Programação Funcional
Folha de Exercícios 08
Entrada e Saída

Prof. Wladimir Araújo Tavares

Entrada e Saída

1. Escreva uma função `seq_ :: [IO a] -> IO ()` que realiza uma lista de ações.

```
Prelude> seq_ [print i | i <- [1..5]]
1
2
3
4
5
```

2. Escreva uma função `elefantes :: Int -> IO ()` tal que, por exemplo, elefantes 5 imprime os seguintes versos:

```
Se 2 elefantes incomodam muita gente,
3 elefantes incomodam muito mais!
Se 3 elefantes incomodam muita gente,
4 elefantes incomodam muito mais!
Se 4 elefantes incomodam muita gente,
5 elefantes incomodam muito mais!
```

   Sugestão: utilize a função `show :: Show a => a -> String` para converter um inteiro numa cadeia de caracteres; pode ainda reutilizar a função `seq_ :: [IO a] -> IO ()` para executar uma lista de ações.

3. 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.

4. 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 = 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.

5. Escreva um programa completo que reproduza a funcionalidade do utilitário wc: ler um ficheiro de entrada e imprime o número de linhas, número de palavras e número de caracteres.

```
$echo a maria tinha um cordeirinho | wc
Linhas: 1
Palavras: 5
Caracteres: 29
```

   Sugestão: Utilize as funções `words :: String -> [String]` e `lines :: String -> [String]`.

6. Faça um programa que leia um número n e imprime n!
7. Faça um programa que leia um número n e imprime "sim", se o número é primo e "nao", caso contrário.
8. Escreva a função `accumulate::[IO a] -> IO[a]`, que realiza uma lista de ações, acumulando o resultado dessas ações em uma lista.
9. Faça programa para que ele leia um número natural n, e então leia outros n números e calcule e exiba a soma destes números usando a função `accumulate`.

Mônadas

1. Mônadas são abstrações sobre construtores de tipos para os quais podem ser definidas duas funções, *return* e `(>>=)` com determinados tipos e propriedades.

   Em Haskell, vamos considerar que mônadas são instâncias da seguinte classe:

```
class Monad m where
  return:: a -> m a
  (>>=):: m a -> (a -> m b) -> m b
```

   return recebe um valor e "encaixota" esse valor na mônada. `(>>=)` recebe um valor monádico (isto é, um valor "encaixotado") e passa o resultado de desencaixotar o valor para o segundo argumento, que retorna o resultado de aplicar o primeiro valor, mas também encaixotado.

   A visão de *IO* como uma mônada serve principalmente para sequenciar ações de entrada e saída e garantir que toda ação de tipo *IO* a, para algum a, não possa ser "desencaixotada" e vista como um valor de tipo a, a não ser como resultado de operações de entrada e saída, usadas de modo sequencial na mônada.

   A instância de *Maybe* é dada a seguir:

```
instance Monad Maybe where
  return = Just
  Nothing >>= _ = Nothing
  Just x  >>= f = f x
```

   Ou seja, *Nothing* é propagado sempre que ocorre (como um caso de falha), e *Just* serve como uma caixa para um valor: `(>>=)` recebe um valor dentro da caixa e aplica a função passada como segundo argumento a esse valor.

   Um equilibrista está treinando equilibrismo na sua fazenda: ele anda sob uma corda esticada segurando uma longa barra de madeira. O problema maior que ele enfrenta é quando pássaros pousam na barra, e o número de pássaros que pousam de um lado da barra é quatro a mais do que os que pousam do outro lado. Sempre que isso ocorre, ele se desequilibra e cai.

   Vamos simular pássaros usando inteiros e a barra de madeira por um par de pássaros que estão pousados em cada lado da barra, esquerdo e direito:

```
type Passaros = Int
type Barra    = (Passaros, Passaros)
```

   Para simular o pouso de pássaros na barra, no lado esquerdo e direito, vamos usar respectivamente as duas funções a seguir:

```
pousoEsq:: Passaros -> Barra -> Maybe Barra
pousoEsq n (esq,dir)
  | abs ((esq + n) - dir) < 4 = Just (esq + n, dir)
  | otherwise                 = Nothing

pousoDir n (esq,dir)
  | abs (esq - (dir+n)) < 4 = Just (esq, dir+n)
  | otherwise               = Nothing
```

   Usando a função `>>=` da mônada *Maybe*, podemos simular o pouso de pássaros na barra sem ter que ficar testando casos de falha explicitamente no código. Por exemplo:

   return `(0,0) >>= pousoDir 2 >>= pousoEsq 2 >>= pousoDir 2`

   retorna *Just* `(2,4)`.

   E:

   return `(0,0) >>= pousoEsq 1 >>= pousoDir 4 >>= pousoEsq (-1) >>= pousoDir (-2)`

   retorna *Nothing*: *return* `(0,0)` retorna *Just* `(0,0)`, depois *pousoEsq* `1` retorna *Just* `(1,0)`, depois *pousoDir* `4` retorna *Just* `(1,4)`, mas *pousoEsq* `(-1)` retorna *Nothing*. E depois *Nothing* é "propagado", ou seja, retornado como resultado de *Nothing* `>>=` *pousoDir* `(-2)`, pela definição de `(>>=)` na mônada *Maybe*.

   Reescreva a função *rotina* a seguir, que trata casos de falha explicitamente, usando a mônada *Maybe* de modo a evitar os casos de tratamento explícito de falha:

```
rotina:: Maybe Barra
rotina = case pousoEsq 1 (0,0) of
           Nothing     -> Nothing
           Just barra1 -> case pousoDir 4 barra1 of
                            Nothing     -> Nothing
                            Just barra2 -> case pousoEsq 2 barra2 of
                                             Nothing     -> Nothing
                                             Just barra3 -> pousoEsq 1 barra3
```

   Escreva uma função `rotina :: Barra -> [Pouso] -> Maybe Barra` que recebe uma barra e uma lista de pousos e devolve uma Maybe Barra.

```
data Pos = Esq | Dir deriving (Eq,Show)
type Pouso = (Pos, Passaros)

b1 = (0,0) :: Barra
p1 = (Esq, 2) :: Pouso
p2 = (Dir, 4) :: Pouso
sq = [p1,p2]
rotina :: Barra -> [Pouso] -> Maybe Barra
rotina b1 sq == Just (2,4)
```

2. Seja

```
data Expr = Number Integer | Neg Expr |
Plus Expr Expr | Minus Expr Expr | Times Expr Expr |
Div Expr Expr | Mod Expr Expr
```

   (a) Escreva a função `eval1 :: Expr -> Maybe Integer` que avalia uma expressão. O valor Nothing deve ser retornado no caso de uma divisão por zero.

(b) Escreva a função `eval2 :: Expr -> [Integer]` que avalia uma expressão. O valor [] deve ser retornado no caso de uma divisão por zero.

(c) Generalize essa função para `eval3 :: Monad m => Expr -> m Integer` que usa uma mônada arbitrária (use return no lugar de Just e fail "Divisao por zero"no lugar de Nothing).

3. Defina o operador `(!?):: [a] -> Int -> Maybe a` tal que `xs !? n` retorna o n-ésimo elemento da lista caso ele exista. Caso contrário, retorne Nothing. A função `(!?)` é o operador seguro de índice (A versão segura do operador !!).

4. Defina a função `swap :: Int -> Int -> [a] -> Maybe [a]` que recebe dois índices de uma lista e troca os elementos destes índices em uma lista. Se os índices estão fora dos limites da lista retorne Nothing. Caso contrário, retorne a lista com os índices trocados.

5. A função mapM mapeia uma função monádica sobre uma lista.

   `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`

   Exemplo: `mapM Just [1..10] == Just [1..10]`

   Use mapM para definir uma função

   `getElts :: [Int] -> [a] -> Maybe [a]`

   que recebe uma lista de índices e uma lista e retorna uma lista dos elementos dos índices em uma mônada Maybe. Se algum dos índices não existe, a função deve retorna Nothing. Use o operador de índice definido por você (!?).

6. Considere o seguinte programa em Haskell:

```
type Person = String
type Family = [(Person,Person,Person)]

p1 = "Bart Simpsons"
p2 = "Lisa Simpsons"
p3 = "Marge Simpsons"
p4 = "Homer Simpsons"
p5 = "Maggie Simpsons"
p6 = "Abraham Simpsons"
p7 = "Mona Simpsons"
p8 = "Ned Flanders"
p9 = "Maude Flanders"
p10 = "Rod Flanders"
p11 = "Todd Flanders"

f = [(p4,p3,p1),
     (p4,p3,p2),
     (p4,p3,p5),
     (p6,p7,p4),
     (p8,p9,p10),
     (p8,p9,p11)]
```

   (a) Faça a função `father :: Family -> Person -> Maybe Person` que dado uma pessoa retorne o pai da pessoa se existir na família f.

   (b) Faça a função `mother :: Family -> Person -> Maybe Person` que dado uma pessoa retorne a mãe da pessoa se existir na família f.

   (c) Faça a função `paternalgrandfather :: Family -> Person -> Maybe Person` que dado uma pessoa retorne o avô paterno da pessoa se existir na família f.

   (d) Faça a função `paternalgrandmother :: Family -> Person -> Maybe Person` que dado uma pessoa retorne a avó paterno da pessoa se existir na família f.

   (e) Faça a função `bothGrandfathers :: Person -> Maybe (Person, Person)` que dado uma pessoa retorna os dois avôs paternos da pessoa se existir na família f.

7. Considere o seguinte programa em Haskell:

```
data Graph = Graph [Int] [(Int,Int)]

search :: Graph v e -> Int -> Int -> Maybe [Int]
search g@(Graph vl el) src dst
    | src == dst = Just [src]
    | otherwise = search' el
       where search' [] =
       search' ((u,v):es)
       | src == u =
                  case search g v dst of
                  Just p ->
                  Nothing ->
       | otherwise =
```

   Complete o programa acima.

```
gr = Graph [0, 1, 2, 3] [(0,1), (0,2), (1,3), (2,3)]
searchAll gr 0 3 :: Maybe [Int] == Just [0,1,3]
```

8. A classe *Functor* é usada em Haskell para permitir a generalização da função *map* para outros construtores de tipo além de listas. Para que mensagens de erro mais simples possam ser mais facilmente emitidas no caso de listas, *map* é chamada de *fmap* no caso geral.

```
class Functor f where
  fmap:: (a -> b) -> f a -> f b
```

Note: f é uma variável que deve ser instanciada para um construtor de tipo (por exemplo, para `[]`, *Maybe*, *Tree*, *IO*), não para um tipo. Por exemplo, `[]` pode ser definida como uma instância de *Functor* como a seguir:

```
instance Functor [] where
  fmap = map
```

Com essa definição de instância, podemos usar *fmap* do mesmo modo como *map*, em listas de qualquer tipo.

Tipos que "encaixotam" valores podem ser vistos como funtores: *fmap* pega uma função e uma caixa para valores de tipo a, aplica essa função aos valores desencaixotados e encaixota os valores de tipo b obtidos pela aplicação da função.

Outro exemplo de instância de *Functor* é do construtor *Maybe*:

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing  = Nothing
```

Com essa definição de instância, podemos usar *fmap* sobre valores construídos com *Maybe*:

```
fmap (++ "def") (Just "abc")
```

obtemos:

```
Just "abcdef"
```

(a) Considere o tipo: `data Tree a = Leaf | Node a (Tree a) (Tree a)` Defina uma instância de *Functor* para *Tree* de modo que possamos usar *fmap* para árvores como usamos *map* para listas.

(b) Considere o tipo: `data MultiTree a = Folha | No a [Tree a]` Defina uma instância de *Functor* para *MultiTree* de modo que possamos usar *fmap* para árvores como usamos *map* para listas.