

Programação Funcional

Unidade 6 – Funções

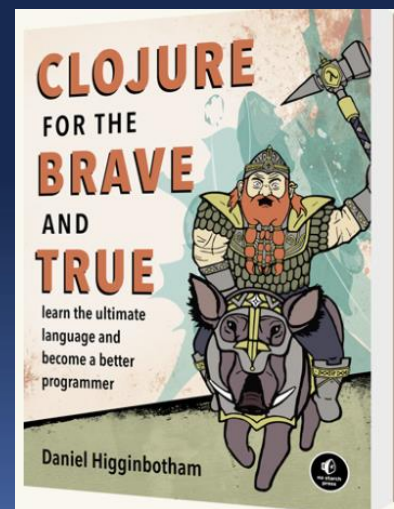
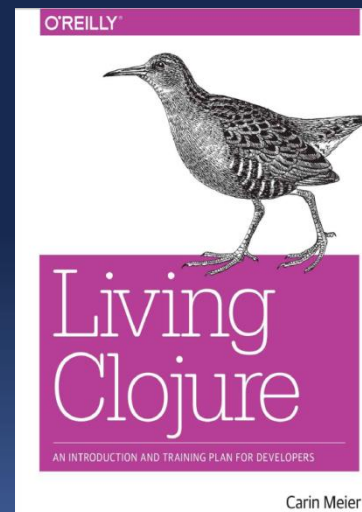
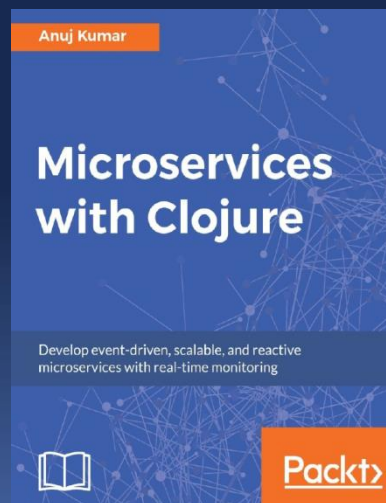
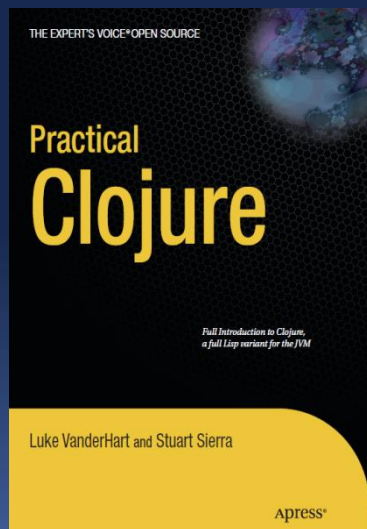
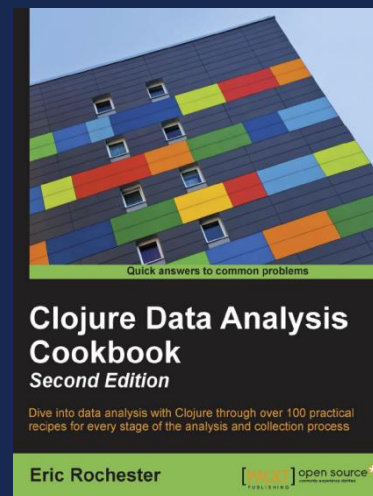
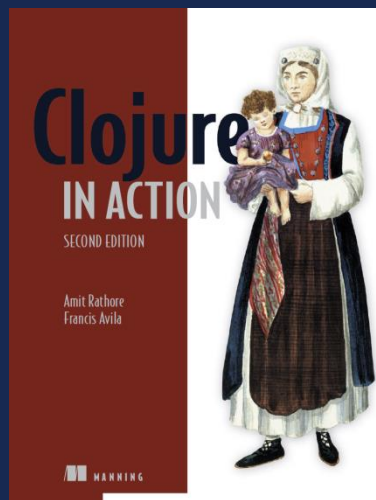
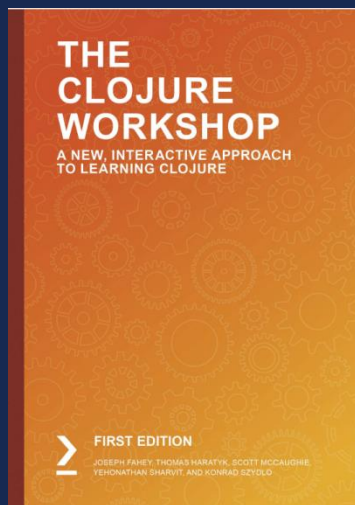


Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecido.freitas@prof.uscs.edu.br
aparecidovfreitas@gmail.com

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



Bibliografia

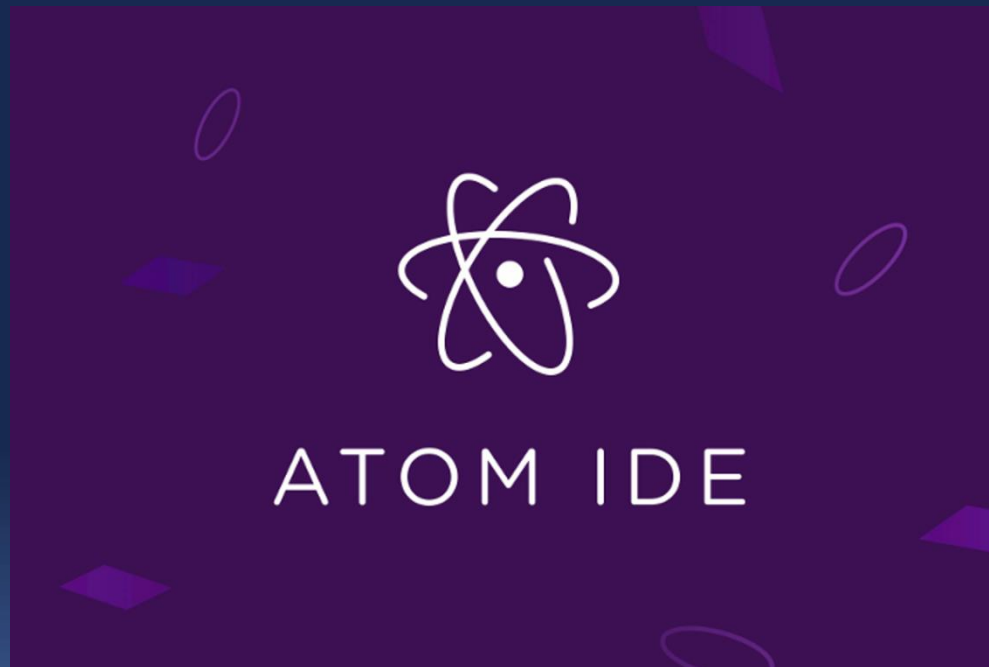


Introdução

- ✓ 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"**.

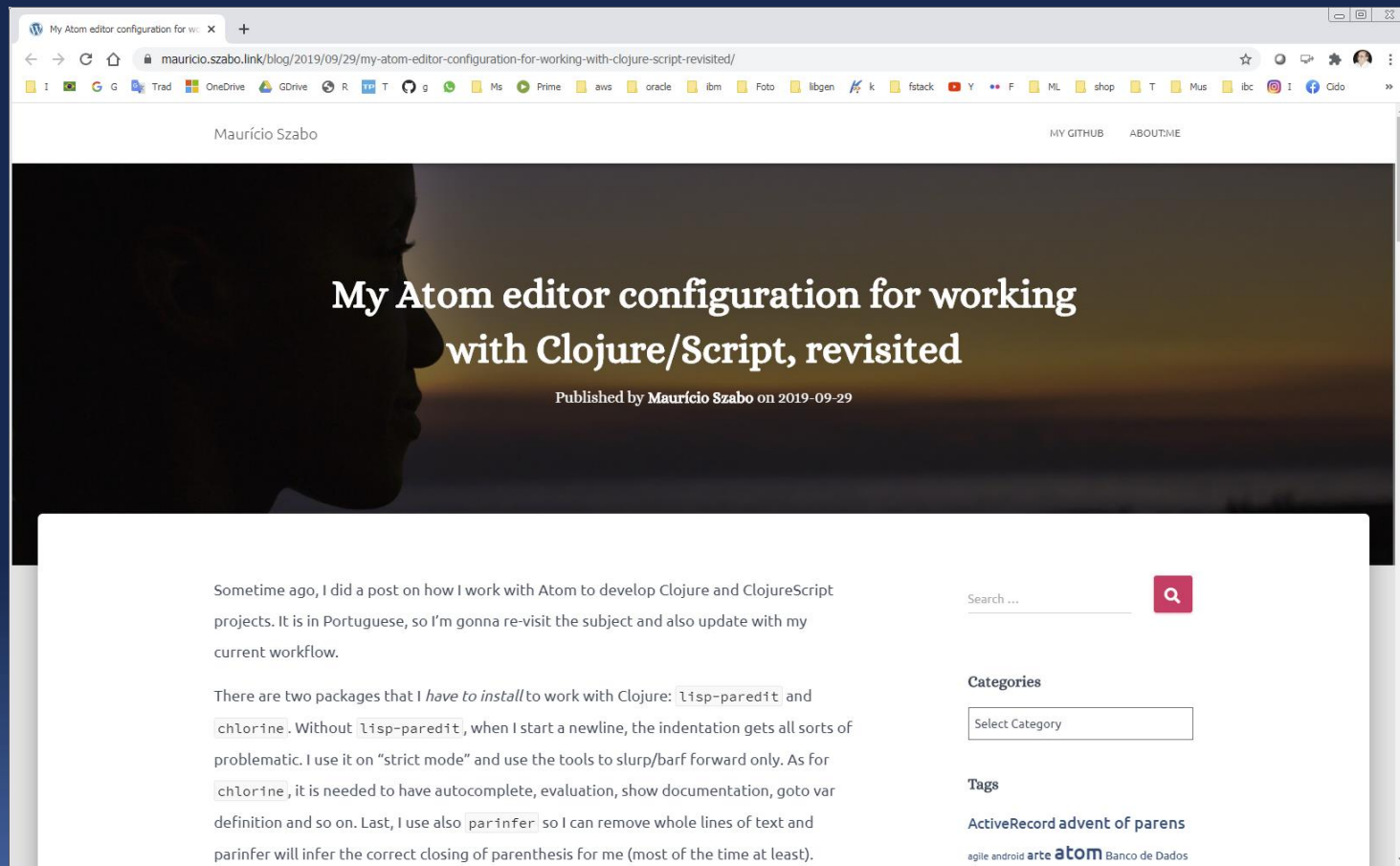


Clojure com Atom



Configuração do Package Chlorine no Atom

<https://mauricio.szabo.link/blog/2019/09/29/my-atom-editor-configuration-for-working-with-clojure-script-revisited/>



Ativando REPL Remoto

```
(do (require 'clojure.core.server)
    (clojure.core.server/start-server
     {:name "socket-repl"
      :port 4444
      :accept 'clojure.main/repl
      :address "localhost"}))
```



Ativando REPL Remoto

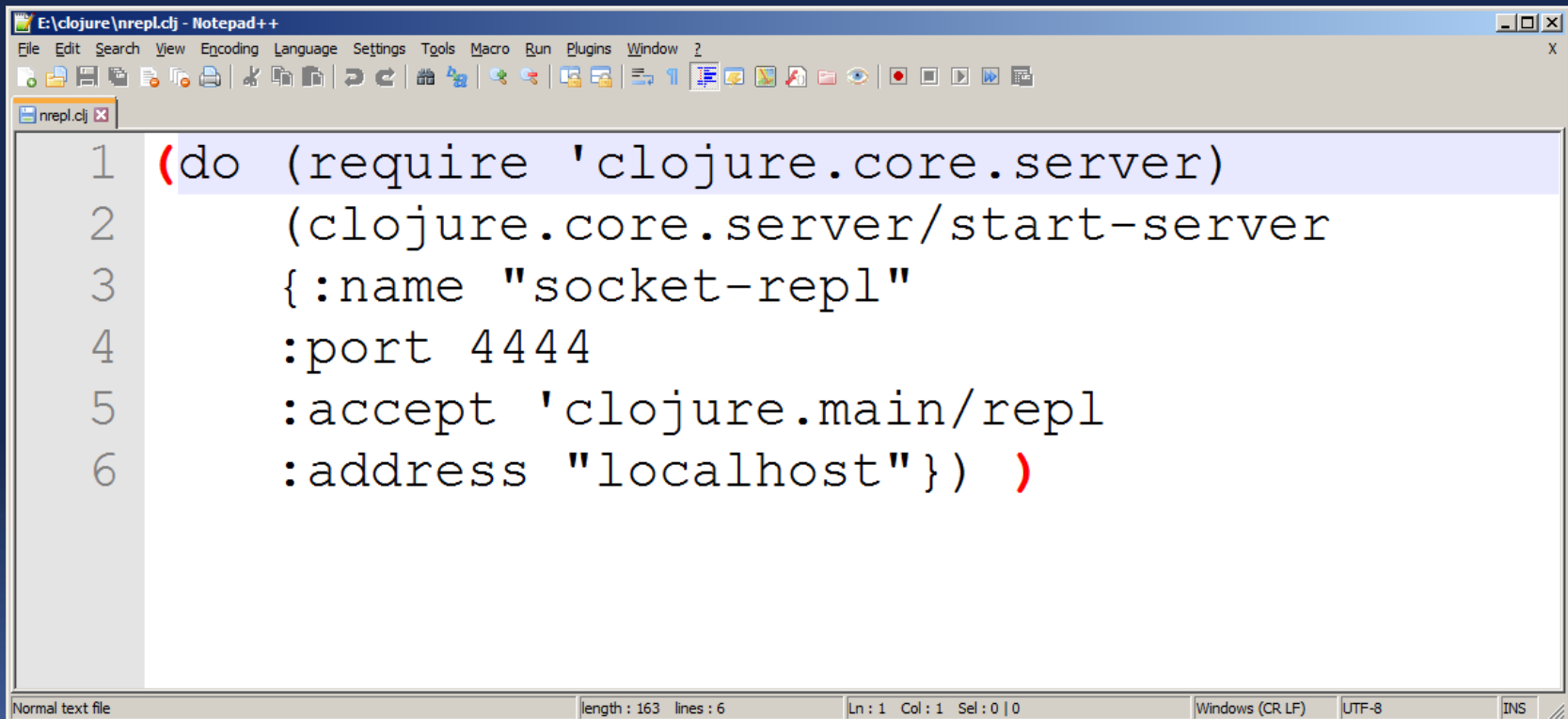
Digitando-se o código na console

```
Clojure 1.10.2-master-SNAPSHOT
user=> (do (require 'clojure.core.server)
(clojure.core.server/start-server
{:name "socket-repl"
:port 4444
:accept 'clojure.main/repl
:address "localhost"}))
#object[java.net.ServerSocket 0x43f82e78 "ServerSocket[addr=localhost/127.0.0.1,
localport=4444]"]
user=> _
```



Salvando código num file .clj

- ✓ Criar arquivo .clj com código
- ✓ Estamos chamando o arquivo de nrepl.clj
- ✓ Será salvo no diretório: E:/clojure



The screenshot shows a Notepad++ window titled "E:\clojure\nrepl.clj - Notepad++". The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, Plugins, Window, and Help. The toolbar contains various icons for file operations and editing. The file "nrepl.clj" is open, and its content is displayed in a monospaced font. The code is as follows:

```
1 (do (require 'clojure.core.server)
2     (clojure.core.server/start-server
3       {:name "socket-repl"
4         :port 4444
5         :accept 'clojure.main/repl
6         :address "localhost"})) )
```

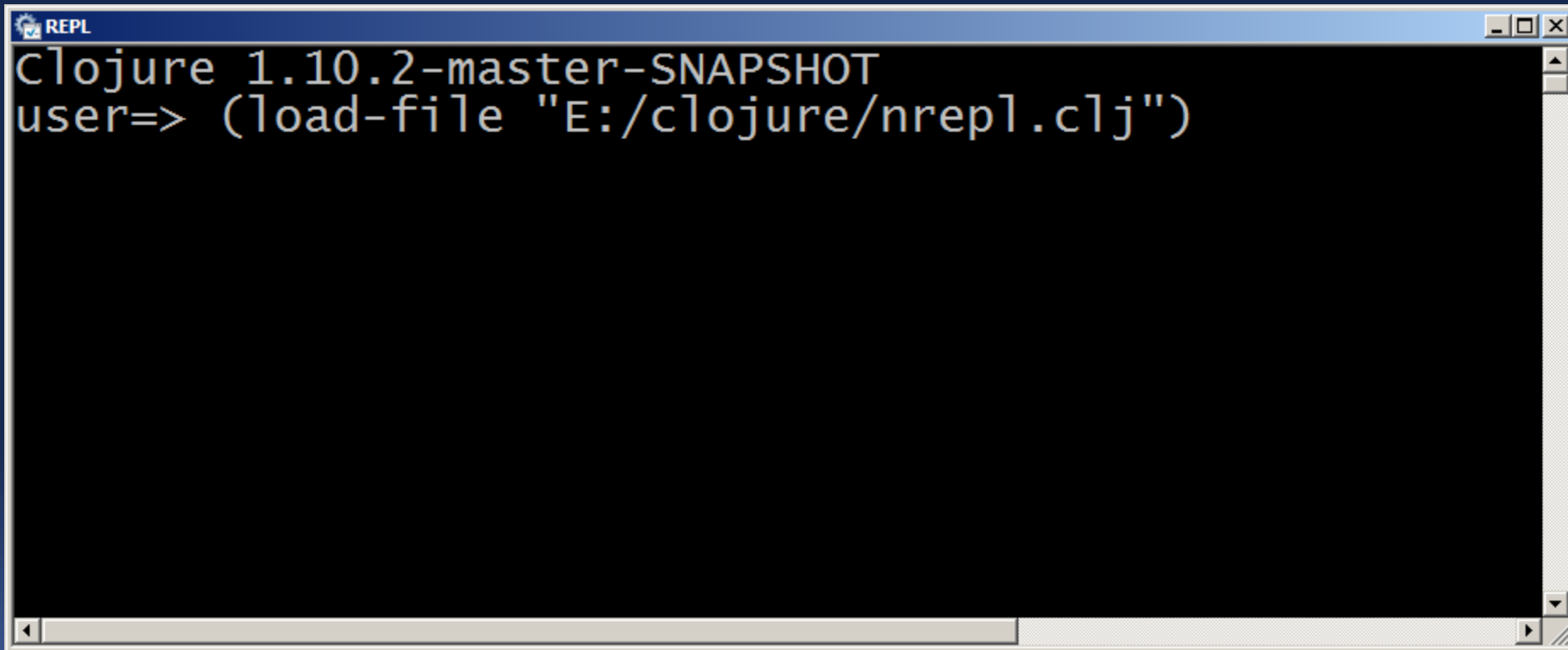
The status bar at the bottom indicates "Normal text file", "length : 163 lines : 6", "Ln : 1 Col : 1 Sel : 0 | 0", "Windows (CR LF)", "UTF-8", and "INS".



Ativando REPL Remoto

Carregando-se arquivo digitado anteriormente

> (load-file "E:/clojure/nrepl.clj")

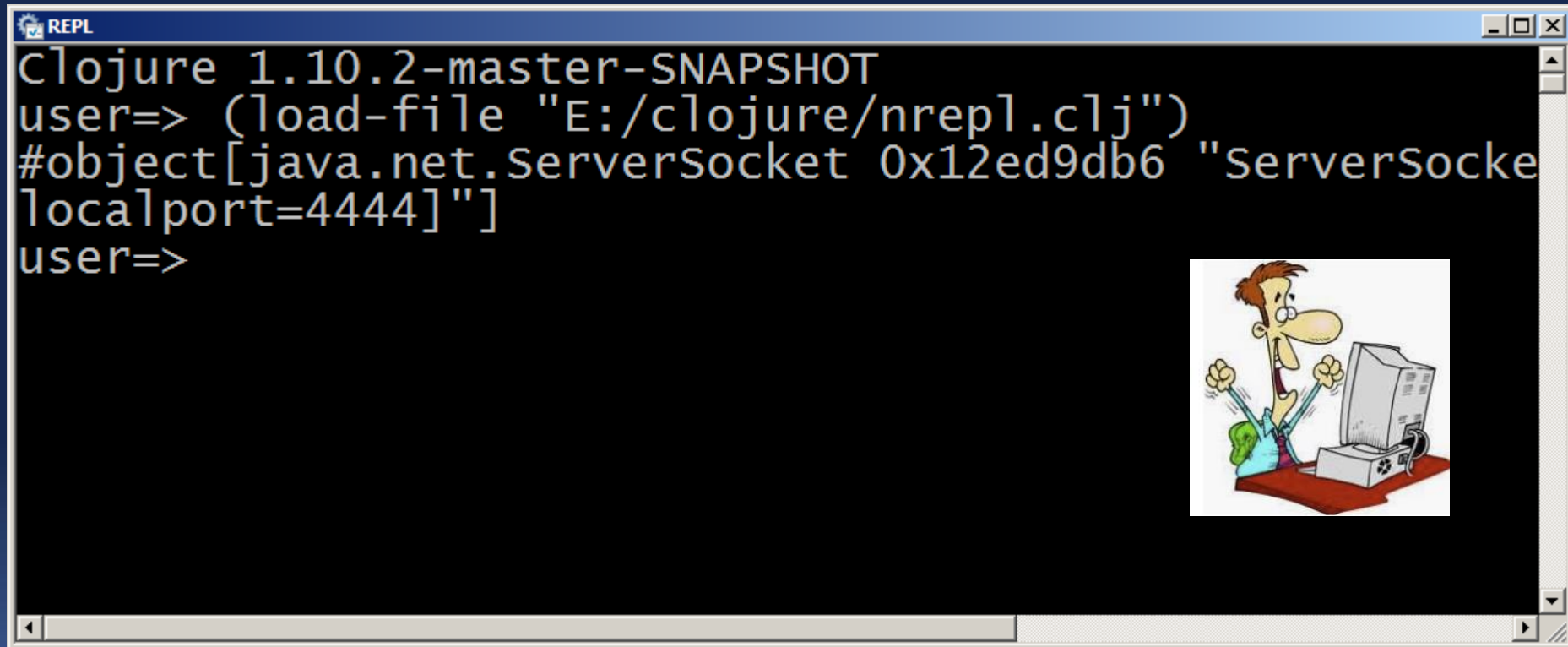


```
REPL
Clojure 1.10.2-master-SNAPSHOT
user=> (load-file "E:/clojure/nrepl.clj")
```

Ativando REPL Remoto

Carregando-se arquivo digitado anteriormente

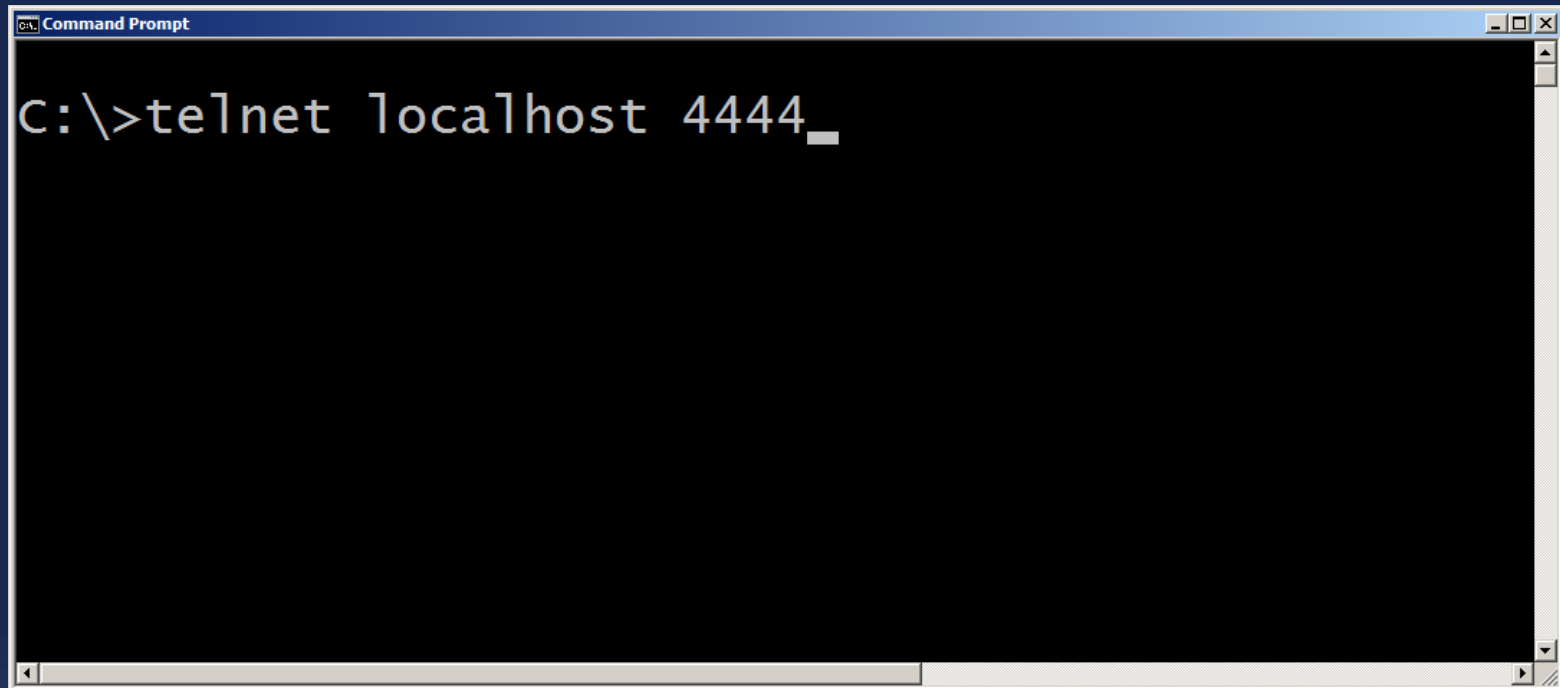
> (load-file "E:/clojure/nrepl.clj")



```
REPL
Clojure 1.10.2-master-SNAPSHOT
user=> (load-file "E:/clojure/nrepl.clj")
#object[java.net.ServerSocket 0x12ed9db6 "ServerSocket
localhost=4444"]
user=>
```

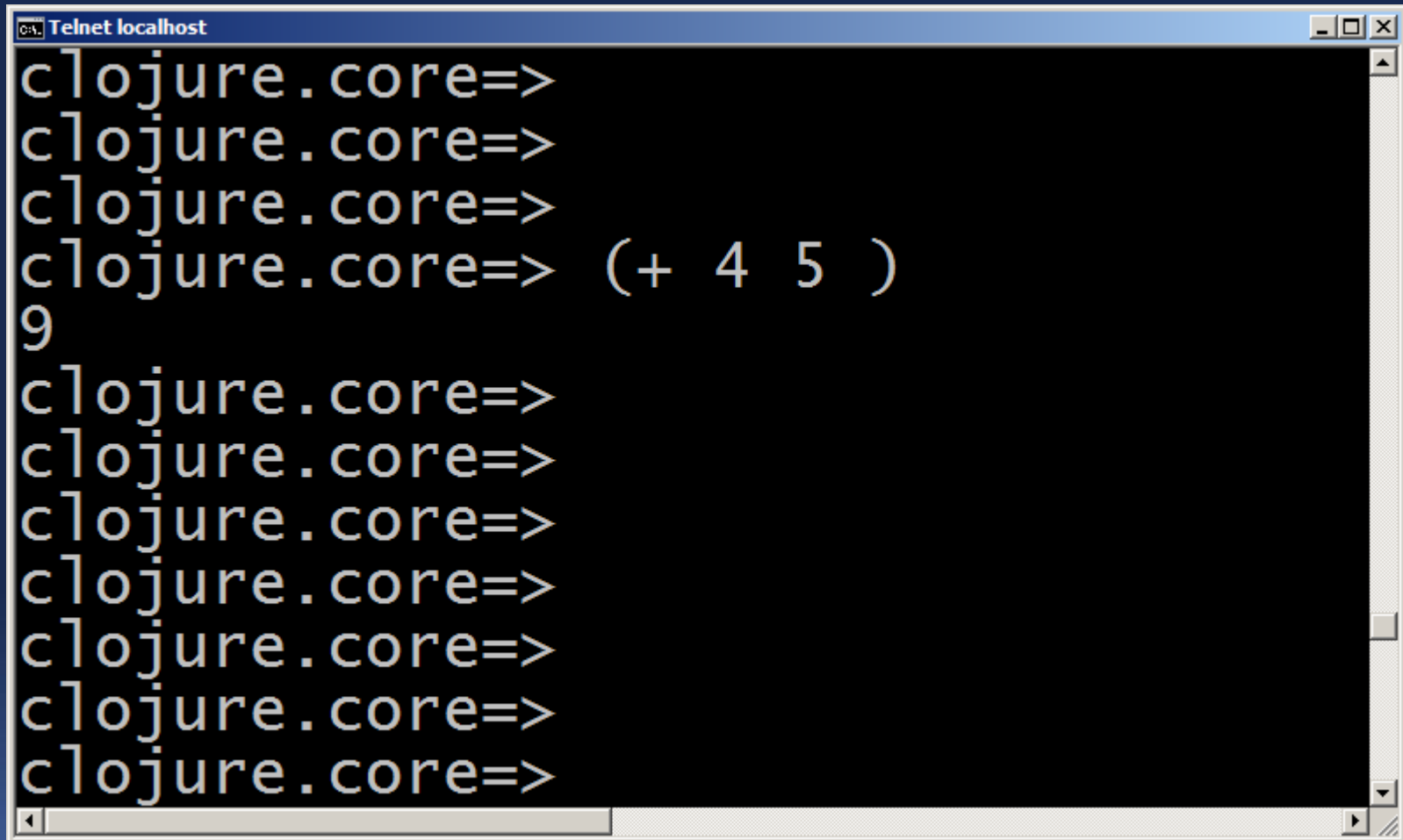


Conectando com Telnet na porta 4444



```
C:\>telnet localhost 4444_
```

Conectando com Telnet na porta 4444



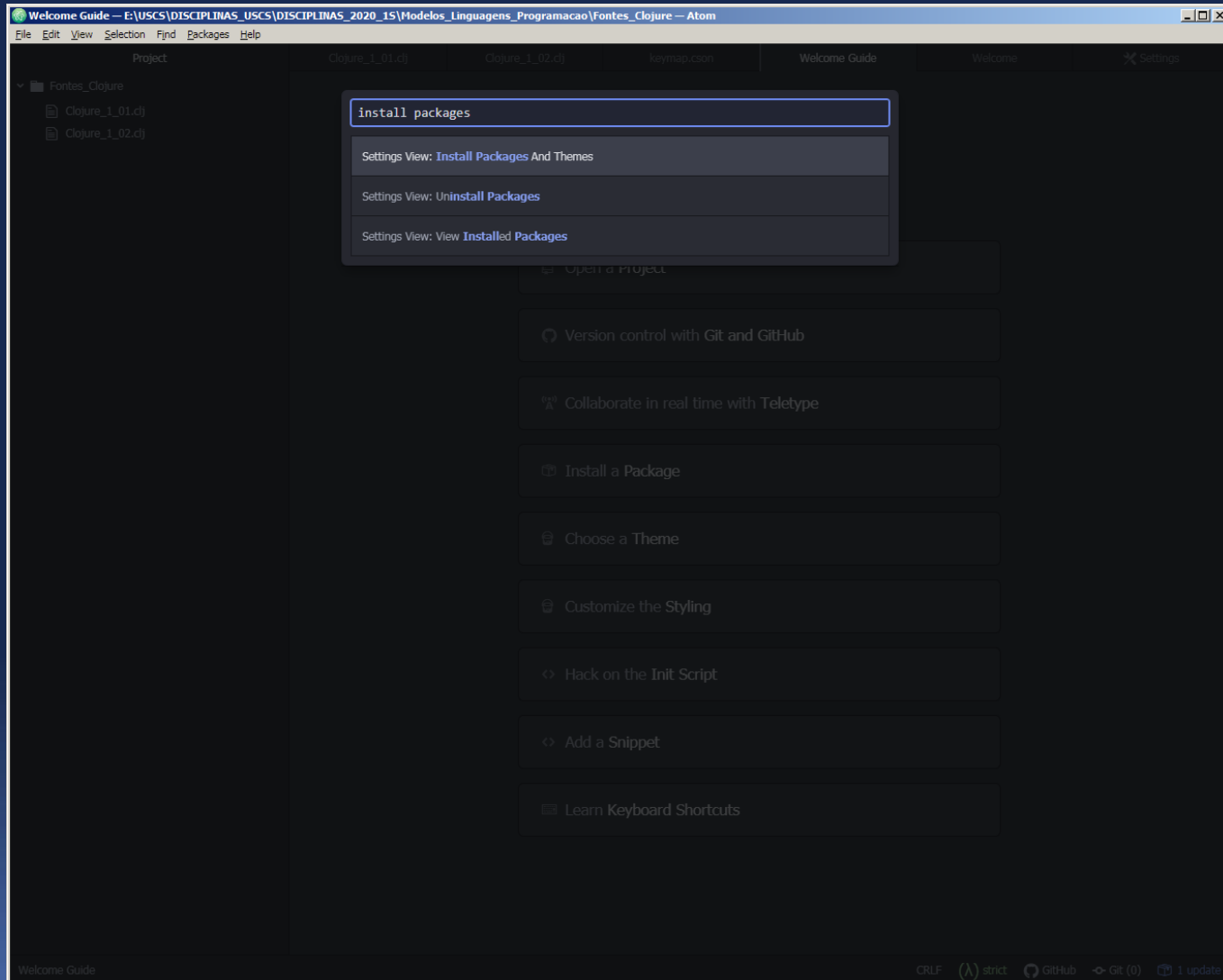
```
GA Telnet localhost
clojure.core=>
clojure.core=>
clojure.core=>
clojure.core=> (+ 4 5 )
9
clojure.core=>
clojure.core=>
clojure.core=>
clojure.core=>
clojure.core=>
clojure.core=>
clojure.core=>
```

Desenvolvimento Clojure com a IDE Atom



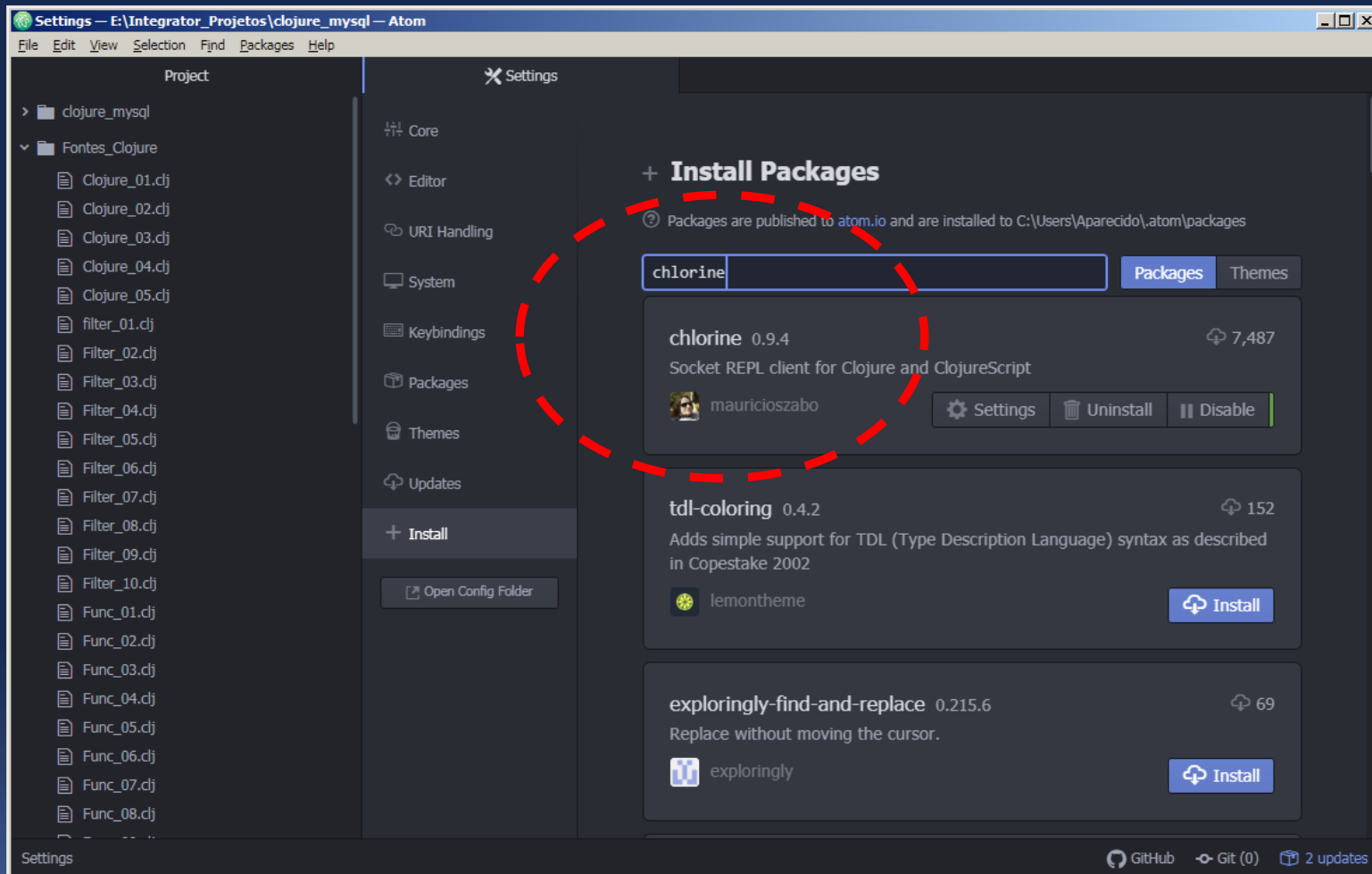
Atom – Package Chlorine

✓ **Ctrl + Shift + p** > **Install Packages and Themes**



Atom – Package Chlorine

✓ Instalar => package **Chlorine**



Chlorine – Configuração de Hot Keys

✓ Instalar => package **Chlorine**

- ✓ Ao se teclar **CTRL+SHIFT+p** e procurar por "**Open your keymap**", Atom irá abrir um arquivo pra customizar hotkeys;
- ✓ Nesse arquivo podem ser incluídas as configurações:

```
'atom-text-editor':  
  'ctrl-k': 'chlorine:clear-console'  
  'ctrl-shift-I': 'chlorine:load-file'  
  'ctrl-shift-l': 'chlorine:clear-inline-results'  
  'ctrl-shift-enter': 'chlorine:evaluate-block'  
  'ctrl-enter': 'chlorine:evaluate-top-block'  
  'ctrl-c': 'chlorine:break-evaluation'  
  'ctrl-d': 'chlorine:doc-for-var'
```



Chlorine – Configuração de Hot Keys

```
# Atom Flight Manual:
```

```
# http://flight-manual.atom.io/using-atom/sections/basic-custom
```

```
'atom-text-editor':
```

```
  'ctrl-k': 'chlorine:clear-console'
```

```
  'ctrl-shift-I': 'chlorine:load-file'
```

```
  'ctrl-shift-L': 'chlorine:clear-inline-results'
```

```
  'ctrl-shift-enter': 'chlorine:evaluate-block'
```

```
  'ctrl-enter': 'chlorine:evaluate-top-block'
```

```
  'ctrl-c': 'chlorine:break-evaluation'
```

```
  'ctrl-d': 'chlorine:doc-for-var'
```



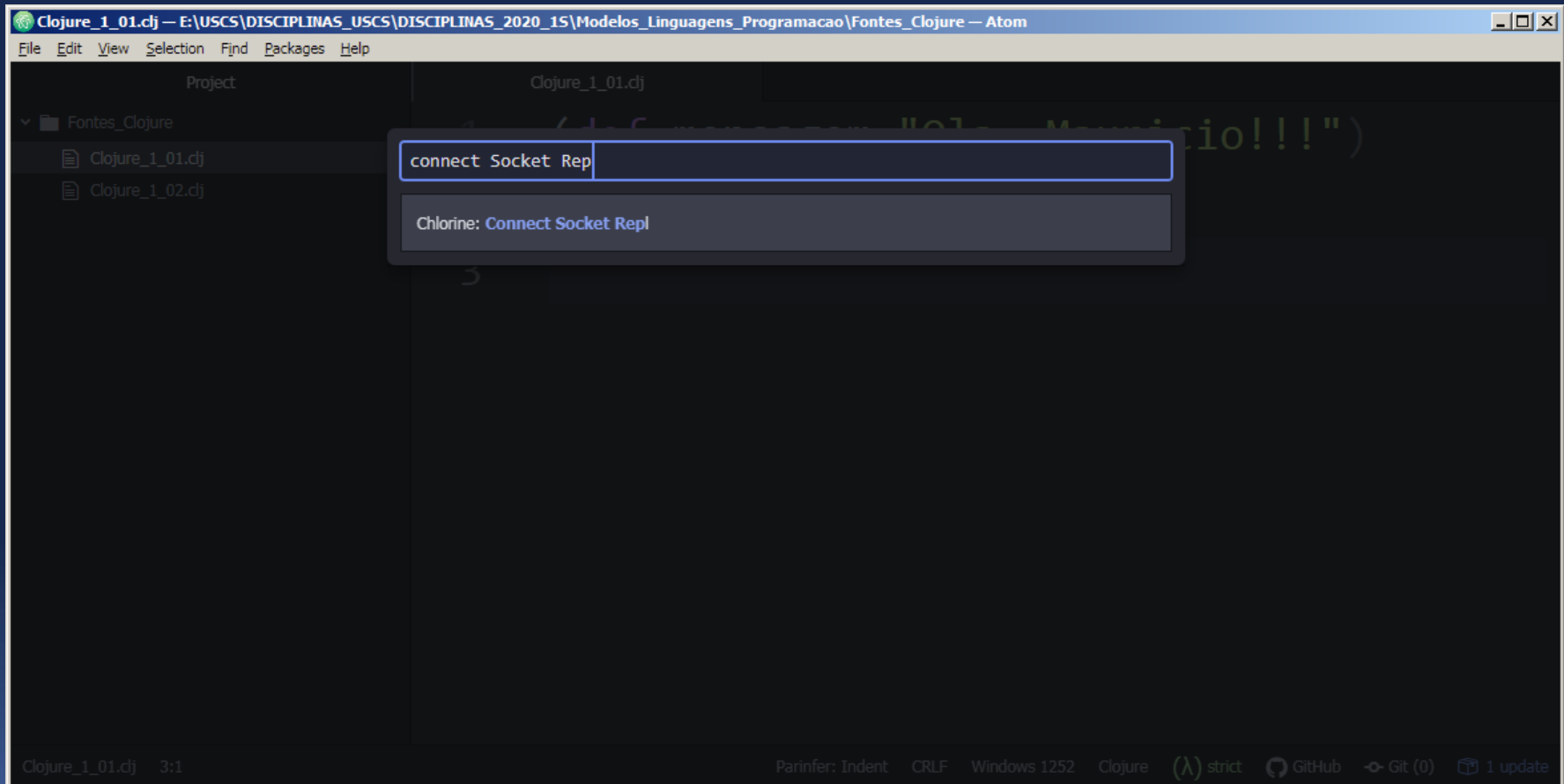


Conectando Atom no REPL remoto



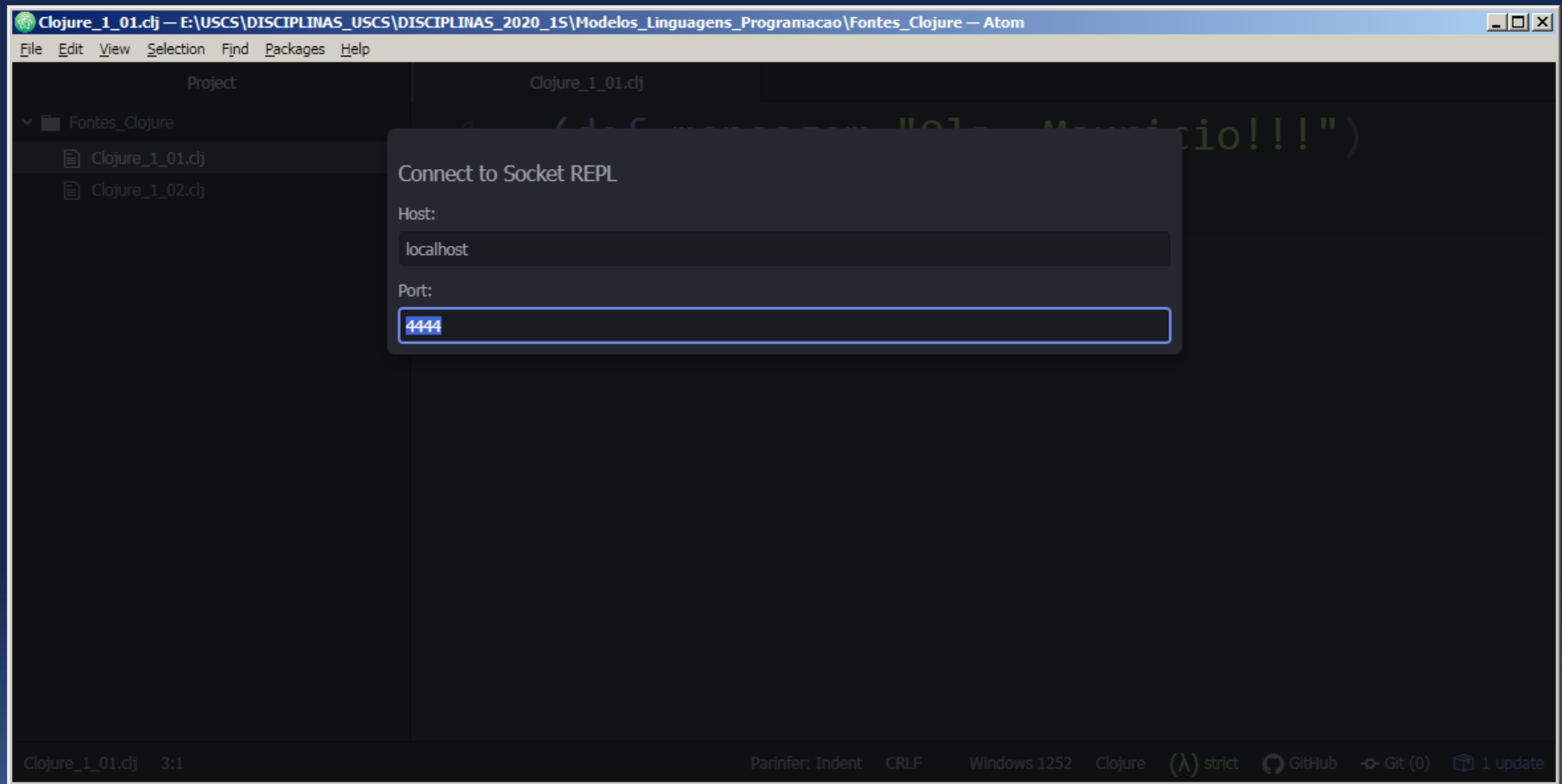
Atom – Package Chlorine

- ✓ Após instalação do Chlorine e subida do socket com REPL remoto
- ✓ **Ctrl + Shift + p** > **Connect Socket Repl**



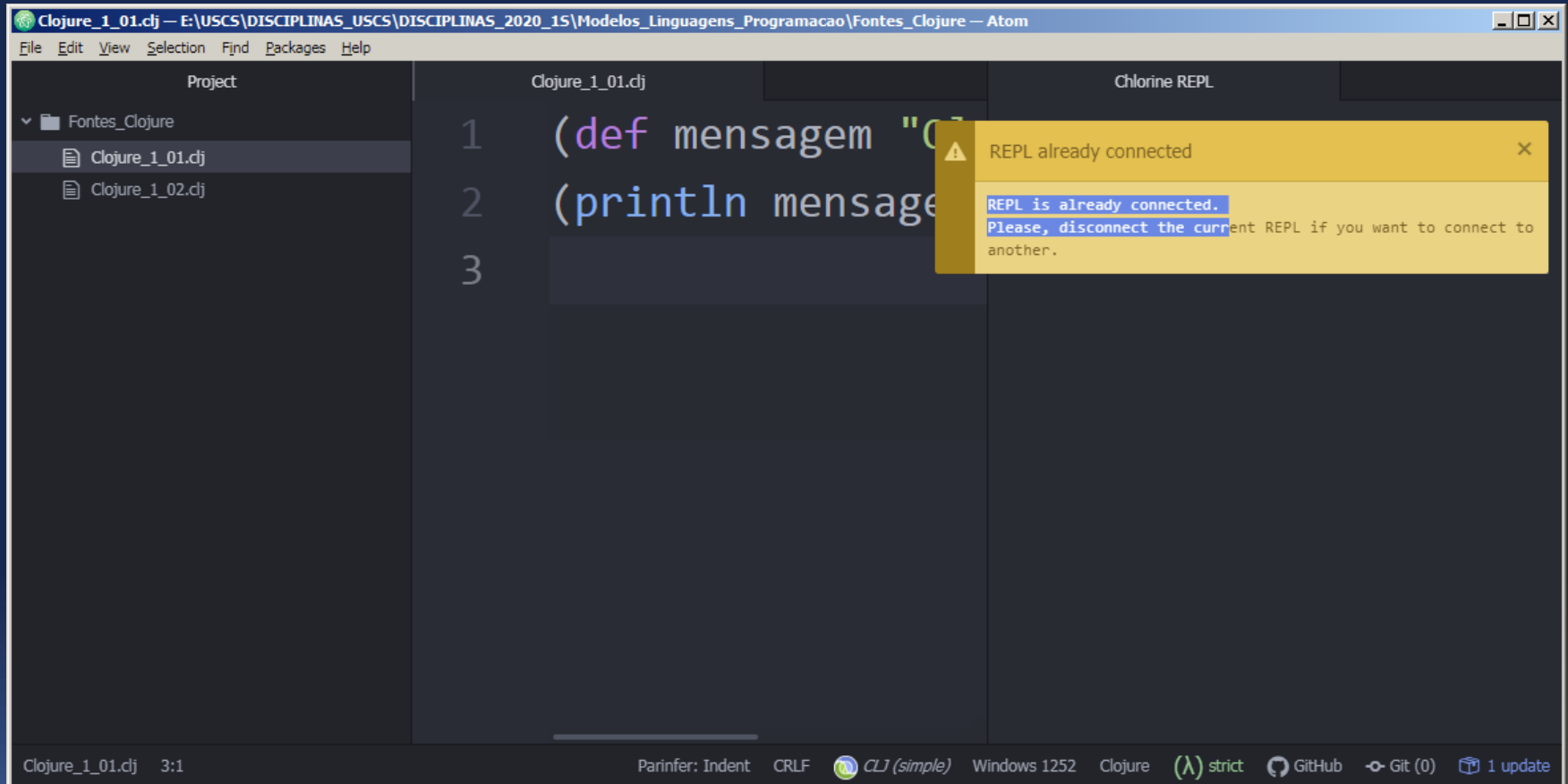
Atom – Package Chlorine

- ✓ Informar Host e porta onde o REPL remoto está sendo executado



Atom – Package Chlorine

✓REPL connected !!!

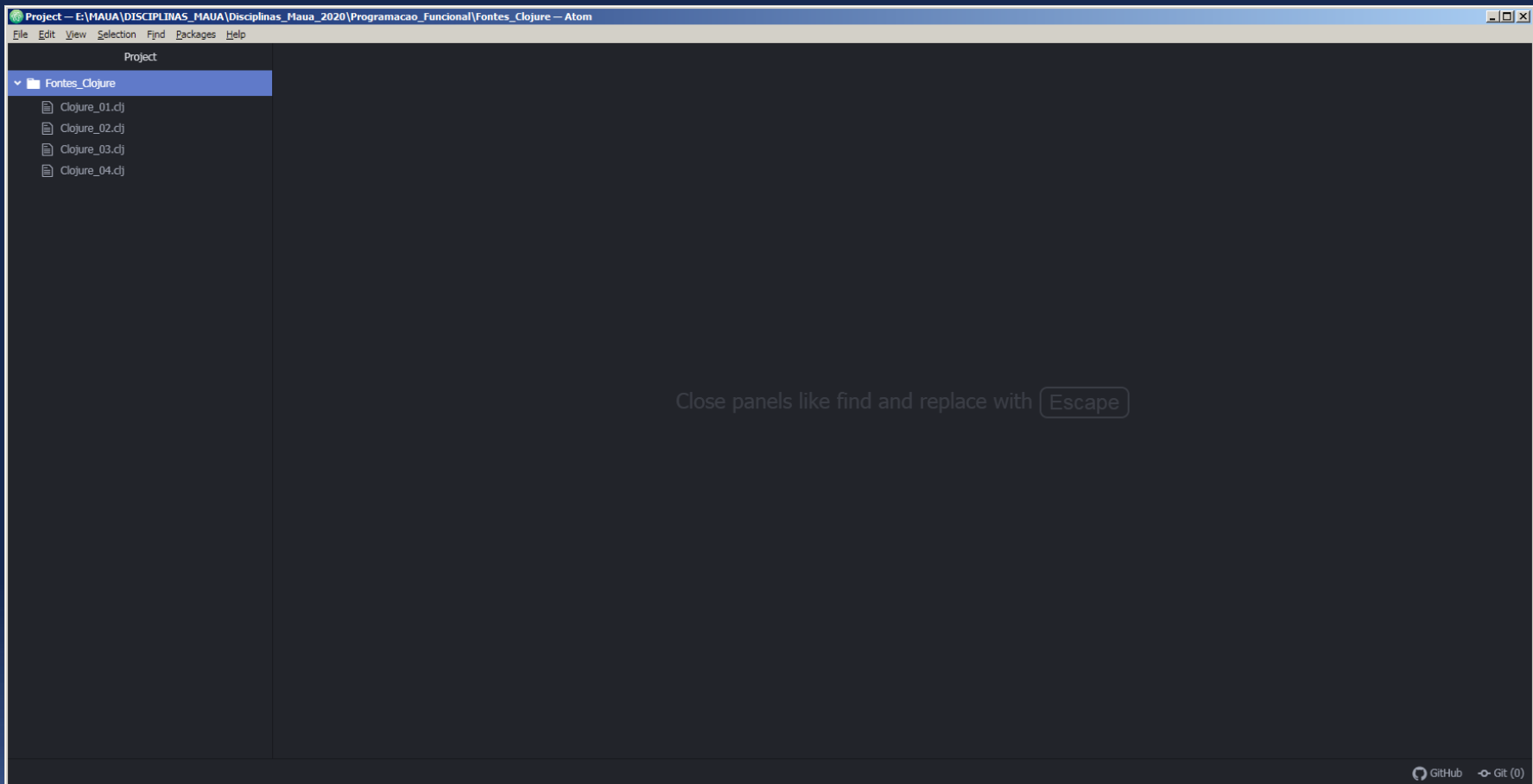


Entrando com código Clojure no Atom



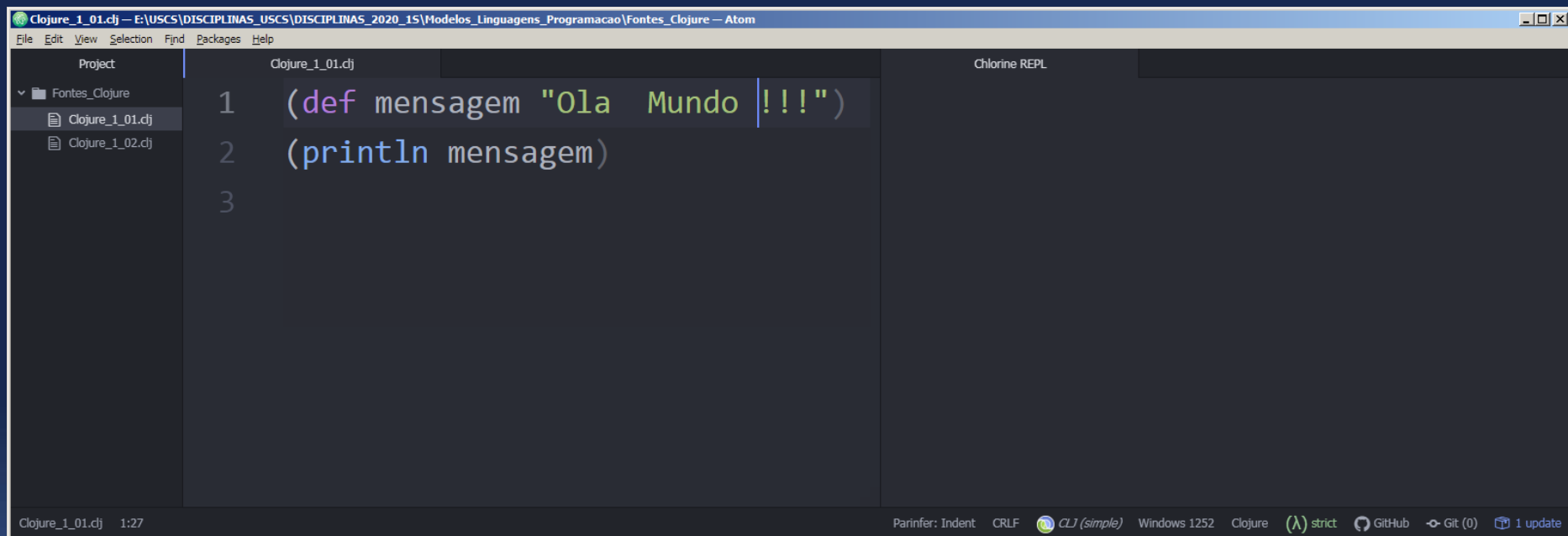
Ativar Atom

✓ File > Open Folder



Atom – Package Chlorine

✓ Escrever código **Clojure** (extensão .clj)



Ativar Atom

✓ Escrevendo código Clojure no Atom

```
Closure_01.cj
1  (def a (+ 20 30))
2
3  (def b (+ 4 3))
4
5  ✓ (defn somar [b]
6      (+ a b ))
7
8  ✓ (defn somar-2 [a b]
9      (+ a b ))
10
11  (somar 10)
12
```



Atom – Package Chlorine

- ✓ Carregar código para Repl remoto
- ✓ **Ctrl + Shift + p** > **Load File**

```
(def a (+ 20 30))
```

```
(def b (+ 4 3))
```

```
(defn somar [b]  
  (+ a b ))
```

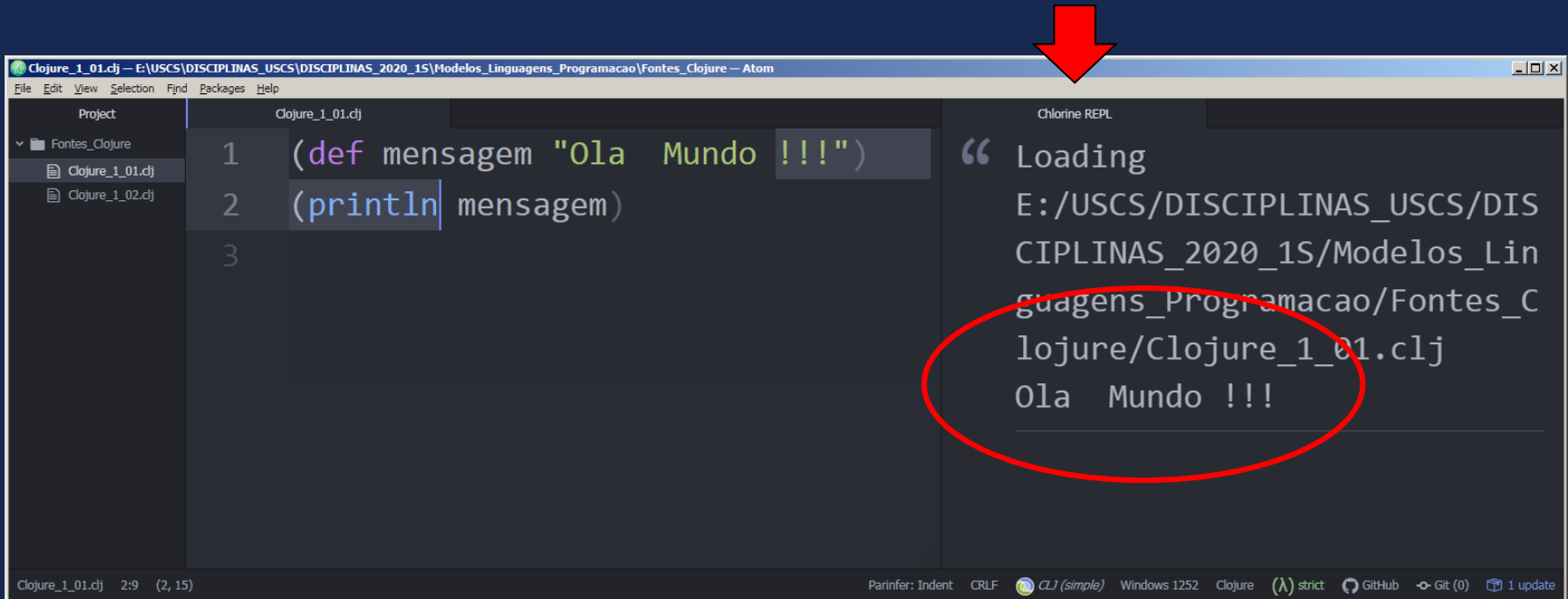
Chlorine: Load File

Ctrl+Shift+I



Atom – Package Chlorine

✓ Carga do código Clojure para o REPL



The screenshot shows the Atom editor interface with the Chlorine REPL package installed. The left sidebar shows the project structure with files `Clojure_1_01.clj` and `Clojure_1_02.clj`. The main editor displays the code in `Clojure_1_01.clj`:

```
1 (def mensagem "Ola Mundo !!!")
2 (println mensagem)
3
```

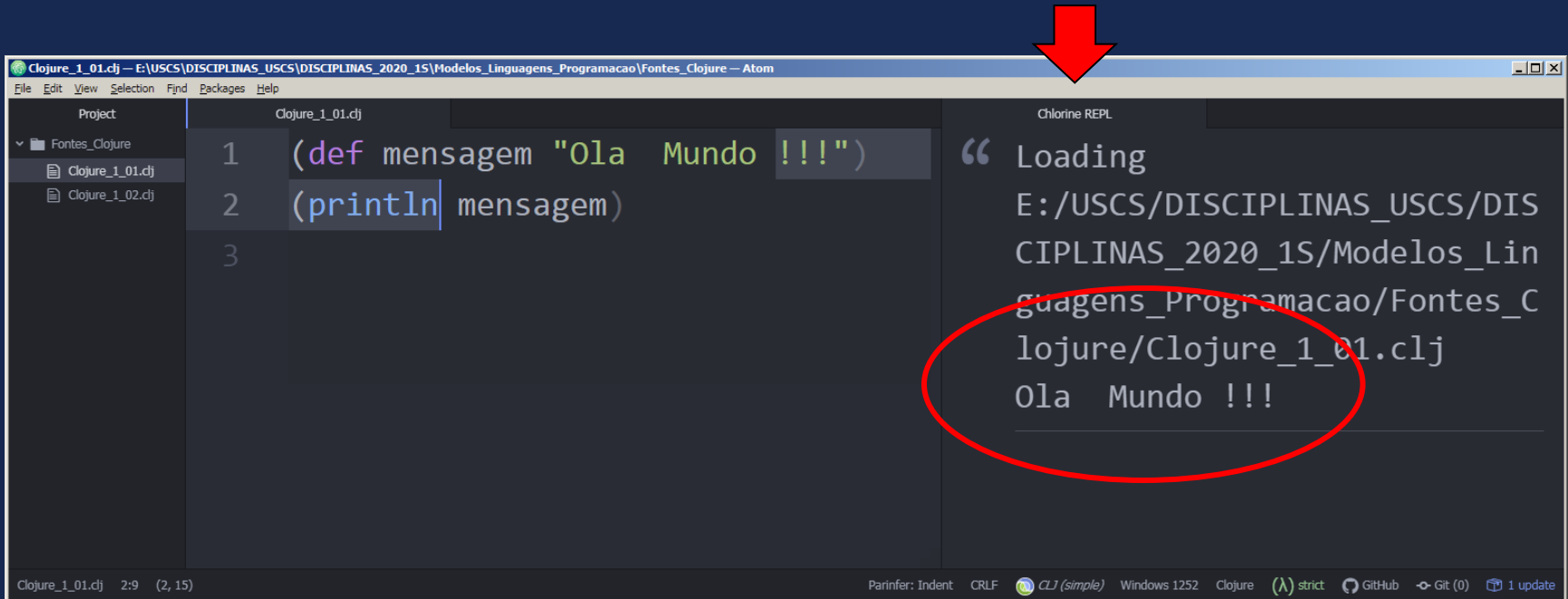
The right sidebar shows the Chlorine REPL output:

```
“ Loading
E:/USCS/DISCIPLINAS_USCS/DISCIPLINAS_2020_1S/Modelos_Linguagens_Programacao/Fontes_Clojure/Clojure_1_01.clj
Ola Mundo !!!
```

The output is circled in red. The status bar at the bottom shows the current file is `Clojure_1_01.clj` at line 2, column 15. The bottom right corner of the status bar shows the Clojure package is active and has 1 update.

Atom – Package Chlorine

✓ Carga do código Clojure para o REPL



The screenshot shows the Atom editor interface with the Chlorine REPL package installed. The left sidebar shows the project structure with files `Clojure_1_01.clj` and `Clojure_1_02.clj`. The main editor displays the code in `Clojure_1_01.clj`:

```
1 (def mensagem "Ola Mundo !!!")
2 (println mensagem)
3
```

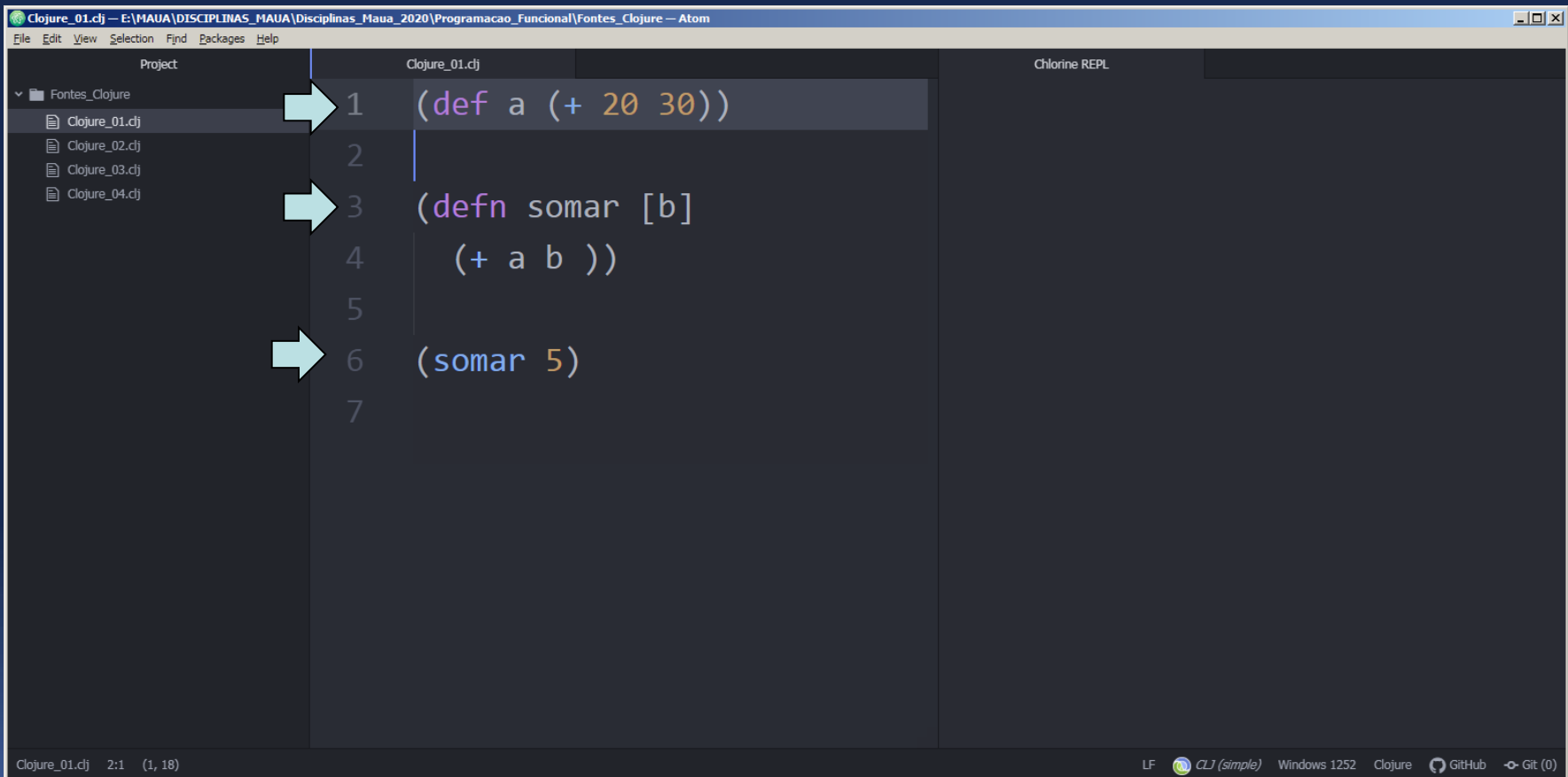
The right sidebar shows the Chlorine REPL output:

```
“ Loading
E:/USCS/DISCIPLINAS_USCS/DISCIPLINAS_2020_1S/Modelos_Linguagens_Programacao/Fontes_Clojure/Clojure_1_01.clj
Ola Mundo !!!
```

A red arrow points to the REPL output, and a red circle highlights the output text.

Evaluate "top-blocks"

- ✓ Hot Key: **Ctrl + Enter**
- ✓ Selecionar o top block desejado e teclar \Rightarrow **Ctrl + Enter**



The screenshot shows the Atom editor interface with a project named 'Fontes_Clojure'. The file 'Clojure_01.cj' is open, displaying the following Clojure code:

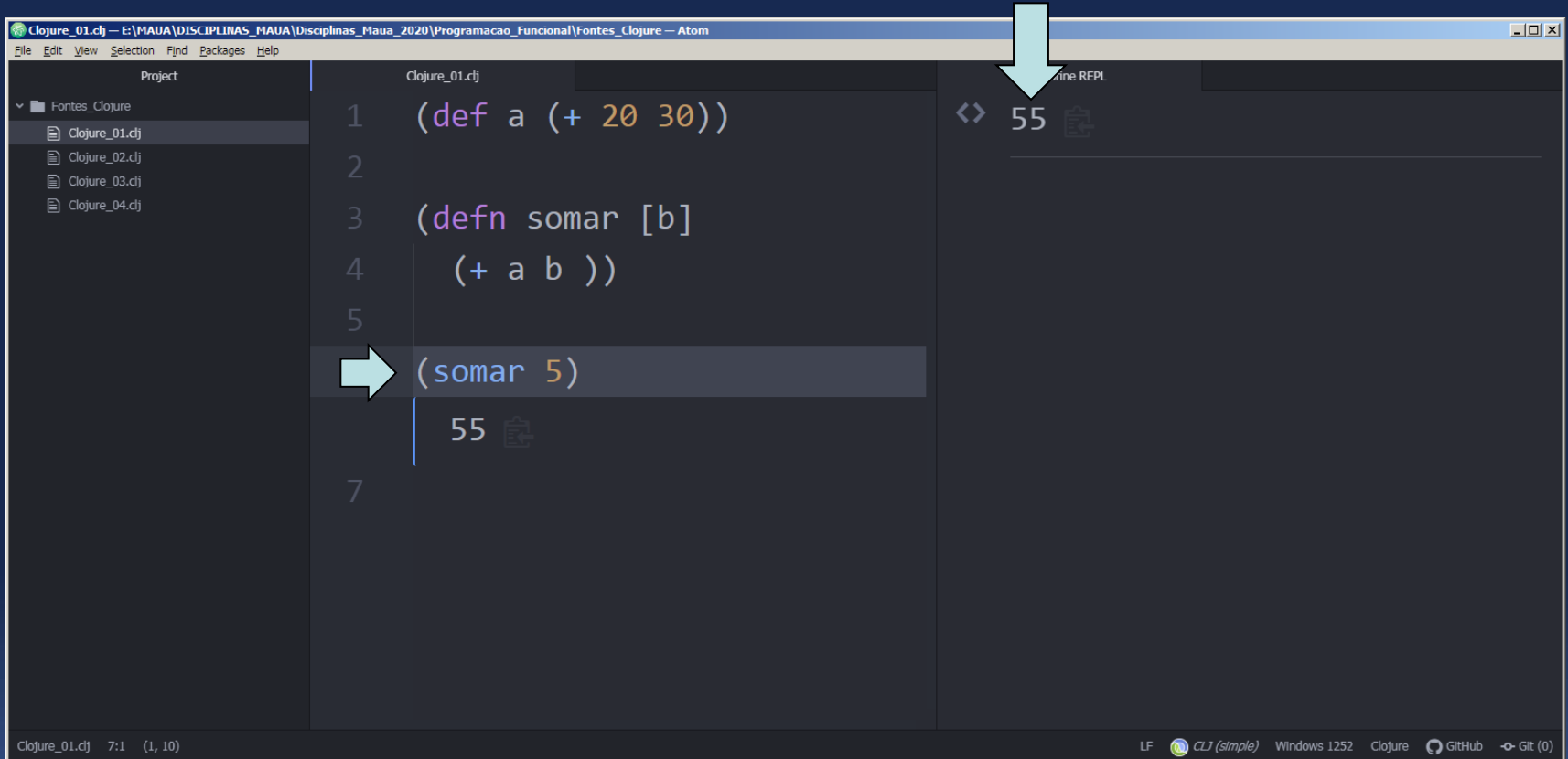
```
1 (def a (+ 20 30))
2
3 (defn somar [b]
4   (+ a b ))
5
6 (somar 5)
7
```

Three arrows point from the left margin to the first, third, and sixth lines of code, indicating the 'top-blocks' to be evaluated. The status bar at the bottom shows 'Clojure_01.cj 2:1 (1, 18)' and various icons including CLJ (simple), Windows 1252, Clojure, GitHub, and Git (0).



Evaluate "top-blocks"

✓ Por exemplo, marcar o top block (somar 5) e teclar => **Ctrl + Enter**



```
Clojure_01.cj
1 (def a (+ 20 30))
2
3 (defn somar [b]
4   (+ a b ))
5
6 (somar 5)
7
55
```

Project: Fontes_Clojure

File Edit View Selection Find Packages Help

Online REPL

55

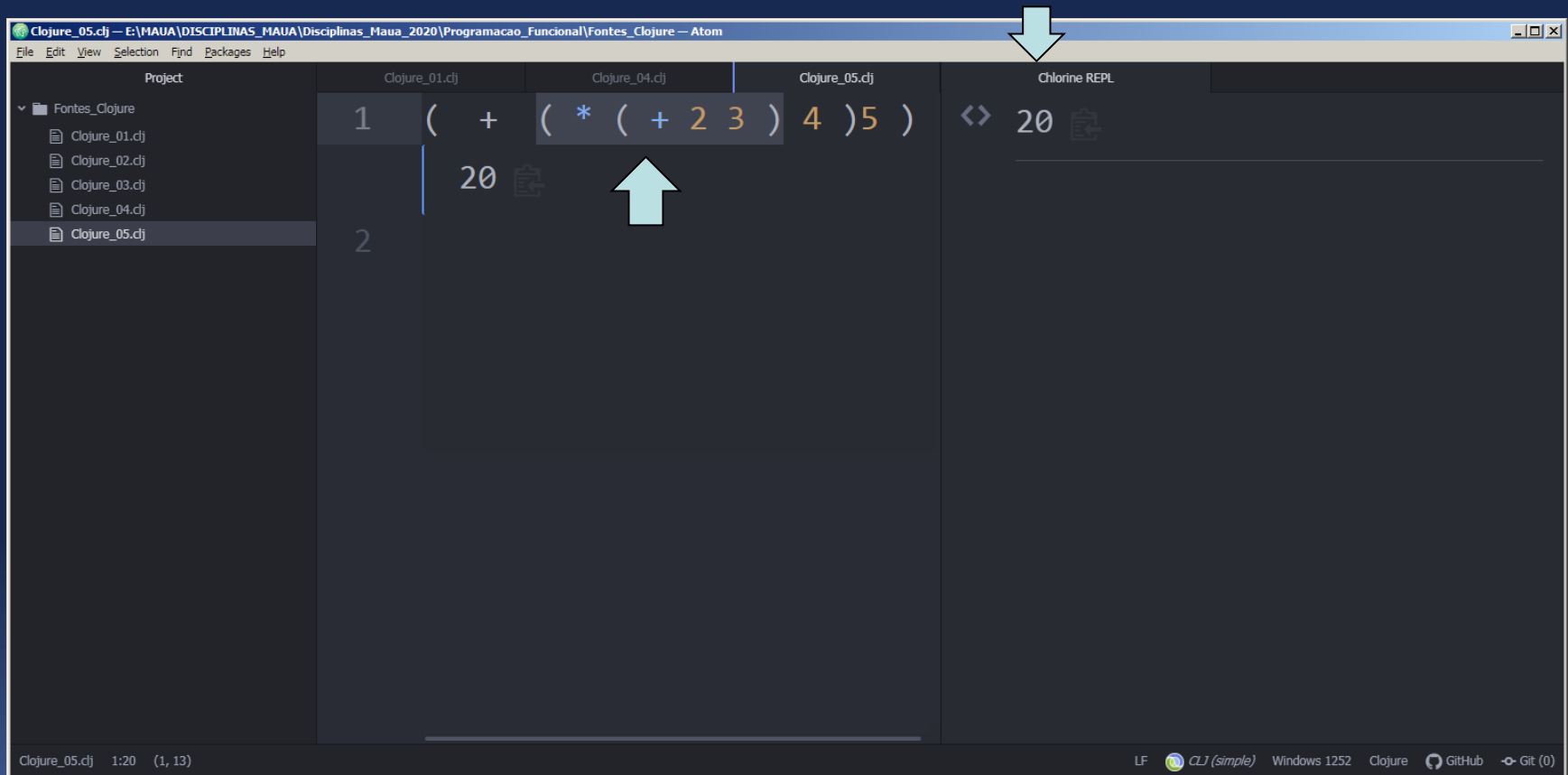
Clojure_01.cj 7:1 (1, 10)

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



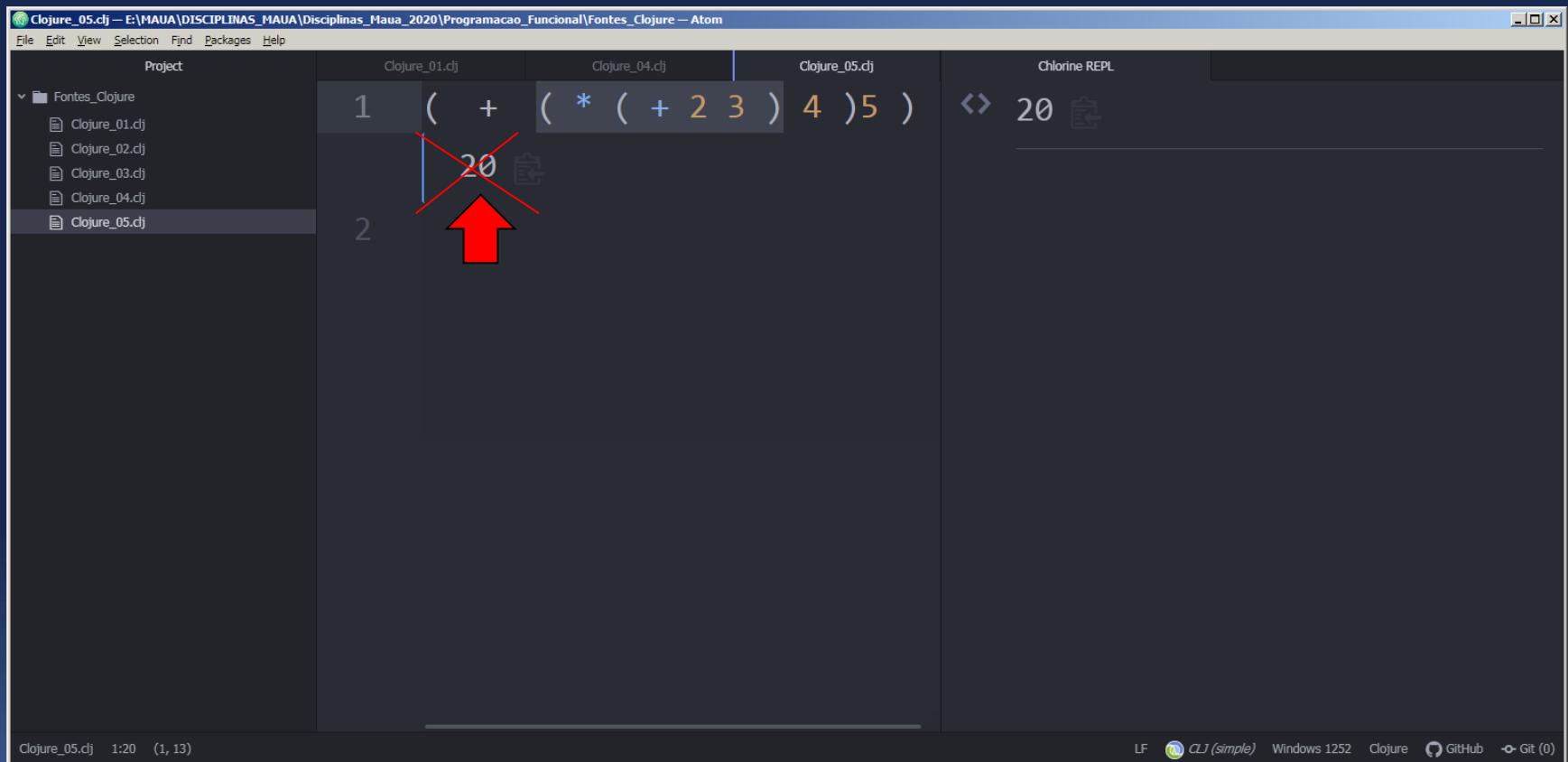
Evaluate "blocks"

- ✓ Hot Key: **Ctrl + Shift + Enter**
- ✓ Selecionar o block desejado e teclar => **Ctrl + Shift + Enter**



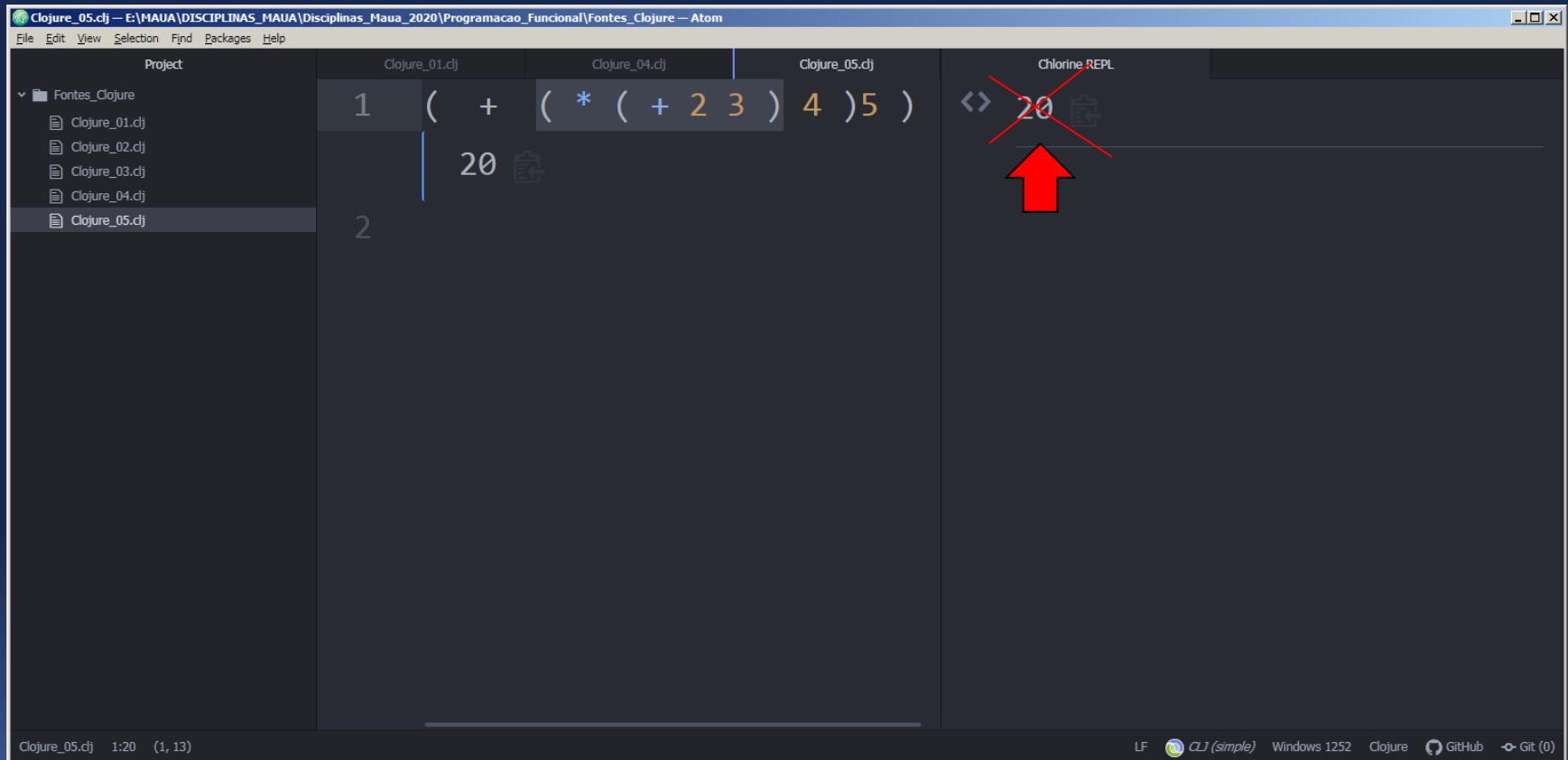
Clear Inline Results

✓ Pode ser feito pela hot key: **Ctrl + Shift + L**



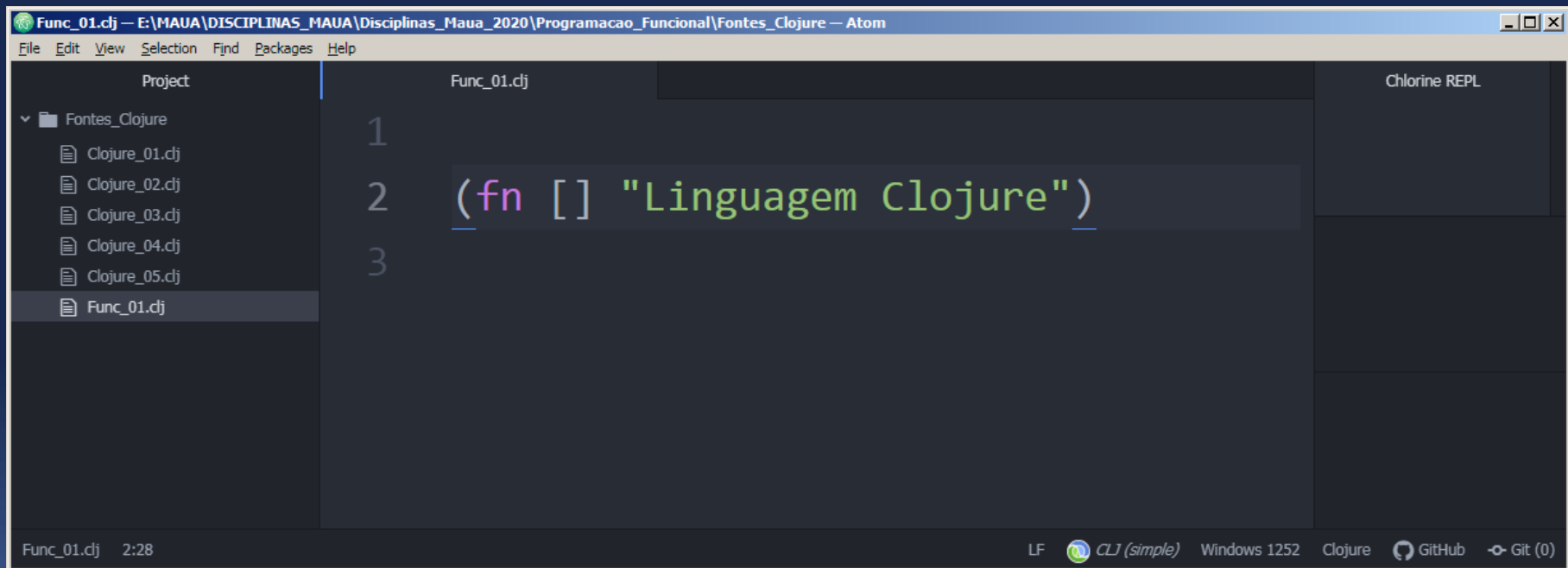
Clear Console

✓ Pode ser feito pela hot key: **Ctrl +k**



Lembrando ... Funções

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

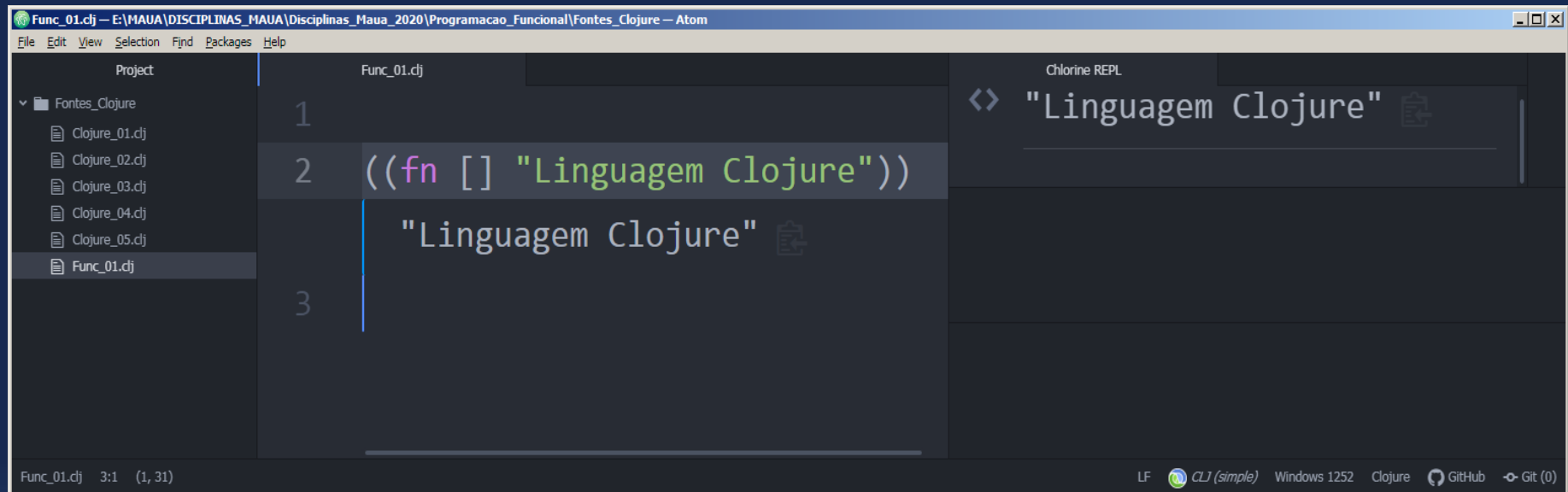


```
Func_01.cj — E:\MAUA\DISCIPLINAS_MAU\Disciplinas_Maua_2020\Programacao_Funcional\Fontes_Clojure — Atom
File Edit View Selection Find Packages Help
Project Fontes_Clojure
  Clojure_01.cj
  Clojure_02.cj
  Clojure_03.cj
  Clojure_04.cj
  Clojure_05.cj
  Func_01.cj
Func_01.cj
1
2 (fn [] "Linguagem Clojure")
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 text editor interface. On the left, a project sidebar displays 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 is open to 'Func_01.cj' and contains the following Clojure code:

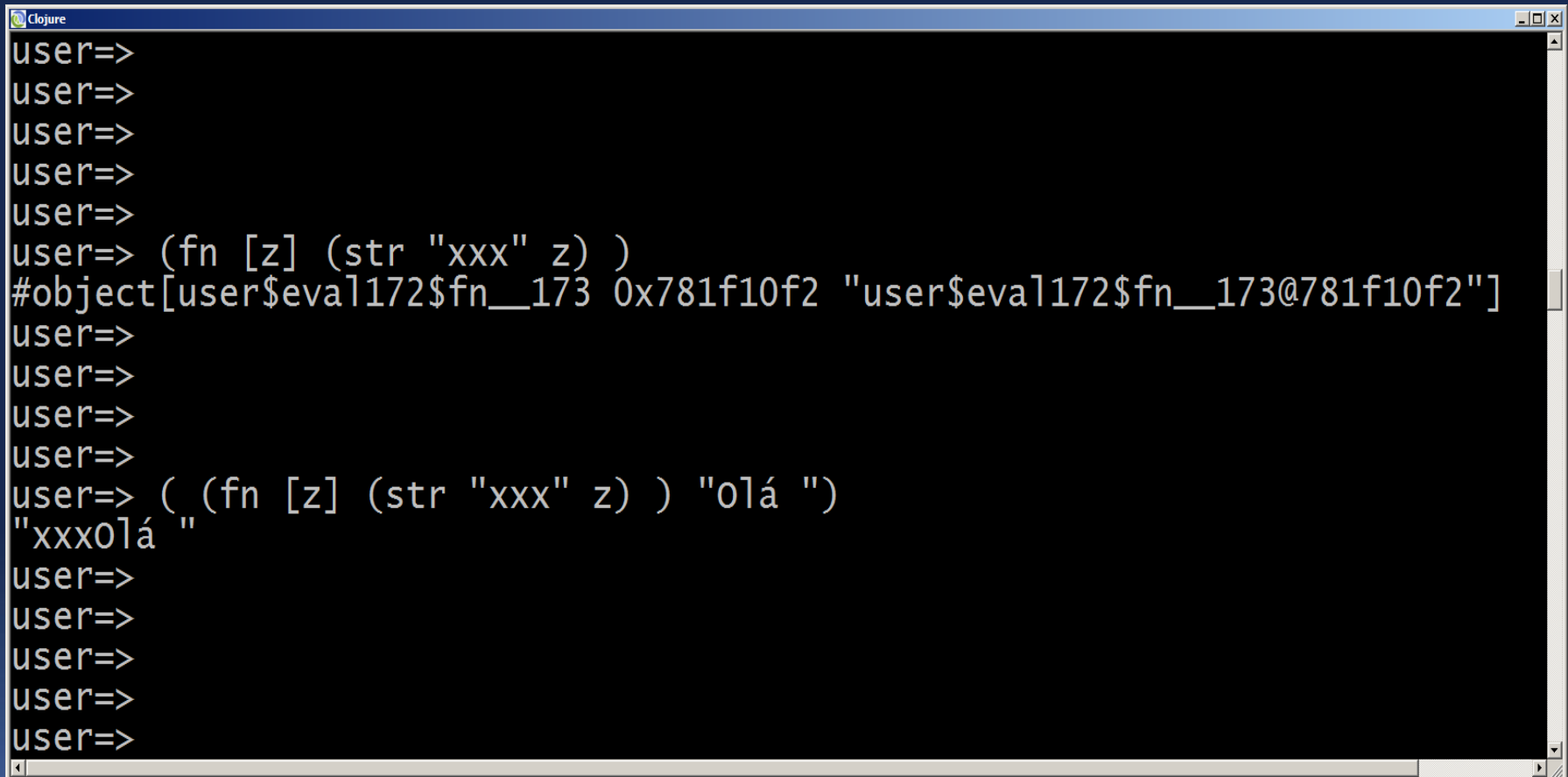
```
1  
2 ((fn [] "Linguagem Clojure"))  
3 "Linguagem Clojure"
```

On the right side of the editor, there is a 'Chlorine REPL' window. It shows the output of the function call: '<> "Linguagem Clojure"'. The status bar at the bottom of the editor indicates the current file is 'Func_01.cj' at line 3, column 1, with a cursor at (1, 31). It also shows the Clojure version 'CLJ (simple)', the operating system 'Windows 1252', and the Clojure version 'Clojure'. There are also links to 'GitHub' and 'Git (0)'.



Lembrando ... Funções

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

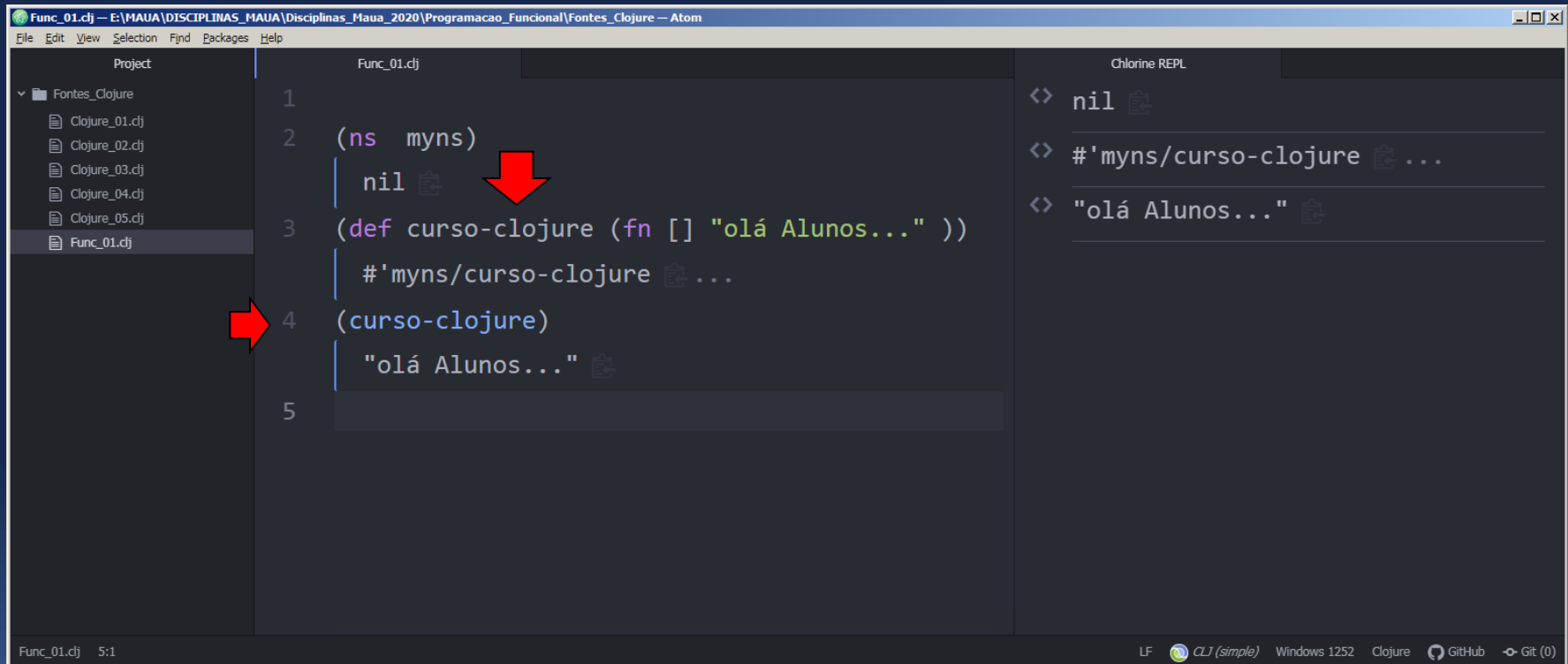


```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (fn [z] (str "xxx" z) )
#object[user$eval172$fn__173 0x781f10f2 "user$eval172$fn__173@781f10f2"]
user=>
user=>
user=>
user=>
user=> ( (fn [z] (str "xxx" z) ) "olá ")
"xxx0lá "
user=>
user=>
user=>
user=>
user=>
```



Lembrando ... Funções

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



```
Func_01.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_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
```

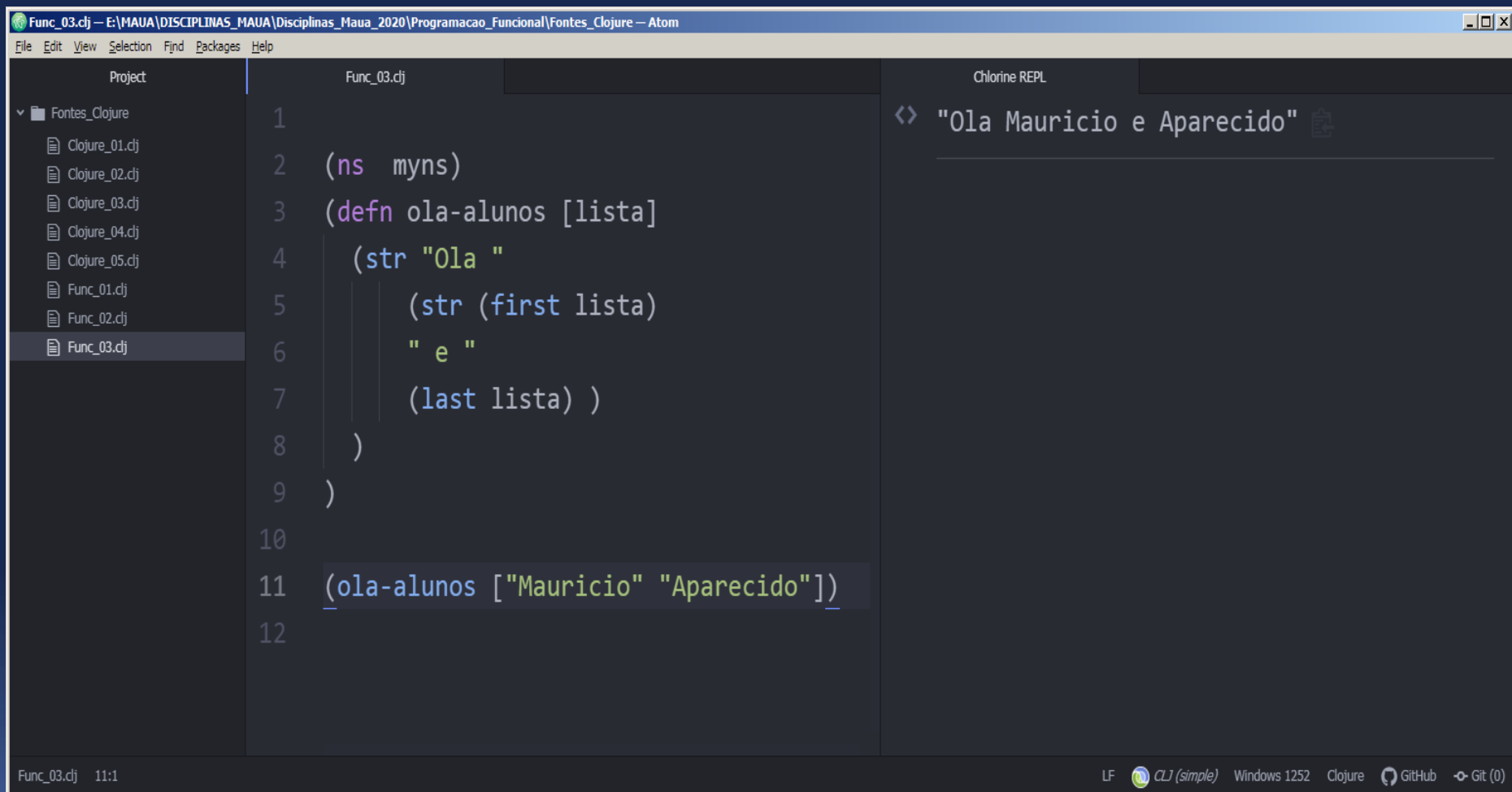
Two red arrows highlight the process: one points to the `defn` keyword on line 3, and the other points to the function call `(ola-alunos)` on line 4.

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. It also shows the 'CLJ (simple)' package is active, along with window and package information.

Argumentos para Funções



The screenshot shows the Atom text editor interface. The left sidebar displays a project tree with a folder named 'Fontes_Clojure' containing several files. The main editor area shows a Clojure script in 'Func_03.cj' with the following 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
```

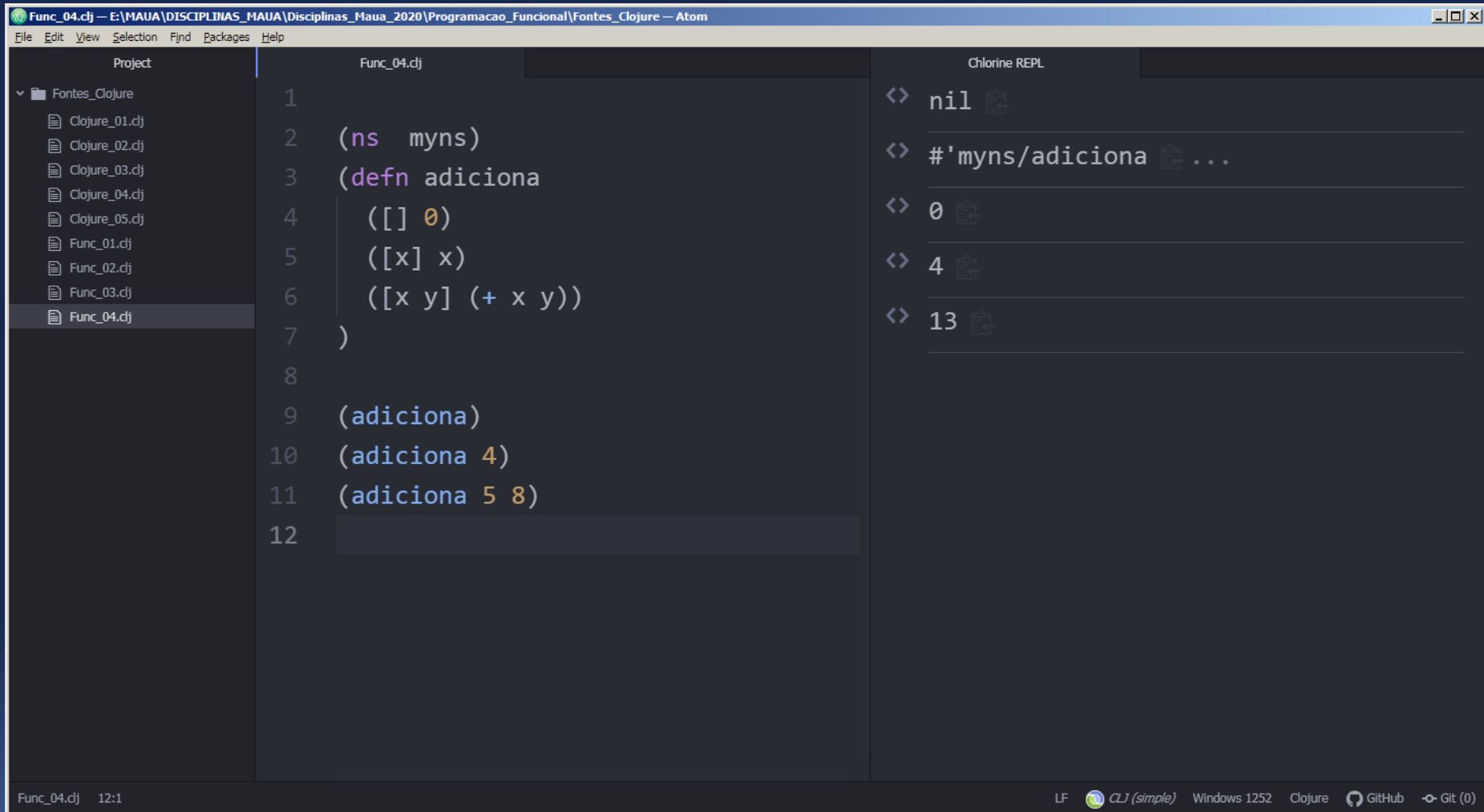
The right sidebar shows the 'Chlorine REPL' with the output of the last expression:

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

The status bar at the bottom indicates the current file is 'Func_03.cj' at line 11, column 1. It also shows various icons for language mode (CLJ), window count (1252), and other tools like Clojure, GitHub, and Git.



Funções com multivariadas



The screenshot displays the Atom editor interface with a project named 'Fontes_Clojure'. The file 'Func_04.clj' is open, showing the following Clojure code:

```
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
```

The Chlorine REPL on the right shows the execution of the code:

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

The status bar at the bottom indicates the file is 'Func_04.clj' at line 12, column 1. The bottom right corner shows the 'CLJ (simple)' REPL running on 'Windows 1252' with 'Clojure' and 'Git (0)' icons.



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**);

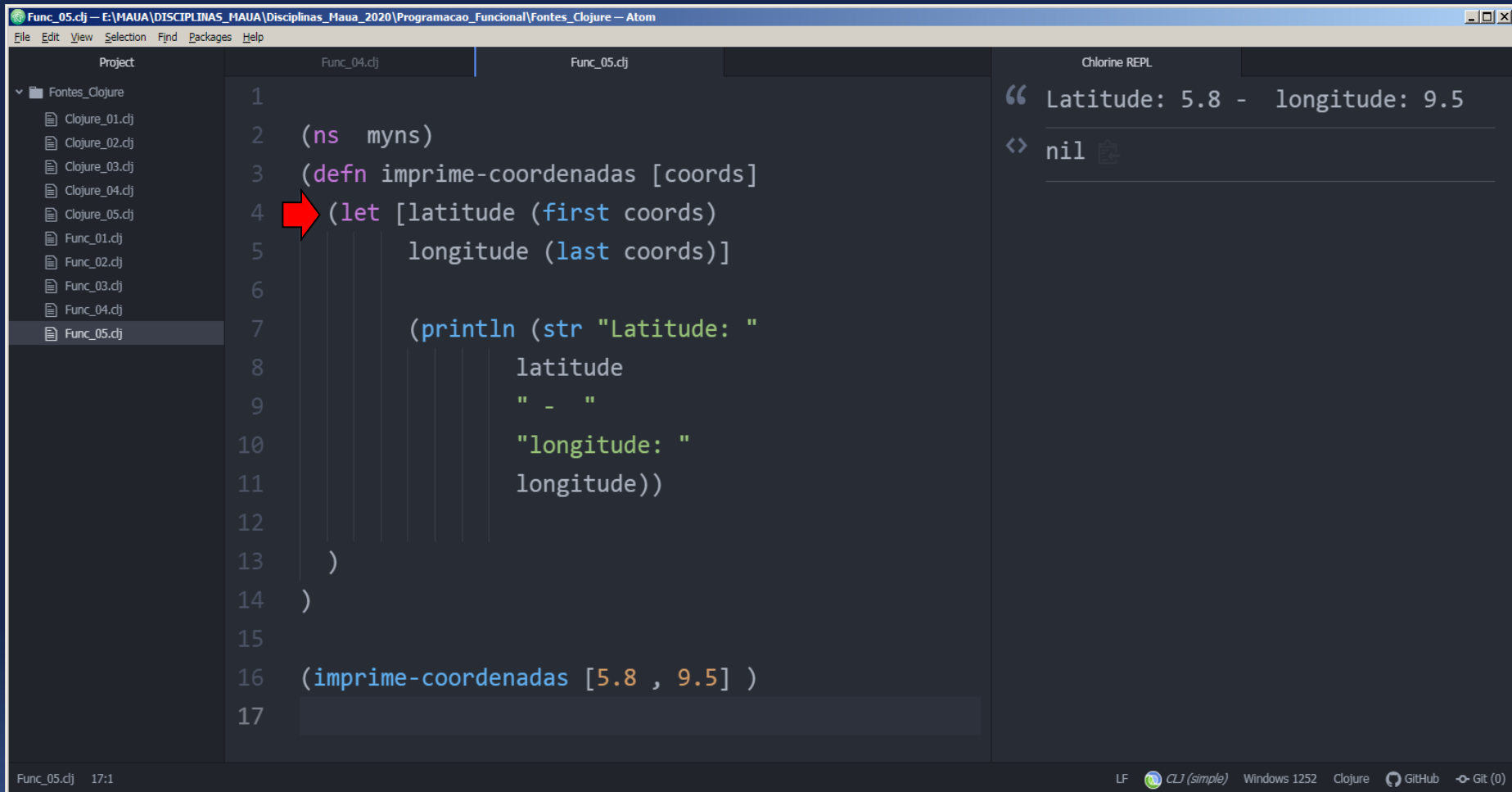


Desestruturação Sequencial

- ✓ 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



The screenshot shows the Atom editor with a project named 'Fontes_Clojure'. The file 'Func_05.clj' is open, showing the following code:

```
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
```

A red arrow points to the `(let` statement on line 4. The right-hand pane shows the Chlorine REPL with the output of the function call:

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



Exemplo – Desestruturação

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

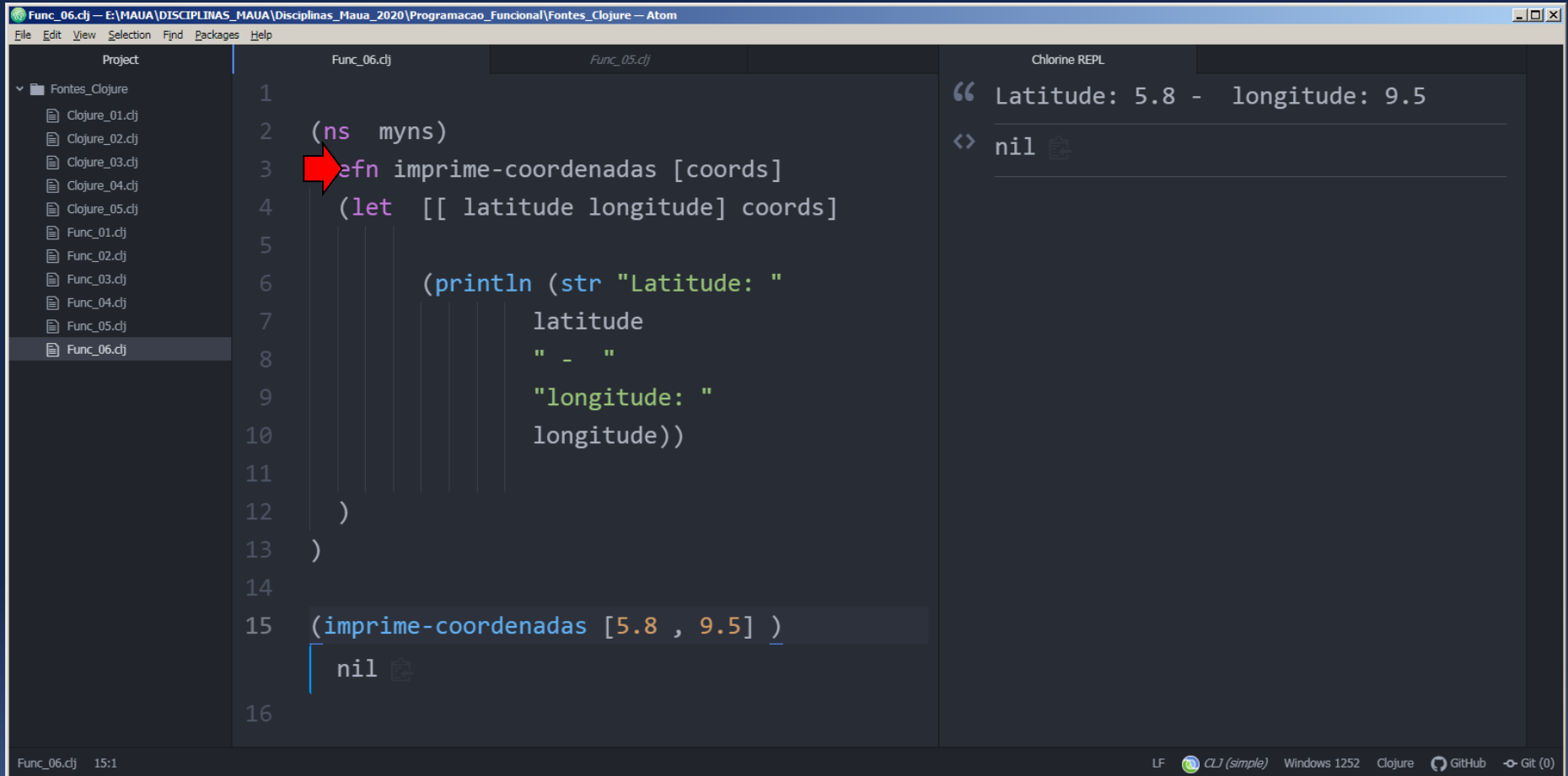
    (println (str "Latitude: "
                  latitude
                  " - "
                  "longitude: "
                  longitude)))
  )
)

(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



The screenshot shows the Atom editor with a project named 'Fontes_Clojure'. The file 'Func_06.clj' is open, showing the following code:

```
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
```

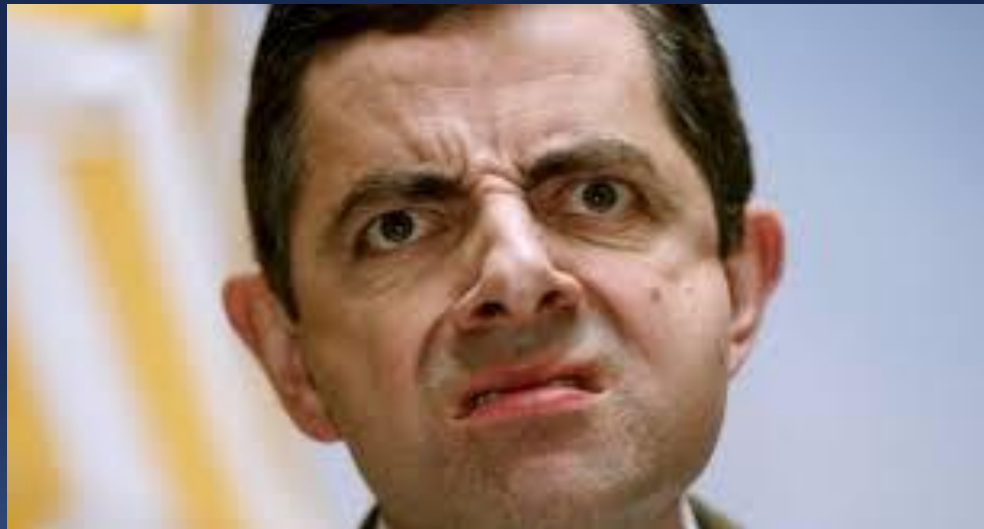
A red arrow points to the **defn** keyword on line 3. The right panel shows the Chlorine REPL output:

```
“ 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 Clojure_05.cj, and Func_01.cj through Func_07.cj.
- Editor:** Displays the content of 'Func_07.cj' with the following code:

```
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 of the code, displaying the output `1 2 3` and the return value `nil`.
- Status Bar:** Indicates the current Clojure version as 'CLJ (simple)' and includes links to GitHub and Git.

- ✓ 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

```
Func_08.clj — E:\MAUA\DISCIPLINAS_MAUUA\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
  Func_06.clj
  Func_07.clj
  Func_08.clj

Func_08.clj
1 (ns myns)
2 (let [[ a b c] '(5 6 7)]
3
4   (println a b c )
5
6 )
7 nil

Chlorine REPL
“ 5 6 7
<> nil
```



Desestruturação Associativa



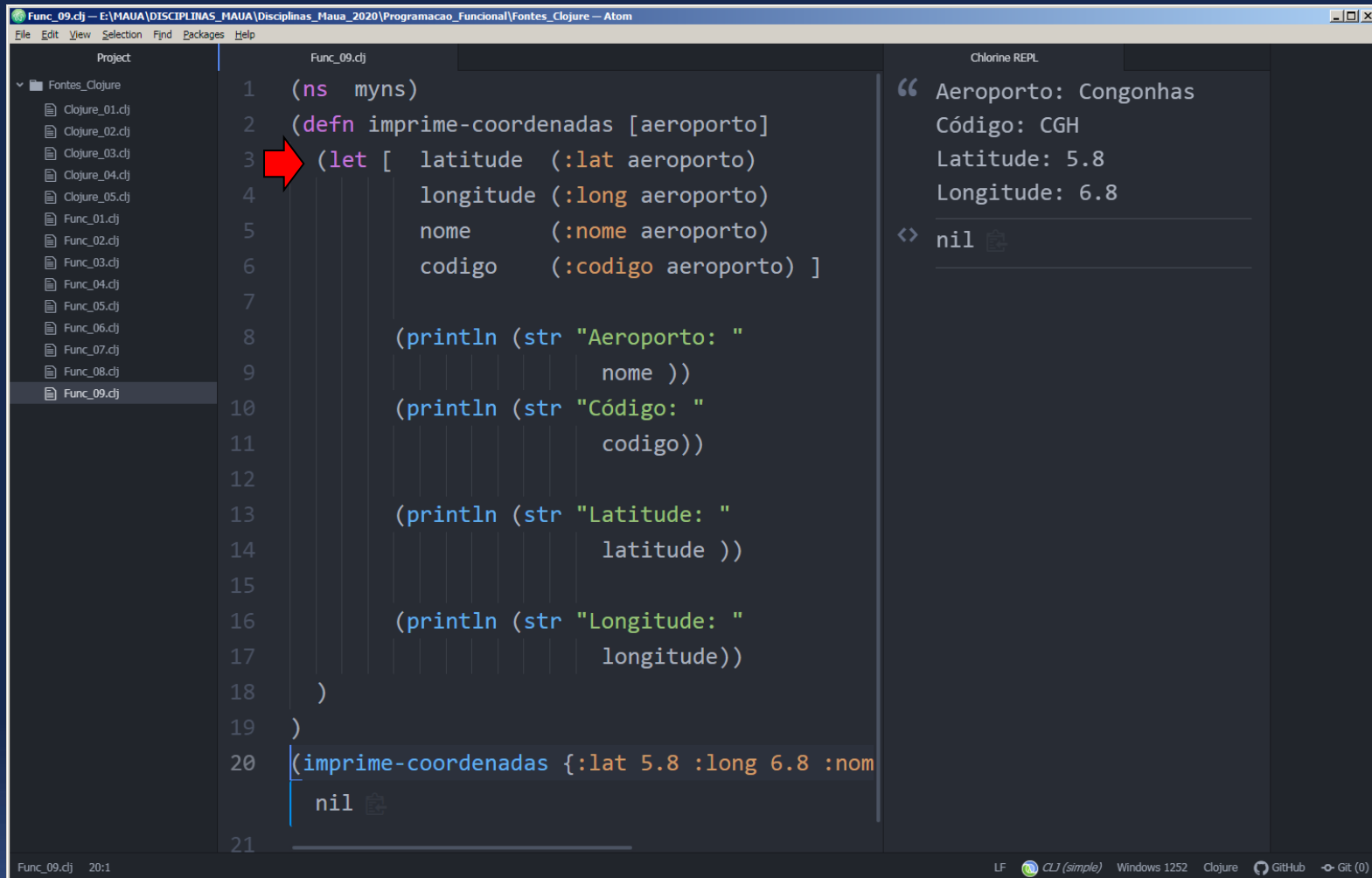
Desestruturação Associativa

- ✓ 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.

```
(ns myns)

(defn imprime-coordenadas [aeroporto]
  (let [ latitude (:lat aeroporto)
          longitude (:long aeroporto)
          nome (:nome aeroporto)
          codigo (:codigo aeroporto) ]

    (println (str "Aeroporto: "
                  nome ))
    (println (str "Código: "
                  codigo))

    (println (str "Latitude: "
                  latitude ))

    (println (str "Longitude: "
                  longitude)))
  )
)

(imprime-coordenadas {:lat 5.8 :long 6.8 :nom
nil
```

Chlorine REPL

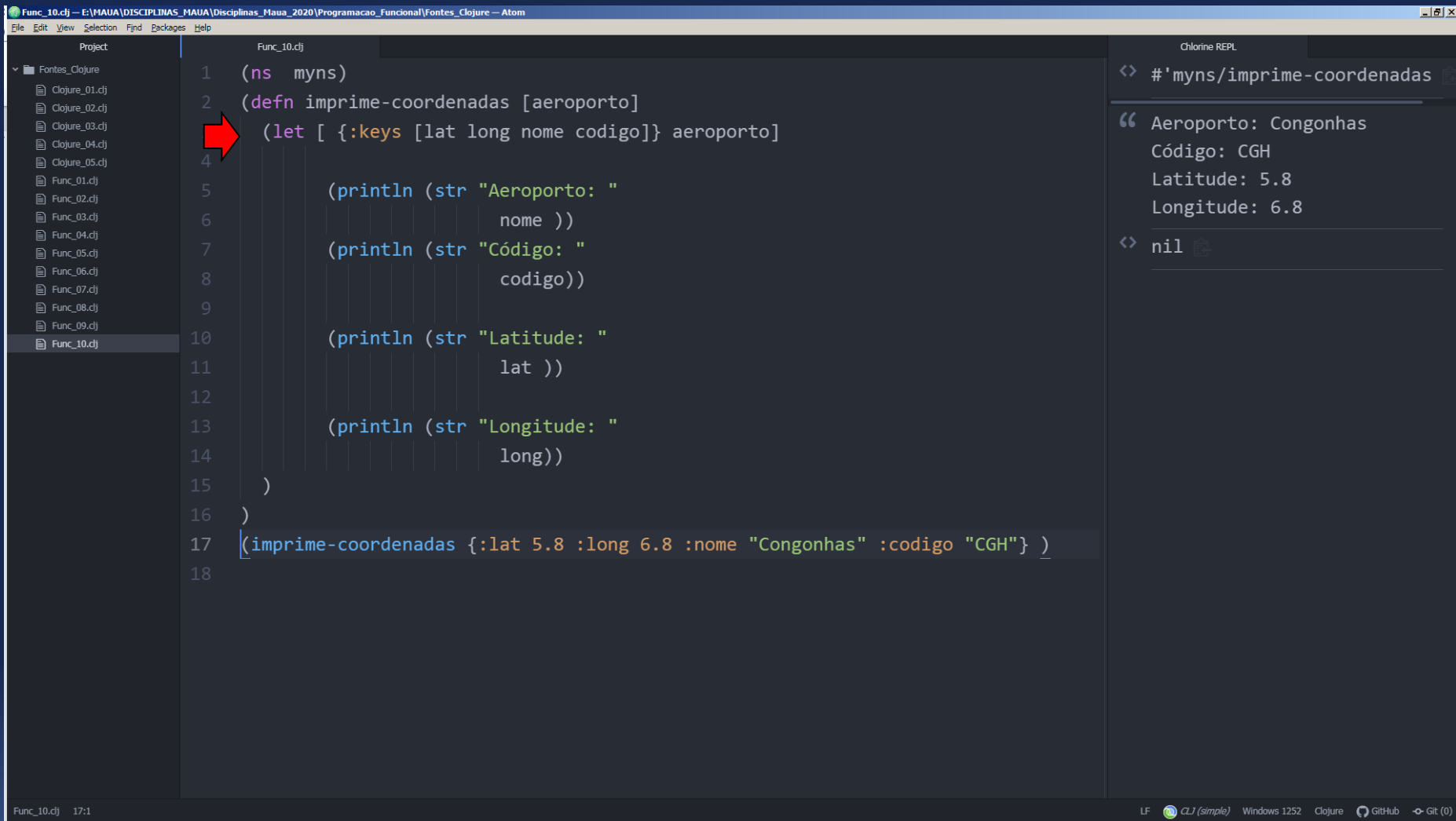
```
" Aeroporto: Congonhas
Código: CGH
Latitude: 5.8
Longitude: 6.8

<> 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



The screenshot shows the Atom editor with a project named 'Func_10.clj'. The left sidebar shows a file explorer with various Clojure files. The main editor displays the following Clojure code:

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

A red arrow points to the destructuring binding in the `let` statement on line 3. The right sidebar shows the 'Chlorine REPL' output:

```
<> #'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 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.clj | Chlorine REPL
(ns myns)

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

(def soma-5 (somador 5))

(soma-5 100)
105
```

