Lambda Cálculo e Programação Funcional

Programação Funcional
Bacharelado em Sistemas de Informação
Maio - 2009

Alonzo Church (1903–1995)

Professor em Princeton, EUA (1929–1967) e UCLA (1967–1990)

Inventou um sistema formal, chamado ***λ-calculus*** (Lambda Cálculo) , e definiu a noção de função computável utilizando este sistema.

O Lambda Cálculo pode ser chamado "a menor linguagem de programação universal" do mundo.

As linguagens de programação funcionais, como Lisp, Miranda, ML, Haskell são baseadas no Lambda Cálculo.

Lambda Cálculo

O lambda cálculo pode ser visto como uma linguagem de programação abstrata em que funções podem ser combinadas para formar outras funções, de uma forma *pura.*

O lambda cálculo trata funções como ***cidadãos de primeira classe***, isto é, entidades que podem, como um dado qualquer, ser utilizadas como argumentos e retornadas como valores de outras funções.

Sintaxe

Uma expressão simples do lambda cálculo:

```
(+ 4 5)
```

Todas as aplicações de funções no lambda cálculo são escritas no formato prefixo.

Avaliando a expressão: `(+ 4 5) = 9`

A avaliação da expressão procede por redução:

```
(+ (* 5 6)(* 4 3)) → (+ 30 (* 4 3))
                   → (+ 30 12) →  42
```

Sintaxe

Uma *abstração lambda* é um tipo de expressão que denota uma função:

```
(λx. + x 1)
```

O λ determina que existe uma função, e é imediatamente seguido por uma variável, denominada *parâmetro formal* da função.

```
(λ      x  .  +      x  1)
```

A função de	x	que	incrementa	x	de	1

Sintaxe

Uma expressão lambda deve ter a forma:

```
<exp> ::=    <constante>
           | <variavel>
           | <exp> <exp>
           | λ.<variavel> <exp>
```

Os parênteses podem ser utilizados nas expressões para prevenir ambiguidades.

Exemplos

(λx. x) y
(λx. f x)
x y
(λx. x) (λx. x)
(λx. x y) z
(λx y. x) t f
(λx y z. z x y) a b (λx y. x)
(λf g. f g) (λx. x) (λx. x) z
(λx y. x y) y
(λx y. x y) (λx. x) (λx. x)
(λx y. x y) ((λx. x) (λx. x))

Análise de Expressões Lambda

A expressão Lambda se estende para a direita

$\lambda f.\ x\ y \quad \equiv \quad \lambda f.(x\ y)$

A aplicação é associativa à esquerda

$x\ y\ z \quad \equiv \quad (x\ y)\ z$

Múltiplos lambdas podem ser omitidos

$\lambda f\ g.\ x \quad \equiv \quad \lambda f\ .\ \lambda g.\ x$

Variáveis Livres e Ligadas

A variável x é **ligada** por λ na expressão: λ x.e

Caso a variável não seja ligada, então é considerada **livre**.

As variáveis livres de um termo são definidas como:

$$FV(x) = \{ x \}$$

$$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$$

$$FV(\lambda\ x\ .\ e) = FV(e) - \{ x \}$$

Para a expressão abaixo, y é ligada e x é uma variável livre.

λ y . x y

Variáveis Livres e Ligadas

(λx. + x y) 4

Para avaliar completamente a expressão, é necessário conhecer o valor global de y. Para x, a variável é local e pertence à função.

Exemplos:

```
λx. + ((λy. + y z) 7) x      // z é livre
+ x ((λx. + x 1) 4)          // o primeiro x é livre
```

Redução

A aplicação de um argumento à uma abstração lambda implica na substituição das ocorrências das variáveis correspondentes ao argumento:

$$(\lambda x.\ +\ x\ 1)\ 4 \rightarrow +\ 4\ 1$$

Esta operação é denominada **β-redução.**

Funções também podem ser passadas como argumento:

```
(λ f. f 3) (λ x. + x 1)    → (λ x. + x 1) 3
                           → + 3 1
                           → 4
```

β-redução

```
(λx. λy. + (- x 1)) x 3) 9
      → (λx. + (- x 1)) 9 3
      → + (- 9 1) 3
      → + 8 3
      → 11
```

```
(λx. λy. + x ((λx.- x 3) y)) 5 6
```

??

Conversão

Considere as duas abstrações lambda:

```
(λx. + x 1)
(λy. + y 1)
```

Claramente as duas abstrações acima são equivalentes e uma **α-conversão** nos permite mudar o nome do parâmetro formal de uma abstração lambda.

Então:

$$(\lambda x.\ +\ x\ 1) \underset{\alpha}{\leftrightarrow} (\lambda y.\ +\ y\ 1)$$

Conversão

Considere as duas expressões:

```
(λx. + 1 x)
(+  1)
```

Ambas tem o mesmo comportamento quando aplicadas à argumento: adicionam 1 ao argumento.

Uma **η–redução** é uma regra expressando tal equivalência:

$$(\lambda x.\ +\ 1\ x\ ) \underset{\eta}{\leftrightarrow} (+\ 1)$$

η–redução (Equivalência)

```
(λx. + 1 x)
(+  1)
```

Em Haskell:

```
Main> (\x → (+) 1 x) 4
5
Main> ( (+) 1) 4
5
```

Resumo

α–conversão (renomeação)	λ x . e	↔	λ y . e	*Quando y não é uma variável livre em e.*
β–redução (aplicação)	(λ x . e1) e2	→	[e2/x] e1	*Operações de renomeação e substituição.*
η–conversão	λ x . e x	→	e	*Elimina uma abstração λ mais complexa.*

Exemplos de funções

Exemplo: Área de um círculo

$F \equiv \pi * r^2, \quad r \in N$

$Q \equiv x * x, \quad x \in N$

$H = F \circ Q$ (função composta)

$H(r) = 3{,}14 * Q(r)$ (Fortran)

Abstração Lambda: $\lambda\, y.\, \pi * y * y$

Em Haskell:

```
Main> (\y -> pi * y * y) 3
28.2743338823081
```

Exemplos de funções

```
int f(x) {
    return x+2;
}
f(5);
```

$(\lambda f.\ f(5))\ (\lambda x.\ x+2)$

Bloco principal — Função declarada

Exemplos de funções

Dada uma função f, defina uma expressão $f \circ f$:

$$\lambda f.\ \lambda x.\ f\ (f\ x)$$

Aplicação para f(x) = x+1:

$(\lambda f.\ \lambda x.\ f\ (f\ x))\ \ (\lambda y.\ y+1)$

$= \lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ \ x)$

$= \lambda x.\ (\lambda y.\ y+1)\ (x+1)$

$= \lambda x.\ (x+1)+1$

Tuplas como funções

Em Haskell:

```
t      = \x -> \y -> x
f      = \x -> \y -> y
pair   = \x -> \y -> \z -> z x y
first  = \p -> p t
second = \p -> p f
```

```
Main> first (pair 1 2)
1
Main> first (second (pair 1 (pair 2 3)))
2
```

Exercícios

1. Calcular a média das notas dos alunos de um curso:

```
type Aluno = (Int, String, Float)
             -- N° Aluno, Nome, Nota
type Curso = [Aluno]

listaAlunos :: Curso
listaAlunos = [(1234, "Jose Azevedo", 13.2),
               (2345, "Carlos Silva", 9.7),
               (3456, "Rosa Mota", 17.9)]

media :: Curso->Float
media l = (/) (sum (map (\(_,_,n)->n) l))
              (fromIntegral (length l))
```

Exercícios

```
Main> media listaAlunos
13.6
```

Funções utilizadas:

`fromIntegral` : função que transforma o argumento em um número da classe Fractional

Exemplo:

```
divisao a b = (fromIntegral a) / (fromIntegral b)
```

`map` : função que aplica uma função a todos os elementos de uma lista.

Exemplo:

```
Main> dobro 3
9
```

Exercícios

```
Main> map dobro [1,2,3]
[1,4,9]
Main> map (\(_,n) -> n) [(1,2),(3,4)]
[2,4]

sum : função que soma os elementos de uma lista.
Main> sum [1,2,3]
6
Main> sum ['a','b','c']
ERROR - Cannot infer instance
*** Instance   : Num Char
*** Expression : sum ['a','b','c']

Main> sum [1.2,3.3]
4.5
```

Exercícios

2. Implementar uma função que calcula o sucessor de um número inteiro usando expressão lambda ($\lambda x.\ x+1$). Em seguida, definir uma função *duasVezes* para aplicar uma função nela mesma. Finalmente, construir uma função para mapear a aplicação de *duasVezes* sobre uma lista de inteiros.

```
sucessor::(Int -> Int)
sucessor = \x -> x + 1

Main> sucessor 1
2
Main> sucessor 5
6
Main> sucessor 'a'
ERROR - Type error in application
```

Exercícios

Função *duasVezes* para aplicar uma função nela mesma:

```
sucessor::(Int -> Int)
sucessor = \x -> x + 1

duasVezes :: (a->a) ->a ->a
duasVezes f x = f (f x)


Main> duasVezes sucessor 5
7
Main> duasVezes sucessor 100
102
Main> duasVezes sucessor 5.6
ERROR - Cannot infer instance
```

Exercícios

Função *mapear* para aplicar uma função à uma lista de Valores (sem usar a função **map**)

```
sucessor::(Int -> Int)
sucessor = \x -> x + 1

duasVezes :: (a->a) ->a ->a
duasVezes f x = f (f x)

mapear::(a->b)->[a]->[b]
mapear f [] = []
mapear f (x:xs) = (f x): mapear f xs
```

Exercícios

```
Main> mapear sucessor [1,3,5]
[2,4,6]

Main> mapear (duasVezes sucessor) [1,3,5]
[3,5,7]

Main> mapear (duasVezes (+ 1)) [1,3,5]
[3,5,7]

Main> mapear (duasVezes (\x -> x * x)) [3,4,5]
[81,256,625]

Main> mapear (\y -> y ++ y) ["na","ta","la"]
["nana","tata","lala"]

Main> mapear (duasVezes (\y -> y ++ y)) ["na","ta","la"]
["nananana","tatatata","lalalala"]
```

Exercícios

3. Dados um elemento e uma lista, intercale o elemento na lista, em todas as posições.

```
intercala :: a -> [a] -> [[a]]
intercala x [] = [[x]]
intercala x (l:ls) =  [x:l:ls] ++
                      map (l:) (intercala x ls)
```

```
Main> intercala 1 [2,3]
[[1,2,3],[2,1,3],[2,3,1]]
```

Exercícios

4. Reduza as expressões avaliando-as e escreva o código correspondente em Haskell, testando no ambiente Hugs.

```
1) (λ x.2*x + 1) 3
2) (λ xy.x+y) 5 7
3) (λ yx.x+y) 5 7
4) (λ xy.x-y) (λz.z/2)
5) (λ x.xx) (λ y.y)
6) (λ x.λ y. x + ((λ x.8) - y)) 5 6
7) (λ x.λ y. - x y) 9 4
```