Princípios de Programação

Exercícios

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Engenharia Informática

2019/2020

Recursão

Tópicos endereçados neste capítulo: Definição de funções por recursão.

1. Defina as seguintes funções:

   (a) `sum':: Num a =>[a] -> a`, que devolve a soma dos elementos de uma lista.

   (b) `replicate':: Int -> a -> [a]`, que produz uma lista com `n` elementos idênticos. Se `n` for negativo, produz a lista vazia.

   (c) `maximo :: Ord a =>[a] -> a`, que devolve o maior elemento de uma lista não vazia.

   (d) `elem' :: Eq a =>a -> [a] -> Bool`, que decide se um dado elemento existe numa dada lista.

   (e) `substitui :: Eq a =>a -> a -> [a] -> [a]`, que substitui o primeiro elemento pelo segundo elemento na lista argumento. Por exemplo:

   ```
   ghci> substitui 'a' 'o' "as balas"
   "os bolos"
   ```

   (f) `altera:: Ord a =>[a]-> a -> a -> [a]`, que substitui todos os elementos da lista argumento que sejam menores que o segundo argumento, pelo terceiro argumento. Por exemplo,

   ```
   ghci> altera [10,0,23,4,14,2,11] 10 5
   [10,5,23,5,14,5,11]
   ```

(g) `multiplos :: [Int] -> Int -> [Int]` que devolve uma lista contendo os elementos de uma dada lista que são múltiplos de um também dado número inteiro. Exemplo:

```
ghci> multiplos [1,3,6,2,5,15,3,5,7,18] 3
[3,6,15,3,18]
```

(h) `zip'::` `[a] -> [b] -> [(a,b)]`, que produz uma lista de pares a partir de duas listas. A lista resultante tem tantos pares quantos o número de elementos da lista argumento mais curta.

(i) `potencias :: Integer -> [Integer] -> [Integer]`, que devolve uma lista com potências cuja base é o número dado no primeiro argumento e cujos expoentes são dados pelos valores do segundo argumento. Exemplo:

```
ghci> potencias 3 [1..10]
[3,9,27,81,243,729,2187,6561,19683,59049]
```

(j) `posicoes :: [Int] -> Int -> [Int]` que devolve uma lista contendo as posições dos elementos da lista dada como primeiro argumento que são múltiplos do segundo argumento. Exemplo:

```
ghci> posicoes [1,3,6,2,5,15,3,5,7,18] 3
[1,2,5,6,9]
```

(k) `frase :: Int -> [(Int,String)]-> String` que devolve a *string* resultante de concatenar (mantendo a ordem) as *strings* dos pares contidos na lista dada como segundo argumento, que são iguais ao valor do primeiro argumento. Exemplo:

```
ghci> frase 3 [(3,"As "),(1,"Sete ")
  (3,"armas "), (5,"Amor "),
  (3,"e os "), (1,"anos "), (3,"baroes ")]
"As armas e os baroes "
```

(l) `trocaPares :: [a] -> [a]`, que troca cada elemento de uma lista com o elemento seguinte, repetindo o processo de par em par de elementos. Se a lista contiver um número ímpar de elementos, o último elemento não é modificado. Exemplo:

```
ghci> trocaPares [1 .. 5]
[2,1,4,3,5]
```

(m) `fusao :: (Ord a, Num b)=>[(a,b)] -> [(a,b)] -> [(a,b)]`, que dadas duas *listas associação*, devolve uma outra lista associação, obtida por junção das duas listas. Uma lista associação é uma lista de pares chave-valor que não contém dois pares com a mesma chave. Neste exercício estamos apenas interessados em listas ordenadas, por ordem crescente dos valores das chave. No caso da função `fusao`, a lista resultante deve conter tantos pares quantas as chaves distintas existentes nas duas listas argumento e o

valor associado a cada chave será a soma dos valores correspondentes nas duas listas (se cada uma das listas contiver um par com essa mesma chave) ou o valor associado à ocorrência dessa chave no caso de ocorrer somente numa das listas. Exemplo:

```
ghci> fusao [('b',8),('g',2),('m',6),('v',4)]
    [('a',3),('g',5),('m',2)]
[('a',3),('b',8),('g',7),('m',8),('v',4)]
```

2. Um número inteiro positivo d pode ser convertido para representação binária através do seguinte algoritmo:

   i) se $d < 2$, a sua representação binária é o próprio d;

   ii) caso contrário, divide-se d por 2. O resto (0 ou 1) dá-nos o último dígito (o mais à direita) da representação binária;

   iii) os dígitos precedentes da representação binária são dados pela representação binária do quociente de d por 2.

   Escreva uma função que dado num inteiro devolve a sua representação binária. Por exemplo,

   ```
   ghci> repBinaria 23
   10111
   ```

3. Escreva uma função `odioso :: Int -> Bool` que decide se um dado número é um número odioso. Um número odioso é um número não negativo que tem um número ímpar de uns na sua expansão binária. Os primeiros números odiosos são 1, 2, 4, 7, 8, 11.

4. Escreva uma função que recebe dois inteiros i e j e que devolve a representação de i na base j, com $2 \leq j \leq 36$. Esta é uma generalização da função do exercício da representação binária. Para $j > 9$ utilize letras maiúsculas para representar os respectivos dígitos dessas bases ($A = 10$, $B = 11, \ldots$).

5. Programe o seguinte algoritmo de ordenação por inserção em dois passos.

   a) Defina uma função `insert :: Ord a =>a-> [a] -> [a]` que insere um elemento na posição correcta dentro de uma lista *ordenada*. Por exemplo:

   ```
   ghci> insert 3 [1,2,4,5]
   [1,2,3,4,5]
   ```

   b) Defina uma função `insertSort :: Ord a =>[a] -> [a]` que implementa o algoritmo de ordenação por inserção, definido pelas duas regras: (i) a lista vazia está ordenada; (ii) uma lista não vazia pode ser ordenada ordenando a cauda e inserindo a cabeça no resultado.

6. Programe o seguinte algoritmo de ordenação por fusão em dois passos.

   a) Defina uma função recursiva
   `merge :: Ord a =>[a] -> [a] -> [a]` que funde duas listas *ordenadas*, produzindo uma lista *ordenada*. Por exemplo:

   ```
   ghci> merge [1,3,5] [2,4]
   [1,2,3,4,5]
   ```

   b) Defina uma função recursiva
   `mergeSort :: Ord a =>[a] -> [a]` que implementa o algoritmo de ordenação por fusão, definido pelas duas regras: (i) listas de comprimento ≤ 1 estão ordenadas; (ii) as outras listas podem ser ordenadas ordenando as suas duas metades e fundindo os resultados.

7. Teste os três algoritmos (incluindo o `quicksort` do livro de texto).

   Para isso temos de gerar uma lista grande meio desordenada. A função `randomList` prepara uma lista de comprimento arbitrário.

   ```
   randomList :: Int -> [Int]
   randomList n = take n randomInfiniteList

   randomInfiniteList :: [Int]
   randomInfiniteList = iterate f 1234
        where
           f x = (1343 * x + 997) `mod` 1001
   ```

   Agora podemos fazer os nossos testes.

   ```
   ghci> :set +s
   ghci> let xs = randomList 100000
   ghci> isort xs
   ghci> mergeSort xs
   ghci> qsort xs
   ```

   De notar que este exercício apenas dá uma ideia da complexidade em termos de espaço e de tempo dos três algoritmos.