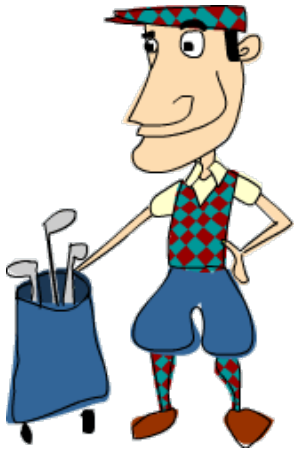
Tipos e Typeclasses Índice Recursão

Sintaxe em Funções

Pattern matching



Este capítulo irá cobrir algumas construções sintáticas interessantes do Haskell, e começaremos por pattern matching. Pattern matching consiste na pesquisa por padrões em determinados dados e, caso tenha sucesso, fazer algo com ele.

Ao definir funções, você pode definir códigos específicos para cada padrão. Isso gera um código mais conciso, simples e legível. Você pode avaliar qualquer tipo de dado — números, caracteres, listas, tuplas... Vamos fazer uma função bem simples que verifica se o número dado é sete ou não.

```
lucky :: (Integral a) => a -> String
lucky 7 = "SETE! BINGO!"
lucky x = "Desculpe, tente novamente!"
```

Ao chamar **`lucky`**, os padrões serão testados de cima para baixo e, de acordo com o resultado, executado ou não seu corpo. Aqui, o único modo dessa pattern ter sucesso é se ela for 7. Senão, vamos para a segunda, que será executada de qualquer maneira devido ao **`x`**. Essa função ainda poderia ser implementada por uma estrutura *if*. Mas e se quisermos uma função que diga o número se o parâmetro for de 1 até 5 ou **`"Não está entre 1 e 5"`** para outros? Sem *pattern matching*, teríamos que escrever uma estrura *if then else* bem confusa. No entanto, com ele podemos escrever assim

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "Um!"
sayMe 2 = "Dois!"
sayMe 3 = "Três!"
sayMe 4 = "Quatro!"
sayMe 5 = "Cinco!"
sayMe x = "Não está entre 1 e 5"
```

Veja que se movessemos a última *pattern* (o *else* geral) para o topo, sempre diria **`"Não está entre 1 e 5"`**, porque seria sempre executado para qualquer número e as linhas seguintes não teriam oportunidade de serem executadas.

Lembra da função fatorial que implementamos anteriormente? Definimos o fatorial de um número **`n`** como sendo **`product [1..n]`**. Nós ainda podemos usar recursividade, que é como geralmente é definida na matemática. Começamos explicitando que o fatorial de 0 é 1. Então dizemos que o fatorial de qualquer número inteiro positivo é ele multiplicado pelo fatorial do predecessor. Assim fica traduzido para o idioma Haskell.

```
factorial :: (Integral a) => a -> a
```

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Essa é a primeira vez que definimos uma função recursiva. Recursão é importante em Haskell e iremos ver com calma mais tarde. Mas para dar água na boca, veremos agora o que acontece ao tentarmos calcular o fatorial de, digamos, 3: tentamos descobrir o resultado de `3 * factorial 2`. O fatorial de 2 é `2 * factorial 1`, então até agora temos `3 * (2 * factorial 1)`. `factorial 1` é `1 * factorial 0`, então `3 * (2 * (1 * factorial 0))`. É ai que temos o truque — definimos o fatorial de 0 como sendo 1 com a intenção de não executar a terceira linha, então só retornamos 1. O resultado final será o equivalente a `3 * (2 * (1 * 1))`. Se tivéssemos colocado a linha para a recursão como primeira da função, o zero seria incluido e o cálculo seria infinito. Por isso a ordem é muito importante ao definir *patterns* e é melhor colocar as mais específicas no início e deixar as gerais para o fim.

Pattern matching também podem falhar, como no caso de definirmos uma função como essa:

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

e tentarmos chamá-la com um parâmetro inesperado:

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
"*** Exception: tut.hs:(53,0)-(55,21): Non-exhaustive patterns in function charName
```

Isso acontece por não termos uma *pattern* genérica. Ao criar *patterns*, sempre devemos incluir *pattern* que cubra todas as excessões para que o programa não pare de modo indevido caso usemos um valor inesperado.

Pattern matching também pode ser usada em tuplas. Se precisamos fazer uma função que recebe dois vetores em um plano bidimensional (em forma de pares) e o some, somamos seus x primeiro e depois seus y. Assim seria como faríamos se não conhecessemos *pattern matching*:

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

Bom, funciona. Mas existe um modo muito melhor de se fazer a mesma coisa. Vamos modificar essa função para que passe a usar *pattern matching*.

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Aí está. Muito melhor. Veja que temos um *pattern* padrão. O tipo de `addVectors` (em ambos casos) é `addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)`, então garantimos que receberemos dois pares como parâmetros.

`fst` e **`snd`** também precisam de parâmetros em pares. Mas e com triplas? Não há nenhuma função que trabalhe com elas, mas podemos fazer a nossa própria.

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

_ significa o mesmo que em compreensão de listas. Se o compilador não deve se importar com o que há ali, usamos o **_**.

O que me lembra que *patterns matching* também podem ser usadas em compreensão de listas. Veja...

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

Se a *pattern* falhar, se moverá para o próximo elemento.

As prórias listas podem ser usadas em pattern matching. Você pode casar um padrão com uma lista vazia **`[]`** ou qualquer padrão que involva **`:`** e uma lista vazia. Sabendo que **`[1,2,3]`** é apenas um açucar sintático para **`1:2:3:[]`**, você também pode utilizar o padrão original. Uma pattern como **`x:xs`** irá colocar a cabeça de uma lista em **`x`** o resto dela em **`xs`**, ainda que o único elemento em **`xs`** seja uma lista vazia.

> **Nota**: O padrão **`x:xs`** é muito usado, principalmente em funções recursivas. O (possível) problema é de **`:`** funcionar apenas com lista de length 1 ou maior.

Se você quer deixar, digamos, os três primeiros elementos de uma lista numa variável e o resto em outra, você pode escrever algo como **`x:y:z:zs`**. Só funcionará com lista de 3 ou mais elementos.

Agora que conseguimos comparar um *pattern* com uma lista, vamos criar nossa própria implementação da função **`head`**.

```
head' :: [a] -> a
head' [] = error "Proibido chamar head em uma lista vazia, amador!"
head' (x:_) = x
```

Vejamos se funciona:

```
ghci> head' [4,5,6]
4
ghci> head' "Olá"
'O'
```

Excelente! Saiba ainda que se você quiser especificar diversas variáveis (mesmo que uma delas seja `_` e que isso não seja realmente usado em algo), deveremos coloca-las entre parênteses. Perceba também a função **`error`** que usamos. Ela recebe uma string e gera um runtime error, usando a string para nos informar o erro ocorrido. Já que ele causa a finalização do programa, é bom usa-lo moderadamente. Mas chamar **`head`** com uma lista vazia não faz sentido, então...

Vamos então criar uma função muito útil que nos diz os primeiros elementos de uma lista de um modo mais (des)interessante.

```
tell :: (Show a) => [a] -> String
tell [] = "A lista esta vazia"
tell (x:[]) = "A lista tem apenas um elemento: " ++ show x
tell (x:y:[]) = "A lista tem dois elementos: " ++ show x ++ " e " ++ show y
tell (x:y:_) = "Esta lista esta longa demais. Veja os dois primeiros elementos: " ++ show x ++
```

Essa função é considerada segura porque prevê uma lista vazia, uma lista com apenas um elemento, uma com dois e outra com mais elementos. O **`(x:[])`** e **`(x:y:[])`** poderiam ser escritos como **`[x]`** e **`[x,y]`** (como é algo mais simples, despensa-se os parênteses). Ao contrário de **`(x:y:_)`**, que são para listas de mais de dois elementos e não podem ser escritos com colchetes.

Nós já implementamos a nossa própria **`length`** usando compreensão de listas. Agora tentaremos fazer usando pattern matching e um pouco de recursão:

```
length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs
```

Isso é similar à função de fatorial que escrevemos anteriormente. Primeiro definimos o resultado do valor conhecido — a lista vazia. Isso é conhecido também como edge condition. Então no segundo pattern dividimos a coitada da lista em primeiro elemento e resto. Dissemos que a length da lista é igual a 1 somado com a length do "resto". Usamos `_` para o primeiro elemento porque não nos importamos com o seu valor. Perceba que nos preocupamos com os dois lados: uma lista vazia na primeira linha e uma com mais de um elemento na segunda.

Então vamos descobrir o que acontece ao chamar **`length'`** com **`"ham"`**. Primeiro, é testado se é uma lista vazia. Já que não é, passa-se para o segundo pattern. Aí passamos no teste e o resultado da length é **`1 + length' "am"`**, porque dividimos a lista em primeiro e resto e descartamos o primeiro elemento. Até aí tudo bem. **`length'`** de **`"am"`** é, semelhantemente, **`1 + length' "m"`**. Então agora temos **`1 + (1 + length' "m")`**. **`length' "m"`** é **`1 + length' ""`** (que também poderia ser escrito como **`1 + length' []`**). E já definimos que **`length' []`** é **`0`**. Logo, no fim temos **`1 + (1 + (1 + 0))`**.

Vamos implementar **`sum`**. Sabemos que a soma de uma lista vazia deve ser 0. Escrevemos então como um pattern. E também sabemos que a soma de uma lista é o primeiro elemento mais a soma do resto da lista. Se usarmos o mesmo método, temos:

```
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

Ainda existe uma coisa chamada de ***as patterns***. Eles são uma mão na roda quando precisamos usar um pattern e continuar com ele acessível, mesmo depois de usado. Você cria isso colocando um `@` na frente do nome do pattern. Por exemplo, o pattern **`xs@(x:y:ys)`**. Esse *as patterns* irá fazer exatamente a mesma coisa que **`x:y:ys`**, mas você consegue obter a lista inteira usando apenas **`xs`**, ao invés repetir tudo novamente digitando **`x:y:ys`** no corpo da função novamente. Um exemplo bem simples:

```
capital :: String -> String
capital "" = "String vazia, oops!"
capital all@(x:xs) = "A primeira letra de " ++ all ++ " é " ++ [x]
```

```
ghci> capital "Dracula"
"A primeira letra de Dracula é D"
```

Normalmente usamos esse recurso *as patterns* para evitar repetição de grandes patterns usadas durante a declaração de uma função.

Mais uma coisa — você não pode usar **`++`** em patterns. Se você tentar usar o pattern **`(xs ++ ys)`**, qual seria a primeira e qual seria a segunda lista? Não faz muito sentido. Faria algum sentido usar **`(xs ++ [x,y,z])`** ou **`(xs ++ [x])`**, mas pela estrutura das listas isso não é possível.

Guardas, guardas!

Enquanto patterns é um jeito de ter certeza de que um valor tenha a forma desejada e torna possível desmembra-lo, guards (guardas) são uma forma de testar se uma (ou mais) propriedades são verdadeiras ou falsas. Se isso te parece demais com um if, está no caminho certo. A diferença é que guards são mais fáceis de ler quando têm muitas condições e funcionam bem com patterns.

Ao invés de parar para explicar a sintaxe, vamos direto criar uma função usando guards. Faremos uma função simples que repreende o usuário de uma forma específica, dependendo do seu IMC. Seu IMC é igual à sua massa dividida pelo quadrado de sua altura. Se seu IMC está abaixo de 18,5, você é considerado magro. Se está entre 18,5 e 25, é saudável. Entre 25 e 30, acima do peso. Maior, obeso. Então, esta é a função (ela não calcula, somente mostra a mensagem)

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "Você esta abaixo do peso!"
    | bmi <= 25.0 = "Supostamente você esta normal. Pfff, aposto que você é feio!"
    | bmi <= 30.0 = "Você esta gordo! Faça uma dieta, gorducho!"
    | otherwise   = "Você é uma baleia, meus parabéns!"
```

Guards são marcados por pipes, seguidos do nome de uma função e seus parâmetros. Geralmente, são alinhados e identados um pouco a direita. Um guarda geralmente é uma expressão booleana. Se é **`True`**, a função correspondente é executada. Se **`False`**, o resto da linha não é executada e é passado para a próxima. Se chamarmos essa função com **`24.3`**, será testado se é menor ou igual a **`18.5`**. Já que não é, o próximo guard é testado. A verificação tem sucesso ao passar pelo segundo guard, já que 24,3 é menor que 25,0. Assim, é retornado o resultado do segundo guard.

Os guards são remanescentes de uma árvore encadeada de if/elses da linguagens imperativas, mas muito mais legíveis. Apesar de árvores de if/elses encadeados serem extremamente desaconselhados, às vezes um problema é definido de um modo que não há muitas alternativas. Em Haskell, Guards são uma solução.

Em vários casos, o último guard é **`otherwise`**. **`otherwise`** é definido com **`otherwise = True`** e aprova tudo. É muito parecido com patterns, que testam por padrões ao invés de condições booleanas. Se todos os guards de uma função derem **`False`** (e não tivermos especificado um guard **`otherwise`**), é testado o próximo **pattern**. É assim que patterns e guards funcionam bem em conjunto. Se nenhum guard ou pattern se aplicar, é lançado um erro.

E é claro que podemos usar guards com funções que recebam quantos parâmetros desejarmos. Ao invés de obrigarmos o usuário a calcular o seu próprio IMC antes de usar a função, vamos modificá-la para receber altura e peso e descobrir automaticamnte.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "Você esta abaixo do peso!"
    | weight / height ^ 2 <= 25.0 = "Supostamente você esta normal. Pfff, aposto que você é fe
    | weight / height ^ 2 <= 30.0 = "Você esta gordo! Faça uma dieta, gorducho!"
    | otherwise                   = "Você é uma baleia, meus parabéns!"
```

Vamos descobrir se eu estou gordo...

```
ghci> bmiTell 85 1.90
"Supostamente você esta normal. Pfff, aposto que você é feio!"
```

Ei! Não estou gordo! Mas Haskell me chamou de feio mesmo assim. Que seja.

Veja que não há um **=** depois do nome da função e seus parâmetros, mas antes do primeiro guard. Muitos iniciantes recebem erros de sintaxe por colocarem aí.

Mais uma outra bem simples: vamos implementar nosso próprio **`max`**. Se não se lembra, ele recebe dois parâmetros e retorna o maior.

```
max' :: (Ord a) => a -> a -> a
max' a b
    | a > b     = a
    | otherwise = b
```

Guards também podem ser escritos em apenas uma linha, no entanto eu não recomendo graças a sua pouca legibilidade, mesmo em funções curtas. Mas para fins de demonstração, poderíamos escrever nosso **`max'`** assim:

```
max' :: (Ord a) => a -> a -> a
max' a b | a > b = a | otherwise = b
```

Ugh! Nem um pouco legível! Próxima: nossa própria impremetação de **`compare`** usando guards.

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
```

```
    | a > b     = GT
    | a == b    = EQ
    | otherwise = LT


ghci> 3 `myCompare` 2
GT
```

Nota: Do mesmo modo que podemos chamar funções infixas usando crases em volta do seu nome, podemos defini-las. Às vezes se torna mais fácil de se ler.

Onde!?

Na seção anterior, definimos uma função de calculadora de IMC que era algo parecido com isso:

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "Você esta abaixo do peso!"
    | weight / height ^ 2 <= 25.0 = "Supostamente você esta normal. Pfff, aposto que você é fe
    | weight / height ^ 2 <= 30.0 = "Você esta gordo! Faça uma dieta, gorducho!"
    | otherwise                   = "Você é uma baleia, meus parabéns!"
```

Note ainda que nos repetimos três vezes. Nós nos repetimos três vezes. Repetir-se (três vezes) no código é tão desejável quando levar um chute bem no meio da testa. Já que repetimos a expressão três vezes, seria melhor se calculássemos apenas uma vez, gravássemos numa variável para usarmos nos próximos comandos ao invés da expressão. Para isso, modificamos nossa função para isso:

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "Você esta abaixo do peso!"
    | bmi <= 25.0 = "Supostamente você esta normal. Pfff, aposto que você é feio!"
    | bmi <= 30.0 = "Você esta gordo! Faça uma dieta, gorducho!"
    | otherwise   = "Você é uma baleia, meus parabéns!"
    where bmi = weight / height ^ 2
```

Colocamos a palavra-chave **`where`** (geralmente identamos até onde estão os pipes) e definimos variáveis ou funções. Esses nomes são visíveis dentro dos guards e nos permitem não ficar nos repetindo. Se decidirmos que iremos calcular o IMC de um modo diferente, precisamos mudar apenas uma vez. Podemos avançar um pouco e deixar nossa função assim:

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "Você esta abaixo do peso!"
    | bmi <= normal = "Supostamente você esta normal. Pfff, aposto que você é feio!"
    | bmi <= fat    = "Você esta gordo! Faça uma dieta, gorducho!"
    | otherwise     = "Você é uma baleia, meus parabéns!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

Os nomes criados na seção where só são visíveis dentro da função, então não temos de nos preocupar com elas poluindo o namespace de outras funções. Note também que todos os nomes foram alinhados na mesma coluna. Se

não o fizéssemos, Haskell ficaria confuso e não saberia que eles fazem parte do mesmo bloco.

Associações *where* não são compartilhadas entre diferentes patterns. Se você quiser que vários patterns de uma mesma função compartilhem um determinado nome, você deverá especificá-los como global.

Você também pode usar *where* em conjunto com **pattern match**! Poderíamos reescrever a parte *where* da nossa função anterior como:

```
...
where bmi = weight / height ^ 2
      (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

Vamos fazer agora outra função extremamente necessária que recebe nome e sobrenome e retorna suas iniciais.

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

Poderíamos fazer esse *pattern matching* diretamente nos parâmetros da função (o que resultaria em um código mais limpo) mas é só para mostrar que é possível também fazer isso usando o *where*.

Assim como definimos constantes em blocos where, você também pode definir funções. Voltando ao nosso tema de programação saúdavel, vamos fazer uma função que pega uma lista de pesos/altura e retorna sua lista de IMC.

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
    where bmi weight height = weight / height ^ 2
```

E por hoje é só! A razão pela qual demos o `IMC` como exemplo de função é que não poderíamos calcular diretamente nos seus parâmetros. Se vermos a lista passada pela função, veremos que cada par possui um IMC diferenciado.

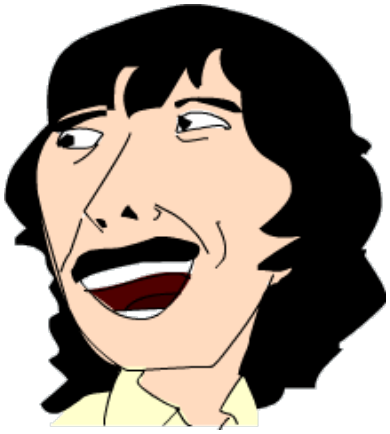
Associações *where* também podem ser aninhadas. É comum criar uma função e definir algum helper com suas cláusulas com funções e daí criar funções helper com suas próprias cláusulas *where*.

Deixe estar

Muito semelhantes às associações where, existem as **let**. Associações where te permitem dar valores a variáveis no fim de uma função, ao mesmo tempo que permitem que toda função a veja, incluindo guards. Associações let dão valores a funções e também são expressões, mas tem um escopo mais restrito por não serem acessíveis dentro de guards. Assim como toda construção em Haskell que permite dar valores a nomes, construções let podem ser usadas em pattern matching. Vejamos em ação! Poderíamos fazer assim uma função que nos dá a área da superfície de um cilíndro baseado em sua altura e raio:

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

Deve-se seguir a forma **`let <bindings> in <expression>`**. Os nomes definidos na parte *let* são acessíveis mesmo na expressão seguinte *in*. Como você deve ter percebido, poderíamos ter definido essa associação com um *where*. Perceba ainda que os nomes também estão alinhados na mesma coluna. Então qual é a diferença dos dois? Por hora, *let* coloca as associações antes da expressão que as usa, o contrário da *where*.



Outra diferença é que associações *let* são expressões por si próprias. *where* são apenas construções sintáticas. Lembre-se de quando vimos o if e explicamos que a estrutura if/else é uma expressão que pode ser enfiada em qualquer lugar?

```
ghci> [if 5 > 3 then "Woo" else "Boo", if 'a' > 'b' then "Foo" else "Bar"]
["Woo", "Bar"]
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

Isso também pode ser feito com associações let.

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

Elas ainda podem ser alternativas para criação e uso de funções de escopo local:

```
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

Se quisermos definir várias variáveis na mesma linha, obviamente não podemos alinhá-las em uma coluna. É por isso que também podemos separá-las por pontos e vírgulas.

```
ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "there!" in foo ++ bar)
(6000000,"Hey there!")
```

Você não necessariamente precisa-se colocar um ponto e vírgula depois da última associação. Como já dissemos, podemos usar pattern match em associações *let*. Isso é muito útil para quebrar uma tupla e dar nomes a cada componente.

```
ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

Também pode-se colocar associações *let* dentro de compreensão de listas. Vamos reescrever nosso exemplo anterior que recebe listas de pares massa-altura, usar um *let* e dispensar a definição de uma função auxiliar interna com *where*.

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

Incluímos o *let* dentro da compreensão de listas como predicado, mas não com a intenção de filtrar a lista, mas sim para apenas dar nomes aos elementos. Nomes definidos num *let* dentro de uma compreensão de listas são visíveis à

função a que pertence (o que antecede o `|`) e a todos os predicados e seções que seguem à associação. Então simplesmente podemos fazer nossa funão retornar o IMC apenas de pessoas gordas:

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

Só não podemos usar o nome **`imc`** em **`(w, h) <- xs`** porque é executado antes da definição do *let*.

Omitimos o *in* do *let* porque usamos-o em uma compreensão de listas, que já pré-define a visibilidade de nomes. No entanto, poderíamos usar *let in* no predicado e seus nomes serem acessíveis apenas dentro do predicado. O *in* ainda pode ser omitido ao definir funções e constantes diretamente no GHCI. Caso o façamos, os nomes serão visíveis por toda a sessão interativa.

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
<interactive>:1:0: Not in scope: `boot'
```

Você pode estar se perguntando: Se associações *let* são tão perfeitas, porque não usá-las ao invés de *where*? Bom, já que associações *let* são mais restritivas quanto ao escopo, elas não podem ser usadas entre guards. Algumas pessoas preferem usar *where* porque os nomes vêm depois da função que os usam. Desse modo, o corpo da função fica próximo de nomes e declaração de tipo, tornando mais legível.

Expressões case

Se você já programou em outras linguagens imperativas (C, C++, Java, etc.), sabe do que estou falando. É pegar uma variável e executar blocos de código específicos para cada valor pré-definido e possivelmente definir um bloco padrão no caso dela ter um valor inesperado.

Haskell pega o conceito e o expande. Como o próprio nome sugere, expressões case são, bom, expressões, e muito semelhante com expressões if/else e associações *let*. Não só podemos testar expressões baseadas em seu valor, como podemos também usar pattern matching. Hmmmmm, pegar uma variável, procurar por um pattern, executar blocos de código para cada resultado... onde já vi isso? Ah claro, pattern matching em parâmetros de funções! Mas isso é só um atalho para executar uma expressão case. Esses dois códigos fazem a mesma coisa, então pode escolher qual você prefere:

```
head' :: [a] -> a
head' [] = error "Não existe head em listas vazias!"
head' (x:_) = x
```

```
head' :: [a] -> a
head' xs = case xs of [] -> error "Não existe head em listas vazias!"
                      (x:_) -> x
```

Como vê, a sintaxe de expressões case são extremamente simples:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

`expression` é testada por alguns patterns. Pattern matching funciona exatamente como se presume: o primeiro pattern que verificar-se verdadeiro é executado. Se todo o case for executado e nenhum pattern se encaixar, acontece um runtime error.

Enquanto pattern matching dentro dos parâmetros só podem ser feitos ao definir funções, expressões case podem ser colocadas praticamente em qualquer lugar. Olha só:

```
describeList :: [a] -> String
describeList xs = "A lista é " ++ case xs of [] -> "vazia."
                                             [x] -> "uma lista unitária."
                                             xs -> "uma lista grande."
```

Elas são úteis para usar pattern matching no meio de expressões. Já que pattern matching em definições de função são um atalho para expressões case, poderíamos ainda ter feito desse jeito:

```
describeList :: [a] -> String
describeList xs = "A lista é " ++ what xs
    where what [] = "vazia."
          what [x] = "uma lista unitária."
          what xs = "uma lista grande."
```

Tipos e Typeclasses Índice Recursão