Programação Funcional
Folha de Exercícios 05

Prof. Wladimir Araújo Tavares

1. A função `foldl :: (a -> b -> a) -> a -> [b] -> a` é tal que (`foldl f z xs`) devolve o resultado da aplicação sucessiva de f usando como primeiro argumento da função f, o valor z na primeira a aplicação e o resultado da aplicação anterior nas outras aplicações; como segundo argumento de f, os valores da lista xs a partir da esquerda. Por exemplo,

   ```
   foldl (/) 64 [4,2,4]  == ( ( ( 64 / 4 ) / 2 ) / 4 ) == 2.0
   ```

   A definição recursiva de foldl é:

   ```
   foldl f z [ ] = z
   foldl f z (x : xs) = foldl f (f z x ) xs
   ```

   A função `scanEsq` guarda o resultado das aplicações sucessivas realizadas por `foldl`.

   (a) Defina função `scanEsq :: (b -> a -> b) -> b -> [a] -> [b]` tal que (`scanEsq f z xs`) é uma lista de sucessivas aplicações da função f com os valores da lista xs a partir da esquerda. Por exemplo,

   ```
   scanEsq 64 (/) [4,2,4] = [64.0, 16.0, 8.0, 2.0]
   ```

2. Use a função `scanEsq` para definir a lista infinita `fatorialLista :: [Int]` definida da seguinte maneira:

   ```
   fatorialLista = [1!, 2!, 3!, 4!, 5!, ...]
   ```

3. Use a lista infinita fatorialLista, para definir a função `fatorial :: Int -> Int` tal que `fatorial n` é n!.

4. O número e pode ser calculado por meio da série de Taylor como a soma da seguinte série infinita:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \ldots \quad (1)$$

   (a) Crie uma lista infinita `parcelas :: [Double]` com as parcelas da série infinita.

   ```
   take 4 parcelas == [1.0,1.0,0.5,0.16666666666666666]
   ```

   (b) Crie uma lista infinita `e :: [Double]` com as somas parciais da série infinita.

   ```
   take 4 e == [1.0,2.0,2.5,2.6666666666666665]
   ```

   (c) Defina a funcão calcE n que calcula o valor de e somando n parcelas da série. Calcule o valor de e para 10, 100, 1000 parcelas.

   (d) O processo de soma de parcelas pode ser repetido várias vezes até que $e^(k)$ esteja muito próximo do valor $e^(k-1)$. Dada uma precisão ϵ, o valor x^k será escolhido como uma solução aproximada da solução exata se:

   i. Erro absoluto: $|e^{(k)} - e^{(k-1)}| < \epsilon$

   ```
   absolutoE 0.01 == 2.7166666666666663
   absolutoE 0.0001 == 2.71827876984127
   ```

   Dica: Crie uma lista infinita com os valores consecutivos de e e remova os pares com o erro absoluto maior que ϵ usando a função `dropWhile`.

   ii. Erro relativo: $|\frac{e^{(k)} - e^{(k-1)}}{e^{(k-1)}}| < \epsilon$

   ```
   relativoE 0.01   == 2.7166666666666663
   relativoE 0.0001 == 2.7182539682539684
   ```

5. Considere a seguinte série (i.e. somas infinitas) que convergem para π:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \ldots \quad (2)$$

   (a) Construa uma lista infinita com os numerados das parcelas.
   $[4, -4, \ldots, 4(-1)^n]$

   (b) Construa uma lista infinita com os denominadores das parcelas.

   (c) Combine as duas listas usando a função `zipWith`.

   (d) Defina a função `calcPi1` n que calcula o valor de pi somando n parcelas da série. Calcule o valor o somatório para 10, 100 e 1000 parcelas.

   (e) Qual é o valor de pi considerando como critério de parada erro absoluto 0.01?

   (f) Qual é o valor de pi considerando como critério de parada erro relativo 0.01?

6. A lista infinita de números naturais `[1,2,3,4,..]` , `naturais :: [Int]`, pode ser definida de várias maneiras em Haskell:

   ```
   naturais = [1..]
   ```

   ```
   naturais = 1 :  zipWith (+) naturais [1,1..]
   ```

   ```
   naturais = scanEsq (+) 1 [1,1..]
   ```

   ```
   naturais = 1 :  [ x+y | (x,y) <- zip naturais [1,1..]  ]
   ```

   Os números triangulares são os números da seguinte forma $T_0 = 0$ e $T_n = T_{n-1} + n$. Defina uma lista infinita dos números triangulares `triangular :: [Int]`. Por exemplo,

   ```
   take 10 triangular == [0,1,3,6,10,15,21,28,36,45]
   ```

7. O fecho de Kleene é uma operação unária aplicada a conjuntos. A aplicação do fecho de Kleene num conjunto A é escrito como A^*. Se A é um alfabeto de uma linguagem, então A^* é o menor superconjunto de A que contém ϵ (string vazia) e e fechado para operação de concatenação. A^* também pode ser descrito como o conjunto de todos os elementos que podem ser formados através da concatenação de zero ou mais elementos de A.

   O fecho de Kleene pode ser definido recursivamente da seguinte maneira:

$$\begin{aligned} A_0 &= \{\epsilon\} \\ A_{i+1} &= \{wv : w \in A_i, v \in A\} \end{aligned}$$

   Logo,

$$A^* = \bigcup_{i=0}^{\infty} A_i = A_0 \cup A_1 \cup \ldots$$

   Defina a função `kleene :: [a] -> [[a]]` tal que (`kleene xs`) devolve o fecho de Kleene do conjunto xs. Por exemplo,

   ```
   take 12 (kleene "01") == [,"0","1","00","01","10","11","000","001",
   "100","101","010"]
   ```

   ```
   take 20 (kleene "01") ==[,"0","1","00","01","10","11","000","001",
   "100","101","010","011","110","111","0000","0001","1000","1001","0100"]
   ```

8. Defina a função `repete :: a -> [a]` tal que (repete x) é uma lista infinita cujos elementos são x. Por exemplo,

   ```
   repete 5 = [5,5,5,5,...
   take 3 repete 5 = [5,5,5]
   ```

   (a) Defina usando compreensão de listas

   (b) Defina usando a definição recursiva de lista

9. Defina a função `ciclo :: [a] -> [a]` tal que (ciclo xs) repete infinitamente da lista xs. Por exemplo,

   ciclo [1,2] = [1,2,1,2,1,2,1,2,1,2,...

10. Defina, por compreensão, a função `eco :: [a] -> [a]` tal que `eco xs` é uma lista obtida a partir da lista xs repetindo cada elemento o número de vezes indicada pela sua posição. O primeiro elemento é repetido 1 vez, o segundo 2 vezes e assim sucessivamente. Por exemplo,

    ```
    eco "abcd" == "abbcccdddd"
    take 6 (eco [1..]) = [1,2,2,3,3,3]
    ```

11. Defina a função `itera :: ( a->a ) -> a -> [a]` tal que (`itera f x`) é uma lista cujo primeiro elemento é x e os seguintes são obtidos aplicando a função f ao elemento anterior, ou seja,

    ```
    [x, f x, f (f x), f (f x), ...]
    ```

    Por exemplo,

    ```
    itera (+1) 3 == [3,4,5,6,7,8,9,...
    itera (*2) 1 == [1,2,4,8,16,32,...
    ```

    (a) Defina por compreensão de listas infinitas.

    (b) Defina por recursão em listas infinitas.

12. Defina a função `potenciaIterada :: Int -> (a -> a) -> a -> a` tal que (`potenciaIterada n f x`) é o resultado de aplicar n vezes a função f ao valor x. Por exemplo,

    ```
    potenciaIterada 3 (*10) 5 == 5000
    potenciaIterada 4 (+10) 5 == 45
    ```

    Use a função `take :: Int -> [a] -> [a]` e `iterate :: (a->a) -> a -> [a]`.

13. Defina a função `paresOrdenados :: [a] -> [(a,a)]` tal que (`paresOrdenados xs`) é uma lista de todos os pares de elementos (x, y) de xs tal que x aparece em xs antes de y. Por exemplo,

    ```
    paresOrdenados [3,2,5,4] == [(3,2),(3,5),(3,4),(2,5),(2,4),(5,4)]
    paresOrdenados [3,2,5,3] == [(3,2),(3,5),(3,3),(2,5),(2,3),(5,3)]
    ```

14. Considere o seguinte algoritmo que gera uma sequência de inteiros. Comece com um inteiro positivo n. Se n é par, divida por 2. Se n é ímpar, multiplique por 3 e some 1. Repita o processo com o novo valor de n, o processo termina quando n = 1. Por exemplo, comece com n = 6,

    ```
    [6, 3, 10, 5, 16, 8, 4 , 2, 1]
    ```

    Esta sequência é conhecida como sequência Collatz. Uma conjectura bastante conhecida defende que este algoritmo de fato termina para todo n positivo.

(a) Defina a funcao `seguinte :: Integer -> Integer` tal que seguinte n é o próximo número obtido na sequência de Collatz.

(b) Defina a funcão `Collatz :: Integer -> [Integer]` tal que collatz n é a sequência de Collatz obtida comecando com o valor n usando a função `itera`. Por exemplo,

```
collatz 13 = [13,40,20,10,5,16,8,4,2,1]
```

15. A função `digitos :: Integer -> [Integer]`, dado um número n, devolve uma lista com os dígitos de n . Ela pode ser definida da seguinte maneira:

```
digitos n = [ read [x] :: Int | x <- show n]
```

A função `show :: Show a => a -> String` devolve a representação em `String` de um tipo `Show a` e a função `read :: Read a => String -> a` dada uma representação em `String` converte para um tipo `Read a`.

(a) Defina a função `listaInteger :: [Integer] -> Integer` tal que (`listaInteger xs`) é o número formado pelos dígitos da lista xs usando a read e show. Por exemplo,

```
listaInteger [5]       == 5
listaInteger [1,3,4,7] == 1347
listaInteger [0,0,1]   == 1
```

(b) Defina a função `juntaNumero :: Integer -> Integer -> Integer` tal que `juntaNumero x y` é o numero resultante de "concatenar"os digitos dos dois números x e y. Por exemplo,

```
juntaNumero 12 987 == 12987
juntaNumero 1204 7 == 12047
```

(c) Defina a função `inverso :: Integer -> Integer` tal que inverso n é o número obtido escrevendo os dígitos de n em ordem inversa. Por exemplo,

```
inverso 42578 == 87524
inverso 203   == 302
```

16. Defina, por compreensão, a lista dos números inteiros `inteiros :: [Int]` tal que inteiro é uma lista obtida pela enumeração do números inteiros. Por exemplo,

```
take 10 inteiros == [0,-1,1,-2,2,-3,3,-4,4,-5]
```

Dica: use a função `concat :: [[a]] -> [a]` que concatena uma lista de listas.

17. Defina a função `divisoresEm :: Int -> [Int] -> Bool` tal que (`divisoresEm x ys`) verifica se x pode ser expresso por fatores em ys. Por exemplo,

```
divisoresEm 12 [2,3,5] == True   (12 = 2*2*3)
divisoresEm 14 [2,3,5] == False  (14 = 2*7)
```

18. Defina a função `intercala :: a -> [a] -> [[a]]` tal que (`intercala x ys`) é uma lista de listas obtidas intercalando x entre os elementos de ys. Por exemplo,

```
intercala 1 [2,3] == [[1,2,3], [2,1,3], [2,3,1]]
```

19. Defina a função `permutacao :: [a] -> [[a]]` tal que `permutacao xs` é uma lista de todas as permutações da lista xs usando a função intercala. Por exemplo,

```
permutacoes "abc"== ["abc","bac","bca","acb","cab","cba"]
```