Recursão

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

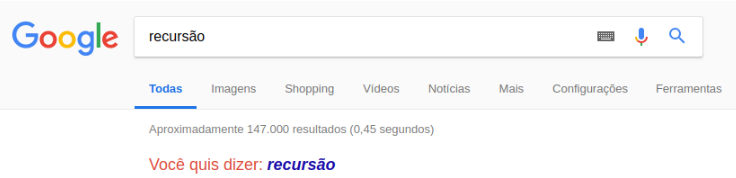
Disclaimer

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

BY SA

Recursão

Recursão


Google
recursão
Todas
Imagens
Shopping
Vídeos
Notícias
Mais
Configurações
Ferramentas
Aproximadamente 147.000 resultados (0,45 segundos)
Você quis dizer: recursão

- A recursividade permite expressar ideias declarativas.
- Composta por um ou mais casos bases (para que ela termine) e a chamada recursiva.

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

- Caso base:

$$1! = 0! = 1$$

- Para $n = 3$:

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

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

```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

UFABC

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
```

- Mesmo a pilha de expressão pode estourar!
- Recursão caudal também é útil no Haskell.

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

```haskell
f x = f x'

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

Contra-exemplos de recursão caudal:

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

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

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

- 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)
```

- A variável `r` é chamada de variável acumuladora

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
```

Pergunta

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

Pergunta

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

Resposta

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

- Podemos forçar a avaliação (forçar uma avaliação estrita) com a função `seq`

```haskell
seq :: a -> b -> b
-- Tem o seguinte comportamento
⊥ `seq` b = ⊥ -- ⊥ representa bottom
a `seq` b = b
```

- ”Magicamente”após `seq`, o seu primeiro argumento é avaliado (avaliação estrita) e o seu retorno é o segundo argumento (avaliação não estrita)

UFABC

```haskell
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
```

Dessa forma temos:

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

Tail Call Optimization - The Musical

https://www.youtube.com/watch?v=-PX0BV9hGZY

Máximo Divisor Comum

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

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

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

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

```haskell
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 😄

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

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

UFABC

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

Exemplo:

Regra	m	n	r
1	14	12	0
2	7	24	24
2	3	48	72
3	1	96	168

Exercício 1 - Multiplicação Etíope

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

Recursão em Listas

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

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

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

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

- Por que não usar Pattern Matching?

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

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

Exercício 2

UFABC

Faça a versão caudal dessa função:

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

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

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

Produtória

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

```haskell
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
```

E a função `length`?

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

E a função `length`?

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

- 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.

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 pelos casos bases:

- o inverso de uma lista vazia, é vazia

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

Invertendo uma lista

Vamos começar pelos casos bases:

- o inverso de uma lista vazia, é vazia
- o inverso de uma lista com um elemento, é ela mesma

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

Invertendo uma lista

Vamos começar pelos casos bases:

- o inverso de uma lista vazia, é vazia
- o inverso de uma lista com um elemento, é ela mesma
- o inverso de uma lista com dois elementos é...

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

Invertendo uma lista

Vamos começar pelos casos bases:

- o inverso de uma lista vazia, é vazia
- o inverso de uma lista com um elemento, é ela mesma
- o inverso de uma lista com dois elementos é...
- 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]
```

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

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

```
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)]
```

Temos como casos bases:

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

E o caso recursivo:

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

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

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

Exercício 4

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

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

Em alguns casos o retorno da função recursiva é a chamada dela mesma **múltiplas** vezes:

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

UFABC

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

```haskell
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:

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

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

Não confunda recursão múltipla com recursão mútua.

Vamos verificar a execução:

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

Dicas para recursão

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:

```haskell
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:

```haskell
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:

```haskell
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`.

```haskell
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:

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

Exemplo - Cálculo de potências

- Suponha que temos que calcular x^n para n inteiro positivo.
- Como calcular de forma recursiva?

Exemplo - Cálculo de potências

x^n é:

- 1, se $n = 0$.
- xx^{n-1}, caso contrário.

Exemplo - Cálculo de potências

1. Defina a assinatura da função
2. Enumere os casos
3. Defina os casos simples
4. Defina os casos restantes
5. Simplifique

Exemplo - Cálculo de potências

1. Defina a assinatura da função
2. Enumere os casos
3. Defina os casos simples
4. Defina os casos restantes
5. Simplifique

```haskell
pot :: (Num a, Integral b) => a -> b -> a
```

Exemplo - Cálculo de potências

1. Defina a assinatura da função ✓
2. **Enumere os casos**
3. Defina os casos simples
4. Defina os casos restantes
5. Simplifique

```haskell
pot :: (Num a, Integral b) => a -> b -> a
pot b 0 =
pot b 1 =
pot b e =
```

Exemplo - Cálculo de potências

1. Defina a assinatura da função ✓
2. Enumere os casos ✓
3. **Defina os casos simples**
4. Defina os casos restantes
5. Simplifique

```haskell
pot :: (Num a, Integral b) => a -> b -> a
pot b 0 = 1
pot b 1 = b
pot b e =
```

Exemplo - Cálculo de potências

1. Defina a assinatura da função ✓
2. Enumere os casos ✓
3. Defina os casos simples ✓
4. Defina os casos restantes
5. Simplifique

```haskell
pot :: (Num a, Integral b) => a -> b -> a
pot b 0 = 1
pot b 1 = b
pot b e = b * pot b (e - 1)
```

Exemplo - Cálculo de potências

1. Defina a assinatura da função ✓
2. Enumere os casos ✓
3. Defina os casos simples ✓
4. Defina os casos restantes ✓
5. Simplifique

```haskell
pot :: (Num a, Integral b) => a -> b -> a
pot _ 0 = 1
pot b e = b * pot b (e - 1)
```

Exemplo - Cálculo de potências

```haskell
pot :: (Num a, Integral b) => a -> b -> a
pot _ 0 = 1
pot b e = b * pot b (e - 1)
```

Pergunta

Daria para melhorar?

```haskell
pot :: (Num a, Integral b) => a -> b -> a
pot _ 0 = 1
pot b e = b * pot b (e - 1)
```

Pergunta

Daria para melhorar?

Resposta

Podemos fazer uma versão com recursão de cauda (tente fazer em casa!). Mas tem outra saída melhor ainda...

Exemplo - Cálculo de potências

E se definirmos a potência de forma diferente?

x^n é:

- se $n = 0$, então $x^n = 1$.
- se $n > 0$ e n é par, então $x^n = (x^{n/2})^2$.
- se $n > 0$ e n é ímpar, então $x^n = x(x^{(n-1)/2})^2$.

Note que aqui também definimos a solução do caso maior em termos de casos menores.

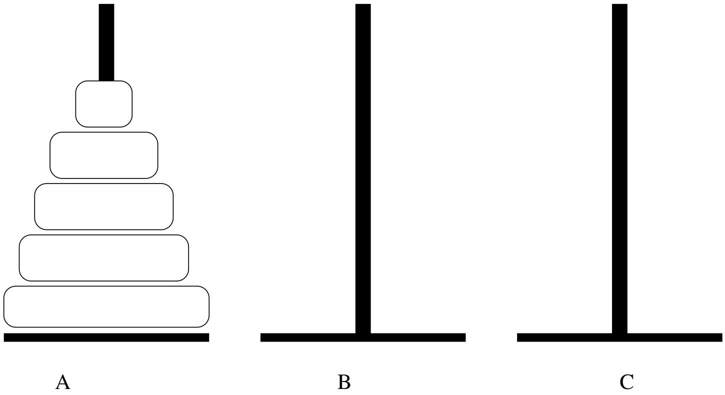
Exemplo - Cálculo de potências

```haskell
pot :: (Num a, Integral b) => a -> b -> a
pot _ 0 = 1
pot b e
  | even e      = aux * aux
  | otherwise   = b * aux * aux
  where
    aux = pot b (e `div` 2)
```

Pergunta

O algoritmo acima é mais eficiente que o anterior. Por quê?

Exemplo - Torres de Hanoi



Exemplo - Torres de Hanoi

- Inicialmente temos 5 discos de diâmetros diferentes na estaca A.
- O problema das torres de Hanoi consiste em transferir os cinco discos da estaca A para a estaca C (pode-se usar a estaca B como auxiliar).
- Porém deve-se respeitar as seguintes regras:
  - Apenas o disco do topo de uma estaca pode ser movido.
  - Nunca um disco de diâmetro maior pode ficar sobre um disco de diâmetro menor.

- Vamos considerar o problema geral onde há n discos.
- Vamos usar indução para obtermos um algoritmo para este problema.

Exemplo - Torres de Hanoi

- Base: $n = 1$. Neste caso temos apenas um disco. Basta mover este disco da estaca A para a estaca C.
- Hipótese: Sabemos como resolver o problema quando há $n - 1$ discos.
- Passo: Devemos resolver o problema para n discos.
  - Por hipótese de indução, sabemos mover os $n - 1$ primeiros discos da estaca **A** para **B** usando **C** como auxiliar.
  - Depois de movermos estes $n - 1$ discos, movemos o maior disco (que continua na estaca **A**) para a estaca **C**.
  - Novamente pela hipótese de indução, sabemos mover os $n - 1$ discos da estaca **B** para **C** usando **A** como auxiliar.
- Com isso temos uma solução para o caso onde há n discos.
- A indução nos fornece um algoritmo e ainda por cima temos uma demonstração formal de que ele funciona!

Problema: Mover n discos de **A** para **C**.

- Se $n = 1$, então mova o único disco de **A** para **C** e pare.
- Caso contrário ($n > 1$) desloque de forma recursiva os $n - 1$ primeiros discos de **A** para **B**, usando **C** como auxiliar.
- Mova o último disco de **A** para **C**.
- Mova, de forma recursiva, os $n - 1$ discos de **B** para **C**, usando **A** como auxiliar.

Exemplo - Torres de Hanoi

- Escreva uma função com a assinatura abaixo que computa a solução para o problema

```haskell
hanoi :: Int -> Char -> Char -> Char -> [(Int, Char, Char)]
```

- A função recebe um inteiro representando o número de discos, e os idenficadores das estacas (ex. `'A'`, `'B'` e `'C'`).
- A sua função deve devolver uma lista com triplas onde o primeiro elemento é o disco a ser movido, o segundo a estaca de origem e o terceiro a estaca de destino