Criando seus próprios tipos e typeclasses | Índice | Resolvendo Problemas Funcionalmente

Input e Output

Nós mencionamos que Haskell é uma linguagem puramente funcional. Enquanto nas linguagens imperativas você usualmente faz as coisas passando ao computador uma série de passos para executar, em programação funcional define-se mais como as coisas são. Em Haskell, uma função não pode mudar algum estado, como mudar o conteúdo de uma variável (quando uma função muda um estado, dizemos que a função tem *efeitos colaterais*). A única coisa que uma função pode fazer em Haskell é nos devolver um resultado baseado nos parâmetros que passamos a ela. Se uma função é chamada duas vezes com os mesmos parâmetros, ela tem que devolver o mesmo resultado. Embora possa parecer um pouco limitador quando você vem de um mundo imperativo, temos visto que isso é na verdade muito legal. Em uma linguagem imperativa, você não tem garantia que uma simples função que deveria apenas moer alguns números não vai incendiar sua casa, sequestrar seu cachorro e arranhar seu carro com uma batata enquanto mói esses números. Por exemplo, quando estamos fazendo uma árvore binária de busca, nós não inserimos um elemento em uma árvore modificando alguma árvore existente. Nossa função ao inserir em uma árvore binária de busca na verdade retorna uma nova árvore, porque nós não podemos modificar a original.

Embora funções incapazes de mudar estado sejam boas porque nos ajudam a manter o foco no nosso programa, existe um problema com elas. Se uma função não pode mudar qualquer coisa no mundo, como ela poderá nos dizer o que ela calculou? Com intúito de nos dizer o que ela calculou, ela tem que mudar o estado de um dispositivo de saída (usualmente o estado da tela), a qual então emite fótons que viajam para o nosso cérebro e mudam o estado da nossa mente.

Não se desespere, nem tudo está perdido. Percebe-se que Haskell na verdade tem um sistema bastante hábil para lidar com funções que têm efeitos colaterais que nitidamente separa a parte pura do nosso programa da parte impura, a qual faz todo o trabalho sujo de se comunicar com o teclado e com a tela. Com essas duas partes separadas, nós ainda podemos nos concentrar na parte pura do nosso programa e levar vantagem de todas as coisas que a pureza oferece, como preguiça, robustez e modularidade enquanto nos comunicamos eficientemente com o mundo externo.

Olá, mundo!

Até agora, nós sempre carregamos nossas funções no GHCI para testá-las e brincar com elas. Nós também exploramos a biblioteca padrão de funções dessa maneira. Mas agora, após oito capítulos ou mais, nós finalmente vamos escrever nosso primeiro programa Haskell *real*! E com bastante certeza, vamos usar o bom e velho artifício do `"olá, mundo"`.

Ei! No decorrer desse capítulo, irei assumir que você está utilizando um ambiente baseado em unix para aprender Haskell. Se você esta no Windows, eu sugiro que você faça o download do Cygwin, que é um ambiente baseado em Linux para Windows, também conhecido como "exatamente o que você procura".

Então, para os iniciantes, digite o seguinte no seu editor de texto favorito:

```
main = putStrLn "hello, world"
```

Nós acabamos de definir um nome chamado **`main`** e nele nós chamamos uma função **`putStrLn`** com o parâmetro **`"hello, world"`**. Parece medíocre, mas não é, como nós veremos dentro de poucos momentos. Salve esse arquivo como **`helloworld.hs`**.

E agora, nos vamos fazer algo que nunca fizemos antes. Vamos finalmente compilar nosso programa! Estou muito animado! Abra o seu terminal e navegue até o diretório onde está o arquivo **`helloworld.hs`** e faça o seguinte:

```
$ ghc --make helloworld
[1 of 1] Compiling Main             ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

Ok! Com alguma sorte, você verá algo como isto e agora você poderá executar seu programa através do comando **`./helloworld`**.

```
$ ./helloworld
hello, world
```

E aqui estamos nós, nosso primeiro programa compilado que imprimiu alguma coisa no terminal. Que extraordinariamente chato!

Vamos analizar o que nós escrevemos. Primeiro, vamos olhar para o tipo da função **`putStrLn`**.

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

Nós podemos ler o tipo de **`putStrLn`** assim: **`putStrLn`** pega uma string e retorna uma **ação de I/O** que tem um resultado do tipo **`()`** (ou seja: uma tupla vazia, também conhecida como unidade). Uma ação de I/O é alguma coisa que, quando executada, irá realizar uma ação com um efeito colateral (que é usualmente ler do dispositivo de entrada ou imprimir algo na tela) e irá também conter algum tipo de valor de retorno dentro dela. Imprimir uma string no terminal realmente não tem nenhum tipo de valor de retorno significativo, então um valor irrelevante **`()`** é usado.

A tupla vazia é um valor **`()`** e também tem um tipo **`()`**.

Então, quando será executada uma ação de I/O? Bem, será onde **`main`** estiver. Uma ação de I/O será executada quando nós dermos a ela um nome **`main`** e então executarmos nosso programa.

Seu programa ser somente uma ação de I/O parece um tipo de limitação. Por isso nós podemos usar a sintaxe *do* para unir várias ações de I/O em uma só. Veja o seguinte exemplo:

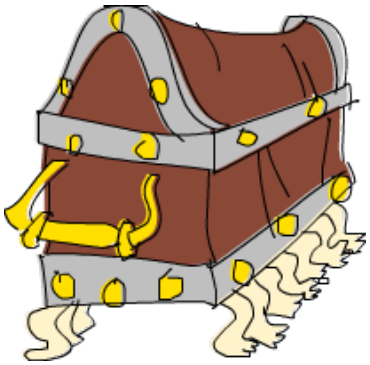
```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

Ah, interessante, nova sintaxe! E isso parece muito mais com um programa imperativo. Se você compilar e tentar executar, ele irá se comportar exatamente como você esperaria. Note que nós escrevemos *do* e então escrevemos uma série de passos, como faríamos em um programa imperativo. Cada um desses passos é uma ação de I/O. Colocando-as juntas com a sintaxe *do*, nós as transformamos em uma única ação de I/O. A ação que nós obtivemos tem um tipo **`IO ()`**, porque esse é o tipo da última ação de I/O dentro dela.

Por causa disso, **`main`** sempre tem uma assinatura de tipo **`main :: IO alguma coisa`**, em que ***`alguma coisa`*** é algum tipo concreto. Por convenção, nós usualmente não especificamos uma declaração de tipo para **`main`**.

Uma coisa interessante que nós não tínhamos visto antes é a terceira linha, que diz **`name <- getLine`**. Parece que uma linha é lida do dispositivo de entrada e armazenada em uma variável chamada **`name`**. É realmente isso? Bem, vamos examinar o tipo de **`getLine`**.

```
ghci> :t getLine
getLine :: IO String
```



Aha, ok. **`getLine`** é uma ação de I/O que contém um resultado do tipo **`String`**. Isso faz sentido, porque ela irá aguardar que o usuário digite algo no terminal e então que algo seja representado como uma string. Então o que acontece com **`name <- getLine`**? Você pode ler esse trecho de codigo assim: **execute a ação de I/O `getLine` e então associe o valor do seu resultado a `name`**. **`getLine`** tem um tipo **`IO String`**, então **`name`** terá um tipo **`String`**. Você pode pensar em uma ação de I/O como sendo uma caixa com pequenos pés que irá sair para o mundo real e fazer algo lá (como grafitar algo em uma parede) e talvez trazer consigo alguma informação. Uma vez que ela tenha trazido as informações a você, a única maneira de abrir a caixa e obter a informação dentro dela é usando o operador **`<-`**. E se nós estamos tirando dados de uma ação de I/O, só podemos fazê-lo quando estamos dentro de outra ação de I/O. É dessa forma que Haskell nitidamente separa as partes pura e impura do nosso código. Desse ponto de vista **`getLine`** é impura porque não há garantia de que o seu resultado será o mesmo quando executada duas vezes. Isso ocorre devido a uma espécie de *contaminação* pelo uso do construtor de tipo **`IO`** e nós somente podemos retirar esses dados da ação de I/O dentro de código de I/O. Além disso, devido ao código de I/O também ser contaminado, qualquer computação que dependa de informação de I/O contaminada terá um resultado contaminado.

Dê uma olhada nesse trecho de código. Ele é válido?

```
nameTag = "Hello, my name is " ++ getLine
```

Se você disse não, vá comer um biscoito. Se disse sim, beba um copo de lava derretida. Brincadeira, não beba! A razão pela qual isso não funciona é que **`++`** requer que ambos os parâmetros sejam listas do mesmo tipo. O parâmetro da esquerda tem o tipo **`String`** (ou **`[Char]`** se quiser), enquanto **`getLine`** tem tipo **`IO String`**. Você não pode concatenar uma string e uma ação de I/O. Primeiro temos que tirar o resultado de dentro da ação de I/O para obter um valor do tipo **`String`** e a única maneira de fazer isso é usar algo como **`name <- getLine`** dentro de alguma outra ação de I/O. Se nós queremos lidar com informação impura, temos que fazê-lo em um ambiente impuro. Então a contaminação da impureza se espalha em volta quase como a praga dos zumbis e é do nosso maior interesse manter os trechos de I/O do nosso código o menor possível.

Toda ação de I/O executada tem um resultado encapsulado dentro dela. Por isso nosso exemplo anterior poderia ter sido escrito assim:

```
main = do
    foo <- putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

Contudo, **`foo`** teria apenas um valor **`()`**, então fazer isso seria um pouco questionável. Observe que nós não associamos o resultado do último **`putStrLn`** a nada. Isso porque em um bloco *do*, **o resultado da última ação não pode ser associado a um nome** como o das duas primeiras foram. Veremos exatamente por que isto acontece um pouco mais tarde, quando nos aventurarmos pelo mundo dos monads. Por enquanto, você pode pensar nisso como se um bloco *do* extraisse automaticamente o resultado da última ação e o associasse ao seu próprio resultado.

Exceto pela última linha, todas as linhas em um bloco *do* que não têm seus resultados associados a nenhum nome podem ser escritas como uma associação. Então **`putStrLn "BLAH"`** pode ser escrito como **`_ <- putStrLn "BLAH"`**. Mas isso é inútil, então nós omitimos o **`<-`** para ações de I/O que não contêm um resultado importante, como **`putStrLn`** ***`alguma coisa`***.

Iniciantes algumas vezes pensam que escrevendo

```
name = getLine
```

irá ler do dispositivo de entrada e então associar o valor a **`name`**. Bem, não irá, tudo que isso faz é dar à ação de I/O **`getLine`** um nome diferente chamado **`name`**. Lembre-se, para retirar o valor de uma ação de I/O, você tem que executá-la dentro de outra ação de I/O associando seu resultado a um nome com **`<-`**.

Ações de I/O serão executadas somente se for dado a elas um nome **`main`** ou quando elas estiverem dentro de uma ação de I/O maior que nós fizermos com um bloco *do*. Nós podemos também usar um bloco *do* para unir algumas ações de I/O e então podemos usar a ação de I/O resultante em outro bloco *do* e assim por diante. De qualquer forma, elas serão executadas somente se eventualmente cairem em **`main`**.

Ah, certo, existe ainda mais um caso em que ações de I/O serão executadas. Quando nós digitamos uma ação de I/O no GHCI e pressionamos enter, ela é executada.

```
ghci> putStrLn "HEEY"
HEEY
```

Mesmo quando nós digitamos um número ou a invocação de uma função no GHCI e pressionamos enter, ele irá avaliar (se for necessário) e invocar **`show`** para o resultado da avaliação e então irá imprimir essa string no terminal usando **`putStrLn`** implicitamente.

Lembra das associações *let*? Se não se lembra, refresque sua memória lendo essa seção. Elas são construções da forma **`let` *`associações`* `in` *`expressão`***, em que ***`associações`*** são nomes que serão utilizados em uma expressão e ***`expressão`*** é a expressão a ser avaliada que utilizará esses nomes. Também dissemos que em list comprehensions, a parte *in* não é necessária. Bem, você pode usá-las em blocos *do* quase da mesma forma que você as usa em list comprehensions. Dê uma olhada nisso:

```
import Data.Char

main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn $ "hey " ++ bigFirstName ++ " " ++ bigLastName ++ ", how are you?"
```

Viu como as ações de I/O no bloco *do* estão alinhadas? Percebeu também como o *let* está alinhado com as ações de I/O e os nomes do *let* estão alinhados uns com os outros? Esta é uma boa prática, porque indentação é importante em Haskell. Agora, nós escrevemos **`map toUpper firstName`**, o que transforma algo como **`"John"`** em uma string mais legal, como **`"JOHN"`**. Nós associamos strings em maiúsculas a um nome e depois as utilizamos em outra string que imprimimos no terminal.

Você pode estar se perguntando quando usar **`<-`** e quando usar associações *let*? Bem, lembre-se, **`<-`** é usado (por enquanto) para executar ações de I/O e associar seus resultados a nomes. **`map toUpper firstName`**, contudo, não é uma ação de I/O. É uma expressão pura em Haskell. Então, use **`<-`** quando você quiser associar resultados de ações de I/O a nomes e associações *let* para associar expressões puras a nomes. Escrevendo algo como **`let firstName = getLine`**, só teríamos dado a ação de I/O **`getLine`** um nome diferente e ainda teríamos que utilizar **`<-`** para executá-la.

Agora iremos fazer um programa que continuamente lê uma linha e imprime a mesma linha com as palavras invertidas. A execução do programa irá terminar quando nós digitarmos uma linha em branco. Este é o programa:

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

Para se ter uma noção do que isso faz, você pode executá-lo antes de analizarmos o código.

Dica de profissional: Para rodar um programa você pode compilá-lo e rodar o arquivo executável escrevendo **`ghc --make helloworld`** e então **`./helloworld`** ou você pode usar o comando **`runhaskell`** dessa forma: **`runhaskell helloworld.hs`** e o seu programa será executado dinamicamente.

Primeiro, vamos dar uma olhada na função **`reverseWords`**. Ela é apenas uma função normal que pega uma string como **`"hey there man"`** e então invoca **`words`** com ela para gerar uma lista de palavras como **`["hey","there","man"]`**. Então mapeamos a lista com **`reverse`**, obtendo **`["yeh","ereht","nam"]`**, juntamos as palavras de volta em uma string usando **`unwords`** e o resultado final é **`"yeh ereht nam"`**. Veja que usamos composição de funções aqui. Sem composição de funções, teríamos que escrever algo como **`reverseWords st = unwords (map reverse (words st))`**.

E quanto ao **`main`**? Primeiro, obtemos uma linha digitada no terminal usando **`getLine`** e chamamos essa linha de **`line`**. E agora, temos uma expressão condicional. Lembre-se que em Haskell, todo *if* deve ter um *else* porque toda expressão tem que ter algum tipo de valor de retorno. Escrevemos um *if* de forma que quando a condição for verdadeira (em nosso caso, a linha que digitamos está em branco), executamos uma ação de I/O e quando não for, a ação de I/O no *else* é executada. Isto porque em um bloco *do* de I/O, *if*s tem que ser da forma **`if` *`condição`* `then` *`ação de I/O`* `else` *`ação de I/O`***.

Primeiro vamos dar uma olhada no que acontece no bloco *else*. Porque temos que ter exatamente uma ação de I/O depois do *else*, nós usamos um bloco *do* para unir duas ações de I/O em uma só. Você também poderia escrever esse trecho como:

```
else (do
    putStrLn $ reverseWords line
    main)
```

Isso deixa mais explícito que o bloco ***do*** pode ser visto como uma ação de I/O, mas é mais feio. De qualquer forma, dentro do bloco ***do***, nós chamamos **`reverseWords`** passando como parâmetro a linha que obtivemos com **`getLine`** e então imprimimos o resultado no terminal. Depois disso, nós executamos **`main`**. Ela é chamada recursivamente sem problemas, porque a própria **`main`** é uma ação de I/O. De certo modo, nós voltamos para o início do programa.

Agora, o que acontece quando **`null line`** é verdadeiro? Nesse caso, o que está depois do *then* é executado. Se procurarmos, veremos que isso é **`then return ()`**. Se você usa linguagens imperativas como C, Java ou Python, você provavelmente está pensando que sabe o que esse **`return`** faz e talvez você já tenha pulado esse parágrafo realmente longo. Bem, essa é a questão: o **`return`** **em Haskell não tem nada a ver com** **`return`** **na maioria das outras linguagens!**. Ele tem o mesmo nome, o que confunde muitas pessoas, mas na verdade ele é um pouco diferente. Em linguagens imperativas, **`return`** normalmente encerra a execução de um método ou subrotina e faz ele devolver algum tipo de valor a quem o chamou. Em Haskell (especificamente em ações de I/O), ele cria uma ação de I/O com um valor puro. Se você pensar na analogia feita antes sobre a caixa, ele pega o valor e coloca dentro de uma caixa. A ação de I/O resultante na verdade não faz qualquer coisa, ela apenas tem aquele valor encapsulado como seu resultado. Então, em um contexto de I/O, **`return "haha"`** terá um tipo **`IO String`**. Qual é o sentido de colocar um valor puro dentro de uma ação de I/O que faz coisa nenhuma? Por que contaminar nosso programa com mais **`I/O`** do

que o necessário? Bem, precisamos de alguma ação de I/O para executar no caso de uma entrada em branco. Por isso nós simplesmente forjamos uma ação de I/O que faz nada, escrevendo **`return ()`**.

Usar **`return`** não faz com que o bloco de I/O ***do*** encerre sua execução ou algo parecido. Por exemplo, esse programa irá muito felizmente executar cada linha até a última:

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn line
```

Tudo o que esses **`return`**s fazem é criar ações de I/O que na verdade fazem nada além de ter um resultado encapsulado e esses resultados são descartados porque eles não são associados a um nome. Podemos usar **`return`** combinado com **`<-`** para associar coisas a nomes.

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

Como você pode ver, **`return`** é como um de oposto **`<-`**. Enquanto **`return`** pega um valor e o coloca em uma caixa, **`<-`** pega uma caixa (e a executa) e retira o valor de dentro dela, associando a um nome. Mas isso é redundante, especialmente porque você pode usar associações ***let*** em blocos ***do*** para associar valores a nomes, dessa forma:

```
main = do
    let a = "hell"
        b = "yeah"
    putStrLn $ a ++ " " ++ b
```

Quando estamos lidando com blocos *do* de I/O, nós usamos **`return`** principalmente porque precisamos criar uma ação de I/O que faz coisa nenhuma porque não queremos que essa ação de I/O criada a partir de um bloco ***do*** tenha o valor resultante da sua última ação, mas queremos que ela tenha um valor resultante diferente, então usamos **`return`** para criar uma ação de I/O que sempre encapsula o valor desejado por nós e a colocamos no final.

> Um bloco ***do*** também pode ter somente uma ação de I/O. Nesse caso, isso é o mesmo que simplesmente escrever a ação de I/O. Algumas pessoas prefeririam escrever **`then do return ()`** nesse caso porque o *else* também tem um ***do***.

Antes de passarmos aos arquivos, vamos dar uma olhada em algumas funções que são úteis quando lidamos com I/O.

`putStr` é bem parecida com **`putStrLn`** porque ela recebe uma string como parâmetro e retorna uma ação de I/O que imprimirá essa string no terminal, porém **`putStr`** não quebra a linha depois de imprimir a string enquanto **`putStrLn`** quebra.

```
main = do   putStr "Hey, "
```

```
            putStr "I'm "
            putStrLn "Andy!"
```

```
$ runhaskell putstr_test.hs
Hey, I'm Andy!
```

Sua assinatura de tipo é **`putStr :: String -> IO ()`**, então o resultado encapsulado na ação de I/O resultante é a tupla vazia. Um valor forjado, então não faz sentido associá-lo a um nome.

`putChar` pega um caractere e retorna uma ação de I/O que o imprimirá no terminal.

```
main = do   putChar 't'
            putChar 'e'
            putChar 'h'
```

```
$ runhaskell putchar_test.hs
teh
```

`putStr` na verdade é definida recursivamente utilizando **`putChar`**. A condição de parada de **`putStr`** é a string vazia, então se estamos imprimindo uma string vazia, somente retornamos uma ação de I/O que não faz qualquer coisa usando **`return ()`**. Se a string não é vazia, então imprimimos o seu primeiro caractere usando **`putChar`** e então imprimimos o restante usando **`putStr`**.

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

Observe como podemos usar recursão em I/O, exatamente como podemos usar em código puro. Exatamente como em código puro, definimos a condição de parada e então pensamos no que de fato é o resultado. Essa é uma ação que primeiro imprime o primeiro caractere e então imprime o restante da string.

`print` pega um valor de qualquer tipo que seja uma instância de **`Show`** (o que quer dizer que sabemos como representá-lo como uma string), invoca **`show`** com esse valor para convertê-lo em string e então imprime a saída no terminal. Basicamente, ela é apenas **`putStrLn . show`**. Primeiro ela roda **`show`** em um valor e então passa o resultado para **`putStrLn`**, que retorna uma ação de I/O que imprimirá nossa entrada.

```
main = do   print True
            print 2
            print "haha"
            print 3.2
            print [3,4,3]
```

```
$ runhaskell print_test.hs
True
2
"haha"
3.2
[3,4,3]
```

Como você pode ver, é uma função bem útil. Lembra que nós falamos que ações de I/O são executadas somente quando elas aparecem em **`main`** ou quando tentamos avaliá-las no prompt do GHCI? Quando digitamos um valor (como `3` ou **`[1,2,3]`**) e pressionamos enter, o GHCI na verdade usa **`print`** nesse valor para mostrá-lo no terminal!

```
ghci> 3
3
ghci> print 3
3
ghci> map (++"!") ["hey","ho","woo"]
["hey!","ho!","woo!"]
ghci> print (map (++"!") ["hey","ho","woo"])
["hey!","ho!","woo!"]
```

Quando queremos imprimir strings, normalmente usamos **`putStrLn`** porque não queremos as aspas circundando-as, mas para imprimir valores de outros tipos no terminal, **`print`** é mais usada.

`getChar` é uma ação de I/O que lê um caractere do dispositivo de entrada. Sendo assim, sua assinatura de tipo é **`getChar :: IO Char`**, porque o resultado contido na ação de I/O é um **`Char`**. Observe que graças à bufferização, a leitura de caracteres na verdade não acontecerá até que o usuário pressione enter.

```
main = do
    c <- getChar
    if c /= ' '
        then do
            putChar c
            main
        else return ()
```

Esse programa aparentemente deveria ler um caractere e então checar se é um espaço em branco. Se é, interrompe a execução e se não, imprime-o no terminal e então faz a mesma coisa novamente. Bem, de certa forma ele faz isso, mas não do modo que você esperaria. Dê uma olhada nisso:

```
$ runhaskell getchar_test.hs
hello sir
hello
```

A segunda linha é a entrada. Nós digitamos **`hello sir`** e pressionamos enter. Graças à bufferização, a execução do programa irá iniciar somente depois que tivermos pressionado enter e não depois que qualquer caractere for digitado. Uma vez que tivermos pressionado enter, ele atuará sobre tudo o que tivermos digitado até então. Experimente brincar com esse programa para ter uma noção sobre isso!

A função `when` faz parte de **`Control.Monad`** (então, escreva **`import Control.Monad`** para acessá-la). Ela é interessante porque em um bloco ***do*** ela se parece com uma declaração de fluxo de controle, mas na verdade é uma função comum. Ela pega um valor booleano e uma ação de I/O , se o valor booleano for **`True`**, retorna a mesma ação de I/O que nós passamos. Todavia, se for **`False`**, ela retorna a ação **`return ()`**, que é uma ação de I/O que faz nada. Eis um modo como podemos reescrever o trecho de código anterior, onde demonstramos **`getChar`**, usando **`when`**:

```
import Control.Monad

main = do
    c <- getChar
```

```
    when (c /= ' ') $ do
        putChar c
        main
```

Então, como você pode perceber, ela é útil para encapsular o padrão **`if alguma coisa then do alguma ação de I/O else return ()`**.

`sequence` pega uma lista de ações de I/O e retorna uma ação de I/O que irá executar as ações passadas uma após a outra. O resultado contido nessa ação de I/O será uma lista dos resultados de todas as ações de I/O que foram executadas. Sua assinatura de tipo é **`sequence :: [IO a] -> IO [a]`**. Fazer isso:

```
main = do
    a <- getLine
    b <- getLine
    c <- getLine
    print [a,b,c]
```

É exatamente o mesmo que fazer isso:

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

Então **`sequence [getLine, getLine, getLine]`** cria uma ação de I/O que irá executar **`getLine`** três vezes. Se associarmos essa ação a um nome, o resultado é uma lista de todos os resultados, no nosso caso, uma lista de três coisas que o usuário digitou no terminal.

Um padrão comum usando **`sequence`** é quando usamos funções como **`print`** ou **`putStrLn`** para mapear listas. Escrevendo **`map print [1,2,3,4]`** não iremos criar uma ação de I/O. Isso irá criar uma lista de ações de I/O, porque isso é como escrever **`[print 1, print 2, print 3, print 4]`**. Se quisermos transformar essa lista de ações de I/O em uma única ação de I/O, temos que sequenciá-la.

```
ghci> sequence (map print [1,2,3,4,5])
1
2
3
4
5
[(),(),(),(),()]
```

O que é esse **`[(),(),(),(),()]`** no final? Então, quando avaliamos uma ação de I/O no GHCI, ela é executada e o seu resultado é impresso, a menos que o resultado seja **`()`**, caso em que nada é impresso. Daí porque avaliar **`putStrLn "hehe"`** no GHCI só imprime **`hehe`** (porque o resultado contido em **`putStrLn "hehe"`** é **`()`**). Mas quando escrevemos **`getLine`** no GHCI, o resultado dessa ação de I/O é impresso, porque **`getLine`** tem o tipo **`IO String`**.

Devido ao uso de uma função que retorne uma ação de I/O para mapear uma lista e depois sequenciá-la ser algo tão comum, as funções `mapM` e `mapM_` foram introduzidas. **`mapM`** pega uma função e uma lista, mapeia a lista com a função e então sequencia o mapeamento. **`mapM_`** faz o mesmo, mas descarta o resultado no final. Geralmente usamos **`mapM_`** quando não nos importamos com os resultados que nossas ações de I/O sequenciadas têm.

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

`forever` pega uma ação de I/O e retorna uma ação de I/O que somente repete para sempre a ação de I/O passada. Ela faz parte de **`Control.Monad`**. Esse pequeno programa irá aguardar indefinidamente alguma entrada do usuário e mandar de volta a ele, EM MAIÚSCULAS:

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

`forM` (localizada em **`Control.Monad`**) é parecida com **`mapM`**, porém tem os parâmetros invertidos. O primeiro parâmetro é a lista e o segundo é a função que será usada para mapear a lista, que será então sequenciada. Por que isto é útil? Bem, com algum uso criativo de lambdas e notação *do*, podemos fazer coisas como essa:

```
import Control.Monad

main = do
    colors <- forM [1,2,3,4] (\a -> do
        putStrLn $ "Which color do you associate with the number " ++ show a ++ "?"
        color <- getLine
        return color)
    putStrLn "The colors that you associate with 1, 2, 3 and 4 are: "
    mapM putStrLn colors
```

O **`(\a -> do ... )`** é uma função que pega um número e retorna uma ação de I/O. Temos que envolvê-la com parênteses, senão o lamba irá pensar que as últimas duas ações de I/O pertencem a ele. Observe que escrevemos **`return color`** no bloco ***do*** mais interno. Fizemos isso para que a ação de I/O definida pelo bloco ***do*** tenha nossas cores contidas nele como resultado. Na verdade não temos que fazer isso, porque **`getLine`** já as contém como resultado. Escrever **`color <- getLine`** e depois **`return color`** é somente desempacotar o resultado de **`getLine`** e em seguida empacotá-lo novamente, então é o mesmo que escrever somente **`getLine`**. O **`forM`** (chamado com seus dois parâmetros) produz uma ação de I/O, cujo resultado nós associamos a **`colors`**. **`colors`** é apenas uma lista de strings comum. No final, imprimimos todas essas cores escrevendo **`mapM putStrLn colors`**.

Você pode pensar que **`forM`** é: faça uma ação de I/O para cada elemento nessa lista. O que cada ação de I/O fará pode depender do elemento que é usado para criar a ação. Finalmente, execute essas ações e associe seus resultados a alguma coisa. Nós não temos que associá-las, também podemos simplemente descartá-las.

```
$ runhaskell form_test.hs
Which color do you associate with the number 1?
white
Which color do you associate with the number 2?
blue
```

```
Which color do you associate with the number 3?
red
Which color do you associate with the number 4?
orange
The colors that you associate with 1, 2, 3 and 4 are:
white
blue
red
orange
```

Poderíamos ter feito isso sem **`forM`**, porém com **`forM`** fica mais legível. Geralmente escrevemos **`forM`** quando queremos mapear e sequenciar algumas ações que definimos lá no local utilizando a notação *do*. Seguindo esse raciocínio, poderíamos ter substituido a última linha por **`forM colors putStrLn`**.

Nesta seção, aprendemos o básico sobre ***input*** e ***output*** (I/O) em Haskell. Também descobrimos o que são ações de I/O, como elas nos habilitam a realizar operações de *input* e *output* e quando elas são realmente executadas. Reiterando, ações de I/O são valores muito parecidos com qualquer outro valor em Haskell. Podemos passá-los como parâmetros para funções e funções podem retornar ações de I/O como resultado. A diferença é que se elas estiverem na função **`main`** (ou forem o resultado de uma linha no GHCI), elas são executadas. O que significa que elas escrevem coisas na sua tela ou tocam Yakety Sax através das suas caixas de som. Cada ação de I/O também pode encapsular um resultado com o qual ela lhe diz o que ela obteve do mundo real.

Não pense em uma função como a **`putStrLn`** como sendo uma função que pega uma string e a imprime na tela. Pense nela como uma função que pega uma string e retorna uma ação de I/O. Essa ação de I/O irá, quando executada, imprimir poesias bonitas no seu terminal.

Arquivos e streams

`getChar` é uma ação de I/O que lê um único caractere do terminal. **`getLine`** é uma ação de I/O que lê uma linha do terminal. As duas são bem diretas e a maioria das linguagens de programação tem algumas funções ou declarações que são similares a elas. Mas agora, vamos conhecer `getContents`. **`getContents`** é uma ação de I/O que lê qualquer coisa do dispositivo de entrada padrão até encontrar um caractere de fim de arquivo. Seu tipo é **`getContents :: IO String`**. O que é legal em **`getContents`** é que ela faz I/O "preguiçosa". Quando escrevemos **`foo <- getContents`**, ela não lê toda a entrada de uma vez, armazena na memória e associa a **`foo`**. Não, ela é preguiçosa! Ela dirá: *"Sim sim, eu vou ler a entrada do terminal mais tarde a medida em que avançamos, quando você realmente precisar dela!"*.

`getContents` é realmente útil quando estamos direcionando a saída de um programa para a entrada do nosso programa. Caso você não saiba como piping funciona em sistemas baseados em unix, aqui vai uma explicação rápida. Vamos criar um arquivo de texto que contém o seguinte pequeno haicai:

```
Sou um pequeno bule
O que há com essa comida de avião, hein?
Ela é tão pequena, sem gosto
```

Sim, o haicai é uma porcaria, e daí? Se alguém souber de algum bom tutorial de haicai, compartilhe comigo.

Agora, relembre o pequeno programa que escrevemos quando estávamos apresentando a função **`forever`**. Ele aguardava que o usuário digitasse uma linha, retornava essa linha para ele em MAIÚSCULAS e depois fazia tudo de novo, indefinidamente. Só para você não ter que voltar tudo de novo, aqui está ele novamente:

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

Iremos salvar esse programa como **`capslocker.hs`** ou algum outro nome e compilá-lo. Depois, vamos usar um pipe unix para passar nosso arquivo de texto diretamente para o nosso pequeno programa. Vamos ter o auxílio do programa GNU *cat*, que imprime um arquivo que lhe é dado como argumento. Dê uma olhada, é excelente!

```
$ ghc --make capslocker
[1 of 1] Compiling Main             ( capslocker.hs, capslocker.o )
Linking capslocker ...
$ cat haiku.txt
Sou um pequeno bule
O que há com essa comida de avião, hein?
Ela é tão pequena, sem gosto
$ cat haiku.txt | ./capslocker
SOU UM PEQUENO BULE
O QUE HÁ COM ESSA COMIDA DE AVIÃO, HEIN?
ELA É TÃO PEQUENA, SEM GOSTO
capslocker <stdin>: hGetLine: end of file
```

Como você pode observar, o piping da saída de um programa (no nosso caso do *cat*) para a entrada de outro (*capslocker*) é feito com o caractere `|`. O que fizemos é equivalente a simplesmete executar *capslocker*, digitando nosso haicai no terminal e depois fornecendo um caractere de fim de arquivo (isso normalmente é feito pressionando Ctrl-D). Isso é como executar *cat haiku.txt* e dizer: “Espera, não imprima isso no terminal, passe para *capslocker* ao invés disso!”.

Então o que essencialmente estamos fazendo com esse **`forever`** é pegar a entrada e transformá-la em alguma saída. Por isso nós podemos usar **`getContents`** para tornar nosso programa ainda melhor e mais curto:

```
import Data.Char

main = do
    contents <- getContents
    putStr (map toUpper contents)
```

Nós executamos a ação de I/O **`getContents`** e chamamos a string que ela produz de **`contents`**. Então, usamos **`toUpper`** para mapear essa string e imprimí-la no terminal. Tenha em mente que devido às strings serem basicamente listas, que são preguiçosas, e **`getContents`** ser I/O preguiçosa, o programa não tentará ler todo o conteúdo de uma

vez e guardá-lo na memória antes de imprimir a versão em maiúsculas. Ao invés disso, ele imprimirá a versão em maiúsculas à medida que faz a leitura, porque ele somente lerá uma linha da entrada quando realmente precisar.

```
$ cat haiku.txt | ./capslocker
SOU UM PEQUENO BULE
O QUE HÁ COM ESSA COMIDA DE AVIÃO, HEIN?
ELA É TÃO PEQUENA, SEM GOSTO
```

Legal, isso funciona. O que acontece se simplesmente executarmos *capslocker* e tentarmos digitar as linhas no terminal?

```
$ ./capslocker
hey ho
HEY HO
lets go
LETS GO
```

Nós encerramos pressionando Ctrl-D. Muito bom! Como você pode perceber, ele imprime nossa saída em maiúsculas linha por linha. Quando o resultado de **`getContents`** é associado a **`contents`**, ele não é representado na memória como uma string real, mas como uma promessa de que irá produzir aquela string eventualmente. Quando usamos **`toUpper`** para mapear **`contents`**, isso também é uma promessa de usar essa função para mapear um possível conteúdo dessa string. E finalmente quando **`putStr`** acontece, ela diz para a promessa anterior: *"Ei, eu preciso de uma linha em maiúsculas!"*. Ela não tem linhas ainda, então diz para **`contents`**: *"Ei, que tal agora pegar uma linha do terminal?"*. É aí que **`getContents`** realmente lê do terminal e fornece uma linha para o código que pediu a ela para produzir algo real. Esse código então usa **`toUpper`** para mapear a linha e passa ela para **`putStr`**, que a imprime. Depois, **`putStr`** diz: *"Ei, eu preciso da próxima linha, vamos!"* e isto se repete até que não haja mais entradas, o que é verificado pela presença de um caractere de final de arquivo.

Vamos criar um programa que recebe alguma entrada e imprime somente as linhas que têm menos de 15 caracteres. Observe:

```
main = do
    contents <- getContents
    putStr (shortLinesOnly contents)

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 15) allLines
        result = unlines shortLines
    in  result
```

Fizemos a parte de I/O do nosso programa o menor possível. Devido ao fato de que o nosso programa deve ler uma entrada e imprimir alguma saída baseada nessa entrada, podemos implementá-lo lendo o conteúdo da entrada, aplicando uma função sobre eles e depois imprimindo o que a função retornou.

A função **`shortLinesOnly`** funciona dessa forma: ela recebe uma string, como **`"curta\nlooooooooooooooonga\ncurta denovo"`**. Essa string tem três linhas, duas delas são curtas e a do meio é longa. Ela aplica a função **`lines`** nessa string, convertendo-a em **`["curta", "looooooooooooooonga", "curta denovo"]`**, que depois associamos ao nome **`allLines`**. Essa

lista de string é então filtrada de modo que somente aquelas linhas que têm menos de 15 caracteres permanecem na lista, produzindo **`["curta", "curta denovo"]`**. E finalmente, **`unlines`** junta essa lista em uma única string com quebras de linha, obtendo **`"curta\ncurta denovo"`**. Vamos testá-la.

```
sou curta
eu sou
sou uma linha looooooooooonga!!!
sim sou longa e dai hahahaha!!!!!!
linha curta
loooooooooooooooooooooooooonga
curta
```

```
$ ghc --make shortlinesonly
[1 of 1] Compiling Main             ( shortlinesonly.hs, shortlinesonly.o )
Linking shortlinesonly ...
$ cat shortlines.txt | ./shortlinesonly
sou curta
eu sou
curta
```

Fazemos um pipe do conteúdo de *shortlines.txt* para a entrada de *shortlinesonly* e como saída, obtemos somente as linhas curtas.

Esse padrão de obter alguma string do dispositivo de entrada, transformá-la com uma função e depois imprimí-la é tão comum que existe uma função, chamada `interact`, que torna sua implementação ainda mais fácil. **`interact`** recebe uma função do tipo **`String -> String`** como parâmetro e retorna uma ação de I/O que irá receber alguma entrada, aplicar a função nela e depois imprimir o resultado da função. Vamos modificar o nosso programa para usá-la.

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

Só para mostar como esse resultado pode ser alcançado com muito menos código (mesmo achando que será menos legível) e para demonstrar nossa habilidade com composição de funções, vamos retrabalhá-lo mais um pouco.

```
main = interact $ unlines . filter ((<10) . length) . lines
```

Wow, nós realmente o reduzimos a uma única linha, o que é muito legal!

Até agora, nós trabalhamos com I/O imprimindo coisas no terminal e lendo dele. Mas e quanto a ler e escrever em arquivos? Bem, de certo modo, nós já estamos fazendo isso. Um modo de visualizar a leitura do terminal é imaginar que é como ler de um arquivo (um tanto especial). O mesmo vale para escrever no terminal, que é como escrever em um arquivo. Chamamos esses dois arquivos de **`stdout`** e **`stdin`**, que significam *saída padrão* e *entrada padrão*, respectivamente. Mantendo isso em mente, veremos que escrever e ler arquivos é muito parecido com escrever na saída padrão e ler da entrada padrão.

Iremos iniciar com um programa realmente simples que abre um arquivo chamado *girlfriend.txt*, que contém um verso do hit nº 1 de Avril Lavigne, *Girlfriend*, e somente o imprime no terminal. Aque está *girlfriend.txt*:

```
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

E aqui está o nosso programa:

```
import System.IO

main = do
    handle <- openFile "girlfriend.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

Executando-o, obtemos o resultado esperado:

```
$ runhaskell girlfriend.hs
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

Vamos estudá-lo linha a linha. A primeira linha são só quatro exclamações, para atrair sua atenção. Na segunda linha, Avril nos diz que ela não gosta do nosso parceiro romântico atual. A terceira linha serve para enfatizar essa desaprovação, enquando a quarta linha sugere que deveríamos procurar uma nova namorada.

Vamos estudar também o programa linha por linha! Nosso programa são várias ações de I/O unidas por um bloco *do*. Na primeira linha do bloco *do*, nós observamos uma nova função chamada `openFile`. Esta é sua assinatura de tipo: **`openFile :: FilePath -> IOMode -> IO Handle`**. Se você lê-la em voz alta, ela diz: **`openFile`** recebe um caminho de arquivo e um **`IOMode`** e retorna uma ação de I/O que abrirá um arquivo e terá o handle associado a esse arquivo encapsulado como seu resultado.

`FilePath` é só um tipo sinônimo para **`String`**, simplesmente definido como:

```
type FilePath = String
```

`IOMode` é um tipo que definimos da seguinte forma:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Exatamente como nosso tipo que representa os sete possíveis valores para os dias da semana, esse tipo é uma enumeração que representa o que nós queremos fazer com nossos arquivos abertos. Muito simples. Só repare que esse tipo é **`IOMode`** e não **`IO Mode`**. **`IO Mode`** seria o tipo de uma ação de I/O que tem um valor de algum tipo **`Mode`** como seu resultado, mas **`IOMode`** é somente uma simples enumeração.

Por fim, é retornada uma ação de I/O que abrirá o arquivo especificado no modo especificado. Se associarmos essa ação a algo obtemos um **`Handle`**. Um valor do tipo **`Handle`** representa onde o nosso arquivo está. Usaremos esse handle para saber de qual arquivo ler. Seria estúpido ler um arquivo e não associá-lo a um handle porque não seríamos capazes de fazer qualquer coisa com o arquivo. Então, no nosso caso, associamos o handle a **`handle`**.

Na próxima linha, vemos uma função chamada `hGetContents`. Ela recebe um **`Handle`**, de forma que ela sabe de qual arquivo obter o conteúdo e retorna uma **`IO String`** — uma ação de I/O que guarda como seu resultado o conteúdo do arquivo. Essa função é bem parecida com **`getContents`**. A única diferença é que **`getContents`** lerá automaticamente da entrada padrão (que é o terminal), enquanto **`hGetContents`** tem um handle de arquivo que lhe diz de qual arquivo ler. Em todos os outros aspectos, elas funcionam igualmente. E exatamente como **`getContents`**, **`hGetContents`** não tentará ler um arquivo de uma vez e guardá-lo na memória, mas ele lerá quando necessário. Isso é bem interessante porque podemos tratar **`contents`** como o conteúdo inteiro do arquivo, mas na verdade ele não está carregado na memória. Então se for um arquivo bem enorme, escrever **`hGetContents`** não iria esgotar sua memória, mas iria ler somente o que precisasse, quando necessário.

Veja a diferença entre o handle usado para identificar um arquivo e o conteúdo de um arquivo, associados no nosso programa a **`handle`** e **`contents`**, respectivamente. O handle é somente algo pelo qual nós sabemos o que é o nosso arquivo. Se você pensar em todo seu sistema de arquivos como sendo um livro bem grande e cada arquivo é um capítulo no livro, o handle é um marcador de livros que mostra onde você está lendo atualmente (ou escrevendo) um capítulo, enquanto que o conteúdo é o capítulo real.

Com **`putStr contents`** nós só imprimimos o conteúdo na saída padrão e depois escrevemos `hClose`, que recebe um handle e retorna uma ação de I/O que fecha o arquivo. Você mesmo tem que fechar o arquivo após abrí-lo com **`openFile`**!

Outra maneira de o que acabamos de fazer é usar a função `withFile`, que tem a assinatura de tipo **`withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a`**. Ela recebe um caminho para um arquivo, um **`IOMode`** e depois recebe uma função que pega um handle e retorna alguma ação de I/O. O que ela retorna é uma ação de I/O que abre aquele arquivo, faz algo que queremos com ele e depois o fecha. O resultado encapsulado na ação de I/O final que é retornada é o mesmo resultado que a ação de I/O que nós passamos retornaria. Isso pode soar um pouco complicado, mas é muito simples, especialmente com lambdas. Aqui está o nosso exemplo anterior reescrito usando **`withFile`**:

```
import System.IO

main = do
    withFile "girlfriend.txt" ReadMode (\handle -> do
        contents <- hGetContents handle
        putStr contents)
```

Como você vê, é muito semelhante com o trecho de código anterior. **`(\handle -> ... )`** é a função que recebe um handle e retorna uma ação de I/O e ela geralmente é escrita assim, com um lambda. A razão para ela ter que receber uma função que retorna uma ação de I/O ao invés de simplesmente receber uma ação de I/O para executar e depois

fechar o arquivo é que a ação de I/O que nós passamos a ela não saberia sobre qual arquivo operar. Desse modo, **`withFile`** abre o arquivo e então passa o handle para a função que demos a ela. Depois pega de volta uma ação de I/O daquela função e cria uma ação de I/O igual, só que fecha o arquivo depois. Eis um modo como podemos fazer a nossa própria função **`withFile`**:

```
withFile' :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile' path mode f = do
    handle <- openFile path mode
    result <- f handle
    hClose handle
    return result
```

Sabemos que o resultado será uma ação de I/O, então podemos começar com um *do*. Primeiro abrimos o arquivo e obtemos o handle dele. Depois, passamos **`handle`** para a nossa função para obter uma ação de I/O que faz todo o trabalho. Associamos essa ação a **`result`**, fechamos o handle e então escrevemos **`return result`**. Ao usar **`return`** para retornar o valor encapsulado na ação de I/O que obtivemos de **`f`**, fizemos com que nossa ação de I/O encapsule o mesmo resultado que a que nós obtivemos de **`f handle`**. Então se **`f handle`** retorna uma ação que leria algumas linhas da entrada padrão, as imprimiria em um arquivo e teria encapsulado como seu resultado o número de linhas que foram lidas, se usássemos ela com **`withFile'`**, a ação de I/O resultante também teria como seu resultado o número de linhas lidas.

Exatamente como temos **`hGetContents`** que funciona como **`getContents`** mas para um arquivo específico, também temos `hGetLine`, `hPutStr`, `hPutStrLn`, `hGetChar`, etc. Elas funcionam exatamente como suas equivalentes sem o *h*, porém recebem um handle como parâmetro e operam naquele arquivo específico ao invés de operar sobre a entrada padrão ou saída padrão. Exemplo: **`putStrLn`** é uma função que recebe uma string e retorna uma ação de I/O que imprimirá essa string no terminal e criará uma nova linha depois. **`hPutStrLn`** recebe um handle e uma string e retorna uma ação de I/O que escreverá essa string no arquivo associado ao handle e criará uma nova linha depois dessa string. Do mesmo modo, **`hGetLine`** recebe um handle e retorna uma ação de I/O que lê uma linha do seu arquivo.

Abrir arquivos e depois tratar seu conteúdo como strings é tão comum que nós temos essas três pequenas e agradáveis funções que tornam o nosso trabalho ainda mais fácil:

`readFile` tem a assinatura de tipo **`readFile :: FilePath -> IO String`**. Lembre-se, **`FilePath`** é só um nome fantasia para **`String`**. **`readFile`** recebe um caminho para um arquivo e retorna uma ação de I/O que irá ler aquele arquivo (preguiçosamente, é claro) e associar seu conteúdo a algum nome como uma string. Isso é geralmente mais conveniente do que executar **`openFile`**, associá-lo a um handle e então executar **`hGetContents`**. Eis um modo como poderíamos ter escrito nosso exemplo anterior com **`readFile`**:

```
import System.IO

main = do
    contents <- readFile "girlfriend.txt"
    putStr contents
```

Devido a não obtermos um handle com o qual identificaríamos nosso arquivo, não temos como fechá-lo manualmente, então Haskell faz isso por nós quando usamos **`readFile`**.

`writeFile` tem o tipo **`writeFile :: FilePath -> String -> IO ()`**. Ela recebe um caminho para um arquivo e uma string para escrever nesse arquivo e retorna uma ação de I/O que fará a escrita. Se o arquivo já existir, ele terá seu conteúdo apagado antes de ser escrita a string nele. Aqui está como transformar *girlfriend.txt* em uma versão em MAIÚSCULAS e escrevê-la em *girlfriendcaps.txt*:

```
import System.IO
import Data.Char

main = do
    contents <- readFile "girlfriend.txt"
    writeFile "girlfriendcaps.txt" (map toUpper contents)
```

```
$ runhaskell girlfriendtocaps.hs
$ cat girlfriendcaps.txt
HEY! HEY! YOU! YOU!
I DON'T LIKE YOUR GIRLFRIEND!
NO WAY! NO WAY!
I THINK YOU NEED A NEW ONE!
```

`appendFile` tem uma assinatura de tipo que é exatamente igual à de **`writeFile`**, mas **`appendFile`** não apaga o conteúdo do arquivo se ele já existir, mas acrescenta coisas a ele.

Digamos que nós temos um arquivo *todo.txt* que em cada linha tem uma tarefa que nós temos que fazer. Agora vamos criar um programa que pega uma linha da entrada padrão e adiciona ela à nossa lista de tarefas.

```
import System.IO

main = do
    todoItem <- getLine
    appendFile "todo.txt" (todoItem ++ "\n")
```

```
$ runhaskell appendtodo.hs
Secar a louça
$ runhaskell appendtodo.hs
Lavar o cachorro
$ runhaskell appendtodo.hs
Tirar a salada do forno
$ cat todo.txt
Secar a louça
Lavar o cachorro
Tirar a salada do forno
```

Precisamos adicionar o **`"\n"`** no final de cada linha porque **`getLine`** não nos dá um caractere de nova linha no final.

Ahh, mais uma coisa. Falamos sobre como escrever **`contents <- hGetContents handle`** não faz o arquivo inteiro ser lido de uma vez e armazenado na memória. A I/O é preguiçosa, então fazer isso:

```
main = do
    withFile "something.txt" ReadMode (\handle ->
        contents <- hGetContents handle
```

```
        putStr contents)
```

é como ligar com um cano do arquivo até a saída. Exatamente como você pode pensar em listas como streams, você também pode pensar em arquivos como streams. Isso irá ler uma linha de cada vez e imprimí-la no terminal à medida em que a leitura avança. Então você deve estar perguntando, quão largo é esse cano afinal? Com que frequência o disco será acessado? Bem, para arquivos de texto, a bufferização padrão é usualmente bufferização por linha. O que significa que a menor parte do arquivo a ser lida de uma vez é uma linha. Por isso que nesse caso, na verdade, uma linha é lida e impressa na saída, a próxima linha é lida e impressa, etc. Para arquivos binários, a bufferização padrão é geralmente a bufferização por bloco. O que significa que o arquivo será lido bloco a bloco. O tamanho do bloco é algum tamanho que o seu sistema operacional achar razoável.

Você pode controlar exatamente como a bufferização é feita usando a função **`hSetBuffering`**. Ela recebe um handle e um **`BufferMode`** e retorna uma ação de I/O que determina o modo de bufferização. **`BufferMode`** é um tipo de dados simples de enumeração e os possíveis valores que ele pode assumir são: **`NoBuffering`**, **`LineBuffering`** ou **`BlockBuffering (Maybe Int)`**. O **`Maybe Int`** serve para especificar o quão grande o bloco deve ser, em bytes. Se for **`Nothing`**, então o sistema operacional determina o tamanho do bloco. **`NoBuffering`** significa que será lido um caractere de cada vez. **`NoBuffering`** geralmente é uma porcaria porque tem que acessar o disco com muita frequência.

Aqui está o nosso trecho de código aterior, porém sem ler linha por linha, mas o arquivo inteiro em blocos de 2048 bytes.

```
main = do
    withFile "something.txt" ReadMode (\handle ->
        hSetBuffering handle $ BlockBuffering (Just 2048)
        contents <- hGetContents handle
        putStr contents)
```

Ler arquivos em blocos maiores pode ajudar se quisermos minimizar o acesso ao disco ou quando nosso arquivo é na verdade um recurso de rede lento.

Podemos também usar `hFlush`, que é uma função que recebe um handle e retorna uma ação de I/O que irá descarregar o buffer do arquivo associado ao handle. Quando estamos usando bufferização por linha, o buffer é descarregado depois de cada linha. Quando usamos bufferização por bloco, descarrega-se quando lemos um bloco. O buffer também é descarregado quando fechamos um handle. Isso significa que quando encontramos um caractere de nova linha, o mecanismo de leitura (ou escrita) transfere todos os dados até o momento. Porém, podemos usar **`hFlush`** para forçar a transferência dos dados que foram lidos até agora. Após descarregar, os dados ficam disponíveis para outros programas que estão rodando ao mesmo tempo.

Pense na leitura de um arquivo bufferizada por bloco como: sua privada está configurada para desccarregar-se assim que tenha quatro litros de água dentro dela. Então você começa a jogar água e uma vez que a marca de quatro litros seja atingida, a água é automaticamente descarregada e a informação na água que você jogou até agora é lida. Porém, você também pode descarregar a privada manualmente pressionando o botão nela. Isso faz a privada descarregar e toda a água (informação) dentro dela é lida. Caso você não tenha percebido, descarregar o vaso manualmente é uma metáfora para **`hFlush`**. Essa não é uma analogia muito boa para os padrões de analogia de programação, mas eu queria um objeto do mundo real que pudesse ser descarregado para exemplificar.

Nós já fizemos um programa para adicionar um novo item à nossa lista de tarefas em *todo.txt*, agora faremos um programa para remover um item. Eu vou somente colar o código e depois vamos analisar o programa juntos para você ver que ele é muito fácil. Usaremos algumas novas funções de **`System.Directory`** e uma nova função de **`System.IO`**, mas elas serão explicadas.

De qualquer modo, aqui está o programa para remover um item de *todo.txt*:

```
import System.IO
import System.Directory
import Data.List

main = do
    handle <- openFile "todo.txt" ReadMode
    tempdir <- getTemporaryDirectory
    (tempName, tempHandle) <- openTempFile tempdir "temp"
    contents <- hGetContents handle
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStrLn "Essas são as suas tarefas:"
    putStr $ unlines numberedTasks
    putStrLn "Qual delas você deseja excluir?"
    numberString <- getLine
    let number = read numberString
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile "todo.txt"
    renameFile tempName "todo.txt"
```

Primeiramente, nós apenas abrimos *todo.txt* no modo leitura e associamos seu handle a **`handle`**. Depois, usamos uma ação de I/O de **`System.Directory`** chamada `getTemporaryDirectory`. Ela é uma ação de I/O que pergunta ao seu sistema operacional qual é o diretório adequado para armazenar um arquivo temporário e retorna isso como seu resultado. Seu tipo é **`getTemporaryDirectory :: IO String`**, embora fosse mais claro nesse caso pensar nisso como sendo **`getTemporaryDirectory :: IO FilePath`**. Nós associamos o caminho do diretório temporário a **`tempdir`**.

Em seguida, usamos uma função que não conhecíamos que pertence a **`System.IO`** — `openTempFile`. Seu nome é bem auto-explicativo. Ela recebe um caminho para um diretório temporário e um modelo de nome para um arquivo e abre um arquivo temporário. Usamos **`"temp"`** como modelo de nome para o arquivo temporário, o que significa que o arquivo temporário será nomeado *temp* mais alguns caracteres aleatórios. Ela retorna uma ação de I/O que cria o arquivo temporário e o resultado nessa ação de I/O é um par de valores: o nome do arquivo temporário e um handle. Poderíamos simplesmente abrir um arquivo normal chamado *todo2.txt* ou algo parecido, mas é uma boa prática usar **`getTemporaryDirectory`** e **`openTempFile`** porque você garantirá que não está sobrescrevendo nada.

Em seguida, associamos o conteúdo de *todo.txt* a **`contents`**. Depois, dividimos essa string em uma lista de strings, cada uma sendo uma linha. Assim, **`todoTasks`** agora é algo como **`["Secar a louça", "Lavar o cachorro", "Tirar a salada do forno"]`**. Nós mesclamos os números de 0 em diante e essa lista com uma função que pega um número, como 3, uma string, como **`"hey"`** e retorna **`"3 - hey"`**, assim **`numberedTasks`** é **`["0 - Secar a louça", "1 - Lavar o cachorro" ...`**. Juntamos essa lista de strings em uma única string com quebras de linha usando **`unlines`** e a imprimimos essa string no terminal. Observe que ao invés de fazer isso, também poderíamos ter feito **`mapM putStrLn numberedTasks`**

Perguntamos ao usuário qual atividade ele quer excluir e esperamos ele digitar um número. Digamos que ele queira excluir a atividade 1, que é **`Lavar o cachorro`**, então ele digita **`1`**. **`numberString`** agora é **`"1"`** e porque nós queremos um número, não uma string, aplicamos **`read`** a ela para obter **`1`** e associamos isso a **`number`**.

Lembre-se das funções **`delete`** e **`!!`** de **`Data.List`**. **`!!`** retorna um elemento de uma lista com determinado índice e **`delete`** exclui a primeira ocorrência de um elemento em uma lista e retorna uma nova lista sem aquela ocorrência. **`(todoTasks !! number)`** (number agora é **`1`**) retorna **`"Lavar o cachorro"`**. Associamos **`todoTasks`** sem a primeira ocorrência de **`"Lavar o cachorro"`** a **`newTodoItems`** e então juntamos isso em uma única string usando **`unlines`** antes de escrever no arquivo temporário que abrimos. O arquivo antigo está intacto e o arquivo temporário contém todas as linhas que o antigo contém, exceto pela linha que excluímos.

Depois disso fechamos tanto o arquivo original quanto o temporário e então removemos o original com `removeFile`, que, como você pode ver, recebe um caminho para um arquivo e o exclui. Depois de deletar o antigo *todo.txt*, usamos `renameFile` para renomear o arquivo temporário para *todo.txt*. Tome cuidado, **`removeFile`** e **`renameFile`** (que estão ambos em **`System.Directory`** a propósito) recebem caminhos para arquivos como parâmetros, não handles.

Assunto encerrado! Poderíamos ter feito isso em ainda menos linhas, mas fomos bem cuidadosos para não sobrescrever algum arquivo existente e educadamente perguntamos ao sistema operacional sobre onde poderíamos colocar nosso arquivo temporário. Agora vamos testar o programa!

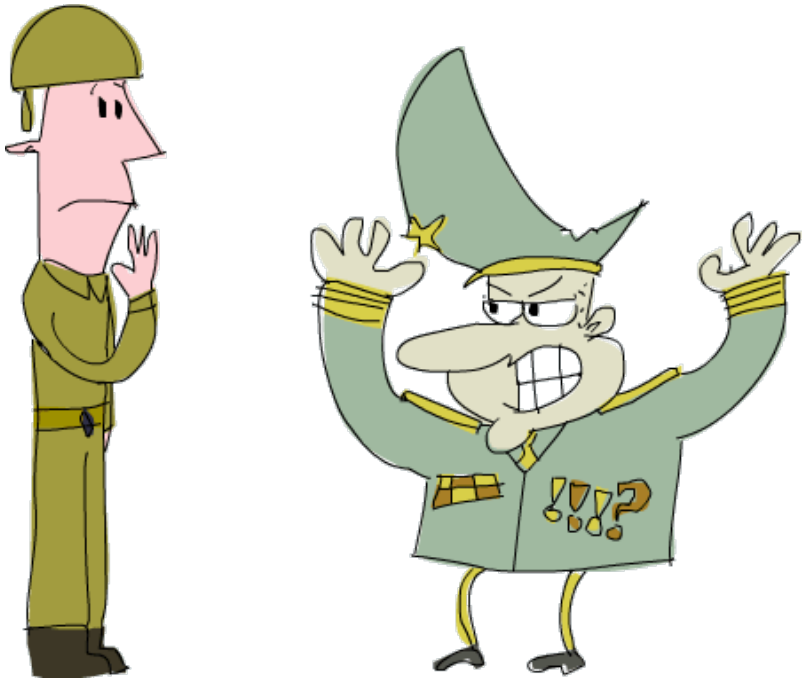
```
$ runhaskell deletetodo.hs
Essas são as suas tarefas:
0 - Secar a louça
1 - Lavar o cachorro
2 - Tirar a salada do forno
Qual delas você deseja excluir?
1

$ cat todo.txt
Secar a louça
Tirar a salada do forno

$ runhaskell deletetodo.hs
Essas são as suas tarefas:
0 - Secar a louça
1 - Tirar a salada do forno
Qual delas você deseja excluir?
0

$ cat todo.txt
Tirar a salada do forno
```

Argumentos de linha de comando

Lidar com argumentos de linha de comando é quase uma necessidade se você quiser criar um script ou aplicação que rode em um terminal. Felizmente, a biblioteca padrão de Haskell tem um método elegante de obter os argumentos de linha de comando de um programa.

Na seção anterior, fizemos um programa para adicionar um item à nossa lista de tarefas a fazer e um programa para remover um item. Existem dois problemas com a nossa abordagem. O primeiro é que simplesmente embutimos o nome do arquivo contendo a lista no nosso código. Nós simplesmente decidimos que o arquivo se chamaria *todo.txt* e que o usuário nunca teria a necessidade de lidar com várias listas de coisas a fazer.

Um modo de resolver isso é sempre perguntar ao usuário qual arquivo ele quer usar como sua lista. Usamos essa abordagem quando queríamos saber qual item o usuário queria excluir. Isso funciona, mas não é tão bom, porque requer que o usuário execute o programa, espere que o programa pergunte algo e então responda a ele. Isso é chamado de programa interativo e a dificuldade com programas de linha de comando interativos é a seguinte — e se você quiser automatizar a execução desse programa, como feito em um script de lotes? É mais difícil criar um script que interaja com um programa do que um script que simplesmente invoque um programa ou vários deles.



Por isso que algumas vezes é melhor que o usuário diga ao programa o que ele quer quando ele executar o programa, ao invés de o programa perguntar ao usuário quando for executado. E que melhor maneira de o usuário dizer ao programa o que ele quer que seja feito do que usando argumentos de linha de comando.

O módulo **`System.Environment`** tem duas ações de I/O interessantes. Uma é `getArgs`, que tem o tipo **`getArgs :: IO [String]`** e é uma ação de I/O que receberá os agumentos com os quais o programa foi invocado e terá como resultado encapsulado uma lista com os argumentos. `getProgName` tem o tipo **`getProgName :: IO String`** e é uma ação de I/O que contém o nome do programa.

Aqui está um pequeno programa que demonstra como as duas funcionam:

```
import System.Environment
import Data.List

main = do
   args <- getArgs
   progName <- getProgName
   putStrLn "Os argumentos são:"
   mapM putStrLn getArgs
   putStrLn "O nome do programa é:"
   putStrLn progName
```

Nós associamos **`getArgs`** e **`progName`** a **`args`** e **`progName`**. Escrevemos **`Os argumentos são:`** e então para todo argumento em **`args`**, executamos **`putStrLn`**. Finalmente, também imprimimos o nome do programa. Vamos compilar isto como **`arg-test`**.

```
$ ./arg-test first second w00t "multi word arg"
Os argumentos são:
first
second
w00t
multi word arg
O nome do programa é:
arg-test
```

Perfeito. Munido desse conhecimento você poderia criar algumas aplicações legais de linha de comando. Na verdade, vamos prosseguir e criar uma. Na seção anterior, fizemos um programa para adicionar tarefas e outro para excluí-las. Agora, iremos juntá-los em um único programa, que irá funcionar dependendo dos argumentos de linha de comando. Também o faremos operar com diferentes arquivos, não somente com *todo.txt*.

Iremos chamá-lo simplesmtente de *todo* e ele será capaz de fazer três coisas diferentes:

- Visualizar tarefas
- Adicionar tarefas
- Excluir tarefas

Por enquanto não iremos nos preocupar muito com possíveis entradas erradas.

Nosso programa será feito de uma forma que se quisermos adicionar a tarefa **`Encontrar a espada mágica do poder`** ao arquivo *todo.txt*, teremos que digitar **`todo add todo.txt "Encontrar a espada mágica do poder"`** no terminal. Para visualizar as tarefas simplesmente escreveremos **`todo view todo.txt`** e para remover a tarefa com o índice 2, escreveremos **`todo remove todo.txt 2`**.

Vamos começar criando uma lista de associações de redirecionamento. Ela consiste em uma lista de associações simples que tem argumentos de linha de comando como chaves e funções como seus valores correspondentes. Todas essas funções serão do tipo **`[String] -> IO ()`**. Elas irão receber a lista de argumentos como um parâmetro e retornar uma ação de I/O que fará a vizualização, adição, remoção, etc.

```
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: [(String, [String] -> IO ())]
dispatch =  [ ("add", add)
            , ("view", view)
            , ("remove", remove)
            ]
```

Temos ainda que definir **`main`**, **`add`**, **`view`** e **`remove`**, então vamos começar com **`main`**:

```
main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args
```

Primeiro, pegamos os argumentos e os associamos a **`(command:args)`**. Se você se lembra de casamento de padrões, isso significa que o primeiro argumento será associado a **`command`** e o resto deles será associado a **`args`**. Se executarmos nosso programa usando a linha de comando **`todo add todo.txt "Espancar o macaco"`**, **`command`** será **`"add"`** e **`args`** será **`["todo.txt", "Espancar o macaco"]`**.

Na próxima linha, procuramos nosso comando na lista de redirecionamentos. Como **`"add"`** está associado a **`add`**, obtemos **`Just add`** como resultado. Usamos casamento de padrão novamente para extrair nossa função do **`Maybe`**. O

que acontece se nosso comando não estiver na lista de redirecionamento? Bem, então a busca retornará **`Nothing`**, mas dissemos que não nos preocuparemos muito em falhar elegantemente, então o casamento de padrões falhará e nosso programa lançará uma exceção.

Finalmente, chamamos nossa função **`action`** (que agora é, digamos, **`add`**) com o restante da lista de argumentos. Isso retornará uma ação de I/O que adiciona um item, mostra uma lista de itens ou remove um item e devido a essa ação ser parte do bloco *do* **`main`**, ela será executada.

Excelente! Tudo que falta agora é implementar **`add`**, **`view`** e **`remove`**. Vamos começar com **`add`**:

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
```

Se chamarmos nosso programa com a linha de comando **`todo add todo.txt "Espancar o macaco"`**, o **`"add"`** será associado a **`command`** no primeiro casamento de padrões no bloco **`main`**, enquanto **`["todo.txt", "Espancar o macaco"]`** será passada à função que nós obtemos da lista de redirecionamento. Então, como não estamos lidando com entradas incorretas por enquanto, nós apenas reconhecemos imediatamente um padrão que é uma lista com esses dois elementos e retornamos uma ação de I/O que acrescenta a linha ao final do arquivo, juntamente com um caractere de nova linha.

Agora, vamos implementar a funcionalidade de visualização da lista. Se quisermos visualizar os itens em um arquivo, escrevemos **`todo view todo.txt`**. Assim, no primeiro casamento de padrão, **`command`** será **`"view"`** e **`args`** será **`["todo.txt"]`**.

```
view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStr $ unlines numberedTasks
```

Nós já fizemos praticamente a mesma coisa no programa que apenas remove tarefas quando estamos mostrando as tarefas para que o usuário possa escolher uma para remoção, só que aqui nós apenas mostramos as tarefas.

Por fim, vamos implementar **`remove`**. Ela será bem parecida com o programa que apenas remove tarefas, então se você não entender como a remoção de itens funciona aqui, dê uma olhada na explicação sobre aquele programa. A principal diferença é que não estamos embutindo *todo.txt* no código, mas obtendo com um argumento. Nós também não estamos perguntando ao usuário o número da tarefa a remover, estamos obtendo ele como um argumento.

```
remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    tempdir <- getTemporaryDirectory
    (tempName, tempHandle) <- openTempFile tempdir "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
```

```
    renameFile tempName fileName
```

Nós abrimos o arquivo com o nome **`fileName`** e abrimos um arquivo temporário, removemos a linha com o índice que o usuário deseja excluir, escrevemos isso no arquivo temporário, removemos o arquivo original e renomeamos o arquivo temporário para **`fileName`**.

Aqui está o programa todo de uma vez, em toda sua glória!

```
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: [(String, [String] -> IO ())]
dispatch =  [ ("add", add)
            , ("view", view)
            , ("remove", remove)
            ]

main = do
    (command:args) <- getArgs
    let (Just getAction) = lookup command dispatch
    getAction args

add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")

view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStr $ unlines numberedTasks

remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    tempdir <- getTemporaryDirectory
    (tempName, tempHandle) <- openTempFile tempdir "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

Para resumir nossa solução: fizemos uma associação de redirecionamento que mapeia comandos para funções que recebem alguns argumentos de linha de comando e retornam uma ação de I/O. Nós vemos o que é o comando e obtemos a função apropriada através da lista de redirecionamento. Chamamos essa função com o restante dos argumentos da linha de comando para obter uma ação de I/O que fará o procedimento apropriado e então simplesmente executamos essa ação!

Em outras linguagens, poderíamos ter implementado isso com um grande bloco switch ou algo parecido, mas usar funções de alta ordem nos permite simplesmente dizer à lista de redirecionamento para nos dar a função apropriada e então dizer a essa função para nos dar uma ação de I/O para alguns argumentos de linha de comando.

Vamos testar nossa aplicação!

```
$ ./todo view todo.txt
0 - Secar a louça
1 - Lavar o cachorro
2 - Tirar a salada do forno

$ ./todo add todo.txt "Pegar as crianças na lavanderia"

$ ./todo view todo.txt
0 - Secar a louça
1 - Lavar o cachorro
2 - Tirar a salada do forno
3 - Pegar as crianças na lavanderia

$ ./todo remove todo.txt 2

$ ./todo view todo.txt
0 - Secar a louça
1 - Lavar o cachorro
2 - Pegar as crianças na lavanderia
```

Outra coisa legal sobre isto é que é fácil adicionar funcionalidades adicionais. Basta adicionar uma entrada na lista de associações de redirecionamento e implementar a função correspondente e não terá nada mais para se preocupar! como um exercício, você pode tentar implementar uma função **`bump`** que irá receber um arquivo e um número de tarefa e retornar uma ação de I/O que coloca essa tarefa no topo da lista de coisas a fazer.

Você poderia fazer esse programa falhar de um modo um pouco mais elegante no caso de más entradas (por exemplo, se alguém escrever **`todo UP YOURS HAHAHAHA`**) criando uma ação de I/O que somente notifique que houve um erro (digamos, **`errorExit :: IO ()`**), depois verificando possíveis entradas inconsistentes e se existirem, executa-se a ação de I/O de notificação de erros. Outro modo é usar exceções, que nós estudaremos em breve.

Aleatoriedade

Muitas vezes quando se esta programando, você precisa pegar dados aleatórios. Talvez você esteja fazendo um jogo onde um dado precisa ser jogado ou que você precisa gerar alguns dados de teste para testar seu programa. Há uma série de usos para dados aleatórios enquanto programamos. Bem, atualmente, pseudoaleatório, porque todos nós sabemos que a única verdadeira fonte de aleatoriedade é um macaco em um monociclo com um queijo em uma mão e a outra na bunda. Nesta seção, vamos dar uma olhada em como fazer Haskell gerar dados aparentemente aleatórios.

Na maioria das outras linguagens de programação, você tem funções que lhe dão de volta um número aleatório. Cada vez que você chama essa função, você recebe de volta um número aleatório diferente(espero). E em Haskell? Bem, lembre, Haskell é uma linguagem funcional pura. O que isto significa é que ele tem transparência referencial. O que ISTO significa é que uma função, se dado os mesmos parâmetros duas vezes, deve produzir o mesmo resultado duas vezes. Isso é muito legal, pois nos permite raciocinar de forma diferente sobre os programas e nos permite adiar a avaliação até que realmente precisemos dela. Se eu chamar uma função,

posso ter certeza que ela não vai fazer qualquer coisa engraçada antes de me dar os resultados. Tudo o que importa são os resultados. No entanto, isso faz com que seja um pouco complicado para obter números aleatórios. Se eu tenho uma função como esta:

```
randomNumber :: (Num a) => a
randomNumber = 4
```

Isso não é muito útil como uma função de número aleatório, porque ela vai sempre retornar **`4`**, mesmo que eu possa garantir que o 4 é completamente aleatório, porque eu usei um dado para determinar ele.

Como outras linguagens fazem números aparentemente aleatórios? Bem, elas pegam várias informações do seu computador, como a hora atual, quanto e onde você moveu o mouse e que tipo de ruídos que você fez atrás de seu computador para com base nisso lhe dar um número que parece muito aleatório. A combinação desses fatores (de aleatoriedade) é provavelmente diferente em um determinado momento no tempo, de modo a obter um número aleatório diferente.

Ah. Então em Haskell, nós podemos fazer um número aleatório, se em seguida fizermos uma função que recebe como parâmetro essa aleatoriedade e com base nisso retornar um número (ou outro tipo de dados).

Veja o módulo **`System.Random`**. Ele tem todas as funções que satisfazem nossa necessidade de aleatoriedade. Vamos mergulhar em uma das funções que ele exporta depois, chamada `random`. Este é o seu tipo: **`random :: (RandomGen g, Random a) => g -> (a, g)`**. Opa! Algumas typeclasses novas nessa declaração de tipo pintando por aqui! A typeclass `RandomGen` é para tipos que podem atuar como fontes de aleatoriedade. A typeclass `Random` é para coisas que podem assumir valores aleatórios. Um valor booleano pode assumir um valor aleatório, isto é **`True`** ou **`False`**. Um número também pode assumir uma infinidade de valores aleatórios diferentes. Pode uma função assumir um valor aleatório? Eu não penso assim, provavelmente não! Se tentarmos traduzir a declaração de tipo de **`random`** para Inglês, teremos algo como: isto pega um gerador aleatório (essa é a nossa fonte de aleatoriedade) e retorna um valor aleatório e um novo gerador aleatório. Porque isso sempre retorna um novo gerador bem como um valor aleatório? Então, veremos em seguida.

Para utilizar nossa função **`random`**, temos que ter em nossas mãos um desses geradores aleatórios. O módulo **`System.Random`** exporta um tipo bem legal chamado de `StdGen` que é uma instância do tipo de classe **`RandomGen`**. Podemos fazer um **`StdGen`** manualmente ou podemos dizer ao sistema que nos dê um baseado em uma infinidade de coisas aleatórias.

Para criar manualmente um gerador aleatório, use a função `mkStdGen`. Ela tem o tipo **`mkStdGen :: Int -> StdGen`**. Ela pega um inteiro e baseada nele nos dá um gerador aleatório. Beleza então, vamos tentar usar **`random`** e **`mkStdGen`** em parceria para nos dar um (super randômico) número.

```
ghci> random (mkStdGen 100)
```

```
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Random a' arising from a use of `random' at <interactive>:1:0-20
    Probable fix: add a type signature that fixes these type variable(s)
```

O que é isto? Ah, certo, a função **`random`** pode retornar um valor de qualquer tipo, que faz parte da typeclass **`Random`**, por isso temos que informar ao Haskell que TIPO de tipo que queremos. Também não vamos esquecer que ele retorna um valor aleatório e um gerador aleatório juntos.

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Finalmente! Um número que parece meio aleatório! O primeiro componente da tupla é o nosso número, enquanto que o segundo componente é uma representação textual do nosso novo gerador aleatório. O que acontece se nós chamarmos **`random`** com o mesmo gerador aleatório novamente?

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Claro. O mesmo resultado para os mesmos parâmetros. Vamos tentar lhe dar um gerador aleatório diferente como parâmetro.

```
ghci> random (mkStdGen 949494) :: (Int, StdGen)
(539963926,466647808 1655838864)
```

Certo, legal, excelente, um número diferente. Podemos usar a notação de tipos para obter tipos diferentes de volta a partir da função.

```
ghci> random (mkStdGen 949488) :: (Float, StdGen)
(0.8938442,1597344447 1655838864)
ghci> random (mkStdGen 949488) :: (Bool, StdGen)
(False,1485632275 40692)
ghci> random (mkStdGen 949488) :: (Integer, StdGen)
(1691547873,1597344447 1655838864)
```

Vamos fazer uma função que simula jogar uma moeda três vezes. Se **`random`** não retornou um novo gerador, juntamente com um valor aleatório, nós teríamos que fazer esta função pegar três geradores aleatórios como um parâmetro e, em seguida, retornar aos lançamentos de moeda para cada um deles. Mas isso soa mal, porque se um gerador pode fazer um valor aleatório de tipo **`Int`** (o qual pode assumir uma abundância de valores diferentes), ele deve ser capaz de fazer três lançamentos de moeda (que pode assumir precisamente oito combinações). Portanto, aqui é onde **`random`** retornando um novo gerador, juntamente com um valor realmente vem a calhar.

Vamos representar uma moeda com um simples **`Bool`**. **`True`** é a cauda, **`False`** é a cabeça.

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

Chamamos **`random`** com o gerador que pegamos como parâmetro para obter uma moeda e um novo gerador. Então, nós chamamos de novo, só que desta vez com o nosso novo gerador, para obter a segunda moeda. Fazemos o mesmo para a terceira moeda. Se tivéssemos chamado com o mesmo gerador todas vezes, todas as moedas teriam o mesmo

valor e nós seríamos capazes somente de obter **`(False, False, False)`** ou **`(True, True, True)`** como resultado.

```
ghci> threeCoins (mkStdGen 21)
(True,True,True)
ghci> threeCoins (mkStdGen 22)
(True,False,True)
ghci> threeCoins (mkStdGen 943)
(True,False,True)
ghci> threeCoins (mkStdGen 944)
(True,True,True)
```

Observe que nós não temos que fazer **`random gen :: (Bool, StdGen)`**. Isso porque já especificamos que queremos booleanos na declaração do tipo da função. É por isso que Haskell pode inferir que queremos um valor booleano nesse caso.

Então, o que acontece se quisermos virar quatro moedas? Ou cinco? Bem, há uma função chamada `randoms` que pega um gerador e retorna uma sequência infinita de valores com base no gerador.

```
ghci> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507,545074951,-1015194702,-1622477312,-502893664]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True,True,True,True,False]
ghci> take 5 $ randoms (mkStdGen 11) :: [Float]
[7.904789e-2,0.62691015,0.26363158,0.12223756,0.38291094]
```

Por que **`randoms`** não retorna um novo gerador, assim como uma lista? Poderíamos implementar a função **`randoms`** muito facilmente assim:

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in value:randoms' newGen
```

Uma definição recursiva. Obtemos um valor randômico qualquer e um novo gerador a partir do nosso gerador atual, então criamos uma lista contendo um valor como sendo a cabeça e um valor randômico com base no nosso novo gerador como sendo a cauda. Como precisamos ser capazes de gerar uma quantidade infinita de números, não queremos ter de volta um novo gerador randômico.

Podemos fazer uma função que gera um fluxo finito de números e um novo gerador da seguinte forma:

```
finiteRandoms :: (RandomGen g, Random a, Num n) => n -> g -> ([a], g)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
    let (value, newGen) = random gen
        (restOfList, finalGen) = finiteRandoms (n-1) newGen
    in  (value:restOfList, finalGen)
```

De novo, uma definição recursiva. Dizemos que se quisermos 0 números, apenas retornamos uma lista vazia e o gerador que nos foi dado. Para qualquer outro número de valores aleatórios, primeiro obtemos um número aleatório e um novo gerador. Essa será a cabeça. Então dizemos que a cauda será *n - 1* números gerados com o novo gerador. Então retornamos a cabeça e o resto da lista se junta e o gerador final que iremos obter será dado por *n - 1* números aleatórios.

E se quisermos um valor aleatório em algum tipo de intervalo? Todos os números inteiros aleatórios até agora foram escandalosamente grande ou pequeno. E se nós queremos para jogar um dado? Bem, usamos `randomR` para esse propósito. Ela tem um tipo de **`randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)`**, significando que é como um tipo **`random`**, ela pega apenas como primeiro parâmetro um par de valores que são os limites mínimos e máximos e o valor final produzido estará no intervalor desses límites.

```
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
```

Há também `randomRs`, que produz um fluxo de valores aleatórios dentro de nossos intervalos definidos. Veja só:

```
ghci> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

Legal, se parece com uma senha super secreta ou algo assim.

Você pode estar se perguntando, o que este capítulo tem a ver com I/O? Nós não fizemos nada relacionado a I/O até agora. Bem, até agora nós sempre fizemos o nosso gerador de números aleatórios manualmente, fazendo-o com algum inteiro arbitrário. O problema é que se fizermos isso em nossos programas reais, eles irão sempre retornar os mesmos números aleatórios, o que definitivamente é nada bom para nós. É por isso que **`System.Random`** oferece a ação de I/O `getStdGen`, que tem um tipo de **`IO StdGen`**. Quando o programa é iniciado, ele pede ao sistema por um bom gerador de números aleatórios e armazena ele no chamado gerador global. **`getStdGen`** obtém esse gerador aleatório global quando você o relaciona a algo.

Aqui está um simples programa que gera uma string aleatória.

```
import System.Random

main = do
    gen <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen)
```

```
$ runhaskell random_string.hs
pybphhzzhuepknbykxhe
$ runhaskell random_string.hs
eiqgcxykivpudlsvvjpg
$ runhaskell random_string.hs
nzdceoconysdgcyqjruo
$ runhaskell random_string.hs
bakzhnnuzrkgvesqplrx
```

Be careful though, just performing **`getStdGen`** twice will ask the system for the same global generator twice. If you do this:

Tenha cuidado, ao executar **`getStdGen`** duas vezes ele vai pedir ao sistema para o mesmo gerador global duas vezes. Se você fizer isso:

```
import System.Random
```

```
main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen2 <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen2)
```

você terá a mesma string impressa duas vezes! Uma maneira de obter duas strings diferentes de tamanho 20 é criando um fluxo infinito e em seguida, pegar os 20 primeiros caracteres e imprimi-los em uma linha e, logo a seguir, pegar os próximos 20 caracteres e imprimi-los na segunda linha. Para isso, podemos usar a função **`splitAt`** de **`Data.List`**, que divide uma lista com um índice qualquer e retorna uma tupla que tem a primeira parte como sendo o primeiro componente e a segunda parte como sendo o segundo componente.

```
import System.Random
import Data.List

main = do
    gen <- getStdGen
    let randomChars = randomRs ('a','z') gen
        (first20, rest) = splitAt 20 randomChars
        (second20, _) = splitAt 20 rest
    putStrLn first20
    putStr second20
```

Another way is to use the `newStdGen` action, which splits our current random generator into two generators. It updates the global random generator with one of them and encapsulates the other as its result.

Outra maneira é usar a ação `newStdGen`, que divide o nosso atual gerador aleatório em dois geradores. Ele atualiza o gerador aleatório global com um deles e encapsula o outro como seu resultado.

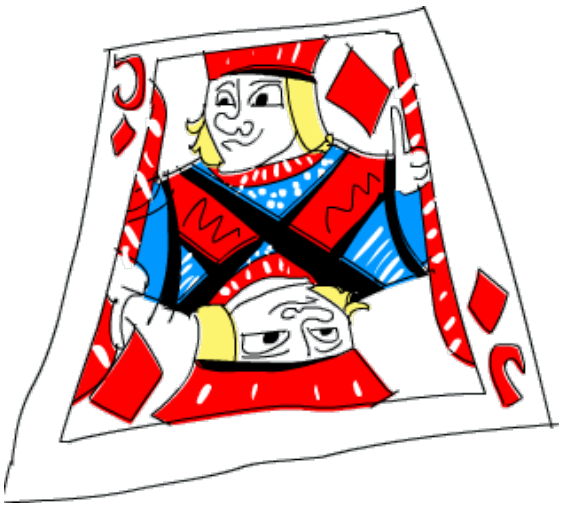
```
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen' <- newStdGen
    putStr $ take 20 (randomRs ('a','z') gen')
```

Não obtemos apenas um novo gerador aleatório quando relacionamos **`newStdGen`** há algo, o global também é atualizado, portanto se fizermos **`getStdGen`** novamente e vinculá-lo a algo, teremos um gerador que não é o mesmo que **`gen`**.

Aqui está um pequeno programa que irá convidar o usuário a adivinhar qual número o programa está pensando.

```
import System.Random
import Control.Monad(when)

main = do
    gen <- getStdGen
    askForNumber gen

askForNumber :: StdGen -> IO ()
askForNumber gen = do
    let (randNumber, newGen) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
```

```
    let number = read numberString
    if randNumber == number
        then putStrLn "You are correct!"
        else putStrLn $ "Sorry, it was " ++ show randNumber
    askForNumber newGen
```



Fazemos uma função **`askForNumber`**, que pega um gerador de números aleatórios e retorna uma ação I/O que irá solicitar ao usuário um número e dizer-lhe se ele adivinhou certo. Nesta função, primeiro geramos um número aleatório e um novo gerador baseado no gerador que obtemos como parâmetro e os chamamos de **`randNumber`** e **`newGen`**. Digamos que o número gerado foi **`7`**. Então dizemos ao usuário para adivinhar qual o número que o programa está pensando. Executamos **`getLine`** e associamos seu resultado a **`numberString`**. Quando o usuário digita **`7`**, **`numberString`** torna-se **`"7"`**. Depois, usamos **`when`** para verificar se a string digitada pelo usuário é uma string vazia. Se for, um ação I/O vazia de **`return ()`** é executada, que efetivamente finaliza o programa. Se não for, a ação que existe no bloco *do* é executada. Usamos **`read`** em **`numberString`** para convertê-la em um número, assim **`number`** agora é **`7`**.

> **Desculpe-me!** Se o usuário nos der aqui alguma informação que **`read`** não puder ler (como **`"haha"`**), nosso programa irá quebrar com uma mensagem de erro bem feia. Se você não quer que seu programa falhe com entradas errôneas, use `reads`, que retorna uma lista vazia quando ele não consegue ler uma string. Quando ele consegue, ele retorna uma lista única com uma tupla que tem o nosso valor desejado como um componente e uma string com o que ele não consumiu como o outro.

Nós verificamos se o número que damos como entrada é igual ao gerado aleatoriamente e damos ao usuário a mensagem apropriada. E então chamamos **`askForNumber`** recursivamente, só que desta vez com o novo gerador que obtemos, o que nos dá uma ação I/O que é igual à que foi realizada, ela apenas depende de um gerador diferente e então executamos isso.

`main` consiste em apenas obter um gerador aleatório do sistema e chamar **`askForNumber`** com ele para obter a ação inicial.

Aqui está o nosso programa em ação!

```
$ runhaskell guess_the_number.hs
Which number in the range from 1 to 10 am I thinking of? 4
Sorry, it was 3
Which number in the range from 1 to 10 am I thinking of? 10
You are correct!
Which number in the range from 1 to 10 am I thinking of? 2
Sorry, it was 4
Which number in the range from 1 to 10 am I thinking of? 5
Sorry, it was 10
Which number in the range from 1 to 10 am I thinking of?
```

Outra maneira de fazer este mesmo programa seria assim:

```haskell
import System.Random
import Control.Monad(when)

main = do
    gen <- getStdGen
    let (randNumber, _) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        newStdGen
        main
```

É muito semelhante à versão anterior, só que em vez de fazer uma função que recebe um gerador e, em seguida, chama a si mesmo de forma recursiva com o novo gerador atualizado, nós fazemos todo o trabalho em **`main`**. Depois de dizer ao usuário se eles estavam corretos em sua suposição ou não, nós atualizamos o gerador global e em seguida chamamos **`main`** novamente. Ambas as abordagens são válidas, mas eu gosto mais da primeira, uma vez que faz menos coisas em **`main`** e também nos fornece uma função que podemos reutilizar facilmente.

Bytestrings

Listas são legais e muito úteis como estrutura de dados. Por enquanto, estamos usando elas bastante por todos lugares. Existe uma grande variedade de funções que operam nelas e a avaliação preguiçosa de Haskell nos permite trocar os loops *for* e *while* das outras linguagens por filtros e mapeamentos nas listas, como a avaliação ocorre apenas uma vez isso precisa ser assim, então coisas como listas infinitas (inclusive infinitas listas de listas infinitas!) não são um problema para nós. Esse é o porque de listas poderem ser usadas também para representar *streams*, mesmo quando lemos a partir de uma entrada padrão ou quando a partir de arquivos. Podemos apenas abrir um arquivo e lê-lo como uma string, mesmo no caso dele ser acessado apenas quando for necessário.

Entretanto, processar arquivos como strings tem um inconveniente: isso costumar ser bem lento. Como você sabe, **`String`** é um tipo sinônimo de **`[Char]`**. **`[Char]`** não tem um tamanho fixo, porque precisamos de vários bytes para um único caracter, como por exemplo um Unicode. Além disso, listas realmente são preguiçosas. Se você tem uma lista como **`[1,2,3,4]`**, ela será avaliada somente quando for estritamente necesário. Então a lista inteira é ordenada pela premissa de uma lista. Lembre-se que **`[1,2,3,4]`** é um açucar sintático para **`1:2:3:4:[]`**. Quando o primeiro elemento da lista é forçado a ser avaliado (vamos supor, ao imprimi-lo), o resto da lista **`2:3:4:[]`** ainda é apenas uma promessa de lista, e assim por diante. Então você deve pensar em listas como promessas para os próximos elementos que serão entregues quando realmente tiverem que ser entregues, e junto com ela, a promessa do elemento a seguir. Não requer um grande esforço mental para se concluir que processar uma simples lista de números é uma série de promessas e que isso provavelmente não é a coisa mais eficiente do mundo.

Essa sobrecarga não nos afeta muito o tempo inteiro, mas pode se tornar uma deficiência quando estivermos lendo grandes arquivos e os manipulando. Por isso que Haskell tem **bytestrings**. Bytestrings são como listas, cada elemento tem o tamanho de um byte (ou 8 bits). A forma como ele realiza a avaliação preguiçosa também é diferente.

Bytestrings vem em dois sabores: as restritas e as preguiçosas. As bytestrings restritas encontram-se em **`Data.ByteString`** e elas eliminam a avaliação preguiçosa completamente. Não existem promessas envolvidas; elas reprentam uma série de bytes em um array e ponto. Você não tem com elas coisas como bytestrings infinitas. Se você avaliar o primeiro byte de uma bytestring restrita, você terá que avalia-lá por inteiro. A parte boa é que temos bem menos sobrecargas porque não temos *thunks* (o termo técnico para *promessas*) envolvidas. A parte ruim é que elas irão ocupar a sua memória mais rapidamente porque elas lêem da memória de uma só vez.

O outro tipo de bytestrings encontram-se em **`Data.ByteString.Lazy`**. Elas são preguiçosas, tão preguiçosas quanto listas. Como dissemos antes, existem tantos *thunks* (promessas) em uma lista quanto elementos. Dai o porque desse tipo ser tão lento para certos propósitos. Bytestrings preguiçosas lançam mão de uma abordagem diferente - elas são armazenadas em pedaços (não confuda com thunks!!), cada pedaço tem um tamanho de 64K. Então se avaliarmos uma bytestring preguiçosa (imprimindo ela ou algo assim), o primeiro 64K será avaliado. Depois disso, ela será apenas uma promessa para o restos dos pedaços. Bytestrings preguiçosas são como listas de bytestrings restritas com um tamanho de 64K. Quando você processa um arquivo com bytestring preguiçosa, ela será lida pedaço por pedaço. O que é legal porque o uso de memória não vai lá nas alturas e 64K provavelmente fecha perfeitamente em CPU's de cache L2.

Se você der uma olhada na documentação para **`Data.ByteString.Lazy`**, você verá que tem um monte de funções que tem os mesmos nomes de **`Data.List`**, só a assinatura de tipo que tem **`ByteString`** ao invés de **`[a]`** e **`Word8`** ao invés **`a`** nelas. As funções com os mesmos nomes geralmente funcionam da mesma forma que funcionam nas listas. Como os nomes são os mesmos, nós vamos dar um *import* "qualificado" no script e então carregar o script no GHCI pra brincar com bytestrings.

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

`B` contém tipos e funções de bytestring preguiçosa, enquanto **`S`** contém as restritas. Geralmente vamos usar a versão preguiçosa.

A função `pack` tem a assinatura de tipo **`pack :: [Word8] -> ByteString`**. Significando que ela recebe uma lista de bytes com o tipo **`Word8`** e retorna uma **`ByteString`**. Você pode imaginar que ela recebe uma lista, que é preguiçosa, e a torna menos preguiçosa, significando então que ela será preguiçosa somente nos intervalos de 64K.

Qual que é o esquema daquele tipo **`Word8`**? Bem, ele é como um **`Int`**, que possui um intervalo bem menor, normalmente algo entre 0~255. Ele representa um número de 8-bit. E assim como **`Int`**, ele também esta dentro da typeclass **`Num`**. Por exemplo, sabemos que o valor **`5`** é polifórmico e pode agir como qualquer tipo numérico. Então, ele também pode ser do tipo **`Word8`**.

```
ghci> B.pack [99,97,110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwx" Empty
```

Como você pode notar, normalmente não nos preocupamos muito em relação ao **`Word8`**, porque o sistema de tipos pode fazer com o que o número escolha qual o seu tipo. Se você tentar usar um número grande, **`336`**, por exemplo, como sendo um **`Word8`**, ele só irá empacotar em torno de **`80`**.

Nós empacotamos somente um punhado de valores em uma **`ByteString`**, que se encaixam dentro de um pedaço. O **`Empty`** é como o **`[]`** para listas.

`unpack` é a função inversa de **`pack`**. Ela pega uma bytestring e converte ela em uma lista de bytes.

`fromChunks` pega uma lista de bytestrings restritas e as converte em bytestring preguiçosas. `toChunks` pega uma lista de bytestrings preguiçosas e as converte em bytestring restritas.

```
ghci> B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], S.pack [46,47,48]]
Chunk "()*" (Chunk "+,-" (Chunk "./0" Empty))
```

Isso é bom se você tem um monte de pequenas bytestrings restritas e você quer processa-las eficientemente sem junta-las em uma bytestring restrita em memória primeiro.

A versão em bytestring para **`:`** chama-se `cons`. Ela recebe um byte e uma bytestring e coloca o byte no começo. Entretanto ela é preguiçosa, então ela fará um novo pedaço mesmo que o primeiro pedaço na bytestring não estiver preenchido. Por essa razão que é sempre melhor usar a versão restrita de **`cons`**, `cons'` se você estiver pensando em adicionar um monte de bytes no começo de uma bytestring.

```
ghci> B.cons 85 $ B.pack [80,81,82,84]
Chunk "U" (Chunk "PQRT" Empty)
ghci> B.cons' 85 $ B.pack [80,81,82,84]
Chunk "UPQRT" Empty
ghci> foldr B.cons B.empty [50..60]
Chunk "2" (Chunk "3" (Chunk "4" (Chunk "5" (Chunk "6" (Chunk "7" (Chunk "8" (Chunk "9" (Chunk
<"
Empty))))))))))
ghci> foldr B.cons' B.empty [50..60]
Chunk "23456789:;<" Empty
```

Como você pode perceber `empty` cria uma bytestring vazia. Notou a diferença entre **`cons`** e **`cons'`**? Com o **`foldr`**, começamos com uma bytestring vazia e então obtivemos uma lista de números a partir da direita, adicionando cada número ao inicio da nossa bytestring. Quando usamos **`cons`**, terminamos com um pedaço para cada byte, e meio que falhamos no nosso propósito.

Por outro lado, o módulo bytestring carrega um bando de funções análogas as de **`Data.List`**, incluindo, mas não limitada somente a, **`head`**, **`tail`**, **`init`**, **`null`**, **`length`**, **`map`**, **`reverse`**, **`foldl`**, **`foldr`**, **`concat`**, **`takeWhile`**, **`filter`**, etc.

Esse módulo inclui também funções com os mesmos nomes e comportamentos das funções encontradas em **`System.IO`**, somente as **`String`**s são substituidas pelas **`ByteString`**s. Por exemplo, a função **`readFile`** em **`System.IO`** tem o tipo **`readFile :: FilePath -> IO String`**, enquanto que a `readFile` do módulo bytestring tem o tipo **`readFile :: FilePath -> IO ByteString`**. Veja bem, se você estiver usando bytestring restritas e

precisar ler um arquivo, ele será lido em memória uma única vez! Com bytestring preguiçosas, ele será lido em vários pedaços.

Vamos codar um programinha bem simples que pega dois arquivos como argumentos na linha de comando e copia o primeiro arquivo no segundo. Note que **`System.Directory`** já tem uma função chamada **`copyFile`**, mas vamos implementar assim mesmo a nossa própria função para copiar arquivos.

```
import System.Environment
import qualified Data.ByteString.Lazy as B

main = do
    (fileName1:fileName2:_) <- getArgs
    copyFile fileName1 fileName2

copyFile :: FilePath -> FilePath -> IO ()
copyFile source dest = do
    contents <- B.readFile source
    B.writeFile dest contents
```

Acabamos de fazer nossa própria função que recebe dois **`FilePath`**s (lembre-se, **`FilePath`** é apenas um sinônimo para **`String`**) e retornamos uma ação I/O que irá copiar um arquivo dentro do outro usando bytestring. Na função **`main`**, apenas recebemos os argumentos e chamamos nossa função que tem a ação I/O, que fará o trabalho.

```
$ runhaskell bytestringcopy.hs something.txt ../../something.txt
```

Note que um programa que não utiliza bytestrings poderia muito bem se parecer com esse, a unica diferença é que usamos **`B.readFile`** e **`B.writeFile`** ao invés de **`readFile`** e **`writeFile`**. Na maioria das vezes você poderá converter um programa que usa strings normais em um programa que faz uso de bytestrings fazendo apenas os *imports* necessários e colocando o módulo qualificado na frente de alguma função. As vezes, você terá que converter funções que você mesmo escreveu para funcionar com strings para que funcionem com bytestrings, mas isso não é difícil.

Sempre que você precisar de um boa performance em um programa que lê um monte de dados em uma string, tente usar bytestrings, as chances são de você ter uma bom ganho de performance com um pouco de esforço da sua parte. Eu geralmente escrevo meus programas com strings normais e então os converto para usar bytestrings se a performance não estiver satisfatória.

Exceções

Todas linguagens tem procedures, funções e pedaços de código que em algum momento falham. Essas coisas acontecem na vida. Para diferentes linguagens existem diferentes maneiras de lidar com essas falhas. Em C, normalmente retornamos algum valor anormal (como **`-1`** ou um ponteiro nulo) para indicar que aquilo que a função esta retornando não deve ser tratado como um valor normal. Java e C#, por outro lado, preferem usar exceções para lidar com falhas. Quando uma exceção é levantada, o fluxo de controle pula para algum código que definimos para fazer algum tratamento e talvez levantar novamente alguma exceção que nos levará para algum outro código que irá lidar com o erro e tomar conta dele.

Haskell tem um belo sistema de tipos. Tipos de dados algébrico que permitem usarmos tipos como **`Maybe`** e **`Either`** em valores que esses tipos vão representar e que podem ter ou não algum valor. Em C, ao retornar, digamos, **`-1`** para

uma falha, é uma convenção amplamente aceita. Mas ela tem significado apenas para humanos. Se não tomarmos cuidado, podemos cair no erro de tratar esses valores anormais como algo normal e causar um caos completo no nosso código. O sistema de tipos em Haskell nos fornece algo muito mais seguro nesse aspecto. A função `a -> Maybe b` claramente indica que isso pode produzir um `b` envolto por um `Just` ou pode nos retornar `Nothing`. O tipo é diferente de simplesmente `a -> b` e se tentarmos usar alternadamente essas duas funções, o compilador irá reclamar para nós.

Apesar de ter tipos expressivos que suportam computações falhas, Haskell ainda tem suporte para exceções, porque elas fazem muito sentido no contexto de I/O. Um bando de coisas podem dar errado quando lidamos com o mundo externo porque ele simplesmente não é confiável. Por exemplo, quando abrimos um arquivo, um monte de coisas podem dar errado. O arquivo pode estar bloqueado, ele pode não estar mais lá ou o HD ou alguma outra pode não estar mais disponível. Então é interessante ser capaz de pular para uma parte em no nosso código que irá lidar com o erro quando ele aparecer.

Beleza, então código I/O (ou código impuro) pode lançar exceções. Isso faz todo sentido. Mas e código puro? Também pode lançar exceções. Pense sobre as funções `div` e `head`. Elas tem tipos de `(Integral a) => a -> a -> a` e `[a] -> a`, respectivamente. Sem `Maybe` ou `Either` no tipo retornado e assim mesmo ambas podem falhar! `div` explode na nossa cara se tentarmos dividir por zero e `head` tem chiliques se dermos uma lista vazia pra ele.

```
ghci> 4 `div` 0
*** Exception: divide by zero
ghci> head []
*** Exception: Prelude.head: empty list
```

Código puro pode levantar exceções, mas elas apenas podem ser capturas no nível de I/O do nosso código (quando estamos dentro do bloco *do* que esta dentro de `main`). Isso porque você não sabe quando (ou se) alguma coisa será avaliada em seu código puro, devido a avaliação preguiçosa e por não termos uma ordem bem definida de execução, enquanto que códigos I/O tem.

Anteriormente, falamos sobre como devemos gastar o mínimo de tempo possível no parte de I/O do nosso programa. A lógica do nosso programa deve residir principalmente dentro das nossas funções puras, porque seus resultados dependem apenas dos parâmetros com que as funções são chamadas. Ao lidar com funções puras, você só tem que pensar sobre o que a função retorna, porque ela não pode fazer nada além disso. Isso torna sua vida muito mais fácil. Apesar de ser necessário ter alguma lógica em I/O (como a abertura de arquivos e semelhantes), de preferência elas devem ter apenas o mínimo possível. Funções puras são preguiçosas por padrão, o que significa que nós não sabemos quando ela será avaliada e isso realmente não importa. No entanto, uma vez que as funções puras começarem a jogar exceções, será importante quando elas serão avaliadas. É por isso que só podemos pegar exceções lançadas a partir de funções puras na parte

I/O do nosso código. E isso é bem ruim, porque queremos manter a parte de I/O o mais enxuto possível. No entanto, se não pegá-las na parte I/O do nosso código, nosso programa irá quebrar. A solução? Não misture exceções e código puro. Tire vantagem de poderoso sistema tipo de Haskell e use tipos como **`Either`** e **`Maybe`** para representar os resultados que podem falhar.

Esse é o porque que vamos apenas dar uma olhada em como usar exceções I/O por enquanto. Exceções I/O são exceções que são causadas quando alguma coisa der errado enquanto estivermos nos comunicando com o mundo externo em uma ação I/O que faça parte de **`main`**. Por exemplo, quando tentamos abrir um arquivo e ele foi deletado ou coisa assim. Dê uma olhada nesse programa que abre um arquivo cujo nome é dado através da linha de comando do terminal e nos diz quantas linhas o arquivo tem.

```
import System.Environment
import System.IO

main = do (fileName:_) <- getArgs
          contents <- readFile fileName
          putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
```

Um programa super simples. Nós processamos a ação I/O **`getArgs`** e atribuimos a primeira string da lista que é dada por **`fileName`**. Então damos ao conteúdo do arquivo o nome de **`contents`**. Finalmente, aplicamos **`lines`** nesse conteúdo para obter uma lista de linhas para saber então o tamanho dessa lista e devolver isso para **`show`** que terá uma string representando esse número. Ela funciona conforme o esperado, mas e se acontecer dela receber o nome de um arquivo que não existe?

```
$ runhaskell linecount.hs i_dont_exist.txt
linecount.hs: i_dont_exist.txt: openFile: does not exist (No such file or directory)
```

Aha, recebemos um erro do GHC, nos dizendo que o arquivo não existe. Nosso programa quebrou. E se quisermos imprimir uma bela mensagem informando que o arquivo não existe? Um jeitão de fazer isso é verificando se o arquivo existe antes de tentar abrir ele usando a função `doesFileExist` do módulo **`System.Directory`**.

```
import System.Environment
import System.IO
import System.Directory

main = do (fileName:_) <- getArgs
          fileExists <- doesFileExist fileName
          if fileExists
              then do contents <- readFile fileName
                      putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!
              else do putStrLn "The file doesn't exist!"
```

Fizemos **`fileExists <- doesFileExist fileName`** porque **`doesFileExist`** tem o tipo de **`doesFileExist :: FilePath -> IO Bool`**, que significa que ele irá retornar uma ação I/O que tem o seu resultado representado por um valor boleano que irá nos dizer se o arquivo existe. Não podemos simplesmente usar **`doesFileExist`** em um *if* diretamente.

Outra solução aqui pode ser o uso de exceções. Elas são perfeitamente aceitáveis dentro desse contexto. Um arquivo não existir é uma exceção a nível de I/O, e pegar isso em I/O é tranquilo e belezinha.

Para lidar com isso usando exceções, podemos tirar vantagem da função `catch` de **`System.IO.Error`**. O tipo dela é **`catch :: IO a -> (IOError -> IO a) -> IO a`**. Ela recebe dois parâmetros. O primeiro é uma ação I/O. Que pode ser por exemplo uma ação I/O que tenta abrir um arquivo. O segundo é um pseudo handler. Se a primeira ação I/O fornecida para o **`catch`** lançar uma exceção I/O, essa exceção será passada para o handler, que vai decidir o que fazer. Então o resultado final é uma ação I/O que irá agir de acordo com o primeiro parametro ou que irá fazer o que o handler quiser caso a primeira ação I/O levantar uma exceção.

Se você já é familiarizado com os blocos *try-catch* de linguagens como Java ou Python, então a função **`catch`** é semelhante a eles. O primeiro parâmetro é aquela coisa que vamos tentar, o tipo de coisa que é feito em blocos *try* de linguagens imperativas. O segundo parâmetro é o handler que cuida da exceção, assim como a maioria dos blocos *catch* que pegam a exceção que você quer dar uma examinada pra ver o que aconteceu. O handler é envocado se uma exceção é levantada.

O handler pega um valor do tipo **`IOError`**, que é um valor que significa que uma exceção I/O aconteceu. Isso também carrega infomação em relação ao tipo de exceção que foi levantada. Como esse tipo é implementado depende da implementação da própria linguagem, que significa que não podemos inspecionar valores do tipo **`IOError`** por pattern matching, assim como não podemos usar pattern match em valores do tipo **`IO qualquer-coisa`**. Podemos usar vários predicados úteis para encontrar coisas sobre valores do tipo **`IOError`** que é o que já iremos aprender dentro de alguns instantes.

Vamos colocar então o nosso novo amiguinho **`catch`** pra funcionar!

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e = putStrLn "Whoops, had some trouble!"
```

Primeiramente, notamos que se colocarmos aqueles *backticks* (pequeno caracter: `) ao redor nós podemos usa-lá como uma função infixa, porque ela recebe dois parâmetros. Usando como uma função infixa ela fica mais legível. Então **`toTry `catch` handler`** é o mesmo que **`catch toTry handler`**, que encaixa bem com o seu tipo. **`toTry`** é a ação I/O que nós tentamos executar e **`handler`** é a função que recebe um **`IOError`** e retorna uma ação para ser executada no caso de exceção.

Vamos colocar isso pra rodar:

```
$ runhaskell count_lines.hs i_exist.txt
The file has 3 lines!
```

```
$ runhaskell count_lines.hs i_dont_exist.txt
Whoops, had some trouble!
```

No handler, nós não checamos para ver qual o tipo de **`IOError`** recebemos. Simplesmente dizemos **`"Ooppss, deu algum problema!"`** para qualquer tipo de erro. Apenas pegar todos os tipos de exceções em um handler é uma má pratica em Haskell assim como é também na maioria das outras linguagens. E se acontecer alguma outra exceção que não queremos pegar, quando nós mesmos interrompendo o programa ou algo assim? Esse é porque que vamos fazer o mesmo que fazemos em outras linguagens: verificar qual o tipo de exceção estamos recebendo. Se for o tipo de exceção que estamos esperando receber, fazemos o que queremos fazer. Se não for, jogamos essa exceção de volta pro mato. Vamos alterar nosso programa para pegar somente as exceções causadas por um arquivo não existente.

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = putStrLn "The file doesn't exist!"
    | otherwise = ioError e
```

Tudo continua igual exceto o handler, que modificamos para pegar somente um certo grupo de exceções I/O. Aqui nós usamos duas novas funções de **`System.IO.Error`** — [function]isDoesNotExistError e [function]ioError. **`isDoesNotExistError`** é um predicado dos **`IOError`**s, que significa que aquela função vai receber um **`IOError`** e retornar um **`True`** ou **`False`**, significando que ele tem um tipo **`isDoesNotExistError :: IOError -> Bool`**. Nós usamos ele na exceção que irá passar para o nosso handler dar uma olhada se tem um erro causado pelo arquivo não existir. Nós usamos a sintaxe de guardas aqui, mas poderiamos também ter usado um *if else*. Se ela não foi causada porque o arquivo não existe, nós levantamos novamente a exceção que foi passada pelo handler com a função **`ioError`**. Isso tem o tipo **`ioError :: IOException -> IO a`**, que então recebe um **`IOError`** e executa uma ação I/O que irá levanta-lá. A ação I/O tem o tipo **`IO a`**, porque ela nunca de fato produz um resultado, então ela pode se comportar como **`IO anything`**.

Quando a exceção levantada na ação I/O **`toTry`** que colocamos ao lado do bloco *do* não for causada por um arquivo não existente, **`toTry `catch` handler`** irá pegar isso e em seguida levanta-la novamente. Muito lega, hein?

Existem vários predicados que atuam em **`IOError`** e se um guarda não avalia-lo como **`True`**, a avaliação passa para o próximo guarda. Os predicados que atuam em **`IOError`** são:

- isAlreadyExistsError
- isDoesNotExistError
- isAlreadyInUseError
- isFullError
- isEOFError

- isIllegalOperation
- isPermissionError
- isUserError

Muitos deles são auto-explicativos. **isUserError** avalia para **True** quando usamos a função [function]userError para criar uma exceção, que será usada para criar exceções do nosso código e equipa-lo com uma string. Por exemplo, você pode fazer **ioError $ userError "remote computer unplugged!"**, embora seja preferível que você use tipos como **Either** e **Maybe** para expressar a possibilidade de falhas ao invés de levantar exceções você mesmo com **userError**.

Então você pode ter um handler que se parece mais ou menos com isso:

```
handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = putStrLn "The file doesn't exist!"
    | isFullError e = freeSomeSpace
    | isIllegalOperation e = notifyCops
    | otherwise = ioError e
```

Onde **notifyCops** e **freeSomeSpace** são ações I/O que definimos. Certifique-se de levantar novamente as exceções se elas não atenderem a nenhum dos critérios, caso contrário seu programa irá falhar silenciosamente em alguns casos onde ele não deveria.

System.IO.Error também exporta funções que nos habilitam a pedir para nossas exceções por alguns atributos, como onde que a manipulação de um arquivo causou um erro, ou qual o nome do arquivo. Os que iniciam com **ioe** você pode ver uma lista completa deles na documentação. Digamos que queremos imprimir o nome do arquivo que causou o erro. Não podemos imprimir o **fileName** que recebemos de **getArgs**, porque somente o **IOError** foi passado para o handler e o handler não sabe nada além disso. A função depende somente dos paramêtros com que ela foi chamada. Esse é o porque podemos usar a função [function]ioeGetFileName, que tem o tipo **ioeGetFileName :: IOError -> Maybe FilePath**. Ela pega um **IOError** como parâmetro e talvez retorna um **FilePath** (que é só um sinônimo para **String**, lembre-se, por isso é meio que a mesma coisa). Basicamente o que isso faz é extrair o caminho do arquivo do **IOError**, se ele tiver. Vamos mudar o nosso programa para imprimir o caminho do arquivo responsável pela exceção ocorrida.

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e =
        case ioeGetFileName e of Just path -
> putStrLn $ "Whoops! File does not exist at: " ++ path
                                   Nothing -
> putStrLn "Whoops! File does not exist at unknown location!"
    | otherwise = ioError e
```

No guarda onde **`isDoesNotExistError`** é **`True`**, nós usamos a expressão *case* para chamar **`ioeGetFileName`** com **`e`** e então casar o padrão contra o valor retornado por **`Maybe`**. O uso da expressão *case* é normalmente usada quando queremos casar padrões contra alguma coisa sem provocar uma nova função.

Você não precisa usar um handler para **`catch`** exceto em todas as partes de I/O. Você pode apenas cobrir certas partes do seu código I/O com **`catch`** ou você pode cobrir várias delas com **`catch`** e usar diferentes handlers para ele, como por exemplo:

```
main = do toTry `catch` handler1
          thenTryThis `catch` handler2
          launchRockets
```

Aqui, **`toTry`** usa **`handler1`** com seu handler e **`thenTryThis`** usa **`handler2`**. **`launchRockets`** não é um parâmetro de **`catch`**, então seja qual for a exceção ele pode lançar o que esta quebrando nosso programa, a não ser que **`launchRockets`** use **`catch`** internamente para manipular como sua própria exceção. Claro que code]toTry, **`thenTryThis`** e **`launchRockets`** são ações I/O que devem estar grudadas com *do* e hipoteticamente definidos em outro lugar. Isso é meio que similar com blocos *try-catch* de outras linguagens, onde você pode cercar seu programa inteiro com um único *try-catch* ou pode usar uma abordagem mais refinada e usar vários outros em diferentes partes do seu código para contralar que tipo de erro aconteceu e onde.

Agora você sabe como lidar com exceções I/O! Levantar exceções em código puro e trabalhar com elas não foi coberto aqui, principalmente porque, como dissemos antes, Haskell oferece soluções muito melhores para indicar erros ao invés de reverte-lo em I/O para captura-los. Mesmo quando colamos ações I/O juntas elas podem falhar, eu prefiro ter meu tipo algo como **`IO (Either a b)`**, significando que são ações I/O normais mas o resultado que elas produzem são do tipo **`Either a b`**, ou seja, é tanto **`Left a`** ou **`Right b`**.

Criando seus próprios tipos e typeclasses | Índice | Resolvendo Problemas Funcionalmente