Programação Funcional
Lista de Exercícios 04

Prof. Wladimir Araújo Tavares

Entrada e Saída

1. Escreva um programa em Haskell que solicita ao usuário para digitar uma frase, lê a frase (uma linha) da entrada padrão e testa se a string lida é uma palíndrome, exibindo uma mensagem apropriada.
2. Faça um programa que leia um número n e imprime n!.
3. Faça um programa que leia um número n e imprime "sim"se o número é primo, caso contrário, imprime "nao".
4. Escreva um programa que solicita ao usuário três números em ponto flutuante, lê os números, e calcula e exibe o produto dos números.
5. Escreva um programa em Haskell que solicita ao usuário uma temperatura na escala Fahrenheit, lê esta temperatura, converte-a para a escala Celsius, e exibe o resultado. Para fazer a conversão, defina uma função celsius :: Double -¿ Double que recebe a temperatura na escala Fahrenheit e resulta na temperatura correspondente na escala Celsius. Use a seguinte equação para a conversão:

$$C = \frac{5}{9}(F - 32) \tag{1}$$

   onde F é a temperatura na escala Fahrenheit e C é a temperatura na escala Celsius. Use a função celsius na definição de main. A digitação da temperatura em Fahrenheit deve ser feita na mesma linha onde é exibida a mensagem que a solicita.
6. A prefeitura de Quixadá abriu uma linha de crédito para os funcionários estatutários. O valor máximo da prestação não poderá ultrapassar 30% do salário bruto. Fazer um programa que permita entrar com o salário bruto e o valor da prestação, e informar se o empréstimo pode ou não ser concedido.
7. Considere o seguinte programa:

```
module Main where
main
        = do {
        tests <- getLine;
        contents <- getContents;
        putStrLn $ show $take (read tests) (lines contents)
        }
```

   Modifique o programa para que ele leia um número natural n, e então leia outros n números e calcule e exiba a soma destes números.
8. Escreva um programa completo que reproduza a funcionalidade do utilitário wc de Unix: ler um arquivo de texto da entrada-padrão e imprimir o número de linhas, número de palavras.

```
module Main where

main = do       contents <- getContents;
```

   Use a função lines, words:

   - lines quebra uma string em uma lista de strings usando o caractere new line como separador.
   - words quebra uma string em uma lista de palavras usando o caractere espaço como separador

   Exemplo:

```
aslfjasfl
salfsa
asflsafsa aslfsaf
alsfhsa aslfjas aslfjaflk
```

   Saída:

```
linhas: 4
palavras: 7
```

9. A função `interact :: (String -> String) -> IO ()` é muito utilizada para construir programas com entrada e saída simples. Considere o seguinte programa:

```
module Main where
main = do       interact (show.length.lines)
```

   O programa acima imprime o número de linhas do arquivo de entrada.

   Faça um programa completo que lê linhas de texto da entrada-padrão e imprime cada linha invertida usando a função interact.

   Dica: Use as funções lines, unlines, map reverse.
10. Faça um programa que lê uma linha de um arquivo de texto da entrada-padrão e diga se a palavra é palindrome ou não usando a função interact.

    Dica: Use a seguinte função

```
respondPalindromes :: String -> String
respondPalindromes =
    unlines . map (\xs -> if isPalindrome xs
                          then "palindrome"
                          else "nao palindrome") . lines
```

11. Considere o seguinte programa:

```
import Control.Monad
main = do
        n <- getLine
        v <- forM [1..read n] (\a -> do
                valor <- getLine
                return valor)
        putStrLn (show v)
```

    Entrada:

```
4
1
2
3
4
```

    Saída:

```
["1","2","3","4"]
```

    Modifique o programa para imprimir o somatório dos valores digitados.

Tautologia

```
import Data.List

data Prop =
    Const Bool
    | Var Char
    | Neg Prop
    | Conj Prop Prop
    | Disj Prop Prop
    | Impl Prop Prop


type Interpretacao = [(Char, Bool)]

valor :: Interpretacao -> Prop -> Bool
valor _ (Const b) = b
valor i (Var x) = busca x i
valor i (Neg p) = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Disj p q) = valor i p || valor i q
valor i (Impl p q) = valor i p <= valor i q

primeiro (a,b) = a
segundo  (a,b) = b

busca :: Eq c => c -> [(c,v)] -> v
busca c [] = error "Interpretacao insuficiente"
busca c (x:xs) = if c == (primeiro x) then segundo x
                 else busca c xs

variaveis :: Prop -> [Char]
variaveis (Const x) = []
variaveis (Var x)   = [x]
variaveis (Neg p)   = variaveis p
variaveis (Conj p q) = nub ( variaveis p ++ variaveis q )
variaveis (Disj p q) = nub ( variaveis p ++ variaveis q )
variaveis (Impl p q) = nub ( variaveis p ++ variaveis q )

bits :: Int -> [[Bool]]
bits 0 = [ [] ]
bits n = [ x:xs | x <- [True, False], xs <- bits (n-1)]

interpretacoes :: Prop -> [Interpretacao]
interpretacoes p =  [ zip var b |  b <- bits (length var) ]
    where var = (variaveis p)

tautologia :: Prop -> Bool
tautologia p = and [ valor i p | i <- interpretacoes p]
```

12. Escreva uma definição recursiva da função `dual :: Prop -> Prop` que obtém o dual duma proposição, i.e. a proposição que resulta de substituir todas as conjunções por disjunções (e vice-versa) e as constantes True por False (e viceversa); as variáveis e negações são inalteradas. Exemplos:

```
dual (Neg (Var 'a'))
= Neg (Var 'a')
dual (Disj (Var 'x') (Neg (Var 'x')))
= Conj (Var 'x') (Neg (Var 'x'))
dual (Conj (Var 'a') (Disj (Var 'b') (Const False)))
= Disj (Var 'a') (Conj (Var 'b') (Const True))
```

13. Escreva uma função `contar :: Prop -> [(Char, Int)]` cujo resultado é uma lista de associações entre cada variável e o número de vezes que ocorre na proposição.

    Exemplo: `contar (Conj (Var 'a') (Disj (Var 'b') (Var 'a')) = [('a', 2), ('b', 1)]`

    (a ordem dos pares no resultado não é importante).

14. Escreva uma definição duma função `showProp :: Prop -> String` para converter uma proposição em texto;

    Alguns exemplos:

```
> showProp (Neg (Var 'a'))
"(~a)"
> showProp (Disj (Var 'a') (Conj (Var 'a') (Var 'b')))
"(a || (a && b))"
> showProp (Impl (Var 'a') (Impl (Neg (Var 'a')) (Const False))
"(a -> ((~a) -> F))"
```

15. Escreva uma definição da função `satisfaz :: Prop -> Bool` que verifica se uma proposição é satisfazível, isto é, se existe uma atribuição de valores às variáveis que a torna verdadeira.
16. Escreva uma definição da função `equiv :: Prop -> Prop -> Bool` que verifica se duas proposições são equivalentes, isto é, tomam o mesmo valor de verdade para todas as atribuições de variáveis.

    Sugestão: p e q são equivalentes se e só se Conj (Impl p q) (Impl q p) for uma tautologia.

Árvore de Pesquisa

17. Complete as seguintes definições recursivas para uma árvore binária:

```
data Arv a = Vazia | No a (Arv a) (Arv a)
```

    (a) Escreva uma função recursiva para calcular o número de nós de uma árvore.

```
tamanho :: Arv a -> Int
tamanho Vazia          =
tamanho (No x esq dir) =
```

    (b) Escreva uma função recursiva para calcular a altura de uma árvore.

```
altura :: Arv a -> Int
altura Vazia          =
altura (No x esq dir) =
```

    (c) Escreva uma função recursiva para soma todos os valores de uma árvore binária de números

```
sumArv :: Num a => Arv a -> a
sumArv Vazia          =
sumArv (No x esq dir) =
```

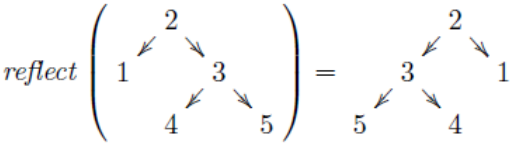
    (d) Escreva uma definição recursiva `nivel :: Int -> Arv a -> [a]` tal que nivel n arv é a lista ordenada dos valores da árvore no nível n, isto é, a uma altura n (considerando que a raiz tem altura 0).

```
nivel :: Int -> Arv a -> [a]
nivel _ Vazia          =
nivel 0 (No x esq dir) =
nivel n (No x esq dir) =
```

    (e) Escreva uma definição da função de ordem superior `mapArv :: (a -> b) -> Arv a -> Arv b` tal que mapArv f t aplica uma função f a cada valor duma árvore t. árvore t.

```
mapArv :: (a -> b) -> Arv a -> Arv b
mapArv f Vazia          =
mapArv f (No x esq dir) =
```

    (f) Escreva uma definição da função `reflect :: Arv a -> Arv a` que recursivamente troca os lados esquerdos e direitos de uma árvore. Exemplo:



    (g) Escreva uma definição da função que insere um valor numa árvore de pesquisa ordenada. Deve manter invariante a propriedade ordenação da árvore e não inserir outra cópia do valor se este já ocorrer na árvore.

```
inserir :: Ord a => a -> Arv a -> Arv a
inserir x Vazia =
inserir x (No y esq dir)
          | x<y =
          | x>y =
          | otherwise =
```

    (h) Escreva uma definição da função que remove um valor numa árvore de pesquisa ordenada. Deve manter invariante a propriedade ordenação da árvore.

```
mais_esq :: Arv a -> a
mais_esq (No x Vazia _) = x
mais_esq (No _ esq _) = mais_esq esq


remover :: Ord a => a -> Arv a -> Arv a
remover x Vazia = Vazia -- não ocorre
remover x (No y Vazia dir) -- um descendente
remover x (No y esq Vazia) -- um descendente

remover x (No y esq dir) -- dois descendentes
| x<y =
| x>y =
| x==y =
```

    Se a árvore tem dois descendentes, substitua x pelo menor valor da árvore da direita e depois remova o menor valor da árvore direita.

    (i) Escreva uma função recursiva para listar os elementos de uma árvore de pesquisa em ordem decrescente.

```
listar :: Ord a => Arv a -> [a]
listar Vazia          =
listar (No x esq dir) =
```

Multiconjuntos

18. Considere a definição em Haskell dum tipo de dados para multiconjuntos (i.e. coleções sem ordem mas com repetições) representado como árvore de pesquisa:

```
data MConj a = Vazio | No a Int (MConj a) (MConj a)
```

    Cada nó contém um valor e a sua multiplicidade (i.e. o número de repetições); para facilitar a pesquisa, a árvore deve estar ordenada pelos valores.

    Por exemplo:

```
No 'A' 2 Vazio (No 'B' 1 Vazio Vazio)
```

    representa o multi-conjunto {A, A, B} com dois carateres 'A' e um 'B'.

    (a) Escreva uma definição recursiva da função `ocorre :: Ord a => a -> MConj a -> Int` que procura o número de ocorrências de um valor num multi-conjunto; o resultado deve ser 0 se o valor não pertencer ao multi-conjunto.

    (b) Escreva uma definição recursiva da função `inserir :: Ord a => a -> MConj a -> MConj a` que insere um valor num multi-conjunto mantendo a árvore de pesquisa ordenada.

    (c) Escreva uma definição recursiva da função `listar :: MConj a -> [a]` para listar os elementos de um multi-conjunto.

    (d) Escreva uma definição recursiva da função `tamanho :: MConj a -> Int` para calcular o número de elementos de um multiconjunto.

    (e) Escreva uma definição recursiva da função `sumMConj :: MConj a -> Int` para calcular o somatório de todos os elementos de um multiconjunto.

Relações

19. Considere a representação de uma relação binária nos inteiros como uma lista de pares.

```
type Rel = [(Int , Int)]
```

    (a) Escreva uma função `reflexiva :: [Int]->Rel -> Bool` que verifique se uma relação R em A é reflexiva (R é reflexiva se e só se $\forall x \in A, (x, x) \in R$)

    Exemplos:

```
reflexiva [1,2,3] [(1, 1), (2, 2), (1, 2),(3, 3)] = True
reflexiva [1,2,3] [(1, 2), (2, 3)] = False
```

    (b) Escreva uma função `simetrica :: Rel -> Bool` que verifique se uma relação é transitiva (R é transitiva se e só se $(x, y) \in R \Rightarrow (y, x) \in R$ para todos x, y).

    Exemplos:

```
simetrica [(1, 3), (3, 1), (2, 2)] = True
simetrica [(1, 2), (2, 3)] = False
```

    (c) Escreva uma função `transitiva :: Rel -> Bool` que verifique se uma relação é transitiva (R é transitiva se e só se $(x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$ para todos x, y, z).

    Exemplos:

```
transitiva [(1, 3), (1, 2), (2, 3)] = True
transitiva [(1, 2), (2, 3)] = False
```

Grafos

20. Considere a representação de um grafo dirigido de vértices inteiros como um par ordenado uma lista de vértices e uma lista de arestas (isto é, pares ordenados de vertices).

```
type Vert = Int
type Grafo = ([Vert],[(Vert ,Vert)])
```

    Escreva uma função caminho :: `Grafo -> [Vert] -> Bool` tal que caminho g xs é True se xs é uma lista de vértices que representa um caminho no grafo (isto é, se cada dois vértices consecutivos correspondem a uma aresta) e False, caso contrário.

    Exemplos:

```
G = ([1,2,3], [(1, 2), (2, 1), (2, 3)])
caminho  G [1, 2, 1, 2, 3] = True
caminho  G [1, 2, 1, 3] = False
```