

Programação Funcional

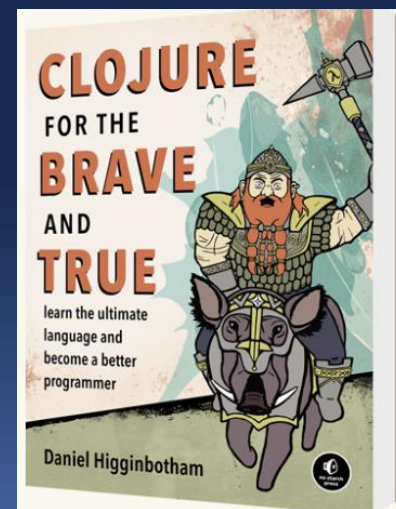
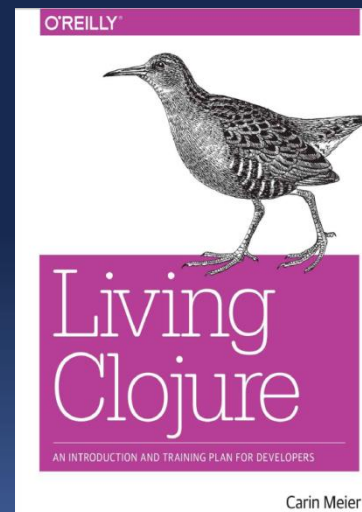
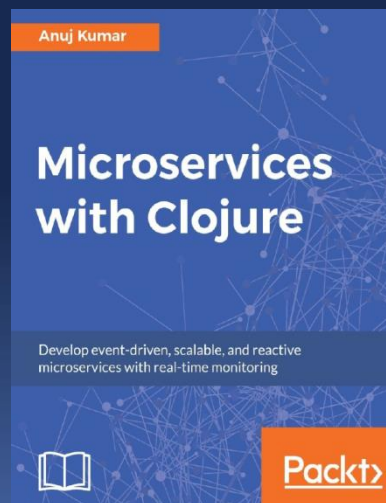
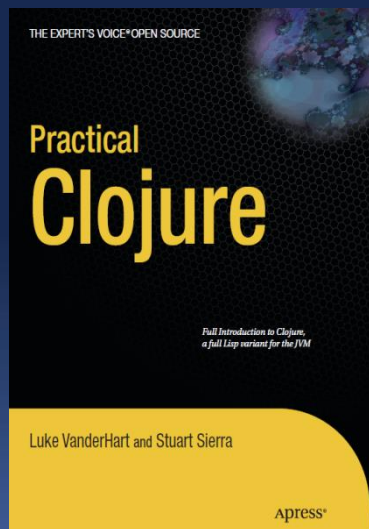
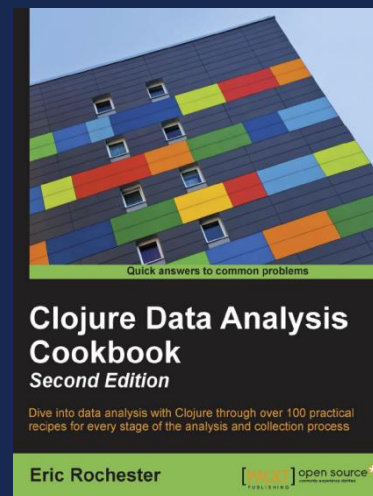
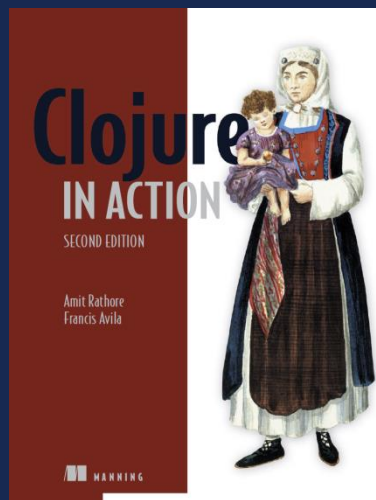
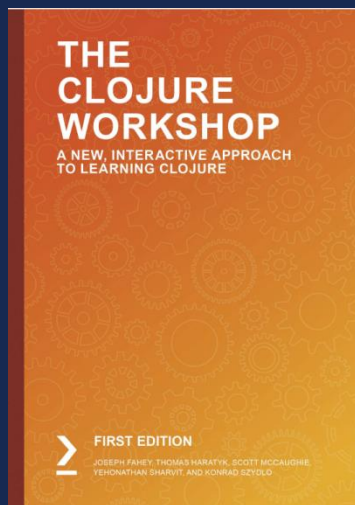
Unidade 8 – Recursão e Looping



Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecido.freitas@prof.uscs.edu.br
aparecidovfreitas@gmail.com



Bibliografia



Introdução

- ✓ Dados a serem manipulados em programas **nem sempre** tomam o formato **linear** no qual as funções **map** ou **reduce** são particularmente adaptadas;
- ✓ As técnicas vistas no último capítulo **não** tratam de travessias não-lineares em estruturas como **árvores** ou **grafos**.
- ✓ No entanto, a linguagem **Clojure** fornece também ferramentas que oferecem mais controle ao programador, as quais baseiam-se em **recursão**.



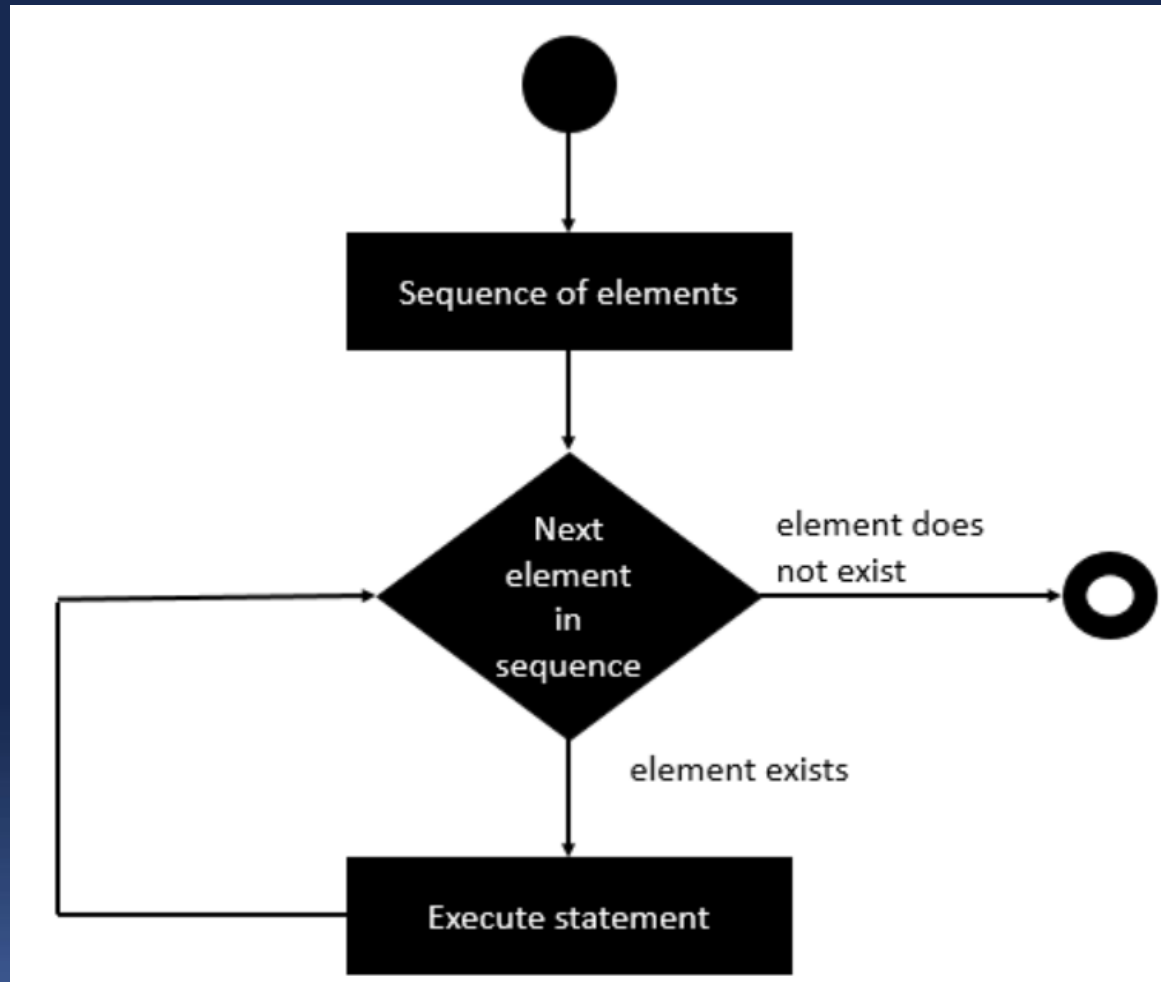
Macro `doseq`

- ✓ É uma das alternativas procedurais mais próximas para loopings;
- ✓ Assemelha-se ao comando **`foreach`** encontrado em outras linguagens de programação;
- ✓ É basicamente empregada para **iterar** sobre uma sequência.

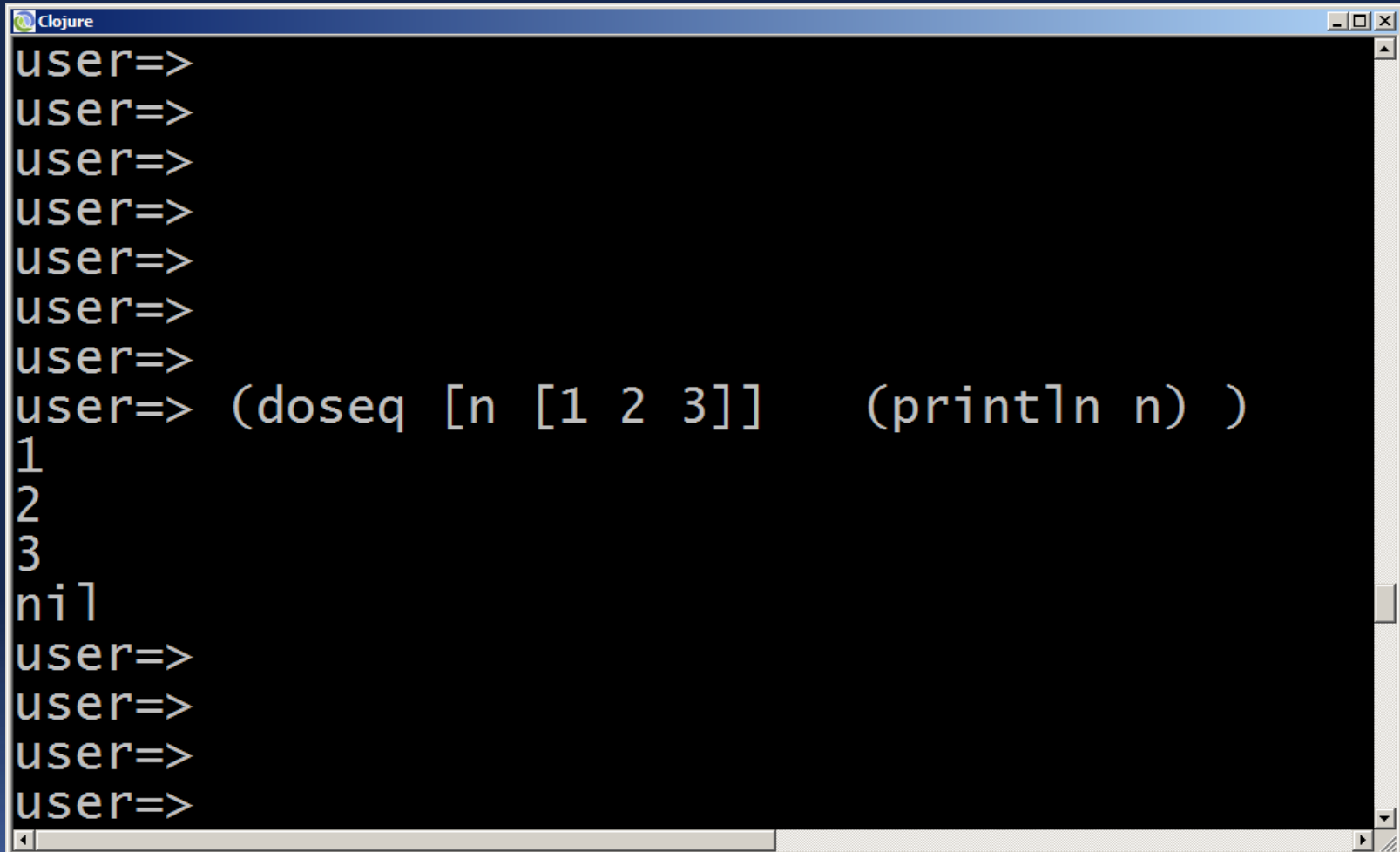
```
(doseq (sequence)  
      statement#1)
```



Macro doseq



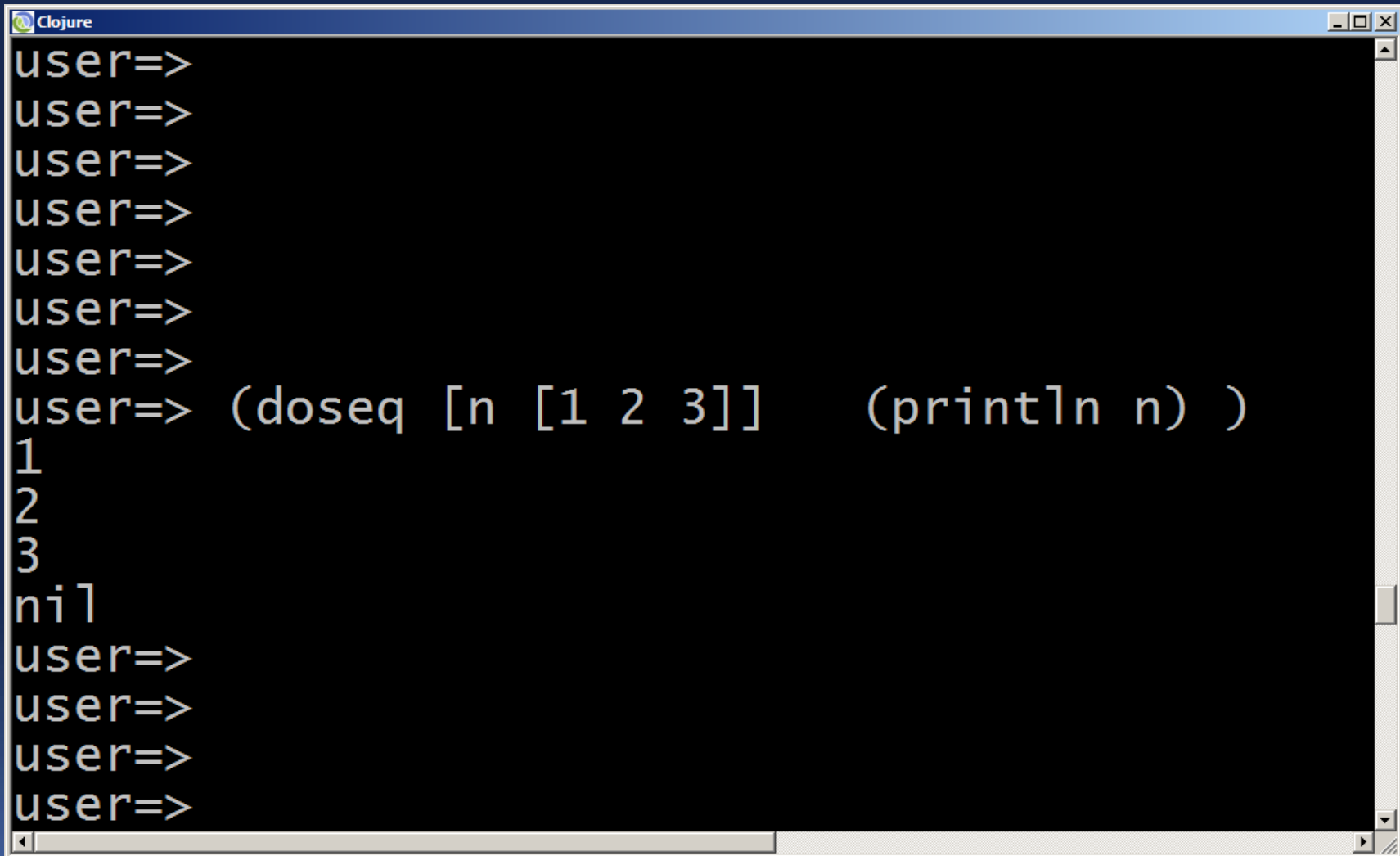
Macro doseq



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (doseq [n [1 2 3]] (println n) )
1
2
3
nil
user=>
user=>
user=>
user=>
```



Macro doseq



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (doseq [n [1 2 3]] (println n) )
1
2
3
nil
user=>
user=>
user=>
user=>
```



Macro doseq

Recursao_01.dj

```
(ns myns)
```

```
(doseq [ n (range 5)]  
  (println (str "Linha " n)))
```

```
)
```

```
nil
```

Chlorine REPL

```
“ Linha 0
```

```
Linha 1
```

```
Linha 2
```

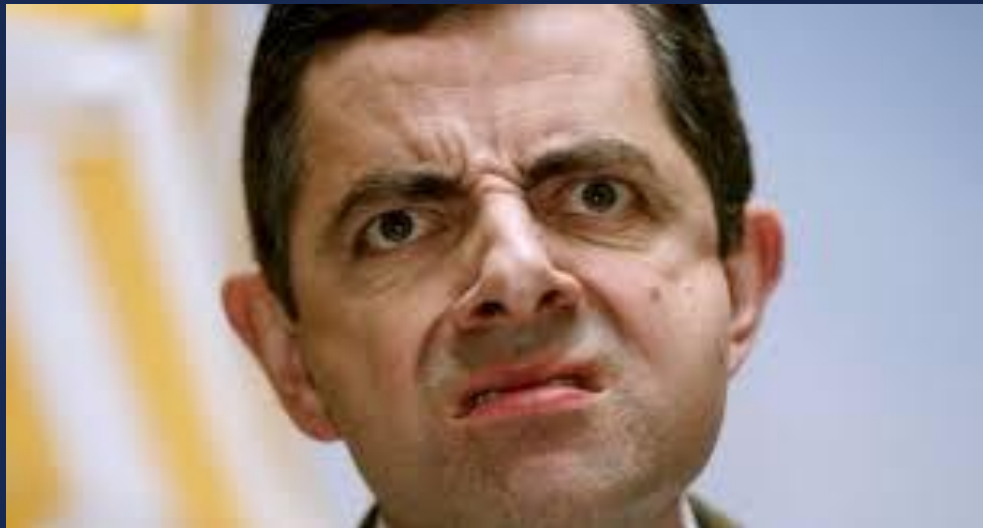
```
Linha 3
```

```
Linha 4
```

```
<> nil
```



Vê ... Não entendi !!! Porque `doseq` retorna `nil` ?



Recursao_01.dj	Chlorine REPL
<pre>(ns myns)</pre>	“ Linha 0
	Linha 1
	Linha 2
	Linha 3
	Linha 4
<pre>(doseq [n (range 5)] (println (str "Linha " n)))</pre>	<> nil
<pre>nil</pre>	

Porque `doseq` retorna `nil`?

- ✓ Traduzindo a função: 'Para cada inteiro `n` de 0 a 4, imprima um string com a palavra "Linha" e o inteiro `n`.'
- ✓ A função `doseq` sempre retorna `nil`.
- ✓ Ou seja, `doseq` não produz nenhum valor;
- ✓ Seu propósito é apenas executar "*side effects*"

```
Recursao_01.clj                                     Chlorine REPL

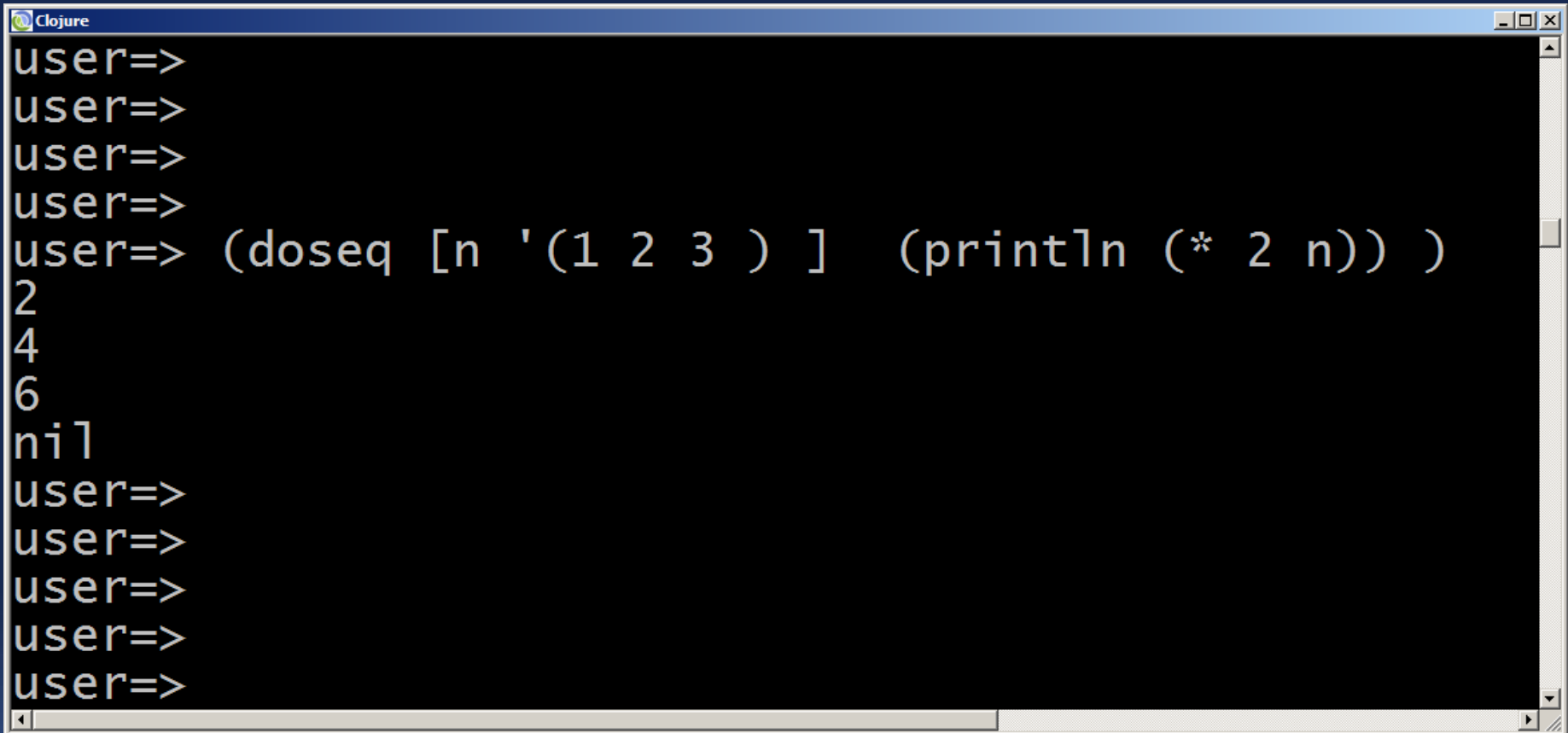
(ns myns)

(doseq [ n (range 5)]
  (println (str "Linha " n)))
)
nil
```

```
“ Linha 0
  Linha 1
  Linha 2
  Linha 3
  Linha 4
<> nil
```



Porque `doseq` retorna **`nil`** ?



```
Clojure
user=>
user=>
user=>
user=>
user=> (doseq [n '(1 2 3)] (println (* 2 n)))
2
4
6
nil
user=>
user=>
user=>
user=>
user=>
```

Vê ... Ao invés de usar `doseq`
eu não poderia usar `map` ?



Código alternativo com map

- ✓ Como foi visto no capítulo anterior a função **map** é **lazy**. Isso significa que **não** se pode garantir que a sequência inteira será executada;
- ✓ Em contrapartida **doseq** **não** é lazy;
- ✓ Com **map**, **filter**, **reduce** e todas as outras funções de tratamento de sequência, deve-se sempre tentar usar **funções puras**, ou seja, funções sem "**side effects**".



Código alternativo com map

Recursao_01.cj

Recursao_02.cj

Chlorine REPL

```
(ns myns)
```

```
(map (fn [i] (println "Linha " i)) (range 5))  
>(nil nil nil nil nil)
```

```
“ Linha 0  
Linha 1  
Linha 2  
Linha 3  
Linha 4
```

```
<> >(nil nil nil nil nil)
```



Exemplo doseq

Recursao_03.clj

```
(ns myns)
```

```
(doseq [n (range 5)]
```

```
  (when (odd? n)
```

```
    (println (str "Linha " n))))
```

```
)
```

```
nil
```

Chlorine REPL

```
“ Linha 1  
  Linha 3
```

```
<> nil
```



Código equivalente com filter

Recurso_04.cj

Chlorine REPL

```
(ns myns)
```

```
“ Linha 1  
  Linha 3
```

```
(doseq [n (filter odd? (range 5)) ]  
  (println (str "Linha " n))  
)
```

```
<> nil
```

```
nil
```



Qual dos dois códigos é mais aconselhável de usar ?



Qual dos dois códigos é mais aconselhável de usar ?



```
Recursao_03.cj  
  
(ns myns)  
  
(doseq [n (range 5)]  
  (when (odd? n)  
    (println (str "Linha " n))))  
  
nil
```

Chlorine REPL

```
“ Linha 1  
  Linha 3  
<> nil
```



```
Recursao_04.cj  
  
(ns myns)  
  
(doseq [n (filter odd? (range 5)) ]  
  (println (str "Linha " n))  
)  
  
nil
```

Chlorine REPL

```
“ Linha 1  
  Linha 3  
<> nil
```



Qual dos dois códigos é mais aconselhável de usar ?

- ✓ **Não** há nada de **errado** com os códigos apresentados;
- ✓ Entretanto, uma **boa prática** na Programação Funcional é remover-se tanto quanto possível a lógica interna ao corpo da função **doseq**;



Qual dos dois códigos é mais aconselhável de usar ?

- ✓ Com a **remoção** do código interno ao corpo da função **doseq**, organiza-se melhor o código e também prepara-se o código para que num futuro se possa incluir, se necessário, linhas de código internas ao corpo da função;
- ✓ Se isso for necessário, o código já estará escrito de forma apropriada e, possivelmente, se beneficiando da "**lazy evaluation**".



Looping Shortcut

- ✓ Como boa prática, deve-se **evitar** a escrita de loops reais;
- ✓ **Clojure** provê funções interessantes que podem auxiliar casos simples no qual o que se deseja é simplesmente executar-se uma repetição;



Looping Shortcut

- ✓ Algumas dessas funções podem retornar sequências **lazy**;
- ✓ Um exemplo dessa função é **repeat**.

Recursao_05.cj

Chlorine REPL

```
(ns myns)
```

```
<> >("Hello..." "Hello..." "Hello..." "Hello..." "Hello...")
```

```
(take 5 (repeat "Hello..."))
```



Recursão

Recursao_06.cj

Chlorine REPL

```
(ns myns)
```

```
<> 12
```

```
(defn soma-recursiva [ n lista-numeros]
  (if (first lista-numeros)
      (soma-recursiva (+ n (first lista-numeros))
                      (rest lista-numeros) )
      n
  )
)
#'myns/soma-recursiva ...
```

```
(soma-recursiva 0 [1 3 8])
```

```
12
```

