

Paradigma Funcional

Fundamentos e Primeiras Linguagens Funcionais



ERICK GALANI MAZIERO
erick.maziero@ufla.br

Departamento de Ciências da Computação
Universidade Federal da Lavras



Introdução

O paradigma funcional trata a computação como avaliação de funções matemáticas.

Esse estilo de programação é suportado por linguagens de programação funcional, ou linguagens aplicativas.

Linguagens funcionais possuem alto nível de abstração e estilo declarativo: especifica-se o que deve ser computado ao invés de como.

Alguns exemplos de linguagens funcionais: LISP, Scheme, ML e Haskell.



Exemplos de linguagens funcionais





Funções matemáticas

Uma função matemática é um mapeamento de membros de um conjunto, chamado de conjunto **domínio**, para outro, chamado de conjunto **imagem**.

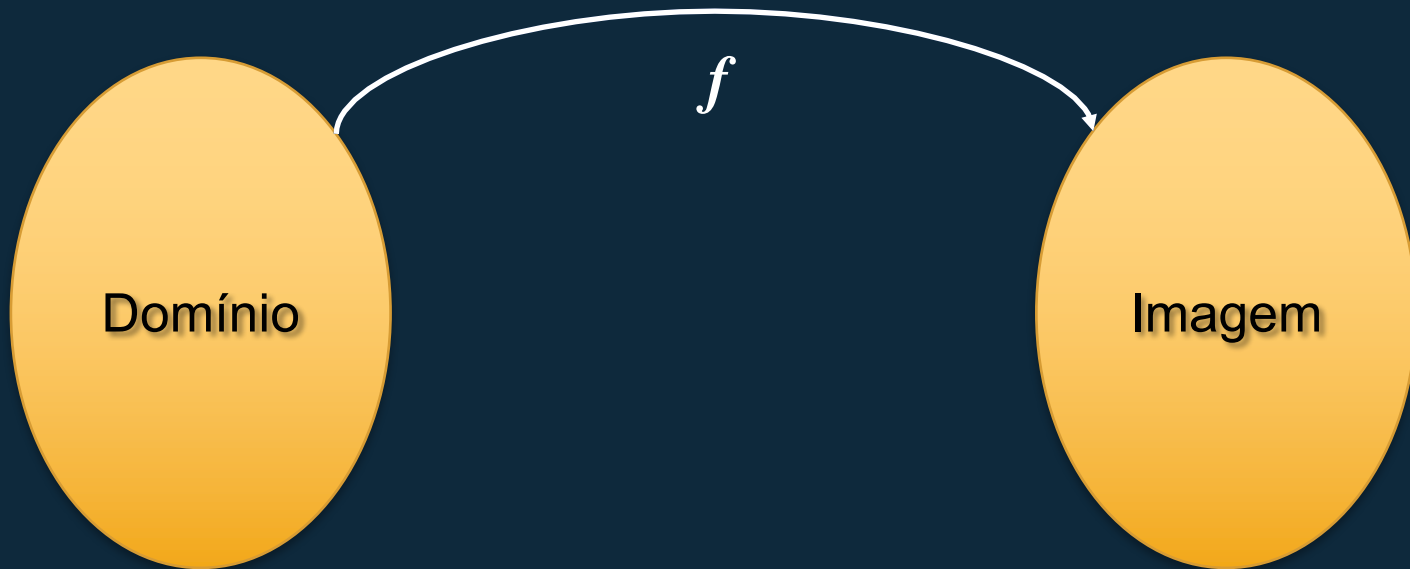
As funções são geralmente aplicadas a um elemento em particular do conjunto domínio, fornecido como um **parâmetro** para a função.

Uma função leva a, ou retorna, um elemento do conjunto imagem.

Em funções matemáticas, a ordem de avaliação de suas expressões de mapeamento é controlada por **recursão e expressões condicionais**, e não por sequência e repetição iterativa, como nas linguagens imperativas.

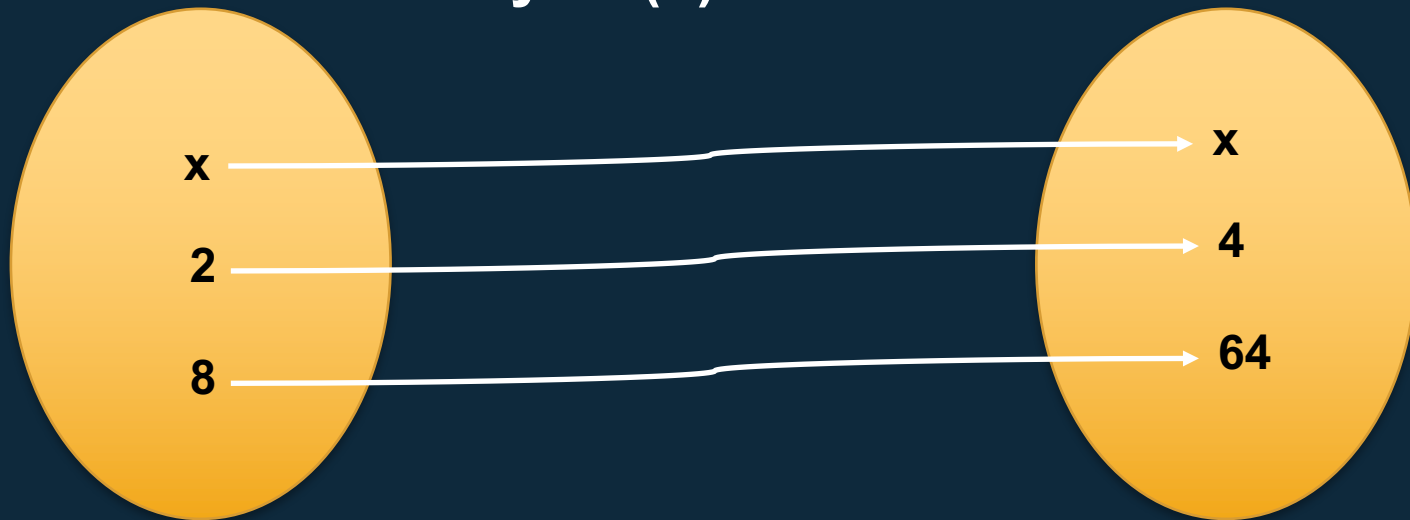


Funções Matemáticas



Funções Matemáticas

$$y = f(x) = x^2$$





Fundamentos da Programação Funcional - I

O objetivo do projeto de uma linguagem de programação funcional é **mimetizar funções matemáticas ao máximo possível**.

Em uma linguagem imperativa, uma expressão é avaliada e o resultado é armazenado em uma **posição de memória**, representada como uma variável em um programa.

Uma linguagem de **programação puramente funcional não usa variáveis**, nem sentenças de atribuição. Sem variáveis, as construções de iteração não são possíveis, já que elas são controladas por variáveis.





Fundamentos da Programação Funcional - II

Na programação funcional, as repetições devem ser especificadas com recursão em vez de estruturas de repetição.

Uma linguagem funcional fornece:

- um conjunto de **funções primitivas**;
- um conjunto de **formas funcionais** para construir funções complexas a partir das funções primitivas;
- uma **operação de aplicação de função**;
- alguma estrutura ou **estruturas para representar dados**.





Transparência Referencial

Programa funcional não tem 'estado'

Não tem atribuição: o programador não precisa se preocupar com variáveis

Dada uma função, podemos substituí-la por seu valor de retorno sem causar impacto na aplicação





Transparência Referencial

O resultado de uma função é determinado unicamente por seus valores de entrada. Coisa alguma fora da função pode afetar a sua saída.

Isso é não tem efeito colateral!!





Uso da recursão

Principal causa da perda de performance, pois é computacionalmente caro realizar a recursão.

Se recursão for *de cauda* interpretador por mudar para iteração.

Pesquise o que significa recursão de cauda

Indicado é sempre tentar **recursão de cauda**





Funções Simples

Exemplo de função que calcula o cubo de x . x pertence ao domínio dos números reais

◇ $\text{cube}(x) \equiv x * x * x, \quad x \in \mathbb{R}$

◇ $\text{cube}: \mathbb{R} \rightarrow \mathbb{R}$

O resultado da função `cube` depende apenas de x . Nenhum outro valor interfere no resultado: não tem efeito colateral

◇ O símbolo \equiv significa *é definida como*






Funções Lambda

Alonzo Church, 1941, especificou funções não nomeadas

São funções que, geralmente, são utilizadas num escopo menor e, portanto, não precisam de um nome para ser referenciada em contexto mais amplo. λ é uma letra grega, nomeada lambda.

$$\lambda(x)x * x * x$$


$$(\lambda(x)x * x * x)(2)$$

Resulta em 8





Formas funcionais

Nem tudo se resolve com funções simples, como a função cubo. Então, as linguagens funcionais permitem as funções de ordem superior. Exemplos:

- ◇ Composição de funções
- ◇ Aplicar-a-todos





Composição de funções

$$h \equiv f \circ g$$

Na composição de funções, duas ou mais funções simples, são compostas para formar uma função mais complexa

$$f(x) \equiv x + 2$$

$$g(x) \equiv 3 * x$$

$$h(x) \equiv f(g(x))$$

$$h(x) \equiv (3 * x) + 2$$






Aplicar a todos

Denotada como α recebe uma única função como parâmetro e uma lista de argumentos

A função f é aplicada a todos argumentos da lista



$$f(x) \equiv x * x$$
$$\alpha(f, (2,3,4))$$

Resulta em (4, 9, 16)





A primeira linguagem funcional

- ◇ LISP está para Fortran, como Funcional está para Imperativo, no histórico das linguagens de programação
- ◇ Mas, inclui recursos imperativos: variáveis e, portanto, não pode ser considerada uma linguagem puramente funcional





LISP: Tipos e Estruturas de dados

- ◇ Em LISP, os dados podem ser:
 - ◇ Átomos, ou
 - ◇ Listas
 - Desprovidos de tipos
- ◇ Exemplos de Listas em LISP
 - (A B C D)
 - (A (B C) D (E (F G)))





LISP: Primeiro interpretador

- ◇ LISP usa a Notação-M que foi definida para a linguagem FORTRAN
 - Notação-M para linguagem de máquina (máquina IBM 704)
- ◇ Função EVAL é o Interpretador LISP, definido em 1965
 - Tem grande relação com o estudo da computabilidade, que vocês estudarão na disciplina Teoria da Computação





LISP: Primeiro interpretador

- ◇ Convenção Cambridge Polonesa definia a seguinte sintaxe para os comandos LISP
 - `(nome_funcao param_1 ... param_n)`
 - `(+ 5 7)`
 - `(nome_funcao (LAMBDA(arg_1, ..., arg_n) expressao))`





Scheme

- ◇ É um dialeto de LISP, surgiu no MIT em 1970, com as seguintes vantagens:
 - Tamanho pequeno
 - Escopo estático
 - Sintaxe simples
 - Bom para fins didáticos





Scheme: Interpretador

- ◇ Laço infinito de leitura-avaliação-escrita: Lê o comando, faz a sua avaliação (execução) e retorna o resultado
- ◇ Isso é feito pela função EVAL
 1. Cada expressão de parâmetro é avaliada
 2. Função primitiva é aplicada aos parâmetros
 3. Valor resultante é mostrado





Scheme: Funções Numéricas Primitivas

- ◇ $+$, $-$, $/$, e $*$ são os operadores aritméticos
- ◇ $*$ \rightarrow 1 (se inserir apenas o operador $*$, Scheme retorna 1)
- ◇ $+$ \rightarrow 0 (se inserir apenas o $+$, retorna 0, porque?)
- ◇ Considere os seguintes comandos e retornos:
 - $(* 3 7)$ retorna 21
 - $(- 15 7 2)$ retorna 6
 - $(-24 (* 4 3))$ retorna 12





Scheme: definição de funções

- ◇ Baseado na notação LAMBDA
- ◇ $(\text{LAMBDA}(x)(* \ x \ x))$
- ◇ $((\text{LAMBDA}(x)(* \ x \ x))7)$
 - Resulta em 49
 - x é variável vinculada e nunca muda na expressão





Scheme: definição de funções

- ◇ DEFINE é utilizado para vincular nome a um valor ou expressão
 - Não define variável!
 - Pode criar constante
- ◇ `(DEFINE símbolo expressão)`
- ◇ `(DEFINE pi 3.14159)`
- ◇ `(DEFINE two_pi (* 2 pi))`





Scheme: Função de saída

- ◇ (DISPLAY expressão)
 - Como o print do Python
- ◇ (NEWLINE)






Scheme: Predicado numérico

Retorna valor booleano:

◇ #T

◇ #F

Função	Significado
=	Igual
<>	Diferente
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
EVEN?	É número par?
ODD?	É número ímpar?
ZERO?	É zero?



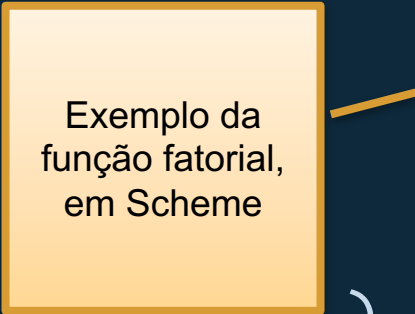


Scheme: Controle de fluxo

(IF predicado expressão_então
expressão_senão)

```
(DEFINE (factorial n)
  (IF (= n 0)
      1
      (* n (factorial(- n 1)))
  )
)
```

Exemplo da
função fatorial,
em Scheme





Scheme: Funções de lista


Uso principal das primeiras linguagens funcionais foi o processamento de listas

Então a linguagem tem *seletores* para listas:

- ◇ CAR: retorna primeiro elemento da lista
- ◇ CDR: retorna a lista menos o primeiro elemento
- ◇

```
(DEFINE (second lst) (CAR (CDR lst)))
```

Retorna o segundo elemento da lista `lst`.
Consegue ver isso?



Scheme: QUOTE

- ◇ Evita que um parâmetro seja avaliado
- ◇ (QUOTE A)
 - `A
- ◇ (QUOTE (A B C))
 - `(A B C)

Retorna A, sem
saber o que A
seja





Scheme: Funções de lista

CONS é um construtor de listas. Constrói a partir de dois argumentos

Exemplo apenas ilustrativo que constrói a lista lst a partir do primeiro elemento dela (CAR) e do restante (CDR).

Geralmente insere o primeiro argumento como CAR do segundo argumento, geralmente uma lista (CDR)

—— (CONS (CAR lst) (CDR lst))





Scheme: Funções de PREDICADO

◇ (EQ? `A `A)

- Retorna #T

◇ (EQ? `A `B)

- Retorna #F

◇ (LIST? `(X,Y))

- Retorna #T

◇ (LIST? `X)

- Retorna #F

◇ (NULL? `(A,B))

- Retorna #F

◇ (NULL? `())

- Retorna #T



Scheme: Outros Exemplos

◇ (member `B (A B C))

```
(DEFINE (member atm lst)
  (COND
    ((NULL? lst) #F)
    ((EQ? atm (CAR lst)) #T)
    (ELSE (member atm (CDR lst))))
)
```

Define a função
member, que
verifica se atm
pertence à lista
lst



Scheme: Outros Exemplos

- ◇ `(append '(A B) '(C D E))`
 - Retorna a lista `(A B C D E)`

```
(DEFINE (append lst1 lst2)
  (COND
    ((NULL? lst1) lst2)
    (ELSE (CONS (CAR lst1)
                  (append(CDR lst1) lst2))))
)
```

Define a função
append, que
concatena as
listas lst1 e
lst2






Scheme: Recursão de cauda

```
(DEFINE (member atm lst)
  (COND
    ((NULL? lst) #F)
    ((EQ? atm (CAR lst)) #T)
    (ELSE (member atm (CDR lst))))
  )
)
```

Função member
com recursão
de cauda: a
recursão é
sempre o último
comando da
função recursiva



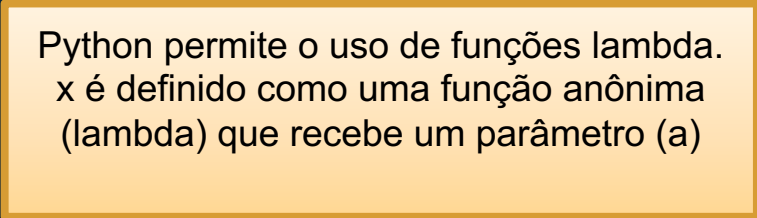


Linguagens imperativas com Funcionais


Diversas linguagens imperativas têm construções funcionais, principalmente para o tratamento de listas

Exemplo em Python:

```
x = lambda a : a + 10  
print(x(5))
```



Python permite o uso de funções lambda. x é definido como uma função anônima (lambda) que recebe um parâmetro (a)



Linguagens imperativas com Funcionais

```
def myfunc(n):  
    return lambda a : a * n
```

A função myfunc recebe um parâmetro (n) e retorna n vezes o parâmetro da função que for definida com myfunc.

Veja com atenção o exemplo! Interessante, não!?

```
mydoubler = myfunc(2)  
mytripler = myfunc(3)
```

```
print(mydoubler(11))  
print(mytripler(11))
```





Referência Bibliográfica

Sebesta, R. W. (2011). *Conceitos de Linguagens de Programação*. 9 ed. Bookman.

Capítulo 15

