Programação Funcional (MCCC015-23)
Lista de Exercícios 4

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

14 de agosto de 2024

Exercícios com Folds[1]

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.

---

[1] Estes exercícios foram preparados tomando como base aqueles elaborados pelo Prof. Antoni Diller da Universidade de Birmingham.

Exercício 1

Usando a função de alta ordem `foldr`, defina uma função `sumsq` que receba um inteiro n como argumento e retorne a soma dos quadrados dos primeiros `n` inteiros. Ou seja:

$$\text{sumsq } n = 1^2 + 2^2 + 3^2 + ... + n^2$$

Não use a função `map`.

Resposta

```
sumsq1 :: Integral a => a -> a
sumsq1 n = foldr (\x acc -> x * x + acc) 0 [1..n]
```

Exercício 2

Defina `length`, que retorna o número de elementos em uma lista, usando `foldr`. Redefina-a usando `foldl`.

Resposta

```
length0 :: [Int] -> Int
length0 = foldr (\_ acc -> 1 + acc) 0

length1 :: [Int] -> Int
length1 = foldl (\acc _ -> acc + 1) 0
```

Exercício 3

Defina `minlist`, que retorna o menor inteiro em uma lista não vazia de inteiros, usando `foldr1`. Redefina-a usando `foldl1`.

Resposta

```
minlist0 :: [Int] -> Int
minlist0 = foldr1 min

minlist1 :: [Int] -> Int
minlist1 = foldl1 min
```

Exercício 4

Defina `reverse`, que inverte a ordem dos elementos de uma lista, usando `foldr`.

Resposta

```
reverse0 :: [a] -> [a]
reverse0 = foldr (\x acc -> acc ++ [x]) []
```

Exercício 5

Usando `foldr`, defina uma função `remove` que receba duas strings como argumentos e remova da segunda string todas as letras que aparecem na primeira. Por exemplo, `remove "first" "second" == "econd"`.

Resposta

```haskell
remove :: String -> String -> String
remove rm = foldr (\x acc -> if x `elem` rm then acc else x : acc)  ""
```

Exercício 6

Defina `filter` usando `foldr`. Redefina `filter` novamente usando `foldl`.

Resposta

```haskell
filter0 :: (a -> Bool) -> [a] -> [a]
filter0 f = foldr (\x acc -> if f x then x : acc else acc) []

-- nessa implementação alternativa em vez de usar ++ que aumenta o
-- tempo de execução para O(n^2) usamos um reverse ao final o que
-- garante que o tempo continue linear.
filter1 :: (a -> Bool) -> [a] -> [a]
filter1 f = reverse . foldl (\acc x -> if f x then x : acc else acc) []
```

Exercício 7

A função `remdups` remove duplicatas adjacentes de uma lista. Por exemplo:

```haskell
remdups [1, 2, 2, 3, 3, 3, 1, 1] == [1, 2, 3, 1]
```

Defina `remdups` usando `foldr`. Dê outra definição usando `foldl`.

Resposta

```haskell
condCons :: Eq a => a -> [a] -> [a]
condCons x0 [] = [x0]
condCons x0 xs@(x1:_)
  | x0 == x1  = xs
  | otherwise = x0 : xs

remdups0 :: Eq a => [a] -> [a]
remdups0 = foldr condCons []

remdups1 :: Eq a => [a] -> [a]
remdups1 = reverse . foldl (flip condCons) []
```

Exercício 8

A função `inits` retorna a lista de todos os segmentos iniciais de uma lista. Assim, `inits "ate" = [[], "a", "at", "ate"]`. Defina `inits` usando `foldr`.

Resposta

```haskell
inits0 :: [a] -> [[a]]
inits0 = foldr (\x acc -> [] : fmap (x:) acc) [[]]

inits1 :: [a] -> [[a]]
inits1 xs = foldr (\x acc -> take x xs : acc) [] [0 .. length xs]
```

Exercício 9

Usando `foldl`, defina `approxe n` tal que

$$\text{approxe } n = \sum_{i=0}^{n} \frac{1}{i!}$$

Por exemplo:

$$\begin{aligned} \text{approxe } 4 &= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} \\ &= 1 + 1 + 0,5 + 0,16666... + 0,0416666... \\ &= 2,7083333... \end{aligned} \tag{1}$$

Resposta

```haskell
approxe ::  (Integral i, Fractional f) => i -> f
approxe n = foldl (\acc i -> acc + recip (fact i)) 0 [0 .. n]
  where
    fact i = fromIntegral $ product [2 .. i]
```

Exercício 10

Usando `scanl`[2], defina uma função `sae` (aproximações sucessivas de `e`) tal que:

$$\text{sae } n = \left[\sum_{i=0}^{0} \frac{1}{i!}, \sum_{i=0}^{1} \frac{1}{i!}, \sum_{i=0}^{2} \frac{1}{i!}, \ldots, \sum_{i=0}^{n} \frac{1}{i!}\right]$$

Resposta

```haskell
sae :: (Integral i, Fractional f) => i -> [f]
sae n = scanl (\acc i -> acc + recip (fact i)) 1 [1 .. n - 1]
  where
    fact i = fromIntegral $ product [2 .. i]
```

[2] https://hackage.haskell.org/package/base-4.20.0.1/docs/Prelude.html#v:scanl

Exercício 11

Defina `iterate`[3] usando `scanl`.

Resposta

```
iterate0 :: (t -> t) -> t -> [t]
iterate0 f v = scanl (flip ($)) v (repeat f)
```

Exercício 12

Defina `shift`, que coloca o primeiro elemento de uma lista no final. Assim, `shift [1, 2, 3] = [2, 3, 1]` e `shift "eat"= "ate"`. Usando `foldl` e `shift`, defina `rotate`, que produz todas as rotações de uma lista. Por exemplo, `rotate [1, 2, 3] = [[1, 2, 3], [2, 3, 1], [3, 1, 2]]`.

Resposta

```
shift :: [a] -> [a]
shift [] = []
shift (x:xs) = xs <> [x]

rotate1 :: [a] -> [[a]]
rotate1 xs = reverse $ foldl (\acc f -> f xs : acc) [] shfts
  where
    shfts = take (length xs) (iterate0 (shift .) id)
```

Exercício 13

A função `add` pode ser definida em termos de:

```
succ i = i + 1
pred i = i - 1
```

com as equações:

```
add i 0 = i
add i j = succ (add i (pred j))
```

(a) Dê uma definição semelhante de `mult` que usa apenas `add` e `pred`. Dê uma definição de `exp` que usa apenas `mult` e `pred`. Qual é a próxima função nessa sequência?

(b) A função de dobra em inteiros `foldi` pode ser definida como segue:

```
foldi :: (a -> a) -> a -> Int -> a
foldi f q 0 = q
foldi f q i = f (foldi f q (pred i))
```

Defina as funções `add`, `mult` e `exp` em termos de `foldi`.

(c) Defina as funções `fat` (fatorial) e `fib` (números de Fibonacci) usando a função `foldi`.

[3] https://hackage.haskell.org/package/base-4.20.0.1/docs/Prelude.html#v:iterate

Resposta

```haskell
mult :: Int -> Int -> Int
mult _ 0 = 0
mult x 1 = x
mult x y = x + mult x (pred y)

-- A próxima função da sequência é exp.

add :: Int -> Int -> Int
add = foldi succ

mult1 :: Int -> Int -> Int
mult1 x = foldi (add x) 0

exp1 :: Int -> Int -> Int
exp1 b = foldi (mult1 b) 1

fat :: Int -> Int
fat = fst . foldi passo (1, 1)
  where
    passo (acc, i) = (mult1 acc i, succ i)

fib :: Int -> Int
fib = fst . foldi passo (0, 1)
  where
    passo (atual, anterior) = (add atual anterior, atual)
```