Departamento de Computação
Universidade Federal de Ouro Preto
Programação Funcional
Prof. Lucília Figueiredo

Lista de Exercícios 01 -Recursão, Listas

1. Quais so os tipos das seguintes funções:

```
second xs = head )(tail xs)

swap (x,y) = (y,x)

pair x y = (x,y)

palindrome xs = reversae xs == xs

twice f x = f (f x)
```

2. Defina cada uma das funções a seguir. Procure usar funções pré-definidas em Haskell:
   (a) `prodMn :: Int -> Int -> Int` que, dados dois valores inteiros m e n, retornar o produto de todos os valores inteiros entre m e n (inclusive).
   (b) `sumQuad :: Int -> Int` que, dado um valor inteiro positivo n, retorna a soma dos quadrados de todos os inteiros compreendidos entre 0 e n.
   (c) `sumFats :: Int -> Int` que, dado um valor inteiro positivo n, retornar a soma dos fatoriais de todos os inteiros entre 0 e n.
   (d) `duplicate :: Int -> String -> String` que, dado um inteiro n e um string s retorna um novo string igual ao string dado duplicado n vezes. Se n = 0, retorna o string vazio.
   (e) `makespaces :: Int -> String` que, dado um número inteiro n, retorna um string de n espaços em branco. (Use a função definida no item anterior).
   (f) `rjustify :: Int -> String -> String` que, dado o comprimento de uma linha e um string, alinha esse string à direita na linha, inserindo o número de espaços em branco necessários à sua esquerda. Caso o comprimento do string seja maior que o comprimento da linha, ele deve ser quebrado em mais linhas.

3. A função `length`, que computa o número de elementos de uma lista, pode ser definida do seguinte modo:

```
length xs = length' 0 xs
   where length' n []     = n
         length' n (x:xs) = length' (n+1) xs
```

   Essa função usa a função auxiliar `length'`, que possui um parâmetro adicional para acumular o resultado. A função `length'` é definida usando *recursão de cauda*, uma vez que a chamada recursiva `length' (n+1) xs`, usada no lado direito da definição, não ocorre dentro de nenhum argumento de outra função. Use essa técnica de *recursão de cauda* para definir as seguintes funções:
   (a) `fac :: Int -> Int`, que computa o fatorial de um número natural
   (b) `reverse :: [a] -> [a]`, que inverte uma lista

Existem alguma vantagem em se usar recursão de cauda nas definições acima?
Existe alguma diferença se é usada a estratégia de avaliação *lazy* ou *strict*?

4. Considere as seguintes definições das funções `take` e `from`:

```
take ::  Int -> [a] -> [a]
take 0     xs     = []
take (n+1) (x:xs) = x:take n xs

from ::  Int -> [Int]
from n = n:from (n+1)
```

   Mostre os passos de redução da avaliação *lazy* da expressão `take 3 (from 8)`.

5. Considere a seguinte definição de função em Haskell:

```
until p f x = if p x then x else until p f (f x)
```

   (a) Qual é o tipo da função `until`?

   (b) Qual é o resultado da avaliação da expressão `until (<10) square 2`?

   (c) Use a função `until` para definir uma função que, dado um string `s`, retorne o string obtido removendo-se todos os caracteres iguais a branco que ocorrem no início de `s`.

6. A lista de números primo pode ser gerada, de modo eficiente, por meio do algoritmo denominado *crivo de Erastotenes* (proposto pelo matemático grego Erastotenes).

   (a) Defina a função `crivo :: [Int] -> [Int]` que, dada uma lista de inteiros, retorna uma nova lista tal que nenhum dos elementos dessa lista é um divisor de qualquer dos elementos posteriores a ele na lista. Para dedefinir a função `crivo`, podemos pensar recursivamente, do seguinte modo: a) se a lista passada como argumento é vazia, ent ao a solução é trivial; b) se a lista é não vazia, da forma (p:ps), então a solução é a lista (p:l), em que l é obtida aplicando crivo, recursivamente, a uma lista que não contêm nenhum múltiplo de p.

   (b) Usando a função `crivo`, definida no item anterior, defina a função `primos :: Int -> [Int]` que, dado um inteiro n, retorna a lista dos números primos de 1 a n.

7. Defina uma função `positions :: a -> [a] -> [Int]` tal que, dado um valor `y` e uma lista de valores `xs`, retorna a lista de todas as posições em que `x` ocorre na lista `xs`. (*Dica*: Use a função `zip` e *list compreension*).

8. Uma lista `xs` é um *prefixo* de uma lista `ys` se `ys=xs++zs` para alguma lista `zs`. Por exemplo "ban" é um prefixo de "banana". Defina uma função

```
prefix ::  Eq a => [a] -> [a] -> Bool
```

   tal que `prefix xs ys` retorna `True` se `xs` é um prefixo de `ys` e retorna `False` em caso contrário.

9. Uma lista `xs` é uma *subseqüência* de uma lista `ys` se `ys=as++xs++zs` para alguma lista `zs` e alguma lista `as`. Por exemplo "ana" é uma subseqüência de "banana". Defina uma função

```
subsequence ::  Eq a => [a] -> [a] -> Bool
```

   tal que `subsequence xs ys` retorna `True` se `xs` é uma subseqüência de `ys` e retorna `False` em caso contrário.

10. O programa `grep` do UNIX pesquisa pelas ocorrências de um dado string em um arquivo, e imprime toda linha que contém tal ocorrência. Defina uma função similar:

```
grep ::  String -> String -> [String]
```

cujos argumentos são um string a ser procurado e o conteúdo de um arquivo no qual deve ser feita a pesquisa, e cujo resultado é igual á saída do comando `grep` do UNIX. Por exemplo:

```
grep "ana" "Suzana é filha de José.\n
            José é irmão de Pedro,\n
            que é pai de Ana"
  ▷ ["Suzana é filha de José.","que é pai de Ana"]
```

11. Uma *rotação* de uma lista é obtida removendo-se qualquer número de elementos do início da lista e inserindo esses elementos no fim da lista. (*Dica:* Use *list comprehension*). Por exemplo, as rotações de `[1,2,3]` são `[1,2,3]`, `[2,3,1]` e `[3,1,2]`. Defina uma função:

```
rotations ::  [a] -> [[a]]
```

que retorna a lista de todas as possíveis rotações da lista passada como argumento. Por exemplo,

```
rotations [1,2,3] ▷ [[1,2,3],[2,3,1],[3,1,2]]
```

12. *Problema das n-rainhas:* Esse problema consiste em determinar as possíveis posições em que podem ser colocadas n rainhas em um tabuleiro de xadrez (n x n), de maneira que nenhuma rainha seja atacada pelas demais (uma rainha ataca outra se as duas estiverem na mesma linha, ou na mesma coluna, ou na mesma diagonal). Considere que uma posição no tabuleiro de xadrez é representada por um par de valores inteiros que indicam a linha e a coluna no tabuleiro. Uma solução para o problema pode ser então representada por uma lista contendo as posições das n rainhas. As possíveis soluções para o problema consistem, portanto, de uma lista de listas de n posições:

```
type Pos = (Int,Int)
rainhas ::  Int -> [[Pos]]
```

Para obter uma solução para esse problema, podemos pensar recursivamente, do seguinte modo:

- Se o número de rainhas a serem colocadas no tabuleiro é igual a `0`, ou igual a `1`, então o problema pode ser resolvido trivialmente.
- A solução do problema de colocar `n` rainhas pode ser obtida, a partir da solução para o problema de colocar `(n-1)` rainhas, acrescentando a essa solução as posições em que podem ser colocadas a última rainha, de maneira que ela não ataque nenhuma das outras já colocadas no tabuleiro. Note que a i-ésima rainha deve ser colocada na i-ésima linha do tabuleiro, para `i = 1..n`|.

Para obter um programa que determina a solução do problema, siga os passos sugeridos a seguir.

(a) Defina a função `ataca :: Pos -> Pos -> Bool` que determina se as rainhas colocadas nas posições passadas como argumentos atacam uma à outra.

(b) Defina a função `posOK :: [Pos] -> Pos -> Bool` que retorna True se a rainha colocada na posição passada como argumento não ataca nenhuma das demais rainhas colocadas anteriormente, nas posições contidas na lista passada como argumento; e retorna False em caso contrário.

(c) Defina a função `rainhas :: Int -> [[Pos]]` que, dado o número de rainhas a serem colocadas no tabuleiro, retorna a lista das possíveis soluções para o problema.

13. O método de Newton-Raphson para cálculo da raiz quadrada de um número `x` pode ser descrito da seguinte maneira. Começando com uma aproximação inicial a (por exemplo, `x/2`), obtemos aproximações consecutivas, cada vez melhores, em cada iteração n, usando a regra

$$a(n+1) = (a(n) + x/a(n))/2$$

Se essas aproximações convergem para um limite `a`, então

$$a = (a + x/a)/2e$$

Portanto:

$$2a = a + x/aa = x/aa * a = xa = squareRoot(x)$$

De fato, as aproximações convergem rapidamente para um limite. Programas usuais para cálculo de raiz quadrada têm como argumento um fator de precisão `fp`, e terminam a seqüência de aproximações quando a diferença entre duas delas é menor que `fp`. Como o algoritmo de Newton-Raphson computa uma seqüência de aproximações, é natural representar isso em um programa por meio de uma lista de aproximações. Para construir o programa para cálculo da raiz quadrada de um número, baseado nesse método, siga os seguintes passos:

(a) Defina a função `next :: Float -> Float -> Float` que, dada um valor `x` e uma aproximação `a`, calcula a próxima aproximação para a raiz quadrada de `x`, de acordo com a regra acima.

(b) Defina a funç ao `repeat :: (Float -> Float) -> Float -> [Float]` que, dada uma função `f` e um valor inicial `a0`, constrói a lista `[a0, f a0, f(f a0), f(f(f a0)), ...]` . Note que a seqüência de aproximações para a raiz de `x` pode ser, portanto, obtida pela expressão a seguir, onde `a0` é uma primeira aproximação para a raiz de `x`: `repeat (next x) a0`

(c) Para obter a raiz quadrada com uma aproximação menor ou igual a um fator de precisão `fp` (próximo de 0), devemos construir uma função para testar quando dois valores consecutivos da lista diferem de um valor manor que `fp`. Defina a função `within :: Float -> [Float] -> Float` que, dado um fator de precisão `fp` e uma lista de valores `xs`, retorna o primeiro elemento da lista tal que a diferença entre ele e o anterior é menor ou igual a `fp`.

(d) Colocando tudo isso junto, podemos então definir a função para cálculo da raiz quadrada como:

```
sqrt1 a0 fp x = within fp (repeat (next x) a0)
```

(e) Note que, nesse programa, a parte que computa a seqüência de aproximações `repeat (next x)` é independente da parte que verifica se foi obtida a precisão desejada. Suponha que desejamos modificar o programa acima, de maneira que a retornar um valor quando a razão entre as aproximações sucessivas se aproxima de um determinado fator de precisão `fp` (nesse caso, próximo de 1). Construa uma nova versão do seu programa (`sqrt2`) que opera dessa forma.