Linguagens de Programação

Programação Funcional e Haskell
Funções e Expressões Lambda

Thiago Alves

Introdução

A forma mais direta de definição de funções é pela combinação de funções existentes.

```
isDigit  :: Char -> Bool
isDigit c = (c >= '0') && (c <= '9')

even :: a -> bool
even a = n `mod` 2 == 0

splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)
```

Defina a função

```
halve :: [a] → ([a], [a])
```

que divide uma lista de tamanho par em duas metades

```
halve xs = splitAt (length xs `div` 2) xs
```

Defina a função

```
halve :: [a] → ([a], [a])
```

que divide uma lista de tamanho par em duas metades

Expressões Condicionais

Como na maioria das linguagens de programação, funções podem ser definidas usando expressões condicionais.

```
abs  :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs recebe um inteiro n e retorna n se é não negativo e -n, caso contrário.

Defina a função signum que recebe um inteiro e retorna -1 quando o inteiro é negativo, retorna 1 quando o inteiro positivo e retorna 0 quando o inteiro é nulo

```
signum  :: Int → Int
```

Expressões condicionais podem ser aninhadas:

```
signum  :: Int → Int
signum n = if n < 0 then -1 else if n == 0 then 0 else 1
```

```
signum  :: Int → Int
signum n = if n < 0 then -1 else
                    if n == 0 then 0 else 1
```

Ambiguidade de Condicionais

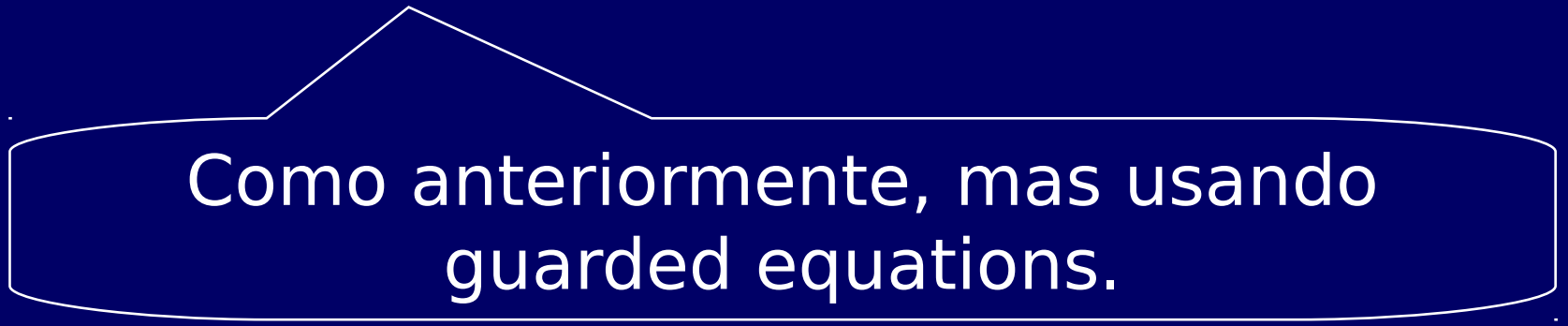
- Em Haskell, expressões condicionais devem sempre ter else
- Evitando possíveis problemas de ambiguidade com condicionais aninhados

```
if True then if False then 1 else 2
```

Guarded Equations

Como alternativa aos condicionais, funções podem ser definidas usando guarded equations.

```
abs n | n ≥ 0      = n
      | otherwise = -n
```



Use Guarded equations para escrever uma definição para a função signum:

Guarded equations podem ser usadas para tornar definições envolvendo múltiplas condições mais fáceis de ler:

```
signum n | n < 0       = -1
         | n == 0      = 0
         | otherwise = 1
```

- otherwise trata todos os outros casos para não ocorrer erro
- Não é obrigatório

Pattern Matching

Várias funções possuem uma definição mais clara usando pattern matching nos seus argumentos.

```
not      :: Bool → Bool
not False = True
not True  = False
```

not mapeia False para True, e True para False.

- O underscore _ é um padrão coringa que casa qualquer valor de argumento.
- Cada ocorrência de _ funciona como um variável diferente

```
True  && b = b
False && _ = False
```

- Padrões são casados na ordem. Por exemplo, a seguinte definição sempre retorna False:

```
_    && _    = False
True && True = True
```

- Padrões não devem repetir variáveis. Por exemplo, a seguinte definição retorna um erro:

```
b && b = b
_ && _ = False
```

Padrões de Tupla

Uma tupla de padrões é um padrão

```
fst :: (a,b) → a
fst (x,_) = x

snd :: (a,b) → b
snd (_,y) = y
```

Padrões de Lista

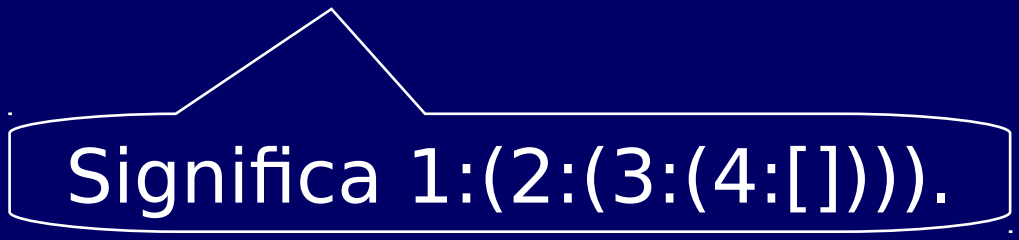
Uma lista de padrões é um padrão que dá match com listas de mesmo tamanho em que os elementos dão match com o padrão correspondente

```
firstAThree :: [Char] → Bool
firstAThree ['a',_,_] = True
firstAThree _ = False
```

Padrões de Lista

Internamente, toda lista não-vazia é construída pelo uso repetido do operador (:) chamado “cons” que adicona um elemento ao início da lista.

[1,2,3,4]



Funções em listas podem ser definidas usando padrões x:xs.

```
head :: [a] → a
head (x:_)  = x

tail :: [a] → [a]
tail (_:xs) = xs
```

- Padrões x:xs apenas casam com listas não-vazias:

```
> head []
ERROR
```

- Padrões (x:xs) devem ser parentisados
- A função tem prioridade mais alta
- A seguinte definição retorna um erro:

```
head x:_ = x
```

Escreva uma definição para a função firsta que recebe uma lista de caracteres e retorna True se o primeiro elemento da lista é 'a', e retorna False, caso contrário.

```
firsta :: [Char] -> Bool
```

Escreva uma definição para a função firsta que recebe uma lista de caracteres e retorna True se o primeiro elemento da lista é 'a', e retorna False, caso contrário.

```
firsta :: [Char] -> Bool
firsta ('a':_) = True
firsta _ = False
```

Padrões de Inteiros

Haskell permite padrões de inteiros da forma n + k em que n é variável e k>0 é constante

```
pred :: Int → Int
pred 0       = 0
pred (n + 1) = n
```

Considere a função safetail que se comporta da mesma forma que tail, exceto que safetail mapeia a lista vazia para a lista vazia, enquanto tail retorna um erro nesse caso. Defina safetail usando:

(a) uma expressão condicional;
(b) guarded equations;
(c) pattern matching.

Dica: a função null :: [a] $\rightarrow$ Bool pode ser usada para testar se uma lista é vazia.

```
safeTail xs = if null xs then [] else tail xs
```

```
safeTail xs
    | null xs = []
    | otherwise tail xs
```

```
safeTail [] = []
safetail (x:xs) = xs
```

Expressões Lambda

Funções podem ser construídas sem nome usando expressões lambda.

$\lambda x \rightarrow x + x$

A função sem nome que recebe um número x e retorna o resultado x + x.

- O símbolo λ é a letra Grega lambda, e é digitado no teclado como \.
- Em Haskell, o uso do símbolo λ para funções sem nome vem do lambda calculus, a teoria de funções na qual Haskell é baseada.
- Funções definidas com expressões lambdas podem ser usadas da forma usual

```
> (\x -> x + x) 2
4
```

Utilidade do Lambda?

Expressões lambda podem ser usadas para fornecer um significado para as funções definidas usando currificação.

Por exemplo:

```
add x y = x + y
```

significa

```
add = \x → (\y → x + y)
```

Expressões lambda também são úteis para definição de funções que retornam funções como resultados.

```
power :: Int → (Int → Int)
power n = \x → x^n

square = power 2
cube = power 3
```

Expressões lambda podem ser usadas para evitar nomear funções que só são referenciadas uma vez.

Por exemplo:

```
f x = x*2 + 1
odds n = map f [0..n-1]
```

Pode ser simplificado para

Aplicada em cada elemento da lista

```
odds n = map (\x → x*2 + 1) [0..n-1]
```

Uma empresa quer dar um aumento de 10% para os funcionários e quer saber o gasto total depois do aumento.

Defina uma função novoGasto que recebe uma lista de salários e retorna o gasto da empresa com os salários atualizados.

```
novoGasto :: [Float] -> Float
```

Uma empresa quer dar um aumento de 10% para os funcionários e quer saber o gasto total depois do aumento.

Defina uma função novoGasto que recebe uma lista de salários e retorna o gasto da empresa com os salários atualizados.

```
novoGasto :: [Float] -> Float
novoGasto xs = sum (map (\x → x*1.1) xs)
```

Seções

Um operador escrito entre seus dois argumentos pode ser convertido em uma função currificada escrita antes dos seus dois argumentos usando parênteses.

Por exemplo:

```
> 1+2
3

> (+) 1 2
3
```

Essa convenção também permite um dos argumentos do operador ser incluído nos parênteses.

Por exemplo:

```
> (1+) 2
3

> (+2) 1
3
```

Em geral, se ⊕ é um operador então funções da forma (⊕), (x⊕) and (⊕y) são chamadas seções.

```
(⊕) = \x → (\y → x ⊕ y)
(x⊕) = \y → x ⊕ y
(⊕y) = \x → x ⊕ y
```

Utilidade das Seções?

- Funções úteis podem ser definidas de forma mais simples usando seções
- Além disso, não é preciso dar nome

Por exemplo:

`(1+)` - função sucessor

`(1/)` - função inversa multiplicativa

`(*2)` - função dobro

`(/2)` - função metade

Utilidade das Seções?

Usar operadores como argumentos para funções

```
novoSal :: [Float] → [Float]
novoSal xs = map (*1.1) xs
```