Variantes da Recursão Primitiva e Recursão Geral

Programação Funcional
DCOMP-UFS

Fugindo um pouco do esquema

As vezes o caso base não é para a lista vazia.
Por exemplo

```
maior :: [Int] -> Int
maior [] = ?
maior (b:bs) = …
```

```
maior :: [Int] -> Int
maior [b] = b
maior (b:bs) = max b (maior bs)
```

Exercícios

Definir propriedades da função `maior` e testar com `quickCheck`

Definir uma função `menor` para calcular o menor de uma lista de inteiros

Definir propriedades da função `menor` e testar com `quickCheck`

Fugindo do esquema

Pode ser necessário ter mais de um caso base

```
sorted :: [Int] -> Bool
sorted [] = True
sorted [x] = True
sorted (x1:x2:xs) = x1 <= x2 && sorted (x2:xs)
```

Testando iSort

A lista produzida por `iSort` efetivamente está ordenada

```
prop_Sorted_iSort :: [Int] -> Bool
prop_Sorted_iSort xs = sorted (iSort xs)
```

```
Main> quickCheck prop_Sorted_iSort
...
```

Exercício

- Definir a função
  ```
  filtraPosicoesPares :: [Int] -> [Int]
  ```
  que retorna todos os elementos da lista de entrada que estão em posições impares

- Definir a função
  ```
  filtraPosicoesImpares :: [Int] -> [Int]
  ```
  que retorna todos os elementos da lista de entrada que estão em posições pares

Recursão Primitiva numa combinação de argumentos

Para `m`, `n` inteiros tais que `m<=n`, queremos calcular a soma de `m` até `n`

```
somaDeAte m n = m + (m+1) + (m+2) + … + n
```

Recursão em que variável?
Qual é o caso base?

Observe:

Se `m == n`, `somaDeAte m n = m`

Se `m < n`, `somaDeAte m n = somaDeAte m (n-1) + n`

Recursão primitiva em `n-m`

Uma forma de raciocinar sobre a “parada” da indução é pensar num “tamanho do problema”

```
somaDeAte m n
    | m == n     = m
    | otherwise = somaDeAte m (n-1) + n
```

O tamanho do problema é `n-m`

- O tamanho é decrescido em 1 a cada chamada recursiva
- Temos as garantia que o caso base vai ser alcançado (em algum momento teremos que `m==n`)

Outra versão de `somaDeAte`
Tamanho do problema também é `n-m`
Recursão primitiva em `n-m`

```
somaDeAte m n
    | m == n     = m
    | otherwise = m + somaDeAte (m+1) n
```

Recursão Geral

Definição recursiva em termos de valores menores (não necessariamente `n-1`)

```
par :: Int -> Bool
par 0 = True
par n = par (n-2)
```

Algum problema?

```
par :: Int -> Bool
par 0 = True
par 1 = False
par n = par (n-2)
```

O tamanho do problema é `n`

- O tamanho é decrescido em 2 a cada chamada recursiva
- Temos as garantia que um dos casos bases será alcançado (em algum momento teremos que `n==1` ou `n==0`)

Recursão geral: a chamada rercursiva é feita com um tamanho do problema menor (não necessariamente `n-1`)

A definição dos casos base deve ser cuidadosa para evitar loops (computos infinitos)

`div` e `mod` por subtrações sucessivas

```
meuDiv m n
     | m < n       = 0
     | otherwise = 1 + meuDiv (m-n) n
```

```
meuMod m n
     | m < n       = m
     | otherwise = meuMod (m-n) n
```

O tamanho do problema é `m`

- O tamanho é decrescido em `n` a cada chamada recursiva
- O caso base ocorre quando não podemos decrescer o tamanho do problema em `n`
- Temos a garantia que o caso base será alcançado?

O que acontece com `meuDiv` e `meuMod` quando `n = 0`?

Exercícios

Sem usar nem `div` nem `mod`, defina uma função chamada `divMod` que calcula ao mesmo tempo a divisão inteira e o resto da divisão inteira

Testar usando `quickCheck`

O algoritmo de Euclides para o `mdc`

Uma propriedade do `mdc` (máximo divisor comum) é que se `a > b`, então

```
mdc a b = mdc (a-b) b
```

Podemos usar esta propriedade para darmos uma definição recursiva para a função `mdc`

```
mdc :: Int -> Int -> Int
mdc a b
  | a == b     = a
  | a < b      = mdc (a, b-a)
  | otherwise = mdc (a-b, b)
```

Exercício: Euclides estendido

Outra propriedade do `mdc` (máximo divisor comum) é que se `a > b`, então

```
mdc a b = mdc (a `mod` b) b
```

Usar esta propriedade para dar uma definição recursiva para a função `mdc`.
Considere que o caso base acontece quando um número divide o outro.

Exercício

```
pot m n
    | n == 0     = 1
    | otherwise = m * pot m (n-1)
```

Quantas chamadas recursivas são feitas para calcular `pot m n`?

Podemos melhorar muito a eficiência de `pot` usando as seguintes propriedades

$$m^n = (m^k)^2, \text{ se } n \text{ é par com } n = 2k$$

$$m^n = m(m^k)^2, \text{ se } n \text{ é impar com } n = 2k+1$$

Dê uma definição para `pot` que explore as propriedades acima. Nesta versão, quantas chamadas recursivas são feitas para calcular `pot m n`?

Fibonacci

A sequência de Fibonacci começa com 0 e 1 e depois cada subsequente número é a soma dos dois anteriores.

0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Uma função que calcula o n-ésimo número de fibonacci é

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

Recursão com `n-1` e `n-2`, simultaneamente
Termina sempre?
Eficiência? … Exemplo de cálculo

Fibonacci com recursão primitiva

```
-- fib_ n retorna um par:
--  o primeiro é o n_ésimo fibonacci
--  o segundo é o próximo fibonacci

fib_ :: Int -> (Int, Int)
fib_ 0 = (0,1)
fib_ n = (b, a+b)
  where (a, b) = fib_ (n-1)

fib :: Int -> Int
fib n = first (fib_ n)
```

Recursão geral em listas

```
last :: [Int] -> Int
last [a] = a
last (x:xs) = last xs
```

```
last :: [Int] -> Int
last [] = error "last is undefined for []"
last [a] = a
last (x:xs) = last xs
```

Recursão geral em listas

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

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

Exercício

- O que acontece com `take` e `drop` quando o primeiro argumento é negativo?

- Redefina `take` de tal forma que `take n xs` retorne `[]` quando `n < 0`.

- Redefina `drop` de tal forma que `drop n xs` retorne `xs` quando `n < 0`.

```
palindroma :: [Char] -> Bool
palindroma [] = True
palindroma [x] = True
palindroma xs = head xs == last xs &&
                palindroma (init (tail xs))
```

```
palindroma :: [Char] -> Bool
palindroma xs = length xs <= 1 ||
                ( head xs == last xs &&
                  palindroma (init (tail xs))
```

Quick sort

```
qSort :: [Int] -> [Int]
qSort [] = []
qSort (x:xs) = qSort menores ++ [x] ++ qsort maiores
   where menores = [ y | y <- xs, y <= x ]
         maiores = [ y | y <- xs, y > x ]
```

```
qSort :: [Int] -> [Int]
qSort [] = []
qSort (x:xs) = qSort menores ++ (x : qsort maiores)
   where menores = [ y | y <- xs, y <= x ]
         maiores = [ y | y <- xs, y > x ]
```

Exercício

- Defina a função
  `merge :: [Int] -> [Int] -> [Int]`
  de tal maneira que ao receber duas listas ordenadas `xs` e `ys`, `merge xs ys` retorna todos os elementos de `xs` e `ys` numa única lista também ordenada.
- Usando `merge`, defina a função
  `mergeSort :: [Int] -> [Int]`
  que ordena a lista de entrada. A ideia é dividir a lista de entrada em duas partes de tamanho igual ou no máximo variando em um, recursivamente ordenar cada parte e então aplicar `merge`.

Recursão mútua

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

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

Exercício

- Definir as funções

```
filtraPosicoesPares :: [Int] -> [Int]
filtraPosicoesImpares :: [Int] -> [Int]
```

usando recursão mútua

Exercício

Usando `quickCheck`, podemos testar as seguintes propriedades da função `maior`

- `maior xs` é maior ou igual que todos os elementos de `xs`
- `xs` contém um elemento que é igual a `maior xs`

```
prop_Maior_maior :: [Int] -> Bool
prop_Maior_maior ns =
     ns == [] || and [ m <= n | n <- ns]
  where m = menor ns
```

Exercício

Defina uma função para a propriedade:

- `xs` contém um elemento que é igual a `maior xs`

Teste usado `quickCheck`