Programação Funcional

Funções de alta ordem

Prof. Edson Alves

Faculdade UnB Gama

2020

Sumário

1. Funções de alta ordem

2. Mapas, filtros e reduções

3. Funções anônimas

Exemplos de funções das bibliotecas do Haskell

1. A função `lines` recebe uma string e retorna um vetor de strings, o qual corresponde às linhas contidas na string original

```
ghci> :type lines
lines :: String -> [String]

ghci> lines "Hello\nWorld"
["Hello", "World"]
```

2. A função `unlines` é sua inversa: ela recebe um vetor de strings, e une todas elas em uma única string, adicionando o terminador de linha entre elas

```
ghci> :type unlines
unlines :: [String] -> String

ghci> unlines ["a", "b", "c"]
"a\nb\nc\n"
```

3. A função `last` retorna o último elemento da lista

```
ghci> last "ABC"
'C'
```

Exemplos de funções das bibliotecas do Haskell

4. A função complementar de `last` é a função `init`, que retorna todos, menos o último, elementos da lista

```
ghci> init "ABCDE"
"ABCD"
```

5. A função (++) une duas listas em uma única lista

```
ghci> [1..5] ++ [2..4]
[1, 2, 3, 4, 5, 2, 3, 4]
```

6. A função `concat` generaliza este comportamento, recebendo uma lista de listas e as concatenando em uma única lista

```
ghci> :type concat
concat :: [[a]] -> [a]

ghci> concat ["um", "dois", "tres"]
"umdoistres"
```

Exemplos de funções das bibliotecas do Haskell

7. A função `reverse` recebe uma lista `xs` e retorna uma nova lista, com todos os elementos de `xs` em ordem inversa

```
ghci> reverse [1..5]
[5, 4, 3, 2, 1]
```

8. As funções `and` e `or` aplicam as operações lógicas binárias (`&&`) e (`||`) em todos os elementos da lista, até que reste apenas um elemento

```
ghci> and [True, False, True]
False

ghci> or [True, False, True]
True
```

9. A função `splitAt` recebe um inteiro `i` e uma lista `xs`, e retorna um par de listas `(xs[1..i], xs[(i+1)..n])`

Exemplos de funções das bibliotecas do Haskell

10. A função `zip` recebe duas listas `xs` e `ys` e gera uma lista de pares `zs`, cujo tamanho é mesmo da menor dentre as duas, cujos elementos `(xi, yi)` são oriundos destas listas, nesta ordem

```
ghci> :type zip
zip :: [a] -> [b] -> [(a, b)]

ghci> zip [1..] "Teste"
[(1,'T'),(2,'e'),(3,'s'),(4,'t'),(5,'e')]
```

11. As funções `zip3`, `zip4`, ..., `zip7` são as equivalentes para três, quatro, etc, até sete listas
12. A função `words` quebra uma string em uma lista de palavras, delimitadas por qualquer caractere que corresponda a espaços em branco:

```
ghci> :type words
words :: String -> [String]

ghci> words "A B\tC\nD\rE"
["A","B","C","D","E"]
```

Funções infixadas

- Haskell utiliza, por padrão, a notação prefixada, de modo que, na aplicação da função `f` aos argumentos `x` e `y`, o nome da função precede os argumentos, que são separados por espaços em branco

```
z = f x y
```

- Se a função recebe dois ou mais argumentos, é possível que a notação infixada traga uma melhor compreensão e leitura
- Para utilizar a notação infixada, basta colocar o nome da função entre crases (`), tanto em uma definição quanto em uma chamada
- Ambas formas são intercambiáveis

```haskell
import Data.Bits

bitwise_or :: Int -> Int -> Int
bitwise_or a b = a .|. b

main = print (x, y) where       -- saída: (3, 7)
    x = bitwise_or 1 2
    y = 3 `bitwise_or` 5
```

Exemplos de funções infixadas

1. A função `elem` recebe um elemento `x` e uma lista de elementos `xs` e retorna verdadeiro se `x` pertence a `xs`

```
ghci> :type elem
elem :: a -> [a] -> Bool

ghci> 'x' `elem` "Teste"
False
```

2. A negação de `elem` é a função `notElem`

```
ghci> 'x' `notElem` "Teste"
True
```

3. A função `isPrefixOf` do módulo `Data.List` recebe os mesmos parâmetros, e retorna verdadeiro se `x` é prefixo de `xs`
4. As funções `isInfixOf` e `isSuffixOf` do mesmo módulo tem comportamento semelhante, retornando verdadeiro se `x` é uma sublista de `xs` ou se `x` é sufixo de `xs`, respectivamente

Funções de alta ordem

- Uma função é dita de **alta ordem** se ela recebe uma ou mais funções como parâmetro ou retorna uma função
- Por exemplo, a função `break` recebe um predicado `P` e uma lista `xs`, e retorna uma par de listas `(ys, zs)`, onde `xs = ys ++ zs` e `zs` tem início no primeiro elemento `x` de `xs` tal que a expressão '`P x`' é verdadeira

```
ghci> :type break
break :: (a -> Bool) -> [a] -> ([a], [a])

ghci> break even [1, 1, 2, 3, 5, 8]
([1, 1], [2, 3, 5, 8])
```

- A função `all` recebe um predicado `P` e uma lista `xs` e retorna verdadeiro se '`P x`' é verdadeira para todos `x` em `xs`
- A função `any` recebe os mesmos parâmetros, e retorna verdeiro se '`P x`' é verdadeira para ao menos um elemento de `xs`

Exemplos de funções de alta ordem

1. A função `takeWhile` recebe um predicado **P** e uma lista `xs` e retorna uma lista `ys` cujos elementos são todos dentre os primeiros elementos `x` de `xs` tais que '**P** x' é verdadeira
2. Sua complementar é a função `dropWhile`, que recebe os mesmo parâmetros e retorna uma lista `ys` cujo primeiro elemento é o primeiro elemento `x` de `xs` para o qual a expressão '**P** x' é falsa

```
Prelude Data.Char> takeWhile isUpper "FGAmaDF"
"FGA"

Prelude Data.Char> dropWhile isUpper "FGAmaDF"
"maDF"
```

3. A função `span` retornam um par de listas com as duas partes resultantes da chamada de `takeWhile`

```
Prelude Data.Char> span isUpper "FGAmaDF"
("FGA", "maDF")
```

Laços em Haskell

- Diferentemente das linguagens imperativas, Haskell não oferece construtos equivalentes aos laços `for` e `while`
- Para contornar este fato pode-se valer de algumas técnicas distintas
- Uma maneira é utilizar recursão
- Outra forma é utilizar funções de alta ordem e abstrações
- Esta diferença de abordagem tende a ser um fator que dificulta a aprendizagem de Haskell, e linguagens funcionais em geral, para programadores acostumados com linguagens imperativas

Exemplo de uso de recursão: Capitalização

- O primeiro exemplo de uso de recursão para substituir laços é o problema de capitalizar as palavras de uma string dada
- Por capitalizar uma palavra entende-se:
  1. tornar a primeira letra maiúscula; e
  2. transformar todas as demais em minúsculas
- Para ilustrar as diferenças entre as abordagens imperativa e funcional, será apresentado um código C++ que capitaliza strings
- Em seguida, será apresentado um código equivalente em Haskell, utilizando recursão em substituição ao laço
- Para focar apenas no processo de capitalização, o parâmetro das funções será uma lista de palavras, abstraindo-se assim o processo de tokenização

Implementação da capitalização em C++

```
vector<string> capitalize(const vector<string>& xs)
{
    vector<string> ys;

    for (auto x : xs)
    {
        auto y = x;

        if (not y.empty())
        {
            y[0] = toupper(x[0]);

            for (size_t i = 1; i < x.size(); ++i)
                y[i] = tolower(x[i]);
        }

        ys.push_back(y);
    }

    return ys;
}
```

Implementação da capitalização em Haskell

```haskell
import Data.Char

capitalize [] = []
capitalize (x:xs) = cap x : capitalize xs where
    cap [] = []
    cap (y:ys) = toUpper y : lower ys
    lower [] = []
    lower (z:zs) = toLower z : lower zs

main = putStr $ unlines $ capitalize xs where
    xs = ["abc", "XYZ", "Teste", "iPod"]
```

Exemplo de uso de recursão: Lista de Aprovados

- Um segundo exemplo de uso de recursão em substituição a laços é o de gerar uma lista de aprovados, a partir de uma lista de alunos e suas menções
- O critério de aprovação é ter nota final igual ou superior a 5 pontos
- Os alunos serão representados por uma estrutura que contém o nome do aluno e sua nota final
- Novamente será apresentado um código em C++, que utiliza laços
- O código em Haskell novamente substituirá o laço por recursão
- Atente que neste exemplo, e no anterior, a recursão é composta por um (ou mais) caso(s) base(s), e uma chamada recursiva

Implementação da lista de aprovados em C++

```cpp
struct Student
{
    string name;
    int score;
};

vector<string> aprovados(const vector<Student>& xs)
{
    vector<string> ys;

    for (auto [name, score] : xs)
        if (score >= 5)
            ys.push_back(name);

    return ys;
}
```

Implementação da lista de aprovados em Haskell

```haskell
data Student = Student String Int

aprovados [] = []
aprovados (x:xs) | score >= 5  = name : aprovados xs
                 | otherwise   = aprovados xs
                 where (Student name score) = x

main = putStr $ unlines $ aprovados xs where
    xs = [ Student "Ana" 8, Student "Beto" 3, Student "Carlos" 5,
           Student "Daniel" 4, Student "Edgar" 7 ]
```

Exemplo de uso de recursão: Produto Escalar

- O terceiro exemplo de uso de recursão em substituição aos laços é o cálculo do produto escalar entre dois vetores
- Segundo a Álgebra Linear, dados dois vetores $\vec{u}, \vec{v} \in \mathbb{R}^n$, o produto escalar entre $\vec{u}$ e $\vec{v}$ é dado por

$$\vec{u} \cdot \vec{v} = \sum_{i=1}^{n} u_i v_i$$

- Observe que a própria definição sugere o uso de um laço, representado pelo somatório
- Para utilizar a recursão, é preciso reinterpretar esta solução
- No caso base, se $\vec{u}, \vec{v} \in \mathbb{R}^0$, então $\vec{u} \cdot \vec{v} = 0$
- Se $\vec{u}, \vec{v} \in \mathbb{R}^n$, então

$$\vec{u} \cdot \vec{v} = u_1 v_1 + \vec{r} \cdot \vec{s},$$

onde $\vec{r} = (u_2, u_3, \ldots, u_n)$ e $\vec{s} = (v_2, v_3, \ldots, v_n)$

Implementação do produto interno em C++

```cpp
double dot_product(const vector<double>& xs, const vector<double>& ys)
{
    auto res = 0.0;

    for (size_t i = 0; i < xs.size(); ++i)
        res += xs[i] * ys[i];

    return res;
}
```

Implementação do produto interno em Haskell

```haskell
dot_product [] [] = 0.0
dot_product (x:xs) (y:ys) = x*y + dot_product xs ys

main = print $ dot_product xs ys where
    xs = [ 1.2, -0.8, 5.5, 3.7 ]
    ys = [ 2.8, 1.3, -4.9, 5.0 ]
```

Exemplo de uso de recursão: Verificação de primalidade

- O último exemplo de uso de recursão para substituir laços é o teste de primalidade
- Dado um inteiro positivo n, a função `is_prime(n)` deve retornar verdadeiro se n é primo, e falso, caso contrário
- A complexidade da função que será apresentada é $O(\sqrt{n})$, pois se vale do fato de que, se n é composto, ele tem ao menos um divisor próprio d tal que $d \leq \sqrt{n}$
- Contudo, para evitar erros de precisão, a função `sqrt` não é utilizada explicitamente

Implementação da verificação de primalidade em C++

```cpp
bool is_prime(int n)
{
    if (n < 2)
        return false;

    if (n == 2)
        return true;

    if (n % 2 == 0)
        return false;

    for (int d = 3; d * d <= n; d += 2)
        if (n % d == 0)
            return false;

    return true;
}
```

Implementação da verificação de primalidade em Haskell

```haskell
is_prime 2 = True
is_prime n  | n < 2          = False
            | mod n 2 == 0   = False
            | otherwise      = test n 3
            where test n d | d * d > n      = True
                           | mod n d == 0   = False
                           | otherwise      = test n (d + 2)

main = do
    n <- readLn :: IO Int
    putStrLn $ show n ++ " eh primo? " ++ show (is_prime n)
```

Mapas, filtros e reduções

- Os **mapas**, os **filtros** e as **reduções** são funções de alta ordem fundamentais em programação funcional
- Elas abstraem três conceitos fundamentais:
  1. A partir de uma lista `xs`, criar uma nova lista `ys` tal que $y_i = f(x_i)$ para uma função f dada (mapa)
  2. A partir de uma lista `xs`, criar uma nova lista `ys` formada pelos elementos `x` de `xs` que atendem a um predicado `P` (filtro)
  3. Gerar um elemento `y` a partir de uma lista `xs` através da aplicação sucessiva de uma operação binária `op` e um valor inicial `x0` (redução)
- Todas as três técnicas recebem uma função como parâmetro
- A aplicação destas técnicas substituem, em vários casos, a necessidade dos laços das linguagens imperativas

Mapas

- Em Haskell, os mapas são implementados por meio da função map

```
    ghci> :type map
    map :: (a -> b) -> [a] -> [b]
```

- Um mapa recebe uma função f que transforma um elemento do tipo a em um elemento do tipo b e uma lista de elementos do tipo a
- O retorno é uma lista de elementos do tipo b, onde $b_i = f(a_i)$
- O uso de mapas simplifica o código e o torna mais legível

```haskell
-- Capitalização utilizando mapas
import Data.Char

capitalize xs = map cap xs where
    cap [] = []
    cap (y:ys) = toUpper y : map toLower ys
```

Filtros

- Em Haskell, os filtros são implementados por meio da função `filter`:

```
ghci> :type filter
filter :: (a -> Bool) -> [a] -> [a]
```

- Um filtro recebe um predicado **P** e uma lista de elementos do tipo `[a]` e retorna uma nova lista do tipo `[a]`
- Um elemento a da lista de entrada estará na lista de saída se, e somente se, a expressão '`P a`' for verdadeira
- A ordem relativa dos elementos é preservada

```haskell
import Data.Char

main = print (filter isHexDigit s) where
    s = "Coordenadas (20A, 38F, 40X)"

-- saída: "Cdeada20A38F40"
```

Lista de aprovados usando filtros e mapas

```haskell
-- Lista de aprovados usando filtros
data Student = Student {
    studentName   :: String,
    studentScore  :: Int
}

aprovados xs = map studentName (filter f xs) where
    f x = studentScore x >= 5

main = putStr $ unlines $ aprovados xs where
    xs = [ Student "Ana" 8, Student "Beto" 3, Student "Carlos" 5,
           Student "Daniel" 4, Student "Edgar" 7 ]
```

Recursão de cauda

- Uma função é dita **recursiva de cauda** (*tail recursive*) se ela ou retorna valores (casos-base) ou retorna chamadas de si mesma, com diferentes parâmetros
- Nem toda função recursiva é recursiva de cauda
- Por exemplo, a definição da função fatorial abaixo é recursiva, mas não recursiva de cauda:

$$
n! = \begin{cases} 1, & \text{se } n = 0 \text{ ou } n = 1 \\ n \cdot (n-1)!, & \text{caso contrário} \end{cases}
$$

- Isto porque, na chamada recursiva, o retorno consiste no produto da chamada de $(n-1)!$ pelo parâmetro n

Recursão de cauda

- Contudo, esta definição pode ser modificada para que se torne recursiva de cauda:

$$f(n,m) = \begin{cases} m, & \text{se } n = 0 \text{ ou } n = 1 \\ f(n-1, n \cdot m), & \text{caso contrário} \end{cases}$$

- Desde modo, $n! = f(n, 1)$
- Observe que a chamada recursiva agora consiste apenas em uma invocação da função f
- O parâmetro m é denominado **acumulador**
- A recursão de cauda permite a **otimização de chamada de cauda** (*tail call optimization – TCO*)
- Isto porque, neste caso, é possível evitar o uso da pilha de execução, reaproveitando um único registro de ativação a cada chamada

Reduções

- Em Haskel, as reduções são implementadas por meio de **dobras** (*folds*)
- A **dobra à esquerda** (*left fold*) abstraí o seguinte padrão:
  - i. fazer algo a cada elemento da lista;
  - ii. atualizar o acumulador a cada ação; e
  - iii. retornar o acumulador ao final do processo.
- A função `foldl` pode ser definida como

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl step zero (x:xs) = foldl step (step zero x) xs
foldl _    zero []     = zero
```

- `step` é uma função que recebe dois parâmetros dos tipos `a` e `b`, respectivamente, e retorna um elemento do tipo `a`
- `zero` é o acumulador, e na chamada da função `foldl` é o termo inicial da expansão
- O terceiro parâmetro é a lista a ser processada

Dobra à esquerda

- A função `step` atualizará o acumulador, utilizando o valor acumulado e um elemento da lista
- `foldl` é uma dobra à esquerda porque consome os elementos da lista da esquerda para a direita
- Por exemplo, a função `accumulate` abaixo retorna a soma dos elementos da lista `xs`

```
accumulate xs = foldl (+) 0 xs
```

- Os parêntesis constituem uma notação para funções (ou operadores) binárias
- Outro exemplo seria a implementação da função fatorial:

```
factorial n = foldl (*) 1 [1..n]
```

Exemplo de expansão de uma dobra à esquerda

```
factorial 4 = foldl (*) 1 [1..4]
            == foldl (*) (1 * 1) [2..4]
            == foldl (*) ((1 * 1) * 2) [3..4]
            == foldl (*) (((1 * 1) * 2) * 3) [4..4]
            == foldl (*) ((((1 * 1) * 2) * 3) * 4) []
            == ((((1 * 1) * 2) * 3) * 4)
            == (((1 * 2) * 3) * 4)
            == ((2 * 3) * 4)
            == (6 * 4)
            == 24
```

Dobra à direita

- A função foldr realiza a dobra à direita
- Ela pode ser definida como

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr step zero (x:xs) = step x (foldr step zero xs)
foldr _    zero []     = zero
```

- A diferença entre as duas dobras pode ser vista através da expansão

```
factorial 3 = foldr (*) 1 [1..3]
            == 1 * foldr (*) 1 [2..3]
            == 1 * (2 * foldr (*) 1 [3..3])
            == 1 * (2 * (3 * foldr (*) 1 []))
            == 1 * (2 * (3 * 1))
            == 1 * (2 * 3)
            == 1 * 6
            == 6
```

- Esta função pode ser interpretada da seguinte maneira: troque o construtor da lista por step e a lista vazio por zero:

```
[1..3]      == 1 : (2 : (3 : []))
factorial 3 == 1 * (2 * (3 * 1))
```

Dobra à direita

- À primeira vista, a dobra à direita parece menos útil na prática do que a dobra à esquerda, uma vez que processa os elementos do último para o primeiro
- Contudo, ela pode ser utilizada para implementar a função `filter`
- Considere a implementação de `filter` abaixo, que utiliza recursão explícita:

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

- Esta implementação pode ser reescrita como

```haskell
filter p xs = foldr step [] xs
    where step x ys | p x       = x : ys
                    | otherwise = ys
```

Dobra à direita

- A assinatura da função `filter` nos diz que ela retorna uma lista do mesmo tipo que ela consome
- O caso base então ocorre com a lista vazia, que inicializará o acumulador
- A função `foldr` chamará a função `step` passando um elemento da lista e o acumulador
- Se o elemento atender ao predicado `p` ele deve ser adicionado ao acumulador, caso contrário deve ser descartado
- Funções que podem ser implementadas com `foldr` são denominadas **recursivas primitivas**

Dobra à direita

- A função map também pode ser implementada por meio da função foldr:

```haskell
map :: (a -> b) -> [a] -> [b]
map f xs = foldr step [] xs
    where step x ys = f x : ys
```

- Até mesmo foldl pode ser escrita em termos de foldr:

```haskell
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z xs = foldr step id xs z
    where step x g a = g (f a x)
```

- Esta definição não é óbvia, e envolve a aplicação parcial de funções (*currying*)
- Para compreender esta equivalência, o ideal é escrever as equivalências de uma aplicação

Exemplo de expansão de `foldl` definido como `foldr`

```haskell
    accumulate [1..3] = foldl (+) 0 [1..3]
                      == foldr step id [1..3] 0
                      == step 1 (foldr step id [2..3]) 0
                      == step 1 (step 2 (foldr step id [3..3])) 0
                      == step 1 (step 2 (step 3 (foldr step id []))) 0
                      == step 1 (step 2 (step 3 id)) 0
                      == step 2 (step 3 id) (+ 0 1)
                      == step 3 id (+ (+ 0 1) 2)
                      == id (+ (+ (+ 0 1) 2) 3)
                      == (+ (+ (+ 0 1) 2) 3)
                      == (((0 + 1) + 2) + 3)
```

Exemplo de produto escalar definido usando foldr

```
dot_product xs ys = foldr step 0.0 (zip xs ys) where
    step (x, y) z = x*y + z

main = print $ dot_product xs ys where
    xs = [ 1.2, -0.8, 5.5, 3.7 ]
    ys = [ 2.8, 1.3, -4.9, 5.0 ]
```

`foldl` e evaluação não-estrita

- A função `foldl` deve ser usada como cuidado, por causa da evaluação não-estrita
- Isto significa que, até que o caso baso seja atingido, todas as expressões intermediárias sejam armazenadas sem serem computadas
- Estas expressões ocupam mais memória do que os valores que elas representam
- Se a expressão for muito extensa, a pilha de execução pode estourar

```
ghci> foldl (+) 0 [1..10^7]
*** Exception: stack overflow
```

- Este tipo de erro é denominado *space overflow*, pois o programa consome muito mais espaço (em memória) do que deveria

Funções lambda

- Haskell permite a definição de funções anônimas, denominadas **funções lambda**
- A sintaxe para a definição de uma função lambda é

```
\var1 var2 ... varN -> expression
```

- A lista de parâmetros var1, var2, ..., varN pode conter casamentos de padrões
- A expressão, contudo, não pode conter guardas
- A depender do contexto, pode ser necessário usar parêntesis para delimitar o corpo da função lambda
- Por exemplo, a função abaixo imprime os inteiros de 1 a n, substituindo os múltiplos de m pela palavra "Pim"

```haskell
-- Abaixo um exemplo para n = 10 e m = 3
pim n m = map (\x -> if mod x m == 0 then "Pim" else show x) [1..n]

main = putStr $ unlines $ pim 10 3
```

Currying

- A aplicação parcial de uma função (*currying*, termo derivado do nome do lógico Haskell Curry) é uma técnica de programação funcional que permite obter uma nova função através da aplicação incompleta de seus parâmetros
- Na sintaxe de tipo de uma função, as setas (->) indicam a sequência de aplicações parciais
- Por exemplo, o tipo da função `take` é

```
ghci> :type take
take :: Int -> [a] -> [a]
```

- Uma leitura possível deste tipo seria: "a função `take` recebe dois parâmetros – um inteiro e uma lista de elementos do tipo `a`"
- Porém, efetivamente este tipo significa que a função `take`, ao receber como parâmetro um número inteiro, retorna uma função `f` cujo tipo é `[a] -> [a]`

Currying

- Por exemplo,

```
ghci> take3 = take 3

ghci> :type take3
take3 :: [a] -> [a]

ghci> take3 [1..10]
[1, 2, 3]
```

- De fato, em Haskell, todas as funções recebem um único parâmetro
- A cada aplicação de um parâmetro, o retorno é uma nova função, que recebe "um parâmetro a menos" que a anterior
- Outra forma de interpretar este comportamento é pensar que, se f é uma função com múltiplos parâmetros, a cada aplicação um dos parâmetros tem seu valor fixado, e os parâmetros ainda não definidos são os parâmetros da função resultante

Currying

- A aplicação parcial permite a definição de novas funções baseadas em funções preexistentes, simplificando o código e facilitando a leitura do mesmo
- Por exemplo, a função head pode ser definida em termos de uma aplicação parcial de take

```
head xs = take 1 xs
```

- De fato, o parâmetro xs pode ser omitido desta definição, resultando em

```
head = take 1
```

- Outro exemplo de função definida por meio de aplicação parcial:

```
-- retorna apenas os elementos ímpares da lista
odds = filter odd
```

Seções

- Haskell tem uma sintaxe especial para aplicação parcial de funções em notação infixada, denominada seção
- Para tal, basta envolver o operador entre parêntesis, e fornecer o operador da esquerda ou da direita

```
-- Dobra o valor de todos os elementos da lista
ghci> doubles = map (2*)

ghci> doubles [1..5]
[2, 4, 6, 8, 10]

ghci> squares = map (^2)

ghci> squares [1..5]
[1, 4, 9, 16, 25]

ghci> hasx = ('x' `elem`)
ghci> hasX = ('X' `elem`)
ghci> anyx xs = hasx xs || hasX xs
```

Padrão ‘como’

- Ao escrever funções que recebem listas como parâmetros, é comum checar dois padrões: a lista vazia (`[]`) e a lista com ao menos um elemento (padrão `(x:xs)`)
- Este segundo padrão desconstrói a lista, de modo que se a lista toda for necessária na expressão que se segue, ela precisa se reconstruída
- O padrão ‘**como**’ (*as-pattern*) permite checar este segundo caso, preservando a lista original, de modo que ela pode ser utilizada na expressão diretamente, evitando novas reconstruções
- Por exemplo, a função abaixo lista todos os sufixos não nulos de uma string dada:

```haskell
suffixes :: String -> [String]
suffixes xs@(_:xs') = xs : suffixes xs'
suffixes [] = []

ghic> suffixes "Teste"
ghic> ["Teste", "este", "ste", "te", "e"]
```

Composição de funções

- Algumas funções podem ser implementadas através de uma cadeia de chamadas de funções, onde o resultado da aplicação da função anterior ao parâmetro da chamada é o parâmetro de entrada da próxima função da cadeia
- Por exemplo, a função `tails` do módulo `Data.List` tem comportamento quase idêntico ao da função `suffixes` implementada anteriormente:

```
ghci> :module Data.List

ghci> tails "Teste"
ghic> ["Teste", "este", "ste", "te", "e", ""]
```

- Assim, a função `suffixes` pode ser implementada utilizando-se as funções `tails` e `init`:

```
suffixes xs = init (tails xs)
```

- Este padrão corresponde à composição de funções em matemática

Composição de funções

- Em Haskell, funções podem ser compostas por meio do operador ponto final (`.`)
- De fato, ele é um operador como os demais operadores da linguagens, associativo à esquerda, como nível 9 de precedência:

```
ghci> :info (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
infixr 9 .
```

- Assim, a função `suffixes` pode ser implementada como

```
suffixes = init . tails
```

- A função abaixo contabiliza o número de palavras de uma string que começam em maiúscula:

```
ghci> :module Data.Char
ghci> capCount = length . filter (isUpper . head) . words

ghci> capCount "Paradigmas de Programação"
2
```

Referências

1. **SHALOM**, Elad. *A Review of Programming Paradigms Througout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
2. **SULLIVAN**, Bryan O.; **GOERZEN**, John; **STEWART**, Don. *Real World Haskell*, O'Relly.