Recursão

Programação Funcional

Prof. Rodrigo Ribeiro

Objetivos

- Definir funções recursivas em Haskell
- Definir algoritmos de ordenação sobre listas.

Setup

```haskell
module Aula04 where

import Prelude hiding ( replicate
                      , elem
                      , sum
                      , reverse )

main :: IO ()
main = return ()
```

Funções recursivas

- Exemplos clássicos.

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

Funções recursivas

- Necessariamente, devem ter um caso base.
- Chamadas recursivas devem ser feitas sobre argumentos de tamanho “menor”.

Como obter os tipos mostrados

- Em versões atuais de Haskell, algumas funções foram "generalizadas".
- Logo, o tipo que apresentarei nas aulas iniciais não será o mesmo obtido ao executar `:t nome_função` no GHCi.

Como obter os tipos mostrados

- Para obter os tipos mostrados, inicie o interpretador usando `stack ghci --ghci-options -XTypeApplications`.
- E então use: `:t sum @[]`

Recursão sobre listas

```
length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs
```

Exemplo

```haskell
length :: [a] -> Int
length [] = 0                    -- 1
length (_ : xs) = 1 + length xs -- 2
```

- Execução

```haskell
length "abc"     == (2)
1 + length "bc" ==
```

Exemplo

```haskell
length :: [a] -> Int
length [] = 0                          -- 1
length (_ : xs) = 1 + length xs -- 2
```

- Execução

```haskell
length "abc"            == (2)
1 + length "bc"         == (2)
1 + (1 + length "c") ==
```

Exemplo

```haskell
length :: [a] -> Int
length [] = 0                        -- 1
length (_ : xs) = 1 + length xs -- 2
```

- Execução

```haskell
length "abc"           == (2)
1 + length "bc"        == (2)
1 + (1 + length "c") == (2)
1 + (1 + (1 + length "")) ==
```

Exemplo

```haskell
length :: [a] -> Int
length [] = 0                    -- 1
length (_ : xs) = 1 + length xs -- 2
```

- Execução

```haskell
length "abc"          == (2)
1 + length "bc"       == (2)
1 + (1 + length "c") == (2)
1 + (1 + (1 + length "")) ==
1 + (1 + (1 + 0)) == 3
```

Recursão sobre listas

- Concatenação de listas

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

Exemplo

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys  -- 1
(x : xs) ++ ys = x : (xs ++ ys) -- 2
```

- Execução

```
[1,2] ++ [3,4,5,6] == (2)
1 :([2] ++ [3,4,5,6]) ==
```

Exemplo

```haskell
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys  -- 1
(x : xs) ++ ys = x : (xs ++ ys) -- 2
```

- Execução

```haskell
[1,2] ++ [3,4,5,6] == (2)
1 :([2] ++ [3,4,5,6]) == (2)
1 : (2 : ([] ++ [3,4,5,6])) ==
```

Exemplo

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys  -- 1
(x : xs) ++ ys = x : (xs ++ ys) -- 2
```

- Execução

```
[1,2] ++ [3,4,5,6] == (2)
1 :([2] ++ [3,4,5,6]) == (2)
1 : (2 : ([] ++ [3,4,5,6])) == (1)
1 : (2 : [3, 4,5,6]) ==
[1,2,3,4,5,6]
```

Exemplo

- Definir a função `replicate`:

```
replicate :: Int -> a -> [a]
```

Exemplo

```haskell
replicate :: Int -> a -> [a]
replicate n x
  | n <= 0 = []
  | otherwise = x : replicate (n - 1) x
```

- Como definir `replicate` usando list comprehensions?

Receita para funções recursivas

1. Defina o tipo da função.
2. Enumere os casos.
3. Defina os casos base.
4. Defina os casos recursivos.

Exemplo: Testando pertinência

1. Definindo o tipo da função

```haskell
elem :: Eq a => a -> [a] -> Bool
```

Exemplo: Testando pertinência

2. Enumerando os casos.

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = _
elem x (y : ys) = _
```

Exemplo: Testando pertinência

3. Definindo caso base.

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y : ys) = _
```

Exemplo: Testando pertinência

4. Definindo caso recursivo.

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

Exemplo

- Desenvolver uma função para somar elementos de uma lista de inteiros.

Passo 1: Definir o tipo.

```
sum :: [Int] -> Int
```

Exemplo

Passo 2: Enumere os casos.

```
sum :: [Int] -> Int
sum [] = _
sum (x : xs) = _
```

Exemplo

Passo 3: Defina o(s) caso(s) base

```
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = _
```

Exemplo

Passo 4: Defina o(s) caso(s) recursivos.

- Como obter `sum (x:xs)` a partir do resultado de `sum xs`?
  - Fácil! Basta somar `x` a `sum xs`!

Exemplo

- Solução

```
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + sum xs
```

Invertendo listas

- Passo 1. Definir o tipo.

```
reverse :: [a] -> [a]
```

Invertendo listas

- Passo 2. Enumere os casos.

```
reverse :: [a] -> [a]
reverse [] = _
reverse (x : xs) = _
```

Invertendo listas

- Passo 3. Definindo o caso base.

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x : xs) = _
```

Invertendo listas

- Passo 4. Definindo o caso recursivo.

```
reverse :: [a] -> [a]
reverse [] = [] -- 1
reverse (x : xs) = reverse xs ++ [x] -- 2
```

Invertendo listas

- Exemplo

```
reverse [True, False, False] == (2)
reverse [False, False] ++ [True] ==
```

Invertendo listas

- Exemplo

```
reverse [True, False, False] == (2)
reverse [False, False] ++ [True] == (2)
(reverse [False] ++ [False]) ++ [True] ==
```

Invertendo listas

- Exemplo

```
reverse [True, False, False] == (2)
reverse [False, False] ++ [True] == (2)
(reverse [False] ++ [False]) ++ [True] == (2)
((reverse [] ++ [False]) ++ [False]) ++ [True] ==
```

Invertendo listas

- Exemplo

```
reverse [True, False, False] == (2)
reverse [False, False] ++ [True] == (2)
(reverse [False] ++ [False]) ++ [True] == (2)
((reverse [] ++ [False]) ++ [False]) ++ [True] == (1)
(([] ++ [False]) ++ [False]) ++ [True] ==
[False, False, True]
```

Invertendo listas

- Problema: Essa versão de `reverse` usa ++.
- Como ++ é $O(n)$, temos que reverse é da ordem de $O(n^2)$.

Invertendo listas

- Em linguagens imperativas, é possível inverter listas em tempo $O(n)$.
- Como implementar esse algoritmo de forma eficiente?

Invertendo listas

- Podemos fazer isso eficientemente, usando um acumulador.

```
rev :: [a] -> [a]
rev xs = rev' xs []
   where
     rev' [] acc        = acc -- 1
     rev' (z : zs) acc = rev' zs (z : acc) -- 2
```

Invertendo listas

- Exemplo.

```
rev [1,2,3] ==
rev' [1,2,3] [] ==
```

Invertendo listas

- Exemplo.

```
rev [1,2,3] ==
rev' [1,2,3] [] == (2)
rev' [2,3] (1 : []) ==
```

Invertendo listas

- Exemplo.

```
rev [1,2,3] ==
rev' [1,2,3] [] == (2)
rev' [2,3] (1 : []) == (2)
rev' [3] (2 : (1 : [])) ==
```

Invertendo listas

- Exemplo.

```
rev [1,2,3] ==
rev' [1,2,3] [] == (2)
rev' [2,3] (1 : []) == (2)
rev' [3] (2 : (1 : [])) == (2)
rev' [] (3 : (2 : (1 : []))) ==
```

Invertendo listas

- Exemplo.

```
rev [1,2,3] ==
rev' [1,2,3] [] == (2)
rev' [2,3] (1 : []) == (2)
rev' [3] (2 : (1 : [])) == (2)
rev' [] (3 : (2 : (1 : []))) == (1)
(3 : (2 : (1 : []))) == [3,2,1]
```

Inserindo em listas

- Inserindo elementos em uma lista ordenada de inteiros.
- Função utilizada pelo algoritmo insertion sort.

Inserindo em listas

- Passo 1. Definindo o tipo.

```
insert :: Int -> [Int] -> [Int]
```

Inserindo em listas

- Passo 2. Enumere os casos

```
insert :: Int -> [Int] -> [Int]
insert x [] = _
insert x (y : ys) = _
```

Inserindo em listas

- Passo 3. Defina o caso base.

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y : ys) = _
```

Inserindo em listas

- Passo 4. Defina o caso recursivo.
- Dividimos esse em dois casos.

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

Inserindo em listas

- Passo 4. Defina o caso recursivo.
- Caso `x <= y`: `x` é a nova cabeça da lista.

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

Inserindo em listas

- Passo 4. Defina o caso recursivo.
- Caso `x > y`: `x` é inserido na cauda.

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

Insertion sort

- Passo 1. Definindo o tipo.

```
isort :: [Int] -> [Int]
```

Insertion sort

- Passo 2. Enumerando os casos

```
isort :: [Int] -> [Int]
isort [] = _
isort (x:xs) = _
```

Insertion sort

- Passo 3. Definindo o caso base.

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = _
```

Insertion sort

- Passo 4. Definindo o caso recursivo.

```
isort :: [Int] -> [Int]
isort [] = []
isort (x : xs) = insert x (isort xs)
```

Mergesort

- Intercalação de listas ordenadas.
- Passo 1. Qual o tipo da função?

Mergesort

```
merge :: [Int] -> [Int] -> [Int]
```

Mergesort

Passo 2. Defina os casos.

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = _
merge xs [] = _
merge (x : xs) (y : ys) = _
```

Mergesort

Passo 3. Defina os casos base.

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge xs [] = xs
merge (x : xs) (y : ys) = _
```

Mergesort

Passo 4. Defina o caso recursivo.

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

Mergesort

```haskell
msort :: [Int] -> [Int]
msort [] = []
msort [x] = [x]
msort xs = merge (msort ys) (msort zs)
    where
      n = length xs `div` 2
      ys = take n xs
      zs = drop n xs
```

Exercício

- Defina a função `minList` que retorna o menor inteiro de uma lista de números fornecida como entrada. Siga cada um dos passos anteriores para sua solução.

Exercício

- Implemente a função `andList` que produz a conjunção de uma lista de booleanos fornecida como entrada.
- Implemente a função `orList` que produz a disjunção de uma lista de booleanos fornecida como entrada.

Exercício

- Implemente a função `indexOf` que, a partir de um inteiro `x` e uma lista de inteiros `xs`, retorna a posição de `x` na lista `xs`. Caso `x` não pertença a lista, o valor -1 deve ser retornado.

Exercício

- A função `takeList` recebe como entrada um número inteiro *n* e uma lista *xs* e retorna, como resultado, uma lista contendo os primeiros *n* de *xs*. Implemente `takeList` seguindo os passos apresentados para definir uma função recursiva.

Exercício

- A função `dropList` recebe como entrada um número inteiro n e uma lista xs e remove os n primeiros elementos de xs. Implemente `dropList` seguindo os passos apresentados para definir uma função recursiva.

Exercício

- Implemente a função `removeAll` que, a partir de um inteiro `x` e uma lista de inteiros `xs`, remove todas as ocorrências de `x` da lista `xs`. Apresente duas implementações: 1) usando recursividade e 2) usando list comprehensions.

Exercício

- Implemente a função `countPos` que, a partir de uma lista de números inteiros, retorna a sua quantidade de números positivos.