Haskell – Lista 2

Aluno: Mílton Cézar Correia Segundo
Matrícula: 1075128
Disciplina: Programação Funcional
Professor: Francisco Vieira

Nota: Embora, as questões da lista de exercícios estejam descontínuadas(i. e. Da questão um se vá direto à questão 3), a ordem aqui colocada corresponde a ordem sequencial.

1. A implementação deste item foi realizada utilizando uma função f2 auxiliar, que retorna a lista de elementos a ser concatenada baseada no vetor de posições, e a função utilitária *map*. A função *map* juntamente com a função implementada *dec* foi usada por conveniencia, para realizar a indexação dos elementos a partir de um, embora por default o operador **!!** use indexação baseada em zero.

```
dec :: Int -> Int
dec x = x-1

f2 :: [Int] -> [t] -> [t]
f2 [] (y:ys) = []
f2 (x:xs) (y:ys) = ((y:ys)!!x):(f2 xs (y:ys))

f :: [Int] -> [t] -> [t]
f [] [] = []
f (x:xs) [] = []
f [] (y:ys) = (y:ys)
f (x:xs) (y:ys) = (y:ys)++(f2 (map (dec) (x:xs)) (y:ys))
```

Obs: O exemplo do enunciado está errado. Considerando-se que a função indexa os elementos da lista a partir do elemento 1 temos:
f [2,1,4] ["a", "b", "c", "d"] = ["a", "b", "c", "d", "b", "a", "d"]

Pois os elementos [2, 1, 4] correspondem a “b”, “a”, e “d” da lista da esquerda, respectivamente. Ao contrário do que se apresenta no enunciado:
f [2,1,4] ["a", "b", "c", "d"] = ["a", "b", "c", "d", "d", "a", "b"]

2. Na implementação deste exercício foi utilizada a biblioteca **Data.Matrix** que pode ser instalada no linux com o utilitário cabal(*apt-get install cabal-install*) com o comando *cabal install matrix*. Segue o código:

```
import Data.Matrix

my_sum ::  (Num t) => [t] -> Int -> t
my_sum [] _ = 0
my_sum (x:xs) 1 = x + (my_sum xs (-1))
my_sum (x:xs) (-1) = (-1)*x + (my_sum xs 1)

my_minor :: (Num t) => Matrix t -> Int -> Matrix t
my_minor x a = minorMatrix 1 a x

my_det :: (Num t) => Matrix t -> t
my_det x | (nrows x) == 2 = ((getElem 1 1 x)*(getElem 2 2 x) - (getElem 1 2 x)*(getElem 2 1
x))
```

```
              | otherwise = (my_sum (map (my_det) (map (my_minor x) [1 .. (ncols x)])) 1)

mat :: Matrix Int
mat = fromList 3 3 [1..]
```

Com a declaração de mat é possível realizar o teste da função *my_det* que recebe como argumento uma matriz de números e retorna o determinante dessa matriz, que no caso é zero. Foi utilizada a ideia recursiva de que o determinante de uma matriz é igual ao somatório do determinante multiplicado por um fator, que se alterna entre -1 e 1(função *my_sum*) das submatrizes obtidas escolhendo-se uma linha e removendo, isoladamente, a coluna e o elemento de cada elemento desta linha até a coluna *ncols* x , onde x é uma matrix passada como parâmetro.

3. As funções **valid** e **validAux** são usadas com propósito de filtrar o espaço de configurações do tabuleiro de modo que as rainhas não conflitem em linha ou diagonal. A verificação por coluna não é necessária visto que é considerado que as rainhas são colocadas por padrão em colunas distintas, sendo o retorno de chess8 uma lista com as posições das linhas das rainhas em ordem de colunas. A função **chess8Aux** retorna um conjunto de soluções possíveis quando passada com o parâmetro 8, e a função **chess** retorna a solução de posição 1 na lista de listas retornada por **chess8Aux**.

```
validAux :: Int -> Int -> [Int] -> Bool
validAux _ _ [] = True
validAux p c (x:xs) | p == x = False --rainhas na mesma linha
                    | ((x-p) == c) || ((x-p) == (-c)) = False -- rainhas na mesma diagonal
                    | otherwise = validAux p (c+1) xs

valid :: [Int] -> Bool
valid [] = True
valid (x:xs) = (validAux x 1 xs) && (valid xs)


--chess8Aux n
--resolve problema das 8 rainhas. O parametro n eh o numero de colunas restantes
chess8Aux :: Int -> [[Int]]
chess8Aux 0 = [[]]
chess8Aux 1 = [[1]]
chess8Aux n = filter valid [x:y | y <- (chess8Aux (n-1)), x <- [1 .. 8]]

chess8 :: [Int]
chess8 = (chess8Aux 8)!!1
```

4. Neste apartado, utiliza-se uma função auxiliar com dois parâmetros auxiliares do tipo *Bool* e [[t]]. O primeiro serve para saber a qual lista o elemento pertence. Se o parâmetro Bool é verdadeiro então o elemento pertence à primeira lista e se é falso, assume-se que é da segunda lista.

```
f2 :: [t] -> Bool -> [[t]] -> [[t]]
f2 [] _ x = x
f2 (x:xs) True [a, b] = f2 xs False [a++[x], b]
f2 (x:xs) False [a, b] = f2 xs True [a, b++[x]]

f :: [t] -> [[t]]
f x = f2 x True [[], []]
```

5.

a) Para converter a lista de inteiros em string utilizamos as funções auxiliares toAst e toSpace, que repetem astericos e espaços uma quantidade n de vezes.

```
toAst :: Int -> String
toAst 0 = ""
toAst n = "*"++(toAst (n-1))

toSpace :: Int -> String
toSpace 0 = ""
toSpace n = " "++(toSpace (n-1))

toString :: [Int] -> String
toString [] = ""
toString (x:[]) = toAst x
toString (x:y:ys) = (toAst x) ++ (toSpace y) ++ (toString ys)
```

b) Agrupa-se cada três digitos em uma lista

```
type Linha = String
toLinhas :: String -> [Linha]
toLinhas [] = []
toLinhas (x:[]) = [[x]]
toLinhas (x:y:z:zs) = [x,y,z]:(toLinhas zs)
```

c) Para cada elemento da lista se insere um '\n'.

```
showLinhas :: [Linha] -> String
showLinhas [] = ""
showLinhas (x:xs) = x++"\n"++(showLinhas xs)
```

d)

```
juntaLinhas :: [Linha] -> [Linha] -> [Linha]
juntaLinhas [] [] = []
juntaLinhas (x:xs) (y:ys) = (x++" "++y):(juntaLinhas xs ys)
```

e) Para esta implementação foi utilizada uma função **ndig** como função de apoio para definir as ações específicas para números de 1, 2 e 3 digítos. Para números de 1 dígito basta utilizar a recursão **showLinhas → toLinhas → toString**. Para números de mais de um dígito devemos concatenar o resultado de toLinhas de cada dígito, de tal forma que a possamos posicionar o \n ao final de cada linha **i** corretamente. Para tal utilizamos a função que contruímos juntaLinhas.

```
ndig :: Int -> Int
ndig n | n < 10 = 1
               | n < 100 = 2
               | otherwise = 3

numeros :: [[Int]]
numeros = [zero,um,dois,tres,quatro,cinco,seis,sete,oito,nove]

tolcd :: Int -> String
tolcd n | (ndig n) == 1 =  showLinhas (toLinhas (toString (numeros!!n)))
        | (ndig n) == 2
```

```
        =  showLinhas
        (juntaLinhas
              (toLinhas (toString (numeros!!((mod n 100) `div` 10))))
              (toLinhas (toString (numeros!!(mod n 10))))
        )
        | (ndig n) == 3
        = showLinhas
        (juntaLinhas
          (juntaLinhas
            (toLinhas
              (toString (numeros!!((mod n 1000) `div` 100))))
            (toLinhas
              (toString (numeros!!((mod n 100) `div` 10)))) )
          (toLinhas (toString (numeros!!((mod n 10)))))
        )
```

f)

```
type Estado = Bool
tcomp :: String -> Estado -> Int -> [Int]
tcomp [] False n | (n > 0) = [n]
                 | otherwise = []
tcomp [] True n | (n > 0) = [n]
                | otherwise = []
tcomp (x:xs) True n | x == '*' = tcomp (xs) True (n+1)
                    | otherwise = n:(tcomp (x:xs) False 0)
tcomp (x:xs) False n | x == ' ' = tcomp (xs) False (n+1)
                     | otherwise = n:(tcomp (x:xs) True 0)

toCompact :: String -> [Int]
toCompact s = tcomp s True 0
```

A implementação da função toCompact é passada à função auxiliar tcomp que acrescenta dois parâmetros. Um deles é uma flag *Boolean*. O valor *True* indica que a função está contabilizando asteriscos e o valor falso indica que a função está contabilizando espaços em brancos. O outro parâmetro é um inteiro n, que indica a quantidade do caractér atual sendo lida

6. Para retornar todas as permutações de uma lista de elementos, tal que essas permutações tenham um tamanho maior ou igual que um número n é necessário percorrer todo o espaço de busca. Recursivamente, podemos definir todas essas permutações em uma lista haskell do tipo (x:xs), como o conjunto de todas as permutações de tamanho maiores ou iguais que n que não contém a cabeça x unido ao conjunto da concatenação da cabeça x a todas as permutações possíveis de tamanho n-1 na lista xs. A função filter garante a propriedade “>=n”, enquanto a condição (length (x:xs)) >= n foi utilizada para fins de desempenho, podando onde as cadeias não podem alcançar a longitude mínima desejada.

```
pred_length :: Int -> [t] -> Bool
pred_length n x = (length x) >= n

list_perms :: [t] -> Int -> [[t]]
list_perms [] _ = [[]]
list_perms (x:xs) n | (length (x:xs)) >= n
         = filter (pred_length n) ([x:y | y <- (list_perms xs (n-1))] ++ (list_perms xs n) )
         | otherwise = [[]]
```

7. Para implementar o reconhecimento de strings, podemos simular um autômato que reconheça a substring1 na substring3 fazendo com que os estados do autômato correspondam aos caracteres da substring1, realizando a transição dos caracteres um a um. Mantemos a string a ser reconhecida em memória como o primeiro parâmetro da função *occur* e a cada passo fazemos uma chamada recursiva à mesma, que equivale a um movimento na fita. A primeira posição do cabeçote é mantida no terceiro parâmetro, juntamente com o restante a ser reconhecido. Quando não nos restam mais caracteres a serem reconhecidos(O terceiro parâmetro é a lista vazia), temos a certeza de que reconhecemos todos os caracteres e a substring1 pertence a substring3. Após o reconhecimento, podemos realizar a substituição com segurança. Em primeiro lugar separamos a string nas duas partes à esquerda e à direita da substring1, utilizando a função *sep*. Por último, a função *substr* realiza o resto do trabalho concatenando a substring2 no meio das duas substrings resultantes de *sep*.

```
occur :: String -> String -> String -> Bool
occur [] _  _ = True
occur x _ [] = True
occur _ [] _ = False
occur x (y:ys) (a:as) | y == a  = occur x ys as
                      | otherwise = occur x (ys) x

getT :: Int -> (String, String) -> String
getT 0 (a,b) = a
getT 1 (a,b) = b

-- sep x y z b c
-- retorna uma tupla contendo as string anteriores e posteriores a string x na cadeia y
-- x: A string a ser encontrada
-- y: String onde deveremos buscar a String x
-- z: Restante dos caracteres de x a serem reconhecidos
-- b: Parte da String que coincidiu com o inicio de x, porem ainda esta em reconhecimento
-- c: Parte da String que foi lida mas que nao foi reconhecida
sep :: String -> String -> String -> String -> (String, String)
sep [] y _ _ _= (y, [])
sep x y [] b c = (c, y)
sep _ [] _ b c = ((c++b), [])
sep x (y:ys) (a:as)  b   c  | y == a  = sep x ys as (b++[a]) c
                            | otherwise = sep x ys x [] (c++b++[y])



substr :: String -> String -> String -> String
substr x y z | (occur x z x) =  (getT 0 (sep x z x [] [])) ++ y ++ (getT 1 (sep x z x [] [
[]))
                  | otherwise = z
```

8. Onde está o banco de dados ao qual a questão se refere?

9. Todas as permutações possíveis da lista **(x:xs)** são todas as permutações de **(xs)** e todas as concatenações de x com cada uma das cadeias do conjunto de permutações de **xs**.

```
perms :: [t] -> [[t]]
perms [] = [[]]
```

```
perms (x:xs) = [x:y | y <- (perms xs)] ++ (perms xs)
```