Cálculo λ

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC que por sua vez foi baseado em `https://github.com/nadia-polikarpova/cse130-sp18/blob/master/lectures/01-lambda.md`.

Cálculo λ

UFABC

- Computabilidade é uma área de estudo central da Ciência da Computação. Ela estuda a possibilidade de resolver um problema seguindo um algoritmo.

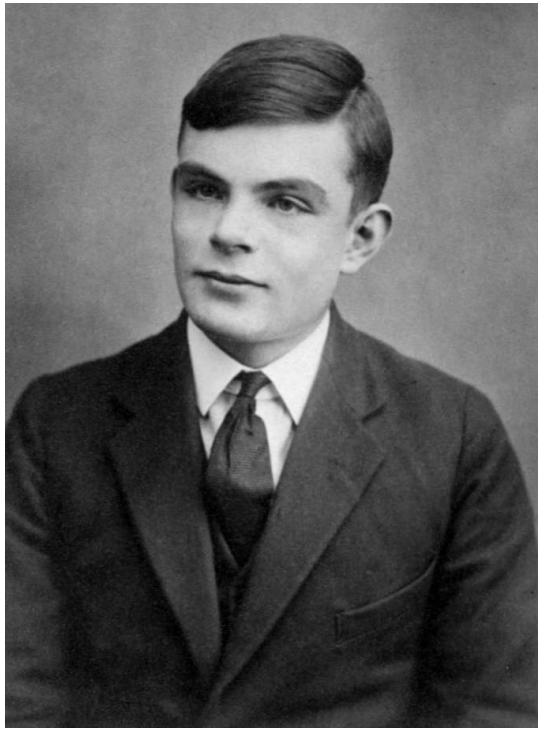
Problemas associados:

- **Decisão:** verifica se um elemento $s \in S$ também está contido em T. Exemplo: testar se um número é primo, $x \in \mathbb{N}, x \in P$.
- **Função:** calcular o resultado da aplicação de uma função $f : S \to T$. Exemplo: inverter uma *string*.
- **Busca:** verificar se xRy em uma relação binária R. Exemplo: buscar um clique em um grafo.
- **Otimização:** encontrar a solução x^* entre todas as soluções do espaço de busca S de tal forma a maximizar ou minimizar uma função $f(x)$. Exemplo: quanto devo colocar em cada possível investimento para maximizar meus lucros.

Máquina de Turing

UFABC

- Modelo matemático de computação criado por Alan Turing em 1936. Consiste de uma Máquina de Estado Finita cuja entrada é provida por uma fita de execução de tamanho arbitrário.
- A máquina permite ler, escrever, e *andar* por essa fita.



Wikipedia - Domínio Público

Qual a menor linguagem universal?

UFABC

Nas linguagens que vocês aprenderam até agora, temos:

- Atribuição (`x = x + 1`)
- Booleanos, inteiros, floats, caracteres,...
- Condicionais
- Laços
- Funções
- Recursão
- Ponteiros
- Objetos, classes

Mas do que realmente precisamos para programar?

- Sistema formal para expressar computação baseado em abstração de funções e aplicação usando apenas atribuições de nomes e substituições.
- Criado por Alonzo Church na década de 1930
  - Apresentado em 1932 e refinado até 1940 quando apresentou a sua versão tipada.
  - Church foi o orientador de doutorado do Alan Turing, que publicou o paper descrevendo máquinas de Turing em 1936

Wikipedia

UFABC

Descreve computação apenas utilizando... funções!!

- ~~Atribuição (`x = x + 1`)~~
- ~~Booleanos, inteiros, float, caracteres,...~~
- ~~Condicionais~~
- ~~Laços~~
- Funções
- ~~Recursão~~
- ~~Ponteiros~~
- ~~Objetos, classes~~

UFABC

Uma linguagem deve ser descrita em função de sua sintaxe e semântica (vocês estudarão isso em compiladores), ou de maneira menos formal, como você escreve e o que significa.

Wikipedia

”Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.”

Peter Landin, 1966

”The next 700 programming languages”. Communications of the ACM. Association for Computing Machinery. 9 (3): 157–166. https://doi.org/10.1145%2F365230.365257

Sintaxe do Cálculo λ

UFABC

- Composto de 3 elementos
  - Variáveis
  - Definição de funções
  - Aplicação de funções
- Em sua sintaxe original a letra grega minúscula lambda (λ) é usada para definir funções. Por exemplo:
  - Função identidade: $\lambda x.x$
  - Função que devolve a função identidade: $\lambda x.(\lambda y.y)$
- Nestes slides usamos a sintaxe de Haskell que troca
  - λ por \
  - . por ->

```
1  e ::= x
2      | \x -> e
3      | e1 e2
```

Sintaxe do Cálculo λ

UFABC

Um programa é definido por uma expressão `e`, ou termos-λ que podem assumir uma de três formas:

- Variável: `x, y, z`, um nome que assumirá um valor durante a computação.
- Abstração: (ou função anônima ou função λ)
  - `\x -> e`,
  - `x` é o parâmetro formal, `e` é o corpo da função
  - para qualquer valor `x` compute `e`
- Aplicação: (ou chamada de função)
  - `e1 e2`, aplique o argumento `e2` na função `e1` (`e1(e2)`).
  - Todo `e_i` é uma expressão!

Exercício 1

UFABC

- Qual dos seguintes termos está sintaticamente incorreto?
  (a) `\(\x -> x) -> y`
  (b) `\x -> x x`
  (c) `\x -> x (y x)`
  (d) Alternativas *a* e *c*
  (e) Todas acima

Resposta - Exercício 1

UFABC

- Qual dos seguintes termos está sintaticamente incorreto?
  (a) `\(\x -> x) -> y`
  (b) `\x -> x x`
  (c) `\x -> x (y x)`
  (d) Alternativas *a* e *c*
  (e) Todas acima
- Alternativa A

Exemplos

UFABC

```haskell
\x -> x                -- função identidade
\x -> (\y -> y)        -- retorna a função identidade
\f -> f (\x -> x)      -- função que aplica seu argumento
                       -- (que é uma função)
                       -- na função identidade
```

Funções de dois ou mais argumentos

UFABC

```haskell
\x -> (\y -> y) -- recebe dois args e retorna o segundo
\x -> (\y -> x) -- recebe dois args e retorna o primeiro
```

Syntatic Sugar

UFABC

original	syntatic sugar
`(((e1 e2) e3) e4)`	`e1 e2 e3 e4`
`\x -> (\y -> (\z -> e))`	`\x -> \y -> \z -> e`
`\x -> \y -> \z -> e`	`\x y z -> e`

```
1 \x y -> y
```

Escopo de uma variável

UFABC

Escopo indica a visibilidade de uma variável. Em C, Java:

```
1  int x; /* x está visível aqui, mas y não */
2  {
3    int y;  /* x e y estão visíveis */
4  }
5  /* y deixou de existir :( */
```

Escopo de uma variável

UFABC

- Na expressão `\x -> e`
  - `x` é uma variável
  - A expressão `e` é o escopo de `x`
  - Qualquer ocorrência de `x` em `e` está ligada (*bound*) por `\x`:

```
1 \x -> x
2 \x -> \y -> x
```

Escopo de uma variável

UFABC

Por outro lado **x** está livre (*free*) se não está dentro de uma abstração:

```
1  x y                  -- não tem \
2  \y -> x y            -- x vem de outro lugar
3  (\x -> \y -> x) x -- o segundo x é diferente do primeiro
```

Exercício 2

UFABC

Na expressão abaixo, **x** é ligado ou livre?

```
1  (\x -> x) x
```

Expressões fechadas

UFABC

- Se **e** não tem variáveis livres, então **e** é uma expressão fechada.
  - Expressões fechadas também são chamadas de combinadores (*combinators*)

Qual é a menor expressão fechada possível?

```
\x -> x
```

UFABC

Podemos reescrever as expressões utilizando duas regras:

- Passo α: renomeia um parâmetro formal.
- Passo β: aplica uma expressão utilizando uma variável livre (ou, de maneira mais simples, chama uma função).

Redução β

UFABC

```
1  (\x -> e1) e2 => e1[x := e2]
```

- Onde `e1[x := e2]` significa, "e1 com todas as ocorrências livres de `x` substituídas por `e2`."
- Computação por substituição e busca
  - Se você vir uma abstração aplicada a um argumento, pegue o corpo da abstração e substitua todas as ocorrências livres do parâmetro formal pelo argumento
  - Dizemos que `(\x -> e1) e2`, β-reduz para `e1[x := e2]`

Redução β

UFABC

```
(\x -> x) 2 => 2

(\f -> f (\x -> x)) (somar 1) => (somar 1) (\x -> x)
```

Exercício 3

UFABC

```
1  (\x -> (\y -> y)) 3 => ???
```

Equivalência α

UFABC

- Renomeia as variáveis de uma função para evitar conflito:

```
1  \x -> x => \y -> y
```

Forma normal (*Normal form*)

UFABC

- Um termo-λ na forma `(\x -> e1) e2` é chamado de expressão redutível (*reducible expression*) ou redex
- O termo está em sua forma normal se não contém nenhum redex.

Exercício 4

UFABC

Quais dos termos abaixo não está na forma normal?

```
1  x
2  x y
3  (\x -> x) y
4  x (\y -> y)
```

Avaliação

- Um termo-λ `e` é avaliado para `e'` se existe uma sequência de passos:

```
e => e1 => ... => en => e'
```

- ... tal que `e'` seja uma forma normal.

Avaliação

UFABC

```haskell
-- Exemplo 1
(\x -> x) 3 => 3

--Exemplo 2
(\f -> f (\x -> x)) (\x -> x)
    => (\x -> x) (\x -> x)
    => (\x -> x)
```

Non-Terminating Evaluation

```
1  (\x -> x x) (\x -> x x)
2    =β> (\x -> x x) (\x -> x x)
```

- Ops... Conseguimos escrever programas que a cada redução nos leva a eles próprios.
- É impossível chegar a uma forma normal.
- Este combinador é chamado Ω

- O que ocorre se passarmos Ω como argumento para outra função?

```
1 let OMEGA = (\x -> x x) (\x -> x x)
2
3 (\x -> \y -> y) OMEGA
```

- Essa expressão pode ser reduzida pra forma normal? Tente em casa. 

Programando com λ

- Linguagens reais tem uma pancada de funcionalidades:
  - Booleanos
  - Registros (structs, tuplas, ...)
  - Números
  - Funções ✓
  - Recursão
- Vamos ver como representar essas funcionalidades com cálculo λ

UFABC

- Como expressamos o conceito de Verdadeiro e Falso utilizando funções?

UFABC

Antes, é razoável nos perguntarmos:

- Para que utilizamos Verdadeiro e Falso?

UFABC

- Para que utilizamos Verdadeiro e Falso?
  - Decisões no formato: `if b then e1 else e2`.
- Nós já implementamos as funções necessárias para essa definição em outros slides 😀

Booleanos

UFABC

```
1  Verdadeiro = \x y -> x
2  Falso      = \x y -> y
3  IF         = \b x y -> b x y
```

UFABC

```
1  IF Verdadeiro 2 3
2    => (\b x y -> b x y) Verdadeiro 2 3
3    => (\x y -> Verdadeiro x y) 2 3
4    => (\y -> Verdadeiro 2 y) 3
5    => Verdadeiro 2 3
6    => (\x y -> x) 2 3
7    => (\y -> 2) 3
8    => 2
```

Exercício 5

UFABC

Defina as seguintes funções:

```
1  NOT = \b -> ???
2  AND = \b1 b2 -> ???
3  OR  = \b1 b2 -> ???
```

UFABC

- Linguagens reais tem uma pancada de funcionalidades:
  - Booleanos ✓
  - Registros (structs, tuplas, ...)
  - Números
  - Funções ✓
  - Recursão

Registros

UFABC

- Vamos começar com registros com dois elementos (ou, pares)
- Para que e como utilizamos pares?
  - Criamos um par dados dois itens
  - Pegamos o primeiro item de um par
  - Pegamos o segundo item de um par

A API de Pares

UFABC

- Precisamos definir três funções:

```
1  PAIR = \x y -> ???  -- Cria um par com elementos x e
   ↪  y
2                      -- { fst : x, snd : y }
3  FST  = \p -> ???    -- Devolve o primeiro elemento
4                      -- p.fst
5  SND  = \p -> ???    -- Devolve o segundo elemento
6                      -- p.snd
```

- Tal que

```
1  FST (PAIR abacate banana)
2    => abacate
3  SND (PAIR abacate banana)
4    => banana
```

Implementando Pares

UFABC

- Um par de **x** e **y** é apenas algo que permite você escolher entre **x** e **y**!
  - Ou seja, uma função que recebe um booleano e devolve ou **x** ou **y**
  - Onde você já viu isso antes? 😜

Implementando Pares

UFABC

```haskell
PAIR = \x y -> (\b -> IF b x y)
FST  = \p -> p Verdadeiro -- Chamando com Verdadeiro, pega o primeiro elemento
SND  = \p -> p Falso      -- Chamando com Falso, pega o segundo elemento
```

Exercício 6 - Trio/Terno/Tripla

UFABC

- Como implementar um registro com três valores?

```haskell
TRIPLE = \x y z -> ???
FST3   = \t -> ???
SND3   = \t -> ???
TRD3   = \t -> ???
```

Solução - Exercício 6

UFABC

```haskell
TRIPLE = \x y z -> PAIR x (PAIR y z)
FST3   = \t -> FST t
SND3   = \t -> FST (SND t)
TRD3   = \t -> SND (SND t)
```

UFABC

- Linguagens reais tem uma pancada de funcionalidades:
  - Booleanos ✓
  - Registros (structs, tuplas, ...) ✓
  - Números
  - Funções ✓
  - Recursão

Números Naturais

UFABC

- Considere os números naturais $0, 1, 2, \ldots$
- Para que (e como) os utilizamos?
  - Contagem: 0, `inc`, `dec`
  - Aritmética: `+`, `-`, `*`
  - Comparações: `==`, `<`, ...

Números Naturais

UFABC

- Vamos começar definindo os números:

```
1  ZERO = ???
2  UM   = ???
3  DOIS = ???
4  ...
```

UFABC

- **Números de Church:** um número `N` é codificado como a chamada de uma função `N` vezes:

```
1  ZERO = ???
2  UM   = \f x -> f x
3  DOIS = \f x -> f (f x)
4  TRES = \f x -> f (f (f x))
5  ...
```

UFABC

E o ZERO?

```
1  ZERO = ???
2  UM   = \f x -> f x
3  DOIS = \f x -> f (f x)
4  TRES = \f x -> f (f (f x))
5  ...
```

UFABC

Com que essa definição parece?

```
1  ZERO = \f x -> x
2  UM   = \f x -> f x
3  DOIS = \f x -> f (f x)
4  TRES = \f x -> f (f (f x))
5  ...
```

UFABC

```
1  ZERO  = \f x -> x
2  Falso = \x y -> y
```

UFABC

- Função `INC` deve adicionar 1 ao número `n`:

```
1  INC = \n -> ???
```

UFABC

Função `INC` deve adicionar 1 ao número `n`:

```
1  INC = \n -> ???
2  INC ZERO = UM
```

UFABC

Substituindo pelas definições:

```
1 INC = \n -> ???
2 INC (\f x -> x) = \f x -> f x
```

A operação que deve ser feita para encontrar o novo **x** é em função do ZERO:

```
1  INC = \n -> (\f x -> ???)
2  INC (\f x -> x) = \f x -> f x
3  INC (\f x -> x) = \f x -> f ((\f x -> x) ???)
```

UFABC

Se eu passar como argumento de ZERO o f e o x, obtemos:

```
1  INC = \n -> (\f x -> ???)
2  INC (\f x -> x) = \f x -> f x
3  INC (\f x -> x) = \f x -> f ((\f x -> x) f x)
4      => \f x -> f x
```

UFABC

Se eu passar como argumento de ZERO o f e o x, obtemos:

```
1  INC = \n -> (\f x -> f (n f x))
2  INC (\f x -> x) = \f x -> f x
3  INC (\f x -> x) = \f x -> f ((\f x -> x) f x)
4      => \f x -> f x
```

Exercício 7

UFABC

Como implementar a função **ADD**?

```
1  ADD = \n m -> ???
2  ADD DOIS UM = TRES
3  ADD (\f x -> f (f x)) (\f x -> f x) = (\f x -> f (f (f
   ↪  x)))
```

- Linguagens reais tem uma pancada de funcionalidades:
  - Booleanos ✓
  - Registros (structs, tuplas, ...) ✓
  - Números ✓
  - Funções ✓
  - Recursão

UFABC

- Como podemos fazer chamadas recursivas se as funções são anônimas (não tem nome)?

```haskell
SUM = \n -> ???  -- 1 + 2 + ... + n
```

UFABC

- Nas linguagens tradicionais basta fazer:

```
1 sum 0 = 0
2 sum n = n + sum (n-1)
```

- E no cálculo λ?

UFABC

```
1  \n -> IF (ISZERO n)
2            ZERO
3            (ADD n (SUM (DEC n)))
```

SUM não existe, não tem nome ainda!

```
1  \n -> IF (ISZERO n)
2            ZERO
3            (ADD n (SUM (DEC n)))
```

UFABC

Vamos criar uma função intermediaria, chamada STEP que recebe uma expressão **rec**:

```
1  STEP =
2     \rec -> \n -> IF (ISZERO n)
3                     ZERO
4                     (ADD n (rec (DEC n)))
```

Nosso objetivo é passar a definição de **rec** como parâmetro de STEP.

UFABC

Queremos criar uma função `FIX` que faça:

```
1 FIX STEP => STEP (FIX STEP)
```

UFABC

Dessa forma teríamos:

```
1  SUM = FIX STEP
2
3  SUM UM
4      => FIX STEP UM
5      => STEP (FIX STEP) UM
6      => \rec -> \n -> IF (ISZERO n) ZERO
7                       (ADD n (rec (DEC n))) (FIX STEP) UM
8      => \n -> IF (ISZERO n) ZERO
9                       (ADD n ((FIX STEP) (DEC n))) UM
10     => IF (ISZERO UM) ZERO
11                      (ADD UM ((FIX STEP) (DEC UM)))
12     => ADD UM ((FIX STEP) (DEC UM))
13     => ADD UM ((FIX STEP) ZERO)
```

Dessa forma teríamos:

```
1  => ADD UM (STEP (FIX STEP) ZERO)
2  => ADD UM (\rec -> \n -> IF (ISZERO n) ZERO
3             (ADD n (rec (DEC n))) (FIX STEP) ZERO)
4  => ADD UM (\n -> IF (ISZERO n) ZERO
5             (ADD n ((FIX STEP) (DEC n))) ZERO)
6  => ADD UM (IF (ISZERO ZERO) ZERO
7             (ADD ZERO ((FIX STEP) (DEC ZERO))))
8  => ADD UM ZERO
9  => UM
```

Nosso objetivo é passar a definição de **rec** como parâmetro de STEP.

- Já fizemos algo nessa linha quando falamos do combinador Ω
- Basta misturarmos a ideia do Ω com uma função que queremos que seja ”carregada”junto
- **Haskell** Brooks **Curry** 😛 foi o primeiro a fazê-lo!
  - Haskell foi orientado de doutorado do David Hilbert

```
1  FIX = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
```

UFABC

```
1  FIX STEP
2    => (\stp -> (\x -> stp (x x)) (\x -> stp (x x))) STEP
3    => (\x -> STEP (x x)) (\x -> STEP (x x))
4    => STEP ((\x -> STEP (x x) (\x -> STEP (x x))
```

UFABC

- Linguagens reais tem uma pancada de funcionalidades:
  - Booleanos ✓
  - Registros (structs, tuplas, ...) ✓
  - Números ✓
  - Funções ✓
  - Recursão ✓

Para saber mais

UFABC

- Exemplos de funções: *Principles of Programming Languages. Handout. The University of Birmingham*
- *Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach*. Ken Slonneger, Barry L. Kurtz. Capítulo 5. Disponível gratuitamente em: `http://www.divms.uiowa.edu/~slonnegr/plf/Book/`