PARADIGMA FUNCIONAL

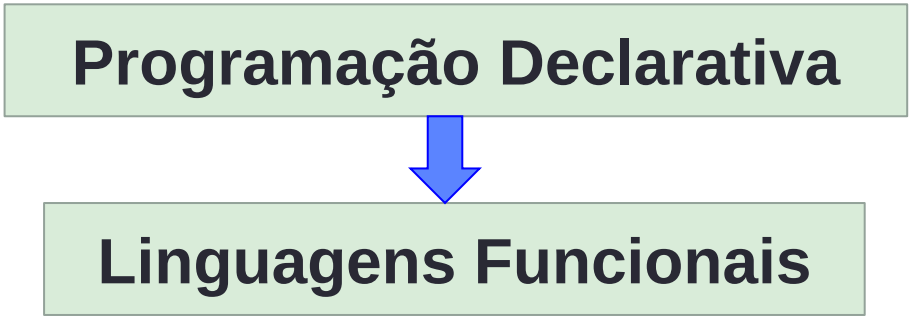
Everton L. G. Alves
everton@computacao.ufcg.edu.br

Motivação

- Como lidar com o **tamanho** e a **complexidade** dos programas modernos?
- Como reduzir o **tempo** e o **custo** do desenvolvimento?
- Como podemos aumentar nossa **confiança** de que os programas funcionam corretamente?

Motivação

- Uma abordagem para resolver esses problemas é a concepção de **novas linguagens** de programação que:
  - Permitam que programas sejam escritos de forma **clara**, **concisa**, e com um **alto nível de abstração**
  - Suportem componentes de software **reutilizáveis**
  - Incentivem o uso de **verificação formal**
  - Permitam **prototipagem rápida**



Introdução – LPs Imperativas

- LPs como C, Fortran, Java são classificadas como imperativas
  - Baseiam sua programação na modificação do estado de variáveis

$$a = a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n = a'$$

Antes da execução, o estado possui um valor inicial α, representando as entradas do programa. Durante a execução, os comandos alteram o estado. Ao final, o estado tem um novo valor α'

  - Exemplo: um programa que realiza a ordenação de um conjunto
    - Estado incial: um array de valores
    - Estado intermeriário: o array ainda não ordenado (modificado para atingir o objetivo)
    - Estado final: o mesmo array ordenado

Introdução – LPs Imperativas

- Mudanças de estado são tipicamente realizadas usando **comandos de atribuição**
  - v = E ou v:= E
- **Comandos condicionais** e de **repetição** impõem fluxos de execução
  - if/else
  - while
- O programa é um conjunto de instruções que indicam como mudar o estado das variáveis
  - **Estilo imperativo**!

Paradigma Funcional

- Forte ruptura do modelo imperativo

- Um **programa** funcional é essencialmente uma **função** que pode ser uma **composição** de funções

- Assumindo um programa funcional, sua saída é completamente determinada pelas entradas que recebe. Portanto:

$$a \Leftarrow f(a)$$

Paradigma Funcional

- Programação funcional é baseada no conceito de **funções matemáticas**

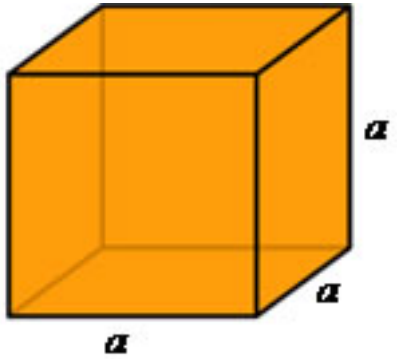
- **Recursões** e **expressões condicionais**, ao invés de sequenciamento e iterações

- **Transparência Referencial**
  - Sem efeitos colaterais!

- Funções são **valores de primeira ordem**

Função - Exemplo

- Função volume de um cubo
  - cubo (x) = x * x * x



  - **Domínio**: conjunto dos número reais
  - **=** : a função é definida como
  - **x** : pode representar qualquer elemento do domínio

Nesse caso, o x tem a mesma semântica de uma variável no paradigma imperativo? NÃO

Programação Funcional

- Tem como padrão a busca por "imitar" funções matemáticas

- Funções Matemáticas:

cubo (x) = x * x * x

$$mdc(m,n) = \begin{cases} n & , \text{se } m = 0 \\ mdc(n \bmod m,\ m) & , \text{se } m > 0 \end{cases}$$

- Programas Haskell:

```
cubo x = x * x *x
```

```
mdc m n
| m == 0 = n
| m > 0 = mdc (n `mod` m) m
```

Programação Funcional

- Na programação funcional não existe a noção de mudança de estado
  - Instruções de atribuição tendem a ter caráter limitado

- Exemplo: Soma dos naturais de 1 a 10

Exemplo - Java

```
int total = 0
for (int i = 1; i<=10; i++)
    total = total + i
```

- A computação é realizada via mutação das variáveis *total* e *i*

passo	instrução	i	total
1	total = 0	?	0
2	int i = 1	1	0
3	total = total + i	1	1
4	i++	2	1
5	total = total + i	2	3
6	i++	3	3
7	total = total + i	3	6
...			
21	i++	10	45
22	total = total + i	10	55

Exemplo - Haskell

- Em Haskell, a mesma computação se dá via aplicação da função *sum*, passada uma lista de inteiros [1..10]

```
sum [1..10]
```

- Na execução passo a passo, a expressão original é reduzida até que não possa mais ser simplificada

sum [1..10] = sum [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
= 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
= 3 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
= 6 + 4 + 5 + 6 + 7 + 8 + 9 + 10
...
= 55

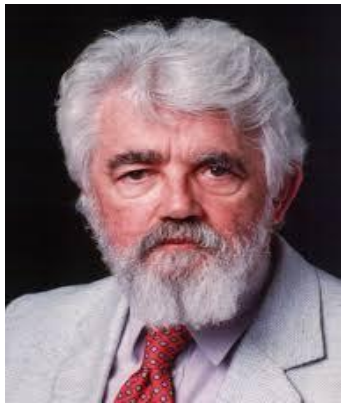
Por que Estudar Programação Funcional?

- Visão clara de conceitos fundamentais:
  - Abstração e tipos abstratos de dados
  - Recursão
  - Genericidade, polimorfismo, sobrecarga
- Programação com um alto nível de abstração, possibilitando:
  - Alta produtividade
  - Programas mais concisos
  - Menos erros
  - Provas de propriedades sobre programas
- Primeiro paradigma ensinado em várias universidades importantes
  - Stanford, Berkeley

Histórico

- Na década de 1930 Alonzo Church desenvolve o **cálculo lambda**, uma teoria de funções simples, mas poderosa

- Na década de 1950 John McCarthy desenvolve **Lisp**, a primeira linguagem funcional, com influências do cálculo lambda, mas mantendo as atribuições de variáveis



- Na década de 1960 Peter Landin desenvolve **ISWIM**, a **primeira linguagem funcional pura** baseada no cálculo lambda, sem atribuições

Histórico

- Na década de 1970 Robin Milner e outros desenvolvem **ML**, a primeira linguagem funcional moderna, que introduziu a inferência de tipos e tipos polimórficos

- Na década de 1980 surgiu **Haskell**

Usos Práticos

- Compiladores, provadores de teoremas, sistemas Web, serviços de chat de grande escala, IA, etc.

- Erlang
  - Serviço de chat do Facebook
- Scala (linguagem híbrida, parte funcional)
  - Serviço de filas de mensagens no Twitter
- Scheme e LISP
  - Ensino de programação em várias universidades
- ML, F#
  - Verificação de HW e SW na Microsoft e na Intel

Haskell

- Um comitê surgiu durante a FPCA 87 (Functional Programming Languages and Computer Architecture)
  - Consolidar as LP funcionais existentes em uma nova LP padronizada

- Haskell foi proposta em 1990
- Versão mais recente: Haskell 2010

Haskell

- Linguagem **funcional** de uso genérico
- Concebida para ensino e também para o desenvolvimento de aplicações reais

- Haskell no mundo real:
  - **GHC** o compilador de Haskell é escrito em Haskell
  - **Darcs** um sistema distribuido para gestão de código-fonte
  - **Chordify** extração de acordes musicais (https://chordify.net/)
  - https://wiki.haskell.org/Haskell_in_industry

Haskell - GHC

- Compilador Haskell
- Suporta Haskell 98, Haskell 2010 e muitas extensões
- Grande conjunto de bibliotecas, etc.
- Inclui também o interpretador *ghci*
  - Um programa ***Haskell*** pode ser **compilado** (em código binário nativo) ou ser **interpretado** (na forma de um *script*).

- Disponível em: https://www.haskell.org/ghc/

Haskell

- Características importantes de Haskell
  - Sintaxe similar a ML
  - **Estaticamente** e **fortemente** tipada
  - **Puramente** funcional
    - Não possui variáveis
    - Nem outras características imperativas
  - Possui **avaliação preguiçosa** (lazy evaluation)
  - Permite trabalhar com **listas infinitas**

Haskell – Valores e Tipos

- Haskell é uma LP fortemente tipada
  - Toda função, variável ou constante possui um tipo que sempre poderá ser determinado

- Embora fortemente tipada, Haskell possui um sistema de dedução automática de tipos para funções cujos tipos não forem definidos

Haskell – Valores e Tipos

- Em Haskell escrevemos ***e:: T*** para indicar que a expressão *e* admite o tipo *T*

- Tipos primitivos:
  - *Bool*: valores lógicos
  - *Char*: caracteres simples
  - *String*: sequencias de caracteres
  - *Int*: inteiros de precisão fixa
  - *Integer*: inteiros de precisão arbitrária
  - *Float*: vírgula flutuante de precisão simples
  - *Double*: vírgula flutuantes de precisão dupla

Haskell – Valores e Tipos

- Linguagens funcionais possuem tipos semelhantes aos de outras linguagens de programação

- Porém a modificação de valores é proibida

Programação Imperativa:
l = [1,2,3];
l.add(8);

l agora armazena a lista [1,2,3,8]

Programação Funcional;
l = [1,2,3]
add 8 [1,2,3] retorna [1,2,3,8]

l permanece [1,2,3]

- Ponteiros são manipulados implicitamente em linguagens funcionais
- Objetos são desalocados automaticamente

Haskell – Prototipação de Tipos

- Em Haskell, toda função requer uma prototipação de tipos
  - Sequência dos argumentos da função, sendo o último tipo o do valor de retorno da função

nome_da_funcao :: $Tipo_{arg1}$ -> $Tipo_{arg2}$ ...-> $Tipo_{argsaida}$

  - Ex: Função que verifica se um número inteiro é par

```
par :: Int -> Bool
```

Haskell – Prototipação de Tipos

- Exemplos:
  - Função que verifica se um caractere está presente em uma string

```
findChar :: String -> Char -> Bool
```

  - Função que calcula o mmc de dois números

```
mmc:: Int -> Int -> Int
```

Haskell – Funções

- Funções em Haskell são normalmente definidas pelo uso de equações
- Ex: função soma pode ser escrita:

```
soma :: Int -> Int -> Int
soma x y = x + y
```

- Invocação da função:

```
>soma 10 20
30
```

Haskell – Composição de Funções

- Um dos pontos chaves da programação funcional é o uso da **composição de funções**

```
soma :: Int -> Int -> Int
soma x y = x + y
```

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

Incremento da soma de 10 e 20?

```
> incrementa (soma 10 20)
```

Exemplos

- Escreva uma função para calcular o dobro de um número

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

- Reusando a função definida anteriormente, escreva uma função para quadruplicar um número

```
quad:: Int -> Int
quad x = 2 * (dobro x)
```

Definição de Funções

- Linguagens funcionais normalmente oferecem diversos mecanismos para definição de funções

- Em Haskell:
  - **Expressões condicionais**
  - **Alternativas com guardas**
  - **Casamento de padrões**
  - Expressões Case
  - Expressões Lambda
  - Seções

Definição de Funções – Expressões Condicionais

- Usa uma expressão lógica para escolher entre **dois resultados** de **mesmo tipo**
- Exemplo: Função que retorna o valor absoluto de um número:

```
abs:: Int -> Int
abs n = if n > 0 then n
        else -n
```

- Sintaxe de uma expressão de seleção bidirecional:

```
if <condição>
    then <resultado 1>
    else <resultado 2>
```

- O ELSE é obrigatório!

Definição de Funções – Expressões Condicionais

- O else é obrigatório. Por que?
  - O if em Haskell é uma expressão.
  - Uma expressão é um trecho de código que retorna um valor (e.g., 5 é uma expressão porque retorna 5; 4 + 8 é uma expressão porque retorna 12)
  - Toda expressão deve retornar alguma coisa
  - Por isso, o else é obrigatório, caso contrário, se a condição do if não fosse satisfeita, não haveria retorno

```
menor :: Int -> Int -> Int
menor x y = if x <= y then x
            else y
```

Definição de Funções – Expressões Condicionais

- A expressão condicional é uma **expressão**, portanto **sempre tem um valor**
- Assim uma expressão condicional pode ser usada dentro de outra(s) expressão(ões).

- Exemplos:

```
5 * if even 2 then 10 else 20

(if even 2 then 10 else 20) + 1

if even 2 then 10 else 20 + 1
```

Definição de Funções – Alternativas com guardas

- Expressão **multidirecional** para escolher entre uma **sequência de resultados** de mesmo tipo

- As guardas permitem estabelecer uma distinção entre casos diferentes da definição de uma função

```
nome_funcao arg1 ... argn
        | guarda_1 = exp_1
        ...
        | guarda_m = exp_m
```

```
abs n | n >= 0 = n
      | otherwise = -n
```

Definição de Funções – Alternativas com guardas

- Função para retornar o maior entre três números:

```
maxTres :: Int -> Int -> Int -> Int
maxTres x y z
      | x >= y && x >= z        = x
      | y >= z                  = y
      | otherwise               = z
```

```
> maxTres 6 (4+3) 5
?? 6 >= (4+3) && 6 >= 5
?? 6 >= 7 && 6 >= 5
?? False && True
?? False
?? 7 >= 5
?? True
7
```

Exemplo

- Escreva uma função **conceito** que recebe uma nota como argumento e retorne o conceito correspondente
  - Nota >= 9.0, conceito A
  - Nota >= 7.5 e < 9.0, conceito B
  - Nota >= 6.0 e < 7.5, conceito C
  - Nota >= 4.0 e < 6.0, conceito D
  - Nota < 4.0, conceito E

```
conceito :: Float -> Char
conceito n
      | n >= 9 = ‘A’
      | n >= 7.5 = ‘B’
      | n >= 6 = ‘C’
      | n >= 4 = ‘D’
      | otherwise = ‘E’
```

Definição de Funções – Casamento de Padrões

- Uma sequência de "**padrões**" é usada para escolher entre uma **sequência de valores** de mesmo tipo
- Se existe um **match** (casamento) entre o padrão e o argumento, este é escolhido e demais não são testados

```
not :: Bool -> Bool
not False = True
not True = False
```

```
totalSales :: Int -> Int
totalSales 0 = sales 0
totalSales n = totalSales (n-1) + sales n
```

Definição de Funções – Casamento de Padrões

- Função definida por várias equações, cada uma com um padrão no seu lado esquerdo

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Vantagens:

- Simplifica o uso de recursividade
- Concisão e clareza
- Mais próxima de uma definição matemática

Cláusulas *where*

- Equivalentes às “variáveis locais”
  - Mas não iguais!
  - Lembre que não existem variáveis na programação funcional.
- Evitam a repetição de código e recálculos

```
bmiTell :: Float -> Float -> String
bmiTell weight height
    | bmi <= 18.5 = ”Underweight!"
    | bmi <= 25.0 = ”Regular"
    | otherwise   = ”Overweight!"
    where bmi = weight / height ^ 2
```

Exercício

- Defina a função *roots* que calcula as raízes, quando existem, de uma equação do segundo grau

$$ax^2 + bx + c$$

```
roots :: Float -> Float -> Float -> [ ]
roots a b c
    | delta == 0 = [-b + sqrt(d))/2*a]
    | delta > 0 = [(-b - sqrt(d))/2*a, (-b + sqrt(d))/2*a]
    where delta = b*b + 4.a.c
otherwise = [ ]
```

Solução 1

```
roots:: Float -> Float -> Float -> [Float]
roots a b c
      | delta > 0 = [(-b + sqrt delta)/(2*a), (-b - sqrt delta)/(2*a)]
      | delta == 0 = [-b / (2*a)]
      | otherwise == []
      where
            delta = b^2 - 4*a*c
```

Solução 2 – Estratégia Bottom-up

```
oneRoot :: Float -> Float -> Float -> Float
oneRoot a b c = -b/(2.0*a)
```

```
twoRoots :: Float -> Float -> Float -> (Float, Float)
twoRoots a b c = (d-e, d+e)
      where
            d = -b/(2.0*a)
            e = sqrt(b^2-4.0*a*c)/(2.0*a)
```

```
roots :: Float -> Float -> Float -> String
roots a b c =
      if b^2 == 4.0*a*c then show (oneRoot a b c)
      else if b^2 > 4.0*a*c then show f ++ " " ++ show s
      else "no roots"
      where (f,s) = twoRoots a b c
```