Primeira Prova de PLC (IF686)

Data: 15 de outubro de 2010. **Tempo disponível: 2h00m.** **Prof.:** Fernando Castor

1. (1,5) Descreva em no máximo **uma página** o projeto do seu grupo para essa disciplina. Forneça uma visão geral sobre o que é o projeto (forneça exemplos de código ou explique a mecânica do jogo, conforme a necessidade) e descreva as principais funções (com as assinaturas) e tipos de dados implementados no desenvolvimento do projeto.

Obs.: Cada décimo abaixo de 1,0 na nota desta questão implica na perda de 0,2 ponto na nota **final** do projeto (considerando as duas partes). Exemplo: um aluno que tirar 0,5 nesta questão (ou seja, acertar apenas 33% dela) terá um redutor de 1,0 ponto na nota final do projeto.

2. (2,5) Faça o que é pedido:

(a) (0,5 pto.) Crie um tipo algébrico chamado `Habitantes` para representar algumas das raças de habitantes da Terra Média: Elfos, Humanos, Anões e Hobbits. Indivíduos de cada uma dessas raças têm um nome, um *string* associado a valores do tipo `Habitantes`. Valores desse tipo algébrico devem ser exibíveis/imprimíveis. Além disso, deve ser possível comparar valores do tipo `Habitantes` para saber se são iguais.

(b) (0,5 pto.) Defina uma classe chamada `Comp` com apenas uma função chamada `maisImportante`, polimórfica, que tem dois parâmetros de um mesmo tipo e fornece como resultado um `Bool`.

(c) (1,5 ptos.) Defina uma instância da classe `Comp` que implemente `maisImportante` para dois argumentos do tipo `Habitante`. Essa função deve devolver `True` se o primeiro argumento for mais importante que o segundo e `False` caso contrário. Considere que Elfos são mais importantes que Humanos, que são mais importantes que Anões, que, obviamente, são mais importantes que Hobbits. Se os dois argumentos corresponderem a habitantes da mesma raça, o primeiro é mais importante que o segundo se seu nome for maior, de acordo com o operador >, já definido para *strings*. (Dica: pense também nos casos em que o primeiro elemento é **menos** importante.)

a)

```
data Habitantes = Elfos String | Humanos String | Anoes String | Hobbits String deriving (Show, Eq)
```

b)

```
class Comp t where
maisImportante :: t -> t -> Bool
```

c)

```
instance Comp Habitantes where
   maisImportante (Elfos n1) (Humanos n2) = True
   maisImportante (Elfos n1) (Anoes n2) = True
   maisImportante (Elfos n1) (Hobbits n2) = True
   maisImportante (Humanos n1) (Elfos n2) = False
   maisImportante (Humanos n1) (Anoes n2) = True
   maisImportante (Humanos n1) (Hobbits n2) = True
   maisImportante (Anoes n1) (Elfos n2) = False
   maisImportante (Anoes n1) (Humanos n2) = False
   maisImportante (Anoes n1) (Hobbits n2) = True
   maisImportante (Hobbits n1) (Elfos n2) = False
   maisImportante (Hobbits n1) (Humanos n2) = False
   maisImportante (Hobbits n1) (Elfos n2) = False
   maisImportante (Elfos n1) (Elfos n2) = (n1 < n2)
   maisImportante (Humanos n1) (Humanos n2) = (n1 < n2)
   maisImportante (Anoes n1) (Anoes n2) = (n1 < n2)
   maisImportante (Hobbits n1) (Hobbits n2) = (n1 < n2)

-- maisImportante (Elfos "Tulio") (Humanos "Jesus") -- True
```

3. (5,0) Faça o que é pedido:

(a) (0,5 pto.) Defina um tipo de dados para árvores binárias (`Arvore`) polimórficas, ou seja, cada nó da árvore guarda um valor de um tipo arbitrário (embora todos os nós guardem valores do mesmo tipo). Valores do tipo `Arvore` devem ser exibíveis/imprimíveis.

(b) (2,5 ptos.) Defina uma função chamada `criarArvoreDeImportancia` que, dada uma lista de valores de um tipo que pertence à classe `Comp` da questão anterior, cria uma árvore tal que à esquerda da raiz estão todos os elementos menos importantes que a raiz e à direita todos os mais importantes. Essa política é aplicada recursivamente a cada sub-árvore (ou seja, funciona como uma árvore de busca não-balanceada).

(c) (2.0 ptos.) Defina uma função chamada `filhos` que, dada uma árvore resultante de `criarArvoreDeImportancia`, devolve uma função que recebe um valor do tipo `Habitante` como parâmetro e, ao ser invocada, produz uma lista com todos os habitantes que, na árvore original, são mais importantes que o `Habitante` fornecido como argumento, contanto que este **esteja** na árvore (caso contrário, produz uma lista vazia). Ou seja, dada a árvore `arvore_anoes = No (Anao "Gimli") (No (Anao "Bombur") Nil Nil) (No (Anao "Gloin") Nil (No (Anao "Thorin") Nil Nil))`, a chamada `filhos arvore_anos` fornece como resultado a lista `[Anao "Gloin", Anao "Thorin"]` (os elementos dessa lista não têm nenhuma ordem em particular).

a)

```
data Tree t = NilT | Node t (Tree t) (Tree t) deriving (Show, Eq)
```

b)

```
criarArvoreDeImportancia :: [Habitantes] -> Tree Habitantes
criarArvoreDeImportancia lista = aux lista NilT

aux :: [Habitantes] -> Tree Habitantes -> Tree Habitantes
aux [] arvore = arvore
aux (a:as) arvore = aux as (inserir a arvore)

inserir:: Habitantes -> Tree Habitantes -> Tree Habitantes
inserir habitante NilT = (Node habitante NilT NilT)
inserir habitante1 (Node habitante2 esquerda direita)
   | maisImportante habitante1 habitante2  = (Node habitante2 esquerda (inserir habitante1
direita))
   | maisImportante habitante2 habitante1 =  (Node habitante2 (inserir habitante1 esquerda)
direita)

-- criarArvoreDeImportancia [Elfos "Castor", Humanos "Tulio", Humanos "Jesus", Elfos "AulaPLC"]
-- Node (Elfos "Castor") (Node (Humanos "Tulio") NilT (Node (Humanos "Jesus") NilT NilT)) (Node
(Elfos "AulaPLC") NilT NilT)
```

c)

```
filhos :: Tree Habitantes -> (Habitantes -> [Habitantes])
filhos NilT _ = []
filhos (Node no (esquerda) (direita)) habitante
   | maisImportante no habitante =  filhos esquerda habitante
   | maisImportante habitante no =  filhos direita habitante
   | otherwise = exibir direita

exibir :: Tree Habitantes -> [Habitantes]
exibir NilT = []
exibir (Node no (esquerda) (direita)) = no : (exibir esquerda) ++ (exibir direita)

arvore_anoes = Node (Anoes "Gimli") (Node (Anoes "Bombur") NilT NilT) (Node (Anoes "Gloin")
NilT (Node (Anoes "Thorin") NilT NilT))

-- filhos arvore_anoes (Anoes "Gimli") -- [Anoes "Gloin",Anoes “Thorin"]
```

4. (2,0) 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.) `(+ 2).((*) 2).(2 /)`

(b) (1,0 pto.) `(.).filter`

a)

((*) 2) . (2 /):

(2 /) :: Fractional a => a -> a
. :: (c - > d) -> (b -> c) -> b -> d
((*) 2) :: Num e => e -> e

(c -> d) ~ Fractional a => a -> a
c ~ d ~ Fractional a => a
d ~ Fractional a => a
(b -> c) ~ Num e => e -> e
b ~ c ~ Num e => e

b -> d ~ Fractional a => a -> a

((*) 2) . (2 /): Fractional a => a -> a

(+ 2) . ((*) 2) . (2 /):

(+ 2) :: Num f => f -> f
. :: (h -> i) -> (g -> h) -> g -> i
((*) 2) . (2 /): Fractional a => a -> a

(h -> i) ~ Num f => f -> f
h ~ i ~ Num f => f
(g -> h) ~ Fractional a => a -> a
g ~ h ~ Fractional a => a

g -> i ~ Fractional a => a -> a

(+ 2) . ((*) 2) . (2 /) :: Fractional a => a -> a

b)

(.).filter :

. :: (b -> c) -> (a -> b) -> a -> c
. :: (e -> f) -> (d -> e) -> d -> f
filter :: (g -> Bool) -> [g] -> [g]

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

comparando e:
b ~ [g]
c ~ [g]

d -> f ~ (g -> Bool) -> (a -> [g]) -> a -> [g]

(.).filter :: (g -> Bool) -> (a -> [g]) -> a -> [g]