Paradigmas de Programação

Fabrício Olivetti de França

28 de Junho de 2018

Recursão

Recursão

A recursividade permite expressar ideias declarativas.

Composta por um ou mais casos bases (para que ela termine) e a chamada recursiva.

$$n! = n.(n-1)!$$

Recursão

Caso base:

$$1! = 0! = 1$$

Recursão

Para $n = 3$:

3! = 3 . 2! = 3 . 2 . 1! = 3 . 2 . 1 = 6

Recursão

```haskell
fatorial :: Integer -> Integer
fatorial 0 = 1
fatorial 1 = 1
fatorial n = n * fatorial (n-1)
```

Recursão

```haskell
fatorial :: Integer -> Integer
fatorial 0 = 1
fatorial 1 = 1
fatorial n = n * fatorial (n-1)
```

Casos bases primeiro!!

Fatorial

O Haskell avalia as expressões por substituição:

```
> fatorial 4
      => 4 * fatorial 3
      => 4 * (3 * fatorial 2)
      => 4 * (3 * (2 * fatorial 1))
      => 4 * (3 * (2 * 1))
      => 4 * (3 * 2)
      => 4 * 6
      => 24
```

Fatorial

Ao contrário de outras linguagens, ela não armazena o estado da chamada recursiva em uma pilha, o que evita o estouro da pilha.

```
> fatorial 4
      => 4 * fatorial 3
      => 4 * (3 * fatorial 2)
      => 4 * (3 * (2 * fatorial 1))
      => 4 * (3 * (2 * 1))
      => 4 * (3 * 2)
      => 4 * 6
      => 24
```

Fatorial

A pilha recursiva do Haskell é a expressão armazenada, ele mantém uma pilha de expressão com a expressão atual. Essa pilha aumenta conforme a expressão expande, e diminui conforme uma operação é avaliada.

```
> fatorial 4
      => 4 * fatorial 3
      => 4 * (3 * fatorial 2)
      => 4 * (3 * (2 * fatorial 1))
      => 4 * (3 * (2 * 1))
      => 4 * (3 * 2)
      => 4 * 6
      => 24
```

Recursão Caudal

Mesmo a pilha de expressão pode estourar!

Recursão caudal também é útil no Haskell.

Recursão Caudal

A **recursão caudal** é uma função recursiva cujo valor de retorno consiste **apenas** da chamada recursiva:

```
f x = f x'
```

```
g x y = g x' y'
```

Recursão Caudal

Contra-exemplos de recursão caudal:

```
f x = 1 + f x'
```

```
g x y = y * (g x' y')
```

```
f x = f x' + f x''
```

Recursão Caudal

A função `fatorial` pode ser reescrita como:

```
fatorial :: Integer -> Integer
fatorial 0 = 1
fatorial 1 = 1
fatorial n = fatorial' n 1
  where
    fatorial' 1 r = r
    fatorial' n r = fatorial' (n-1) (n*r)
```

Recursão Caudal

Dessa forma temos:

```
> fatorial 4
      => fatorial' 4 1
      => fatorial' 3 (4*1)
      => fatorial' 2 (3*4*1)
      => fatorial' 1 (2*3*4*1)
      => (2*3*4*1)
      => 24
```

Recursão Caudal

Por que o primeiro parâmetro é avaliado e o segundo mantém uma expressão?

Precisamos saber o valor do primeiro parâmetro para o Pattern Matching, o segundo só é necessário no final

Recursão Caudal

Podemos forçar a avaliação com a função seq:

```
fatorial :: Integer -> Integer
fatorial 0 = 1
fatorial 1 = 1
fatorial n = fatorial' n 1

fatorial' 1 r = r
fatorial' n r = r' `seq` fatorial' (n-1) r'
  where r' = n*r
```

Recursão Caudal

Dessa forma temos:

```
> fatorial 4
      => fatorial' 4 1
      => fatorial' 3 4
      => fatorial' 2 12
      => fatorial' 1 24
      => 24
```

Máximo Divisor Comum

O algoritmo de Euclides para encontrar o Máximo Divisor Comum (*greatest common divisor* - gcd) é definido matematicamente como:

```
gcd :: Int -> Int -> Int
gcd a 0 = a
gcd a b = gcd b (a `mod` b)
```

Máximo Divisor Comum

```
> gcd 48 18
    => gcd 18 12
    => gcd 12 6
    => gcd 6 0
    => 6
```

Máximo Divisor Comum

Se garantirmos que ambos os argumentos são positivos, podemos reescrever a função como:

```
gcd :: Int -> Int -> Int
gcd a b | a == b      = a
        | a > b       = gcd (a-b) b
        | otherwise   = gcd a (b-a)
```

Máximo Divisor Comum

```
> gcd 48 18
    => gcd 30 18
    => gcd 12 18
    => gcd 12 6
    => gcd 6 6
    => 6
```

Um passo extra ☹, mas utilizando subtração ao invés de divisão 😄

Multiplicação Etíope (0.5 pto)

A multiplicação Etíope de dois números m, n é dado pela seguinte regra:

- Se m for par, o resultado é a aplicação da multiplicação em $m/2, n * 2$.
- Se m for ímpar, o resultado a aplicação da multiplicação em $m/2, n * 2$ somados a n.
- Se m for igual a 1, retorne n.

Multiplicação Etíope (0.5 pto)

Exemplo:

m	n	r
14	12	0
7	24	24
3	48	72
1	96	168

Multiplicação Etíope (0.5 pto)

Implemente o algoritmo recursivo da Multiplicação Etíope. Em seguida, faça a versão caudal.

Recursão em Listas

Funções recursivas em listas

Podemos também fazer chamadas recursivas em listas, de tal forma a trabalhar com apenas parte dos elementos em cada chamada:

```
sum :: Num a => [a] -> a
sum [] = 0
sum ns = ???
```

Funções recursivas em listas

Podemos também fazer chamadas recursivas em listas, de tal forma a trabalhar com apenas parte dos elementos em cada chamada:

```
sum :: Num a => [a] -> a
sum [] = 0
sum ns = (head ns) + sum (tail ns)
```

Por que não usar Pattern Matching?

Funções recursivas em listas

Podemos também fazer chamadas recursivas em listas, de tal forma a trabalhar com apenas parte dos elementos em cada chamada:

```
sum :: Num a => [a] -> a
sum []     = 0
sum (n:ns) = n + sum ns
```

Exercício

Faça a versão caudal dessa função:

```
sum :: Num a => [a] -> a
sum []     = 0
sum (n:ns) = n + sum ns
```

Produtória

Como ficaria a função `product` baseado na função `sum`:

```
sum :: Num a => [a] -> a
sum []     = 0
sum (n:ns) = n + sum ns
```

Produtória

Como ficaria a função `product` baseado na função `sum`:

```
product :: Num a => [a] -> a
product []     = 0
product (n:ns) = n + sum ns
```

Produtória

Como ficaria a função `product` baseado na função `sum`:

```
product :: Num a => [a] -> a
product []     = 1
product (n:ns) = n * product ns
```

Tamanho

E a função `length`?

```
sum :: Num a => [a] -> a
sum []     = 0
sum (n:ns) = n + sum ns
```

Tamanho

E a função `length`?

```
length :: [a] -> Int
length []     = 0
length (n:ns) = 1 + length ns
```

Padrões de Programação

Reparem que muitas soluções recursivas (principalmente com listas) seguem um mesmo esqueleto. Uma vez que vocês dominem esses padrões, fica fácil determinar uma solução.

Nas próximas aulas vamos criar funções que generalizam tais padrões.

Invertendo uma lista

Considere a função `reverse`:

```
> :t reverse
reverse :: [a] -> [a]
> reverse [1,2,3]
[3,2,1]
```

Como poderíamos implementá-la?

Invertendo uma lista

Vamos começar pelo caso base, o inverso de uma lista vazia, é vazia:

```
reverse :: [a] -> [a]
reverse [] = []
```

Invertendo uma lista

Vamos começar pelo caso base, o inverso de uma lista com um elemento, é ela mesma:

```
reverse :: [a] -> [a]
reverse []  = []
reverse [x] = [x]
```

Invertendo uma lista

Vamos começar pelo caso base, o inverso de uma lista com dois elementos é:

```
reverse :: [a] -> [a]
reverse []    = []
reverse [x]   = [x]
reverse [x,y] = [y,x]
```

Invertendo uma lista

Vamos começar pelo caso base, o inverso de uma lista com três elementos é:

```
reverse :: [a] -> [a]
reverse []      = []
reverse [x]     = [x]
reverse [x,y]   = [y,x]
reverse [x,y,z] = [z,y,x]
```

Invertendo uma lista

Esse último caso base nos dá uma ideia de como generalizar! Note que:

```
> reverse [1,2,3] == reverse [2,3] ++ [1]
```

Invertendo uma lista

Vamos começar pelo caso base, o inverso de uma lista com três elementos é:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Função zip

Lembrando a função `zip` da aula anterior:

```
> zip [1,2,3] [4,5]
[(1,4), (2,5)]
```

Função zip

Temos como casos bases:

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _ = []
zip _ [] = []
```

Função zip

E o caso recursivo:

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _          = []
zip _ []          = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Exercício

Crie uma função recursiva chamada `insert` que insere um valor `x` em uma lista `ys` ordenada de tal forma a mantê-la ordenada:

```
insert :: Ord a => a -> [a] -> [a]
```

Exercício

Crie uma função recursiva chamada `isort` que utiliza a função `insert` para implementar o Insertion Sort:

```
isort :: Ord a => [a] -> [a]
```

Recusão Múltipla

Em alguns casos o retorno da função recursiva é a chamada dela mesma em múltiplas instâncias:

```haskell
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Exercício

Complete a função `qsort` que implementa o algoritmo Quicksort:

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort menores ++ [x] ++ qsort maiores
  where
    menores = [a | ???]
    maiores = [b | ???]
```

Recursão mútua

Um último caso interessante de recursão é quando a recursão é feita entre duas funções intercaladamente:

```
even :: Int -> Int
even 0 = True
even n = odd (n-1)

odd  :: Int -> Int
odd 0 = False
odd n = even (n-1)
```

Recursão mútua

Vamos verificar a execução:

```
> even 4
    => odd 3
    => even 2
    => odd 1
    => even 0
    True
```

Dicas para recursão

Dicas para criar uma função recursiva

Vamos considerar a função `drop` que remove os `n` primeiros elementos de uma lista:

```
> drop 3 [1..10]
[4,5,6,7,8,9,10]
```

Passo 1: defina a assinatura da função

A função `drop` recebe um `Int` e uma lista e retorna outra lista, sem restrições:

```
drop :: Int -> [a] -> [a]
```

Passo 2: enumere os casos

Para o primeiro argumento da função, podemos ter o caso trivial `0` que não faz nada e o caso genérico `n`.

O segundo argumento pode ter a lista vazia `[]` e o caso genérico `(x:xs)`. Vamos criar as combinações desses casos:

```
drop :: Int -> [a] -> [a]
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

Passo 3: defina os casos simples

Se eu não quero remover nada, retorno a própria lista, se eu quero remover algo de uma lista vazia, o retorno é vazio:

```
drop :: Int -> [a] -> [a]
drop 0 []     = []
drop 0 (x:xs) = x:xs
drop n []     = []
drop n (x:xs) =
```

Passo 4: defina os casos restantes

Como remover o primeiro elemento de `(x:xs)`? Removendo `x` e retornando apenas `xs`.

```
drop :: Int -> [a] -> [a]
drop 0 []     = []
drop 0 (x:xs) = x:xs
drop n []     = []
drop n (x:xs) = drop (n-1) xs
```

Passo 5: generalize e simplifique

O primeiro e terceiro caso são redundantes, o segundo caso não precisa de pattern matching na lista:

```
drop :: Int -> [a] -> [a]
drop _ []     = []
drop 0 xs     = xs
drop n (x:xs) = drop (n-1) xs
```