Programação Funcional
Folha de Exercícios 06
Definição de Tipos

Prof. Wladimir Araújo Tavares

1. A função `log :: Floating a => a -> a` não está definida para números negativos.

```
> log 1000
6.907755278982137
> log (-1000)
''ERROR'' -- runtime error
```

   Defina uma versão segura que evite runtime error usando Maybe.

```
safeLog :: (Floating a, Ord a) => a -> Maybe a
safeLog x
    | x > 0      =
    | otherwise  =
```

2. Considere uma representação de pontos no plano pelo par das suas coordenadas cartesianas: `type Ponto = (Float, Float)`

   (a) Escreva uma definição da função que calcula a distância euclidiana entre dois pontos (x_1, y_1) e (x_2, y_2): $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.
   `dist :: Ponto -> Ponto -> Float`

   (b) Considere agora um percurso dado como uma lista de pontos consecutivos. Escreva uma função `comprimento :: [Ponto] -> Float` que calcule o comprimento total dum percurso (isto é, a soma das distâncias entre pontos consecutivos). Pode usar recursão ou listas em compreensão e deve usar a função dist da alínea anterior para calcular as distâncias. Tenha atenção de tratar corretamente os percursos degenerados (vazios ou com apenas um ponto).

3. Vamos representar pontos de um plano cartesiano com duas coordernadas e regiões desse plano como funções, usando os seguintes tipos:

```
data Ponto = Pt Float Float
type Regiao = Ponto -> Bool
```

   Se r representa uma região do plano, então um ponto p está nessa região do plano se r p é igual a True.

   (a) Defina funções `retang::Ponto->Ponto->Regiao` e `circ::Ponto->Raio->Regiao` tais que:
   retang p q retorne (a região que representa) o retângulo tal que p é o ponto mais à esquerda e mais baixo, e q o ponto mais à direta e mais alto.
   circ p r retorne o círculo de raio r e centro p.
   Lembre-se: regiões são representadas por funções.

   (b) Defina funções `uniao::Regiao->Regiao->Regiao`, `interseccao::Regiao->Regiao->Regiao` e `complemento::Regiao->Regiao` tais que p está em uniao r r' se e somente se p está na uniao das regiões r e r', e analogamente para intersecção e complemento.

4. Considere a representação de um grafo dirigido de vértices inteiros como um par ordenado uma lista de vértices e uma lista de arestas (isto é, pares ordenados de vertices).

```
type Vert = Int
type Grafo = ([Vert],[(Vert ,Vert)])
```

   Escreva uma função caminho :: `Grafo -> [Vert] -> Bool` tal que caminho g xs é True se xs é uma lista de vértices que representa um caminho no grafo (isto é, se cada dois vértices consecutivos correspondem a uma aresta) e False, caso contrário.

   Exemplos:

```
G = ([1,2,3], [(1, 2), (2, 1), (2, 3)])
caminho  G [1, 2, 1, 2, 3] = True
caminho  G [1, 2, 1, 3] = False
```

5. Considere a representação de uma relação binária nos inteiros como uma lista de pares.

   `type Rel = [(Int , Int)]`

   (a) Escreva uma função `reflexiva :: [Int]->Rel -> Bool` que verifique se uma relação R em A é reflexiva (R é reflexiva se e somente se $\forall x \in A, (x, x) \in R$)
   Exemplos:

```
reflexiva [1,2,3] [(1, 1), (2, 2), (1, 2),(3, 3)] = True
reflexiva [1,2,3] [(1, 2), (2, 3)] = False

reflexiva conj rel = and [ elem (x,x) rel | x <- conj ]
```

   (b) Escreva uma função `simetrica :: Rel -> Bool` que verifique se uma relação é transitiva (R é transitiva se e somente se $\forall x, y, (x, y) \in R \Rightarrow (y, x) \in R$).
   Exemplos:

```
simetrica [(1, 3), (3, 1), (2, 2)] = True
simetrica [(1, 2), (2, 3)] = False

simetrica [] == True
simetrica [(1,2),(2,1)] == True
simetrica [(1,2),(2,1),(1,3)] == False
simetrica [(1,2),(2,1),(1,3),(3,1)] == True
```

   (c) Escreva uma função `transitiva :: Rel -> Bool` que verifique se uma relação é transitiva (R é transitiva se e somente se $\forall x, y, z, (x, y) \in R \wedge (y, z) \in R \rightarrow (x, z) \in R$).
   Exemplos:

```
transitiva [(1, 3), (1, 2), (2, 3)] = True
transitiva [(1, 2), (2, 3)] = False

transitiva [(1,2),(2,4),(1,4)] == True
transitiva [(1,2),(2,4),(1,4),(2,3)] == False
transitiva [] == True
```

6. Complete as seguintes definições recursivas para uma árvore binária:

```
data Arv a = Vazia | No a (Arv a) (Arv a) deriving (Eq, Show)
arv1 = Vazia
arv2 = No 2 arv1 arv1
arv3 = No 4 arv2 arv1
arv4 = No 5 arv3 arv2
```

   (a) Escreva uma função recursiva para calcular o número de nós de uma árvore.

```
tamanhoArv :: Arv a -> Int
tamanhoArv Vazia           =
tamanhoArv (No x esq dir) =

tamanhoArv arv4 == 4
```

   (b) Escreva uma função recursiva para calcular a altura de uma árvore.

```
alturaArv :: Arv a -> Int
alturaArv Vazia           =
alturaArv (No x esq dir) =

alturaArv arv4 == 3
```

   (c) Escreva uma função recursiva para soma todos os valores de uma árvore binária de números

```
sumArv :: Num a => Arv a -> a
sumArv Vazia           =
sumArv (No x esq dir) =
```

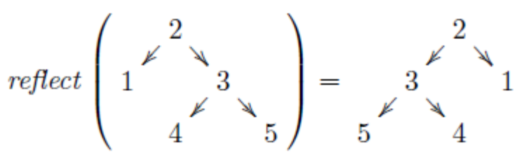
   (d) Escreva uma definição recursiva `nivel :: Int -> Arv a -> [a]` tal que nivel n arv é a lista ordenada dos valores da árvore no nível n, isto é, a uma altura n (considerando que a raiz tem altura 0).

```
nivel :: Int -> Arv a -> [a]
nivel _ Vazia            =
nivel 0 (No x esq dir) =
nivel n (No x esq dir) =
```

   (e) Escreva uma definição da função de ordem superior `mapArv :: (a -> b) -> Arv a -> Arv b` tal que mapArv f t aplica uma função f a cada valor duma árvore t. árvore t.

```
mapArv :: (a -> b) -> Arv a -> Arv b
mapArv f Vazia           =
mapArv f (No x esq dir) =
```

   (f) Escreva uma definição da função `reflect :: Arv a -> Arv` a que recursivamente troca os lados esquerdos e direitos de uma árvore. Exemplo:



   (g) Escreva uma definição da função que insere um valor numa árvore de pesquisa ordenada. Deve manter invariante a propriedade ordenação da árvore e não inserir outra cópia do valor se este já ocorrer na árvore.

```
inserir :: Ord a => a -> Arv a -> Arv a
inserir x Vazia =
inserir x (No y esq dir)
          | x<y =
          | x>y =
          | otherwise =
```

(h) Escreva uma definição da função que remove um valor numa árvore de pesquisa ordenada. Deve manter invariante a propriedade ordenação da árvore.

```haskell
mais_esq :: Arv a -> a
mais_esq (No x Vazia _) = x
mais_esq (No _ esq _) = mais_esq esq
```

```haskell
remover :: Ord a => a -> Arv a -> Arv a
remover x Vazia = Vazia -- não ocorre
remover x (No y Vazia dir) -- um descendente

remover x (No y esq Vazia) -- um descendente

remover x (No y esq dir) -- dois descendentes
 | x<y =
 | x>y =
 | x==y =
```

Se a árvore tem dois descendentes, substitua x pelo menor valor da árvore da direita e depois remova o menor valor da árvore direita.

(i) Escreva uma função recursiva para listar os elementos de uma árvore de pesquisa em ordem decrescente.

```haskell
listar :: Ord a => Arv a -> [a]
listar Vazia          =
listar (No x esq dir) =
```

7. Considere a definição em Haskell dum tipo de dados para multiconjuntos (i.e. coleções sem ordem mas com repetições) representado como árvore de pesquisa:

```haskell
data MConj a = Vazio | No a Int (MConj a) (MConj a)
```

Cada nó contém um valor e a sua multiplicidade (i.e. o número de repetições); para facilitar a pesquisa, a árvore deve estar ordenada pelos valores.

Por exemplo:

```haskell
No 'A' 2 Vazio (No 'B' 1 Vazio Vazio)
```

representa o multi-conjunto {A, A, B} com dois carateres 'A' e um 'B'.

(a) Escreva uma definição recursiva da função `ocorre :: Ord a => a -> MConj a -> Int` que procura o número de ocorrências de um valor num multi-conjunto; o resultado deve ser 0 se o valor não pertencer ao multi-conjunto.

(b) Escreva uma definição recursiva da função `inserir :: Ord a => a -> MConj a -> MConj a` que insere um valor num multi-conjunto mantendo a árvore de pesquisa ordenada.

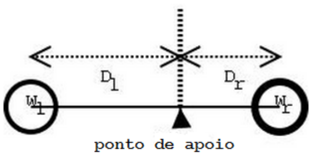
(c) Escreva uma definição recursiva da função `listar :: MConj a -> [a]` para listar os elementos de um multi-conjunto.

(d) Escreva uma definição recursiva da função `tamanho :: MConj a -> Int` para calcular o número de elementos de um multiconjunto.

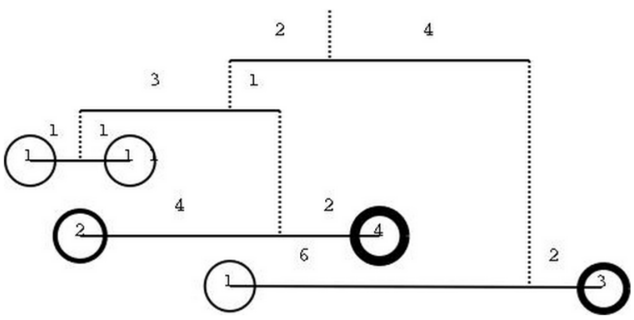
(e) Escreva uma definição recursiva da função `sumMConj :: MConj a -> Int` para calcular o somatório de todos os elementos de um multiconjunto.

8. Um mobile é uma estrutura constituída por uma haste, a partir da qual objetos ponderados ou outras hastes são penduradas.

A figura abaixo ilustra um mobile. É uma haste suspensa por uma corda, com objetos pendurados de cada lado. Também pode ser visto como uma espécie de alavanca com o ponto de apoio sendo o ponto em que a haste está suspensa pela corda. Pelo princípio da alavanca, sabemos que equilibrar um móvel temos que o produto do peso dos objetos pela distância ao ponto de apoio devem ser iguais, ou seja, $Wl \times Dl = Wr \times Dr$ onde Dl é a distância da esquerda, Dr é a distância da direita, WL é o peso da esquerda e Wr é o peso da direita.



Num sistema mais complexo, um objeto pode ser substituído por um sub-mobile, tal como mostrado na figura a seguir. Neste caso, não é tão simples para verificar se um mobile está em equilíbrio por isso precisamos de você para escrever um programa que, recebe um mobile como entrada, verifica se o móvel está em equilíbrio ou não.



```haskell
data Mobile = Haste Mobile Int Mobile Int | Objeto Int deriving (Ec

m1 = Haste (Objeto 1) 6 (Objeto 3) 2
m2 = Haste (Objeto 2) 4 (Objeto 4) 2
m3 = Haste (Objeto 1) 1 (Objeto 1) 1
m4 = Haste (m3) 3 (m2) 1
m5 = Haste (m4) 2 (m1) 6
```

(a) Escreva uma função `peso:: Mobile -> Int` que dado um Mobile retorna o peso sustentado por ele.

(b) Escreva uma função `equilibrio:: Mobile -> Int` que dado um Mobile retorna True, se todo o sistema está em equilíbrio,caso contrário, retorna False.

```haskell
import Data.List

data Prop =
    Const Bool
    | Var Char
    | Neg Prop
    | Conj Prop Prop
    | Disj Prop Prop
    | Impl Prop Prop
    deriving (Eq, Show)

prop1 = Impl (Var 'P') (Var 'Q')
prop2 = Impl (Neg (Var 'P')) (Var 'Q')
prop3 = Impl prop1 prop2


variaveis :: Prop -> [Char]
variaveis (Const x) = []
variaveis (Var x)   = [x]
variaveis (Neg p)   = variaveis p
variaveis (Conj p q) = nub ( variaveis p ++ variaveis q )
variaveis (Disj p q) = nub ( variaveis p ++ variaveis q )
variaveis (Impl p q) = nub ( variaveis p ++ variaveis q )
```

9. Escreva uma definição recursiva da função `removeImpl :: Prop -> Prop` que obtém uma proposição equivalente removendo a implicação. A implicação pode ser removida usando a seguinte regra de equivalência:

$$p \to q \Leftrightarrow \neg p \vee q \qquad (1)$$

```haskell
removeImpl prop1 == Disj (Neg (Var 'P')) (Var 'Q')
removeImpl prop2 == Disj (Neg (Neg (Var 'P'))) (Var 'Q')
removeImpl prop3 == Disj (Neg (Disj (Neg (Var 'P')) (Var 'Q')))
(Disj (Neg (Neg (Var 'P'))) (Var 'Q'))
```

10. Escreva uma definição recursiva da função `dual :: Prop -> Prop` que obtém o dual duma proposição, i.e. a proposição que resulta de substituir todas as conjunções por disjunções (e vice-versa) e as constantes True por False (e viceversa); as variáveis e negações são inalteradas. Exemplos:

```haskell
dual (Neg (Var 'a'))
= Neg (Var 'a')
dual (Disj (Var 'x') (Neg (Var 'x')))
= Conj (Var 'x') (Neg (Var 'x'))
dual (Conj (Var 'a') (Disj (Var 'b') (Const False)))
= Disj (Var 'a') (Conj (Var 'b') (Const True))
```

11. Escreva uma definição recursiva da função `duplaNegacao :: Prop -> Prop` que obtém uma proposição equivalente removendo a dupla negação. A dupla negação pode ser removida usando a seguinte regra de equivalência:

$$\neg\neg p \Leftrightarrow p \qquad (2)$$

```haskell
duplaNegacao $ removeImpl prop1 == Disj (Neg (Var 'P')) (Var 'Q')
duplaNegacao $ removeImpl prop2 == Disj (Var 'P') (Var 'Q')
duplaNegacao $ removeImpl prop3 == Disj (Neg (Disj (Neg (Var 'P'))
(Disj (Var 'P') (Var 'Q'))
```

12. Escreva uma função `contar :: Prop -> [(Char, Int)]` cujo resultado é uma lista de associações entre cada variável e o número de vezes que ocorre na proposição.

Exemplo: `contar (Conj (Var 'a') (Disj (Var 'b') (Var 'a')) = [('a', 2), ('b', 1)]`

(a ordem dos pares no resultado não é importante).

13. Escreva uma definição duma função `showProp :: Prop -> String` para converter uma proposição em texto;

Alguns exemplos:

```haskell
> showProp (Neg (Var 'a'))
"(~a)"
> showProp (Disj (Var 'a') (Conj (Var 'a') (Var 'b')))
"(a || (a && b))"
> showProp (Impl (Var 'a') (Impl (Neg (Var 'a')) (Const False))
"(a -> ((~a) -> F))"
```