Programação Funcional

Capítulo 6

Funções Recursivas

José Romildo Malaquias

Departamento de Computação
Universidade Federal de Ouro Preto

2012.1

1 Funções recursivas

2 Recursividade mútua

3 Recursividade de cauda

Tópicos

1 Funções recursivas

2 Recursividade mútua

3 Recursividade de cauda

Recursividade

- **Recursividade** é uma idéia inteligente que desempenha um papel central na programação funcional e na ciência da computação em geral.
- **Recursividade** é o mecanismo de programação no qual uma definição de função ou de outro objeto refere-se ao próprio objeto sendo definido.
- Assim **função recursiva** é uma função que é definida em termos de si mesma.
- Recursividade é o mecanismo básico para repetições nas linguagens funcionais.
- São sinônimos: <u>recursividade</u>, <u>recursão</u>, <u>recorrência</u>.

Estratégia recursiva

- Estratégia para a definição recursiva de uma função:
  1. dividir o problema em problemas menores do mesmo tipo
  2. resolver os problemas menores (dividindo-os em problemas ainda menores, se necessário)
  3. combinar as soluções dos problemas menores para formar a solução final
- Ao dividir o problema sucessivamente em problemas menores eventualmente os casos simples são alcançados:
  - não podem ser mais divididos
  - suas soluções são definidas explicitamente

Definição recursiva

De modo geral, uma **definição de função recursiva** é dividida em duas partes:

- Há um ou mais **casos base** que dizem o que fazer em situações simples, onde não é necessária nenhuma recursão.
  Nestes casos a resposta pode ser dada de imediato, sem chamar recursivamente a função sendo definida.
  Isso garante que a recursão eventualmente possa parar.
- Há um ou mais **casos recursivos** que são mais gerais, e definem a função em termos de uma chamada *mais simples* a si mesma.

Exemplo: fatorial

- A função que calcula o fatorial de um número natural pode ser definida recursivamente como segue:

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

- A primeira equação estabelece que o fatorial de 0 é 1. Este é o caso base.
- A segunda equação estabelece que o fatorial de um número positivo é o produto deste número e do fatorial do seu antecessor. Este é o caso recursivo.
- Observe que no caso recursivo o subproblema `fatorial (n-1)` é mais simples que o problema original `fatorial n` e está *mais próximo* do caso base `fatorial 0`.

Exemplo: fatorial (cont.)

- Aplicando a função fatorial:

```
fatorial 6
⇝ fatorial 5 * 6
⇝ (fatorial 4 * 5) * 6
⇝ ((fatorial 3 * 4) * 5) * 6
⇝ (((fatorial 2 * 3) * 4) * 5) * 6
⇝ ((((fatorial 1 * 2) * 3) * 4) * 5) * 6
⇝ (((((fatorial 0 * 1) * 2) * 3) * 4) * 5) * 6
⇝ (((((1 * 1) * 2) * 3) * 4) * 5) * 6
⇝ ((((1 * 2) * 3) * 4) * 5) * 6
⇝ (((2 * 3) * 4) * 5) * 6
⇝ ((6 * 4) * 5) * 6
⇝ (24 * 5) * 6
⇝ 120 * 6
⇝ 720
```

Exercício 1

Digite a função `fatorial` em um arquivo fonte Haskell e carregue-o no ambiente interativo de Haskell.

a) *Mostre* que `fatorial 7 = 5040`.

b) Determine o valor da expressão `fatorial 7` usando o ambiente interativo.

c) Determine o valor da expressão `fatorial 1000` usando o ambiente interativo. Se você tiver uma calculadora científica, verifique o resultado na calculadora.

d) Qual é o valor esperado para a expressão `div (fatorial 1000) (fatorial 999)`? Determine o valor desta expressão usando o ambiente interativo.

e) O que acontece ao calcular o valor da expressão `fatorial (-2)`?

Exemplo: potências de 2

- A função que calcula a potência de 2 para números naturais pode ser definida recursivamente como segue:

```
pot2 :: Integer -> Integer
pot2 n
  | n == 0 = 1
  | n > 0  = 2 * pot2 (n-1)
```

- A primeira equação estabelece que $2^0 = 1$. Este é o caso base.
- A segunda equação estabelece que $2^n = 2 \times 2^{n-1}$, sendo $n > 0$. Este é o caso recursivo.
- Observe que no caso recursivo o subproblema `pot2 (n-1)` é mais simples que o problema original `pot2 n` e está *mais próximo* do caso base `pot2 0`.

Exemplo: potências de 2 (cont.)

- Aplicando a função potência de 2:

```
pot2 4
⇝ 2 * pot2 3
⇝ 2 * (2 * pot2 2)
⇝ 2 * (2 * (2 * pot2 1))
⇝ 2 * (2 * (2 * (2 * pot2 0)))
⇝ 2 * (2 * (2 * (2 * 1)))
⇝ 2 * (2 * (2 * 2))
⇝ 2 * (2 * 4)
⇝ 2 * 8
⇝ 16
```

Exemplo: potências de 2 (cont.)

Exercício 2

Considere a seguinte definição para a função potência de 2:

```
pot2' :: Integer -> Integer
pot2' n
  | n == 0    = 1
  | otherwise = 2 * pot2' (n-1)
```

O que acontece ao calcular o valor da expressão `pot2' (-5)`?

Exemplo: multiplicação

- A multiplicação de inteiros está disponível na biblioteca como uma operação primitiva por questões de eficiência. Porém ela pode ser definida usando recursividade em um de seus argumentos:

```haskell
mul :: Int -> Int -> Int
mul m n
  | n == 0    = 0
  | n > 0     = m + mul m (n-1)
  | otherwise = - (mul m (-n))
```

- A primeira equação estabelece que quando o multiplicador é zero, o produto também é zero. Este é o caso base.
- A segunda equação estabelece que $m \times n = m + m \times (n-1)$, sendo $n > 0$. Este é um caso recursivo.
- A terceira equação estabelece que $m \times n = -(m \times (-n))$, sendo $n < 0$. Este é outro caso recursivo.

Exemplo: multiplicação (cont.)

- Aplicando a função multiplicação:

```
mul 7 (-3)
⇝ - (mul 7 3)
⇝ - (7 + mul 7 2)
⇝ - (7 + (7 + mul 7 1))
⇝ - (7 + (7 + (7 + mul 7 0)))
⇝ - (7 + (7 + (7 + 0)))
⇝ - (7 + (7 + 7))
⇝ - (7 + 14)
⇝ - 21
⇝ -21
```

- A definição recursiva da multiplicação formalisa a idéia de que a multiplicação pode ser reduzida a adições repetidas.

Exemplo: multiplicação (cont.)

Exercício 3

Mostre que `mul 5 6 = 30`.

Exemplo: sequência de Fibonacci

- Na seqüência de Fibonacci

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

  os dois primeiros elementos são 0 e 1, e cada elemento subseqüente é dado pela soma dos dois elementos que o precedem na seqüência.

- A função a seguir calcula o n-ésimo número de Fibonnaci, para $n \geq 0$:

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

- A primeira e segunda equações são os casos base.
- A terceira equação é o caso recursivo.

Exemplo: sequência de Fibonacci (cont.)

- Neste caso temos **recursão múltipla**, pois a função sendo definida é usada mais de uma vez em sua própria definição.
- Aplicando a função de fibonacci:

```
fib 5
⇝ fib 3 + fib 4
⇝ (fib 1 + fib 2) + (fib 2 + fib 3)
⇝ (1 + (fib 0 + fib 1)) + ((fib 0 + fib 1) + (fib 1 + fib 2))
⇝ (1 + (0 + 1)) + ((0 + 1) + (1 + (fib 0 + fib 1)))
⇝ (1 + 1) + (1 + (1 + (0 + 1)))
⇝ 2 + (1 + (1 + 1))
⇝ 2 + (1 + 2)
⇝ 2 + 3
⇝ 5
```

Exemplo: sequência de Fibonacci (cont.)

Exercício 4

Mostre que `fib 6 = 8`.

Tópicos

1 Funções recursivas

2 Recursividade mútua

3 Recursividade de cauda

Recursividade mútua

- **Recursividade mútua** ocorre quando duas ou mais funções são definidas em termos uma da outra.

Exemplo: par e ímpar

- As funções da biblioteca `even` e `odd`, que determinam se um número é par ou ímpar, respectivamente, geralmente são definidas usando o resto da divisão por 2.

Exemplo: par e ímpar (cont.)

- No entanto elas também podem ser definidas usando recursividade mútua:

```
par :: Int -> Bool
par n | n == 0    = True
      | n > 0     = impar (n-1)
      | otherwise = par (-n)

impar :: Int -> Bool
impar n | n == 0    = False
        | n > 0     = par (n-1)
        | otherwise = impar (-n)
```

- Zero é par, mas não é ímpar.
- Um número positivo é par se seu antecessor é ímpar.
- Um número positivo é ímpar se seu antecessor é par.
- Um número negativo é par (ou ímpar) se o seu oposto for par (ou ímpar).

Exemplo: par e ímpar (cont.)

- Aplicando as função par e ímpar:

```
par (-5)
⇝ par 5
⇝ impar 4
⇝ par 3
⇝ impar 2
⇝ par 1
⇝ impar 0
⇝ False
```

Tópicos

1 Funções recursivas

2 Recursividade mútua

3 Recursividade de cauda

Recursividade de cauda

- Uma função recursiva apresenta **recursividade de cauda** se o resultado final da chamada recursiva é o resultado final da própria função.
- Se o resultado da chamada recursiva deve ser processado de alguma maneira para produzir o resultado final, então a função não apresenta recursividade de cauda.

- **Exemplo:**
  A função recursiva a seguir não apresenta recursividade de cauda:

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

  No caso recursivo, o resultado da chamada recursiva `fatorial (n-1)` é multiplicado por `n` para produzir o resultado final.

- **Exemplo:**
  A função recursiva a seguir não apresenta recursividade de cauda:

```
par :: Integer -> Bool
par n
  | n == 0 = True
  | n > 0  = not (par (n-1))
```

  No caso recursivo, a função `not` é aplicada ao resultado da chamada recursiva `par (n-1)` para produzir o resultado final.

Recursividade de cauda (cont.)

- **Exemplo:**
  A função recursiva `potencia2'` a seguir apresenta recursividade de cauda:

```
potencia2 :: Integer -> Integer
potencia2 n = potencia2' n 1

potencia2' :: Integer -> Integer -> Integer
potencia2' n y
  | n == 0 = y
  | n > 0  = potencia2' (n-1) (2*y)
```

  No caso recursivo, o resultado da chamada recursiva `potencia2' (n-1) (2*y)` é o resultado final.

Recursividade de cauda (cont.)

Exercício 5

Mostre que `potencia2 5 = 32`.

Exercício 6

Faça uma definição recursiva da função par usando recursividade de cauda.

- Em muitas implementações de linguagens de programação uma chamada de função usa um espaço de memória (**quadro**, *frame* ou registro de ativação) em uma área da memória (**pilha** ou *stack*) onde são armazenadas informações importantes, como:
  - argumentos da função
  - variáveis locais
  - variáveis temporárias
  - endereço de retorno da função

Otimização de chamada de cauda (cont.)

- Uma **chamada de cauda** acontece quando uma função chama outra função como sua última ação, não tendo mais nada a fazer. O resultado final da função é dado pelo resultado da chamada de cauda.
- Em tais situações o programa não precisa voltar para a função que chama quando a função chamada termina.
- Portanto, após a chamada de cauda, o programa não precisa manter qualquer informação sobre a função chamadora na pilha.
- Algumas implementações de linguagem tiram proveito desse fato e na verdade não utilizam qualquer espaço extra de pilha quando fazem uma chamada de cauda.
- Esta técnica é chamada de **eliminação da cauda**, **otimização de chamada de cauda** ou ainda **otimização de chamada recursiva**.
- A **otimização de chamada de cauda** permite que funções com recursividade de cauda recorram indefinidamente sem estourar a pilha.
- Muitas linguagens funcionais não possuem estruturas de repetição e usam funções recursivas para fazer repetições.
- Nestes casos a otimização de chamada de cauda é fundamental para uma boa eficiência dos programas.

Vantagens de usar recursividade

- Muitas funções podem ser naturalmente definidas em termos de si mesmas.
- Propriedades de funções definidas usando recursão podem ser provadas usando **indução**, uma técnica matemática simples, mas poderosa.

Exercícios

Exercício 7

O fatorial duplo de um número natural n é o produto de todos os números de 1 (ou 2) até n, contados de 2 em 2. Por exemplo, o fatorial duplo de 8 é $8 \times 6 \times 4 \times 2 = 384$, e o fatorial duplo de 7 é $7 \times 5 \times 3 \times 1 = 105$.
Defina uma função para calcular o fatorial duplo usando recursividade.

Exercício 8

Defina uma função recursiva que recebe dois números naturais m e n e retorna o produto de todos os números no intervalo $[m, n]$:

$$m \times (m+1) \times \cdots \times (n-1) \times n$$

Exercícios (cont.)

Exercício 9

Usando a função definida no exercício 8, escreva uma definição não recursiva para calcular o fatorial de um número natural.

Exercício 10

Defina uma função recursiva para calcular a soma de dois números inteiros, sem usar os operadores `+` e `-`. Utilize as funções `succ` e `pred` da biblioteca, que calculam respectivamente o sucessor e o antecessor de um valor.

Exercício 11

Defina uma função recursiva para calcular a potência de um número, considerando que o expoente é um número natural. Utilize o método das multiplicações sucessivas.

Exercícios (cont.)

Exercício 12

A raiz quadrada inteira de um número inteiro positivo *n* é o maior número inteiro cujo quadrado é menor ou igal a *n*. Por exemplo, a raiz quadrada inteira de 15 é 3, e a raiz quadrada inteira de 16 é 4.
Defina uma função recursiva para calcular a raiz quadrada inteira.

Exercício 13

Defina duas funções recursivas que calculam o quociente e o resto da divisão inteira de dois números inteiros usando subtrações sucessivas.

Exercícios (cont.)

Exercício 14

Defina uma função recursiva para calcular o máximo divisor comum de dois números inteiros não negativos a e b, usando o algoritmo de Euclides:

$$\text{mdc}(a,b) = \begin{cases} a & \text{se } b = 0, \\ \text{mdc}(b, a \bmod b) & \text{se } b > 0, \\ \text{mdc}(a, -b) & \text{se } b < 0 \end{cases}$$

Nota: o prelúdio já tem a função `gcd :: Integral a => a -> a -> a` que calcula o máximo divisor comum de dois números integrais.

Exercício 15

Faça uma definição recursiva para uma função

```
maior :: (Integer -> Integer) -> Integer -> Integer
```

que recebe uma função *f* e um número inteiro não negativo *n*, e retorna o maior dos valores

`f 0`, `f 1`, `f 2`, `...`, `f (`n`-1)`, `f` n

Por exemplo, considerando a função

```
g :: Integer -> Integer
g x | even x    = 2*x^2 - 3*x + 1
    | otherwise = div (x^3) 2
```

temos

```
maior g 10 ⇝ 364
```

Exercício 16

Considere a seguinte função para calcular o fatorial de um número:

```
fat n = fat' n 1
  where
    fat' n x
      | n == 0 = x
      | n > 0  = fat' (n-1) (n*x)
```

a) Mostre que `fat 6 = 720`.

b) Compare o cálculo de `fat 6` com o cálculo de `fatorial 6` apresentado anteriormente. Qual versão da função fatorial é mais eficiente: `fatorial` ou `fat`? Explique.

Exercício 17

Defina uma função com recursividade de cauda para calcular o n-ésimo ($n \geq 0$) número de Fibonacci.

Fim