Universidade Federal do ABC
MCTA016-13 - Paradigmas de Programação
2023.Q2

Lista quinzenal 04

Prof. Emílio Francesquini e Prof. Mário Leston Rey

16 de julho de 2023

Atenção!

- O seu repositório deverá conter, **além** da implementação das funções abaixo, o seguinte:
  - Um projeto criado com o Stack que compile. Códigos que não compilarem receberão nota 0.
  - O seu projeto deverá ter uma função `main` (executada automaticamente com `stack run` que execute cada função pedida abaixo com os exemplos dados.
  - Um arquivo de teste (tipicamente `Spec.hs`) com pelo menos um teste de unidade para cada exercício. (Veja as listas anteriores para saber como criar um arquivo de testes que são executados automaticamente via `stack test`).
  - Pelo menos **2** testes de propriedade utilizando o QuickCheck (que devem ser rodados juntamente com os testes de unidade via `stack test`) para pelo menos 2 dos exercícios abaixo (**totalizando 4 testes do QuickCheck no mínimo**).
- 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:

```
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 1

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 2

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 3

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

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

Exercício 4

Escreva uma nova versão versão da função do Exercício 2 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 5

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.

Exercício 6

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
```

devolve

```
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
        (TNode Green (tleaf Green :- tleaf Green :- Nil)) :- Nil)
```

Exercício 7

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