Ficha 7

Programaçao Funcional

2015/16

1. Para armazenar uma agenda de contactos telefónicos e de correio electronico 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 funçao `acrescEmail :: Nome -> String -> Agenda -> Agenda` que, dado um nome, um email e uma agenda, acrescenta essa informacão à agenda.
   acrescEmail :: Nome -> String -> Agenda -> Agenda
   acrescEmail nome email [] = [(nome,[Email email])]
   acrescEmail nome email ((name,list):ts) = if (nome == name) then (nome, ((Email email) : list)) : ts else (name,list) : (acrescEmail nome email ts)

   (b) Defina a funçao `verEmails :: Nome -> Agenda -> Maybe [String]` que, dado um nome e uma agenda, retorna a lista dos emails associados a esse nome. Se esse nome nao existir na agenda a funçao deve retornar `Nothing`.
   verEmails :: Nome -> Agenda -> Maybe [String]
   verEmails _ [] = Nothing
   verEmails nome ((name,list):ts) = if (nome == name) then Just (searchEmailsAux list) else verEmails nome ts where
   searchEmailsAux [] = []
   searchEmailsAux ((Email h):ts) = h : (searchEmailsAux ts)
   searchEmailsAux (_:ts) = searchEmailsAux ts

   (c) Defina a funçao `consTelefs :: [Contacto] -> [Integer]` que, dada uma lista de contactos, retorna a lista de todos os números de telefone dessa lista (tanto telefones fixos como telemoveis).
   consTelefs :: [Contacto] -> [Integer]
   consTelefs [] = []
   consTelefs ((Casa x):ts) = x : (consTelefs ts)
   consTelefs ((Trab x):ts) = x : (consTelefs ts)
   consTelefs ((Tlm x):ts) = x : (consTelefs ts)
   consTelefs (_:ts) = consTelefs ts

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

```
casa :: Nome -> Agenda -> Maybe Integer
casa _ [] = Nothing
casa nome ((name,list):ts) = if (nome == name) then casaAux list else casa nome ts
where
 casaAux [] = Nothing
 casaAux ((Casa x):ts) = Just x
 casaAux (_:ts) = casaAux ts
```

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

```
procura :: Nome -> TabDN -> Maybe Data
procura _ [] = Nothing
procura nome ((name,date):ts) = if (name == nome) then Just date else procura
nome ts
```

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

```
calcIdade (D d m a) (D dd mm aa) | (a >= aa) = 0
                                 | ((a < aa) && ((m > mm) || ((m <= mm) && (d
> dd)))) = aa - a - 1
                                 | ((a < aa) && (m <= mm) && (d <= dd)) = aa –
a

idade :: Data -> Nome -> TabDN -> Maybe Int
idade _ _ [] = Nothing
idade dt nome ((name,date):ts) | (name == nome) = Just (calcIdade date dt)
                                                | otherwise = idade dt nome ts
```

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

```
anterior :: Data -> Data -> Bool
anterior (D d m a) (D dd mm aa) = (a < aa) || ((a == aa) && (m < mm)) || ((a == aa)
&& (m == mm) && (d < dd))
```

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

```
insereOrdData :: (Nome,Data) -> TabDN -> TabDN
insereOrdData (nm,dt) [] = [(nm,dt)]
insereOrdData (nm,dt) ((name,date):ts) = if (anterior dt date) then (nm,dt) : ((name,date) : ts) else (name,date) : (insereOrdData (nm,dt) ts)

ordena :: TabDN -> TabDN
ordena tab = oAux [] tab where
        oAux new [] = new
        oAux new (h:ts) = oAux (insereOrdData h new) ts
```

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

```
porIdade :: Data -> TabDN -> [(Nome,Int)]
porIdade dt tab = let (a,b) = (unzip (ordena tab)) in zip a (map (\x -> calcIdade x dt) b)
```

3. Considere o seguinte tipo de dados que descreve a informação de um extracto bancario. 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       -- Dia Mes Ano
  deriving Show
data Extracto = Ext Float [(Data, String, Movimento)]
  deriving Show
```

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

```
extValor :: Extracto -> Float -> [Movimento]
extValor (Ext _ []) _ = []
extValor (Ext x ((_, _, Credito y):ts)) valor = if (y > valor) then (Credito y) : (extValor (Ext x ts) valor) else extValor (Ext x ts) valor
extValor (Ext x ((_, _, Debito y):ts)) valor = if (y > valor) then (Debito y) : (extValor (Ext x ts) valor) else extValor (Ext x ts) valor
```

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

```
filtro :: Extracto -> [String] -> [(Data,Movimento)]
filtro (Ext _ []) filters = []
filtro (Ext v ((date, desc, movimento):ts)) filters = if (elem desc filters) then (date,movimento) : (filtro (Ext v ts) filters) else filtro (Ext v ts) filters
```

(c) Defina a funçao `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.

```
creDeb :: Extracto -> (Float,Float)
creDeb (Ext _ []) = (0,0)
creDeb (Ext v ((_, _, Credito x):ts)) = let (a,b) = creDeb (Ext v ts) in (x + a, b)
creDeb (Ext v ((_, _, Debito y):ts)) = let (a,b) = creDeb (Ext v ts) in (a, b + y)
```

```
creDeb' :: Extracto -> (Float,Float)
creDeb' (Ext valor movimentos) = cDAux movimentos where
    cDAux [] = (0,0)
    cDAux ((_, _, Credito x):ts) = let (a,b) = cDAux ts in (a + x, b)
    cDAux ((_, _, Debito y):ts) = let (a,b) = cDAux ts in (a, b + y)
```

(d) Defina de novo a funcao `creDeb :: Extracto -> (Float,Float)` usando um `foldr`.

```
creDebF (Ext valor movimentos) = cDAux movimentos where
                                    cDAux lista = foldr fAux (0,0) lista
                                    fAux (_, _, Credito x) (a,b) = (x + a, b)
                                    fAux (_, _, Debito y) (a,b) = (a, y + b)
```

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

```
saldo :: Extracto -> Float
saldo (Ext inicial movimentos) = let (a,b) = creDeb (Ext inicial movimentos) in
inicial + a - b
```

(f) Defina de novo a funcão `creDeb :: Extracto -> (Float,Float)` usando um `foldr`.

```
saldoF :: Extracto -> Float
saldoF (Ext inicial movimentos) = foldr fAux inicial movimentos where
  fAux (_, _, Credito x) resto = x + resto
  fAux (_, _, Debito y) resto = resto - y
```