Capítulo 4
Sistemas condicionais sofisticados

No capítulo 1 foi apresentado o “if-them-else statement” – um mecanismo usado para a criar fluxos condicionais simples, os quais são usados para a atribuição de valores a funções. No entanto, caso seja necessário realizar fluxos condicionais mais elaborados, complexos e elegantes, é mais recomendável usar ferramentas mais poderosas. Neste capítulo, aprenderemos duas delas.

4.1 Casamento de padrão

Se trata da especificação de um padrão para a análise de dados. Observe a função abaixo que analisa se um determinado inteiro é ou não 12 para compreender melhor este novo recurso.

```
1.verificador :: (Eq a,Num a) => a -> String --“a” tem que ser instância de “Eq” e de “Num”
2.verificador 12 = "100% doze"
3.verificador x = "0% doze"
```

Caso você teste o código acima, notará que quando a função “verificador” é chamado e tem como parâmetro o número 12, ele retorna “100% doze”. Caso o número seja um número arbitrário *x* diferente de 12, ele retornará “0% doze”. Ao contrário do que é comum de pensar à primeira vista, a utilização do casamento de padrão não se trata de uma redeclaração de uma função já existente, mas sim da definição de como ela se comportará em diferentes cenários.

No entanto, é importante perceber que para o casamento de padrão funcionar, é necessário que todas as “instâncias da função”, por assim dizer, precisam estar implementadas consecutivamente. Logo, o código abaixo gerará um erro, pois há uma *reimplementação* de uma função já existente.

```
1.verificador :: (Eq a,Num a) => a -> String
2.verificador 12 = "100% doze"
3.haskell = "é legal"
4.verificador x = "0% doze"
```

Uma coisa importante a ser notada é que a ordem do casamento de padrão importa, por exemplo, caso a função “verificador” seja implementada da seguinte maneira :

```
1.verificador :: (Eq a,Num a) => a -> String
2.verificador x = "0% doze"
3.verificador 12 = "100% doze"
```

“100% doze” nunca será impresso, já que o 1° padrão sempre será satisfeito independente do número.

Uma vantagem bastante notável do casamento de padrão é a facilidade que ele fornece aos programadores quando é necessário criar funções recursivas.

```
1.somaTudo :: Integral a => a -> a
2.somaTudo 1 = 1
3.somaTudo n = somaTudo (n-1) + n
```

Antes que a recursão seja explicada, é necessário que a lógica que a função emprega seja entendida. Como o próprio nome da função sugere, ela retorna o somatório dos primeiros “n” números partindo do 1. Uma vez que o somatório de 1 é ele próprio, podemos notar que o somatório de 1 à 2 é “somatório de 1 + 2”. Por sua vez, o somatório de 3 é “somatório de 2 + 3”, o que é igual à : somatório de 1 + somatório de 2 + 3”. Portanto, uma conjectura recursiva para o somatório seria :

Somatório(n) = Somatório(n-1) + n

Uma vez que o somatório de um é um, o caso base será 1 – ou seja, retorna ele próprio. Analisando agora o casamento de padrão, notamos que é primeiro é verificado se o parâmetro passado à função é 1, caso seja, será retornado 1. Caso contrário, há a chamada recursiva. Outro exemplo clássico :

```
1. fatorial :: Integral a => a -> a
2. fatorial 1 = 1
3. fatorial n = fatorial(n-1) * n
```

Uma vez que o fatorial de 1 é dado por ele mesmo, ele é um excelente caso base. A chamada recursiva consiste na multiplicação de (n-1) por n, uma vez que o fatorial nada mais é que o produto dos primeiros “n” inteiros.

4.1.1 Um parâmetro genérico nulo para o casamento de padrão

Um parâmetro genérico nulo é um parâmetro que sempre será ignorado pelo haskell. O haskell usa o sublinhado para representar este parâmetro. Observe os exemplos abaixo para entender melhor a sua utilização.

```
1.numerosLegais :: (Num a, Eq a) => a -> String -- “a” tem que ser instância de “Num” e de “Eq”
2.numerosLegais 1 = "Unidade"
3.numerosLegais 2 = "Vida"
4.numerosLegais 3 = "Unidade"
5.numerosLegais _ = "Outro numero legal"
```

Neste exemplo simples, é demonstrado que, desde que o tipo passado para a função pertença a classe de tipo Num e Eq e diferente de um, dois e três, a linha 5 sempre será executada e retornará “Outro número legal”.

Perceba que a utilização do sublinhado não é equivalente à de um número genérico *x,* pois o valor de *x* pode ser levado em consideração para alguma operação interna da função, enquanto o “valor” do sublinhado será ignorado. Embora o exemplo anterior não tenha retratado tão bem isto, pois a

utilização do “x” vs. “_” é, de fato, equivalente neste caso, já que a única utilidade da função é retornar uma string. No entanto, a diferença entre eles é mais evidente na função “somaTudo”. Observe :

```
1.somaTudo :: Integral a => a -> a
2.somaTudo 1 = 1
3.somaTudo _ = somaTudo (_-1) + _ -- Isto gerá um erro, pois o sublinhado não possui valor algum
```

Uma vez que o valor passado para a linha 3 é ignorado, não há forma de aproveitá-lo para realizar a soma.

4.1.2 Casamento de padrão com listas

O casamento de padrão também pode ser utilizado com listas. Observe :

```
goodSaying :: String -> String
goodSaying [] = "A lista está vazia !"
goodSaying ['C', 'S', 'S', 'M', 'L'] = "Uma lista escrita da forma convencional"
goodSaying ('N':'D':'S':'M':'D':[]) = "Uma lista escrita da forma como o haskell lê listas"
goodSaying _ = "Padrão de lista não reconhecido"
```

A função acima verifica se uma lista é vazia, caso seja, ela retornará “A lista está vazia!”, caso a lista seja composta exatamente pelas letras “CSSML” ou “NDSMD”, uma mensagem referente à forma com que a lista foi especificada durante a implementação da função será retornada, caso nenhum desses padrões sejam identificados, será retornado "Padrão de lista não reconhecido".

Uma coisa extremamente importante a ser notada no exemplo anterior é que existem diferentes formas de declarar uma lista, mas todas são equivalentes, por exemplo :

```
1. 'V':'R':'S':'N':'S':'M':'V':[] == ['V','R','S','N','S','M','V']
        True
2. "SMQLIVB" == ['S','M','Q','L','I','V','B']
        True
3. 'I' : "HS" == "IHS"
        True
```

Esta variedade de formas de expressar a mesma coisa é muito útil para o casamento de padrão, observe o código abaixo que lê um texto recursivamente e retorna a primeira letra maiúscula que encontrar.

```
1. leRecursivamente :: String -> String
2. leRecursivamente [] = "Não há letras maiúculas !"
3. leRecursivamente (x:xs) = if x >= 'A' &&  x <= 'Z' then "primeira: " ++ show x else leRecursivamente xs
4. >> leRecursivamente "amDg"
          "primeira: 'D'"
```

Note que a especificação da lista através de “x:xs” na linha 2 é extremamente útil, pois, a pré-anexação do caractere “x” à string “xs”, possibilita que a função “leRecursivamente” analise um caractere por vez (“x”), enquanto o restante da string (“xs”) será analisado somente na próxima chamada recursiva.

A seguir, observe mais alguns exemplos de casamento de padrões que envolvem listas :

```
1.head' :: (Show a) => [a] -> String
2.head' [] = "A lista está vazia !"
3.head' (x:xs) = show x

4.last' :: (Show a) => [a] -> String
5.last' [] = "A lista está vazia !"
6.last' (x:[]) = show x
7.last' (x:xs) = last' xs

8.tail' :: (Show a) => [a] -> String
9.tail' [] = "A lista está vazia !"
10.tail' (x:xs) = show xs

10.length' :: (Show a) => [a] -> Int
11.length' [] = 0
12.length' (x:xs) = 1 + length' xs
```

Das funções implementadas acima, a função «length’» merece uma atenção especial, pois, à primeira vista, pode parecer que na linha 12 há uma soma de um inteiro (1) com uma string (length' xs), mas se a função for analisada com cuidado, pode ser notado rapidamente que ela é o equivalente a um *for loop*, pois à cada chamada recursiva (o equivalente à uma iteração), realiza-se a soma entre um e o restante do tamanho da string, até que ela esteja vazia.

4.1.3 Referenciamento de uma lista completa em um casamento de padrão

Caso seja conveniente referenciar toda a lista durante o casamento de padrão, ao invés de ter acesso somente ao “head” e ao “tail” realizando (x:xs), basta usar um recurso simples para atingir este fim, observe :

```
1.head' :: (Show a) => [a] -> String
2.head' [] = "A lista está vazia !"
3.head' listaInteira@(x:xs) = "lista inteira: " ++ show listaInteira ++ " head: " ++ show x
>> head' [1,3,10,12,14]
        "lista inteira: [1,3,10,12,14] head: 1"
```

Como pode ser observado no exemplo acima, para que você seja capaz de referenciar toda a lista durante em uma função, você pode optar por usar a sintaxe :

<nomeFunção> <nomeLista>@(<nomeHead>:<nomeTail>) = <*statement*>

Claro que também existe a opção de simplesmente pré-anexar o “head” com o “tail”.

```
1.head' :: (Show a) => [a] -> String
2.head' [] = "A lista está vazia !"
3.head' (x:xs) = "lista inteira: " ++ show (x:xs) ++ " head: " ++ show x
>> head' [1,3,10,12,14]
        "lista inteira: [1,3,10,12,14] head: 1"
```

Porém está opção é menos elegante, além de ser ligeiramente mais custosa em termos de computação, uma vez que a pré-anexação teria de ser executada todas as vezes que “listaInteira” tivesse que ser referenciada.

NOTE : A capacidade de referenciar a lista inteira através do “@” diz respeito a lista que está sendo analisada durante a chamada recursiva. Ou seja, não significa que durante 14° chamada recursiva, você será capaz de referenciar a lista da chamada recursiva 0.

NOTE : Não é possível descrever uma lista como a concatenação de 2 listas para o casamento de padrão, por exemplo :

```
padraoEstranho [] = "lista vazia"
padraoEstrado([1,2,3] ++ [4,5,6]) = show 7 --gera um erro
```

4.2 Case expressions

Várias linguagens disponibilizam expressões do tipo *case-switch* e o haskell não é diferente. Este tipo de expressão é bastante útil caso seja necessário fazer fluxos condicionais simples e exaustivos, como as opções de discagem de centrais de suporte, por exemplo. Uma vez que este exemplo é o mais clássico de todos, o código a seguir implementa um pequeno sistema deste tipo para que este mecanismo seja melhor compreendido, observe atentamente a sua sintaxe.

```
centralTelefonica :: Integral a => a -> String
centralTelefonica opcao = case opcao of
  1 -> "Parabens ! Voce discou 1"
  2 -> "Parabens ! Voce discou 2"
  3 -> "Parabens ! Voce discou 3"
  _ -> "Que pena, voce discou outro numero !"
```

Como pode ser observado, a sintaxe de uma *case-expression* é bastante simples, basta que após o “=” seja posto :

case <nomeParametro> of

e em seguida, basta que haja a especificação das opções permitidas e seus respectivos retornos após a seta (“->”). Perceba que o uso da indentação é obrigatório e a sua utilização foi indicada no exemplo através dos espaços amarelos. Além disto, como já discutido no capítulo 1, não importa a quantidade de espaços usados para a indentação, desde que a quantidade adotada se mantenha uniforme em todo o código. No caso do exemplo acima, foi optado pelo uso de dois espaços.