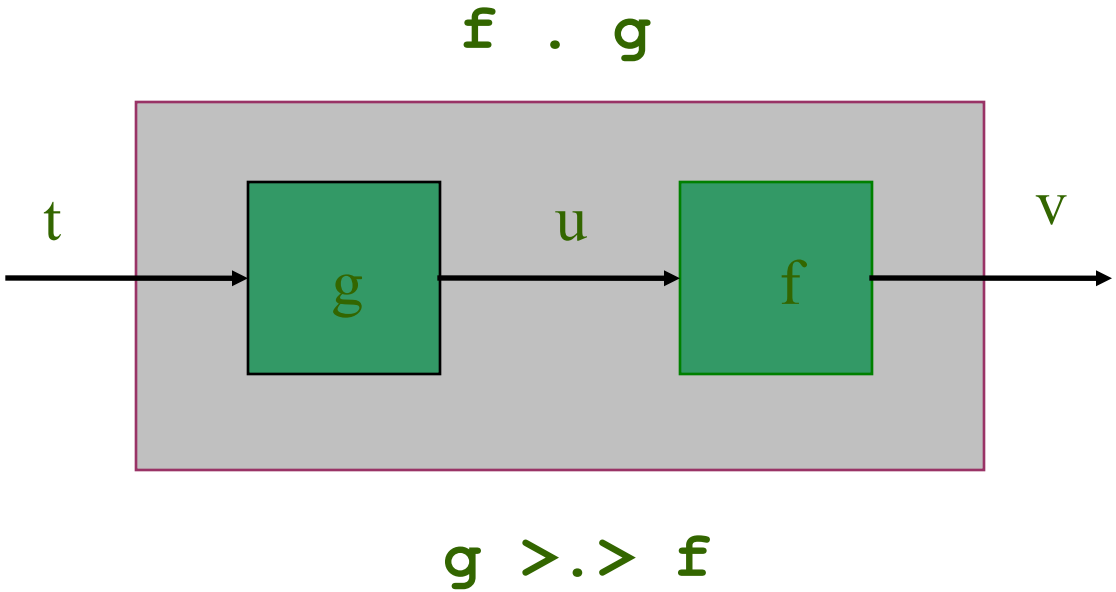
Programação Funcional
Funções como valores

Vander Alves

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)
  ```

- a composição é associativa
  ```

composição para a frente

- `(>.>) :: (t -> u) -> (u -> v) -> (t -> v)`
- `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
  ```

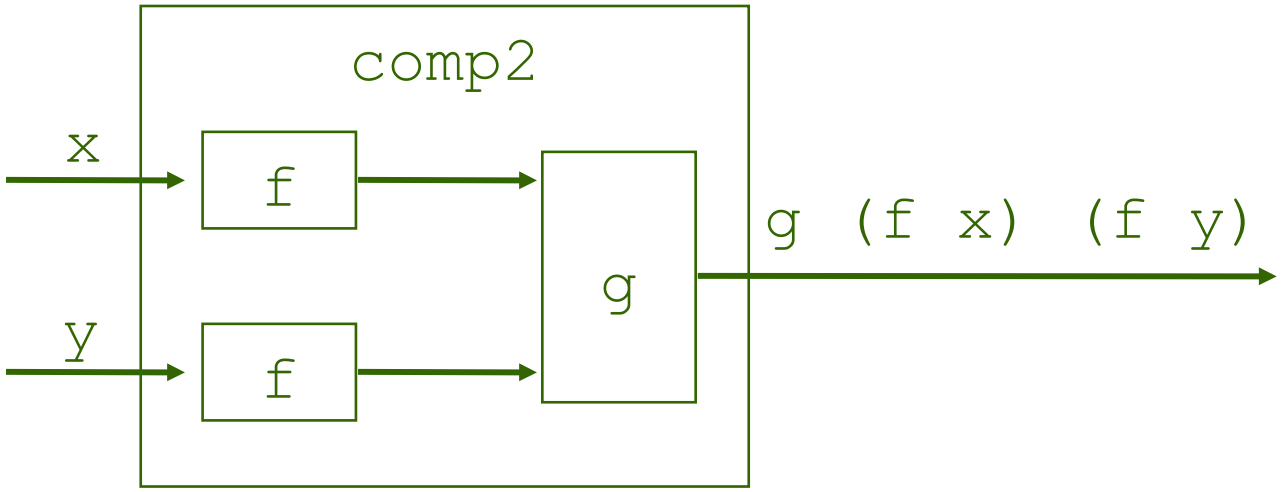
Funções como valores e resultados

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

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)
```
```




```
\x y -> g (f x) (f y)
```

```
comp2 :: (t -> u) -> (u -> u -> v) ->
   (t -> t -> v)
```

```
comp2 f g = (\x y -> g (f x) (f y))
```

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))`

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
    ```
    ```

Quantos argumentos uma função tem?

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

Seções

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

seções

```
getEvens
    = filter ((==0).(`mod` 2))

books db per
    = map snd (filter ((==per).fst) db)
```

Exercícios

- Use aplicação parcial para definir a função `addNum`

currying

- ```
  curry  :: ((t,u) -> v) -> (t -> u -> v)
  curry g a b = g (a,b)
  ```
- ```
  uncurry :: (t -> u -> v) -> ((t,u) -> v)
  uncurry f (a,b) = f a b
  ```
- ```
  flip :: (t -> u -> v) -> (u -> t -> v)
  flip f b a = f a b
  ```
  ```