Características da programação funcional

- Funções puras
- Estado imutável
- Recursão
- Transparência referencial
- Laziness
- Funções de primeira classe
- Funções de alta ordem
- Sistemas de tipos

Função pura

- Função no sentido matemático -> Para cada valor do domínio tem um único valor na imagem.
- Mesmo chamando a função várias vezes, o resultado não é alterado.
- Não tem efeito colateral.
- Sempre tem que retornar alguma coisa, mesmo que seja (return None).

Transparência referencial

- Pode substituir a definição em qualquer lugar que ela apareça.

```
x = F(2)
F(k) = k + 3
x + x
F(2) + F(2)
(2 + 3) + (2 + 3)
```

```
Z = G(x)
(...)
if Z == G(x):
```

O valor pode ter mudado depois de várias linhas (...) Com o conceito de Transparência referencial é garantido que Z = G(x) em qualquer ponto e pode ser substituído.

Preguiça

- Nem todas as linguagens funcionais têm essa definição.
- x = [1.100000000]

No momento nenhuma memória foi gasta, pois só foi definido X.

y = x * 2
z = y + 1
print(x)

O cálculo de y e z não vão ser feitos, pois ele só precisa de x para fazer o print

w = take(z, 10)

Só define w, não faz mais nada.

Se colocar print(w), mesmo que z seja um vetor de tamanho infinito ele só vai calcular os 10 primeiros valores de z, pois é o que é necessário para ter w.

Função primeira classe

- Tratar função como objeto de primeira classe (int, float, char).
    - Ou seja, você pode tratar uma função como uma variável, logo uma variável pode receber uma função, por exemplo. f = g(x)

```
def f(x):
  return x + 1

g = f

print(g(5))
```

```
def f(x):
  return x + 1

def h(x):
  return x - 1

g = [f, h]

print(g[1](5))
```

Função de alta ordem

- Funções que recebem como parâmetro outra função.

Classe de tipo

- Capacidade que os tipos dos elementos possuem
- O tipo Num representa todos tipos que é possível fazer operações aritméticas
- Num, Ord, Eq

Construtores

- Construi tipos
- Lista de int = [int]
    - [3,5] :: (Num a) => [a]
        - (Num a) específica que a é do tipo Num, logo depois disso vem =>, tudo que está antes dessa seta e depois do :: é uma contextualização do tipo, como contextualizar o que é a. Depois da seta vem [a], para indicar que [3, 5] é uma lista de a
- g :: Int -> Integer
    - Uma função que recebe um Int e retorna um Integer
- ->
    - Significa agrupamento a direita
    - Uma sequencia de tipos se agrupa de direita para a esquerda
    - int -> int -> [int]
        - int -> (int -> [int])
            - Uma função que recebe um int e retorna uma outra função que também recebe um int e retorna uma lista de inteiros
    - (int -> int) -> [int]

        - Significa que você tem uma função que recebe uma função do tipo (int -> int) e retorna uma lista de inteiros

```
applyToList :: (Int -> Int) -> [Int]
applyToList f = [f 1, f 2, f 3, f 4, f 5]

double :: Int -> Int
double x = x * 2

result :: [Int]
result = applyToList double
```

O resultado vai ser [2, 4, 6, 8, 10]

    - int -> int -> int -> [int]
        - Traduz para int ->(int -> (int -> [int]))
        - Significa que é uma função que chama uma outra função que recebe um inteiro que chama uma outra função que recebe um inteiro e que retorna uma lista de inteiros

```
f :: Num a => a -> a -> a
f x y = x + y
```

f 3 5 :: (Num a) => a #é uma expressão que resulta em um valor numérico
f 3 :: (Num a) => a -> a #uma função que é a aplicação parcial da função f, logo, isso cria uma nova função que depende de uma mais um valor de a, para que se possa aplicar a função f de vez.

```
h :: (Num a) => a -> a
h = f 1
```

h 6 é a mesma coisa de f 1 6

```
m :: (Num a) => a-> b -> a
m x y = x +3
```

Nesse caso o tipo do b (y) é independente, pois ele não é usado na expressão

```
p x y = 42
```

p :: (Num c) => a -> b -> c #for mais genética
p :: (Num a, Num c) => a-> b -> c #restringe a e c (eles tem que ser num, não necessariamente do mesmo tipo)
p :: (Num a) => a -> b ->a, faria com que x fosse do mesmo tipo de 42

- Eq significa que aquele valor pode ser comparado
    - r :: (Eq a) => a -> a -> Bool
    - r x y = x == y
- Ord significa que pode ser ordenado
    - r :: (Num a, Ord a) => a -> a -> Bool
    - r x y = x + 5 > y
    -
    - s :: (Num a, Ord a) => a -> Bool

    - s x = x > 5

Ord e Eq são necessários, pois como estou declarado eles como uma classe de tipo é necessário dizer se eles podem ser comparados ou não. Se eu fizesse s :: Int -> Int -> Bool aí não seria necessário já que Int por si só pode ser comparado.

- :
    - Operador que junta a cabeça com a calda e retorna uma lista
    - (:) :: a -> [a] -> [a]
        - Rece u, a e uma lista de [a] e retorna uma lista que irá conter a e a lista de a passada

<u>Map</u>
mapa [] = [] #caso base
mapa (x:xs) = (2*x):(mapa xs) #caso recursivo

Ele vai aplicar 2*x, sendo x a cabeça e concatenar com o resultado de mapa xs que é aplicar mapa na calda, logo isso vai aplicar 2*x em todo o vetor.

mapa :: (Num a) => [a] -> [a] #não é a forma mais genérica, pois faz com que o resultado tenha que ser do mesmo tipo do input

map :: (b -> c) -> [b] -> [c] #mais genérico
map f (x:xs) = (f x) : (map f xs)
A declaração do map mostra que map vai receber uma função do tipo (b -> c), uma lista de b e vai retornar uma lista de c

L = vetor
mult = 2x
mapa = map mult l

mapa = map (\x -> 2*x) #uso de lambda e uso parcial da função map
mapa L #termina de aplicar a função

<u>Pureza</u>
Função pura pode usar código puro
Função impura pode usar código puro
Função pura <u>NÃO</u> pode usar código impuro
Função impura pode usar código impuro

“do” é uma palavra reservada e só pode ser utilizada em funções impuras
A main, por definição é uma função impura

IO indica que é impuro (input e output) e () indica que o retorno da função deve ser nulo, lembrando que toda função em haskell deve retornar algo, mesmo que nulo.

```haskell
main :: IO ()

main = do
    let x = 42 {- Tem que sinalizar que é impuro -}
    let str = show x {- Show -> Converte um tipo em String -}
    putStrLn str
```

let

- Dentro de uma função do
    - Dentro de uma função do o let é utilizado para declarar variáveis locais que poderão posteriormente ser utilizadas na função. Tudo que contém um let dentro de uma função do é puro.
- Dentro de uma função pura
    - O let dentro de uma função pura também é utilizado para declarar variáveis localmente, porém ele deve ser acompanhado da expressão in para dizer onde ele será utilizado

```haskell
f :: Int -> Int
f x =
    let y = 2
    in x + y
```

Entrada e saída

```haskell
main = do
    let x = 42 {- Tem que sinalizar que é impuro -}
    let y = f x 3
    let str = show y {- Show -> Converte um tipo em String -}
    putStrLn str

f :: (Num a) => a -> a -> a
f x y = x + y
```

Neste caso é necessário usar o show para poder converter um tipo à uma string para posteriormente poder fazer o print.

```haskell
{- Ler dois númreros do teclado e imprimir a soma deles -}
main :: IO ()
main = do
    l1 <- getLine
    l2 <- getLine
    let x :: Integer
        x = read l1
    let y :: Integer
        y = read l2
    let z = add x y
    let str = show z
    putStrLn str

add :: (Num a) => a -> a -> a
add x y = x + y
```

Agora, ao ler algo do usuário é necessário usar o read, que transforma de string para o tipo declarado da variável.

Haskell é uma linguagem bidimensional, ou seja, tudo tem que ser alinhado.

O let permite que defina várias variáveis em sequência:

```haskell
let v :: Integer
    v = read l {
    x = v
    y = f x 3
```

A classe show são os tipos que podem ser transformados em string.

Guarda
| é chamado de guarda

```haskell
sinal :: Integer -> Integer
sinal n
    | n < 0 = -1
    | n == 0 = 0
    | otherwise = 1
```

é equivalente a

```haskell
sinal2 x =  if x < 0
            then -1
            else if x == 0
                 then 0
                 else 1
```

```haskell
tempo :: Integer -> String
tempo t
    | t < 0 = "Congelante"
    | t < 15 = "Ok"
    | t < 25 = "Agradável"
    | t < 35 = "Quente"
    | t < 45 = "Derretendo"
    | otherwise = "Já era"
```

A ordem na qual se passa nas guardas é sequencial, portanto é importante definir a ordem certa.

```haskell
bmi :: Float -> Float -> String
bmi h w
    | w / (h*h) < 20 = "Abaixo"
    | w / (h*h) < 30 = "Normal"
    | w / (h*h) < 35 = "Sobrepeso"
    | w / (h*h) < 40 = "Obeso I"
    | otherwise = "Obeso II"
```

Dessa maneira ele vai ter que calcular w / (h * h) toda vez, portanto:

```haskell
bmi :: Float -> Float -> String
bmi h w
    | b < 20 = "Abaixo"
    | b < 30 = "Normal"
    | b < 35 = "Sobrepeso"
    | b < 40 = "Obeso I"
    | otherwise = "Obeso II"
    where
        b = w / (h*h)
```

Where pode ter mais de 1 valor

```haskell
baskara :: Float -> Float -> Float -> [Float]
baskara a b c
    | delta > 0 = [(-b + sqrdelta)/(2*a), (-b - sqrdelta)/(2*a)]
    | delta == 0 = [-b/(2*a)]
    | otherwise = []
    where
        sqrdelta = delta ** 0.5
        delta = b**2 - 4*a*c
```

Padrão

Você pode definir valores padrões para uma função.

f 3 = 2

f 10 = 2

f x = x +1

Exemplo

```haskell
fatorial :: Integer -> Integer
fatorial 0 = 1
fatorial n = n * (fatorial(n-1))
```

Existe um valor padrão para fat 0

```haskell
somapares :: [Integer] -> [Integer]
somapares [] = []
somapares [x] = [x] {- Equivalente a (x:[]), apenas cabeça, xs vazio -}
somapares (x1:x2:xs) = (x1+x2):(somapares xs)
```

```haskell
main = do
    putStrLn $ show $ nth 3 [3,4,5,1,2,3,4]

nth :: (Eq a, Num a) => a -> [b] -> b
nth 0 (x:_) = x
nth n (_:xs) = nth (n-1) xs
```

Tem que utilizar Eq pois há uma comparação de igualdade quando nth 0

Tem que utilizar Num pois há n-1 que é uma operação aritmética

$ é um operador que aplica uma função na outra

Ele aplica a função concatenar, depois o resultado aplica na show, e depois o resultado aplica em putStrLn.

```haskell
main = do
    putStrLn $ show $ conc [2,3] [8,0]

conc :: [a] -> [a] -> [a]
conc [] l = l
conc (x:xs) l = x:(conc xs l)
```

Outra coisa que você pode fazer é colocar o nome da função entre os inputs. Isso chama infix

```haskell
pertence :: (Eq a) => a -> [a] -> Bool
_ `pertence` [] = False
x `pertence` (y:ys)
    | x == y  = True
    | otherwise = x `pertence` ys
```

Também pode chamar a função como prefix utilizando ()

```haskell
main = do
    putStrLn $ show $ mapa (\x -> (*) x x) [2,3,4,

mapa :: (a -> b) -> [a] -> [b]
mapa _ [] = []
mapa g (x:xs) = (g x):(mapa g xs)
```

```haskell
main = do
    putStrLn $ show $ mapa (\x -> (*) x x) [2,3,4,5]
    putStrLn $ show $ mapa (\x -> x `mod` 5) [2,3,4,5]
    putStrLn $ show $ mapa (`mod` 5) [2,3,4,5]

mapa :: (a -> b) -> [a] -> [b]
mapa _ [] = []
mapa g (x:xs) = (g x):(mapa g xs)
```

(\x -> x ‘mod’ 5) [2, 3, 4, 5] #isso daqui aplica 2 mod 5, 3 mod 5, 4 mod 5 e assim vai.
(‘mod’ 5) é a mesma coisa do anterior, pois essa expressão é aplicar parcialmente uma função e já vimos que aplicar parcialmente uma função é criar uma nova função que tem como input os inputs restantes.

```haskell
putStrLn $ show $ mapa (10 `mod`) [2,3,4,5]
```

Agora isso daqui faz 10 % 2, 10 % 3, 10 % 4 e 10 % 5

Reduce

```haskell
main = do
    putStrLn $ show $ reduce (\x y -> x + y) 0 [4,4,6,7,8,9]

{- op = operação, b = base, l = lista -}
reduce op b [] = b
reduce op b (x:xs) = op x $ reduce op b xs
```

Basicamente a funcionalidade da função reduce é reduzir um vetor a um valor levando em consideração uma base e uma operação, nesse caso ela vai somar todos os elementos do vetor,

Essa daqui é uma forma melhor de escrever a função reduce

```haskell
main = do
    putStrLn $ show $ reduce (+) 0 [4,4,6,7,8,9]

reduce :: (b -> a -> a) -> a -> [b] -> a
reduce _ base [] = base
reduce op base (x:xs) = op x $ reduce op base xs
```

```haskell
{- Como aplicar o reduce para a função soma -}
soma = reduce (+) 0

{- Como aplicar o reduce para a função multiplicação -}
produto = reduce (*) 1

{- Como aplicar o reduce para a função concatenar duas listas de string
-}
concatenar = reduce (++) ""
```

Um jeito de fazer a concatenação de dois vetores através do reduce é:

concatenaReduce = reduce (:) l1 l2

mapa f l = reduce ((:).f) [] l

fat n = reduce (*) 1 [1..n]

Either

Permite retornar uma coisa ou outra
É necessário definir qual parte usa cada tipo para isso se usa o left e right

```haskell
main = do
    putStrLn $ show $ primeiro [2,3]
    putStrLn $ either id show $ primeiro []

{- Função que retorna o primeiro elemento de uma lista -}
primeiro :: (Show a) => [a] -> Either a String
primeiro [] = Right "Nao ha elementos"
primeiro (x:_) = Left x
```

Maybe

A resposta pode ser que seja um a.

Nesse caso do código você tem que dizer qual a resposta em 3 casos, que é quando não tem ninguém na lista fazendo com que a resposta seja Nada, quando só tem 1 pessoa na lista, logo o maior vai ser ele mesmo e quando tem mais de dois elementos na lista, onde tem que ver o caso da função aplicada no calda. Se o resultado for nada, então é a cabeça, se o resultado for y então tem que verificar se x é maior ou menor que y.

```haskell
main = do
    putStrLn $ show $ maximo ([] :: [Integer])
    putStrLn $ show $ maximo ([2,3,4] :: [Integer])

maximo :: (Ord a) => [a] -> Maybe a
maximo [] = Nothing {- Construtor do maybe, que não da nenhum valor -}
maximo [x] = Just x {- Construtor de valor do maybe -}
maximo (x:xs) = case maximo xs of
    Nothing -> Just x
    Just y -> Just $ if x > y then x else y
```

Listas infinitas

```haskell
main = do
    putStrLn $ show $ take 10 $ primos
    putStrLn $ show $ sum $ takeWhile (<1000) primos {- Soma dos primos menores que mil -}

primos = let
            b (x:xs) = x:(b (filter ((/=0).(`mod` x)) xs))
         in
            b [2..]
```

É interessante, pois primos retorna uma lista infinita

Usando take a gente consegue pegar só os primeiros 10 primos e usando o takeWhile significa que a gente vai ficar pegando os valores enquanto eles forem menor do que 1000

a linha filter ((/=0).(‘mod’ x)) xs) aplicar xs [i] mod x e verifica se isso é 0, se for, ele passa no filtro e é concatenado. O x é o primeiro elemento de b que é 2.

Struct (data) construtores de dados simples
Funciona como struct em Haskell, exemplo:

data DiaDaSemana = Dom | Seg | Ter | Qua | Qui | Sex | Sab

Aqui DiaDaSemana tem todos esses valores

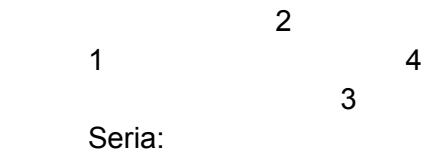
felicidade :: DiaDaSemana -> Integer
felicidade Sex = 10
felicidade Sab = 15
felicidade Qui = 20
felicidade Dom = 8
felicidade _ = 5
Aí dá para definir funções como essas

Definição de uma árvore binária

data ArvoreBB a = Folha | No a (ArvoreBB a) (ArvoreBB a)

Uma árvore pode ter uma folha ou um nó de valor a junto de outras duas árvores

uma forma de definir a árvore:



Seria:

No 2 (No 1 Folha Folha) (No 4 (No 3 Folha Folha) Folha)

Uma função para contar a quantidade de nós seria:

```
contNo :: ArvoreBB -> Int
contNo Folha = 0
contNO ArovreBB _ av1 av2 = 1 + (contNO av1) + (contNO av2)
```

Struct (data) definição de tipos com registro

```
data Aluno = Aluno {nome :: String, curso :: String, media :: Float}
        deriving (Show, Eq, Ord)
```

Esse deriving é para dizer que, nesse caso, você pode printar, verificar se é igual e fazer comparações de maior ou menor.

a1 = Aluno { nome = "Adenilso", curso = "BCC", media = 9.5 }
É impossível trocar o valor de um aluno, logo teria que fazer algo do tipo:
a2 = a1 { media = 9.0 }

Somar a média de um vetor de alunos:

- arr = [a1, a2, a3]
- putStrLn $ show $ sum $ map media $ arr
    - Nesse caso o map pega é média de cada aluno

Somar a média dos alunos que só são do BSI:

- putStrLn $ show $ sum $ map media $ filter ((== “BSI”).curso) $ arr

```
data Vendedor = Vendedor { -- Um Struct Vendedor
   nome :: [Char],
   cpf :: [Char],
   uf :: [Char],
   aniversario :: Data,
   dependentes :: [Idade],
   vendas :: [Venda]}
```

deriving (Show, Read)

Para pegar uma lista dos primeiros nomes dos vendedores separados por espaço.

```
unwords $
map (head.words.nome) $
vendedores
```

words transforma uma string em um array separando por espaço, o unwords faz o inverso

se tiver algum nome repetido é só aplicar nub para tirar os repetidos

Arquivos

```
1    h <- openFile "text.txt" ReadMode
2    c <- hGetContents h
3    hClose h
4    “Operações com c”
```

O problema é que por causa do laziness ele não chegou a ler o conteúdo do arquivo, ele só vai ler quando for ter que fazer uma operação com c, mas aí ele não vai mais conseguir fazer a operações, pois o arquivo já vai ter sido fechado.

O que você vai ter que fazer é:
evaluate $ force c

O force fala que é para ler todas as linhas de c e evaluate para o laziness e le na hora o arquivo.

Javascript

Não tem laziness, sistemas de tipos e estado imutável.

Java
Se usa uma interface para usar a função lambda.

O laziness existe sem ser através do stream.