PROGRAMAÇÃO FUNCIONAL [BCC222]

Segunda prova (2018–1)
13 de junho de 2018

Matrícula: _____

Nome: _____

Departamento de Computação
Universidade Federal de Ouro Preto
Prof. José Romildo Malaquias

Instruções

- Não se esqueça de preencher o nome e o número de matrícula na folha de respostas.
- A avaliação é individual.
- A interpretação das questões faz parte da avaliação.
- Faça as observações que achar necessárias por escrito.
- **Não é permitido o uso de computadores ou aparelho celular**. Os aparelhos celulares deverão permanecer desligados e guardados fora do alcance de suas mãos.
- Qualquer tentativa de consulta não autorizada será punida com rigor, de acordo com as regras da instituição.

Questão 1. [1 PONTO]
Qual é o resultado da compilação e avaliação da expressão a seguir?

```
filter (not . even . (+3)) [5..9]
```

a) erro de sintaxe

b) erro de tipo

c) `[8, 9, 10, 11, 12]`

d) `[False, True, False, True, False]`

e) nenhuma das respostas anteriores

Questão 2. [1 PONTO]
Qual é o resultado da compilação e avaliação da expressão que se segue?

```
case [not (1 /= 2), 5 == 5, length "pedro" /= 2] of
  [True,_,_]  -> 3
  [_,False,_] -> "fim"
  x:_         -> show x
  _           -> "não sei"
```

a) erro de sintaxe

b) erro de tipo

c) `3`

d) `"fim"`

e) `"False"`

f) `"não sei"`

g) nenhuma das respostas anteriores

Questão 3. [2 PONTOS]
Em uma aplicação que simula um caixa automático a carteira de contas é representada por uma lista de contas, onde cada conta tem um número e um histórico de movimentos (uma lista). Cada movimento tem uma data (uma string), uma descrição (abertura, saque, ou depósito), o valor movimentado, e o saldo final.

```haskell
data DescMov = Abertura | Saque | Depósito
               deriving (Eq, Ord, Show)
type Data = String
type Movimento = (Data, DescMov, Double, Double)
type Histórico = [ Movimento ]
type Conta = (Integer, Histórico)
type Carteira = [ Conta ]
```

Exemplo de carteira de contas:

```haskell
exemplo :: Carteira
exemplo = [ (10, [ ("6/6/18", Depósito,  20,  90)
                 , ("6/5/18", Saque   ,  80,  70)
                 , ("3/4/18", Depósito,  50, 150)
                 , ("1/2/18", Abertura, 100, 100)
                 ] )
          , (20, [ ("5/5/18", Depósito,  35,  50)
                 , ("9/2/18", Saque,      5,  15)
                 , ("4/2/18", Depósito,   2,  20)
                 , ("4/2/18", Depósito,  18,  18)
                 , ("3/2/18", Abertura,   0,   0)
                 ] )
          , (30, [ ("9/2/18", Saque,      8,   2)
                 , ("3/5/18", Abertura,  10,  10)
                 ] )
          ]
```

1. Escreva uma função `somaDepósitos :: Carteira -> Double` que recebe uma carteira de contas e resulta na soma de todos os depósitos feitos no banco. Use funções de ordem superior.

   Exemplo:

   ```haskell
   somaDepósitos exemplo ⇝ 125.0
   ```

   Dicas:

   - use variáveis locais para nomear valores intermediários
   - use as funções da biblioteca padrão `map`, `filter`, `concat`, `snd` e `sum`
   - expressões lambda junto com `map` e `filter` podem ser úteis

   ```haskell
   somaDepósitos ... =  ...
     where
       listaDeHistóricos = ... - seleciona os históricos de cada conta
       listaDeMovimentos = ... - todos os históricos concatenados
       listaDeDepósitos  = ... - seleciona apenas os depósitos
       listaDeValores    = ... - seleciona os valores depositados
       soma              = ... - soma dos valores depositados
   ```

2. Usando funções de ordem superior defina uma função `saldoTotal :: Carteira -> Double` para obter o saldo total depositado no banco. Considere que o histórico de cada conta já se encontra ordenado cronologicamente de maneira decrecente, de forma que o primeiro movimento (cabeça da lista) é o mais recente. Considere que o histórico nunca é vazio.

   Dicas:

   - Organize o código de maneira semelhante ao item anterior, mas para obter a soma dos saldos de todas as contas.
   - O saldo de cada conta é obtido a partir do primeiro movimento no histórico.

Questão 4. [2 PONTOS]
Considere a seguinte definição de tipo para representar uma expressão algébrica:

```haskell
data Exp = Cte Double       - contante
         | Var String       - variável
         | Som Exp Exp      - soma
         | Sub Exp Exp      - subtração
         | Mul Exp Exp      - multiplicação
         | Div Exp Exp      - divisão
         | Neg Exp          - simétrico
  deriving (Show)
```

Defina uma função `contaCte :: Exp -> Integer` que recebe uma expressão aritmética e calcula a quantidade de constantes que aparecem na expressão.

Por exemplo:

```haskell
contaCte (Cte 5.6)                                  ⇝ 1
contaCte (Som (Cte 5.6) (Var "x"))                  ⇝ 1
contaCte (Div (Som (Cte 5.6) (Cte 1.4)) (Cte 2)     ⇝ 3
contaCte (Mul (Neg (Cte 5.6)) (Som (Cte 1.4) (Cte 2)) ⇝ 3
```

Questão 5. [4 PONTOS]
Considere a seguinte definição de tipo polimórfico para árvores binárias:

```haskell
data ArvBin t = Vazia | No t (ArvBin t) (ArvBin t)
  deriving (Show)
```

Basicamente uma árvore binária pode ser de duas formas possíveis: vazia (representada pelo construtor constante `Vazia`), ou pode ser um nó (representada pelo construtor `No`) contendo uma informção e duas subárvores do mesmo tipo do nó. Exemplo:

```haskell
arvore1 = (No 5
              (No 9 Vazia Vazia)
              (No 5
                  (No 8
                      Vazia
                      (No 3 Vazia Vazia))
                  Vazia))
```

1. Escreva uma função `pesquisa` que recebe um predicado (uma função de teste, que resulta em verdadeiro ou falso) e uma árvore binária, e resulta possivelmente no primeiro elemento da árvore que passe no teste, quando a árvore é percorrida da esquerda para a direita (*pré ordem*).

   Use o construtor de tipo `Maybe` no resultado da função. Anote o tipo mais geral da sua função.

   Por exemplo:

   ```haskell
   pesquisa odd Vazia
   ⇝ Nothing

   pesquisa even (No 8 Vazia Vazia)
   ⇝ Just 8

   pesquisa ((>5) . length) (No "paulo" Vazia (No "roberto" Vazia Vazia))
   ⇝ Just "roberto"

   pesquisa (<0) arvore1
   ⇝ Nothing
   ```

2. Defina uma função `somaArv` que recebe uma árvore de números e resulta na soma de todos os elementos da árvore.

   Por exemplo:

   ```haskell
   somaArv arvore1 ⇝ 30
   ```

3. Defina uma função `arvLista` para obter a lista de todos os elementos da árvore, percorrida da esquerda para a direita (*pre-order*).