# Recapitulação

Programação Funcional

Prof. Maycon Amaro

# Programação Funcional

É um paradigma de programação que se baseia em funções modeladas por *funções matemáticas*. Programas são combinações de *expressões*.

## Função Matemática

- Uma função é uma relação entre um conjunto de possíveis entradas e um conjunto de possíveis saídas.
- Para ser uma função, cada elemento do domínio só pode estar relacionado a no máximo um elemento do contra-domínio.
- Uma função é total se está definida para todo elemento do domínio.
- Uma função é parcial se existe algum elemento do domínio para o qual ela não está definida.

# Tipos e Conjuntos

- Todo tipo pode ser entendido como um conjunto, cujos elementos s\(\tilde{a}\) os valores desse tipo.
- Uma função é um mapeamento entre tipos assim como é entre conjuntos.

# Transparência Referencial (Função Pura)

► A propriedade de uma função poder ser substituída pelo seu valor de retorno sem afetar o comportamento do programa.

#### Requisitos

- deve sempre retornar o mesmo resultado para uma mesma entrada
- deve não possuir efeitos colaterais

### Lambda Cálculo

- A base das linguagens funcionais.
- ► Sistema baseado em expressões.
- ▶ É um sistema muito simples e muito poderoso.

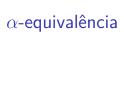
#### Sintaxe

#### Possui apenas três construções:

- ► Variáveis: um nome para um potencial valor.
- Abstrações: definições de funções.
- Aplicações: a aplicação de uma função em seus argumentos.

#### Formalmente:

$$e := v \mid \lambda v.e \mid e e$$



Variáveis ligadas podem ser renomeadas sem alterar o significado da abstração.

 $\beta$ -redução

$$(\lambda v.e_1) e_2 \longrightarrow [v \mapsto e_2]e_1$$

em que  $[v \mapsto e_2]e_1$  significa capture-avoiding substitution das ocorrências livres de v em  $e_1$  pelo termo  $e_2$ .

# Estratégias de Avaliação

- ► Full-beta reduction
- ► Normal-order
- ► Call-by-name (avaliação lazy)
- ► Call-by-value (avaliação estrita)

# Influências em Outras Linguagens

- Funções anônimas
- Closures
- Avaliação lazy

### Haskell

- É uma linguagem funcional pura, de propósito geral, com tipos estáticos.
- Principal implementação é o GHC.
- ► Famoso gerenciador de projetos é o Stack.

# Definindo funções

```
sucessor :: Int -> Int
sucessor x = x + 1
```

### Executando funções

- Entrada e Saída envolve efeitos colaterais
- ► Ambiente interativo (GHCi) é bastante prático

## Consultando o tipo

Dentro do ambiente interativo, digite :t <funcao> ou :t (operador) para verificar o tipo dele.

```
stack ghci
ghci> :t (+)
(+) :: Num a => a -> a -> a
```

#### Casamento de Padrão

```
conjuncao :: Bool -> Bool -> Bool
conjuncao False _ = False
conjuncao True x = x
```

#### Guardas

# Função anônima

```
soma1 :: Int \rightarrow Int
soma1 = (\ x y \rightarrow x + y) 1
```

#### Recursão

- ► Em Haskell não há for, while ou estrutura semelhante. Para repetições, temos que usar recursividade.
- Alguns padrões de recursão estão abstraídos no prelúdio: map, filter, foldr, all, any, etc.
- Por conta da avaliação lazy, analisar o desempenho de funções recursivas em Haskell é mais complicado.

#### **Imutabilidade**

- ► Toda variável ou estrutura em Haskell é imutável.
- ▶ Reuso de nomes no máximo leva a **shadowing**.
- ► Transparência referencial é garantida.

### Calculadora de MDC e MMC

```
euclides :: Int -> Int -> Int
euclides x 0 = x
euclides x y = euclides y (mod x y)

mdc :: Int -> Int -> Int
mdc x y = euclides (abs x) (abs y)

mmc :: Int -> Int -> Int
mmc x y = div ((abs x) * (abs y)) (mdc x y)
```

# Álgebra de Tipos

- ► Tipo vazio é 0
- ► Tipo unitário é 1
- ► Variants e Enumerações são soma
- ► Tuplas e Structs são **produto**
- Funções são análogas à potenciação

### Exemplo em Haskell

Tipo Pessoa com um nome e uma idade.

```
data Pessoa = Pessoa Int String
```

```
exemplo :: Pessoa
exemplo = Pessoa 27 "Amy Winehouse"
```

#### Usando Records

```
data Pessoa = Pessoa { idade :: Int, nome :: String }
exemplo :: Pessoa
exemplo =
  Pessoa { nome = "Amy Winehouse", idade = 27 }
```

# Tipos de Dados Algébricos

- Casamento de padrão funciona normalmente
- ▶ Tipos recursivos são super simples de definir, já que Haskell abstrai o uso dos ponteiros.

# Composição de Função

$$g(x) = x + 1$$
$$f(x) = 2 * x$$

 $h = f \circ g$ 

Isso inspira o point-free style.

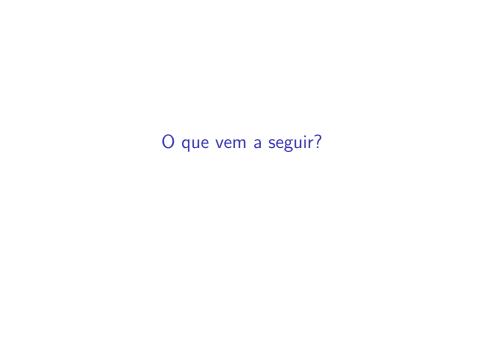
## Point-free style

```
ehPar :: Int -> Bool
ehPar x = even x
se torna:
ehPar :: Int -> Bool
ehPar = even
```

# Exemplo

Um número que não é par, é ímpar:

```
odd :: Int -> Bool odd = not . even
```



# Próximos tópicos

- Programação Genérica e Classes de Tipos
- ► Funtores, Aplicatives e Mônadas
- Noções de Type-level Programming
- ► Tópicos especiais (avaliação lazy, ordenação, etc)
- Testes e Verificação Formal