Programação funcional

AULA 01 - SOMA

{-

N = 4 ⇒ 1 + 2 + 3 + 4 = 10

N = 3 ⇒ 1 + 2 + 3 = 6

soma(n)

se o n=1, então soma(n)=1

se o n>1, então soma(n)=soma(n-1)+1

-}

-- main = putStrLn "Olá, Haskell no VS Code!"

soma 1 = 1 -- caso base

soma n = soma(n-1) + n -- caso recursivo

main = print (soma 4) -- Teste com n = 5

AULA 02 - FATORIAL

*— FATORIAL : 3! = 3*2*2 = 6.*

— função fatorial, recebe parametro n

{- fatorial(n) = 1 se n = 0

*fatorial(n) = fatorial(n-1)*n se n>=1*

main = print (soma 4) -- Teste com n = 4 -}

fatorial 0 = 1

fatorial n = fatorial(n-1)*n

main = print (fatorial 4)

ou por guards

```
fatorial n
| n == 0    = 1
| otherwise = n * fatorial (n - 1)
```

AULA 03 - FIBONACCI

```
{-
se n = 0, então fib(n)=0
se n = 1, então fib(n)=1
se n > 1, então fib(n)=fib(n-1) + fib(n-2)
exemplo: n = 3
fib(3) = fib(2) + fib(1) = 1 + 1 = 2
fib(1) = 1
fib(2) = fib(1) + fib(0) = 1 + 0 = 1
fib(0) = 0
-}
— utilizando guards
fib :: Int → Int
fib n
| n == 0    = 0
| n == 1    = 1
| otherwise = fib (n - 1) + fib (n - 2)
main :: IO ()
main = print (fib 0)
```

AULA 04 - GUARDA

```
guarda x | (x == 0) = 0
         | (x == 1) =1
         | otherwise = 10
```

AULA 05 - VARIÁVEL ANÔNIMA

```
andi :: Bool → Bool → Bool
andi False _ = False
andi _ False = False
andi True True = True
main :: IO ()
main = print (andi False True)
```

AULA 06 - TUPLAS

- conj de dados heterogenios

```
func :: (Int, Int) → (Int, Int) → (Int, Int)
func (a,b) (c,d) = (a+c, b+d)
main :: IO ()
main = print (func (1,2)(2,4))
```

AULA 07 - EXTRAINDO DADOS DE TUPAS

```
nomes :: (String, String, String)
nomes = ("Raissa", "Amanda","João")
selec_prim (x, _, _) = x
selec_sec ( _, y, _) = y
selec_ter ( _, _, z) = z
main :: IO ()
main = print(selec_prim nomes)
```

AULA 08 - DEFININDO NOVOS TIPOS (FUNÇÃO TYPE)

type Nome = String

type Idade = Int

type Linguagem = String

type Pessoa = (Nome, Idade, Linguagem)

pessoa :: Pessoa

pessoa = ("Joao", 20, "Haskell")

main :: IO ()

main = print(pessoa)

// Retorna primeiro elemento da tupa

my_fst :: Pessoa → Nome

my_fst (n, i, l) = n

- - main = print(my_(pessoa))

AULAS - LISTAS

- *listas comprimento*
  size_list []=0
  size_list (a:b) = 1 + size_list b -- retirando a cabeça da lista e contando o corpo/CHAMADA RECURSIVA
- *função que verifica se duas listas são iguais(mesmo elementos e numeros de elementos na mesma posição)*
  comp_listas :: [Int] → [Int] → Bool
  comp_listas [] [] = True
  comp_listas [] _ = False
  comp_listas _ [] = False
  comp_listas (a:b) (c:d)
  | (a == c) = comp_listas b d
  | otherwise = False
- função que retorna o inverso da lista

inv_listas :: [t] → [t]

inv_listas [] = []

inv_listas (a:b) = inv_listas b ++ [a]

- pertence

pertence :: Int → [Int] → Bool

pertence _ [] = False

pertence n (a:b)

| n == a   = True

| otherwise = pertence n b

- maior elemento da lista
  maior :: [Int] → Int
  maior = x
  maior (a:b)
  | (a > maior b) = a
  | otherwise = maior b

AULA 15 - Compreensão de listas

[x | x ← [1,2,3]]   = [1,2,3]

[x + 1 | x ← [1,2,3]] = [2,3,4]

[x*x | x ← [1,2,3]] = [1,4,9]

[1 .. 10] = [1,2,3,4,5,6,7,8,9,10]

par:: Int → Bool

par x = mod x 2 == 0

lista = [x| x ← [1..10], par x, x>5]

tupa de dois elementos

[(x,y) | x ← [1..5], y ← [6..10]]

MOD = resto da divisão!!!

/= diferente

import Data.Char → chr ord tabela haskell

sqrt = raiz quadrada

---

polimorfismo = não importa o tipo / tipo generico

```
my_length :: [a] → Int

my_length [] = 0

my_length (x:xs) = 1 + mylength xs
```

AULA 23

```
-- IF
if_par :: Int → Bool
if_par n = if (mod n 2 == 0) then True else False
-- CASE
case_par :: Int → Bool
case_par n = case (mod n 2 == 0) of
True → True
False → False
-- GUARDA
guarda_par :: Int → Bool
guarda_par n
| (mod n 2 == 0) = True
| otherwise = False
```

WHERE

--> cria variáveis auxiliares ou expressões locais
- utilizada no final

```
quad :: Int → Int
quad n = quad_n
where
quad_n = n * n
```

aula funcional