# PROGRAMAÇÃO FUNCIONAL [BCC222]

**Teste 4 (2017–2)**
11 de dezembro de 2017

Matrícula: _____

Nome: _____

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

**Questão 1. Torres de Hanoi.** Torres de Hanoi (figura 1) é um quebra-cabeça clássico com uma solução que pode ser descrita de forma recursiva.

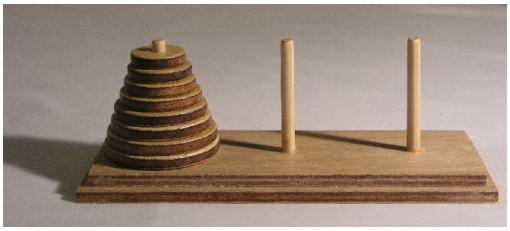

Figura 1: Torres de Hanoi.

Vários discos de tamanhos diferentes são empilhados em três cavilhas. O objetivo é obter, a partir de uma configuração inicial com todos os discos empilhados sobre a primeira cavilha, uma configuração que termina com todos os discos empilhados sobre a última cavilha, como mostrado na figura 2.

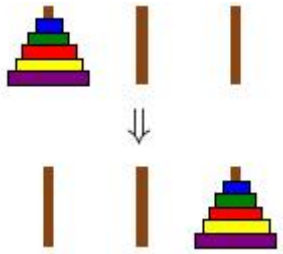

Figura 2: As torres de Hanoi.

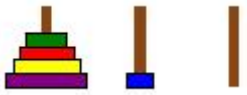

Figura 3: Um primeiro movimento válido.

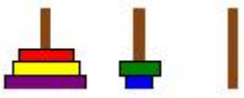

Figura 4: Uma configuração ilegal.

As únicas regras são:

1. somente um disco pode ser movido de cada vez, e
2. um disco maior nunca pode ser empilhado em cima de um menor.

Por exemplo, como o primeiro passo tudo o que você pode fazer é mover o disco menor que se encontra no topo do pino, para outro pino diferente, uma vez que apenas um disco pode ser movido de cada vez. A partir desta situação, é ilegal mover para a configuração mostrada nas figura 4, porque você não tem permissão para colocar o disco verde em cima do disco azul, que é menor.

Para mover $n$ discos (empilhados em ordem crescente de tamanho) de um pino $a$ para um pino $b$ usando um pino $c$ como armazenamento temporário:

1. mova $n-1$ discos de $a$ para $c$ usando $b$ como armazenamento temporário,
2. mova o disco no topo de $a$ para $b$, e
3. mova $n-1$ discos de $c$ para $b$ usando $a$ como armazenamento temporário.

Defina uma função `hanoi` do tipo especificado a seguir.

```
type Pino = String
type Movimento = (Pino,Pino)

hanoi :: Integer
      -> Pino
      -> Pino
      -> Pino
      -> [Movimento]
```

Dados o número de discos e os nomes dos pinos para os três pinos, `hanoi` deve resultar na lista dos movimentos que devem ser realizados para mover a pilha de discos do primeiro pino para o segundo pino, usando o terceiro pino como armazenamento temporário.

Note que uma declaração de tipo usando a palavra chave `type`, como em `type Pino = String`, define um *sinônimo de tipo*. Neste caso `Pino` é declarado como um sinônimo para `String`, e os dois nomes de tipo `Pino` e `String` podem agora ser usados um no lugar do outro. Nomes descritivos para tipos dados desta maneira podem ser usados para dar nomes mais curtos para tipos complicados, ou para simplesmente ajudar na documentação, como foi feito aqui.

Exemplo:

```
hanoi 2 "a" "b" "c"
⇝
[("a","c"), ("a","b"), ("c","b")]
```

Este resultado significa que para transferir dois discos do pino $a$ para o pino $b$ usando o pino $c$ como armazenamento temporário, deve-se mover um disco de $a$ para $c$, um disco de $a$ para $b$, e um disco de $c$ para $b$.

**Questão 2. Conversão para string com segurança.** A função

```
readMaybe :: Read a => String -> Maybe a
```

definida no módulo `Text.Read`, converte uma string em um valor do tipo `a` (que deve ser instância da classe `Read`). A conversão sucede se e somente se há exatamente um resultado válido.

Faça um programa que leia uma temperatura na escala Celsius e calcula e exibe a temperatura correspondente na escala Fahrenheit. O programa deve verificar se a entrada de dados sucede ou falha. Uma nova entrada deve ser feita enquanto a leitura for inválida.

**Questão 3. Um tipo para os números naturais.** Vamos implementar um tipo para representar simbolicamente os números naturais.

1. Defina um tipo algébrico `Nat` para representar números naturais. Um número natural pode ser:

   - zero, ou
   - positivo, sendo neste caso o sucessor de outro número natural

   O seu tipo deve ter dois construtores de dados: `Zero`, um construtor constante, para representar o valor zero, e `Suc`, um construtor de aridade um, para representar um número positivo.

   Observe que o tipo `Nat` deve ser recursivo, já que ele deverá ser usado em sua própria definição.

   Use derivação automática da classe `Show`.

2. Defina as variávies `um`, `dois` e `tres` do tipo `Nat` cujos valores são os números naturais 1, 2 e 3, respectivamente.

3. Defina a função

   ```
   nat2integer :: Nat -> Integer
   ```

   que converte um número natural em um número inteiro. Faça uma definição recursiva onde o caso base corresponde 0, e o caso recursivo corresponde aos números positivos.

4. Defina a função

   ```
   integer2nat :: Integer -> Nat
   ```

   que converte um número inteiro em um número natural.

5. Defina a função

   ```
   natLt :: Nat -> Nat -> Bool
   ```

   que recebe dois números naturais e verifica se o primeiro é menor que o segundo.

6. Defina a função

   ```
   natAdd :: Nat -> Nat -> Nat
   ```

   que recebe dois números naturais e resulta na soma dos números. A função deve ser recursiva no segundo argumento.

7. Defina a função

   ```
   natSub :: Nat -> Nat -> Nat
   ```

   que recebe dois números naturais e resulta na diferença dos números. A função deve ser recursiva no segundo argumento.

8. Defina a função

   ```
   natMul :: Nat -> Nat -> Nat
   ```

   que recebe dois números naturais e resulta no produto dos números. A função deve ser recursiva no segundo argumento.

9. Defina as funções

   ```
   natDiv :: Nat -> Nat -> Nat
   ```

   e

   ```
   natMod :: Nat -> Nat -> Nat
   ```

   que recebem dois números naturais e resultam no quociente e no resto dos números, respectivamente.