Primeira Prova de PLC (IF686)

Data: 22 de abril de 2010. **Tempo disponível: 2h00m.** **Prof.:** Fernando Castor

1. (2,0) Responda as perguntas abaixo.

(a) (1,0 ptos.) Quais são as diferenças entre tipificação estática e dinâmica em linguagens de programação? Quais são as vantagens e desvantagens de cada abordagem?

A tipificação estática é verificada em tempo de execução, enquanto a dinâmica é verificada em tempo de compilação. O primeiro tipo de abordagem garante de antemão que o programa estará correto para todas as possíveis entradas, ma limita a liberdade do programador. Já o segundo tipo de abordagem possibilita essa liberdade, porém pode ocasionar em erros durante a execução.

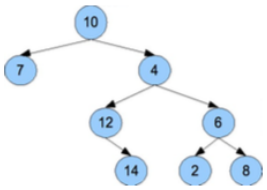
(b) (0,5 ptos.) Que tipo de tipificação Haskell usa? Estática ou dinâmica? E Java? Por que em Haskell, normalmente, não é necessário especificar o tipo de variáveis e funções?

Haskell e Java utilizam a tipificação estática, visto que verificam os erros em tempo de compilação. Porém, em Haskell não é necessário especificar o tipo de variáveis e funções porque a linguagem não é fortemente tipada, ou seja, a declaração do tipo não é obrigatória.

(c) (0,5 ptos.) Por que o uso de produtos cartesianos não é suficiente para definir classes em linguagens de programação orientadas a objetos?

Porque as classes não são apenas compostas por atributos, mas também possuem uma série de métodos.

2. (3,5 ptos.) Defina um tipo de dados para árvores binárias (`Tree`) polimórficas, ou seja, cada nó da árvore guarda um valor de um tipo arbitrário (embora todos os nós guardem valores do mesmo tipo). Defina também uma função chamada `dfs` que, dada uma árvore, a percorre em profundidade (*depth-first*) imprimindo o conteúdo de cada nó ao passar por ele. Note que a função deve ser capaz de imprimir árvores para qualquer tipo de valor "imprimível" (não apenas inteiros, reais, etc.). A raiz de cada sub-árvore deve ser impressa **antes** dos nós da sub-árvore à esquerda. Ou seja, dada a árvore



uma chamada a `dfs` passando tal árvore como argumento **deve imprimir** o seguinte (incluindo as vírgulas e o ponto): `10, 7, 4, 12, 14, 6, 2, 8.`

```
data Tree t = NilT | Node t (Tree t) (Tree t)

aux :: Show t => Tree t -> String
aux NilT = ""
aux (Node no (esquerda) (direita)) = "," ++ (show no) ++ (aux esquerda) ++ (aux direita)

dfs :: Show t => Tree t -> String
dfs tree = (tail (aux tree)) ++ "."
```

-- Ex.: dfs (Node 10 (Node 7 NilT NilT) (Node 4 (Node 12 NilT (Node 14 NilT NilT)) (Node 6 (Node 2 NilT NilT) (Node 8 NilT NilT))))

3. (2,5 ptos.) Quais são os tipos das funções abaixo? Mostre como você obteve esses tipos. Se for necessário, identifique as classes dos parâmetros polimórficos. Caso não seja possível determinar o tipo de alguma das funções, explique o porquê.

(a) (1,0 pto.) `map.map.foldr`

(b) (1,5 ptos.) `map.((.)  (foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]])))`

a)

map.foldr:

map :: (a->b) -> [a] -> [b]
. :: (d -> e) -> (c -> d) -> c -> e
foldr :: (f -> g -> g) -> g -> [f] -> g

(d -> e) ~ (a ->b) -> [a] -> [b]
d ~ (a -> b)
e ~ [a] -> [b]
(c -> d) ~ (f -> g -> g) -> g -> [f] -> g
c ~ (f -> g -> g)
d ~ g -> [f] -> g
comparando d:
a ~ g
b ~ [f] -> g

c -> e ~ (f -> g -> g) -> [a] -> [b] ~ (f -> g -> g) -> [g] -> [[f] -> g]

map.foldr :: (f -> g -> g) -> [g] -> [[f] -> g]

map.map.foldr:

map :: (h -> i) -> [h] -> [i]
. :: (k -> l) -> (j -> k) -> j -> l
map.fold :: (f -> g -> g) -> [g] -> [[f] -> g]

(k -> l) ~ (h -> i) -> [h] -> [i]
k ~ (h -> i)
l ~ [h] -> [i]
(j -> k) ~ (f -> g -> g) -> [g] -> [[f] -> g]
j ~ (f -> g -> g)
k ~ [g] -> [[f] -> g]
comparando k
h ~ [g]
i ~ [[f] -> g]

j -> l ~ (f -> g -> g) -> [h] -> [i] ~ (f -> g -> g) -> [[g]] -> [[[f] -> g]]

map.map.fold :: (f -> g -> g) -> [[g]] -> [[[f] -> g]]

b)

map.((.) (foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]])))

foldr (++) [] [[1], [2], [4,5,6], [3]] = [1,2,3,4,5,6,3] ~ Num a => [a]

foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]]):

foldr :: (b -> c -> c) -> c -> [b] -> c
(++) :: [d] -> [d] -> [d]
foldr (++) [] [[1], [2], [4,5,6], [3]] :: Num a => [a]

(b -> c -> c) ~ [d] -> [d] -> [d]
b ~ [d]
c ~ [d]
c ~ Num a -> [a]
b ~ [d] ~ c ~ Num a => [a]

[b] -> c ~ Num a => [[a]] -> [a]

foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]]) :: Num a => [[a]] -> [a]

((.) (foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]]))):

. :: (f -> g) -> (e -> f) -> e -> g
foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]]) :: Num a => [[a]] -> [a]

(f -> g) ~ Num a => [[a]] -> [a]
f ~ Num a => [[a]]
g ~ Num a => [a]

(e -> f) -> e -> g ~ Num a => (e -> [[a]]) -> e -> [a]

((.) (foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]]))) :: Num a => (e -> [[a]]) -> e -> [a]

map.((.) (foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]]))):

map :: (h -> i) -> [h] -> [i]
. :: (k -> l) -> (j -> k) -> j -> l
((.) (foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]]))) :: Num a => (e -> [[a]]) -> e -> [a]

(k -> l) ~ (h -> i) -> [h] -> [i]
k ~ (h -> i)
l ~ [h] -> [i]
 (j -> k) ~ Num a => (e -> [[a]]) -> e -> [a]
j ~ Num a => (e -> [[a]])
k ~ Num a => e -> [a]
comparando k:
h ~ e
i ~ Num a => [a]

j -> l ~ Num a => (e -> [[a]]) -> [h] -> [i] ~ Num a => (e -> [[a]]) -> [e] -> [[a]]

map.((.) (foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]]))):: Num a => (e -> [[a]]) -> [e] -> [[a]]

4. (3,0 ptos.) Defina, usando as construções de Haskell que você aprendeu até agora, um tipo de dados chamado Grafo, que implementa um grafo direcionado cujos nós contém números inteiros (os rótulos dos nós). Defina em seguida uma função chamada `mapEdges` cuja assinatura é a seguinte:

```
mapEdges::(Int->Int->Int)->(Grafo Int)->(Int->[Int])
```

A função `mapEdges` retorna uma nova função que, quando invocada com o valor guardado em um nó `N` do grafo, devolve uma lista cujos elementos são o resultado de aplicar a função recebida como argumento por `mapEdges` (do tipo `Int->Int->Int`) aos valores armazenados nos nós das arestas que partem de `N`. Por exemplo, usando a árvore (`arvore`) da questão 2 como um grafo direcionado, uma chamada `mapEdges (+) arvore` produziria uma função que, ao ser chamada com o argumento `10` (o valor armazenado na raiz de `arvo re`), produziria como resultado a lista `[17, 14]` (resultado de aplicar `(+)` a 10 e 7 e a 10 e 4).

```
data Grafos = Grafo [(Int, [Int])] -- primeiro int = no, segundo int = nos adjacentes

mapEdges :: (Int -> Int -> Int) -> (Grafos) -> Int -> [Int]
mapEdges f grafo no = [f no x | x <- (auxMap grafo no)]

auxMap :: (Grafos) -> Int -> [Int]
auxMap (Grafo ((x,y):as)) no
   | x == no = y
   | otherwise = auxMap (Grafo as) no

-- Ex.: mapEdges (+) (Grafo [(10, [7,4]), (7,[]), (4, [12,6]), (12, [14]), (14,[]), (6, [2,8]), (2,[]),
(8,[])]) 10
```