

# Programação Funcional

## Unidade 6 – Funções



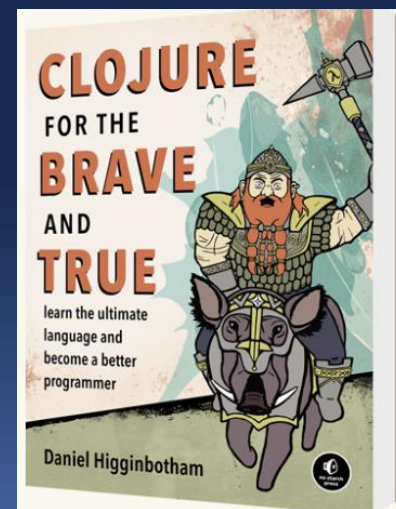
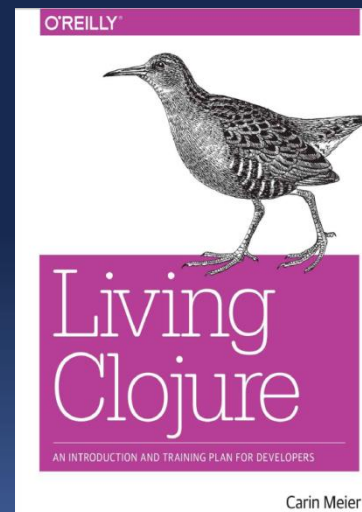
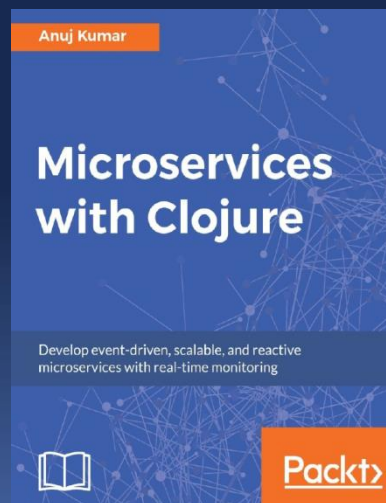
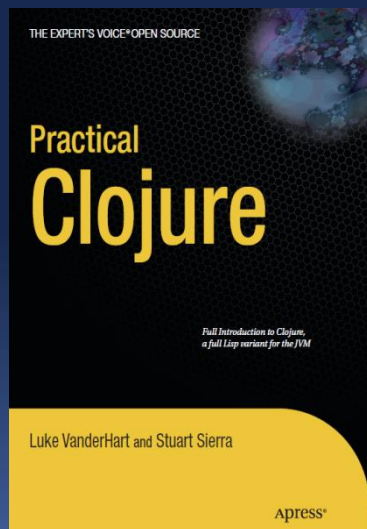
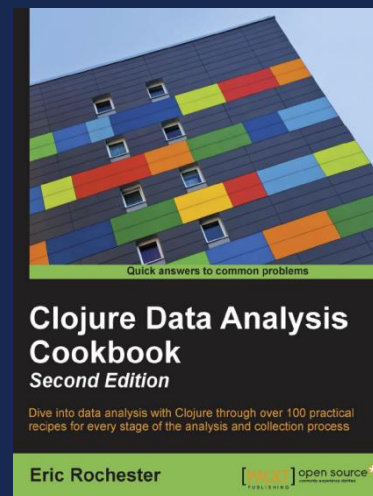
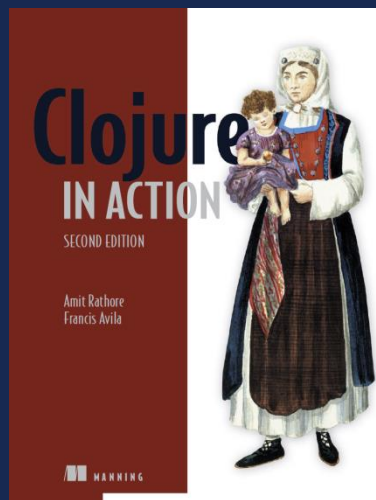
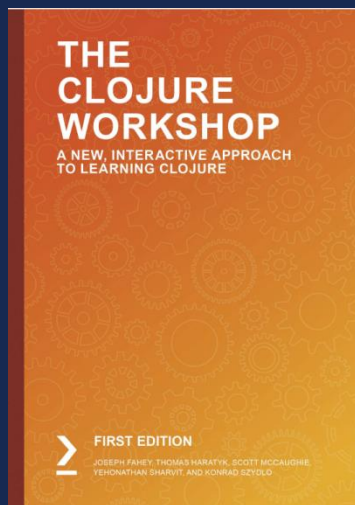
Prof. Aparecido V. de Freitas  
Doutor em Engenharia  
da Computação pela EPUSP  
[aparecido.freitas@prof.uscs.edu.br](mailto:aparecido.freitas@prof.uscs.edu.br)  
[aparecidovfreitas@gmail.com](mailto:aparecidovfreitas@gmail.com)



Revisão Técnica: Mauricio Szabo  
[mauricio.szabo@gmail.com](mailto:mauricio.szabo@gmail.com)



# Bibliografia



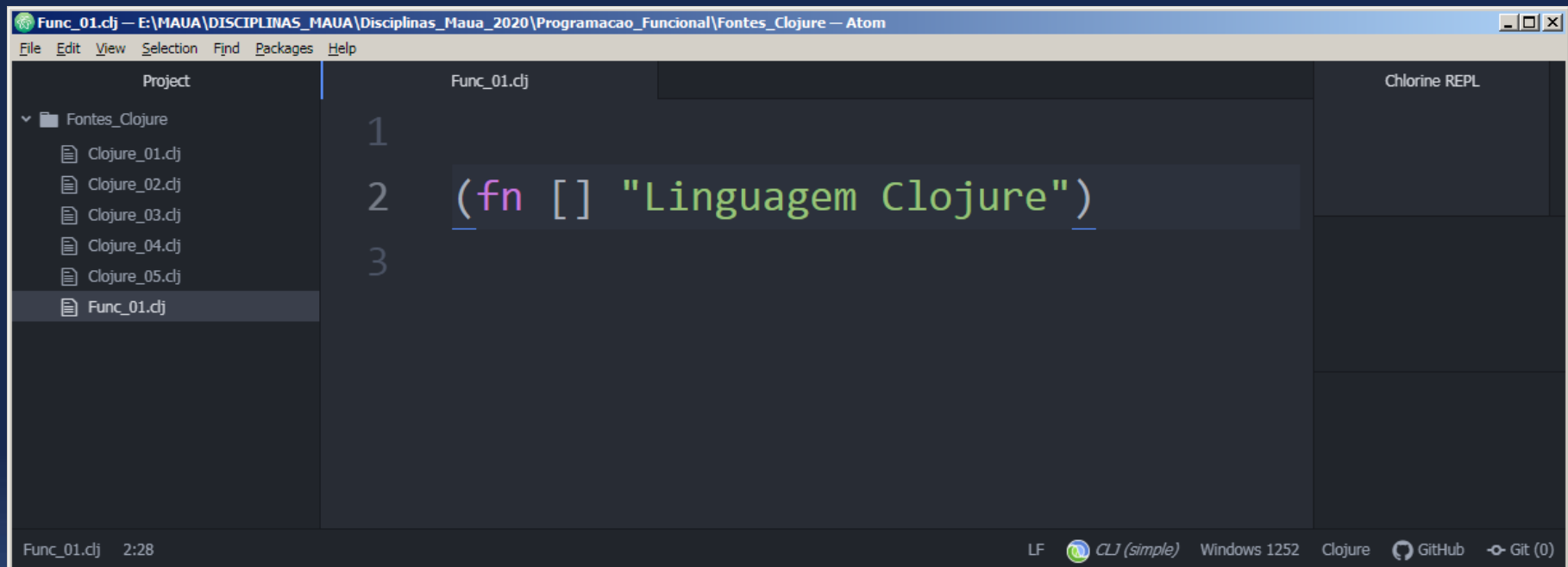
# Introdução

- ✓ A linguagem **Clojure** é funcional, e funções são de fundamental importância para o programador **Clojure** e para a programação funcional;
- ✓ Na programação funcional, evita-se mudanças de estado e emprega-se de forma intensa estruturas de dados imutáveis;
- ✓ Funções em **Clojure** são **"first-class citizens"** pois pode-se passar uma função para outra função, armazená-las em variáveis, ou ainda retorná-las de outras funções;
- ✓ Funções em **Clojure** são também chamadas de **"first-class functions"**.



# Lembrando ... Funções

- ✓ Uma função ao ser avaliada retorna sua última avaliação;
- ✓ Funções podem ser criadas por `fn`.



```
Func_01.cj
```

```
1  
2 (fn [] "Linguagem Clojure")  
3
```

Chlorine REPL

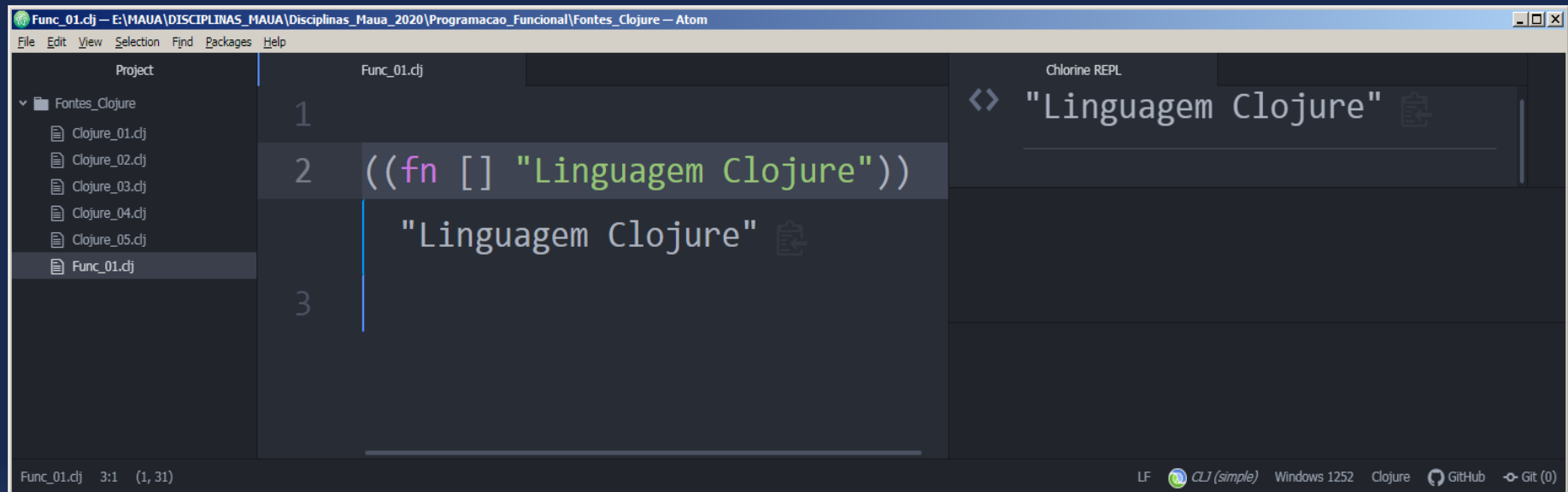
Func\_01.cj 2:28

LF CLJ (simple) Windows 1252 Clojure GitHub Git (0)



# Lembrando ... Funções

- ✓ Para processarmos uma função, devemos chamá-las entre parênteses.



The screenshot shows the Atom editor interface. On the left, a project tree shows a folder named 'Fontes\_Clojure' containing several files: 'Clojure\_01.cj', 'Clojure\_02.cj', 'Clojure\_03.cj', 'Clojure\_04.cj', 'Clojure\_05.cj', and 'Func\_01.cj'. The main editor window displays the content of 'Func\_01.cj' with the following code:

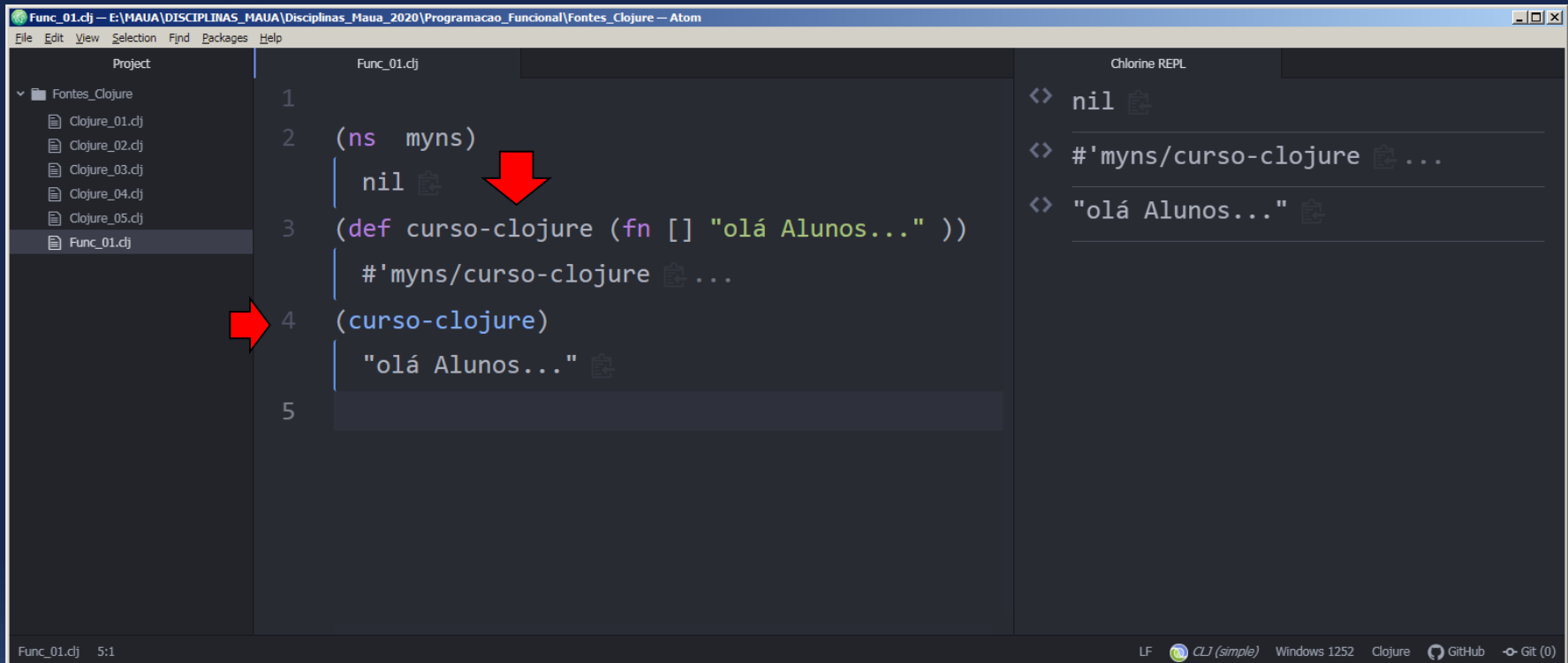
```
1  
2 ((fn [] "Linguagem Clojure"))  
3 "Linguagem Clojure"
```

On the right, the 'Chlorine REPL' panel shows the output of the function call: '<> "Linguagem Clojure"'. The status bar at the bottom indicates the current file is 'Func\_01.cj' at line 3, column 1, and provides links to 'LF', 'CLJ (simple)', 'Windows 1252', 'Clojure', 'GitHub', and 'Git (0)'.



# Lembrando ... Funções

✓ **Símbolos** podem ser associados à funções.



```
Func_01.cj — E:\MAUA\DISCIPLINAS_MAU\Disciplinas_Maua_2020\Programacao_Funcional\Fontes_Clojure — Atom
File Edit View Selection Find Packages Help

Project
  Fontes_Clojure
    Clojure_01.cj
    Clojure_02.cj
    Clojure_03.cj
    Clojure_04.cj
    Clojure_05.cj
    Func_01.cj

Func_01.cj
1
2 (ns myns)
   nil
3 (def curso-clojure (fn [] "olá Alunos..." ))
   #'myns/curso-clojure ...
4 (curso-clojure)
   "olá Alunos..."
5

Chlorine REPL
<> nil
<> #'myns/curso-clojure ...
<> "olá Alunos..."

Func_01.cj 5:1
LF CLJ (simple) Windows 1252 Clojure GitHub Git (0)
```



# Abreviando o processo com **defn**

✓ **defn** simplifica o processo de criar uma função e atribuindo-a um **símbolo**.

The screenshot shows the Atom editor interface with a project named 'Fontes\_Clojure'. The main editor window displays the file 'Func\_02.cj' with the following Clojure code:

```
1  
2 (ns myns)  
   nil  
3 (defn ola-alunos [] "Ola alunos...")  
   #'myns/ola-alunos ...  
4 (ola-alunos)  
   "Ola alunos..."  
5
```

Two red arrows highlight the process: one points to the `defn` keyword, and the other points to the function call `(ola-alunos)` on line 4.

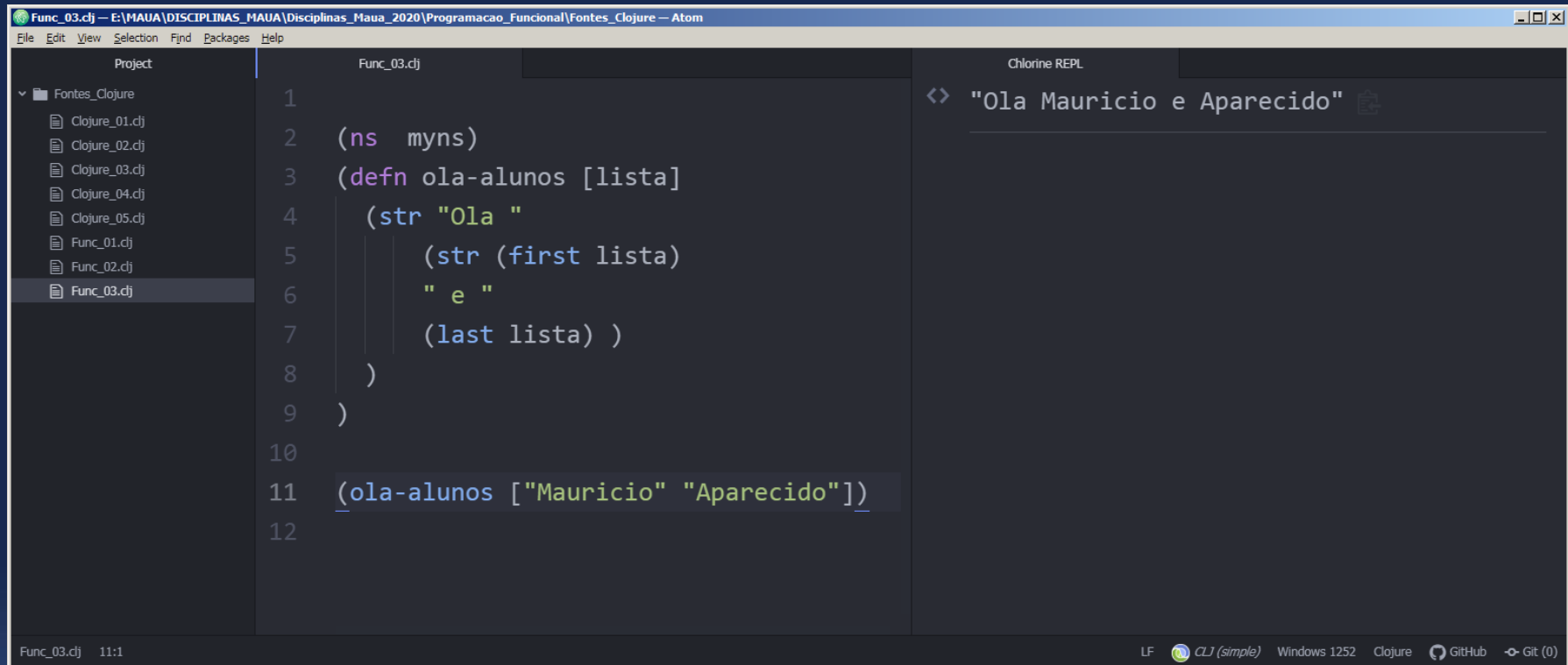
The right-hand pane shows the 'Chlorine REPL' with the following output:

```
<> nil  
<> #'myns/ola-alunos ...  
<> "Ola alunos..."
```

The status bar at the bottom indicates the current file is 'Func\_02.cj' at line 5, column 1. The bottom right corner shows the 'CLJ (simple)' REPL running on 'Windows 1252' encoding, with 'Clojure' as the language and 'GitHub' as the provider.



# Argumentos para Funções



The screenshot shows the Atom editor interface with a project named 'Fontes\_Clojure'. The file 'Func\_03.cj' is open, displaying the following Clojure code:

```
1  
2 (ns myns)  
3 (defn ola-alunos [lista]  
4   (str "Ola "  
5       (str (first lista)  
6           " e "  
7       (last lista) )  
8   )  
9 )  
10  
11 (ola-alunos ["Mauricio" "Aparecido"])  
12
```

The right-hand pane shows the 'Chlorine REPL' with the output of the function call:

```
<> "Ola Mauricio e Aparecido"
```

The status bar at the bottom indicates the file is 'Func\_03.cj' at line 11, column 1. It also shows icons for 'LF', 'CLJ (simple)', 'Windows 1252', 'Clojure', 'GitHub', and 'Git (0)'.



# Funções com multivariadas

```
Func_04.cj — E:\MAUA\DISCIPLINAS_MAUUA\Disciplinas_Maua_2020\Programacao_Funcional\Fontes_Clojure — Atom
File Edit View Selection Find Packages Help

Project
  Fontes_Clojure
    Clojure_01.cj
    Clojure_02.cj
    Clojure_03.cj
    Clojure_04.cj
    Clojure_05.cj
    Func_01.cj
    Func_02.cj
    Func_03.cj
    Func_04.cj

Func_04.cj
1
2 (ns myns)
3 (defn adiciona
4   ([ ] 0)
5   ([x] x)
6   ([x y] (+ x y)))
7 )
8
9 (adiciona)
10 (adiciona 4)
11 (adiciona 5 8)
12

Chlorine REPL
<> nil
<> #'myns/adiciona ...
<> 0
<> 4
<> 13

Func_04.cj 12:1
LF CLJ (simple) Windows 1252 Clojure GitHub Git (0)
```

# Desestruturação

- ✓ Desestruturação corresponde ao procedimento de se **remover** elementos de sua estrutura ou **desmontar** uma estrutura de dados;
- ✓ Há duas formas principais de se desestruturar dados: **sequencialmente** (com **vectors**) ou de forma **associativa** (com **maps**);

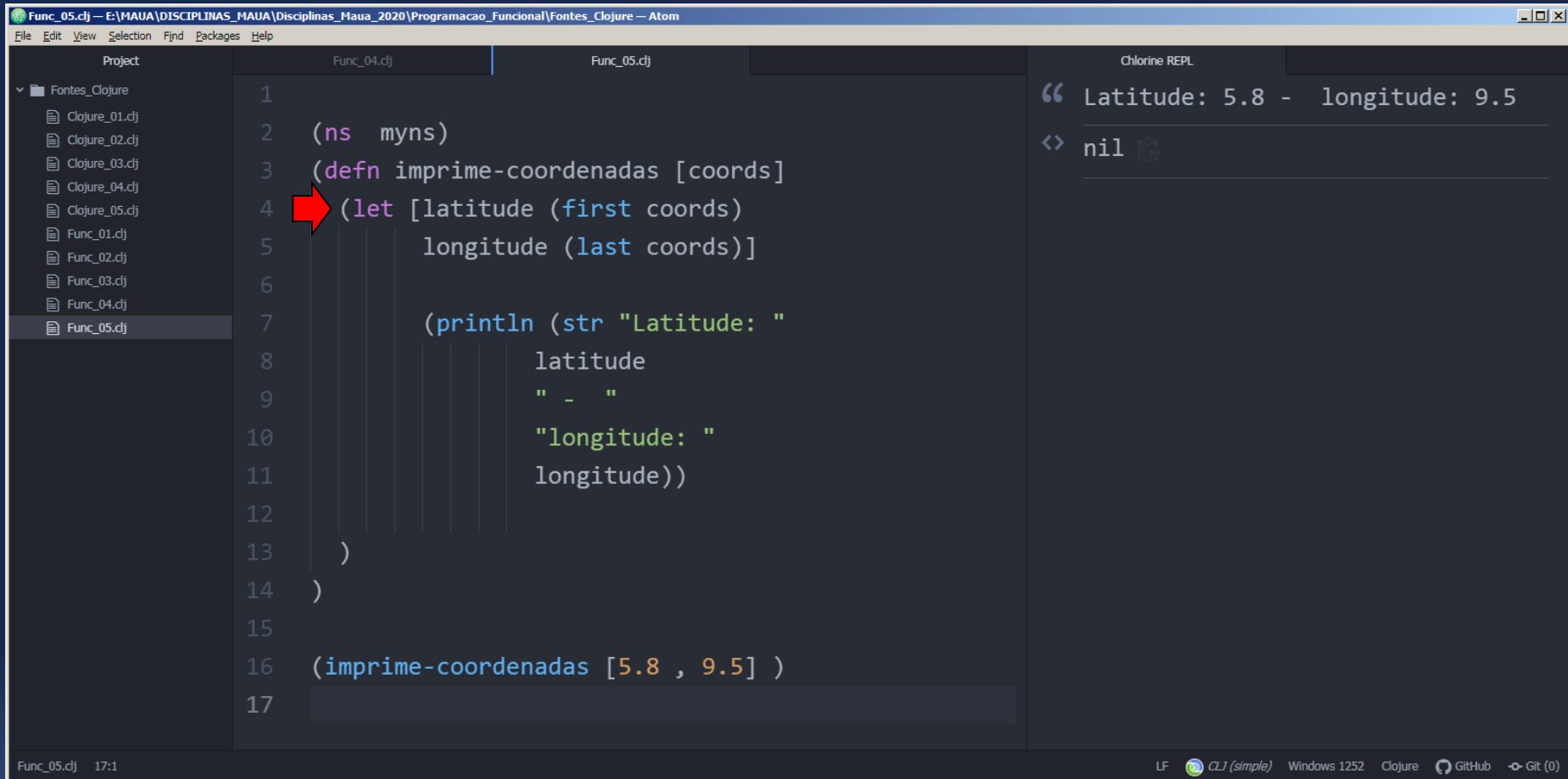


## Desestruturação Sequencial

- ✓ Considere, por exemplo, a necessidade de se escrever uma função que imprime - de forma formatada - um string que corresponde a uma tupla de coordenadas. Por exemplo: [5.8 , 9.5].
- ✓ Poderíamos escrever a seguinte função `imprime_coordenadas`.



# Exemplo – Desestruturação



```
Func_05.clj — E:\MAUA\DISCIPLINAS_MAU\Disciplinas_Maua_2020\Programacao_Funcional\Fontes_Clojure — Atom
File Edit View Selection Find Packages Help

Project
  Fontes_Clojure
    Clojure_01.clj
    Clojure_02.clj
    Clojure_03.clj
    Clojure_04.clj
    Clojure_05.clj
    Func_01.clj
    Func_02.clj
    Func_03.clj
    Func_04.clj
    Func_05.clj

1
2 (ns myns)
3 (defn imprime-coordenadas [coords]
4   (let [latitude (first coords)
5         longitude (last coords)]
6
7         (println (str "Latitude: "
8                       latitude
9                       " - "
10                      "longitude: "
11                      longitude)))
12   )
13 )
14
15
16 (imprime-coordenadas [5.8 , 9.5] )
17

Chlorine REPL
“ Latitude: 5.8 - longitude: 9.5
<> nil
```



# Exemplo – Desestruturação

```
(ns myns)
(defn imprime-coordenadas [coords]
  (let [latitude (first coords)
        longitude (last coords)]

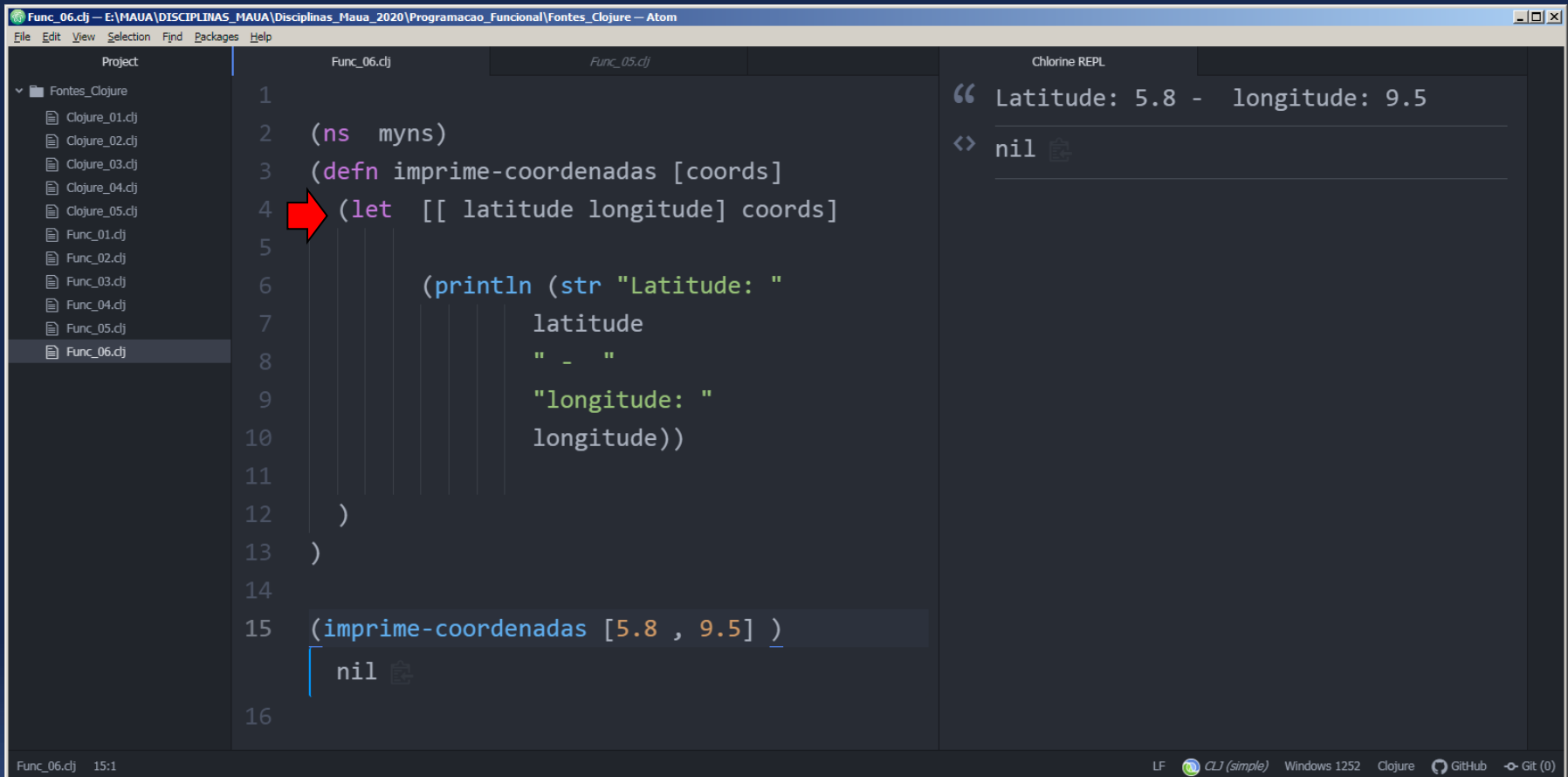
    (println (str "Latitude: "
                  latitude
                  " - "
                  "longitude: "
                  longitude)))
  )
)

(imprime-coordenadas [5.8 , 9.5] )
```

- ✓ Neste exemplo, a função `imprime-coordenadas` recebe um tupla de coordenadas como parâmetro e as imprime para a console de maneira formatada;
- ✓ O que se está fazendo nessa função é um **binding** do primeiro elemento para o símbolo `latitude` e o segundo para `longitude`. Assim, estamos **desestruturando** a estrutura passada como parâmetro.



# Outra sintaxe para Desestruturação



```
Func_06.clj — E:\MAUA\DISCIPLINAS_MAUU\Disciplinas_Mauu_2020\Programacao_Funcional\Fontes_Clojure — Atom
File Edit View Selection Find Packages Help

Project
  Fontes_Clojure
    Clojure_01.clj
    Clojure_02.clj
    Clojure_03.clj
    Clojure_04.clj
    Clojure_05.clj
    Func_01.clj
    Func_02.clj
    Func_03.clj
    Func_04.clj
    Func_05.clj
    Func_06.clj

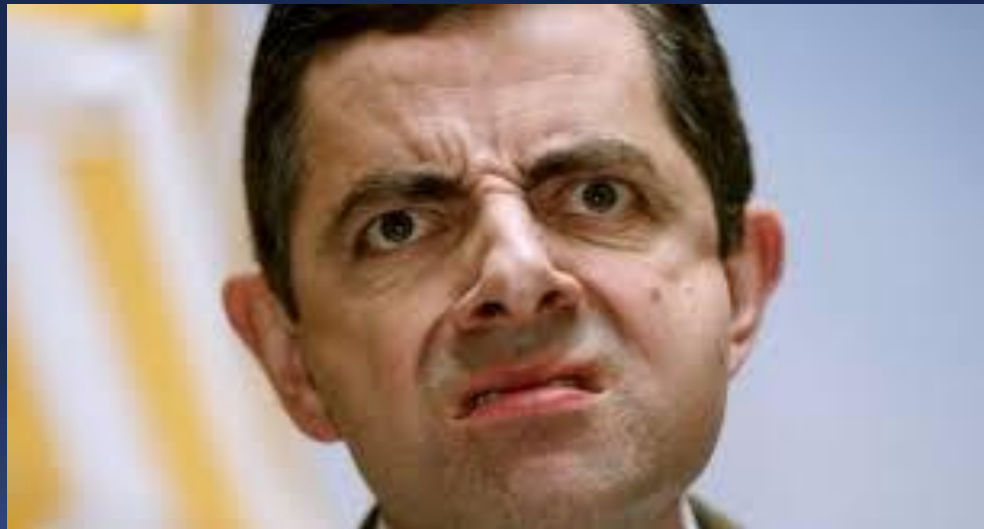
1
2 (ns myns)
3 (defn imprime-coordenadas [coords]
4   (let [[latitude longitude] coords]
5     (println (str "Latitude: "
6                 latitude
7                 " - "
8                 "longitude: "
9                 longitude)))
10
11
12 )
13 )
14
15 (imprime-coordenadas [5.8 , 9.5] )
16 nil

Chlorine REPL
“ Latitude: 5.8 - longitude: 9.5
<> nil
```



Vê ... Não entendi !!!

Eu não preciso escrever first e last ???



## Outra sintaxe para Desestruturação

```
(ns myns)
(defn imprime-coordenadas [coords]
  → (let [[latitude longitude] coords]
      (println (str "Latitude: "
                    latitude
                    " - "
                    "longitude: "
                    longitude)))
  )
)
```

- ✓ Com esta sintaxe o código está mais expressivo que o anterior;
- ✓ Não precisamos usar as funções `first` e `last`;
- ✓ Simplesmente expressamos os símbolos que desejamos recuperar;
- ✓ Com essa sintaxe, `latitude` é "mapeada" para o primeiro elemento do vector;
- ✓ E `longitude` é "mapeada" para o segundo elemento do vector.





# Outro exemplo para Desestruturação

The screenshot shows the Chlorine REPL interface with the following components:

- Project Panel:** Lists files in the `Fontes_Clojure` directory, including `Clojure_01.cj` through `Func_07.cj`.
- Editor:** Displays the content of `Func_07.cj` with line numbers 1 through 7. The code is:

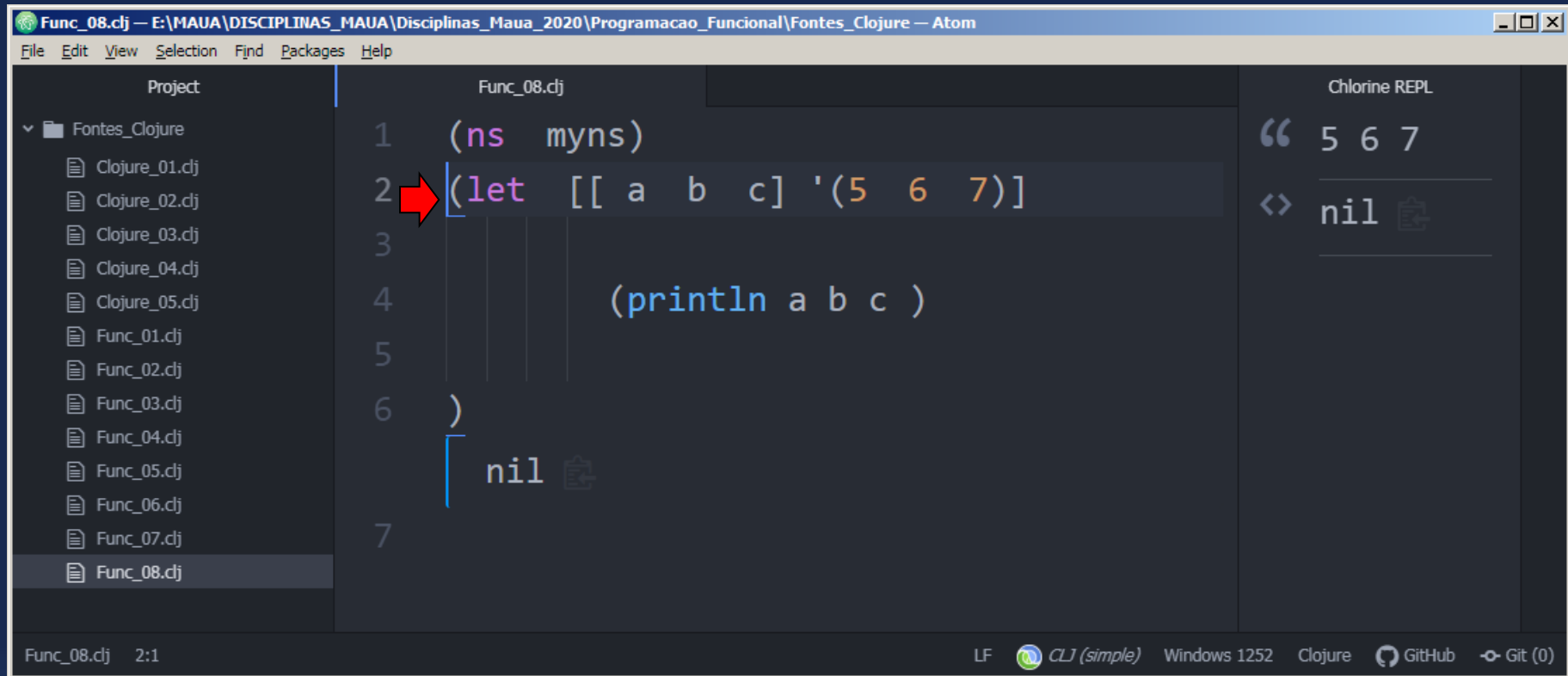
```
1 (ns myns)
2 (let [[ a b c] [ 1 2 3]]
3
4     (println a b c )
5
6 )
7
```

A red arrow points to the opening parenthesis of the `(let` expression on line 2.
- REPL Panel:** Shows the execution results. The first prompt `"` is followed by `1 2 3`. The second prompt `<>` is followed by `nil` and a clipboard icon.
- Status Bar:** At the bottom, it shows `Chlorine REPL`, `CLJ (simple)`, `GitHub`, and `Git (0)`.

- ✓ Neste exemplo, os bindings são criados de acordo com a ordem sequencial do vector e a ordem sequencial dos símbolos definidos no vector `[ a b c ]`;
- ✓ Os símbolos resultantes do binding estão sendo usados no `println`.



# Uma lista também pode ser desmontada



The screenshot shows the Atom editor interface with a file named `Func_08.cj` open. The left sidebar displays a project tree with a folder `Fontes_Clojure` containing several files. The main editor area shows the following Clojure code:

```
1 (ns myns)
2 (let [[ a b c] '(5 6 7)]
3
4     (println a b c )
5
6 )
7 nil
```

A red arrow points to the opening square bracket of the `(let` expression on line 2. The right sidebar shows the `Chlorine REPL` with the output of the code execution:

```
“ 5 6 7
<> nil
```

The status bar at the bottom indicates the current file is `Func_08.cj` at line 2, column 1. It also shows the Clojure version `CLJ (simple)` and the operating system `Windows 1252`.





# Desestruturação Associativa



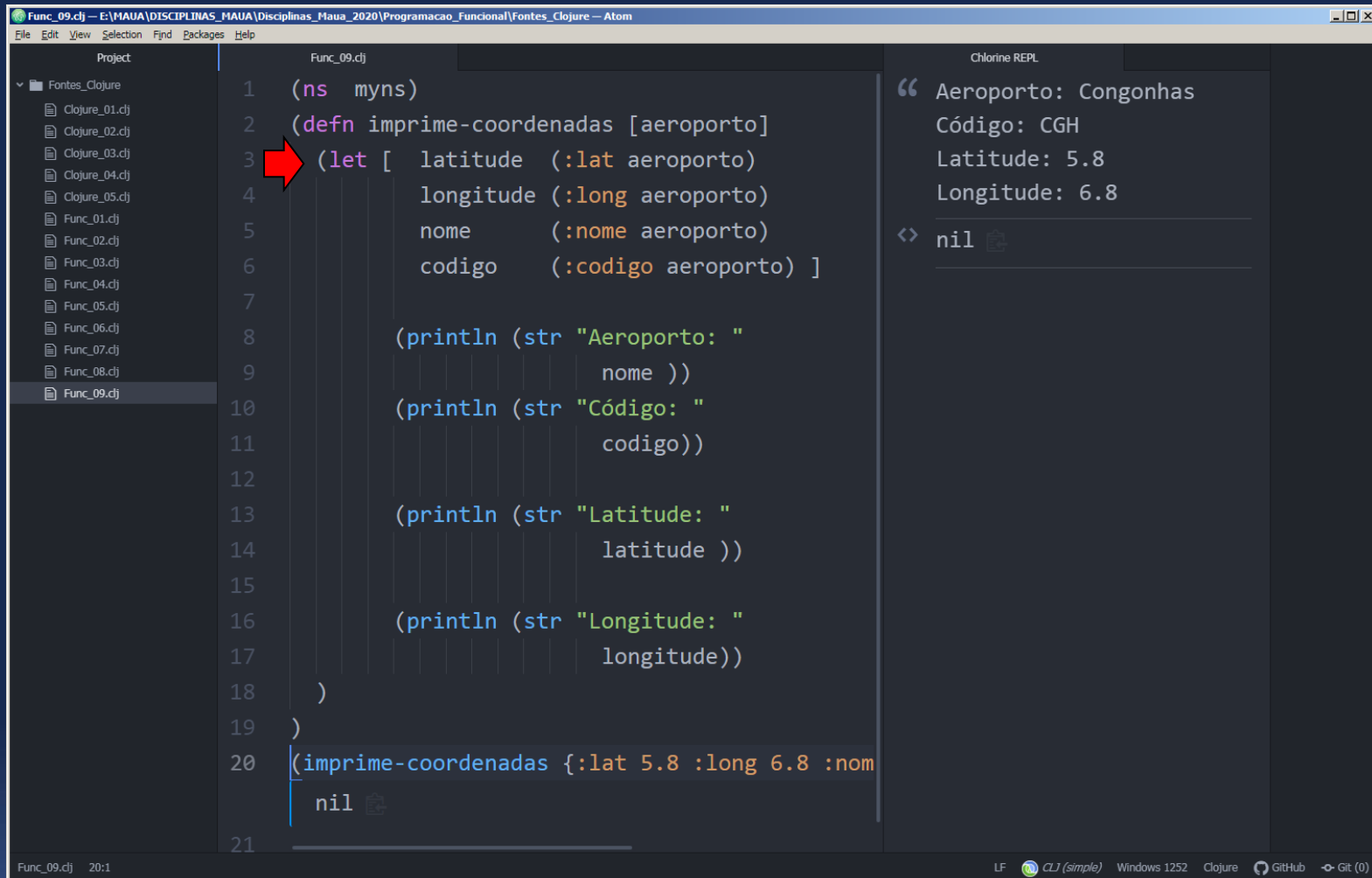
# Desestruturação Associativa

- ✓ Vamos considerar o exemplo visto anteriormente com a função `imprime-coordenadas`;
- ✓ Porém, ao invés dessa função receber uma tupla de valores correspondentes às coordenadas, vamos imaginar que o parâmetro fosse um `map`, com os seguintes pares `key-value`:

➡ { `:lat` 5.8      `:long` 6.8      `:nome` "Congonhas"      `:codigo` "CGH" }



# Função imprime-coordenadas



The screenshot shows the Atom editor with a project named 'Fontes\_Clojure'. The file 'Func\_09.cj' is open, displaying a Clojure function 'imprime-coordenadas' that takes an 'aeroporto' argument and prints its details. A red arrow points to the 'let' binding in the function definition. The REPL on the right shows the output of the function call, displaying the airport name, code, latitude, and longitude.

```
1 (ns myns)
2 (defn imprime-coordenadas [aeroporto]
3   (let [ latitude (:lat aeroporto)
4         longitude (:long aeroporto)
5         nome (:nome aeroporto)
6         codigo (:codigo aeroporto) ]
7
8     (println (str "Aeroporto: "
9                  nome ))
10    (println (str "Código: "
11                 codigo))
12
13    (println (str "Latitude: "
14                 latitude ))
15
16    (println (str "Longitude: "
17                 longitude))
18  )
19 )
20 (imprime-coordenadas {:lat 5.8 :long 6.8 :nom
21 nil
```

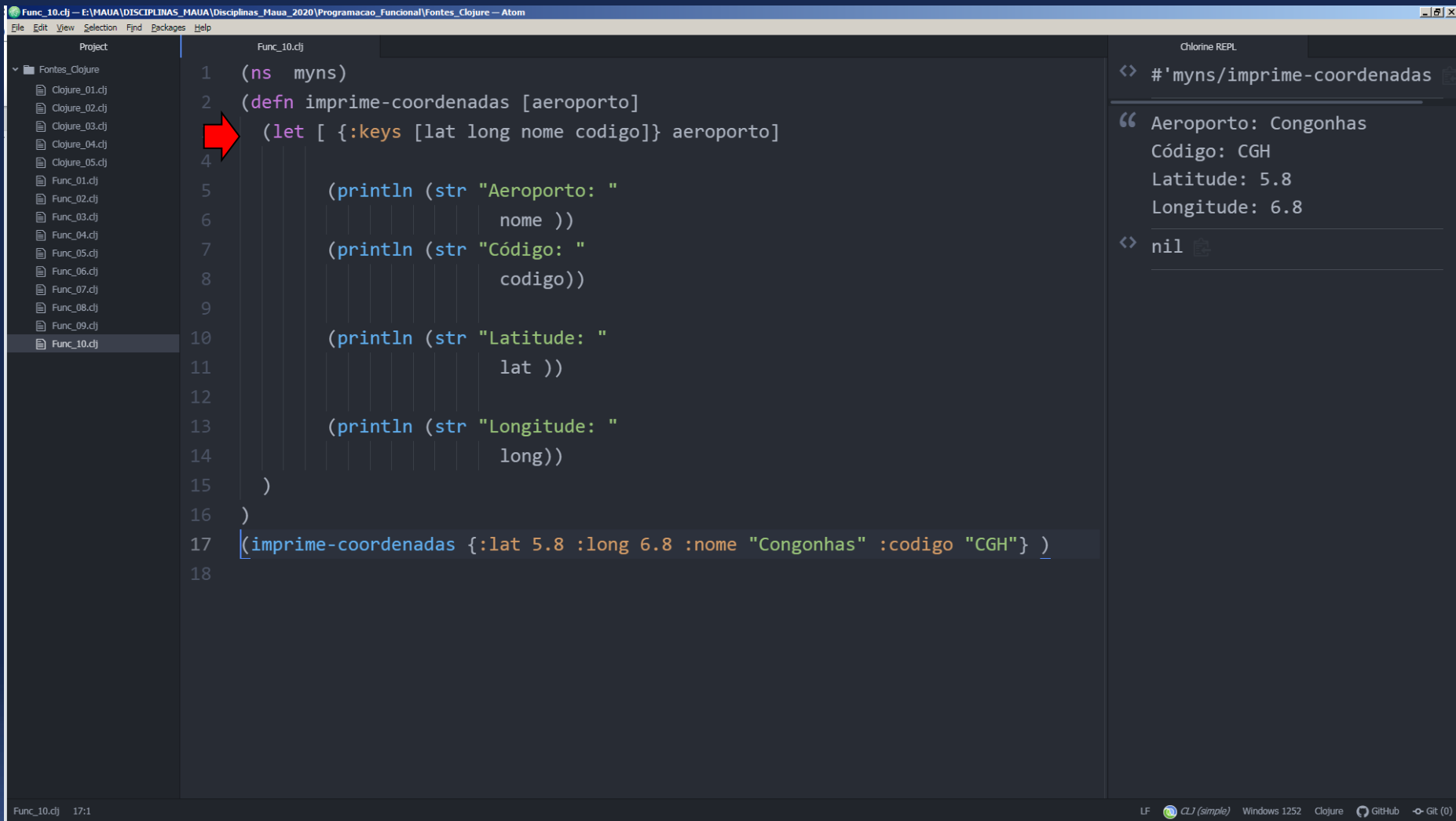
Chlorine REPL

```
" Aeroporto: Congonhas
Código: CGH
Latitude: 5.8
Longitude: 6.8
<> nil
```

- ✓ Nesse exemplo, ainda **não** utilizamos desestruturação associativa;
- ✓ Aqui apenas recuperamos os valores do map com o uso de keywords como funções na expressão let.



# Função imprime-coordenadas



```
1 (ns myns)
2 (defn imprime-coordenadas [aeroporto]
3   (let [ {:keys [lat long nome codigo]} aeroporto]
4
5     (println (str "Aeroporto: "
6                  nome ))
7
8     (println (str "Código: "
9                  codigo))
10
11    (println (str "Latitude: "
12                lat ))
13
14    (println (str "Longitude: "
15                 long))
16  )
17  (imprime-coordenadas {:lat 5.8 :long 6.8 :nome "Congonhas" :codigo "CGH"} )
18
```

Chlorine REPL

```
<> #'myns/imprime-coordenadas
“ Aeroporto: Congonhas
  Código: CGH
  Latitude: 5.8
  Longitude: 6.8
<> nil
```

- ✓ Aqui sim, estamos usando a técnica de desestruturação associativa;
- ✓ Bindings das chaves do map são associados à símbolos que são posteriormente utilizados em printlns para a console.





# Desestruturando parâmetros de Funções



# Desestruturando parâmetros de funções

```
1 (ns myns)
2 (defn imprime-voo [[[lat1 long1] [lat2 long2]]]
3
4     (println "Voando de:")
5     (println (str "Latitude 1: "
6                   lat1 ))
7     (println (str "Longitude 1: "
8                   long1))
9
10    (println "para:")
11
12    (println (str "Latitude 2: "
13                 lat2 ))
14
15    (println (str "Longitude 2: "
16                 long2))
17 )
18 )
19 (imprime-voo [ [5.7 8.5] [3.8 3.2]] )
```

<> #'myns/imprime-voo ...

“ Voando de:  
Latitude 1: 5.7  
Longitude 1: 8.5  
para:  
Latitude 2: 3.8  
Longitude 2: 3.2

<> nil

- ✓ Neste exemplo se está empregando desestruturação sequencial nos parâmetros da função!





# Arity Overloading

- ✓ Clojure suporta "**arity overloading**", o que significa que pode-se efetuar **sobrecarga** de uma função com outra função de mesmo nome especificando-se parâmetros extra para a nova função;
- ✓ Assim, essas duas funções têm o mesmo nome, mas têm diferentes implementações;
- ✓ O corpo da função é escolhido com base no número de argumentos fornecidos.

```
1 (ns myns)
2 (defn overload
3   ([ ] "Sem argumentos...")
4   ([a] (str "Um argumento: " a ))
5   ([a b] (str "Dois argumentos: " a " " b )))
6 )
7
8 (overload)
9 (overload "Hello")
10 (overload 10 99)
```

```
<> "Sem argumentos..."
<> "Um argumento: Hello"
<> "Dois argumentos: 10 99"
```





# Higher-Order Programming



# Higher Order Programming

- ✓ Permite que uma **função** pode ser **passada** como parâmetro para outra **função**;
- ✓ Da mesma forma uma **função** pode ser **retornada** por outra função;
- ✓ A escrita de funções simples aumenta sua **modularidade**;
- ✓ Da mesma forma, a escrita de funções **puras** aumenta a **robustez** e **confiabilidade** do código;
- ✓ Funções puras **não** causam **side effects**, uma vez que sempre - quando aplicadas - **retornam o mesmo valor** quando a ela são passados os mesmos parâmetros.
- ✓ Como **boa prática**, deve-se escrever funções puras tanto quanto possível.



# Higher Order Programming

- ✓ Conceito de extrema importância em qualquer linguagem do Paradigma Funcional;
- ✓ Higher order functions permitem a composição de funções;
- ✓ Isto significa que podemos escrever funções menores e combiná-las para criar funções maiores (modularidade);
- ✓ Como um jogo de LEGO no qual pequenas peças são compostas para formar uma peça maior;



# Funções como Argumentos

- ✓ Vejamos as duas funções abaixo:

```
(ns myns)

(defn dobro-soma [a b]
  (* 2 (+ a b) ))

(defn dobro-produto [a b]
  (* 2 (* a b) ))

(double-soma 2 3 )
10

(dobro-produto 2 3)
12
```

- ✓ As funções dobro-soma e dobro-produto compartilham um padrão comum. Elas somente se diferem pelo nome e pela função usada na computação.



# Funções como Argumentos

```
Func_15.clj      Chlorine REPL

1  (ns myns)
2
3  (defn f [op a b]
4    (op a b))
5
6  (defn dobro-f [f op a b]
7    (* 2 (f op a b)))
8
9
10 (f + 2 3)
    5
11 (f * 2 3)
    6
12
13 (dobro-f f * 4 6)
    48
14 (dobro-f f + 10 5)
    30
```

- ✓ A função `f` foi passada como parâmetro para a função `dobro-f`.



# Funções retornando Funções

- ✓ A primeira função será chamada **somador**. Ela recebe um número **x**, como único argumento, e **retorna** uma função;
- ✓ A função retornada pelo **somador** também recebe um simples número **a**, como seu único argumento, e retorna **x + a**;
- ✓ A função **somador** é um "**closure**". Isto significa que ela pode acessar todas as variáveis que estavam no escopo quando ela foi criada.
- ✓ A função **soma-5** tem acesso à **x** mesmo estando fora da definição de **somador** !



```
Func_17.cj                                     Chlorine REPL
(ns myns)                                       <> 105

(defn somador [x]
  (fn [a] (+ x a))
)

(def soma-5 (somador 5))

(soma-5 100)
105
```

