# Leitor de Código de Barras

BCC22 Programação Funcional — Aula Prática 06

Lucília Camarão de Figueiredo

A prática deve ser realizada antes da aula prevista para a discussão da mesma. Verifique a data prevista para a discussão de cada aula prática no calendário do curso disponível em `www.dcc.ufmg.br/~lucilia/cursos/func`. Traga para a aula de discussão o código que você implementou. A discussão não será eficaz se você não tentar fazer os exercícios previamente.
Você pode trabalhar em conjunto com outros colegas, mas deve entender o que está fazendo, pois não irá contar com a ajuda de colegas durante os exames.
A realização das práticas e presença nas aulas são obrigatórias e valem pontos na disciplina.

## 1 Leitor de Código de Barras

Nesta prática você vai aprender um pouco sobre a implementação de um leitor de código de barras. Infelizmente não vamos poder contar com um scanner de código de barras, vamos apenas implementar a pesquisa por um item com um dado código de barras em um banco de dados. Os dados do banco de dados serão lidos de um arquivo e armazenados em duas diferentes estruturas de dados, para testar qual delas provê melhor velocidade para acesso aos dados.

Os arquivos que você vai usar são `pratica06.hs`, `KeymapList.hs`, e `KeymapTree.hs`. Além desses, também o arquivo que contém os nossos dados: `database.csv` (`csv` significa 'comma-separated values').

Comece abrindo o arquivo `KeymapList.hs` em um editor de textos. Esse arquivo define um tipo de dados abstrato para representar um mapeamento (ou tabela) chave-valor (*keymap*). O arquivo começa com a seguinte cláusula:

```
module KeymapList ( Keymap, size, ...  )  where
```

Essa declaração significa que o arquivo `KeymapList.hs` é um módulo que pode ser usado em outros arquivos de programas Haskell, da mesma forma que os módulos `Data.Char` e `Test.QuickCheck` que você já usou anteriormente. As funções e construtores relacionadas entre parênteses (`Keymap`, `size`, etc.) são exportados pelo módulo e podem ser importados em outro arquivo qualquer.

Vamos agora dar uma olhada na declaração do tipo `Keymap`:

```
newtype Eq k => Keymap k a = K [(k,a)]
```

Essa declaração define o tipo polimórfico `Keymap`. O primeiro argumento, `k`, é o tipo das chaves e o segundo argumento, `a`, é o tipo dos valores da tabela. Por exemplo, uma tabela pode ter tipo `Keymap Int String`, indicando que é um mapeamento de chaves de tipo `Int` para dados de tipo `String`.

Nessa declaração, a restrição `Eq k =>` indica que a o tipo da chave, `k`, deve ser um tipo para o qual esteja definido o teste de igualdade. Em outras palavras, se desejamos valores de um determinado tipo como chave, esse tipo deve ser tal que a função `(==)` deve estar definida para expressões desse tipo (caso contrário não teríamos como encontrar a chave na tabela).

A definição propriamente dita — `K [(k,a)]` indica que a tabela é uma lista de pares *chave-valor*. O construtor `K` é simplesmente um invólucro, que evita que valores do tipo `Keymap` possam ser diretamente manipulados por meio de funções definidas sobre listas. A razão disso é que queremos restringir as operações que podem ser feitas sobre a nossa tabela, de maneira a evitar confusão e possíveis erros.

Vamos agora dar uma olhada nas funções definidas nesse módulo. Note que todos os tipos das funções incluem a restrição `Eq k =>`, uma vez que essa restrição ocorre no tipo de dado `Keymap`.

- `size :: Eq k => Keymap k a -> Int`
  Retorna o número de entradas na tabela `Keymap`.
- `get :: Eq k => k -> Keymap k a -> Maybe a`
  Pesquisa na tabela pela chave dada e retorna o valor associado a essa chave. Nessa função, os valores de chaves da tabela são comparados com a chave dada por meio da função `(==)`; por isso é requerida a restrição `Eq k =>` no tipo da função. O valor retornado não é um valor de tipo `a`, mas sim do tipo `Maybe a`, pois pode ocorrer que a chave dada não seja encontrada na tabela. O tipo `Maybe a` será explicado mais adiante.
- `set :: Eq k => k -> a -> Keymap k a -> Keymap k a`
  Dados uma chave e um valor, insere o par chave-valor na tabela. Caso a chave já se encontre na tabela, o valor associado a essa chave é substituído pelo valor dado como argumento; caso contrário a chave é inserida na tabela, juntamente com o valor associado.
- `del :: Eq k => k -> Keymap k a -> Keymap k a`
  Essa função exclui da tabela a entrada correspondente à chave dada como argumento.
- `select :: Eq k => (a -> Bool) -> Keymap k a -> Keymap k a`
  Essa função retorna a tabela obtida restringindo-se a tabela dada como segundo argumento apenas aos valores que satisfazem o predicado dado como primeiro argumento.
- `tolist :: Eq k => Keymap k a -> [(k,a)]`
  Essa função exporta a tabela como uma lista de pares chave-valor.
- `fromList :: Eq k => [(k,a)] -> Keymap k a`
  Essa função constrói uma tabela a partir de uma lista de pares chave-valor.

Agora que já sabemos como é uma tabela `KeymapList`, podemos começar a trabalhar na `pratica06.hs`. Logo abaixo do início desse arquivo você verá as seguintes declarações:

```
import KeymapList
type Barcode = String
type Product = String
type Unit = String

type Item = (Product,Unit)

type Catalog = Keymap Barcode Item
```

Primeiro, o programa importa o módulo `KeymapList`. Em seguida, declara os tipos sinônimos `Barcode`, `Product`, e `Unit`, que são strings, e o tipo sinônimo `Item`, que é um par `(Product,Unit)`. Finalmente, declara o tipo sinônimo `Catalog` para se referir a uma tabela `Keymap` que associa chaves do tipo `Barcodes` com valores do tipo `Item`. Abaixo disso, você encontra a definição de um pequeno banco de dados de teste.

**Exercícios**

1. Antes de trabalhar com o banco de dados, você precisa implementar funções que permitam visualizar os dados armazenados. Se você tentar avaliar `testDB` ou `show testDB` no prompt do GHCi, verá que nada será impresso. Ao invés disso, você deve usar o seguinte comando:

   ```
   Main> toList testDB
   ```

   Entretanto, o formato de saída é muito pouco legível. Nos exercícios a seguir vamos implementar uma função para exibir os dados do banco de dados em um formato mais adequado para visualização.

   (a) Escreva a função `longestProduct`, que encontra na tabela o produto (ou item) com nome mais longo.

   (b) Escreva a função `formatLine` que, dado um número (que especifica o comprimento do nome do produto no formato), imprime o código de barra, o nome do produto e a informação de unidade, em uma única linha, separando essas informações por pontos (se o comprimento real do nome do produto for menor do que o especificado para o formato, a diferença deve ser preenchida com pontos). Por exemplo:

   ```
   Main> formatLine 7 ("0001",("product","unit"))
   "0001...product...unit"
   Main> formatLine 7 ("0002",("thing","unknown"))
   "0002...thing.....unknown"
   ```

   Você pode supor que o nome de um produto nunca é maior do que o comprimento especificado como argumento.

   (c) Escreva a função `printCatalog` que imprime o catálogo `Catalog` formatado. Você vai precisar usar a função `toList` (definida no módulo `KeymapList`). Teste sua função escrevendo no prompt do GHCi:

   ```
   Main> putStr (showCatalog testDB)
   ```

2. Agora vamos começar a usar a função `get`. Primeiro, avalie a seguinte expressão:

   ```
   Main> get "9780201342758"testDB
   ```

   Você verá que a resposta desejada é precedida pelo construtor `Just`. Agora tente:

   ```
   Main> get "000"testDB
   ```

   Qual é a resposta agora? O tipo de dado `Maybe a` é definido do seguinte modo:

   ```
   Maybe a = Nothing
           | Just a
   ```

   (a) Qual é o tipo do valor retornado pelao função `get`? Quais são os possíveis valores retornados por essa função, para o pequeno banco de dados de teste dados definido no programa? (*dica*: são cinco possíveis valores).

   (b) Escreva a função `getItems` que, dada uma lista de códigos de barra, retorna a lista dos itens correspondentes. Teste seu código avaliando a seguinte expressão:

   ```
   Main> getItems ["0001","9780201342758","0003"] testDB
   ```

   O resultado deve ser uma lista contendo apenas um item, correspondente ao livro texto desse curso. Para definir essa função você provavelmente vai querer definir funções auxiliares. (*Dica*: quando encontrar um item, construa uma lista que contém apenas ele próprio; caso você não encontre, retorne uma lista vazia; então "concatene" todas as listas resultantes em uma única lista.)

## 2 O Banco de Dados Real

Seu arquivo `pratica06.hs` faz uso de algumas funções que ainda não foram explicadas. A função `makeDB` lê o arquivo `database.csv`, armazenando-o na mamória, na forma de uma tabela `Keymap`. Para ler o banco de dados para a memória, basta chamar a função `makeDB` no prompt do CHCi (isso pode demorar alguns segundos):

```
Main> makeDB
```

O banco de dados é armazenado na variável `theDB :: Catalog`, e permanecerá na memória até que você recarregue o seu programa (ou carregue um novo programa, ou termine o GHCi). O banco de dados é bastante grande, por isso não é uma boa idéia tentar imprimi-lo na tela. Mas você pode perguntar qual é o tamanho do BD:

```
Main> size theDB
```

Outra função disponível é a função `getSample`, que retorna um código de barras aleatório; tente:

```
Main> getSample
```

Para obter o item associado ao código de barras retornado pelo comando anterior, você pode usar:

```
*Main> get it theDB
```

('it' é uma variável especial do GHCi, que refere-se ao valor retornado pela última expressão avaliada).

Vamos agora testar quão eficiente é a nossa implementação da tabela dada por `KeymapList`. Primeiro, precisamos ativar o recurso de medida de tempo de execução de programas do GHCi. Digite o seguinte no prompt do GHCi:

```
Main> :set +s
```

Pode parecer que não mudou nada, mas a partir de agora o GHCi vai prover informação sobre o tempo de execução (e uso de memória) para cada expressão submetida para avaliação.

**Exercícios**

4. (a) Recarregue seu arquivo e carregue o banco de dados novamente (usando `makeDB`). Anote quanto tempo isso demorou.
   (b) Selecione pelo menos 10 códigos de barra (usando getSample) e use `get` para obter o item associado a cada um desses códigos. Anote quanto tempo é gasto para encontrar cada item. Calcule o tempo médio gasto para recuperar um item.
   (c) Os tempos que você obteve nos testes acima provavelmente são diferentes para cada item. Qual é a razão disso? Se o banco de dados fosse 2 vezes maior, quanto tempo, em média você acha que seria necessário para recuperar um item? Quantos itens da tabela `KeymapList` a função `get` tem que inspecionar antes de encontrar o item desejado, caso ele seja o último item da tabela?

## 3 Tabelas implementadas como árvores

Nessa parte do exercício vamos implementar a tabela de itens de forma diferente. Ao invés de implementá-la como uma lista, tal como no arquivo `KeymapList.hs`, vamos implementá-la como uma árvore. Abra o arquivo

`KeymapTree.hs`. Esse arquivo contém uma nova declaração para o tipo de dado `Keymap`, assim como esqueletos de definições das mesmas funções existentes no arquivo `KeymapList.hs` (mais duas funções extras: `depth` e `merge`).

Você vai implementar no arquivo `KeymapTree.hs` as mesmas funções do arquivo `KeymapList.hs`. Do ponto de vista do módulo que usa essas funções elas parecerão as mesmas; internamente, entretanto, elas serão diferentes, e também terão desempenho diferente.

A idéia da implementação da tabela como uma árvore é explicada na seção 16.7 (página 315) do livro texto adotado no curso. Basicamente, os dados são armazenados nos nodos da árvore. O ramo à esquerda de cada nodo $x$ armazena apenas dados cujas chaves são menores do que a chave contida em $x$; o ramo à direita de $x$ armazena apenas dados cujas chaves são maiores do que a chave contida em $x$.

Primeiro vamos dar uma olhada na declaração do tipo de dado `Keymap`:

```
data Ord k => Keymap k a = Node k a (Keymap k a) (Keymap k a)
                         | Leaf
```

Esse tipo de dados define um mapeapento entre chaves, de tipo `k`, e valores, de tipo `a`. Para que as chaves possam ser ordenadas, é necessária a restrição `Ord k =>`. Isso significa que as chaves usadas em um mapeamento apenas podem ser de um tipo sobre o qual estejam definidas as funções `(==)`, `(>=)` e `(<=)`.

Os construtores de valores do tipo `Keymap` são

```
Node k a (Keymap k a) (Keymap k a)
Leaf
```

O segundo construtor (`Leaf`) representa uma folha da árvore, assim como o construtor de lista vazia `[]` na implementação anterior `Leaf` não armazena nenhum dado. O primeiro construtor `Node` faz todo o trabalho. Ele combina:

- uma chave, de tipo `k`
- o valor correspondente, de tipo `a`
- a subárvore esquerda, que é uma árvore de tipo `Keymap k a`
- a subárvore direita, também uma árvore de tipo `Keymap k a`

Quando construímos uma tabela na forma de uma árvore, devemos garantir que a árvore permaneça ordenada. Isso é, para cada nodo, com uma determinada chave, todas as chaves da subárvore esquerda devem ser menores do que essa chave, e todos as chaves da subárvore direita devem ser maiores do que essa chave. Para garantir essa propriedade, é necessário assegurar que o usuário da tabela `Keymap` apenas tenha acesso a ela por meio de funções "confiáveis".

**Exercícios**

5. Modifique o arquivo `pratica06.hs` de maneira a importar a nova implementação de tabelas, na forma de árvore; ou seja, substitua a linha:

   ```
   import KeymapList
   ```

   por

```
import KeymapTree
```

Recarregue novamente o arquivo `pratica06.hs` no GHCi. O que você acha que deveria ser retornado pela avaliação da expressão:

```
size ( Node "0001just some item"Leaf Leaf )
```

Avalie essa expressão usando o GHCi. Qual é o resultado? O que acontece é que, como os construtores `Node` e `Leaf` não são exportados, outros módulos não podem definir funções sobre árvores.

Vamos agora completar as definições das funções especificadas no módulo `KeymapTree.hs`.

**Exercícios**

6. (a) Dê uma olhada no código da função `size`. Como ela funciona? Como podemos fazer recursão sobre árvores?
   (b) Defina a função `depth`, que toma uma árvore como argumento e retorna a profundidade máxima dessa árvore, isso é, retorna o maior caminho desde a raiz até uma folha da árvore. Uma folha deve ter profundidade 0.
   (c) Carregue o arquivo `KeymapTree.hs` no GHCi e teste as funções `size` e `depth` para a pequena árvore `testTree` definida no arquivo (os resultados devem ser `size 4` e `depth 3`).
7. Defina a função `toList`, que toma como argumento uma árvore e retorna a lista de pares (chave, valor) correspondente. Tente também implementar uma versão dessa função que retorna a lista ordenada por chave.
8. Dê uma olhada na definição da função `set`. Note que é definida uma função auxiliar para fazer a recursão, de modo a evitar repetir as váriáveis `key` e `value` frequentemente na definição.
   (a) Explique o que a função `f` faz quando ela encontra uma folha.
   (b) Explique o que a função `f` faz quando ela encontra um nodo. O que acontece se ela encontra a chave que está procurando? Se ela não encontra a chave, onde a pesquisa continua?
9. Complete a função `get`. Lembre-se que o valor retornado deve ser de tipo `Maybe a`. Quando a função deve retornar `Nothing` e quando ela deve retornar `Just` um valor? Teste sua função sobre a árvore de teste `testTree` e em seguida use a propriedade `prop_set_get` para verificar sua função com o `QuickCheck`.
10. Defina a função `fromList` que converte uma lista em árvore. Você deve usar a função `set` para adicionar cada elemento na árvore. Como deve ser a árvore inicial? Você pode usar recursão sobre a lista de entrada, mas também pode usar `foldr` e `uncurry` (você deve tentar isso, pois é mais conciso e elegante). Use a propriedade `prop_toList_fromList` para testar suas soluções. Se sua função `toList` retorna listas ordenadas, então você pode também testar sua função `fromList` usando a propriedade `prop_toList_fromList_sorted`.

Agora você já definiu funções suficientes sobre `KeymapTree` para começar a avaliar o desempenho dessa implementação.

**Exercícios**

11. Salve o arquivo `KeymapTree.hs` e abra o arquivo `pratica06.hs`

(a) Carregue novamente o banco de dados, avaliando a expressão `makeDB` no prompt do GHCi. Dessa vez o banco de dados será armazenado na forma de uma árvore. Quanto tempo leva essa operação?

(b) Selecione pelo menos 10 exemplos de chave para testar quão rápido opera a sua função `get`. Lembre-se:

```
Main> getSample
retorna um código de barra aleatório
Main> get it theDB
pesquisa o item associado ao código de barras retornado.
```

(c) Quantos códigos de barra a sua função `get` precisa inspecionar, no pior caso, até encontrar no banco de dados a chave desejada?

# 4 Exercícios adicionais

11. Observe a definição da função `merge`. Você sabe para que serve o símbolo especial `_`? Pode adivinhar para que ele serve pela maneira como é usado nessa definição? Explique como a função `merge` funciona.

12. Defina a função `del`, que toma como argumentos uma chave e uma árvore e retona uma árvore idêntica à que foi dada como argumento, exceto que a entrada correspondente à chave dada é deletada. Você vai precisar da função merge para implementar a função `del`.

13. Defina a função `select`, que toma como argumentos um predicado e uma árvore e retorna uma nova árvore, que contém apenas as entradas da árvore original que satisfazem o predicado dado.