Programação Funcional
Lista de Exercícios 05

Prof. Wladimir Araújo Tavares

Mônadas Maybe

1. A função `log :: Floating a => a -> a` não está definida para números negativos.

```
> log 1000
6.907755278982137
> log (-1000)
''ERROR'' -- runtime error
```

   Defina uma versão segura que evite runtime error usando Maybe.

```
safeLog :: (Floating a, Ord a) => a -> Maybe a
safeLog x
    | x > 0     =
    | otherwise =
```

2. A função `sqrt :: Floating a => a -> a` não está definida para números negativos. Defina uma versão segura evitando erros usando Maybe.
3. Faça a versão segura da função composta $log \circ sqrt(x) = log(sqrt(x))$ usando com a notação do.

```
safeLogSqrt :: (Floating a, Ord a) => a -> Maybe a
safeLogSqrt x = do {
                                }
```

4. Faça a versão segura da função composta $log \circ sqrt(x) = log(sqrt(x))$ usando sem a notação do usando **>>=**.

```
safeLogSqrt' :: (Floating a, Ord a) => a -> Maybe a
safeLogSqrt' x =
```

5. 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.

6. 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]
```

7. Vamos representar pontos de um plano cartesiano com duas coordernadas e regiões desse plano como funções, usando os seguintes tipos:

```
data Ponto = Pt Float Float
type Regiao = Ponto -> Bool
```

   Se r representa uma região do plano, então um ponto p está nessa região do plano se r p é igual a True.

   (a) Defina funções `retang::Ponto->Ponto->Regiao` e `circ::Ponto->Raio->Regiao` tais que:

   retang p q retorne (a região que representa) o retângulo tal que p é o ponto mais à esquerda e mais baixo, e q o ponto mais à direta e mais alto.

   circ p r retorne o círculo de raio r e centro p.

   Lembre-se: regiões são representadas por funções.

   (b) Defina funções `uniao::Regiao->Regiao->Regiao`, `interseccao::Regiao->Regiao->Regiao` e `complemento::Regiao->Regiao` tais que p está em uniao r r' se e somente se p está na uniao das regiões r e r', e analogamente para intersecção e complemento.