda
timeList :: List Sem -> Int
timeList Nil       = 0
timeList (x :- xs) =
    1 + timeList (mapl next xs2)
  where
    xs2 = if x == Red then x :- xs else xs

-- Versão com recursão de cauda
timeList2 :: List Sem -> Int
timeList2 = go (0, id)
  where
    foldFun (n, f) c
```

```
      | f c == Red = (n + 2, next . next . f)
      | otherwise  = (n + 1, next . f)

  go acc Nil       = fst acc
  go acc (x :- xs) = go (foldFun acc x) xs
```

Exercício 10

Escreva uma versão da função `foldl` para valores do tipo `List a`:

```
redl :: (b -> a -> b) -> b -> List a -> b
```

- **Resposta**

```
redl :: (b -> a -> b) -> b -> List a -> b
redl _ b Nil       = b
redl f b (x :- xs) = redl f (f b x) xs
```

Exercício 11

Escreva uma nova versão versão da função do Exercício 9 usando a função `redl`.

- **Resposta**

```
timeListRedl :: List Sem -> Int
timeListRedl  = fst . redl foldFun (0, id)
  where
    foldFun (n, f) c
      | f c == Red = (n + 2, next . next . f)
      | otherwise  = (n + 1, next . f)
```

Vamos agora resolver o mesmo problema em uma estrutura um pouco mais geral. Considere o seguinte ADT:

```
data BT a = BEmpty | BNode a (BT a) (BT a)
    deriving Show
```

cujo propósito é o de modelar uma árvore binária. Para facilitar, vamos definir a função:

```
bleaf :: a -> BT a
bleaf x = BNode x BEmpty BEmpty
```

O interesse aqui reside em valores do tipo `BT Sem`.
Por exemplo,

```
bt :: BT Sem
bt = BNode Green
    (
        BNode Green (bleaf Red) (bleaf Green)
    )
    (
        BNode Yellow (bleaf Red) (bleaf Green)
    )
```

Exercício 12

Para cada folha `f` de uma árvore binária há um único caminho – sequência de nós – que leva de `f` até a raiz da árvore. Em uma árvore binária de semáforos tal caminho determina uma sequência de semáforos; tal sequência, denotada por s_f, é dita *ancorada* em f. Escrevemos $\tau(s_f)$ para denotar o tempo que leva para um automóvel atravessar s_f.

Escreva uma função

```
timeBT :: BT Sem -> Int
```

que recebe `t :: BT Sem` e devolve

$$\min\{\tau(s_f) \mid f \text{ é folha de } t\}.$$

Este último número é denotado por $\mu(t)$. Por exemplo, a chamada

```
timeBT bt
```

devolve 4.

- **Resposta**

```haskell
-- fold associativo a esquerda que toma o primeiro elemento da
-- lista como acumulador inicial
redl1 :: (a -> a -> a) -> List a -> a
redl1 f (x :- xs) = redl f x xs

-- Devolve o elemento mínimo de uma lista
minimuml :: Ord a => List a -> a
minimuml = redl1 min

-- Dada uma árvore binária de a, devolve todos os caminhos das folhas
-- à raiz. Para evitar concatenações de listas, esta implementação
-- utiliza uma variável acumuladora o que acaba aumentando
-- ligeiramente a complexidade da solução.
allPaths :: BT a -> List (List a)
allPaths = go (Nil, False, Nil)
  where
    --    (atual, incluir, caminhos acumulados)
    go :: (List a, Bool, List (List a)) -> BT a -> List (List a)
    go (curr, True,  ps) BEmpty        = curr :- ps
    go (_,    False, ps) BEmpty        = ps
    go (curr, _,     ps) (BNode x l r) = psr
      where
        -- para evitar repetições, só adiciona uma das folhas
        -- terminadoras
        psl = go (x :- curr, True,  ps)  l
        psr = go (x :- curr, False, psl) r

timeBT :: BT Sem -> Int
timeBT = minimuml . mapl timeListRedl . allPaths
```

Exercício 13

Escreva uma função

```haskell
bestBT :: BT Sem -> List Sem
```

que recebe `t :: BT Sem` e devolve uma lista de semáforos obtida de uma sequência s_f ancorada em alguma folha `f` de `t` tal que o tempo para atravessar s_f é $\mu(t)$. Por exemplo, a chamada

```
timeBT bt
```

pode devolver (note que pode haver várias respostas válidas em caso de empate):

```
Red :- Yellow :- Green :- Nil
```

- **Resposta**

```
-- Devolve a cabeça da lista passada como parâmetro
headl :: List a -> a
headl (x :- _) = x

-- Dado um par de listas, devolve uma lista de pares contendo os
-- valores das listas recebidas
zipl :: List a -> List b -> List (a, b)
zipl Nil _ = Nil
zipl _ Nil = Nil
zipl (x :- xs) (y :- ys) = (x, y) :- zipl xs ys

-- Filtra os elementos da lista recebida como parâmetro de acordo com
-- o predicado fornecido
filterl :: (a -> Bool) -> List a -> List a
filterl _ Nil = Nil
filterl p (x :- xs)
  | p x       = x :- filterl p xs
  | otherwise = filterl p xs

-- Concatena duas listas
appendl :: List a -> List a -> List a
appendl xs ys = redl (flip (:-)) ys xs

-- Ordena uma lista utilizando como critério a função de comparação
-- fornecida
sortByl :: (a -> a -> Bool) -> List a -> List a
sortByl _ Nil = Nil
sortByl f (x :- xs) = appendl menores (x :- maiores)
  where
    menores = filterl (`f` x) xs
    maiores = filterl (\a -> not $ f a x) xs
```

```haskell
bestBT :: BT Sem -> List Sem
bestBT xs = snd . headl . sortByl (\(t0,_) (t1, _) -> t0 <= t1) $ zipl ts ps
  where
    ps = allPaths xs
    ts = mapl timeListRedl ps
```

Finalmente, considere agora o seguinte ADT

```haskell
data Tree a = TEmpty | TNode a List (Tree a)
```

cujo propósito é o de representar uma árvore.
É conveniente, para simplificar, definir a função:

```haskell
tleaf :: a -> Tree a
tleaf x = TNode x Nil
```

Por exemplo, eis uma árvore de semáforos:

```haskell
tree :: Tree Sem
tree = TNode Green
       ((TNode Green (tleaf Red :- tleaf Green :- Nil)) :-
       (TNode Yellow
         (TNode Red (tleaf Red :- tleaf Green :- Nil) :- tleaf Yellow :- Nil)) :-
       (TNode Green (tleaf Green :- tleaf Green :- Nil)) :- Nil)
```

Exercício 14

Repita os últimos dois exercícios definindo funções que recebem árvores em vez de árvores binárias.

- **Resposta**

```haskell
-- Transforma uma lista de listas em uma lista simples contendo todos
-- os elementos
concatl :: List (List a) -> List a
concatl Nil = Nil
concatl (xs :- xss) = xs `appendl` concatl xss

-- mapeia uma função qque devolve uma lista em uma lista e devolve a
-- concatenação dos resultados
concatMapl :: (a -> List b) -> List a -> List b
```

```haskell
concatMapl f = concatl . mapl f

-- Dada uma árvore de a, devolve todos os caminhos das folhas
-- à raiz.
allPathsT :: Tree Sem -> List (List Sem)
allPathsT = go Nil
  where
    go :: List Sem -> Tree Sem -> List (List Sem)
    go path TEmpty = path :- Nil
    go path (TNode x Nil) = (x :- path) :- Nil
    go path (TNode x cs) = concatMapl (go (x :-path)) cs

timeT :: Tree Sem -> Int
timeT = minimuml . mapl timeListRedl . allPathsT

bestT :: Tree Sem -> List Sem
bestT xs = snd . headl . sortByl (\(t0,_) (t1, _) -> t0 <= t1) $ zipl ts ps
  where
    ps = allPathsT xs
    ts = mapl timeListRedl ps
```