1ª Lista de Exercícios de Paradigmas de Linguagens Computacionais
Professor: Fernando Castor
Monitor: Francisco Soares Neto
CIn-UFPE – 2010.1
Disponível desde: 01/04/2010
Entrega: 20/04/2010

A lista deverá ser respondida **em dupla**. A falha em entregar a lista até a data estipulada implicará na perda de 0,5 ponto na nota do primeiro Exercício Escolar da disciplina para os membros da dupla. Considera-se que uma lista na qual **menos que 30%** das respostas estão corretas não foi entregue. A entrega da lista com **pelo menos 70%** das questões corretamente respondidas implica em um acréscimo de 0,5 ponto na nota do primeiro Exercício Escolar da disciplina para os membros da dupla. Se **qualquer situação de cópia de respostas** for identificada, os membros **de todas as duplas envolvidas** perderão 1 ponto na nota do primeiro Exercício Escolar. O mesmo vale para respostas obtidas a partir da Internet. As respostas deverão ser entregues **exclusivamente em formato texto ASCII** (nada de .pdf, .doc, .docx ou .odt) e deverão ser enviadas para o professor (castor@cin.ufpe.br) com cópia para o monitor (xfrancisco.soares@gmail.com). Tanto podem ser organizadas em arquivos separados, um por questão (e, neste caso, deverão ser zipadas), quanto em um único arquivo texto onde a resposta de cada questão está devidamente identificada e é auto-contida (uma parte da resposta de uma questão que seja útil para responder uma outra deve estar duplicada neste última).

Implemente as seguintes funções:

1) Função 'itemN', que recebe um numero inteiro positivo *n* e uma lista e retorna o *n*-ésimo elemento da lista.
Ex.:
**Main> itemN 2 ["primeiro", "segundo", "terceiro"]*
"segundo"

2) Função 'elemento', que recebe um elemento qualquer e uma lista e verifica se o elemento está presente na lista.
Ex.:
**Main> elemento 2 [1,2,3]*
True
**Main> elemento 5 [1,2,3]*
False

3) Função 'ocorrenciasN', que recebe um elemento e uma lista, e retorna quantas vezes o elemento ocorre na lista.
Ex.:
**Main> ocorrenciasN 3 [1,2,3,4,5]*
1
**Main> ocorrenciasN 0 [0,1,0,2,0,3,0,4,0,5]*
5

4) Função 'only1', que recebe um item qualquer e uma lista e informa se há somente uma instância desse item na lista.
Ex.:
**Main> only1 1 [1,2,3,4,5]*
True
**Main> only1 123 [1,2,3,4,5]*
False
**Main>only1 3 [2,3,1,23,9,3]*
False

5) Função 'parImpar', que recebe uma lista de inteiros e retorna uma tupla cujo primeiro elemento é uma lista dos números pares da lista de entrada e o segundo é uma lista dos números ímpares da lista de entrada.
Ex.:
**Main> parImpar [1,2,3,4,5]*
([2,4],[1,3,5])

6) Função 'estaOrdenado', que recebe uma lista e retorna um Bool que indica se a lista está ordenada ou não.
Ex.:
**Main> estaOrdenado [1,2,3]*
True

7) Função 'sort', que receba uma lista e a ordene.
Ex.:
**Main> sort [1,3,5,2,9,8,7]*
[1,2,3,5,7,8,9]

8) Função 'getWord', que recebe uma String e retorna a primeira palavra dessa String – sem contar pontuação.
Ex.:
**Main> getWord "Olá, Mundo!"*
"Olá"

9) Um crivo de Eratóstenes em Haskell – uma função que receba um número inteiro positivo *n* e retorne uma lista com todos os números primos menores que *n*.
Ex.:
**Main> eratostenes 12*
[2,3,5,7,11]

Obs. A resolução desta questão não deve levar em conta questões de eficiência.

10) Função 'split', que recebe uma função *f* do tipo *(Int->Bool)* e uma lista *L* de números inteiros e retorna uma tupla onde o primeiro elemento é uma lista dos itens *i* de *L* tais que *f i* é verdadeiro e o segundo elemento é uma lista dos itens *i'* de *L* tais que *f i'* é falso.
Ex.:
**Main> split (>5) [1,2,3,4,5,6,7,8]*
([6,7,8],[1,2,3,4,5])

11) Função 'combinar', que recebe uma função *f* do tipo *((a,b)->c)* e duas listas *l1* e *l2* dos tipos *[a]* e *[b]* como argumentos e aplica *f* aos elementos das duas listas na ordem em que aparecem (Ex. se *x* é o primeiro elemento de *l1* e *y* é o primeiro de *l2,* o primeiro elemento da lista resultante será *f x y*). Note que *l1* e *l2* podem ter quantidades de elementos distintas. Neste caso, os elementos extras da lista de maior comprimento devem ser incluídos diretamente na lista resultante (Ex. se *l1* tem mais elementos e *z* é um elemento de *l1* que não tem uma contraparte em *l2*, *z* deve ser incluído na lista resultante.

Ex.:
**Main> combinar (\x,y->2*x + 3*y) [1,2,3] [1,2,3,4]*
[5,10,15,4]

12) Função 'foldi', que receba como entra uma função *f::(a->a->a)->a->[a]->a* e percorra a lista

correspondente ao seu último argumento do último para o primeiro elemento, aplicando a função correspondente ao seu primeiro argumento. O segundo argumento é o caso base da função, caso a lista seja vazia.

foldi (-) 0 [1,2,3,4,5,6,7] == 7 – (6 – (5 – (4 – (3 – (2 – (1 – 0))))))

<u>As questões seguintes são bons exemplos de possíveis questões de prova!</u>

13) Dada a função:
func a b = a++b
Informe se as seguintes aplicações dela funcionam, porque funcionam e o que seria necessário para que funcionassem, caso não funcionem.
a) *func ["Olah,"] " Mundo!"*
b) func 'O' "lah, Mundo!"
c) func "Olah, Mundo" 3
d) func "Olah, Mundo" ['!']

14) Implemente uma função que, dada uma lista como entrada, retorna uma árvore binária **balanceada**
Ex.:
**Main> list2tree [1,2,3,4,5]*
No 1 (No 2 (No 4 Nil Nil) Nil) (No 3 (No 5 Nil Nil) Nil)

Obs. Se quiser que o valor retornado pela função seja impresso pelo GHCi, ao definir o tipo algébrico para árvores, inclua no final a cláusula "deriving Show" (sem as aspas). Isso ainda não foi coberto em sala de aula, mas é o assunto da aula da próxima terça, 06/04/2010.

15) Implemente uma função que, dados uma lista de valores e um número inteiro *N* como entradas, caso seja possível, retorna uma árvore binária com profundidade menor ou igual *N*. Caso não seja possível, a função deve retornar uma árvore vazia (sem raiz).

16) Quais são os tipos das funções abaixo? Apresente uma lista de argumentos válida para uma aplicação total dessas funções, caso isso seja possível. Caso não seja, explique o porquê.

(a) *map.foldr*
(b) *foldr.map*
(c) *filter.map*
(d) *map.map*
(e) *map.foldr.map.foldr*

17) Defina, usando as construções de Haskell que você aprendeu até agora, um tipo de dados chamado Grafo, que implementa um grafo não direcionado com pesos, e uma função '*dijkstra*' que receba um valor do tipo Grafo e dois nós desse grafo e retorne o custo do menor caminho entre eles. A forma como o tipo de dados Grafo será definida fica à sua escolha.