Cálculo Lambda e Funções Genéricas

Em linguagem Haskell, a expressão `\x` indica uma abstração lambda (usada para definir funções sem nome) sobre a variável x, como no exemplo:

```
\x -> mod n x /= 0
```

Esta expressão define que se o resto da divisão de um número *n* por *x* (variável ligada) for diferente de zero, então o resultado é verdadeiro, caso contrário, falso.

A abstração acima pode ser usada para o teste de primalidade (que decide se um número inteiro é primo):

```
primo n = n > 1 && all (\x -> mod n x /= 0) [2..n-1]
```

Nesta verificação, utilizamos a função `all` que checa se todos os valores de uma lista são conformes à uma determinada propriedade, neste caso, resto da divisão por n ser diferente de zero.

Utilizando a função `primo` definida acima, podemos manipular os números primos contidos num intervalo usando função genérica:

```
> map (primo) [3,4,2,1]
[True,False,True,False]

> filter primo [4,5,3,7,12]
[5,3,7]

> foldr1 (+) (filter (primo) [1..50])
328
```

Uma outra função de redução é a foldr, que recebe uma função, uma variável e uma lista como parâmetros:

```
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Dada esta função, outras três podem ser definidas:

```
sum = foldr (+) 0
product = foldr (*) 1
and = foldr (&&) True
```

O nome `foldr` é devido à ordem de avaliação (à direita - *right* em inglês) dos valores.

```
foldr (+) a [b,c,d,e]
```

Pode ser descrita como: `(b + (c + (d + (e + a))))`

Exercícios:

1. Mostre o resultado obtido pela execução das expressões Haskell (ex. da Lista 3):

```
> (\x -> x + 3) 5
> (\x -> \y -> x * y + 5) 3 4
> map (\x -> x ^3) [2,4,6]
> deriv (\x -> x*x*x) 0.0001 1
         where deriv f dx = \x -> (f(x + dx) - f(x)) / dx
> (\(x,y) -> x * y^2) (3,4)
> (\(x,y,_) -> x * y^2) (3,4,2)
> map (\(x,y,z) -> x + y + z) [(3,4,2),(1,1,2),(0,0,4)]
> filter (\(x,y) -> x `mod` y == 0) [(4,2),(3,5),(6,3)]
> (\xs -> zip xs [1,2,3]) [4,5,6]
> map (\xs -> zip xs [1..]) [[4,6],[5,7]]
> foldr1 (+) [1,2,3]
> foldr1 (\x -> \y -> x + y + 7) [1,2,3,4,5]
```

2) Sejam as definições e funções abaixo para um programa que manipula datas importantes. Dada uma data, no formato (dd,mm,aaaa) queremos saber qual o dia da semana correspondente. Para responder a esta questão, definimos a função `diaSemana` bem como outras funções necessárias:

```
type Dia = (Int,Int,Int)

nomesDias = ["Domingo","Segunda-Feira", "Terca-Feira", "Quarta-
Feira",
             "Quinta-Feira", "Sexta-Feira", "Sabado"]

diaSemana::Dia->[Char]
diaSemana (d,m,a) = nomesDias !! numeroDoDia (d,m,a)

numeroDoDia::Dia->Int
numeroDoDia (d,m,a) = mod ( (a-1)*365
      + div (a-1) 4
      - div (a-1) 100
      + div (a-1) 400
      + sum (take (m-1) (meses a))
      + d
      ) 7

meses::Int->[Int]
meses a = [31, fev, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
       where fev | bissexto a = 29
                 | otherwise = 28

bissexto::Int->Bool
bissexto a = (mod a 4 == 0) &&
             (not (mod a 100 == 0) || (mod a 400 == 0))
```

Utilizando funções genéricas faça:

A) Teste as funções acima e verifique o dia da semana de uma sequência de datas. Para isso utilize a função genérica de mapeamento)e uma lista de entrada, como a lista *datasImportantes:*

```
datasImportantes::[Dia]
datasImportantes = [(01,01,2010),(10,06,2010),(03,12,2010),(05,02,2009),
                    (01,10,2007),(02,12,2005),(14,03,2002),(11,10,2003),
                    (31,07,2000),(22,06,1997),(05,05,1995),(13,11,1990)]
```

B) Dada uma lista de datas, retorne apenas aquelas em que o ano esteja entre 1995 e 2005.

C) A partir de uma lista de datas, retorne apenas aquelas que correspondem à uma segunda-feira.

D) Crie, usando lista por compreensão, todas as datas entre dois anos (exemplo: entre 2000 e 2005). Retorne, em seguida, o número de sextas-feiras 13 existentes no intervalo.

3) Seja a função nub abaixo. Analise a função, faça testes com diferentes entradas e descreva a sua funcionalidade.

```
nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

4) Para calcular as combinações de notas para devolver o troco durante um pagamento, podemos definir a função:

```
notas:: [Int]
notas = [1,2,5,10,20,50,100]

notasTroco :: Int -> [[Int]]
notasTroco 0 = [[]]
notasTroco valor = [v:vs | v <- notas, valor >= v,
                           vs <- notasTroco (valor-v) ]

> notasTroco 4
[[1,1,1,1],[1,1,2],[1,2,1],[2,1,1],[2,2]]
```

Como podemos observar, a função `notasTroco` retorna várias listas iguais. Faça funções para eliminar as listas repetidas.

Bom trabalho!!