

# Recursão Primitiva

Programação Funcional  
DCOMP-UFS

# Função fatorial

1. `fatorial(0) = 1`

2. **Se** `n > 0`

$$\begin{aligned}\text{fatorial}(n) &= n * \underline{(n-1) * \dots 2 * 1} \\ &= n * \text{fatorial}(n-1)\end{aligned}$$

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

```
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)
```

<b>n</b>	<b>fatorial</b>
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

$$0+1+2+\dots+n$$

Podemos definir a função `somaAte` explorando os seguintes fatos

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

Ou seja:

$$\text{somaAte}(n) = \text{somaAte}(n-1) + n$$

```
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:  
 $0! + 1! + 2! + \dots + n!$
5. Escreva uma função que calcule  
 $2^0 + 2^1 + 2^2 + \dots + 2^n$

5. Dada uma função  $f$  de  $\text{Int}$  em  $\text{Int}$ , defina por recursão primitiva uma função  $\text{maiorF}$  que aceite um natural  $n$  e devolva o maior valor dentre  $f\ 0, f\ 1, \dots$  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  $\text{Int}$  em  $\text{Int}$ , defina por recursão primitiva uma função  $\text{algumF0}$  que aceite um natural  $n$  e devolva o booleano  $\text{True}$  se e somente se um ou mais valores de  $f\ 0, f\ 1, \dots, f\ n$  é zero. Teste com diferentes definições de  $f$ .
8. Dada um função  $f$  de  $\text{Int}$  em  $\text{Bool}$ , defina por recursão primitiva uma função  $\text{algumFentre}$  que aceite um natural  $n$  e devolva o booleano  $\text{True}$  se e somente se  $f\ i$  é  $\text{True}$  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 cabeça e uma lista resto
- 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