Primeiro EE de PLC (IF686)

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.

```
f F _ _ = []
f x m n = n (***) (m show x) []

g _ F = F
g i (R a e d) = R (i a) (g i e) (g i d)

h j F l = l
h j (R a e d) l
  | (R a e d) == (R "␣" F F) = "" ++ (h j e l) `j` (h j d l)
  | otherwise = a `j` (h j e l) `j` (h j d l)

(***) a b = a ++ "␣" ++ b

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)

(***) Concatena Strings, logo: (***) :: String -> String -> String

Como em h se tem uma árvore de String e em result uma de Int, temos: data T t = F | R t (T t) (T t)

De acordo com result: f :: (T a) -> tipo de g -> tipo de h (sabendo que o retorno é uma lista)

g aplica i nos nós da árvore, logo: g :: (a -> b) -> (T a) -> (T b)

h :: tipo de j -> (T a) -> String -> String, pois retorna uma String ou l (que é o terceiro parâmetro). E como j concatena strings, incluindo da árvore, logo h :: (String -> String -> String) -> (T String) -> String -> String

Voltando para f, como temos que h recebe strings, então g seria do tipo g :: (a -> String) -> (T a) -> (T Stringo), logo, f :: (T a) -> ((a -> String) -> (T a) -> (T String)) -> ((String -> String -> String) -> (T String) -> String -> String)

Sendo assim, f recebe uma árvore de inteiros, transforma em árvore de strings com g e finalmente transforma em uma única string com h, semelhante a uma função show.

b)

map.f
map:: (a -> b) -> [a] -> [b]
. :: (d -> e) -> (c -> d) -> c -> e
f :: (T t) -> ((t -> String) -> (T t) -> (T String)) -> ((String -> String -> String) -> (T String) -> String -> String)

(d -> e) ~ (a -> b) -> [a] -> [b]
d ~ (a -> b)
e ~ [a] -> [b]

(c -> d) ~ (T t) -> ((t -> String) -> (T t) -> (T String)) -> ((String -> String -> String) -> (T String) -> String -> String)
c ~ (T t)
d ~ ((t -> String) -> (T t) -> (T String)) -> ((String -> String -> String) -> (T String) -> String -> String)

comparando d
(a -> b) ~ ((t -> String) -> (T t) -> (T String)) -> ((String -> String -> String) -> (T String) -> String -> String)
a ~ ((t -> String) -> (T t) -> (T String))
b ~ ((String -> String -> String) -> (T String) -> String -> String)

c -> e ~ (T t) -> [((t -> String) -> (T t) -> (T String))] -> [((String -> String -> String) -> (T String) -> String -> String)]

F :: (T t)

(map.f) F :: [((t -> String) -> (T t) -> (T String))] -> [((String -> String -> String) -> (T String) -> String -> String)]

2. (5,0 ptos.) 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)

g = Grafo [(1, [2,3,5]),(2,[4]), (3, [1,2]), (4, [5]), (5,[])] -- para testes

```haskell
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 (removeAberto
(setAbertos grafo) inicio) inicio (retornaPossiveis grafo inicio (removeAberto (setAbertos
grafo)inicio)))

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 nós alcançáveis do início
caminhos grafo abertos inicio [] = []
caminhos grafo abertos inicio (a:as) = a : (caminhos grafo (removeAberto abertos a) inicio as) ++
(caminhos grafo (removeAberto abertos a) a (retornaPossiveis grafo a (removeAberto abertos a)))

caminho :: Eq t => t -> [t] -> Bool
caminho vertice caminhos = ([x | x <- caminhos, x == vertice] /= [])
```