Recursão Primitiva

Programação Funcional
DCOMP-UFS

Função fatorial

1. `fatorial(0) = 1`
2. **Se** `n>0`

```
fatorial(n) = n*(n-1)*...2*1
            = n*fatorial(n-1)
```

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ \text{n}*(n\text{-}1)! & \text{se } n > 0 \end{cases}$$

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

Recursão através de uma tabela

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

n	**fatorial**
0	1
1	1 * fatorial 0 = 1 * 1 = 1
2	2 * fatorial 1 = 2 * 1 = 2
3	3 * fatorial 2 = 3 * 2 = 6
...	...

Cálculo de funções com recursão

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

```
fatorial 4 ↝
   4 * fatorial (4-1) ↝
   4 * fatorial 3 ↝
   4 * 3 * fatorial (3-1) ↝
   4 * 3 * fatorial 2 ↝
   4 * 3 * 2 * fatorial (2-1) ↝
   4 * 3 * 2 * fatorial 1 ↝
   4 * 3 * 2 * 1* fatorial (1-1) ↝
   4 * 3 * 2 * 1 * fatorial 0 ↝
   4 * 3 * 2 * 1 * 1 ↝
   … ↝
   24
```

Recursão Primitiva

Esquema de definição **sobre os naturais**

Como outro exemplo, queremos calcular a soma
  0+1+2+...+n

Podemos definir a função `somaAte` explorando os seguintes fatos

1. `somaAte(0) = 0`
2. Se `n>0`
   0+1+2+...+n = (0+1+2+...+(n-1)) + n

   Ou seja:
   `somaAte(n) = somaAte(n-1) + n`

```
somaAte :: Int -> Int
somaAte 0 = 0
somaAte n = somaAte(n-1) + n
```

Exemplo de cálculo ...

Esquema geral da recursão primitiva

Para definir uma função f sobre os naturais podemos usar o seguinte padrão

```
f :: A -> … -> Int … -> X
f … 0 … = exp_base
f … n … = exp_indutiva[f(..., n-1, ... ), n, … ]
```

Para a parte indutiva pensamos assim:

Se já tenho a solução para n-1, como construo a solução para n?

Exemplo: soma usando sucessor

Preciso escolher em que argumento fazer a indução

```
soma:: Int -> Int -> Int
soma 0 m = m
soma n m = ...
```

Se tenho `soma (n-1) m`, como calculo `soma n m`?

```
soma:: Int -> Int -> Int
soma 0 m = m
soma n m = suc (soma(n-1) m)
```

Exemplo: soma usando sucessor

Neste exemplo poderia ter escolhido fazer a indução no segundo argumento

```
soma:: Int -> Int -> Int
soma n 0 = n
soma n m = suc (soma n (m-1))
```

Exemplos de cálculo com soma

```
soma:: Int -> Int -> Int
soma 0 m = m
soma n m = suc (soma(n-1) m)
```

Exercícios

Usando recursão primitiva:

1. Defina a função multiplica que multiplica dois números. Não pode usar o operador (*), mas pode usar (+).

2. Escreva uma função que receba n e devolva 2^n

3. Escreva uma função `pot` que receba um inteiro m e um natural n e calcule m^n

4. Escreva uma função que dado n, calcule:
   $0! + 1! + 2! + \ldots + n!$

5. Escreva uma função que calcule
   $2^0 + 2^1 + 2^2 + \ldots + 2^n$

5. Dada uma função `f` de `Int` em `Int`, defina por recursão primitiva uma função `maiorF` que aceite um natural `n` e devolva o maior valor dentre `f 0`, `f 1` , ... e `f n`.
   Para testar ou validar, use a seguinte definição para `f`

```
f :: Int -> Int
f m
  | m == 0 = 8
  | m == 1 = 44
  | m == 2 = 17
  | otherwise = 0
```

7. Dada um função `f` de `Int` em `Int`, defina por recursão primitiva uma função `algumF0` que aceite um natural `n` e devolva o booleano `True` se e somente se um ou mais valores de `f 0`, `f 1`, ..., `f n` é zero. Teste com diferentes definições de `f`.

8. Dada um função `f` de `Int` em `Bool`, defina por recursão primitiva uma função `algumFentre` que aceite um natural `n` e devolva o booleano `True` se e somente se `f i` é `True` para algum `i` entre `0` e `n`. Teste com diferentes definições de `f`.

9. Defina por recursão primitiva uma função que calcule a raiz quadrada inteira de n (o maior natural cujo quadrado é menor ou igual a n)

Recursão Primitiva em Listas (e Strings)

Programação Funcional
DCOMP-UFS

Listas são construções indutivas

- Uma lista é
  - Vazia, ou
  - Construída com um elemento *<u>cabeça</u>* e uma <u>lista</u> *<u>resto</u>*
- Em Haskell
  - `[]`
  - `b : bs` uma lista cuja cabeça é `b` e cujo resto é a lista `bs`
- Exemplos

```
[]
1 : [] = [1]
1 : (4 : (0 : []))) = [1, 4, 0]
```

- (:) é associativo à direita, assim, a última expressão acima pode ser escrita sem parênteses

```
1 : 4 : 0 : [ ] = [1, 4, 0]
```

Recursão primitiva sobre listas

Para definir uma função f sobre listas podemos usar o seguinte padrão

```
f :: A -> … -> [B] … -> X
f … [] … = exp_base
f … (a:as) … = exp_indutiva[… f as… , n, … ]
```

Para a parte indutiva pensamos assim:

Se já tenho a solução para a lista `as`, como construo a solução para `(a : as)`?

Exemplos

```
longitude :: [Int] -> Int
longitude [] = 0
longitude (b:bs) = 1 + longitude bs
```

```
soma :: [Int] -> Int
soma [] = 0
soma (b:bs) = b + soma bs
```

```
pares :: [Int] -> [Int]
pares [] = []
pares (b:bs)
   | b `mod`2 == 0 = b : pares bs
   | otherwise     = pares bs
```

- Tabela para longitude

- Cálculo com funções recursivas
  - longitude
  - soma
  - pares

Exercícios

- Usando recursão primitiva sobre listas (não pode usar compreensões), defina funções para
  - O produto dos elementos de uma lista de inteiros
  - Filtrar (eliminar) os números pares, ou seja, ficar somente com os ímpares
  - Verificar se um string é formado somente por caracteres alfanuméricos (letras e numerais). Use a função

    ```
    isAlphaNum :: Char -> Bool
    ```

    da biblioteca `Data.Char`
  - Eliminar a primeira ocorrência de um dado elemento, se ele ocorrer, senão retornar a lista original
  - Eliminar todas as ocorrências de um dado elemento
  - Inverter um string

Recursão primitiva em listas - mais exemplos

```
isAlphaNumStr :: [Char] -> Bool
isAlphaNumStr [] =  True
isAlphaNumStr (c:cs)
    | isAlphaNum c = isAlphaNumStr cs
    | otherwise    = False
```

Outra forma de escrever:

```
isAlphaNumStr :: [Char] -> Bool
isAlphaNumStr [] =  True
isAlphaNumStr (c:cs)= isAlphaNum c &&
                      isAlphaNumStr cs
```

```
elem :: Int -> [Int] -> Bool
elem x []      = False
elem x (y:ys) = x == y || elem x ys
```

```
(++) :: [Int] -> [Int] -> [Int]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Observe que não dá certo se tentamos fazer a recursão primitiva no segundo argumento.
Aprender a escolher qual é argumento para fazer a recursão é crítico

```
dobrarTodos :: [Int] -> [Int]
dobrarTodos [] = []
dobrarTodos (x:xs) = 2*x : dobrarTodos xs
```

Uma versão sem recursão, usando compreensões

```
dobrarTodos :: [Int] -> [Int]
dobrarTodos xs = [2*x | x <- xs]
```

Compreensões são bastante expressivas, mas, internamente, Haskell trabalha com funções recursivas.

Dada uma lista `xss` cujos elementos são listas, a função `concat` concatena todas as listas em `xss` ilegíveis.

```
concat :: [[Int]] -> [Int]
concat []       = []
concat (xs:xss) = xs ++ concat xss
```

Consegue definir a função `concat` usando somente compreensões?

A função `and` aplica o operador e lógico `&&` a todos os elementos de uma lista. Por exemplo:

```
and [False, True, False]
   ⇝ False && True && False ⇝ … ⇝ False
```

```
and :: [Bool] -> Bool
and []     = True
and (b:bs) = b && and bs
```

Exercício

A função `or` aplica o operador ou lógico `||` a todos os elementos de uma lista. Por exemplo:

```
or [False, True, False]
   ↝ False || True || False ↝ … ↝ True
```

Dê uma definição recursiva para a função `or`.

Ordenação por inserção

```
iSort :: [Int] -> [Int]
iSort []       = []
iSort (x:xs) = ins x (iSort xs)
```

```
ins :: Int -> [Int] -> [Int]
ins x []          = [x]
ins x (y:ys)
    | x <= y      = x : y : ys
    | otherwise = y : ins x ys
```

Testando iSort

- Duas propriedades para testar
  - A lista produzida por iSort está efetivamente ordenada
  - A lista produzida por iSort é uma permutação da lista original
- Foquemos na permutação
  - `xs` é permutação de `ys` sse para qualquer `n`, o número de ocorrências de `n` em `xs` e em `ys` é o mesmo

```haskell
prop_Perm_iSort :: Int -> [Int] -> Bool
prop_Perm_iSort n ns =
    ocorrencias n ns == ocorrencias n (iSort ns)


ocorrencias :: Int -> [Int] -> Int
ocorrencias n [] = 0
ocorrencias n (m:ms)
   | n == m     = 1 + ocorrencias n ms
   | otherwise = ocorrencias n ms
```

```
Main> quickCheck prop_Perm_iSort
...
```