Ficha 1

Programaçao Funcional

2015/16

1. Usando as seguintes funçoes pré-definidas do Haskell:
   - `length l`: o número de elementos da lista `l`
   - `head l`: a cabeca da lista (nao vazia) `l`
   - `tail l`: a cauda lista (nao vazia) `l`
   - `last l`: o último elemento da lista (nao vazia) `l`
   - `sqrt x`: a raiz quadrada de `x`
   - `div x y`: a divisao inteira de `x` por `y`
   - `mod x y`: o resto da divisão inteira de `x` por `y`

   defina as seguintes funçoes:

   (a) `perimetro` – que calcula o perímetro de uma circunferência, dado o comprimento do seu raio.
   ```
   -- O valor de pi está predefinido no Haskell.
   -- perimetro :: Float -> Float
   perimetro :: Floating a => a -> a  --Aqui o tipo abrange todos os tipos flutuantes (Double, Float, etc).
   perimetro r = 2 * pi * r
   ```

   (b) `dist` – que calcula a distancia entre dois pontos no plano Cartesiano. Cada ponto é um par de valores do tipo `Float`.
   ```
   dist :: (Float,Float) -> (Float,Float) -
   > Float
   dist (a,b) (x,y) = sqrt(((x - a)**2) +
   ((y - b)**2))
   ```

   (b) `primUlt` – que recebe uma lista e devolve um par com o primeiro e o último elemento dessa lista.
   ```
   primUlt :: [a] -> (a,a)
   primUlt l = (head l, last l)
   ```

   (c) `multiplo` – tal que `multiplo m n` testa se o número inteiro `m` é múltiplo de `n`.
   ```
   multiplo :: Int -> Int -> Bool
   multiplo m n = mod m n == 0
   -- mesmo que: multiplo m n = if(mod m n == 0) then True else False
   ```

   (e) `truncaImpar` – que recebe uma lista e, se o comprimento da lista for ímpar retira-lhe o primeiro elemento, caso contrario devolve a propria lista.
   ```
   truncaImpar :: [a] -> [a]
   truncaImpar l = if(mod (length l) 2 /= 0) then tail l else l
   ```

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

```
max2 :: Int -> Int -> Int
max2 a b = if(a > b) then a else b
```

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

```
max3 :: Int -> Int -> Int -> Int
max3 a b c = max2 (max2 a b) c
```

2. Definia as seguintes funçoes sobre polinómios de 2ª grau:

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

```
nRaizes :: (Floating a, Ord a) => a -> a -> a -> Int
nRaizes a b c | delta > 0    = 2
              | delta == 0   = 1
              | delta < 0    = 0
                  where delta = (b**2) - (4 * a * c)
```

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

```
raizes :: (Floating a, Ord a) => a -> a -> a -> [a]
raizes a b c | nR == 0  = []
             | nR == 1  = [(-b) / (2 * a)]
             | nR == 2  = [((-b) + sqrt((b**2) - (4 * a * c))) / (2 * a), ((-b) -
sqrt((b**2) - (4 * a * c))) / (2 * a)]
          where nR = nRaizes a b c
```

3. Vamos representar um ponto por um par de números que representam as suas coordenadas no plano Cartesiano.

```
type Ponto = (Float,Float)
```

   (a) Defina uma funcao que recebe 3 pontos que são os vértices de um triangulo e devolve um tuplo com o comprimento dos seus lados.

```
ladosTri :: Ponto -> Ponto -> Ponto -> (Float,Float,Float)
ladosTri x y z = (dist x y, dist y z, dist z x)
```

   (b) Defina uma funcão que recebe 3 pontos que são os vértices de um triangulo e calcula o perímetro desse triangulo.

```
periTri :: Ponto -> Ponto -> Ponto -> Float
periTri x y z = a + b + c where (a,b,c) = ladosTri x y z
```

   (c) Defina uma funcao que recebe 2 pontos que sao os vértices da diagonal de um rectangulo paralelo aos eixos e constroi uma lista com os 4 pontos desse rectângulo.

```
listaPontosRect :: Ponto -> Ponto -> [Ponto]
listaPontosRect (a,b) (x,y) = [(a,b), (a,y), (x,y), (x,b)]
```

4. 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 funçoes para:

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

horaValida :: Hora -> Bool
horaValida (h,m) = (h >= 0) && (h < 24) && (m >= 0) && (m < 60)

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

horaDepois :: Hora -> Hora -> Bool
horaDepois (a,b) (x,y) = (a < x) || ((a == x) && (b < y))

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

hora2Mins :: Hora -> Int
hora2Mins (h,m) = (h * 60) + m

(c) converter um valor em minutos para horas;

mins2Hora :: Int -> Hora
mins2Hora m = (div m 60, mod m 60)

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

diferHoras :: Hora -> Hora -> Int
diferHoras a b = if(horaDepois a b) then (hora2Mins b) - (hora2Mins a) else (hora2Mins a) - (hora2Mins b)

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

adicionaMinutos :: Hora -> Int -> Hora
adicionaMinutos hora mins = mins2Hora ((hora2Mins hora) + mins)