

Construção de Funções por Indução

Aula 7

Indução na Construção de Funções

- O princípio da indução matemática pode ser utilizado na construção de funções.
- Construção por indução gera funções naturalmente **recursivas**.
- Uma função é recursiva quando sua definição usa a própria função.
 - Ex: $fatorial(n) = n.fatorial(n-1)$
- Indução Fraca leva a funções com **recursão primitiva**.
- Caso base -> condição de parada das funções recursivas.
- Hipótese de indução -> chamada recursiva da função para uma entrada de tamanho menor; no caso da indução fraca para $n = k-1$.
- Caso geral -> estratégia de solução onde a solução obtida para $n=k-1$ é estendida para se obter a solução para o caso em que $n=k$.

Um exemplo bem simples

- Construa uma função para calcular a^n , $n \geq 0$. Por convenção, $0^0 = 1$.
- Construção por indução e formulação recursiva.
 - Variável de indução: n .
 - **Caso base:** $n_0 = 0$. Neste caso a função retorna 1 pois $a^0 = 1$.
 - **Hipótese de indução:** Suponha que sabemos calcular para uma entrada $n = k-1$. Ou seja sabemos calcular a^{k-1}
 - **Caso geral:** O objetivo é calcular quando $n = k$, ou seja a^k . Mas queremos usar a HI. Ou seja, precisamos criar uma estratégia que calcule a^k a partir do valor de a^{k-1} , que supomos já calculado.

Usando álgebra sabemos que $a^k = a \cdot a^{k-1}$. Então nossa estratégia consiste em apenas multiplicar por a o valor de a^{k-1} !!!

Um exemplo bem simples

- Construa uma função para calcular a^n , $n \geq 0$. Por convenção, $0^0 = 1$.
- Poderíamos escrever a função, se considerarmos apenas o caso de sucesso

```
pot :: Float -> Int -> Float
pot _ 0 = 1                -- caso base
pot a k = a * pot a (k-1)  -- caso geral
```

- Para tratar o caso de insucesso poderíamos fazer

```
pott :: Float -> Int -> Float
pott a k
  | k < 0 = error "expoente negativo"
  | otherwise = pot a k
```


Um exemplo bem simples

- Avaliação

pot 2 3 \approx 2 * (pot 2 2) \approx

2 * 2 * (pot 2 1) \approx

2 * 2 * 2 * (pot 2 0) \approx

2 * 2 * 2 * 1 \approx 8

Exemplos

- A maioria das funções básicas de listas já vistas pode ser construída por indução fraca. Veremos algumas delas.
- Construa uma função para calcular o tamanho de uma lista de inteiros. Esta seria a função `length` aplicada a uma lista de inteiros.
- Entendendo o problema $\text{length } [] = 0$; $\text{length } [5] = 1$;
 $\text{length } [2, 3, 4] = 3$
 - Variável de indução: tamanho da lista, n .
 - **Caso base:** $n_0 = 0$, ou seja a lista vazia. Neste caso a função retorna o (zero), porque a quantidade de elementos de uma lista vazia é zero.
 - **Hipótese de indução:** Suponha que sabemos calcular o tamanho de uma lista para $n=k-1$. Ou seja sabemos calcular o tamanho de uma lista de inteiros com $k-1$ elementos.
 - **Caso geral:** O objetivo é calcular quando $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que calcule o tamanho de uma lista de k inteiros a partir do tamanho de uma lista de $k-1$ inteiros, que já supomos saber calcular.

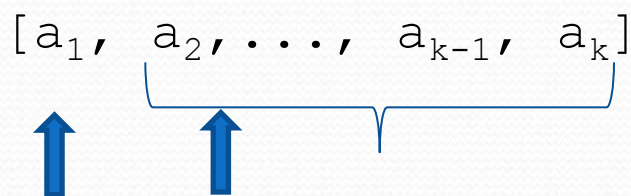
Exemplos

- **Caso geral:** O objetivo é calcular quando $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que calcule o tamanho de uma lista de k inteiros a partir do tamanho de uma lista de $k-1$ inteiros, que já supomos saber calcular.

Estratégia:

tamanho da lista de k elementos = 1 + tamanho da lista de $k-1$ elementos

$[a_1, a_2, \dots, a_{k-1}, a_k]$



x xs , lista de $k-1$ elementos, onde posso aplicar a HI (chamada recursiva)

`meuLength :: [Int] -> Int`

`meuLength [] = 0` `-- caso base`

`meuLength (x:xs) = 1 + meuLength xs` `-- caso geral`

Exemplos

- Avaliação

```
meuLength [3,10,7] ~ 1 + (meuLength [10,7]) ~  
1 + 1 + (meuLength [7]) ~  
1 + 1 + 1 + (meuLength []) ~  
1 + 1 + 1 + 0 ~ 3
```


Exemplos

- Construa uma função para efetuar a soma dos números de uma lista de inteiros de tamanho n . Esta seria a função `sum` aplicada numa lista de inteiros.
 - Variável de indução: tamanho da lista, n .
 - **Caso base:** $n_0 = 0$, ou seja a lista vazia. Neste caso a função retorna o (zero), porque zero é o elemento neutro da soma.
 - **Hipótese de indução:** Suponha que sabemos calcular para uma entrada $n = k - 1$. Ou seja sabemos calcular a soma de uma lista de inteiros de tamanho $k - 1$.
 - **Caso geral:** O objetivo é calcular quando $n = k$, usando a HI. Ou seja, precisamos criar uma estratégia que calcule a soma de uma lista de k inteiros a partir da soma de uma lista de $k - 1$ inteiros, que já supomos saber calcular.

Exemplos

- Observe que somar os elementos de $[a_1, a_2, \dots, a_{k-1}, a_k]$ é o mesmo que somar o valor de a_1 com a soma dos elementos de $[a_2, \dots, a_{k-1}, a_k]$. Esta lista tem $k-1$ elementos, e por HI eu já sei calcular esta soma

$[a_1, a_2, \dots, a_{k-1}, a_k]$



x



xs

lista de $k-1$ elementos, onde posso aplicar a HI

- Poderíamos escrever a função:

```
meuSum :: [Int] -> Int
```

```
meuSum [] = 0 -- caso base
```

```
meuSum (x:xs) = x + meuSum xs -- caso geral
```


Exemplos

- Avaliação

$$\begin{aligned}\text{meuSum } [3, 10, 7] &\sim 3 + (\text{meuSum } [10, 7]) \sim \\ &3 + 10 + (\text{meuSum } [7]) \sim \\ &3 + 10 + 7 + (\text{meuSum } []) \sim \\ &3 + 10 + 7 + 0 \sim 20\end{aligned}$$

Exemplos

- Construa uma função para calcular o fatorial de um número inteiro n , $n \geq 0$.
 - Variável de indução: n .
 - **Caso base:** $n_0 = 0$. Neste caso a função retorna 1, porque 1 é o fatorial de 0 (zero).
 - **Hipótese de indução:** Suponha que sabemos calcular para uma entrada $n = k-1$. Ou seja sabemos calcular o fatorial de $k-1$.
 - **Caso geral:** O objetivo é calcular quando $n = k$, usando a HI. Ou seja, precisamos criar uma estratégia que calcule o fatorial de k a partir do fatorial de $k-1$, que já supomos saber calcular.

Exemplos

- Observe que o fatorial de k é dado por $1.2.3.4.5....(k-1).k$. Como o produto é associativo isto é o mesmo que $(1.2.3....k-1).k$.
- Logo a estratégia é calcular o fatorial dos $k-1$ elementos usando HI e depois multiplicar o resultado por k

- Poderíamos escrever a função:

```
meuFat :: Int -> Int
```

```
meuFat 0 = 1 -- caso base
```

```
meuFat n = meuFat (n-1) * n -- caso geral
```

- Como posso ter caso de insucesso...

```
meuFatorial :: Int -> Int
```

```
meuFatorial n
```

```
    | n < 0 = "error numero negativo"
```

```
    | otherwise = meuFat n --
```

Exemplos

- Avaliação

`meuFat 3 ~ (meuFat 2) * 3 ~`
`(meuFat 1) * 2 * 3 ~`
`(meuFat 0) * 1 * 2 * 3 ~`
`1 * 1 * 2 * 3 ~ 6`

Exercício de Fixação

- Construa uma função para efetuar o produto dos números de uma lista de inteiros de tamanho n . Esta seria a função `product` aplicada numa lista de inteiros.

Exercício de Fixação

- Construa uma função para efetuar o produto dos números de uma lista de inteiros de tamanho n . Esta seria a função `product` aplicada numa lista de inteiros.

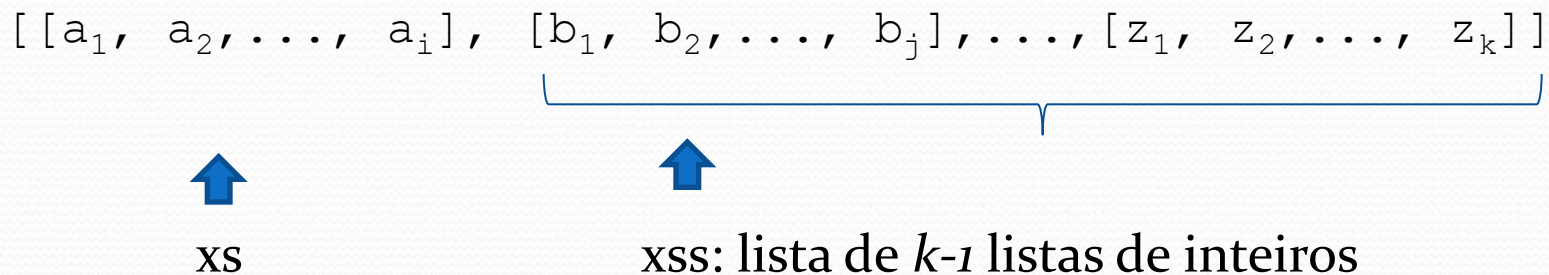
```
meuProd :: [Int] -> Int
meuProd [] = 1 -- caso base
meuProd (x:xs) = x * meuProd xs -- caso geral
```


Exemplos

- Construa uma função para concatenar as listas de uma lista de listas de inteiros. Esta seria a função `concat` aplicada a uma lista de listas de inteiros.
 - Variável de indução: n , tamanho da lista
 - **Caso base:** $n_0 = []$. Neste caso a função retorna `[]`, que o resultado da concatenação de lista vazia.
 - **Hipótese de indução:** Suponha que sabemos concatenar para uma entrada $n=k-1$. Ou seja sabemos realizar a concatenação de $k-1$ listas de inteiros.
 - **Caso geral:** O objetivo é concatenar quando $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que concatene k listas de inteiros a partir do resultado da concatenação de $k-1$ listas de inteiros, que já supomos saber calcular.

Exemplos

- **Caso geral:** O objetivo é concatenar quando $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que concatene k listas de inteiros a partir do resultado da concatenação de $k-1$ listas de inteiros, que já supomos saber calcular.



```
meuConcat :: [[Int]] -> [Int]
```

```
meuConcat [] = [] -- caso base
```

```
meuConcat (xs:xss) = xs ++ meuConcat xss -- caso geral
```


Exemplos

- Avaliação

```
meuConcat [[3,10,7],[2,4],[5]] ~  
[3,10,7]++ meuConcat [[2,4],[5]] ~  
[3,10,7]++[2,4]++meuConcat[[5]] ~  
[3,10,7]++[2,4]++ [5] ++ meuConcat[] ~  
[3,10,7]++[2,4]++ [5] ++ [] ~  
[3,10,7]++[2,4]++ [5] ++ [] ~  
[3,10,7]++[2,4]++ [5] ~  
[3,10,7]++[2,4,5] ~  
[3,10,7,2,4,5]
```

Exemplos

- Construa uma função para reverter a ordem dos elementos de uma lista de listas de inteiros. Esta seria a função `reverse` aplicada a uma lista de inteiros.
 - Variável de indução: n , tamanho da lista
 - **Caso base:** $n_0 = []$. Neste caso a função retorna `[]`, pois não há elementos a reverter na lista vazia.
 - **Hipótese de indução:** Suponha que sabemos reverter para uma entrada $n=k-1$. Ou seja sabemos reverter a ordem dos elementos de uma lista de inteiros de tamanho $k-1$.
 - **Caso geral:** O objetivo é reverter quando $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que reverte a ordem dos k elementos de uma lista de inteiros a partir do resultado da reversão dos $k-1$ elementos de uma lista de inteiros, que já supomos saber calcular.

Exemplos

- **Caso geral:** O objetivo é reverter quando $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que reverta a ordem dos k elementos de uma lista de inteiros a partir do resultado da reversão dos $k-1$ elementos de uma lista de inteiros, que já supomos saber calcular.

Estratégia:

$[a_1, a_2, \dots, a_{k-1}, a_k] \rightarrow [a_k, a_{k-1}, \dots, a_2, a_1]$

x ↑

xs ↑

lista de $k-1$ elementos, onde posso aplicar a HI (chamada recursiva)

```
meuReverse :: [Int] -> [Int]
```

```
meuReverse [] = [] -- caso base
```

```
meuReverse (x:xs) = meuReverse xs ++ [x] -- caso geral
```

Exemplos

- Avaliação

`meuReverse [3,10,7] ~`

`meuReverse [10,7] ++ [3] ~`

`meuReverse [10] ++ [7] ++ [3] ~`

`meuReverse [] ++ [10] ++ [7] ++ [3] ~`

`[] ++ [10] ++ [7] ++ [3] ~`

`[] ++ [10] ++ [7,3] ~`

`[] ++ [10,7,3] ~`

`[10,7,3]`

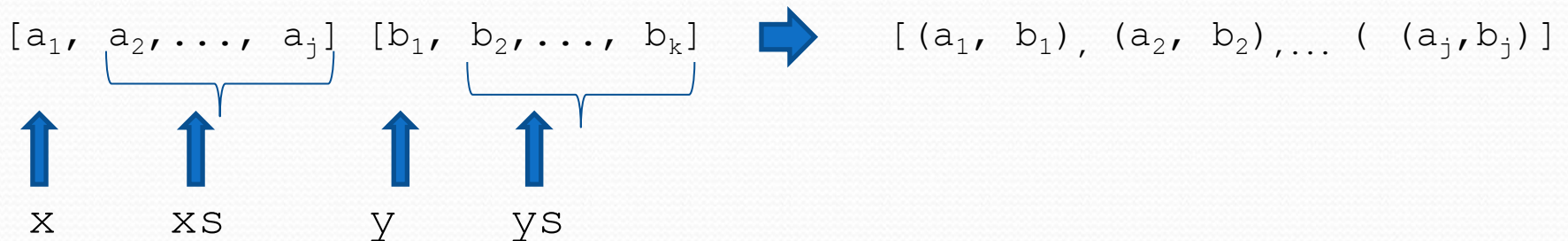
Exemplos

- Construa uma função similar à função `zip` para duas listas de inteiros. Ex: `zip [1, 2] [3, 4, 5] = [(1, 3), (2, 4)]`.
 - Variável de indução: n , tamanho da menor lista
 - **Caso base**: pelo menos uma lista é `[]`. Neste caso a função retorna `[]`, pois não há elementos a transformar em pares.
 - **Hipótese de indução**: Suponha que sabemos “zipar” para uma entrada $n=k-1$. Ou seja sabemos “zipar” os elementos de duas listas de inteiros de tamanho pelo menos $k-1$.
 - **Caso geral**: O objetivo é “zipar” quando $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que “zipe” duas lista de inteiros com pelo menos k elementos a partir do resultado da “zipagem” de listas de inteiros com pelo menos $k-1$ elementos, que já supomos saber calcular.

Exemplos

- **Caso geral:** O objetivo é “zipar” quando $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que “zipe” duas lista de inteiros com pelo menos k elementos a partir do resultado da “zipagem” de listas de inteiros com pelo menos $k-1$ elementos, que já supomos saber calcular.

Estratégia: Supondo $j \leq k$,



```
meuZip :: [Int] -> [Int] -> [(Int, Int)]  
meuZip _ [] = [] -- caso base  
meuZip [] _ = []  
meuZip (x:xs) y:ys = (x,y) : meuZip xs ys -- caso geral
```


Exemplos

- Avaliação

```
meuZip [1,2] [3,4,5] ~
```

```
    (1,3) : meuZip [2] [4,5] ~
```

```
    (1,3) : (2,4) : meuZip [] [5] ~
```

```
    (1,3) : (2,4) : [] ~
```

```
    (1,3) : [(2,4)] ~
```

```
    [(1,3), (2,4)]
```

Exemplos

- Construa uma função similar à função `take` para uma lista de inteiros. Ex: `take 2 [3, 4, 5] = [3, 4]`.
 - Variáveis de indução: menor entre i e n , quantidade de itens e tamanho da lista, respectivamente
 - **Casos bases**: (a) $i=0$ ou lista = []. Nestes casos a função retorna [], pois não há elementos para serem coletados.
 - **Hipótese de indução**: Suponha que sabemos coletar os elementos para uma entrada com $m = j-1$ e $n=k-1$.
 - **Caso geral**: O objetivo é coletar elementos quando $m = j$ e $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que colete j elementos de listas de inteiros com pelo menos k elementos, a partir da coleta de $j-1$ elementos de uma lista de inteiros com pelo menos $k-1$ elementos que já supomos saber calcular.

Exemplos

- **Caso geral:** O objetivo é coletar elementos quando $m = j$ e $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que colete j elementos de listas de inteiros com pelo menos k elementos, a partir da coleta de $j-1$ elementos de uma lista de inteiros com pelo menos $k-1$ elementos que já supomos saber calcular.

Estratégia:

```
meuTake :: Int -> [Int] -> [Int]
meuTake _ [] = []                                -- caso base
meuTake 0 _ = []
meuTake i (x:xs) = x: meuTake (i-1) xs          -- caso geral
```

Tratando o caso especial quando $i < 0$. Por convenção, adota-se retornar [].

```
meuTakeGeral :: Int -> [Int] -> [Int]
meuTakeGeral i ls
  | i < 0 = []
  | otherwise = meuTake i ls
```

Exemplos

- Avaliação

```
meuTake 2 [3,4,5] ~  
      3 : meuTake 1 [4,5] ~  
      3: 4: meuTake 0 [5] ~  
      3: 4: [] ~  
      3: [4] ~  
      [3,4]
```


Exemplos

- Construa uma função para retornar os elementos pares de uma lista de inteiros. Ex: `coletaPares [2, 4, 5] = [2, 4]`.
 - Variáveis de indução: n , tamanho da lista
 - **Caso base**: `lista = []`. Nestes casos a função retorna `[]`, pois não há elementos para serem coletados.
 - **Hipótese de indução**: Suponha que sabemos coletar os elementos pares de uma lista de inteiros com $n=k-1$ elementos.
 - **Caso geral**: O objetivo é coletar elementos pares de uma lista de inteiros de tamanho $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que colete os elementos pares de listas de inteiros com k elementos, a partir da coleta de pares numa lista com $k-1$ elementos, que já supomos saber calcular.

Exemplos

- **Caso geral:** O objetivo é coletar elementos pares de uma lista de inteiros de tamanho $n=k$, usando a HI. Ou seja, precisamos criar uma estratégia que colete os elementos pares de listas de inteiros com k elementos, a partir da coleta de pares numa lista com $k-1$ elementos, que já supomos saber calcular.

Estratégia:

```
coletaPares :: [Int] -> [Int]
coletaPares [] = []                                -- caso base
coletaPares x:xs
  | ehPar x = x: coletaPares xs                      -- caso geral
  | otherwise = coletaPares xs
where ehPar p = (mod p 2 == 0)
```


Exemplos

- Avaliação

coletaPares [2,4,5] \rightsquigarrow

2 : coletaPares [4,5] \rightsquigarrow

2 : 4 : coletaPares [5] \rightsquigarrow

2 : 4 : coletaPares [] \rightsquigarrow

2 : 4 : [] \rightsquigarrow

2 : [4] \rightsquigarrow

[2,4]

Exercícios Recomendados

- Elabore funções recursivas que realizem o mesmo que as funções `replicate`, `and`, `or`, `drop` e `unzip`.
- Elabore uma função recursiva para calcular a soma dos números pares de uma lista
- Elabore uma função recursiva para calcular o produto dos números ímpares de uma lista
- Elabore uma função recursiva para dado uma string qualquer devolver uma string contendo apenas os dígitos da string.
- Elabore uma função recursiva `elemNum` para dados uma lista de inteiros e um número inteiro, devolve a quantidade de vezes que este número ocorre na lista.
- Usando a função `elemNum` defina uma função `unicos` que dada uma lista de inteiros `ls` devolve uma lista contendo os elementos de `ls` que ocorrem apenas uma vez. Ex: `unicos [4,2,1,2,3,3]` retorna `[4,1]`.

Exercícios Recomendados

- Elabore uma função recursiva para dados um inteiro e uma lista de inteiros retornar `True` se o elemento está na lista e `False` caso contrário.
- Refaça o exercício da biblioteca definindo funções recursivas para:
 - Dada uma pessoa, encontre os livros que ela emprestou;
 - Dado um livro, encontre quem emprestou este livro, assumindo que o livro pode ter mais de um exemplar;
 - Dado um livro, desejamos saber se o mesmo se encontra emprestado ou não;
 - Dada uma pessoa, desejamos saber a quantidade de livros que ela tomou emprestado;
 - Dado um par (Pessoa, Livro), queremos removê-lo da lista de emprestados, sinalizando a sua devolução.