
nu bank

QUEM SOMOS?

Leonardo Iacovini

leonardo.iacovini@nubank.com.br

Rafael Leal

rafael.leal@nubank.com.br

Nubank

- Cartão de Crédito (2014)
- Rewards (2016)
- Conta (2017)
- Cartão de Débito (2018)
- PJ (2019)
- Empréstimos (2019)

- 20 milhões de clientes
- Mais de 550 Engenheiros
- Mais de 350 Microsserviços Clojure

Tecnologias

- Microsserviços
- Clojure
- Scala
- Kafka
- Datomic
- Kubernetes
- AWS
- ...

Clojure

- LISP
- Funcional
- JVM
- REPL

Programação Funcional

Paradigma de programação que procura tratar problemas computacionais como a avaliação de funções matemáticas, tem suas origens no cálculo Lambda

- Clojure
- Elixir
- Scala
- Haskell
- F#

cálculo-λ

- Turing complete.
- Tudo são funções, os únicos símbolos definidos são λ e .

λx.x
λ -> Delimita uma função
x. -> indica o argumento
x -> corpo da função

https://blog.caelum.com.br/comecando-com-o-calculo-lambda-e-a-programacao-funcional-de-verdade/

cálculo-λ

T ≡ λab.a ≡ λa.λb
F ≡ λab.b ≡ λb.λa
NOT ≡ λx.xFT

NOT ≡ λx.xFT

NOT é uma função que recebe um booleano, e devolve a aplicação desse booleano nos parâmetros F e T

NOT(T) => T(F, T) => F
NOT(F) => F(F, T) => T

https://blog.caelum.com.br/comecando-com-o-calculo-lambda-e-a-programacao-funcional-de-verdade/

- Funções puras
- Imutabilidade
- Recursão (com cuidado!)
- Idempotência (livre de *side-effects*)
- Funções são valores (First Class Functions)
- Transparência referencial
- Programação declarativa

Funcional?

```
void fill_message(Person* person) {
    int age_in_seconds = person->age;
    person->message = person->name + " is " + (int)
age_in_seconds/3600 + "years old";
 }
```

"Uma relação f:A⟶B é uma função quando para qualquer x ∈ A existe um único (x,y) ∈ f."

Funções Puras

Os dados recebidos não são alterados. Um novo valor é criado e retornado

Não dependem de nenhum estado externo

Para um mesmo conjunto de argumentos o resultado sempre será o mesmo (determinística)

São funções livre de efeitos colaterais

Efeitos colaterais são ações que alteram o estado do mundo (Eg: enviar um email, fazer uma transação no DB, etc...)

Funções puras são fáceis de entender, testar e seguras de executar! (idempotentes)

Funções Puras

```
let resultArray = []

someIntermediateArray.forEach(o => {
 resultArray.push(o.someAttribute)
})

return resultArray
```

```
return someIntermediateArray
  .map(o => o.someAttribute)
```

Imutabilidade

- “Variáveis” não tem seu valor alterado
  Se **a** = 1, então **a** sempre será 1

- Funções não alteram o valor de suas entradas, elas produzem uma saída com as alterações feitas.

f(a) → b onde b != a
(outro “*objeto/instância*” - não é a mesma referência)

Vantagens

- Programação concorrente/paralelismos (thread safety)
- Fica mais fácil entender o que o código está fazendo
- Facilidade para testar

Performance

Isso não deveria piorar a performance? (Listas por exemplo)

Preciso ficar copiando as estruturas toda manipulação ?

NÃO

Clojure (e a maioria das linguagens de funcionais) usam estruturas de dados otimizadas para operações imutáveis, a maioria das operações é feita em O(1). - essas estruturas são geralmente conhecidas como *persistent data structures*

Se utiliza de técnicas de reaproveitamento de memória (já que os dados não mudam - copy-on-write)

Read more:
http://lampwww.epfl.ch/papers/idealhashtrees.pdf

Imutabilidade elimina grande parte dos problemas que ocasionam *dead-locks* e problemas de concorrência, simplificando nosso código

```
lock.lock();
doSomething();
lock.unlock();
```

```
synchronized (Lock lock) {
    doSomething()
}
```

Mutável

```
int factorial(int n) {
   int f = 1;
   while(n > 0)
     f *= n--;
   return f;
}
```

Imutável

```
(defn factorial [n]
  (if (= n 0)
    1
    (* n (factorial (dec n)))))
```

Idempotência

Em matemática e ciência da computação, a idempotência é a propriedade que algumas operações têm de poderem ser aplicadas várias vezes sem que o valor do resultado se altere após a aplicação inicial.

```
f(f(x)) = f(x)
```

Mas e daí?

Quando tivermos que lidar com estado externo, ainda podemos pensar em funções ao invés de subrotinas!

Ao invés de pensarmos em:

`f(x) → void`

pensamos:

`f(x, world) → new-world`

Em caso de falha, podemos tentar novamente sem medo de dupla execução!

Funções são Valores

Funções são tratadas como demais tipos (inteiros, float, strings, etc...), e podem ser atribuídas a variáveis, passadas como argumentos ou retornadas por outras funções.

High Order Functions

São funções que recebem outras funções como argumento

Ou, funções que retornam outras funções

$$f(g(x)) === f \circ g$$

$$f(g, a) \rightarrow b$$

$$f(a, b) \rightarrow g$$

onde `f` e `g` são funções

Transparência Referencial

O resultado de uma função depende apenas dos argumentos que ela recebe

Característica de funções puras

Testes

Escrever testes em linguagens funcionais é muito mais fácil e seguro.

Não dependemos de estado externo (fixtures).

Podemos escrever testes generativos (testes baseado em propriedades)

```clojure
(def incrementa (fn [a] (+ a 1)))
(map incrementa [0 1 2 3 4]) ;; => [1 2 3 4 5]
```

```clojure
(def even? (complement odd?)) ;; even? é um função

(even? 2) ;; => false
(even? 3) ;; => true
```

```clojure
(defn f [a] (+ a 1)))
(defn g [a] (* a 2))
(def fg (comp f g)) ;; => (fg a) === (f (g a))
(fg 10) ;; => 21
```

HoF Comuns

```clojure
(map inc [1 2 3 4]) ;; => [2 3 4 5]
(filter even? [1 2 3 4]) ;; => [2 4]
(reduce + [1 2 3 4])  ;; => 10
```

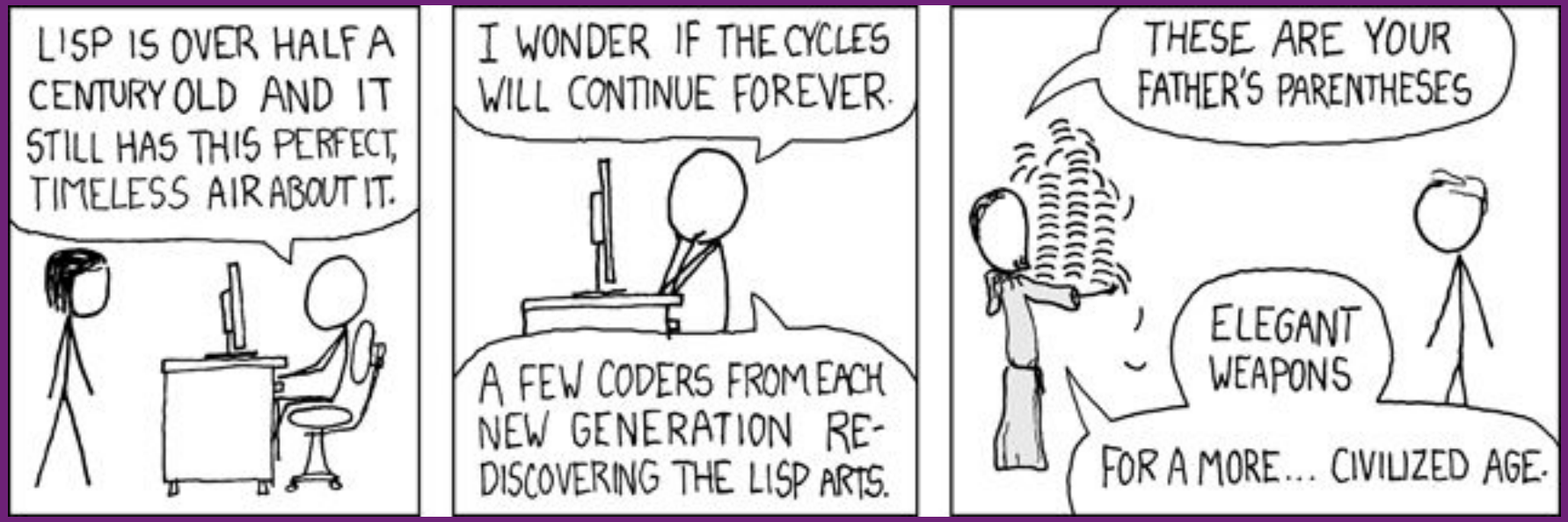
Operações em sequências!

Clojure

https://clojure.org

LISP

- Parênteses!
- Code is Data
- Primeira aparição em 1958
- Seu nome vem de LISt Processing
- Common Lisp, Scheme, Clojure (entre outros)!


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

JS

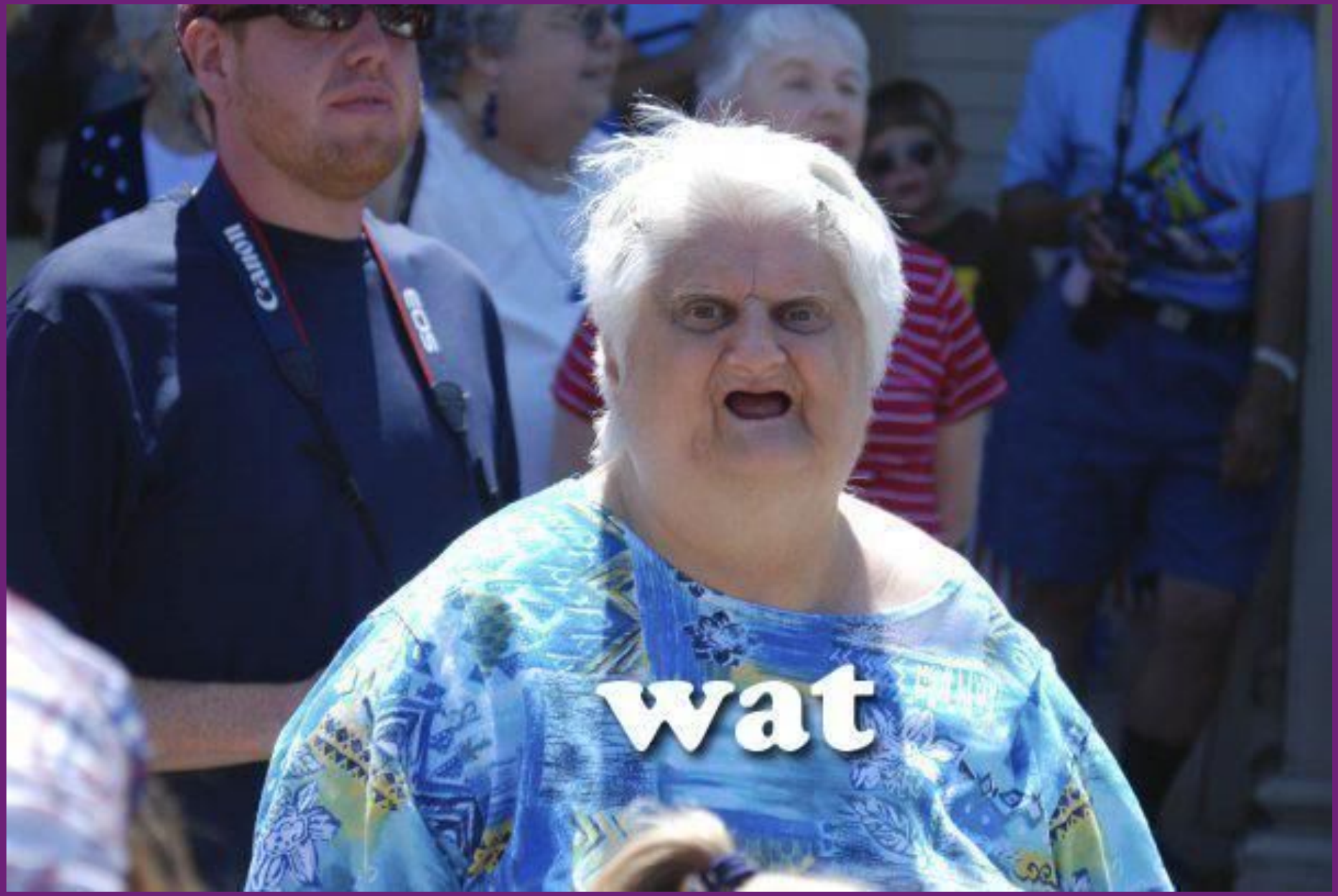
```
function(arg1, arg2)
```

Clojure

```
(function arg1 arg2)
```

Clojure é um *dialeto* LISP

Dialeto é uma variante linguística constituída por características fonológicas, sintáticas, semânticas e morfológicas próprias.


wat

Clojure segue a mesma estrutura gramatical e sintática de um LISP, porém possui suas características próprias de execução e padrões

Exemplo (Fatorial)

```
(defn factorial [n]
  (if (= n 0)
    1
    (* n (factorial (dec n)))))
```

- Roda na JVM, tem interop com outras linguagens que também rodam nela (Java, Scala, Groovy, etc...)
- Dinâmica
- REPL Driven Development
- Criada por Rich Hickey
- Simples e intuitiva!

Tipos

```clojure
1                ;; long (interger)
1.2              ;; double (float)
"text"           ;; string
true             ;; boolean
:hey             ;; keywords
1/3              ;; Ratio
1N               ;; BigInteger
10.24M           ;; BigDecimal
(1 2 3)          ;; List
[1 2 3]          ;; Vector
{:a "ola"}       ;; Maps
#{1 2 3}         ;; Sets
(fn [])          ;; Funções
nil              ;; nil
```

"It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures."

- **Alan Perlis**, first recipient of the Turing Award

Números

BigDecimal e **BigInteger** são nativos do Clojure, você pode fazer operações com extrema alta precisão usando todos métodos e operadores nativos. (muito útil para uma instituição financeira)

Ratio é um tipo de Clojure que permite expressar frações e proporções com precisão infinita, como por exemplo **1/3**

Funções

Funções em Clojure podem ser criadas usando `fn`

```
(fn [arg] body)
(def my-fn (fn [arg] body))
(defn my-fn [arg] body)
#(inc %) ;; => (fn [a] (inc a))
```

Mapas

Mapas `{}` são usados muito em Clojure para carregar valores juntos (o que geralmente seria um objeto), mas ele contém apenas os valores imutáveis e não operações que alteram seu estado.

Geralmente **keywords** são usadas como chaves para os mapas

```
{:name           "John"
 :account-number 3456
 :branch          1}
```

Sequences

Clojure conta com 2 tipos para sequências: `List '()` e **Vector [ ]**

Ambas são bem semelhantes e funcionam de forma da mesma maneira em diferentes funções e cenários, porém existem algumas diferenças de comportamentos, como por exemplo:

`Lists` tem elementos adicionados no seu inícios, `Vectors` tem elementos adicionados no final, quando usando a função `conj`

Sequências são elementos fundamentais em programação funcional, e em Clojure. Processar e transformar sequências de dados é parte central de muitos programas

Blocos

```
(def a 2)
(def incrementa (fn [a] (+ a 1)))
(defn incrementa [a] (+ a 1))

(if (pos? age)
  :verdade
  :falso) ;; :verdade

(let [a 10
      b (+ a 10)]     ;; => b = 20
  (+ (incrementa a)
     (incrementa b))) ;; => 32
```

Recursão

Recursão é uma ferramenta importante em programação funcional, pois permite que executemos "loops" sem alterar o valor das variáveis

```
(defn factorial [n]
  (if (= n 0)
    1
    (* n (factorial (dec n)))))

(defn tail-recur [sum cnt]
   (if (= cnt 0)
      sum
      (recur (+ cnt sum) (dec cnt))))
```

Loop

Permite realizar loops dentro de funções, como se tivéssemos uma recursão localizada.

```
(loop [value 0]
  (if (< value 10)
    (recur (inc value))
    (println "Done")))
```

Dica!

Prefira sempre utilizar sempre que possível funções como **map**, **filter**, **reduce**, etc... E composições delas, ao invés de recursão e **loop**.

Elas deixam seu código mais limpo, funcional e fácil de entender. Recursão deve ser utilizada quando ela for realmente a melhor opção (ou única)

Keywords

keywords são um tipo muito usado em Clojure, geralmente são usadas como chaves de mapas, ou identificadores (como valores de um Enum)

keywords podem (e muitas vezes são) usadas com namespaces, como por exemplo: `:person/name`. Isso ajuda a qualificar a keywords

keywords também são funções! Você pode acessar valores de um mapa usando elas:

```
(:name {:name "John"}) => "John"
```

Sets

sets #{1 2 3} são coleções como sequências, só que sem uma ordem definida, e não podem ter elementos repetidos.

sets também são funções! Usando um set, podemos saber se um elemento está contido nele:

```
(#{1 2 3} 1) => 1

(#{1 2 3} 5) => nil
```

Code is Data (Homoiconicity)

O código Clojure em sí é uma sequência de símbolos em uma **list**

Sets

São funções que operam sobre a estrutura do código para modificá-lo antes da compilação ou execução

Exemplo: Thread Last Macro `->>`

```
(->> [1 2 3]
     (map inc)
     (filter even?))
 ;; => (filter even? (map inc [1 2 3]))
```

DEMO

```
(defn all-bills [account as-of db http rollout routes]
 (let [[latest & historical] (latest+historical-bills account as-of db http)
       open-bill             (open-bill-w-cache account as-of db http)
       future-bills          (->> (future-bills account as-of db http)
                                  :bills
                                  (time/sort-latest-first :bill/due-date))
       card-interests        (c-acc/all-card-interests (:account/id account) http)
       future+open           (conj (vec future-bills) (:bill open-bill))
       future+open-wire      (mapv #(a-bill/bill->full-preview-wire % account card-interests routes) future+open)
       historical-wire       (mapv #(a-bill/bill->full-preview-wire % account card-interests routes) historical)]
   (if latest
     (let [financing-info (http/financing-info latest account http)
           latest-wire    (a-bill/latest-bill->full-preview-wire latest account card-interests financing-info
(:line-items open-bill) routes)]
       (into (conj future+open-wire latest-wire) historical-wire))
     (into future+open-wire historical-wire))))
```

Interop Java

Clojure consegue realizar interop com Java, e com isso ganhamos todo acervo que a JVM nos fornece.

```
(let [d (java.util.Date.)]
  (.getTime d))  ; ⇒ 1349819873183

(Math/floor 5.677)          ; ⇒ 5.0
(Boolean/valueOf "false")   ; ⇒ false
(Boolean/valueOf "true")    ; ⇒ true
```

(não abuse do interop - use somente quando necessário - Clojure fornece uma biblioteca repleta de recursos)

Concorrência e Paralelismo

Clojure é feito pensando em concorrência, ele nos provê nativamente várias funções para ajudar com isso!

- `pmap` -> Como o `map` só que paraleliza automaticamente o processamento
- `future`
- `promise`

Mas…
E quando precisamos de estado?

Estado Mutável

Quando precisamos trabalhar com estado mutável, Clojure possui estruturas de dados próprias para isso, como por exemplo o atom

```
(def car (atom {:make "Audi"
                :model "Q3"}))

@car
;;{:make "Audi", :model "Q3"}

(swap! car assoc :model "Q5")
;;{:make "Audi", :model "Q5"}

(reset! car {:make "" :model ""})
;;{:make "", :model ""}
```