**Programação funcional**

Tem que entregar 7 atividades.

-> Uma função em C não garante que f(5) retorna sempre o mesmo valor (pode ter alguma variável global)

-> Em programação funcional, as funções podem ser de fato funções -> f(5) sempre retorna o mesmo valor. Por isso que posso chamar função em função: f(g(x)). Vamos estar trabalhando com muita matemática

-> Haskell é uma (e a primeira) linguagem puramente funcional e bem mais segura que C, embora seja pouco útil.

-> Máquina de Turing X Lambda cálculo

-> Lambda cálculo: f 5 = (λx . x+1)5 = 5+1 = 6

-> g = λx.(λy. x+1) -> Os parênteses são dispensáveis (não altera a conta) -> Faço o primeiro com x e o segundo com y

T = λx . λy.x -> retorna o primeiro valor de dois (TF ele retorna T)

F = λx . λy.y

NOT = λp.p F T

AND = λp. λq. p q p

OR = λp. λq. p p q

Ex: AND TT = (λp. λq. p q p)TT = (λq. T q T)T = TTT = (λx . λy.x)TT = T ( λx . λy.x retorna o primeiro valor de T e T)

Ex: AND TF = (λp. λq. p q p)TF = (λq. T q T)F = TFT = (λx . λy.x)FT = F

* **Função pura** é aquela que nunca muda para uma mesma entrada. Print é não pura.
* Efeito colateral
* **Estado imutável (imutabilidade de estado)**: não altera as variáveis depois de criadas

Há como uma função imutável não ter transparência referencial? Sim, usar variável receber duas coisas diferentes em um if else

Em haskell não tem problema usar variável global

* Grande utilização de recursão

Em linguagens funcionais não precisa se preocupar com stackoverflow

* **Transparência referencial**: previsibilidade e substituibilidade de expressões
* **Preguiça (lazyness)**: não faz o cálculo até ser necessário. Só na hora de usar o valor é que ele vai fazer o cálculo: Ex em Haskell: x = [1…100000000000000] ; y = x\*2; (isso tem custo zero). Isso é excelente pq não preciso ter que pensar no tamanho do vetor que vou alocar. Ela é usada por causa do estado imutável, função pura.

Ex: EM DEFINIÇÕES NÃO ACONTECE NADA

x = [1..]; (isso tem todos números naturais)

y = x\*2; -> não acontece nada

z = take(y,10) -> não acontece nada

print(z) -> acontece alguma coisa

* **Função de primeira classe**: você trata inteiro e função como se fosse a mesma coisa.
* **Função de alta ordem**: chama uma função como parâmetro de outra função
* **Sistemas de tipos**: fortemente tipado (tudo tem tipo e tudo tem que ser compatível)

def somavec(a);

if len(a)== 0:

return 0

else:

return a[0] + somavec(a[1:]) # primeiro termo mais cauda

def processavec(a,comb): # função de alta ordem

if(len(a) == 1):

return cabeca(a)

else:

m = processavec(cauda(a),comb)

c = cabeca(a)

return comb(c,m) #operação que se quer fazer (soma do vetor, mult, menor elemento, etc)

def2(x,y):

return x + y

Não dê ctrl c + ctrl v -> faça uma função geral que recebe mais um parametro

def select(a, test): # seleciona dada uma condição test

if len(a) ==0:

return []

else:

c = cabeca(a)

m = select(cauda(a), test)

if test(c):

return [c] + m

else:

return m

def qs(a): # quicksort

if len(a) <= 1:

return a

else:

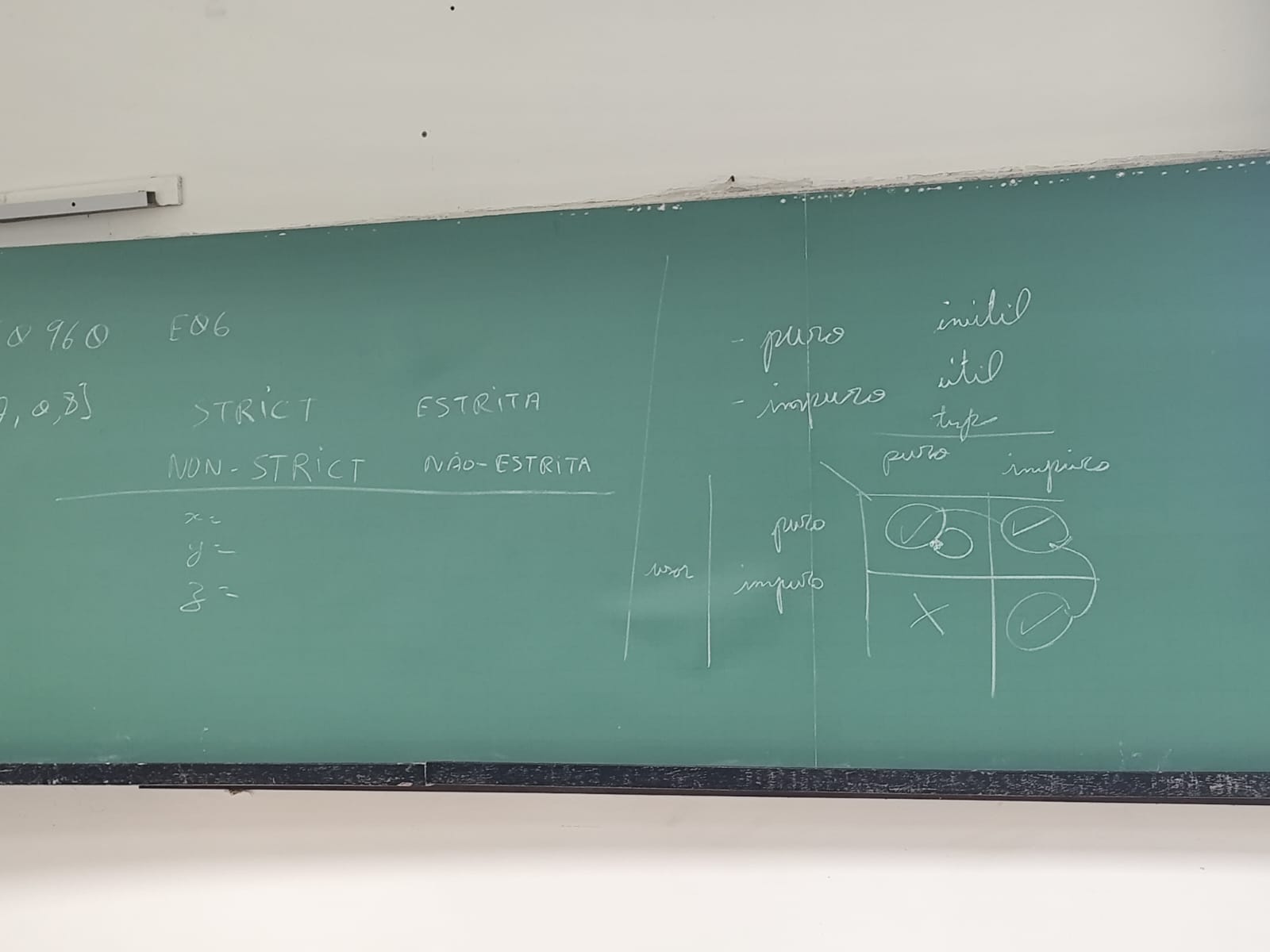
pivot = cabeca(a)

return(

qs(select(a, lambda x: x < pivot)) +

select(a, lambda x: x == pivot) +  
 qs(select(a, lambda x: x > pivot))

)



-- Qualquer expressão ou parte de expressão em haskell tem um tipo

-- Em py tem Integer e Int

-- Haskell é fortemente tipado

-- SISTEMAS DE TIPOS:

-- f x y = x+y

-- Classe de tipo

-- Num é uma classe de tipo: representa todos os tipos que consegue fazer operação aritmética

-- 5 :: Int

-- 5 :: Integer

-- 5 :: Rational

-- '5' :: Char

-- [5,5] :: [Int]

-- f x = x + x

-- classe de tipo

-- Num

-- construtores

-- Int

-- [Int]

-- [Char]

-- [[Int]]

-- 5 :: (Num a) => a

-- [3,5] :: (Num a) => [a]

-- g :: Int -> Integer

-- [Int] -> Int

-- [[Int] -> Int] -> [[Int]]

-- Int -> Int -> [Int] -- Função que, dado um inteiro, retorna uma função que, dado um inteiro, retorna uma lista de inteiro

f x y = x+y

f :: (Num a) => a -> a -> a

m x y = x + 3

-- m:: (Num a) => a -> (b -> a)

Int -> Int -> Int

Int -> Int -> [Int]

(Int -> Int) -> Int X

Int -> (Int -> [Int] x)

f :: (Num a) => a -> a -> a

(f 3 5) :: (Num a) => a

(f 3) :: (Num a) => a -> a

h = f 1

h :: (Num a) => a -> a

h 6

h 41

n x y = x

n:: b -> a ->b

p x y = 42

p :: (Num c) => a -> b -> c

-- ou

p :: (Num a, Num c) => a -> b -> c

g x y = x + 3

f x y = x + y

g :: (Eq a) => a -> a -> Bool

f :: (Num a, Ord a) => a -> a -> Bool

f x = x > 5

[] :: [a]

x = [5, 8] :: (Num a) => [a]

y = 3 :: (Num a) => a

(y : x) :: (Num a) => [a]

(:) :: a -> [a] -> [a]

(:) :: (Num a) => a -> [a] -> [a]

mapa [] = []

mapa (x:xs) = (2\*x):(mapa xs)

mapa :: Num a => [a] -> [a]

-- Def do map

map f [] = []

map f (x: xs) = (f x) : (map f xs)

map:: (b->c) => [b] -> [c]

-- -------------

mult2 x = 2\*x

mapa l = map mult2 l

mapa l = map(\x->2\*x)l -- mais direto

mapa map(\x -> 2\*x)

-- --------------

m x y = x + 3

(1;0)(1)

g x y = x = y

9 : (f q q) = a → a → Bool

f x y = x + y

Σ: (Num a, Ord a) => a → a → Bool

δ x = x > 5

Aula 1:

main :: IO ()

main = do

l1 <- getLine

l2 <- getLine

let x :: Integer

x = read l1

let y :: Integer

y = read l1

let z = add x y

let str = show z

putStrLn str

add :: (Num a) => a -> a -> add

add x y = x + y

-- strict e nao strint

-- nao strict é legal trabalhar com SO q eh lazy

-- nao strict calcula o valor que vc passou na função se necessário (diferente de C)

Primeiro código:

-- Coloque os tipos nas coisas

main :: IO () -- o IO marca um tipo

-- main = putStrln "Hello World"

main = do -- Só funções impuras usam do, por causa da imperativo.

putStrLn "Hello"

putStrLn "W"

l <- getLine -- l é constante e impura

-- let v :: Integer

let v = read l

-- let dummy = getLine Isso dá errado

-- O let faz o mostra que é algo puro dentro de uma funcao impura (no caso, o da main)

let x = f v --- Seja x f (41)

let y = x\*2

let str = show y

putStrLn str

f :: (Num a) => a -> a

f x = x +1

-- Em haskell vc nunca vai ter um função pura supreeendementemente virando algo impuro

h:: (Num a, Ord a) => a->a->a->

h x y z = if x>10

then y

else z

w1 = sum [1..100]

w2 = sum [1..]