Linguagens de Programação

Funções Recursivas

Samuel da Silva Feitosa

Aula 10

Funções recursivas

Funções Recursivas

- **Recursividade** é uma ideia que desempenha um papel central na programação funcional.
  - É um mecanismo de programação no qual uma definição de função refere-se a ela mesma no código que a define.
  - É uma função que é definida em termos de si mesma.
- Recursividade é o mecanismo básico para criar **repetições** em linguagens funcionais.

Estratégia

- Definição recursiva de uma função:
    - **Dividir** o problema em problemas menores do mesmo tipo.
    - **Resolver** os problemas menores (dividindo-os em problemas ainda menores, se necessário).
    - **Combinar** as soluções dos problemas menores para formar a solução final.
- Ao dividir o problema, casos mais simples são alcançados.
    - Problema não pode mais ser dividido.
    - Suas soluções são divididas explicitamente.

Funções Recursivas

- De modo geral, uma definição de função recursiva é dividida em duas partes.

1. Há um ou mais **casos base** que dizem o que fazer em situações simples.
   - Nestes casos, a resposta pode ser dada de imediato, sem chamar recursivamente a função.
   - Isso garante que a recursão eventualmente pode parar.
2. 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

- Considere o cálculo do fatorial de um número.

```
fatorial :: Integer -> Integer
fatorial n = if n == 0
                 then 1
                 else fatorial (n - 1) * n
```

- Nesta definição:
  - A condição verdadeira (**then**) estabelece que o fatorial de 0 é 1. Este é o caso base.
  - A condição falsa (**else**) estabelece que o fatorial de um número é o produto desse número com o fatorial do seu antecessor. Este é o caso recursivo.
- Observe que no caso recursivo o subproblema fatorial (n - 1) é mais simples que o original.

Aplicando a função *fatorial*

- Vejamos os passos que o computador está executando para o fatorial de 4.

```
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
```

Tarefas sobre fatorial

- Mostre que **fatorial 7 = 5040** usando a sequência de passos de simplificação.
- Determine o valor da expressão **fatorial 7** usando o ambiente interativo.
- Determine o valor da expressão **fatorial 1000** usando o ambiente interativo. Confira se o cálculo está correto usando uma calculadora.
- Qual é o valor esperado para a expressão **div (fatorial 1000) (fatorial 999)**? Determine o valor desta expressão com ambiente interativo.

Recursão - Outro exemplo

- A função que calcula a potência de dois (isto é, a base é dois) para números naturais pode ser definida recursivamente como:

```
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 estabelece que 2^n = 2 x 2^(n-1), sendo n>0. Este é o caso recursivo.

Recursão - Outro exemplo

- Observe que no caso recursivo o subproblema potDois (n-1) é mais simples que o original.
  - Ou seja, está mais próximo do caso base.
- Aplicando a função potência de dois:

```
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
```

Recursão - Multiplicação

- A multiplicação pode ser definida usando adição e recursividade em um de seus argumentos.

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

- 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 diz que m x n = m + m x (n-1), sendo n>0. Este é o caso recursivo.

Recursão - Multiplicação

- A definição recursiva da multiplicação formaliza a ideia de que a multiplicação pode ser reduzida a adições repetidas.
- **Mini exercício:**
  - Mostre que mul 5 6 = 30.

Recursão com Listas

- Vamos definir nossas próprias funções que manipulam listas.
- Precisamos também considerar os dois casos gerais:
    - O que fazer quando a lista está vazia.
    - O que fazer quando a lista tem algum elemento inicial e uma cauda.
- A funções terão o seguinte esqueleto básico:

```
if null list
    then <caso para lista vazia>
    else <faz algo com (head list) e (tail list)>
```

Recursão com Listas

- Verificando o comprimento de uma lista.

```
comp :: [Int] -> Int
comp list = if null list
                then 0
                else 1 + comp (tail list)
```

- Podemos escrever a mesma função utilizando pattern matching.

```
comp' :: [Int] -> Int
comp' [] = 0
comp' (x:xs) = 1 + comp' xs
```

Concatenação de Listas

- Neste exemplo, vamos utilizar um nome de função que utiliza símbolos **(+++).**
  - Na definição usaremos: **lst1 +++ lst2**
- Vamos usar os dois casos gerais:
  - Quando concatenando uma lista vazia com qualquer outra lista, retornaremos apenas a segunda lista, pois a primeira não adiciona nenhum elemento.
  - Quando a primeira lista tiver elementos, vamos adicionar o primeiro (**head**) na resposta e chamar recursivamente a função **+++** com a cauda (**tail**) e a segunda lista.

Concatenação de Listas

- Traduzindo esta ideia em código:

```haskell
(+++) :: [Int] -> [Int] -> [Int]
lst1 +++ lst2 = if null lst1
                   then lst2
                   else (head lst1) : (tail lst1 +++ lst2)
```

- Fazendo o mesmo usando pattern matching.

```haskell
(++++) :: [Int] -> [Int] -> [Int]
[] ++++ lst2 = lst2
(h:t) ++++ lst2 = h : (t ++++ lst2)
```

Invertendo uma Lista

- Vamos seguir novamente a metodologia de separar o trabalho da função.
- Vamos usar os dois casos gerais:
    - Caso base: se a lista estiver vazia, retorna uma lista vazia.
    - Caso recursivo: inverte a cauda (**tail**) da lista e adiciona a cabeça (**head**) da lista ao final da lista invertida.

Invertendo uma Lista

- Traduzindo esta ideia em código:

```
reverse' :: [Int] -> [Int]
reverse' lst = if null lst
                  then []
                  else reverse' (tail lst) ++ [head lst]
```

- A mesma ideia usando pattern matching.

```
reverse'' :: [Int] -> [Int]
reverse'' [] = []
reverse'' (h:t) = reverse'' t ++ [h]
```

Considerações Finais

- Estudamos outro conceito muito importante na programação funcional, o uso da **recursão** na definição de funções.
- Também vimos o uso de **guardas** condicionais.
- Verificamos o formato geral de uma definição recursiva:
  - Tratar o **caso base**.
  - Chamar a função **recursivamente** com um **problema menor** do que o inicial.
- Utilizamos recursão tanto para números quando para listas.

Exercícios

- Defina uma função recursiva que recebe dois números naturais m e n, e retorna o produto de todos os números no intervalo $[m, n]$:
    - $m \times (m + 1) \times \ldots \times (n - 1) \times n$
- Defina uma função recursiva para calcular o n-ésimo ($n >= 0$) número da sequência de Fibonacci.