Ficha 3

Programaçao Funcional

2015/16

1. Indique como é que o interpretador de haskell avalia as expressões das alíneas que se seguem, apresentando a cadeia de reduçao de cada uma dessas expressões.

   (a) Considere a seguinte definição

   ```
   p :: Int -> Bool
   p 0 = True
   p 1 = False
   p x | x > 1 = p (x-2)
   ```

   Diga, justificando, qual é o valor de `p 5`.

   (b) Considere a seguinte definição

   ```
   f l = g [] l
   g l [] = l
   g l (h:t) = g (h:l) t
   ```

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

   (c) Considere a seguinte definição

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

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

2. Defina as seguintes funçoes sobre listas de tuplos:

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

   (b) `nosPrimeiros :: (Eq a) => a -> [(a,b)] -> Bool` que testa se um elemento aparece na lista como primeira componente de algum dos pares.
   nosPrimeiros :: (Eq a) => a -> [(a,b)] -> Bool
   nosPrimeiros _ [] = False
   nosPrimeiros x (h:ts) = if ((fst h) == x) then True else nosPrimeiros x ts

(c) `minFst :: (Ord a) => [(a,b)] -> a` que calcula a menor primeira componente.
Por exemplo, `minFst [(10,21), (3, 55), (66,3)] = 3`

```
minFst :: (Ord a) => [(a,b)] -> a
minFst list = mfAux list (fst (head list)) where
  mfAux [] w = w
  mfAux ((x,y):ts) w = if (x < w) then mfAux ts x else mfAux ts w
```

(d) `sndMinFst :: (Ord a) => [(a,b)] -> b` que calcula a segunda componente associada à menor primeira componente.
Por exemplo, `sndMinFst [(10,21), (3, 55), (66,3)] = 55`

```
sndMinFst :: (Ord a) => [(a,b)] -> b
sndMinFst list = smfAux list (head list) where
  smfAux [] (x,y) = y
  smfAux ((w,z):ts) (x,y) = if (w < x) then smfAux ts (w,z) else smfAux ts (x,y)
```

(e) `sumTriplos :: (Num a, Numb, 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)`

```
sumTriplos :: (Num a, Num b, Num c) => [(a,b,c)] -> (a,b,c)
sumTriplos [] = (0,0,0)
sumTriplos ((x,y,z):ts) = somaTriplo (x,y,z) (sumTriplos ts) where somaTriplo (x,y,z)
(xs,ys,zs) = (x + xs,y + ys,z + zs)
```

(f) `maxTriplo :: (Ord a, Num a) => [(a,a,a)] -> a` que calcula o maximo valor da soma das componentes de cada triplo de uma lista.
Por exemplo, `maxTriplo [(10,-4,21), (3, 55,20), (-8,66,4)] = 78`

```
maxTriplo :: (Ord a, Num a) => [(a,a,a)] -> a
maxTriplo ((x,y,z):ts) = mTAux ts (x + y + z) where
  mTAux [] big = big
  mTAux ((x,y,z):ts) big = if ((x + y + z) > big) then mTAux ts (x + y + z) else
mTAux ts big
```

3. Defina recursivamentes as seguintes funções sobre números inteiros nao negativos:

(a) `(><) :: Int -> Int -> Int` para multiplicar dois números inteiros (por somas sucessivas).

```
(><) :: Int -> Int -> Int
(><) x y = if (x > y) then mAux x y 0 else mAux y x 0 where
  mAux _ 0 res = res
  mAux x y res = mAux x (y - 1) (res + x)
```

(b) `div, mod :: Int -> Int -> Int` que calculam a divisão e o resto da divisão inteiras por subtraccoes sucessivas.

```
div' :: Int -> Int -> Int
div' x y = if (x < y) then 0 else 1 + (div (x - y) y)
mod' :: Int -> Int -> Int
mod' x y = if (x < y) then x else mod (x-y) y
```

(c) `power :: Int -> Int -> Int` que calcula a potência inteira de um número por multiplicaçoes sucessivas.

```
power :: Int -> Int -> Int
power x 1 = x
power x y = x * (power x (y-1))
```

4. 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):

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

Por exemplo, se uma viagem for

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

significa que teve três etapas:

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

Para este problema, vamos trabalhar apenas com viagens que começam e acabam no mesmo dia.

Utilizando as funcoes 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 sao validas).

```
etapaValida :: Etapa -> Bool
etapaValida (x,y) = (horaValida x) && (horaValida y) && (horaDepois x y)
```

(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).

```
viagemValida :: Viagem -> Bool
viagemValida [] = True
viagemValida [x] = etapaValida x
viagemValida ((x,xs):(y,ys):ts) = if ((etapaValida (x,xs)) && (horaDepois xs y))
then viagemValida ((y,ys):ts) else False
```

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

```
chegadaPartidaViagem :: Viagem -> (Hora,Hora)
chegadaPartidaViagem list = cPVAux list (head list) where
 cPVAux [] x = x
 cPVAux ((x,y):ts) (st,end) = cPVAux ts (st,y)
```

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

```
tempoEmViagem :: Viagem -> Hora

tempoEmViagem [] = (0,0)

tempoEmViagem ((x,y):ts) = adicionaMinutos (tempoEmViagem ts)
(diferHoras x y)
```

(d) Calcular o tempo total de espera.

```
tempoEmEspera :: Viagem -> Hora
tempoEmEspera [] = (0,0)
tempoEmEspera [x] = (0,0)
tempoEmEspera ((x,xs):(y,ys):ts) = adicionaMinutos
(tempoEmEspera ((y,ys):ts)) (diferHoras xs y)
```

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

```
tempoTotalViagem :: Viagem -> Hora
tempoTotalViagem vg = let (horaComeco,horaFim) = chegadaPartidaViagem vg
                      in mins2Hora (diferHoras horaComeco horaFim)
```