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

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

17 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
```

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"
```

3. Defina uma função para seguinte assinatura:

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

4. Defina uma função

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

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

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
```

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

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.

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

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

usando dessa vez uma única expressão condicional.

9. Defina a única função possível para a assinatura

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

10. Defina a única função possível para a assinatura

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

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.

12. Defina uma função

    ```
    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.

13. Usando compreensão de listas, defina a função

    ```
    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]
    ```

14. Defina a função

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

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

15. Escreva uma função recursiva

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

    que verifica se os elementos da lista formam um palíndromo.

16. Defina uma função

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

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

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.

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.

    (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].

    (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: $[6,5,12,4] = 6 + 5 + 1 + 2 + 4 = 18$

(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
```

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.
```

20. Escreva uma função recursiva

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

que verifica se um valor está contido numa lista

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
```

22. Defina a função recursiva

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

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

23. Defina a função

```
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"
```

24. Escreva a função

```
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"
```

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]
```

26. Utilizando a função merge implementada anteriormente, defina

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

que implementa o algoritmo merge sort. Para auxiliar, defina

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

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

27. Defina duas funções

```
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.

28. Defina a função

```
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 []
[]
```

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]
```

30. Defina uma função

```
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]
```

31. Usando *folding*, defina a função

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

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

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

32. Utilize *folding* para definir as seguintes funções em 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]
```

33. Escreva duas versões para a função

```
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*.