Introdução Índice Tipos e Typeclasses

Começando

Preparar, apontar, foi!

Ok, vamos começar! Se você é o tipo de pessoa horrível que não lê as introduções das coisas e se você pulou ela, então de qualquer maneira você irá querer ler a última seção da introdução, porque explica o que você precisa para acompanhar este tutorial e como vamos fazer para carregar as funções. A primeira coisa que vamos fazer é rodar o GHC no modo interativo e chamar alguma função para obter uma base para começar a sentir o Haskell. Abra seu terminal e digite **`ghci`** . Você será agraciado com algo parecido com isto:

```
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude>
```

Meus parabéns, você esta no GHCi! O prompt aqui é **`Prelude>`** no entando como isto é demasiadamente longo quando você carrega algumas coisas na sua sessão, nós iremos usar **`ghci>`**. Se você quer ter o mesmo prompt, digite **`:set prompt "ghci> "`**.

Veja algumas simples operações aritméticas.

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
ghci>
```

Isto é bastante auto-explicativo. Nós podemos utilizar todos operadores comuns em uma só linha e todas suas regras usuais precedentes serão obedecidas. Nós também podemos utilizar parênteses para declarar de forma explicita a operação precedente ou para muda-la.

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

Muito legal hein?! Sim, eu sei que não... mas tenha paciência comigo. Uma pequena armadilha a se observar aqui é a negação de números. Se você quiser obter números negativos, sempre tenha a operação determinada por parênteses.

Fazendo `5 * -3` o GHCi irá gritar com você, porém fazendo `5 * (-3)` irá funcionar perfeitamente.

Álgebra booleana também é bastante simples. Você provavelmente já sabe que `&&` refere-se a expressão booleana *and*, `||` refere-se a expressão booleana *or*. `not` faz a negação de um `True` ou de um `False`.

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

Testando para igualdades fica algo assim.

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
True
```

Que tal se fizermos `5 + "llama"` ou `5 == True`? Bem, se nós tentarmos o primeiro trecho destes código, nós teremos uma grande e assustadora mensagem de erro!

```
No instance for (Num [Char])
arising from a use of `+' at <interactive>:1:0-9
Possible fix: add an instance declaration for (Num [Char])
In the expression: 5 + "llama"
In the definition of `it': it = 5 + "llama"
```

Credo! O que o GHCi esta dizendo para nós aqui é que `"llama"` não é um número e que ele não sabe como adicionar isto ao número 5. Se isso não fosse um `"llama"` mas um `"four"` ou um `"4"`, Haskell ainda assim não iria considerá-lo como um número. `+` espera-se que tenha a sua esquerda e a sua direita um número. Se nós tentarmos fazer `True == 5`, o GHCi irá nos dizer que os tipos são diferentes. Visto que o `+` funciona somente em coisas consideradas números, o `==` já funciona em quaisquer coisas que podem ser comparadas. O problema é que ambos devem ser do mesmo tipo. Não podemos comparar maças com laranjas. Iremos dar uma olhada em tipos um pouco mais tarde.

Nota: Você pode fazer `5 + 4.0` porque `5` é adaptável e pode se comportar como um inteiro ou um ponto flutuante (float). `4.0` não pode se comportar como um inteiro, então o `5` é o único que pode ser adaptado.

Talvez você não saiba mas nós estamos usando função agora o tempo todo. Por exemplo, `*` é uma função que pega dois números e então os multiplica. Como você percebe, nós invocamos ela colocando no meio dos dois números. Isto é o que nós chamamos de função *infixa*. Muitas funções não são usadas com números pois são as funções de *prefixo*. Vamos dar uma olhada então nisto.

Funções normalmente são *prefixo*, porém agora não iremos declarar explicitamente que a função é do formato *prefixo*, iremos apenas assumir isto. Na maioria das linguagens imperativas as funções são chamadas escrevendo o nome da função e então escrevendo os seus parâmetros entre parênteses, normalmente separados por vírgula. Em Haskell, as funções são chamadas escrevendo o nome da função, com um espaço e então os parâmetros separados por espaços. Inicialmente, vamos tentar chamar uma das mais chatas funções em Haskell.

```
ghci> succ 8
9
```

A função **`succ`** pega qualquer coisa e então define o seu sucessor e o retorna. Como você pode ver, nós apenas separamos o nome da função dos parâmetros com um espaço. Chamar a função com alguns parâmetros também é bastante simples. As funções **`min`** e **`max`** pegam duas coisas que podem ser colocadas em ordem (como um número!) e retorna aquele que for o menor ou maior. Veja você mesmo:

```
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
```

Aplicar uma função (chamando a função colocando um espaço depois dela, e depois ir digitando os seus parâmetros) tem sempre a maior precedência. O que isto significa para nós é que estas duas declarações são equivalentes.

```
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9) + (max 5 4) + 1
16
```

No entanto, se nós quisermos ter o sucessor do produto dos números 9 e 10, nós não podemos escrever **`succ 9 * 10`** senão isto irá pegar o sucessor do 9, que seria então a multiplicação por 10. Resultando em 100. Nós devemos escrever **`succ (9 * 10)`** para obter 91.

Se a função recebe dois parâmetros, nós podemos também chamá-la como uma função *infixa* colocando antes dela aspas simples. Por exemplo, se a função **`div`** recebe dois inteiros e realiza a divisão deles. Fazendo **`div 92 10`** o resultado será 9. Porém quando nós realizamos este tipo de chamada ela fica meio confusa para saber qual número esta dividindo e qual esta sendo dividido. Nós podemos invocar esta função como uma função *infixa* fazendo **`92 `div` 10`** e logicamente isto ficará muito mais claro.

Muitas pessoas que vieram de linguagens imperativas tendem a colocar a notação dos parênteses para especificar o uso de funções. Por exemplo, em C, você utiliza parênteses para chamar funções como **`foo()`**, **`bar(1)`** ou **`baz(3, "haha")`**. Como dissemos, espaços são utilizados para a aplicação de funções em Haskell. Então estas funções em Haskell deveriam ser **`foo`**, **`bar 1`** e **`baz 3 "haha"`**. Se você enxergar algo como **`bar (bar 3)`**, não significa que aquela chamada de **`bar`** recebe **`bar`** e **`3`** como parâmetros. Isto significa que nós primeiros chamamos a função **`bar`** com o parâmetro **`3`** para obter algum número e então chamar **`bar`** novamente com este número. Em C, isso seria algo como **`bar(bar(3))`**.

Primeiras funções do seu filho

No capítulo anterior nós aprendemos a idéia básica para chamada de funções. Agora vamos tentar fazer a nossa própria! Abra o seu editor de texto favorito e escreva a seguinte função que recebe um número e o multiplica por dois.

```
doubleMe x = x + x
```

Funções são definidas de forma semelhante como são chamadas. O nome da função é seguido por parâmetros separados por espaços. Porém quando definimos funções, terá um **=** e depois nós iremos definir o que a função faz. Salve isto como **`baby.hs`** ou algo parecido. Agora vá até o local onde o arquivo foi salvo e rode o **`ghci`** a partir dai. Dentro do GCHI, digite **`:l baby`**. Agora o script estará carregado e nós já podemos brincar com a função que definimos.

```
ghci> :l baby
[1 of 1] Compiling Main             ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

Como o **+** funciona muito bem tanto com números inteiros como com números de ponto flutuante (ou qualquer outra coisa que possa ser considerada um número) a nossa função também funcionára com qualquer número. Vamos fazer uma função que recebe dois números e multiplica cada um deles por dois, e após realiza a soma deles.

```
doubleUs x y = x*2 + y*2
```

Simples. Nós podemos definir isso também como **`doubleUs x y = x + x + y + y`**. Testar isto produzirá um resultado bastante previsível (lembre-se de anexar esta função no arquivo **`baby.hs`**, salva-lo e então rodar **`:l baby`** dentro do GHCI).

```
ghci> doubleUs 4 9
26
ghci> doubleUs 2.3 34.2
73.0
ghci> doubleUs 28 88 + doubleMe 123
478
```

Como esperado, você pode chamar suas próprias funções dentro de outras funções que você fez. Com isto em mente, nós iremos redefinir **`doubleUs`** da seguinte forma:

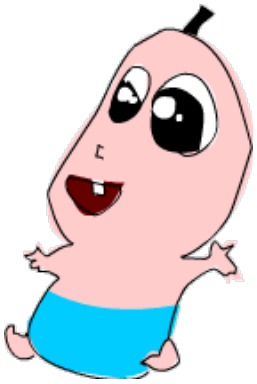
```
doubleUs x y = doubleMe x + doubleMe y
```

Este é um exemplo bastante simples de um padrão comum que você verá ao longo de Haskell. Ir fazendo funções básicas que são obviamente corretas e então combiná-las em funções mais complexas. Desta forma você também acabará evitando repetições. E se alguns matemáticos descobrissem que 2 é na verdade 3 e você tivesse que mudar o seu programa? Você deveria então simplesmente redefinir o **`doubleMe`** para ser **`x + x + x`** e desde que **`doubleUs`** chamasse **`doubleMe`**, já estaria funcionando automaticamente neste estranho mundo onde 2 é 3.

Funções em Haskell não precisam estar em uma ordem em particular, portanto não importa se você definir primeiro o **`doubleMe`** e depois o **`doubleUs`** ou se fizer o contrário.

Agora vamos fazer uma função que multiplicará um número por 2 porém somente se este número for menor ou igual a 100, porque números maiores que 100 já são grandes o suficiente.

```
doubleSmallNumber x = if x > 100
                      then x
                      else x*2
```



Neste ponto iremos introduzir o comando *if* do Haskell. Você provavelmente esta familiarizado com o comando *if* de outras linguagens. A diferença entre o comando *if* do Haskell e o comando *if* das linguagens imperativas é que a parte do *else* é obrigatória em Haskell. Em linguagens imperativas você pode simplesmente pular uma séria de etapas se a condição do *if* não for satisfatória, porém em Haskell toda expressão e função devem retornar alguma coisa. Poderíamos também escrever o comando *if* em uma só linha, mas eu acho esta forma mais legível. Outra coisa sobre o comando *if* em Haskell é que ele é uma *expressão*. Uma *expressão* é basicamente um pedaço de código que retorna um valor. **`5`** é uma expressão porque isto retorna 5, **`4 + 8`** é uma expressão, **`x + y`** é uma expressão porque retorna a soma de **`x`** e **`y`**. Por causa do *else* ser obrigatório, um *if* sempre retornará alguma coisa porque ele é uma *expressão*. Se quisermos adicionar 1 para cada número que será produzido na nossa função anterior, podemos escrevê-la da seguinte forma.

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

Se tivéssemos omitido os parênteses ele teria adicionado 1 somente se **`x`** não fosse maior do que 100. Note o **`'`** no final do nome da função. Aquela apóstrofe tem nenhum significado especial na sintaxe do Haskell. Ele é um caracter válido para ser usado em um nome de função. Normalmente nós utilizamos o **`'`** para designar uma versão especifica de uma função (aqueles que não são preguiçosos) ou em uma versão levemente modificada de uma função ou variável. Pelo fato do **`'`** ser um caractere válido em funções, nós podemos fazer uma função como esta:

```
conanO'Brien = "Este sou eu, Conan O'Brien!"
```

Há duas observações bem interessantes aqui. A primeira é que no nome da função nós não podemos escrever o nome próprio Conan's em letras maiúsculas. Isto porque funções não podem começar com letras maiúsculas. Veremos o porque disto mais adiante. A segunda observação é que esta função não tem nenhum parâmetro. Quando uma função não tem nenhum parâmetro, nós normalmente chamamos isto de *definição* (ou um *nome*). Como não podemos alterar o que os nomes (e funções) significam após termos os definido, **`conanO'Brien`** e a string **`"Este sou eu, Conan O'Brien!"`** podem ser usadas alternadamente.

Uma introdução às listas

Assim como as listas de compras no mundo real, listas em Haskell são extremamente úteis. Esta é a estrutura de dados mais utilizada e pode ser utilizada em muitas e variadas formas para modelar e resolver uma série de problemas. Listas são MUITO legais. Neste capitulo nós iremos dar uma olhada no básico de listas, strings (que são listas) e compreensões de listas.

Em Haskell, listas são estruturas de dados homogêneas. Ela armazena vários elementos do mesmo tipo. Isto quer dizer que podemos ter uma lista com inteiros ou uma lista de caracteres, porém não podemos ter uma lista com inteiros e alguns caracteres. E agora, uma lista!

Nota: Nós podemos utilizar a palavra-chave `let` para definir o nome correto no GHCi. Fazer `let a = 1` dentro do GHCi é o equivalente a escrever `a = 1` em um script e carregá-lo.

```
ghci> let lostNumbers = [4,8,15,16,23,48]
ghci> lostNumbers
[4,8,15,16,23,48]
```

Como você pode ver, listas são caracterizadas por utilizar colchetes e os valores nas listas são separados por vírgulas. Se nós tentarmos uma lista como `[1,2,'a',3,'b','c',4]`, Haskell irá reclamar que os caracteres (que são, alias, demarcados entre aspas simples) não são números. Falando em caracteres, strings são simplesmente listas de caracteres . `"hello"` é só um açúcar sintático para `['h','e','l','l','o']`. Como strings são listas, nós podemos utilizar funções de listas nelas, o que é realmente útil.

Uma prática comum é colocar duas listas juntas. Fazemos isso utilizando o operador `++`.

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

Veremos repetidamente o uso do operador `++` em strings grandes. Quando juntamos duas listas (mesmo se você adicionar uma lista de um elemento só em outra lista, como por exemplo: `[1,2,3] ++ [4]`), internamente o Haskell irá percorrer toda a lista que esta do lado esquerdo do `++`. Isto não é um problema quando não estamos lidando com listas muito grandes. Porém colocar algo no final de uma lista com cinqüenta milhões de elementos irá demorar um pouco. No entanto, colocar alguma coisa no início de uma lista utilizando o operador `:` (também chamado de *contra* operador) será instantâneo.

```
ghci> 'Q':" GATINHA"
"Q GATINHA"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

Observe que o `:` recebe um número e uma lista de números ou um caractere e uma lista de caracteres, enquanto o `++` recebe duas listas. Mesmo que você esteja adicionando um elemento ao final de uma lista com `++`, você terá que demarcá-la com colchetes para que se torne lista.

`[1,2,3]` é na verdade açúcar sintático para `1:2:3:[]`. `[]` é uma lista vazia. Se acrescentarmos `3` nele, ele irá se tornar `[3]`. Se acrescentarmos `2`, se tornará `[2,3]` e assim por diante.

Nota: `[]`, `[[]]` e `[[],[],[]]` são coisas diferentes. O primeiro é uma lista vazia, o segundo é uma lista que contém uma lista vazia, o terceiro é uma lista que contém três listas vazias.

Se você deseja obter um elemento de uma lista pelo seu índice, utilize `!!`. O índice inicia a partir de 0.

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

Mas se você tentar obter o sexto elemento de uma lista que contém somente quatro elementos, você obterá um erro portanto seja cuidadoso!

Listas também podem conter listas. Elas também podem conter listas que contêm listas que contêm listas...

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

As listas dentro de uma lista podem conter *lengths* diferentes, mas eles não podem ser de tipos diferentes. Assim como você não pode ter uma lista que tem alguns caracteres e alguns números, você não pode ter uma lista que contém algumas listas de caracteres e algumas listas de números.

As listas poderão ser comparadas se as coisas que elas contêm puderem ser comparadas. Quando utilizamos **<**, **<=**, **>** e **>=** para comparar listas, eles são comparados na ordem lexicográfica. Primeiro os cabeçalhos são comparados. Se eles forem iguais então os segundos elementos são comparados, etc.

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] > [3,4]
True
ghci> [3,4,2] > [2,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

O que mais você pode fazer com listas? Aqui estão algumas funções básicas para operações em listas.

`head` recebe uma lista e retorna o seu *head*. O *head* (cabeça) de uma lista é basicamente o primeiro elemento.

```
ghci> head [5,4,3,2,1]
5
```

`tail` recebe uma lista e retorna a sua "cauda". Em outras palavras, ele decepa a cabeça de uma lista e retorna a cauda.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

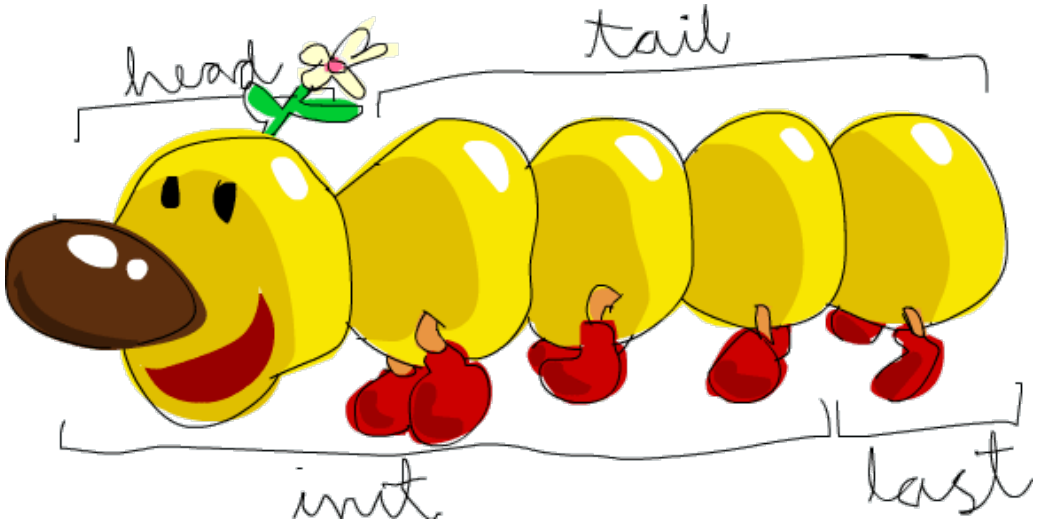
`last` recebe uma lista e retorna o seu último elemento.

```
ghci> last [5,4,3,2,1]
1
```

`init` recebe uma lista e retorna tudo com exceção do último elemento.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

Se pensarmos em uma lista como um monstro, aqui está o que é o que.



Mas o que acontece se tentarmos obter o *head* de uma lista vazia?

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

WOW! Tudo explode bem na nossa cara! Isso não é um mostro, não tem cabeça! Quando for utilizar **`head`**, **`tail`**, **`last`** e **`init`**, tenha o cuidado para não utilizar em listas vazias. Este erro não pode ser capturado em tempo de compilação por isso é sempre bom tomar certos cuidados antes de pedir para o Haskell lhe dar algum elemento de uma lista vazia.

`length` recebe uma lista e retorna o seu *length* (tamanho), obviamente.

```
ghci> length [5,4,3,2,1]
5
```

`null` verifica se a lista é vazia. Se for, então retorna **`True`**, senão retorna **`False`**. Utilize esta função no lugar de **`xs == []`** (Caso você tiver uma lista chamada **`xs`**)

```
ghci> null [1,2,3]
False
ghci> null []
True
```

`reverse` reverte uma lista.

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

`take` recebe um número e uma lista. Ele extrai a quantidade de elementos desde o início da lista.. Observe.

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

Observe que se tentarmos tirar mais elementos do que há na lista, ele retorna só a lista. Se nós tentarmos retornar 0 elementos, receberemos uma lista vazia.

`drop` funciona de forma similar, só que retira o número de elementos a partir do ínicio da lista.

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 1 [7,6,5,4]
[6,5,4]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

`maximum` recebe uma lista de coisas que podem ser colocadas em algum tipo de ordem e retorna o seu maior elemento.

`minimum` retorna o menor.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

`sum` recebe uma lista de números e retorna a sua soma.

`product` recebe uma lista de números e retorna o seu produto.

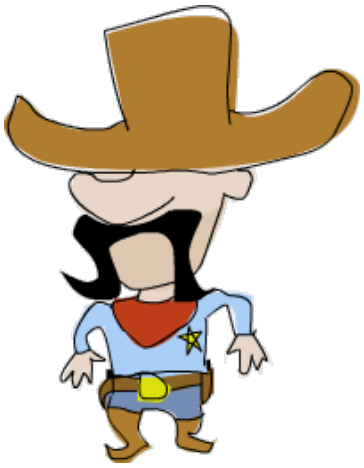
```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

`elem` recebe alguma coisa e uma lista de coisas e nos diz se esta coisa é um elemento da lista. Geralmente é chamado como uma função *infixa* porque é mais fácil de ler dessa maneira.

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

Estas são algumas funções básicas para operações em listas. Iremos dar mais uma olhada em funções de listas depois.

Texas ranges



E se você precisar de uma lista com todos os números entre 1 e 20? Claro, iríamos digitando todos eles, porém obviamente esta não seria uma solução para cavalheiros que desejam excelência em sua linguagem de programação. Em função disso, nós utilizamos ranges. Ranges é um jeito de construir listas que são uma seqüência aritmética de elementos devidamente enumerados. Números podem ser enumerados. Um, dois, três, quatro, etc. Caracteres também podem ser enumerados. O alfabeto é uma enumeração de caracteres de A a Z. Nomes também podem ser enumerados. O que vem depois de "João"? Não faço idéia.

Para constituir uma lista que contenha todos os números naturais de 1 a 20, você simplesmente deve escrever `[1..20]`. Isto é o equivalente a ter escrito `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]` e não há diferença entre um e outro com a exceção de que escrever longas seqüências enumeradas manualmente é algo bastante estúpido.

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

Ranges também são legais porque você pode definir uma etapa específica. E se nós quisermos todos os números ímpares entre 1 e 20? Ou se nós quisermos sempre ter o terceiro número entre 1 e 20?

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

Esta é uma questão simples de separar os primeiros dois elementos com uma vírgula e então especificar qual o limite máximo. Embora bastante esperto, ranges com diversas etapas não são tão espertos como muitas pessoas esperam que eles sejam. Você não pode fazer `[1,2,4,8,16..100]` e esperar que isso lhe retorne a potência de 2. Primeiramente porque você pode especificar somente uma etapa. E segundo que algumas seqüências não são aritméticas, mas sim ambíguas se tivermos somente alguns dos primeiros termos.

Para ter uma lista com todos os números entre 20 e 1, você não pode simplesmente fazer `[20..1]`, você deve fazer `[20,19..1]`.

Veja como utilizamos números ponto flutuante em ranges! Como eles não são completamente precisos (por definição), o uso deles em ranges pode retornar alguns resultados que cheiram bem mal.

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

Meu conselho é que não se faça uso deles em ranges de listas.

Você também pode utilizar ranges para fazer listas infinitas, basta não especificar qual o limite máximo. Depois nós iremos adentrar com mais detalhes nas listas infinitas. No momento, vamos examinar como que você pode ter os primeiros 24 múltiplos de 13. Claro, você quer fazer `[13,26..24*13]`. Porém eis aqui um jeito melhor: `take 24 [13,26..]`. Como Haskell é preguiçoso, ele não vai tentar analisar uma lista infinita imediatamente porque isto nunca vai acabar. Ele vai ficar esperando pra ver no que esta lista infinita vai dar. E aqui ele vê que você quer apenas os primeiros 24 elementos e de bom grado lhe retorna eles.

```
ghci> take 24 [13,26..]
[13,26,39,52,65,78,91,104,117,130,143,156,169,182,195,208,221,234,247,260,273,286,299,312]
```

Algumas funções para produzirmos listas infinitas:

`cycle` recebe uma lista e gera ciclos infinitos dela. Se você tentar mostrar o resultado, continuará para sempre até que você tente cortá-lo fora em algum lugar.

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

`repeat` recebe um elemento e produz uma lista infinita dele. Isto é como o ciclo de uma lista com somente um elemento.

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

No entanto será mais simples utilizar a função `replicate` caso você deseje algumas repetições do mesmo elemento em uma lista. `replicate 3 10` retornará `[10,10,10]`.

Eu sou a compreensão de lista

Se você já esteve em um curso de matemática, você provavelmente já viu compreensão de conjuntos. Isto é normalmente utilizado para a construção de conjuntos mais específicos de conjuntos em geral. Uma *compreensão* básica para um conjunto que contém os dez primeiros números naturais seria $S = \{2 \cdot x \mid x \in \mathbb{N},\ x \leq 10\}$. A parte depois do *pipe* é chamada de output da função, **`x`** é a variável, **`N`** é o input e **`x <= 10`** o predicado. Isso significa que o conjunto contém o dobro de todos os números naturais que satisfazem o predicado.

Se quisermos escrever isto em Haskell nós podemos fazer algo como **`take 10 [2,4..]`**. Mas se nós não quisermos dobrar os 10 primeiros números naturais mas sim aplicar alguma função mais complexa neles? Nós podemos utilizar compreensão de lista para isto. Compreensão de lista é bastante similar com compreensão de conjuntos. Vamos começar com os 10 primeiros números pares agora. A compreensão de lista que podemos usar é **`[x*2 | x <- [1..10]]`**. **`x`** é traçado a partir de **`[1..10]`** e para cada elemento em **`[1..10]`** (que temos vinculado a **`x`**) teremos esse elemento dobrado. Aqui esta a compreensão em ação.

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

Como você vê, obtemos os resultados desejados. Agora vamos adicionar uma condição (ou um predicado) nesta compreensão. Predicados ficam depois dessa parte obrigatória sendo separado por vírgula. Digamos que nós queremos somente os elementos que, dobrados, são maiores ou igual a 12.

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
```

Legal, isso funciona. E se nós quisermos todos os números entre 50 e 100 onde o resto da divisão pelo número 7 fosse 3? Fácil.

```
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

Sucesso! Note que reduzir as listas por predicados é chamado de **filtragem**. Nós temos uma lista de números e nós os filtraremos pelo predicado. Vamos a outro exemplo. Digamos que você queira uma *compreensão* que substitua cada número ímpar maior do que 10 com **`"BANG!"`** e cada número ímpar menor do que 10 com **`"BOOM!"`**. Se o número não for ímpar, queremos ele fora da lista. Convenientemente nós colocamos esta *compreensão* dentro de uma função para reutilizá-la mais facilmente.

```
boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
```

A última parte da *compreensão* é o predicado. A função **`odd`** retorna **`True`** quando o número for ímpar e **`False`** quando for par. O elemento é incluído na lista somente se todos os predicados examinados sejam **`True`**.

```
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

Podemos também incluir vários predicados. Se nós quisermos todos os números entre 10 e 20 que não sejam 13, 15 ou 19, podemos fazer:

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

Não apenas podemos ter vários predicados em uma compreensão de lista (um elemento deve satisfazer todos os predicados para ser incluso no resultado da lista), como também podemos extrair várias listas. Quando extraímos diversas listas, a compreensão produz todas as combinações da lista desejada e junta com a função de output que nós fornecemos. Uma lista produzida pela compreensão da extração a partir de duas listas com length 4 terá um length de 16, desde que não seja filtrado. Se eu tiver duas listas, `[2,5,10]` e `[8,10,11]` e quiser obter o produto de todas as combinações possíveis entre números destas listas, aqui esta o que devemos fazer.

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

Como esperado, o length da nova lista é 9. E se eu quiser todos os possíveis produtos que sejam maiores do que 50?

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

Que tal uma compreensão de lista que combina uma lista de adjetivos e uma lista de substantivos... só para dar umas risadas.

```
ghci> let substantivos = ["gerente","programador","cliente"]
ghci> let adjetivos = ["malemolente","chato","fofoqueiro"]
ghci> [substantivos ++ " " ++ adjetivos | substantivos <- substantivos, adjetivos <- adjetivos
["gerente malemolente","gerente chato","gerente fofoqueiro","programador malemolente",
"programador chato","programador fofoqueiro","cliente malemolente","cliente chato",
"cliente fofoqueiro"]
```

Já sei! Vamos escrever a nossa própria versão do **`length`**! Chamaremos isto de **`length'`**.

```
length' xs = sum [1 | _ <- xs]
```

_ significa que não me importo com o que vamos extrair da lista, ao invés de escrever o nome da variável que nunca será usada, basta escrever _. Esta função substitui todos os elementos da lista com `1` e soma todos eles. A idéia é ter o resultado da soma e obter depois o length da nossa lista.

Lembrete de amigo: como strings são listas, nós devemos usar compreensão de listas para processar e produzir strings. Eis uma função que recebe uma string e remove tudo exceto letras maiúsculas dela.

```
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Testando isto:

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "EUnaoGOSTODEGATOS"
"EUGOSTODEGATOS"
```

O predicado aqui faz todo o serviço. Ele dirá ao caractere que ele será incluído em uma nova lista somente se ele for um elemento da lista `['A'..'Z']`. Aninhar compreensões de lista também é possível se você estiver operando em uma lista que contém listas. Uma lista que contém diversas listas de números. Vamos remover todos os números ímpares sem diminui-los.

```
ghci> let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

Você pode escrever compreensão de lista em diversas linhas. Se você não estiver no GHCI, é melhor que quebre longas compreensões de lista em multiplas linhas, especialmente se forem aninhadas.

Tuplas

Em alguns casos, tuplas são como listas — um jeito de armazenar muitos valores em um lugar só. No entanto, existem algumas diferenças fundamentais. Uma lista de números é uma lista de números. Esse é o seu tipo e não importa se tiver um único número nela ou uma quantidade infinita de números. Tuplas, no entanto, são usadas quando você sabe exatamente quantos valores você quer combinar e o seu tipo depende de quantos componentes ela tem e os tipos dos componentes. Tuplas são caracterizadas por parênteses com seus componentes separados por vírgulas.

Outra diferença fundamental é que elas não precisam ser homogêneas. Ao contrário de uma lista, uma tupla pode conter uma combinação de vários tipos.

Pense em como você representaria um vetor bi-dimensional em Haskell. Um jeito seria você usar uma lista. Esse seria um jeito de fazer. E se você quiser colocar alguns vetores em uma lista para representar pontos de uma superfície plana em duas dimensões? Você poderia fazer algo como `[[1,2],[8,11],[4,5]]`. O problema com este método é que podemos fazer algumas coisas como `[[1,2],[8,11,5],[4,5]]`, com Haskell não tem problema desde que isso seja uma lista de listas com números, mas meio que não faz sentido. Mas uma tupla de tamanho dois (também chamado de um par) é o seu próprio tipo, sabendo disto aquela lista não pode ter alguns pares e depois umas triplas (uma tupla de tamanho três), mas vamos utilizar mesmo assim. Em vez de contornar os vetores com colchetes, usamos parênteses: `[(1,2),(8,11),(4,5)]` E se eu quiser uma superfície como `[(1,2),(8,11,5),(4,5)]`? Bem, isto me dará um erro:

```
Couldn't match expected type `(t, t1)'
against inferred type `(t2, t3, t4)'
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of `it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

Ele esta me dizendo que tentei usar um par e um triplo na mesma lista, isso não pode acontecer. Você não pode fazer uma lista como `[(1,2),("One",2)]` porque o primeiro elemento da lista é um par de números e o segundo elemento é um par que consiste em uma string e um número. Tuplas podem também ser usadas para representar uma variedade

de dados. Por exemplo, se eu quiser representar alguns nomes e idades em Haskell, eu poderia utilizar uma tripla: **`("Christopher", "Walken", 55)`**. Como vimos nesse exemplo, tuplas também podem conter listas.

Utilize tuplas quando você souber com antecedência que muitos componentes contêm algumas partes de dados já esperadas. Tuplas são muito mais rígidas porque cada diferença de tamanho da tupla é o seu próprio tipo, você não pode escrever uma função geral para anexar elementos para uma tupla - você deve escrever uma função para anexar elementos a um par, uma função para anexar para uma tripla, uma função para uma 4-tuplas, etc.

Enquanto você tiver listas únicas, não existirá tal coisa como uma tupla única. Realmente não faz muito sentido quando você pensa sobre isso. Uma tupla única seria apenas o valor que ele contém e, como tal, não teria nenhum benefício para nós.

Assim como as listas, tuplas podem ser comparadas umas com as outras se seus componentes puderem ser comparados. Você só não pode comparar duas tuplas de tamanho diferente, uma vez que você pode comparar duas listas de tamanhos diferentes. Duas funções úteis que operam em pares:

`fst` recebe um par e retorna seu primeiro componente.

```
ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
```

`snd` recebe um par e retorna seu segundo componente. Surpresa!

```
ghci> snd (8,11)
11
ghci> snd ("Wow", False)
False
```

Nota: estas funções operam somente em pares. Não funcionam com triplas, 4-tuplas, 5-tuplas, etc. Falaremos sobre a extração de dados de tuplas de diferentes maneiras um pouco mais tarde.

Uma função legal que produz uma lista de pares: `zip`. Ela pega duas listas e então as comprimem juntamente em uma única lista juntando os elementos que casarem em pares. Esta é realmente uma função muito simples mas tem uma infinidade de usos. Ela é bastante útil especialmente quando precisamos combinar duas listas em um único formato ou cruzar duas listas simultaneamente. Veja uma demonstração:

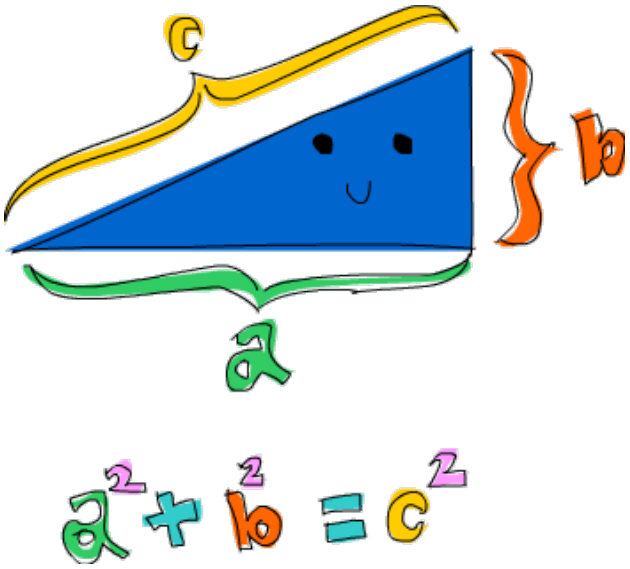
```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["um", "dois", "tres", "quatro", "cinco"]
[(1,"um"),(2,"dois"),(3,"tres"),(4,"quatro"),(5,"cinco")]
```

Ela pareia todos os elementos e produz uma nova lista. O primeiro elemento vai com o primeiro, o segundo com o segundo, etc. Observe que como pares podem ter tipos diferentes entre eles, **`zip`** pode pegar duas listas que contém tipos diferentes e comprimir em uma só. O que acontece se os tamanhos das listas forem diferentes?

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["eu","sou","uma", "tartaruga"]
[(5,"eu"),(3,"sou"),(2,"uma"),(6,"tartaruga")]
```

A lista maior é simplesmente cortada para combinar com o tamanho da outra menor. Como Haskell é preguiçoso, nós podemos "zippar" listas finitas com listas infinitas:

```
ghci> zip [1..] ["banana", "laranja", "melancia", "uva"]
[(1,"banana"),(2,"laranja"),(3,"melancia"),(4,"uva")]
```



$$a^2 + b^2 = c^2$$

Aqui esta um problema para combinar tuplas e compreensão de listas: Que triângulo retângulo que tem números inteiros para todos os lados e todos os lados iguais ou menores que 10 e tem um perímetro de 24? Primeiro, vamos tentar gerar todos os triângulos com lados iguais ou menores que 10:

```
ghci> let triangles = [ (a,b,c) | c <- [1..10], b <- [1..10], a <- [1..10] ]
```

Estamos apenas desenhando a partir de três listas e a nossa função de output irá combiná-los em uma tripla. Se você avaliar isso digitando **`triangles`** no GHCI, você terá uma lista de todos os possíveis triângulos com os lados menores ou iguais a 10. A seguir, iremos adicionar uma condição que todas elas têm de ser triângulos retângulos. Vamos também modificar esta função levando em consideração que o lado b não é maior do que a hipotenusa e esse lado não é maior do que o lado b.

```
ghci> let rightTriangles = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b], a^2 + b^2 == c^
```

Estamos quase terminando. Agora, acabamos de modificar a função, dizendo que queremos aqueles em que o perímetro é igual a 24.

```
ghci> let rightTriangles' = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b], a^2 + b^2 == c
ghci> rightTriangles'
[(6,8,10)]
```

E cá esta nossa resposta! Este é um padrão comum em programação funcional. Você tem um conjunto inicial de soluções e em seguida você aplica transformações a essas soluções e as filtram até chegar ao resultado correto.

Introdução Índice Tipos e Typeclasses