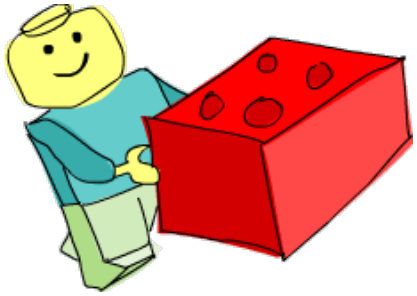
Funções de alta ordem Índice Criando seus próprios tipos e typeclasses

Módulos

Carregando módulos



Um módulo Haskell é uma coleção de funções, tipos e typeclasses. Um programa Haskell é uma coleção de módulos, onde o módulo principal carrega outros e usa suas funções para fazer algo de útil. Ter o código dividido em vários módulos tem várias vantagens. Se o módulo é genérico o bastante, quando exportadas, suas funções podem ser úteis a um bando de softwares. Se o seu próprio código está separado em módulos que não requerem uns aos outros (muito), eles podem ser utilizáveis em outros projetos. É uma grande vantagem perder algum tempo separando seu código em algumas partes com propósitos adequados.

A biblioteca padrão Haskell é dividida em módulos, que contém tipos e funções semelhantes ou que têm um uso semelhante. Existe um módulo para manipulação de listas, um módulo para programação concorrente, um módulo para lidar com números complexos, etc. Todas as funções, tipos e typeclasses que trabalhamos até agora fazem parte do módulo **`Prelude`**, que já é importado automaticamente. Nesse capítulo, daremos uma olhada em alguns módulos úteis e suas funções. Mas primeiro, como importar módulos.

A sintaxe de importação módulos em um script Haskell é **`import <nome do módulo>`**. Isso deve ser feito antes de definir qualquer função, por isso importações são feitas geralmente no início do arquivo. Obviamente um script pode importar vários módulos, só é preciso colocar o código apropriado em linhas distintas. Então vamos importar o módulo **`Data.List`** que possui várias funções úteis para se trabalhar com listas, inclusive uma que servirá para nos dizer quantos elementos únicos existem numa dada lista.

```
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

Quando você dá um **`import Data.List`**, todas as funções de **`import Data.List`** tornam-se disponíveis no namespace global, o que significa que a partir daí podemos usar qualquer uma delas no momento que quisermos. **`nub`** é uma das funções definidas em **`Data.List`** que recebe uma lista e retira todos os elementos repetidos. Misturar **`length`** e **`nub`** fazendo **`length . nub`** resulta em uma função equivalente a **`\xs -> length (nub xs)`**.

Você também pode colocar funções de módulos no global namespace no GHCI. Se estiver no GHCI e quiser chamar uma das funções de **`Data.List`**, você pode fazer:

```
ghci> :m + Data.List
```

Se o que você quer é carregar vários módulos dentro do GHCI, não é necessário digitar **`:m +`** milhares de vezes porque há a possibilidade de fazer o mesmo em apenas uma linha.

```
ghci> :m + Data.List Data.Map Data.Set
```

Contudo se já carregou um script que carrega módulos, não é necessário usar **`:m +`** novamente.

Se seu interesse é em apenas meia dúzia de funções, você pode importar apenas elas. Para carregar somente a **`nub`** e a **`sort`** do **`Data.List`**, fazemos assim:

```
import Data.List (nub, sort)
```

Você ainda pode importar - com exceções - todas as funções de um módulo. Isso é útil quando módulos exportam funções de mesmo nome e você não quer sofrer com conflitos. Já que já falamos da função **`nub`**, vamos importar todas as funções do **`Data.List`** menos ela:

```
import Data.List hiding (nub)
```

Outro meio de lidar com conflitos de nome é qualificar importações. O módulo **`Data.Map`** que provê formas de pesquisar valores por chaves em estruturas de dados, exporta várias funções de nomes repetidos como **`Prelude`**, **`filter`** e **`null`**. Então quando importamos **`Data.Map`** e chamamos **`filter`**, Haskell não sabe qual função você quer. E é assim que resolvemos esse problema:

```
import qualified Data.Map
```

Isso quer dizer que quando quisermos referenciar a **`filter`** do **`Data.Map`**, temos que colocar **`Data.Map.filter`**, e que **`filter`** continua sendo nosso **`filter`** já conhecido e amado. O problema é que digitar **`Data.Map`** na frente de cada função que pode dar problema é um pouco trabalhoso. Por isso que também podemos renomear importações qualificadas para algo mais curto:

```
import qualified Data.Map as M
```

Agora para referenciar a **`filter`** do **`Data.Map`** fica apenas **`M.filter`**.

Leia essa ótima referência para ver quais módulos vêm na biblioteca padrão. Um bom modo de aprender algo novo de Haskell é mergulhar na biblioteca padrão para descobrir módulos e funções carregadas automaticamente. Você ainda pode ler seus códigos-fonte. Esse também é um dos melhores jeitos de aprender e adquirir um conhecimento sólido em Haskell.

Para pesquisar sobre e ver a localização de funções, use o Hoogle. Nesse incrível sistema de busca de Haskell, você pode pesquisar por nome, módulo ou até mesmo declarações de tipo.

Data.List

O módulo **`Data.List`** obviamente lida com listas, provendo funções muito úteis que inclusive já conhecemos algumas (como **`map`** e **`filter`**) por **`Prelude`** já importar algo por conveniência. Você não precisará importar **`Data.List`** via importação qualificada porque não conflita com nada do **`Prelude`** (exceto os já roubados automaticamente). Vamos dar uma olhada em algumas que ainda não conhecemos.

`intersperse` recebe um elemento e uma lista e o repete entre cada um dos elementos da lista. Veja só:

```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

`intercalate` pega uma lista de listas e uma lista simples e a insere entre cada uma, retornando apenas uma lista com o resultado.

```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

`transpose` transpõe uma lista de listas. Vendo uma lista de listas como uma matriz, as colunas viram linhas e vice-versa.

```
ghci> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
ghci> transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
```

Supondo que queremos somar os polinômios $3x^2 + 5x + 9$, $10x^3 + 9$ e $8x^3 + 5x^2 + x - 1$. Podemos representá-los por meio das listas **`[0,3,5,9]`**, **`[10,0,0,9]`** e **`[8,5,1,-1]`**. Agora para somar, precisamos fazer só o seguinte:

```
ghci> map sum $ transpose [[0,3,5,9],[10,0,0,9],[8,5,1,-1]]
[18,8,6,17]
```

Quando transpomos essas três listas, as potências de 3 estão na primeira linha, as de 2 na segunda e assim por diante. Mapear **`sum`** desses valores produz o resultado esperado.

`foldl'` e `foldl1'` são versões mais trabalhadoras do que suas respectivas versões preguiçosas. Ao usar folds preguiçosas em listas grandes, você pode se deparar com um estouro de pilha. O culpado disso é exatamente a natureza preguiçosa das funções, que ao invés de realmente realizar a operação com o acumulado, somente diz que quando for pedido um resultado, ele irá executar mais essa outra conta. O problema é quando todo esse bando de promessas ultrapassa o limite da memória. Os folds severos não são tão passíveis de bugs por já realizar as operações intermediárias ao invés de encher a pilha de falsas promessas. Então se você tiver esse tipo de problema, tente usar suas versões trabalhadoras.

`concat` concatena uma lista de listas em apenas uma lista.

```
ghci> concat ["foo","bar","car"]
"foobarcar"
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

Somente removerá um nível de camadas. Então se você quer simplificar `[[[2,3],[3,4,5],[2]],[[2,3],[3,4]]]` - uma lista de listas, precisa concatenar duas vezes.

`concatMap` é o mesmo que mapear e depois concatenar uma lista.

```
ghci> concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]
```

`and` recebe uma lista de valores booleanos e retorna **`True`** se todos forem **`True`**.

```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

`or` é relativamente parecido com o **`and`**, mas retorna **`True`** se no mínimo um dos valores for **`True`**.

```
ghci> or $ map (==4) [2,3,4,5,6,1]
True
ghci> or $ map (>4) [1,2,3]
False
```

`any` e `all` verificam se um predicado é verdadeiro para no mínimo um ou todos os elementos da lista (respectivamente). Geralmente usamos uma delas ao invés de mapear e depois usar **`and`** ou **`or`**.

```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
ghci> all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
ghci> any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
```

`iterate` toma uma função e um valor inicial. Aplica a função ao valor, depois aplica a função ao seu resultado, então ao seu resultado novamente... Retorna os resultados na forma de uma lista infinita.

```
ghci> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
ghci> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

`splitAt` recebe um número e uma lista. Então separa a lista em elementos, retornando uma tupla.

```
ghci> splitAt 3 "heyman"
("hey","man")
ghci> splitAt 100 "heyman"
("heyman","")
ghci> splitAt (-3) "heyman"
("","heyman")
ghci> let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
```

`takeWhile` é uma função simples. Vai pegando os elementos da lista enquanto a verificação resulta satisfatoriamente. Pode ser bastante útil.

```
ghci> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
ghci> takeWhile (/=' ') "This is a sentence"
"This"
```

Digamos que queira saber a soma de todas as potências menores de 10000. Não podemos mapear **`(^3)`**, aplicar um filtro e tentar somar já que filtros em listas infinitas nunca terminam. Você pode saber que os elementos são ascendentes, mas Haskell não. Por isso que podemos fazer isso:

```
ghci> sum $ takeWhile (<10000) $ map (^3) [1..]
53361
```

Aplicamos **`(^3)`** em uma lista infinita e, logo que o elemento 10000 é encontrado, a lista é cortada. Então somamos facilmente.

`dropWhile` é semelhante, com a diferença dele descartar todos os elementos com resultado verdadeiro. Quando **`False`** é encontrado, retorna o resto da lista. Uma função fascinante.

```
ghci> dropWhile (/=' ') "This is a sentence"
" is a sentence"
ghci> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
```

Nos entregaram uma lista que representa os valores de estoque por datas. A lista é feita por tuplas onde o primeiro componente é o estoque, o segundo é o ano, o terceiro é o mês e o quarto é o dia. Queremos saber quando o estoque excederá os 1000 dólares pela primeira vez.

```
ghci> let stock = [(994.4,2008,9,1),(995.2,2008,9,2),(999.2,2008,9,3),(1001.4,2008,9,4),
(998.3,2008,9,5)]
ghci> head (dropWhile (\(val,y,m,d) -> val < 1000) stock)
(1001.4,2008,9,4)
```

`span` é algo como o **`takeWhile`** mas que retorna duas listas. A primeira lista contém tudo que o **`takeWhile`** retornaria com o mesmo teste. A segunda é a lista que seria descartada.

```
ghci> let (fw, rest) = span (/=' ') "This is a sentence" in "First word:" ++ fw ++ ", the rest
"First word: This, the rest: is a sentence"
```

Enquanto que **`span`** toma os elementos enquanto não aparecer o primeiro true como resultado do teste, `break` para ao perceber que a função retornou true. Fazer **`break p`** é o equivalente a **`span (not . p)`**.

```
ghci> break (==4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
ghci> span (/=4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

Ao usar o **`break`**, a segunda lista terá o primeiro elemento como o que satisfaz o teste.

`sort` simplesmente ordena uma lista. O tipo dos elementos da lista deve ser parte da typeclass **`Ord`**, já que como seus elementos não podem ser colocados em uma ordem, obviamente não podem ser ordenados.

```
ghci> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
ghci> sort "This will be sorted soon"
"    Tbdeehiillnooorssstw"
```

`group` recebe uma lista e agrupa em novas listas os elementos iguais.

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

Se ordenarmos e depois agruparmos uma lista, podemos descobrir quantas vezes cada elemento aparece.

```
ghci> map (\l@(x:xs) -> (x,length l)) . group . sort $ [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
```

`inits` e `tails` são semelhantes a **`init`** e **`tail`**, com a diferença que as primeiras vão aplicando recursivamente o método até o fim da lista. Confira:

```
ghci> inits "w00t"
["","w","w0","w00","w00t"]
ghci> tails "w00t"
["w00t","00t","0t","t",""]
ghci> let w = "w00t" in zip (inits w) (tails w)
[("","w00t"),("w","00t"),("w0","0t"),("w00","t"),("w00t","")]
```

Vamos usar fold para implementar uma pesquisa por sublistas em listas.

```
search :: (Eq a) => [a] -> [a] -> Bool
search needle haystack =
    let nlen = length needle
    in  foldl (\acc x -
> if take nlen x == needle then True else acc) False (tails haystack)
```

Primeiro chamamos **`tails`** com a lista que procuramos. Então compara-se cada "cauda" com a lista.

Com isso, criamos algo com o funcionamento da função `isInfixOf`. `isInfixOf` procura por uma sublista em uma lista e retorna **`True`** se for encontrada em algum lugar.

```
ghci> "cat" `isInfixOf` "im a cat burglar"
True
ghci> "Cat" `isInfixOf` "im a cat burglar"
False
ghci> "cats" `isInfixOf` "im a cat burglar"
False
```

`isPrefixOf` e `isSuffixOf` procuram por sublistas no fim e início de uma lista.

```
ghci> "hey" `isPrefixOf` "hey there!"
True
ghci> "hey" `isPrefixOf` "oh hey there!"
False
ghci> "there!" `isSuffixOf` "oh hey there!"
True
ghci> "there!" `isSuffixOf` "oh hey there"
False
```

`elem` e `notElem` verificam se um elemento está ou não dentro de uma lista.

`partition` recebe uma lista e um predicado e retorna um par de listas. A primeira lista resultante contém todos os elementos que satisfazem o predicado, a segunda que não.

```
ghci> partition (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOBMORGAN","sidneyeddy")
ghci> partition (>3) [1,3,5,6,3,2,1,0,3,7]
([5,6,7],[1,3,3,2,1,0,3])
```

É importante saber diferenciar de **`span`** e **`break`**:

```
ghci> span (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOB","sidneyMORGANeddy")
```

Enquanto **`span`** e **`break`** terminam ao encontrar o primeiro elemento que não satisfaz o predicado, **`partition`** passa por toda a lista e separa ao encontrar o predicado.

`find` recebe uma lista e um predicado e retorna o primeiro elemento que satisfaz o predicado (dentro de um valor **`Maybe`**). Nós iremos ver tipos de dados algébricos melhor no próximo capítulo, mas por enquanto: um valor **`Maybe`** (possivelmente) pode ser **`Just something`** (apenas alguma coisa) ou **`Nothing`** (nada). Do mesmo jeito que uma lista pode ter nenhum ou vários elementos, um valor **`Maybe`** pode ter nada ou um valor. Do mesmo jeito que uma lista de números inteiro é **`[Int]`**, o tipo que possivelmente tem um inteiro é **`Maybe Int`**. Bom, testemos o **`find`**.

```
ghci> find (>4) [1,2,3,4,5,6]
Just 5
ghci> find (>9) [1,2,3,4,5,6]
Nothing
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

Veja o tipo de **`find`**. É **`Maybe a`**. Isso é muito parecido com ser **`[a]`**, mas o **`Maybe`** pode conter um ou zero elementos, enquanto uma lista pode conter um, zero ou vários elementos.

Lembra-se de quando procuramos a primeira vez que o estoque ultrapassou $1000. Fizemos **`head (dropWhile (\(val,y,m,d) -> val < 1000) stock)`**. Lembre-se que **`head`** não é muito seguro. O que acontece se nosso estoque nunca chegou a $1000? Nosso uso da **`dropWhile`** retornaria uma lista vazia e ao pegarmos o primeiro elemento de nada seria gerado um erro. No entanto, se reescrevêssemos para **`find (\(val,y,m,d) -> val > 1000) stock`**, seria muito mais seguro. Se o estoque nunca chegou a $1000 (que nenhum elemento satisfez o predicado), seria retornado **`Nothing`**. Mas uma resposta válida dessa lista seria, digamos, **`Just (1001.4,2008,9,4)`**.

`elemIndex` é semelhante ao **`elem`**, mas que não retorna um valor booleano. Provavelmente retornará a posição do elemento. Caso o elemento não exista na lista, **`Nothing`**.

```
ghci> :t elemIndex
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
ghci> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
ghci> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
```

`elemIndices` é semelhante ao **`elemIndex`**, mas retorna uma lista de posições, no caso do nosso elemento aparecer várias vezes. Como estamos usando uma lista para representar os índices, não precisamos de um tipo **`Maybe`**, já que o erro pode ser representado por uma lista vazia, muito semelhante ao **`Nothing`**.

```
ghci> ' ' `elemIndices` "Where are the spaces?"
[5,9,13]
```

`findIndex` é parecido com o find, mas retorna apenas a posição do primeiro elemento que satisfaz o predicado. `findIndices` retorna os índices de todos os elementos que satisfaz o predicado na forma de uma lista.

```
ghci> findIndex (==4) [5,3,2,1,6,4]
Just 5
ghci> findIndex (==7) [5,3,2,1,6,4]
Nothing
ghci> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0,6,10,14]
```

Já vimos **`zip`** e **`zipWith`**. Elas combinam ("transpõem") duas listas em tuplas ou por uma função binária (o que significa precisar de dois parâmetros). Mas e se quisermos combinar três listas? Ou combinar três listas com uma função de três parâmetros? Para isso temos as `zip3`, `zip4`, etc. e `zipWith3`, `zipWith4`, etc. Variações que vão até 7. Pode parecer uma gambiarra, mas funciona muito bem, além de não ser sempre que é preciso transpor 8 listas. Ainda existe um modo muito prático de combinar listas infinitas, mas não avançamos o suficiente para ver essa parte.

```
ghci> zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,2,2] [2,2,3]
[7,9,8]
ghci> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
[(2,2,5,2),(3,2,5,2),(3,2,3,2)]
```

Igualmente à transposição comum, listas maiores que a menor são cortadas nessa quantidade de elementos.

`lines` é uma função muito útil ao lidar com arquivos ou qualquer outro tipo de entrada. Recebe uma string e retorna cada linha da string como uma lista.

```
ghci> lines "first line\nsecond line\nthird line"
["first line","second line","third line"]
```

`'\n'` é o caractere para uma nova linha Unix. Barras invertidas sempre têm um significado especial em strings e caracteres em Haskell.

`unlines` é o inverso da função **`lines`**. Recebe uma lista de strings e as une pelo caractere **`'\n'`**.

```
ghci> unlines ["first line", "second line", "third line"]
"first line\nsecond line\nthird line\n"
```

`words` e `unwords` são para separar uma linha de texto em palavras ou unir na ordem contrária. Muito útil.

```
ghci> words "hey these are the words in this sentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> words "hey these           are    the words in this\nsentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> unwords ["hey","there","mate"]
"hey there mate"
```

Já mencionamos a função `nub`. Recebe uma lista e retira os elementos duplicados, retornando uma lista que temos a certeza dos elementos não possuírem nenhum gêmeo perdido. Mas essa função tem um nome bem curioso. A expressão "nub" significa a parte essencial ou prioritária de algo. Na minha opinião, nomes de funções deveriam ter nomes mais parecidos com o seu real significado ao invés de se prender a antigas definições.

```
ghci> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
[1,2,3,4]
ghci> nub "Lots of words and stuff"
"Lots fwrdanu"
```

`delete` recebe um elemento e uma lista e remove sua primeira ocorrência.

```
ghci> delete 'h' "hey there ghang!"
"ey there ghang!"
ghci> delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere ghang!"
ghci> delete 'h' . delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere gang!"
```

`\\` é uma função de diferença de listas. Resumindo, traz a diferença. Para cada elemento da lista 2, remove-se o correspondente (se existente) na 1.

```
ghci> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
ghci> "Im a big baby" \\ "big"
```

```
"Im a  baby"
```

Fazer `[1..10] \\ [2,5,9]` é o mesmo que `delete 2 . delete 5 . delete 9 $ [1..10]`.

`union` também faz uma operação de conjuntos. Retorna a união de duas listas. Funciona colocando cada elemento da lista 2 na 1, se não existente.

```
ghci> "hey man" `union` "man what's up"
"hey manwt'sup"
ghci> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
```

`intersect` significa intersecção. Retorna apenas os elementos encontrados em ambas listas.

```
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

`insert` recebe um elemento e uma lista de elementos que podem ser ordenados. Então, o insere logo depois que o menor ou igual mais próximo. Se usarmos **`insert`** em uma lista ordenada, o resultado permanecerá ordenado.

```
ghci> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
ghci> insert 'g' $ ['a'..'f'] ++ ['h'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> insert 3 [1,2,4,3,2,1]
[1,2,3,4,3,2,1]
```

`length`, **`take`**, **`drop`**, **`splitAt`**, **`!!`** e **`replicate`** têm em comum receberem um **`Int`** como parâmetro, mesmo sabendo que o código ficaria mais genérico aceitando todos tipos de **`Integral`** ou **`Num`** (dependendo de cada função). Isso é feito por razões de compatibilidade. Mudar isso com certeza quebraria muito código antigo. Por isso que **`Data.List`** implementou versões suas mais genéricas: `genericLength`, `genericTake`, `genericDrop`, `genericSplitAt`, `genericIndex` e `genericReplicate`. **`length`** tem tipo **`length :: [a] -> Int`**. Se você tentar descobrir a média de números com **`let xs = [1..6] in sum xs / length xs`** terá um erro, já que você não pode usar **`/`** com **`Int`**. **`genericLength`** por outro lado, tem tipo **`genericLength :: (Num a) => [b] -> a`**. Como **`Num`** pode agir como ponto flutuante, você pode conseguir o resultado esperado com **`let xs = [1..6] in sum xs / genericLength xs`**.

`nub`, **`delete`**, **`union`**, **`intersect`** e **`group`** têm também versões genéricas com nomes `nubBy`, `deleteBy`, `unionBy`, `intersectBy` e `groupBy`. A diferença é que as primeiras versões usam **`==`** para testar por igualdade, enquanto suas versões *By* requerem também uma função para comparação. **`group`** é o mesmo que **`groupBy (==)`**.

Por hora, digamos que temos uma lista que descreve os valores de uma segunda e nós queremos dividir em sublistas baseado se o elemento está abaixo ou acima de zero. Se usássemos o **`group`** comum, apenas agruparia valores adjacentes. Mas queremos classificar quanto à posição de zero. É aí que **`groupBy`** mostra sua utilidade! A função de igualdade das funções *By* precisa receber dois elementos de mesmo tipo e retornar **`True`**, independente dos seus critérios para isso.

```
ghci> let values = [-4.3, -2.4, -1.2, 0.4, 2.3, 5.9, 10.5, 29.1, 5.3, -2.4, -14.5, 2.9, 2.3]
ghci> groupBy (\x y -> (x > 0) == (y > 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

Logo identificamos como fazer nosso teste. A função de igualdade retornará **`True`** somente se ambos forem negativos ou positivos. Essa função de igualdade pode ser escrita por **`\x y -> (x > 0) && (y > 0) || (x <= 0) && (y <= 0)`**, apesar de achar que do primeiro jeito ficou muito mais legível. Um modo muito melhor de escrever funções *By* é importando a função `on` de **`Data.Function`**. **`on`** é definida assim:

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
f `on` g = \x y -> f (g x) (g y)
```

Fazendo **`(==) `on` (> 0)`** retorna uma função de igualdade semelhante a **`\x y -> (x > 0) == (y > 0)`**. **`on`** é muito usada por funções *By*, já que é possível fazer isso:

```
ghci> groupBy ((==) `on` (> 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

Muito mais legível! Agora você pode ler em voz alta: Agrupe por igualdade sempre que elementos forem maiores do que zero.

Semelhantemente, **`sort`**, **`insert`**, **`maximum`** e **`minimum`** também têm versões genéricas. Funções como **`groupBy`** recebem uma função para determinar quando dois elementos são iguais. `sortBy`, `insertBy`, `maximumBy` e `minimumBy` recebem uma função que diz se um elemento é maior, menor ou igual a outro. O tipo de **`sortBy`** e **`sortBy :: (a -> a -> Ordering) -> [a] -> [a]`**. Se você ainda se lembra, o tipo **`Ordering`** pode assumir valor **`LT`**, **`EQ`** ou **`GT`**. **`sort`** é o equivalente a **`sortBy compare`**, porque *compare* pega dois tipos de **`Ord`** e retorna sua relação quanto ordenação.

Listas podem ser comparadas, mas quando o são, são comparados lexicograficamente. E se quisermos ordenar uma lista de listas não baseado nas listas internas, mas em seus tamanhos? Como deve ter imaginado, **`sortBy`** é a nossa solução.

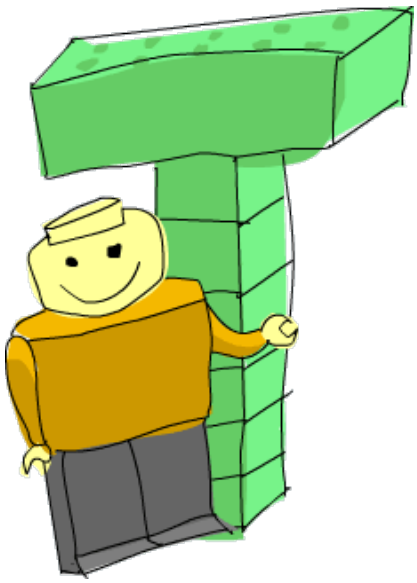
```
ghci> let xs = [[5,4,5,4,4],[1,2,3],[3,5,4,3],[],[2],[2,2]]
ghci> sortBy (compare `on` length) xs
[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]
```

Incrível! **`compare `on` length`** ... cara, isso mais parece inglês puro! Se você não percebeu bem como **`on`** funciona aqui, **`compare `on` length`** é o equivalente a **`\x y -> length x `compare` length y`**. Quando você está trabalhando com funções *By* que precisam de uma função de igualdade, você geralmente vai usar **`(==) `on` something`** e com funções *By* que precisem de outra de ordenação, você usa **`compare `on` something`**.

Data.Char

O módulo **`Data.Char`** é exatamente o que o seu nome sugere. Exporta funções que lidam com caracteres. É extremamente útil para filtrar ou mapear strings, que nada mais são do que listas de caracteres.

`Data.Char` também exporta vários predicados de caracteres. Isto é, funções que recebem um caracter e nos diz se uma proposição é verdadeira ou falsa. Alguns exemplos:



`isControl` testa se um caracter é um caracter de controle.

`isSpace` testa se um caracter é um espaço em branco, o que inclui espaços, tabs, novas linhas, etc.

`isLower` testa se um caracter é minúsculo.

`isUpper` testa se um caracter é maiúsculo.

`isAlpha` testa se um caracter é uma letra.

`isAlphaNum` testa se um caracter é uma letra ou um número.

`isPrint` testa se um caracter é imprimível. Caracteres de controle, por exemplo, não são.

`isDigit` testa se um caracter é um dígito.

`isOctDigit` testa se um caracter é um dígito octal.

`isHexDigit` testa se um caracter é um dígito hexadecimal.

`isLetter` testa se um caracter é uma letra.

`isMark` testa se um caracter é um caracter de marcação Unicode, que quando combinados com letras precedentes, formam letras com acentos. Use se for francês.

`isNumber` testa se um caracter é numérico.

`isPunctuation` testa se um caracter é uma pontuação.

`isSymbol` testa se o caracter é simbolo, inclusive simbolos matemáticos ou de moedas.

`isSeparator` testa se um caracter é espaço ou separador Unicode.

`isAscii` testa se um caracter é um dos primeiros 128 Unicode.

`isLatin1` testa se um caracter é um dos primeiros 256 Unicode.

`isAsciiUpper` testa se um caracter é ASCII e maiúsculo.

`isAsciiLower` testa se um caracter é ASCII e minúsculo.

Todos esses predicados têm tipos **`Char -> Bool`**. Na maior parte das vezes eles serão usados para filtrar algo de strings. Por exemplo, digamos que estamos fazendo um programa com login que necessita que o usuário seja composto apenas por caracteres alfanuméricos. Poderíamos usar a função **`all`** do **`Data.List`** em combinção com predicados do **`Data.Char`** para testar sua validade.

```
ghci> all isAlphaNum "bobby283"
True
ghci> all isAlphaNum "eddy the fish!"
False
```

Que bacana! Caso não se lembre, **`all`** recebe um predicado e uma lista e retorna **`True`** apenas se o predicado aprova todos os elementos.

Podemos ainda usar a **`isSpace`** para simular o funcionamento da função **`words`** do **`Data.List`**.

```
ghci> words "hey guys its me"
["hey","guys","its","me"]
ghci> groupBy ((==) `on` isSpace) "hey guys its me"
["hey"," ","guys"," ","its"," ","me"]
ghci>
```

Hmmm, muito parecido mesmo, mas manteve os espaços em branco. O que poderíamos fazer? Eu sei, pelo menos. Vamos filtrar esse pessoal.

```
ghci> filter (not . any isSpace) . groupBy ((==) `on` isSpace) $ "hey guys its me"
["hey","guys","its","me"]
```

Ah.

A **`Data.Char`** ainda exporta o tipo de dados **`Ordering`**. O **`Ordering`** pode ter valor **`LT`**, **`EQ`** ou **`GT`**. É como uma enumeração. Descreve os possíveis resultados ao comparar dois elementos. O tipo **`GeneralCategory`** também é uma enumeração. Tem algumas possíveis categorias que um caracter pode se encaixar. A principal função de conseguir a categoria geral de caracteres é **`generalCategory`**. Tem tipo **`generalCategory :: Char -> GeneralCategory`**. Existem ainda outras 31 categorias que não listaremos aqui, mas vamos dar uma olhada na função.

```
ghci> generalCategory ' '
Space
ghci> generalCategory 'A'
UppercaseLetter
ghci> generalCategory 'a'
LowercaseLetter
ghci> generalCategory '.'
OtherPunctuation
ghci> generalCategory '9'
DecimalNumber
ghci> map generalCategory " \t\nA9?|"
[Space,Control,Control,UppercaseLetter,DecimalNumber,OtherPunctuation,MathSymbol]
```

Já que o tipo **`GeneralCategory`** é parte da typeclass **`Eq`**, poderíamos testar algo como **`generalCategory c == Space`**.

`toUpper` converte um caracter para maiúsculo. Espaços, números e outros permanecem iguais.

`toLower` converte um caracter para minúsculo.

`toTitle` converte um caracter para title-case. Para a maioria dos caracteres, titlecase é o mesmo que maiúsculas.

`digitToInt` converte um caracter para **`Int`**. Para que funcione, o caracter deve estar nas ranges **`'0'..'9'`**, **`'a'..'f'`** ou **`'A'..'F'`**.

```
ghci> map digitToInt "34538"
[3,4,5,3,8]
ghci> map digitToInt "FF85AB"
[15,15,8,5,10,11]
```

`intToDigit` é o inverso da função **`digitToInt`**. Recebe um **`Int`** na range **`0..15`** e converte para um caracter minúsculo.

```
ghci> intToDigit 15
'f'
ghci> intToDigit 5
'5'
```

As funções `ord` e **`chr`** convertem caracteres para seus correspondentes em números e vice-versa:

```
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```

Os valores **`ord`** de dois caracteres é referente à tabela Unicode. O Código de César é um método primitivo de codificar mensagens ao associar caracteres por um correspondente (seguindo a mesma sequência). Nós mesmos poderíamos construir nosso código criptografado sem precisar debruçar sobre um alfabeto.

```
encode :: Int -> String -> String
encode shift msg =
    let ords = map ord msg
        shifted = map (+ shift) ords
    in  map chr shifted
```

Aqui, primeiro convertemos nossa string para uma lista de números. Então aumentamos ou diminuimos uma certa quantidade de inteiros de toda a lista e convertemos de volta para caracteres. Se você já está craque na criação de funções, já escreveu **`map (chr . (+ shift) . ord) msg`**. Vamos tentar criptografar algumas mensagens.

```
ghci> encode 3 "Heeeeey"
"Khhhhh|"
ghci> encode 4 "Heeeeey"
"Liiiii}"
ghci> encode 1 "abcd"
"bcde"
ghci> encode 5 "Marry Christmas! Ho ho ho!"
```

```
"Rfww~%Hmwnxyrfx&%Mt%mt%mt&"
```

Parece que está certo. Para decodificar a mensagem é só necessário voltar o número de posições em todos caracteres, trazendo-os à posição inicial.

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
```

```
ghci> encode 3 "Im a little teapot"
"Lp#d#olwwoh#whdsrw"
ghci> decode 3 "Lp#d#olwwoh#whdsrw"
"Im a little teapot"
ghci> decode 5 . encode 5 $ "This is a sentence"
"This is a sentence"
```

Data.Map

Listas de associação (também chamadas de dicionários) são listas usadas para armazenar pares de chave e valor, onde sua ordem não é considerada. Por exemplo, podemos usar uma lista de associação para guardar números de telefone (que seriam os valores) junto do nome dos donos (chaves). A ordem que esses dados estão armazenados não importa para nós, desde que consigamos o número certo ao pesquisar por uma pessoa.

O modo mais óbvio de armazenar listas de associação em Haskell seria uma lista de pares. Onde o primeiro componente do par deve ser a chave, e o segundo componente o valor. Aqui vai um exemplo de uma lista de associação com números de telefone:

```
phoneBook =
    [("mari","555-2938")
    ,("carol","452-2928")
    ,("patrícia","493-2928")
    ,("luciana","205-2928")
    ,("aline","939-8282")
    ,("camila","853-2492")
    ]
```

Tirando a identação bizarra, isso é apenas uma lista de pares de strings. O objetivo mais comum ao lidar com listas de associação é a busca por chave. Vamos criar uma função que procura por uma dada chave.

```
findKey :: (Eq k) => k -> [(k,v)] -> v
findKey key xs = snd . head . filter (\(k,v) -> key == k) $ xs
```

Extremamente simples. A função recebe uma chave e uma lista, filtra a lista para que permaneçam apenas os que têm chave correspondente e retorna o primeiro par chave-valor. Mas o que acontece ao procurarmos por uma chave inexistente? Bem, nós tentaríamos pegar o primeiro elemento de uma lista vazia, gerando um *runtime error*. Mas precisamos fazer os nossos programas para que sejam um pouco mais difíceis de quebrar, então usaremos o tipo **`Maybe`**. Se não encontrar uma chave, retornará **`Nothing`**. Se encontrar, retornará **`Just alguma-coisa`**, onde "alguma-coisa" é o valor correspondente à chave.

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key [] = Nothing
findKey key ((k,v):xs) = if key == k
```

```
        then Just v
        else findKey key xs
```

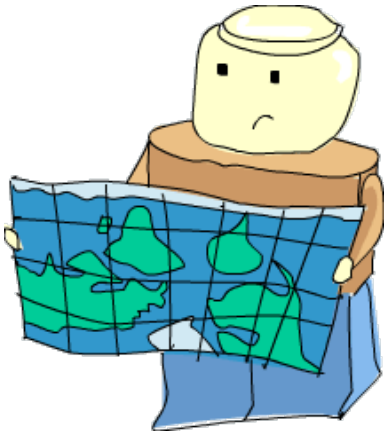
Preste atenção à declaração de tipo. Ela recebe uma chave que pode ser igualada, uma lista de associação e talvez devolva um valor. Parece correto.

Essa é uma função de agenda de contatos que trabalha com uma lista. Temos a condição limite, quebramos a lista em uma cabeça e uma cauda, temos chamadas recursivas, esta tudo ali. Esse é o caso clássico de uso do padrão de folding. Vamos tentar implementar então com folds.

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key = foldr (\(k,v) acc -> if key == k then Just v else acc) Nothing
```

Dica: Geralmente é melhor usar folds para listas de padrões recursivos ao invés de escrever explicitamente a recursão porque é mais fácil de se ler. Todos sabem o que é um fold ao ver a chamada à função **`foldr`**, mas toma mais tempo para entender todo seu processo de recursão.

```
ghci> findKey "camila" phoneBook
Just "853-2492"
ghci> findKey "mari" phoneBook
Just "555-2938"
ghci> findKey "wilma" phoneBook
Nothing
```



Funciona que é uma beleza! Agora só falta você usar sua nova agenda para ver se não fica só na pesquisa.

Nós acabamos de implementar a função **`lookup`** do **`Data.List`**. Se quisermos achar um valor correspondente a uma chave, precisamos percorrer todos os elementos até encontrá-lo. O módulo **`Data.Map`** oferece listas de associação muito mais rápidas (porque elas são implementadas internamente com árvores) e ainda provê algumas funções úteis. De agora em diante usaremos mapas ao invés de listas de associação.

Já que **`Data.Map`** exporta funções que conflitam com outras de **`Prelude`** e **`Data.List`**, faremos uma importação qualificada.

```
import qualified Data.Map as Map
```

Coloque esse comando para importação no script e carregue-o via GHCI.

Chega de enrolar e vamos ver o que o **`Data.Map`** nos guarda! Aí vai um resumo rápido de suas funções.

A função `fromList` recebe uma lista de associações (na forma de uma lista mesmo) e retorna um mapa com a mesma estrutura.

```
ghci> Map.fromList [("mari","555-2938"),("carol","452-2928"),("luciana","205-2928")]
fromList [("mari","555-2938"),("carol","452-2928"),("luciana","205-2928")]
ghci> Map.fromList [(1,2),(3,4),(3,2),(5,5)]
fromList [(1,2),(3,2),(5,5)]
```

Se existirem chaves duplicadas na lista de associação original, as duplicadas são descartadas. Essa é a assinatura de tipo de **`fromList`**.

```
Map.fromList :: (Ord k) => [(k, v)] -> Map.Map k v
```

Ela diz que *fromList* recebe uma lista de pares de tipos **`k`** e **`v`** e retorna um mapa com chaves de tipo **`k`** e tipo **`v`**. Perceba que quando você está criando listas de associação com listas, as chaves só precisam ser possíveis de serem igualadas (terem tipos pertencentes à typeclass **`Eq`**) mas não necessariamente ordenáveis. Essa é a essência do módulo **`Data.Map`**. Ele precisa que as chaves sejam ordenáveis para organizá-las na forma de uma árvore.

É sempre aconselhável usar o **`Data.Map`** para associações chave-valor, a menos que o tipo não esteja na typeclass **`Ord`**.

`empty` representa um mapa vazio. Não necessita de nenhum argumento, só retorna um mapa vazio.

```
ghci> Map.empty
fromList []
```

`insert` recebe uma chave, um valor e um mapa e retorna um novo mapa igual ao antigo, mas com chaves e valor adicionados.

```
ghci> Map.empty
fromList []
ghci> Map.insert 3 100 Map.empty
fromList [(3,100)]
ghci> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100  Map.empty))
fromList [(3,100),(4,200),(5,600)]
ghci> Map.insert 5 600 . Map.insert 4 200 . Map.insert 3 100 $ Map.empty
fromList [(3,100),(4,200),(5,600)]
```

Nós podemos implementar nosso próprio **`fromList`** usando um mapa vazio, **`insert`** e um fold. Veja só:

```
fromList' :: (Ord k) => [(k,v)] -> Map.Map k v
fromList' = foldr (\(k,v) acc -> Map.insert k v acc) Map.empty
```

Um fold bastante simples. Começamos com um mapa vazio e dobramos à direita, inserindo os pares de chave-valor no acumulador em cada passo.

`null` testa se um mapa está vazio.

```
ghci> Map.null Map.empty
True
ghci> Map.null $ Map.fromList [(2,3),(5,5)]
False
```

`size` reporta o tamanho de um mapa.

```
ghci> Map.size Map.empty
0
ghci> Map.size $ Map.fromList [(2,4),(3,3),(4,2),(5,4),(6,4)]
5
```

`singleton` recebe uma chave e um valor e cria um mapa com apenas esses dados.

```
ghci> Map.singleton 3 9
fromList [(3,9)]
ghci> Map.insert 5 9 $ Map.singleton 3 9
fromList [(3,9),(5,9)]
```

`lookup` funciona como o **`Data.List lookup`**, mas opera apenas em mapas. Retorna **`Just something`** se encontrar algo pela chave procurada, **`Nothing`** se encontrar nada.

`member` é um predicado que recebe uma chave e um mapa e diz se a chave existe ou não no mapa.

```
ghci> Map.member 3 $ Map.fromList [(3,6),(4,3),(6,9)]
True
ghci> Map.member 3 $ Map.fromList [(2,5),(4,5)]
False
```

`map` e `filter` funcionam semelhantemente aos homônimos das listas.

```
ghci> Map.map (*100) $ Map.fromList [(1,1),(2,4),(3,9)]
fromList [(1,100),(2,400),(3,900)]
ghci> Map.filter isUpper $ Map.fromList [(1,'a'),(2,'A'),(3,'b'),(4,'B')]
fromList [(2,'A'),(4,'B')]
```

`toList` é o inverso da **`fromList`**.

```
ghci> Map.toList . Map.insert 9 2 $ Map.singleton 4 3
[(4,3),(9,2)]
```

`keys` e `elems` retornam listas de chaves e valores, respectivamente. **`keys`** é o equivalente a **`map fst . Map.toList`** e **`elems`** é o de **`map snd . Map.toList`**.

`fromListWith` é uma função muito legal. Age como **`fromList`**, mas ao invés de discartar as chaves duplicadas, recebe também uma função que decide o que fazer com elas. Imagine que uma garota pode ter vários números e temos uma lista de associação como essa:

```
phoneBook =
    [("mari","555-2938")
    ,("mari","342-2492")
    ,("carol","452-2928")
    ,("patrícia","493-2928")
    ,("patrícia","943-2929")
    ,("patrícia","827-9162")
    ,("luciana","205-2928")
```

```
      ,("aline","939-8282")
      ,("camila","853-2492")
      ,("camila","555-2111")
      ]
```

Agora se quisermos usar **`fromList`** para criar um mapa, perderemos alguns números! Então é assim que vamos fazer:

```
phoneBookToMap :: (Ord k) => [(k, String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith (\number1 number2 -> number1 ++ ", " ++ number2) xs
```

```
ghci> Map.lookup "patrícia" $ phoneBookToMap phoneBook
"827-9162, 943-2929, 493-2928"
ghci> Map.lookup "aline" $ phoneBookToMap phoneBook
"939-8282"
ghci> Map.lookup "mari" $ phoneBookToMap phoneBook
"342-2492, 555-2938"
```

Se uma chave duplicada for encontrada, a função que passamos será usada para combinar os valores das chaves em outro valor. Poderíamos primeiro ter listas únicas para os valores e usarmos **++** para fazer a combinação.

```
phoneBookToMap :: (Ord k) => [(k, a)] -> Map.Map k [a]
phoneBookToMap xs = Map.fromListWith (++) $ map (\(k,v) -> (k,[v])) xs
```

```
ghci> Map.lookup "patrícia" $ phoneBookToMap phoneBook
["827-9162","943-2929","493-2928"]
```

Muito elegante! Outro caso de uso é quando criamos um mapa com base em uma lista de associação de números e quando encontramos uma chave duplicada, queremos o maior valor da chave para mante-lo.

```
ghci> Map.fromListWith max [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,100),(3,29),(4,22)]
```

Ou podemos optar por adicionar valores junto a mesma chave.

```
ghci> Map.fromListWith (+) [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,108),(3,62),(4,37)]
```

`insertWith` é para **`insert`** o que **`fromListWith`** é para **`fromList`**. Ele insere um par chave-valor em um mapa, porém se o mapa já tiver a chave, ele usa a função passada para determinar o que fazer.

```
ghci> Map.insertWith (+) 3 100 $ Map.fromList [(3,4),(5,103),(6,339)]
fromList [(3,104),(5,103),(6,339)]
```

Estas são apenas algumas funções do **`Data.Map`**. Você pode ainda ver uma lista completa de funções na documentação.

Data.Set

O módulo **`Data.Set`** nos oferece conjuntos. Conjuntos matemáticos, isso mesmo. Conjuntos são algo entre listas e mapas. Todos os elementos em um conjunto são únicos. E por serem internamente implementados por árvores (semelhantemente a mapas do **`Data.Map`**), são ordenados. Checar existência, inserção, deleção, etc. é muito mais rápido do que com listas. As operações mais comuns falando-se de conjuntos é inserir, checar existência e converter para lista.

Pelos nomes do **`Data.Set`** frequentemente conflitarem com membros de **`Prelude`** e **`Data.List`**, fazemos uma importação qualificada.

Coloque essa linha no seu script:

```
import qualified Data.Set as Set
```

E então carregue-o via GHCI.

Temos dois pedaços de um texto. Queremos descobrir quais caracteres são usados em ambos.

```
text1 = "Eu tive um sonho com animes. Anime... Realidade... Tem alguma diferenca?"
text2 = "O velhote deixou a lixeira dele la fora e agora o lixo dele esta todo espalhado no me
```

A função `fromList` faz exatamente o que você imagina. Recebe uma lista e converte-a em um conjunto.

```
ghci> let set1 = Set.fromList text1
ghci> let set2 = Set.fromList text2
ghci> set1
fromList " .?AERTacdefghilmnorstuv"
ghci> set2
fromList " !Oadefghilmnoprstuvx"
```

Como pode ver, os itens são ordenados e cada elemento são únicos. Usaremos `intersection` para descobrir quais estão em ambos.

```
ghci> Set.intersection set1 set2
fromList " adefghilmnorstuv"
```

Podemos ainda usar `difference` para ver quais letras estão no primeiro mas não no segundo conjunto e vice-versa.

```
ghci> Set.difference set1 set2
fromList ".?AERTc"
ghci> Set.difference set2 set1
fromList "!Opx"
```

Ou também podemos gerar um terceiro conjunto com as letras que aparecem em qualquer um dos dois primeiros usando `union`.

```
ghci> Set.union set1 set2
```

```
fromList " !.?AEORTacdefghilmnoprstuvx"
```

As funções `null`, `size`, `member`, `empty`, `singleton`, `insert` e `delete` você já deve imaginar para que servem.

```
ghci> Set.null Set.empty
True
ghci> Set.null $ Set.fromList [3,4,5,5,4,3]
False
ghci> Set.size $ Set.fromList [3,4,5,3,4,5]
3
ghci> Set.singleton 9
fromList [9]
ghci> Set.insert 4 $ Set.fromList [9,3,8,1]
fromList [1,3,4,8,9]
ghci> Set.insert 8 $ Set.fromList [5..10]
fromList [5,6,7,8,9,10]
ghci> Set.delete 4 $ Set.fromList [3,4,5,4,3,4,5]
fromList [3,5]
```

Nós ainda podemos procurar por subconjuntos ou superconjuntos. O conjunto A é um subconjunto de B se B contém todos os elementos que A também tem. O conjunto A é um superconjunto de B se B contém todos os elementos de A e mais alguns.

```
ghci> Set.fromList [2,3,4] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isProperSubsetOf` Set.fromList [1,2,3,4,5]
False
ghci> Set.fromList [2,3,4,8] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
False
```

Podemos ainda usar a `map` e `filter` para filtrá-los.

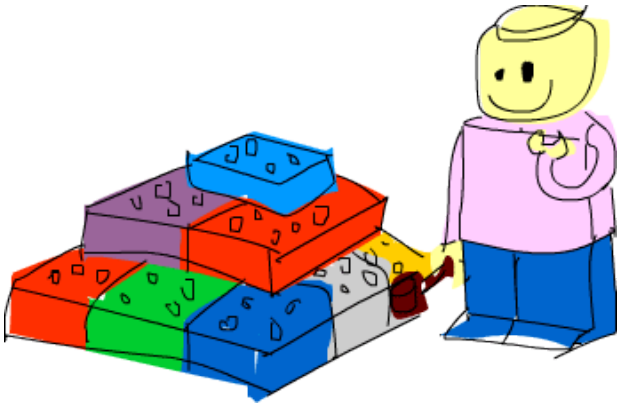
```
ghci> Set.filter odd $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,5,7]
ghci> Set.map (+1) $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
```

Conjuntos geralmente são usados para eliminar de uma lista valores duplicados transformando em **`fromList`** e convertendo de volta para uma lista com `toList`. A **`Data.List`** **`nub`** já faz isso, mas remover duplicados com listas grandes é muito mais rápido convertendo primeiro para um conjunto e depois convertendo de volta. Mas usar **`nub`** requer que os tipos dos elementos da lista estejam na typeclass **`Eq`**, enquanto para converter a lista em um elemento, deve estar em **`Ord`**.

```
ghci> let setNub xs = Set.toList $ Set.fromList xs
ghci> setNub "HEY WHATS CRACKALACKINE"
" ACEHIKLNRSTWY"
ghci> nub "HEY WHATS CRACKALACKIN"
"HEY WATSCRKLIN"
```

`setNub` geralmente é mais rápido do que **`nub`** em listas grandes, mas como pode ver, **`nub`** preserva a ordem dos elementos, enquanto **`setNub`** não.

Fazendo seus próprios módulos



Temos visto vários módulos interessantes, mas como fazemos o nosso próprio? Quase toda linguagem de programação permite dividir o código em vários arquivos e Haskell não é diferente. Ao escrever programas, uma boa prática é juntar funções e tipos que tem o mesmo propósito num módulo. Desse modo, você pode facilmente reutilizar funções de outros programas simplesmente importando módulos.

Vamos aprender como criar um módulo tendo como exemplo algumas funções de cálculo de volume e área de objetos geométricos. Podemos criar um arquivo chamado **`Geometry.hs`**.

Dizemos que um módulo *exporta* funções. Ao importar um módulo, eu passo a poder usar as funções que foram exportadas. Um módulo também pode definir funções que possuem chamadas a funçoes internas, mas só podemos ver e usar as explicitamente exportadas.

No início de um módulo, especificamos seu nome. Se nosso arquivo se chama **`Geometry.hs`**, nosso módulo deve se chamar **`Geometry`**. Especificamos as funções que serão exportadas e então podemos começar a escrever nossas próprias funções.

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
```

Como pode ver, calcularemos área e volume de esferas, cubos e prismas. Enfim, as próprias definições:

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c

cuboidArea :: Float -> Float -> Float -> Float
```

```
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

A boa e velha geometria. Mas temos algumas coisas a atentar. Como um cubo nada mais é do que um caso especial de prisma, definimos sua área e volume como um prisma com todas medidas iguais. Ainda definimos uma função auxiliar chamada **`rectangleArea`**, que calcula a área de um retângulo baseado nas medidas dos seus lados. Extremamente simples já que é pura multiplicação. Perceba que apesar de a usarmos (como também **`cuboidArea`** e **`cuboidVolume`**) no módulo, não a exportamos! Isso é devido ao módulo tratar-se apenas de funções de objetos tridimensionais (mas que não deixam de precisar da **`rectangleArea`**).

Ao criar um módulo, geralmente exportamos apenas as funções adequadas e essenciais ao seu propósito, deixando as outras internas. Se alguém estiver usando o módulo **`Geometry`**, não precisa se preocupar com outras funções alheias a seu interesse. Podemos decidir modificar ou mesmo deletar uma dessas funções em uma próxima versão (como deletar **`rectangleArea`** e substituir por **`*`**) e ninguém perceberia a mudança.

Para usar o módulo, é só fazer isso:

```
import Geometry
```

`Geometry.hs` deve estar na mesma pasta que o programa que o importa.

Módulos também podem assumir estruturas hierárquicas. Cada módulo pode ter submódulos que também podem ter seus submódulos. Vamos testar o conceito para que o **`Geometry`** tenha três submódulos, um para cada tipo de objeto.

Primeiro, criamos uma pasta chamada **`Geometry`**. Atenção ao G maiúsculo. Nela, teremos três arquivos: **`Sphere.hs`**, **`Cuboid.hs`**, e **`Cube.hs`**. O que cada arquivo conterá:

`Sphere.hs`

```
module Geometry.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

`Cuboid.hs`

```
module Geometry.Cuboid
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c
```

```
area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Cube.hs

```
module Geometry.Cube
( volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

Perfeito! O primeiro é **Geometry.Sphere**. Veja que colocamos tudo na pasta **Geometry** e definimos o módulo como **Geometry.Sphere**. Fazemos o mesmo que o prisma. Perceba também que em todos submódulo, temos funções de mesmo nome. Só podemos fazer isso por se tratarem de módulos separados. Queremos usar as funções de **Geometry.Cuboid** em **Geometry.Cube** mas não podemos apenas dar um **import Geometry.Cuboid** por exportar funções de mesmo nome que **Geometry.Cube**. Por isso que fazemos uma importação qualificada e tudo funciona.

Se estivesse num arquivo no mesmo nível da pasta **Geometry**, podemos, digamos:

```
import Geometry.Sphere
```

E então chamamos **area** e **volume** que devolverá área e volume de uma esfera. Mas se quisermos usar dois ou mais módulos, teríamos que fazer malabarismos. Ou... fazer importações qualificadas. Então só fazemos isso:

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

Só chamar **Sphere.area**, **Sphere.volume**, **Cuboid.area**, etc. e que cada um vai calcular área ou volume do objeto correspondente.

Logo que você se ver em um arquivo muito grande e com várias funções, tente perceber quais funções tem um propósito em comum e estudar se pode colocá-las num módulo separado. Você pode simplesmente importá-lo na próxima vez que um programa necessitar da mesma funcionalidade.

Funções de alta ordem Índice Criando seus próprios tipos e typeclasses