For a Few Monads More Índice

Zippers

Embora a pureza de Haskell seja acompanhada de vários benefícios, ela nos faz abordar alguns problemas diferentemente do que em linguagens impuras. Por causa da transparência referencial, um valor em Haskell é igual a outro se eles representem a mesma coisa.

Então se tivermos uma árvore cheia de cincos e quisermos mudar um desses para seis, nós temos que saber exatamente qual cinco da nossa árvore nós queremos mudar. Temos que saber onde ele esta na nossa árvore. Em linguagens impuras, nós podemos só verificar na memória onde o cinco está localizado e muda-lo. Mas em Haskell, um cinco é igual ao outro, então não podemos diferencia-lo baseado na sua posição na memória. Nós também não podemos *alterar* qualquer valor; quando falamos em mudar uma árvore, isso significa que estamos recebendo uma árvore e devolvendo uma nova árvore semelhante a original, mas levemente diferente.

Uma coisa que podemos fazer é lembrar o caminho da raiz da árvore até o elemento que queremos mudar. Nós podemos dizer, pegue a árvore, vá para esquerda, vá para direita e então esquerda novamente e mude o elemento que está nessa posição. Embora isso funcione, pode ser ineficiente. Se quisermos mais tarde mudar o elemento vizinho ao elemento previamente atualizado, temos que andar todo o caminho da raiz da árvore até o nosso elemento novamente!

Nesse capítulo, nós iremos ver como podemos manipular uma estrutura de dados de forma fácil e eficiente. Legal!

Caminhando

Como nós aprendemos na aula de biologia, existem muitos tipos de árvores, então vamos pegar a semente que usaremos para plantar a nossa. Aqui está:

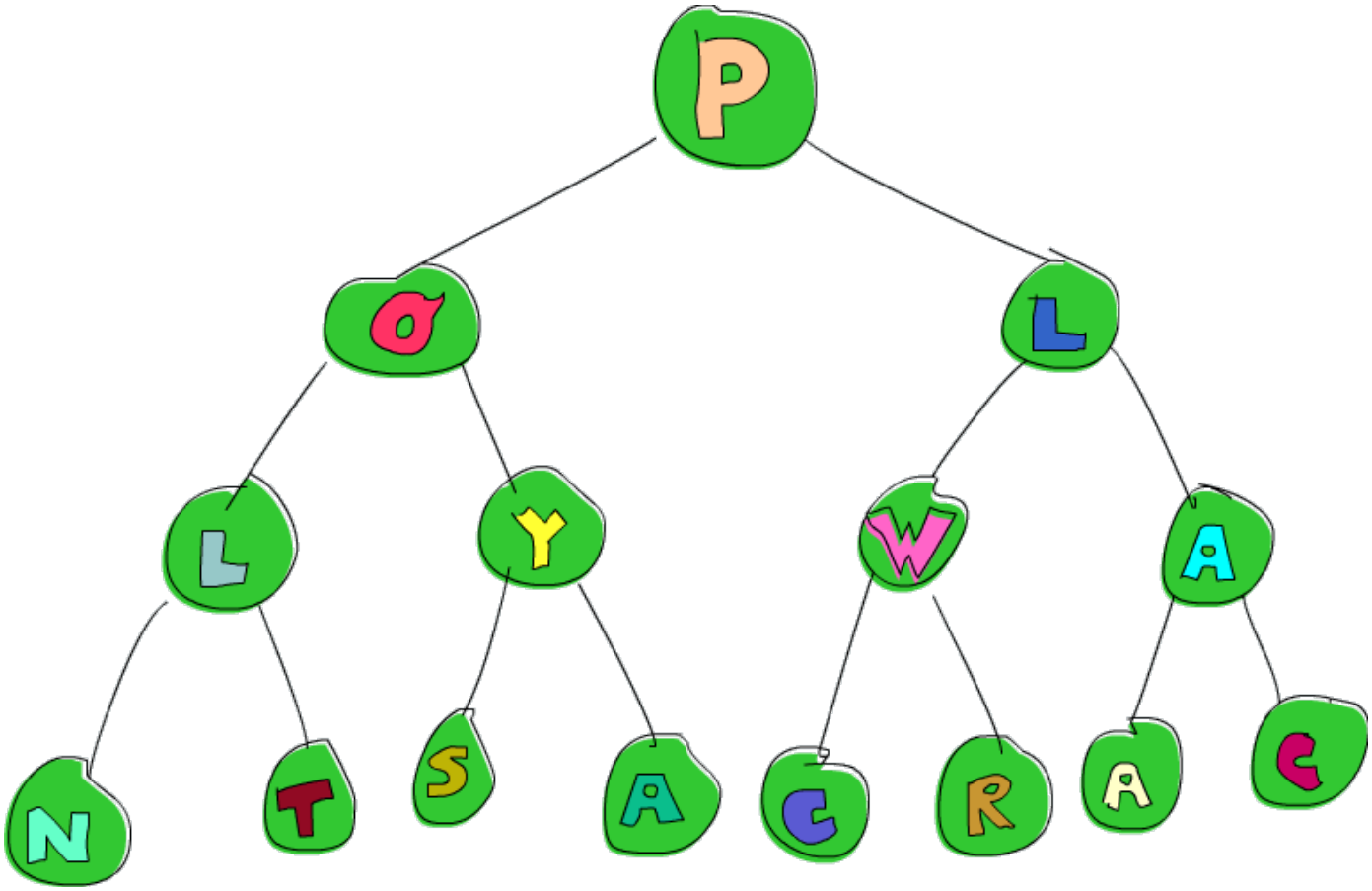
```haskell
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
```

Então nossa árvore ou está vazia ou contém um nó que tem um elemento e duas sub-árvores. Aqui está um ótimo exemplo de uma árvore, a qual eu estou dando para você, caro leitor, de graça!

```haskell
freeTree :: Tree Char
freeTree =
    Node 'P'
        (Node 'O'
            (Node 'L'
                (Node 'N' Empty Empty)
                (Node 'T' Empty Empty)
            )
            (Node 'Y'
                (Node 'S' Empty Empty)
                (Node 'A' Empty Empty)
            )
        )
        (Node 'L'
```

```
            (Node 'W'
                (Node 'C' Empty Empty)
                (Node 'R' Empty Empty)
            )
            (Node 'A'
                (Node 'A' Empty Empty)
                (Node 'C' Empty Empty)
            )
        )
```

E aqui está a representação gráfica da árvore:



Notou o `W` na árvore? Digamos que nós queremos muda-lo para um `P`. Como é que vamos fazer isso? Bem, uma forma seria percorrendo nossa árvore até acharmos um elemento que seja localizado indo primeiramente para direita e depois para esquerda e mudando esse elemento. Aqui está o código para fazer isso:

```
changeToP :: Tree Char -> Tree Char
changeToP (Node x l (Node y (Node _ m n) r)) = Node x l (Node y (Node 'P' m n) r)
```

Eca! Não só isso é feio, mas também bastante confuso. O que aconteceu aqui? Bem, nós percorremos nossa árvore a partir do elemento raiz `x` (que vem a ser o elemento `'P'` na raiz) e sua sub-árvore `l`. Ao invés de nomear a sub-árvore direita, nós continuamos percorrendo a árvore. Nós continuamos percorrendo até atingirmos a súb-árvore na qual a raiz é o `'W'`. Uma vez feito isso, nós reconstruímos a árvore mudando a sub-árvore que contém o `'W'` como raiz mas agora alterando seu valor para `'P'`.

Existe uma maneira melhor de fazer isso? Que tal nós fazermos nossa função usar a árvore juntamente com uma lista de direções. As direções irão ser ou `E` ou `D`, representando esquerda ou direita respectivamente. Com isso iremos alterar o elemento que alcançarmos ao seguir as direções fornecidas. Aqui está:

```haskell
data Direction = L | R deriving (Show)
type Directions = [Direction]

changeToP :: Directions-> Tree Char -> Tree Char
changeToP (L:ds) (Node x l r) = Node x (changeToP ds l) r
changeToP (R:ds) (Node x l r) = Node x l (changeToP ds r)
changeToP [] (Node _ l r) = Node 'P' l r
```

Se o primeiro elemento na nossa lista de direções for **`E`**, nós construímos uma nova árvore que seja parecida com a antiga, só se diferenciando por ter na sub árvore esquerda um elemento alterado para **`'P'`**. Quando chamamos recursivamente **`chageToP`**, nós passamos só a cauda da lista de direções, pois nós já fomos para esquerda. Nós fazemos a mesma coisa em caso da direção ser **`D`**. Se a lista de direções estiver vazia, isso significa que nós estamos no destino, então retornamos uma árvore parecida com a que foi passada, porém tem **`'P'`** como valor da raiz.

Para evitar imprimir a árvore toda, vamos fazer uma função que receba uma lista de direções e retorne qual elemento é alcançado pela lista:

```haskell
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```

Essa função é bastante similar a **`changeToP`**, só se difere que ao invés de guardar todo caminho ao longo das direções, ela ignora tudo exceto o destino. Aqui nós mudamos o elemento **`'W'`** para **`'P'`** e ver a mudança na nossa nova árvore:

```haskell
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

Legal, parece que funciona. Nessas funções, a lista de direções age com *foco*, por isso conseguimos exatamente a sub-árvore desejada. Uma lista de direções com **`[D]`** foca na sub-árvore que está a direita da raiz, por exemplo. Uma lista de direções vazia foca na árvore principal como um todo.

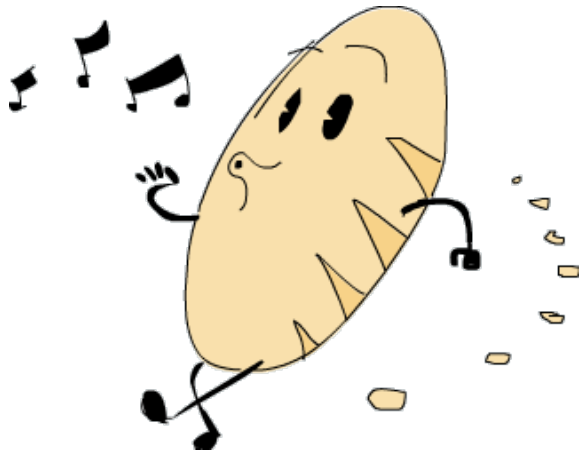
Enquanto essa técnica parece legal, pode ser uma pouco ineficiente, especialmente quando queremos mudar os elementos várias vezes. Digamos que tenhamos um árvore realmente grande e uma longa lista de direções que aponta para algum elemento que está localizado na base da árvore. Nós usamos nossa lista de direções para caminhar ao longo da árvore e mudar o elemento da base. Se nós quiséssemos mudar outro elemento que está próximo ao elemento que acabamos de mudar, nós temos que começar tudo de novo a partir da raiz e percorrer o mesmo caminho novamente! Que saco!

Na próxima seção, nós iremos encontrar uma alternativa melhor para focar em um sub-árvore, uma que permita uma troca eficiente de foco em sub-árvores vizinhas.

Uma trilha de migalhas de pão

Então, para focar em um sub-árvore queremos algo melhor do que só uma lista de direções para seguir da raiz da nossa árvore. Será que ajudaria se começássemos a partir da raiz da árvore e percorremos tanto para a esquerda quanto para a direita, um passo de cada vez e deixássemos migalhas espalhadas? É isso, quando formos para a

esquerda vamos nos lembrar que viemos da esquerda e quando formos para direita vamos lembrar que viemos da direita. Claro, nós podemos tentar isso.



Para representar as migalhas, nós também podemos usar uma lista de **`Direction`** (qual pode ser **`E`** ou **`D`**), só que ao invés de chamarmos de **`Directions`** nós podemos dizer que são **`Breadcrumbs`**, pois agora o sentido será o inverso já que estamos deixando a medida que descemos nossa árvore:

```
type Breadcrumbs = [Direction]
```

Aqui está a função que recebe a árvore e algumas migalhas percorrendo a sub-árvore esquerda enquanto adiciona **`E`** para a cabeça da lista que representa as migalhas:

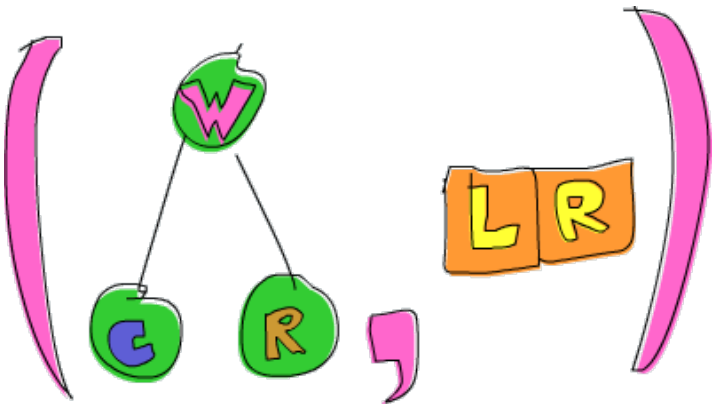
```
goLeft :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goLeft (Node _ l _, bs) = (l, L:bs)
```

Nós ignoramos o elemento da raiz e a sub-árvore direita retornando a sub-árvore esquerda juntamente com a lista de migalhas antiga com **`E`** como cabeça da lista. Aqui está a função para ir para direita:

```
goRight :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goRight (Node _ _ r, bs) = (r, R:bs)
```

Funciona do mesmo jeito. Vamos usar essas funções para receber nossa **`freeTree`** e ir para direita e para esquerda:

```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```



Ok, então agora nós temos a árvore que tem `'W'` na raiz, `'C'` na raiz da sub-árvore esquerda e `'R'` na raiz da sub-arvore direita. As migalhas são **`[E,D]`**, pois primeiro fomos para direita para depois ir para a esquerda.

Para percorrer a árvore de maneira mais clara, nós podemos usar a função **`-:`** que definimos assim:

```
x -: f = f x
```

Essa função nos permite aplicar as funções nos valores primeiramente escrevendo o valor, depois a função **`-:`** e por último a função. Então, ao invés de **`goRight (freeTree, [])`**, nós podemos escrever

`(freeTree, []) -: goRight`. Usando isso podemos reescrever o código acima de modo mais evidente de que estamos indo primeiro para direita e depois para esquerda:

```
ghci> (freeTree, []) -: goRight -: goLeft
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

Voltando para cima

E se agora nós quiséssemos voltar para cima na nossa árvore? Por causa das nossas migalhas nós sabemos que a árvore atual é ou a sub-arvore direita ou esquerda de seus pais, mas é só isso. Elas não nos dizem o suficiente sobre o pai da sub-arvore analisada para que nós sejamos capazes de voltar para o elemento pai na árvore. Parece que além da direção que tomamos, uma única migalha também deve conter todos os outros dados que precisamos para voltar a subir. Neste caso, que é o elemento na árvore do pai junto com sua sub-árvore direita.

Em geral uma única migalha deve conter todas as informações necessárias para reconstruir o nó pai. Por isso, devem ter a informação de todos os caminhos que não levam e também deve saber a direção que tomou, mas não deve conter a sub-árvore que estamos atualmente focando. Isso acontece porque nós já temos a sub-árvore no primeiro componente da tupla, então se tivéssemos também nas migalhas nós teríamos informação duplicada.

Vamos modificar nossa migalha para que elas também contenham as informações sobre tudo que anteriormente nós ignoramos quando caminhamos para esquerda e direita. Ao invés de `Direction`, nós iremos fazer uma novo tipo de dados:

```
data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)
```

Agora, ao invés de só `L`, nós temos a `LeftCrumb` que também contém o elemento do nó de onde viemos e a árvore direita que não visitamos. Ao invés de `D`, nós temos `RightCrumb`, que contém o elemento do nó de onde viemos e a árvore esquerda que nós não visitamos.

Essas migalhas agora contém todas as informações necessárias para recriar a árvore pelo qual nós caminhamos. Ao invés de ser só migalhas comuns, elas agora são mais como disquetes que nós deixamos pelo meio do caminho, pois elas contém mais informações do que só direções que nós tomamos.

Em essência, toda migalha é agora como um nó de árvore com um buraco. Quando nos movemos para o interior da árvore, a migalha carrega toda a informação do nó de onde viemos *exceto* a sub-árvore que nós escolhemos para focar. Também temos que notar onde o buraco está. Nesse caso da `LeftCrumb`, nós sabemos que nós viemos da esquerda, então a sub-árvore que está faltando é a esquerda.

Vamos também mudar nosso tipo `Breadcrumbs` para um sinônimo que reflita isso:

```
type Breadcrumbs a = [Crumb a]
```

A seguir, temos que modificar as funções `goLeft` e o `goRigth` para guardarem as informações sobre os caminhos que nós não fomos nas migalhas, ao invés de ignorar essa informação como fazíamos antes. Aqui está o `goLeft`:

```
goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
```

```
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

Você pode notar que é bastante semelhante ao nosso **`goLeft`** anterior, a diferença está que ao invés de só adicionar **`L`** a cabeça da nossa lista de migalhas, nós adicionamos a **`LeftCrumb`** para que signifique que viemos da esquerda e equipamos a **`LeftCrumb`** com o elemento do nó de que viemos (isso é o **`x`**) e a sub-árvore direita que decidimos não visitar.

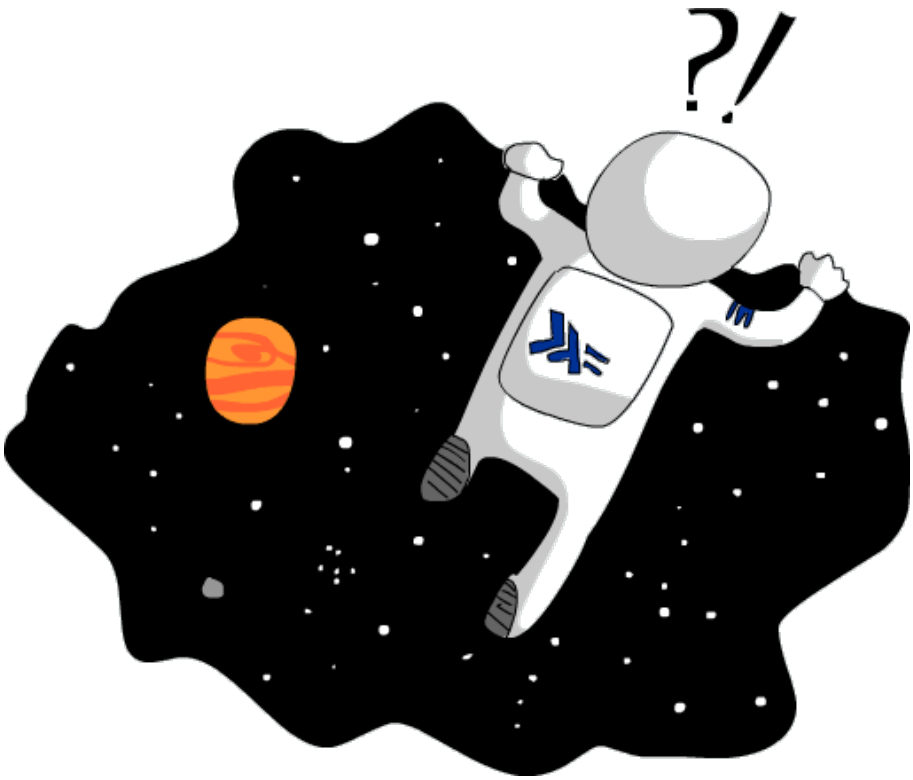
Note que essa função assume que a árvore em foco não está **`Empty`**. Uma árvore vazia não possui sub-árvores, então se nós tentarmos ir para esquerda em uma árvore vazia um erro acontecerá pois o caminho do **`Node`** não existe e não existe representação para o valor **`Empty`**.

A função **`goRigth`** é similar:

```
goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goRight (Node x l r, bs) = (r, RightCrumb x l:bs)
```

Nós éramos capazes de ir para direita e para esquerda. O que conseguimos agora é a habilidade de ir para trás pois conseguimos relembrar informações sobre os pais dos nós e os caminhos que não visitamos. Aqui está a função **`goUp`**:

```
goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```



Nós estamos focando na árvore **`t`** e checamos qual é a última *migalha*. Se a última migalha for uma **`LeftCrumb`**, então nós construímos uma nova árvore onde nossa árvore **`t`** é a sub-árvore esquerda e nós usamos a informação sobre a sub-árvore direita que nós não visitamos e o elemento para completar o resto do **`Node`**. Como nós voltamos para trás, por assim dizer, e pegamos a última migalha para recriar a árvore-mãe, a nova lista de migalhas não contém a migalha usada.

Note que essa função causa um erro se tentarmos voltar a partir do topo da árvore. Mais tarde iremos usar a monad **`Maybe`** para representar essa possibilidade de falha quando mudamos o foco.

Com um par de **`Tree a`** e **`Breadcrumbs a`**, nós temos toda a informação para reconstruir toda a árvore e também temos o foco na sub-árvore. Esse esquema também permiti facilmente mover para cima, direita ou esquerda na árvore. O par que contém a parte em foco da estrutura de dados e seus arredores é chamado de zíper, pois movendo seu foco

para cima e para baixo da estrutura de dados lembra o movimento de um zíper de uma calça comum. Então é legal fazer um tipo com esse significado:

```
type Zipper a = (Tree a, Breadcrumbs a)
```

Eu preferia chamar esse tipo de **`Focus`** pois faz mais sentido já que estamos focando em uma parte da estrutura de dados, mas o termo zíper é mais amplamente usado para descrever tal configuração, então vou ficar com **`Zipper`**.

Manipulando árvores sobe o foco

Agora que podemos nos mover para cima e para baixo, vamos fazer uma função que modifique um elemento na raiz de uma sub-árvore que o zipper está focado:

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

Se nós focamos em um nó, modificamos a raiz do elemento com a função **`f`**. Se focarmos em uma árvore vazia, deixamos ela como ela é. Agora nós podemos começar com árvore, percorrer qualquer caminho que quisermos e modificar um elemento tudo isso enquanto mantemos o foco naquele elemento podendo facilmente ir para cima ou para baixo. Um exemplo:

```
ghci> let newFocus = modify (\_ -> 'P') (goRight (goLeft (freeTree,[])))
```

Nós vamos para esquerda, depois para direita e depois modificamos o elemento da raiz para o valor **`'P'`**. Fica mais legível se usarmos a função **`-:`**:

```
ghci> let newFocus = (freeTree,[]) -: goLeft -: goRight -: modify (\_ -> 'P')
```

Nós podemos ir para cima se quisermos e substituir o elemento com o misterioso **`'X'`**:

```
ghci> let newFocus2 = modify (\_ -> 'X') (goUp newFocus)
```

Ou se escrevermos com **`-:`**:

```
ghci> let newFocus2 = newFocus -: goUp -: modify (\_ -> 'X')
```

Mover para cima é fácil pois as migalhas que deixamos formam uma parte da estrutura de dados que não estamos focando, mas se invertermos, torna-se mais difícil. É por isso que quando queremos mover para cima nós não temos que começar da raiz e ir fazendo nossa descida, mas nós só pegamos o topo de uma árvore invertida, assim, deixando de ser invertida e adicionando uma parte ao foco.

Cada nó tem duas sub-árvores, mesmo que essas sub-árvores sejam vazias. Então, se nós focarmos em um sub-árvore vazia, uma coisa que podemos fazer é substituir ela por uma sub-árvore não vazia inserindo uma árvore a um nó folha. O código para isso é simples:

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_, bs) = (t, bs)
```

Nós recebemos uma árvore e um zipper e retornamos um novo zipper que tem o foco mudado para a árvore fornecida. Não só podemos estender árvores desse modo nós podemos substituir sub-árvores vazias por novas árvores e substituir uma sub-árvore por completo. Vamos adicionar uma árvore ao lado esquerda da nossa **`freeTree`**:

```
ghci> let farLeft = (freeTree,[]) -: goLeft -: goLeft -: goLeft -: goLeft
ghci> let newFocus = farLeft -: attach (Node 'Z' Empty Empty)
```

`newFocus` é focada agora na árvore que acabamos de adicionar e o resto permanece contido nas migalhas. Se nós usarmos a função **`goUp`** para caminhar até o topo da árvore, irá ser a mesma árvore **`freeTree`** porém com um elemento **`'Z'`** adicionado a sua esquerda.

Eu vou direto para o topo, oh yeah, até onde o ar é fresco e limpo!
Fazer uma função que percorre todo caminho até o topo da árvore, independente de onde está o foco, é bem fácil. Aqui está:

```
topMost :: Zipper a -> Zipper a
topMost (t,[]) = (t,[])
topMost z = topMost (goUp z)
```

Se nossa trilha de migalhas reforçadas estiver vazia siginifica que nós já estamos na raiz da árvore, nós só voltamos o foco. Caso contrário, nós vamos até o topo até conseguir o foco do pai e aplicamos recursivamente a função **`topMost`** para isso. Então agora nós podemos percorrer toda a árvore usando as funções **`modify`** e **`attach`** à medida que avançamos e, em seguida, quando terminarmos de fazer as nossas modificações, utilizamos **`topMost`** para focar na raiz da árvore e ver as modificações na pespectiva certa.

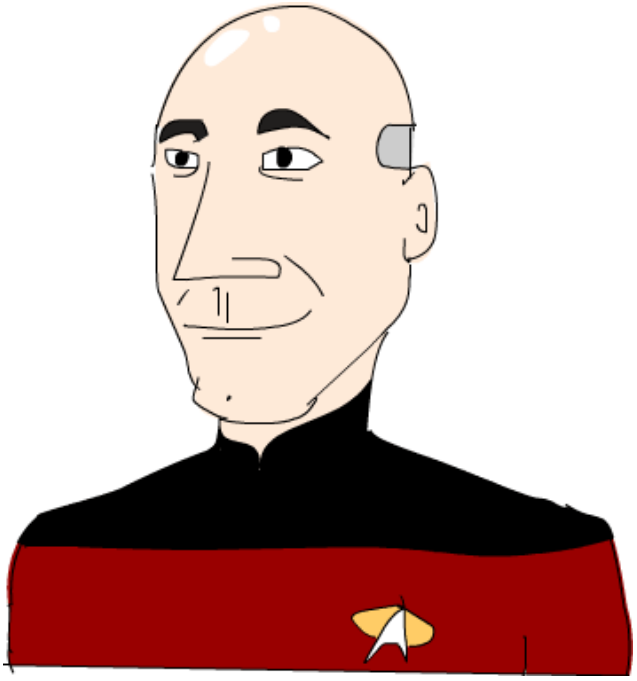
Focando nas listas

Zippers também podem ser usados como qualquer outra estrutura de dados, o que não é nenhuma surpresa já que eles podem ser usados para focar em sub-listas de listas. Afinal de contas, listas são bastante parecidas com árvores, apenas quando um nó em uma árvore tem um elemento (ou não) e várias sub-árvore, um nó em uma lista tem um elemento e apenas uma única sub-lista. Quando nós implementamos a nossa própria lista, definimos nosso tipo de dados da seguinte forma:

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

Compare isso com a nossa definição de árvore binária e será fácil ver como listas podem ser vistas árvores quando cada nó tem somente uma única sub-árvore.

Uma lista como `[1,2,3]` pode ser escrita como `1:2:3:[]`. Contendo a cabeça da lista, que será `1` e então a cauda da lista, que será `2:3:[]`. Por sua vez, `2:3:[]` também tem a sua cabeça, que é `2` e uma cauda, que é `3:[]`. Com `3:[]`, o `3` é a cabeça e a cauda é a lista vazia `[]`.



Vamos fazer um zipper para listas. Para mudar o nosso foco de sub-listas em listas, nós nos movemos para frente ou para trás (enquanto que em árvores nos movemos para esquerda ou para direita). A parte focada será uma sub-árvore e juntos com isso nós vamos deixar migalhas de pão à medida que avançamos. Agora, em que será que consiste uma migalha de pão para uma lista? Quando estamos lidando com árvores binária, dizemos que a migalha de pão terá que segurar o elemento na raiz do nó pai, juntamente com todas as sub-árvores que não escolhemos. Ele também terá que lembrar se fomos para esquerda ou para direita. Então, ele terá que ter todas as informações que um nó tem exceto para a sub-árvore que escolheu para se concentrar.

Listas são mais simples que árvores, então não precisamos lembrar se fomos para esquerda ou direita, isso porque existe somente uma forma de ir mais fundo em listas. Como há somente uma única sub-árvore em cada nó, não precisamos lembrar do caminho que fizemos para chegar lá. E ao que tudo indica, tudo o que precisamos lembrar é do elemento anterior. Se temos uma lista como `[3,4,5]` e sabemos que o elemento anterior é `2`, então nós podemos voltar apenas inserindo aquele elemento na cabeça de nossa lista, obtendo assim `[2,3,4,5]`.

Como uma migalha de pão aqui é apenas um elemento, não precisamos necessariamente colocar ela dentro de um tipo de dados, da forma como fizemos antes com o tipo de dados `Crumb` para a árvore de zippers:

```
type ListZipper a = ([a],[a])
```

A primeira lista representa a lista em que estamos focando e a segunda lista representa a migalha de pão. Vamos criar uma função que vai pra frente e para trás dentro das listas:

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs, x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

Quando estamos indo para frente, nós focamos na cauda da lista atual e deixamos a cabeça do elemento como uma migalha de pão. Quando nos movemos para trás, nós pegamos a última migalha de pão e colocamos ela no inicio da lista.

Aqui esta as duas funções em ação:

```
ghci> let xs = [1,2,3,4]
ghci> goForward (xs,[])
([2,3,4],[1])
ghci> goForward ([2,3,4],[1])
([3,4],[2,1])
ghci> goForward ([3,4],[2,1])
([4],[3,2,1])
ghci> goBack ([4],[3,2,1])
```

```
([3,4],[2,1])
```

Percebemos que as migalha de pão no caso de listas são nada mais do que apenas a parte reversa da nossa lista. O elemento que nos afastamos sempre vai para a cabeça da migalha de pão, por isso que é fácil se mover para trás apenas pegando elemento da cabeça da nossa migalha e usando isso como nosso foco.

Isso também torna mais fácil ver por que chamamos isso de zíper, porque isso realmente se parece com um ziper que se move para cima e para baixo.

Se você estiver fazendo um editor de texto, você pode usar uma lista de strings para representar as linhas de textos que estão atualmente abertas e você pode então usar o zipper para saber em qual linha o cursor esta atualmente focado. Ao usar um zipper, pode ser mais fácil também inserir uma linha em qualquer lugar do texto ou deletar alguma.

A very simple file system

Now that we know how zippers work, let's use trees to represent a very simple file system and then make a zipper for that file system, which will allow us to move between folders, just like we usually do when jumping around our file system.

If we take a simplistic view of the average hierarchical file system, we see that it's mostly made up of files and folders. Files are units of data and come with a name, whereas folders are used to organize those files and can contain files or other folders. So let's say that an item in a file system is either a file, which comes with a name and some data, or a folder, which has a name and then a bunch of items that are either files or folders themselves. Here's a data type for this and some type synonyms so we know what's what:

```
type Name = String
type Data = String
data FSItem = File Name Data | Folder Name [FSItem] deriving (Show)
```

A file comes with two strings, which represent its name and the data it holds. A folder comes with a string that is its name and a list of items. If that list is empty, then we have an empty folder.

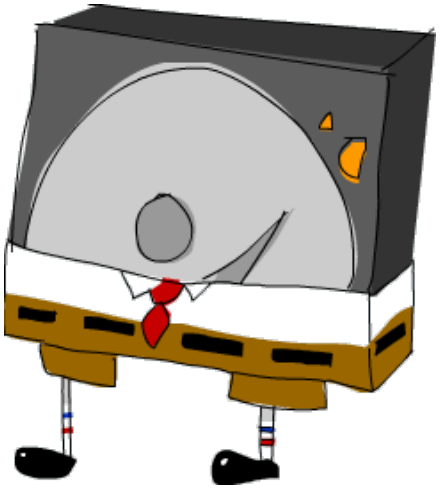
Here's a folder with some files and sub-folders:

```
myDisk :: FSItem
myDisk =
    Folder "root"
        [ File "goat_yelling_like_man.wmv" "baaaaaa"
        , File "pope_time.avi" "god bless"
        , Folder "pics"
            [ File "ape_throwing_up.jpg" "bleargh"
            , File "watermelon_smash.gif" "smash!!"
            , File "skull_man(scary).bmp" "Yikes!"
            ]
        , File "dijon_poupon.doc" "best mustard"
        , Folder "programs"
            [ File "fartwizard.exe" "10gotofart"
            , File "owl bandit.dmg" "mov eax, h00t"
            , File "not_a_virus.exe" "really not a virus"
            , Folder "source code"
                [ File "best_hs_prog.hs" "main = print (fix error)"
                , File "random.hs" "main = print 4"
                ]
            ]
```

```
        ]
```

That's actually what my disk contains right now.

A zipper for our file system



Now that we have a file system, all we need is a zipper so we can zip and zoom around it and add, modify and remove files as well as folders. Like with binary trees and lists, we're going to be leaving breadcrumbs that contain info about all the stuff that we chose not to visit. Like we said, a single breadcrumb should be kind of like a node, only it should contain everything except the sub-tree that we're currently focusing on. It should also note where the hole is so that once we move back up, we can plug our previous focus into the hole.

In this case, a breadcrumb should be like a folder, only it should be missing the folder that we currently chose. Why not like a file, you ask? Well, because once we're focusing on a file, we can't move deeper into the file system, so it doesn't make sense to leave a breadcrumb that says that we came from a file. A file is sort of like an empty tree.

If we're focusing on the folder `"root"` and we then focus on the file `"dijon_poupon.doc"`, what should the breadcrumb that we leave look like? Well, it should contain the name of its parent folder along with the items that come before the file that we're focusing on and the items that come after it. So all we need is a `Name` and two lists of items. By keeping separate lists for the items that come before the item that we're focusing and for the items that come after it, we know exactly where to place it once we move back up. So this way, we know where the hole is.

Here's our breadcrumb type for the file system:

```
data FSCrumb = FSCrumb Name [FSItem] [FSItem] deriving (Show)
```

And here's a type synonym for our zipper:

```
type FSZipper = (FSItem, [FSCrumb])
```

Going back up in the hierarchy is very simple. We just take the latest breadcrumb and assemble a new focus from the current focus and breadcrumb. Like so:

```
fsUp :: FSZipper -> FSZipper
fsUp (item, FSCrumb name ls rs:bs) = (Folder name (ls ++ [item] ++ rs), bs)
```

Because our breadcrumb knew what the parent folder's name was, as well as the items that came before our focused item in the folder (that's `ls`) and the ones that came after (that's `rs`), moving up was easy.

How about going deeper into the file system? If we're in the `"root"` and we want to focus on `"dijon_poupon.doc"`, the breadcrumb that we leave is going to include the name `"root"` along with the items that precede `"dijon_poupon.doc"` and the ones that come after it.

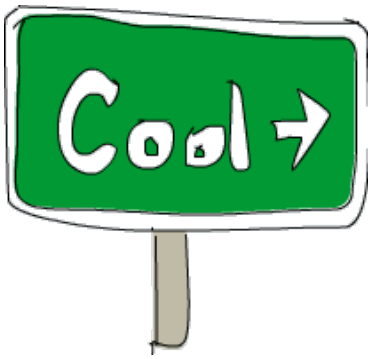
Here's a function that, given a name, focuses on a file of folder that's located in the current focused folder:

```
import Data.List (break)

fsTo :: Name -> FSZipper -> FSZipper
fsTo name (Folder folderName items, bs) =
    let (ls, item:rs) = break (nameIs name) items
    in  (item, FSCrumb folderName ls rs:bs)

nameIs :: Name -> FSItem -> Bool
nameIs name (Folder folderName _) = name == folderName
nameIs name (File fileName _) = name == fileName
```

`fsTo` takes a **`Name`** and a **`FSZipper`** and returns a new **`FSZipper`** that focuses on the file with the given name. That file has to be in the current focused folder. This function doesn't search all over the place, it just looks at the current folder.



First we use **`break`** to break the list of items in a folder into those that precede the file that we're searching for and those that come after it. If you remember, **`break`** takes a predicate and a list and returns a pair of lists. The first list in the pair holds items for which the predicate returns **`False`**. Then, once the predicate returns **`True`** for an item, it places that item and the rest of the list in the second item of the pair. We made an auxilliary function called **`nameIs`** that takes a name and a file system item and returns **`True`** if the names match.

So now, **`ls`** is a list that contains the items that precede the item that we're searching for, **`item`** is that very item and **`rs`** is the list of items that come after it in its folder. Now that we have this, we just present the item that we got from **`break`** as the focus and build a breadcrumb that has all the data it needs.

Note that if the name we're looking for isn't in the folder, the pattern **`item:rs`** will try to match on an empty list and we'll get an error. Also, if our current focus isn't a folder at all but a file, we get an error as well and the program crashes.

Now we can move up and down our file system. Let's start at the root and walk to the file **`"skull_man(scary).bmp"`**:

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsTo "skull_man(scary).bmp"
```

`newFocus` is now a zipper that's focused on the **`"skull_man(scary).bmp"`** file. Let's get the first component of the zipper (the focus itself) and see if that's really true:

```
ghci> fst newFocus
File "skull_man(scary).bmp" "Yikes!"
```

Let's move up and then focus on its neighboring file **`"watermelon_smash.gif"`**:

```
ghci> let newFocus2 = newFocus -: fsUp -: fsTo "watermelon_smash.gif"
ghci> fst newFocus2
File "watermelon_smash.gif" "smash!!"
```

Manipulating our file system

Now that we know how to navigate our file system, manipulating it is easy. Here's a function that renames the currently focused file or folder:

```
fsRename :: Name -> FSZipper -> FSZipper
fsRename newName (Folder name items, bs) = (Folder newName items, bs)
fsRename newName (File name dat, bs) = (File newName dat, bs)
```

Now we can rename our **`"pics"`** folder to **`"cspi"`**:

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsRename "cspi" -: fsUp
```

We descended to the **`"pics"`** folder, renamed it and then moved back up.

How about a function that makes a new item in the current folder? Behold:

```
fsNewFile :: FSItem -> FSZipper -> FSZipper
fsNewFile item (Folder folderName items, bs) =
    (Folder folderName (item:items), bs)
```

Easy as pie. Note that this would crash if we tried to add an item but weren't focusing on a folder, but were focusing on a file instead.

Let's add a file to our **`"pics"`** folder and then move back up to the root:

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsNewFile (File "heh.jpg" "lol") -: fsUp
```

What's really cool about all this is that when we modify our file system, it doesn't actually modify it in place but it returns a whole new file system. That way, we have access to our old file system (in this case, **`myDisk`**) as well as the new one (the first component of **`newFocus`**). So by using zippers, we get versioning for free, meaning that we can always refer to older versions of data structures even after we've changed them, so to speak. This isn't unique to zippers, but is a property of Haskell because its data structures are immutable. With zippers however, we get the ability to easily and efficiently walk around our data structures, so the persistence of Haskell's data structures really begins to shine.

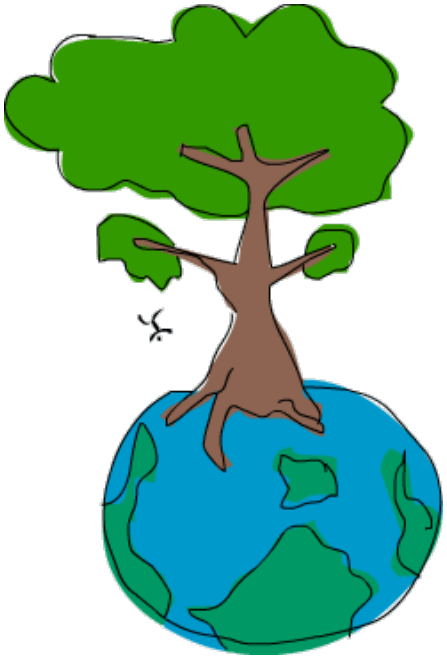
Watch your step

So far, while walking through our data structures, whether they were binary trees, lists or file systems, we didn't really care if we took a step too far and fell off. For instance, our **`goLeft`** function takes a zipper of a binary tree and moves the focus to its left sub-tree:

```
goLeft :: Zipper a -> Zipper a
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

But what if the tree we're stepping off from is an empty tree? That is, what if it's not a **`Node`**, but an **`Empty`**? In this case, we'd get a runtime error because the pattern match would fail and we have made no pattern to handle an empty tree, which doesn't have any sub-trees at all. So far, we just assumed that we'd never try to focus on the left sub-tree of an

empty tree as its left sub-tree doesn't exist at all. But going to the left sub-tree of an empty tree doesn't make much sense, and so far we've just conveniently ignored this.



Or what if we were already at the root of some tree and didn't have any breadcrumbs but still tried to move up? The same thing would happen. It seems that when using zippers, any step could be our last (cue ominous music). In other words, any move can result in a success, but it can also result in a failure. Does that remind you of something? Of course, monads! More specifically, the **`Maybe`** monad which adds a context of possible failure to normal values.

So let's use the **`Maybe`** monad to add a context of possible failure to our movements. We're going to take the functions that work on our binary tree zipper and we're going to make them into monadic functions. First, let's take care of possible failure in **`goLeft`** and **`goRight`**. So far, the failure of functions that could fail was always reflected in their result, and this time is no different. So here are **`goLeft`** and **`goRight`** with an added possibility of failure:

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (Node x l r, bs) = Just (l, LeftCrumb x r:bs)
goLeft (Empty, _) = Nothing

goRight :: Zipper a -> Maybe (Zipper a)
goRight (Node x l r, bs) = Just (r, RightCrumb x l:bs)
goRight (Empty, _) = Nothing
```

Cool, now if we try to take a step to the left of an empty tree, we get a **`Nothing`**!

```
ghci> goLeft (Empty, [])
Nothing
ghci> goLeft (Node 'A' Empty Empty, [])
Just (Empty,[LeftCrumb 'A' Empty])
```

Looks good! How about going up? The problem before happened if we tried to go up but we didn't have any more breadcrumbs, which meant that we were already in the root of the tree. This is the **`goUp`** function that throws an error if we don't keep within the bounds of our tree:

```
goUp :: Zipper a -> Zipper a
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

Now let's modify it to fail gracefully:

```
goUp :: Zipper a -> Maybe (Zipper a)
goUp (t, LeftCrumb x r:bs) = Just (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = Just (Node x l t, bs)
goUp (_, []) = Nothing
```

If we have breadcrumbs, everything is okay and we return a successful new focus, but if we don't, then we return a failure.

Before, these functions took zippers and returned zippers, which meant that we could chain them like this to walk around:

```
gchi> let newFocus = (freeTree,[]) -: goLeft -: goRight
```

But now, instead of returning **`Zipper a`**, they return **`Maybe (Zipper a)`**, so chaining functions like this won't work. We had a similar problem when we were dealing with our tightrope walker in the chapter about monads. He also walked one step at a time and each of his steps could result in failure because a bunch of birds could land on one side of his balancing pole and make him fall.

Now, the joke's on us because we're the ones doing the walking, and we're traversing a labyrinth of our own devising. Luckily, we can learn from the tightrope walker and just do what he did, which is to exchange normal function application for using **`>>=`**, which takes a value with a context (in our case, the **`Maybe (Zipper a)`**, which has a context of possible failure) and feeds it into a function while making sure that the context is taken care of. So just like our tightrope walker, we're going to trade in all our **`-:`** operators for **`>>=`**. Alright, we can chain our functions again! Watch:

```
ghci> let coolTree = Node 1 Empty (Node 3 Empty Empty)
ghci> return (coolTree,[]) >>= goRight
Just (Node 3 Empty Empty,[RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight
Just (Empty,[RightCrumb 3 Empty,RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight >>= goRight
Nothing
```

We used **`return`** to put a zipper in a **`Just`** and then used **`>>=`** to feed that to our **`goRight`** function. First, we made a tree that has on its left an empty sub-tree and on its right a node that has two empty sub-trees. When we try to go right once, the result is a success, because the operation makes sense. Going right twice is okay too; we end up with the focus on an empty sub-tree. But going right three times wouldn't make sense, because we can't go to the right of an empty sub-tree, which is why the result is a **`Nothing`**.

Now we've equipped our trees with a safety-net that will catch us should we fall off. Wow, I nailed this metaphor.

Our file system also has a lot of cases where an operation could fail, such as trying to focus on a file or folder that doesn't exist. As an exercise, you can equip our file system with functions that fail gracefully by using the **`Maybe`** monad.

For a Few Monads More Índice