DCC909 – Programação Funcional

AULA 05

Carlos Bruno Oliveira Lopes

Engenheiro de Computação

Mestre em Ciência da Computação

HASKELL

Recursividade

– é o mecanismo de programação no qual uma definição de função ou de outro objeto refere-se ao próprio objeto sendo definido.

Função recursiva

– é uma função que é definida em termos de si mesma.
– Recursividade é o mecanismo básico para *repetições* nas linguagens funcionais.

HASKELL

Função 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

HASKELL

Função 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 pode 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.

HASKELL

Função recursiva

Ex.:

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

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

Nesta definição:

- A primeira guarda estabelece que o fatorial de 0 é 1. Este é o caso base.
- A segunda guarda estabelece que o fatorial de um número positivo é o produto deste número e do fatorial do seu antecessor. Este é o caso recursivo.

Obs.: 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.

HASKELL – exercício

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

a) Mostre que fatorial 7 = 5040 usando uma sequência de passos de simplificação.
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 se calcular o valor da expressão fatorial (-2)?

HASKELL

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

Função recursiva

Ex.:

– Função que calcula a potência de dois (base dois) para números naturais.

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

Nesta definição:

- A primeira cláusula estabelece que $2^0 = 1$. Este é o caso base.
- A segunda cláusula estabelece que $2^n = 2 \times 2^{n-1}$, sendo $n > 0$. Este é o caso recursivo.

Obs.: no caso recursivo o subproblema potDois (n-1) é mais simples que o problema original potDois n e está mais próximo do caso base potDois 0.

HASKELL – exercício

Considere a seguinte definição para a função potência de dois:

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

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

HASKELL

- Função recursiva

  Ex.:

  – A multiplicação de inteiros.

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

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

Nesta definição:

– A primeira cláusula estabelece que quando o multiplicador é zero, o produto também é zero. Este é o caso base.

– A segunda cláusula estabelece que $m \times n = m + m \times (n - 1)$, sendo $n > 0$. Este é o caso recursivo.

– A terceira cláusula estabelece que $m \times n = -(m \times (n - 1))$, sendo $n < 0$. Este é outro caso recursivo.

HASKELL – exercício

Mostre que **mul 5 6 = 30**.

HASKELL

```
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
```

Função recursiva

Ex.:

– A sequência de Fibonacci. Na sequência de Fibonacci $0, 1, 1, 2, 3, 5, 8, 13 \ldots$ os dois primeiros elementos são 0 e 1, e cada elemento subsequente é dado pela soma dos dois elementos que o

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

Nesta definição:

- A primeira e a segunda cláusula estabelece são os caso base.
- A terceira cláusula é o caso recursivo.

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

HASKELL – exercício

Mostre que fib 6 = 8.

HASKELL

Recursividade Mútua

– Ocorre quando duas ou mais funções são definidas em termos uma da outra.

Ex.:

Definida pelo resto

```haskell
par, impar :: Int -> Bool

par n = mod n 2 == 0

impar n = not (par n)
```

Definida pela recursividade

```haskell
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)
```

HASKELL

Recursividade Mútua

Ex.:

```haskell
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)
```

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

- Nestas definições observamos que:
  - 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).

HASKELL

Recursividade de cauda

– Uma função recursiva apresenta recursividade de cauda se o resultado final da chamada recursiva é o resultado final da própria função.

Ex.:

```haskell
pdois :: Integer -> Integer
pdois n = pdois' n 1

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

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

HASKELL – exercício

1. Mostre que **pdois 5 = 32**.

HASKELL

Estruturas de repetição

- Muitas linguagens funcionais não possuem estruturas de repetição;
- Elas usam funções recursivas para fazer repetições

HASKELL – Exercícios

1. (**Fatorial duplo**) 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.
2. (**Multiplicação em um intervalo**) Defina uma função recursiva que recebe dois números naturais m e n, e retorna o produto de todos os números no intervalo [m; n]:
$$m \times (m + 1) \times \cdots \times (n - 1) \times n$$
3. (**Fatorial**) Usando a função definida no exercício 2, escreva uma definição não recursiva para calcular o fatorial de um número natural.
4. (**Adição**) Defina uma função recursiva para calcular a soma de dois números naturais, sem usar os operadores + e -. Utilize as funções **succ** e **pred** da biblioteca, que calculam respectivamente o sucessor e o antecessor de um valor.