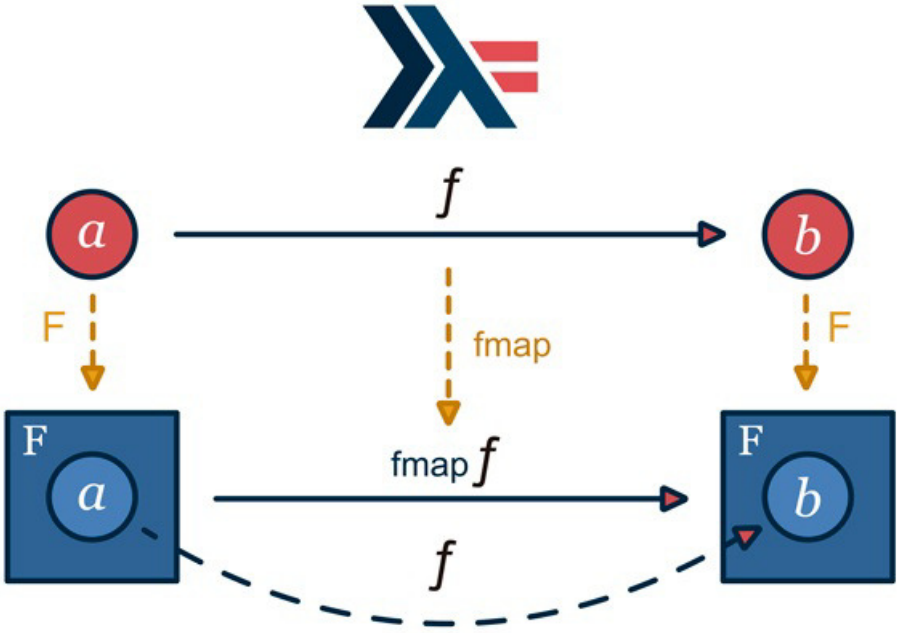
Haskell

Uma introdução à programação funcional



Casa do Código

ALEXANDRE GARCIA DE OLIVEIRA

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-5519-273-9

EPUB: 978-85-5519-274-6

MOBI: 978-85-5519-275-3

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

AGRADECIMENTOS

A Deus, pela força e perseverança que sempre tive. À minha família, pelo suporte emocional. À minha namorada, Rosanne, por me aguentar sempre.

Aos meus professores da USP e da Fatec, que contribuíram para eu chegar até aqui.

Ao pessoal da Romefeller (http:\romefeller.io), por me fazerem crescer na linguagem Haskell.

Aos meus alunos que contribuíram de alguma forma, com suas dúvidas e/ou orientação de TCCs.

À editora, Vivian Matsui, por tirar minhas dúvidas nas horas mais inoportunas.

SOBRE O AUTOR

O autor é formado Técnologo em Processamento de Dados na Faculdade de Tecnologia da Baixada Santista Rubens Lara (FATEC-RL) em 2004, e Bacharel em Matemática pelo Instituto de Matemática e Estatística da Universidade de São Paulo em 2012, onde também se formou Mestre em Matemática Aplicada em 2015. Leciona há 7 anos no Centro Estadual de Educação Tecnológica Paula Souza (CEETEPS), sendo há 5 anos pelas Fatecs, onde passou pelos campos de Santos, Praia Grande e São Caetano do Sul.

Programa em Haskell há cerca de 4 anos e conheceu a linguagem através de um TCC orientado na Fatec de São Caetano. Ele também é apaixonado pelo Santos Futebol Clube, Álgebra, Final Fantasy 7, viagens de navio e algumas séries e animes.

PREFÁCIO

Haskell é uma linguagem de programação que traz novas formas de se escrever programas corretos, eficientes e fáceis de manter. Com um modelo mental novo e uma caixa de ferramentas de programação extremamente moderna, essa linguagem pura e funcional nos dá insights valiosos para a construção de aplicações, mesmo quando precisamos deles em outras tecnologias, aplicados em outros contextos.

Este livro foi escrito como uma introdução à linguagem Haskell, para leitores já familiares com ao menos uma linguagem de programação imperativa e dispostos a dedicar tempo para entender o que a programação puramente funcional tem a nos oferecer. Interessante e divertida, a linguagem é única em ser funcional, pura, estaticamente tipada, com avaliação preguiçosa por padrão e com versões compiladas e interpretadas.

Com a crescente necessidade da indústria em escrever software altamente concorrente e paralelo, e capaz de ser jogado na nuvem - software que precisa, sem grandes mudanças, suportar um ou N contextos de execução paralelos para escalar -, o modelo de programação sem o uso implícito de estado está em alta. Esse modelo, aqui tomando a forma de programação funcional ou declarativa, nos incentiva a modularizar as menores unidades possíveis da computação: funções.

Usando a composição de declarações do que uma computação é, em vez de sequências de instruções de como uma computação ocorre, quebramos problemas em partes mais determinísticas.

Assim, em vários casos, ganhamos o suporte ao paralelismo e concorrência de graça.

Além disso, nossos programas são cada vez maiores e mais complexos, e o trabalho de manutenção de programas está maior e mais suscetível a falhas humanas. Por ser puro e estaticamente tipado, Haskell nos permite estabelecer garantias teóricas (a partir de provas e/ou modelos matemáticos expressos no código, propriedades de como expressões são avaliadas) e estáticas (a partir de mais metadados no seu rico sistema de tipos, programas "finais" sem casos não tratados).

Haskell é muito usado para a pesquisa em linguagens de programação e tem um dos sistemas de tipos mais avançados disponíveis. A promessa é a de entregar programas sólidos mais rápido ao mercado, mesmo que um primeiro protótipo demore mais a ser escrito.

O professor Alexandre convida-os a descobrir a sintaxe da linguagem Haskell, um pouco do ethos da programação funcional e alguns dos conceitos formais baseados na teoria das categorias que nos entregam essas garantias e possibilidades, permeando essa tecnologia. E ele faz isso com a iniciativa improvável e bem-sucedida de dois anos de aulas na FATEC-RL, ensinando seus alunos a beleza e aplicação do paradigma puramente funcional e do Haskell.

Este livro é um primeiro pé em um longo caminho, no qual eu ainda tenho muito a percorrer; uma nova empreitada na literatura de linguagens de programação brasileira que espero ser tão útil para vocês quanto é animadora para mim.

Pedro Tacla Yamada - Colaborador na HaskellBR, e desenvolvedor e consultor na Beijaflor Software.

INTRODUÇÃO

Este livro destina-se a alunos interessados em aprender sobre este paradigma que vem sendo adotado pelo mercado cada vez mais. Eu geralmente não recomendo um curso específico como pré-requisito, porém um semestre de lógica de programação e estruturas de dados não machucam, e servem para entender um ou dois exemplos do livro. Um aluno que tem facilidade com matemática e que tenha paixão pelo assunto consegue seguir o livro sem ter feito um curso formal nos assuntos indicados também.

O livro constrói o conhecimento na linguagem Haskell, começando do zero, desde a sua instalação até o conceito de Mônadas que é o ponto alto do livro. Ele possui nove capítulos mais apêndice e referências, e foi escrito com base nas notas de aula do meu curso de Programação Funcional, na Faculdade de Tecnologia da Baixada Santista Rubens Lara (FATEC-RL). Depois de quase dois anos ministrando esta disciplina e notado a dificuldade dos alunos de achar material em nossa língua mãe, eis que surgiu a ideia do presente trabalho.

O primeiro capítulo aborda questões gerais da linguagem e do paradigma. No segundo, vemos a instalação do ambiente, alguns exemplos preliminares, manipulação de listas e tuplas. No terceiro, introduzimos os tipos (um dos maiores pontos fortes da linguagem) e suas manipulações.

Já no quarto, vemos como a linguagem trata as funções através dos conceitos de **lambdas**, **currying** e funções de alta ordem. No

quinto, alavancamos o poder dos tipos usando o conceito de polimorfismo paramétrico, classes de tipos e vemos um exemplo muito especial: os monoides.

No sexto, há uma breve introdução informal sobre Teoria das Categorias. O sétimo explora os funtores, um dos conceitos-chave para o entendimento do assunto a ser abordado no oitavo capítulo: as mônadas. Finalmente, no último capítulo, são dados vários exemplos práticos de mônadas através do `IO`, que é a representação da computação com efeitos aqui no Haskell. Confira ainda o apêndice com todo o código do miniprojeto e referências para seguir seus estudos.

Sumário

1 Programação funcional **1**

1.1 Linguagem Haskell 2

1.2 Haskell na web 4

1.3 Conclusão 5

2 # Primeiros exemplos **7**

2.1 Primeiro contato com os tipos de dados e funções 9

2.2 Operação com listas 11

2.3 Compreensão de listas 15

2.4 Tuplas 16

2.5 Exercícios 17

2.6 Conclusão 18

3 Declarando novos tipos de dados **20**

3.1 Pattern matching 21

3.2 Campos de um construtor 26

3.3 Record syntax 28

3.4 Miniprojeto: RH de uma empresa de TI 29

3.5 Exercícios 32

3.6 Conclusão 37

4 Um pouco mais sobre funções 38

4.1 Lambdas 38

4.2 Funções de alta ordem 39

4.3 Currying 41

4.4 Exemplos de funções de alta ordem 42

4.5 Sintaxe em funções 52

4.6 Recursão 53

4.7 Miniprojeto: RH de uma empresa de TI 56

4.8 Exercícios 58

4.9 Conclusão 59

5 Polimorfismo paramétrico 61

5.1 Tipos com parâmetros 61

5.2 Restrição de tipos em funções 65

5.3 Classes de tipos 66

5.4 Outras classes 70

5.5 Monoides 76

5.6 Miniprojeto: trabalhando com parsers 81

5.7 Exercícios 83

5.8 Conclusão 86

6 Teoria das Categorias 87

6.1 Categorias 87

6.2 Noção matemática de funtor 92

6.3 Função identidade em Haskell 93

6.4 Conclusão 95

7 Funtores **96**

7.1 Funtor Maybe 98

7.2 Criando seu funtor 100

7.3 Funtores Aplicativos 101

7.4 Funtores Contravariantes 105

7.5 Miniprojeto: continuação usando funtores 110

7.6 Exercícios 111

7.7 Conclusão 112

8 Mônadas **114**

8.1 Transformações naturais 115

8.2 Definição 116

8.3 Notação DO 122

8.4 A mônada [] 125

8.5 Exercícios 126

8.6 Conclusão 127

9 Mônada IO **129**

9.1 Compilando um programa "Olá Mundo" 130

9.2 Exemplos práticos 132

9.3 Manipulando arquivos 139

9.4 Miniprojeto final 142

9.5 Exercícios 152

9.6 Conclusão 154

10 Apêndice **155**

11 Referências **160**

Versão: 20.8.29

Capítulo 1

PROGRAMAÇÃO FUNCIONAL

A programação funcional é um paradigma de programação que trata apenas de aplicação de funções matemáticas, evitando alteração de estado e mutabilidade de dados. Ou seja, assim que uma variável é alocada na memória e um valor é associado a este local, tal valor não pode ser mudado e sim transformado por uma aplicação de função.

Uma das características da programação funcional é o estilo de estrutura declarativa que se opõe ao estilo imperativo. Nela não há descrição de estruturas de controle, e seu estilo descreve o que o programa faz (*what to do*) e não como ele deve ser feito (*how to do*). O uso desse estilo visa minimizar os impactos dos efeitos externos, **side effects**. Denomina-se este conceito como funções de ordem superior, ou forma funcional.

A fundamentação matemática rigorosa da programação funcional nos permite escrever testes de software mais precisos e baseados em propriedades matemáticas que permitem escrever uma prova matemática de modo a validar um código.

Em uma linguagem funcional, funções são tratadas como

valores comuns, sendo que existe a possibilidade de: uma função ser assinalada a uma constante localmente, ser passada via parâmetro ou até mesmo ser retornada por uma outra função.

Atualmente, algumas das linguagens principais do mercado (por exemplo, Java e C#) adotaram recentemente algumas ferramentas da programação funcional, como o uso de lambdas – que é um tratamento de um método como se fosse um valor e este poderá ser passado via parâmetro ou retornado como foi descrito anteriormente. Linguagens como JavaScript, Python, Ruby e muitas outras, hoje em dia, possuem algum suporte para este paradigma, e isto mostra o crescente interesse das comunidades de desenvolvedores em torno disso.

Muitas linguagens que suportam apenas o paradigma funcional estão crescendo no mercado. Entre elas podemos citar: Haskell, Erlang, Clojure, Scala, OCaml e algumas linguagens que compilam para JavaScript, como ClojureScript, Elm, PureScript, entre outras.

A empresa RedMonk elaborou um ranking no começo de 2016 baseando-se nas atividades das plataformas GitHub e Stackoverflow. Neste ranking, as linguagens totalmente funcionais Scala, Haskell e Clojure aparecem respectivamente em 14ª, 15ª e 19ª colocações, um bom índice para um paradigma que há 10 anos era usado apenas em ambientes acadêmicos (PEYTON JONES, 2009).

1.1 LINGUAGEM HASKELL

Esta linguagem começou em 1987, de acordo com Peyton Jones (2007) durante uma conferência de programação funcional.

Neste evento, um comitê de intelectuais se formou para criar um novo padrão de programação funcional.

Além das características do paradigma funcional descritas anteriormente, outros fatores presentes são: *laziness*, o fato de ser uma linguagem de programação funcional pura e ser estaticamente tipada (PEYTON JONES, 2009).

O conceito de laziness (ou processamento preguiçoso) é o ato de a linguagem só calcular expressões quando realmente for necessário (HUGHES, 1990). Isto evita alguns processamentos desnecessários. Por exemplo, a função `++` em Haskell significa concatenação de listas. Se tivermos a seguinte expressão:

```
[3,6,7,3*10^89,0] ++ [-1, 9]
```

Ela produzirá a lista `[3,6,7,3*10^89,0, -1, 9]` , sem precisar calcular a expressão `3*10^89` , economizando tempo de processamento, neste caso. Em Hughes (1990) e Peyton Jones (2010) , é possível ver que, antigamente, o conceito de computação preguiçosa e efeitos externos (leitura e escrita de arquivos, por exemplo) não poderiam coexistir.

"We have described lazy evaluation in the context of functional languages, but surely so useful a feature should be added to nonfunctional languages - or should it? Can lazy evaluation and side-effects coexist? Unfortunately, they cannot: Adding lazy evaluation to an imperative notation is not actually impossible, but the combination would make the programmer's life harder, rather than easier. Because lazy evaluation's power depends on the programmer giving up any direct control over the order in which the parts of a program are executed, it would make programming with side effects rather difficult, because predicting in what order -or even

whether- they might take place would require knowing a lot about the context in which they are embedded" (HUGHES, 1990).

Isso quer dizer que uma lista de ações com efeito externo, por exemplo, printar um caractere na tela não teria uma sequência dos comandos, aparente tornando impossível a existência de efeitos externos na linguagem, e assim condenando-a a viver apenas em ambientes acadêmicos e tornando impossível o desenvolvimento de aplicações para o mercado de trabalho.

Felizmente, como vemos em Peyton Jones (2010), o conceito de Mônadas (ou **Monads**) tornou possível o trabalho conjunto desses dois conceitos e ainda recebe-se de graça um belo formalismo matemático deste conceito, trazido à tona baseando-se na Teoria de Categorias.

A tipagem estática do Haskell é uma poderosa ferramenta da linguagem, permitindo que se possa usar o conceito de *type-safety* que nos permite controlar erros de programação oriundos de conversões implícitas de tipos, como é comumente visto em outras linguagens (O'SULLIVAN et al., 2012).

O conceito de programação funcional pura nos diz que toda função aqui é pura, ou seja, deve ter o mesmo retorno a partir de um mesmo argumento. Isso indica que não há mudança em variáveis e pontos globais de acesso. Para se manter pura, a linguagem usa conceitos matemáticos, vistos aqui para a manipulação de dados que sofram interferência externa (**input** e **output**).

1.2 HASKELL NA WEB

Hoje, é possível desenvolver para web usando a linguagem Haskell, e alguns frameworks que usam esta linguagem são notórios, tais como: Yesod, Scotty, Happstack e Snap. Cada um deles possui características diferentes. Aqui daremos ênfase ao Yesod, que foi desenvolvido por Michael Snoyman (SNOYMAN, 2012).

A maior característica do Yesod é o fato de termos segurança de tipos (*type-safety*) nas URLs. Ou seja, uma aplicação web desenvolvida com este framework não terá problemas de imagens ou links quebrados, pois o conceito em foco garante que tais erros sejam checados em tempo de compilação e não de execução, garantindo aplicações com menos erros.

Conforme Snoyman (2012), o Yesod possui suporte também para webservices REST, padrão de internacionalização i18n, persistência de dados (dependendo do pacote usado, isto pode ser feito de modo type-safe também), interpoladores para criação de páginas dinâmicas e templatização usando os shakespearean templates que consistem em DSLs (*Domain Specific Languages*) para HTML, CSS e JavaScript chamadas de Hamlet, Lucius (Cassius) e Julius. É possível concluir que é possível programar usando Haskell, e colocar aplicações em produção, o que antigamente não era possível.

1.3 CONCLUSÃO

Vimos algumas informações básicas sobre a linguagem Haskell e o paradigma funcional, bem como algumas características importantes da linguagem. A partir de agora, apresentaremos a configuração de um ambiente apropriado para programarmos

nesta linguagem, e também os primeiros conceitos e trechos de código do livro.

CAPÍTULO 2

PRIMEIROS EXEMPLOS

A linguagem Haskell é uma linguagem de tipagem forte e estática, ou seja, toda expressão possui um tipo definido em tempo de execução. Qualquer problema com tipos será pego em tempo de compilação, por exemplo, se um parâmetro do tipo inteiro é esperado e um tipo char é passado.

Será necessária a instalação do Haskell Platform, que pode ser baixado em https://www.haskell.org/platform/. O compilador a ser baixado chama-se `ghc` (*The Glasgow Haskell Compiler*) e sua versão atual é a 8.0.1.

Para começarmos, usaremos a ferramenta `ghci` , e ela já vem inicialmente no mesmo pacote de instalação citado, que é um REPL (*Read, Evaluate, Print, Loop*). Basta digitar em linha de comando a palavra `ghci` e teremos o seguinte prompt:

```
Prelude>
```

A palavra `Prelude` significa que os módulos básicos da linguagem foram carregados. Para carregar um módulo, basta executar o comando:

```
Prelude> :l Teste.hs
```

O `Teste.hs` é um arquivo na mesma pasta na qual o

comando `GHCi` foi digitado. Um módulo começa da seguinte forma:

```
module Teste where
```

Daí em diante, poderemos fazer nossos tipos de dados e funções, e proceder com o desenvolvimento de um módulo. Observe também que, como chamamos o módulo de `Teste` , o seguinte comando também carregará este módulo:

```
Prelude> :l Teste
```

Isto suprime a extensão `.hs` . Se a do começo do módulo for suprimida, não haverá problema algum e o nome deste será chamado pelo `ghc` de `Main` . Se o nome do arquivo e o nome do módulo forem diferentes, deveremos carregá-los pelo nome do arquivo apenas.

Após o módulo ser carregado com sucesso (e passar pelo processo de compilação), qualquer alteração feita precisará ser carregada novamente com o módulo, usando o comando:

```
Teste> :r
```

Este dará um reload no módulo e compilará novamente, sem a necessidade de usar o `:l` com o nome do arquivo. Aqui você poderá também calcular valores de expressões que dependem ou não do módulo carregado.

```
Teste> 2+2
4

Teste> 3<4
True

Teste> "Ola " ++ "Mundo"
"Ola Mundo"
```

A resposta aparecerá logo abaixo do indicador do prompt. Dessa forma, é possível testar as funções implementadas em um módulo de maneira fácil.

Finalmente, um dos comandos mais úteis do `ghci` é o `:t`, que inspeciona o tipo de dado de um valor:

```
Teste>:t True
True :: Bool
```

A resposta obtida pode ser lida como `True`, e tem o tipo `Bool`. Note que o símbolo `::` foi lido como "tem o tipo".

2.1 PRIMEIRO CONTATO COM OS TIPOS DE DADOS E FUNÇÕES

O arquivo de extensão `.hs` é onde ficam as funções declaradas, como visto na seção anterior, explicitamente de modo a serem usadas, por enquanto, pelo GHCi. Toda função deve seguir o seguinte padrão:

```
nomeDaFuncao p1 p2 p3 ... pN = expressão que depende dos N parâme
tros.
```

Note que a convenção utilizada para nomear funções é a *CamelCase* (equivalente para nomeação de métodos em Java) e, para nomear os parâmetros, todas as letras devem ser minúsculas. Funções também possuem tipo e podem ser declaradas explicitamente. Isto é considerado uma boa prática e, sempre que possível, devemos colocar tipos em nossas funções, como o seguinte exemplo:

```
maiorQue :: Int -> Int -> Bool
maiorQue x y = x > y
```

Esse exemplo mostra uma função que possui dois parâmetros inteiros, e sua expressão possui um retorno booleano. Portanto, o tipo da função é `Int -> Int -> Bool`.

Uma dica a seguir é que o número de flechas (`->`) acompanha o número de parâmetros que a função admite, que neste caso, são chamados de `x` e `y`. A função `(>)` também pode ser chamada de maneira infixa, como segue:

```
maiorQue :: Int -> Int -> Bool
maiorQue x y = (>) x y
```

Como a função `(>)` recebe dois parâmetros, o uso dos parênteses faz a chamada desta função de maneira infixa.

Observe agora esta função declarada a seguir:

```
u :: Int
u = 7
```

Neste exemplo, temos uma função `u` que não recebe parâmetros e possui um retorno inteiro constante. Um lembrete aos programadores imperativos é que, aqui em Haskell, o sinal de igual não representa atribuição como em muitas linguagens, e sim definição. Pelo fato de Haskell ter seus dados como sendo imutáveis, o trecho a seguir produzirá um erro de ambiguidade de definição.

```
u :: Int
u = 7

u = 6
```

Como pode-se ver, a partir do momento em que tentamos compliar o trecho anterior, não é possível distinguir as duas definições, causando, assim, um erro em tempo de compilação.

```
Teste.hs:6:1: error:
    Multiple declarations of ‘u’
    Declared at: Teste.hs:4:1
                 Teste.hs:6:1
```

2.2 OPERAÇÃO COM LISTAS

O `Prelude` vem recheado de funções para operar com listas, e nesta seção vamos ver algumas delas. Uma lista é um tipo de dado que assume um valor vazio `[]` ou, se possuir algum elemento, como um inteiro, escrevemos na forma `[1,2,3]`.

Listas possuem elementos do mesmo tipo e podem tê-los adicionados ou removidos, fazendo com que seu tamanho seja variável. Podemos inspecionar o tipo de uma lista no `ghci` desta forma:

```
Prelude>:t [True, False, True]
[True, False, True] :: [Bool]
```

A resposta obtida é lida da mesma forma que anteriormente. `[True, False, True]` tem o tipo `[Bool]`. A função `(++)` representa a concatenação entre duas listas, por exemplo:

```
Prelude> [1,2] ++ [3,4]
[1,2,3,4]
```

Também temos a maneira infixa:

```
Prelude> (++) [1,2] [3,4]
[1,2,3,4]
```

Observe que, se um dos parâmetros for uma lista vazia `[]`, obteremos os seguintes resultados:

```
Prelude> [1,2] ++ []
[1,2]
```

```
Prelude> [] ++ [1,2]
[1,2]
```

Estes resultados serão úteis mais à frente. Vale notar que a função `(++)` opera sob listas de qualquer tipo de dado, por exemplo:

```
Prelude> [True,False] ++ [True]
[True, False, True]

Prelude> "ABCD" ++ "EFG"
"ABCDEFG"
```

Não podemos nos esquecer que uma String (aspas duplas) é um apelido para lista de Char (aspas simples) ou, em Haskell, `[Char]`. O fato de esta função funcionar para listas de quaisquer tipos será explorado mais à frente.

As funções `head`, `last` e `tail` extraem o primeiro, o último e uma lista sem o primeiro elemento, respectivamente. Veja:

```
Prelude> head "ABCDEFG"
'A'

Prelude> last "ABCDEFG"
'G'

Prelude> tail "ABCDEFG"
"BCDEFG"
```

Porém, se estas funções forem calculadas em uma lista vazia `[]`, obteremos:

```
Prelude> head []
*** Exception: Prelude.head: empty list

Prelude> last []
*** Exception: Prelude.last: empty list

Prelude> tail []
```

```
*** Exception: Prelude.tail: empty list
```

Tais erros decorrem do fato de as três funções não estarem definidas na lista vazia. Tais funções são chamadas de funções parciais e devem ser evitadas, pois são fontes de erros de execução, os quais tentamos diminuir ao mínimo quando usamos a linguagem Haskell.

A função `reverse` inverte a ordem de uma lista de qualquer tipo:

```
Prelude> reverse "HASKELL"
"LLEKSAH"

Prelude> reverse [1,2,3]
[3,2,1]

Prelude> reverse []
[]
```

Observe que é possível deduzir que a função `last` é a composta de `head` com `reverse`:

```
Prelude> (head . reverse) "HASKELL"
'L'
Prelude> last "HASKELL"
'L'
```

Mais detalhes sobre a função que compõe duas funções `(.)` serão dados mais adiante. Não se preocupe.

A função `!!`, usada de maneira infixa, recebe uma lista e um número inteiro, e devolve o elemento na posição informada.

```
Prelude> [1,2,3,4] !! 2
3
Prelude> [1,2,3,4] !! 0
1
```

Tal função também é parcial, visto que:

```
Prelude> [1,2,3,4] !! (-2)
*** Exception: Prelude.!!: negative index
Prelude> [] !! 2
*** Exception: Prelude.!!: index too large.
```

A primeira exceção nos indica que houve uma ocorrência de índice negativo, no caso, `-2`. Já a última exceção que tentamos acessar era uma posição maior que o tamanho total da lista. Neste caso, temos uma lista de tamanho `0` e tentamos acessar a terceira posição `2`.

A função cons `(:)` recebe um elemento e uma lista, e devolve este elemento na frente da lista:

```
Prelude> 3 : [3,4,5,6,-1]
[3,3,4,5,6,-1]
Prelude> 'A' : "BCDE"
"ABCDE"
Prelude> 2 : []
[2]
Prelude> 'A' : []
"A"
```

Também é possível chamar esta função sucessivas vezes. Veja:

```
Prelude> 3 : [4,5,2]
[3,4,5,2]
Prelude> 3 : 4 : [5,2]
[3,4,5,2]
Prelude> 3 : 4 : 5 : [2]
[3,4,5,2]
Prelude> 3 : 4 : 5 : 2 : []
[3,4,5,2]
```

Guarde bem esse exemplo, pois será de suma importância quando o conceito de *pattern matching* for introduzido.

Finalmente a função `length` recebe uma lista e retorna a quantidade de elementos contida nela.

```
Prelude> length [1,2,3]
3
Prelude> length ['a']
1
Prelude> length "a"
1
Prelude> length []
0
```

2.3 COMPREENSÃO DE LISTAS

Em Haskell, é possível construir listas de quaisquer tipos usando expressões que podem ser distribuídas a todos os elementos de um dado vetor, usando *compreensão de listas* (ou *list comprehensions*). De maneira geral:

```
   [EXPRESSÃO(var) | var<-LISTA, FILTRO_1, FILTRO_2, ..., FILTRO_
]
```

A expressão é qualquer função que será distribuída nos elementos da lista, representados por `var`, com os elementos que passem na condição dos filtros. Veja um exemplo:

```
dobroLista :: [Int] -> [Int]
dobroLista xs = [2*x | x<-xs]
```

Essa função possui como parâmetro a lista de inteiros `x` e ela devolve uma lista de inteiros contendo o dobro de cada elemento `x`, contido em `xs`. Tendo em mente a descrição da função anterior, é possível enxergar o uso da sintaxe `[Int] -> [Int]`, que representa que a função possui um parâmetro do tipo lista de inteiros e esta retorna uma lista de inteiros.

O tipo definido após a última `->` representa o retorno da função. No exemplo anterior, a expressão é a função `2*x`, e a lista na qual a função será distribuída é `xs`.

```
lista :: [Int]
lista = [2*x+1 | x<-[0 .. 10], x/=5]
```

Neste exemplo, vemos que a função `2*x+1` se distribuirá a todos elementos da lista `[0 .. 10]`, com exceção do número 5 que não passa no filtro indicado. Portanto, a lista tem como conteúdo `[1, 3, 5, 7, 9, 13, 15, 17, 19, 21]`.

2.4 TUPLAS

Diferentemente das listas, que só carregam dados de um tipo só com um número variável de elementos, as tuplas carregam diversos tipo ao mesmo tempo e possuem um número fixo de elementos. Não é possível usar a função cons (`:`), nem concatenar (`++`) nada a elas.

Tuplas são imutáveis. O número de elementos em uma tupla é fixo, e cada local no qual um elemento reside é chamado de coordenada.

```
Prelude>:t ('A',"ALO")
('A',"ALO") :: (Char, [Char])
```

A resposta obtida indica que `('A',"ALO")` tem o tipo `(Char, [Char])`, que implica que a primeira coordenada possui o tipo `Char` e a segunda `[Char]` ou `String`. É possível fazer funções usando tupla:

```
foo :: Char -> Int -> (Int, String)
foo x y = (y+9, x:[x])
```

Vamos testar no GHCi:

```
Prelude> foo 'E' 2
(11,"EE").
```

As funções `fst` e `snd` projetam a primeira e a segunda coordenada de uma tupla, respectivamente.

```
Prelude> fst ('A',"ALO")
'A'
Prelude> snd ('A',"ALO")
"ALO"
```

2.5 EXERCÍCIOS

2.1) Gere as listas:

a) `[1,11,121,1331,14641,161051,1771561]`

b) `[1,2,3,5,6,7,9,10,11,13,14,15,17,18,19,21,22,23,25,26,27,29,30,31,33,34,35,37,38,39]`

c) `["AaBB", "AbBB", "AcBB", "AdBB", "AeBB", "AfBB", "AgBB"]`

d) `[5,8,11,17,20,26,29,32,38,41]`

e) `[1.0,0.5,0.25,0.125,0.0625,0.03125]`

f) `[1,10,19,28,37,46,55,64]`

g) `[2,4,8,10,12,16,18,22,24,28,30]`

h) `['@','A','C','D','E','G','J','L']`

2.2) Crie uma função que verifique se o tamanho de uma String é par ou não. Use `Bool` como retorno.

2.3) Escreva uma função que receba um vetor de Strings e retorne uma lista com todos os elementos em ordem reversa.

2.4) Escreva uma função que receba um vetor de Strings e retorne uma lista com o tamanho de cada String. As palavras de tamanho par devem ser excluídas da resposta.

2.5) Escreva a função `head` como composição de duas outras.

2.6) Faça uma função que receba uma String e retorne `True` se esta for um palíndromo; caso contrário, `False` .

2.7 Faça uma função que receba um inteiro e retorne uma tupla, contendo: o dobro deste número na primeira coordenada, o triplo na segunda, o quádruplo na terceira e o quíntuplo na quarta.

2.6 CONCLUSÃO

Neste capítulo, vimos o básico para a instalação do compilador do Haskell (GHC) e de seu **REPL** (GHCi), alguns comandos básicos e como calcular expressões no GHCi. A principal noção abordada foi a de tipos e funções, que são conceitos recorrentes em toda a linguagem.

Listas e tuplas são tipos de dados e foram introduzidos também aqui, pois são ferramentas básicas para conceitos mais complexos que serão vistos mais adiante. A manipulação de listas dá uma boa ideia de como é a escrita em programação funcional, e fornece uma noção ao leitor de como será daqui para a frente e o quanto se difere das linguagens imperativas comumente usadas no mercado.

É sempre bom lembrar da importância da resolução dos exercícios e do acompanhamento dos exemplos dados no GHCi. Sem isso, o entendimento dos capítulos vindouros pode ser comprometido.

O próximo capítulo é consequência natural dos conceitos apresentados aqui. Para avançar, certifique-se de que você entendeu como construir sua função, como ler o seu tipo e como funciona sua escrita básica.

CAPÍTULO 3

DECLARANDO NOVOS TIPOS DE DADOS

Este capítulo é uma introdução ao abrangente sistema de tipos da linguagem. Nele, é abordado como criar e manipular tipos de dados na linguagem Haskell.

Uma das possibilidades de criar novos tipos em Haskell é utilizando a palavra reservada `data`. Considere o seguinte exemplo:

```
data Dia = Segunda | Terca | Quarta |
           Quinta | Sexta | Sabado | Domingo
```

Lê-se que o novo tipo chamará `Dia` e possuirá *value constructors* `Segunda` ou `Terca` ou `Quarta` ou `Quinta` ou `Sexta` ou `Sabado` ou `Domingo`. Logo, nota-se que `|` pode ser lido como `ou` e o tipo `Dia` pode assumir um entre vários valores. O tipo `Dia` é comumente chamado na literatura de *sum type*.

Os *values constructors* são valores assumidos pelos tipos (chamados também de *data constructors*). Se usarmos `:t` no GHCi, é possível enxergar o tipo de cada *value constructor*:

```
Prelude>:t Segunda
Segunda :: Dia
Prelude>:t Quinta
```

```
Quinta :: Dia
```

Podemos agora criar funções com o novo tipo criado:

```
agenda :: Dia -> String
agenda Domingo = "TV..."
agenda Sabado = "Festa"
agenda _ = "Trabalho"
```

Este exemplo toma como parâmetro um `Dia` e retorna uma String que representa a tarefa determinada pela agenda. Note que, nesta função, foi aplicado o conceito de *pattern matching*, que será estudado mais de perto na próxima seção, uma importante parte da sintaxe de funções do Haskell.

Neste exemplo, se o `Dia` passado for `Domingo`, o retorno será `"TV..."`; se for `Sabado`, `"Festa!"`; e qualquer outro dia que não os dois retornará `"Trabalho!"`.

3.1 PATTERN MATCHING

Sempre que desejarmos enxergar partes de um tipo em um determinado ponto do programa, usamos o *pattern matching*. Esta técnica permite a inspeção de partes menores de um *value constructor* contra um padrão determinado pelo programador.

É possível enxergar isto como aquele brinquedo que possui formas geométricas com buracos e suas respectivas peças. Se tentarmos encaixar a peça em forma de quadrado no buraco em forma de círculo, não será possível.

Vamos analisar, no exemplo a seguir, o tipo `(Int,Int)`, que é uma tupla com um inteiro em cada uma das duas coordenadas. Serão listadas algumas combinações válidas deste tipo, na entrada

de uma função qualquer `f`:

```
f :: (Int, Int) -> Int
f (0,0) = 0
f (0,1) = 1
f (1,0) = 1
f (x,0) = x
f (0,y) = y
f (x,y) = x+y
```

A primeira coisa a se observar é a forma do tipo de entrada da função `(Int,Int)` com as entradas da função. Todas elas possuem o mesmo tamanho de tupla e os valores constantes que aparecem são do tipo `Int`, no caso `0` e `1`. Agora vamos executar tal função no GHCi:

```
Prelude> f (0,0)
0
```

Note que o argumento passado para a função foi `(0,0)`, e este padrão foi encontrado na linha `f (0,0) = 0`, logo o valor retornado será 0. Observe:

```
Prelude> f (0,1)
1
```

O argumento passado foi `(0,1)`, e o padrão encontrado está na linha `f (0,1) = 1`, retornando então `1`. O padrão da linha seguinte, quarta linha da função `f`, é análogo a este. Se calcularmos a expressão a seguir no GHCi:

```
Prelude> f (0,-90)
-90
```

O valor retornado é `-90`, pois o padrão encontrado é o da sexta linha de `f`, ou seja, `f (0,y) = y`. O padrão `(0,y)` pode ser lido como: *valor 0 fixo na primeira coordenada e um valor qualquer na segunda coordenada que chamaremos de y.*

O padrão `(x,0)` pode ser lido analogamente como *valor qualquer na primeira coordenada que chamaremos de x e um valor fixo 0 na segunda*. O nome para os valores variáveis podem ser repetidos em várias linhas, porém, não na mesma. Poderíamos ter o padrão `(0,x)` em vez de `(0,y)` sem problemas. Então, temos este outro exemplo:

```
Prelude> f (10,3)
13
```

Ele se encaixará no último padrão, pois, os únicos valores fixos são 0 e 1. Note que, no último padrão `f (x,y) = x+y` , teremos que `x` valerá `10` , e `y` será igual a `3` . Em suma, para tuplas, temos os seguintes padrões:

- `(0,0)` – valores fixos em ambas as coordenadas;
- `(x,0)` – valor variável na primeira e fixo na segunda;
- `(0,y)` – valor fixo na primeira e variável na segunda;
- `(x,y)` – valores variáveis em ambas coordenadas (devem possuir nomes diferentes).

```
g :: (Int, Int) -> Int
g (7,7) = 7
g (_,_) = 0
```

Uma observação é a que toda vez que o `_` (underline) aparecer, significa variável ignorada. Ou seja, tanto o valor da

primeira variável quanto o da segunda são variáveis, porém, com o `_`, o efeito é o de ignorar os valores. Neste caso:

```
Prelude> g (34,58)
0
```

Assim, 34 não é igual a 7, e 58 não é igual a 7, desviando a chamada da função para o padrão contido na segunda linha, `g (_,_) = 0`. Os valores 34 e 58 foram ignorados pelo `_`. É possível também ignorar o valor todo:

```
g :: (Int, Int) -> Int
g (7,7) = 7
g _ = 0
```

O efeito será o mesmo. Agora o exemplo será com listas:

```
h :: [Int] -> Int
h [] = 0
h (_:[]) = 1
h (_:x:[]) = 2+x
h (x:y:z:[]) = 3+x+y+z
h (x:_:_:w:[]) = 4+x+w
h (x:xs) = x
```

Do capítulo anterior, sabemos que `:` é o operador cons, e que `1:2:3:[]` se equivale a `[1,2,3]`. No GHCi, podemos calcular:

```
Prelude > h []
0
```

Ou seja, a função `h` calculada no valor fixo `lista vazia` produz o valor `0`. Note que o valor constante `[]` é fixo no primeiro padrão. Agora se calcularmos `h` com o argumento `[1,2,3]`:

```
Prelude > h [213]
1
```

Obteremos como retorno o valor `1`, pois o argumento é `[213] = 213:[]` e o padrão encontrado é o da linha `h (_:[]) = 1`, que é o padrão para listas de **exatamente** um elemento. O elemento da lista `213` foi ignorado pelo `_`.

```
Prelude > h [5,6]
8
```

Nesse código temos que `[5,6] = 5:6:[]`, portanto, o padrão é o da linha `h (_:x:[]) = 2+x`, sendo que o elemento `5` foi ignorado pelo `_`, e o valor `6` atrelado à variável `x`. Assim, o valor de retorno é `2+6`, que é 8. Este padrão `(_:x:[])` é para **exatamente** dois elementos.

```
Prelude > h [1,2,3]
9
Prelude > h [1,2,3,4]
9
```

Estes exemplos anteriores retornaram `9` em consequência de os argumentos terem sido `[1,2,3] = 1:2:3:[]`, que se encaixavam no padrão `h (x:y:z:[]) = 3+x+y+z`, como `x` valendo `1`, `y` valendo `2` e `z` valendo `3`. Já `[1,2,3,4] = 1:2:3:4:[]` se encaixa em `h (x:_:_:w:[]) = 4+x+w`, com `x` valendo `1`, `w` valendo `4` e os outros dois valores ignorados pelo `_`.

Esse dois padrões são para 3 e 4 elementos, respectivamente. Para testar o último padrão da função `h`, basta que a lista tenha mais de 4 elementos. Veja um exemplo:

```
Prelude > h [1,2,3,4,5]
1
```

Ou seja, como a lista tem cinco elementos, ela não bate com nenhum padrão de elementos fixos (aqueles que terminam com

`[]`), mas sim com o de elemento variável (que terminam com `xs` , qualquer outro nome ou `_`). Logo, temos `[1,2,3,4,5] = 1:[2,3,4,5]` , com `x` valendo `1` e `xs` valendo `[2,3,4,5]` .

Pode-se ler o padrão `(x:xs)` como **um ou mais elementos**, assim como `(x:y:xs)` **dois ou mais elementos**, `(x:_:_:_:a:as)` **cinco ou mais elementos**, e assim por diante. Só para fixar, um último exemplo: considere `[1,2,3,4,5,6]` . Este se encaixa, por exemplo, em `(x:_:c:b:xs)` , com `x=1` , `c=3` , `b=4` e `xs=[5,6]` .

3.2 CAMPOS DE UM CONSTRUTOR

Todo value constructor pode possuir campos.

```
data Pessoa = Fisica String Int | Juridica String
```

Nesse exemplo, o tipo `Pessoa` possui dois values constructors: `Fisica` , que possui dois campos; e `Juridica` , que possui apenas um. Inspecionando o novo tipo no GHCi, é possivel notar que todo value constructor é, na realidade, uma função que retorna um valor do tipo `Pessoa` , e os campos são seus parâmetros.

```
Aula3> :t Fisica
Fisica :: String -> Int -> Pessoa
Aula3> :t Juridica
Fisica :: String -> Pessoa
```

Este conceito de chamar um função com menos argumentos do que o esperado é chamado de *currying*, e será explorado em mais detalhes no próximo capítulo. Com a ajuda do *pattern matching*, vamos criar uma função teste que recebe uma `Pessoa` como parâmetro, e esta retornará uma tupla contendo duas

Strings, informando nome e idade.

No caso de `Pessoa` do tipo `Fisica`, a função retornará uma mensagem que mostra o nome na primeira coordenada, assim como a idade na segunda. Para o *value constructor* `Juridica`, será mostrado o nome na primeira coordenada e uma mensagem informando que não há o campo idade na segunda.

```
teste :: Pessoa -> (String, String)
teste (Fisica x y) = ("Nome: " ++ x, "Idade: " ++ show y)
teste (Juridica x) = ("Nome: " ++ x, "Não há idade")
```

A primeira parte da função usa o *pattern matching* para encaixar os parâmetros `x` e `y` dentro do padrão encontrado no tipo `Pessoa String Int`, logo, `x` possui o tipo `String` e `y`, o tipo `Int`. A segunda parte procede da mesma forma, porém, o *value constructor* `Juridica` possui apenas um parâmetro String, logo, o *pattern matching* faz com que `x` seja `String`.

Uma outra maneira de se declarar um tipo é o uso da palavra `newtype`, porém, nesse caso, só podemos ter um *value constructor* e, no máximo, um campo. Por exemplo:

```
newtype Dado = Dado Int
```

Ele serve para uma espécie de apelido para um tipo existente (no caso, `Int`). Outra diferença é que `data` tem uma valoração preguiçosa, enquanto `newtype` não. Veja um exemplo:

```
newtype Dado = Dado Int
data Dado1 = Dado1 Int
```

Se tentarmos fazer algma operação com o valor `Dado undefined`, teremos erro pela ausência da valoração preguiçosa, enquanto `Dado1 undefined` não causará erro algum, desde que

você não acesse o campo. Note que o valor `undefined` está presente em todo tipo e representa a ausência de computação.

3.3 RECORD SYNTAX

Outra forma de declarar novos tipos com parâmetros é o *record syntax*, que permite escrever tais tipos usando nomes para cada parâmetro. O uso do *record syntax* permite extrair os valores contidos no campo (associado a *value constructor*) da mesma maneira que um `getter` da programação orientada a objetos (lembre-se de que aqui não há o conceito de modificadores de acesso ou coisa parecida).

O seu uso é simples. Apenas daremos nomes aos campos que os *values construtors* carregam e só. Fazendo isso, já estaremos usando este conceito. Por exemplo, o tipo `Ponto` , que pode ser definido como:

```
data Ponto = Ponto Double Double
```

Pode ser escrito assim:

```
data Ponto = Ponto {xval,yval :: Double}
```

A vantagem é a semântica do fato dos campos terem nomes. Outro ponto positivo é que cada nome dado também pode ser usado como função de projeção de valores (algo parecido com o `getter` da programação orientada a objetos). Podemos inspecionar o tipo das funções de projeção `xval` e `yval` no GHCi.

```
Prelude> :t xval
xval :: Ponto -> Double
Prelude> :t yval
yval :: Ponto -> Double
```

Em suma, o *record syntax* provê, além de um nome aos campos de um *value constructor*, funções de projeção para obter o valor do campo correspondente.

Uma observação é que o *type constructor* `Ponto` pode ter o mesmo nome do seu *value constructor*, e isto não causa ambiguidade nenhuma ao compilador.

```
Aula3> Ponto 1.1 2
Ponto {xval = 1.1, yval = 2.0}
```

Para fazer uma função que calcula a distância do ponto à origem, podemos proceder de três formas.

1. Primeira forma:

```
distOrig :: Ponto -> Double
distOrig (Ponto x y) = sqrt(x**2 + y**2)
```

2. Segunda forma:

```
distOrig :: Ponto -> Double
distOrig (Ponto {xval=x, yval=y}) = sqrt(x**2 + y**2)
```

3. Terceira forma:

```
distOrig :: Ponto -> Double
distOrig p = sqrt(xval p**2 + yval p**2)
```

As duas primeiras maneiras usam o *pattern matching* para encaixar os parâmetros `x` e `y` no padrão que possui o tipo `Ponto`. Já a última usa `xval` e `yval`, funções de projeção adquiridas no uso do *record syntax*.

3.4 MINIPROJETO: RH DE UMA EMPRESA DE TI

Após a discussão dos conceitos ao final de alguns capítulos, um miniprojeto será mostrado de forma a "juntar" os conceitos vistos e motivar os leitores com aplicações um pouco mais práticas do que os exemplos vistos. Esse miniprojeto aumentará de tamanho significativamente ao final de cada capítulo, até virar um minissistema pronto.

Lembre-se de que o foco deste livro não é o desenvolvimento web (que será pensado em próximas versões), mas sim plantar a semente da programção funcional. Acompanhar estes miniprojetos é de fundamental importância para o aprendizado. A história se passa em uma empresa de TI que quer fazer um sistema em Haskell para controle de seu RH, como experiência para a adoção da linguagem futuramente.

Então, temos empresa de TI com um pequeno sistema de RH e, nesse minúsculo sistema, são mantidos nomes e os cargos. O salário dos funcionários são constantes conforme o cargo (não há programadores que ganhem mais que os outros). Os cargos são: estagiário, programador, analista e gerente.

Há mobilidade de cargos na estrutura hierárquica e é gerada uma folha com os detalhes da empresa como um pequeno relatório, em formato JSON. Antes de começar, crie um arquivo chamado `Projeto.hs` .

```
module Projeto where

data Cargo = Estagiario | Programador | Coordenador | Gerente der
iving Show

data Pessoa = Pessoa {cargo :: Cargo, nome :: String} deriving Sh
ow

verSalario :: Pessoa -> Double
```

```
verSalario (Pessoa Estagiario _) = 1500
verSalario (Pessoa Programador _) = 5750.15
verSalario (Pessoa Coordenador _) = 8000
verSalario (Pessoa Gerente _) = 10807.20

verFolha :: Pessoa -> String
verFolha p = "{nome: \"" ++ (nome p) ++
             "\", cargo: \"" ++ show (cargo p) ++
             "\", salario: " ++ show (verSalario p) ++ "}"

promover :: Pessoa -> Pessoa
promover (Pessoa Estagiario n) = Pessoa Programador n
promover (Pessoa Programador n) = Pessoa Coordenador n
promover (Pessoa Coordenador n) = Pessoa Gerente n
promover (Pessoa _ n) = Pessoa Gerente n
```

Foi criado os tipos `Cargo` e `Pessoa` (linhas 2 e 3 do programa). O tipo `Cargo` possui quatro *values constructors* (uso do `deriving Show` será analisado no *capítulo 5*, agora é só importante saber que essa cláusula deve ser colocada para a função `show` funcionar, e que a função `show` transforma qualquer tipo em uma `String`).

O tipo `Pessoa` possui um *value constructor* contendo dois campos (`nome` e `cargo`), escritos em formato de *record syntax*. A função `verSalario` usa o *pattern matching* contra o campo `cargo` do tipo `Pessoa`, e ignora o campo `nome`. Esta retorna um valor fixo de salário para cada *value constructor* do tipo `Cargo`.

A função `verFolha` recebe uma parâmetro do tipo `Pessoa` e retorna uma representação deste tipo em formato `JSON`. Note que usamos o `++` para concatenar as strings e a função de projeção `cargo`, definida no *record syntax* do tipo `Pessoa`. Também usamos o retorno da função `verSalario` para montar esta estrutura.

Finalmente, a função `promover` recebe um parâmetro do tipo `Pessoa` e faz um *pattern matching* contra o campo `cargo`. Em todos eles usou-se a letra `n` para representar o campo `nome` do tipo `Pessoa`. O último *pattern matching* de `promover` nos diz que não há promoção para cargos de `Gerente`.

3.5 EXERCÍCIOS

3.1) Crie o tipo `Pergunta` com os values constructors `Sim` ou `Nao`. Faça as funções seguintes, determinando seus tipos explicitamente.

- `pergNum`: recebe via parâmetro uma `Pergunta`. Retorna `0` para `Nao` e `1` para `Sim`.
- `listPergs`: recebe via parâmetro uma lista de `Perguntas`, e retorna `0`s e `1`s correspondentes aos constructores contidos na lista.
- `and'`: recebe duas `Perguntas` como parâmetro e retorna a tabela `verdade` do `and` lógico, usando `Sim` como verdadeiro e `Nao` como falso.
- `or'`: idem ao anterior, porém deve ser usado o `ou` lógico.
- `not'`: idem aos anteriores, porém usando o `not` lógico.

3.2) Faça o tipo `Temperatura` que pode ter valores `Celsius`, `Farenheit` ou `Kelvin`. Implemente as funções:

- `converterCelsius`: recebe um valor `double` e uma temperatura, e faz a conversão para Celsius.
- `converterKelvin`: recebe um valor `double` e uma

temperatura, e faz a conversão para Kelvin.

- `converterFarenheit` : recebe um valor `double` e uma temperatura, e faz a conversão para Farenheit.

3.3) Implemente uma função que simule o vencedor de uma partida de pedra, papel e tesoura usando tipos criados. Casos de empate devem ser considerados em seu tipo.

3.4) Faça uma função que retorne uma string, com todas as vogais maiúsculas e minúsculas eliminadas de uma string passada por parâmetro usando *list compreenshion*.

3.5) Sabe-se que as unidades imperiais de comprimento podem ser `Inch` , `Yard` ou `Foot` (há outras ignoradas aqui). Sabe-se que `1in=0.0254m` , `1yd=0.9144m` , `1ft=0.3048` . Faça a função `converterMetros` que recebe a unidade imperial e o valor correspondente nesta unidade. Esta função deve retornar o valor em metros.

Implemente também a função `converterImperial` , que recebe um valor em metros e a unidade de conversão. Esta função deve retornar o valor convertido para a unidade desejada.

3.6) Faça um novo tipo chamado `Mes` , que possui como valores todos os meses do ano. Implemente:

- A função `checaFim` , que retorna o número de dias que cada mês possui (considere fevereiro tendo 28 dias).
- A função `prox` , que recebe um mês atual e retorna o próximo mês.
- A função `estacao` , que retorna a estação do ano de acordo com o mês e com o hemisfério.

Use apenas tipos criados pela palavra `data` aqui.

3.7) Faça uma função que receba uma `String` e retorne `True` se esta for um palíndromo; caso contrário, `False`.

3.8) Faça uma função que elimine todos os números pares, todos os ímpares múltiplos de 7 e negativos de uma lista de inteiros passada via parâmetro. Você deve retornar esta lista em ordem reversa em comparação a do parâmetro.

3.9) Faça uma função que recebe três Strings `x`, `y` e `z` como parâmetro. A função retorna uma tupla com três coordenadas contendo a ordem reversa em cada. A primeira coordenada deve conter string reversa do primeiro parâmetro, e assim por diante.

3.10) Faça uma função chamada `revNum`, que receba uma String `s` e um `Int n`. Esta deverá retornar as **n** primeiras letras em ordem reversa e o restante em sua ordem normal. Exemplo:

```
revNum 4 "FATEC" = "ETAFC"
```

3.11) Crie o tipo de dado `Binario` que pode ser `Zero` ou `Um`. Faça outro tipo de dado chamado `Funcao` que pode ser `Soma2`, `Maior`, `Menor` ou `Mult2`. Implemente a função `aplicar` que recebe uma `Funcao` e dois `Binarios`. Seu retorno consiste em executar a operação desejada. Exemplo:

```
aplicar Soma2 Um Um  = Zero
```

3.12) Faça uma função chamada `binList`, usando *list compreeshion*, que recebe uma lista de `Binarios` (ver exercício anterior) e retorna outra lista com elemento somado `Um` e convertido para `Int`. Exemplo:

```
binList [Um, Zero, Zero, Um, Zero] = [0,1,1,0,1]
```

3.13) Faça um novo tipo chamado `Metros`, que possui um `\textit{value constructor}` de mesmo nome, cujos parâmetros são: um `Int` que representa a dimensão, e um `Double` que representa o valor da medida e outro chamado `MetragemInvalida`. Implemente as funções:

- `areaQuadrado :: Metros -> Metros`: calcula a área de um quadrado.
- `areaRet :: Metros -> Metros -> Metros`: calcula a área de um retângulo.
- `areaCubo :: Metros -> Metros`: calcula a área de um cubo.

Exemplo:

```
Prelude> areaQuadrado (Metros 1 2.0)
Metros 2 4.0
```

Use o *pattern matching* para ignorar as metragens erradas (calcular a área de um quadrado com um lado de dimensão 4 não é válido).

3.14) Faça o novo tipo `Valido` que possui dois *value constructors* `Sim` e `Nao`. O value constructor `Sim` possui um parâmetro (campo) `String`. Implemente uma função `isNomeValido` que recebe um nome e retorna `Nao` caso a `String` seja vazia; caso contrário, `Sim`.

3.15) Refaça o exercício 3 do capítulo anterior usando *record syntax* e tipos com parâmetro (siga o exemplo da conversão de medidas SI para imperial).

3.16) Faça o tipo `Numero`, que possui um value constructor `Ok` com um campo `double` e outro value constructor `Erro`

com um campo `String`. Faça a função `dividir` que divida dois números e, caso o segundo número seja 0, emita um erro (use o *pattern matching*). Exemplo:

```
Prelude> dividir (Numero 6) (Numero 5)
Numero 1.2.
```

3.17) Faça o tipo `Cripto` que possua dois values constructors `Mensagem` e `Cifrado`, ambos com um campo `String` e um value constructor `Erro`. Faça as funções `encriptar` e `decriptar`, seguindo cada exemplo a seguir.

```
Prelude> encriptar (Mensagem  "FATEC")
Cifrado "GBUFD"

Prelude> decriptar (Cifrado  "DBTB")
Mensagem "CASA"
```

Veja que a encriptação deve empurrar cada letra a frente e a decriptação faz o inverso, empurrando uma letra para trás. Use as funções `succ` e `pred`, e também *list compreeshions*. Não é possível encriptar mensagens cifradas e decriptar mensagens.

3.18) Faça uma função `encriptarTodos` que encripta (ou dá erro) todos os elementos de um vetor de `Cripto`.

3.19) Tendo como base o exercício de conversão de medidas, crie uma função que faça conversão de câmbio. Você deve criar o tipo `Cambio` contendo os value constructors `Euro`, `Real` e `Dollar`. Crie também o tipo `Moeda` que possui os campos `val :: Double` e `cur :: Cambio`. Use record syntax e as taxas de conversão do dia no qual você fez o exercício.

3.20) Crie a função `converterTodosReal` que recebe uma lista de moedas e retorna outra lista de moedas com todos os seus

elementos convertidos para `Real`. Use *list compreenshion*.

3.21) Crie a função `maxMoeda` que recebe uma lista de moedas e retorna o valor máximo absoluto (sem conversão alguma) dentre os campos `val` desta lista. Exemplo:

```
Prelude> maxMoeda [Moeda 3 Real, Moeda 7 Dollar, Moeda 1 Euro]
7
```

Use a função `maximum`.

3.6 CONCLUSÃO

Neste capítulo, vimos como manipular tipos de dados, o conceito de *record syntax* com suas funções de projeção semelhantes aos *getters* da programação orientada a objetos, e o importantíssimo conceito de *pattern matching*, que facilita escrever funções com comportamentos que dependem de *value constructors*.

Este capítulo possui vários exercícios que devem ser feitos, além de ser o primeiro a propor um miniprojeto de uma aplicação feita em Haskell. O entendimento deste capítulo é fundamental para o entendimento do funcionamento desta linguagem. Afinal, temos uma linguagem com um sistema de tipos brilhante que pode ajudar a reduzir muito custos com teste de software, como explicado na introdução deste livro.

Capítulo 4

UM POUCO MAIS SOBRE FUNÇÕES

Em programação funcional, funções são tratadas como valores comuns, assim como `5` é `Int` ou `"Olá"` é uma `String`. Ou seja, funções podem ser passadas como parâmetro e retornadas por outras funções.

Além disso, se uma função espera três parâmetros e você passar dois, não há problema algum nisso. Podemos inclusive criar funções sem um corpo, como vimos até o presente momento.

Este capítulo é dedicado a entendermos mais a fundo como o Haskell trabalha a estrela maior da linguagem: funções. Trabalharemos também um pouco de sintaxe dentro de funções e recursão.

4.1 LAMBDAS

Funções podem ser usadas como valores e não necessariamente em um contexto explícito, como fizemos até aqui. Por exemplo, não é preciso declarar a função em um contexto para usá-la. Em vez disso, é possível usá-las naturalmente como se estivessem declaradas. De uma maneira geral, lambdas têm a seguinte forma:

```
\p1 p2 p3 p4 ... pn -> EXPR(p1, p2, p3, p4, ... , pn)
```

Isto pode ser lido como: receba os parâmetros `p1` , `p2` , `p3` , `p4` , ... , `pn` , e devolva uma expressão que dependa destes **n** parâmetros. Veja um exemplo:

```
Prelude> (\x -> 2*x) 4
8
```

Ele indica que estamos usando um lambda que recebe um parâmetro `x` e devolve o dobro dele mesmo. Uma expressão lambda só pode ser usada em contextos locais. Veja outro exemplo:

```
Prelude> (\x xs -> x : reverse xs) 'A' "UOIE"
"AEIOU"
```

Este recebe um elemento `x` e uma lista de elementos `xs` , devolvendo o primeiro elemento na frente da lista `xs` em ordem reversa. Todos os conceitos apresentados neste capítulo valem sempre para expressões lambda.

4.2 FUNÇÕES DE ALTA ORDEM

Como dito na introdução do presente capítulo, funções podem ser passadas como parâmetro ou retornar outras funções. Uma função que possua alguma das duas características citadas é dita como uma **função de alta ordem**, ou *high-order function*.

```
ev :: (Int -> Int) -> Int
ev f = 1 + f 5
```

Isto é uma função de alta ordem. O tipo de `ev :: (Int -> Int) -> Int` consiste em uma entrada do tipo `Int -> Int` e uma saída inteira. Os parênteses indicam que a entrada tem o tipo de uma função em vez de dois inteiros.

A expressão de retorno de `ev` pode ser em um primeiro momento esquisita, porém é de fácil entendimento. Ela significa: calcule a função `f`, que foi recebida como argumento, ao valor `5` e depois some `1`. Considere as funções auxiliares:

```
dobro :: Int -> Int
dobro x = 2*x

triplo :: Int -> Int
triplo x = 3*x
```

Vamos testar a chamada de `ev` no GHCi:

```
Prelude> ev dobro
11
Prelude> ev triplo
16
```

Tal chamada nos mostra que, primeiramente, a função `dobro` foi argumento de `ev` e depois `triplo`. Vamos analisar de perto o que aconteceu com a primeira (a outra chamada segue o mesmo raciocínio).

```
ev dobro = 1 + dobro 5 = 1 + 2*5 = 1 + 10 = 11
```

A função `dobro` foi plugada em `f`, assim, ela é chamada com o argumento `5`, resultando em `10` como retorno. Neste retorno foi somado `1` ao final, dando `11`. Exemplos de funções que retornam outras terão mais sentido nas seções seguintes e serão suprimidos no momento.

Note que é possível usar lambdas em vez das funções `dobro` e `triplo`, como segue:

```
Prelude> ev (\x -> 2*x)
11
Prelude> ev (\x -> 3*x)
16
```

Escrevendo desta maneira, é permitido uma maior produtividade, pois, não será necessário escrever o corpo das funções e recarregar o módulo novamente.

4.3 CURRYING

Currying é uma técnica que consiste em transformar a chamada de uma função (retorno valorado), que recebe múltiplos argumentos, em uma avaliação de uma sequência de funções. Você pode fixar uma quantidade de argumentos e deixar o restante variável.

```
somarTresNum :: Int -> Int -> Int -> Int
somarTresNum x y z = x+y+z

somarCurr :: Int -> Int
somarCurr = somarTresNum 4 5
```

A função `somarCurr` possui fixo os parâmetros `x` e `y` da função `somarTresNum`, deixando `z` livre para variar. Portanto, podemos inspecionar o tipo de `somarCurr`.

```
Prelude>  :t  somarCurr
somarCurr :: Int -> Int
```

Como esperado, o tipo de `somarCurr` é `Int`, pois, apenas uma variável foi mantida livre na definição de `somarCurr`. O interessante é que podemos agrupar as últimas duas flechas usando parênteses no tipo de `somarTresNum`. Veja:

```
somarTresNum :: Int -> Int -> (Int -> Int)
```

Fazendo isso, fica claro que, a partir da função `somarTresNum`, se passarmos dois argumentos a ela, o retorno será uma função de um parâmetro `Int` e seu retorno `Int`,

fazendo com que ela se torne de **alta ordem**. Sempre que houver parênteses encobrindo flechas e tipos, consideraremos que o parâmetro ou o retorno é uma função.

Em suma, sempre que uma função tiver algum parâmetro suprimido, o retorno será uma função. Esta possuirá como parâmetro todas as variáveis livres, e sua avaliação levará em conta todos os parâmetros que foram deixados fixos e os que ficaram livres também.

```
Prelude> somarCurr 1
10
```

Agora vamos analisar um exemplo mais trabalhado, usando tipos e a estrutura de um programa em Haskell.

4.4 EXEMPLOS DE FUNÇÕES DE ALTA ORDEM

Nesta seção, serão apresentados ao leitor alguns exemplos de funções de alta ordem que facilitam o dia a dia do programador que utiliza este paradigma. Também será demonstrada a utilidade dos conceitos de funções de alta ordem e *currying*.

Map/Foldl/Filter

A função `map` tem como objetivo aplicar uma função `f`, recebida via parâmetro, a todos os elementos de uma lista `l`, também recebida via parâmetro. O retorno desta função é uma nova lista, na qual seus elementos são as saídas da função `f`, tendo como argumento os elementos da lista.

```
Aula4> :t map
```

```
map :: (a -> b) -> [a] -> [b]
```

O tipo de entrada da lista deve ser o mesmo tipo da entrada da função e o tipo da saída será o mesmo da saída da função. Usando o conceito de *currying* e *funções de alta ordem*, podemos enxergar o `map` como:

```
map :: (a -> b) -> ([a] -> [b])
```

Ao suprimir a lista de tipo `[a]`, como parâmetro da função `map`, o retorno obtido é uma função do tipo `[a] -> [b]`. Ou seja, sua função foi levantada de `a -> b` para `[a] -> [b]`, isto é, a função que inicialmente trabalhava com estruturas simples passa agora a trabalhar com listas.

```
Aula4> map (+2) [1..5]
[3,4,5,6,7]
```

Neste exemplo, a função `map` recebe via parâmetro a função `(+2)`, que é um *currying* da função `(+)`, tendo o valor `2` fixo em seu segundo argumento, e distribui essa função a todos os elementos da lista `[1,2,3,4,5]`. Assim, é efetuado: `1+2`, `2+2`, `3+2`, `4+2` e `5+2`. Note que o uso do `map` se assemelha ao *list compreeshion* visto no primeiro capítulo.

Para a função `foldl`, devem ser passados uma função `f` com dois parâmetros e também um valor inicial. Essa função dobra a lista começando da esquerda, isto é, a função `f` é aplicada ao valor inicial e ao primeiro elemento da lista. O retorno dela mais o segundo elemento devem ser os parâmetros para a nova aplicação de `f` até acabarem os elementos da lista. Você pode pensar que a função `foldl` se comporta como um acumulador se comparada ao paradigma imperativo.

```
Aula4> :t foldl
```

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

O primeiro parâmetro (de `foldl`) deve ser uma função na qual sua primeira entrada seja o mesmo tipo `b` que o valor inicial (segundo parâmetro de `foldl`), e sua segunda entrada com o mesmo tipo `a` dos elementos contidos na lista do terceiro parâmetro (de `foldl`).

```
Aula4> foldl (+) 0 [1..4]
10
```

A função soma `(+)` terá como parâmetro o valor inicial `0` e o primeiro valor da lista `1`. A função soma será aplicada a estes dois parâmetros, resultando em 1. O valor 1 do acúmulo (`0+1`) será um novo parâmetro, junto com o valor `2` da lista, e ambos sofrerão a aplicação da soma novamente, resultando em `3`. Esse valor `3`, juntamente com o valor `3` da lista, sofrerá a aplicação de `(+)` novamente, resultando em `6`. O valor `6` e o elemento `4` da lista sofrerão pela última vez a aplicação de `(+)`, resultando em `10`.

```
(+) 0 1 [2,3,4]
(+) 1 2 [3,4]
(+) 3 3 [4]
(+) 6 4 []
10
```

Note que, em Haskell, `0 + 1` é a versão infixa de `(+) 0 1`, e ambas produzem o mesmo valor, `1`.

```
Aula4> foldl ( \xs   x -> x:xs )  [] ''FATEC''
''CETAF''
```

Isto produzirá as seguintes chamadas:

```
'F':[] "ATEC"
'A':'F':[] "TEC"
'T':'A':'F':[] "EC"
```

```
'E':'T':'A':'F':[] "C"
'C':'E':'T':'A':'F':[] ""
''CETAF''
```

Observe que `(\xs x-> x : xs )` é um *lambda*, que nada mais é do que uma função sem corpo. Esta função recebe uma lista `xs` e um valor `x` , e retorna o elemento `x` inserido à frente da lista `xs` .

O `filter` é uma função que recebe uma outra função `f` de retorno booleano e uma lista de elementos. Ele retorna uma outra lista contendo os elementos que foram argumentos de `f` e que tiveram `True` como retorno.

```
Aula4> filter (>0) [-4..4]
[1,2,3,4]
```

Neste exemplo, é filtrado todos os elementos maiores que zero. A função recebida `(>0)` se equivale ao *lambda* `\x-> x > 0` , que recebe um valor `x` . Este retorna `True` caso seja maior que zero, e `False` caso contrário. Os elementos `[1,2,3,4]` são todos que satisfazem a condição da função `(>0)` e, no caso, são retornados pela função `filter` .

```
Aula4> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

O tipo da função `filter` nos diz que a função do primeiro parâmetro precisa retornar uma expressão booleana. A entrada da função do primeiro parâmetro deve ter um tipo `a` , que deve ser o mesmo tipo dos elementos do segundo parâmetro, de tipo `[a]` .

Função .

A composição de funções, introduzida no segundo capítulo, é

uma noção muito importante em programação funcional e merece toda atenção aqui nesta seção. A composição de funções é algo cotidiano na vida do programador. É o simples ato de chamar duas funções, ou mais, sendo que o retorno de uma é o argumento da outra.

Para acompanhar os exemplos a seguir, vamos considerar as funções:

```
traseira :: String -> String
traseira [] = []
traseira (x:xs) = xs

contar :: String -> Int
contar = length
```

A primeira função retorna uma lista vazia, caso o argumento seja vazio, e o fim da lista `xs`, caso o argumento tenha pelo menos um elemento (ignorando assim o primeiro elemento e devolvendo o resto). A função `contar` é apenas uma renomeação da função `length`, que conta elementos de uma lista de qualquer tipo. Nesse caso, deixamos a entrada como `String` por questões didáticas.

Na expressão `contar(traseira "Haskell")`, a função `traseira` será executada em um primeiro momento e terá como retorno `"askell"`. Após este passo, obtemos a chamada `contar "askell"` - note que o retorno de `traseira` é o argumento de `contar` - que produzirá o retorno `6`.

O exemplo parece simples, porém, na programação funcional, tem um motivo teórico muito bem fundamentado. Antes de continuar, note que `contar(traseira "Haskell")` se equivaleria a `contar(traseira("Haskell"))`, que é um estilo

corriqueiro nas linguagens de programação imperativa.

A expressão em destaque `contar(traseira "Haskell")` pode ser melhorada e escrita de uma forma mais próxima da matemática, usando a função infixa `.` (ponto). O nome desta função tem motivação na notação matemática para composição de funções.

```
Prelude> (contar . traseira) "Haskell"
6
```

Essa função `.` é de alta ordem. Basta checar seu tipo:

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Ela diz o seguinte: essa função recebe duas outras de tipos `(b -> c)` e `(a -> b)`, retornando uma outra de tipo `(a -> c)`. Note que `a`, `b` e `c` são qualquer tipo da linguagem (`Int`, `Double`, `[Int]`, `String`, `(Int, String)` etc.).

Retomando a frase "o retorno de uma função deve ser o argumento da outra", é possível ver que o tipo incumbido de realizar esta façanha é o `b`, pois é a saída do segundo parâmetro de tipo `(a -> b)` e a entrada do primeiro parâmetro de tipo `(b -> c)`. Portanto, o tipo da função composta (retorno de `.`) se dá pela entrada da primeira função `a` e pela saída da segunda `c`.

Vamos analisar agora o exemplo da expressão `(contar . traseira) "Haskell"`:

```
Prelude> :t traseira
tail' :: String -> String
Prelude> :t contar
contar :: String -> Int
```

Podemos agora inspecionar a função composta dada pela expressão `contar . traseira`. O primeiro argumento de `.` é a função `contar`, portanto, `b=String` e `c=Int`, pois, o tipo de `contar` é `String -> Int`, e a função conta com um argumento de tipo `b -> c`.

Analogamente, o segundo argumento de `contar . traseira` é a função `traseira`, que tem o tipo `String -> String`. Como a função `.` espera algo do tipo `a -> b`, temos que `a=String` e `b=String`, concluindo que o tipo do retorno de `contar . traseira` é `String -> Int`, já que o retorno de `.` é `a -> c`.

É bom observar `b=String`, nos dois casos, fazendo que com que a composição funcionasse corretamente (saída da segunda função é a entrada da primeira). Se tentássemos compor `traseira . contar`, obteríamos um erro de compilação, pois `traseira` espera uma `String` e o retorno de `contar` é um `Int`.

Função $

A função infixa `$` recebe uma função e um valor, e aplica a função neste valor. Veja um exemplo:

```
Prelude> contar $ "Ola"`
3
```

Aparentemente, não há diferença alguma com uma chamada de função simples:

```
Prelude> contar "Ola"`
3
```

Porém, a diferença está quando usamos os parênteses para dar

prioridade nas operações. Quando calculamos o tamanho de uma lista usando a concatenação, devemos, por exemplo, usar parênteses:

```
Prelude> contar ("Ola" ++ "Alo")
6
```

Fazemos isso para indicar que a concatenação será efetuada e seu retorno entrará na função `contar`.

A função `$` é uma maneira fácil de se livrar da poluição causada pelo uso excessivo de parênteses - como é habitual em muitas linguagens -, deixando o código mais claro.

```
Prelude> contar $ "Ola" ++ "Alo"
6
```

Como `$` possui uma precedência à direita maior que `++`, primeiro será calculado `"Ola" ++ "Alo"`, e depois `contar "OlaAlo"`, retornando `6`. Inspecionando o tipo de `$`, podemos ver:

```
Prelude> :t ($)
($) :: (a -> b) -> a -> b
```

Ela recebe uma função que recebe uma função de tipo `a` e retorno `b`, e um valor de tipo `a` devolvendo um valor de tipo `b`. Em nosso exemplo, o primeiro argumento de `$` é a função `contar`. Desta forma, `a` é uma `String` e `b` um `Int`, pois o tipo de `contar` é `String -> Int`. Já o segundo argumento é a `String "OlaAlo"`, tendo `6` como retorno e este do tipo `Int`.

Tanto `.` quanto `$` são funções infixas definidas na base do GHC. O próximo exemplo será criado por mim e ajudará bastante em alguns conceitos que serão abordados nos próximos capítulos.

Função |>

Podemos nos valer do poder que a linguagem Haskell nos oferece para criar a seguinte função infixa:

```
(|>) :: a -> (a -> b) -> b
(|>) x f = f x

infixl 9 |>
```

Vale ressaltar que o operador que acabamos de definir, existe na linguagem e está definido no módulo `Data.Function` com o nome `(&)`. Para não confundir o leitor, pois `(&)` pode ser facilmente confundido com `(&&)` (`and` lógico), usaremos este `(|>)` que acabamos de definir (ou seja, `(|>) = (&)`). Ela recebe um valor de tipo `a`, e uma função que recebe `a` e retorna `b`, retornando assim um valor de tipo `b`. Note que essa função é apenas `$` com os parâmetros trocados de ordem.

O `infixl` indica alta precedência à esquerda. A linguagem de programação funcional para **front-end** `Elm` usa muito esta função e, em seu ambiente, este conceito é chamado de **pipelining**. Aqui em Haskell, este operador terá mais utilidade mais à frente, porém usando-o é possível reescrever a seguinte função:

```
func :: String -> String
func x = x ++ (tail (take 3 (reverse x)))
```

Da seguinte forma:

```
funcI :: String -> String
funcI x = x
    |> reverse
    |> take 3
    |> tail
    |> (x ++)
```

Isso tem um estilo parecido com as linguagens imperativas, dando a impressão de que está sendo executada uma instrução por linha. Primeiramente, é importante frisar que `func e funcI` são as mesmas funções escritas, mas em estilos diferentes. A função `func` foi escrita usando composição de funções, como de costume.

Nela, concatenaremos a expressão `(tail (take 3 (reverse x)))` ao parâmetro recebido `x`. Vale ressaltar que `(tail (take 3 (reverse x)))` pode também ser escrito como `(tail . (take 3) . reverse) x`, ou `tail . (take 3) . reverse $ x`. Essa expressão nos diz que o retorno de `reverse` entrará na função `take 3`. Graças ao conceito de **currying**, o retorno dela entrará em `tail`, finalizando com a concatenação.

Em `funcI`, temos o mesmo fenômeno, porém em outra escrita, sendo esta mais parecida com linguagens imperativas. Nela, temos que o parâmetro `x` entra em `reverse`, e seu retorno é passado para `take 3`. Lembre-se sempre do *currying* aqui, depois para `tail` e depois para a expressão `(x ++)`, que é a concatenação de `x` com o valor passado em cada função desde o começo (note o uso do *currying* aqui também).

Este último exemplo foi muito rico e usou todos os conceitos discutidos neste capítulo. É de extrema importância o entendimento da função infixa `|>`. Com ela, o leitor estará preparado para encarar o conceito de **Monads**, que é tão importante neste linguagem. A partir dele, consguiremos nos conectar com o mundo real, ou seja, aplicações web, arquivos, banco de dados etc.

4.5 SINTAXE EM FUNÇÕES

Os *guards* são uma maneira de testar várias condições em uma função, de maneira similar a um `if` encadeado. Por exemplo, se quisermos calcular o IMC de uma pessoa e, a partir deste valor, mostrar uma mensagem na tela indicando se ela está acima do peso ou não, é possível usando *guards*. Podemos escrever as condições de uma maneira limpa.

```
imc p a
| p/(a*a) <= 18.5 = "Abaixo do peso"
| p/(a*a) < 25.0 = "Peso ideal"
| p/(a*a) <= 30 = "Acima do peso"
| otherwise = "Obesidade"
```

Um possível tipo da função anterior é `Double -> Double -> Double` . Os parâmetros `p` e `a` representam peso e altura. A expressão `p/(a*a)` representa o cálculo do IMC. A partir deste cálculo, as condições são verificadas em ordem até que alguma seja `True` e o retorno da função (mensagem) será executada.

Caso a condição seja `False` , a próxima condição será verificada até chegar ao `otherwise` , que sempre será `True` . A cláusula `where` pode ajudar a facilitar a escrita da função de IMC.

```
imc p a
| valorImc <= 18.5 = "Abaixo do peso"
| valorImc < 25.0 = "Peso ideal"
| valorImc <= 30 = "Acima do peso"
| otherwise = "Obesidade"
where
 valorImc = p/(a*a)
```

O uso do `where` ajuda a não escrever a expressão `p/(a*a)` em toda condição. Note que, para cada padrão do *pattern matching*, é possível ter *guards* próprios.

4.6 RECURSÃO

Recursão é um método de resolução de problemas que consiste na solução de pequenas instâncias do problema até achar a solução global. A recursão precisa de uma condição de base (de parada ou inicial) para que não se caia em loops infinitos.

Em Haskell, é uma técnica fundamental para resolver problemas, pois, não há instruções de loop como `repeat` ou `for`.

```
fat n
| n <= 1 = 1
| otherwise = n*fat(n-1)
```

Essa expressão realiza o cálculo de um fatorial que é um problema recursivo. A condição base é quando temos `fat 1 = 1`. Como `fat 0 = 1` e não queremos que a função exploda com valores negativos, a condição `n <= 1` foi escolhida (por enquanto, é a melhor opção).

Um possível tipo para a função anterior é `Int -> Int`. Para se reverter a ordem de uma `String`, é possível proceder de forma recursiva, como mostra o trecho de código adiante.

As operações são realizadas conforme a ordem de chamada da função. Por exemplo, se calcularmos o fatorial de `5`:

```
Prelude> fat 5
120
```

A primeira expressão a ser calculada é `fat 5 = 5*fat 4`, e a próxima chamada será `fat 4`. O quadro seguinte mostra a ordem das operações:

```
fat 5 = 5*fat 4
```

```
fat 4 = 4*fat 3
fat 3 = 3*fat 2
fat 2 = 2*fat 1
```

Na memória, as chamadas são empilhadas, criando algo parecido como o quadro:

```
fat 1
fat 2
fat 3
fat 4
fat 5
```

Cada retorno não recursivo da função faz com que a chamada seja eliminada da memória. Em nosso caso, a chamada `fat 1` corresponde a um retorno não recursivo conforme definido, logo, `fat 1` retornará `1` e a próxima expressão a ser calculada está em `fat 2`. O quadro a seguir mostra a ordem de cálculo das expressões.

```
fat 1 = 1
fat 2 = 2*fat 1 = 2*1 = 2
fat 3 = 3*fat 2 = 3*2 = 6
fat 4 = 4*fat 3 = 6*4 = 24
fat 5 = 5*fat 4 = 24*5 = 120.
```

O próximo exemplo ilustra chamadas recursivas usando o conceito de *pattern matching*.

```
reverse' :: String -> String
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Se a função `reverse'` for chamada com um argumento `"Haskell"`, obteremos `"lleksaH"`. O padrão `(x:xs)` quebra a String `"Haskell"` em `'H':"askell"`. A expressão da função `reverse'` concatena a chamada da função novamente com o argumento `"atec"`, concatenando a String `"F"` (equivalente a

`['F']`) ao final, trocando assim a ordem da palavra.

A função procede recursivamente até achar a condição de parada `reverse' []=[]` , que quer dizer que se a String vazia for encontrada, será retornada ela mesma, não modificando a palavra a ser revertida e encerrando o processo recursivo.

Analogamente ao exemplo anterior, podemos ver o mesmo processo para esta função:

```
reverse' "Haskell" = reverse' "askell" ++ ['H']
reverse' "askell" = reverse' "skell" ++ ['a']
reverse' "skell" = reverse' "kell" ++ ['s']
reverse' "kell" = reverse' "ell" ++ ['k']
reverse' "ell" = reverse' "ll" ++ ['e']
reverse' "ll" = reverse' "l" ++ ['l']
reverse' "l" = reverse' [] ++ ['l']
```

E na memória:

```
reverse' []
reverse' "l"
reverse' "ll"
reverse' "ell"
reverse' "kell"
reverse' "skell"
reverse' "askell"
reverse' "Haskell"
```

Como `reverse' []` é uma chamada não recursiva (caso base), obtemos o quadro a seguir contendo a ordem do cálculo das expressões:

```
reverse' [] = []
reverse' "l" = reverse' [] ++ ['l'] = "l"
reverse' "ll" = reverse' "l" ++ ['l'] = "ll"
reverse' "ell" = reverse' "ll" ++ ['e'] = "lle"
reverse' "kell" = reverse' "ell" ++ ['k'] = "llek"
reverse' "skell" = reverse' "kell" ++ ['s'] = "lleks"
reverse' "askell" = reverse' "skell" ++ ['a'] = "lleksa"
```

```
reverse' "Haskell" = reverse' "askell" ++ ['H'] = "lleksaH".
```

Observe que `reverse'` e `reverse` são iguais. O conceito de recursão que acabamos de ver é de importância na substituição de blocos de repetição (`do` , `while` , `for`) usados em outras linguagens de programação.

4.7 MINIPROJETO: RH DE UMA EMPRESA DE TI

Neste capítulo, continuaremos o projeto iniciado, mantendo as funções, o mesmo arquivo `Projeto.hs` e também o mesmo cabeçalho. Aqui será implementado funções para contratar um funcionário novo, calcular a média salarial de uma lista de pessoas, contratação de vários funcionários em cargos iniciais e uma rotina para promoção.

```
contratarInicial :: String -> Pessoa
contratarInicial = Pessoa Estagiario

mediaSalarial :: [Pessoa] -> Double
mediaSalarial ps = (foldl calculo 0 ps) / (fromIntegral $ length
ps)
                   where
                       calculo salario pessoa = salario + verSala
rio pessoa

contratarVariosEstag :: [String] -> [Pessoa]
contratarVariosEstag ps = map (contratarInicial Estagiario) ps

rotinaPromocao :: Pessoa -> String
rotinaPromocao p = p
                |> promover
                |> verFolha
```

A função `contratarInicial` age como se fosse um

construtor – sim, aquele velho conhecido da programação orientada a objetos. O `cargo` sempre será `Estagiario`. Esta função é de alta ordem, pois não recebe entrada alguma e retorna uma função do tipo `String -> Pessoa`.

Note que aqui está sendo usado o conceito de *currying*, do fato de `Pessoa Estagiario` não ser do tipo `pessoa`, mas sim `String -> Pessoa`. Ou seja, para a chamada ser completa e o tipo ser `Pessoa`, necessita-se do nome.

A função `contratarVariosEstag` usa o `map` para jogar a função `contratarInicial Estagiario`. Não esqueça do *currying*, para dentro desta lista de `String ps`. Do funcionamento do `map`, todo elemento `String` da lista `ps` será argumento de `contratarInicial Estagiario`, e seu retorno será uma `Pessoa` contratada pela empresa.

Na função `mediaSalarial`, a cláusula `where` é usada de forma a facilitar a sua escrita. O tipo da função interna cálculo é `Double -> Pessoa -> Double`, que é exatamente um tipo de função exigida pelo `foldl`. Lembre-se de que tal função deve receber algo do mesmo tipo que `0` e um elemento de mesmo tipo que a lista `ps`, como também deve retornar algo de mesmo tipo que `0`.

A função `calculo` soma o salário que vem do primeiro argumento do `foldl` com `verSalario pessoa`, que verifica o salário de uma pessoa que é elemento do segundo argumento do `foldl` (no caso, `ps`). A soma realizada pelo `foldl` é dividida pela quantidade de elementos dada pela expressão `fromIntegral $ length ps`. Esta converte o tamanho inteiro da lista para um `Double`, usando a função `fromIntegral`. Note o uso de `$` para

evitar os parênteses.

Finalmente, a rotina de promoção simplesmente promove um funcionário e exibe sua nova folha. Como duas ações estão sendo realizadas, a composição é uma das alternativas de escrita, porém, usamos o `|>`. A função `rotinaPromocao` é lida como: "Promova p e depois veja sua folha". A mesma função pode ser escrita também usando `.` e `$`, como:

```
rotinaPromocao :: Pessoa -> String
rotinaPromocao p = verFolha . promover $ p
```

Isso reforça que `|>` dá um som mais parecido com as linguagens imperativas.

4.8 EXERCÍCIOS

4.1) Faça uma função que retorne a média de um `[Double]`, usando `foldl`.

4.2) Faça uma função que receba uma `[String]` e retorne todos os elementos palíndromos. Ver exercício 3.7.

4.3) Implemente uma função que filtre os números pares e outra que filtre os ímpares de uma lista recebida via parâmetro.

4.4) Filtre os números primos de uma lista recebida por parâmetro.

4.5) Implemente uma função que receba uma lista de inteiros e retorne o dobro de todos, eliminando os múltiplos de 4.

4.6) Faça uma função `func` que receba uma função `f` de tipo `(String -> String)`, e uma `String s` que retorna o reverso

de `s` concatenado com aplicação da função `f` em `s`.

4.7) Crie um tipo `Dia` contendo os dias da semana. Faça uma função que receba uma lista de `Dias` e filtre as `Terças`.

4.8) Implemente o tipo `Dinheiro` que contenha os campos `valor` e `correncia` (`Real` ou `Dolar`), e uma função que converta todos os "dinheiros" de uma lista para dólar (e outra para real). Com isso, implemente funções para:

- Filtrar todos os `Dolares` de uma lista de `Dinheiro`.
- Somar todos os `Dolares` de uma lista.
- Contar a quantidade de `Dolares` de uma lista.

4.9) Usando a função `foldl`, crie lambdas para:

- Contar números negativos de uma lista de `Int`.
- Contar letras 'P' de uma `String`.
- Para contar `Sabados` em uma lista de um `[DiaSemana]`.
- Para, a partir de uma lista de `[DiaSemana]`, retornar a soma dos dias. Exemplo: `[Segunda, Segunda, Quarta]` deve retornar `5`. Use uma função auxiliar para converter `DiaSemana` para `Int`.

4.10) Reescreva os exercícios anteriores usando: `.`, `$` e `|>`.

4.9 CONCLUSÃO

Este capítulo foi fundamental para entendermos como escrever um programa usando o paradigma funcional. Nele, foram mostradas as armas para escrever código da maneira que lhe for

mais conveniente. Também foi visto que podemos ter estruturas condicionais e de loop usando **guards** e **recursão**.

Aprendemos também outra alternativa aos **list comprehensions** do *capítulo 2* usando os famosos `map` , `filter` e `foldl` . Foi mostrado ao leitor também como a linguagem trata as funções no estudo dos conceitos de **funções de alta ordem** e **currying**.

Pode-se se dizer que, até aqui, se você compreendeu os conceitos, rodou os miniprojetos e fez os exercícios, você possui um conhecimento básico da linguagem. Você sabe ver um tipo, identificar tipos de funções, escrever funções e entender o funcionamento básico do paradigma. Os próximos capítulos lhe deixarão numa posição de nível intermediário na linguagem. Aperte os cintos!

Capítulo 5

POLIMORFISMO PARAMÉTRICO

5.1 TIPOS COM PARÂMETROS

Os tipos com parâmetros (ou em inglês, *type parameters*) são o equivalente aos generics do Java aqui no Haskell. É uma forma de se implementar o conceito de polimorfismo paramétrico, ou seja, chamaremos de contêiner ou caixa um tipo que possuir um *type parameter*, por causa de sua semelhança com uma caixa de tipos. Ele poderá guardar um ou mais elementos de vários tipos, sem a necessidade de especificar qual.

Um exemplo real são as listas, corriqueiras no dia a dia do programador. Elas podem ser enxergadas como caixas contendo vários elementos de um mesmo tipo. Graças a este conceitos, podemos ter no mesmo programa listas de inteiros `[Int]`, de booleanos `[Bool]`, caracteres `String` ou `[Char]` e afins. A ideia é que possamos operar listas (ou qualquer contêiner) sem a necessidade de olhar "o que há dentro".

Ou seja, a partir do momento em que um tipo vai para dentro de um contêiner, estaremos preocupados com o contêiner em si, e não com o que foi para dentro. Podemos também transformar o

que está dentro do contêiner, mas mesmo assim, não importa quem ou o que está ali. Vamos usar um exemplo mais simples ainda do que uma lista, de modo a fazer você se ambientar com o conceito.

Uma `Coisa` representará algo que podemos guardar nenhum, um ou dois elementos de um mesmo tipo:

```
data Coisa a = UmaCoisa a | DuasCoisas a a | ZeroCoisa
```

O tipo anterior possui um *type parameter* `a` que significa que qualquer tipo poderá ser passado ao contêiner `Coisa`. O *value constructor* `DuasCoisas` carrega dois campos do tipo `a`, enquanto o *value constructor* `UmaCoisa` carrega um, e o `ZeroCoisa` é apenas um *value constructor* sem campos.

```
:t (DuasCoisas "OLA" "Mundo")
(DuasCoisas "OLA" "Mundo") :: Coisa String

:t (UmaCoisa True)
(UmaCoisa True) :: Coisa Bool
```

Observe que poderemos ter contêiners de vários tipos ao mesmo tempo, por exemplo, `Coisa String`, `Coisa Bool`, `Coisa Int`, `Coisa Pessoa` etc., assim como temos com as listas. Aqui o que importa é o contêiner `Coisa` e não o que há dentro.

Ao trabalharmos com contêiners, o comando `:kind` do GHCi vai indicar quantos *types parameters* existem em seu tipo. Quanto mais existirem, mais complexo será trabalhar com ele.

```
Prelude> :kind Int
Int :: *

Prelude> :kind Coisa
Coisa :: * -> *
```

```
Prelude> :kind []
[] :: * -> *
```

O `kind` enxerga os tipos como funções de tipos. Como `Int` não possui parâmetro de tipo, seu `kind` é `*` (podemos pensar como **kind 1**, como abuso de linguagem) ao passo que `Coisa` possui um parâmetro de tipo, logo seu `kind` é `* -> *` (ou **kind 2**, abusando mais um pouco da linguagem). Divagando um pouco, se declarássemos o seguinte tipo:

```
data Foo a b = Foo a b
```

E também inspecionássemos seu `kind`, obteríamos `* -> * -> *`, pois `Foo` possui dois parâmetros de tipo `a` e `b`. Fazendo um comparativo com o Java, a classe `HashMap` (estrutura similar às listas que possuem índices de quaisquer tipo) é um exemplo de um tipo de `kind * -> * -> *`.

Em Haskell, há vários tipos de `kinds`, porém o foco maior deste livro será em tipos de `kind *` e `* -> *`. Isto é, nossas caixas poderão guardar elementos de mesmo tipo apenas (e não de tipos diferentes, como `Foo`).

Se tentarmos inspecionar o tipo de `DuasCoisas True 'A'`, obteremos erro de compilação, pois, `DuasCoisas` possui dois campos genéricos de mesmo tipo. Em contrapartida, veja:

```
Prelude> :t (Foo True 'A')
(Foo True 'A') :: Foo Bool Char
```

Esse código ocorre sem erros, por ser um contêiner de `kind * -> * -> *`.

Podemos usar o conceito de polimorfismo paramétrico para

criar tipos recursivos (tipos que possuem algum campo contendo ele próprio) como listas ligadas e árvores. Uma árvore binária pode ser escrita usando tipos polimórficos, por exemplo:

```
data Arvore a = Nulo |
                Leaf a |
                Branch a (Arvore a) (Arvore a) deriving Show
```

O tipo `Arvore` possui três *value constructors*: `Nulo`, que não possui nenhum campo; `Leaf a`, que possui um campo de tipo `a`; e `Branch a (Arvore a) (Arvore a)`, que possui um campo `a`, um campo para representar o nó do filho esquerdo de tipo `Arvore a` (que pode ser novamente `Nulo`, `Leaf` ou `Branch`) e um campo para representar o nó do filho direito de tipo `Arvore a`.

Os campos de tipo `a` representam o elemento a ser guardado na árvore, enquanto os campos de tipo `Arvore a` fazem a continuação da estrutura, tanto para esquerda quanto para a direita.

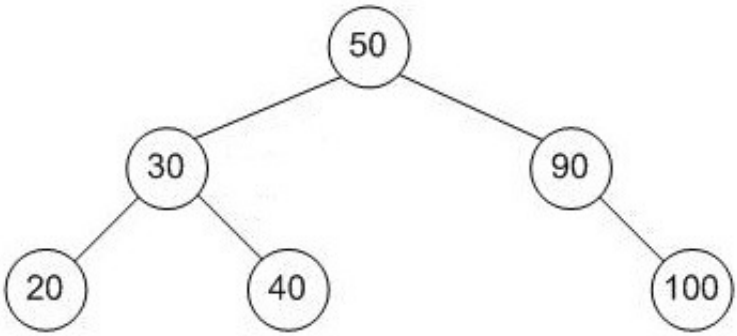


Figura 5.1: Árvore binária

A árvore anterior pode ser representada pela expressão:

```
Branch 50 (Branch 30 (Leaf 20) (Leaf 40)) (Branch 90 Nulo (Leaf 1
00))
```

O percurso em ordem, por exemplo, pode ser escrito com ajuda da *recursão* e do *pattern matching*:

```
emOrdem :: Arvore a -> [a]
emOrdem (Branch x l r) = emOrdem l ++ [x] ++ emOrdem r
emOrdem (Leaf x) = x
emOrdem Nulo = []
```

Note que a condição de parada da função anterior se baseia no fato do *pattern matching* encontrar um `Leaf` ou `Nulo` interrompendo o processo recursivo dos `Branch` es. A função `emOrdem` é polimórfica e, como é possível notar, não utiliza o tipo `a` associado ao contêiner `Arvore`. Ou seja, funções polimórficas não podem ver ou operar sobre *type variables*.

No caso da função `emOrdem`, sua entrada é do tipo `Arvore a`, logo, essa função valerá para qualquer tipo `a`, e seu retorno `[a]` indica que será uma lista desse mesmo tipo. A conclusão maior que podemos tirar é que `emOrdem` troca o contêiner sem mexer nos elementos.

Pense em um presente que você coloca em uma caixa azul, porém você não gostou da cor da caixa e resolve trocar para uma caixa vermelha. Você apenas trocou a caixa - essa é a ideia com funções deste tipo.

5.2 RESTRIÇÃO DE TIPOS EM FUNÇÕES

Às vezes, é necessário operar com o elemento que possui um tipo genérico `a`, por exemplo. Em uma função, é necessário exibir um elemento dentro do contêiner. E se criássemos esta função da

seguinte forma:

```
foo :: a -> String
foo x = "O valor de tipo a é: " ++ show x
```

Teríamos um problema, pois a entrada aceita um tipo qualquer e este pode não ter uma instância de `Show` em sua declaração. Em outras palavras, a expressão `deriving Show`.

Para resolver isto, temos de avisar que não pode ser qualquer `a` criando uma restrição que avisa ao compilador que tipos serão barrados fora desse contexto em tempo de compilação. Veja um exemplo:

```
foo :: Show a => a -> String
foo x = "O valor de tipo a é: " ++ show x
```

Com esta função, conseguimos ter uma dica do que pode ser `a` como indicado na restrição `Show a`. Ou seja, tipos que possuem a expressão `deriving Show` serão aceitos pela função `foo`, em tempo de compilação, ao passo que funções sem esta palavra não serão aceitas. Há um nome específico para `Show`, **typeclasses**, que será abordado a seguir.

5.3 CLASSES DE TIPOS

Um *typeclass* (classe de tipos) é uma estrutura do Haskell que habilita um operador (ou mais) ou uma função (ou mais) a ser usada de forma diferente, dependendo de um *type parameter*. Para cada tipo, uma instância deverá ser definida e, com ela, a definição do operador ou da função que o *typreclass* provê.

Fazendo uma analogia com o paradigma orientado a objetos, um *typeclass* se assemelha a uma interface, e o operador/função

inerente a ele se assemelha a um método abstrato. Se usarmos o tipo:

```
data Coisa a = Nada | UmaCoisa a | DuasCoisas a a
```

Não poderíamos mostrá-lo na tela nem comparar (usando a função `==`). Para mostrar qualquer tipo na tela, seria necessário o uso do *typeclass* `Show` e, para comparar, usaríamos o `Eq`. Reescrevendo:

```
data Coisa a = Nada | UmaCoisa a | DuasCoisas a a deriving (Show,
 Eq)
```

Temos agora que o tipo `Coisa a` pode ser mostrado na tela e é comparável. O Haskell provê para o tipo `Coisa a` a seguinte regra de igualdade: dois valores do tipo `Coisa a` são iguais se, e somente se, seus *values constructors* são iguais e os campos também.

```
DuasCoisas 5 5 == DuasCoisas 5 5
True

DuasCoisas 7 3 == DuasCoisas 3 7
False
```

Classe Eq

Esta regra de igualdade para o tipo em questão vem de forma gratuita, apenas usando `deriving Eq` na definição. Se a regra de igualdade para este tipo for criada pelo leitor, a instância de `Eq` para o tipo `Coisa a` se faz necessária.

```
instance Eq a => Eq (Coisa a) where
    (DuasCoisas x1 y1) == (DuasCoisas x2 y2) = x1 == y2
    (UmaCoisa x) == (UmaCoisa y) = x==y
    Nada == Nada = True
    _ == _ = False
```

Note que o tipo `a` dentro da estrutura `Coisa` deve ser comparável (instância de Eq) como mostra o trecho `Eq a => Eq (Coisa a)`. Não pode ser qualquer `a`, sendo assim, faz-se necessária a restrição `Eq a`.

Se o tipo `a` não for uma instância de `Eq`, não há como comparar os elementos de dentro de `Coisa`, logo, o tipo `Coisa a` não pode ser comparável. Logo, é preciso apagar o `Eq` da palavra `deriving`. Veja:

```
data Coisa a = Nada | UmaCoisa a | DuasCoisas a a deriving Show
```

Ainda nesse exemplo, mudou-se a regra de igualdade apenas para o *value constructor* `DuasCoisas` e, com isso, temos:

```
DuasCoisas 7 3 == DuasCoisas 3 7
True
```

A operação binária diferente (`/=`) vem de graça com a negação da igualdade (`==`).

Podemos também escrever uma instância de `Show` para `Coisa a`, diferente da que foi provida pelo Haskell. Veja o exemplo:

```
Prelude> DuasCoisas 7 9
DuasCoisas 7 9

Prelude> Nada
Nada

Prelude> UmaCoisa True
UmaCoisa True
```

O valor, por exemplo, `DuasCoisas 7 9` apareceu porque `Coisa` possui a palavra `deriving Show`, que cria a instância de `Show` para `Coisa a` automaticamente.

Classe Show

A classe `Show` é usada para a saída de qualquer dado. Nos *frameworks web* em Haskell, usamos o `Show` para injetar saídas das funções em um template HTML. Há uma instância "padrão" de `Show` que fornece, na tela, `DuasCoisas 7 9`, por exemplo. Podemos mudar isso removendo o `Show` do `deriving`:

```
data Coisa a = Nada | UmaCoisa a | DuasCoisas a a
```

E também criando a instância:

```
instance Show a => Show (Coisa a) where
    show Nada = "Nadinha..."
    show (UmaCoisa x) = "Coisa com o elemento " ++ show x
    show (DuasCoisas x y) = "Coisa com os elementos " ++ (show x)
 ++ " e " ++ (show y)
```

Note que a classe `Show` exige a definição da função `show` de tipo `a -> String`. Testando a instância anterior:

```
Prelude> DuasCoisas 7 9
Coisa com os elementos 7 e 9

Prelude> Nada
Nadinha...

Prelude> UmaCoisa True
Coisa com o elemento True
```

Desta forma, a instância padrão do Haskell foi modificada.

Classe Read

Este *typeclass* é outro de extrema importância, pois converte uma String em um tipo `a` definido em sua instância. Ele ajuda muito na hora de fazer conversão de dados. Por exemplo, podemos converter algo digitado, via entrada padrão, e transformar isto em

qualquer tipo (é mais usual com números).

A criação de uma instância para este tipo pode ser bem complicada. Mostraremos o seu uso acompanhado do `deriving` apenas (instância padrão do Haskell). Vamos usar nosso tipo como teste:

```
module Cap5 where

data Coisa a = Nada | UmaCoisa a | DuasCoisas a a deriving Read

lerCoisa :: Coisa Int -> Coisa Int
lerCoisa Nada = UmaCoisa 0
lerCoisa (UmaCoisa x) = UmaCoisa (x+1)
lerCoisa (DuasCoisas x y) = DuasCoisas (2*x) (y-3)
```

E no GHCi teremos:

```
*Cap5> lerCoisa (read "Nada")
UmaCoisa 0
*Cap5> lerCoisa (read "UmaCoisa 8")
UmaCoisa 9
*Cap5> lerCoisa (read "DuasCoisas 1 2")
DuasCoisas 2 (-1)
```

Perceba que a entrada de `lerCoisa` foi obtida a partir de uma `String` após a conversão feita por `read`. O problema é que, se a `String` não estiver formatada corretamente, obteremos um erro de execução que não é bem-vindo por aqui.

```
*Cap5> lerCoisa (read "DuasCoisas 1 2 3")
*** Exception: Prelude.read: no parse.
```

A *exception* `no parse` indica que a conversão da `String` `"DuasCoisas 1 2 3"` em um tipo `Coisa Int` falhou.

5.4 OUTRAS CLASSES

Outras classes importantes são: `Num`, `Fractional`, `Real`, `Integral`, `Enum`, `Ord` e `Bounded`.

Classe Num

A classe `Num` é definida por:

```
class  Num a  where
    (+), (-), (*)       :: a -> a -> a
    abs                 :: a -> a
    signum              :: a -> a
    fromInteger         :: Integer -> a
```

Ela provê uma interface para operações aritméticas básicas, como `+` (mais), `-` (menos) e `*` (vezes). Além disso, ela possui as funções:

- `abs`: para transformar valores negativos em positivos;
- `signum`: indica, em seu retorno, se um número é positivo, negativo ou zero;
- `fromInteger`: para converter um número inteiro (`Integer` representa números inteiros grandes) para o tipo `a` indicado na instância.

No caso de `Int`, por exemplo, temos que `signum` retorna `1` se o número for positivo, `0` caso seja o zero, e `-1` para número negativo.

Classe Fractional

A classe `Fractional` é definida por:

```
class  (Num a) => Fractional a  where
    (/)                 :: a -> a -> a
```

```
fromRational        :: Rational -> a
```

Ela provê um jeito de efetuar a divisão para seu tipo, a partir da função `(/)`. A função `fromRational` converte uma fração, `3/5` por exemplo, em seu tipo `a`.

Note que todo tipo que instancia de `Fractional` deve ser também uma instância de `Num`, como indicado na restrição `Num a` da linha `(Num a) => Fractional a`.

Classe Real

A classe `Real` de tipo `(Num a, Ord a) => Real a` provê a função `toRational :: a -> Rational`, que converte seu tipo `a` em uma fração. Todo tipo instância de `Real` também deve, obrigatoriamente, ser instância de `Num` e `Ord`.

Classe Integral

A classe `Integral` determina que um tipo `a` é capaz de realizar divisões inteiras com quociente. As funções a serem implementadas são:

- `quotRem :: a -> a -> (a, a)`, que retorna o quociente na primeira coordenada da tupla e o resto na segunda, além de ter a função.
- `toInteger :: a -> Integer`, que converte seu tipo `a` em um `Integer`.

Todo tipo instância da classe `Integral` deve ser instância também de `Real` e `Enum`, conforme a declaração `class (Real a, Enum a) => Integral a`.

Classe Enum

A classe `Enum` possui uma interface para enumerações de tipo, por exemplo:

```
data Dia = Domingo | Segunda | Terca | Quarta | Quinta | Sexta |
Sabado
```

Isso pode ser associado a um inteiro, por exemplo, `Domingo = 1` , ..., `Sabado = 7` . Esta classe possui a definição:

```
class Enum a where
    toEnum :: Int -> a
    fromEnum :: a -> Int
```

Ela converte seu tipo `a` nesse inteiro representando uma contagem. Todo tipo instância de `Enum` ganha de graça as funções `succ` e `pred` , que indicam o sucessor e o predecessor de um valor. Se colocarmos `deriving Enum` no tipo `Dia` , teremos a enumeração anterior como:

```
Prelude> succ Domingo
Segunda

Prelude> pred Segunda
Domingo
```

O resultado foi como esperávamos. Os *typeclasses* podem ter funções já definidas previamente como `succ` e `pred` que valem para qualquer `a` . Note que `toEnum` e `fromEnum` não são funções previamente definidas, e elas esperam uma instância para ter sua definição algo que lembra interfaces no Java.

Classe Ord

A classe `Ord` indica que seu tipo possui ordenação por meio do operador `<=` . A classe `Ord` é definida como:

```
class Eq a => Ord a where
    (<=) :: a -> a -> Bool
```

Note que basta definir como funciona `<=`, que `>`, `<`, `>=`, `max` e `min` são ganhos de graça. Note também que todo tipo ordenável (instância de `Ord`) deve ser instância de `Eq`. A classe `Ord` facilita, por exemplo, o trabalho com lista (`Strings` especificamente):

```
Prelude> "HASKELL" > "OLA"
True
```

Isso porque facilita a escrita ao ocultar a função `length`, provendo uma escrita mais limpa e eficaz.

Classe Bounded

Finalmente, vemos a classe `Bounded`:

```
class Bounded a where
    minBound, maxBound :: a
```

As funções que não possuem entrada `minBound` e `maxBound` representam os limites mínimo e máximo do seu tipo. Para o tipo `Dia`, poderíamos criar a instância:

```
instance Bounded Dia where
    minBound = Domingo
    maxBound = Sabado
```

Note que poderíamos usar o `Bounded`, assim como todas essas classes, no `deriving`. A classe `Bounded` nos cria listas de todos os valores de tipo `[minBound .. maxBound]`, de forma geral e sem precisar saber qual tipo está sendo trabalhado.

O intuito de usar os *typeclasses* é sempre o de criar uma função que valha para o tipo mais genérico possível e para o maior

número de tipos possíveis. Obtém-se assim um nível de abstração maior em seu código, evitando uma quantidade enorme de linhas de código que fazem a mesma coisa.

Criando uma classe

Quando é necessário, por exemplo, ler diferentes tipos de arquivo nos quais o conteúdo deve ser representado por um tipo em Haskell, interface para validação de vários tipos de dados, precisamos fazer a criação de uma classe.

```
class SimNao a where
    simnao :: a -> Bool
```

Esse *typeclass* provê uma função `simnao` que deve ser definida para todo `a` do qual desejarmos criar uma instância. Ela representa uma maneira de fazer algum tipo de validação de dados dependente de algum tipo. Pode-se criar instâncias do *typeclass* `SimNao a` para qualquer tipo, como faremos para os seguintes:

```
instance SimNao Int where
    simnao x
        | x < 18 = False
        | otherwise = True
```

A instância de `SimNao` para `Int` pode conter validação de idades.

```
instance SimNao [a] where
    simnao [] = False
    simnao _ = True
```

Para listas (`String` , por exemplo), pode indicar que, ao digitar um nome, o campo foi deixado vazio.

```
Prelude> simnao (1::Int)
False
```

```
Prelude> simnao "João"
True
```

Note que `(1::Int)` foi usado para indicar que quermos um inteiro e não um `Num a`.

5.5 MONOIDES

Um monoide é um conjunto (em Haskell, um tipo) `m` equipado com a função infixa `<>` de tipo `m -> m -> m`, que satisfaz as seguintes leis:

- **Associatividade:** `(a <> b) <> c = a <> (b <> c)` para quaisquer valores `a`, `b`, `c` de `m`.
- **Elemento neutro:** existe um elemento `e` em `m`, tal que `a <> e = a` para todo valor `a` de `m`.

No Haskell, o *elemento neutro* é chamado de *mempty*, e a operação binária (funções infixas que recebem dois parâmetros, `<>`) de *mappend*.

Um *monoide* em Haskell é uma classe de tipos *typeclass* que deve ser importada do módulo `Data.Monoid`. Basta chamar a palavra reservada `Import` e o nome do módulo na linha subsequente à palavra `module`.

Essa classe deve ser usada toda vez que tivermos uma operação binária que recebe dois valores de mesmo tipo retornando um outro desse mesmo tipo. Os monoides são interfaces para operações que juntam dois valores de um mesmo tipo em um, algo mais abstrato que somas e multiplicações.

Ao se pensar em monoides, devemos achar um elemento que

não faz nada perante esta operação. Devemos verificar se esta operação é associativa (não importa de onde você comece a fazer a operação, pela esquerda ou direita, o resultado é o mesmo).

Se considerarmos o tipo `Int`, vemos que quatro possíveis operações binárias são candidatas a ser monoides, e elas são as quatro operações aritméticas. Por exemplo:

```
instance Monoid Int where
    mempty = 0
    mappend = (+)
```

Ou:

```
instance Monoid Int where
    mempty = 1
    mappend = (*)
```

Note que o elemento neutro da soma é o número 0, `x+0=x` para qualquer número `x`; e o da multiplicação é o número 1, `y*1=y` para qualquer número `y`. Como a soma e a multiplicação são associativas, por exemplo, `(5+3)+2 = 5+(3+2) = 10` e `(5*3)*2 = 5*(3*2) = 30`, temos que, para o tipo `Int`, conseguimos formar monoides tanto com a soma quanto a multiplicação em qualquer instância de `Num`.

Porém, em Haskell, só conseguimos ter uma instância de qualquer classe, e não só `Monoid`, para cada tipo por vez, causando uma ambiguidade. Será que devemos escolher a soma ou a multiplicação? Bom, para resolver esse problema, há dois tipos de kind `* -> *` no módulo `Data.Monoid` que possuem instâncias de monoide prontas. São eles `Sum` e `Product`, declarados como:

```
newtype Sum a = Sum { getSum :: a }
newtype Product a = Product { getSum :: a }
```

Eles possuem as instâncias:

```
instance Num a => Sum a where
    mempty = Sum 0
    mappend (Sum x) (Sum y) = Sum (x+y)

instance Num a => Product a where
    mempty = Product 1
    mappend (Product x) (Product y) = Product (x*y)
```

Com isso, é possível somar ou multiplicar sem usar os sinais `+` e `*` e, em vez disso, usar `mappend` ou `<>`, que realizará a operação pelo tipo envolvido. Antes de dar o exemplo, é necessário a chamada do módulo `Data.Monoid`, ou no programa usando `import Data.Monoid`, ou no GHCi com o comando `:m módulo`, que carrega um módulo externo pronto para seu uso.

```
Prelude> :m Data.Monoid

Prelude Data.Monoid> Sum 5 <> Sum 6
Sum 11

Prelude Data.Monoid> Product 5 <> Product 6
Product 30
```

Esse é o grande benefício de usar monoides. Podemos calcular ações representadas por uma operação binária usando uma única função, `mappend` ou `<>`.

TOME CUIDADO!

O compilador não verificará se seu operador tem realmente um `mempty` verdadeiro, ou se ele é associativo. Isso fica a cargo do programador. Porém, sugiro **fortemente** que siga essas regras, pois facilita na hora de fazer os chamados testes de propriedades.

Um teste de propriedade testaria sua função com base em alguma regra matemática, como por exemplo, a associatividade. Se você faz um teste com sucesso dessa propriedade, não precisará testar vários casos, barateando essa atividade.

Você pode estar se perguntando sobre a subtração e divisão, porém elas não formam monoides, pois não são operações binárias associativas. Há, no entanto, `mempty` para ambas, que seria 0 e 1, respectivamente. A associatividade falha nestes casos, ou seja, `3-(2-1) = 1` ao passo que `(3-2)-1 = 0` (a divisão você pode achar com facilidade um exemplo que faz a associatividade falhar).

A divisão possui um exemplo contrário análogo, `(10/5)/2 = 1` enquanto `10/(5/2) = 4`. Não transforme em instâncias de monoide operações que não são associativas, ou que há um falso `mempty`, pois quem sairá perdendo será você.

A associatividade é uma propriedade muito importante na área da programação paralela, daí a importância de se usar tal estrutura matemática. Pense em 3 processos rodando ao mesmo tempo, não

há como saber quem acaba primeiro. Se eles estiverem envolvidos em uma operação binária associativa, não há por que se preocupar com qual deles acaba primeiro. Se não houver a associatividade, os resultados serão diferentes, causando muito prejuízo.

Instâncias de monoide

Uma lista possui uma instância de monoide já implementada:

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

Seu `mappend` é a concatenação e, com isso, seu `mempty` é a lista vazia, pois ao concatenarmos qualquer lista à lista vazia, não teremos efeito algum. Essa operação é claramente associativa, então, não importa por onde começamos a concatenar, o resultado será o mesmo. Por exemplo: `[1,2,4] ++ ([0,9] ++ [8,5]) = ([1,2,4] ++ [0,9]) ++ [8,5] = [1,2,4,0,9,8,5]`.

A maior vantagem da instância de monoide para listas é que `String` é um `[Char]` e, acredite, isso facilita muito quando os tipos `Text` e `ByteString` são usados. Tanto um quanto o outro são usados em alguns pacotes de persistência de dados em um banco de dados e frameworks web, e são mais rápidos que a String convencional. O problema é que `String`, `Text` e `ByteString` possuem a função `concat` em comum, podendo dar ambiguidade em qual chamar. Considere o trecho:

```
import qualified Text as T
import qualified ByteString as BS
```

Então, usaríamos `concat`, `T.concat` e `BS.concat` para concatenarmos dois desses tipos: `String`, `Text` e `Bytestring`,

respectivamente. Com o uso das instâncias de monoide comum aos três, basta usar `<>` e estamos feitos, algo bastante útil quando estamos em um ambiente de produtividade.

Outro benefício do uso de uma instância de monoide é a função `mconcat`, que já é implementada da seguinte forma:

```
mconcat :: (Monoid a) => [a] -> a
mconcat xs = foldl mappend mempty
```

Essa função recebe uma lista de um tipo `a` (obrigada a ser instância de monoide, se não obteremos erro de compilação) e retorna um valor de tipo `a` que representa a sucessiva aplicação de `mappend`, acumulando em um único valor e tendo como valor inicial da acumulação `mempty`.

Pense em uma soma de inteiros de um vetor, então começamos do zero e vamos somando até acabar os elementos. Veja um exemplo:

```
Prelude Data.Monoid> mconcat [Sum 7, Sum 3, Sum 10]
Sum 20

Prelude Data.Monoid> mconcat ["Ola Mundo", " Haskell", "!!!"]
"Ola Mundo Haskel!!!"
```

5.6 MINIPROJETO: TRABALHANDO COM PARSERS

No *capítulo 3*, vimos que há uma função chamada `verFolha`, que transforma um valor do tipo funcionário em uma String no formato JSON. A função em questão é:

```
verFolha :: Pessoa -> String
verFolha p = "{nome: \"" ++ (nome p) ++
             "\", cargo: \"" ++ show (cargo p) ++
```

```
            "\", salario: " ++ show (verSalario p) ++ "}"
```

Apesar de não estar errada, podemos melhorar isto e generalizar, usando um *typeclass*. Isto é, fazer uma rotina, de modo que funcione para todo tipo que necessitarmos transformar em JSON. Caso quisermos ter um tipo para cada entidade da empresa, e for necessário uma conversão para JSON, basta criarmos uma instância deste *typeclass* para cada um desses tipos.

O intuito aqui é mostrar como usar os *typeclasses*, e também criar nossas próprias classes, além de saber quando usar tal conceito. Será criado um tipo chamado `Projeto` que carregará as informações de orçamento do projeto, nome e uma lista com índice das pessoas envolvidas (o índice será implementado no *capítulo 7*).

Como será necessário ver o total de pessoas em todos os projetos e o orçamento de todos eles, tornaremos `Projeto` instância de monoide.

```
module Projeto where

-- funcionalidades implementadas previamente

data Projeto = Projeto {nomeProjeto :: String,
                        budget :: Double,
                        envolvidos :: [Int]} deriving Show

class ToJSON a where
    toJSON :: a -> String

instance ToJSON Pessoa where
    toJSON p = "{nome: \"" ++ (nome p) ++
               "\", cargo: \"" ++ show (cargo p) ++
               "\", salario: " ++ show (verSalario p) ++ "}"

instance ToJSON Projeto where
    toJSON p = "{nome: \"" ++ (nomeProjeto p) ++
```

```
            "\", orçamento: \"" ++ show (budget p) ++
            "\", envolvidos: " ++ show (envolvidos p) ++ "}"

instance Monoid Projeto where
    mempty = Projeto "" 0 []
    mappend (Projeto nome1 budget1 env1) (Projeto nome2 budget2 e
nv2) =
        Projeto (nome1 ++ ", " ++ nome2) (budget1 + budget2) (en
v1 ++ env2)
```

Note que agora não será mais necessária a função `verFolha`, e a classe `ToJSON` consegue trabalhar com cada tipo que quisermos transformar em um JSON. Com a instância de `Monoid` para `Projeto`, conseguimos juntar dois projetos em um só, de modo a concatenar seus nomes, somar seus orçamentos e juntar as duas listas de pessoas envolvidas.

Outro benefício de ter a instância de `Monoid` para `Projeto` é que a função `mconcat` juntará informações de vários projetos ao mesmo tempo.

5.7 EXERCÍCIOS

5.1) Crie o tipo `TipoProduto` que possui os *values constructors* `Escritorio`, `Informatica`, `Livro`, `Filme` e `Total`. O tipo `Produto` possui um value constructor - de mesmo nome - e os campos `valor` (`Double`), `tp` (`TipoProduto`) e um value constructor `Nada`, que representa a ausência de um `Produto`.

Deseja-se calcular o valor total de uma compra, de modo a não ter nenhuma conversão para inteiro e de forma combinável. Crie uma instância de monoide para `Produto`, de modo que o retorno sempre tenha `Total` no campo `tp` e a soma dos dois produtos

em `valor`. Explique como seria o exercício sem o uso de monoides. Qual(is) seria(m) a(s) diferença(s)?

5.2) Crie uma função `totalGeral` que recebe uma lista de produtos e retorna o preço total deles usando o monoide anterior.

5.3) A função `min` no Haskell retorna o menor entre dois números, por exemplo, `min 4 5 = 4`.

- Crie um tipo `Min` com um campo inteiro, que seja instância de `Ord`, `Eq` e `Show` (deriving).
- Crie uma instância de `Monoid` para `Min` (`maxBound` representa o maior inteiro existente no Haskell).
- Quanto vale a expressão `Min (-32) <> Min (-34) <> Min (-33)`?
- Explique sua escolha para o `mempty`.

5.4) Crie uma função `minAll` que recebe um `[Min]` e retorna um `Min` contendo o menor valor.

5.5) Crie o tipo `Paridade` com os *values constructors* `Par` e `Impar`. Crie o *typeclass* `ParImpar` que contém a função `decide :: a -> Paridade` e possui as instâncias:

- Para `Int`: noção de Par/Impar de `Int`.
- Para `[a]`: uma lista de elementos qualquer é `Par` se o número de elementos o for.
- `Bool`: `False` como `Par`, `True` como `Impar`.

5.6) A função `max` no Haskell retorna o maior entre dois números, por exemplo: `max 4 5 = 5`.

- Crie um tipo `Max` com um campo inteiro que seja instância de `Ord`, `Eq` e `Show` (deriving).
- Crie uma instância de `Monoid` para `Max` (`minBound` representa o menor inteiro existente no Haskell).
- Quanto vale a expressão `Max 10 <> Max 13 <> Max 5`?
- Explique sua escolha para o `mempty`.
- Crie uma função `maxAll` que recebe um `[Max]` e retorna um `Max` contendo o maior valor.

5.7) Usando a estrutura de árvore, monte uma função `mapa`, que jogue uma função passada por parâmetro para todos os elementos de uma árvore. Deixe explícito o tipo desta função.

5.8) Usando o exercício anterior, some 5 a cada elemento de uma árvore de inteiros.

5.9) Uma lista ordenada é uma lista cujos elementos são inseridos de forma ordenada (crescente). Usando o tipo `ListaOrd a = a :?: (ListaOrd a) | Nulo deriving Show`, crie as funções:

- `inserir :: (Ord a) => a -> ListaOrd a -> ListaOrd a`
- `remover :: (Eq a) => a -> ListaOrd a -> ListaOrd a`
- `tamanho :: ListaOrd a -> Int`

Observação: a função `remover` deve buscar um elemento. Se não achar, a lista deve se manter intacta.

5.10) Usando a estrutura de árvore vista, faça uma função que some todos os elementos de uma árvore de números.

5.11) Implemente os percursos pós-ordem e pré-ordem. Via comentário, faça os "testes de mesa" para os dois percursos da árvore `Raiz 15 (Raiz 11 (Folha 6) (Raiz 12 (Folha 10) Nula)) (Raiz 20 Nula (Raiz 22 (Folha 21) Nula))`.

5.12) Faça uma função para inserir um elemento em uma árvore de busca binária (use a mesma estrutura vista).

5.13) Faça uma função que, a partir de uma lista de elementos de tipo, insira todos os elementos desta lista na árvore e retorne-a, usando o exercício anterior.

5.8 CONCLUSÃO

Neste capítulo, as coisas ficaram mais sérias, e conceitos importantes como polimorfismo paramétrico, classes de tipos e monoides foram apresentados ao leitor. Você também viu a estrutura dos números em Haskell a partir de seus **typeclasses**.

Os próximos capítulos que completam este livro dependem completamente dos conceitos apresentados aqui. De todos, este é o que mais apresenta conceitos que o leitor já está habituado, caso tenha estudado algumas linguagens orientadas a objetos (há uma semelhança entre os conceitos).

O próximo capítulo é de caráter opcional ao leitor, pois será uma introdução para explicar de onde vem os conceitos explicados nos capítulos subsequentes. Caso queira um continuação sem se preocupar com a matemática, pule para o *capítulo 7*.

CAPÍTULO 6

TEORIA DAS CATEGORIAS

Neste capítulo, será introduzido, de maneira informal, o conceito matemático de Categoria, no qual a linguagem Haskell se baseia para nortear alguns conceitos de programação. Ainda que seja uma exposição breve, é importante o leitor ter este embasamento para captar a ideia do que vamos utilizar nos próximos capítulos.

Um programador Haskell não precisa ser um matemático ou um expert em Teoria das Categorias, porém, um entendimento (mesmo que básico) do assunto permitirá um entendimento mais profundo da linguagem, dando um diferencial poderoso no mercado, pelo aumento da capacidade de abstração. O uso da Teoria das Categorias em Engenharia de Software é uma tendência fora do país, e provavelmente logo sairá da escuridão, aos olhos do mercado.

6.1 CATEGORIAS

Uma categoria C é uma estrutura abstrata composta de objetos e morfismos. Um objeto é uma representação abstrata do que se quer estudar (conjuntos, tabelas, palavras, pessoas, animais etc.) e um morfismo é uma relação de objetos que se compõem de forma

associativa. Existe também o morfismo neutro, chamado identidade. São coleções simples com três componentes:

- Coleção de todos os objetos chamada de `ob( C )`.
- Coleção de todos os morfismos chamada de `hom( C )`: um morfismo relaciona dois objetos A e B. Um morfismo `f :: A -> B` relaciona um objeto de entrada A com um de saída B (generalização do conceito de função).
- Noção de composição dos morfismos: `(.)`: Se `g :: A -> B` e `f :: B -> C`, então `f.g :: A -> C`.

Em toda categoria, é necessário existir a função identidade. Em outras palavras, para cada objeto X de uma categoria C, existe um morfismo `idX :: X -> X`, tal que para cada morfismo `f :: A -> B` satisfaça:

```
f . idA = f = idB . f
```

Ou seja, a função identidade não faz absolutamente nada. Ela age como elemento neutro, assim como o 0 na soma e o 1 na multiplicação.

E finalmente a composição deve ser associativa. Isto é, se `f :: A -> B`, `g :: B -> C` e `h :: C -> D`, obtemos:

```
(h . g) . f = h . (g . f)
```

Note que a notação para a função composta usada é em termos da função `(.)`, apresentada no *capítulo 4*. Vale lembrar de que `(f.g) x = f(g(x))`, ou seja, primeiro calculamos `g` tendo como argumento `x` e, após o retorno, calculamos `f`.

Um exemplo de categoria

Banco de Dados, conforme Spivak (2014), pode ser visto como uma categoria, em que seus objetos seriam as tabelas, e seus morfismos, as colunas. Uma declaração de regra de negócio é dada pela composição dos morfismos. Considere as tabelas Funcionário e Departamento a seguir, respectivamente.

Tabela de Funcionário

IdFunc	NomeFunc	SobrenomeFunc	Chefe	Dpt
101	João	Silva	103	10
102	José	Santos	102	02
103	Ernesto	Oliveira	103	10

Tabela de Departamento

idDept	NomeDept	Secr
10	Compras	101
02	Vendas	102

O campo `Secr` indica um secretário de departamento. Analisando esta situação sob a ótica de uma categoria, temos:

- `ob( C )` é composto de Funcionário, Departamento (as tabelas) e os tipos de dados `Int` (inteiros) e `String` (palavras).
- `Hom( C )` é composto por: `IdFunc :: Funcionário -> Int`, `NomeFunc :: Funcionário -> String`, `Chefe :: Funcionário -> Funcionário`, `Dept :: Funcionário -> Departamento`, `idDept :: Departamento -> Int`, `NomeDept :: Departamento -> String` e `Secr ::`

```
Departamento -> Funcionário .
```

Foram escolhidos `IdFunc :: Funcionário -> Int` e `idDept :: Departamento -> Int` pelo fato de não relacionarem tabelas. As chaves estrangeiras relacionam tabelas, logo, a autorrelação e as duas chaves estrangeiras são definidas como `Chefe :: Funcionário -> Funcionário` , `Dept :: Funcionário -> Departamento` e `Secr :: Departamento -> Funcionário` . Os outros campos não citados relacionam as tabelas com `String` ou `Int` .

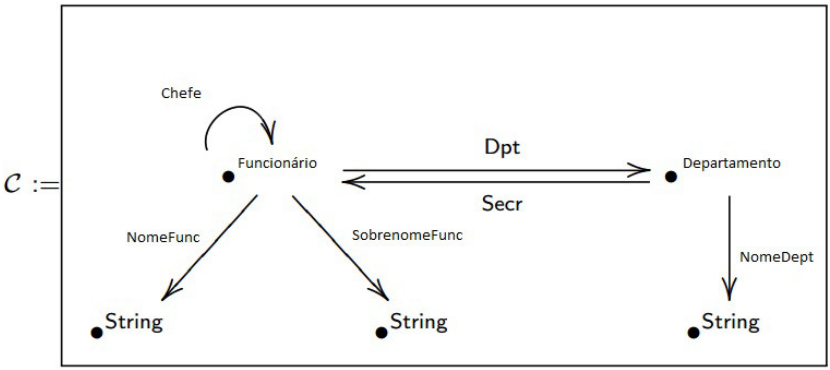


Figura 6.1: Representação de um banco de dados sob a ótica de uma categoria Spivak (2014), traduzido pelo autor

A figura anterior mostra os objetos e os morfismos identificados. Os círculos são os objetos, e as flechas, morfismos. Um exemplo de regra de negócio seria `Dpt . Chefe` , que indica a qual departamento um chefe está associado.

A categoria Hask

A categoria mais importante aqui é a Hask, que representa os tipos contidos na linguagem Haskell. Em outras palavras, temos

que:

- `Ob(Hask)` é constituída dos tipos, por exemplo: `Int`, `Bool`, `[a]`, `Char`, `Double` etc.
- `Hom(Hask)` é constituída das funções, por exemplo: `length :: [a] -> Int`, `show :: a -> String`, `dobro :: Double -> Double` etc.
- A noção de composição é atribuída ao operador `.` (visto no *capítulo 4*).
- O operador `.` é associativo.
- Para todo tipo, há a função `id :: a -> a`, representando a identidade (a ser explicada mais a fundo nas próximas seções).
- A função `id` se comporta como elemento neutro.

Há uma classe que possui a noção de categoria chamada `Category` no módulo `Control.Category`:

```
class Category cat where
    -- | identidade
    id :: cat a a

    -- | composição
    (.) :: cat b c -> cat a b -> cat a c
```

Note que esta classe trabalha com tipos de kind `* -> * -> *`. A categoria Hask acontece na instância de `Category` para `->` (da funções), que é kind `* -> * -> *`. O contêiner `->` tem esse kind, pois `a -> b` é equivalente a `(->) a b`, formando um tipo de dado com duas variáveis **type variable**.

Tendo em mente o tipo de `id :: cat a a`, se substituirmos `cat` por `->`, obteremos `id :: (->) a a`, ou seja, `id :: a -> a` quando há a substituição de `cat` por `->`. Analogamente, `(.)`

`:: cat b c -> cat a b -> cat a c` se torna `(.) :: (->) b c -> (->) a b -> (->) a c`, nos dando `(.) :: (b -> c) -> (a -> b) -> (a -> c)` como esperado.

A instância em questão é definida por:

```
instance Category (->) where
    id = GHC.Base.id
    (.) = (GHC.Base..)
```

Isso indica o que já sabíamos: `id` é a identidade, e `.`, a composição.

6.2 NOÇÃO MATEMÁTICA DE FUNTOR

Um funtor é uma relação entre duas categorias `C` e `D`, e mapeia objetos em objetos e morfismos em morfismos. Ou seja, todo objeto da categoria C terá um correspondente em D, e todo morfismo da categoria C terá um correspondente em D.

Pense em bolas brancas de papel ligadas por fios brancos. Se você trocar as bolas brancas por vermelhas e os fios brancos por vermelhos, entenderá o intuito do funtor.

A categoria C teria como objetos as bolas brancas, e a categoria D, as vermelhas. A categoria C teria como morfismos os fios brancos, e a categoria D, os vermelhos. O funtor seria o ato de trocas das bolas e dos fios, sem bagunçar o desenho que elas formam. Esse é o pensamento. Matematicamente, isso pode ser representado como:

- **Associação de valores:** para todo A de `ob(C)`, é associado `F(A)` em `ob(D)`.
- **Associação de morfismos:** `f :: A -> B` é associado

a `F(f) :: F(A) -> F(B)`.

Isso deve satisfazer:

```
F(idX) = idF(X)
F(f.g) = F(f) . F(g)
```

Todo funtor em Haskell é um endofuntor, ou seja, a categoria de entrada dos funtores é **Hask**, assim como a sua saída. Um funtor em Haskell possui kind `* -> *` e deve ser uma instância do typeclass **Functor**, que será visto a fundo no próximo capítulo.

6.3 FUNÇÃO IDENTIDADE EM HASKELL

Em Haskell (categoria Hask), temos a função `id`, que representa a função identidade apresentada anteriormente. Seu tipo segue a seguinte definição:

```
Prelude> :t id
id :: a -> a
```

A partir da definição de polimorfismo paramétrico, isso indica que podemos ter `id`s para todo tipo `a`, ou seja, `id :: Int -> Int`, `id :: Bool -> Bool`, e assim por diante. Como dito, a função identidade age como elemento neutro, logo:

```
Prelude> id 5
5

Prelude> id True
True
```

Note que uma função de tipo `a -> a` só pode executar essa tarefa: nada. É difícil pensar em uma função que valha ao mesmo tempo para **todos** os tipos sem restrição alguma, por isso a única tarefa possível para `id` é não fazer nada.

A função `id` é usada, na prática, para, por exemplo, ignorar um argumento (e o **currying** também) de uma função de alta ordem. Isto é, uma função que pede uma função, e você precisa que este argumento não realize operação alguma. Veja um exemplo:

```
module Cap6 where

data Produto = Produto {preco :: Double, desconto :: Double} deri
ving (Show, Eq, Ord)

buscaProd :: (Produto -> a) -> [Produto] -> ([a] -> a) -> a
buscaProd f ps g =  g (map f ps)
```

A função `buscaProd` possui um tipo esquisito, porém bem útil. Ela possui como parâmetro uma função de tipo `Produto -> a` (qualquer função de projeção de `Produto` entra aqui, `preco` ou `desconto`, pois `preco :: Produto -> Double` e `desconto :: Produto -> Double`), uma lista de produtos e uma função do tipo `[a] -> a`.

Aqui podemos fazer qualquer tipo de funções parecidas com um fold, somar a lista, maior da lista, menor da lista etc. Se quisermos o maior desconto do `Produto`, faremos:

```
Cap6> buscaProd desconto [Produto 20.99 0.2, Produto 4.99 0.7, Pr
oduto 1.99 0.4] minimum
0.2
```

Ou seja, ele vai extrair o `desconto` de todos os produtos, deixando a lista como `[0.2, 0.7, 0.4]`, e vai extrair o menor que é `0.2` (80% de desconto), o maior desconto. Se quisermos a soma dos `precos`, faremos:

```
Cap6> buscaProd valor [Produto 20.99 0.2, Produto 4.99 0.7, Produ
to 1.99 0.4] sum
27.969999999999995
```

Também, se quisermos o `Produto` com maior valor e menor desconto, primeiramente implementaremos a função:

```
maiorPrecoMenorDesc :: Produto -> Produto -> Produto
maiorPrecoMenorDesc (Produto p1 d1) (Produto p2 d2) = Produto (ma
x p1 p2) (max d1 d2)
```

Lembra-se de que o `foldl` transforma uma lista em um valor? Nesse caso, usaremos `foldl maiorPrecoMenorDesc (Produto 0 0)`, que executa a função `maiorPrecoMenorDesc` em todos os produtos, tendo como retorno o produto de maior preço e menor desconto. Para ajudar nesta tarefa, implementaremos:

```
todos :: [Produto] -> Produto
todos ps = foldl maiorPrecoMenorDesc (Produto 0 0) ps
```

Para testar, não vamos extrair valor algum de `Produto`, ou seja, queremos ignorar a extração dos valores. Logo, a única função que pode nos ajudar é aquela que não faz nada: `id`.

```
Cap6> buscaProd id [Produto 20.99 0.2, Produto 4.99 0.7, Produto
1.99 0.4] todos
Produto {preco = 20.99, desconto = 0.7}
```

6.4 CONCLUSÃO

Neste capítulo, abordamos, de maneira leve e informal, o conceito matemático de categorias. Se o conceito matemático de alguma forma foi pesado para o leitor, é necessário ter em mente apenas as nomenclaturas: (endo)funtor, morfismo, objeto e categoria Hask.

Capítulo 7

FUNTORES

No capítulo anterior, foi introduzida a noção de funtor de acordo com a Teoria das Categorias. Neste, abordaremos o uso dos funtores na linguagem Haskell.

Em Haskell, um funtor é simplesmente uma classe (ou *typeclass*) que possui a função `fmap` a definir para toda instância. Ou seja, todo tipo de kind `* -> *` que seja instância de `Functor` deve "saber" como levar uma função `g` de tipo `a -> b` para dentro de um contêiner de tipo `f a`, resultando assim em algo do tipo `f b`.

A entrada de uma função `g` de tipo `a -> b` é claramente de tipo `a`, fazendo com que algo do tipo `f a` seja barrado com **type mismatch** pelo compilador, já que `a` e `f a` são valores de tipos diferentes. A classe `Functor` é definida como:

```
class  Functor f  where
    fmap :: (a -> b) -> f a -> f b
```

A função `fmap` faz com que `g` de tipo `a -> b` "seja jogada para dentro" de `f a`, e esta é a intuição sobre os funtores. A partir deste raciocínio, é possível notar que a lista `[]` (que é um tipo de kind `* -> *`) possui uma instância de `Functor`, e seu `fmap` é a conhecida função `map`.

Podemos pensar também que os funtores aqui em Haskell generalizam de alguma maneira a ação do `map` em relação às listas. Um exemplo lúdico é o ato de fazer carinho em um gato. A ação do carinho é uma função. Se o gato entrar em uma caixa, grande de preferência, você nâo fará carinho na caixa (`f a` não pode ser entrada de `g`), então você deve entrar na caixa. O ato de entrar na caixa para fazer o carinho seria o tal `fmap` . Vamos checar seus tipos:

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]


Prelude> :t fmap
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

Nisso, notamos que, ao trocar `[]` na função `map` , por um tipo `f` de kind `* -> *` ,
qualquer que seja instância de `Functor` , obtemos o tipo de `fmap` . O tipo de `fmap` nos diz que ela possui como entrada uma função de tipo `a -> b` e um valor dentro de um contêiner `f` de tipo `f a` para devolver algo de tipo `f b` .

Se usarmos o *currying*, podemos enxergar `fmap g` como uma função que recebe algo de tipo `f a -> f b` , ou seja, algo que recebe `f a` e devolve `f b` . Todo tipo instância de `Functor` deve ser implementado segundo as leis dos funtores (assim como nos monoides).

```
fmap id x = x
fmap (g.h) x = (fmap g . fmap h) x
```

Estas regras valem para todo valor `x` de tipo `f a` . As leis acima indicam que se jogarmos para dentro a função `id` não acontecerá nada e que se houver uma composição de funções no

argumento de `fmap` equivalerá a composição listada na expressão `(fmap g . fmap h) x`, ou seja, será efetuado `fmap h x` primeiro e o resultado disso será argumento de `fmap g`. Ressalto que o compilador permite que um funtor não siga essas regras, porém para efeito de testes por propriedades é recomendado seguir.

7.1 FUNTOR MAYBE

O tipo `Maybe` é um tipo de kind `* -> *` definido como:

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

O *value constructor* `Just`, que possui um campo de tipo variável `a`, pode representar uma computação validada, ao passo que `Nothing` representa um erro ou ausência de uma computação. Este tipo é usado para validações de dados, como:

```
divisao :: Double -> Double -> Maybe Double
divisao x 0 = Nothing
divisao x y = Just (x/y)
```

A função `divisao` recebe dois parâmetros que serão testados pelo *pattern matching*. No caso de o segundo ser `0`, o retorno será `Nothing`, pois não há divisão por zero.

O último padrão indica que o segundo parâmetro é diferente de zero, então a divisão será colocada dentro do *value constructor* `Just`, indicando uma operação válida. A função está correta e possui uma boa regra de validação, porém ela não opera com a função `2*`, por exemplo:

```
Prelude> 2 * (divisao 10 2)

<interactive>:15:1:
```

```
    Non type-variable argument in the constraint: Num (Maybe a)
    (Use FlexibleContexts to permit this)
    When checking that ‘it’ has the inferred type
      it :: forall a. Num (Maybe a) => Maybe a
```

Isso fará com que o compilador procure uma instância de `Num` para `Maybe a` (que não há) e, mesmo que houvesse, a expressão anterior é errada e não compilaria, já que `Double` é diferente de `Maybe Double`.

O tipo `Double` representa um número que é tangível e pode ser usado, já `Maybe Double` pode não haver um número dentro e, assim, representa uma operação mal sucedida (uma divisão por zero, por exemplo). Por isso, é necessária uma maneira de operarmos um `Double` com um `Maybe Double`, genericamente `a` com `Maybe a`. Essa "mistura" se dá através de uma instância de `Functor` (já implementada):

```
instance Functor Maybe where
    fmap g Nothing = Nothing
    fmap g (Just x) = Just (g x)
```

Quando o parâmetro de `fmap g` (lembre-se do *currying*) é `Nothing`, nada deve ser feito, pois é um contêiner vazio e não há como fazer a função passar para dentro. Quando o parâmetro for `Just x`, passamos a função `g` para dentro do contêiner `Just (g x)`, e nada mais.

Agora é possível calcular a expressão anterior usando o `fmap`:

```
Prelude> fmap (2*) (divisao 10 2)
Just 10
```

Em vez de ficarmos usando estruturas condicionais (como em outras linguagens), o funtor `Maybe` nos dá uma solução prática para erro, como no exemplo a seguir:

```
Prelude> fmap (2*) (divisao 10 0)
Nothing
```

Assim, nos dá um controle melhor sobre situações indesejadas, semelhantes a dividir por zero.

Há um operador que representa o `fmap` chamado `<$>`, e ele nada mais é do que o `fmap` de modo infixo:

```
Prelude> (2*) <$> (divisao 10 2)
Just 10
```

A escolha de um ou de outro é questão de estilo de código. O operador `<$>` é usado frequentemente junto aos *Funtores Aplicativos* a serem discutidos nas próximas seções desse mesmo capítulo.

7.2 CRIANDO SEU FUNTOR

Os contêiners podem ser vistos como um rótulo. No exemplo anterior, `Maybe a` rotula um tipo `a` qualquer, indicando que pode haver um valor ou um erro lá. As listas `[a]` indicam que pode haver vários valores dispostos em forma de lista. Há também o `IO a`, que indica que o tipo `a` é oriundo de uma computação com efeitos (leitura e escrita de arquivos, por exemplo, que será estudado a fundo no *capítulo 9*).

Os funtores ajudam as funções a operarem mais facilmente com tipos similares aos citados. É possível criar nosso contêiner contendo dois espaços para um mesmo tipo `a`, como por exemplo, o tipo:

```
data Dupla a = Dupla a a deriving Show
```

Este tipo é análogo a `(a,a)`. O tipo `Dupla` nos ajuda a

trabalhar uma instância de funtor diferente do que a provida pela linguagem. Vamos analisar:

```
instance Functor Dupla where
    fmap g (Dupla x y) = Dupla (g x) (g y)
```

Nesse código, vemos que `Dupla x y` é um valor de tipo `Dupla a`, assim sendo `x :: a` (`x` tem o tipo `a`) e `y :: a` também. Logo, o resultado de `fmap g` calculado em `Dupla x y` nos dá `Dupla (g x) (g y)`, resultando em algo do tipo `Dupla b`, que é exatamente a definição de `fmap`.

Moral da história: não importa quantos campos de mesmo tipo seu contêiner possua, a função `g` pode se distribuir por mais de um campo, se necessário.

7.3 FUNTORES APLICATIVOS

Quando temos uma função dentro de um funtor (`f (a -> b)`), e queremos operar em cima de um valor `f a`, temos a necessidade de usar o operador `<*>` definido na classe `Applicative`, que rege os funtores aplicativos. Outro problema é que o `fmap` sozinho funciona apenas para funções com um parâmetro, como o `2*` usado anteriormente.

Um funtor aplicativo ajuda a fazer uma função receber um valor como parâmetro quando tanto a função como o(s) parâmetro(s) estão dentro de tipos de kind `* -> *`. A classe `Applicative` pode ser definida como:

```
class Functor f => Applicative f where
    pure :: a -> f a
    <*> :: f (a -> b) -> f a -> f b
```

Note que `<*>` está envolvido com funções `f (a -> b)` e valores `f a`, dentro de um funtor `f`. Preste bem atenção agora, e compare as expressões a seguir:

```
($) :: (a -> b) -> a -> b
<*> :: f (a -> b) -> f a -> f b
```

Notou alguma semelhança? Se sim, você está muito bem! Se não, experimente apagar os `f's` de `<*>`. Recairemos em `$`. Mágico, certo?

Você pode pensar em `<*>` como um nome *gourmet* para o `$`. Eles fazem a mesma coisa (aplicam uma função sobre um valor), porém, no caso do primeiro, há um funtor envolvido. A função `pure` simplemente coloca um valor dentro de um funtor aplicativo, e nada mais.

Para testar os exemplos, devemos importar o módulo `Control.Applicative`. Para os testes, vamos usar as instâncias de `Applicative` de nossos dois funtores favoritos: `Maybe` e `[]`.

```
instance Applicative Maybe where
    pure x = Just x
    (Just f) <*> (Just x) = Just (f x)
    _ <*> _ = Nothing

instance Applicative [] where
    pure x    = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Na instância de `Applicative` para `Maybe`, vemos que o `pure` joga um valor `x` qualquer de tipo `a` dentro do `Just`, como `pure 5 = Just 5` nesse caso. O operador `<*>` provê a aplicação de `f` em `x`, sendo que tanto `f` de tipo `a -> b` quanto `x` de tipo `a` estão presos dentro do funtor `Maybe`. Com isso, temos `(Just f) <*> (Just x) = Just (f x)`, ou seja, `f`

`x` dentro do funtor `Maybe` (*value constructor* `Just`).

No caso de `[]`, o `pure` coloca um valor dentro de uma lista, neste caso, `pure 5 = [5]`. No caso do operador de aplicação, serão aplicadas todas as funções dentro da lista `fs` em todos os valores da lista `xs`.

A *list comprehension* `[f x | f <- fs, x <- xs]` diz que, para todo `f` dentro de `fs` e todo `x` dentro de `xs`, ocorrerá a aplicação (execução da função) `f x`.

```
Prelude> Just (2*) <*> Just 10
Just 20

Prelude> [(2*), id, (3*)] <*> [1,2]
[2, 1, 3, 4, 2, 6]
```

Uma conclusão possível de se tirar é `fmap f x = pure f <*> x`. Outra aplicação legal dos funtores aplicativos é que se, por exemplo, quisermos dividir `Just 10` por `Just 2`, teríamos de extrair o 10 e o 2, e depois dividir.

Lembre-se de que extrair valores de tipos com instância funtor nem sempre é legal. Pense em como extrair algum valor de uma lista vazia `[]` ou de um erro `Nothing`. Para executar a divisão anterior, bastaria fazer:

```
Prelude> pure (/) <*> Just 10 <*> Just 2
Just 5
```

Ou mais comumente:

```
Prelude> (/) <$> Just 10 <*> Just 2
```

A primeira expressão indica o uso do `pure` e a segunda do `fmap` (`<$>`). As expressões se associam à esquerda e fazem o uso

do *currying* para funcionar. Por exemplo, na primeira expressão:

```
pure (/) <*> Just 10 <*> Just 2 =
(pure (/) <*> Just 10) <*> Just 2 =
(Just (/) <*> Just 10) <*> Just 2 =
Just (10/) <*> Just 2 =
Just (10/2) =
Just 5
```

Note que a primeira linha representa a expressão a calcular, e a segunda foi usada na associação à esquerda. Já na terceira, temos a definição do `pure` para o funtor aplicativo `Maybe`; na quarta foi usado o *currying* (lembre-se de que `10/` é o *currying* da função divisão, pois falta o segundo argumento) e a definição de `<*>` para `Maybe`; e na quinta, temos a mesma definição e, finalmente, é computado `10/2`.

Na segunda expressão, temos:

```
((/) <$> Just 10) <*> Just 2 =
Just (10/) <*> Just 2 =
Just (10/2) =
Just 5
```

Da primeira para segunda linha, foi usado o `fmap` e o *currying* e, na próxima, a definição de `<*>`.

No framework para desenvolvimento de aplicações web `Yesod`, são usados funtores aplicativos para ler formulários de um HTML e fazer uma requisição `GET` ou `POST` para uma rota economizando muitas linhas. O exemplo usado lá é parecido com isso.

Suponha haver um funtor aplicativo `Form`, que representa um valor digitado em campo `input type=text`, por exemplo. Considere o tipo `Pessoa = Pessoa String Int`, que possui um

nome `String` e uma idade `Int`. O tipo `Pessoa` é na verdade uma função `String -> Int -> Pessoa` pelo *currying*. Logo, se digitarmos `João` e `40`, teremos `Form João` e `Form 40` no Haskell.

```
Pessoa <$> Form "João" <*> Form 40
Form (Pessoa "João") <*> Form 40
Form (Pessoa "Jõao" 40)
```

Ao passar o tipo `Pessoa` construído "para dentro" de `Form`, poderíamos salvar o registro digitado no formulário em uma tabela do banco de dados usando o `Persistent`.

7.4 FUNTORES CONTRAVARIANTES

Esta seção é opcional e você pode pulá-la sem problema algum no aprendizado do restante do livro. Mas se você é um leitor curioso e que realmente quer ir além na linguagem, e ter um *boost* em sua capacidade de pensamento e abstração, esta seção é para você.

Primeiramente, a classe `Functor` vista aqui, na realidade, tem um "sobrenome" chamado **Covariante**. Uma pergunta que pode ter surgido é: "será que qualquer tipo pode ser instância de functor?".

Em um primeiro momento, somos levados a crer que sim, a partir do momento em que criamos o tipo `Dupla` (que carrega dois valores de tipo `a`) e vimos que o tipo `[]` carrega vários valores de tipo `a`. Vamos analisar, então, o tipo:

```
module Cap07 where

data Predicado = Predicado {runPred :: a -> Bool}
```

Ele possui um *value constructor*, chamado `Predicado`, e um campo de tipo `a -> Bool`, chamado `runPred`. A função `runPred` recebe um valor de um tipo qualquer `a` e retorna um valor `Bool`.

Esse tipo generaliza a noção de validação de dados, ou seja, tem várias regras de validação "dentro" deste tipo de kind `* -> *`. A pergunta que fica é: podemos ter uma instância de `Functor` para `Predicado`, dado que ele tem kind `* -> *`? Para respondê-la, vamos entender o tipo `Predicado`.

```
ehMenor4 :: Predicado Int
ehMenor4 = Predicado (\x -> x < 4)
```

A função monta um predicado que checa se um número é negativo ou não. Note que o tipo `Predicado` carrega uma função dentro e esta tem o tipo `Int -> Bool` (por causa do tipo de `ehMenor4` ser `Predicado Int`). Portanto, o parâmetro `x` do lambda `\x -> x < 4` é um `Int`.

Veja um outro pequeno exemplo:

```
tamanhoOito :: Predicado String
tamanhoOito = Predicado (\x -> length x == 8)
```

Ele verifica se uma `String` tem tamanho 8 ou não. Como o tipo da função é `Predicado String`, a entrada do lambda dentro de `Predicado` será uma `String`. Para usar as funções anteriores, fazemos:

```
Cap07> let func = runPred ehMenor4
Cap07> func 8
False
Cap07> func (-5)
True
```

Primeiramente, utilizamos uma expressão usando o `let`, para podermos usar `func` sempre que precisarmos na sessão ativa do GHCi. A função `func` contém `runPred ehMenor4`, que extrai a função de dentro de `Predicado`. Logo, `func = \x -> x < 0` ou, em outras palavras, `func x = x < 0`, e ela possui o tipo `Int -> Bool`.

As expressões `func 8` e `func (-5)` testam se os números passados no argumento são negativos ou não:

```
Cap07> let func = runPred tamanhoOito
Cap07> func "Haskell"
False
Cap07> func "HASKELL "
True
```

O mesmo ocorre com a função `tamanhoOito`. Entendido o tipo `Predicado` (que é realmente um pouco diferente), vamos agora recapitular o tipo de `fmap`.

```
Cap07> :t fmap
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

A pergunta que fica no ar é: por conta do `Predicado` possuir kind `* -> *`, é possível torná-lo instância de `Functor`? A resposta é um grande **não**! E tal resposta causa estranheza, pois até agora qualquer tipo mostrado era possível tornar um funtor.

Tendo em mente `(a -> b) -> f a -> f b`, como nossa tentativa de funtor é o `Predicado`, precisaríamos ter `(a -> b) -> Predicado a -> Predicado b`. Teríamos então duas possibilidades de instância de `Functor` para `Predicado`. A primeira seria:

```
instance Functor Predicado where
    fmap g (Predicado p) = Predicado (g . p)
```

E a segunda seria:

```
instance Functor Predicado where
    fmap g (Predicado p) = Predicado (p . g)
```

Nos dois casos, só poderíamos ter uma composição, pois `(a -> b) -> Predicado a -> Predicado b` possui dois parâmetros que envolvem: uma função de tipo `a -> b` e um `Predicado a` que possui dentro dele também uma função, que no caso é de tipo `a -> Bool`. O *pattern matching* `Predicado p` extrai a nossa função `p` de tipo `a -> Bool` de dentro.

Agora vamos analisar `g . p`. O `p` tem entrada `a` e saída `Bool`, enquanto `g` tem entrada `a`, portanto, temos um erro de compilação. Analisando `p . g`, temos que `g` tem entrada `a` e saída `b`, porém a entrada de `p` é `a`, ocorrendo um erro novamente.

Logo, não é possível criarmos uma instância de `Functor` para `Predicado`. Porém, eis que há uma classe que nesse caso nos salva:

```
class Contravariant f where
    contramap :: (a -> b) -> f b -> f a
```

Um Functor Contravariante é um funtor na categoria oposta de Hask. A categoria oposta é aquele que inverte todas as direções das da funções `->`. Ou seja, se temos uma função de tipo `Int -> String` em Hask, a mesma terá `String -> Int` na categoria oposta de Hask (que chamaremos de `Hask*`, por conveniência).

Voltemos ao `contramap`. A intuição é a mesma de um funtor: jogar uma função para dentro do contêiner, só que dessa vez a função jogada tem de ter a "flecha invertida". Pense que `a -> b`

vira `b -> a` quando jogada para dentro de `f b`, para ter o tipo de dentro de `f` batendo com a entrada de `b -> a`. No caso de `Predicado`, conseguimos que ele tenha uma instância de Funtor Contravariante:

```
instance Contravariant Predicado where
    contramap g (Predicado p) = Predicado (p . g)
```

Como a função `contramap` tem tipo `(a -> b) -> Predicado b -> Predicado a` (no caso do `Predicado`), temos que `p` da expressão `Predicado (p . g)` tem tipo `b -> Bool`, então temos que `g` possui entrada `a` e saída `b`. Esta saída entra em `p` que tem entrada `b` e saída `Bool`, nos dando `p . g :: a -> Bool` e, por fim, `Predicado (p . g) :: Predicado a`, como queríamos.

Agora, toda vez que quisermos trocar as entradas das funções de dentro de `Predicado`, usamos o `contramap`. Por exemplo, se tivermos uma função `String -> Int`, podemos jogar para dentro de `ehNegativo :: Predicado Int`, nos retornando algo de tipo `Predicado String`. Testando no GHCi:

```
Cap07> let func = contramap length ehMenor4
Cap07> let g = runPred func
Cap07> g "Haskell"
False
Cap07> g "ABA"
True
```

Podemos observar que `length` é "jogado para dentro" de `ehMenor4 :: Predicado Int` através do `contramap`. Na segunda linha, a nova função é extraída a partir de `runPred`, logo, `g` possui o tipo `String -> Bool` (`func` possui o tipo `Predicado String`) e verifica se uma `String` possui o tamanho menor que 4 e não mais um `Int`.

A partir daqui, você já tem um conhecimento um pouco mais fundo sobre funtores e está preparado para encarar **Bifuntores** e **Profuntores**, que são funtores para tipos de kind `* -> * -> *`. Os bifuntores jogam duas funções "sem mudar a seta" para dentro, enquanto os profuntores jogam uma função "sem mudar a seta" e outra "mudando a seta" para dentro do contêiner.

Apesar de meu profundo interesse em bifuntores e profuntores, a abordagem destes conceitos pode elevar a complexidade do livro que tem como objetivo ser uma introdução à programação funcional utilizando a linguagem Haskell.

7.5 MINIPROJETO: CONTINUAÇÃO USANDO FUNTORES

Para facilitar buscas de funcionários e projetos em arquivos (buscas em qualquer tipo de arquivo, não só para eles), devemos usar uma espécie de índice (assim como uma chave primária). Esse tipo que representa índice será um tipo de kind `* -> *` e instância de `Functor`.

```
module Projeto where

data Indice a = Indice {indice :: Int, dados :: a}

-- rotinas previamente implementadas

instance Functor Indice where
    fmap f (Indice i dados) = Indice i (f dados)
```

O tipo `Indice` possui um *value constructor* chamado `Indice`, com dois campos de tipos `Int` e `a`. Esse tipo representa um valor indexado. `Indice Pessoa` e `Indice Projeto` indicam que estamos criando campos indexados sem

mexer nos tipos já implementados, além de separar índice de dados (o pacote `persistent` propõe algo similar ao mexer com dados oriundos de uma tabela de banco de dados).

Como `Indice` é algo de kind `* -> *`, aproveitamos para criar uma instância de funtor. Fazendo isso, não precisaremos ficar extraindo os dados do índice toda vez que for necessário operar com `Pessoa` ou `Projeto`.

Por exemplo, a função `promover :: Pessoa -> Pessoa` não pode ter a entrada como `Indice Pessoa`, então temos de "jogar para dentro" a função `promover` através da expressão `Indice i (f dados)`. Note que o parâmetro `i` é ignorado, pois não é de tipo variável e, portanto, fica fora da ação do `fmap`.

Com a instância de `Functor` para índice finalizada, é possível operar com o contêiner `Indice` sem a necessidade de ficar extraindo o valor de dentro.

7.6 EXERCÍCIOS

7.1) Faça uma instância de `Functor` para o tipo `Coisa`, definido no início do *capítulo 5*. A função `g` deve "ir para dentro" em todas as coordenadas de `Coisa`. No caso de `ZeroCoisa`, o `fmap` deve retornar `ZeroCoisa`.

7.2) Aproveitando o exercício anterior, faça uma instância de `Applicative Functor` para o tipo `Coisa`.

7.3) Crie a função `mult234 :: Double -> Coisa Double` que multiplica por 2 a primeira coordenada, por 3 a segunda, e por 4 a terceira. Use a instância de `Applicative` feita no exercício

anterior.

7.4) Escreva uma instância para `Functor` e `Applicative` para o tipo `Arvore`, visto no *capítulo 5*.

7.5) Escreva uma instância de `Functor` para o tipo `data Fantasma a = Fantasma`.

7.6) Escreva uma possível instância de `Functor` para o tipo `data Dupla a = Dupla a Int a`.

7.7) É possível criar uma instância de `Functor` para o tipo `Derp a = Derp {runDerp :: Bool -> a}` ? Justifique e, em caso positivo, crie a instância de `Functor` e `Applicative`.

7.8) (Opcional) Implemente:

- O tipo `NovoPred` possui um *value constructor* de mesmo nome, contendo o campo `runNovoPred` de tipo `Maybe a -> Bool`;
- Crie uma instância de Funtor Contravariante para o tipo `NovoPred`.

7.7 CONCLUSÃO

Este capítulo foi dedicado a mais uma classe (*typeclass*) especial, os funtores. Diferentemente do capítulo anterior, que mostrou uma noção matemática, aqui foi abordado o uso prático de um funtor.

Todo tipo de kind `* ->*` pode ser pensado como um rótulo para o tipo, uma caixa rotulante, como: `[]` representa uma estante onde pode ser guardadas várias coisas de tipo `a`; `Maybe`

representa um valor `a` ausente ou não; um funtor é apenas como vamos jogar uma ação dentro dessas caixas.

O Funtor Aplicativo ajuda no caso no qual uma função `f` está dentro desses tipos de kind `* -> *`, e que `<*>` é o análogo de `$` para tipos de kind `* -> *`. O entendimento deste capítulo prepara o aluno para o entendimento do último conceito teórico do livro, as famosas **Monads**.

Capítulo 8

MÔNADAS

Monad (ou Mônada, em português) é um conceito oriundo da Teoria das Categorias e serve, entre outras coisas, para representar matematicamente a computação com efeitos colaterais (**side-effects**), como leitura e escrita de arquivos, aplicações **web**, banco de dados, entre outras coisas. A partir deste capítulo, você será capaz de entender e desenvolver aplicações em Haskell, e entender os benefícios do uso deste conceito para a confecção de aplicações mais robustas.

Para começar, precisamos introduzir uma nomenclatura presente na literatura sobre a linguagem Haskell quando se pesquisa o termo **Monad**. Tal nomenclatura será útil para o entendimento do conceito, mas no dia a dia sua presença é meramente opcional. A intenção é analisar a categoria dos endofuntores do **Hask**, e não **Hask** em si, pois mônadas são estruturas que moram nesta nova categoria.

Ressalta-se ao leitor que não é necessário ser um *expert* em Teoria das Categorias. Entretanto, ter um conhecimento básico do que está por trás dos conceitos aqui em Haskell lhe proporcionará um diferencial incrível frente ao mercado de trabalho, já que você poderá enxergar tais estruturas em sua linguagem de trabalho e escrever um código de melhor manutenção, que precise de menos

testes e em um alto nível de abstração. Basta olhar para os promises do AngularJS após o entendimento deste capítulo para sentir o poder destes conceitos.

8.1 TRANSFORMAÇÕES NATURAIS

Uma mônada, assunto principal deste capítulo, é composta por um funtor e duas transformações naturais. Para entender melhor o conceito em foco, será introduzida primeiramente a definição de transformações naturais.

Uma **transformação natural** é outro conceito emprestado da Teoria das Categorias. Aqui podemos enxergá-la como uma função entre dois funtores. Como o formalismo matemático está além do escopo deste livro, podemos pensar em transformações naturais como funções de tipo `F a -> G a`, em que `a` é um tipo de kind `*` qualquer (novamente lembrando `Int`, `String`, `Char`, `Bool` etc.), `F` e `G` um tipo de kind `* -> *` qualquer (`[]`, `Identity`, `Maybe`, `IO` etc.).

Uma observação quanto ao funtor `Identity`, do módulo `Control.Monad.Identity`, é que ele não faz nenhuma mudança de contexto, ou seja, `Identity Int` é apenas `Int`, assim como `Int` pode ser visto como `Identity Int`. Em instantes, veremos o motivo de pensarmos neste funtor que não faz nada.

As transformações naturais são usadas quando precisamos de funções entre contêiners (tipos de kind `* -> *`) sem nos preocuparmos com o que há dentro. Vamos agora analisar alguns exemplos de transformações naturais:

```
maybeLista :: Maybe a -> [a]
maybeLista (Just x) = [x]
maybeLista Nothing = []
```

Esta transformação natural troca o funtor `Maybe` pelo funtor `[]`. A entrada desta função possui um *pattern matching* contra `Maybe`. Caso tivermos o *value constructor* `Just x` - lembrando de que `x :: a` -, este elemento é colocado dentro de uma lista usando o símbolo `[]`; caso seja `Nothing`, a lista vazia é retornada.

Os próximos exemplos serão mais importantes para continuarmos.

```
toLista :: a -> [a]
toLista x = [x]

Prelude> toLista 6
[6]
```

A função `toLista` pode ser pensada como uma transformação natural que troca o funtor `Identity` (por conta de `Identity a = a`) pelo funtor `[]`, simplesmente colocando um elemento `x :: a` dentro desta lista. Essa função nos mostra que, ao criarmos uma lista em um programa, estamos lidando com uma transformação natural entre `Identity` e `[]`.

Como dito na introdução deste capítulo, é importante analisarmos a categoria dos endofuntores, cujos seus objetos são os funtores (quaisquer tipo de kind `* -> *` que seja instância de `Functor`). Seus morfismos são as transformações naturais que acabamos de ver.

8.2 DEFINIÇÃO

Matematicamente, uma mônada na categoria *Hask* consiste, basicamente, em três "coisas":

- Um endofuntor `m`, ou seja, um tipo de kind `* -> *`, instância de `Functor`.
- Uma transformação natural que troca `Identity` por `m`, chamada `return`. Note que `return :: a -> m a`.
- Uma transformação natural que troca `m m` por `m`, chamada `join`. Note que `join :: m (m a) -> m a`.

E devem seguir as leis das mônadas. Essas leis são as mesmas que as leis válidas para monoides, pois é sabido que uma mônada em um Hask é um monoide na categoria dos endofuntores de Hask (Os objetos dessa categoria são Funtores, os morfismos são as transformações naturais e a noção de composição não é dada por `(.)` e sim por `(<=<)` ao qual é um caso especial de `(>>=)` a ser visto abaixo. A estrutura de monoide vem do fato de `return` agir como `mempty` e o `join` como `mappend`).

Em Haskell, as mônadas são representadas pelo **typeclass Monad**, estão implementadas no módulo `Control.Monad` e possuem kind `* -> *`. Isto quer dizer que, dado um tipo `m` de kind `* -> *` instância de `Functor`, é possível transformá-lo em uma mônada, se soubermos como definir as transformações naturais `return` e `join`.

Porém, em Haskell, `join` é uma função que vem de graça, pois ela é escrita em termos da função infixa `>>=`, chamada **bind**.

Conclusão: para usarmos instâncias de `Monad` para qualquer

funtor `m`, é necessário sabemos como implementar `return` e `>>=` para este funtor `m`, já que cada um possuirá suas peculiaridades.

Os próximos tópicos nos trarão informação sobre `return`, `>>=` e `join`.

return

A transformação natural `return` representa como devemos pegar um valor de tipo `a` e colocar dentro de um contêiner de kind `* -> *` `m`, que possui uma instância de funtor. Ludicamente, pense que `a` tem o tipo `Livro` e `m` o tipo `Caixa`, então o ato de por seu livro na caixa chama-se `return`.

Para fixar, em vez de falar "Colocarei o livro na caixa", fale "return livro", que dará no mesmo! Outro exemplo é se estivermos focados no funtor `[]` e no funtor `Maybe`, então:

```
Prelude> return 6 :: [Int]
[6]

Prelude> return 6 :: Maybe Int
Just 6
```

Lembre sempre de que `::` indica o tipo que queremos resolver na expressão anterior. A primeira expressão coloca o número `6` no funtor `[]` e, na seguinte, coloca o mesmo número dentro do funtor `Maybe`. Se não colocássemos `::`, teríamos o seguinte tipo para a expressão `return 5`:

```
Prelude> :t return 5
return 5 :: (Monad m, Num a) => m a
```

Isso é uma restrição para a classe `Num` em `a`, e outra para a

classe `Monad` em `m`, ou seja, `m` é alguma mônada não especificada ainda. Lembre-se, `return` não faz nunca operação alguma com o valor de dentro, ele só coloca dentro de um funtor.

join e bind (>>=)

A transformação natural (de agora em diante, chamaremos de função mesmo) `join` e a função infixa `>>=` estão intimamente ligadas. Se o Haskell pedisse para definirmos `join`, ganharíamos de graça `>>=`. Porém, a linguagem nos pede para definir sempre `>>=`, logo, ganhamos de graça `join`.

O "ganhar de graça" quer dizer que basta definir um que o outro já fica definido automaticamente, sem qualquer intervenção humana.

A função `join` simplesmente junta dois funtores em um só, ou seja, `join` transforma `[[]]` em `[]`, `Maybe Maybe` em `Maybe`, `IO IO` em `IO`, e assim por diante (note que os funtores devem ser iguais). Ludicamente, vamos supor que sua mãe pegue um livro seu, coloque-o em uma caixa, e depois pegue esta caixa e coloque-a em outra. O que você teria? Seu livro dentro da caixa da caixa. O que o join faz? Jogaria uma caixa fora e seu livro ficaria dentro de uma caixa apenas. Veja um exemplo:

```
Prelude> :m Control.Monad

Prelude Control.Monad> join [[4]]
[4]

Prelude Control.Monad> join (Just (Just True))
Just True

Prelude Control.Monad> join [[[20]]]
[[20]]
```

```
Prelude Control.Monad> join ["W"]
"W"
```

Esses códigos nos mostra uma camada de funtor sendo ignorada nos três exemplos. A pergunta que vem na cabeça é: e se tivermos mais de um elemento nas listas? A resposta será dada na próxima seção.

Entendido a função `join`, é necessário o entendimento de `>>=`, que talvez, até aqui, seja a função mais importante vista. Primeiramente, vamos inspecionar seu tipo:

```
Prelude> :m Control.Monad

Prelude Control.Monad>:t (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Primeiro, note que há uma restrição em `m`, dizendo que este deve ser instância de `Monad` (obviamente). A função de alta ordem `>>=` possui como entrada *um valor* `m a` e *uma função de tipo* `a -> m b`, e na saída temos um valor de tipo `m b`. Vamos relembrar da função `|>` vista no *capítulo 4*; ela foi definida como:

```
(|>) :: a -> (a -> b) -> b
(|>) x f = f x
```

Agora compare, sem a restrição para ativarmos nossa habilidade de reconhecer padrões.

```
(>>=) :: m a -> (a -> m b) -> m b
(|>) :: a -> (a -> b) -> b
```

Notou algo de similar? Claro que sim. Se retirarmos o `m` de `>>=`, caímos em `|>`, logo `>>=` é o `|>` para mônadas! Só para lembrar, há o operador `=<<`, que não vamos focar aqui, mas ele possui o tipo:

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b
```

Compare este com a função `$`, vista no *capítulo 4*:

```
($) :: (a -> b) -> a -> b
```

Logo, `=<<` é o `$` para mônadas! Só um detalhe: se escolhermos `m` como o funtor `Identity`, ou seja, `Identity a = a`, obteríamos:

```
(>>=) :: Identity a -> (a -> Identity b) -> Identity b
```

Isso nos daria **`(>>=) :: a -> (a -> b) -> b`**, ou seja, `>>=` e `|>` são os mesmos sob o funtor `Identity` (assim como `=<<` e `$`)! Como `>>=` recai em algo que já vimos, só precisamos entendê-lo nesse novo contexto e estaremos prontos para trabalhar em casos mais concretos.

Suas entradas são um valor de tipo `m a` e uma função de tipo `a -> m b`, temos então uma caixa `m` e um item `a`. Note que o tipo `a -> m b` possui entrada `a`, ou seja, poderíamos jogar tal função, via `fmap`, dentro da caixa `m a` e esta transformaria o elemento de dentro em `m b`. O resultado seria algo de tipo `m (m b)` que, com a ajuda do `join`, transformaria em `m b` novamente, que é o retorno da função `>>=`. Logo, teríamos:

```
x >>= f = join (fmap f x)
```

Isso é exatamente o processo descrito. Note que o `join` se faz extremamente necessário, pois é é preciso acabar com um dos `m`. Vale ressaltar, mais uma vez, que tanto `>>=` quanto `|>` trabalham com o **valor na função, e não a função no valor**. Basta relembrar:

```
x |> f = f x
```

Note a ausência de `join` e `fmap` para este caso sem mônadas. Vamos a um exemplo. Vamos analisar a seguinte expressão:

```
Prelude Control.Monad> Just "Olá" >>= (\x -> Just (x ++ " Mundo")
)
Just "Olá Mundo"
```

O valor de entrada de `>>=` tem tipo `Maybe String`, e a função de entrada tem tipo `String -> Maybe String`, logo `a = String`, `b = String` e `m = Maybe`. Para entender melhor o resultado `Just "Olá Mundo"`, vamos analisar friamente o cálculo da expressão anterior.

```
Just "Olá" >>= (\x -> Just (x ++ " Mundo")) =
join (fmap (\x -> Just (x ++ " Mundo")) (Just "Olá")) =
join (Just ((\x -> Just (x ++ " Mundo") "Olá")) =
join (Just (Just "Olá Mundo")) =
Just "Olá Mundo"
```

Primeiramente, o lambda `(\x -> Just (x ++ " Mundo")` entra para dentro de `Just` presente no valor `Just "Olá"`, com ajuda da função `fmap`. Depois, `"Olá"` vira argumento do lambda `(\x -> Just (x ++ " Mundo")`, retornando `Just(Just "Olá Mundo)"`, que é argumento de `join`, finalizando o cálculo.

8.3 NOTAÇÃO DO

A escrita de funções que usam os operadores monádicos `join`, `return` e `>>=` pode ser bastante complicada para novatos na linguagem. Por isso, existe um facilitador na sintaxe chamado notação `do`, que permite o uso das funções anteriores de uma forma mais convencional e parecida com linguagens imperativas.

Primeiramente, vamos relembrar alguns conceitos para depois introduzir a notação. Vimos que `>>=` e `|>` são "irmãos" e, com isso, temos que ambos se associam à esquerda com máxima prioridade. Lembre-se do exemplo do *capítulo 4*:

```
module Monadas where

import Control.Monad

funcI :: String -> String
funcI x = x
    |> reverse
    |> take 3
    |> tail
    |> (x ++)
```

Uma função análoga, usando o funtor `Maybe`, pode ser escrita e, em vez de usarmos `|>`, usaremos `>>=`.

```
funcMaybe :: String -> Maybe String
funcMaybe x = return x
    >>= \y -> return (reverse y)
    >>= \z -> return (take 3 z)
    >>= \w -> return (tail w)
    >>= \u -> return (x ++ u)
```

Com as chamadas:

```
Prelude> :l Monadas.hs

Monadas> funcI "Haskell"
"Haskelle"
Monadas> funcMaybe "Haskell"
Just "Haskelle"
```

Isso nos mostra as similaridades das funções em questão e como elas se associam à esquerda com alta prioridade. Para entender `funcMaybe`, devemos entender a instância de `Monad` para `Maybe` já implementada na linguagem.

```
instance Monad Maybe where
    return x = Just x
    (Just x) >>= f = f x
    Nothing >>= f = Nothing
```

Note que `(Just x) >>= f = f x` ignora o primeiro `Just` e faz com que `x` seja a entrada da função `f`, pois ambas possuem o mesmo tipo `x :: a` e `f :: a -> m b`.
Voltando a `funcMaybe`, o primeiro cálculo é `return x >>= \y -> return (reverse y)`, que se traduz em `Just "Haskell" >>= \y -> Just (reverse y)`.

Como visto na definição de `Monad` para `Maybe`, temos que "Haskell" vira argumento do lambda `\y -> Just (reverse y)`, logo `y="Haskell"` nos dando `Just "lleksaH"`.

No próximo `Just "lleksaH" >>= \z -> return (take 3 z)`, teremos o retorno `z="lleksaH"`, portanto, o resultado `Just "lle"`. A expressão continua com `Just "lle" >>= return (tail w)`, assim temos `w="lle"` e retornamos `Just "le"`.

Por fim, temos `Just "le" >>= \u -> return (x ++ u)` com `x=Haskell` e `u=le`, retornando, finalmente, `Just "Haskelle"`. Lembre-se de que tudo isso foi graças a alta prioridade à esquerda, que `>>=` possui. Graças a isto, podemos transformar expressões parecidas com as que vimos em `funcMaybe` nestas usando a notação `do`.

```
funcMaybe :: String -> Maybe String
funcMaybe x = do
    y <- return x
    z <- return (reverse y)
    w <- return (take 3 z)
    u <- return (tail w)
    return (x ++ u)
```

Esta notação transforma a escrita funcional, que estávamos habituados, em uma escrita parecida com as linguagens de programação imperativas! Dê uma olhada: `<-` lembra a atribuição, ausente aqui graças a um truque com os lambdas.

Tendo isso em mente, quando vemos `return x >>= \y -> return (reverse y) >>= \z ...`, é só ignorarmos `->` e transformar `>>=` em `<-`, e pronto! Podemos de agora em diante escrever de uma maneira "imperativa" aqui no Haskell, sem nos preocuparmos com *side-effects* ou valores impuros.

Algo a se frisar: se o retorno de uma função não for monádico (algo como `m a`), então sempre devemos chamar a função `return` ou a notação `do` quando usarmos `>>=`. Por isso, em `funcMaybe`, as expressões `x`, `reverse y`, `take 3 z`, `tail w` e `x ++ u` são argumentos de `return` (nenhuma possui algum retorno monádico).

Outra coisa: nunca confunda o `return` do Haskell com o do Java, por exemplo. São duas coisas completamente diferentes.

8.4 A MÔNADA []

Finalmente, vamos mostrar a instância de `Monad` para listas (já implementadas na linguagem).

```
instance Monad [] where
    return x = [x]
    xs >>= f = [y | x <- xs, y <- f x]
```

A função `return` simplesmente coloca um elemento `x` na lista, enquanto `>>=` pega todos os valores `x<-xs` dela e coloca-os como entrada de `f`. Veja um exemplo:

```
Prelude Control.Monad> [2,3] >>= \x -> [x+1]
[3,4]
```

Caso a função tenha várias expressões em cada posição, os valores de entrada serão os mesmos. Observe:

```
Prelude Control.Monad> [2,3] >>= \x -> [x+1,x-1]
[3,4,1,2]
```

O valor de retorno `[3,4,1,2]` representa o cálculo de `[2+1,3+1,2-1,3-1]`. Como listas são mônadas, a notação `do` pode ser usufruída também seguindo as mesmas regras vistas na seção anterior.

```
expr :: [Int]
expr = do
    x <- [2,3]
    [x+1,x-1]
```

8.5 EXERCÍCIOS

8.1) Faça um tipo `Caixa` com um *type parameter* `a` e três construtores chamados `Um`, `Dois` e `Tres` possuindo um, dois e três campos de tipo `a`, respectivamente.

- Faça uma instância de Functor para o tipo `Caixa`. A função deve ser aplicada em todas as coordenadas dos valores (`Um`, `Dois` ou `Tres`).
- Crie uma instância de `Monad` para o tipo `Caixa`. Seu `return` deve ser o *value constructor* de `Um`.

Observação: quando definir `>>=` para `Caixa`, o valor a entrar em `f` segue as regras:

- `Um`: o único campo entra na função (análogo ao

`Maybe`);

- `Dois` : o segundo campo entra;
- `Tres` : o terceiro campo entra.

8.2) Crie uma função `mult234 :: Double -> Caixa Double` que receba uma parâmetro x e devolva o dobro de x na primeira coordenada, o triplo na segunda e o quádruplo na terceira usando o operador `>>=` .

8.3) Determine o valor das expressões a seguir (caso seja possível), sem usar o GHCi:

- `Tres 1 2 3 >>= mult234 >>= mult234`
- `Dois 2 4 >>= mult234`
- `:kind Coisa`
- `Dois 2 3 >>= \_ -> Dois 7 7`

8.4) Faça um exemplo, usando a notação `do` , de um trecho qualquer de código usando sua `Monad Caixa` .

8.6 CONCLUSÃO

Neste capítulo finalizamos o ferramental teórico necessário para programar em Haskell em plena produtividade. O conceito de mônada é fundamental para ir além na linguagem e fazer aplicações reais.

Vimos aqui os conceitos de transformações naturais, mônadas e sua tríade de funções. Por último, vimos o importantíssimo `>>=` e os benefícios de sua existência, dando luz à notação `do` que permite ao programador se sentir em um ambiente similar ao encontrado nas linguagens imperativas.

No próximo capítulo, veremos apenas a parte prática com a mônada `IO` e como ler dados da entrada padrão e leitura/escrita em arquivos texto. A leitura e compreensão deste capítulo situam o leitor em uma posição quase avançada na linguagem, e permitirão uma maior tranquilidade quando se depararem com aplicações web e banco de dados.

Capítulo 9

MÔNADA IO

Neste capítulo, apresentaremos a mônada IO, talvez a mais importante da linguagem, pois é ela que realiza a conexão da linguagem Haskell com o complicado "mundo real". Esta mônada é responsável pela leitura e escrita de arquivos e conexão com a entrada padrão, por exemplo.

Os frameworks web e as ferramentas de persistência da linguagem Haskell são consequência desta mônada. Este capítulo será totalmente prático, pois a teoria para entender o que há por trás da mônada IO foi construída previamente.

O tipo `IO` possui kind `* -> *` e instâncias para as importantes classes `Monad`, `Functor` e `Applicative`. Vale relembrar sempre que não é permitido ao programador retirar o valor de dentro de `IO`.

Por exemplo, se o número `5 Int` for lido do teclado, este terá o tipo `IO Int`, implicando que não podemos somar diretamente este valor lido do teclado com um valor `7` fixo e puro (sem ser lido de fonte externa), ou até mesmo somado com outro valor lido pelo teclado. O rótulo `IO` impedirá essas manipulações de dados e manterá um estado imutável por toda a aplicação.

9.1 COMPILANDO UM PROGRAMA "OLÁ MUNDO"

Graças à mônada `IO` , finalmente será possível compilar um programa `"Olá mundo"` . Basta nomearmos o módulo de `Main` e definirmos a função `main` que terá o tipo `IO ()` , análogo ao `void` do Java.

O tipo `()` é algo com um *value constructor* apenas chamado `()` , algo como `data () = ()` . Ele é importante para representar funções "sem retorno", como por exemplo, imprimir um caractere na tela.

Com isso em mente, podemos escrever nosso primeiro programa (até agora usamos um REPL pelo GHCi). Precisamos criar um arquivo chamado `Principal.hs` e, em seguida, escrever a função `main` , como vemos no exemplo a seguir.

```
module Main where

main :: IO ()
main = putStrLn "Olá Mundo!"
```

A função `main` representa o início de seu programa, e é ela que o GHCi vai procurar para gerar o executável. Para compilar, basta digitar em um terminal a linha:

```
$ ghc Principal.hs -o Principal
```

Então, aparecerá a seguinte mensagem:

```
[1 of 1] Compiling Main             ( Principal.hs, Principal.o )
Linking Princpal ...
```

E para executar, digite no terminal (usando aqui bash):

```
$ ./Principal
```

```
Olá Mundo!
```

Pronto, nosso primeiro programa está concluído. Note que é bem simples a escrita dele, porém a teoria foi importante para entender o que há por trás do `IO`.

A função `putStrLn` possui o tipo `String -> IO ()`, recebe um valor `String` e imprime seus caracteres na tela retornando `()` para simular o `void`. Ou seja, nenhum valor útil será retornado após a execução da função.

Essa função imprime caracteres linha a linha. Se quiséssemos outra linha, poderíamos nos beneficiar da notação `do`, vista no capítulo anterior.

```
module Main where

main :: IO ()
main = do
    putStrLn "Olá Mundo!"
    putStrLn "Haskell"
```

Após os passos de compilação, teríamos duas linhas de impressão. Vale notar que podemos escrever essa versão do `main` em termos do `bind (>>=)`, ou melhor, usando `>>` que é a versão do bind para funções que retornam `()`. Algo do tipo `valor >>= \_ -> função`.

Note que o valor pode ser oriundo de uma computação sem retorno e, assim, é ignorado pelo *pattern matching* `_` na entrada da função. Desta forma, a expressão `valor >>= \_ -> função` pode ser reescrita como `valor >> função`, imitando o sequenciamento visto no quadro anterior. Reescrevendo essa função, obteremos:

```
module Main where
```

```
main :: IO ()
main = putStrLn "Olá Mundo!" >> putStrLn "Haskell"
```

Apesar de `>>` ser amplamente usado, manteremos a preferência pela notação `do`.

9.2 EXEMPLOS PRÁTICOS

Nesta seção, estudaremos a fundo as funções `putStrLn`, `print`, `readLn` e `getLine`, sendo as duas primeiras de impressão na tela e as duas últimas para leitura da entrada padrão. A função `putStrLn` só funciona com `String`, enquanto `print` funciona com qualquer tipo que seja instância de `Show` (como visto no *capítulo 5*).

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
Prelude> :t print
print :: Show a => a -> IO ()
```

O `getLine` lê uma `String` da entrada padrão, enquanto `readLn` é mais geral, executando uma conversão de tipos, de `String` para o tipo desejado, desde que este tipo seja instância de `Read`.

```
Prelude> :t getLine
getLine :: IO String

Prelude> :t readLn
readLn :: Read a => IO a
```

Em casos de ambiguidade, teremos de forçar a conversão para o tipo em questão usando `:: IO Tipo`. Por exemplo, `readLn :: IO Int` indica a leitura de algo a ser convertido para inteiro. O cuidado que devemos ter é que se precisarmos ler um inteiro e

outro tipo é encontrado, como `Char`, teremos problema no `read`, causando uma exceção **Prelude.readIO: no parse**.

Agora mostraremos vários exemplos de programas feitos em Haskell para que o leitor se acostume com a escrita da notação `do` e também aprenda a usar a mônada `IO`.

Para todos os exemplos, lembre-se da primeira linha `module Main where`. Você pode compilar os programas um a um, ou renomear todas as funções e usá-las no GHCi (este executa funções de IO sem nenhum problema).

Exemplo 1: soma de dois números

```
main :: IO ()
main = do
    putStrLn "Digite um número: "
    x <- readLn
    putStrLn "Digite outro número: "
    y <- readLn
    putStrLn "Resultado: "
    print $ x + y
```

Note que a função `putStrLn` exibe na tela apenas `String`, enquanto o `print` exibe a soma `x + y`, pois toda instância de `Num` possui instância para `Show`. Se quiséssemos ler `x` e `y` como `Double`, por exemplo, poderíamos usar `readLn :: IO Double` como mencionando anteriormente.

Exemplo 2: loop while funcional

```
main :: IO ()
main = do
    let loop = do
        putStrLn "Qual seu nome?"
        nome <- getLine
        if (nome == "") then do
```

```
            putStrLn "Erro... "
            loop
        else
            putStrLn $ "Ola " ++ nome
    loop
    putStrLn "Fim"
```

O programa anterior executa um `loop`, similar ao `while`, de uma forma funcional utilizando o conceito de recursão. Note que, dentro da função `main`, definimos a função auxiliar `loop` (similar ao `where`, a cláusula `let` permite-nos definir funções localmente, antes de serem usadas).

A função `loop` imprime na tela a mensagem `"Qual seu nome"`, e pede ao usuário digitar uma `String` lida pelo `getLine` e associada a `nome`. A função `if then else` é similar às estruturas condicionais de uma linguagem imperativa. Se o `nome` for vazio, será impresso na tela uma mensagem de erro e a função `loop` será chamada novamente (recursão); caso contrário, será mostrado um cumprimento de olá, seguido do valor em `nome`.

A cláusula `let` apenas definiu a função `loop` localmente. Como dito anteriormente, a função só é chamada na linha 11 com a instrução `loop`, fazendo a primeira chamada dela. Após o loop ser completado, uma mensagem de `"Fim"` é exibida. Esse exemplo realça a semelhança da notação `do` com as linguagens imperativas.

Exemplo 3: loop for funcional

```
import Control.Monad

main :: IO ()
main = do
    z <- readLn
    forM_ [1..z] $ \i -> do
```

```
        print i
```

Nesse exemplo, um inteiro `z` será lido na entrada padrão, e ele será o limite máximo para a expressão `[1..z]`, que criará uma lista de números de 1 até `z`.

A função `forM_` repetirá a execução do lambda `\i -> print i` por `z` vezes. Nesse caso, a variável de entrada do lambda `i` receberá cada valor da lista `[1..z]`, primeiramente 1, depois 2 e assim por diante. Para cada valor da lista, esse será impresso na tela conforme a expressão `print i`. Esse exemplo mostra na tela cada elemento da lista em linhas diferentes.

Exemplo 4: for funcional

```
import Control.Monad

main :: IO ()
main = do
    z <- readLn
    let dentro i = do
        putStrLn $ "Número " ++ (show i)
        readLn
    ns <- mapM dentro [1..z]
    putStrLn $ "Resultado: " ++ (show $ sum ns)
```

Nesse programa, é mostrada na tela, em cada linha, cada linha da lista `[1..z]` (números de 1 a `z` de um em um), em que `z` foi informado pelo usuário. Ao final, é mostrada a soma dos números.

A função `mapM :: Monad m => (a -> m b) -> [a] -> m [b]` joga uma função de ação monádica de tipo `a -> m b` dentro de uma lista de tipo `[a]`, retornando uma lista de tipo `[b]` dentro da mônada `m`, especificada na função passada como parâmetro.

O intuito é o mesmo do `map`, porém com uma mônada envolvida. A cláusula `let` define uma função local de nome `dentro` que recebe um `Int` e executa as ações da mônada `IO`: mostrando cada número na saída padrão e lendo um número da entrada padrão. Assim, teríamos algo parecido com tipo `Int -> IO Int`, entrada `i` de tipo `Int`, e saída `IO Int` por causa do `readLn`.

A função `mapM` faz com que a função `dentro` seja chamada, tendo como argumento todos da lista `[1..z]` e, ao final, mostra na saída a soma dos números digitados.

```
import Control.Monad

main :: IO ()
main = do
    z <- readLn
    ns <- forM [1..z] $ \i -> do
            putStrLn $ "Numero " ++ show i
            readLn
    putStrLn $ "Resultado: " ++ (show $ sum ns)
```

Aqui é feito a mesma ação, porém escrita de outro estilo. Neste caso, temos algo mais próximo das linguagens imperativas (lembra mais ou menos o `for` do Python). Note que a função `forM` é a mesma que `mapM`, porém as duas possuem os parâmetros trocados de posição. Vamos inspecionar os tipos no GHCi:

```
Prelude> :t mapM
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
Prelude> :t forM
forM :: Monad m => [a] -> (a -> m b) -> m [b]
```

Então, vemos a troca de posições dos parâmetros. Isso é recorrente na linguagem Haskell e, dependendo da posição dos argumentos, podemos escrever uma função de várias formas

(lembremos de `$` e `|>` do *capítulo 4*). Há uma função no Haskell, chamada `flip`, que troca dois parâmetros de posição. Um leitor perspicaz identificará que `forM = flip mapM` e que `|> = flip ($)`.

Exemplo 5: adivinhando uma carta do baralho

Vamos construir um jogo simples de cartas feito em programação funcional pura. Esse exemplo nos mostra que jogos também podem ser desenvolvidos nesse tipo de paradigma. Nele mostraremos o uso de números aleatórios usando o pacote `random`.

Para rodar o exemplo a seguir, devemos instalar o pacote `random` pelo comando `cabal install random` em seu terminal.

```
data Naipe = Ouros | Espadas | Copas | Paus deriving (Show, Enum)

data Valor = Dois | Tres | Quatro | Cinco | Seis |
             Sete | Oito | Nove | Dez | J | Q | K | A
             deriving (Show, Enum)

data Carta = Carta {valor :: Valor, naipe :: Naipe} deriving Show

main :: IO ()
main = do
    let acertou True = "Você acertou"
        acertou False = "Errou..."
    baralho <- return [Carta x y | x<-[Dois .. A], y<-[Ouros .. P
aus]]
    cartaNum <- randomRIO (1, length baralho)
    carta <- return $ baralho !! cartaNum
    putStrLn "Escreva a carta para adivinhar: "
    palpite <- readLn
    putStrLn $ "Sua carta foi " ++ show (valor carta) ++ " de " +
+ show (naipe carta)
    putStrLn $ acertou $  carta == palpite
```

Primeiramente, definimos a função `acertou`, localmente com o `let`, de tipo `Bool -> String`, usando *pattern matching* no parâmetro booleano. A `[Carta x y | x<-[Dois .. A], y<-[Ouros .. Paus]]` cria uma lista com as 52 cartas de um baralho.

Note que a expressão `Carta x y` representa o tipo `Carta` a ser montado com o `x` (tipo `Valor`) e o `y` (tipo `Naipe`). O `x` varia entre todos os *value constructors* de `Valor` e o `y` entre os de `Carta`. Lembre-se de que isto é possível graças ao `..` e à instância de `Enum` (o operador `..` depende da função `succ` definida em uma instância de `Enum`).

A lista montada ficaria da forma `[Carta Dois Ouros, Carta Dois Espadas, Carta Dois Copas, Carta Dois Paus, Carta Tres Ouros, ... , Carta A Paus]`. E o `return` coloca a lista dentro da mônada `IO`.

Um número aleatório entre 1 e 52 (`length baralho`) é gerado a partir de `randomRIO (1, length baralho)`. A função `randomRIO` é do módulo `System.Random` e recebe uma tupla com os limites mínimo e máximo para gerar um número aleatório. Note que esta ação é monádica, evitando o `return`.

A expressão `return $ baralho !! cartaNum` coloca dentro da mônada `IO` uma carta do baralho escolhida aleatoriamente sob o nome de `carta`.

O usuário deve escrever o nome da carta em formato de *record syntax*, e a instância de `Read` dos três tipos nos garante isso. Mas o leitor pode optar por fazer o tipo `Carta` sem o *record syntax* para evitar isso. Basta usar a linha `data Carta = Carta Valor Naipe deriving Show` no lugar do tipo `Carta` definido no

trecho anterior.

O programa mostrará a carta escolhida, instância de `Show` dos três tipos, e verificará se são iguais graças ao `Eq` dos três tipos. Veja um exemplo de execução:

```
Adivinha a carta:
Carta {valor = A, naipe = Ouros}
Sua carta foi Sete de Espadas
Errou...
```

Note como o usuário teve de escrever a carta. A função `read` da classe `Read`, em caso de existência de *record syntax*, lerá a entrada apenas desta forma.

9.3 MANIPULANDO ARQUIVOS

Vamos estudar três funções básicas para manipulação de arquivos:

```
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Observe que o tipo `FilePath` é um sinônimo para `String` (`type FilePath = String`). Para ler arquivos, usamos a função `readFile`; para escrever um novo arquivo, `writeFile`; e para escrever ao fim de um arquivo, o `appendFile`. Assim como na seção anterior, vamos aprender a manipular tais funções por meio de alguns exemplos.

Exemplo 6: salário total e maior de uma lista de funcionários

Vamos supor que precisamos calcular os salários a partir de

uma lista de funcionários de uma empresa, dispostos em um arquivo texto. Para isto, precisamos do arquivo `func.dat` que deverá possuir o formato:

```
Func1 3000.00
Func2 2300.00
Func3 9999.99
Func4 23000.00
```

Para realizar este cálculo, considere o trecho:

```
import Text.Printf

main :: IO ()
main = do
    lista <- fmap (map words . lines) $ readFile "func.dat"
    salarios <- return $ map (\(_:vl:_) -> read vl) lista :: IO [
Double]
    printf "%.2f\n" $ sum salarios
    print $ maximum salarios
```

Observe que, na primeira linha após o `do`, temos a leitura do arquivo a partir da expressão `readFile "func.dat"`. Tal expressão retorna em um `IO String` representando todo o arquivo e, por isso, há o `fmap` que joga "para dentro" do funtor `IO` (note que estamos usando IO como funtor aqui) a função composta de duas `(map words . lines)`.

A função `lines` quebra a `String` toda do arquivo pelo separador `\n`, retornando em um `[String]` cada linha do arquivo. Por exemplo, `Func1 3000.00\nFunc2 2300.00` é transformado em `["Func1 3000.00", "Func2 2300.00"]`.

A função `map words` faz a função `words` ser jogada "para dentro" da lista acima, e ela quebra a `String` pelo separador `" "`. Por exemplo, `"Func1 3000.00"` fica `["Func1","3000.00"]`. Logo, `lista` fica `[["Func1","3000.00"],`

`["Func2","2300.00"],["Func3","9999.99"], ["Func4","23000.00"]]` se o arquivo lido for o do exemplo anterior.

A segunda linha extrai o segundo campo das listas anteriores, restando só os valores que são `String` e devem ser convertidos para número. A expressão `map (\(_:vl:_) -> read vl) lista` faz com que o lambda `\(_:vl:_) -> read vl` seja jogado "para dentro" de uma lista de formato análogo ao de cima. Temos então que este lambda extrai a segunda posição via **pattern matching** `_:vl:_` , e converte para `Double` usando a função `read` .

Como o `map` trabalha fora do `IO` , devemos colocar a lista de salários dentro do `IO` usando o `return` . Como vimos no capítulo anterior, ou a ação é monádica ou devemos colocar um `return` para torná-la monádica.

O `:: IO Double` indica que o tipo que o `read` deve converter a `String` é `Double` . O `printf "%.2f\n"` faz o truncamento de duas casa decimais (esta função está no módulo `Text.Printf`). A soma e o máximo são realizados pelas funções `sum` e `maximum` .

Exemplo 7: uso do write e append

Nesse exemplo, o leitor aprenderá a diferença entre `write` e `append` , além de usar a função `doesFileExist` , que verifica se um arquivo existe ou não. O exemplo mostra como gravar uma `String` em um arquivo.

```
import System.Directory
```

```
main :: IO ()
main = do
    putStrLn "Digite o nome do arquivo. Será criado caso não exis
ta"
    arq <- getLine
    putStrLn "Digite uma mensagem"
    mensagem <- getLine
    existe <- doesFileExist arq
    if existe then
        appendFile arq ("\n" ++ mensagem)
    else
        writeFile arq mensagem
```

O programa tem o intuito de verificar se um arquivo existe ou não. Se existir, a mensagem digitada será escrita em uma nova linha aproveitando o conteúdo `appendFile` ; se não existir, será criado um novo `writeFile` .

Note que `arq` se refere ao nome do arquivo em questão e `mensagem` ao conteúdo a ser salvo. A função `doesFileExist` , do módulo `System.Directory` , retorna um `IO Bool` (por isso que não foi direto no `if`), indicando se o arquivo cujo nome está em `arq` existe ou não. Para a mensagem ir para a linha subsequente, em um arquivo existente, usamos a expressão `"\n" ++ mensagem` .

Os dois exemplos dados ilustram como manipular arquivos físicos na linguagem Haskell, e ressaltam a estrutura de mônada que possui o tipo de kind `* -> * IO` .

9.4 MINIPROJETO FINAL

Chegou o momento de aplicar tudo o que aprendemos para a conclusão do miniprojeto. Completaremos o sistema de RH que começou nos capítulos anteriores com as funcionalidades de

buscas e cadastro de funcionários e projetos através da mônada IO.

Com a mônada `IO`, podemos fazer buscas e cadastro de funcionários e projetos para nosso pequeno sistema de RH. Vamos mostrar o código aos poucos para facilitar o seu entendimento.

O código completo pode ser apreciado no apêndice. O leitor é encorajado a implementar as rotinas introduzidas nos capítulos anteriores.

Para a conclusão do sistema, comecemos com a redefinição das instâncias dos tipos usados e com uma instância da classe `ToJSON`, criada no *capítulo 5*, para o tipo `Indice` criado no *capítulo 7*.

```
{-# LANGUAGE BangPatterns #-}
module Main where

import System.Directory
import Control.Monad
import Control.Applicative

data Cargo = Estagiario | Programador | Coordenador | Gerente der
iving (Show, Read)

data Pessoa = Pessoa {cargo :: Cargo, nome :: String} deriving (S
how, Read)

data Indice a = Indice {indice :: Int, dados :: a} deriving (Show
, Read)

data Projeto = Projeto {nomeProjeto :: String,
                        budget :: Double,
                        envolvidos :: [Int]} deriving Show

instance ToJSON a => ToJSON (Indice a) where
    toJSON p = "{id: " ++ show (indice p) ++ ", dados: " ++ (toJS
ON $ dados p) ++ "}"
```

Precisamos estabelecer uma execução de código não

preguiçosa, ou seja, não há uma computação sob demanda e ela deve ser executada com urgência. Essa extensão, vista na primeira linha com o `pragma {-# LANGUAGE BangPatterns #-}`, permite o uso de um `!` (exclamação) na frente de um parâmetro usado dentro de uma função. Se não usarmos o `!`, a computação sob demanda vai travar o arquivo `func.dat` por causa da preguiça, pois pode acontecer uma tentativa de leitura e de escrita ao mesmo tempo.

Repare que os tipos possuem agora instâncias de `Read` para serem usadas nas buscas. A instância de `ToJSON` para `Indice` possui uma restrição em `a`. Lembrando de que vimos no *capítulo 5* que não é todo `Indice a` que poderá ser instância de `ToJSON`, apenas os `a`s que forem instâncias de `ToJSON`. Vamos introduzir agora a função `main` que representa o fluxo principal do programa.

```
main :: IO ()
main = do
    let projetos = "projetos.dat"
        pessoas = "func.dat"
    existeProj <- doesFileExist projetos
    existeFunc <- doesFileExist pessoas
    when (not existeProj) (writeFile projetos "")
    when (not existeFunc) (writeFile pessoas "")
    opcao <- menu
    case opcao of
        1 -> cadastroPessoa
        3 -> buscaPessoas
        6 -> todosSalarios
        7 -> exportarPessoas
        _ -> print "Lol"
```

Primeiramente, vamos analisar a função principal `main`, pois ela é a mais modularizada possível e vai nortear o entendimento do código como um todo. A função começa com a cláusula `let`, que

define duas `Strings` para representar o nome dos arquivos a serem usados. O primeiro será para os projetos e o último para os funcionários.

As quatro próximas linhas verificarão se existem ou não os arquivos. Se existirem, não faremos nada; caso contrário, os arquivos `projetos.dat` e `pessoas.dat` serão criados.

A função `when` é um `if` sem `else`, e seu primeiro argumento é um `Bool` que indica se a ação monádica de criação de arquivo `writeFile` será executada ou não. A expressão `opcao <- menu` indica a chamada da função `menu :: IO Int` que cria nosso menu e retorna a opção correta digitada pelo usuário.

O uso do `case` é análogo a qualquer outra linguagem: é testado os valores de `opcao` até encontrar o verdadeiro. A vantagem do `case` aqui no Haskell é que podemos usar o *pattern matching*, como por exemplo, usar a opção de ignorar `_`.

Cada opção de `case` chamará uma função de tipo `IO ()` para ter o mesmo tipo do `main`. Vamos analisar agora as funções `menu` e as que são chamadas no `case`.

```
menu :: IO Int
menu = do
        putStrLn "Lambda Systemas v1.0"
        putStrLn "Digite uma opção... "
        putStrLn "1- Cadastro de Pessoas"
        putStrLn "2- Cadastro de Projetos"
        putStrLn "3- Busca de Pessoas"
        putStrLn "4- Busca de Projetos"
        putStrLn "5- Total gasto por projeto"
        putStrLn "6- Ver salários"
        putStrLn "7- Exportar Pessoas (JSON)"
        putStrLn "8- Exportar Projetos (JSON)"
        opcao <- readLn
        if opcao < 0 || opcao > 8 then do
```

```
            putStrLn "Opção inválida"
            menu
        else
            return opcao
```

A função `menu` segue o mesmo exemplo de `loop` dado na seção de exemplos, usando a recursão para controlar opções inválidas. Note que o retorno é `IO Int`, que é definido pela expressão `menu`, dentro da parte verdadeira do `if`, e pelo `return opcao`, que coloca o inteiro digitado dentro da mônada `IO`.

Cadastro de pessoas

Aqui será apresentado o cadastro de pessoas como previsto na função `main`. Essa função lerá dados do teclado e os salvará em um arquivo texto.

```
cadastroPessoa :: IO ()
cadastroPessoa = do
    !pessId <- fmap (length . lines) (readFile "func.dat")
    pess <- Pessoa <$> (putStrLn "Cargo: " >> readLn) <*> (putStr
Ln "Nome: " >> getLine)
    appendFile "func.dat" (show (Indice (1+pessId) pess) ++ "\n")
    putStrLn "Usuário cadastrado com sucesso!"
```

Para contar quantas pessoas foram cadastradas, de modo a incrementar o índice, usamos a expressão `fmap (length . lines) (readFile "func.dat")`. Ela joga a função `length . lines` para dentro de um `IO String` representado pelo `readFile`. Note que a função de cadastro de pessoas começa com um parâmetro de lambda com nome `!pessId`, que indica o uso do `BangPatterns` visto anteriormente.

A função `lines :: String -> [String]` quebra uma `String` em várias usando o separador `\n`, e o `length` dá o

tamanho desta lista. Logo, esta linha conta os registros.

Precisamos montar o tipo `Pessoa` para salvarmos no arquivo. Esse dado fará parte do tipo `Indice`, de modo a termos como recuperá-lo de volta a partir de uma busca. Para realizar essa tarefa, usamos a expressão `pess <- Pessoa <$> (putStrLn "Cargo: " >> readLn) <*> (putStrLn "Nome: " >> getLine)`.

Utilizamos o `Applicative` para colocar `Pessoa` dentro da mônada `IO`, como explicado no fim da seção sobre funtores aplicativos. As expressões `putStrLn "Cargo: " >> readLn` (que é um `IO Cargo`) e `putStrLn "Nome: " >> getLine` (um `IO String`) entram como argumentos na função `Pessoa :: Cargo -> String -> Pessoa` (como visto no *capítulo 3*).

Usamos apenas `<$>` e `<*>`, não havendo a necessidade nenhuma de outra conversão. Lembre-se de que esse truque foi explicado usando `Form` em vez de `IO`, que no fundo são "farinha do mesmo saco".

Na linha `appendFile "func.dat" (show (Indice (1+pessId) pess) ++ "\n")`, escrevemos no arquivo `func.dat` o `Indice`, contendo um autoincremento, e a `Pessoa`, aproveitando a instância de `Show` destes tipos. O `"\n"` ao final indica quebra de linha. A função termina com uma mensagem de sucesso.

Cadastro de projetos

O código a seguir mostra como cadastrar projetos lidos da entrada padrão, em um arquivo texto.

```
cadastroProjeto :: IO ()
```

```
cadastroProjeto = do
    !projId <- fmap (length . lines) (readFile "projetos.dat")
    proj <- Projeto <$> (putStrLn "Nome: " >> readLn) <*> (putStr
Ln "Orçamento: " >> readLn) <*> (putStrLn "Envolvidos: " >> readL
n)
    appendFile "func.dat" (show (Indice (1+projId) proj) ++ "\n")
    putStrLn "Projeto cadastrado com sucesso!"
```

Essa função é análoga ao cadastro de pessoas. O seu ponto fraco é que o usuário terá de digitar o índice das pessoas envolvidas, como `[3,4,9]` indica que o projeto possui os funcionários de índice 3, 4 e 9 no projeto.

Buscas

A função a seguir funciona para buscar tanto pessoas como projetos. Ela foi feita para evitar a criação de duas funções idênticas (a menos de nomes dos lambdas) e deixar o programa o mais modularizado possível.

```
buscaPorId :: (Read a, Show a) => Int -> String -> IO (Maybe (Ind
ice a))
buscaPorId tid arq = do
    todosReg <- fmap lines (readFile arq)
    todosConv <- return $ map read todosReg
    res <- return $ filter (\ips -> indice ips == tid) todosConv
    case res of
        [] -> return Nothing
        _ -> return $ Just (head res)
```

Essa função busca qualquer pessoa ou projeto pelo índice. Ela tem como parâmetro o `id` inteiro do projeto ou pessoa, e uma `String` `arq` representando o arquivo. O arquivo `arq` é lido e transformado em uma lista de `String` na expressão `todosReg <- fmap lines (readFile arq)`. Convertemos também todas as `String` para o tipo `Indice a` (qualquer `a` instância de `Show` e `Read`, pois o tipo da função é `(Read a, Show a) => Int ->`

`String -> IO (Maybe (Indice a))`) como visto na expressão `todasPess <- return $ map read todasPessoas` .

Note que é jogado o `read` para dentro da lista, de `String` , `todosReg` ,
nos dando uma lista de `Indice a` chamada `todosConv` . Ao final, isso é colocado dentro da mônada `IO` pelo `return` (note que `map read todosReg` não é da mônada `IO` e necessita de `return`).

Filtramos da lista apenas o `Indice a` pelo inteiro `tid` , a partir de `res <- return $ filter (\ips -> indice ips == tid) todosConv` . Na expressão de filtragem, o lambda `(\ips -> indice ips == pessId)` expressa a igualdade entre o índice do registrado e o passado por parâmetro.

O `case` é usado para fazer *pattern matching* em `res` . Caso for `[]` (vazio), retornará `Nothing` ; caso contrário, será retornado o registro desejado dentro de `Just` . Essa função busca apenas o registro, não mostrando para o usuário.

Gostaríamos de que o usuário pudesse fazer uma consulta no nosso arquivo por meio de um `id` . Para isto, precisamos de funções que pedem para ele digitar.

```
buscaPessoas :: IO ()
buscaPessoas = do
    putStrLn "Digite o id do funcionário"
    pessId <- readLn
    pess <- buscaPorId pessId "func.dat" :: IO (Maybe (Indice Pes
soa))
    case pess of
        Nothing -> putStrLn "Erro..."
        Just p -> print p

buscaProjetos :: IO ()
```

```
buscaProjetos = do
    putStrLn "Digite o id do projeto"
    projId <- readLn
    proj <- buscaPorId projId "projetos.dat" :: IO (Maybe (Indice
Projeto))
    case proj of
        Nothing -> putStrLn "Erro..."
        Just (Indice i (Projeto nm bd env)) -> do
            print $ "id :" ++ show i
            print $ "nome :" ++ nm
            print $ "orçamento :" ++ show bd
            forM_ env $ \envId -> do
                pess <- buscaPorId envId "func.dat" :: IO (Maybe
Indice Pessoa))
                case pess of
                     Nothing -> print "Pessoa inválida"
                     Just p -> print p
```

Essas funções são parecidas, porém a busca de projetos mostra todas as informações da pessoa simulando, assim como um `join` do SQL. Ambas funções começam pedindo para o usuário digitar um `id`.

Após isto, a chamada da função `buscaPorId` é feita, como `buscaPorId projId "projetos.dat" :: IO (Maybe (Indice Projeto))` indica que queremos buscar um `Projeto`. Note que `:: IO (Maybe (Indice Projeto))` foi necessário pelo fato do tipo do retorno de `buscaPorId` ser `IO (Maybe (Indice Projeto))`, ou seja, forçamos que `a` seja `Projeto` aqui.

Como a busca pode não ser bem-sucedida, usamos o `case` para fazer **pattern matching** em `pess` ou `proj`. Caso sejam `Nothing`, uma mensagem de erro será mostrada; caso seja `Just`, o dado de dentro dele será mostrado na tela.

No caso de `buscaProjetos`, usamos outro *pattern matching* dentro de `Just`, dando-nos o inteiro `i` do `Indice`, o nome do

projeto `nm`, o orçamento `bd` e o envolvido `env`, que é um `[Int]`. Como ele é uma lista e queremos mostrar as pessoas a partir dos `id`s contidos, usaremos o `forM_` para jogar a ação monádica (de `IO`) para dentro dos `id`s.

```
\envId -> do
        pess <- buscaPorId envId "func.dat" :: IO (Maybe (Indice
Pessoa))
        case pess of
             Nothing -> print "Pessoa inválida"
             Just p -> print p
```

Logo, o lambda vai ter como entrada um `id` desta lista, e vai realizar a busca por `id` no arquivo de funcionários `pess <- buscaPorId envId "func.dat" :: IO (Maybe (Indice Pessoa))`. E se achar, mostrará na tela; caso contrário, mostrará um erro.

Outras funcionalidades

Seria importante para o projeto uma funcionalidade para listar os salários dos funcionários e outra para exportar o nosso arquivo texto para outro formato, em nosso caso, o `JSON`.

```
todosSalarios :: IO ()
todosSalarios = do
    putStrLn "Listando salários..."
    todasPessoas <- fmap lines (readFile "func.dat")
    todasPess <- return $ map read todasPessoas :: IO [Indice Pes
soa]
    forM_ todasPess $ \(Indice _ pessoa) ->
        print $ "Nome : " ++ (nome pessoa) ++ ", Salário: " ++ sh
ow (verSalario pessoa)

exportarPessoas :: IO ()
exportarPessoas = do
    writeFile "func.json" ""
    todasPessoas <- fmap lines (readFile "func.dat")
    todasPess <- return $ map read todasPessoas :: IO [Indice Pes
```

```
soa]
    mapM_ (\x -> appendFile "func.json" (toJSON x ++ "\n")) todas
Pess
```

As funções `todosSalarios` e `exportarPessoas` são parecidas. No caso de exportar, criamos primeiramente um arquivo `.json`. A busca é feita, assim como a conversão. Note que, nesse caso, não temos, intencionalmente, a generalidade de `buscaPorId`, pois deixaremos a cargo do leitor como exercício fazer com que essas duas funções funcionem tanto para `Pessoa` quanto `Projeto`, assim como na função citada.

```
forM_ todasPess $ \(Indice _ pessoa) ->
        print $ "Nome : " ++ (nome pessoa) ++ ", Salário: " ++ sh
ow (verSalario pessoa)
```

Em `todosSalarios`, a linha faz um *pattern matching* no tipo `Indice` para mostrar as pessoas na tela. Em `exportarPessoas`, o lambda `(\x -> appendFile "func.json" (toJSON x ++ "\n"))` representa uma inserção de registro em `func.json`, de acordo com a expressão `(toJSON x ++ "\n")` que converte `x` para `JSON`, e pula uma linha. Esse lambda será jogado na lista de `Indice Pessoa`, `todasPess`.

O leitor é convidado a implementar, por exemplo, a opção 5 e um cálculo da média salarial dos funcionários conforme função previamente feita, como exercício prático. Aproveite os vários exemplos e ponha a mão na massa, só assim para você aprender com efetividade. O código completo do projeto estará disponível no apêndice.

9.5 EXERCÍCIOS

9.1) Faça um programa que faça o usuário digitar um número,

e mostre na saída padrão se ele é par ou ímpar.

9.2) Faça um programa que mostre uma palavra em ordem reversa a partir de uma digitada pelo usuário.

9.3) Baseando-se no exemplo 5, faça um jogo de Pedra, Papel e Tesoura.

9.4) Faça um programa que calcule uma equação do segundo grau, a partir dos dados digitados pelo usuário.

9.5) Converta para o "estilo funcional" os 7 exemplos dados neste capítulo. Basta escrever os trechos da notação `do` pelo `>>= (bind)` e `>>`.

9.6) Modificando o exemplo 5, faça com que o programa mostre duas cartas aleatórias, retirando a adivinhação.

9.7) Faça um programa que peça para o usuário entrar com um número inteiro `n` e, a partir dele, o usuário deve digitar `n` linhas e estas devem ser gravadas em um arquivo.

9.8) Leia um arquivo que tenha o seguinte formato:

```
1 2
2 4
9 7
455 300
```

E verifique o maior número de cada linha e, ao final, o maior número entre todos. Estes números devem ser mostrados em um arquivo.

9.9) Uma senha é válida se há, pelo menos, 8 caracteres e, desses oito, pelo menos um deve ser numérico. Faça um programa que indique se a senha digitada pelo usuário é válida ou não.

9.10) A partir de um arquivo de senhas, mostre em outro arquivo apenas as senhas válidas e, em sua última linha, a quantidade delas.

9.11) Faça um programa que crie um arquivo contendo 6 números de 1 a 60, gerados aleatoriamente.

9.6 CONCLUSÃO

Neste capítulo, vimos como manipular entrada e saídas de dados em Haskell, tanto na saída padrão quanto em arquivos. Este capítulo deixou claro o uso da estrutura de uma mônada.

A partir daqui, você tem um conhecimento intermediário da linguagem Haskell e está preparado para aprender persistência de dados, desenvolvimento web e concorrência, por exemplo, sem muita dificuldade.

Capítulo 10

APÊNDICE

O código usado nos miniprojetos dos capítulos está disponível completamente aqui.

```
{-# LANGUAGE BangPatterns #-}
module Main where

import System.Directory
import Control.Monad
import Control.Applicative

data Cargo = Estagiario | Programador | Coordenador | Gerente der
iving (Read, Show)

data Pessoa = Pessoa {cargo :: Cargo, nome :: String} deriving (S
how, Read)

data Indice a = Indice {indice :: Int, dados :: a} deriving (Show
, Read)

data Projeto = Projeto {nomeProjeto :: String,
                        budget :: Double,
                        envolvidos :: [Int]} deriving (Read, Show


instance Functor Indice where
    fmap f (Indice i dados) = Indice i (f dados)

(|>) :: a -> (a -> b) -> b
(|>) x f = f x
```

```
infixl 9 |>

class ToJSON a where
    toJSON :: a -> String

instance ToJSON a => ToJSON (Indice a) where
    toJSON p = "{id: " ++ show (indice p) ++ ", dados: " ++ (toJS
ON $ dados p) ++ "}"

instance ToJSON Pessoa where
    toJSON p = "{nome: \"" ++ (nome p) ++
               "\", cargo: \"" ++ show (cargo p) ++
               "\", salario: " ++ show (verSalario p) ++ "}"

instance ToJSON Projeto where
    toJSON p = "{nome: \"" ++ (nomeProjeto p) ++
               "\", orçamento: \"" ++ show (budget p) ++
               "\", envolvidos: " ++ show (envolvidos p) ++ "}"

instance Monoid Projeto where
    mempty = Projeto "" 0 []
    mappend (Projeto nome1 budget1 env1) (Projeto nome2 budget2 e
nv2) =
         Projeto (nome1 ++ ", " ++ nome2) (budget1 + budget2) (en
v1 ++ env2)


verSalario :: Pessoa -> Double
verSalario (Pessoa Estagiario _) = 1500
verSalario (Pessoa Programador _) = 5750.15
verSalario (Pessoa Coordenador _) = 8000
verSalario (Pessoa Gerente _) = 10807.20

verFolha :: Pessoa -> String
verFolha p = "{nome: \"" ++ (nome p) ++
             "\", cargo: \"" ++ show (cargo p) ++
             "\", salario: " ++ show (verSalario p) ++ "}"

contratar :: Cargo -> (String -> Pessoa)
contratar cargo = Pessoa cargo

promover :: Pessoa -> Pessoa
promover (Pessoa Estagiario n) = Pessoa Programador n
promover (Pessoa Programador n) = Pessoa Coordenador n
```

```
promover (Pessoa Coordenador n) = Pessoa Gerente n
promover (Pessoa _ n) = Pessoa Gerente n

mediaSalarial :: [Pessoa] -> Double
mediaSalarial ps = (foldl calculo 0 ps) / (fromIntegral $ length
ps)
                   where
                       calculo salario pessoa = salario + verSala
rio pessoa

contratarVariosEstag :: [String] -> [Pessoa]
contratarVariosEstag ps = map (contratar Estagiario) ps

pesquisarPorNome :: String -> [Pessoa] -> [Pessoa]
pesquisarPorNome nomePesq ps = filter (\(Pessoa _ n) -> nomePesq
== n) ps

rotinaPromocao :: Pessoa -> String
rotinaPromocao p = p
                |> promover
                |> verFolha

cadastroPessoa :: IO ()
cadastroPessoa = do
    !pessId <- fmap (length . lines) (readFile "func.dat")
    pess <- Pessoa <$> (putStrLn "Cargo: " >> readLn) <*> (putStr
Ln "Nome: " >> getLine)
    appendFile "func.dat" (show (Indice (1+pessId) pess) ++ "\n")
    putStrLn "Usuário cadastrado com sucesso!"

cadastroProjeto :: IO ()
cadastroProjeto = do
    !projId <- fmap (length . lines) (readFile "projetos.dat")
    proj <- Projeto <$> (putStrLn "Nome: " >> getLine) <*> (putSt
rLn "Orçamento: " >> readLn) <*> (putStrLn "Envolvidos: " >> read
Ln)
    appendFile "projetos.dat" (show (Indice (1+projId) proj) ++ "
\n")
    putStrLn "Projeto cadastrado com sucesso!"

buscaPessoas :: IO ()
buscaPessoas = do
    putStrLn "Digite o id do funcionário"
    pessId <- readLn
    pess <- buscaPorId pessId "func.dat" :: IO (Maybe (Indice Pes
```

```
soa))
    case pess of
        Nothing -> putStrLn "Erro..."
        Just p -> print p

buscaProjetos :: IO ()
buscaProjetos = do
    putStrLn "Digite o id do projeto"
    projId <- readLn
    proj <- buscaPorId projId "projetos.dat" :: IO (Maybe (Indice
Projeto))
    case proj of
        Nothing -> putStrLn "Erro..."
        Just (Indice i (Projeto nm bd env)) -> do
            print $ "id :" ++ show i
            print $ "nome :" ++ nm
            print $ "orçamento :" ++ show bd
            forM_ env $ \envId -> do
                pess <- buscaPorId envId "func.dat" :: IO (Maybe
Indice Pessoa))
                case pess of
                     Nothing -> print "Pessoa inválida"
                     Just p -> print p

buscaPorId :: (Read a, Show a) => Int -> String -> IO (Maybe (Ind
ice a))
buscaPorId tid arq = do
    todosReg <- fmap lines (readFile arq)
    todosConv <- return $ map read todosReg
    res <- return $ filter (\ips -> indice ips == tid) todosConv
    case res of
        [] -> return Nothing
        _ -> return $ Just (head res)

todosSalarios :: IO ()
todosSalarios = do
    putStrLn "Listando salários..."
    todasPessoas <- fmap lines (readFile "func.dat")
    todasPess <- return $ map read todasPessoas :: IO [Indice Pes
soa]
    forM_ todasPess $ \(Indice _ pessoa) ->
        print $ "Nome : " ++ (nome pessoa) ++ ", Salário: " ++ sh
ow (verSalario pessoa)

exportarPessoas :: IO ()
```

```
exportarPessoas = do
    writeFile "func.json" ""
    todasPessoas <- fmap lines (readFile "func.dat")
    todasPess <- return $ map read todasPessoas :: IO [Indice Pes
soa]
    mapM_ (\x -> appendFile "func.json" (toJSON x ++ "\n")) todas
Pess

menu :: IO Int
menu = do
        putStrLn "Lambda Systemas v1.0"
        putStrLn "Digite uma opção... "
        putStrLn "1- Cadastro de Pessoas"
        putStrLn "2- Cadastro de Projetos"
        putStrLn "3- Busca de Pessoas"
        putStrLn "4- Busca de Projetos"
        putStrLn "5- Total gasto por projeto"
        putStrLn "6- Ver salários"
        putStrLn "7- Exportar Pessoas (JSON)"
        putStrLn "8- Exportar Projetos (JSON)"
        opcao <- readLn
        if opcao < 0 || opcao > 8 then do
            putStrLn "Opção inválida"
            menu
        else
            return opcao

main :: IO ()
main = do
    let projetos = "projetos.dat"
        pessoas = "func.dat"
    existeProj <- doesFileExist projetos
    existeFunc <- doesFileExist pessoas
    when (not existeProj) (writeFile projetos "")
    when (not existeFunc) (writeFile pessoas "")
    opcao <- menu
    case opcao of
        1 -> cadastroPessoa
        2 -> cadastroProjeto
        3 -> buscaPessoas
        4 -> buscaProjetos
        6 -> todosSalarios
        7 -> exportarPessoas
```

Capítulo 11

REFERÊNCIAS

HUGHES, J. *Why Functional Programming Matters*. Research Topics in Functional Programming, Addison-Wesley, 1990, pp 17-42. Disponível em: https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf.

LIPOVACA, M. *Learn You a Haskell for Great Good!* A Beginner's Guide. 2011.

O'GRADY, Stephen. *The RedMonk Programming Language Rankings: January 2016.* Fev. 2016. Disponível em: http://redmonk.com/sograd y/2016/02/19/language-rankings-1-16/.

O'SULLIVAN, B.; GOERZEN, J.; STEWART, D. *Real World Haskell:* Code You Can Believe. O'Reilly, 2008.

SNOYMAN, M. *Developing Web Applications with Haskell and Yesod.* O'Reilly, 2012.

SPIVAK, D. *Category Theory for the Sciences.* MIT Press, 2014.

PEYTON JONES, S.; HUDAK, P.; WADLER, P.; HUGHES, J. *A history of Haskell:* being lazy with class. Proceedings of the III ACM SIGPLAN, Jun. 2007.

PEYTON JONES, S. *Haskell and Erlang:* growing up together. Erlang factory meeting, 2009.

PEYTON JONES, S. *Tackling the Awkward Squad:* monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Maio, 2002. Disponível em: https://www.researchgate.net/publication/2472798_Tackling_the_Awkward_Squad_monadic_inputoutput_concurrency_exceptions_and_foreign-language_calls_in_Haskell.