

Data: 10 de abril de 2012. Tempo disponível: 2h00m.

Prof.: Fernando Castor

Turma: Ciência da Computação

1. (5,0 ptos) Estude cuidadosamente o trecho de código abaixo e faça o que é pedido.

```
1 f F _ _ = []
2 f x m n = n (***) (m show x) []
3
4 g _ F = F
5 g i (R a e d) = R (i a) (g i e) (g i d)
6
7 h j F l = l
8 h j (R a e d) l
9 | (R a e d) == (R " " F F) = "" ++ (h j e l) 'j' (h j d l)
10 | otherwise = a 'j' (h j e l) 'j' (h j d l)
11
12 (***) a b = a ++ " " ++ b
13
14 result = f (R 1 (R 2 F F) (R 3 F F)) g h
```

(a) (3,0 ptos.) Determine assinaturas das funções `f`, `g`, `h` e `(***)` no trecho de código acima de modo que o código compile (incluindo o valor de `result`). Considere que `F` e `R` são construtores para valores de um mesmo tipo `T` (que **você** terá que definir de modo que o código compile!) e que os casamentos de padrões realizados pelas funções `g` e `h` são exaustivos para argumentos do tipo `T`. %

(b) (2,0 ptos.) Determine o tipo da expressão

`(map.f) F`

a)

`data T t = F | R t (T t) (T t)`

`(***) :: String -> String -> String`

`g :: (t -> u) -> (T t) -> (T u)`

2. (5,0 pts.) Defina uma estrutura de dados **Grafo t** (um grafo orientado) com um tipo algébrico polimórfico. Defina uma função

```
popularGrafo :: Eq t => [t] -> [(t,t)] -> Grafo t
```

que recebe uma lista de vértices, uma lista de arestas no formato (origem, destino) e retorna um novo **Grafo**. Agora defina uma função

```
buscaEmLargura :: Eq t => Grafo t -> t -> t -> Bool
```

que recebe um grafo, o vértice inicial, o vértice final e, usando uma busca em largura, devolve **True** se for possível alcançar o vértice final usando o vértice inicial como ponto de partida e **False** caso contrário. A função é auto explicativa, deverá usar o algoritmo de busca em largura e o tipo **t** deverá ser comparável. No grafo não deverá haver nós repetidos/iguais, de forma que, para os valores **v1** e **v2** armazenados em quaisquer dois nós distintos, **(==) v1 v2** produz **False**. Exemplos:

```
Prelude> let grafo = popularGrafo [1,2,3,4] [(1,2), (1,3), (3,4)]
```

```
Prelude> buscaEmLargura grafo 1 4
```

```
True
```

```
Prelude> let grafo = popularGrafo ['a','b','c','d'] [('a','b'), ('a','c'), ('d','b')]
```

```
Prelude> buscaEmLargura grafo 'a' 'd'
```

```
False
```

data Grafos t = Grafo [(t, [t])] deriving Show -- lista de (vertices e lista de vertices adjacentes)
-- o show foi colocado só a nível de teste no ghci

```
g = Grafo [(1, [2,3]),(2,[1]), (3, [1,4]), (4, [])] -- para testes
```

```
popularGrafo :: Eq t => [t] -> [(t,t)] -> Grafos t
```

```
popularGrafo vertices arestas = Grafo [(v, (retornaArestas v arestas)) | v <- vertices]
```

```
retornaArestas :: Eq t => t -> [(t,t)] -> [t]
```

```
retornaArestas vertice arestas = [a | (v, a) <- arestas, (v == vertice)]
```

```
buscaEmLargura :: Eq t => Grafos t -> t -> t -> Bool
```

```
buscaEmLargura grafo inicio destino = caminho destino (caminhos grafo (setAbertos grafo) inicio  
(retornaPossiveis grafo inicio (setAbertos grafo)))
```

```
setAbertos :: Grafos t -> [t]
```

```
setAbertos (Grafo []) = []
```

```
setAbertos (Grafo ((v,a):bs)) = v : setAbertos (Grafo bs)
```

```
removeAberto :: Eq t => [t] -> t -> [t]
```

```
removeAberto vertices vertice = [v | v <- vertices, v /= vertice]
```

```
retornaPossiveis :: Eq t => Grafos t -> t -> [t] -> [t]
```

```
retornaPossiveis grafo vertice abertos = [a | a <- abertos, ([x | x <- (adjacentes grafo vertice), (x ==  
a)] /= [])]
```

```
adjacentes :: Eq t => Grafos t -> t -> [t]
```

```
adjacentes (Grafo []) vertice = []
```

```
adjacentes (Grafo ((v,a):bs)) vertice
```

```
  | (vertice == v) = a
```

```
  | otherwise = adjacentes (Grafo bs) vertice
```

```
caminhos :: Eq t => Grafos t -> [t] -> t -> [t] -> [t] -- retorna uma lista de lista de nós alcançáveis do  
início
```

```
caminhos grafo abertos inicio possiveisArestas = foldr (++) [] [(x : caminhos grafo (removeAberto  
abertos inicio) x (retornaPossiveis grafo x (removeAberto abertos inicio))) | x <- possiveisArestas]
```

```
caminho :: Eq t => t -> [t] -> Bool
```

```
caminho vertice caminhos = ([x | x <- caminhos, x == vertice] /= [])
```