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.

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

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

para árvores de busca do tipo

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

Exercício 3

Defina uma função

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

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

Exercício 4

Considere o tipo de árvores binárias

```
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

```
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.

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.

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.

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.

- 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:

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

Eis alguns valores da variedade finita de listas:

```
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:

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

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

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

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

```
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

```
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`.

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`.

Exercício 10

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

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

Exercício 11

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

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

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

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

```
timeBT bt
```

devolve 4.

Exercício 13

Escreva uma função

```
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
```

Finalmente, considere agora o seguinte ADT

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

cujo propósito é o de representar uma árvore.

É conveniente, para simplificar, definir a função:

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

Por exemplo, eis uma árvore de semáforos:

```
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.