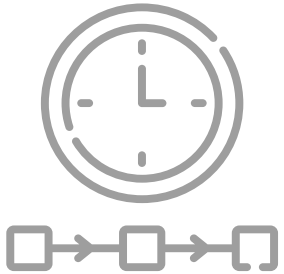
Minicurso de Haskell

Introdução
Programação Funcional
Haskell

Cronograma



Sala 334 - CT

Para o Certificado:

- 50% de presença (4 aulas)
- 70% das atividades

A maioria das aulas vai ter uma pequena atividade via Google Forms

- Você não precisa acertar apenas tentar

Material e atividades em github.com/Fleivio/Minicurso_Haskell

Baixar?

GHCup

Instalador

GHC

Compilador

HLS

Lang Server

Cabal

Gerenciador de pacotes

Para este minicurso você pode se contentar com interpretadores online.

STOP DOING HASKELL

- CODING LANGUAGES WERE NOT SUPPOSED TO BE PURE
- YEARS OF CODING yet NO REAL-WORLD USE FOUND for IMMUTABLE DATA
- Wanted to make data immutable anyway for a laugh? We had a tool for that: It was called **CONST**
- "Yes please make iteration unviable . Please force me to use currying." - Statements dreamed up by the utterly Deranged

LOOK at what Mathematicians have been demanding your Respect for all this time, with the pure coding environment we built for them

(This is REAL Haskell syntax done by tortured computer scientists**):**

```haskell
concatEvenOdd :: [String] -> (String, String)
concatEvenOdd xs = foldl accumulation ("","") (zip [0,1..] xs)
    where
      accumulation :: (String, String) -> (Int, String) -> (String, String)
      accumulation (evenStr, oddStr) (i, str)
        | even i = (evenStr ++ str, oddStr)
        | otherwise = (evenStr, oddStr ++ str)
```

?????

```haskell
(<#) :: Ord a => [a] -> [a] -> Int
[] <# _ = 0
_ <# [] = 0
(x:xs) <# (y:ys)
    | x < y = 1 + xs <# ys
    | otherwise = xs <# ys
```

???????

```haskell
listSearch :: Eq a => [a] -> [a] -> Maybe Int
listSearch _ [] = Nothing
listSearch xs ys = parseList xs ys 0
    where
      parseList _ [] _ = Nothing
      parseList xs ys n
        | xs `isPrefixOf` ys = Just n
        | otherwise = parseList xs (tail ys) (n + 1)
```

???????????????

"Hello I would like `x <- [1,2..]` apples please"

They have played us for absolute fools

Programação Funcional

Funções de Alta Ordem

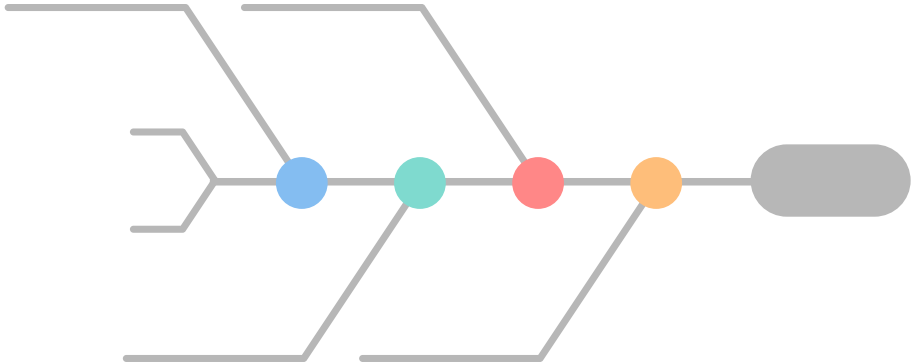
Funções são cidadãos de primeira classe

Imutabilidade

Não existem variáveis

Puridade

Não existem efeitos colaterais



Haskell!

Sintaxe
Aplicação parcial
Puridade

Sintaxe do Haskell

Nada de Parênteses

Enquanto outras linguagens adoram *f(x, y),* em Haskell fazemos apenas *f x y*

```
> id 3
3

> mod 11 2
1

> div 11 2
5
```

Sintaxe do Haskell

Definindo Funções

Todas as funções do Haskell são fortemente e estaticamente tipadas

```
mult2 :: Int -> Int
mult2 x = 2 * x
```

-- Função que dobra um número

```
concatStr :: String -> String -> String
concatStr x y = x ++ y
```

-- Função que concatena duas strings

Sintaxe do Haskell

Funções de Alta Ordem

Funções podem receber e retornar outras funções

```haskell
app2 :: (Int -> Int) -> Int -> Int
app2 f x = f (f x)
-- Aplica uma função duas vezes
```

```haskell
comp :: (Int -> Int)
     -> (Int -> Int)
     -> Int -> Int
comp f g x = f (g x)
-- Compõe duas funções
```

Pequeno Exercício

Defina as seguintes funções

```
mult2 :: Int -> Int
mult2 x = 2 * x
   -- Lembre-se da sintaxe
```

```
mult4 :: Int -> Int
-- que multiplia um número por 4
```

```
quadSum :: Int -> Int -> Int
-- que retorna o quadrado da
soma de dois números
```

Sintaxe do Haskell

Aplicação Parcial

Funções não precisam receber todos os seus parâmetros de uma só vez

A sintaxe da tipagem tem um motivo

```
> :t quadSum
quadSum :: Int -> Int -> Int

> :t quadSum 3
quadSum :: Int -> Int

> :t quadSum 3 2
quadSum :: Int
```

-- O primeiro parâmetro foi saturado, mas o Haskell ainda sabe que falta um Int ainda

Sintaxe do Haskell

Aplicação Parcial

É como se *quadSum* fosse definida da seguinte forma

```
quadSum :: Int -> (Int -> Int)
```

Para cada parâmetro saturado, uma nova função é retornada, só que com um parâmetro a menos

Sintaxe do Haskell

Point Free Syntax

A aplicação parcial abre margem para que funções possam ser declaradas sem expressarmos seus argumentos

```
doisDiv :: Int -> Int
doisDiv x = div 2 x
doisDiv = div 2
```

```
mult4 :: Int -> Int
mult4 x = 4 * x
mult4 = (4*)
```

Pequeno Exercício

Redefina a função app2 utilizando a função comp

```haskell
app2 :: (Int -> Int) -> Int -> Int
app2 f x = f (f x)
```

```haskell
comp :: (Int -> Int)
     -> (Int -> Int)
     -> Int -> Int
comp f g x = f (g x)
```

```haskell
mult4 :: Int -> Int
mult4 = (4*)
```

-- Lembre-se da sintaxe

Sintaxe do Haskell

Aplicação Parcial

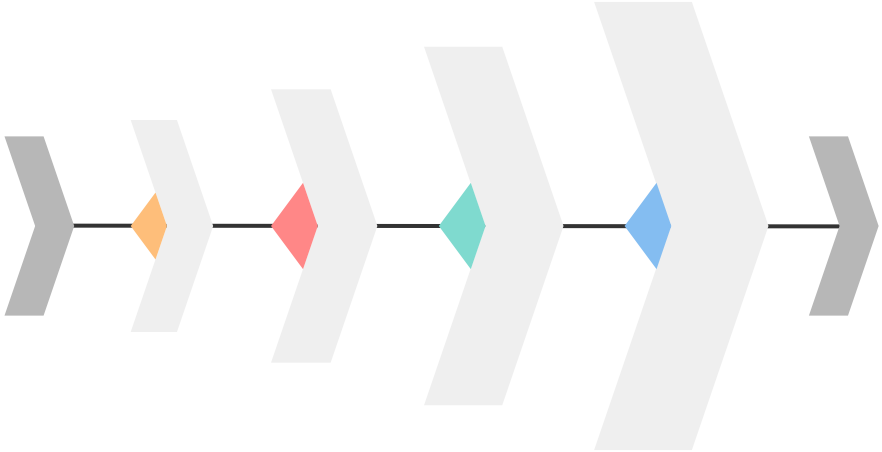
É como se *comp* fosse definida da seguinte forma

```
comp :: (Int -> Int)
     -> (Int -> Int)
     -> (Int -> Int)
```

Para o Haskell, ter ou não os parênteses em verde é a mesma coisa

Puridade e Imutabilidade

Eu só queria fazer um
Hello World



Questão (10pts). Sendo $g(x)$ uma função, tal que

$$g(3) = 4$$

Quanto vale $g(3)$?

Uma função é pura se

Não possui side-effects

Não acessa arquivos

Não altera variáveis fora do seu escopo (globais ou passados por referência)

Não tem variáveis estáticas locais

É transparente de referencial

Quando chamada com os mesmos argumentos, quantas vezes se queira, sempre retorna o mesmo resultado

Em Haskell, todas as definições são imutáveis

Se você disse que *x = 3*
, então *x = 3* para sempre.

Você não pode simplesmente decidir que agora *x=4*. Você disse que *x=3* antes. Você é algum tipo de mentiroso por acaso?

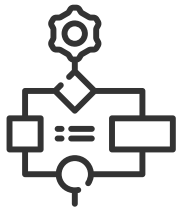
Da mesma forma, para funções, se f(3) = 4, então f(3) = 4

Puridade  Imutabilidade

Laziness


Puridade
Transparência do referencial
Nada de side-effects
Não importa a ordem de avaliação
Imutabilidade
Nada de estado global
Função não mudam

No Haskell vai haver uma divisão muito clara entre o que é puro e impuro



Parte pura

Quase tudo

Avaliado de forma preguiçosa

Parte impura

Tipo IO

Ordena a execução