

Programação Funcional

Unidade 7 – Mapping , Filtering e Reducing



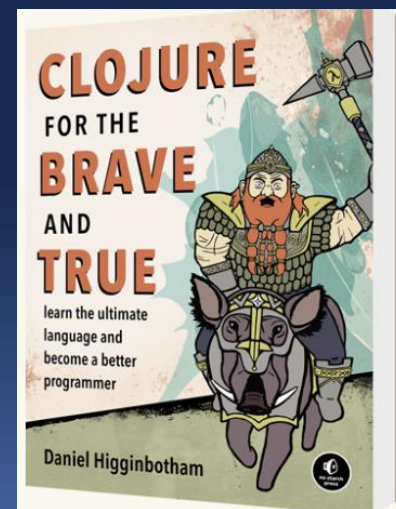
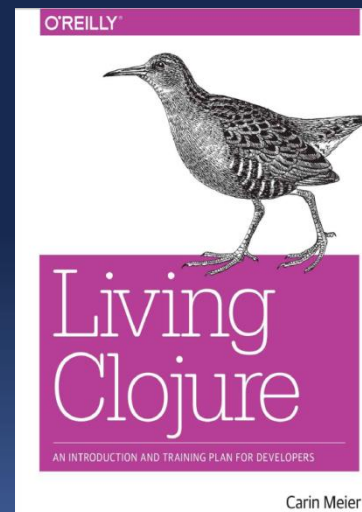
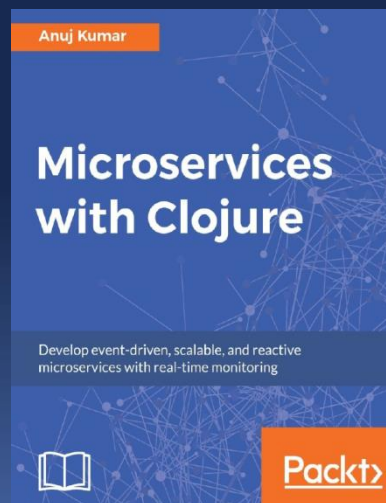
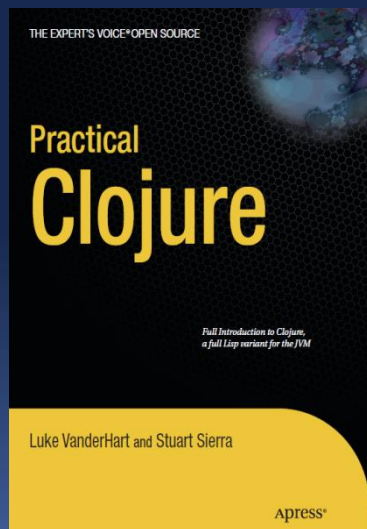
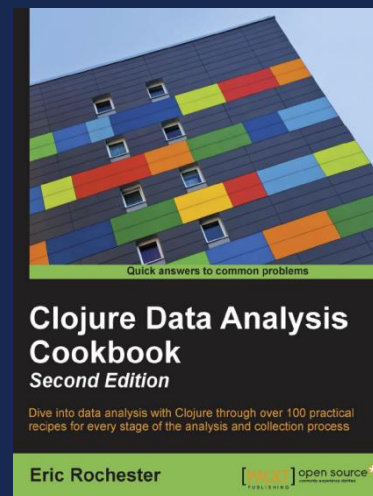
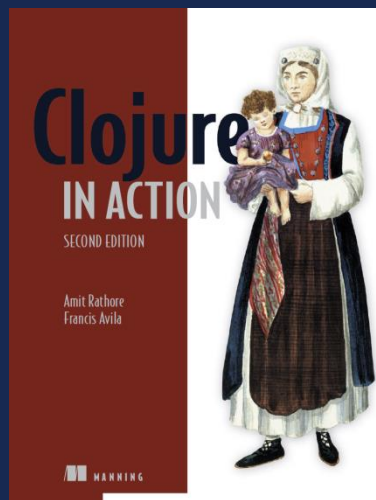
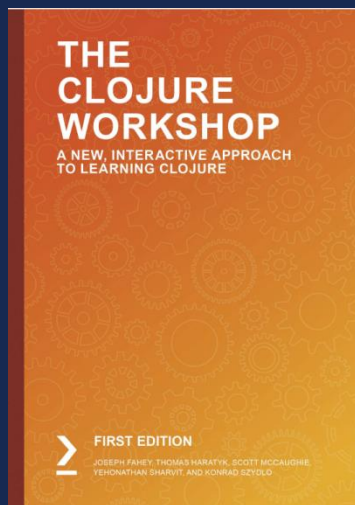
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecido.freitas@prof.uscs.edu.br
aparecidovfreitas@gmail.com



Revisão Técnica: Mauricio Szabo
mauricio.szabo@gmail.com

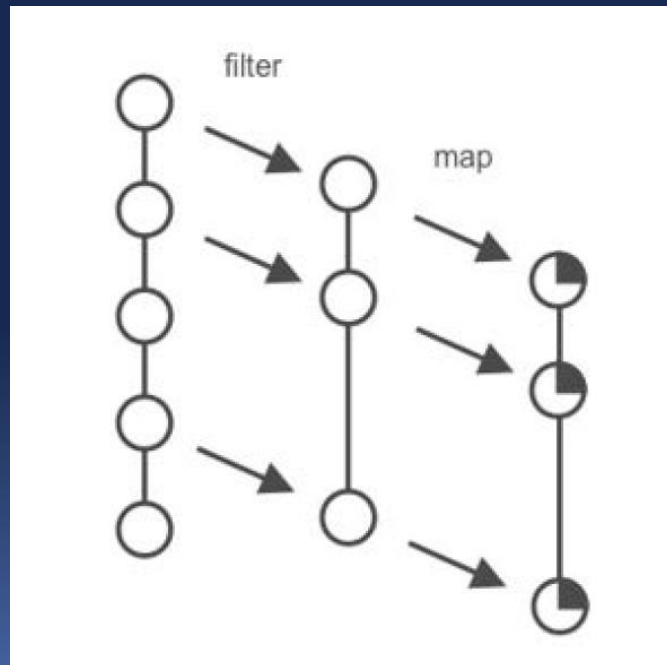


Bibliografia



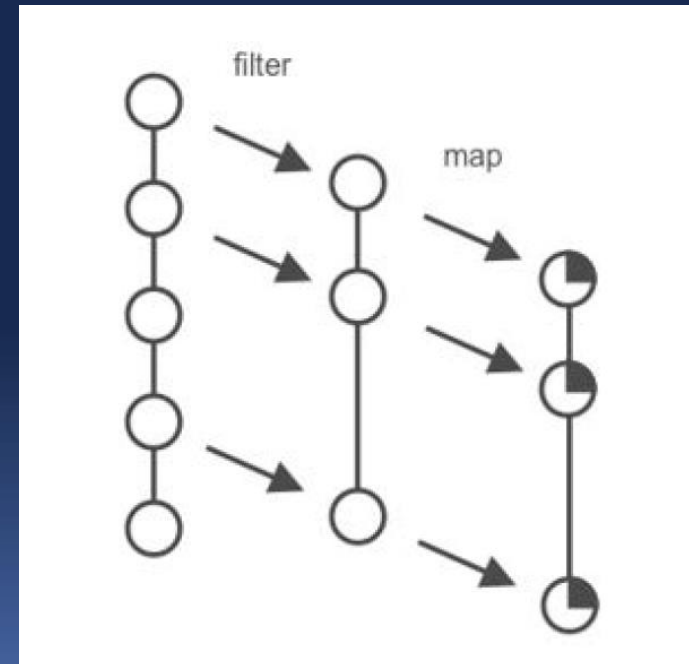
Introdução

- ✓ Funções **map** e **filter** fazem parte fundamental de um grupo maior de funções para tratar sequências;
- ✓ A finalidade destas funções é **modificar sequências**;
- ✓ Aceitam uma ou mais sequências como input e retornam outra sequência.



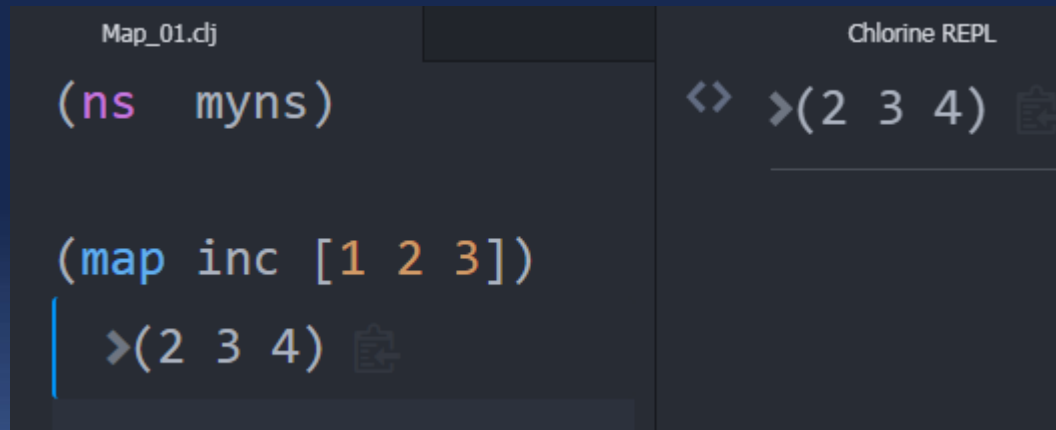
Introdução

- ✓ Funções **map** e **filter** fazem parte fundamental de um grupo maior de funções para tratar sequências;
- ✓ A finalidade destas funções é modificar sequências;
- ✓ Aceitam uma ou mais sequências como input e retornam outra sequência;
- ✓ A figura mostra **map** e **filter** trabalhando juntos. **Filter elimina** itens da lista original enquanto que **map** os **modifica**.



map

- ✓ Como a maioria das funções em **Clojure** para trabalhar com sequências, o primeiro argumento de **map** sempre é uma função;
- ✓ A função fornecida será chamada em cada item da sequência a ser iterada;



```
Map_01.cj  
(ns myns)  
  
(map inc [1 2 3])  
=>(2 3 4)
```

The screenshot shows a Clojure REPL environment. The left pane, titled 'Map_01.cj', contains the code: `(ns myns)`, `(map inc [1 2 3])`, and the result `>(2 3 4)`. The right pane, titled 'Chlorine REPL', shows the same result `>(2 3 4)` with a copy icon.

map

```
Map_02.clj                                     Chlorine REPL
(ns myns)

(defn produto-10 [i] (* i 10))

(map produto-10 [ 1 2 3 4 5 ])
>(10 20 30 40 50)
```

- ✓ Neste exemplo, a função **map** simplesmente aplica a função **produto-10** à todos os elementos da lista de inteiros passada como parâmetro;
- ✓ O mapeamento produz, como consequência, uma **nova lista** dos elementos multiplicados por **10**;
- ✓ A equivalência **um-para-um** é óbvia, mas é uma característica chave do **map**;
- ✓ Com **map**, a sequência produzida tem sempre o mesmo tamanho da sequência de entrada.



map

```
Map_02.clj                                     Chlorine REPL
(ns myns)

(map (fn [i] (* i 10)) [1 2 3 4 5])
>(10 20 30 40 50)
```

- ✓ O mesmo exemplo anterior, porém o **mapeamento** é feito com **função anônima**.

1	—	<i>fn</i>	→	10
2	—	<i>fn</i>	→	20
3	—	<i>fn</i>	→	30
4	—	<i>fn</i>	→	40
5	—	<i>fn</i>	→	50



map

- ✓ Ao se trabalhar com sequências, a função **count** tem grande aplicabilidade;
- ✓ Considerando que **Clojure** considera um **string** como uma lista de caracteres, a função **count** também pode ser usada para encontrar o **tamanho** de um **string**.

Map_03.cj

```
(ns myns)
```

```
(map count ["Ola" "alunos" "Tudo bem?"])
```

```
| >(3 6 9) 
```

Chlorine REPL

```
<> >(3 6 9)
```



map – count (outro exemplo)

- ✓ Exemplo anterior com mais legibilidade.

Map_04.cj

```
(ns myns)
```

```
(defn conta-tam-string [s]  
  (str s ": " (count s))  
)
```

```
(map conta-tam-string ["Ola" "alunos" "Tudo bem?"])  
=>("Ola: 3" "alunos: 6" "Tudo bem?: 9")
```

Chlorine REPL

```
<> >("Ola: 3" "alunos: 6" "Tudo bem?: 9")
```

- ✓ Como visto no exemplo, **map** trabalha em cada elemento de uma lista, produzindo um novo valor em uma nova lista.



filter

- ✓ Diferentemente de **map**, **filter** pode, e frequentemente faz, produzir uma sequência de resultados contendo menos elementos que a sequência de entrada;
- ✓ Uma chamada de **filter** parece basicamente com uma chamada de **map**.

```
filter_01.cj x
(ns myns)

(filter keyword? ["a" "xx" 99 4 :a 88 :b true])
>(:a :b)
```



filter

- ✓ Como com **map**, a função que é passada como argumento ao **filter** é aplicada em cada item na sequência;
- ✓ A diferença é que, no caso do **filter**, a função está sendo aplicada como um **predicado**, podendo retornar um valor **true** ou **false**;
- ✓ Quando um valor **true** é retornando, o item em questão será incluído na sequência de resultados.
- ✓ Assim, uma diferença chave com **map** é que o predicado que é providenciado pelo **filter** somente serve para decidir quando o item deve ou não ser incluso na saída.
- ✓ Portanto, com **filter** não se modificam os itens de qualquer maneira;
- ✓ O result set de **filter** sempre é um **subset** da sequência de entrada.



filter

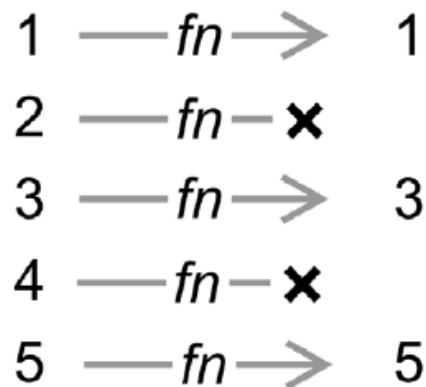
Filter_02.dj	Chlorine REPL
<pre>(ns myns)</pre>	<pre><> true</pre>
<pre>(odd? 7)</pre>	<pre><> false</pre>
<pre> true</pre>	<pre><> >(1 3 5)</pre>
<pre>(odd? 8)</pre>	
<pre> false</pre>	
<pre>(filter odd? [1 2 3 4 5])</pre>	
<pre> >(1 3 5)</pre>	

remove

- ✓ A função **remove** faz exatamente o inverso de **filter**;
- ✓ Quando o predicado retorna **true**, o item é removido.

```
Filter_03.cj  
(ns myns)  
  
(remove odd? [ 1 2 3 4 5 ])  
➤(2 4)
```

Chlorine REPL



constantly

- ✓ A função `constantly` em `Clojure` faz com que o predicado sempre retorne um simples valor.

```
Filter_03.cj | Filter_04.cj | Chlorine REPL
(ns myns)

(filter (constantly true) [ 1 2 3 4 5 ])
>(1 2 3 4 5)

(filter (constantly false) [ 1 2 3 4 5 ])
>()
```

- ✓ Assim, `filter` sempre retorna uma sequência (pode ser tudo ou nada) !



take

- ✓ A função **take** retorna os primeiros n itens de uma lista.

```
Filter_05.cj  
(ns myns)  
  
(take 2 [1 2 3 4 5])  
  >(1 2)
```

Chlorine REPL

<> >(1 2)



drop

- ✓ A função `drop` retorna a lista original sem os primeiros `n` itens.

Filter_06.cj

```
(ns myns)
```

```
(drop 2 [1 2 3 4 5])
```

```
>(3 4 5)
```

Chlorine REPL

```
<> >(3 4 5)
```



take-while

- ✓ A função **take-while** retorna todos os itens da lista enquanto o predicado for **verdadeiro**. A pesquisa é cessada, diferentemente do filter, quando o predicado for false.

```
Filter_06.cj | Filter_07.cj | Chlorine REPL

(ns myns)

(take-while neg? [ -1 -2 -3 9 0 1 -2 -5])
| >(-1 -2 -3)

(take-while neg? [ 0 1 -2 -3 9 0 1 -2 -5])
| >()

(take-while neg? [ 1 2 3 4 5 6 7 ])
| >()

(take-while neg? [ -1 -2 -3 ])
| >(-1 -2 -3)
```



drop-while

- ✓ A função **drop-while** retorna todos os itens **restantes** da lista a partir do primeiro predicado **falso**.

Filter_08.clj	Chlorine REPL
<pre>(ns myns)</pre>	<pre>>(9 0 1 -2 -5)</pre>
<pre>(drop-while neg? [-1 -2 -3 9 0 1 -2 -5])</pre>	<pre>>(0 1 -2 -3 9 0 1 -2 -5)</pre>
<pre> >(9 0 1 -2 -5)</pre>	<pre>>(1 2 3 4 5 6 7)</pre>
<pre>(drop-while neg? [0 1 -2 -3 9 0 1 -2 -5])</pre>	<pre>>()</pre>
<pre> >(0 1 -2 -3 9 0 1 -2 -5)</pre>	
<pre>(drop-while neg? [1 2 3 4 5 6 7])</pre>	
<pre> >(1 2 3 4 5 6 7)</pre>	
<pre>(drop-while neg? [-1 -2 -3])</pre>	
<pre> >()</pre>	





Usando map e filter juntos



Usando map e filter

- ✓ Muito do poder da linguagem **Clojure** nas funções de sequência está na combinação de **map** e **filter**.
- ✓ Embora **map** é escrito no início, a avaliação se inicia com a chamada do **filter**.

```
Filter_09.clj                                     Chlorine REPL
(ns myns)

(map (fn [n] (* 10 n))
     (filter odd? [1 2 3 4 5]))
>(10 30 50)
```



Usando map e filter

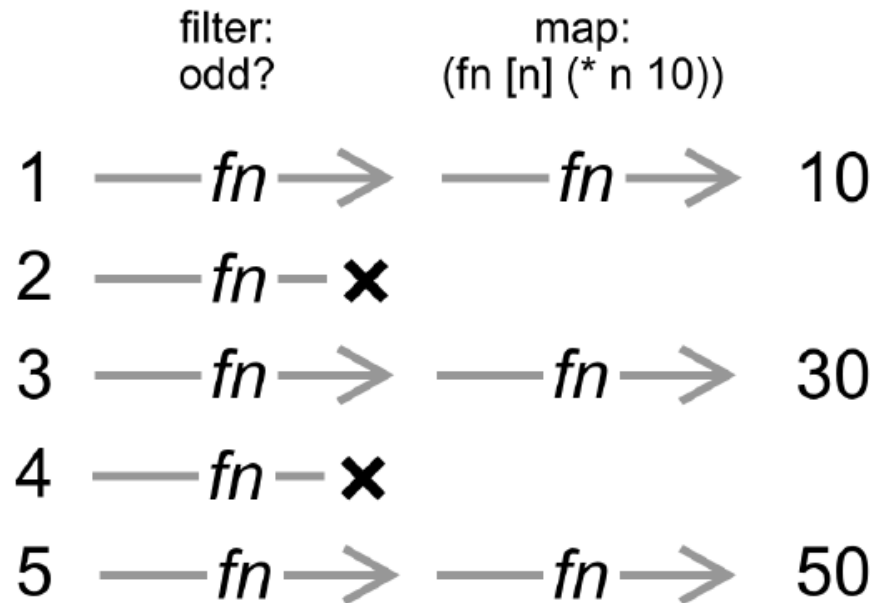
Filter_09.dj

```
(ns myns)
```

```
(map (fn [n] (* 10 n))  
  (filter odd? [1 2 3 4 5]))  
>(10 30 50)
```

Chlorine REPL

```
<> >(10 30 50)
```



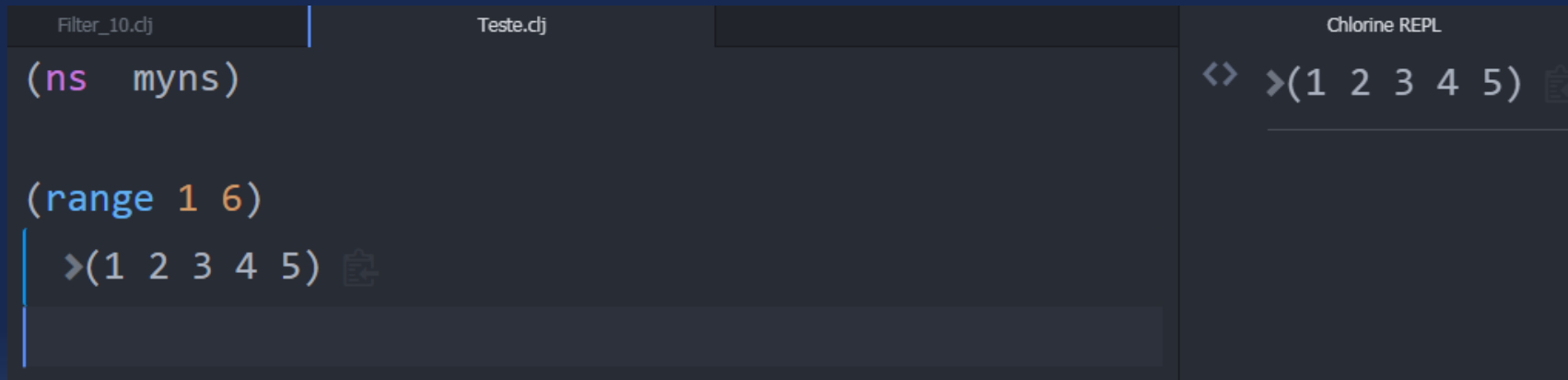
Reescrevendo o exemplo

Filter_09.cj	Filter_10.cj	Chlorine REPL
<pre>(ns myns) (def filtro (filter odd? [1 2 3 4 5])) #'myns/filtro ... (map (fn [n] (* 10 n)) filtro) >(10 30 50)</pre>		<pre><> >(10 30 50)</pre>



Lazy sequence

- ✓ Ao se usar **map** e **filter**, avaliação **lazy** tem uma importante consideração;
- ✓ Nos exemplos anteriores utilizamos como input o **literal vector** [1 2 3 4 5];
- ✓ Poderíamos ao invés de usar o **literal vector**, usar a função **(range 1 6)**.



The screenshot shows a code editor with two tabs: 'Filter_10.cj' and 'Teste.cj'. The 'Teste.cj' tab is active and contains the following Clojure code:

```
(ns myns)  
  
(range 1 6)  
  >(1 2 3 4 5)
```

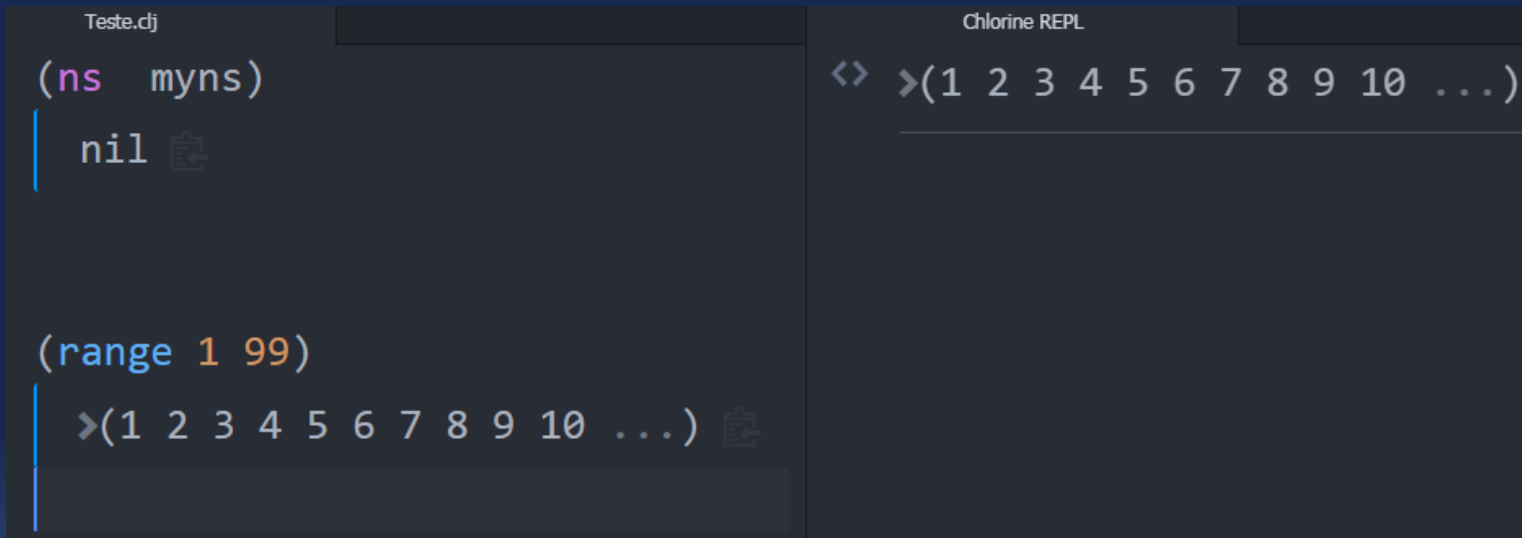
To the right of the code editor is a 'Chlorine REPL' window. It shows the command `>(1 2 3 4 5)` and the output `(1 2 3 4 5)`.

- ✓ A função **range** tem uma importante característica: é **Lazy** !



range é lazy !

- ✓ A função **range** tem uma importante característica: é **Lazy** !
- ✓ A função **range** cria uma lista de inteiros chamando a função **inc** tantas vezes quantas forem necessárias;



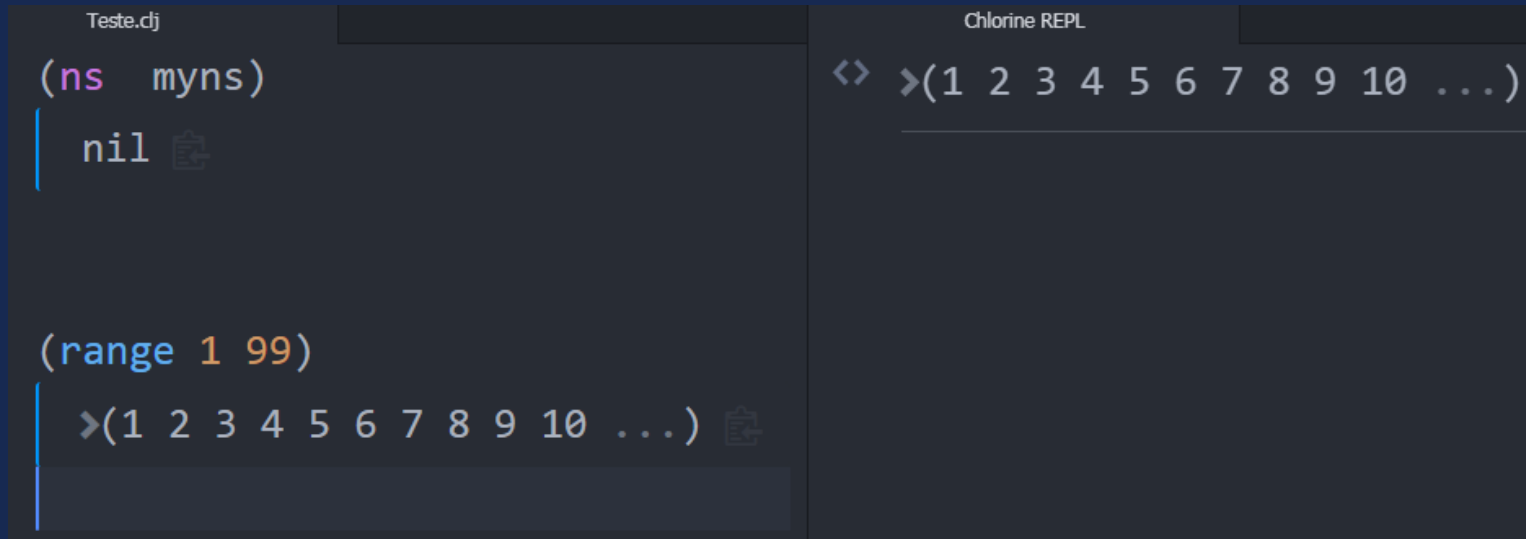
The screenshot shows a REPL environment with two panes. The left pane, titled 'Teste.cj', contains the following code:

```
(ns myns)  
nil  
  
(range 1 99)  
>(1 2 3 4 5 6 7 8 9 10 ...)
```

The right pane, titled 'Chlorine REPL', shows the output of the range function as a lazy sequence:

```
<> >(1 2 3 4 5 6 7 8 9 10 ...)
```


range é lazy !



The screenshot shows a REPL environment with two panes. The left pane, titled 'Teste.clj', contains the following code:

```
(ns myns)

nil

(range 1 99)

>(1 2 3 4 5 6 7 8 9 10 ...)
```

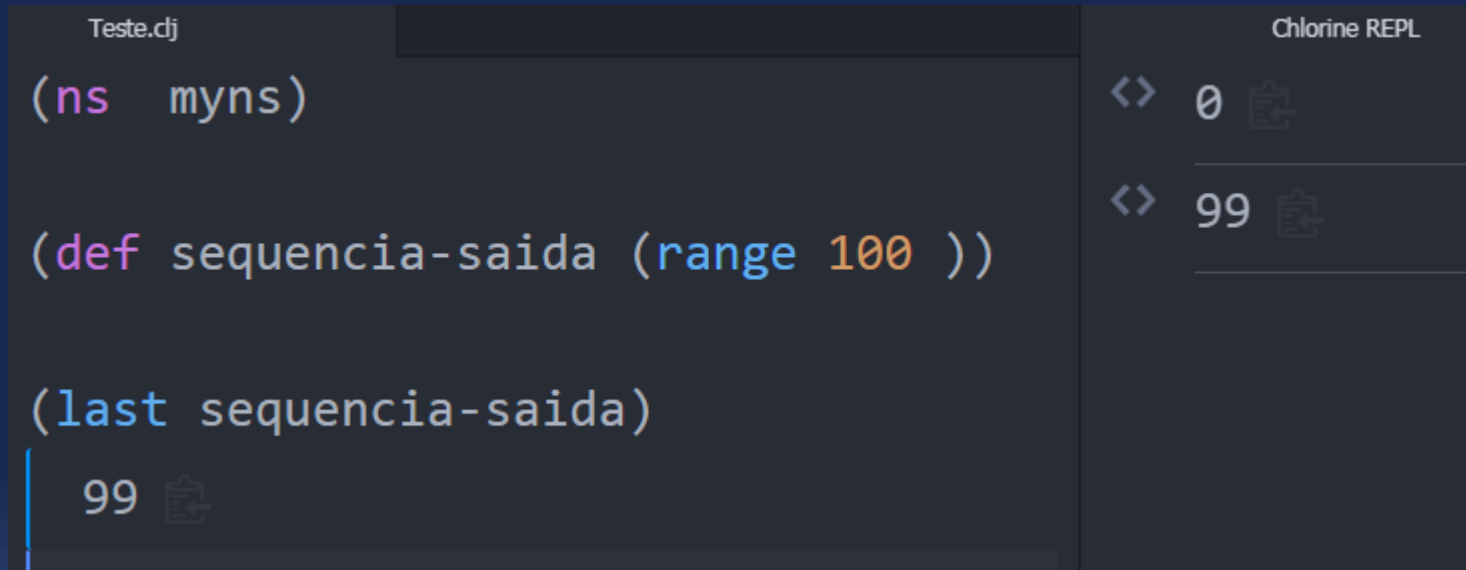
The right pane, titled 'Chlorine REPL', shows the output of the range function as a lazy sequence: `>(1 2 3 4 5 6 7 8 9 10 ...)`. The ellipsis indicates that the sequence is not fully realized.

- ✓ É fácil percebermos que o último elemento é **99**. Mas, esse valor **não** é obtido enquanto toda a aritmética necessária tiver sido executada.
- ✓ Isto significa que quando pesquisamos o primeiro elemento, somente um item é conhecido.



range é lazy !

- ✓ Mas, se pesquisarmos o **último item**, então toda a computação intermediária será executada.



The screenshot shows a Clojure REPL session in a dark-themed editor. The left pane, titled 'Teste.cj', contains the following code: `(ns myns)`, `(def sequencia-saida (range 100))`, and `(last sequencia-saida)`. The right pane, titled 'Chlorine REPL', shows the execution results. The first line shows `<> 0` with a clipboard icon. The second line shows `<> 99` with a clipboard icon, indicating that the `last` function triggered the evaluation of the entire `range 100` sequence.

```
(ns myns)

(def sequencia-saida (range 100 ))

(last sequencia-saida)

99
```



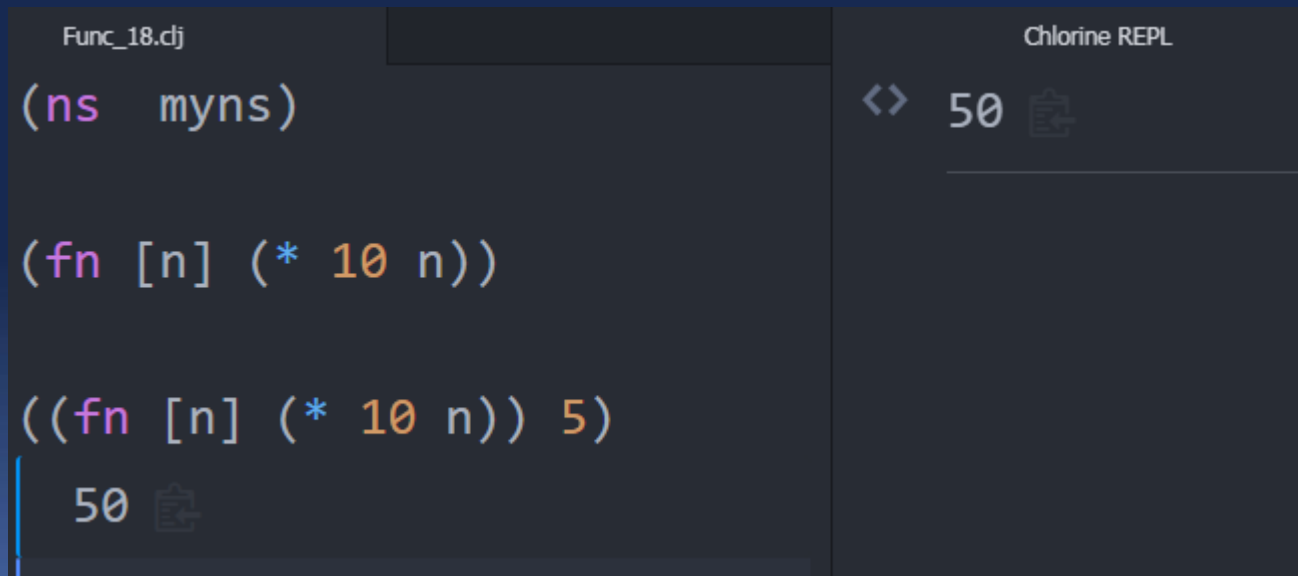
map, filter e remove também são lazy!

- ✓ Isso significa que quando fazemos uma chamada numa sequência **lazy**, **não** será forçada a computação de toda a sequência;
- ✓ Essencialmente, sequências **lazy** efetuam a computação apenas quando necessário;
- ✓ Funções como: **count**, **sort** ou **last**, por outro lado, **não são lazy**;
- ✓ Obviamente, para se contar todos os itens de uma sequência é preciso que se faça uma varredura na lista inteira.



Funções Anônimas

- ✓ Considerando que ao se usar funções de tratamento de sequências, frequentemente passamos funções como parâmetros, em **Clojure** é muito comum que essas funções passadas sejam **anônimas**;
- ✓ A forma canonical **fn** é usualmente empregada para se escrever funções **anônimas**, como no exemplo abaixo:



```
Func_18.cj  
(ns myns)  
  
(fn [n] (* 10 n))  
  
((fn [n] (* 10 n)) 5)  
50
```

Chlorine REPL

<> 50

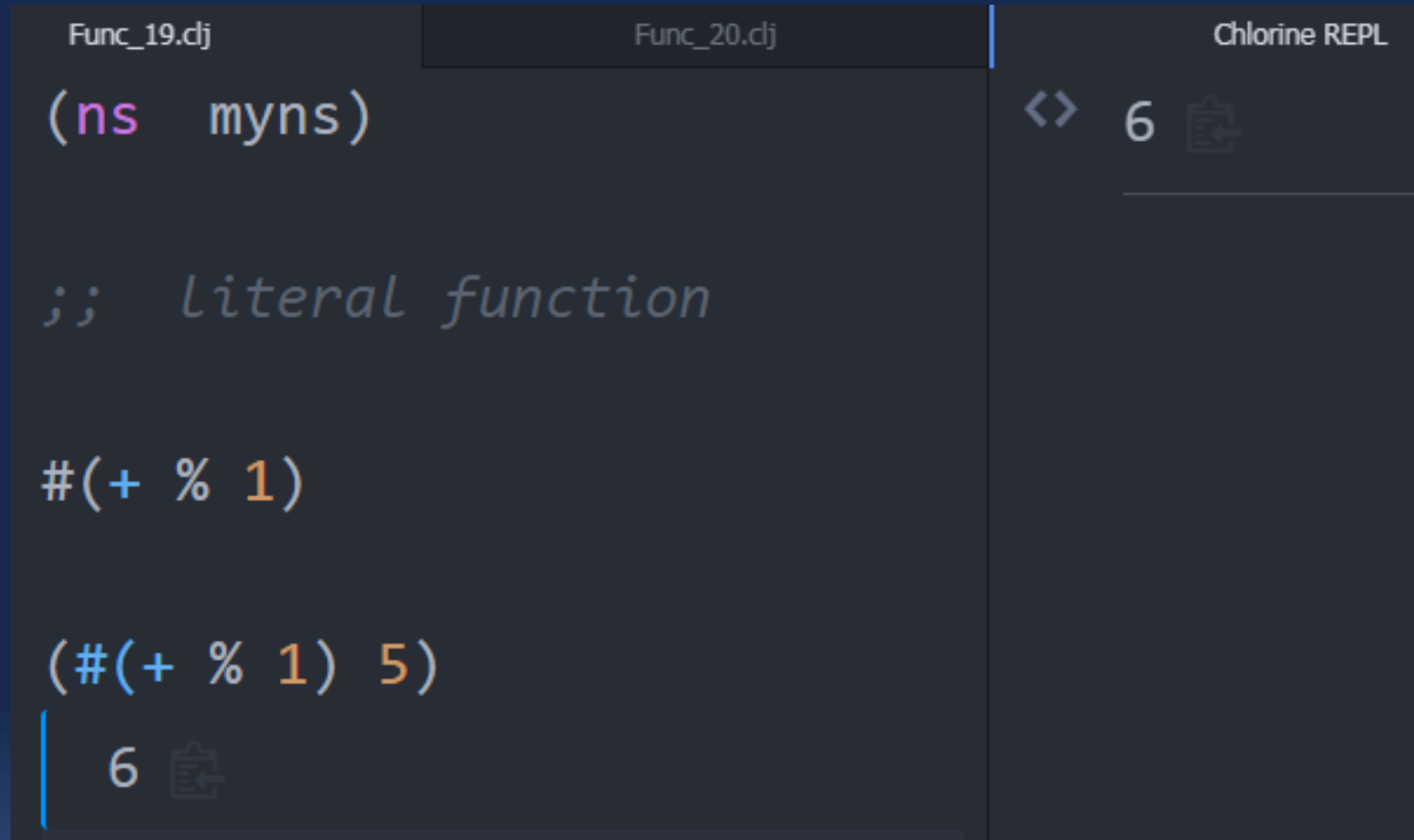


Funções Anônimas – Literal Function

- ✓ Funções anônimas também podem ser escritas por meio de uma **literal function**;
- ✓ Uma **literal function** é uma versão simplificada de `fn`;
- ✓ Com **literal function**, o símbolo `fn` e a lista de argumentos desaparecem;
- ✓ Na notação **literal function** fica somente o coração da função e um operador `#` imediatamente **antes** da abertura dos parênteses;
- ✓ Assim, o operador `#` identifica que o que se segue é uma função anônima;
- ✓ Não se escreve a lista de parâmetros, entretanto adota-se um **padrão** - ao invés de se nomear os parâmetros, como na maioria das funções, os argumentos são automaticamente nomeados com um **pattern**. O primeiro argumento é nomeado com `%` e todos os outros são nomeados automaticamente por `%2` , `%3` , `%4`, e assim por diante.



Funções Anônimas – Literal Function



```
Func_19.cj  Func_20.cj  Chlorine REPL

(ns myns)

;; literal function

#(+ % 1)

(#(+ % 1) 5)
6
```



Funções Anônimas – Literal Function

```
Func_19.clj      Func_20.clj      Chlorine REPL
```

```
(ns myns)
```

```
;; literal function
```

```
#(* (+ % %2) 2)
```

```
(#(* (+ % %2) 2) 6 8)
```

```
28
```

```
<> 28
```



Reducing

- ✓ Funções **map** e **filter** fazem processamento em sequência um para um. Ou seja, map e filter processam uma determinada sequência e produzem outra sequência como saída;
- ✓ Mas, nem sempre é isso que queremos;
- ✓ Muitas vezes, queremos que o resultado da operação à uma determinada sequência seja reduzido a um número, um string, um map, etc;
- ✓ Na Linguagem **Clojure**, a função **reduce** pode ser usada para gerar um resultado a partir de uma determinada sequência.



A base de reduce

- ✓ Para compreendermos como **reduce** trabalha, vamos considerar o exemplo de um código **JavaScript** que retorna a soma de uma lista de inteiros.

```
var integers = [8, 4000, 10, 300];  
var sum = 0;  
for (var i = 0; i < integers.length; i++) {  
    sum = sum + integers[i];  
}  
console.log(sum);
```



A base de reduce

```
var integers = [8, 4000, 10, 300];  
var sum = 0;  
for (var i = 0; i < integers.length; i++) {  
    sum = sum + integers[i];  
}  
console.log(sum);
```

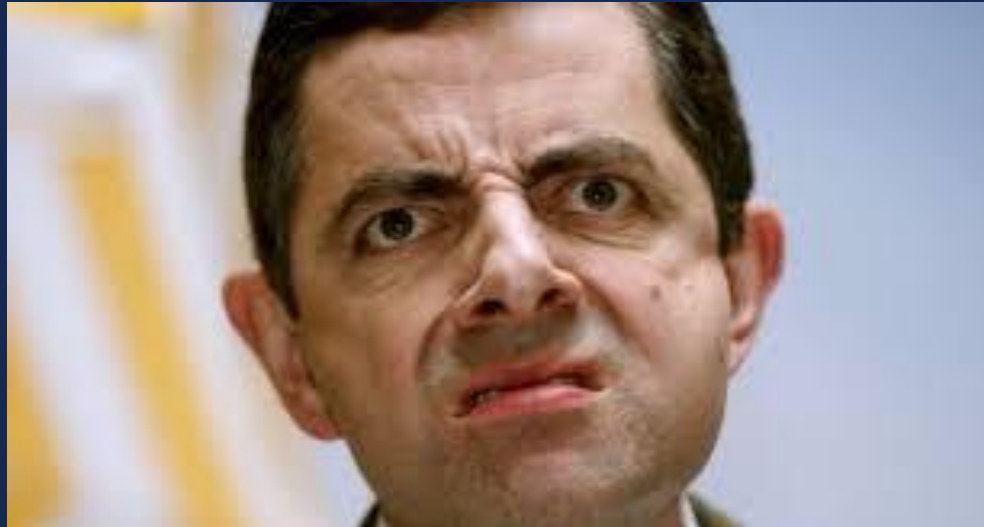
✓ Segue a versão **Clojure** deste exemplo:

```
user> (reduce (fn [sum-so-far item] (+ sum-so-far item)) [8 4000 10 300])  
4318
```



Vê ... Não entendi !!!

Como isso funciona ???



```
user> (reduce (fn [sum-so-far item] (+ sum-so-far item)) [8 4000 10 300])  
4318
```

Como o reduce opera?

- ✓ A expressão tem alguma semelhança com **map** e **filter** vistos anteriormente;
- ✓ Nessa expressão temos uma função (**reduce**), uma função anônima (**fn**) e um **vector** de inteiros;
- ✓ Essa expressão simplesmente adiciona inteiros a partir da sequência fornecida, processando uma iteração sobre os elementos da sequência, de modo semelhante ao **map**;
- ✓ A diferença, no entanto, é que com **reduce**, a função “**se lembra**” do cálculo resultante da avaliação prévia.

```
Reduce_01.dj  
(ns myns)  
  
(reduce (fn [somador item] (+ somador item)) [ 8 4000 10 300])  
4318
```

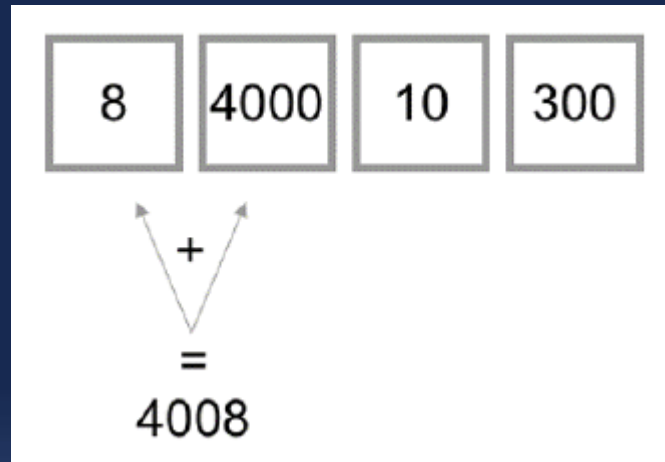
Chlorine REPL

<> 4318



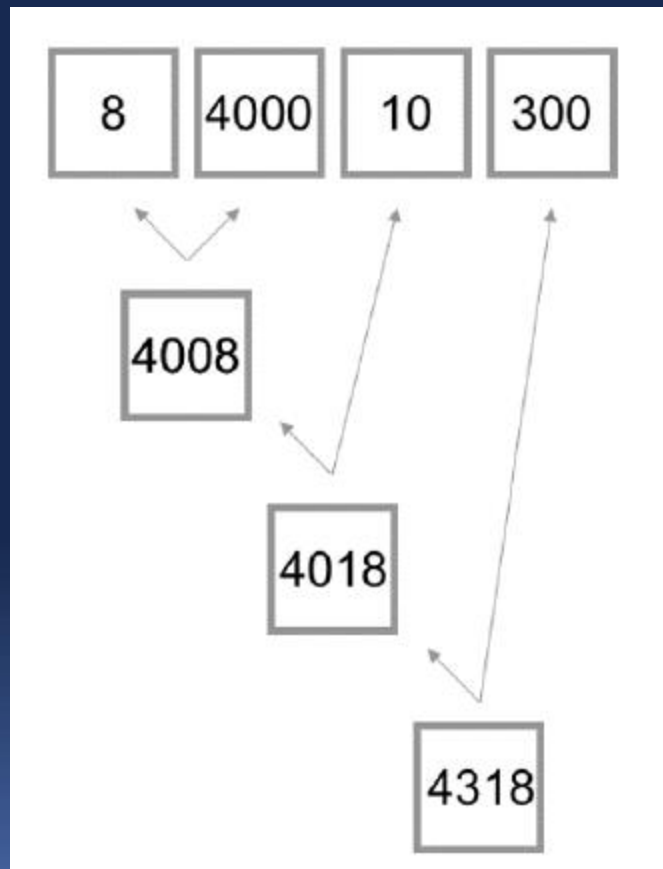
Como o reduce opera?

- ✓ Na primeira vez a função **reduce** faz a chamada da função anônima: `(fn [somador item] (+ somador item))`, sendo os argumentos os primeiros elementos da lista.



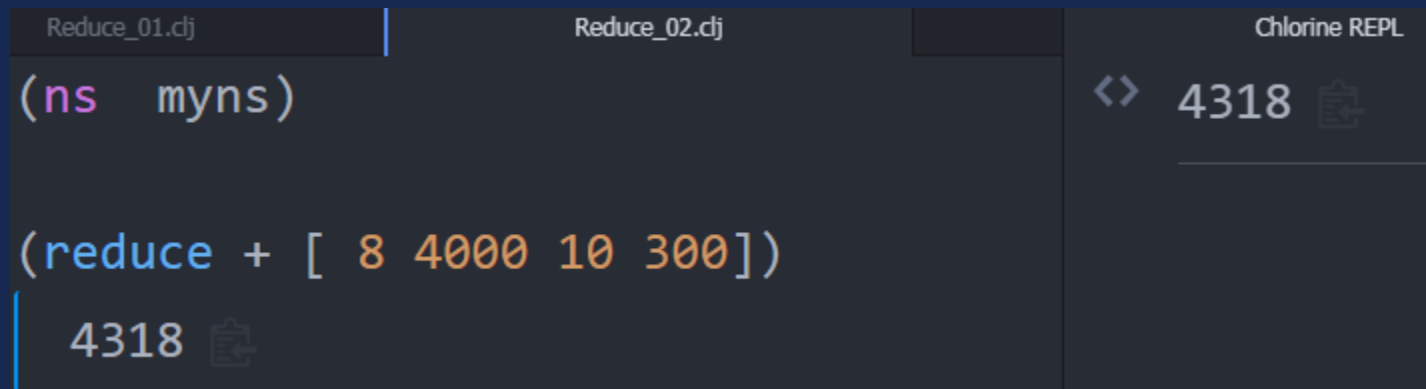
Como o reduce opera?

- ✓ Para cada uma das próximas chamadas, somador corresponde ao resultado do cálculo prévio e item o próximo inteiro da lista.



reduce

- ✓ Poderíamos simplificar a expressão, substituindo a função anônima pela função +;



```
Reduce_01.cj | Reduce_02.cj | Chlorine REPL
(ns myns)
(reduce + [ 8 4000 10 300])
4318
```

The screenshot shows a REPL interface with two tabs: 'Reduce_01.cj' and 'Reduce_02.cj'. The 'Chlorine REPL' panel on the right shows the result '4318' with a copy icon. The code in the editor is a Clojure expression that uses the `reduce` function with the `+` function and a list of numbers to calculate the sum.



reduce

Reduce_03.cj

```
(ns myns)
```

```
(reduce max [ 8 4000 10 300])
```

```
4000
```

Chlorine REPL

```
<> 4000
```



reduce

```
Reduce_03.cj  
(ns myns)  
  
(reduce min [ 8 4000 10 300])  
8
```

Chlorine REPL

<> 4000

<> 8

