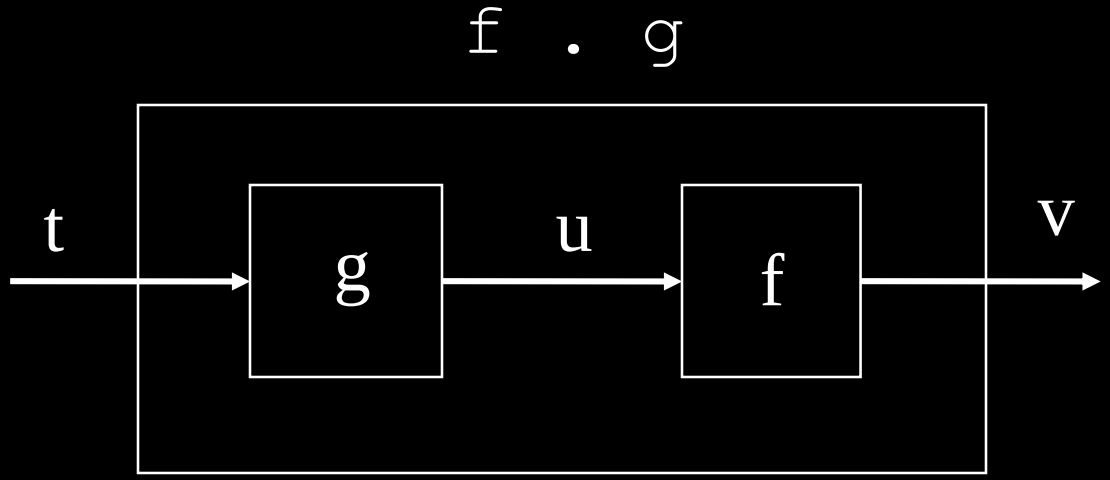
Funções como Valores

André Santos

(a partir de *slides* elaborados por André Santos, Sérgio Soares, Fernando Castor e Márcio Lopes)

A função de composição

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



A função de composição

```
(.) :: (u -> v) -> (t -> u) ->
       (t -> v)
(.) f g x = f (g x)
==
(.) f g = \x->f (g x)
:type (.)
(.)::(b->c)->(a->b)->a->c
```

Composição de funções

```
(>.>) :: (a -> b) -> (b -> c) ->
         (a -> c)
g >.> f = f . g
(g >.> f) x = (f.g) x = f (g x)
```

Funções como valores e resultados

- ```
  twice :: (t -> t) -> (t -> t)
  twice f = f . f
  ```
- ```
  (twice succ) 12
  = (succ . succ) 12
  = succ (succ 12)
  = 14
  ```

Funções como valores e resultados

```
iter :: Int -> (t -> t) -> (t -> t)
iter 0 f = id
iter n f = (iter (n-1) f).f

(iter 10 double) 3
```

Expressões que definem funções

```
addOne x = x +1
map addOne [1,2,3]
```

- Notação Lambda

```
\x -> x + 1
map (\x -> x + 1)[1,2,3]
```

Expressões que definem funções

- ```
  addNum :: Int -> (Int -> Int)
  addNum n = h
     where
     h m = n + m
  ```
- Notação Lambda

```
\m -> 3+m
addNum n = (\m -> n+m)
```
```

Exercício

- Dada uma função `f` do tipo `t -> u -> v`, defina uma expressão da forma

  `(\... -> ...)`

  para uma função do tipo `u -> t -> v` que se comporta como `f` mas recebe seus argumentos na ordem inversa

Aplicações parciais

- ```
  multiply :: Int -> Int -> Int
  multiply a b = a*b
  ```
- ```
  doubleList :: [Int] -> [Int]
  doubleList = map (multiply 2)
  ```
- ```
  (multiply 2) :: Int -> Int
  ```
- ```
  map (multiply 2) :: [Int] -> [Int]
  ```

Aplicações parciais

- `whiteSpace = " "`
- `elem :: Char -> [Char] -> Bool`
- `elem ch whiteSpace`
- `\ch -> elem ch whiteSpace`
- `filter (\ch -> not(elem ch whitespace))`

Qual o tipo da função acima?

associatividade

- `f a b = (f a) b`
- `f a b ≠ f (a b)`
- `t -> u -> v = t -> (u -> v)`
- `t -> u -> v ≠ (t -> u) -> v`

```
g :: (Int -> Int) -> Int
g h = h 0 + h 1
```

associatividade

- `f a b = (f a) b`
- `f a b ≠ f (a b)`
- `t -> u -> v = t -> (u -> v)`
- `t -> u -> v ≠ (t -> u) -> v`

```
g :: (Int -> Int) -> Int
g h = h 0 + h 1
```

Aplicação de ->

- Aplicação de função é associativo à esquerda
  - `f x y = (f x) y`
- Símbolo de função ‘->’ é associativo à direita
- `a -> b -> c` significa

`a -> (b -> c)`

A função de composição (revisitada)

```
(.) :: (u -> v) -> (t -> u) ->
       (t -> v)
(.) f g x = f (g x)
==
(.) f g = \x->f (g x)
:type (.)
(.)::(b->c)->(a->b)->a->c
```

Revisitando `iter`

```
iter 10 double 3 ==
iter 10 ((*) 2) 3 ==
iter 10 (2 (*)) 3 ==
iter 10 (2 *) 3 ==
iter 10 (* 2) 3 ?

iter 10 (/ 2) 2000 ==
iter 10 ((/) 2) 2000 ?
```

Quantos argumentos uma função tem?

- `multiply :: Int -> Int -> Int`
- `multiply :: Int -> (Int -> Int)`
- `multiply 4`
- `(multiply 4) 5`

Mais exemplos …

- `(+2)`
- `(>2)`
- `(3:)`
- `(++ "\n")`
- `filter (>0).map (+1)`
- `double = map (*2)`

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

Mais exemplos …

- `dificil = map.filter`
- `maisdificil = map.foldr`
- `maisainda = foldr.map`