2ª Lista de Exercícios de Paradigmas de Linguagens Computacionais
Professor: Fernando Castor
Monitores:
Paulo Barros <pbsf>,
Leonardo Brayner <lbs2>,
Cleivson Siqueira de Arruda <cleivson.tb@gmail.com>,
Irineu Moura <imlm2>,
Caio Sabino Silva <ccss2>,
CIn-UFPE – 2010.2
Disponível desde: 04/10/2010
Entrega: 15/10/2010

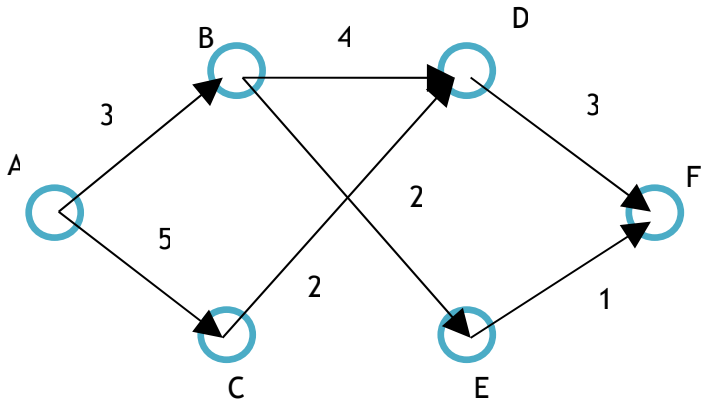
A lista deverá ser respondida **em dupla** . A falha em entregar a lista até a data estipulada implicará na perda de 0,25 ponto na **média** da disciplina para os membros da dupla. Considera-se que uma lista na qual **menos que 30%** (menos que três) das respostas estão corretas não foi entregue. A entrega da lista com **pelo menos 70%** (sete ou mais) das questões corretamente respondidas implica em um acréscimo de 0,125 ponto na média da disciplina para os membros da dupla. Se **qualquer situação de cópia de respostas** for identificada, os membros **de todas as duplas envolvidas** perderão 0,5 ponto na média da disciplina. 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 monitor responsável por sua dupla, **sem** cópia para o professor. 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).

1º) Defina a função `variancia :: [Float] -> Float`, que retorna a variância dos valores de uma lista de pontos flutuantes.

2º) Defina um tipo de dados polimórfico Set t, que representa um conjunto de elementos (isto é, sem elementos repetidos). Lembre-se que na definição do tipo polimórfico, só deve ser aceito um tipo t que se possa comparar a igualdade entre dois elementos. Após isso, defina uma função `diferencaSimetrica :: Set t -> Set t -> Set t`. A diferença simétrica de dois conjuntos A e B é o conjunto formado pelos elementos que existem só em A ou só em B. Defina também uma função `readSet :: [t] -> Set t`. Os valores recebidos por readSet serão listas que representam conjuntos válidos.

```
*Main> diferencaSimetrica (readSet [1,2,3,4]) (readSet [3,4,5])
Set [1,2,5]
```

3º) Nesta questão, você deverá definir uma função dijkstraFunction de menor distância num grafo em relação a um vértice. Tal função deve computar a menor distância de qualquer vértice em relação a um vértice escolhido. Dado o grafo abaixo, chamando a função `dijkstraFunction` escolhendo o vértice "A", obtém-se como resultado uma função que aplicada ao valor de um vértice do grafo devolve a menor distância de A ao dado nó. Ou seja, `dijkstraFunction` retorna uma função que mapeia todas as menores distâncias de um vértice.

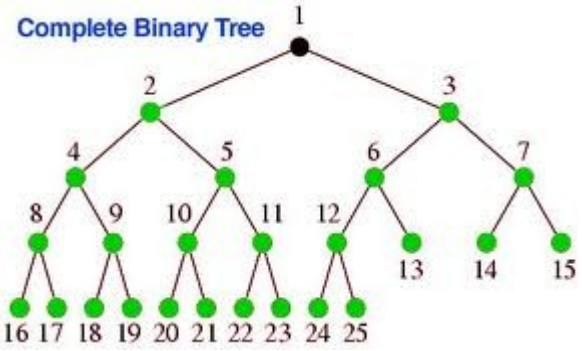


A função deve ser da forma `dijkstraFunction :: Graph t -> t -> (t -> Int)`. Para isso, deve ser definido um **tipo algébrico polimórfico** Graph que representa um grafo **direcionado** e cujas arestas têm **pesos positivos** e deve haver uma função `readGraph :: [t] -> [(t,t,Int)] -> Graph t`, que recebe o conjunto de vértices e arestas, como no exemplo abaixo:

```
*Main> (dijkstraFunction (readGraph [“A”, “B”, “C”, “D”, “E”, “F”] [(“A”, “B”, 3), (“A”, “C”, 5), (“B”, “D”, 4), (“D”, “F”, 3), (“C”, “D”, 2), (“B”, “E”, 2), (“E”, “F”, 1)]) “A”) “F”
6
*Main> (dijkstraFunction (readGraph [“A”, “B”, “C”, “D”, “E”, “F”] [(“A”, “B”, 3), (“A”, “C”, 5), (“B”, “D”, 4), (“D”, “F”, 3), (“C”, “D”, 2), (“B”, “E”, 2), (“E”, “F”, 1)]) “A”) “A”
0
```

Observação: Considere que a função resultante do `dijkstraFunction` só será aplicada a vértices alcançáveis a partir do vértice dado.

4) Defina o tipo `Tree` como foi feito em sala de aula. Defina uma função `criar_arvore_completa :: Int -> Tree a`. Essa função recebe um inteiro n e cria uma árvore binária completa com n nós. Uma árvore binária completa é definida da seguinte forma: Todos os níveis da árvore, exceto talvez o último, estão completamente preenchidos por nós e no último nível os nós ficam o mais à esquerda possível. Por exemplo, a árvore binária abaixo representa graficamente o resultado de chamada `criar_arvore_completa` com o argumento 25.



5) Defina uma função `eh_arvore_completa :: Tree a -> Bool`, que recebe uma árvore e retorna como resultado um booleano dizendo se a árvore binária é completa ou não.

6) Defina em Haskell um construtor de tipo (tipo polimórfico) Function a que modele as seguintes funções matemáticas:

- **Função constante**
  f(x) = c , onde *c* é uma constante qualquer
- **Função identidade**
  f(x) = x
- **Função soma**
  f(x) = g(x) + h(x) , onde *h* e *g* são duas funções quaisquer
- **Função subtração**
  f(x) = g(x) - h(x) , onde *h* e *g* são duas funções quaisquer
- **Função multiplicação**
  f(x) = g(x) * h(x) , onde *h* e *g* são duas funções quaisquer
- **Função divisão**
  f(x) = g(x) /h(x) , onde *h* e *g* são duas funções quaisquer
- **Função de exponenciação**
  $f(x) = e^{g(x)}$, onde g(x) é uma função qualquer
- **Função log-natural**
  f(x) = log(g(x)) , onde g(x) é uma função qualquer

Defina agora uma função ***aplicar*** que recebe um valor do tipo Float e uma Function e retorna como resultado a aplicação da Function ao número de ponto flutuante fornecido como argumento.
Defina também uma função ***derivar*** que recebe uma Function e retorna a derivada da Function (ou seja, outra Function) passada como parâmetro. Considere que somente serão passadas como parâmetros funções contínuas e diferenciáveis.

Lembrando que:

f(x) = c	->	f'(x) = 0
f(x) = x	->	f'(x) = 1
f(x) = g(x) + h(x)	->	f'(x) = g'(x) + h'(x)
f(x) = g(x) - h(x)	->	f'(x) = g'(x) - h'(x)
f(x) = g(x) * h(x)	->	f'(x) = g'(x) * h(x) + g(x)*h'(x)
f(x) = g(x) / h(x)	->	f'(x) = ((h(x)*g'(x)) - (h'(x)*g(x)))/(h(x))^2
$f(x) = e^{g(x)}$	->	$f'(x) = (e^{g(x)})*g'(x)$
f(x) = log(g(x))	->	f'(x) = (g'(x))/g(x)

7º) Considere as seguintes funções:

```
dividir :: (a -> Bool) -> [a] -> ([a], [a])
juntar :: [a] -> [b] -> [(a, b)]
map :: (a -> b) -> [a] -> [b]
```

Qual o tipo 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) juntar . (juntar [1..])
b) juntar. juntar . (juntar [1..])
c) map.dividir
d) map.map.dividir

8º) Defina uma instância de Show para Tree a. Onde deverão ser implementados os métodos da classe Show para o tipo Tree a. A impressão da árvore deve imprimir somente os elementos em pré-ordem, separados por vírgula e com um ponto após o último elemento.

*Main> show (Node 1 (Node 2 (Node 3 Nil Nil) Nil) (Node 4 Nil Nil))

"1,2,3,4."

9º) Defina a função readTreeFromFile que lê operações de um arquivo, cria uma árvore completa e manipula a árvore a partir delas. O padrão do arquivo é:

- Uma linha para a definição do tipo da árvore. O tipo deverá ser comparável para poder ser removido, imprimível e será escrito exatamente como o tipo primitivo de Haskell equivalente. Os tipos possíveis são Int, String, Char, Bool.
- Número de linhas de código.
- Linhas para manipulação da árvore:
  - add x (onde x é do mesmo tipo definido no início do arquivo)
  - remove x (onde x é do mesmo tipo definido no início do arquivo)
  - print ordem (onde ordem é pre, pos, in. Só os elementos devem ser printados. Ponto extra se eles forem separados por vírgula e com um ponto no último elemento)

Ex. de arquivo:

```
Char
8
add c
add l
add e
print in
add j
print pos
remove j
print pre
```

O resultado será:

```
"lce"
"jlec"
"cle"
```