Linguagens de Programação

Cálculo-λ

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Introdução
- Cálculo-λ (não tipado)
- Combinadores: *Idiot*, *Mockingbird*, *Kestrel, Kite, Cardinal*
- Lógica de Church
- Combinadores: *Bluebird*, *Thrush*
- Aritmética de Church
- Combinadores: *Vireo*
- Estrutura de dados
- Combinadores: *Starling*
- Base mínima
- Resumo

INTRODUÇÃO

Linguagens de Programação

Funções

- Funções são mapeamentos de entradas, membros de um conjunto (domínio), em saídas, membros de outro conjunto (imagem)

(entrada) $\mapsto$ (saída)

$$0 \mapsto 1$$
$$1 \mapsto 4$$
$$2 \mapsto 7$$
$$3 \mapsto 10$$
$$4 \mapsto 13$$
$$5 \mapsto 16$$
$$\ldots$$

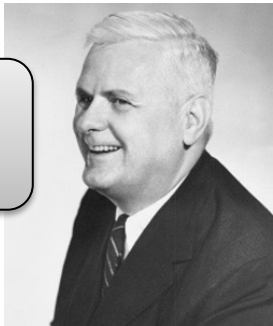
$$F(x) = 3 \times x + 1$$

(reescrita em formato anônimo)

$$x \mapsto 3 \times x + 1$$

Que função é essa?

É possível representar funções anônimas de outra forma?



Abstração-λ

- Uma forma diferente de representar funções é através de abstrações-λ

Abstrações-λ não possuem nomes, são como funções anônimas

$$\lambda x . 3 \times x + 1$$

Símbolo usado como marcador

Apenas uma variável como parâmetro

Uma expressão de retorno (corpo)

Qual a origem do símbolo λ?♣

Como lidar com múltiplos parâmetros?

Como lidar com números e operações aritméticas?

♣: https://en.wikipedia.org/wiki/Lambda_calculus#Origin_of_the_%CE%BB_symbol

α-Equivalência

- Considere as funções a seguir, o que elas têm em comum?

α-convertíveis

$$\lambda\, x.\, 3 \times x + 1$$
$$\lambda\, y.\, 3 \times y + 1$$
$$\lambda\, z.\, 3 \times z + 1$$
$$\lambda\, \star .\, 3 \times \star + 1$$
$$\lambda\, \circ .\, 3 \times \circ + 1$$
$$\lambda\, \Delta .\, 3 \times \Delta + 1$$

Como todas essas funções são similares, elas são **α-equivalentes**

Como **calcular** o resultado de uma função?

Aplicação com β-redução

- Como calcular o resultado para $\mathrm{F}(2)$ em notação matemática?

$$\mathrm{F}(x) = 3 \times x + 1$$

$$\begin{aligned}\mathrm{F}(2) &= 3 \times 2 + 1 \\ &= 6 + 1 \\ &= 7\end{aligned}$$

- E como calcular usando notacão λ?

Esse passo é dado usando uma β-redução

$$\lambda\, x.\, 3 \times x + 1$$

$$\begin{aligned}(\lambda\, x.\, 3 \times x + 1)\ 2 \quad &\text{Aplicação} \\ = 3 \times 2 + 1 \\ = 6 + 1 \\ = 7\end{aligned}$$

Como essa ideia é capturada por linguagens de programação?

Abstrações-λ e Aplicações em LPs

```
f = x => 3*x+1;
console.log(f(2));
```

```
f = lambda x: 3*x+1
print(f(2))
```

```
f = -> (x) { 3*x+1 }
puts f.call(2)
```

```
let f = |x: i32| -> i32 { 3*x+1 };
println!("{}", f(2));
```

```
Function<Integer,Integer> f = x -> 3*x+1;
System.out.println(f.apply(2));
```

```
f = \x -> 3*x+1
print(f 2)
```

```
Func<int, int> f = x => 3*x+1;
Console.WriteLine(f(2));
```

```
val f = (x: Int) => 3*x+1
println(f(2))
```

```
auto f = [](int x) { return 3*x+1; };
std::cout << f(2);
```

```
f = x -> 3*x+1
println(f(2))
```

```
f = fn x -> 3*x+1 end
IO.puts f.(2)
```

```
val f: (Int) -> Int = { x -> 3*x+1 }
println(f(2))
```

```
val f = fn x => 3*x+1;
print (Int.toString (f 2));
```

Cálculo-λ só permite receber um parâmetro como entrada, como lidar com funções com mais parâmetros?

Usando Múltiplos Parâmetros

- Considere o problema de encontrar a soma dos quadrados de 2 e 3
  - Como calcular em notação matemática usando a forma anônima?

Dica: uma ideia é passar uma tupla como (único) parâmetro

$$((x,y) \mapsto x^2 + y^2)\,(2,3)$$
$$= 2^2 + 3^2$$
$$= 13$$

  - E como calcular em SML/NJ usando funções anônimas?

```
- (fn (x,y) => x*x + y*y) (2,3);
val it = 13 : int
```

Será que existe outra forma de fazer isso?

Funções de Ordem Superior e Currying

- Funções de ordem superior são funções que permitem
  - Receber outras funções como entrada
  - Retornar outras funções como saída

- Uma função equivalente à da soma dos quadrados em notação matemática poderia ser escrita como

$$x \mapsto (y \mapsto x^2 + y^2)$$

Essa técnica é chamada de *currying*

$$\begin{aligned}
&\left(x \mapsto (y \mapsto x^2 + y^2)\right)\ 2\ 3 \\
&= (y \mapsto 2^2 + y^2)\ 3 \\
&= 2^2 + 3^2 \\
&= 13
\end{aligned}$$

- Em SML/NJ, essa mesma função poderia ser rescrita como

```
- (fn x => fn y => x*x + y*y) 2 3;
val it = 13 : int
```

Como representar números e operações aritméticas agora?

Abstrações

- Em cálculo-λ tudo são funções, portanto é preciso criar abstrações para operações aritméticas, números, entre outros...
  - $ADD \equiv \lambda x.\lambda y.\cdots$
  - $MUL \equiv \lambda x.\lambda y.\cdots$
  - $N1 \equiv \cdots$
  - $N3 \equiv \cdots$

- Como reescrever a função $F(x) = 3 \times x + 1$ usando essas abstrações?

$$F \equiv \lambda x.\, 3 \times x + 1$$

(converter para a notação aritmética pré-fixada)

$$F \equiv \lambda x.\, + (\times\ 3\ x)\ 1$$

(usar abstrações para as operações aritméticas)

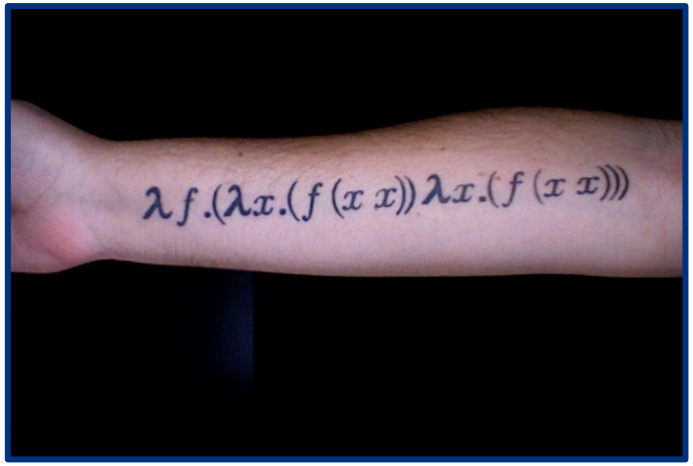
$$F \equiv \lambda x.\, ADD\ (MUL\ 3\ x)\ 1$$

(usar abstração para os números)

$$F \equiv \lambda x.\, ADD\ (MUL\ N3\ x)\ N1$$

Como fazer isso em cálculo-λ?

CÁLCULO-λ (NÃO TIPADO)♠



Linguagens de Programação

♠: https://www.youtube.com/watch?v=233A3bN0UBE&list=PLNwzBl6BGLwOKBFVbvp-GFjAA_ESZ--q4

λ-Termos

- **Definição** (conjunto Λ de λ-termos não tipados):
  - Variável: se $x \in V$, então $x \in \Lambda$
  - Aplicação: se $M, N \in V$, então $(M\ N) \in \Lambda$
  - Abstração: se $x \in V$, então $(\lambda x.M) \in \Lambda$

- Exemplos
  - $(\lambda x.x)$
  - $(\lambda x.(\lambda y.x))$
  - $(\lambda m.(\lambda f.(\lambda x.(f\ ((m\ f)\ x)))))$

Convenções

- Convenções para termos
  - Letras minúsculas x, y, z, ... para denotar variáveis
  - Letras maiúsculas L, M, N, ... para denotar λ-termos
  - O símbolo $\equiv$ para denotar identidade sintática

- Exemplos
  - $(\lambda x.\, x)$
  - $(M\ N)$
  - $(\lambda x.\, M)$, tal que $M \equiv (\lambda y.\, x)$

E os parênteses, como simplificar?

Convenções

- Convenções para parênteses
  - Parênteses externos podem ser omitidos: $(\lambda x.\, M) \equiv \lambda x.\, M$
  - Aplicação é associativa à esquerda: $(M\ N)\ L\ \equiv M\ N\ L$
  - Abstração é associativa à direita: $\lambda x.\, (\lambda y.\, M) \equiv \lambda x.\, \lambda y.\, M$
  - Aplicação tem precedência sobre abstração: $\lambda x.\, (M\ N) \equiv \lambda x.\, M\ N$

- Exemplos
  - $(\lambda x.\, x) \equiv \lambda x.\, x$
  - $(\lambda x.\, (x\ x)) \equiv \lambda x.\, x\ x$
  - $(\lambda x.\, (\lambda y.\, (\lambda z.\, (x\ (y\ z))))) \equiv \lambda x.\, \lambda y.\, \lambda z.\, x\ (y\ z)$

Convenções

- Convenção para abstrações-λ
  - Variáveis em termos *currificados* podem ser agrupadas

$$\lambda x.\lambda y.M \equiv \lambda xy.M$$

Cuidado: abstrações-λ ainda recebem somente uma variável como entrada; isso é apenas açúcar sintático para facilitar a leitura de funções *currificadas*

- Exemplos
  - $\lambda x.\lambda y.\lambda z.x\ (y\ z) \equiv \lambda xyz.x\ (y\ z)$
  - $\lambda m.\lambda f.\lambda x.f\ (m\ f\ x) \equiv \lambda mfx.f\ (m\ f\ x)$

Variáveis Livres e Ligadas

- **Definição** (o conjunto de variáveis livres FV de um λ-termo):
  - Variável: $FV(x) = \{x\}$, se $x \in V$
  - Aplicação: se $FV(M\ N) = FV(M) \cup FV(N)$
  - Abstração: $FV(\lambda x.M) = FV(M) \setminus \{x\}$

Como seria a definição do conjunto de variáveis ligadas BV?

- Exemplos
  - $\lambda x.x\ y$

$$FV(\lambda x.x\ y)$$
$$= FV(x\ y) \setminus \{x\}$$
$$= \big(FV(x) \cup FV(y)\big) \setminus \{x\}$$
$$= (\{x\} \cup \{y\}) \setminus \{x\}$$
$$= \{x, y\} \setminus \{x\}$$
$$= \{y\}$$
$$BV(\lambda x.x\ y) = \{x\}$$

  - $x\ (\lambda x.x\ y)$

$$FV(x\ (\lambda x.x\ y))$$
$$= FV(x) \cup FV(\lambda x.x\ y)$$
$$= \{x\} \cup \{y\}$$
$$= \{x, y\}$$
$$BV(x\ (\lambda x.x\ y)) = \{x\}$$

O que fazer quando se tem uma variável, como x, livre e ligada?

α-Conversão

- **Definição** (α-Conversão): dois λ-termos $M, N \in \Lambda$ são α-convertíveis, denotados por $M \equiv N$, se e somente se eles se diferem somente pelos nomes de suas variáveis ligadas

- Exemplos
  - $x\ (\lambda x.\, x\ y) \equiv x\ (\lambda z.\, z y)$
  - $x\ (\lambda x.\, x\ y) \not\equiv z\ (\lambda x.\, x y)$

Cuidado: não renomeie variáveis livres

Convenção de Barendregt

- **Definição** (convenção de Barendregt)**:** todas as variáveis ligadas em um λ-termos $M \in \Lambda$ devem ser diferentes entre si e que sejam escolhidas de forma que não entrem em conflito com os nomes de variáveis livres no termo

- Exemplos
  - $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$ não segue a convenção, pois o termo possui duas variáveis ligadas com o mesmo nome (x)
  - $(\lambda x.\, (\lambda y.\, x\ y))\ y$ também não segue a convenção, pois há conflito entre a variável livre y e a variável ligada y no termo
  - $(\lambda x.\, (\lambda z.\, x\ z))\ y$ segue a convenção, já que não há conflito entre os nomes de todas as suas variáveis

Cuidado: novamente, não renomeie variáveis livres

Substituição

- **Definição** (substituição):
  - Variável: $x[x := N] \equiv N$
    $\mathrm{y}[x := N] \equiv y$, se $x \not\equiv y$
  - Aplicação: $(P\ Q)[x := N] \equiv (P[x := N])\ (Q[x := N])$
  - Abstração: $(\lambda y.\, P)[x := N] \equiv \lambda y.\, (P[x := N])$, se $y \notin FV(N)$

- Exemplos
  - $(\lambda y.\, y\ x)[x := \lambda z.\, z\ y] \not\equiv \lambda y.\, y\ (\lambda z.\, z\ y)$, pois a variável y do termo aplicado era livre e depois da aplicação ficou ligada
  - $(\lambda y_1.\, y_1 x)[x := \lambda z.\, z\ y] \equiv \lambda y_1.\, y_1 (\lambda z.\, z\ y)$, já que não há conflito entre as variáveis agora

β-Redução de um Passo ($\rightarrow_\beta$)

- **Definição** (β-redução de um passo):
  - Caso base: $(\lambda x.\, M)N \rightarrow_\beta M[x := N]$
  - Passo indutivo: se $M \rightarrow_\beta N$, então $M\ L \rightarrow_\beta N\ L$, $L\ M \rightarrow_\beta L\ N$ e $\lambda x.\, M \rightarrow_\beta \lambda x.\, N$

A computação é dada pela β-redução

- Exemplos
  - $(\lambda x.\, x\ (x\ y))\ N \rightarrow_\beta (x\ (x\ y))[x := N] \equiv N\ (N\ y)$
  - $(\lambda x.\, (\lambda y.\, x\ y)\ z)\ v \rightarrow_\beta (\lambda x.\, (x\ y)[y := z])\ v \equiv (\lambda x.\, x\ z)\ v \rightarrow_\beta v\ z$

    $(\lambda x.\, (\lambda y.\, x\ y)\ z)\ v \rightarrow_\beta ((\lambda y.\, x\ y)\ z)[x := v] \equiv (\lambda y.\, v\ y)\ z \rightarrow_\beta v\ z$
  - $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \rightarrow_\beta (x\ x)[x := \lambda x.\, x\ x] \equiv (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \rightarrow_\beta \cdots$

Como dar vários passos?

β-Redução de Múltiplos Passos ($\rightarrow_\beta^*$)

- **Definição** (β-redução de um múltiplos passos): $M \rightarrow_\beta^* N$ se existir um $n \geq 0$ e termos $M_0, \ldots, M_n \in \Lambda$ tal que $M_0 \equiv M$, $M_n \equiv N$ e $M_i \rightarrow_\beta M_{i+1} \forall i, 1 \leq i \leq n$

- Exemplos
  - $M \rightarrow_\beta^* M$
  - $(\lambda x.\, x\ (x\ y))\ N \rightarrow_\beta^* N\ (N\ y)$
  - $(\lambda x.\, (\lambda y.\, x\ y)\ z)\ v \rightarrow_\beta^* v\ z$
  - $\Omega \equiv (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \rightarrow_\beta^* \Omega \rightarrow_\beta^* \cdots$

Quantos passos foram dados em cada um desses exemplos?

β-Conversão ($=_\beta$)

- **Definição** (β-conversão): $M =_\beta N$ se existir um $n \geq 0$ e termos $M_0, \ldots, M_n \in \Lambda$ tal que $M_0 \equiv M$, $M_n \equiv N$ e para todo $1 \leq i \leq n$ tem-se ou $M_i \to_\beta M_{i+1}$ ou $M_{i+1} \to_\beta M_i$

- Exemplos
  - $(\lambda y.\, y\ v)\ z =_\beta (\lambda x.\, z\ x)\ v$

Essa é uma computação de duas vias

$$(\lambda y.\, y\ v)\ z \to_\beta z\ v\ {}_\beta\!\leftarrow (\lambda x.\, z\ x)\ v$$

  - $(\lambda x.\, x\ (x\ y))\ N =_\beta N\ (N\ y) =_\beta (\lambda z.\, N\ (N\ z))\ y$

β-Forma Normal

- **Definição** (β-forma normal):
  - M está na β-forma normal se M não contiver nenhuma redução
  - M tem uma β-forma normal se existir um $N \in \Lambda$ em β-forma normal tal que $M =_\beta N$

O Cálculo-λ é Turing completo; isso só é possível porque existem termos que não possuem β-forma normal

- Exemplos
  - $(\lambda x.(\lambda y.x\ y)\ z)\ v$ tem uma β-forma normal $v\ z$
  - $\Omega \equiv (\lambda x.x\ x)\ (\lambda x.x\ x)$ não tem uma β-forma normal
  - $(\lambda v.u)\ \Omega \equiv (\lambda v.u)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$ tem uma β-forma normal u

$$(\lambda v.u)\ \Omega \rightarrow_\beta u$$

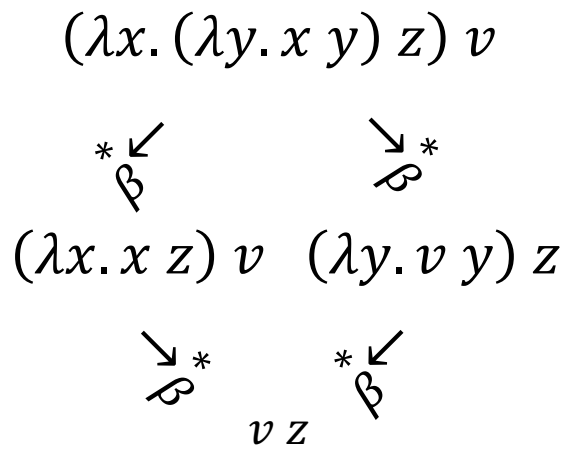
Dica: mesmo que Ω não tenha β-forma normal, esse termo pôde ser reduzido a u

Teorema de Church-Rosser

- **Teorema** (*Church-Rosser*): Sejam $M, N_1, N_2 \in \Lambda$ termos-λ com $M \rightarrow_\beta^* N_1$ e $M \rightarrow_\beta^* N_2$. Então existe um termo-λ $N_3 \in \Lambda$ tal que $N_1 \rightarrow_\beta^* N_3$ e $N_2 \rightarrow_\beta^* N_3$

Dica: esse teorema diz que a ordem de aplicação das β-reduções não é relevante

- Exemplo
  - $(\lambda x.(\lambda y.x\ y)\ z)\ v$



Cálculo-λ

- Linguagem
  - $x \in \Lambda$ } Variável
  - $(M\ N) \in \Lambda$ } Aplicação
  - $(\lambda x.\, M) \in \Lambda$ } Abstração

A linguagem é só isso; o que se poderia adicionar a ela para torná-la mais útil?

- Reduções
  - α-Conversão ($\equiv_\alpha$) } Equivalência
  - β-Conversão ($=_\beta$) } Computabilidade

- Biblioteca (Abstrações)
  - Booleanos (`true`, `false`) e operações lógicas (`not`, `and`, `or`, ...)
  - Números (`zero`, `um`, ...) e operações aritméticas (`add`, `mul`, ...)
  - Condicionais (*se/senão*)
  - Repetições (*recursões*)
  - Estruturas de dados (pares, listas, ...)

Tem mais abstrações?

Cálculo–λ com JavaScript

λ	JS	
x	`x`	Variáveis
(a)	`(a)`	
$f\ a$	`f(a)`	Aplicações
$f\ a\ b$	`f(a)(b)`	
$(f\ a)\ b$	`(f(a))(b)`	
$f\ (a\ b)$	`f(a(b))`	
$\lambda a.\, b$	`a => b`	Abstrações
$\lambda a.\, b\ \mathrm{x}$	`a => b(x)`	
$(\lambda a.\, b)\ x$	`(a => b)(x)`	
$\lambda a.\, \lambda b.\, \mathrm{a}$	`a => b => a`	
$\lambda a.\, (\lambda b.\, \mathrm{a})$	`a => (b => a)`	
$\lambda ab.\, \mathrm{a}$	~~`(a, b) => a`~~ `a => b => a`	

Lembrete: aplicações são associativas à esquerda

Lembrete: abstrações são associativas à direita

COMBINADORES♣

Linguagens de Programação

♣: https://github.com/glebec/lambda-talk

Combinadores

- **Definição** (Combinadores): são termos-λ sem variáveis livres

- Quais expressões a seguir são combinadores e quais não são?
  - $\lambda b.b$
  - $\lambda ab.a$
  - $\lambda abC.C\ (\lambda e.b)$

  São combinadores

  - $\lambda b.a$
  - $\lambda a.ab$

  Não são combinadores

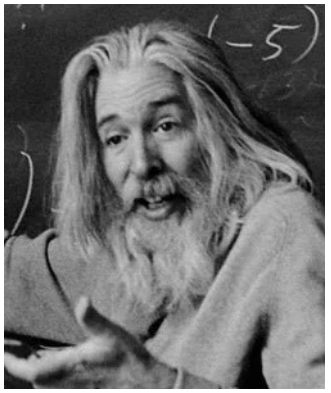
O que combinadores têm a ver com pássaros?

Sobre Combinadores e Pássaros

Moses Schönfinkel

Haskell Curry

Raymond Smullyan



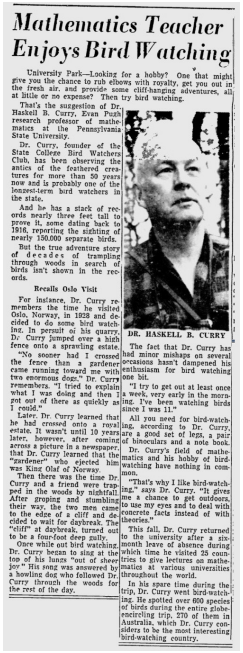
Identitätsfunktion — **I**

Konstanzfunktion — **K**

ver**S**chmelzungsfunktion — **S**

ver**T**auschungsfunktion — **C**

Zusammensetzungsfunktion — **B**



Mathematics Teacher Enjoys Bird Watching

University Park—Looking for a hobby? One that might give you the chance to rub elbows with royalty, get you out in the fresh air, and provide some cliff-hanging adventures, all at little or no expense? Then try bird watching.

That's the suggestion of Dr. Haskell B. Curry, Evan Pugh research professor of mathematics at the Pennsylvania State University.

Dr. Curry, founder of the State College Bird Watchers Club, has been observing the antics of the feathered creatures for more than 50 years now and is probably one of the longest-term bird watchers in the state.

And he has a stack of records nearly three feet tall to prove it, some dating back to 1916, reporting the sighting of nearly 150,000 separate birds.

But the true adventure story of decades of trampling through woods in search of birds isn't shown in the records.

Recalls Oslo Visit

For instance, Dr. Curry remembers the time he visited Oslo, Norway, in 1928 and decided to do some bird watching. In pursuit of his quarry, Dr. Curry jumped over a high fence onto a sprawling estate.

"No sooner had I crossed the fence than a gardener came running toward me with two enormous dogs," Dr. Curry remembers. "I tried to explain what I was doing and then I got out of there as quickly as I could."

Later, Dr. Curry learned that he had crossed onto a royal estate. It wasn't until 10 years later, however, after coming across a picture in a newspaper, that Dr. Curry learned that the "gardener" who ejected him was King Olaf of Norway.

Then there was the time Dr. Curry and a friend were trapped in the woods by nightfall. After groping and stumbling their way, the two men came to the edge of a cliff and decided to wait for daybreak. The "cliff" at daybreak, turned out to be a four-foot deep gully.

Once while out bird watching, Dr. Curry began to sing at the top of his lungs "out of sheer joy." His song was answered by a howling dog who followed Dr. Curry through the woods for the rest of the day.

DR. HASKELL B. CURRY

The fact that Dr. Curry has had minor mishaps on several occasions hasn't dampened his enthusiasm for bird watching one bit.

"I try to get out at least once a week, very early in the morning. I've been watching birds since I was 11."

All you need for bird-watching, according to Dr. Curry, are a good set of legs, a pair of binoculars and a note book.

Dr. Curry's field of mathematics and his hobby of bird-watching have nothing in common.

"That's why I like bird-watching," says Dr. Curry. "It gives me a chance to get outdoors, to use my eyes and to deal with concrete facts instead of with theories."

This fall, Dr. Curry returned to the university after a six-month leave of absence during which time he visited 25 countries to give lectures on mathematics at various universities throughout the world.

In his spare time during the trip, Dr. Curry went bird-watching. He spotted over 600 species of birds during the entire globe-encircling trip, 270 of them in Australia, which Dr. Curry considers to be the most interesting bird-watching country.

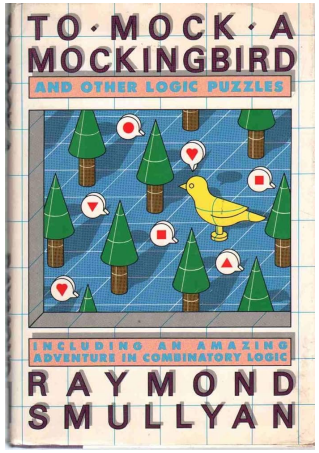
Idiot — **I**

Kestrel — **K**

Starling — **S**

Cardinal — **C**

Bluebird — **B**



COMBINADORES > *IDIOT*

$\lambda a.\, a$

I: *Idiot*

$$I \equiv \lambda a.\, a$$

Qual o nome dessa função na matemática?

Dica: em Haskell esse combinador é implementado pela função `id`

```
> I = a => a
[Function: I]
> I(5)
5
> I("oi")
'oi'
> I(I)
[Function: I]
```

```
ghci> i = id
ghci> i 5
5
ghci> i "oi"
"oi"
ghci> i i 3.14
3.14
```

Cuidado: em Haskell variáveis e funções devem começar com letras minúsculas e não é possível "imprimir" uma função

COMBINADORES > *MOCKINGBIRD*

$\lambda f . f f$

M: *Mockingbird*

$$M \equiv \lambda f.ff$$
$$\lambda f.ff \equiv \omega$$
$$\omega\ \omega \equiv \Omega$$

Esse combinador também é conhecido como ω e a aplicação em si próprio de Ω

Dica: essa capacidade de aplicar uma função a ela mesma é fundamental para implementação de funções recursivas

```
> M = f => f(f)
[Function: M]
> M(I)
[Function: I]
> M(M)
Uncaught RangeError: Maximum
call stack size exceeded
    at M (REPL2:1:10)
    at M (REPL2:1:10)
    at M (REPL2:1:10)
    ...
```

Cuidado: não é possível definir essa função em Haskell

```
ghci> m = \f -> f f
<interactive>:1:13: error:
...
```

COMBINADORES > *KESTREL*

$\lambda ab.\, a$

K: *Kestrel*

$$K \equiv \lambda ab.\, a$$

Dica: em Haskell esse combinador é implementado pela função `const`

```
> K = a => b => a
[Function: K]
> K(I)(M)
[Function: I]
> K(M)(I)
[Function: M]
> K(K)(M)
[Function: K]
```

```
ghci> k = const
ghci> k 7 2
7
```

Por que essa função é chamada de `const`?

COMBINADORES > *KITE*

$\lambda ab.b$

KI: Kite

$$KI \equiv \lambda ab.\, b$$
$$KI \equiv K\ I$$

`KI` é equivalente a aplicar `K I`

Dica: combinadores podem ser formados por outros combinadores

JS

```
> KI = a => b => b
[Function: KI]
> KI(I)(M)
[Function: M]
> KI(M)(I)
[Function: I]
> KI(K)(KI)
[Function: KI]
```

JS

```
> KI = K(I)
[Function: KI]
> KI(7)(2)
2
```

Dica: em Haskell pode ser usar a combinação de `const` com `id` para implementar esse combinador

```
ghci> ki = const id
ghci> ki 7 2
2
```

Qual o resultado de `K I x y`?

COMBINADORES > *CARDINAL*

$\lambda fab.\, fba$

C: Cardinal

$$C \equiv \lambda fab.fba$$

Dica: em Haskell esse combinador é implementado pela função `flip`

```
> C = f => a => b =>
... f(b)(a)
[Function: C]
> C(K)(I)(M)
[Function: M]
```

```
ghci> c = flip
ghci> c k 7 2
2
```

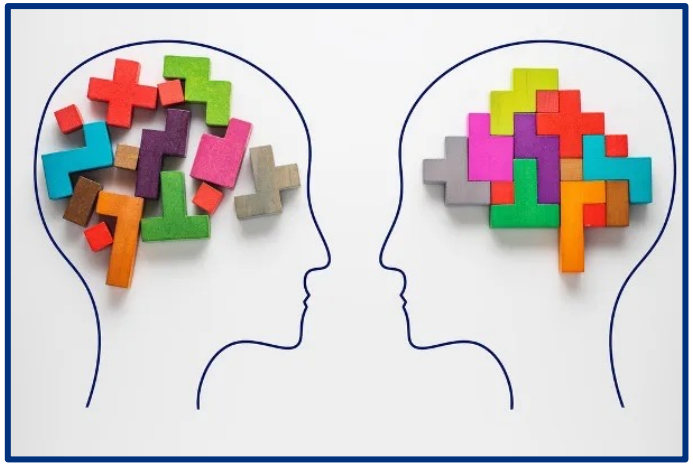
O que `C K` está fazendo?

```
> KI = C(K)
[Function (anonymous)]
> KI(7)(2)
2
```

```
ghci> ki = c k
ghci> ki 7 2
2
```

E o que `C KI` faria?

LÓGICA DE CHURCH



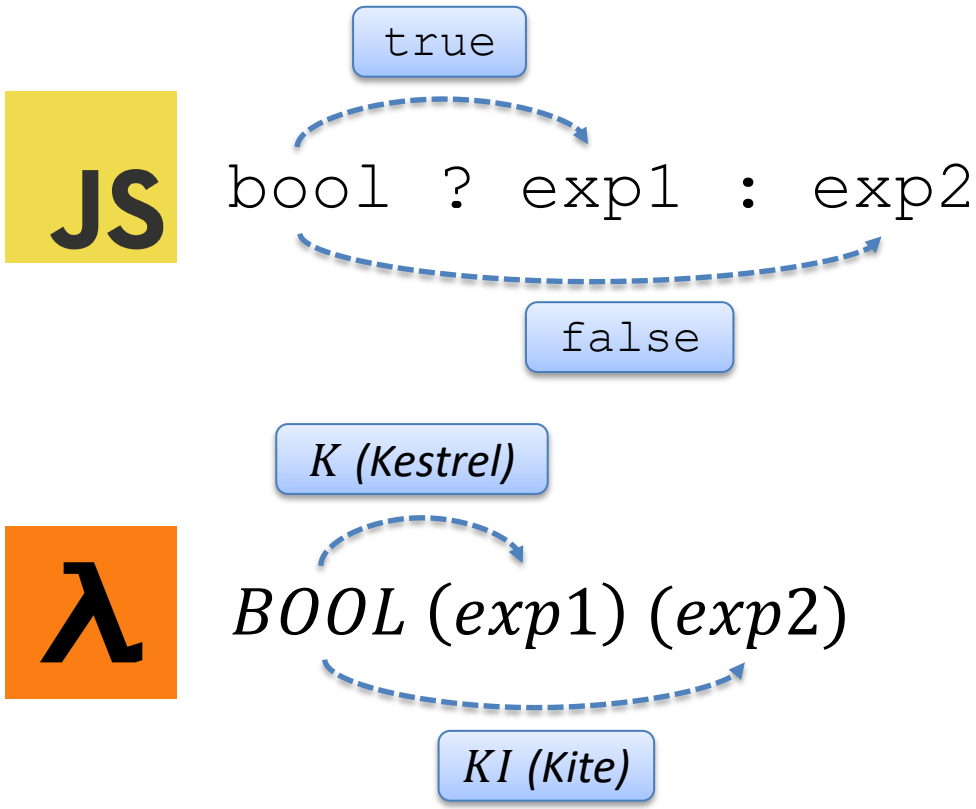
Linguagens de Programação

Representações

- Como o conceito de *falso* pode ser representado?
  - Em linguagens de programação
    - `false` (Java)
    - `False` (Python)
    - `0` (C)
    - `nil` (Ruby)
    - `null` (JavaScript)
  - Em lógica
    - $\bot$ (*bottom*)
    - $\neg True$ (*não-verdadeiro*)
  - Em conjuntos
    - $\emptyset$ (vazio)

Como representar booleanos em cálculo-λ?

Booleanos


true
JS
bool ? exp1 : exp2
false
K (Kestrel)
λ
BOOL (exp1) (exp2)
KI (Kite)

Codificação de Church: Booleanos

$$TRUE \equiv \lambda ab.a \equiv K$$
$$FALSE \equiv \lambda ab.b \equiv KI$$

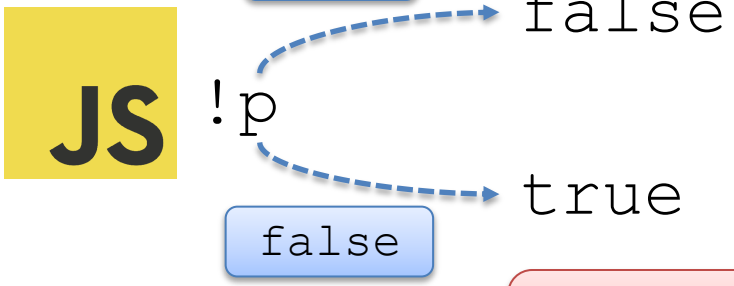
```
> TRUE = K
[Function: K]
> TRUE[util.inspect.custom] = () => "TRUE (K: Kestrel)";
[Function (anonymous)]
> TRUE
TRUE (K: Kestrel)
```

```
> FALSE = KI
[Function (anonymous)]
> FALSE[util.inspect.custom] = () => "FALSE (KI: Kite)";
[Function (anonymous)]
> FALSE
FALSE (KI: Kite)
```

Dica: usado para renomear funções em JS

Como fazer operações booleanas como NOT, AND, OR, ...?

NOT: Negação



$$NOT \equiv \lambda p.\ p\ FALSE\ TRUE$$

K (Kestrel)

KI (Kite)

Cuidado: usando `not'` para não entrar em conflito com a função interna `not`

Tem outro jeito de fazer?

JS

```
> NOT = p => p(FALSE)(TRUE)
[Function: NOT]
> NOT(TRUE)
FALSE (KI: Kite)
> NOT(FALSE)
TRUE (K: Kestrel)
```

```
ghci> not' = \p -> p false true
ghci> not' false 7 2
7
ghci> not' true 7 2
2
```

NOT: Negação

$\lambda ab.a$ $\lambda ba.a$

$NOT\ K \equiv KI$

$NOT\ KI \equiv K$

$\lambda ba.a$ $\lambda ab.a$

Qual combinador inverte os parâmetros?

$NOT \equiv C$

JS

```
> NOT_ = C
[Function: C]
> NOT_(TRUE)
[Function (anonymous)]
> NOT_(TRUE)(7)(2)
2
> NOT_(FALSE)(7)(2)
7
```

Dica: a função anônima retornada é **extensionalmente** equivalente à função $FALSE$ (KI)

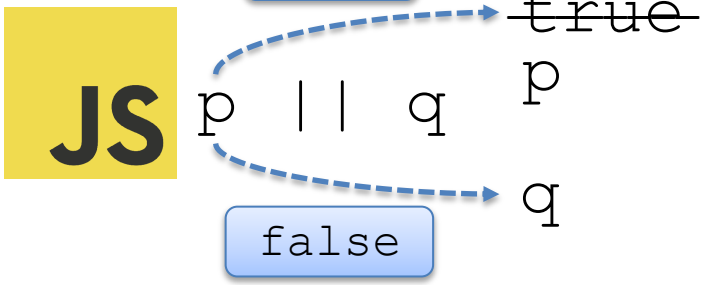
AND: *E Lógico*

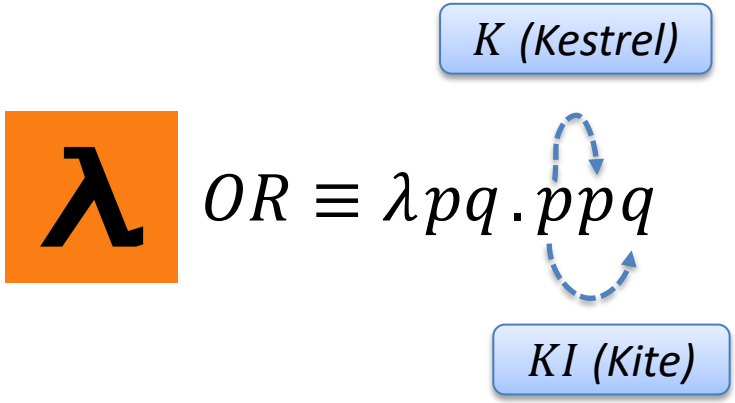
JS

```
> AND = p => q => p(q)(p)
[Function: AND]
> AND(FALSE)(FALSE)
FALSE (KI: Kite)
> AND(FALSE)(TRUE)
FALSE (KI: Kite)
> AND(TRUE)(FALSE)
FALSE (KI: Kite)
> AND(TRUE)(TRUE)
TRUE (K: Kestrel)
```

OR: *OU Lógico*

Dica: também usa curto-circuito





JS

```
> OR = p => q => p(p)(q)
[Function: OR]
> OR(FALSE)(FALSE)
FALSE (KI: Kite)
> OR(FALSE)(TRUE)
TRUE (K: Kestrel)
> OR(TRUE)(FALSE)
TRUE (K: Kestrel)
> OR(TRUE)(TRUE)
TRUE (K: Kestrel)
```

Tem outro jeito de fazer?

OR: OU Lógico

Qual combinador aplicado a x obtém $x\ x$?

λ

$$(\lambda pq.ppq)\ x\ y =_\beta x\ x\ y$$

$$M\ x\ y =_\beta x\ x\ y$$

$$OR \equiv M^*$$

Dica: o asterisco significa *once removed* (*currifica* os parâmetros)

JS

```
> OR = M
[Function: M]
> OR(FALSE)(FALSE)
FALSE (KI: Kite)
> OR(FALSE)(TRUE)
TRUE (K: Kestrel)
> OR(TRUE)(FALSE)
TRUE (K: Kestrel)
> OR(TRUE)(TRUE)
TRUE (K: Kestrel)
```

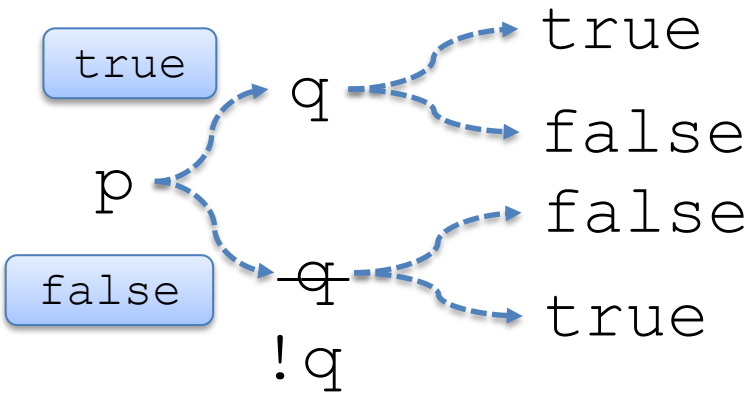
BEQ: Igualdade Booleana

JS `p === q`

λ $BEQ \equiv \lambda pq.pq(NOT\ q)$

K (Kestrel)

KI (Kite)



JS

```
> BEQ = p => q => p(q)(NOT(q))
[Function: BEQ]
> BEQ(FALSE)(FALSE)
TRUE (K: Kestrel)
> BEQ(FALSE)(TRUE)
FALSE (KI: Kite)
> BEQ(TRUE)(FALSE)
FALSE (KI: Kite)
> BEQ(TRUE)(TRUE)
TRUE (K: Kestrel)
```

Lei de De Morgan (Uma delas)

$$\neg(P \wedge Q) \leftrightarrow (\neg P) \vee (\neg Q)$$


```
!(p && q) === (!p) || (!q)
```

λpq.BEQ (NOT (AND p q)) (OR (NOT p) (NOT q))


```
> DM = p => q => BEQ(NOT(AND(p)(q)))(OR(NOT(p))(NOT(q)))
[Function: DM]
> DM(FALSE)(FALSE)
TRUE (K: Kestrel)
> DM(FALSE)(TRUE)
TRUE (K: Kestrel)
> DM(TRUE)(FALSE)
TRUE (K: Kestrel)
> DM(TRUE)(TRUE)
TRUE (K: Kestrel)
```

COMBINADORES > *BLUEBIRD*

$$\lambda f\, g\, a.\, f\,(g\, a)$$

B: Bluebird

O que esse combinador faz?

$$B \equiv \lambda f\, g\, a.\, f\,(g\, a)$$

Composição de funções da direita para a esquerda

```
> B = f => g => a =>
... f(g(a))
[Function: B]
> even = x => x % 2 == 0
[Function: even]
> not = x => !x
[Function: not]
> odd = B(not)(even)
[Function (anonymous)]
> odd(7)
true
```

```
ghci> odd = not . even
ghci> odd 7
True
```

Dica[1]: em Haskell a composição de funções é feita pelo operador infixo ponto

Dica[2]: poderia-se usar esse operador pré-fixado com parênteses:

```
odd = (.) not even
```

COMBINADORES > *THRUSH*

$\lambda af.fa$

T: *Thrush*

$$T \equiv \lambda af.fa$$

O que esse combinador faz?

Guarda um argumento

```
> T = a => f => f(a)
[Function: T]
> HOLDTRUE = T(TRUE)
[Function (anonymous)]
> HOLDTRUE(NOT)
FALSE (KI: Kite)
```

Tem outro jeito de fazer?

T: *Thrush*

$$C\ I \equiv (\lambda f a b.\ f\ b\ a)\ I =_\beta (\lambda a b.\ I\ b\ a) =_\beta (\lambda a b.\ b\ a) \equiv T$$

JS

```
> T = C(I)
[Function: T]
> HOLDFALSE = T(FALSE)
[Function (anonymous)]
> HOLDFALSE(NOT)
TRUE (K: Kestrel)
```

VS

```
ghci> t = flip id
ghci> holdfalse = t false
ghci> holdfalse not' 7 2
7
```

Dica: em Haskell pode se combinar `flip` (*Cardinal*) com `id` (*Idiot*)

ARITMÉTICA DE CHURCH

Linguagens de Programação

Representações

- Como o conceito de *três unidades* pode ser representado?
  - Numericamente
    - 3 (algarismo algébrico)
    - III (algarismos romanos)
    - 11_2 (base binária)
  - Visualmente
    - △△△
    - 𝍠
  - Em outras línguas
    - *three* (inglês)
    - *tres* (espanhol)
    - *trois* (francês)

Como representar numerais em cálculo-λ?

Codificação de Church: Numerais

$$N0 \equiv \lambda fa.a$$
$$N1 \equiv \lambda fa.fa$$
$$N2 \equiv \lambda fa.f(fa)$$
$$N3 \equiv \lambda fa.f(f(fa))$$

Vocês já viram algum desses combinadores antes?

$\lambda fa.a \equiv_\alpha \lambda ab.b$, *portanto* $N0 \equiv FALSE$

$N1 \equiv I^*$

```
> N0 = FALSE
FALSE (KI: Kite)
> N1 = I
[Function: I]
> N2 = f => a => f(f(a))
[Function: N2]
> N3 = f => a => f(f(f(a)))
[Function: N3]
> N0(NOT)(TRUE)
TRUE (K: Kestrel)
> N1(NOT)(TRUE)
FALSE (KI: Kite)
> N2(NOT)(TRUE)
TRUE (K: Kestrel)
> N3(NOT)(TRUE)
FALSE (KI: Kite)
```

Como transformar numerais de Church em números em JS?

```
> jsnum = n => n(x=>x+1)(0)
[Function: jsnum]
> jsnum(N0)
0
> jsnum(N1)
1
> jsnum(N3)
3
```

E como gerar $N99$?

Números de Peano

$$SUCC\ N0 \equiv N1$$
$$SUCC\ N1 \equiv N2$$
$$SUCC\ N2 \equiv N3 \equiv SUCC\ (SUCC\ N1)$$

Como definir $SUCC$?

$$SUCC\ N2$$
$$\equiv (\lambda nfa.f(n\,f\,a))\ N2$$
$$\equiv \lambda fa.f(N2\ f\,a)$$
$$\equiv \lambda fa.f(f(f(a)))$$
$$\equiv N3$$

$$SUCC \equiv \lambda nfa.f(nfa)$$

```
> SUCC = n => f => a => f(n(f)(a))
[Function: SUCC]
> jsnum(SUCC(N2))
3
> jsnum(SUCC(SUCC(N1)))
3
```

Tem outro jeito?

Números de Peano

$$SUCC \equiv \lambda nfa.f(nfa)$$
$$\equiv \lambda nf.Bf(nf)$$

```
> SUCC = n => f => B(f)(n(f))
[Function: SUCC]
> jsnum(SUCC(N2))
3
> jsnum(SUCC(SUCC(N1)))
3
```

Adição

$$ADD\ N1\ N5 \equiv SUCC\ N5$$
$$ADD\ N2\ N5 \equiv SUCC\ (SUCC\ N5)$$
$$ADD\ N3\ N5 \equiv SUCC\ (SUCC\ (SUCC\ N5))$$

$$ADD\ N3\ N5$$
$$\equiv SUCC\ (SUCC\ (SUCC\ N5))$$
$$\equiv (SUCC \cdot SUCC \cdot SUCC)\ N5$$
$$\equiv N3\ SUCC\ N5$$

$$ADD \equiv \lambda mn.\, m\ SUCC\ n$$

```
> ADD = m => n => m(SUCC)(n)
[Function: ADD]
> jsnum(ADD(N3)(N5))
8
```

Multiplicação

$$MUL\ N1\ N2 \equiv N2$$
$$MUL\ N2\ N2 \equiv N4$$
$$MUL\ N3\ N2 \equiv N6$$

$$MUL\ N3\ N2\ \ f\ a$$
$$\equiv (f \cdot f \cdot f \cdot f \cdot f \cdot f)\ a$$
$$\equiv ((f \cdot f) \cdot (f \cdot f) \cdot (f \cdot f))\ a$$
$$\equiv ((N2\ f) \cdot (N2\ f) \cdot (N2\ f))\ a$$
$$\equiv N3\ (N2\ f)\ a$$

Tem outro jeito?

$$MUL \equiv \lambda mnf.m(nf)$$

```
> MUL = m => n => f => m(n(f))
[Function: MUL]
> jsnum(MUL(N3)(N2))
6
```

Multiplicação

$$MUL \equiv \lambda mnf.m(nf) \equiv_\alpha \lambda fga.f(ga) \equiv B$$

$$MUL\ N3\ N2\ f$$
$$\equiv N3\ (N2\ f)$$
$$\equiv (N3 \cdot N2)\ f$$
$$\equiv B\ N3\ N2$$

```
> MUL = B
[Function: B]
> jsnum(MUL(N3)(N2))
6
```

Potência

$$POW\ N2\ N1 \equiv N2$$
$$POW\ N2\ N2 \equiv N4$$
$$POW\ N2\ N3 \equiv N8$$

$$POW\ N2\ N3$$
$$\equiv (N2 \times N2 \times N2)$$
$$\equiv (N2 \cdot N2 \cdot N2)$$
$$\equiv N3\ N2$$

$$POW \equiv \lambda mn.nm$$

```
> POW = m => n => n(m)
[Function: POW]
> jsnum(POW(N2)(N3))
8
```

Tem outro jeito?

Potência

$$POW \equiv \lambda mn.nm \equiv_{\alpha} \lambda af.fa \equiv T$$

```
> POW = T
[Function (anonymous)]
> jsnum(POW(N2)(N3))
8
```

COMBINADORES > *VIREO*

$\lambda abf.fab$

V: *Vireo*

O que esse combinador faz?

Guarda um par de argumentos

$$V \equiv \lambda abf.fab$$

```
> V = a => b => f =>
... f(a)(b)
[Function: V]
> vim = V(I)(M)
[Function (anonymous)]
> vim(K)
[Function: I]
> vim(KI)
[Function: M]
```

Tem outro jeito?

V: Vireo

$$
\begin{aligned}
& BCT \\
& \equiv (\lambda f\, g\, a.\, f\,(g\, a))\; C\; T \\
& =_\beta \lambda a.\, C\,(T\, a) \\
& \equiv \lambda a.\, C\;((\lambda a_1 f_1.\, f_1\; a_1)\; a) \\
& =_\beta \lambda a.\, C\;(\lambda f_1.\, f_1\; a) \\
& =_\beta \lambda a.\,((\lambda f_2 a_2 b_2.\, f_2\; b_2\; a_2)(\lambda f_1.\, f_1\; a)) \\
& \equiv \lambda a.\,(\lambda a_2 b_2.\,(\lambda f_1.\, f_1\; a)\; b_2\; a_2)) \\
& =_\beta \lambda a.\,(\lambda a_2 b_2.\,(b_2\; a)\; a_2)) \\
& \equiv \quad \lambda a a_2 b_2.\, b_2\; a\; a_2 \\
& \equiv_\alpha \lambda a\; b\;\; f.\, f\;\; a\;\; b \equiv V
\end{aligned}
$$

$V \equiv BCT$


```
> V = B(C)(T)
[Function (anonymous)]
> vim = V(I)(M)
[Function (anonymous)]
> vim(K)
[Function: I]
> vim(KI)
[Function: M]
```

ESTRUTURA DE DADOS

Linguagens de Programação

Pares

$$PAIR \equiv V$$
$$FST \equiv \lambda p.pK$$
$$SND \equiv \lambda p.p(KI)$$

```
> PAIR = V
[Function (anonymous)]
> FST = p => p(K)
[Function: FST]
> SND = p => p(KI)
[Function: SND]
```

```
ghci> p = (3,5)
ghci> fst p
3
ghci> snd p
5
```

```
> p = PAIR(N3)(N5)
[Function (anonymous)]
> FST(p)
[Function: N3]
> SND(p)
[Function: N5]
```

Fibonacci

- Considere a sequência de Fibonacci

Dado qualquer par dessa sequência, como gerar o próximo par?

0 1 1 2 3 5 8 13 21 34 55 ...

$$NEXT \equiv \lambda p.\, PAIR(SND\ p)(ADD(FST\ p)(SND\ p))$$

```
> NEXT = p => PAIR(SND(p))(ADD(FST(p))(SND(p)))
[Function: NEXT]
> p = PAIR(N3)(N5)
[Function (anonymous)]
> n = NEXT(p)
[Function (anonymous)]
> jsnum(FST(n))
5
> jsnum(SND(n))
8
```

Como obter o n-ésimo valor da sequência de Fibonacci?

Fibonacci

$$FIB \equiv \lambda n.FST(n(NEXT)(PAIR(N0)(N1)))$$

```
> FIB = n => FST(n(NEXT)(PAIR(N0)(N1)))
[Function: FIB]
> jsnum(FIB(N0))
0
> jsnum(FIB(N1))
1
> jsnum(FIB(N2))
1
> jsnum(FIB(N3))
2
> jsnum(FIB(N4))
3
> jsnum(FIB(N5))
5
> jsnum(FIB(SUCC(N5)))
8
```

COMBINADORES > *STARLING*

$$\lambda f\, g\, a.\, f\, a\, (g\, a)$$

S: *Starling*

O que esse combinador faz?

Aplica um mesmo argumento (`a`) a duas funções (`f` e `g`) e depois combina seus resultados

$$S \equiv \lambda f\, g\, a.\, f\, a\, (g\, a)$$

```
> S = f => g => a => f(a)(g(a))
[Function: S]
> S(x => y => x+y)(z => 2*z)(3)
9
> jsnum(S(ADD)(MUL(N2))(N3))
9
```

Exemplo usando funções em JS

Exemplo usando funções cálculo-λ

BASE MÍNIMA

$SKIBC$

- Todo λ-termo pode ser expresso em termos dos combinadores: S (*Starling*), K (*Kestrel*), I (*Idiot*), B (*Bluebird*) e C (*Cardinal*)

Mockingbird

$$M \equiv \lambda f.ff \equiv SII$$

Thrush

$$T \equiv \lambda af.fa \equiv CI$$

Kite

$$KI \equiv \lambda ab.b \equiv KI \equiv CK$$

Vireo

$$V \equiv \lambda abf.fab \equiv \cancel{BCT} \equiv BC(CI)$$

Dica: mais detalhes deste algoritmo em https://youtu.be/gnrSedVucXs?t=534

Fibonacci usando $SKIBC$

$$FIB \equiv B(CIK)((BC(CI))(S(B(BC(CI))(CI(KI)))(S(B(CI(SB))(CIK))(CI(KI))))((BC(CI))(K(I))(I)))$$

```
> FIB = B(C(I)(K))(B(C)(C(I))(S(B(B(C)(C(I)))(C(I)(K(I)))))
... (S(B(C(I)(S(B)))(C(I)(K)))(C(I)(K(I)))))(B(C)(C(I))(K(I))(I)))
[Function (anonymous)]
> jsnum(FIB(N5))
5
> jsnum(FIB(SUCC(N5)))
8
> jsnum(FIB(SUCC(SUCC(N5))))
13
> jsnum(FIB(SUCC(SUCC(SUCC(N5)))))
21
```

Dá para fazer com menos que 5 combinadores?

SKI

- É possível obter os combinadores B (*Bluebird*) and C (*Cardinal*) a partir dos combinadores S (*Starling*), K (*Kestrel*) e I (*Idiot*)

Bluebird

$$B \equiv \lambda f\, g\, a.\, f\,(g\, a)$$
$$\equiv S(KS)K$$

Cardinal

$$C \equiv \lambda f\, a\, b.\, f\, b\, a$$
$$\equiv S(S(K(S(KS)K))S)(KK)$$

Idiot

$$I \equiv \lambda a.\, a$$
$$\equiv SKK$$
$$\equiv SKS$$

Dica: embora I possa ser derivado em termos de S e K ele é mantido por conveniência

Fibonacci usando SKI

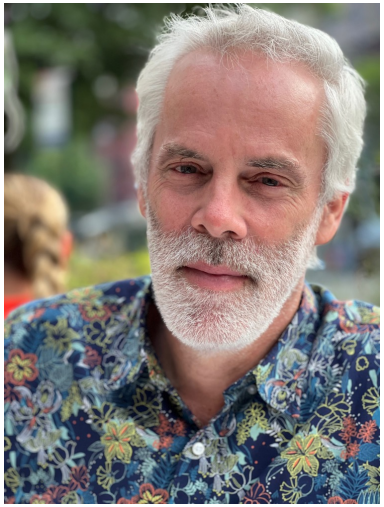
$$FIB \equiv (S(K(S))(K))((S(S(K(S(K(S))(K)))(S))(K(K)))IK)(((S(K(S))(K))(S(S(K(S(K(S))(K)))(S))(K(K)))((S(S(K(S(K(S))(K)))(S))(K(K)))I))(S((S(K(S))(K))((S(K(S))(K))(S(S(K(S(K(S))(K)))(S))(K(K)))((S(S(K(S(K(S))(K)))(S))(K(K)))I))((S(S(K(S(K(S))(K)))(S))(K(K)))I(KI)))(S((S(K(S))(K))((S(S(K(S(K(S))(K)))(S))(K(K)))I(S(S(K(S))(K))))((S(S(K(S(K(S))(K)))(S))(K(K)))IK))((S(S(K(S(K(S))(K)))(S))(K(K)))I(KI))))(((S(K(S))(K))(S(S(K(S(K(S))(K)))(S))(K(K)))((S(S(K(S(K(S))(K)))(S))(K(K)))I))(K(I))(I)))$$

```
> FIB = (S(K(S))(K))((S(S(K(S(K(S))(K)))(S))(K(K)))(I)(K))((S(K(S))(K))
... ((S(S(K(S(K(S))(K)))(S))(K(K))))((S(S(K(S(K(S))(K)))(S))(K(K)))(I))
... (S((S(K(S))(K))((S(K(S))(K))((S(S(K(S(K(S))(K)))(S))(K(K))))
... ((S(S(K(S(K(S))(K)))(S))(K(K)))(I)))((S(S(K(S(K(S))(K)))(S))(K(K)))
... (I)(K(I))))(S((S(K(S))(K))((S(S(K(S(K(S))(K)))(S))(K(K)))(I)
... (S((S(K(S))(K)))))((S(S(K(S(K(S))(K)))(S))(K(K)))(I)(K)))
... ((S(S(K(S(K(S))(K)))(S))(K(K)))(I)(K(I)))))((S(K(S))(K))
... ((S(S(K(S(K(S))(K)))(S))(K(K))))((S(S(K(S(K(S))(K)))(S))(K(K)))(I))
... (K(I))(I)))
[Function (anonymous)]
> jsnum(FIB(SUCC(SUCC(SUCC(N5)))))
21
```

Dá para fazer com menos que 2 combinadores?

ι (iota)

- O combinador ι (iota) foi descoberto por Chris Barker em 2001



$$\iota \equiv \lambda h.\, h \underbrace{(\lambda f\, g x.\, f\, x\, (g x))}_{S} \underbrace{(\lambda a b.\, a)}_{K}$$

Idiot

$$I \equiv \lambda a.\, a$$
$$\equiv \iota\, \iota$$

Kestrel

$$K \equiv \lambda a b.\, a$$
$$\equiv \iota(\iota(\iota\, \iota))$$

Starling

$$S \equiv \lambda f\, g x.\, f\, g x$$
$$\equiv \iota(\iota(\iota(\iota\, \iota)))$$

Fibonacci usando ι

$FIB \equiv ((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i)))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i)))$
$((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i)))))((i(i(i(i(i))))))((i(i(i(i)))((i(i(i(i))))))$
$((i(i))((i(i(i(i)))))(((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i)))))(((i(i(i(i(i)))))$
$((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i))))))((i(i(i(i(i)))))))$
$((i(i(i(i)))((i(i(i(i(i)))))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))((i(i(i(i)))$
$((i(i(i(i(i)))))))((i(i(i(i))))))((i(i(i(i(i)))))))((i(i(i(i)))((i(i(i(i))))))((i(i)))((i(i(i(i(i)))))$
$(((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i)))))(((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))))$
$((i(i(i(i))))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))))$
$((i(i(i(i))))))((i(i(i(i(i)))))))((i(i(i(i)))((i(i(i(i)))))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i)))$
$((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i)))))((i(i(i(i(i))))))((i(i(i(i)))((i(i(i(i))))))$
$((i(i)))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))))$
$((i(i(i(i))))))((i(i(i(i(i)))))))((i(i(i(i)))((i(i(i(i))))))((i(i))((i(i(i(i)))((i(i)))))((i(i(i(i(i)))))$
$(((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i)))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i)))$
$((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i)))))((i(i(i(i(i))))))((i(i(i(i)))((i(i(i(i))))))$
$((i(i))((i(i(i(i(i)))))(((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i))))))))(((i(i(i(i(i)))))$
$((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i))))))((i(i(i(i(i)))))))$
$((i(i(i(i)))((i(i(i(i(i)))))))((i(i))((i(i(i(i))))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))$
$((i(i(i(i)))((i(i(i(i(i)))))))((i(i(i(i))))))((i(i(i(i(i)))))))((i(i(i(i)))((i(i(i(i))))))((i(i))((i(i(i(i)))$
$((i(i))))))(((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i)))))(((i(i(i(i(i)))))((i(i(i(i(i)))))$
$((i(i(i(i)))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i))))))((i(i(i(i))))))((i(i(i(i(i)))))))((i(i(i(i)))$
$((i(i(i(i)))))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))((i(i(i(i)))((i(i(i(i(i)))))))$
$((i(i(i(i))))))((i(i(i(i(i)))))))((i(i(i(i)))((i(i(i(i))))))((i(i)))((i(i(i(i)))((i(i)))((i(i)))))$

Fibonacci usando ι

```
> i = h => h(f => g => x => f(x)(g(x)))(a => b => a)
[Function: i]
> FIB = ((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i))))))((i(i(i(i)))))(((i(i(i(i(i)))))((i(i(i(i(i)))))
... ((i(i(i(i))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i))))))((i(i(i(i))))))((i(i(i(i(i)))))))
... ((i(i(i(i))))((i(i(i(i)))))))((i(i)))((i(i(i(i))))))(((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))))
... ((i(i(i(i))))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))((i(i(i(i)))
... ((i(i(i(i(i)))))))((i(i(i(i)))))))((i(i(i(i(i)))))))((i(i(i(i))))((i(i(i(i)))))))(((i(i(i(i(i)))))
... ((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))))((i(i(i(i)))))))
... ((i(i(i(i(i)))))))((i(i(i(i))))((i(i(i(i)))))))((i(i)))((i(i(i(i(i)))))(((i(i(i(i(i)))))
... ((i(i(i(i))))((i(i(i(i(i)))))))((i(i(i(i))))))(((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))))
... ((i(i(i(i))))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))((i(i(i(i)))
... ((i(i(i(i(i)))))))((i(i(i(i)))))))((i(i(i(i(i)))))))((i(i(i(i))))((i(i(i(i)))))))(((i(i(i(i(i)))))
... ((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))))((i(i(i(i)))))))
... ((i(i(i(i(i)))))))((i(i(i(i))))((i(i(i(i)))))))((i(i))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i)))
... ((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))))((i(i(i(i)))))))((i(i(i(i(i)))))))((i(i(i(i)))
... ((i(i(i(i)))))))((i(i)))((i(i(i(i))))((i(i))))))((i(i(i(i(i)))))(((i(i(i(i(i)))))((i(i(i(i)))
... ((i(i(i(i(i)))))))((i(i(i(i))))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i))))))
... ((i(i(i(i))))((i(i(i(i(i)))))))((i(i(i(i))))))((i(i(i(i(i)))))))((i(i(i(i))))((i(i(i(i)))))))((i(i)))
... ((i(i(i(i(i)))))(((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))))((i(i(i(i))))))))(((i(i(i(i(i)))))
... ((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))))((i(i(i(i)))))))
... ((i(i(i(i(i)))))))((i(i(i(i))))((i(i(i(i)))))))((i(i)))((i(i(i(i))))))(((i(i(i(i(i)))))((i(i(i(i(i))))))
... ((i(i(i(i))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))))((i(i(i(i)))))))((i(i(i(i(i)))))))
... ((i(i(i(i))))((i(i(i(i)))))))((i(i)))((i(i(i(i))))((i(i)))))))(((i(i(i(i(i)))))((i(i(i(i)))
... ((i(i(i(i(i)))))))((i(i(i(i))))))(((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i))))))
... ((i(i(i(i))))((i(i(i(i(i)))))))((i(i(i(i)))))))((i(i(i(i(i)))))))((i(i(i(i))))((i(i(i(i))))))))
... (((i(i(i(i(i)))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))((i(i(i(i))))((i(i(i(i(i)))))))
... ((i(i(i(i)))))))((i(i(i(i(i)))))))((i(i(i(i))))((i(i(i(i)))))))((i(i)))((i(i(i(i))))((i(i)))((i(i))))
[Function (anonymous)]
> jsnum(FIB(SUCC(SUCC(SUCC(N5)))))
21
```

RESUMO

Combinadores

S.	Pássaro	Cálculo-λ	Uso	Haskell
I	*Idiot*	$\lambda a.a$	Identidade	`id`
M	*Mockingbird*	$\lambda f.ff$	Autoaplicação	**(não pode ser definido)**
K	*Kestrel*	$\lambda ab.a$	Verdade, primeiro, *const*	`const`
KI	*Kite*	$\lambda ab.b \equiv \mathrm{KI} \equiv CK$	Falso, segundo	`const id`
C	*Cardinal*	$\lambda fab.fba$	Reverte argumentos	`flip`
B	*Bluebird*	$\lambda fga.f(ga)$	Composição	`(.)`
T	*Thrush*	$\lambda af.fa \equiv CI$	Guarda um argumento	`flip id`
V	*Vireo*	$\lambda abf.fab \equiv BCT$	Guarda um par de argumentos	`flip . flip id`
S	*Starling*	$\lambda fga.fa(ga)$	Aplicar argumento a duas funções e combina os resultados	`<*>`

Codificação de Church: Booleanos

Nome	Cálculo-λ	Uso
$TRUE$	$\lambda ab.a \equiv K \equiv C(KI)$	Codificação de verdadeiro
$FALSE$	$\lambda ab.b \equiv KI \equiv CK$	Codificação de falso
NOT	$\lambda p.p\ FALSE\ TRUE \equiv C$	Negação
AND	$\lambda pq.pqp$	Conjunção
OR	$\lambda pq.ppq \equiv M^*$	Disjunção
BEQ	$\lambda pq.pq(NOT\ q)$	Igualdade booleana

Codificação de Church: Numerais

Nome	Cálculo-λ	Uso
$N0$	$\lambda fa.a \equiv FALSE$	Aplicar `f` zero vezes a `a` (Número zero ou falso)
$N1$	$\lambda fa.fa \equiv I^*$	Aplicar `f` uma vez a `a` (Número um ou identidade)
$N2$	$\lambda fa.f(fa)$	Aplicar `f` duas vezes a `a` (Número dois)
$SUCC$	$\lambda nf.Bf(nf)$	Sucessor de um numeral
ADD	$\lambda mn.m\ SUCC\ n$	Adição
$MULT$	$\lambda mnf.m\ (nf) \equiv B$	Multiplicação
POW	$\lambda mn.nm \equiv T$	Potência

Estrutura de Dados

Nome	Cálculo-λ	Uso
$PAIR$	$\lambda abf.fab \equiv V$	Par de dois argumentos
FST	$\lambda p.pK$	Extrair primeiro do par
SND	$\lambda p.p(KI)$	Extrair segundo do par

ISSO É TUDO, PESSOAL!