Paradigmas de Linguagens Computacionais
Exame Escrito

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco

22 de setembro de 2011

Questão 1 Defina em Haskell as seguintes funções que trabalham com listas que representam conjuntos de elementos:

1. `member`, que recebe um elemento e uma lista e indica se o elemento pertence a lista;
2. `union`, que recebe duas listas e retorna uma lista representando a união das duas recebidas;
3. `inter`, que recebe duas listas e retorna uma lista representando a interseção das duas recebidas;
4. `diff`, que recebe duas listas e retorna uma lista representando a diferença da primeira em relação à segunda.

Evite elementos repetidos nas listas retornadas. Use compreensão de listas para definir `inter` e `diff`, e evite repetição de código. Indique e explique o tipo das funções `member` e `union`. □

1.

Tipo da função member:
Recebe um elemento t, uma lista [t] e retorna True (caso o elemento pertença a lista) ou False (Bool), o (Eq t) foi utilizado porque irá comparar os elementos da lista com a entrada.

```
member :: (Eq t) => t -> [t] -> Bool
member elemento lista = not ([x | x <- lista, x == elemento] == [])
```

2.

Tipo da função union:
Recebe uma lista [t], outra lista [t] e retorna a união das duas listas que também é uma lista [t]. O (Ord t, Eq t) foi utilizado porque a lista será ordenada usando o quicksort e os elementos iguais a algum outro elemento da lista serão removidos pela função removeRepeticao.

```
union :: (Ord t, Eq t) => [t] -> [t] -> [t]
union lista1 lista2 = removeRepeticao $ quickSort (lista1 ++ lista2)

quickSort :: (Ord t) => [t] -> [t]
quickSort [] = []
quickSort (a:as) = quickSort ([ x | x <- as, x <= a]) ++ [a] ++ quickSort ([x | x <- as, x > a])

removeRepeticao :: (Eq t) => [t] -> [t]
removeRepeticao [] = []
removeRepeticao (a:as) = a : removeRepeticao([x | x <- (a:as), x /= a])
```

3.

```
inter :: (Eq t, Ord t) => [t] -> [t] -> [t]
inter lista1 lista2 = removeRepeticao $ quickSort ([x | x <- lista1, member x lista2] ++ [x | x <- lista2, member x lista1])
```

4.

```
diff :: (Eq t, Ord t) => [t] -> [t] -> [t]
diff [] lista2 = []
diff lista1 lista2 = removeRepeticao $ quickSort [x | x <- lista1, not(member x lista2)]
```

Questão 2 Defina em Haskell o tipo de dados `FaceBookPost` e tipos auxiliares. Cada elemento desse tipo é formado pelo autor da mensagem, o texto associado, e uma lista de objetos associados. O autor e o texto podem ser representados por tipos auxiliares definidos como strings. Para não fixar objetos de tipos específicos (fotos, músicas, *links*, etc.), o tipo dos objetos deve ser um parâmetro de `FaceBookPost`. Com base nesse tipo,

1. defina a função `postsDeAmigosInteressantes` que recebe uma lista de autores (amigos) e uma lista de *posts* e retorna apenas os *posts* dos autores passados como parâmetro;
2. para poder comparar e transformar em string os elementos de `FaceBookPost`, defina instâncias das classes `Eq` e `Show`.

```
type Autor = String
type Texto = String
data FaceBookPosts t = FaceBookPost (Autor, Texto, [t]) -- onde t é a lista de objetos

postsDeAmigosInteressantes :: [Autor] -> [FaceBookPosts t] -> [FaceBookPosts t]
postsDeAmigosInteressantes autores posts = [ FaceBookPost (x, y, [z]) | FaceBookPost (x, y, [z]) <- posts, member x autores]

-- postsDeAmigosInteressantes ["tulio", "jesus"] [FaceBookPost ("tulio", "oi", [1]), FaceBookPost ("jesus", "ola", [2]), FaceBookPost ("alguem", "xau", [3])]

instance (Eq a) => Eq (FaceBookPosts a) where
  FaceBookPost (x1, y1, [z1]) == FaceBookPost (x2, y2, [z2]) = (x1 == x2) && (y1 == y2) && ([z1] == [z2])

-- FaceBookPost ("tulio", "oi", [1]) == FaceBookPost ("tulio", "oi", [1]) -- True
-- FaceBookPost ("tulio", "oi", [1]) == FaceBookPost ("daniel", "ola", [2]) -- False

instance (Show a) => Show (FaceBookPosts a) where
    show (FaceBookPost (x1, y1, [z1])) = "Autor: " ++ x1 ++ ". Texto: " ++ y1 ++ ". Objetos" ++ (show [z1])
-- show (FaceBookPost ("tulio", "oi", [1])) -- "Autor: tulio. Texto: oi. Objetos[1]"
```

Questão 3 Defina em Haskell o tipo de dados `Filtro`, que representa uma linguagem de filtragem de *posts* do *Facebook*. Por exemplo, o filtro

```
And (Author ["Pedro","Mariana","Arthur"]) (Not (Text ["Barbie"]))
```

retorna apenas as mensagens que sejam dos autores listados acima e cujo texto não contenha "Barbie". Além dos operadores listados acima, a linguagem deve ter o operador `Or`. Defina então a função `interp`, que recebe um filtro e uma lista de *posts* e retorna uma sublista apenas com os *posts* que satisfazem as regras do filtro passado como parâmetro. Assuma que a função `substring` indica se uma dada string é parte de uma outra string. □

```
interp :: (Eq t, Show t) => Filtro -> [FaceBookPosts t] -> [FaceBookPosts t]
interp filtro [] = []
interp (And regra1 regra2) posts = [x | x <- posts, ((member x (interp regra1 posts)) && (member x (interp regra2 posts)))]
interp (Or regra1 regra2) posts = [x | x <- posts, ((member x (interp regra1 posts)) || (member x (interp regra2 posts)))]
interp (Not regra) posts = [x | x <- posts, not (member x (interp regra posts))]
interp (Author autores) posts = [x | x <- posts, not ([y | y <- autores, (substring y (show x))] == [])]
interp (Text textos) posts = [x | x <- posts, not ([y | y <- textos, (substring y (show x))] == [])]

-- [FaceBookPost ("Pedro", "Barbie", [1]), FaceBookPost ("Marina", "ola", [2]), FaceBookPost ("Arthur", "xau", [3])]
-- [Autor: Marina. Texto: ola. Objetos: [2],Autor: Arthur. Texto: xau. Objetos: [3]]
```

Obs.: para testar basta colocar o "import Data.List;" e trocar substring por "isInfixOf"