Linguagens de Programação Funcionais

Prof. Lucas Ismaily

Sumário

- Introdução
- Funções matemáticas
- Fundamentos de linguagens de programação funcionais
- Programação funcional em Python
  - Familiarizar com o paradigma
- Outras linguagens funcionais
  - Primeira funcional: LISP
  - Scheme
  - COMMON LISP
  - ML
  - Haskell
- Aplicações de linguagens funcionais
- Comparação entre linguagens imperativas e funcionais

Introdução

- O projeto das linguagens imperativas é baseado diretamente na arquitetura de von Neumann
  - Eficiência é a principal preocupação, ao invés da facilidade de desenvolver software

Introdução

- O projeto das linguagens funcionais é baseado em funções matemáticas
  - Já estamos acostumados com representações matemáticas
  - Não se preocupa com a arquitetura da máquina que executará o programa
  - Base teórica sólida: mais fácil de provar corretude
  - Não produz efeitos colaterais

Funções Matemáticas

- Def: Uma função matemática é um mapeamento de elementos de um conjunto (chamado domínio) em elementos de outro conjunto (chamado imagem)
- Uma expressão lambda especifica os parâmetros e o mapeamento de uma função

  Ex.: $\lambda(x)$: x * x * x

  para a função *cubo(x) = x * x * x*

Funções Matemáticas

- Expressões Lambda descrevem funções sem nome
- Avaliação de expressões Lambda

  ex.: $(\lambda(x)\ x * x * x)(3)$

  é avaliado como 27

Funções Matemáticas

- Formas funcionais
  - Função que recebe funções como parâmetro, ou retorna função, ou recebe e retorna

Formas Funcionais

1. Composição de Funções
   - Forma funcional que recebe duas funções, e o resultado é a aplicação da primeira função sobre o resultado da segunda função

   Forma: $h \equiv f \circ g$

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

   Ex.: Para $f(x) \equiv x * x * x$ e $g(x) \equiv x + 3$,

   $h \equiv f \circ g$ vale $(x + 3) * (x + 3) * (x + 3)$

Formas Funcionais

2. Construção
   - Forma funcional que recebe uma lista de funções como parâmetro e produz uma lista de resultados da aplicação de cada função a um dado parâmetro

   Forma: [f, g]

   Ex.: Para f (x) ≡ x * x * x e g (x) ≡ x + 3,

   [f, g] (4) resulta em (64, 7)

Formas Funcionais

3. “Aplicada a todos”
   – Forma funcional que recebe uma função f e uma lista L, e retorna a função f aplicada a cada elemento de L

   Forma: α

   Ex.: Para h (x) $\equiv$ x * x * x

   α(h, (3, 2, 4)) vale (27, 8, 64)

Fundamentos de Linguagens de Programação Funcionais

- O objetivo do projeto de LPFs é imitar as funções matemáticas o máximo possível
- O processo de computação é fundamentalmente diferente das linguagens imperativas
  - Na imperativa, os resultados das operações são armazenados em variáveis para uso posterior
  - O gerenciamento de variáveis é uma preocupação constante e fonte de complexidade para a programação imperativa
- Em LPFs, variáveis não são necessárias, como ocorre na matemática

Fundamentos de Linguagens de Programação Funcionais

- Em LPFs, a avaliação de uma função sempre produz o mesmo resultado quando passamos os mesmos parâmetros
  - Isto se chama transparência referencial

Características de LPFs

- Tudo que pode ser feito com “dados” pode ser feito com funções
  - Ex.: parâmetro e retorno de função, atribuição
- Não utiliza estruturas de controle
  - Ex.: seleção feita com avaliação curto-circuito
  - Loops são implementados com recursão
- Foco no processamento de listas
- “Puramente” funcionais evitam efeitos colaterais

Características de LPFs

- Evita ou proibe o uso de instruções
  - Computação é feita por avaliação de expressões
  - Neste caso, o programa é uma expressão
- O programador se preocupa mais em “o que” será computado, do que em “como” será
- Utiliza com frequência formas funcionais: funções que operam sobre outras funções

Vantagens da Programação Funcional

- Sintaxe e semântica mais simples
- Maior expressividade para aplicações matemáticas
- Programação mais rápida
- Código menor
- Menor chance de cometer erros
- Mais fácil de provar corretude

- Preço: eficiência

Programação Funcional em Python

- Expressões condicionais podem ser imitadas com a avaliação "curto-circuito"

```
if   <c1>: f1()
elif <c2>: f2()
elif <c2>: f3()
else:      f4()
```

É o mesmo que...

```
(<c1> and f1()) or (<c2> and f2()) or (<c3> and f3()) or (f4())
```

Programação Funcional em Python

- Expressões condicionais podem ser imitadas com a avaliação “curto-circuito”

```
>>> a=10
>>> b=20
>>> a and b
20
>>> a=0
>>> a and b
0
```

Programação Funcional em Python

- Expressões Lambda

```
>>> cubo = lambda x: x*x*x
>>> cubo(5)
125
```

Programação Funcional em Python

- Forma funcional "composição"

```
>>> f = lambda x: x*x*x
>>> g = lambda x: x+3
>>> comp = lambda x: f(g(x))
>>> comp(2)
125
```

Programação Funcional em Python

- Forma funcional “aplicada a todos”

```
>>> cubo = lambda x: x*x*x
>>> map(cubo, [1,2,3])
[1, 8, 27]
```

Programação Funcional em Python

- Forma funcional “aplicada a todos”
  - Eliminando o “for”...
    (“while” removido com recursão)

```
>>> L = [3,1,5,2]
>>> for e in L: cubo(e)
27
1
125
8
>>> map(cubo,L)
[27, 1, 125, 8]
```

Programação Funcional em Python

- Forma funcional “reduce”

```
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> soma = lambda n: reduce(lambda x,y: x+y, range(1,n+1))
>>> soma(5)
15
>>> fat  = lambda n: reduce(lambda x,y: x*y, range(1,n+1))
>>> fat(5)
120
```

Programação Funcional em Python

- Forma funcional “filter”

```
>>> par = lambda n: filter(lambda x: x%2==0, range(1,n+1))
>>> par(10)
[2, 4, 6, 8, 10]
```

Programação Funcional em Python

- Definição de listas
  - Permite criar relações

```
>>> [(x,y) for x in (1,2,3,4) for y in (10,15,3,22) if x*y > 25]
[(2, 15),(2, 22),(3, 10),(3, 15),(3, 22),(4, 10),(4, 15),(4, 22)]
```

Programação Funcional em Python

- Definição de listas
  - Permite criar relações

```
>>> [(x,y) for x in (1,2,3,4) for y in (10,15,3,22) if x*y > 25]
[(2, 15),(2, 22),(3, 10),(3, 15),(3, 22),(4, 10),(4, 15),(4, 22)]
```

Código não funcional: permite vários efeitos colaterais..

```
xs = (1,2,3,4)
ys = (10,15,3,22)
bigmuls = []
# ... mais código ...
for x in xs:
    for y in ys:
       # ... mais código ...
        if x*y > 25:
            bigmuls.append((x,y))
            # ... mais código ...
# ... mais código ...
print bigmuls
```

Programação Funcional em Python

- Definição de listas
  - Permite a forma funcional “construção”

```
>>> f = lambda x: x*x*x
>>> g = lambda x: x+3
>>> cons = lambda x: [h(x) for h in (f,g)]
>>> cons(4)
[64, 7]
```

Programação Funcional em Python

- Funções sequenciais

```
>>> executa = lambda f: f()
>>> f1 = lambda: 'texto1'
>>> f2 = lambda: 'texto2'
>>> f3 = lambda: 'texto3'
>>> map(executa, (f1,f2,f3))
['texto1', 'texto2', 'texto3']
```

Programação Funcional em Python

- Exemplo: encontrar um elemento em uma lista

```
>>> lista = ['joao', 'maria', 'jose', 'pedro']
>>> len(lista)
4
>>> lista[0]
'joao'
>>> lista[1:]
['maria', 'jose', 'pedro']

>>> pertence = lambda x,L:
                len(L)>0 and (x==L[0] or pertence(x,L[1:]))
>>> pertence('jose', lista)
True
>>> pertence('paulo', lista)
False
```

Programação Funcional em Python

- Exemplo: determinar se N é primo

```
>>> primo = lambda n: len(
        filter(lambda x: n%x==0, range(2,n))
    ) == 0
>>> filter(primo, range(2,20))
[2, 3, 5, 7, 11, 13, 17, 19]
```

Programação Funcional em Python

- Exemplo: N-ésimo termo da série de Fibonacci

```
>>> fib = lambda n: (n<3 and 1) or
  (fib(n-1)+fib(n-2))
>>> map(fib, range(1,10))
[1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Exercício

- Função que recebe duas listas L1 e L2 e retorna os elementos que pertencem a L1 e a L2 (interseção das listas)
  - Assuma que as listas não tem elementos repetidos
- Função que determina se string é palíndrome
  - Utilize operador de sub-faixa
- Função para ordenar lista

LISP

- LISP: “LISt Processing”
- Tipos de dados: listas e átomos (elementos)
  - Listas armazenadas como lista encadeada simples
  - Forma: átomos e sublistas entre parênteses

    ex.: (A B (C D) E)
- Primeiro interpretador foi apresentado apenas para demonstrar a generalidade computacional da notação

LISP

- Notação Lambda é usada para especificar funções
- Aplicação de função e dados tem a mesma forma!
  - ex.: se (A B C) é interpretada como dados, A, B e C são átomos
  - se é interpretada como aplicação de função, então A é uma função aplicada sobre os parâmetros B e C

Scheme

- Um dialeto do LISP desenvolvido na década de 70
  - Projetada para ser mais simples que os dialetos disponíveis
- Utiliza apenas escopo estático
- Funções são tratadas como dados
  - Podem ser o resultado de uma expressão e elemento de lista
  - Podem ser atribuídas à variáveis e passadas como parâmetro

COMMON LISP

- Combinação de vários recursos dos dialetos do LISP disponíveis no início da década de 80
- Linguagem grande e complexa: o oposto da Scheme

COMMON LISP

- Inclui:
  - Registros
  - Arrays
  - Números complexos
  - Strings
  - Recursos de I/O poderosos
  - Pacotes com controle de acesso
  - Recursos de LP imperativas
  - Estruturas de repetição

COMMON LISP

- Exemplo:

```
(DEFUN iterative_member (atm lst)
 (PROG ()
  loop_1
    (COND
     ((NULL lst) (RETURN NIL))
     ((EQUAL atm (CAR lst))(RETURN T))
          )
  (SETQ lst (CDR lst))
  (GO loop_1)
))
```

ML

- Linguagem com escopo estático e sintaxe que se parece mais com Pascal do que com LISP
- É fortemente tipada e não faz conversão automática (coerção)
- Permite inferência de tipo para variáveis não declaradas
- Inclui tratamento de excessões

ML

- Inclui listas e operações sobre listas
- Declaração de função:

`fun` function_name (formal_parameters) =
function_body_expression;

ex.:

```
fun cube (x : int) = x * x * x;
```

Haskell

- Similar à ML: sintaxe, escopo estático, fortemente tipada, inferência de tipos
- Diferente de ML e da maioria das LPs funcionais por ser puramente funcional:
  - Sem variáveis
  - Sem instrução de atribuição
  - Sem efeitos colaterais

Haskell

- Características mais importantes:
  - Utiliza avaliação preguiçosa: avalia sub-expressões apenas se necessário
  - Oferece “definição de listas”, inclusive listas infinitas

Exemplos em Haskell

1. Número de Fibonacci (ilustra definição de função)

```
fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1)
            + fib n
```

Exemplos em Haskell

2. Fatorial (ilustra condições)

```
fact n
  | n == 0 = 1
  | n > 0 = n * fact (n - 1)
```

A palavra reservada **`otherwise`** pode ser utilizada como condição

Exemplos em Haskell

3. Operações com listas
   - Notação: elementos entre colchetes

   ex.: **`directions = [“north”, “south”, “east”, “west”]`**
   - Tamanho: #

   e.g., **`#directions is 4`**
   - Séries aritméticas com o operador ..

   ex.: **`[2, 4..10] vale [2, 4, 6, 8, 10]`**

Exemplos em Haskell

3. Operações com listas (continuação)
   - Concatenação com ++

   Ex.: `[1, 3] ++ [5, 7]` resulta em

   `[1, 3, 5, 7]`

   - "Primeiro elemento" e "elementos restantes" com operador : (como no Prolog)

   Ex.: `1:[3, 5, 7]` resulta em `[1, 3, 5, 7]`

Exemplos em Haskelles

```
product [] = 1
product (a:x) = a * product x

fact n = product [1..n]
```

Exemplos em Haskell

4. Definição de lista: notação de conjuntos

   ex.: `[n * n | n ← [1..20]]`

   Define uma lista dos quadrados dos primeiros 20 inteiros positivos

```
factors n = [i | i ← [1..n div 2],
                n mod i == 0]
```

Produz todos os divisores de n

Exemplos em Haskell

- Quicksort:

```
sort [] = []
sort (a:x) =
    sort [b | b ← x; b <= a]
    ++ [a] ++
    sort [b | b ← x; b > a]
```

Exemplos em Haskell

5. Avaliação preguiçosa

ex.:

```
positives = [0..]
squares = [n * n | n ← [0..]]
```

(computa apenas aqueles que são necessários)

Ex.: **`member squares 16`**

retornará True

Exemplos em Haskell

- Função member pode ser escrita como:

```
member [] b = False
member(a:x) b =     (a == b) || member x b
```

- Porém, gera loop infinito se b não é quadrado perfeito!
- Melhor fazer assim (assume lista em ordem crescente):

```
member2 (m:x) n
        | m < n= member2 x n
        | m == n     = True
        | otherwise = False
```

Aplicações das Linguagens Funcionais

- LISP é a LPF que teve mais sucesso
  - Amplamente empregada em aplicações de IA:
    - representação do conhecimento
    - aprendizedo de máquina
    - processamento de linguagem natural,
    - modelar fala e visão
  - Ex. fora da IA: editor EMACS e o sistema MACSYMA (para matemática simbólica)
- APL é usada para programação descartável
  - Rápido de escrever, difícil de ler e manter
  - Oferece recursos poderosos para operar matrizes
- Scheme é muito utilizada em universidade para ensinar introdução à programação

Comparação entre as Linguagens Imperativas e Funcionais

- Imperativas:
  - Execução eficiente
  - Sintaxe e semântica complexa
  - Concorrência é projetada pelo programador
- Funcionais:
  - Execução ineficiente
  - Sintaxe e semântica simples
  - Programas podem ser paralelizados automaticamente