Centro de Informática - UFPE
Primeiro Exercício Escolar — IF686 (2017.2)
Data: 03/10/2017 Horário: 13h-15h
Docente: Márcio Lopes Cornélio
Discente:

Instruções para entrega: Enviar dois arquivos para *mlc2@cin.ufpe.br* : (1) arquivo ASCII com todas as respostas (copiar e colar os códigos das respostas) e (2) um arquivo no formato pdf com todas as respostas. Identificar-se nos arquivos (colocar nome). O assunto do email deve estar no formato: <login>_1ee_plc_2017_2

(1,5) 1. Defina uma função `sublistas :: [a] -> [[a]]` que retorna todas as sublistas de uma lista dada como argumento.

> sublistas [1,2,3]

[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]

Solucão:

```
sublistas :: [a] -> [[a]]
sublistas []     =  [[]]
sublistas (x:xs) =  [ x:ys | ys <- subs xs ] ++ subs xs
```

(1,5) 2. Implemente a função `filtrarEInserir :: [[Int]] -> Int -> ([[Int]], Int)` que retorna uma tupla. O primeiro elemento da tupla é constituído de listas de inteiros tais que a soma dos números ímpares é maior que a soma dos números pares. O segundo elemento consiste no produto entre o segundo argumento da função `filtrarEInserir` e a multiplicação da maior soma obtida das listas retornadas. Utilize obrigatoriamente `filter`.

> filtrarEInserir [[2,3,4,5,6], [1, 2, 3], [9]] 5
([[1,2,3], [9]], 45)

> filtrarEInserir [[2,3,4,5], []] 7
([[2,3,4,5]],56)

> filtrarEInserir [] 5
([],0)

Solucão:

```
filtrarEInserir [] n = ([], 0)
filtrarEInserir l n = (p1, p2)
  where
    p1 = listasSomaImparMaiorQuePar l
    p2 = (n *  maximoLista(somaListas (listasSomaImparMaiorQuePar l)))

maximoLista [] = minBound::Int
maximoLista [x] = x
maximoLista (x:xs) = max x (maximoLista xs)

somaListas [] = []
somaListas (x:xs) = (foldr (+) 0) x : somaListas xs

listasSomaImparMaiorQuePar :: [[Int]] -> [[Int]]
listasSomaImparMaiorQuePar [] = []
listasSomaImparMaiorQuePar (x:xs) =
   if (somaListaCond x (\e -> mod e 2 /= 0))
        > (somaListaCond x (\e -> mod e 2 == 0))
   then x:listasSomaImparMaiorQuePar xs
   else listasSomaImparMaiorQuePar xs

somaListaCond :: [Int] -> (Int -> Bool) -> Int
somaListaCond l f = foldr (+) 0 (filter f l)
```

(1,0) 3. Defina a função `altMap :: (a -> b) -> (a -> b) -> [a] -> [b]` que, de forma alternada, aplica as duas funções dadas como argumentos aos elementos sucessivos na lista, respeitando a ordem deles.

> altMap (+10) (+100) [0, 1, 2, 3, 4]

[10, 101, 12, 103, 14]

Solucão:

```
altMap :: (a -> b) -> (a -> b) -> [a] -> [b]
altMap f g l = listaFunc f g l 1 (length l)

listaFunc f g [] n tam    = []
listaFunc f g (x:xs) n tam
  | (mod n 2 == 0) && (n <= tam)  = g x: listaFunc f g xs (n+1) tam
  | (mod n 2 /= 0) && (n <= tam)  = f x: listaFunc f g xs (n+1) tam
```

4. Considere uma função polinomial de grau 2 ($f(x) = ax^2 + bx + c$), onde a, b e c são os coeficientes do polinômio.

(1,0) (a) Defina a função poli :: **Integer** -> **Integer** -> **Integer** -> **Integer** -> **Integer** que recebe como argumentos os coeficientes de uma função polinomial de grau 2 e devolve uma função de inteiro para inteiro (um polinômio)

Solucão:

```
poli :: Integer -> Integer -> Integer -> Integer -> Integer
poli m n p = (\x ->  (m * x * x) + (n * x) + p)
```

(1,0) (b) Defina a função listaPoli :: [(**Integer**,**Integer**,**Integer**)] -> [**Integer**->**Integer**] que aguarda uma lista de triplas de inteiros (coeficientes de um polinômio de segundo grau) e devolve uma lista de funções de inteiro para inteiro (polinômios) .

Solucão:

```
listaPoli :: [(Integer,Integer,Integer)] -> [Integer->Integer]
listaPoli l = [poli x y z | (x,y,z) <- l ]
```

(1,5) (c) Defina a função appListaPoli :: [**Integer**->**Integer**] -> [**Integer**] -> [**Integer**] que recebe uma lista de funções de polinômios e uma lista de inteiros. Esta função devolve uma lista de inteiros que resultam da aplicação de cada polinômio da primeira lista aplicada ao inteiro correspondente na segunda lista. Utilize compreensão de listas.

Solucão:

```
appListaPoli lPoli lInt   = [f x | (f,x) <- zip lPoli lInt]
```

5. Um móbile é constituído de pendentes, fios e barras. Em cada uma das extremidades de uma barra de um mobile é preso um fio, no qual pode estar pendurado um pendente ou uma nova barra. Barras e fios são considerados elementos sem peso e pendentes possuem um peso (representado por um valor inteiro). O seguinte tipo de dado pode ser usado para representar um móbile:

data Mobile = Pendente **Int** | Barra Mobile Mobile

O peso de um mobile é igual à soma dos pesos de todos os seus pendentes. Um móbile é balanceado se ele consiste de um único pendente ou se os pesos dos móbiles pendurados nas duas extremidades da sua barra são iguais e esses móbiles são balanceados. Defina as seguintes funções sobre móbiles:

(1,0) (a) `peso :: Mobile -> Int`, que retorna o peso do móbile dado como argumento.

Solucão:

```
peso :: Mobile -> Int
peso (Pendente n)  = n
peso (Barra m1 m2) = peso m1 + peso m2
```

(1,5) (b) `balanceado :: Mobile -> Bool`, que determina se o móbile dado como argumento é ou não balanceado

Solucão:

```
balanceado :: Mobile -> Bool
balanceado (Pendente _)  = True
balanceado (Barra m1 m2) = peso m1 == peso m2 &&
                           balanceado m1 && balanceado m2
```