Paradigmas de Programação

Paradigma Funcional com Haskell (Parte 2)

Introdução

Este material contém notas de aula sobre o *Paradigma Funcional*. O objetivo é apresentar os conceitos deste paradigma utilizando a linguagem de programação Haskell, em continuidade ao material da Parte 1.

A Linguagem Haskell (continuação)

Anteriormente, aspectos do Paradigma Funcional e uma introdução à Linguagem Haskell foram vistos. Neste material, segue-se apresentando os principais recursos da linguagem.

Estruturas de Decisão

Haskell permite expressar condições com **`if-then-else`**, estrutura presente em muitas linguagens de programação conhecidas, e **guardas**, barras verticais representadas pelo caractere "`|`" seguido de uma expressão lógica a ser avaliada.

Uma construção if-then-else terá a seguinte estrutura:

`if ExpressãoBool then Expressao`$_1$ **`else Expressao`**$_2$

Uma particularidade da construção **`if-then-else`** na linguagem Haskell é a obrigatoriedade do bloco `else`.

Um exemplo com **`if-then-else`**:

```
resultado1 :: Int -> String
resultado1 x = if x > 0 then "Positivo" else "Negativo"
```

O uso de guardas ocorre da seguinte forma:

```
funcaoDeclarada n
      | ExpressaoBool1 = Expressao1
      | ExpressaoBool2 = Expressao2
      | ExpressaoBool3 = Expressao3
      | (...)
      | ExpressaoBooln = Expressaon
```

Uma particularidade das guardas é a necessidade de alinhamento entre elas. Toda guarda precisa estar abaixo da outra, na mesma posição, e com pelo menos um espaço em branco de distância do início da linha. Caso isso não aconteça, um erro será exibido.

Na última linha da declaração acima, **`ExpressaoBool`**$_n$ pode ser substituída por **`otherwise`**, que significa *caso contrário*.

Um exemplo com guardas:

```
resultado2 :: Int -> String
resultado2 x
  | x > 0     = "Positivo"
  | otherwise = "Negativo"
```

Uma terceira forma válida poderia combinar o uso de **`if-then-else`** com expressões lambda:

```
resultado3 :: Int -> String
resultado3 = \x -> if x > 0 then "Positivo" else "Negativo"
```

Um ponto negativo das funções definidas é a chamada para o valor `0` (zero). Neste caso, as funções `resultado1`, `resultado2` e `resultado3` retornam "`Negativo`", o que claramente não condiz com a realidade.

Para melhorar o exemplo, pode-se criar uma função que verifica o sinal de um número. A função deve retornar `1`, se positivo, `-1`, se negativo, e `0` se for neutro (se o número for igual a zero).

A função `sinal1`, definida abaixo, resolverá o problema proposto. Sugere-se salvá-la em um arquivo intitulado `exemplo2.hs`.

```
sinal1 x = if x < 0 then -1 else if x > 0 then 1 else 0
```

A função `sinal1` combina `if`s aninhados pois, se o valor da expressão lógica não avaliar para `True`, é necessário realizar mais um teste, já que o número ainda poderá ser positivo (caso em que retornará `1`) ou nulo (caso em que retornará `0`).

A partir do GHCi, o arquivo `exemplo2.hs` pode ser carregado digitando **`:l exemplo2.hs`**. Após carregá-lo, é possível executar a função com diferentes valores, tais como os sugeridos abaixo:

```
sinal1 9
sinal1 0
sinal1 (-9) -- o valor negativo precisa estar entre parênteses!
```

Outra implementação válida para a mesma função, mas utilizando guardas, está no quadro a seguir:

```
sinal2 x
  | x > 0 = 1
  | x < 0 = -1
  | otherwise = 0
```

Recursividade

Haskell não dispõe de estruturas de repetição. Assim, a forma de realizar iterações é por meio de recursão. A recursividade é uma técnica em que uma função pode chamar a si mesma com argumentos diferentes até atingir uma condição de parada. Isso permite expressar algoritmos de maneira simples e eficiente, aproveitando a imutabilidade dos dados e a avaliação preguiçosa.

Por exemplo, a função de cálculo do Máximo Divisor Comum – MDC em Haskell pode ser definida de forma recursiva da seguinte maneira:

```
mdc a b
   | a > b = mdc (a - b) b
   | a < b = mdc a (b - a)
   | otherwise = a
```

As quatro implementações do cálculo do fatorial de n, presentes no quadro a seguir, também usam recursividade:

```
fatorial1 n = if n == 0 then 1 else n * fatorial1 (n - 1)

fatorial2 n
   | n == 0 = 1
   | otherwise = n * fatorial2 (n - 1)

fatorial3 0 = 1
fatorial3 n = n * fatorial3 (n - 1)

fatorial4 n = product [1..n]
```

Percebe-se que as funções `fatorial1`, `fatorial2` e `fatorial3` chamam explicitamente a própria função. A função `fatorial3` usa um recurso chamado de ***casamento de padrões***, uma forma de definir uma função através de múltiplas equações, cada uma correspondendo a um padrão específico de entrada. No caso da função `fatorial3`, a primeira equação define o

caso base quando o argumento é igual a 0, retornando 1. A segunda equação trata do caso geral, multiplicando n pelo resultado da chamada recursiva de `fatorial3` com $n - 1$ como parâmetro. Já a função `fatorial4` não define uma recursão explicitamente, mas usa a função `product`, que multiplica todos os valores de uma lista recursivamente.

DESAFIO

Declarar uma função para retornar o *n*-ésimo termo da sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8...

Algumas respostas válidas para o desafio estão no quadro a seguir:

```
fibonacci1 :: Integer -> Integer
fibonacci1 0 = error "O n-ésimo valor da sequência deve ser maior que zero."
fibonacci1 1 = 0
fibonacci1 2 = 1
fibonacci1 n = fibonacci1 (n-1) + fibonacci1 (n-2)

fibonacci2 n
  | n == 0 = error "O n-ésimo valor da sequência deve ser maior que zero."
  | n == 1 = 0
  | n == 2 = 1
  | otherwise = fibonacci2 (n-1) + fibonacci2 (n-2)
```

Nota-se o uso da função `error` nos casos em que $n = 0$. No exemplo dado, as funções lançarão uma exceção com a mensagem "O n-ésimo valor da sequência deve ser maior que zero.", pois o valor de n precisa ser maior ou igual a 1.

Let (Seja)

`let` é uma palavra reservada na linguagem Haskell – e em outras linguagens funcionais – para definir declarações aninhadas. A expressão tem escopo definido à direita da declaração, isto é, o valor do parâmetro será utilizado somente nas expressões ali definidas.

Exemplos:

```
let x = 4 in x + 9
let x = (-8) in sinal1 x
let a = 5; b = 171 in mdc a b
let n = 5 in fatorial1 n
let x = (-99) in x + 19 - x^2 + 99 - x * 3 - (x `mod` 99)
letComVariavelLocal x y = let z = x + y in z * 2
```

Quando usar `let`?

1. Em experimentações interativas, como nos exemplos anteriores.
2. Quando o valor testado será aplicado em vários pontos das expressões definidas, como acontece no penúltimo exemplo acima.
3. Quando é preciso definir uma variável ou uma função local, ou seja, que será usada apenas em um escopo limitado, como ocorre no último exemplo acima.

PARA PENSAR

Na penúltima expressão, o valor `-99` precisaria substituir x em todas as suas ocorrências, que são quatro. Valeria a pena repetir o valor `-99` quatro vezes? E se o teste for refeito com o valor `33`? O que é mais prático?

Listas

Listas são conjuntos homogêneos de dados, ou seja, todos os seus dados têm o mesmo tipo. O quadro abaixo tem alguns exemplos que devem ser testados diretamente no GHCi:

```
[1, 2, 3]
:t [1, 2, 3]
-- Poderia ser criada assim:
[1..3]
[]
[True, 1==2, False]
"Ricardo"
1:2:[]
[1] ++ [2]
('a':'b':[] == "ab") == ('a':'b':[] == ['a', 'b'])
:t [1,2,4.9, 5.004, pi]
"A" < "a"
'R':'i':"card" ++ ['o']
```

Toda lista pode ser dividida em duas partes: cabeça (*head*) e cauda (*tail*). A cabeça é o primeiro elemento da lista, e a cauda é a própria lista sem o primeiro elemento. Assim, a lista `[1,2,3]` possui `1` na cabeça e `[2,3]` na cauda.

O operador `:` (*dois pontos*) constrói uma nova lista. A sintaxe do operador é `elem:list`. Assim, a lista resultante da operação é composta por todos os elementos de `list`, acrescidos do elemento `elem` na cabeça da lista (como primeiro elemento).

- `1:[]` resulta em `[1]`
- `1:2:3:4:5:6:7:8:[9]` resulta em `[1,2,3,4,5,6,7,8,9]`

Graças ao operador de construção de listas, em diversos casos costuma-se representar listas na forma `(x:xs)`, onde `x` é um elemento e `xs` é a lista sem o elemento `x`. As funções `cabeca` e `cauda`, definidas no quadro a seguir, retornam a cabeça e a cauda de uma lista, respectivamente. Essas funções são muito parecidas com as funções head e tail, já existentes no módulo `Prelude`.

```
-- Função que obtém a cabeça da lista
cabeca [] = error "Lista vazia!"
cabeca (x:xs) = x
```

```
-- Função que obtém a cauda da lista
cauda [] = error "Lista vazia!"
cauda (x:xs) = xs
```

O operador `++` (*mais mais*) concatena listas. A sintaxe do operador é `list1++list2`. Assim, a lista resultante da operação é composta pela união de todos os elementos das listas `list1` e `list2`, nesta ordem.

- `[1] ++ []` resulta em `[1]`
- `[1, 2] ++ [3, 5] ++ [8, 9]` resulta em `[1,2,3,5,8,9]`

A ordem dos elementos da lista faz diferença. Isso pode ser percebido realizando os dois testes a seguir na ferramenta GHCi:

```
[3,4] == [4,3]
[3,4] == [3,4]
```

Listas podem ser criadas por intervalo, como acontece nos exemplos abaixo:

```
[-10..10]
[2,4..20] -- e se fosse até o 21?
[1,3..20]
[100,10..] -- o que será que vai acontecer aqui? Pressione CTRL+C para interromper!
```

DESAFIO

Como implementar, sem o apoio de outras funções já definidas:

1. **`tamLista`**, função que deve receber uma lista e informar quantos elementos ela tem?
2. **`trocaPorZero`**, função que deve trocar todos os valores de uma lista por zero?
3. **`triplicaValores`**, função que deve triplicar todos os valores de uma lista?

Compreensão de Lista (*List Comprehension*)

Compreensão de Lista é um recurso disponível em Haskell que permite declarar uma lista em notação inspirada na notação de conjuntos (DU BOIS, s.d.).

Os três exemplos do quadro a seguir podem ser testados diretamente no GHCi:

```
[2 * x | x <- [0..10]]
[x | x <- [0..100], x `mod` 2 == 1 ] -- que lista é essa?
[Data.Char.toLower x | x <- "Ricardo de la Rocha Ladeira",
Data.Char.isUpper x]
```

Se desejável, pode-se importar o módulo `Data.Char` antes, depois executar o terceiro comando da seguinte forma:

```
import Data.Char
[toLower x | x <- "Ricardo de la Rocha Ladeira", isUpper x]
```

A primeira parte da declaração consiste no mapeamento, representando como será a expressão de saída. Pegando o primeiro exemplo, seria a expressão `2 * x`, ou seja, o dobro do valor de `x`.

A segunda parte, logo após o *pipe*, indica a que conjunto de entrada a variável pertence. Com base no primeiro exemplo, seria a expressão `x <- [0..10]`. Este trecho também pode ser chamado de *gerador* (DU BOIS, s.d.).

Pode haver ainda uma terceira parte, após o gerador, indicando qual será a filtragem realizada (o predicado). Esta parte é opcional e, quando presente, deve ser precedida por vírgula. Ela não está presente no primeiro exemplo, mas aparece nos outros dois. Tomando como base o segundo exemplo, a expressão utilizada filtra os valores ímpares, eliminando os demais (os pares).

Funções de Alta Ordem

Já que funcionam como dados de tipos quaisquer, em Haskell, funções também podem receber funções como parâmetro ou mesmo retornar funções. As funções que recebem funções em parâmetro(s) ou retornam funções são ditas ***de alta ordem*** ou ***de ordem superior*** (***high order functions***).

Algumas funções de alta ordem conhecidas são:

- `filter`: recebe um predicado e uma lista e faz uma filtragem na lista com base no predicado[1]. Neste caso, o predicado (representado por uma função) será aplicado a cada elemento da lista, realizando assim o filtro e retornando uma nova lista.

```
filtraMenores _ [] = []
filtraMenores n xs = filter (< n) xs
-- A função filtraMenores recebe um número n e uma lista. Ela usa a função filter para definir se cada elemento da lista é menor que o valor n, O retorno é uma lista com todos os valores menores que n.
-- Exemplos de uso:
filtraMenores 4 [1,2,3,4,5]
filtraMenores 3 [2..99]
-- É possível usar a função filter diretamente. Por exemplo:
filter (<4) [2..190]
```

- `map`: realiza o mapeamento de uma função recebida como parâmetro a cada elemento de uma lista também recebida como parâmetro. Em outras palavras, aplica uma função a cada elemento da lista.

```
-- Para testar no GHCi:
map (+1) [1,2,3]
import Data.Char
```

[1] Como mencionado nas Notas de Aula sobre o Paradigma Lógico com Prolog, predicado é uma função cujo contradomínio é um conjunto formado por dois elementos: verdadeiro e falso.

```
map chr [82,105,99,97,114,100,111] -- O que faz esta chamada à função map?
```

- `foldr` (não coberta neste material)
- `foldl` (não coberta neste material)
- `flip` (não coberta neste material)
- `zipWith` (não coberta neste material)

Como ficaria a implementação da função `triplicaValores`, solicitada no quadro de Desafio, utilizando `map`?

Cabe citar que `map` e `filter` são casos especiais de funções mais gerais chamadas de `fold`.

DESAFIO

Como elaborar um código equivalente ao código abaixo utilizando `filter`?

```
[ toLower x | x <- "Ricardo de la Rocha Ladeira", isUpper x]
```

Resposta:

```
map (toLower) (filter (isUpper) "Ricardo de la Rocha Ladeira")
```

Exemplo Real

Um exemplo real de uso da função `map` na resolução de um desafio de Segurança:

1. O desafio começa com um arquivo cheio de números, constantes no quadro a seguir:

```
112 100 60 59 69 60 107 100 93 56 75 96 89 58  105 42 41 35 38 40 110
101 92 93 94 90 90 40 41 25 27 58 94 94 62 59 60 25 100 110 105 91 85
108 101 89 91 85 92 91 112 85 107 105 87 100 90 101 85 104 91 102 98
87 89 91 34 22 103 107 91 22 102 91 104 105 95 105 106 91 100 89 95
87 85 91 95 100 23 57 87 105 101 85 106 91 100 94 87 85 107 105 87 90
101 85 90 91 85 87 98 93 107 99 87 85 92 101 104 99 87 85 87 85 106
```

```
87 88 91 98 87 85 87 105 89 95 95 48 99 107 95 106 101 85 88 91 99 85
102 91 100 105 87 90 101 25 58 91 85 106 101 90 87 85 92 101 104 99
87 34 22 87 103 107 95 85 108 87 95 85 105 107 87 85 92 98 87 93 48
85 90 91 105 87 92 95 101 55 105 89 95 95 62 73 85 80 39 46 41 42 60
58 42 27 92 105 70 101 65 57 108 91 41 32 41 32 60 25 88 94
```

2. A principal suspeita é de que todos os números representam os códigos ASCII (*American Standard Code for Information Interchange*) de caracteres. Assim, basta converter esses códigos usando a função `Data.Char.chr`. Exemplo: `Data.Char.chr 101`
3. No entanto, como a função `chr` do módulo `Data.Char` converte cada número em caractere, é necessário colocar o conjunto de números em uma lista e fazer a conversão utilizando `map`. Assim, para formatar os dados como uma lista, sugere-se o uso do terminal, antes de abrir o GHCi:

```
echo [112 100 60 59 69 60 107 100 93 56 75 96 89 58  105 42 41 35 38
40 110 101 92 93 94 90 90 40 41 25 27 58 94 94 62 59 60 25 100 110
105 91 85 108 101 89 91 85 92 91 112 85 107 105 87 100 90 101 85 104
91 102 98 87 89 91 34 22 103 107 91 22 102 91 104 105 95 105 106 91
100 89 95 87 85 91 95 100 23 57 87 105 101 85 106 91 100 94 87 85 107
105 87 90 101 85 90 91 85 87 98 93 107 99 87 85 92 101 104 99 87 85
87 85 106 87 88 91 98 87 85 87 105 89 95 95 48 99 107 95 106 101 85
88 91 99 85 102 91 100 105 87 90 101 25 58 91 85 106 101 90 87 85 92
101 104 99 87 34 22 87 103 107 95 85 108 87 95 85 105 107 87 85 92 98
87 93 48 85 90 91 105 87 92 95 101 55 105 89 95 95 62 73 85 80 39 46
41 42 60 58 42 27 92 105 70 101 65 57 108 91 41 32 41 32 60 25 88 94]
| tr " " ","
```

4. A lista é gerada e pode ser incluída em uma variável para ser usada no GHCi.

```
list=[112,100,60,59,69,60,107,100,93,56,75,96,89,58,105,42,41,35,38,40
,110,101,92,93,94,90,90,40,41,25,27,58,94,94,62,59,60,25,100,110,105,9
1,85,108,101,89,91,85,92,91,112,85,107,105,87,100,90,101,85,104,91,102
,98,87,89,91,34,22,103,107,91,22,102,91,104,105,95,105,106,91,100,89,9
5,87,85,91,95,100,23,57,87,105,101,85,106,91,100,94,87,85,107,105,87,9
0,101,85,90,91,85,87,98,93,107,99,87,85,92,101,104,99,87,85,87,85,106,
```

87,88,91,98,87,85,87,105,89,95,95,48,99,107,95,106,101,85,88,91,99,85,102,91,100,105,87,90,101,25,58,91,85,106,101,90,87,85,92,101,104,99,87,34,22,87,103,107,95,85,108,87,95,85,105,107,87,85,92,98,87,93,48,85,90,91,105,87,92,95,101,55,105,89,95,95,62,73,85,80,39,46,41,42,60,58,42,27,92,105,70,101,65,57,108,91,41,32,41,32,60,25,88,94]

5. A partir disso, já na ferramenta `ghci`, pode-se converter todos os valores inteiros da lista em caracteres (tipo `Char`) usando a função `chr` como parâmetro de `map`:

```
map Data.Char.chr list
```

6. O retorno obtido não parece legível:

"pd<;E<kd]8K`Y:i*)#&(ne\\]^ZZ()\EM\ESC:^^>;<\EMdni[UleY[U\\[pUkiWdZeUh[fbWY[\"\SYNgk[\SYNf[hi_ij[dY_WU[_d\ETB9WieUj[d^WUkiWZeUZ[UWb]kcWU\\ehcWUWUjWX[bWUWiY__0ck_jeUX[cUf[diWZe\EM:[UjeZWU\\ehcW\"\SYNWgk_UlW_UikWU\\bW]0UZ[iW_e7iY__>IUP'.)*<:*\ESC\\iFeA9l[)) <\EMX^"

7. Para encurtar o trabalho, a alternativa adotada será rotacionar os valores da tabela ASCII dez unidades acima do que consta na lista, pois os valores originais foram codificados em dez unidades abaixo do valor. Assim, o comando completo que é capaz de obter a mensagem original é:

```
map Data.Char.chr (map (+10) list)
```

8. O resultado obtido é a seguinte mensagem:

```
"znFEOFungBUjcDs43-02xofghdd23#%DhhHEF#nxse_voce_fez_usando_replace,
que
persistencia_ein!Caso_tenha_usado_de_alguma_forma_a_tabela_ascii:muit
o_bem_pensado#De_toda_forma,
aqui_vai_sua_flag:_desafioAsciiHS_Z1834FD4%fsPoKCve3*3*F#bh"
```

9. Portanto, a resposta desejada (chamada de *flag*) é **`_desafioAsciiHS_`**

Definição de Sinônimo de Tipo

A palavra reservada `type` introduz um sinônimo para um tipo que já existe e usa os mesmos construtores de dados (HASKELL, 2021). Desta forma, os exemplos a seguir são válidos para definir sinônimos para tipos que já existem:

```
type String = [Char]
type Nome = String
type Nota = Float
type Idade = Int
```

Assim, em qualquer função que o tipo `String` for aceito, um elemento de tipo `Nome` poderá ser usado sem problemas. O quadro a seguir define as funções verNota e verNome, cujas assinaturas utilizam sinônimos de tipos já existentes:

```
verNota :: (Nome, Nota) -> Nota
verNota (a, b) = b
verNome :: (Nome, Nota) -> Nome
verNome (a, b) = a
```

Os testes abaixo são válidos:

```
verNota ("Ricardo", 10.0)
verNome ("Ricardo", 10.0)
```

Mãos à obra! Agora, crie a função `verIdade` (**`verIdade :: (Nome, Nota, Idade) -> Idade`**). A resposta está disponível no arquivo exemplo2.hs.

Conclusão

Esta segunda parte sobre o Paradigma Funcional com Haskell expandiu o entendimento sobre os conceitos fundamentais desse paradigma de programação. Foram explorados aspectos avançados da linguagem Haskell, como estruturas de decisão com `if-then-else` e guardas, recursividade e compreensão de lista. Aprofundou-se o conhecimento sobre funções de alta ordem e suas aplicações, além de discutir a definição de sinônimos de tipo usando a palavra reservada `type`.

Ficou destacada a elegância e a legibilidade da programação funcional, bem como sua ênfase na imutabilidade e na avaliação de expressões. Esses princípios são essenciais para garantir a

confiabilidade e a previsibilidade do código, facilitando a manutenção e o entendimento de quem estuda e trabalha com linguagens funcionais.

É importante destacar que a linguagem Haskell dispõe de diversos outros recursos não cobertos por este material. Assim, o leitor é encorajado a buscar também outras fontes para aprofundar seus conhecimentos na linguagem.

Exercícios

1) Crie a função `maioridade` (`maioridade :: Int -> Bool`), que deve retornar `True` se o valor de `n` for maior ou igual a 18 e `False` caso seja inferior. Exiba uma mensagem de erro se o valor informado for negativo. Faça o exercício utilizando guardas.

2) Resolva o exercício acima utilizando `if-then-else`.

3) Teste as duas funções declaradas nos exercícios anteriores com e sem `let`.

4) Crie a função `modulo` ($|x|$), que tira o sinal de um número.

5) (POSCOMP 2010) Com base nos conhecimentos sobre as linguagens de programação funcionais, considere as afirmativas a seguir.

I. Uma linguagem de programação funcional tem o objetivo de imitar as funções matemáticas, ou seja, os programas são definições de funções e de especificações da aplicação dessas funções.

II. Nas linguagens funcionais, os dados e as rotinas para manipulá-los são mantidos em uma mesma unidade, chamada objeto. Os dados só podem ser manipulados por meio das rotinas que estão na mesma unidade.

III. As rotinas de um programa do paradigma funcional descrevem ações que mudam o estado das variáveis do programa, seguindo uma sequência de comandos para o computador executar.

IV. A linguagem Lisp é um exemplo do paradigma funcional de programação.

Assinale a alternativa correta.

a) Somente as afirmativas I e IV são corretas.
b) Somente as afirmativas II e III são corretas.
c) Somente as afirmativas III e IV são corretas.
d) Somente as afirmativas I, II e III são corretas.
e) Somente as afirmativas I, II e IV são corretas

6) (POSCOMP 2014) Sobre linguagens puramente funcionais, considere as afirmativas a seguir.

I. Programas são definições de funções e de especificações de aplicações dessas funções. A execução desses programas consiste em avaliar tais funções.
II. A avaliação de uma função sempre produz o mesmo resultado, quando invocada com os mesmos argumentos.
III. A passagem de parâmetros para uma função pode ocorrer de duas formas: por valor ou por referência.
IV. O estado interno de uma função é definido por seus parâmetros formais e por variáveis locais estáticas. Estas últimas podem armazenar valores calculados em invocações anteriores da função.

Assinale a alternativa correta.

a) Somente as afirmativas I e II são corretas.
b) Somente as afirmativas I e IV são corretas.
c) Somente as afirmativas III e IV são corretas.
d) Somente as afirmativas I, II e III são corretas.
e) Somente as afirmativas II, III e IV são corretas.

7) (POSCOMP 2012) Em linguagens de programação declarativas, em especial aquelas que seguem o paradigma funcional, a lista é uma estrutura de dados fundamental. Uma lista representa coleções de objetos de um único tipo, sendo composta por dois elementos: a cabeça (head) e o corpo (tail), exceto quando está vazia. A cabeça é sempre o primeiro elemento e o corpo é uma lista com os elementos da lista original, excetuando-se o primeiro elemento. O programa Haskell, a seguir, apresenta uma

função que utiliza essa estrutura de dados.

```
poscomp :: [Int] -> [Int]
poscomp [] = []
poscomp [x] = [x]
poscomp (a:b:c)
    | a > b = b : (a : poscomp c)
    | otherwise = a : (b : poscomp c)
```

Uma chamada a esta função através da consulta **`poscomp [5,3,4,5,2,1,2,3,4]`** produzirá o resultado:

a) `[1,2,2,3,3,4,4,5,5]`
b) `[3,5,4,5,1,2,2,3,4]`
c) `[5,3,4,5,2,1,2,3,4]`
d) `[5,4,3, 2,1]`
e) `[5,3,4,2,1]`

8) Resolva os exercícios da Lista #6.

Referências

DU BOIS, André Rauber. Programação Funcional com a Linguagem Haskell. Disponível em: http://www.inf.ufpr.br/andrey/ci062/ProgramacaoHaskell.pdf. Acesso em: 18 mar. 2024.

HASKELL. Type. 2021. https://wiki.haskell.org/Type#Type_and_newtype. Acesso em: 20 mar. 2024.