Primeiro Exercício Escolar de PLC (IF686)

Data: 10 de maio de 2011.

Início: 10h00m.

Término: 12h00m.

Prof.: Fernando Castor

1. (3,0 ptos.) Faça o que é pedido.

(a) (0,5 pto.) Defina um tipo de dados `Tree` correspondente às árvores binárias cujos nós contém valores que podem ser comparados através de operadores relacionais básicos.

(b) (1,5 ptos.) Implemente uma função chamada `buildSearchTree` que, dada uma lista de valores para os quais os operadores relacionais básicos estão definidos, constrói uma árvore de busca (**não necessariamente balanceada**).

(c) (1,0 ptos.) Implemente uma função chamada `searchTreeSort` que, dada uma lista de valores para os quais os operadores relacionais básicos estão definidos, produz uma lista que contém todos os elementos da lista fornecida como entrada, ordenados do menor para o maior, e necessariamente usando a função `buildSearchTree` para alcançar este fim.

a)
```
data Tree t = Node t (Tree t) (Tree t) | Nil deriving (Show)
```
— neste caso o deriving Show é só para conseguirmos ver a árvore da questão b)

b)
```
buildSearchTree :: (Ord t) => [t] -> Tree t
buildSearchTree lista = aux lista Nil

aux :: (Ord t) => [t] -> Tree t -> Tree t
aux [] arvore = arvore
aux (a:as) arvore = aux as (insereTree arvore a)

insereTree :: (Ord t) => Tree t -> t -> Tree t
insereTree Nil elemento = Node elemento Nil Nil
insereTree (Node no esquerda direita) elemento
   | no > elemento = Node no (insereTree esquerda elemento) direita
   | no < elemento = Node no esquerda (insereTree direita elemento)

-- buildSearchTree [5,2,1,3,7,6,15]
```

c)
```
searchTreeSort :: (Ord t) => [t] -> [t]
searchTreeSort lista = ordena (buildSearchTree lista)

ordena :: Tree t -> [t]
ordena Nil = []
ordena (Node no esquerda direita) = (ordena esquerda) ++ (no:[]) ++ (ordena direita)

searchTreeSort :: (Ord t) => [t] -> [t]
searchTreeSort lista = ordena (buildSearchTree lista)
```

ordena :: Tree t -> [t]
ordena Nil = []
ordena (Node no esquerda direita) = (ordena esquerda) ++ (no:[]) ++ (ordena direita)

-- searchTreeSort [5,2,1,3,7,6,15]

2. Determine os tipos das expressões abaixo, levando em conta as classes dos tipos das funções polimórficas, se necessário.

(a) (2,0 ptos.) Dada uma função `f::Ord a => [a] -> (a -> [b]) -> [[b]]`, determine o tipo da expressão

```
f.map (+1)
```

(b) (2,0 ptos.) `((:).foldr.foldr) (:)`

a)

map (+1) :
map :: (c -> d) -> [c] -> [d]
(+1) :: Num e => e -> e
(c -> d) ~ Num e => e -> e
c ~ Num e => e
d ~ Num e => e

[c] -> [d] ~ Num e => [e] -> [e]
map (+1):: Num e => [e] -> [e]

f.map (+1):
f :: Ord a => [a] -> (a -> [b]) -> [[b]]
. :: (u -> v) -> (t -> u) -> t -> v
map (+1) :: Num e => [e] -> [e]

(u -> v) ~ Ord a => [a] -> (a -> [b]) -> [[b]]
u ~ Ord a => [a]
v ~ Ord a => (a -> [b]) -> [[b]]
(t -> u) ~ Num e => [e] -> [e]
t ~ Num e => [e]
u ~ Num e => [e]
comparando u:
Ord a => [a] ~ Num e => [e]

t -> v ~ (Ord a, Num e) => [e] -> (a -> [b]) -> [[b]] ~ (Ord e, Num e) => [e] -> (e -> [b]) -> [[b]]
f.map (+1):: (Ord e, Num e) => [e] -> (e -> [b]) -> [[b]]

b)

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

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

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

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

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

b -> d ~ (f -> [f] -> [f]) -> [a] -> [a] ~ (f -> [f] -> [f]) -> [[f] -> [[f]] -> [f]] -> [[f] -> [[f]] -> [f]]
(:).foldr.foldr :: (f -> [f] -> [f]) -> [[f] -> [[f]] -> [f]] -> [[f] -> [[f]] -> [f]]

((:).foldr.foldr) (:) :
(:).foldr.foldr :: (f -> [f] -> [f]) -> [[f] -> [[f]] -> [f]] -> [[f] -> [[f]] -> [f]]
(:) :: h -> [h] -> [h]

(f -> [f] -> [f]) ~ h -> [h] -> [h]
f ~ h

[[f] -> [[f]] -> [f]] -> [[f] -> [[f]] -> [f]] ~ [[h] -> [[h]] -> [h]] -> [[h] -> [[h]] -> [h]]
((:).foldr.foldr) (:) :: [[h] -> [[h]] -> [h]] -> [[h] -> [[h]] -> [h]]

3. (4,0 ptos.) Construa uma função `listPartitioner :: Num a => [a] -> ([a]->[[a]])` que, dada uma lista l de tamanho n contendo apenas números não-duplicados, devolve uma outra função que, ao ser chamada com uma lista l' de tamanho $m \geq n$ contendo apenas números como argumento, particiona os elementos de l' em regiões (listas) de acordo com os valores dos elementos de l. O particionamento consiste em criar listas contendo elementos de l' que são menores ou iguais a um elemento de l, mais uma lista contendo apenas os elementos maiores que qualquer elemento de l, se houver algum. O resultado da função é uma lista contendo todas as listas produzidas.

Os seguinte exemplo ilustra o funcionamento de `listPartitioner`. Dada uma lista `[4, 13, 8]`, a chamada

```
listPartitioner [4,13,8]
```

devolve uma função que, se for chamada com a lista `[1..15]`, devolve como resultado a lista

```
[[1,2,3,4], [5,6,7,8], [9,10,11,12,13], [14,15]].
```

No exemplo acima, `[1,2,3,4]` é a lista com todos os elementos da lista `[1..15]` menores ou iguais a `4`, `[5,6,7,8]` é a lista com todos os elementos menores ou iguais a `8`, `[9,10,11,12,13]` é a lista com todos os elementos menores ou iguais a `13` e `[14,15]` é a lista com elementos maiores que o maior elemento da lista `[4,13,8]`. Note que nenhuma das listas recebidas como entrada precisa estar ordenada (mas as saídas precisam).

```
listPartitioner :: (Ord a, Num a) => [a] -> ([a] -> [[a]])
listPartitioner as = filtro (qsort as)

filtro :: (Ord a, Num a) => [a] -> [a] -> [[a]]
filtro (a:[]) bs = [qsort (filter (<=a) bs)] ++ [qsort((filter (>a) bs))]
filtro (a:as) bs = [qsort (filter (<=a) bs)] ++ filtro as (filter (>a) bs)

qsort :: (Ord a, Num a) => [a] -> [a]
qsort [] = []
qsort (a:as) = [ x | x <- as, x < a ] ++ [a] ++ [ x | x <- as, x > a]
```