Programação Funcional (MCCC015-23)
Lista de Exercícios 2 - Haskell Básico
Gabarito

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

29 de julho de 2024

1. Coloque parênteses nas seguintes expressões:

```
2^3*4
2*3+4*5
2+3*4^5
2+3/4-5^6*7
```

   - Resposta

```
(2^3)*4
(2*3)+(4*5)
2+(3*(4^5))
(2+(3/4))-((5^6)*7)
```

2. Sem utilizar qualquer ajuda, determine o valor e o tipo retornado por essas expressões. Em seguida, utilize o *ghci* para confirmar a resposta:

```
(* 9) 6

head [(0,"doge"),(1,"kitteh")]

head [(0 :: Integer ,"doge"),(1,"kitteh")]

if False then True else False

length [1, 2, 3, 4, 5]
```

```
length [1, 2, 3, 4] > length "TACOCAT"
```

- Resposta

```
> (* 9) 6 :: Num a => a
54
> head [(0,"doge"),(1,"kitteh")] :: Num a => (a, [Char])
(0,"doge")
> head [(0 :: Integer ,"doge"),(1,"kitteh")] :: (Integer, [Char])
(0,"doge")
> if False then True else False :: Bool
False
> length [1, 2, 3, 4, 5] :: Int
5
> length [1, 2, 3, 4] > length "TACOCAT" :: Bool
False
```

3. Defina uma função para seguinte assinatura:

```
f :: (a, b) -> (c, d) -> ((b, d), (a, c))
```

- Resposta

```
f :: (a, b) -> (c, d) -> ((b, d), (a, c))
f x y = ((snd x, snd y), (fst x, fst y))
```

4. Defina uma função

```
palindromo :: (Eq a) => [a] -> Bool
```

que verifica se uma string (ou lista) é palíndroma, utilizando a função `reverse`.

- Resposta

```
palindromo :: (Eq a) => [a] -> Bool
palindromo x = x == reverse x
```

5. Mostre que a seguinte função curried pode ser formalizada em termos de expressões lambda:

```
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

- Resposta

```
mult :: Int -> Int -> Int -> Int
mult = x -> (y -> (z ->  x*y*z))
```

6. Mostre como o operador || pode ser definido de quatro modos diferentes usando pattern matching.

   - Resposta

```
-- 1
False || False = False
False || True  = True
True  || False = True
True  || True  = True
-- 2
False || False = False
_     || _     = True
-- 3
False || b = b
True  || _ = True
-- 4
b || c | b == c    = b
       | otherwise = True
```

7. Sem usar outras bibliotecas, funções ou operadores, mostre que a definição por pattern matching de &&

```
True && True = True
_    && _    = False
```

   pode ser formalizada utilizando duas expressões condicionais (*if*) aninhadas.

   - Resposta

```
b `ee` c = if b then (if c then True else False) else False
```

8. Faça o mesmo do exercício anterior para essa definição alternativa de &&:

```
True  && b = b
False && _ = False
```

usando dessa vez uma única expressão condicional.

- Resposta

```
b `ee` c = if c then b else False
```

9. Defina a única função possível para a assinatura

```
c :: a -> b -> a
```

- Resposta

```
c :: a -> b -> a
c x y = x
```

10. Defina a única função possível para a assinatura

```
co :: (b -> c) -> (a -> b) -> a -> c
```

- Resposta

```
co :: (b -> c) -> (a -> b) -> a -> c
co f g x = f (g x)
-- chame (b -> c) de f, (a -> b) de g

-- função composta
co f g x = f (g x)
-- notação infixa com ponto, semelhante a f   g
co f g = f.g
-- notação prefixa
co fg = (.) fg
```

11. Defina uma função

```
penultimo :: [a] -> a
```

que devolve o penúltimo elemento de uma lista, apresentando uma mensagem de erro nos casos de lista vazia e lista com apenas um elemento.

- Resposta

```haskell
penultimo :: [a] -> a
penultimo []       = error "Lista vazia"
penultimo [x]      = error "Lista com um elemento"
penultimo (x:_:[]) = x
penultimo (_:x:xs) = penultimo (x:xs)
```

12. Defina uma função

    ```haskell
    maximoLocal :: [Int] -> [Int]
    ```

    que devolve uma lista com os máximos locais de uma lista de inteiros. Um máximo local é um elemento maior que seu antecessor e que seu sucessor. Por exemplo, em [1,3,4,2,5] 4 é um máximo local, mas 5 não, pois não possui sucessor.

    - Resposta

      ```haskell
      maximoLocal :: [Int] -> [Int]
      maximoLocal (x1:x2:x3:xs)
        | x2 > x1 && x2 > x3 = x2 : maximoLocal (x3:xs)
        | otherwise          = maximoLocal (x2:x3:xs)
      maximoLocal _ = []
      ```

13. Usando compreensão de listas, defina a função

    ```haskell
    perfeitos :: Int -> [Int]
    ```

    que recebe um inteiro n e retorna uma lista dos números perfeitos até n. Um número perfeito é igual à soma de seus fatores, excluindo a si mesmo. O número 28 é perfeito, pois $1 + 2 + 4 + 7 + 14 = 28$. Exemplo:

    ```
    > perfeitos 500
    [6,28,496]
    ```

    - Resposta

      ```haskell
      fatores :: Int -> [Int]
      fatores n = [x | x <- [1..(n-1)], n `rem` x == 0]

      perfeitos :: Int -> [Int]
      perfeitos n = [x | x <- [1..n], x == (sum $ fatores x)]
      ```

14. Defina a função

```
produtoEscalar :: Num a => [a] -> [a] -> a
```

que devolve o produto escalar de dois vetores, usando compreensão de listas.

- Resposta

```
produtoEscalar :: Num a => [a] -> [a] -> a
produtoEscalar u v = sum [i * j | (i,j) <- zip u v]
```

15. Escreva uma função recursiva

```
palindromo :: [Int] -> Bool
```

que verifica se os elementos da lista formam um palíndromo.

- Resposta

```
palindromo :: [Int] -> Bool
palindromo [] = True
palindromo [x] = True
palindromo x
  | head x == last x = palindromo (tail $ init x)
  | otherwise = False
```

16. Defina uma função

```
ordenaListas :: (Num a, Ord a) => [[a]] -> [[a]]
```

que ordene uma lista de listas pelo tamanho de suas sublistas.

- Resposta

```
ordenaListas :: (Num a, Ord a) => [[a]] -> [[a]]
ordenaListas []     = []
ordenaListas (x:xs) = ordenaListas menores ++ [x] ++ ordenaListas maiores
    where
        menores = [a | a <- xs, length a <= length x]
        maiores = [b | b <- xs, length b >  length x]
```

17. Mostre como a compreensão de lista

```
coord :: [a] -> [a] -> [(a,a)]
coord x y = [(i,j) | i <- x, j <- y]
```

que possui duas funções geradoras, pode ser redefinida com duas listas de compreensão aninhadas, cada uma contendo uma única função geradora.

- Resposta

```
coord' :: [a] -> [a] -> [(a,a)]
coord' x y = concat [[(i,j) | j <- y] | i <- x]
```

18. O algoritmo de Luhn para a verificação dos dígitos de um cartão de crédito segue os seguintes passos:

    (a) considere cada dígito como um número

    (b) a partir da direita, dobre os números alternadamente, começando pelo penúltimo

    (c) some todos os dígitos dos números

    (d) se o total for divisível por 10, o número do cartão é válido.

    Em seguida:

    (a) Defina as funções

    ```
    digitosRev :: Int -> [Int]
    ```

    que converte um inteiro em uma lista contendo seus dígitos na ordem reversa.

    - Resposta

    ```
    digitosRev :: Int -> [Int]
    digitosRev n
      | n <= 0    = []
      | otherwise = (n `rem` 10) : digitosRev (n `div` 10)
    ```

    (b) Escreva a função

    ```
    dobroAlternado :: [Int] -> [Int]
    ```

    que recebe uma lista de números e dobra a partir da esquerda o segundo, quarto etc, elemento, devolvendo uma lista atualizada. Note que por termos escrito a função anterior para retornar os dígitos invertidos, podemos fazer essa operação da esquerda ao invés de da direita conforme descrição do algoritmo. Por exemplo, para [3,5,6,4] a a saída é [3,10,6,8].

- Resposta

```
dobroAlternado :: [Int] -> [Int]
dobroAlternado []         = []
dobroAlternado [x]        = [] ++ [x]
dobroAlternado (x:y:xss) = x : (2*y) : dobroAlternado xss
```

(c) Defina a função

```
somaDigitos :: [Int] -> Int
```

que soma todos os dígitos da lista de inteiros. Com o uso função anterior, alguns números possuem dois dígitos, que precisam ser somados individualmente. Exemplo: [6,5,12,4] = 6 + 5 + 1 + 2 + 4 = 18

- Resposta

```
somaDigitos :: [Int] -> Int
somaDigitos x = somaDigitos' x 0
  where
    somaDigitos' [] s                 = s
    somaDigitos' (x:xs) s | x > 9     = somaDigitos' xs (x+s-9)
                          | otherwise = somaDigitos' xs (x+s)
```

(d) Utilize as funções criadas anteriormente para definir a função

```
luhn :: Int -> Bool
```

que verifica se o número é uma sequência válida para um cartão de crédito. Exemplos:

```
> luhn 1784
True
> luhn 4783
False
> luhn 4012888888881881
True
> luhn 4012888888881882
False
```

- Resposta

```
luhn :: Int -> Bool
luhn n
  | (luhn' n) `rem` 10 == 0 = True
  | otherwise = False
  where luhn' n = somaDigitos $ dobroAlternado $ digitosRev n
```

19. Escreva a função McCarthy91 em Haskell

```
mc91 :: Integral a => a -> a
```

que é definida por:

```
MC(n) =  n - 10, se n > 100;
      =  MC (MC (n + 11)), caso contrário.
```

- Resposta

```
mc91 :: Integral a => a -> a
mc91 n
  | n > 100   = n - 10
  | otherwise = mc91 $ mc91 (n + 11)
```

20. Escreva uma função recursiva

```
elem' :: Eq a => a -> [a] -> Bool
```

que verifica se um valor está contido numa lista

- Resposta

```
elem' :: Eq a => a -> [a] -> Bool
elem' _ []     = False
elem' v (x:xs)
  | x == v    = True
  | otherwise = elem' v xs
```

21. Defina a função recursiva

```
euclid :: Int -> Int -> Int
```

que implementa o algoritmo de Euclides para calcular o máximo divisor comum de dois inteiros não-negativos: se dois números são iguais, o resultado é o próprio número, senão, o menor é subtraído do maior e o processo se repete. Exemplo:

```
> euclid 6 27
3
```

- Resposta

```haskell
euclid :: Int -> Int -> Int
euclid n m
  | n < 0 || m < 0 = error "Número negativo"
  | n == m         = n
  | n < m          = euclid n (m - n)
  | otherwise      = euclid (n - m) m
```

22. Defina a função recursiva

    ```haskell
    concat' :: [[a]] -> [a]
    ```

    que recebe uma lista de listas e devolve uma única lista contendo os elementos das listas internas.

    - Resposta

      ```haskell
      concat' :: [[a]] -> [a]
      concat'[]           = []
      concat'([]:xss)     = concat' xss
      concat'((x:xs):xss) = x : concat' (xs:xss)
      ```

23. Defina a função

    ```haskell
    intersperse' :: a -> [a] -> [a]
    ```

    que recebe um char separador e uma lista de chars (ou string) e devolve uma string com os elementos intercalados entre o separador. Exemplo:

    ```
    > intersperse' '-' "abcde"
    "a-b-c-d-e"
    ```

    - Resposta

      ```haskell
      intersperse' :: a -> [a] -> [a]
      intersperse' _ []       = []
      intersperse' _ [x]      = [x]
      intersperse' sep (x:xs) = x : sep : intersperse' sep xs
      ```

24. Escreva a função

    ```haskell
    wordNumber :: Char -> Int -> [Char]
    ```

que recebe um char separador e um inteiro, e devolve uma string com os nomes dos números separados por um char sep. Você poderá utilizar as funções concat' e interspace' definidas anteriormente. Outras funções auxiliares podem ser definidas, como transformar um inteiro em uma lista de dígitos, e substituir cada dígito pelo seu correspondente por extenso. Exemplo:

```
> wordNumber '-' 123
"um-dois-três"
```

- Resposta

```
numToList :: Int -> [Int]
numToList n
  | n == 0 = []
  | otherwise =  numToList (n `div` 10) ++ [(n `mod` 10)]

digitsName :: [Int] -> [String]
digitsName [] = []
digitsName (x:xs)
  | x == 1    = "um"     : digitsName xs
  | x == 2    = "dois"   : digitsName xs
  | x == 3    = "tres"   : digitsName xs
  | x == 4    = "quatro" : digitsName xs
  | x == 5    = "cinco"  : digitsName xs
  | x == 6    = "seis"   : digitsName xs
  | x == 7    = "sete"   : digitsName xs
  | x == 8    = "oito"   : digitsName xs
  | x == 9    = "nove"   : digitsName xs
  | x == 0    = "zero"   : digitsName xs
  | otherwise = []

wordNumber :: Char -> Int -> [Char]
wordNumber sep n = concat' $ intersperse' [sep] $ digitsName $ numToList n
```

25. Defina uma função recursiva

```
merge :: Ord a => [a] -> [a] -> [a]
```

que recebe duas listas ordenadas e devolve uma única lista ordenada. Exemplo:

```
> merge [2,5,6] [1,3,4]
[1,2,3,4,5,6]
```

- Resposta

```haskell
merge :: Ord a => [a] -> [a] -> [a]
merge [] [] = []
merge [] ys  = ys
merge xs []  = xs
merge (x:xs) (y:ys)
  | x <= y      = x : merge xs (y:ys)
  | otherwise   = y : merge (x:xs) ys
```

26. Utilizando a função merge implementada anteriormente, defina

```haskell
msort :: Ord a => [a] -> [a]
```

que implementa o algoritmo merge sort. Para auxiliar, defina

```haskell
halve :: [a] -> ([a],[a])
```

que divide uma lista em duas de mesmo tamanho ou com uma diferença máxima de um elemento.

- Resposta

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

msort :: Ord a => [a] -> [a]
msort []     = []
msort (x:[]) = [x]
msort (x:xs) = merge (msort left) (msort right)
  where
    (left, right) = halve (x:xs)
```

27. Defina duas funções

```haskell
quotRem' :: Integral a => a -> a -> (a, a)
divMod' :: Integral a => a -> a -> (a, a)
```

que implementam a divisão de inteiros como uma subtração recursiva, retornando uma tupla com o quociente da divisão e o resto. Trate o caso de divisão por zero com error.

Existem duas maneiras de se fazer a divisão com números negativos: arredondando para zero (como a função quotRem faz) ou para o infinito negativo (como feito pela função divMod). Para resolver o exercício, comece por quotRem' e use pattern matching listando todos os possíveis casos de dividendos (n), divisores (d), quociente(q) e resto (r) quanto aos sinais:

```
n = 0 <=> q = 0, r = 0
d = 0 => error
n > 0, d > 0 <=> q > 0, r > 0
n < 0, d < 0 <=> q > 0, r < 0
n < 0, d > 0 <=> q < 0, r < 0
n > 0, d < 0 <=> q < 0, r > 0
```

Para definir divMod', utilize quotRem', observando que para n e d com sinais iguais, divMod' devolve os mesmos resultados de quotRem'. Quando os sinais de n e d são opostos, deve-se subtrair 1 do quociente e somar d ao resto.

Por fim, compare seus resultados com as funções quotRem e divMod originais.

- Resposta

```haskell
divPos :: Integral a => a -> a -> a -> (a, a)
divPos n d c
  | n < d     = (c, n)
  | otherwise = divPos (n-d) d (c+1)

negateQ :: Integral a => (a, a) -> (a, a)
negateQ (q, r) = (-q, r)

negateR :: Integral a => (a, a) -> (a, a)
negateR (q, r) = (q, -r)

negateB :: Integral a => (a, a) -> (a, a)
negateB (q, r) = (-q, -r)
```

```haskell
quotRem' :: Integral a => a -> a -> (a, a)
quotRem' _ 0                = error "Divisão por zero!"
quotRem' 0 _                = (0,0)
quotRem' n d | n > 0 && d > 0 = divPos n d 0
             | n < 0 && d < 0 = negateR (divPos (-n) (-d) 0)
             | n < 0 && d > 0 = negateB (divPos (-n) d 0)
             | n > 0 && d < 0 = negateQ (divPos n (-d) 0)

divMod' :: Integral a => a -> a -> (a, a)
divMod' n d | signum r == - signum d = (q-1, r+d)
            | otherwise              = (q, r)
            where (q, r) = quotRem' n d
```

28. Defina a função

```haskell
skips :: [a] -> [[a]]
```

cuja entrada é uma lista e a saída é uma lista de listas. A primeira lista interna é igual à lista de entrada. A segunda lista contém cada segundo elemento da lista de entrada e a enésima lista contém cada enésimo elemento da lista de entrada. Exemplos:

```
> skips "ABCD"
["ABCD", "BD", "C", "D"]
> skips "hello!"
["hello!", "el!", "l!", "l", "o", "!"]
> skips [1]
[[1]]
> skips [True,False]
[[True,False], [False]]
> skips []
[]
```

- Resposta

```haskell
takeNth :: Int -> [a] -> [a]
takeNth _ [] = []
takeNth n xs = go n xs
  where
    maxSize = length xs
    go count xs
      | n > maxSize || count > maxSize = []
```

```
        | otherwise = xs !! (count-1) : go (count + n) xs

skips :: [a] -> [[a]]
skips [] = []
skips xs = skips' xs 1
  where
    maxSize = length xs
    skips' xs count
      | count > maxSize = []
      | otherwise = takeNth count xs : skips' xs (count + 1)
```

29. Sem olhar as definições do Prelude padrão, defina as seguintes funções de alta-ordem:

```
-- Verifica se todos elementos da lista satisfazem um predicado
all' :: (a -> Bool) -> [a] -> Bool

-- Verifica se pelo menos um dos elementos da lista satisfazem um predicado
any' :: (a -> Bool) -> [a] -> Bool

-- Seleciona os elementos da lista enquanto eles satisfizerem um predicado
takeWhile' :: (a -> Bool) -> [a] -> [a]

-- Remove os elementos da lista enquanto eles satisfizerem um predicado
dropWhile' :: (a -> Bool) -> [a] -> [a]
```

- Resposta

```
all' :: (a -> Bool) -> [a] -> Bool
all' p xs = and (map p xs)
-- verifique que podemos definir tb como:
-- all' p = and . map p

any' :: (a -> Bool) -> [a] -> Bool
any' p xs = or (map p xs)
-- verifique que podemos definir tb como:
-- any' p = or . map p

takeWhile' :: (a -> Bool) -> [a] -> [a]
takeWhile' _ [] = []
takeWhile' p (x:xs) | p x       = x : takeWhile' p xs
                    | otherwise = []
```

```haskell
dropWhile' :: (a -> Bool) -> [a] -> [a]
dropWhile' _ [] = []
dropWhile' p (x:xs) | p x       = dropWhile' p xs
                    | otherwise = (x:xs)
```

30. Defina uma função

    ```haskell
    altMap :: (a -> b) -> (a -> b) -> [a] -> [b]
    ```

    que aplica alternadamente as duas funções recebidas como argumento em uma lista. Exemplo:

    ```
    > altMap (+10) (+100) [0,1,2,3,4]
    [10,101,12,103,14]
    ```

    - Resposta

      ```haskell
      altMap :: (a -> b) -> (a -> b) -> [a] -> [b]
      altMap _ _ []     = []
      altMap f g (x:xs) = f x : altMap g f xs
      ```

31. Usando *folding*, defina a função

    ```haskell
    dec2int :: [Int] -> Int
    ```

    que converte uma lista de inteiros em um inteiro. Exemplo:

    ```
    > dec2int [2,3,4,5]
    2345
    ```

    - Resposta

      ```haskell
      dec2int :: [Int] -> Int
      dec2int = foldl (\x y -> 10 * x + y) 0
      ```

32. Utilize *folding* para definir as seguintes funções em Haskell:

    ```haskell
    -- retorna True se pelo menos um booleano da lista for True
    or' :: [Bool] -> Bool

    -- inverte os elementos de uma lista
    reverse' :: [a] -> [a]

    -- filtra os elementos de uma lista de acordo com um predicado
    filter' :: (a -> Bool) -> [a] -> [a]
    ```

- Resposta

```haskell
or' :: [Bool] -> Bool
reverse' = foldl (||) False

reverse' :: [a] -> [a]
reverse' = foldl (\y x -> x : y) []

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x y -> if p x then x : y else y) []
```

33. Escreva duas versões para a função

```haskell
elem' :: Eq a => a -> [a] -> Bool
```

que verifica se um elemento está em uma lista. Uma versão deverá utilizar *folding* e outra a função *any*.

- Resposta

```haskell
elem' :: Eq a => a -> [a] -> Bool
elem' k xs = foldl (\y x -> if x == k then True else y) False xs

elemAny :: Eq a => a -> [a] -> Bool
elemAny x xs = any (==x) xs
-- verifique que podemos definir tb como:
--elemAny = any.(==)
```