Capítulo 5
Guardas

No capítulo anterior foram abordados mecanismos que possibilitavam o tratamento de fluxos condicionais complexos de uma forma simples e elegante: o *casamento de padrão* e as *case-expressions.* Neste capítulo exploraremos um outro recurso que nos permite fazer expressões condicionais: os *guardas.* Recurso este que poderia ter sido facilmente incluindo no capítulo anterior, mas que em virtude de sua importância, merece um capítulo somente dedicado a ele.

5.1 A limitação dos sistemas usados

Apesar de os sistemas condicionais explorados no capítulo anterior serem realmente bons e elegantes, eles possuem algumas limitações. A tabela a seguir expõe as principais:

Casamento de padrão	Case expressions
Limita o sistema condicional à análise de um parâmetro já definido. Na maioria das vezes, este parâmetro se trata de uma lista pela facilidade que o casamento de padrão permite que análises/processamentos sejam feitos recursivamente nestas estruturas.	*Limita* o sistema condicional à análise de um parâmetro já definido. Na maioria das vezes, este parâmetro se trata de uma única expressão pela facilidade que as "case-expressions" têm em executar fluxos condicionais com base em valores alfanuméricos (números ou caracteres)

Para que estas desvantagens se tornem ainda mais evidentes, tomemos como exemplos alguns códigos do capítulo anterior.

```
{-Exemplo com casamento de padrão-}
1.length' :: (Show a) => [a] -> Int
2.length' [] = 0
3.length' (x:xs) = 1 + length' xs

{-Exemplo com "case expressions"-}
1.centralTelefonica :: Integral a => a -> String
2.centralTelefonica opcao = case opcao of
3.  1 -> "Parabens ! Voce discou 1"
4.  2 -> "Parabens ! Voce discou 2"
5.  3 -> "Parabens ! Voce discou 3"
6.  _ -> "Que pena, Voce discou outro numero !"
```

Tendo em vista que o casamento de padrão utilizado na função «length'» tem como objetivo principal a contagem recursiva de caracteres, caso, por algum motivo desconhecido, seja desejo adaptar esta função para que ela não contabilize o caractere "z". Duas soluções poderiam ser empregadas:

- Aplicar um casamento de padrão extra e deixar a função menos elegante e menos legível.
- Abrir mão de trabalhar apenas com casamento de padrão e acrescentar um if-them-else na 3° linha, o que deixará o código menos legível ainda.

Ambas as soluções são apresentadas abaixo.

```
{-1° solução-}
lengthSemZ :: Num a => [Char] -> a
lengthSemZ [] = 0
lengthSemZ ('z':xs) = lengthSemZ xs
lengthSemZ (x:xs) = 1 + lengthSemZ xs

{-2° solução-}
lengthSemZ' :: Num a => [Char] -> a
lengthSemZ' [] = 0
lengthSemZ' (x:xs) = if x=='z' then lengthSemZ' xs else 1+lengthSemZ' xs
```

Como é possível observar na primeira solução, o casamento de padrão continua funcionando, mas, quando se trata deste tipo de tratamento de arbitrariedades, o casamento de padrão deixa muito a desejar em termos de legibilidade. Já a segunda solução, possui um problema de legibilidade ainda mais evidente em virtude da dificuldade de escrita da expressão *if-them-else*. Seria mais desejoso que houvesse um tipo de tratamento condicional que conseguisse analisar arbitrariedades de uma forma simples e elegante.

Analisando agora o exemplo com a “case-expressions”, é bastante evidente que ela carece de recursos simples, como a possibilidade de utilização de operadores relacionais (>,<,==, /=) para a análise de valores, assim como a análise de várias variáveis ao mesmo tempo (A > B && A > C, por exemplo). Um caso em que esta limitação se torna evidente é pontuado abaixo.

```
{-Exemplo com "case expressions"-}
1.centralTelefonica :: Integral a => a -> String
2.centralTelefonica opcao = case opcao of
3.  opcao > 6 && opcao < 12 -> "Parabens ! Voce discou um número maior que 6 e menor que 12"
4.  2 -> "Parabens ! Voce discou 2"
5.  3 -> "Parabens ! Voce discou 3"
```

Como é de se esperar, ganhamos um erro ao tentar carregar este código no ghci.

```
codigos.hs:line:column: error:
    Parse error in pattern: opcao > 6 && opcao < 12
   |
line |   opcao > 6 && opcao < 12 -> "Parabens ! Voce discou um número maior que 6 e menor que 12"
   |   ^^^^^^^^^^^^^^^^^^^^^^^^
Failed, no modules loaded.
```

Apesar de o uso da sintaxe da função “centralTelefonica” estar obviamente errada, o código apresentado ilustra bem a impossibilidade de realização de análises condicionais elaboradas a partir do uso das “case-expressions”, pois elas se limitam a análise dos valores exatos das expressões aritméticas e a realização do retorno correspondente ao valor da expressão. Seria realmente muito bom caso existisse um tipo de expressão condicional que fosse capaz de realizar análises mais elaboradas.

Para nossa sorte, esta ferramenta existe e supre todas as deficiências do casamento de padrão e das “case-expressions” : Estou falando do “guarda” !

NOTE : Apesar de ser possível realizar condicionais simples utilizando *casamento de padrão*, é mais aconselhável utilizá-lo para o processamento recursivo de lista e utilizar uma *case expression* para tratar dos condicionais mais simples.

5.2 Guardas

Sem mais delongas, vamos explorar o funcionamento dos “guardas”. Para tanto, observe o código abaixo que tem como propósito classificar a prioridade de uma tarefa com base em sua pontuação. Quanto maior a pontuação, maior a prioridade.

```
qualificaPrioridade :: (Fractional a, Ord a) => a -> String
qualificaPrioridade tarefa
  |tarefa <=250 = "Pouco importante" --1° guarda
  |tarefa <=500 = "Importância mediana" --2° guarda
  |tarefa <= 750 = "Importante"--3° guarda
  |otherwise = "Extremamente importante"-- guarda "cath-all"
```

Como se pode notar, a função «qualificaPrioridade» recebe um valor pertencente aos reais e, com base nele, há o retorno de uma string que representa a sua classificação. Caso o valor seja menor ou igual a 250 a sua classificação é “Pouco importante”, assim sendo, a função nem sequer executa a condição dos demais guardas e a sua execução é terminada.

Em relação a este comportamento, observe uma coisa interessante: apesar dos guardas apresentados só avaliarem limites superiores da pontuação da tarefa, uma vez que para a condição do k-ésimo guarda seja executada é necessariamente verdade que a condição do guarda em k-1 seja falsa e isto se segue para todo o guarda em uma posição “k” >= 2, na prática, temos a seguinte função:

```
qualificaPrioridade :: (Fractional a, Ord a) => a -> String
qualificaPrioridade tarefa
  |tarefa > 0 && tarefa <=250 = "Pouco importante" --1° guarda
  |tarefa >250 && tarefa <=500 = "Importância mediana" --2° guarda
  |tarefa > 500 && tarefa <= 750 = "Importante" --3° guarda
  |otherwise = "Extremamente importante" -- guarda "cath-all"
```

Além disto, observa-se que, caso os 3 primeiros guardas sejam falsos, se recorre a solução dos covardes: o guarda “otherwise”[1]. Neste contexto, a sua funcionalidade é atribuir o maior nível de importância a uma tarefa independente de sua pontuação e, uma vez que ele está no final da função, isto faz com que toda a tarefa com pontuação superior a 750 seja classificada como “Extremamente importante”.

Outrossim, é bastante notável que a sintaxe de um guarda resume-se ao uso de indentação, seguido de um pipe (“|”) e da condição a ser realizada juntamente ao valor de retorno da função caso a condição a ser avaliada seja verdadeira.

1 “otherwise” significa “de outra forma”

5.3 Where

Apesar da função “qualificaPrioridade” estar correta, de modo geral, é mais desejoso que coisas como a prioridade de uma tarefa sejam calculadas pelo computador e não sejam fornecidas pelo usuário. Tendo isto em mente, a função abaixo expõe uma forma de fazer o cálculo de prioridade tendo em vista três parâmetros:

1. imp: A importância inerente a tarefa;
2. ani: O ânimo que usuário sente em querer executar a tarefa;
3. arr: O arrependimento que o usuário sentirá se não executar a atividade.

```
qualificaPrioridade :: (Fractional a, Ord a) => a -> a -> a -> String
qualificaPrioridade i ani arr
  |i*ani*arr <=250 = "Pouco importante"
  |i*ani*arr <=500 = "Importância mediana"
  |i*ani*arr <= 750 = "Tarefa importante"
  |otherwise = "Tarefa extremamente importante"
```

Apesar de o programa estar correto, nota-se duas coisas desagradáveis a seu respeito:

1. Deselegância: A constate repetição da expressão aritmética “i*ans*arr” deixa o código cansativo de se ler, além de prejudicar a legibilidade;
2. Lentidão: Uma vez que todos os guardas realizam a mesma operação, no pior caso, a expressão “i*ani*arr” será repetido três vezes e tudo isto só pra executar o “otherwise” !

Para solucionar esta inconveniência, existe um recurso que nos permite declarar funções auxiliares no escopo local das funções: o “where”. No contexto da função apresentada, o espírito da coisa consiste em definir o retorno de uma função secundária como sendo “i*ani*arr”, de tal forma que, para executar as comparações, só seja necessário invocar essa função secundária, assim, tornando o código mais limpo e otimizado. Observe:

```
qualificaPrioridade :: (Fractional a, Ord a) => a -> a -> a -> String
qualificaPrioridade i ani arr
  |tarefa <=250 = "Pouco importante"
  |tarefa <=500 = "Importância mediana"
  |tarefa <= 750 = "Tarefa importante"
  |otherwise = "Tarefa extremamente importante"
  where tarefa = i*ani*arr
```

De princípio, o código apresentado pode causar uma certa confusão aos olhos leigos, pois a “variável” «tarefa» está sendo usada constantemente nas condições dos guardas, mas ela não foi passada como parâmetro para a função.

No entanto, se observamos bem, notamos que, logo após o “otherwise” há uma cláusula “where” que indica a declaração de uma função auxiliar chamada «tarefa», de tal forma que o seu valor de retorno seja o produto entre os três parâmetros passados. Portanto, o cálculo da prioridade fica armazenado em «tarefa» e pode ser referenciado no escopo local da função «qualificaPrioridade».

Ainda é possível usar a cláusula where para a criação de *aliases*, isto é, apelidos para os valores que os guardas usam. Em alguns casos, isto pode ajudar a melhorar a legibilidade.

```
qualificaPrioridade :: (Fractional a, Ord a) => a -> a -> a -> String
qualificaPrioridade i ani arr
  |tarefa <= categoria01 = "Pouco importante"
  |tarefa <= categoria02 = "Importância mediana"
  |tarefa <= categoria03 = "Tarefa importante"
  |otherwise = "Tarefa extremamente importante"
  where
    tarefa = i*ani*arr
    categoria01 = 250
    categoria02 = 500
    categoria03 = 750
```

No exemplo acima, houve a declaração de mais três funções auxiliares: “categoria01”, “categoria02” e “categoria03”. A única funcionalidade delas é melhorar a legibilidade do código. No entanto, é bastante notável que elas falharam miseravelmente em alcançar seus objetivos, mas, a depender da função, o uso deste tipo de apelido pode trazer melhorias significativas para a legibilidade do código.

5.3.1 Where e casamento de padrão

No intuito de facilitar a escrita do código, isto é, diminuir o tempo que se escreve a função “qualificaPrioridade”, um programador ainda poderia optar pôr definir os valores de retorno de “categoria01”, “categoria02” e “categoria03” através de um casamento de padrão.

```
qualificaPrioridade :: (Fractional a, Ord a) => a -> a -> a -> String
qualificaPrioridade i ani arr
  |tarefa <= categoria01 = "Pouco importante"
  |tarefa <= categoria02 = "Importância mediana"
  |tarefa <= categoria03 = "Tarefa importante"
  |otherwise = "Tarefa extremamente importante"
  where
     tarefa = i * ani * arr
     (categoria01, categoria02, categoria03) = (250, 500, 750)
```

5.3.2 Casamento de padrão, where e compreensão de listas

A função «qualificaPrioridade» tem um bom potencial no que diz respeito ao cálculo do nível de prioridade de uma tarefa, mas não é cabível que um usuário tenha 10 tarefas, por exemplo, e tenha que calcular a prioridade dessas tarefas individualmente e anotá-las em algum lugar. Por isto, é bastante desejável que uma lista com as tarefas sejam passadas para «qualificaPrioridade» e ela calcule a prioridade das tarefas em lote.

Para tanto, basta que a função receba uma lista cujos elementos sejam “quadruplas” e retorne uma lista de tuplas contendo o nome da tarefa e a sua prioridade. Quanto aos elementos que compõe a “quadrupla”, eles consistem no nome da tarefa e dos 3 parâmetros que foram passados para a «qualificaPrioridade» até agora.

```
qualificaPrioridadeLote :: (Fractional a, Ord a) => [(String, a, a, a)] -> [(String,a)]
qualificaPrioridadeLote tarefas = [calcPrioridade tarefa | tarefa <- tarefas]
 where
   calcPrioridade (nomeTarefa, i, ani, arr) = (nomeTarefa, i*ani*arr)
```

Usando um pouco de licença poética, a função acima pode ser lida da seguinte forma: “calcule a prioridade de tarefa usando «calcPrioridade», onde «calcPrioridade» é dado pelo produto de ‘i’, ‘ani’ e ‘arr3’,de tal forma que tarefa pertença ao conjunto de tarefas.”. Observe a sua execução:

```
>> qualificaPrioridadeLote [("EDA",10,8,10),("Automatos",10,6,10),("Logica", 10, 8, 10), ("Probabilidade", 7, 7, 8), ("Programacao funcional", 10, 6, 10)]

[("EDA",800.0),("Automatos",600.0),("Logica",800.0),("Probabilidade",392.0),("Programacao funcional",600.0)]
```

Caso seja mais desejoso fazer com que o segundo elemento da tupla seja a classificação da tarefa quanto a sua prioridade e não o valor de sua prioridade em sí, ainda poderíamos adaptar um pouco a função apresentada.

```
qualificaPrioridadeLote' :: (Fractional a, Ord a) => [(String, a, a, a)] -> [(String,String)]
qualificaPrioridadeLote' tarefas = [calcPrioridade tarefa | tarefa <- tarefas]
  where
      calcPrioridade (nomeTarefa, i, ani, arr)
        | tarefa' <= 250 = (nomeTarefa, "Pouco importante")
        | tarefa' <= 500 = (nomeTarefa, "Importância mediana")
        | tarefa' <= 750 = (nomeTarefa, "Importante")
        | otherwise = (nomeTarefa, "Extremamente importante")
        where
          tarefa' = i*ani*arr
```

Conseguimos observar coisas bastante interessantes neste código. Abaixo, elas são listadas.

1. Utiliza-se uma cláusula “where” ninhada, ou seja, um “where” dentro de outro “where”. Neste caso a cláusula mais interna «tarefa’» é responsável por calcular a prioridade da tarefa. Uma coisa curiosa de se notar é que, para retornar o produto entre “i”, “ani” e “arr3”, «tarefa’» não necessita receber nenhum parâmetro e isto acontece porque estes dados já foram passados como parâmetros para função declarada no “where” mais externo. Ou seja, as cláusulas mais internas têm acesso aos parâmetros das mais externas.
2. Uma vez que se calcula a prioridade de uma tarefa, é hora de classificá-las de acordo com a sua pontuação. Para tanto, utiliza-se uma lógica semelhante à das funções apresentadas anteriormente, ou seja, compara-se a pontuação com um limite superior de uma dada categoria. No entanto, desta vez uma tupla constituída pelo nome da tarefa e sua classificação é retornada, ao invés de somente a classificação ser retornada.
3. Uma vez que os dois aspectos pontuados anteriormente só são capazes de retorna uma única tupla e não conseguem fazer isto para todos os elementos de uma lista de tuplas. Chegou o momento de aplicar a função «calcPrioridade» em todas as tuplas da lista.

4. Usando licença poética mais uma vez, o código pode ser lido da seguinte forma: “para toda tarefa contida em tarefas, aplique «calcPrioridade», onde «calcPrioridade» é a comparação em cascata[2] entre tarefa’ e um limiteSuperior, onde tarefa’ é o produto entre “i”, “ani” e “arr3”.

5.4 Let

A esta altura você já deve ter notado que, uma vez que o “where” se traduz como “onde” em português, as funções que utilizam este mecanismo se parecem bastante com definições matemáticas. Por exemplo, para encontrar as raízes de uma equações quadráticas poderíamos usar a fórmula de Bháskara e dizer que:

- $x' = \frac{(-b+\sqrt{(\Delta)})}{(2a)}$, onde $\Delta = b^2 - 4ac$, de tal forma que `Δ>=0`

De forma bastante parecida com esta equação, poderíamos definir uma função que encontra x’ em haskell da seguinte forma:

```
x' :: (Floating a, Ord a) => a -> a -> a -> Either String a
x' a b c
   | delta < 0 = Left "Oops!"
   | otherwise = Right ((-b + sqrt delta) / (2 * a))
   where delta = b^2 - 4 * a * c
```

No entanto, na matemática, também é bastante comum usar a seguinte construção:

- Seja $\Delta = b^2 - 4ac$ em x’= $\frac{(-b+\sqrt{(\Delta)})}{(2a)}$

E o haskell também consegue representar este tipo de construção através do “let”! Observe

```
_x' :: (Floating a, Ord a) => a -> a -> a -> a
_x' a b c =
   let delta = b^2 - 4 * a * c
   in (-b + sqrt delta) / (2 * a)
```

Para checar se a descriminante é negativa, basta fazer:

```
__x' :: (Floating a, Ord a) => a -> a -> a -> Either String a
__x' a b c =
   let delta = b^2 - 4 * a * c
   in
      if delta < 0
         then Left "Oops!"
         else Right ((-b + sqrt delta) / (2 * a))
```

2 Os guardas realizam uma comparação em cascata, isto é, executam a primeira comparação e, caso ela seja falsa, executam a segunda e assim sucessivamente até que alguma seja avaliada como sendo verdadeira.

Abaixo, consta alguns exemplos da utilização destas funções. Lembre-se que para parâmetros negativos, é necessário envolvê-lo entre parênteses.

```
>> x' 1 2 3
        Left "Oops!"
>> x' 1 2 (-3)
        Right 1.0
>> _x' 1 2 3
        NaN
>> _x' 1 2 (-3)
        1.0
>> __x' 1 2 3
        Left "Oops!"
>> __x' 1 2 (-3)
        Right 1.0
```

Perceba que "_x' 1 2 3" gera um NaN (not a number), pois a sua descriminante é negativa.

A depender do caso em que o "let" for usado, ele pode melhorar a legibilidade do código, mas a maior desvantagem associada ao seu uso é que ele não trabalha bem com os guardas. Observe como ficaria a primeira implementação da função "qualificaPrioridade" com o "let".

```
qualificaPrioridade :: (Fractional a, Ord a) => a -> a -> a -> String
qualificaPrioridade i ani arr =
  let tarefa = i*ani*arr
  in
    if tarefa <=250 then "Pouco importante"
    else if tarefa <=500 then "Importância mediana"
    else if tarefa <= 750 then "Tarefa importante"
    else  "Tarefa extremamente importante"
```

Ou seja, o "let" baniu por completo a possibilidade de usar os guardas.

NOTE: Caso você seja uma pessoa observadora e inquieta, certamente você perguntou ao professor o porquê do uso do "Either" na assinatura das funções antes mesmo de checar se havia algum "note" explicando este detalhe. Para o seu alívio, este "Note" existe e você está lendo ele!

Pois bem, o "Either" se traduz como "ou isto, ou isso", portanto, já conseguimos observar que ele será usado quando tivermos uma função que poder retornar diferentes tipos de dados dependendo das suas entradas.

No exemplo apresentado, a função "qualificaPrioridade" pode retornar ou String ou (Floating a, Ord a). Perceba que String está sendo passada primeiro, portanto ela está a esquerda do espaço que divide "String" e "a" em "Either **String a"** na assinatura da função (esquerda = left).

De forma análoga, "a" está sendo passada em segundo lugar, logo, em relação ao espaço que divide "String" e "a" em "Either **String a"** ele está à direita (direita = right).

Tendo estes referenciais em mente, quando for necessário retornar uma String, é necessário dizer que você quer retornar o tipo à esquerda do espaço, caso contrário, você irá retornar o tipo à direita do espaço.

Perceba também que é necessário pôr a expressão “(-b + sqrt delta) / (2 * a)” entre parênteses, caso contrário, o haskell tentará dividir dividir Right (-b + sqrt delta) por “2*a”.