Programação Funcional

Ficha 1

Funções não recursivas

1. Usando as seguintes funções pré-definidas do Haskell:

   - `length l`: o número de elementos da lista `l`
   - `head l`: a cabeça da lista (não vazia) `l`
   - `tail l`: a cauda da lista (não vazia) `l`
   - `last l`: o último elemento da lista (não vazia) `l`
   - `sqrt x`: a raiz quadrada de `x`
   - `div x y`: a divisão inteira de `x` por `y`
   - `mod x y`: o resto da divisão inteira de `x` por `y`

   Defina as seguintes funções e os respectivos tipos:

   (a) `perimetro` – que calcula o perímetro de uma circunferência, dado o comprimento do seu raio.

   (b) `dist` – que calcula a distância entre dois pontos no plano Cartesiano. Cada ponto é um par de valores do tipo `Double`.

   (c) `primUlt` – que recebe uma lista e devolve um par com o primeiro e o último elemento dessa lista.

   (d) `multiplo` – tal que `multiplo m n` testa se o número inteiro `m` é múltiplo de `n`.

   (e) `truncaImpar` – que recebe uma lista e, se o comprimento da lista for ímpar retira-lhe o primeiro elemento, caso contrário devolve a própria lista.

   (f) `max2` – que calcula o maior de dois números inteiros.

   (g) `max3` – que calcula o maior de três números inteiros, usando a função `max2`.

2. Defina as seguintes funções sobre polinómios de 2º grau:

   (a) A função `nRaizes` que recebe os (3) coeficientes de um polinómio de 2º grau e que calcula o número de raízes (reais) desse polinómio.

   (b) A função `raizes` que, usando a função anterior, recebe os coeficientes do polinómio e calcula a lista das suas raízes reais.

3. Vamos representar horas por um par de números inteiros:

   ```
   type Hora = (Int,Int)
   ```

   Assim o par `(0,15)` significa *meia noite e um quarto* e `(13,45)` *duas menos um quarto.* Defina funções para:

   (a) testar se um par de inteiros representa uma hora do dia válida;

   (b) testar se uma hora é ou não depois de outra (comparação);

   (c) converter um valor em horas (par de inteiros) para minutos (inteiro);

   (d) converter um valor em minutos para horas;

(e) calcular a diferença entre duas horas (cujo resultado deve ser o número de minutos);

(f) adicionar um determinado número de minutos a uma dada hora.

4. Repita o exercício anterior assumindo agora que as horas são representadas por um novo tipo de dados:

```
data Hora = H Int Int deriving (Show,Eq)
```

Com este novo tipo a hora *meia noite e um quarto* é representada por `H 0 15` e a hora *duas menos um quarto* por `H 13 45`.

5. Considere o seguinte tipo de dados para representar os possíveis estados de um semáforo:

```
data Semaforo = Verde | Amarelo | Vermelho deriving (Show,Eq)
```

(a) Defina a função `next :: Semaforo -> Semaforo` que calcula o próximo estado de um semáforo.

(b) Defina a função `stop :: Semaforo -> Bool` que determina se é obrigatório parar num semáforo.

(c) Defina a função `safe :: Semaforo -> Semaforo -> Bool` que testa se o estado de dois semáforos num cruzamento é seguro.

6. Um ponto num plano pode ser representado por um sistema de coordenadas Cartesiano (distâncias aos eixos vertical e horizontal) ou por um sistema de coordenadas Polar (distância à origem e ângulo do respectivo vector com o eixo horizontal).

```
data Ponto = Cartesiano Double Double | Polar Double Double
             deriving (Show,Eq)
```

Com este tipo o ponto `Cartesiano (-1) 0` pode alternativamente ser representado por `Polar 1 pi`. Defina as seguintes funções:

(a) `posx :: Ponto -> Double` que calcula a distância de um ponto ao eixo vertical.

(b) `posy :: Ponto -> Double` que calcula a distância de um ponto ao eixo horizontal.

(c) `raio :: Ponto -> Double` que calcula a distância de um ponto à origem.

(d) `angulo :: Ponto -> Double` que calcula o ângulo entre o vector que liga a origem ao ponto e o eixo horizontal.

(e) `dist :: Ponto -> Ponto -> Double` que calcula a distância entre dois pontos.

7. Considere o seguinte tipo de dados para representar figuras geométricas num plano.

```
data Figura = Circulo Ponto Double
            | Rectangulo Ponto Ponto
            | Triangulo Ponto Ponto Ponto
              deriving (Show,Eq)
```

O tipo de dados diz que uma figura pode ser um círculo centrado num determinado ponto e com um determinado raio, um rectângulo paralelo aos eixos representado por dois pontos que são vértices da sua diagonal, ou um triângulo representado pelos três pontos dos seus vértices. Defina as seguintes funções:

(a) Defina a função `poligono :: Figura -> Bool` que testa se uma figura é um polígono.

(b) Defina a função `vertices :: Figura -> [Ponto]` que calcula a lista dos vértices de uma figura.

(c) Complete a seguinte definição cujo objectivo é calcular a área de uma figura:

```
area :: Figura -> Double
area (Triangulo p1 p2 p3) =
       let a = dist p1 p2
           b = dist p2 p3
           c = dist p3 p1
           s = (a+b+c) / 2 -- semi-perimetro
       in sqrt (s*(s-a)*(s-b)*(s-c)) -- formula de Heron
...
```

(d) Defina a função `perimetro :: Figura -> Double` que calcula o perímetro de uma figura.

8. Utilizando as funções `ord :: Char -> Int` e `chr :: Int -> Char` do módulo `Data.Char`, defina as seguintes funções:

   (a) `isLower :: Char -> Bool`, que testa se um `Char` é uma minúscula.

   (b) `isDigit :: Char -> Bool`, que testa se um `Char` é um dígito.

   (c) `isAlpha :: Char -> Bool`, que testa se um `Char` é uma letra.

   (d) `toUpper :: Char -> Char`, que converte uma letra para a respectiva maiúscula.

   (e) `intToDigit :: Int -> Char`, que converte um número entre `0` e `9` para o respectivo dígito.

   (f) `digitToInt :: Char -> Int`, que converte um dígito para o respectivo inteiro.

   Note que todas estas funções já estão também pré-definidas nesse módulo.

Programação Funcional

Ficha 2

Funções recursivas sobre listas

1. Indique como é que o interpretador de Haskell avalia as expressões das alíneas que se seguem, apresentando a cadeia de redução de cada uma dessas expressões (i.e., os vários passos intermédios até se chegar ao valor final).

   (a) Considere a seguinte definição:

   ```
   funA :: [Double] -> Double
   funA [] = 0
   funA (y:ys) = y^2 + (funA ys)
   ```

   Diga, justificando, qual é o valor de `funA [2,3,5,1]`.

   (b) Considere seguinte definição:

   ```
   funB :: [Int] -> [Int]
   funB [] = []
   funB (h:t) = if (mod h 2)==0 then h : (funB t)
                                else (funB t)
   ```

   Diga, justificando, qual é o valor de `funB [8,5,12]`.

   (c) Considere a seguinte definição:

   ```
   funC (x:y:t) = funC t
   funC [x] = [x]
   funC [] = []
   ```

   Diga, justificando, qual é o valor de `funC [1,2,3,4,5]`.

   (d) Considere a seguinte definição:

   ```
   funD l = g [] l
   g acc [] = acc
   g acc (h:t) = g (h:acc) t
   ```

   Diga, justificando, qual é o valor de `funD "otrec"`.

2. Defina recursivamente as seguintes funções sobre listas:

   (a) `dobros :: [Float] -> [Float]` que recebe uma lista e produz a lista em que cada elemento é o dobro do valor correspondente na lista de entrada.

   (b) `numOcorre :: Char -> String -> Int` que calcula o número de vezes que um caracter ocorre numa string.

   (c) `positivos :: [Int] -> Bool` que testa se uma lista só tem elementos positivos.

   (d) `soPos :: [Int] -> [Int]` que retira todos os elementos não positivos de uma lista de inteiros.

   (e) `somaNeg :: [Int] -> Int` que soma todos os números negativos da lista de entrada.

   (f) `tresUlt :: [a] -> [a]` devolve os últimos três elementos de uma lista. Se a lista de entrada tiver menos de três elementos, devolve a própria lista.

(g) `segundos :: [(a,b)] -> [b]` que calcula a lista das segundas componentes dos pares.

(h) `nosPrimeiros :: (Eq a) => a -> [(a,b)] -> Bool` que testa se um elemento aparece na lista como primeira componente de algum dos pares.

(i) `sumTriplos :: (Num a, Num b, Num c) => [(a,b,c)] -> (a,b,c)` soma uma lista de triplos componente a componente.
Por exemplo, `sumTriplos [(2,4,11), (3,1,-5), (10,-3,6)] = (15,2,12)`

3. Recorrendo a funções do módulo `Data.Char`, defina recursivamente as seguintes funções sobre strings:

(a) `soDigitos :: [Char] -> [Char]` que recebe uma lista de caracteres, e selecciona dessa lista os caracteres que são algarismos.

(b) `minusculas :: [Char] -> Int` que recebe uma lista de caracteres, e conta quantos desses caracteres são letras minúsculas.

(c) `nums :: String -> [Int]` que recebe uma string e devolve uma lista com os algarismos que ocorrem nessa string, pela mesma ordem.

4. Uma forma de representar polinómios de uma variável é usar listas de monómios representados por pares *(coeficiente, expoente)*

```
type Polinomio = [Monomio]
type Monomio = (Float,Int)
```

Por exemplo, `[(2,3), (3,4), (5,3), (4,5)]` representa o polinómio $2\,x^3 + 3\,x^4 + 5\,x^3 + 4\,x^5$. Defina as seguintes funções:

(a) `conta :: Int -> Polinomio -> Int` de forma a que `(conta n p)` indica quantos monómios de grau `n` existem em `p`.

(b) `grau :: Polinomio -> Int` que indica o grau de um polinómio.

(c) `selgrau :: Int -> Polinomio -> Polinomio` que selecciona os monómios com um dado grau de um polinómio.

(d) `deriv :: Polinomio -> Polinomio` que calcula a derivada de um polinómio.

(e) `calcula :: Float -> Polinomio -> Float` que calcula o valor de um polinómio para uma dado valor de x.

(f) `simp :: Polinomio -> Polinomio` que retira de um polinómio os monómios de coeficiente zero.

(g) `mult :: Monomio -> Polinomio -> Polinomio` que calcula o resultado da multiplicação de um monómio por um polinómio.

(h) `normaliza :: Polinomio -> Polinomio` que dado um polinómio constrói um polinómio equivalente em que não podem aparecer varios monómios com o mesmo grau.

(i) `soma :: Polinomio -> Polinomio -> Polinomio` que soma dois polinómios de forma a que se os polinómios que recebe estiverem normalizados produz também um polinómio normalizado.

(j) `produto :: Polinomio -> Polinomio -> Polinomio` que calcula o produto de dois polinómios

(k) `ordena :: Polinomio -> Polinomio` que ordena um polinómio por ordem crescente dos graus dos seus monómios.

(l) `equiv :: Polinomio -> Polinomio -> Bool` que testa se dois polinómios são equivalentes.

Programação Funcional

Ficha 3

Gestão de informação em listas

1. Assumindo que uma hora é representada por um par de inteiros, uma viagem pode ser representada por uma sequência de etapas, onde cada etapa é representada por um par de horas (partida, chegada):

```
data Hora = H Int Int
          deriving Show

type Etapa = (Hora,Hora)
type Viagem = [Etapa]
```

   Por exemplo, se uma viagem for

```
[(H 9 30, H 10 25), (H 11 20, H 12 45), (H 13 30, H 14 45)]
```

   significa que teve três etapas:

   - a primeira começou às 9 e meia e terminou às 10 e 25;
   - a segunda começou às 11 e 20 e terminou à uma menos um quarto;
   - a terceira começou às 1 e meia e terminou às 3 menos um quarto;

   Para este problema, vamos trabalhar apenas com viagens que começam e acabam no mesmo dia. Utilizando as funções sobre horas que definiu na Ficha 1, defina as seguintes funções:

   (a) Testar se uma etapa está bem construída (i.e., o tempo de chegada é superior ao de partida e as horas são válidas).

   (b) Testa se uma viagem está bem construída (i.e., se para cada etapa, o tempo de chegada é superior ao de partida, e se a etapa seguinte começa depois da etapa anterior ter terminado).

   (c) Calcular a hora de partida e de chegada de uma dada viagem.

   (d) Dada uma viagem válida, calcular o tempo total de viagem efectiva.

   (e) Calcular o tempo total de espera.

   (f) Calcular o tempo total da viagem (a soma dos tempos de espera e de viagem efectiva).

2. Considere as seguinte definição de um tipo para representar linhas poligonais.

```
type Poligonal = [Ponto]
```

   O tipo `Ponto` é idêntico ao definido na Ficha 1. Nas resolução das alíneas seguintes pode utilizar funções definidas nessa ficha.

   (a) Defina a função para calcular o comprimento de uma linha poligonal.

   (b) Defina uma função para testar se uma dada linha poligonal é ou não fechada.

(c) Defina a função `triangula :: Poligonal -> [Figura]` que, dada uma linha poligonal fechada e convexa, calcule uma lista de triângulos cuja soma das áreas seja igual à área delimitada pela linha poligonal. O tipo `Figura` é idêntico ao definido na Ficha 1

(d) Defina uma função para calcular a área delimitada por uma linha poligonal fechada e convexa.

(e) Defina a função `mover :: Poligonal -> Ponto -> Poligonal` que, dada uma linha poligonal e um ponto, dá como resultado uma linha poligonal idêntica à primeira mas tendo como ponto inicial o ponto dado.

(f) Defina a função `zoom :: Double -> Poligonal -> Poligonal` que, dada um factor de escala e uma linha poligonal, dê como resultado uma linha poligonal semelhante e com o mesmo ponto inicial mas em que o comprimento de cada segmento de recta é multiplicado pelo factor dado.

3. Para armazenar uma agenda de contactos telefónicos e de correio electrónico definiram-se os seguintes tipos de dados. Não existem nomes repetidos na agenda e para cada nome existe uma lista de contactos.

```
data Contacto = Casa Integer
              | Trab Integer
              | Tlm Integer
              | Email String
              deriving Show

type Nome = String
type Agenda = [(Nome, [Contacto])]
```

(a) Defina a função `acrescEmail :: Nome -> String -> Agenda -> Agenda` que, dado um nome, um email e uma agenda, acrescenta essa informação à agenda.

(b) Defina a função `verEmails :: Nome -> Agenda -> Maybe [String]` que, dado um nome e uma agenda, retorna a lista dos emails associados a esse nome. Se esse nome não existir na agenda a função deve retornar `Nothing`.

(c) Defina a função `consTelefs :: [Contacto] -> [Integer]` que, dada uma lista de contactos, retorna a lista de todos os números de telefone dessa lista (tanto telefones fixos como telemóveis).

(d) Defina a função `casa :: Nome -> Agenda -> Maybe Integer` que, dado um nome e uma agenda, retorna o número de telefone de casa (caso exista).

4. Pretende-se guardar informação sobre os aniversários das pessoas numa tabela que associa o nome de cada pessoa à sua data de nascimento. Para isso, declarou-se a seguinte estrutura de dados

```
type Dia = Int
type Mes = Int
type Ano = Int
type Nome = String

data Data = D Dia Mes Ano
          deriving Show

type TabDN = [(Nome,Data)]
```

(a) Defina a função `procura :: Nome -> TabDN -> Maybe Data`, que indica a data de nascimento de uma dada pessoa, caso o seu nome exista na tabela.

(b) Defina a função `idade :: Data -> Nome -> TabDN -> Maybe Int`, que calcula a idade de uma pessoa numa dada data.

(c) Defina a função `anterior :: Data -> Data -> Bool`, que testa se uma data é anterior a outra data.

(d) Defina a função `ordena :: TabDN -> TabDN`, que ordena uma tabela de datas de nascimento, por ordem crescente das datas de nascimento.

(e) Defina a função `porIdade:: Data -> TabDN -> [(Nome,Int)]`, que apresenta o nome e a idade das pessoas, numa dada data, por ordem crescente da idade das pessoas.

5. Considere o seguinte tipo de dados que descreve a informação de um extracto bancário. Cada valor deste tipo indica o saldo inicial e uma lista de movimentos. Cada movimento é representado por um triplo que indica a data da operação, a sua descrição e a quantia movimentada (em que os valores são sempre números positivos).

```
data Movimento = Credito Float | Debito Float
               deriving Show

data Data = D Int Int Int
          deriving Show

data Extracto = Ext Float [(Data, String, Movimento)]
              deriving Show
```

(a) Construa a função `extValor :: Extracto -> Float -> [Movimento]` que produz uma lista de todos os movimentos (créditos ou débitos) superiores a um determinado valor.

(b) Defina a função `filtro :: Extracto -> [String] -> [(Data,Movimento)]` que retorna informação relativa apenas aos movimentos cuja descrição esteja incluída na lista fornecida no segundo parâmetro.

(c) Defina a função `creDeb :: Extracto -> (Float,Float)`, que retorna o total de créditos e de débitos de um extracto no primeiro e segundo elementos de um par, respectivamente.

(d) Defina a função `saldo :: Extracto -> Float` que devolve o saldo final que resulta da execução de todos os movimentos no extracto sobre o saldo inicial.

Programação Funcional

Ficha 4

Optimização com *tupling* e acumuladores. Listas por compreensão.

1. Defina a função `digitAlpha :: String -> (String,String)`, que dada uma string, devolve um par de strings: uma apenas com as letras presentes nessa string, e a outra apenas com os números presentes na string. Implemente a função de modo a fazer uma única travessia da string. Relembre que as funções `isDigit,isAlpha :: Char -> Bool` estão já definidas no módulo `Data.Char`.

2. Defina a função `nzp :: [Int] -> (Int,Int,Int)` que, dada uma lista de inteiros, conta o número de valores nagativos, o número de zeros e o número de valores positivos, devolvendo um triplo com essa informação. Certifique-se que a função que definiu percorre a lista apenas uma vez.

3. Defina a função `divMod :: Integral a => a -> a -> (a, a)` que calcula simultaneamente a divisão e o resto da divisão inteira por subtracções sucessivas.

4. Utilizando uma função auxiliar com um acumulador, optimize seguinte definição recursiva que determina qual o número que corresponde a uma lista de digitos.

   ```
   fromDigits :: [Int] -> Int
   fromDigits []    = 0
   fromDigits (h:t) = h*10^(length t) + fromDigits t
   ```

   Note que

   $$\begin{aligned} \texttt{fromDigits [1,2,3,4]} &= 1\times10^3+2\times10^2+3\times10^1+4\times10^0 \\ &= 4+10\times(3+10\times(2+10\times(1+10\times0))) \end{aligned}$$

5. Utilizando uma função auxiliar com acumuladores, optimize a seguinte definição que determina a soma do segmento inicial de uma lista com soma máxima.

   ```
   maxSumInit :: (Num a, Ord a) => [a] -> a
   maxSumInit l = maximum [sum m | m <- inits l]
   ```

6. Optimize a seguinte definição recursiva da função que calcula o `n`-ésimo número da sequência de Fibonacci, usando uma função auxiliar com 2 acumuladores que representam, respectivamente, o `n`-ésimo e o `n+1`-ésimo números dessa sequência.

   ```
   fib :: Int -> Int
   fib 0 = 0
   fib 1 = 1
   fib n = fib (n-1) + fib (n-2)
   ```

7. Defina a função `intToStr :: Integer -> String` que converte um inteiro numa string. Utilize uma função auxiliar com um acumulador onde vai construindo a string que vai devolver no final.

8. Para cada uma das expressões seguintes, exprima por enumeração a lista correspondente. Tente ainda, para cada caso, descobrir uma outra forma de obter o mesmo resultado.

   (a) `[x | x <- [1..20], mod x 2 == 0, mod x 3 == 0]`

   (b) `[x | x <- [y | y <- [1..20], mod y 2 == 0], mod x 3 == 0]`

   (c) `[(x,y) | x <- [0..20], y <- [0..20], x+y == 30]`

   (d) `[sum [y | y <- [1..x], odd y] | x <- [1..10]]`

9. Defina cada uma das listas seguintes por compreensão.

   (a) `[1,2,4,8,16,32,64,128,256,512,1024]`

   (b) `[(1,5),(2,4),(3,3),(4,2),(5,1)]`

   (c) `[[1],[1,2],[1,2,3],[1,2,3,4],[1,2,3,4,5]]`

   (d) `[[1],[1,1],[1,1,1],[1,1,1,1],[1,1,1,1,1]]`

   (e) `[1,2,6,24,120,720]`

Programação Funcional

Ficha 5

Funções de ordem superior

1. Apresente definições das seguintes funções de ordem superior, já pré-definidas no `Prelude` ou no `Data.List`:

   (a) `any :: (a -> Bool) -> [a] -> Bool` que teste se um predicado é verdade para algum elemento de uma lista; por exemplo:
   `any odd [1..10] == True`

   (b) `zipWith :: (a->b->c) -> [a] -> [b] -> [c]` que combina os elementos de duas listas usando uma função específica; por exemplo:
   `zipWith (+) [1,2,3,4,5] [10,20,30,40] == [11,22,33,44]`.

   (c) `takeWhile :: (a->Bool) -> [a] -> [a]` que determina os primeiros elementos da lista que satisfazem um dado predicado; por exemplo:
   `takeWhile odd [1,3,4,5,6,6] == [1,3]`.

   (d) `dropWhile :: (a->Bool) -> [a] -> [a]` que elimina os primeiros elementos da lista que satisfazem um dado predicado; por exemplo:
   `dropWhile odd [1,3,4,5,6,6] == [4,5,6,6]`.

   (e) `span :: (a-> Bool) -> [a] -> ([a],[a])`, que calcula simultaneamente os dois resultados anteriores. Note que apesar de poder ser definida à custa das outras duas, usando a definição

   ```
   span p l = (takeWhile p l, dropWhile p l)
   ```

   nessa definição há trabalho redundante que pode ser evitado. Apresente uma definição alternativa onde não haja duplicação de trabalho.

   (f) `deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]` que apaga o primeiro elemento de uma lista que é "igual" a um dado elemento de acordo com a função de comparação que é passada como parâmetro. Por exemplo:
   `deleteBy (\x y -> snd x == snd y) (1,2) [(3,3),(2,2),(4,2)]`

   (g) `sortOn :: Ord b => (a -> b) -> [a] -> [a]` que ordena uma lista comparando os resultados de aplicar uma função de extracção de uma chave a cada elemento de uma lista. Por exemplo:
   `sortOn fst [(3,1),(1,2),(2,5)] == [(1,2),(2,5),(3,1)]`.

2. Relembre a questão sobre polinómios introduzida na Ficha 3, onde um polinómio era representado por uma lista de monómios representados por pares *(coeficiente, expoente)*

   ```
   type Polinomio = [Monomio]
   type Monomio = (Float,Int)
   ```

   Por exemplo, `[(2,3), (3,4), (5,3), (4,5)]` representa o polinómio $2\,x^3 + 3\,x^4 + 5\,x^3 + 4\,x^5$. Redefina as funções pedidas nessa ficha, usando agora funções de ordem superior (definidas no `Prelude` ou no `Data.List`) em vez de recursividade explícita:

   (a) `selgrau :: Int -> Polinomio -> Polinomio` que selecciona os monómios com um dado grau de um polinómio.

   (b) `conta :: Int -> Polinomio -> Int` de forma a que `(conta n p)` indica quantos monómios de grau `n` existem em `p`.

(c) `grau :: Polinomio -> Int` que indica o grau de um polinómio.

(d) `deriv :: Polinomio -> Polinomio` que calcula a derivada de um polinómio.

(e) `calcula :: Float -> Polinomio -> Float` que calcula o valor de um polinómio para uma dado valor de x.

(f) `simp :: Polinomio -> Polinomio` que retira de um polinómio os monómios de coeficiente zero.

(g) `mult :: Monomio -> Polinomio -> Polinomio` que calcula o resultado da multiplicação de um monómio por um polinómio.

(h) `ordena :: Polinomio -> Polinomio` que ordena um polonómio por ordem crescente dos graus dos seus monómios.

(i) `normaliza :: Polinomio -> Polinomio` que dado um polinómio constrói um polinómio equivalente em que não podem aparecer varios monómios com o mesmo grau.

(j) `soma :: Polinomio -> Polinomio -> Polinomio` que faz a soma de dois polinómios de forma que se os polinómios que recebe estiverem normalizados produz também um polinómio normalizado.

(k) `produto :: Polinomio -> Polinomio -> Polinomio` que calcula o produto de dois polinómios

(l) `equiv :: Polinomio -> Polinomio -> Bool` que testa se dois polinómios são equivalentes.

3. Considere a sequinte definição para representar matrizes:

```
type Mat a = [[a]]
```

Por exemplo, a matriz (triangular superior) $\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix}$ seria representada por `[[1,2,3], [0,4,5], [0,0,6]]`

Defina as seguintes funções sobre matrizes (use, sempre que achar apropriado, funções de ordem superior).

(a) `dimOK :: Mat a -> Bool` que testa se uma matriz está bem construída (i.e., se todas as linhas têm a mesma dimensão).

(b) `dimMat :: Mat a -> (Int,Int)` que calcula a dimensão de uma matriz.

(c) `addMat :: Num a => Mat a -> Mat a -> Mat a` que adiciona duas matrizes.

(d) `transpose :: Mat a -> Mat a` que calcula a transposta de uma matriz.

(e) `multMat :: Num a => Mat a -> Mat a -> Mat a` que calcula o produto de duas matrizes.

(f) `zipWMat :: (a -> b -> c) -> Mat a -> Mat b -> Mat c` que, à semelhança do que acontece com a função `zipWith`, combina duas matrizes. Use essa função para definir uma função que adiciona duas matrizes.

(g) `triSup :: Num a => Mat a -> Bool` que testa se uma matriz quadrada é triangular superior (i.e., todos os elementos abaixo da diagonal são nulos).

(h) `rotateLeft :: Mat a -> Mat a` que roda uma matriz 90º para a esquerda. Por exemplo, o resultado de rodar a matriz acima apresentada deve corresponder à matriz $\begin{bmatrix} 3 & 5 & 6 \\ 2 & 4 & 0 \\ 1 & 0 & 0 \end{bmatrix}$.

Programação Funcional

Ficha 6

Árvores binárias com conteúdo nos nós

1. Considere o seguinte tipo para representar árvores binárias.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
          deriving Show
```

   Defina as seguintes funções:

   (a) `altura :: BTree a -> Int` que calcula a altura da árvore.

   (b) `contaNodos :: BTree a -> Int` que calcula o número de nodos da árvore.

   (c) `folhas :: BTree a -> Int`, que calcula o número de folhas (i.e., nodos sem descendentes) da árvore.

   (d) `prune :: Int -> BTree a -> BTree a`, que remove de uma árvore todos os elementos a partir de uma determinada profundidade.

   (e) `path :: [Bool] -> BTree a -> [a]`, que dado um caminho (`False` corresponde a *esquerda* e `True` a *direita*) e uma árvore, dá a lista com a informação dos nodos por onde esse caminho passa.

   (f) `mirror :: BTree a -> BTree a`, que dá a árvore simétrica.

   (g) `zipWithBT :: (a -> b -> c) -> BTree a -> BTree b -> BTree c` que generaliza a função `zipWith` para árvores binárias.

   (h) `unzipBT :: BTree (a,b,c) -> (BTree a,BTree b,BTree c)`, que generaliza a função `unzip` (neste caso de triplos) para árvores binárias.

2. Defina as seguintes funções, assumindo agora que as árvores são binárias de procura:

   (a) Defina uma função `minimo :: Ord a => BTree a -> a` que determina o menor elemento de uma árvore binária de procura **não vazia**.

   (b) Defina uma função `semMinimo :: Ord a => BTree a -> BTree a` que remove o menor elemento de uma árvore binária de procura **não vazia**.

   (c) Defina uma função `minSmin :: Ord a => BTree a -> (a,BTree a)` que calcula, com uma única travessia da árvore o resultado das duas funções anteriores.

   (d) Defina uma função `remove :: Ord a => a -> BTree a -> BTree a` que remove um elemento de uma árvore binária de procura, usando a função anterior.

3. Considere agora que guardamos a informação sobre uma turma de alunos na seguinte estrutura de dados:

```
type Aluno = (Numero,Nome,Regime,Classificacao)
type Numero = Int
type Nome = String
data Regime = ORD | TE | MEL  deriving Show
data Classificacao = Aprov Int
                   | Rep
                   | Faltou
   deriving Show
type Turma = BTree Aluno  -- árvore binária de procura (ordenada por número)
```

Defina as seguintes funções:

(a) `inscNum :: Numero -> Turma -> Bool`, que verifica se um aluno, com um dado número, está inscrito.

(b) `inscNome :: Nome -> Turma -> Bool`, que verifica se um aluno, com um dado nome, está inscrito.

(c) `trabEst :: Turma -> [(Numero,Nome)]`, que lista o número e nome dos alunos trabalhadores-estudantes (ordenados por número).

(d) `nota :: Numero -> Turma -> Maybe Classificacao`, que calcula a classificação de um aluno (se o aluno não estiver inscrito a função deve retornar `Nothing`).

(e) `percFaltas :: Turma -> Float`, que calcula a percentagem de alunos que faltaram à avaliação.

(f) `mediaAprov :: Turma -> Float`, que calcula a média das notas dos alunos que passaram.

(g) `aprovAv :: Turma -> Float`, que calcula o rácio de alunos aprovados por avaliados. Implemente esta função fazendo apenas uma travessia da árvore.

Programação Funcional

Ficha 7

Outros tipos de árvores

1. Considere o seguinte tipo para representar expressões inteiras.

```
data ExpInt = Const Int
            | Simetrico ExpInt
            | Mais ExpInt ExpInt
            | Menos ExpInt ExpInt
            | Mult ExpInt ExpInt
```

   Os termos deste tipo `ExpInt` podem ser vistos como árvores cujas folhas são inteiros e cujos nodos (não folhas) são operadores.

   (a) Defina uma função `calcula :: ExpInt -> Int` que, dada uma destas expressões calcula o seu valor.

   (b) Defina uma função `infixa :: ExpInt -> String` de forma a que `infixa (Mais (Const 3) (Menos (Const 2) (Const 5)))` dê como resultado `"(3 + (2 - 5))"`.

   (c) Defina uma outra função de conversão para strings `posfixa :: ExpInt -> String` de forma a que quando aplicada à expressão acima dê como resultado `"3 2 5 - +"`.

2. Considere o seguinte tipo para representar árvores irregulares (*rose trees*).

```
data RTree a = R a [RTree a]
```

   Defina as seguintes funções sobre estas árvores:

   (a) `soma :: Num a => RTree a -> a` que soma os elementos da árvore.

   (b) `altura :: RTree a -> Int` que calcula a altura da árvore.

   (c) `prune :: Int -> RTree a -> RTree a` que remove de uma árvore todos os elementos a partir de uma determinada profundidade.

   (d) `mirror :: RTree a -> RTree a` que gera a árvore simétrica.

   (e) `postorder :: RTree a -> [a]` que corresponde à travessia postorder da árvore.

3. Relembre a definição de árvores binárias apresentada na ficha anterior:

```
data BTree a = Empty | Node a (BTree a) (BTree a)
```

   Nestas árvores a informação está nos nodos (as *extermidades* da árvore têm apenas uma marca – `Empty`). É também habitual definirem-se árvores em que a informação está apenas nas extermidades (*leaf trees*):

```
data LTree a = Tip a | Fork (LTree a) (LTree a)
```

   Defina sobre este tipo as seguintes funções

(a) `ltSum :: Num a => LTree a -> a` que soma as folhas de uma árvore.

(b) `listaLT :: LTree a -> [a]` que lista as folhas de uma árvore (da esquerda para a direita).

(c) `ltHeight :: LTree a -> Int` que calcula a altura de uma árvore.

4. Estes dois conceitos podem ser agrupados num só, definindo o seguinte tipo:

```
data FTree a b = Leaf b | No a (FTree a b) (FTree a b)
```

São as chamadas *full trees* onde a informação está não só nos nodos, como também nas folhas (note que o tipo da informação nos nodos e nas folhas não tem que ser o mesmo).

(a) Defina a função `splitFTree :: FTree a b -> (BTree a, LTree b)` que separa uma árvore com informação nos nodos e nas folhas em duas árvores de tipos diferentes.

(b) Defina ainda a função `joinTrees :: BTree a -> LTree b -> Maybe (FTree a b)` que sempre que as árvores sejam *compatíveis* as junta numa só.

Programação Funcional

Ficha 8

Classes de tipos

1. Considere o seguinte tipo de dados para representar fracções

   ```
   data Frac = F Integer Integer
   ```

   (a) Defina a função `normaliza :: Frac -> Frac`, que dada uma fracção calcula uma fracção equivalente, irredutível, e com o denominador positivo. Por exemplo, `normaliza (F (-33) (-51))` deve retornar `F 11 17` e `normaliza (F 50 (-5))` deve retornar `F (-10) 1`. Sugere-se que comece por definir primeiro a função `mdc :: Integer -> Integer -> Integer` que calcula o máximo divisor comum entre dois números, baseada na seguinte propriedade (atribuida a *Euclides*):
   `mdc x y == mdc (x+y) y == mdc x (y+x)`

   (b) Defina `Frac` como instância da classe `Eq`.

   (c) Defina `Frac` como instância da classe `Ord`.

   (d) Defina `Frac` como instância da classe `Show`, de forma a que cada fracção seja apresentada por *(numerador/denominador)*.

   (e) Defina `Frac` como instância da classe `Num`. Relembre que a classe `Num` tem a seguinte definição

   ```
   class (Eq a, Show a) => Num a where
     (+), (*), (-) :: a -> a -> a
     negate, abs, signum :: a -> a
     fromInteger :: Integer -> a
   ```

   (f) Defina uma função que, dada uma fracção `f` e uma lista de fracções `l`, selecciona de `l` os elementos que são maiores do que o dobro de `f`.

2. Relembre o tipo definido na Ficha 7 para representar expressões inteiras. Uma possível generalização desse tipo de dados, será considerar expressões cujas constantes são de um qualquer tipo numérico (i.e., da classe `Num`).

   ```
   data Exp a = Const a
              | Simetrico (Exp a)
              | Mais (Exp a) (Exp a)
              | Menos (Exp a) (Exp a)
              | Mult (Exp a) (Exp a)
   ```

   (a) Declare `Exp a` como uma instância de `Show`.

   (b) Declare `Exp a` como uma instância de `Eq`.

   (c) Declare `Exp a` como instância da classe `Num`.

3. Relembre o exercício da Ficha 3 sobre contas bancárias, com a seguinte declaração de tipos

   ```
   data Movimento = Credito Float | Debito Float
   data Data = D Int Int Int
   data Extracto = Ext Float [(Data, String, Movimento)]
   ```

(a) Defina `Data` como instância da classe `Ord`.

(b) Defina `Data` como instância da classe `Show`.

(c) Defina a função `ordena :: Extracto -> Extracto`, que transforma um extracto de modo a que a lista de movimentos apareça ordenada por ordem crescente de data.

(d) Defina `Extracto` como instância da classe `Show`, de forma a que a apresentação do extracto seja por ordem de data do movimento com o seguinte, e com o seguinte aspecto

```
Saldo anterior: 300
--------------------------------------
Data       Descricao   Credito   Debito
--------------------------------------
2010/4/5   DEPOSITO    2000
2010/8/10  COMPRA                37,5
2010/9/1   LEV                   60
2011/1/7   JUROS       100
2011/1/22  ANUIDADE              8
--------------------------------------
Saldo actual: 2294,5
```

Programação Funcional

Ficha 9

Input / Output

1. A classe `Random` da biblioteca `System.Random` agrupa os tipos para os quais é possível gerar valores aleatórios. Algumas das funções declaradas nesta classe são:

   - `randomIO :: Random a => IO a` que gera um valor aleatório do tipo `a`;
   - `randomRIO :: Random a => (a,a) -> IO a` que gera um valor aleatório do tipo `a` dentro de uma determinada gama de valores.

   Usando estas funções implemente os seguintes programas:

   (a) `bingo :: IO ()` que sorteia os números para o jogo do bingo. Sempre que uma tecla é pressionada é apresentado um número aleatório entre 1 e 90. Obviamente, não podem ser apresentados números repetidos e o programa termina depois de gerados os 90 números diferentes.

   (b) `mastermind :: IO ()` que implementa uma variante do jogo de descodificação de padrões *Mastermind*. O programa deve começar por gerar uma sequência secreta de 4 dígitos aleatórios que o jogador vai tentar descodificar. Sempre que o jogador introduz uma sequência de 4 dígitos, o programa responde com o número de dígitos com o valor correcto na posição correcta e com o número de dígitos com o valor correcto na posição errada. O jogo termina quando o jogador acertar na sequência de dígitos secreta.

2. Uma aposta do EuroMilhões corresponde à escolha de 5 *Números* e 2 *Estrelas*. Os *Números* são inteiros entre 1 e 50. As *Estrelas* são inteiros entre 1 e 9. Para modelar uma aposta destas definiu-se o seguinte tipo de dados:

   ```
   data Aposta = Ap [Int] (Int,Int)
   ```

   (a) Defina a função `valida :: Aposta -> Bool` que testa se uma dada aposta é válida (i.e. tem os 5 números e 2 estrelas, dentro dos valores aceites e não tem repetições).

   (b) Defina a função `comuns :: Aposta -> Aposta -> (Int,Int)` que dada uma aposta e uma chave, calcula quantos *números* e quantas *estrelas* existem em comum nas duas apostas

   (c) Use a função da alínea anterior para:

      i. Definir `Aposta` como instância da classe `Eq`.

      ii. Definir a função `premio :: Aposta -> Aposta -> Maybe Int` que dada uma aposta e a chave do concurso, indica qual o prémio que a aposta tem.
      Os prémios do EuroMilhões são:

Números	*Estrelas*	**Prémio**	*Números*	*Estrelas*	**Prémio**
5	2	**1**	3	2	**7**
5	1	**2**	2	2	**8**
5	0	**3**	3	1	**9**
4	2	**4**	3	0	**10**
4	1	**5**	1	2	**11**
4	0	**6**	2	1	**12**
			2	0	**13**

(d) Para permitir que um apostador possa jogar de forma interactiva:

   i. Defina a função `leAposta :: IO Aposta` que lê do teclado uma aposta. Esta função deve garantir que a aposta produzida é válida.

   ii. Defina a função `joga :: Aposta -> IO ()` que recebe a chave do concurso, lê uma aposta do teclado e imprime o prémio no ecrã.

(e) Defina a função `geraChave :: IO Aposta`, que gera uma chave válida de forma aleatória.

(f) Pretende-se agora que o programa `main` permita jogar várias vezes e dê a possiblidade de simular um novo concurso (gerando uma nova chave). Complete o programa definindo a função `ciclo :: Aposta -> IO ()`.

```
main :: IO ()
main = do ch <- geraChave
          ciclo ch

menu :: IO String
menu = do { putStrLn menutxt
          ; putStr "Opcao: "
          ; c <- getLine
          ; return c
          }
    where menutxt = unlines ["",
                             "Apostar ........... 1",
                             "Gerar nova chave .. 2",
                             "",
                             "Sair .............. 0"]
```

Programação Funcional

1º Ano – LCC/LEF/LEI

Questões

1. Apresente uma definição recursiva da função (pré-definida) `enumFromTo :: Int -> Int -> [Int]` que constrói a lista dos números inteiros compreendidos entre dois limites.

   Por exemplo, `enumFromTo 1 5` corresponde à lista `[1,2,3,4,5]`

2. Apresente uma definição recursiva da função (pré-definida) `enumFromThenTo :: Int -> Int -> Int -> [Int]` que constrói a lista dos números inteiros compreendidos entre dois limites e espaçados de um valor constante.

   Por exemplo, `enumFromThenTo 1 3 10` corresponde à lista `[1,3,5,7,9]`.

3. Apresente uma definição recursiva da função (pré-definida) `(++) :: [a] -> [a] -> [a]` que concatena duas listas.

   Por exemplo, `(++) [1,2,3] [10,20,30]` corresponde à lista `[1,2,3,10,20,30]`.

4. Apresente uma definição recursiva da função (pré-definida) `(!!) :: [a] -> Int -> a` que dada uma lista e um inteiro, calcula o elemento da lista que se encontra nessa posição (assume-se que o primeiro elemento se encontra na posição `0`).

   Por exemplo, `(!!) [10,20,30] 1` corresponde a `20`.

   Ignore os casos em que a função não se encontra definida (i.e., em que a posição fornecida não corresponde a nenhuma posição válida da lista).

5. Apresente uma definição recursiva da função (pré-definida) `reverse :: [a] -> [a]` que dada uma lista calcula uma lista com os elementos dessa lista pela ordem inversa.

   Por exemplo, `reverse [10,20,30]` corresponde a `[30,20,10]`.

6. Apresente uma definição recursiva da função (pré-definida) `take :: Int -> [a] -> [a]` que dado um inteiro `n` e uma lista `l` calcula a lista com os (no máximo) `n` primeiros elementos de `l`.

   A lista resultado só terá menos de que `n` elementos se a lista `l` tiver menos do que `n` elementos. Nesse caso a lista calculada é igual à lista fornecida.

   Por exemplo, `take 2 [10,20,30]` corresponde a `[10,20]`.

7. Apresente uma definição recursiva da função (pré-definida) `drop :: Int -> [a] -> [a]` que dado um inteiro `n` e uma lista `l` calcula a lista sem os (no máximo) `n` primeiros elementos de `l`.

   Se a lista fornecida tiver `n` elementos ou menos, a lista resultante será vazia.

   Por exemplo, `drop 2 [10,20,30]` corresponde a `[30]`.

8. Apresente uma definição recursiva da função (pré-definida) `zip :: [a] -> [b] -> [(a,b)]` constói uma lista de pares a partir de duas listas.

   Por exemplo, `zip [1,2,3] [10,20,30,40]` corresponde a `[(1,10),(2,20),(3,30)]`.

9. Apresente uma definição recursiva da função (pré-definida) `replicate :: Int -> a -> [a]` que dado um inteiro `n` e um elemento `x` constói uma lista com `n` elementos, todos iguais a `x`.

   Por exemplo, `replicate 3 10` corresponde a `[10,10,10]`.

10. Apresente uma definição recursiva da função (pré-definida) `intersperse :: a -> [a] -> [a]` que dado um elemento e uma lista, constrói uma lista em que o elemento fornecido é *intercalado* entre os elementos da lista fornecida.

    Por exemplo, `intersperce 1 [10,20,30]` corresponde a `[10,1,20,1,30]`.

11. Apresente uma definição recursiva da função (pré-definida) `group :: Eq a => [a] -> [[a]]` que agrupa elementos iguais e consecutivos de uma lista.

    Por exemplo, `group [1,2,2,3,4,4,4,5,4]` corresponde a `[[1],[2,2],[3],[4,4,4],[5],[4]]`.

12. Apresente uma definição recursiva da função (pré-definida) `concat :: [[a]] -> [a]` que concatena as listas de uma lista.

    Por exemplo, `concat [[1],[2,2],[3],[4,4,4],[5],[4]]` corresponde a `[1,2,2,3,4,4,4,5,4]`.

13. Apresente uma definição recursiva da função (pré-definida) `inits :: [a] -> [[a]]` que calcula a lista dos prefixos de uma lista.

    Por exemplo, `inits [11,21,13]` corresponde a `[[],[11],[11,21],[11,21,13]]`.

14. Apresente uma definição recursiva da função (pré-definida) `tails :: [a] -> [[a]]` que calcula a lista dos sufixos de uma lista.

    Por exemplo, `tails [1,2,3]` corresponde a `[[1,2,3],[2,3],[3],[]]`.

15. Defina a função `heads :: [[a]] -> [a]` que recebe uma lista de listas e produz a lista com o primeiro elemento de cada lista.

    Por exemplo, `heads [[2,3,4],[1,7],[],[8,5,3]]` corresponde a `[2,1,8]`.

16. Defina a função `total :: [[a]] -> Int` que recebe uma lista de listas e conta o total de elementos (de todas as listas)

    Por exemplo, `total [[2,3,4],[1,7],[],[8,5,3]]` corresponde a `8`.

17. Defina a função `fun :: [(a,b,c)] -> [(a,c)]` que recebe uma lista de triplos e produz a lista de pares com o primeiro e o terceiro elemento de cada triplo.

    Por exemplo, `fun [("rui",3,2), ("maria",5,2), ("ana",43,7)]` corresponde a `[("rui",2), ("maria",2), ("ana",7)]`.

18. Defina a função `cola :: [(String,b,c)] -> String` que recebe uma lista de triplos e concatena as strings que estão na primeira componente dos triplos.

    Por exemplo, `cola [("rui",3,2), ("maria",5,2), ("ana",43,7)]` corresponde a `"ruimariaana"`.

19. Defina a função `idade :: Int -> Int -> [(String,Int)] -> [String]` que recebe o ano, a idade e uma lista de pares com o nome e o ano de nascimento de cada pessoa, e devolve a listas de nomes das pessoas que nesse ano atingirão ou já ultrapassaram a idade indicada.

    Por exemplo, `idade 2021 26 [("rui",1995), ("maria",2009), ("ana",1947)]` corresponde a `["rui","ana"]`.

20. Apresente uma definição recursiva da função,

    ```
    powerEnumFrom ::  Int -> Int -> [Int]
    ```

    que dado um valor `n` e um valor `m` constrói a lista $[\texttt{n}^0, \ldots, \texttt{n}^{\texttt{m}-1}]$.

21. Apresente uma definição recursiva da função,

    ```
    isPrime ::  Int -> Bool
    ```

    que dado um número inteiro maior ou igual a `2` determina se esse número é primo. Para determinar se um número `n` é primo, descubra se existe algum número inteiro `m` tal que $\texttt{2} \leq \texttt{m} \leq \sqrt{\texttt{n}}$ e `mod n m` $= \texttt{0}$. Se um tal número não existir então `n` é primo, e se existir então `n` não é primo.

22. Apresente uma definição recursiva da função (pré-definida) `isPrefixOf :: Eq a => [a] -> [a] -> Bool` que testa se uma lista é prefixo de outra.

    Por exemplo, `isPrefixOf [10,20] [10,20,30]` corresponde a `True` enquanto que `isPrefixOf [10,30] [10,20,30]` corresponde a `False`.

23. Apresente uma definição recursiva da função (pré-definida) `isSuffixOf :: Eq a => [a] -> [a] -> Bool` que testa se uma lista é sufixo de outra.

    Por exemplo, `isSuffixOf [20,30] [10,20,30]` corresponde a `True` enquanto que `isSuffixOf [10,30] [10,20,30]` corresponde a `False`.

24. Apresente uma definição recursiva da função (pré-definida) `isSubsequenceOf :: Eq a => [a] -> [a] -> Bool` que testa se os elementos de uma lista ocorrem noutra pela mesma ordem relativa.

    Por exemplo, `isSubsequenceOf [20,40] [10,20,30,40]` corresponde a `True` enquanto que `isSubsequenceOf [40,20] [10,20,30,40]` corresponde a `False`.

25. Apresente uma definição recursiva da função (pré-definida) `elemIndices :: Eq a => a -> [a] -> [Int]` que calcula a lista de posições em que um dado elemento ocorre numa lista.

    Por exemplo, `elemIndices 3 [1,2,3,4,3,2,3,4,5]` corresponde a `[2,4,6]`.

26. Apresente uma definição recursiva da função (pré-definida) `nub :: Eq a => [a] -> [a]` que calcula uma lista com os mesmos elementos da recebida, sem repetições.

    Por exemplo, `nub [1,2,1,2,3,1,2]` corresponde a `[1,2,3]`.

27. Apresente uma definição recursiva da função (pré-definida) `delete :: Eq a => a -> [a] -> [a]` que retorna a lista resultante de remover (a primeira ocorrência de) um dado elemento de uma lista.

    Por exemplo, `delete 2 [1,2,1,2,3,1,2]` corresponde a `[1,1,2,3,1,2]`. Se não existir nenhuma ocorrência a função deverá retornar a lista recebida.

28. Apresente uma definição recursiva da função (pré-definida) `(\\):: Eq a => [a] -> [a] -> [a]` que retorna a lista resultante de remover (as primeiras ocorrências) dos elementos da segunda lista da primeira.

    Por exemplo, `(\\)[1,2,3,4,5,1] [1,5]` corresponde a `[2,3,4,1]`.

29. Apresente uma definição recursiva da função (pré-definida) `union :: Eq a => [a] -> [a] -> [a]` que retorna a lista resultante de acrescentar à primeira lista os elementos da segunda que não ocorrem na primeira.

    Por exemplo, `union [1,1,2,3,4] [1,5]` corresponde a `[1,1,2,3,4,5]`.

30. Apresente uma definição recursiva da função (pré-definida) `intersect :: Eq a => [a] -> [a] -> [a]` que retorna a lista resultante de remover da primeira lista os elementos que não pertencem à segunda.

    Por exemplo, `intersect [1,1,2,3,4] [1,3,5]` corresponde a `[1,1,3]`.

31. Apresente uma definição recursiva da função (pré-definida) `insert :: Ord a => a -> [a] -> [a]` que dado um elemento e uma lista ordenada retorna a lista resultante de inserir ordenadamente esse elemento na lista.

    Por exemplo, `insert 25 [1,20,30,40]` corresponde a `[1,20,25,30,40]`.

32. Apresente uma definição recursiva da função (pré-definida) `unwords :: [String] -> String` que junta todas as strings da lista numa só, separando-as por um espaço.

    Por exemplo, `unwords ["Programacao", "Funcional"]` corresponde a `"Programacao Funcional"`.

33. Apresente uma definição recursiva da função (pré-definida) `unlines :: [String] -> String` que junta todas as strings da lista numa só, separando-as pelo caracter `'\n'`.

    Por exemplo, `unlines ["Prog", "Func"]` corresponde a `"Prog\nFunc\n"`.

34. Apresente uma definição recursiva da função `pMaior :: Ord a => [a] -> Int` que dada uma lista não vazia, retorna a posição onde se encontra o maior elemento da lista. As posições da lista começam em `0`, i.e., a função deverá retornar `0` se o primeiro elemento da lista for o maior.

35. Apresente uma definição recursiva da função (pré-definida) `lookup :: Eq a => a -> [(a,b)] -> Maybe b` que retorna uma lista construída a partir de elementos de uma lista (o segundo argumento) atendendo a uma condição dada pelo primeiro argumento.

    Por exemplo, `lookup 'a' [('a',1),('b',4),('c',5)]` corresponde à lista `Just 1`.

36. Defina a função `preCrescente :: Ord a => [a] -> [a]` calcula o maior prefixo crescente de uma lista.

    Por exemplo, `preCrescente [3,7,9,6,10,22]` corresponde a `[3,7,9]`.

37. Apresente uma definição recursiva da função `iSort :: Ord a => [a] -> [a]` que calcula o resultado de ordenar uma lista. Assuma, se precisar, que existe definida a função `insert :: Ord a => a -> [a] -> [a]` que dado um elemento e uma lista ordenada retorna a lista resultante de inserir ordenadamente esse elemento na lista.

38. Apresente uma definição recursiva da função `menor :: String -> String -> Bool` que dadas duas strings, retorna `True` se e só se a primeira for menor do que a segunda, segundo a ordem lexicográfica (i.e., do dicionário)

    Por exemplo, `menor "sai" "saiu"` corresponde a `True` enquanto que `menor "programacao" "funcional"` corresponde a `False`.

39. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

    Defina a função `elemMSet :: Eq a => a -> [(a,Int)] -> Bool` que testa se um elemento pertence a um multi-conjunto.

    Por exemplo, `elemMSet 'a' [('b',2), ('a',4), ('c',1)]` corresponde a `True` enquanto que `elemMSet 'd' [('b',2), ('a',4), ('c',1)]` corresponde a `False`.

40. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

    Defina a função `converteMSet :: [(a,Int)] -> [a]` que converte um multi-conjuto na lista dos seus elementos

    Por exemplo, `converteMSet [('b',2), ('a',4), ('c',1)]` corresponde a `"bbaaaac"`.

41. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

    Defina a função `insereMSet :: Eq a => a -> [(a,Int)] -> [(a,Int)]` que acrescenta um elemento a um multi-conjunto.

    Por exemplo, `insereMSet 'c' [('b',2), ('a',4), ('c',1)]` corresponde a `[('b',2), ('a',4), ('c',2)]`.

42. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

    Defina a função `removeMSet :: Eq a => a -> [(a,Int)] -> [(a,Int)]` que remove um elemento a um multi-conjunto. Se o elemento não existir, deve ser retornado o multi-conjunto recebido.

    Por exemplo, `removeMSet 'c' [('b',2), ('a',4), ('c',1)]` corresponde a `[('b',2), ('a',4)]`.

43. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

Defina a função `constroiMSet :: Ord a => [a] -> [(a,Int)]` dada uma lista ordenada por ordem crescente, calcula o multi-conjunto dos seus elementos.

Por exemplo, `constroiMSet "aaabccc"` corresponde a `[('a',3), ('b',1), ('c',3)]`.

44. Apresente uma definição recursiva da função pré-definida `partitionEithers :: [Either a b] -> ([a],[b])` que divide uma lista de *Either*s em duas listas.

45. Apresente uma definição recursiva da função pré-definida `catMaybes :: [Maybe a] -> [a]` que colecciona os elementos do tipo `a` de uma lista.

46. Considere o seguinte tipo para representar movimentos de um robot.

```
data Movimento = Norte | Sul | Este | Oeste
               deriving Show
```

Defina a função `caminho :: (Int,Int) -> (Int,Int) -> [Movimento]` que, dadas as posições inicial e final (coordenadas) do robot, produz uma lista de movimentos suficientes para que o robot passe de uma posição para a outra.

47. Consider o seguinte tipo de dados,

```
data Movimento = Norte | Sul | Este | Oeste
               deriving Show
```

Defina a função `hasLoops :: (Int,Int) -> [Movimento] -> Bool` que dada uma posição inicial e uma lista de movimentos (correspondentes a um percurso) verifica se o robot alguma vez volta a passar pela posição inicial ao longo do percurso correspondente. Pode usar a função `posicao` definida acima.

48. Considere os seguintes tipos para representar pontos e rectângulos, respectivamente. Assuma que os rectângulos têm os lados paralelos aos eixos e são representados apenas por dois dos pontos mais afastados.

```
type Ponto = (Float,Float)
data Rectangulo = Rect Ponto Ponto
```

Defina a função `contaQuadrados :: [Rectangulo] -> Int` que, dada uma lista com rectângulos, conta quantos deles são quadrados.

49. Considere os seguintes tipos para representar pontos e rectângulos, respectivamente. Assuma que os rectângulos têm os lados paralelos aos eixos e são representados apenas por dois dos pontos mais afastados.

```
type Ponto = (Float,Float)
data Rectangulo = Rect Ponto Ponto
```

Defini a função `areaTotal :: [Rectangulo] -> Float` que, dada uma lista com rectângulos, determina a área total que eles ocupam.

50. Considere o seguinte tipo para representar o estado de um equipamento.

```
data Equipamento = Bom | Razoavel | Avariado
                 deriving Show
```

Defina a função `naoReparar :: [Equipamento] -> Int` que determina a quantidade de equipamentos que não estão avariados.