Programação Funcional

(COMP0393)

Leila M. A. Silva

dcomp
departamento de computação

Generalização
Funções como Resultado
(COMP0393)

Aula 12

dcomp
departamento de computação

Funções de Alta Ordem

- Em Haskell, funções são dados e podem ser tratadas como dados de qualquer outro tipo.
- Já vimos que funções podem ser passadas como argumentos
  - Mecanismo poderoso para definir padrões de computo
- Agora veremos que funções podem ser devolvidas como resultado de uma função
  - Funções como valores (computados por outras funções)
- Assim, funções podem ser argumentos e resultados de outras funções, chamadas de *funções de alta ordem*.

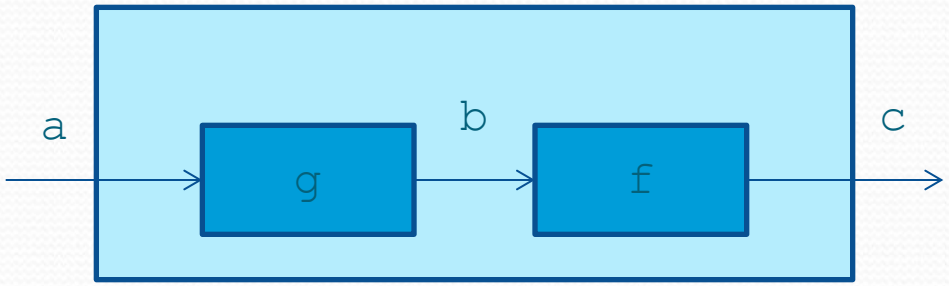
Funções de Alta Ordem

- Em Haskell funções podem ser combinadas usando operadores assim como números, por exemplo.
- É possível definir funções diretamente por expressões usando *abstrações lambda*.
- Haskell permite aplicação parcial de funções e operadores. Permite também funções *currificadas*.

Composição de Funções

- Funções podem ser compostas usando o operador ‘ . ’

```
(f.g) x = f (g x)
```



- Todos os pares de funções podem ser compostos, desde que os tipos sejam consistentes.

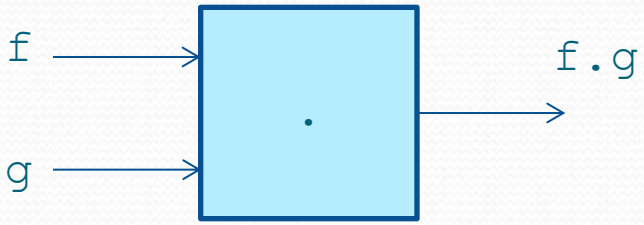
```
rotacione :: Figura -> Figura
rotacione xss = refleteV (refleteH xss)

rotacione :: Figura -> Figura
rotacione xss = (refleteV . refleteH) xss
```

Composição de Funções

- O resultado do operador ‘ . ’ é uma função!!!

```
(f.g) x = f (g x)
```



- Tipo genérico do operador ‘ . ‘

```
(.) :: (b->c) -> (a->b) -> (a->c)
```

tipo de `f` | tipo de `g` | tipo de `f.g`

- Composição é associativa: `f.(g.h) = (f.g).h`

Exemplo

- Dada uma string, somar a posição das letras maiúsculas desta string.
  - Função para indexar a string;
  - Função para coletar as maiúsculas com suas posições;
  - Função para coletar as posições das maiúsculas;
  - Função para somar as posições das maiúsculas - `sum`.

```
indexa :: String -> [(Char, Int)]
indexa str = zip str [1..]

filtrarMaiusculas :: [(Char, Int)] -> [(Char,Int)]
filtrarMaiusculas ps = filter maiusc ps
   where maiusc (c, p) = isUpper c

segundos :: [(Char, Int)] -> [Int]
segundos ps = map snd ps

somaPosMaiusc :: [Int] -> Int
somaPosMaiusc str = sum (segundos (filtrarMaiusculas (indexa str)))
```

Exemplo

- Dada uma string, somar a posição das letras maiúsculas desta string.
  - Função para indexar a string;
  - Função para coletar as maiúsculas com suas posições;
  - Função para coletar as posições das maiúsculas;
  - Função para somar as posições das maiúsculas - `sum`.

```
filtrarMaiusculas :: [(Char, Int)] -> [(Char,Int)]
filtrarMaiusculas ps = filter maiusc ps
   where maiusc (c, p) = isUpper c

filtrarMaiusculas ps = filter maiusc ps
   where maiusc = isUpper . fst

somaPosMaiusc :: String -> Int
somaPosMaiusc str = sum (segundos (filtrarMaiusculas (indexa str)))

somaPosMaiusc :: [Int] -> Int
somaPosMaiusc str = (sum . segundos . filtrarMaiusculas) (indexa str)
```

point free – alta ordem não tem argumento

Mais exemplos

- Função para pular caracteres de uma string que não sejam dígitos

```
pulaNaoDigito :: String -> String
pulaNaoDigito xs = dropWhile (not . isDigit) xs

pulaNaoDigito "rua A no. 54A" ↝ "54A"
```

- Função que recebe uma função de argumento e a aplica duas vezes

```
twice :: (a->a) -> (a->a)
twice  f = f . f

(twice succ) 5 ↝ (succ . succ) 5 ↝ succ (succ 5) ↝ succ 6 ↝ 7
```

CUIDADO!!! Não escrevam `f.g x` pensando ser `(f.g) x`. Lembre-se que a aplicação da função tem precedência. Será interpretado como `f.(g x)`, resultando numa mensagem de erro!

Operador >.>

- A ordem da composição de funções pode gerar alguma confusão, porque em `(f . g) x` primeiro aplica-se a função `g` sobre `x` e depois aplica-se `f` sobre o resultado de `(g x)`.
- Operador `>.>`
  - Mesmo que o operador ‘ . ‘ só que a ordem das funções combinadas aparece invertida na escrita
  - `g >.> f = f . g`
- Tipo genérico do operador ‘ . ‘

```
(>.>) :: (a->b) -> (b->c) -> (a->c)
```

```
rotacione :: Figura -> Figura
rotacione xss = (refleteV . refleteH) xss
```

```
rotacione :: Figura -> Figura
rotacione xss = (refleteH >.> refleteV) xss
```

Operador de aplicação: $

- Em Haskell a notação para a aplicação é por justaposição. Por exemplo, `f x`.
- Mas podemos definir uma notação explícita: `f $ x`
- Para que serve?
  - Uma alternativa para evitar parênteses
  - Por exemplo, ao invés de escrevermos

```
sum ( segundos ( filtrarMaiusculas (zip str [1..])))
```

  podemos escrever

```
sum $ segundos $ filtrarMaiusculas $ zip str [1..]
```

  - O operador $ é associativo à direita
  - Podemos precisar usar a aplicação como uma função. Por exemplo:

```
zipWith ($) [sum, product] [[1, 2], [3, 4]] ↝
    [sum $ [1,2], product $ [3,4]] ↝ [3,12]
```

Exercícios Recomendados

- Defina a generalização de twice

  `iter :: Int -> (a -> a) -> (a -> a)` tal que `iter n f` é a composição de `f` com `f`, `n` vezes.
- Usando `iter` e `dobro`, defina a função `pot2 :: Int -> Int` para calcular 2^n .
- Dada uma lista de funções de mesmo tipo, defina a função `composeList` que compõe uma lista de funções em uma única função composta. Qual o efeito de sua função numa lista vazia de funções?
- Qual o tipo genérico do operador `$`?
- Usando o operador (.) e (>.>) defina as funções:
  - Dado uma string pular espaços em branco da string
  - Dado uma string pegar a primeira palavra da string
- Qual o resultado de

```
zipWith ($) [dobroLista, pares] [[1, 2], [3, 4, 8]]
```

Expressões Lambda

- Haskell permite definir funções sem nome, introduzindo expressões. Ex:

```
adUmTodos xs = map adUm xs
    where adUm n = n + 1

adUmTodos xs = map (\n -> n+1) xs
```

- A expressão `\n -> n+1` é uma expressão lambda. Representa a uma função com argumento `n` que retorna `n+1`.
- O símbolo \ é lido como "lambda".
- Expressões lambda vêm do cálculo lambda, uma teoria de funções criada por Haskell B. Curry

Mais exemplos

```
dodroLista :: [Int] -> [Int]
dobroLista xs = map dobro xs
   where dobro x = 2*x

dodroLista :: [Int] -> [Int]
dobroLista xs = map (\x->2*x) xs

length xs = foldr g 0 xs
    where g :: t -> Int -> Int
          g m n = n + 1

length xs = foldr (\m n -> n+1) 0 xs
```

expressões lambda poodem ter mais de um argumento

Mais exemplos

```
length xs = foldr (\_ n -> n+1) 0 xs

segundos ps = map (\(_,y) -> y) ps

mapFuns :: [a -> b] -> a -> [b]
mapFuns fs x = map apliqueAx fs
  where apliqueAx f = f x

mapFuns fs x = map (\f -> f x) fs

f x y z = resultado
f = \x y z -> resultado
```

podemos usar padrões como argumentos de uma expressão lambda

expressão lambda podem ajudar na legibilidade

Definições equivalentes

Exercícios Recomendados

- Usando expressões lambda, e as funções `not` e `elem` descreva uma função do tipo `Char -> Bool`, que resulta `True` somente se o caracter não for branco, `\n` ou `\t`.
- Sem usar recursão e usando funções pré-definidas, defina a função

```
total :: (Integer -> Integer) -> (Integer-> Integer)
```

de tal forma que total `f` é uma função que para o valor `n` dá o total resultante de `f 0 + f 1 + ...+ f n`

- Dê a definição de uma função que reverte a ordem em que os argumentos são passados na função original

```
invOrdemArg  :: (a -> b -> c) -> (b -> a -> c)
```

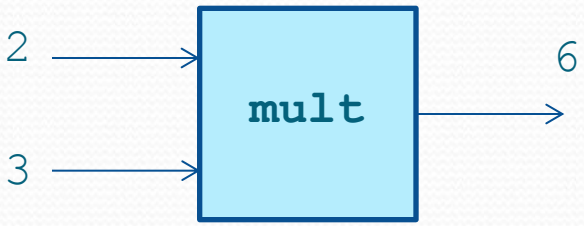
Aplicação Parcial de Funções

- Haskell permite definir aplicar parcialmente funções.Ex:

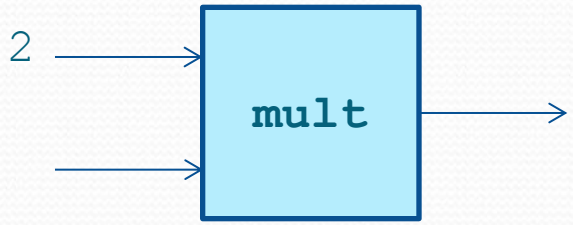
```
mult :: Int -> Int -> Int
mult x y = x*y

mult 2
```

- A função `mult` tem dois argumentos. Haskell permite fixar um deles e construir a função `mult 2`, que ao receber um parâmetro `n`, retorna `2*n`.



uma instância de `mult`



uma função resultante da aplicação parcial de `mult`

Aplicação Parcial de Funções

- Alguns exemplos:

```
mult :: Int -> Int -> Int
mult x y = x*y

dobroLista xs = map (mult 2) xs
dobroLista = map (mult 2)
dobro = mult 2
triplo = mult 3
```

- Só é possível aplicação parcial no primeiro argumento. Não podemos aplicar parcialmente só no segundo argumento, pulando o primeiro.

Aplicação Parcial de Funções

- Se a função tem vários argumentos, podemos aplicar parcialmente só com o primeiro argumento, ou só com o *i-1* primeiros argumentos. Ex:

```
maxTres :: Int -> Int -> Int -> Int
maxTres m n p = max m (max n p)
```

São expressões válidas:

```
maxTres
maxTres 5
maxTres 5 12
maxTres 5 12 8
```

Aplicação Parcial de Operadores: Seções

- Operadores em Haskell também podem ser parcialmente aplicados, pois são funções, e a isto dá-se o nome de *seções*. Ex:

```
(+ 2) -- função que soma 2
(2 +) -- idem
(> 2) -- função que verifica se um número é maior que 2
(2 >) -- função que verifica se um número é menor que 2
(3:)  -- função que acrescenta 3 na cabeça da lista
(++ "\n") -- função que adiciona uma nova linha ao final de
             uma string
```

Regra geral: O operador irá colocar o argumento do lado que está faltando para completar a aplicação.

```
(op x) y = y op x
(x op) y = x op y
```

Aplicação Parcial de Operadores: Seções

- Mais exemplos:

```
filter (> 0) . map (+ 1)

dobroLista :: [Int] -> [Int]
dobroLista = map (* 2)

ehPar :: Int -> Bool
ehPar = (== 0) . (`mod` 2)

ehImpar :: Int -> Bool
ehImpar = not . ehPar
```

Funções *curried* e *uncurried*

- Em Haskell podemos modelar funções de mais de um argumento de formas variadas.

```
mult :: Int -> Int -> Int
mult x y = x*y
```

curried

```
mult :: (Int,Int) -> Int
mult (x,y) = x*y
```

uncurried

- A forma *curried* permite a aplicação parcial e torna a escrita mais elegante.

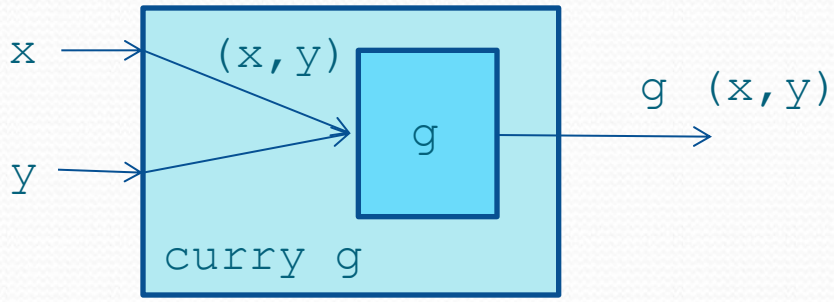
Funções *curried* e *uncurried*

- É possível transformar *curried* para *uncurried* e vice-versa

```
curry ::  ((a,b) -> c) -> (a -> b -> c)
curry g x y = g (x,y)
```

ou

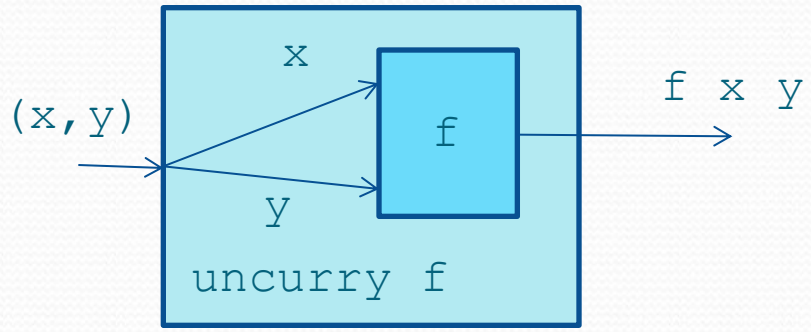
```
curry g = \x y -> g (x,y)
```




```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x, y) = f x y
```

ou

```
uncurry f  = \(x, y) -> f x y
```



Exercícios Recomendados

- Usando seções e combinadores elabore funções para:
  - Dada uma frase pegar a primeira palavra da frase
  - Somar os elementos ímpares, menores que 10.000, do quadrado dos números no intervalo [1..100].
- Refaça o quickSort para usar seções e combinadores.
- Reimplemente o exercício da biblioteca dado na Aula 4 (slide 36) , para usar combinadores, aplicação parcial e expressões lambda.
- Suponha agora que `type Database = [(Person, [Book])]`. Refaça todos os itens do exercício da biblioteca usando combinadores, aplicação parcial e expressões lambda.
- Refaça o exemplo da figura apresentado na Aula 9, usando funções de alta ordem.

Um exemplo mais robusto

Dado um texto queremos criar um índice.

Por exemplo, dado a string

"macaco e gato amarelo\ngato preto gato bravo\ngato amarelo"

queremos criar o índice remissivo

amarelo	1, 3
bravo	2
gato	1, 2, 3
macaco	1
preto	2

Palavras de tamanho menor do que 4 serão desconsideradas

Um exemplo mais robusto

```
type Doc = String
type Linha = String
type Palavra = String
criaIndice :: Doc → [([Int], Palavra)]
```

Projeto orientado pelos dados (foco nos dados intermediários produzidos)

1. Dividir o documento em linhas ⇨ `[Linha]`
2. Acrescentar à linha o seu número ⇨ `[(Int,Linha)]`
3. Dividir as linhas em palavras associadas com o número da linha onde ocorrem ⇨ `[(Int,Palavra)]`
4. Ordenar alfabeticamente sem repetir nenhum par na lista ⇨ `[(Int,Palavra)]`
5. Transformar o número de linha numa lista contendo este número ⇨ `[([Int],Palavra)]`
6. Combinar em uma lista do tipo `[([Int],Palavra)]`
7. Cortar os pares com palavras de tamanho menor que 4

Um exemplo mais robusto

Funções são necessárias para cada ação

- Dividir o documento em linhas ⇨ `linhas:: Doc -> [Linha]`
- Acrescentar à linha o seu número ⇨

  `numLinhas:: [Linha] → [(Int,Linha)]`
- Dividir as linhas em palavras associadas com o número da linha onde ocorrem ⇨ `linhaPalavras :: [(Int,Linha)] → [(Int,Palavra)]`
- Ordenar alfabeticamente as palavras sem repetir nenhum par na lista ⇨

  `ordPal :: [(Int,Palavra)] → [(Int,Palavra)]`
- Transformar o número de linha numa lista contendo este número

  `transfNum :: [(Int,Palavra)] -> [([Int],Palavra)]`
- Combinar em uma lista do tipo `[([Int],Palavra)]`

  `combinaLinPal :: [([Int],Palavra)] → [([Int],Palavra)]`
- Cortar os pares com palavras de tamanho menor que 4

`cortaPal :: [([Int],Palavra)] → [([Int],Palavra)]`

Um exemplo mais robusto

```
criaIndice :: Doc → [ ( [Int], Palavra ) ]
criaIndice =
     linhas         >.>
     numLinhas      >.>
     linhaPalavras  >.>
     ordPal         >.>
     transfNum      >.>
     combinaLinPal  >.>
     cortaPal
```

```
criaIndice :: Doc → [ ( [Int], Palavra ) ]
criaIndice =
     cortaPal   .   combinaLinPal   .
     transfNum . ordPal. linhaPalavras   .
     numLinhas .   linhas
```

Um exemplo mais robusto

Função `linhas` (definida no Prelude `lines`)

```
numLinhas :: [Linha] → [(Int,Linha)]
numLinhas lins = zip [1..] lins
```

```
linhaPalavras :: (Int,Linha) → [(Int,Palavra)]
linhaPalavras = concat . map numPals
  where
      numPals :: (Int, Linha) -> [(Int, Palavra)]
      numPals (n, lin) =  map (\p -> (n, p)) (words lin)
      -- numPals (n, lin) =  [ (n, p) | p <- words lin ]
          -- words está definido no Prelude
```

Um exemplo mais robusto

Função `ordPal` (usa `sortOn` definida no Data.List)

```
ordPal :: [(Int, Palavra)] -> [(Int, Palavra)]
ordPal = sortOn snd
```

```
transfNum :: [(Int, Palavra)] -> [([Int], Palavra)]
transfNum = map tlis
   where tlis (n,p) = ([n],p)
```

```
combinaLinPal :: [([Int], Palavra)] -> [([Int], Palavra)]
combinaLinPal [] =  []
combinaLinPal [np] = [np]
combinaLinPal ((l1, p1) : (l2, p2) : lps) =
  | p1 /= p2    = (l1, p1) : combinalinPal ((l2,p2): lps)
  | otherwise   = combinaLinPal ((l1++l2, p1) : lps)
```

Um exemplo mais robusto

```
cortaPal :: [([Int], Palavra)] -> [([Int], Palavra)]
cortaPal = filter ((>3) . length . snd )
```

Exercício para o Laboratório. Implemente as funções e teste para:

```
"Maria Madalena\nMaria Joana Teresa\nMadalena Maria
Teresa\nJoão "
```

Exercícios Recomendados

- Adapte o quicksort para que ele ordene de acordo com os elementos das duplas, seguindo o critério dado pela função abaixo:

```
orderPair :: (Int, String) → (Int, String) → Bool
orderPair (n1,w1) (n2,w2) =
        w1 < w2  || ( (w1==w2 && n1 < n2)
```

- Defina a função `linhas` usando `takeWhile` e `dropWhile`.
- Use expressões lambda para definir `transfNum`.
- Elabore uma função que modifique a saída da lista de linhas de uma palavra como abaixo:

gato 1,2,3,7,8,9,15  gato 1-3, 7-9, 15

- Modifique a função `criaIndice` para retornar ,para cada palavra, o total de vezes que a palavra ocorre no texto e não as linhas em que ocorre.