

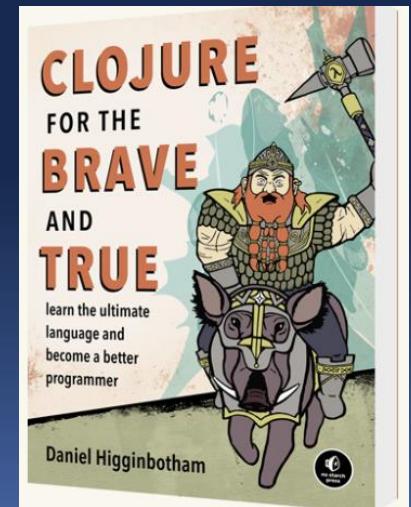
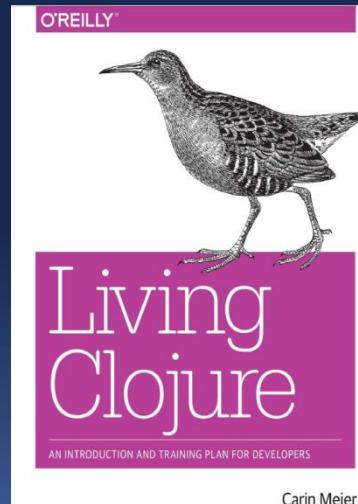
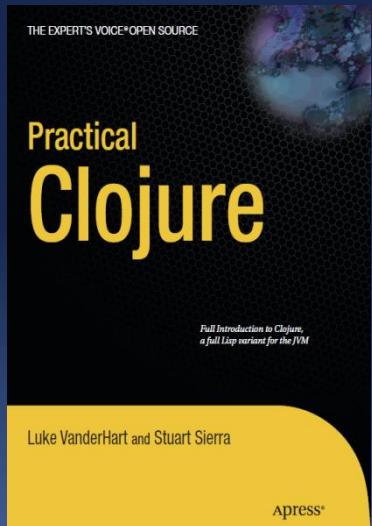
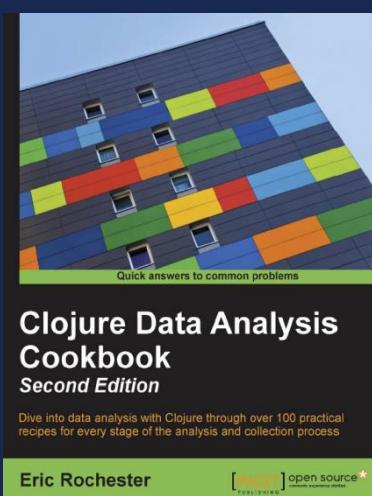
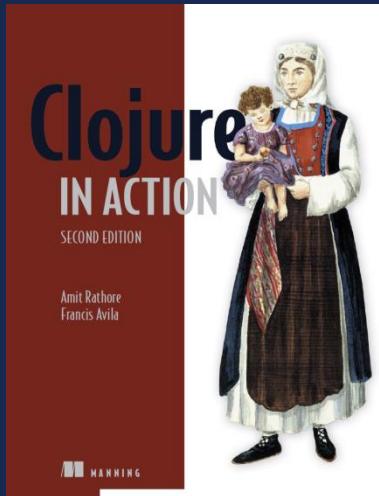
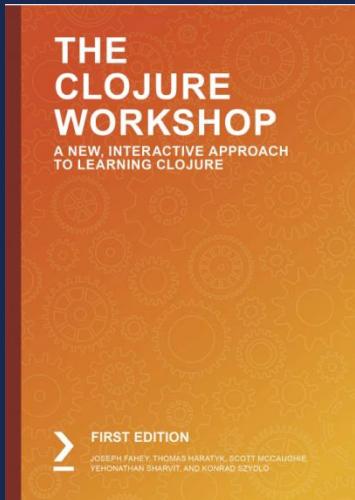
Programação Funcional

Unidade 4 – Visão geral da Linguagem Clojure



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

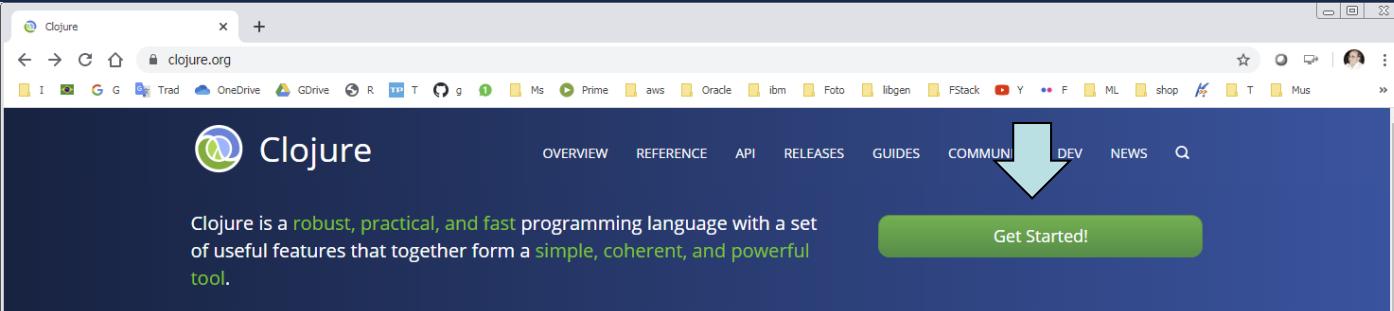
Bibliografia



Antes de iniciar

- ✓ Antes de iniciar a instalação do **Clojure**, esteja certo de que **JDK** está instalado;
- ✓ Clojure é implementado em **Java** e roda na **JVM** (Java Virtual Machine);
- ✓ Clojure é desenhado para ser uma **hosted language**, enquanto que uma outra implementação, chamada **ClojureScript**, roda em qualquer runtime **JavaScript**, por exemplo, um web browser ou Node.js;
- ✓ Clojure **não** requer uma versão particular da Máquina Virtual Java;
- ✓ Mas, recomenda-se que a versão **Java 8** seja instalada;
- ✓ Instruções de instalação em <https://clojure.org/>
- ✓ É possível interagir códigos **Clojure** com libraries (**packages**) escritos em Java (**API'S escritas em Java**);

<https://clojure.org/>



The Clojure Programming Language

Clojure is a dynamic, general-purpose programming language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming. Clojure is a compiled language, yet remains completely dynamic – every feature supported by Clojure is supported at runtime. Clojure provides easy access to the Java frameworks, with optional type hints and type inference, to ensure that calls to Java can avoid reflection.

Clojure is a dialect of Lisp, and shares with Lisp the code-as-data philosophy and a powerful macro system. Clojure is predominantly a functional programming language, and features a rich set of immutable, persistent data structures. When mutable state is needed, Clojure offers a software transactional memory system and reactive Agent system that ensure clean, correct, multithreaded designs.

I hope you find Clojure's combination of facilities elegant, powerful, practical and fun to use.

Rich Hickey
author of Clojure and CTO Cognitect

Companies Succeeding with Clojure

"Our Clojure system just handled its first Walmart black Friday and came out without

"Clojure is a functional programming language from top to bottom. This means

"We discussed the existing Clojure community: the maturity of the language

Learn More

Rationale
A brief overview of Clojure and the features it includes

Getting Started
Resources for getting Clojure up and running

Reference
Grand tour of all that Clojure has to offer

Guides
Walkthroughs to help you learn along the way

Community
We have a vibrant, flourishing community. Join us!



Instruções de Instalação

Local build

Download and build Clojure from source (requires Git, Java, and Maven):

```
git clone https://github.com/clojure/clojure.git  
cd clojure  
mvn -Plocal -Dmaven.test.skip=true package
```

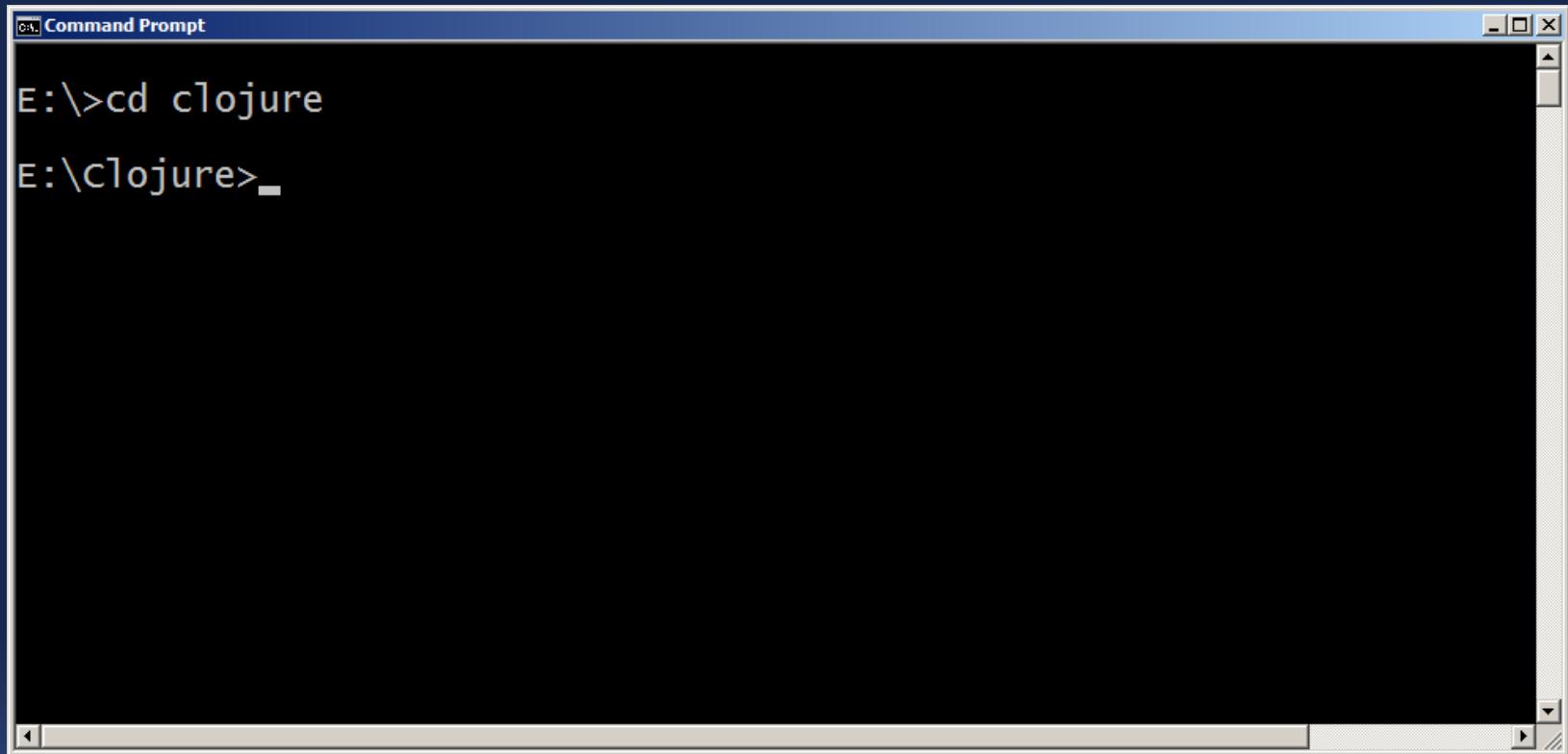
Then start the REPL with the local jar:

```
java -jar clojure.jar
```

Try Clojure online

repl.it provides a browser-based Clojure repl for interactive exploration.

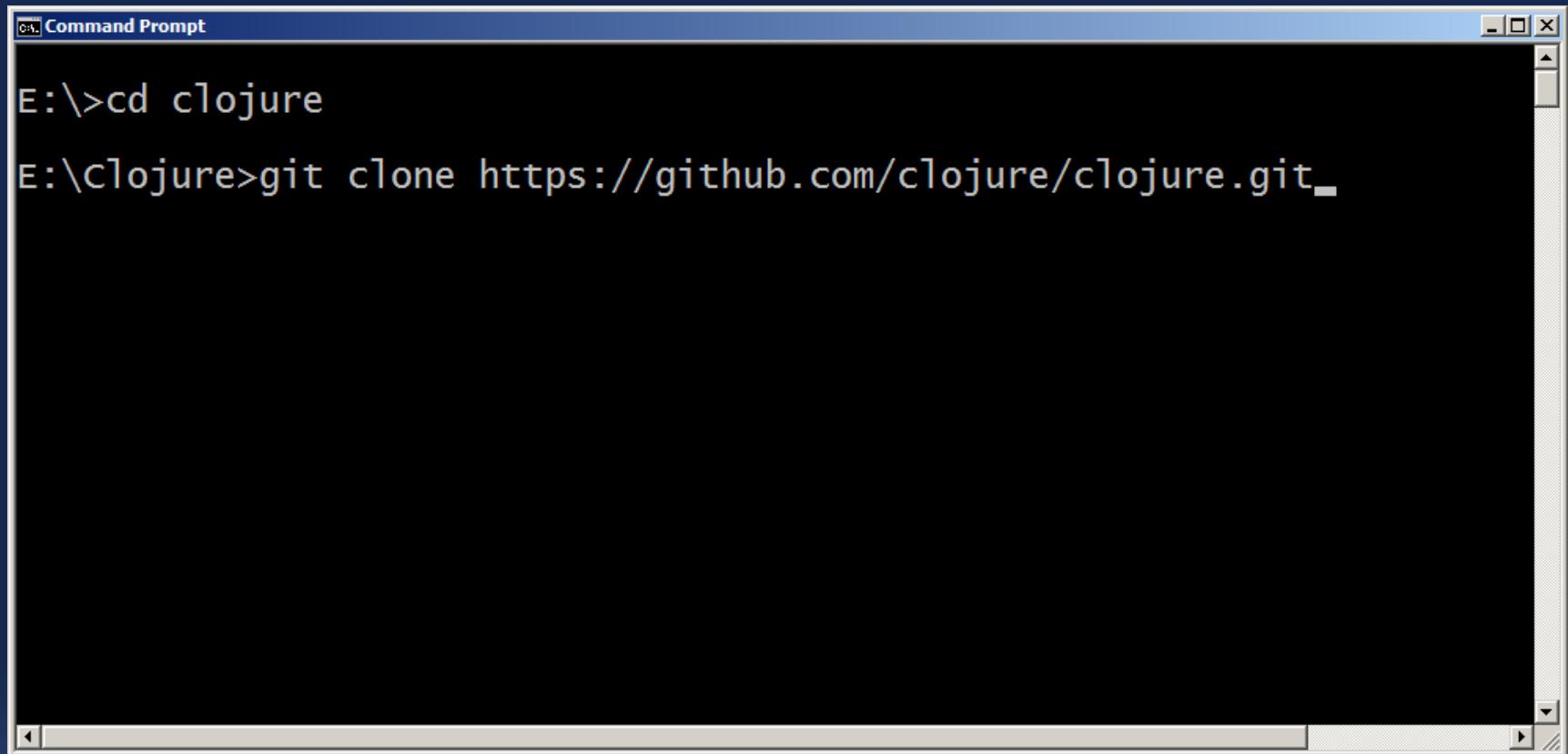
Criar pasta E:\closure



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window has a blue header bar with the title and standard window controls. The main area is black with white text. The user has typed the command "E:\>cd closure" and is now at the prompt "E:\closure>". The window is set against a dark blue background.

```
E:\>cd closure
E:\closure>
```

Baixando clojure



The image shows a screenshot of a Windows Command Prompt window titled "Command Prompt". The window has a blue title bar and a black background. Inside, there are two lines of text in white font:
E:\>cd clojure
E:\Clojure>git clone https://github.com/clojure/clojure.git

Baixando clojure

```
C:\ Command Prompt  
E:\>cd clojure  
  
E:\clojure>git clone https://github.com/clojure/clojure.git  
Cloning into 'clojure'...  
remote: Enumerating objects: 41, done.  
remote: Counting objects: 100% (41/41), done.  
remote: Compressing objects: 100% (28/28), done.  
remote: Total 32714 (delta 11), reused 29 (delta 9), pack-reused 3267  
Receiving objects: 100% (32714/32714), 14.92 MiB | 7.40 MiB/s, done.  
Resolving deltas: 100% (19735/19735), done.  
  
E:\clojure>_
```

Pasta Clojure

Name	Date modified	Type	Size
.git	20-May-20 2:00 PM	File folder	
.idea	20-May-20 2:00 PM	File folder	
doc	20-May-20 2:00 PM	File folder	
src	20-May-20 2:00 PM	File folder	
test	20-May-20 2:00 PM	File folder	
	20-May-20 2:00 PM	Text Document	1 KB
antsetup	20-May-20 2:00 PM	Shell Script	1 KB
build	20-May-20 2:00 PM	XML Document	9 KB
changes	20-May-20 2:00 PM	Markdown Source File	115 KB
clojure.iml	20-May-20 2:00 PM	IML File	2 KB
CONTRIBUTING	20-May-20 2:00 PM	Markdown Source File	1 KB
epl-v10	20-May-20 2:00 PM	Chrome HTML Docu...	13 KB
pom	20-May-20 2:00 PM	XML Document	11 KB
readme	20-May-20 2:00 PM	Text Document	14 KB

Building Clojure com Maven

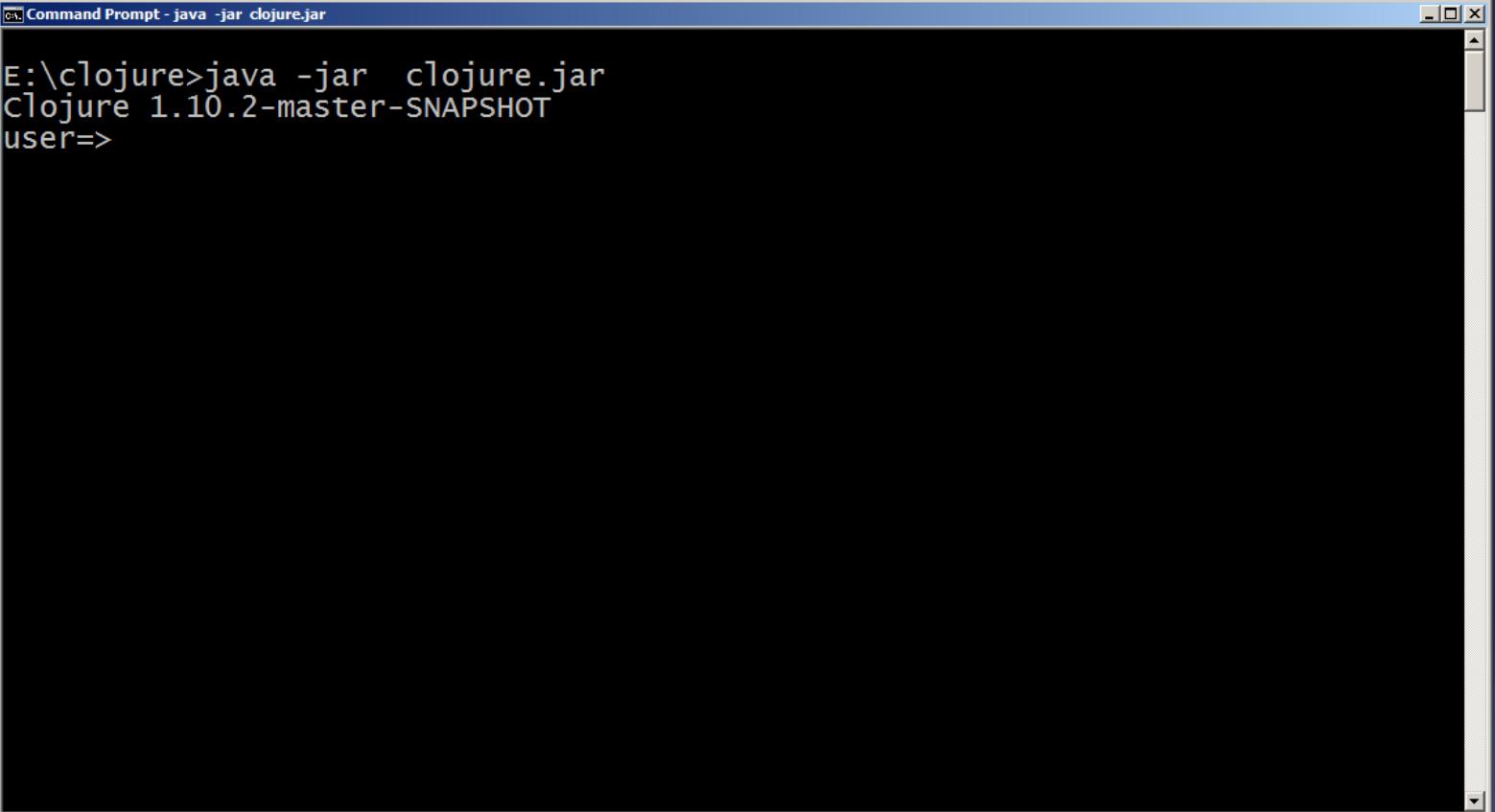
Mvn –Plocal –Dmaven.test.skip=true package

```
Command Prompt
Downloaded from central: https://repo.maven.apache.org/maven2/org/vafer/jdependency/1.2/jdependency-1.2.jar (22 kB at 20 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/com/google/guava/guava/19.0/guava-19.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/ow2/asm/asm-analysis/6.0_BETA/asm-analysis-6.0_BETA.jar (21 kB at 16 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/jdom/jdom/1.1.3/jdom-1.1.3.jar (151 kB at 117 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/eclipse/aether/aether-util/0.9.0.M2/aether-util-0.9.0.M2.jar (134 kB at 100 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/ow2/asm/asm-util/6.0_BETA/asm-util-6.0_BETA.jar (47 kB at 35 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/com/google/guava/guava/19.0/guava-19.0.jar (2.3 MB at 391 kB/s)
[INFO] Including org.clojure:spec.alpha:jar:0.2.187 in the shaded jar.
[INFO] Including org.clojure:core.specs.alpha:jar:0.2.44 in the shaded jar.
[INFO] Including org.clojure:test.check:jar:0.9.0 in the shaded jar.
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 02:28 min
[INFO] Finished at: 2020-05-20T14:12:28-03:00
[INFO] -----
```

E:\clojure>

Iniciando REPL

java -jar clojure.jar



The screenshot shows a Windows Command Prompt window titled "Command Prompt - java -jar clojure.jar". The command entered is "E:\clojure>java -jar clojure.jar". The output shows "Clojure 1.10.2-master-SNAPSHOT" followed by the Clojure user prompt "user=>". The window has a standard Windows title bar and scroll bars.

```
E:\clojure>java -jar clojure.jar
Clojure 1.10.2-master-SNAPSHOT
user=>
```

Testando REPL

java -jar clojure.jar



```
Command Prompt - java -jar clojure.jar
E:\clojure>java -jar clojure.jar
Clojure 1.10.2-master-SNAPSHOT
user=> (+ 2 3 )
5
user=>
```

Clojure installer e CLI tools

Clojure installer e CLI tools

Clojure installer and CLI tools

Clojure provides [command line tools](#) that can be used to start a Clojure repl, use Clojure and Java libraries, and start Clojure programs. See the [changelog](#) for version information.

After following these installation instructions, you should be able to use the `clj` or `clojure` command to start a Clojure repl.

Installation on Windows

An early release version of clj on Windows is available at [clj on Windows](#). Please provide feedback at <https://clojure.atlassian.net/projects/TDEPS>.

clj on Windows

clj on Windows

Alex Miller edited this page 5 days ago · 37 revisions

Currently, `clj` on Windows is in an alpha state. Please try it and provide feedback in the [TDEPS jira](#) or on [#clj-on-windows](#) room on [Clojurians slack](#).

Install

Make sure [PowerShell 5](#) (or later, include PowerShell Core) and [.NET Core SDK 2.1+](#) or [.NET Framework 4.5+](#) (or later) are installed. Then run:

```
Invoke-Expression (New-Object System.Net.WebClient).DownloadString('https://download.clojure.org/install/win-install-1.10.1.561.ps1')
```

clj on Windows

Alternatively, download the latest version of the installer and run the downloaded copy:

- 
- <https://download.clojure.org/install/win-install-1.10.1.561.ps1>

When you run the installer, you will be prompted with several possible install locations:

```
PS Y:\Downloads> .\win-install-1.10.1.561.ps1
Downloading Clojure tools
WARNING: Clojure will install as a module in your PowerShell module path.

Possible install locations:
1) \\Drive\Home\Documents\WindowsPowerShell\Modules
2) C:\Program Files\WindowsPowerShell\Modules
3) C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\
Enter number of preferred install location: 1

Cleaning up existing install
Installing PowerShell module
Removing download
Clojure now installed. Use "clj -h" for help.
```

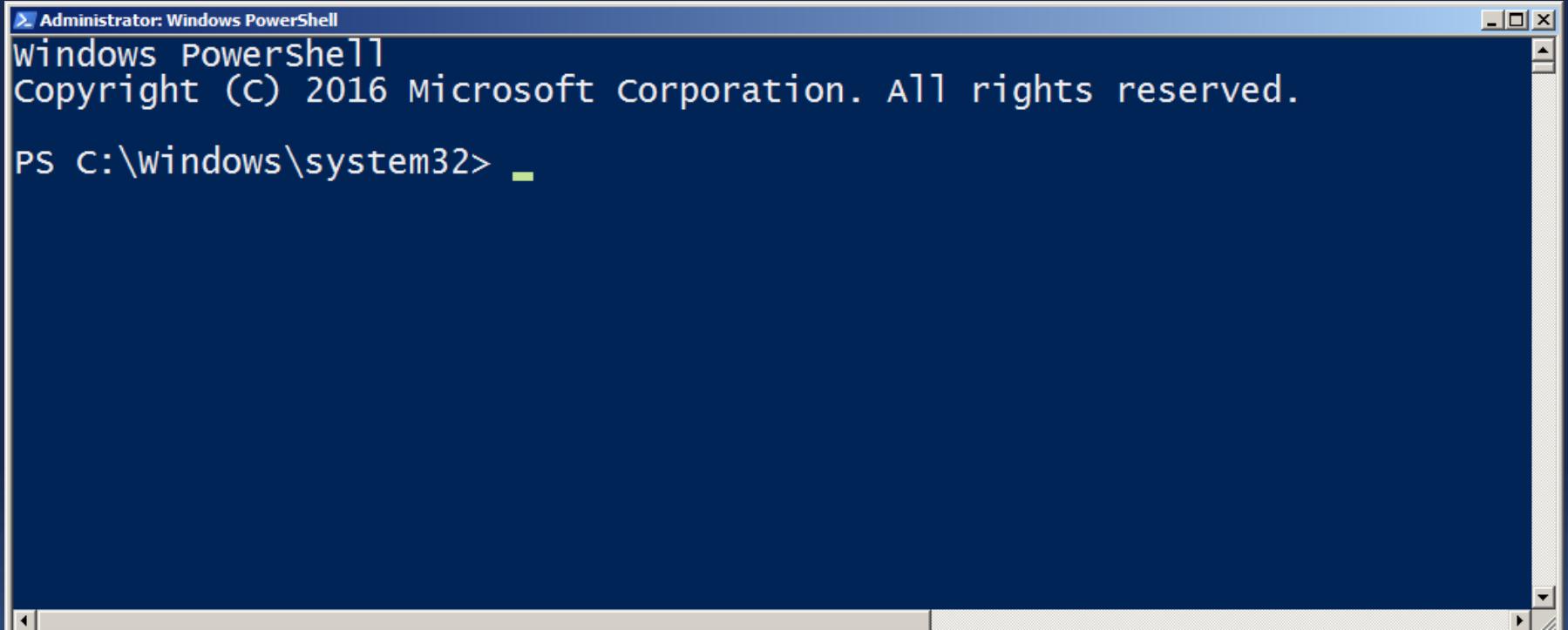
When choosing which location to install consider these tradeoffs:

- #1 can be installed without admin privileges but will create a directory in Documents
- #2 and 3 should probably be run only if you have admin privileges



clj on Windows

- ✓ A instalação será feita com Powershell versão 5
- ✓ Esteja certo de que **PowerShell 5** ou superior esteja instalado em sua máquina;



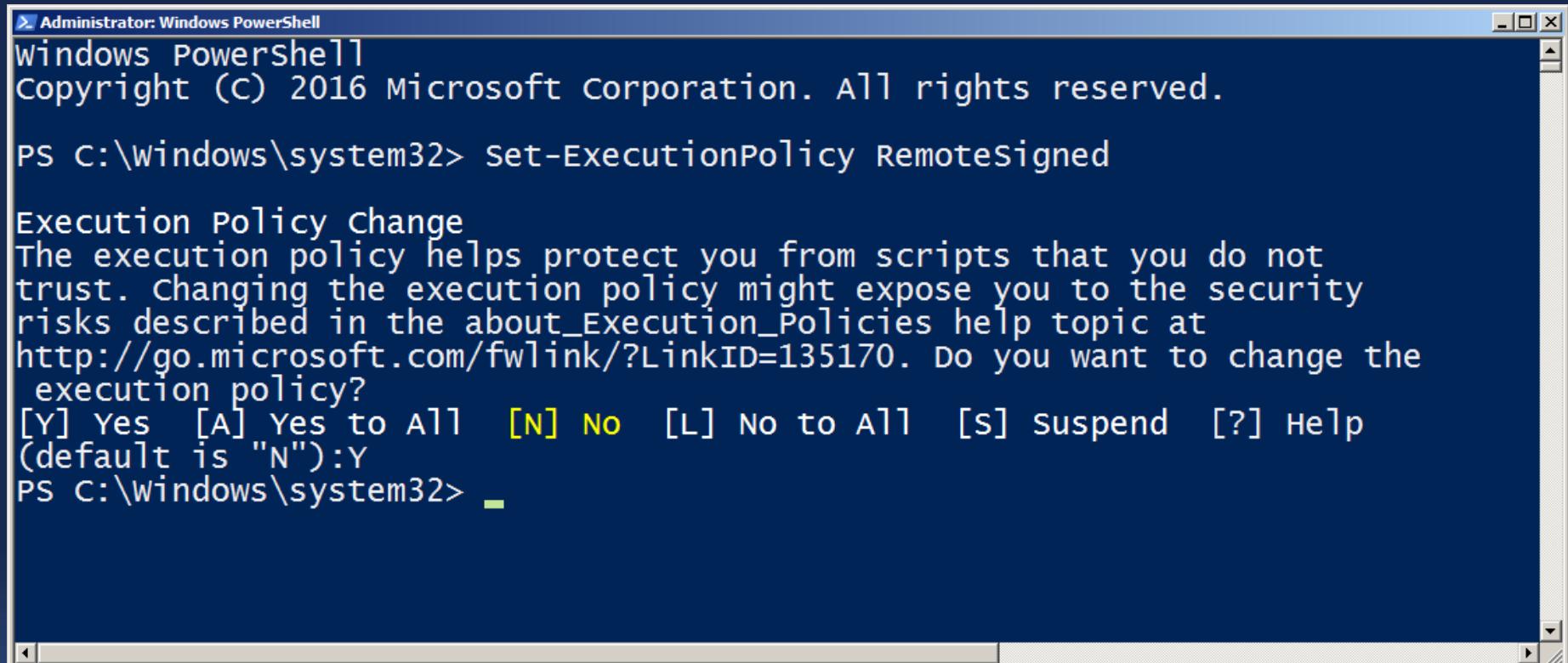
A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The title bar also includes the text "Windows PowerShell" and "Copyright (c) 2016 Microsoft Corporation. All rights reserved.". The command prompt shows "PS C:\windows\system32>". The window has a standard Windows border and title bar.

Configurações de segurança - Powershell



Habilitar PowerShell para execução

- ✓ No prompt do PowerShell: **Set-ExecutionPolicy RemoteSigned**



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The title bar also includes the text "Windows Powershell" and "Copyright (c) 2016 Microsoft Corporation. All rights reserved.". The command "PS C:\windows\system32> Set-ExecutionPolicy RemoteSigned" is entered at the prompt. A confirmation message follows:

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose you to the security risks described in the about_Execution_Policies help topic at <http://go.microsoft.com/fwlink/?LinkID=135170>. Do you want to change the execution policy?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "N"):Y

PS C:\windows\system32> █

Habilitar PowerShell para execução

- ✓ No prompt do PowerShell: (em modo administrador)

`Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass`

Fix for PowerShell Script Not Digitally Signed

When you run a .ps1 PowerShell script you might get the message saying
“.ps1 is not digitally signed. The script will not execute on the system.”

To fix it you have to run the command below to run Set-ExecutionPolicy and change the Execution Policy setting.

```
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
```

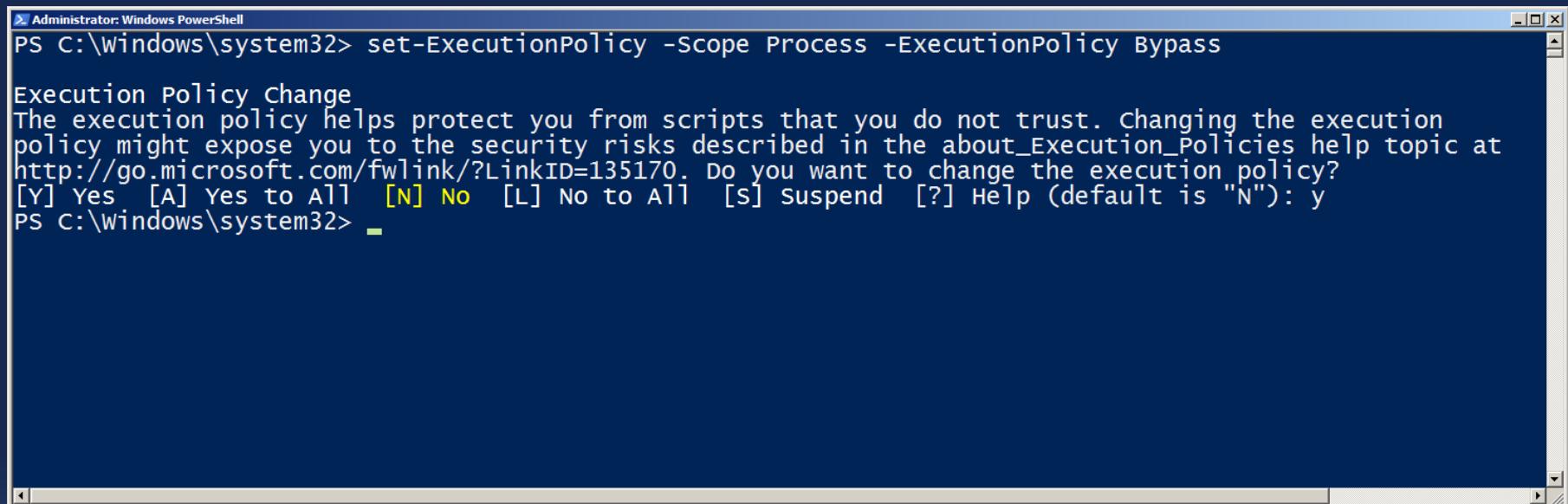
This command sets the execution policy to bypass for only the current PowerShell session after the window is closed, the next PowerShell session will open running with the default execution policy. “Bypass” means nothing is blocked and no warnings, prompts, or messages will be displayed.



Habilitar PowerShell para execução

- ✓ No prompt do PowerShell: (em modo administrador)

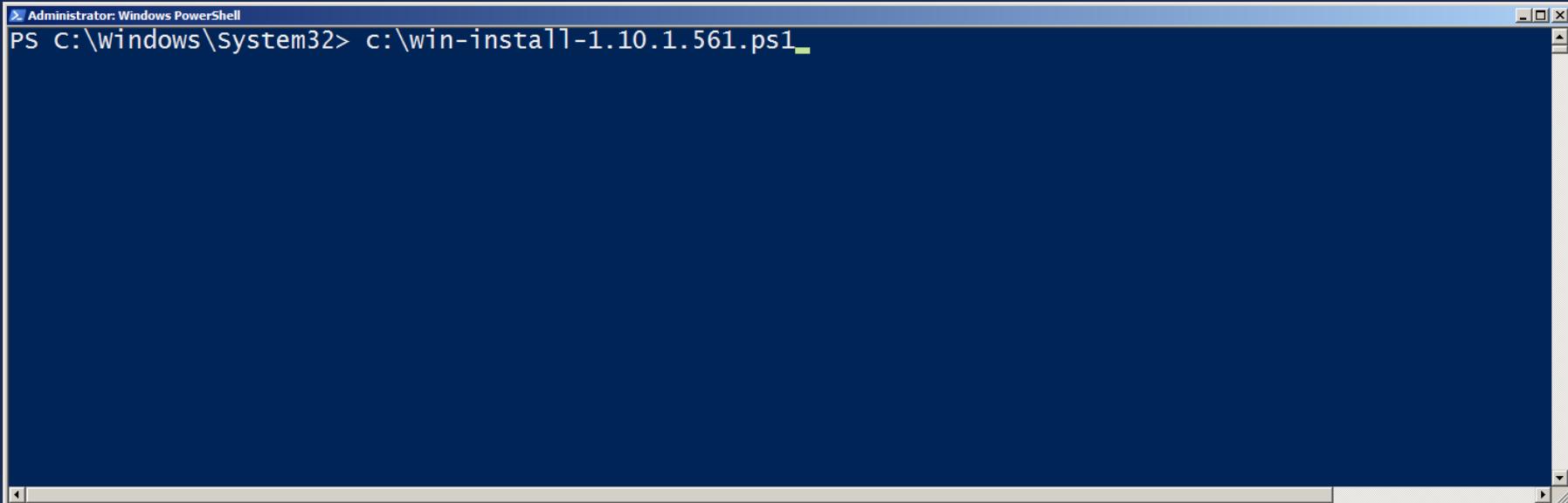
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass



```
Administrator: Windows PowerShell
PS C:\Windows\system32> set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution
policy might expose you to the security risks described in the about_Execution_Policies help topic at
http://go.microsoft.com/fwlink/?LinkId=135170. Do you want to change the execution policy?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): y
PS C:\Windows\system32> _
```

clj on Windows

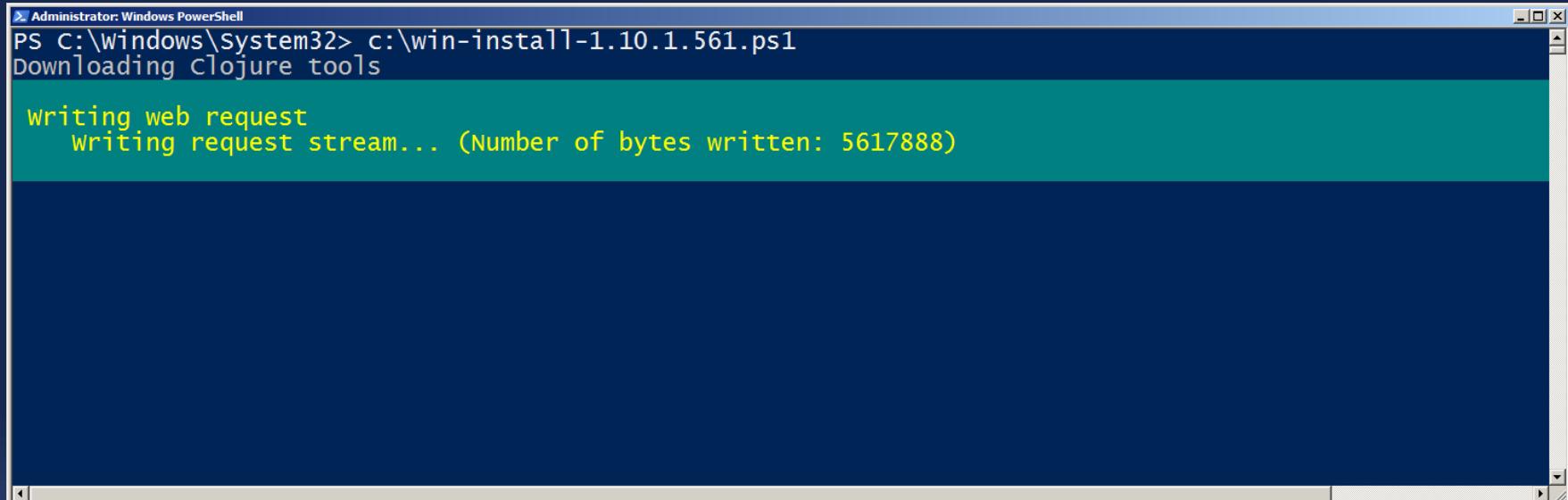
- ✓ Após baixar o instalador do clojure (**win-install-1.10.1.561.ps1**), executá-lo sob o Powershell, em modo administrador.



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The window shows the command "PS C:\Windows\System32> c:\win-install-1.10.1.561.ps1" entered at the prompt. The background of the slide is a dark blue gradient.

clj on Windows

- ✓ Após baixar o instalador do clojure (**win-install-1.10.1.561.ps1**), executá-lo sob o Powershell, em modo administrador.

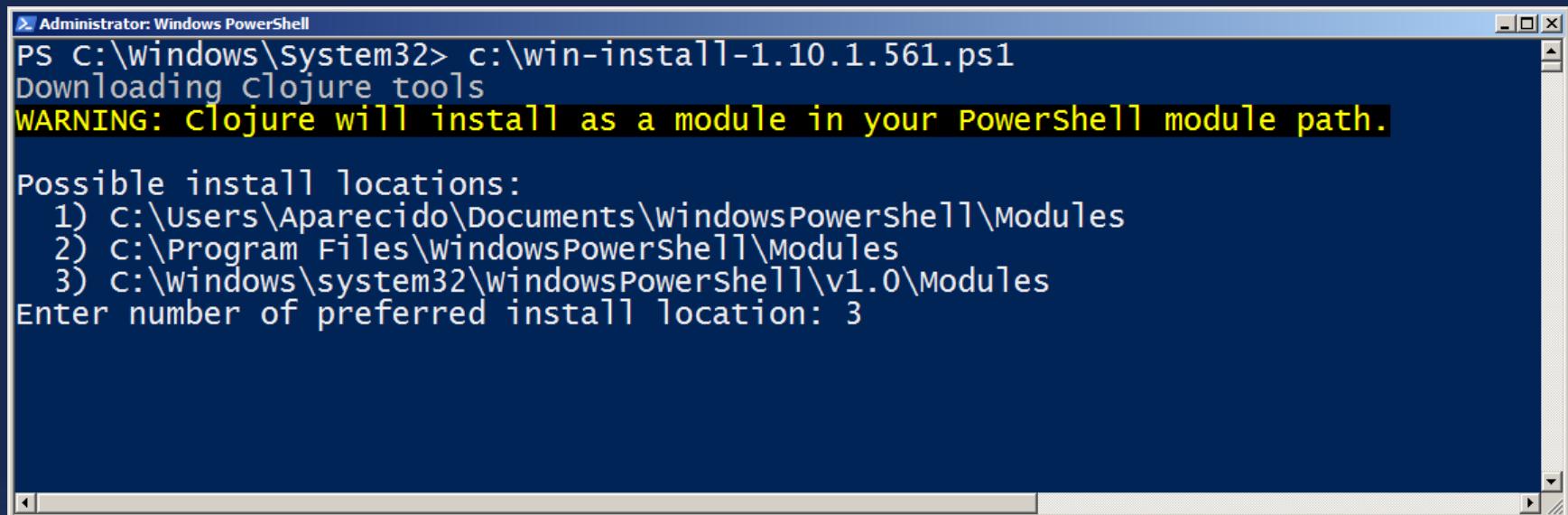


The screenshot shows an Administrator Windows PowerShell window. The command entered is `c:\win-install-1.10.1.561.ps1`. The output shows the script is downloading Clojure tools and then writing a web request and its stream, indicating the progress of the installation.

```
Administrator: Windows PowerShell
PS C:\Windows\System32> c:\win-install-1.10.1.561.ps1
Downloading Clojure tools
Writing web request
Writing request stream... (Number of bytes written: 5617888)
```

clj on Windows

- ✓ Definir opção 3 => c:\Windows\system32\WindowsPowerShell.



```
Administrator: Windows PowerShell
PS C:\Windows\System32> c:\win-install-1.10.1.561.ps1
Downloading Clojure tools
WARNING: Clojure will install as a module in your PowerShell module path.

Possible install locations:
1) C:\Users\Aparecido\Documents\windowsPowershell\Modules
2) C:\Program Files\windowsPowershell\Modules
3) C:\windows\system32\windowsPowershell\v1.0\Modules
Enter number of preferred install location: 3
```

clj on Windows

- ✓ Clojure now installed. Use "clj-h" for help.

```
Administrator: Windows PowerShell
Downloading clojure tools
WARNING: Clojure will install as a module in your PowerShell module path.

Possible install locations:
 1) c:\Users\Aparecido\Documents\WindowsPowerShell\Modules
 2) c:\Program Files\WindowsPowerShell\Modules
 3) c:\windows\system32\WindowsPowerShell\v1.0\Modules
Enter number of preferred install location: 3

Cleaning up existing install
Installing PowerShell module
Removing download
Clojure now installed. use "clj -h" for help.
PS C:\windows\system32>
```



clj on Windows

```
> Administrator: Windows PowerShell
PS C:\Windows\System32> clj
clojure 1.10.1
user=> (println "Hello clojure")
Hello clojure
nil
user=> _
```



Executando clj

- ✓ Sob **BASH**: `powershell -command clj`



```
MINGW64:/c/Users/Aparecido
Aparecido@Aparecido-PC MINGW64 ~
$ (powershell -command clj)
clojure 1.10.1
user=> |
```

Ativando cljs

- ✓ Sob prompt do Windows: **powershell -command cljs**



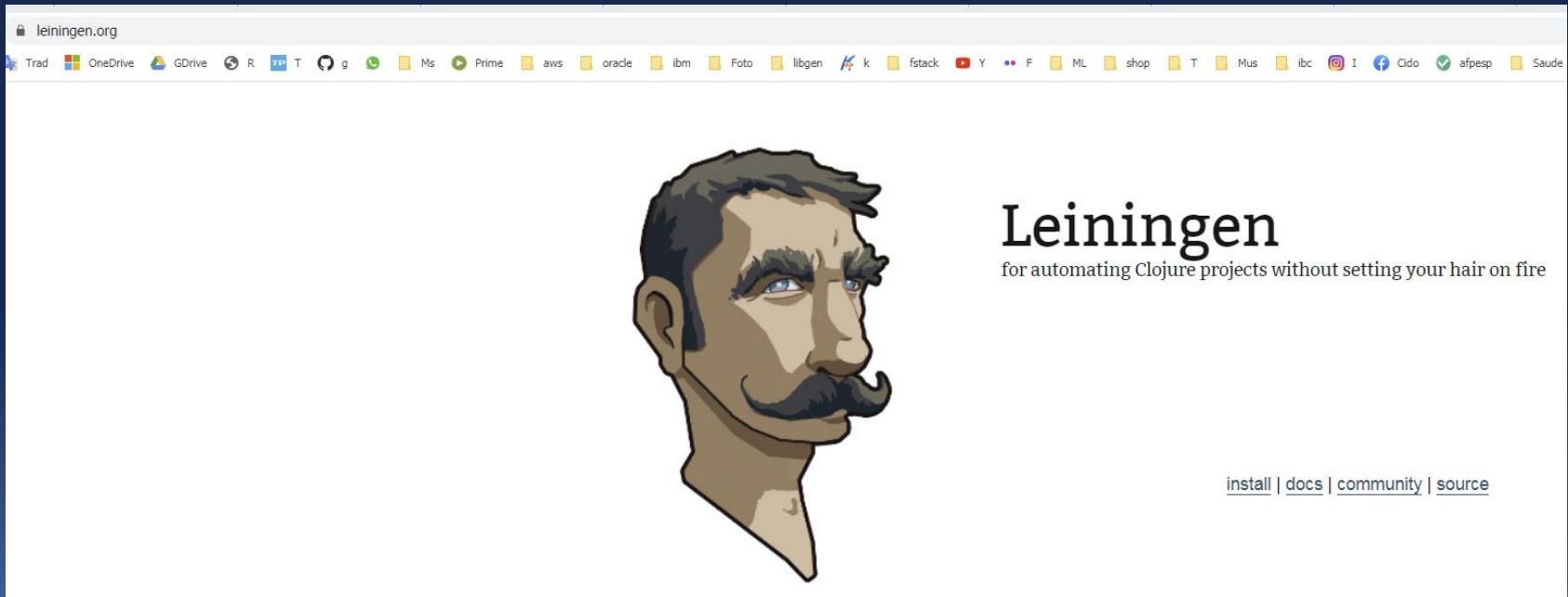
```
C:\Windows\system32\cmd.exe - powershell -command cljs
C:\>powershell -command cljs
clojure 1.10.1
user=>
```

Instalação com Leiningen



Leiningen – www.leiningen.org

- ✓ Leiningen é a forma mais simples de se usar a Linguagem Clojure;
- ✓ Ferramenta para se trabalhar com projetos Clojure;
- ✓ Foco na automação do projeto e configuração declarativa, permite que se foque na criação do código;



Leiningen – Install

Leiningen and Clojure require Java. OpenJDK version 8 is recommended at this time.

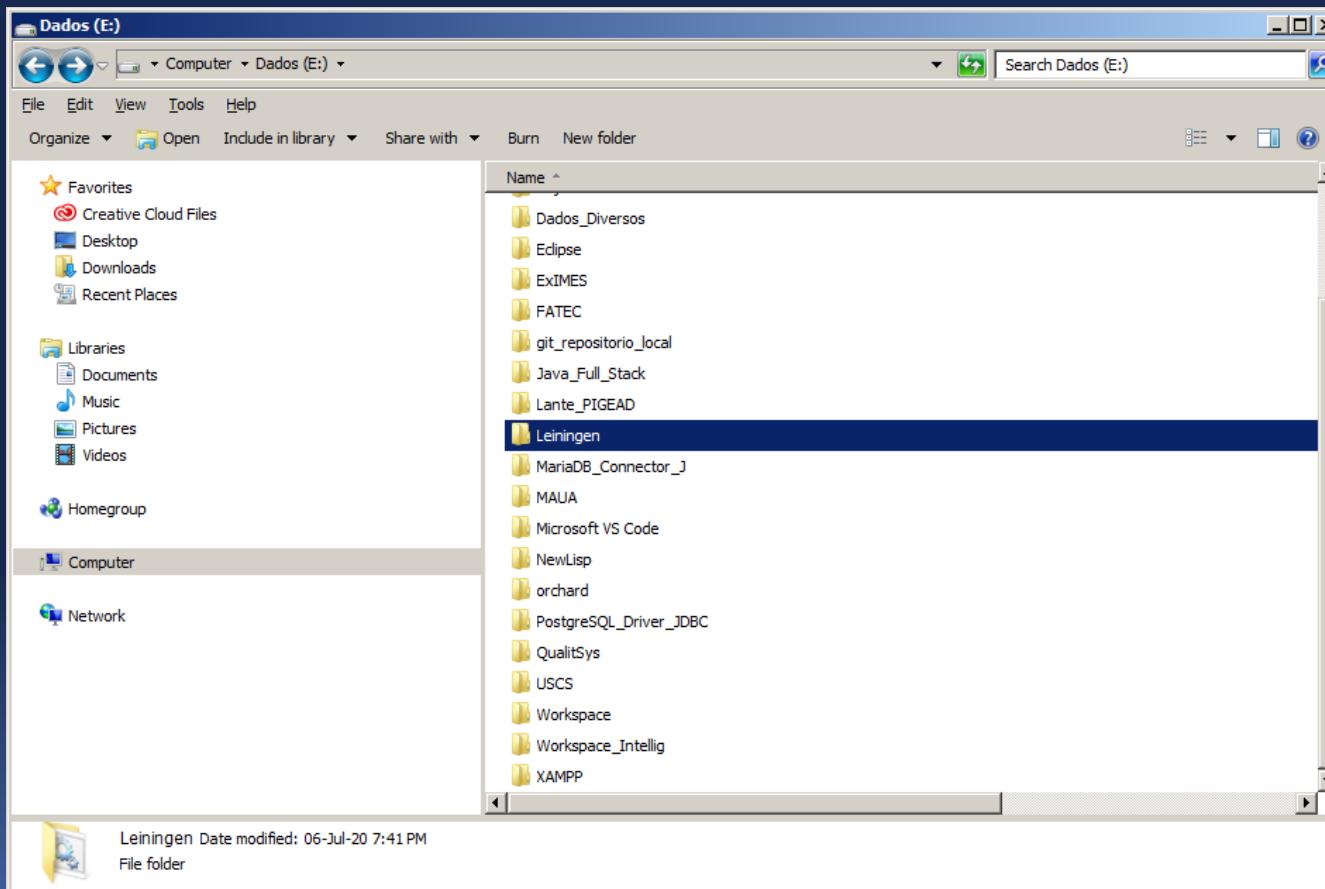
Install

1. Download the [lein script](#) (or on Windows [lein.bat](#))
2. Place it on your \$PATH where your shell can find it (eg. ~/bin)
3. Set it to be executable (`chmod a+x ~/bin/lein`)
4. Run it (`lein`) and it will download the self-install package

You can check your [package manager](#) as well..

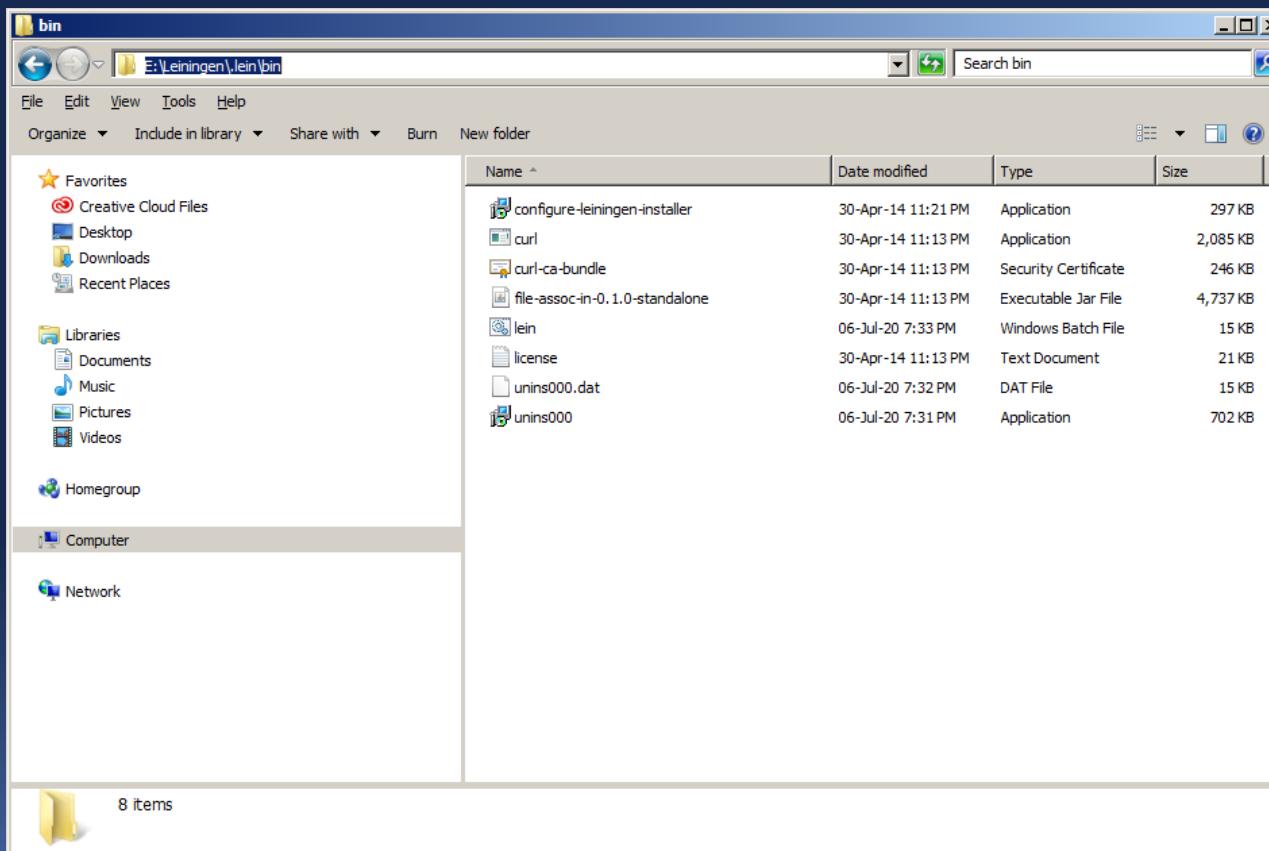
Leiningen - Install

- ✓ Criar pasta para Leiningen, download e execução do instalador



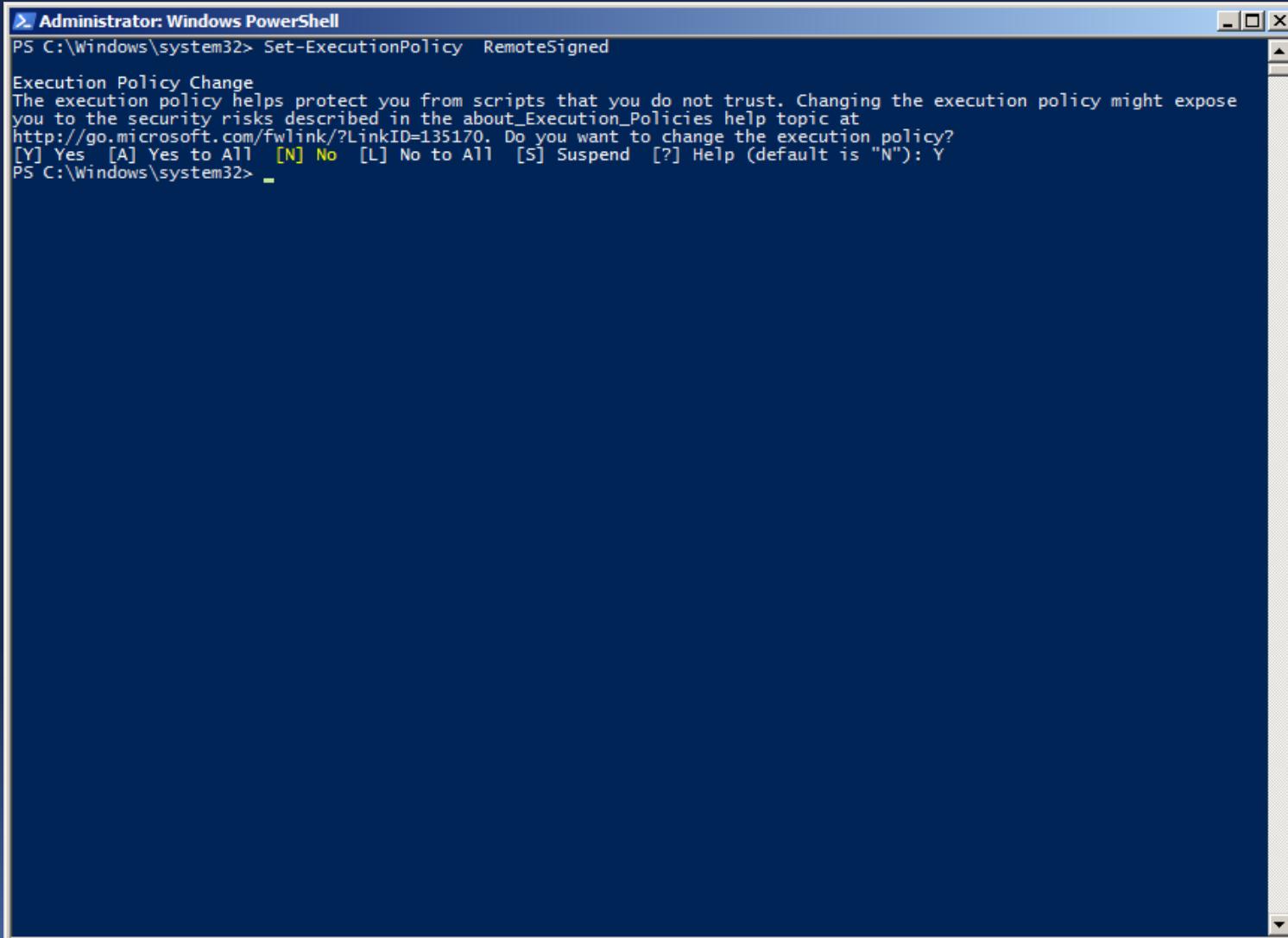
Leiningen - Install

- ✓ Criar pasta para **Leiningen**, download e execução do instalador;
- ✓ Executar o bat file: **lein.bat** (sob Powershell)



Habilitar PowerShell para execução

- ✓ No prompt do PowerShell: **Set-ExecutionPolicy RemoteSigned**



The screenshot shows an Administrator Windows PowerShell window titled "Administrator: Windows PowerShell". The command "PS C:\Windows\system32> Set-ExecutionPolicy RemoteSigned" is entered. A confirmation message about the execution policy change is displayed, stating: "Execution Policy Change. The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose you to the security risks described in the about_Execution_Policies help topic at http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy? [Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N")): Y". The user has responded with "Y".

Habilitar PowerShell para execução

- ✓ No prompt do PowerShell:

`Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass`

Fix for PowerShell Script Not Digitally Signed



Caio Moreno [Follow](#)

Jul 15, 2019 · 1 min read



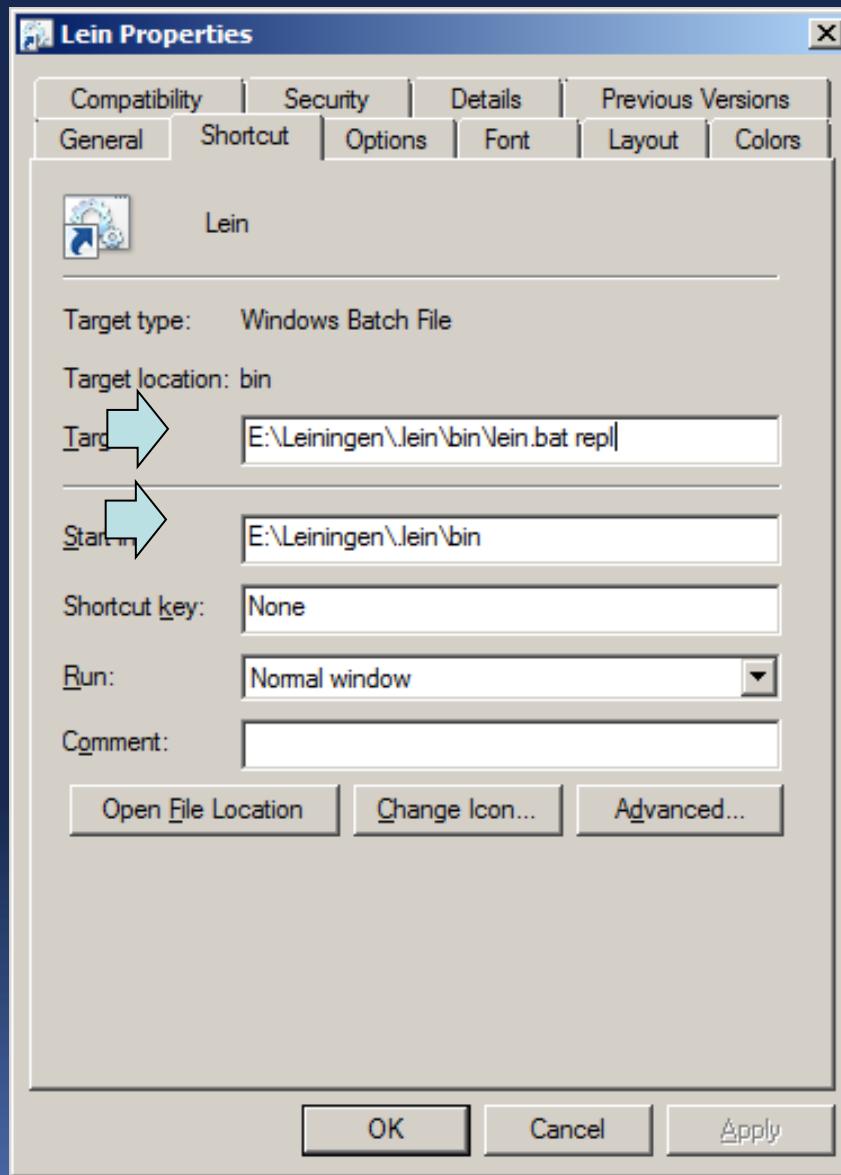
When you run a .ps1 PowerShell script you might get the message saying
“.ps1 is not digitally signed. The script will not execute on the system.”

To fix it you have to run the command below to run Set-ExecutionPolicy and change the Execution Policy setting.

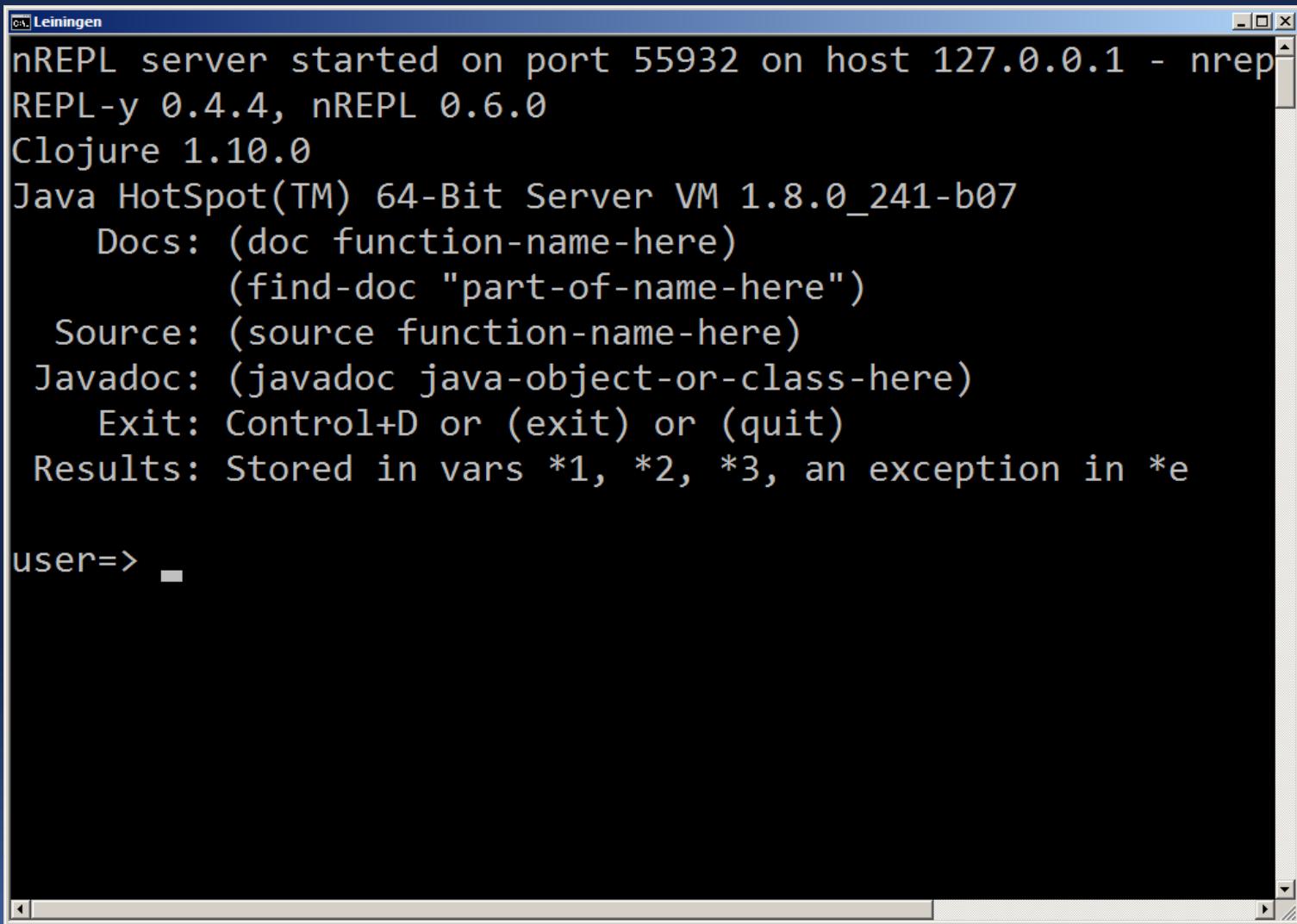
```
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
```

This command sets the execution policy to bypass for only the current PowerShell session after the window is closed, the next PowerShell session will open running with the default execution policy. “Bypass” means nothing is blocked and no warnings, prompts, or messages will be displayed.

Subindo nREPL



Leiningen - nREPL



```
Leiningen
nREPL server started on port 55932 on host 127.0.0.1 - nrepl
REPL-y 0.4.4, nREPL 0.6.0
Clojure 1.10.0
Java HotSpot(TM) 64-Bit Server VM 1.8.0_241-b07
  Docs: (doc function-name-here)
        (find-doc "part-of-name-here")
  Source: (source function-name-here)
Javadoc: (javadoc java-object-or-class-here)
  Exit: Control+D or (exit) or (quit)
Results: Stored in vars *1, *2, *3, an exception in *e

user=> ■
```

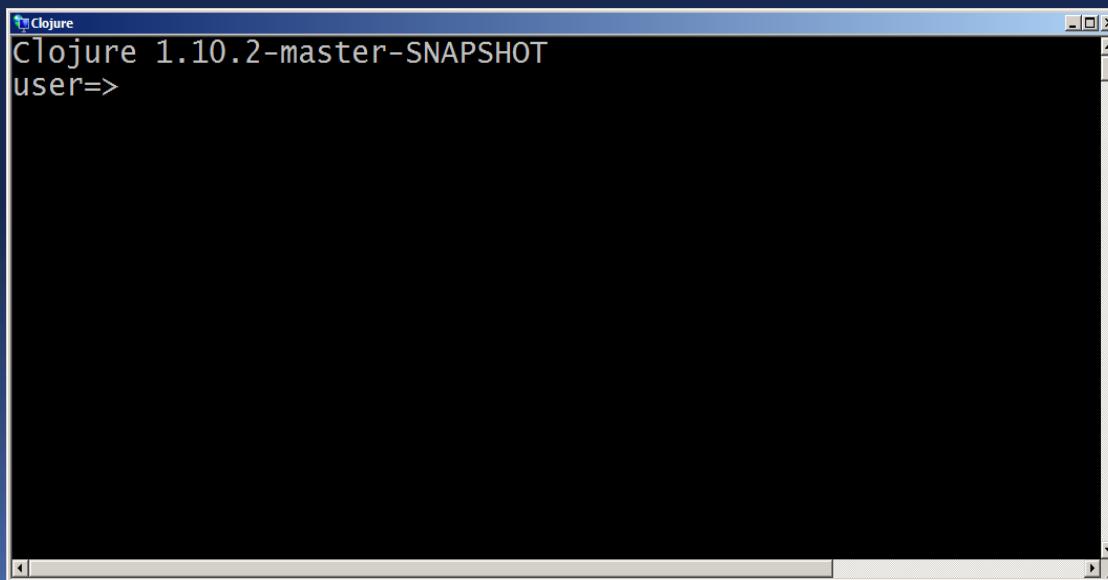
Usando REPL

- ✓ REPL significa **READ EVAL PRINT LOOP** e representa um bom recurso para iniciar o estudo de **Clojure**;
- ✓ É uma **interface de comandos** que permite a avaliação direta de código **Clojure**;
- ✓ No prompt do REPL, a primeira linha representa a **versão** do Clojure, o qual em nosso caso é a **1.10.2**;

A screenshot of a terminal window titled "Clojure". The title bar also shows the text "Clojure 1.10.2-master-SNAPSHOT". The window contains the text "user=>". The window has a standard Windows-style title bar and scroll bars.

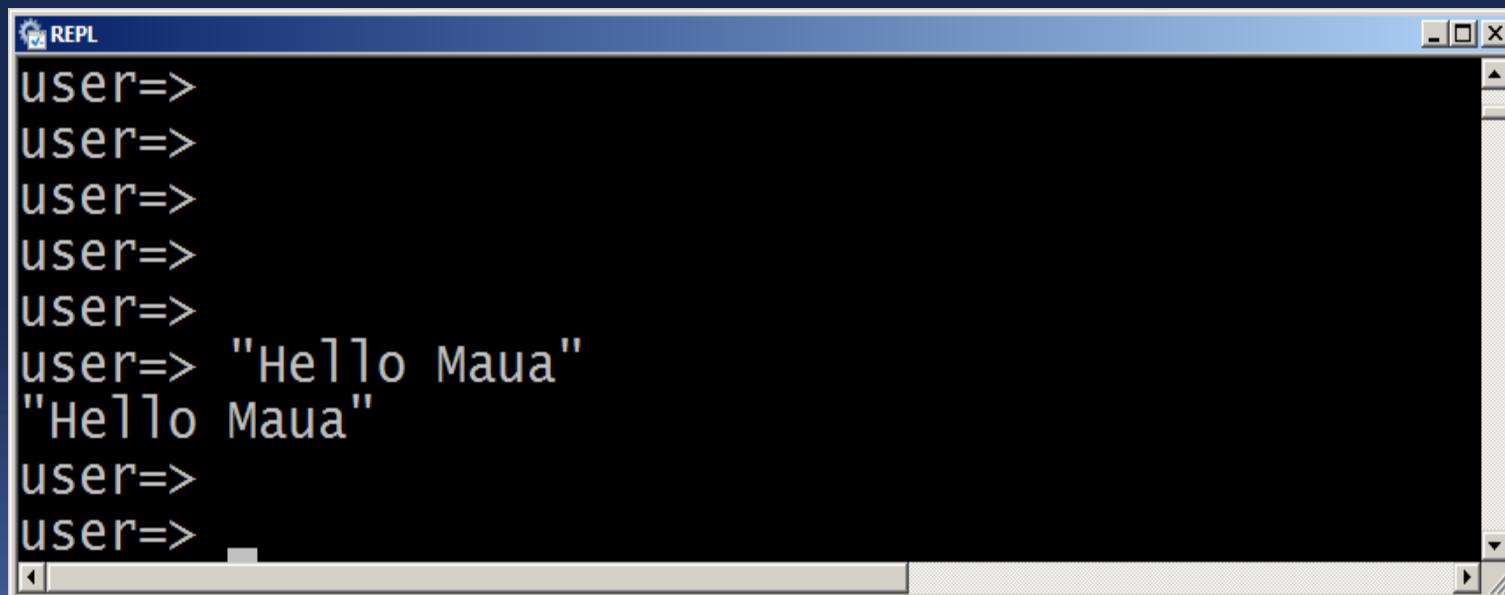
Usando REPL

- ✓ A segunda linha exibe o **namespace** corrente (**user**) e solicita input do usuário;
- ✓ Um **namespace** é um grupo de coisas (tais como funções) que estão agrupadas em um espaço;
- ✓ Aqui, nesse caso, tudo que for criado estará no **namespace user** por default;
- ✓ **REPL** está agora pronto (**ready**).



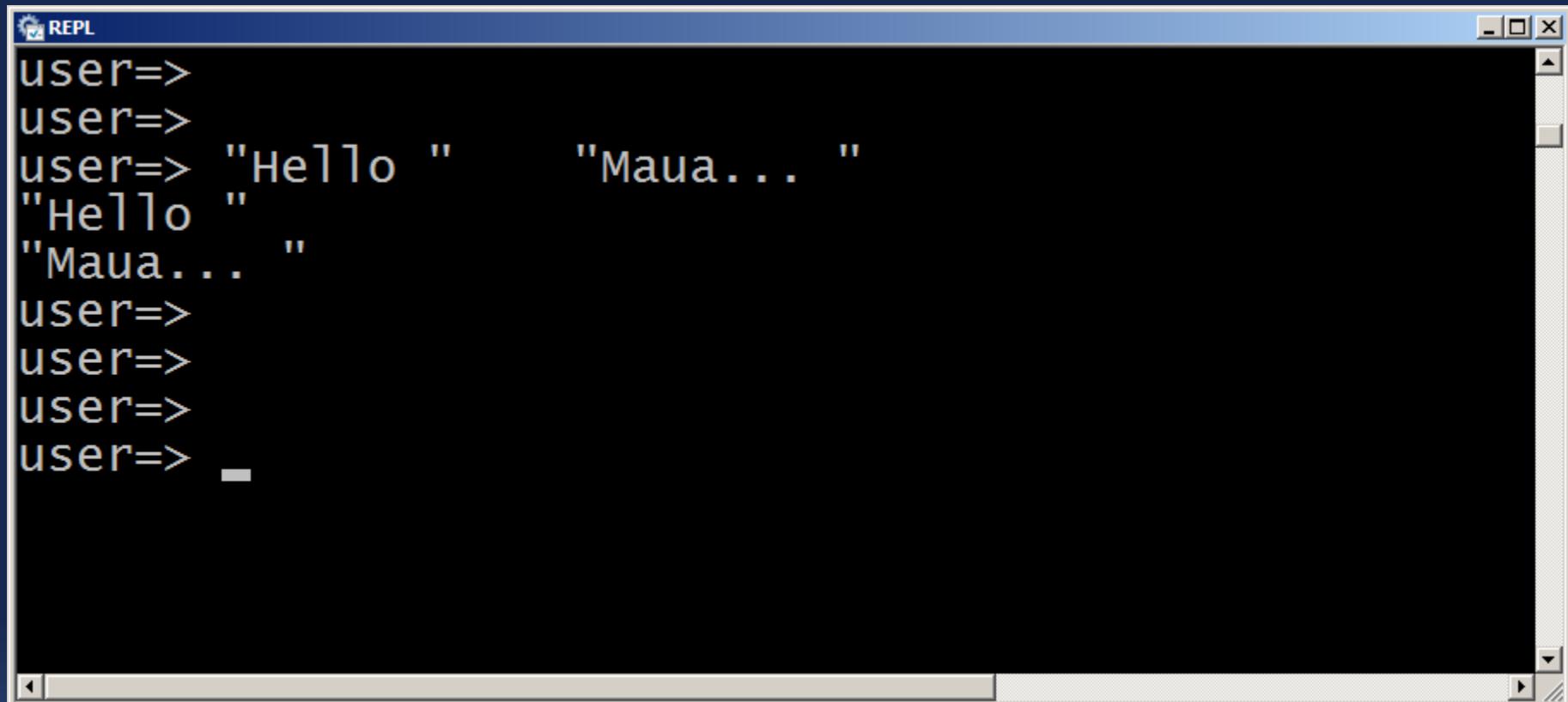
Avaliando expressões

- ✓ Em **Clojure**, **literais string** são criados com aspas duplas, " ";
- ✓ Um **literal** é uma notação para representar valores **fixos** no código fonte.



The screenshot shows a Windows-style application window titled "REPL". The title bar has standard window controls (minimize, maximize, close). The main window is black with white text. It displays the following sequence of text:
user=>
user=>
user=>
user=>
user=>
user=> "Hello Maua"
"Hello Maua"
user=>
user=>

Avaliando múltiplos strings



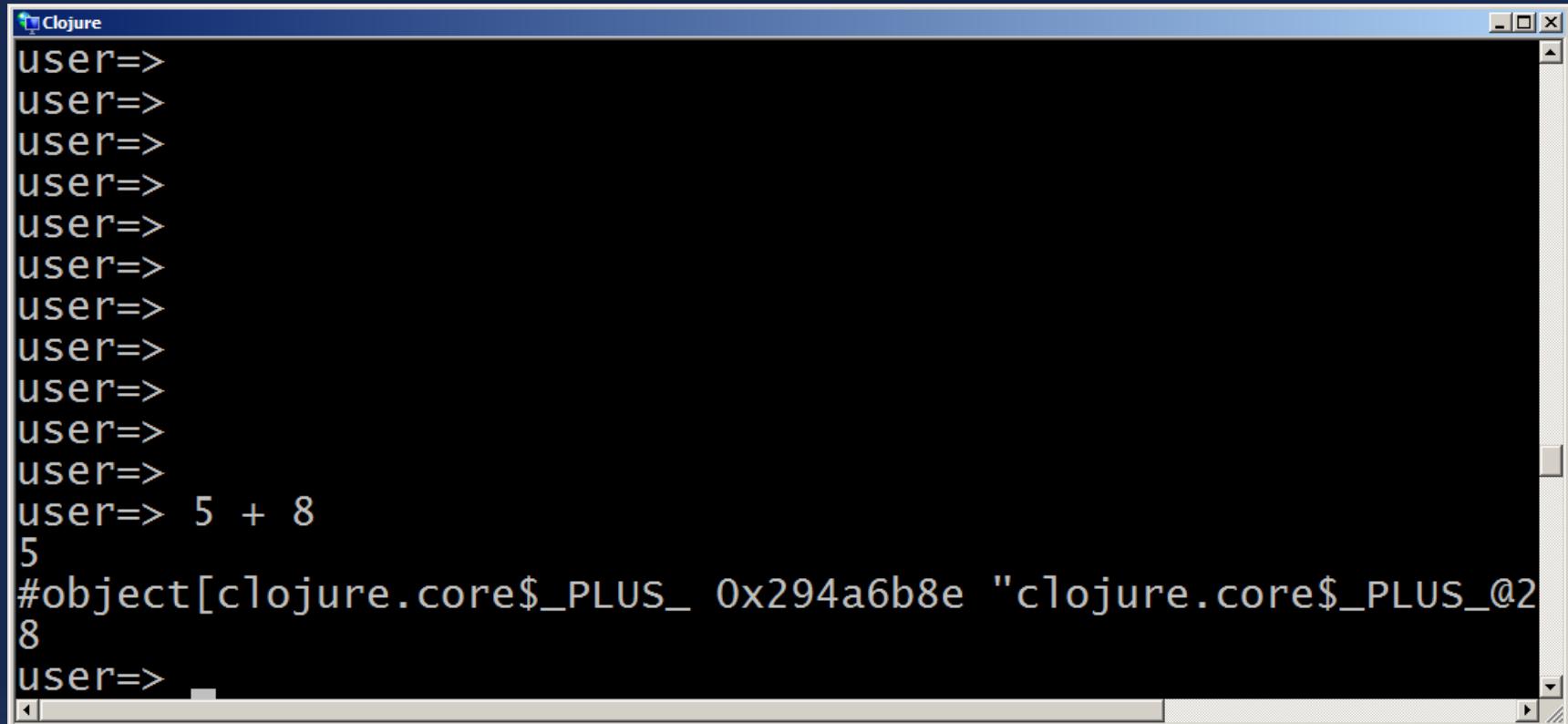
The screenshot shows a Clojure REPL window with the title bar "REPL". The console area displays the following interaction:

```
user=>
user=>
user=> "Hello "      "Maua... "
"Hello "
"Maua... "
user=>
user=>
user=>
user=> _
```

The window has standard operating system window controls (minimize, maximize, close) at the top right and scroll bars on the right and bottom.

- ✓ Nesse exemplo, **duas** expressões **string** foram avaliadas sequencialmente, e cada qual foi **retornada** em duas linhas separadas;

Avaliando expressões aritméticas

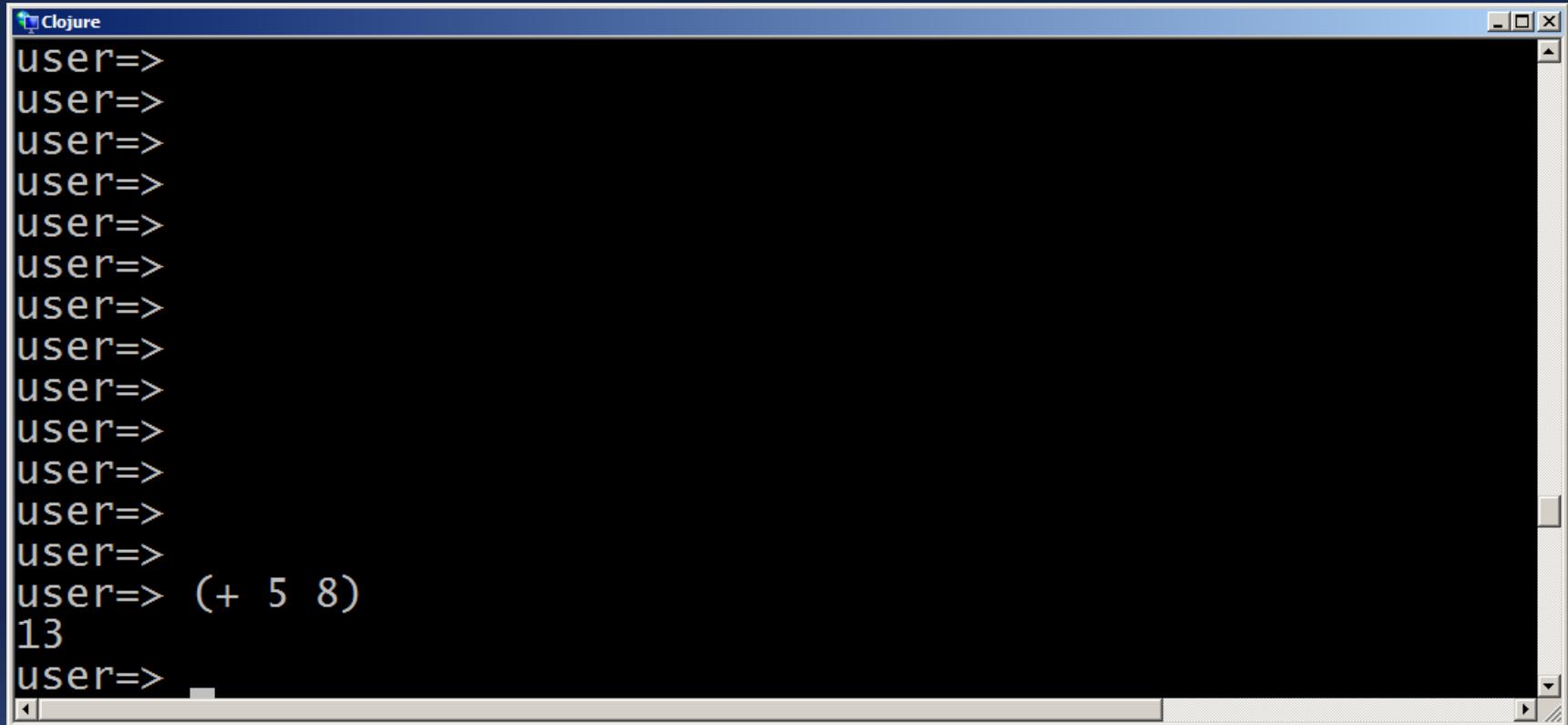


The screenshot shows a Windows-style application window titled "Clojure". Inside, the Clojure REPL is running. The user has typed the expression "5 + 8" and pressed Enter. The REPL has evaluated the first argument "5" and returned it. The second argument "8" is shown as "#object[clojure.core\$PLUS_ 0x294a6b8e "clojure.core\$PLUS_@28"]". The user prompt "user=>" is visible at the bottom.

```
Clojure
user=>
user=> 5 + 8
5
#object[clojure.core$PLUS_ 0x294a6b8e "clojure.core$PLUS_@28"]
user=>
```

- ✓ Clojure retornou **erro**, pois o primeiro argumento deve ser uma **função**. No caso, o símbolo **+** está ligado (**bounded**) à uma função e deve ser o primeiro elemento da lista a ser avaliada, seguido pelos operandos (**argumentos**).

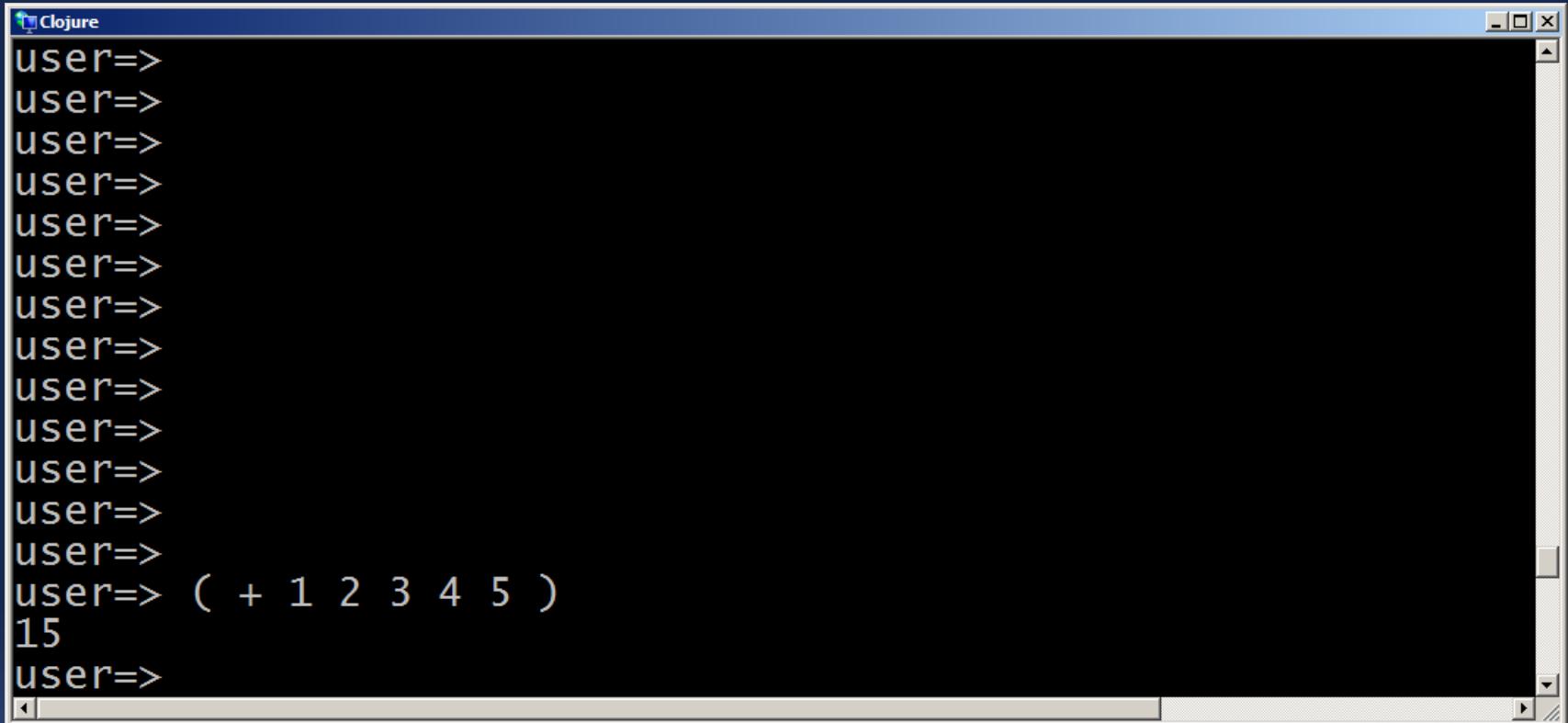
Avaliando expressões aritméticas



The screenshot shows a Windows-style window titled "Clojure". Inside, the Clojure REPL is running. The session starts with several "user=>" prompts, followed by the input of the expression "(+ 5 8)". The REPL then outputs the result "13" and continues with another "user=>". The window has standard operating system window controls (minimize, maximize, close) at the top right.

```
user=>
user=> (+ 5 8)
13
user=>
```

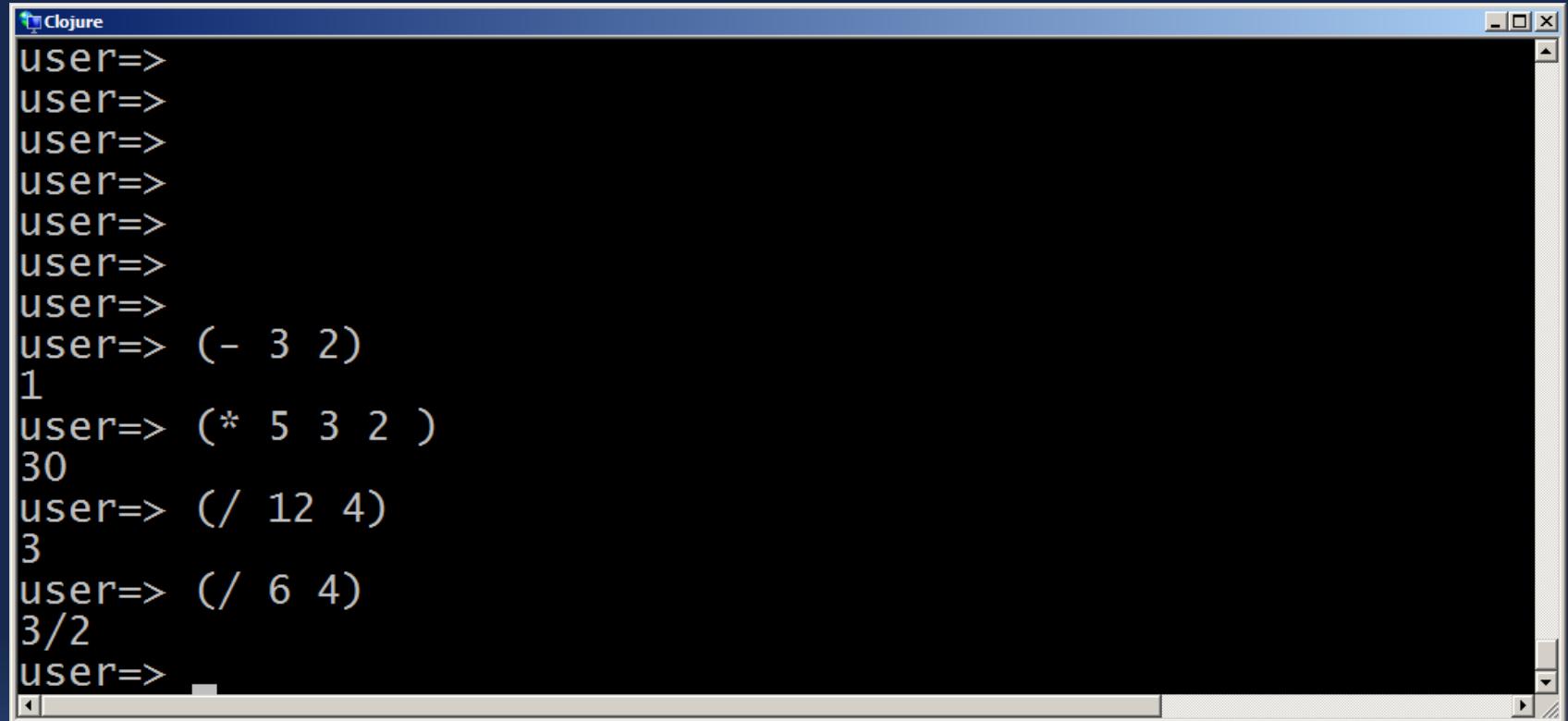
Avaliando expressões aritméticas



The screenshot shows a Windows-style application window titled "Clojure". Inside, the Clojure REPL is running. The user has entered the expression "(+ 1 2 3 4 5)", which is evaluated to the result "15". The REPL prompt "user=>" appears at the end of each line of input or output.

```
Clojure
user=>
user=> ( + 1 2 3 4 5 )
15
user=>
```

Avaliando expressões aritméticas



The screenshot shows a Windows-style application window titled "Clojure". Inside, a Clojure REPL session is running, displaying the following interactions:

```
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (- 3 2)
1
user=> (* 5 3 2 )
30
user=> (/ 12 4)
3
user=> (/ 6 4)
3/2
user=>
```

Função println

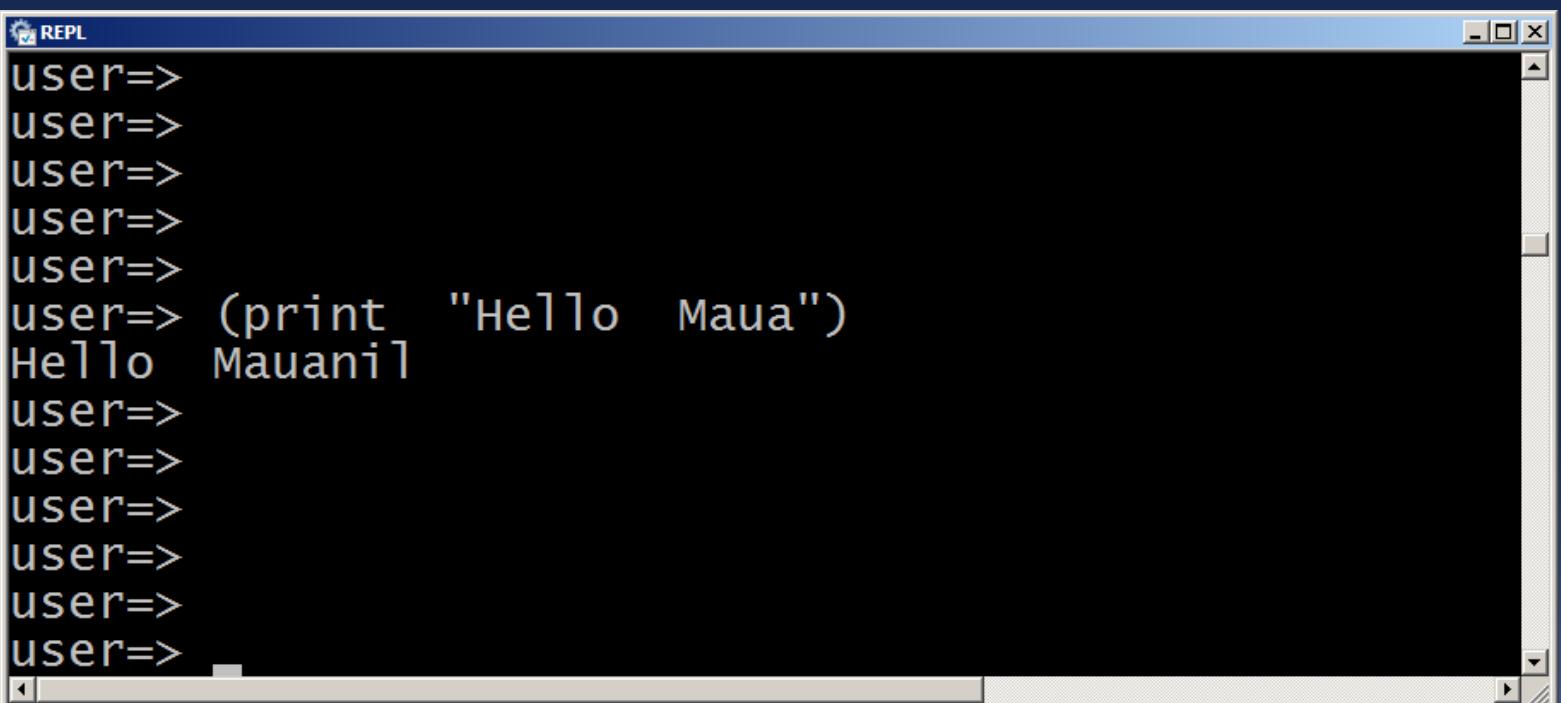


```
REPL
user=>
user=>
user=>
user=>
user=>
user=> (println "Hello Maua")
Hello Maua
nil
user=>
user=>
user=>
user=>
user=>
```

- ✓ O texto impresso pela função `println` foi um **SIDE EFFECT**;
- ✓ A função na verdade **retornou nil**.

nil

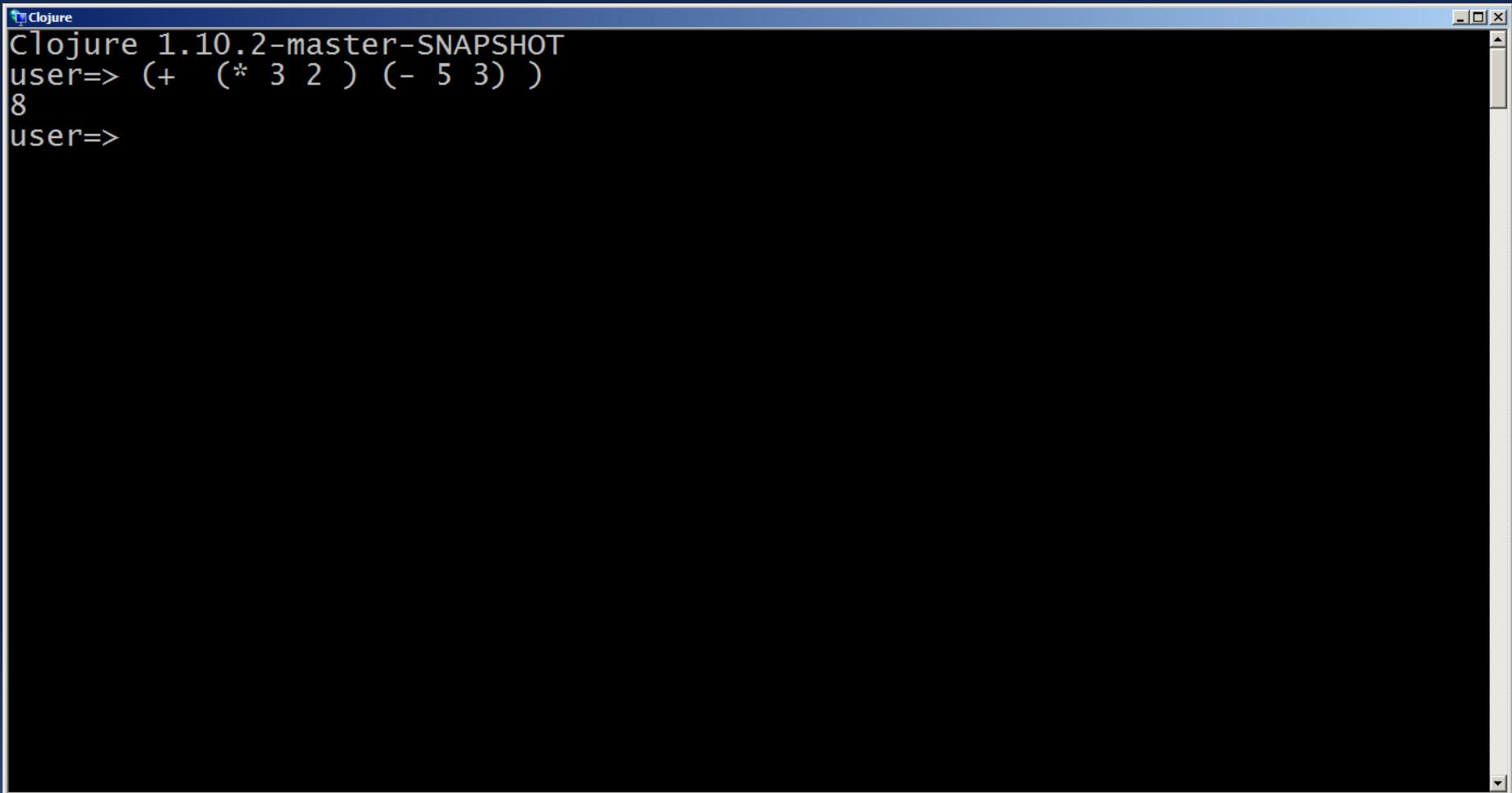
- ✓ Equivale em **Clojure** a um valor “**null**” ou “**nada**”;
- ✓ Ou seja, trata-se da **ausência de significado**;
- ✓ As funções **print** e **println** são usadas para imprimir objetos para a saída padrão e retornam **nil** uma vez que a impressão foi efetivada;



The screenshot shows a Windows-style application window titled "REPL". Inside the window, the Clojure REPL is running. The user has entered several forms, each starting with "user=>". The first five entries are empty lines. The sixth entry is "(print "Hello Maua")". The seventh entry shows the output "Hello Mauanil", with the "l" being a redacted character. A large blue arrow points from the left towards the "(print" line, highlighting it. The window has a standard title bar with icons and a scroll bar on the right side.

```
user=>
user=>
user=>
user=>
user=>
user=> (print "Hello Maua")
Hello Mauanil
user=>
user=>
user=>
user=>
user=>
user=>
```

Funções nested



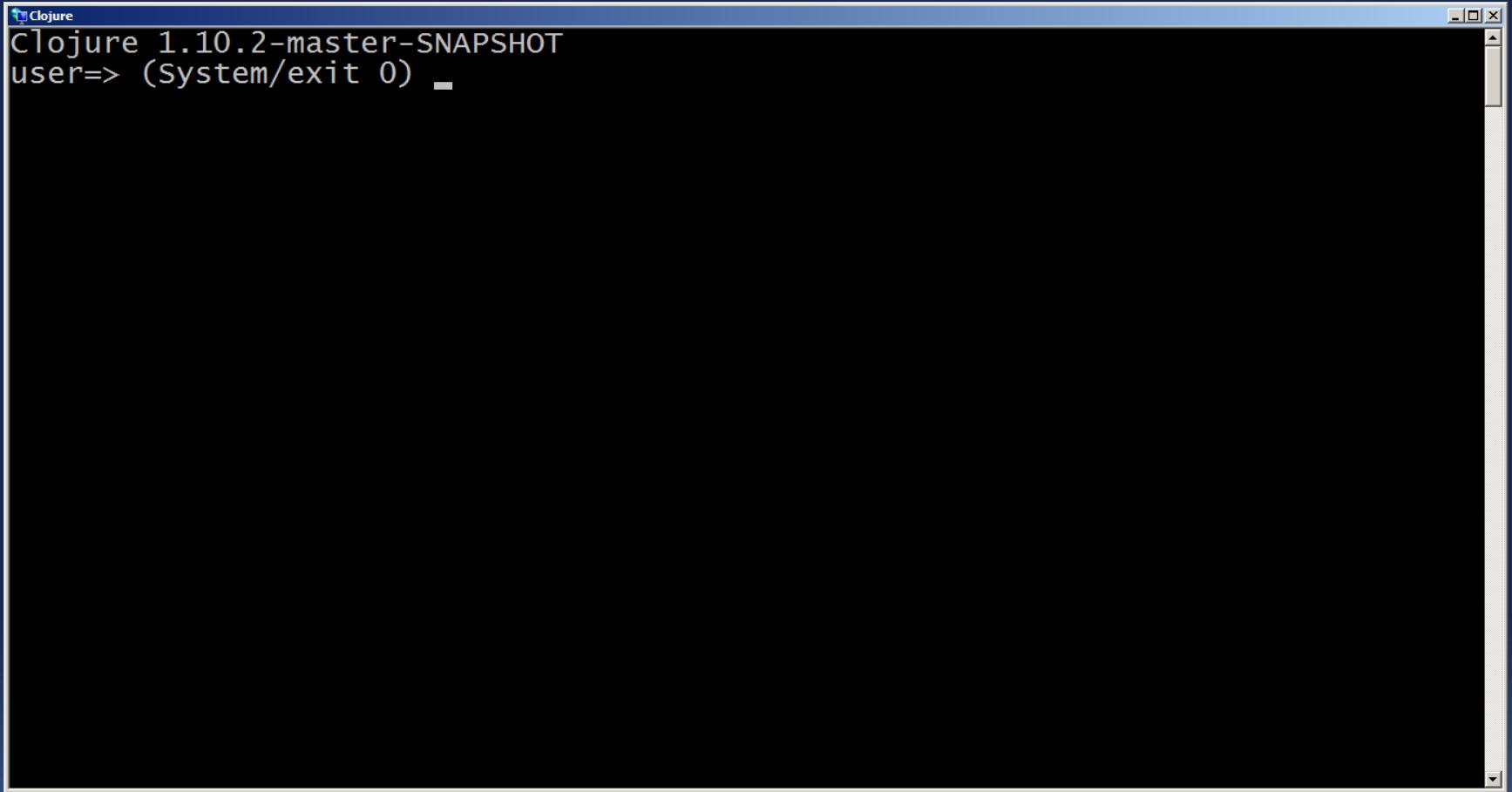
The screenshot shows a Clojure REPL window titled "Clojure". The code entered is:

```
Clojure 1.10.2-master-SNAPSHOT
user=> (+ (* 3 2) (- 5 3))
8
user=>
```



Exit REPL

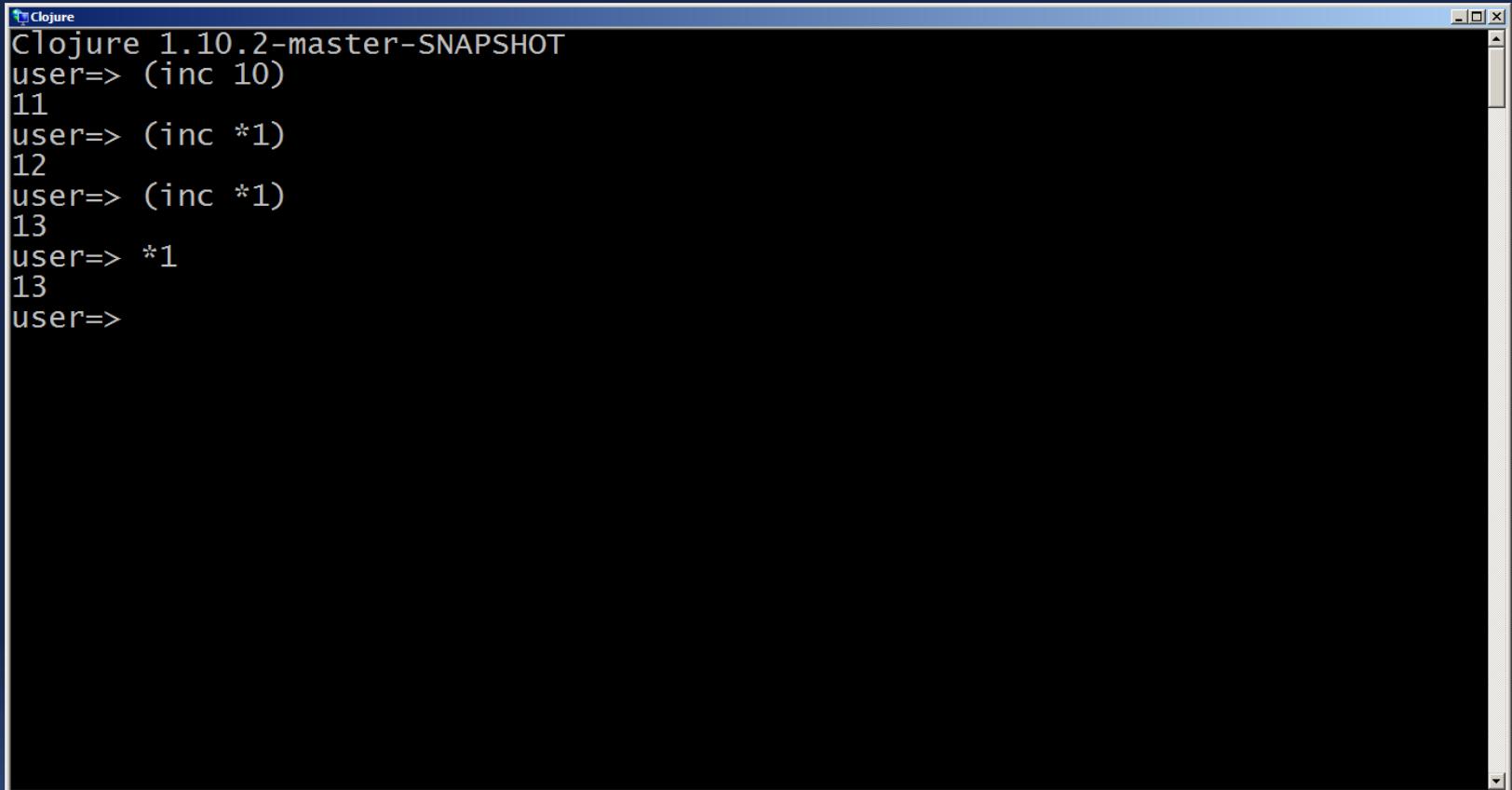
(System/exit 0)



A screenshot of a terminal window titled "Clojure". The window shows the Clojure 1.10.2-master-SNAPSHOT REPL. The user has typed the command `(System/exit 0)` and is pressing the Enter key. The terminal is black with white text, and the window has a standard operating system title bar.

```
Clojure 1.10.2-master-SNAPSHOT
user=> (System/exit 0) -
```

Operando com a última expressão avaliada => *1



The screenshot shows a terminal window titled "Clojure" running on "Clojure 1.10.2-master-SNAPSHOT". The user has entered several commands:

```
Clojure 1.10.2-master-SNAPSHOT
user=> (inc 10)
11
user=> (inc *1)
12
user=> (inc *1)
13
user=> *1
13
user=>
```

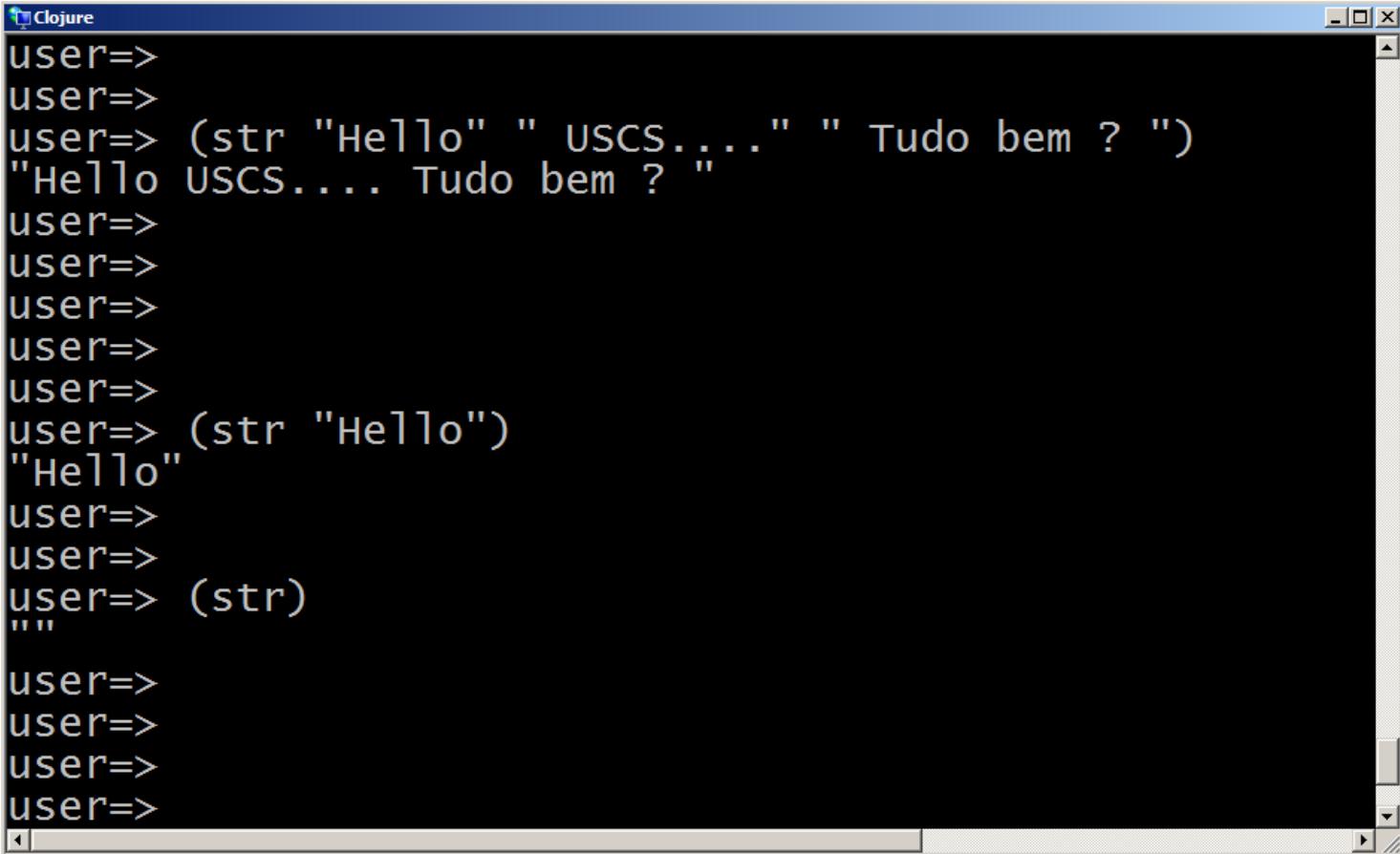
The final output, "13", is highlighted in green, indicating it is the value of the special variable *1.

Operando com a variável que contém o resultado da última exceção => *e

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (/ 1 0)
Execution error (ArithmetricException) at user/eval17 (REPL:1).
Divide by zero
user=>
user=>
user=>
user=>
user=> *e
#error {
  :cause "Divide by zero"
  :via
  [{:type java.lang.ArithmetricException
    :message "Divide by zero"
    :at [clojure.lang.Numbers divide "Numbers.java" 188]}]
  :trace
  [[clojure.lang.Numbers divide "Numbers.java" 188]
   [clojure.lang.Numbers divide "Numbers.java" 3901]
   [user$eval17 invokeStatic "NO_SOURCE_FILE" 1]
   [user$eval17 invoke "NO_SOURCE_FILE" 1]]
```

Concatenando Strings – str

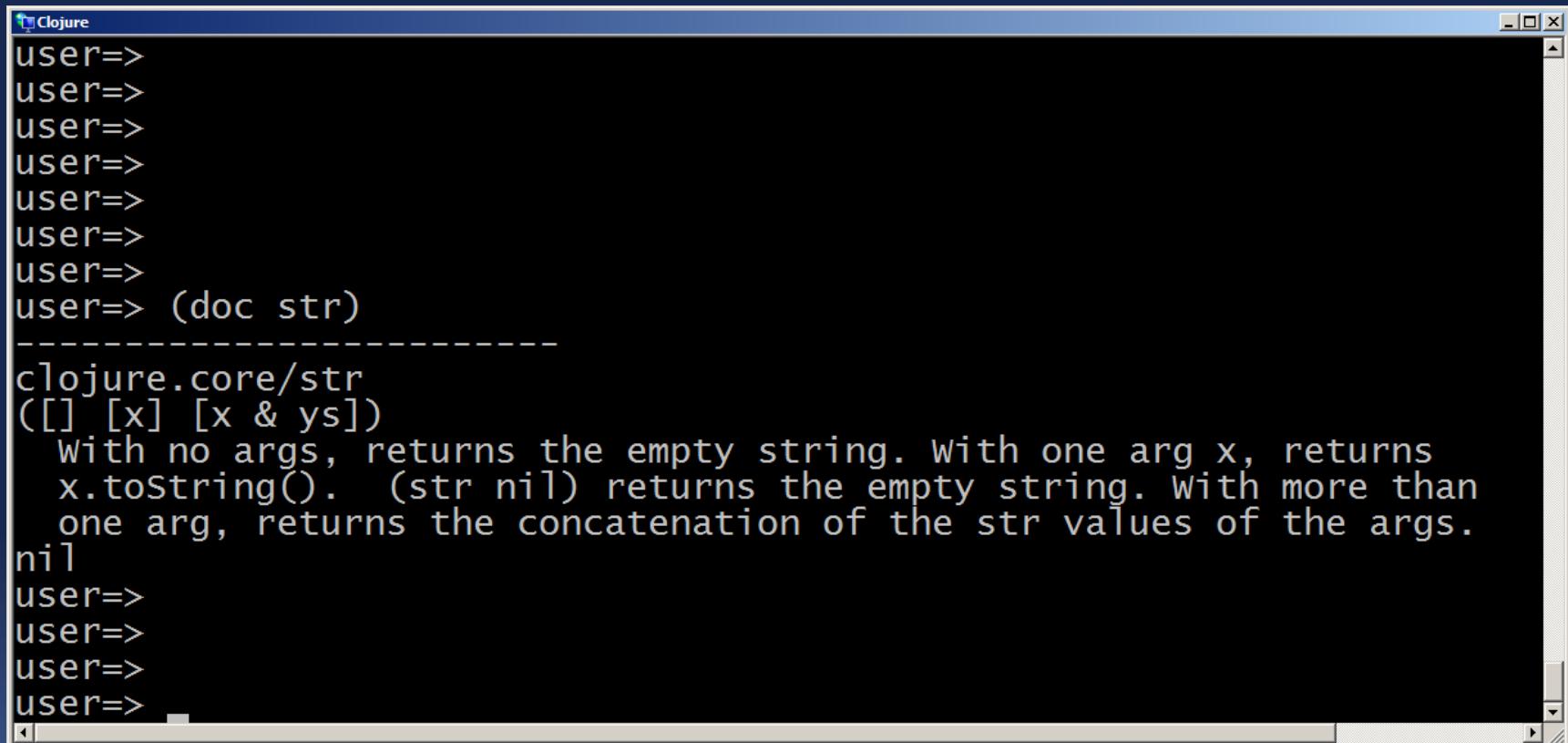
- ✓ A função **str** sem parâmetros retorna o String **nulo** ("")
- ✓ Com um argumento **x** retorna **x.toString()**:
- ✓ Com **mais de um argumento** **x...y** retorna a **concatenação** desses argumentos !



```
Clojure
user=>
user=>
user=> (str "Hello" " USCS...." " Tudo bem ? ")
"Hello USCS.... Tudo bem ? "
user=>
user=>
user=>
user=>
user=>
user=>
user=> (str "Hello")
"Hello"
user=>
user=>
user=> (str)
""
user=>
user=>
user=>
user=>
```

Função doc

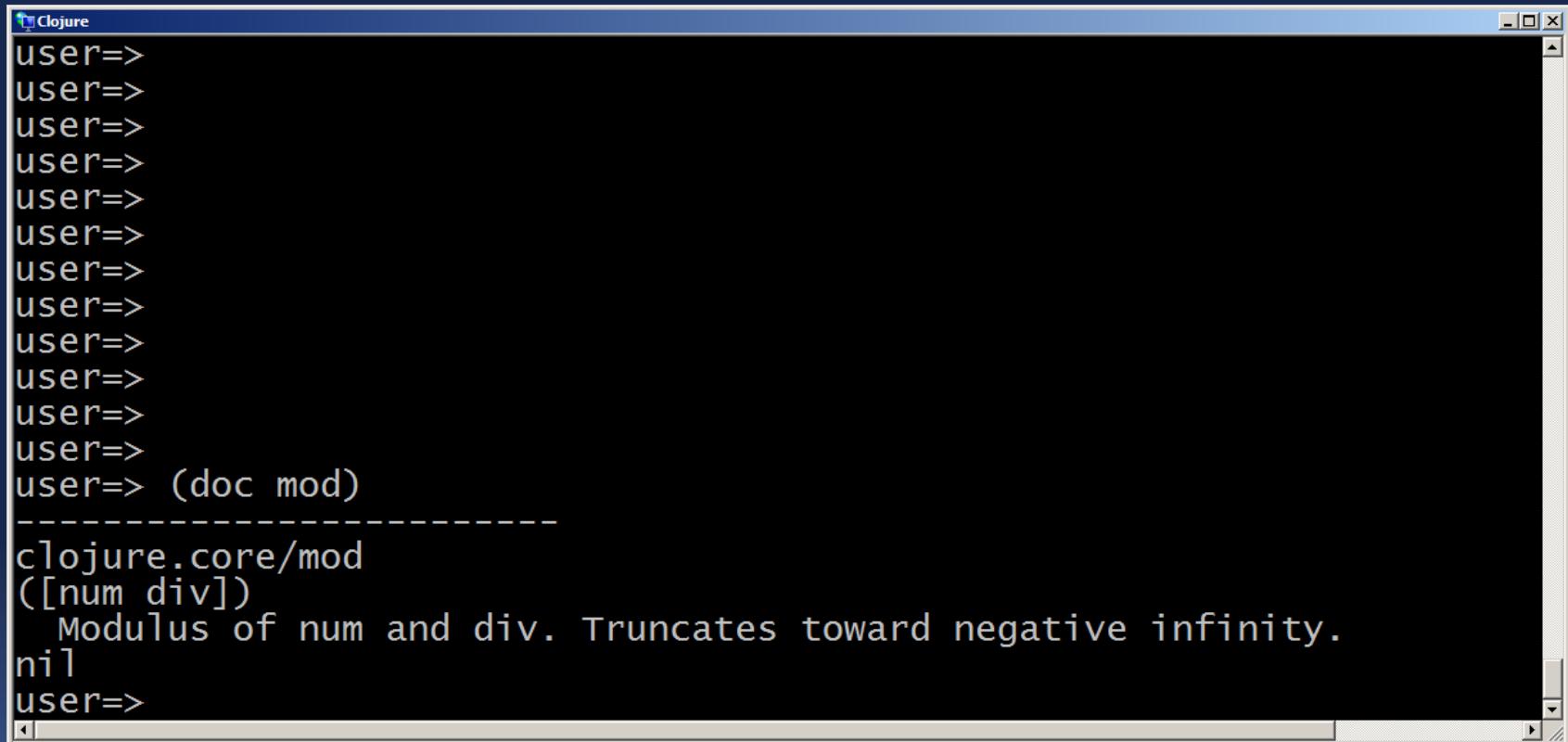
- ✓ A função **doc** permite que se pesquise a **documentação** através de **REPL**;
- ✓ Dado que se conhece o nome da função que se quer usar, pode-se ler a documentação desta função por meio da função **doc**.
- ✓ Exemplo; Obtenção da documentação da função **str**:



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (doc str)
-----
clojure.core/str
([] [x] [x & ys])
With no args, returns the empty string. With one arg x, returns
x.toString(). (str nil) returns the empty string. With more than
one arg, returns the concatenation of the str values of the args.
nil
user=>
user=>
user=>
user=>
```

Função doc

- ✓ Exibe o nome completamente qualificado da função (incluindo o **namespace**) na **primeira linha**, os possíveis **parâmetros** (ou “aridades”) na próxima linha e finalmente a descrição da função que se está pesquisando.
- ✓ Exemplo: **(doc mod)**

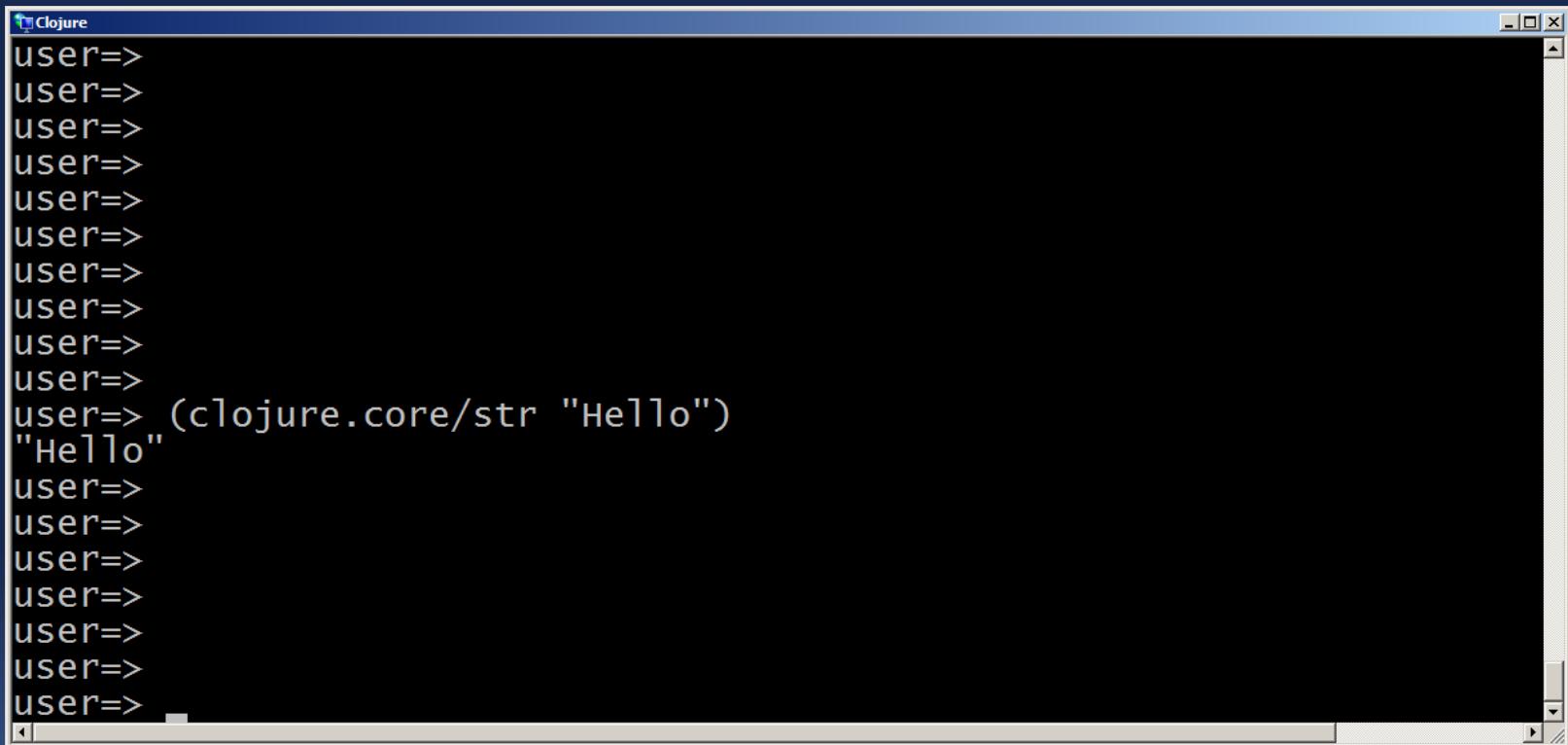


The screenshot shows a terminal window titled "Clojure". The user has entered the command "(doc mod)". The output is as follows:

```
user=>
user=> (doc mod)
-----
clojure.core/mod
([num div])
  Modulus of num and div. Truncates toward negative infinity.
nil
user=>
```

clojure.core

- ✓ Tudo o que está definido em `clojure.core` estará disponível ao seu `namespace` corrente por `default`, dispensando-se assim de se explicitamente requerê-la;
- ✓ Por exemplo: `(str "hello")` ao invés de `(clojure.core/str "hello")`



The screenshot shows a Windows-style application window titled "Clojure". Inside the window, the Clojure REPL is running. The history of input and output is as follows:

```
user=>
user=> (clojure.core/str "Hello")
>Hello
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

(find-doc)

- ✓ Quando **não** se sabe o nome da função, mas se tem uma ideia de sua descrição ou nome que pode conter, pode-se usar a função **find-doc**;

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (find-doc "float")
-----
clojure.pprint/float-parts
([f])
  Take care of leading and trailing zeros in decomposed floats
-----
clojure.pprint/float-parts-base
([f])
  Produce string parts for the mantissa (normalized 1-9) and exponent
-----
clojure.core/aset-float
([array idx val] [array idx idx2 & idxv])
  Sets the value at the index/indices. Works on arrays of float. Returns val.
-----
clojure.core/float
([x])
  Coerce to float
```

Função apropos

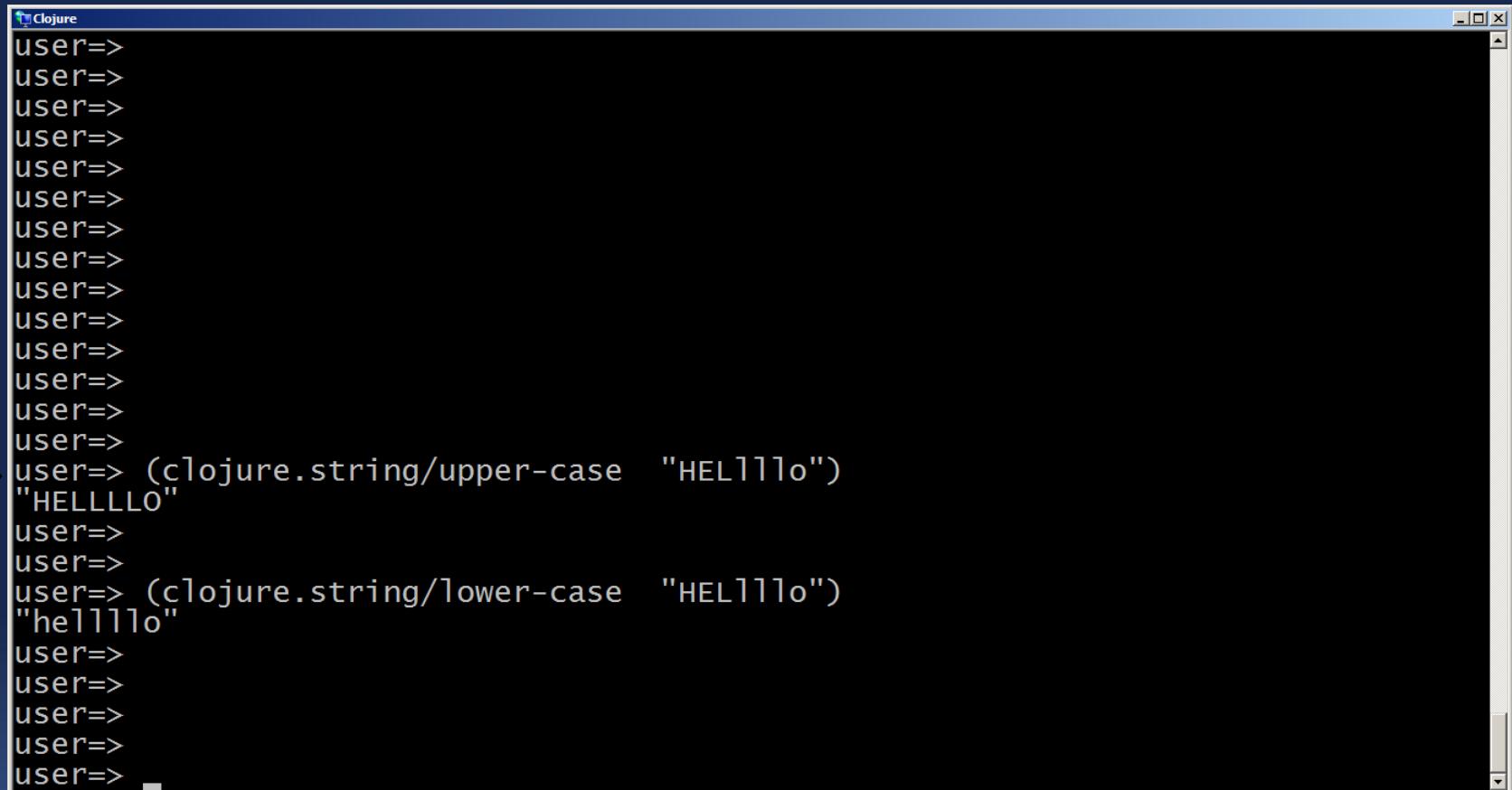
- ✓ Usada para pesquisar funções pelo **nome**, resultando assim uma saída mais sucinta. Por exemplo, pesquisar uma função que transforma um dado string em caracteres maiúsculos ou minúsculos;
- ✓ Observamos na saída que **lower-case** e **upper-case** estão definidas no namespace **clojure.string**.



```
Clojure
user=>
user=> (apropos "case")
(clojure.core/case clojure.string/lower-case clojure.string/upper-case)
user=>
```

lower-case e upper-case

- ✓ Observamos na saída que **lower-case** e **upper-case** estão definidas no namespace **clojure.string**.



```
Clojure
user=>
user=> (clojure.string/upper-case "HELLllo")
"HELLLLO"
user=>
user=>
user=> (clojure.string/lower-case "HELllo")
"hellllo"
user=>
user=>
user=>
user=>
user=>
```

The screenshot shows a Clojure REPL window titled "Clojure". The user has entered several forms. Two specific forms are highlighted with light blue arrows pointing to them:

- (clojure.string/upper-case "HELLllo") which returns "HELLLLO"
- (clojure.string/lower-case "HELllo") which returns "hellllo"

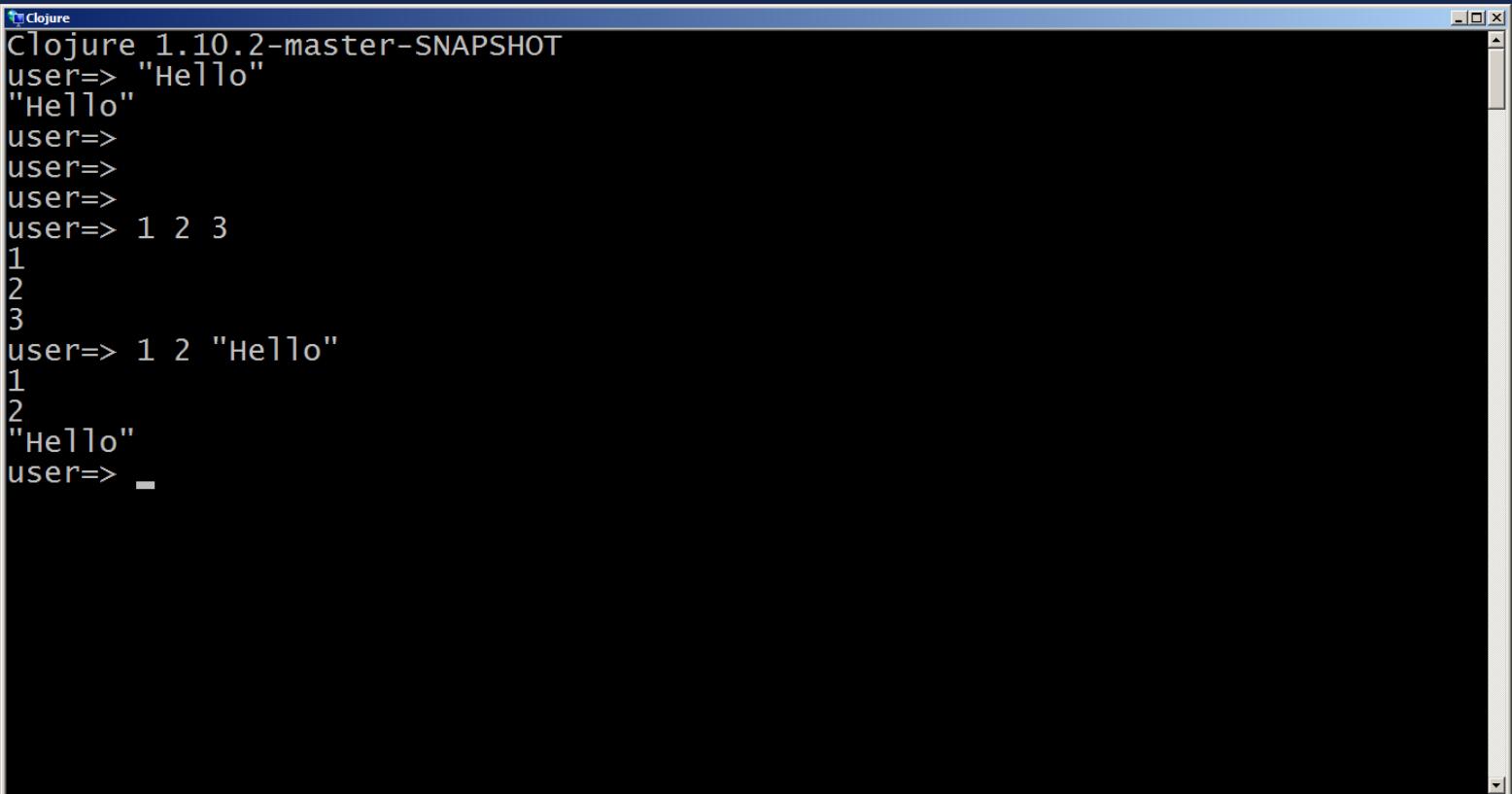
The rest of the session shows standard user input followed by a series of blank responses starting with "user=>".

Atividade 1

- ✓ Abra o **REPL**;
- ✓ Imprima a mensagem: “Eu não tenho medo de parênteses”
- ✓ Some 1, 2 e 3 e multiplique o resultado por 10 menos 3;
- ✓ Imprima a mensagem “Atividade Concluída...”;
- ✓ Encerre o **REPL**.

Avaliação de código em Clojure

- ✓ Em Clojure, literais são expressões válidas e ao serem avaliadas retornam os próprios literais.



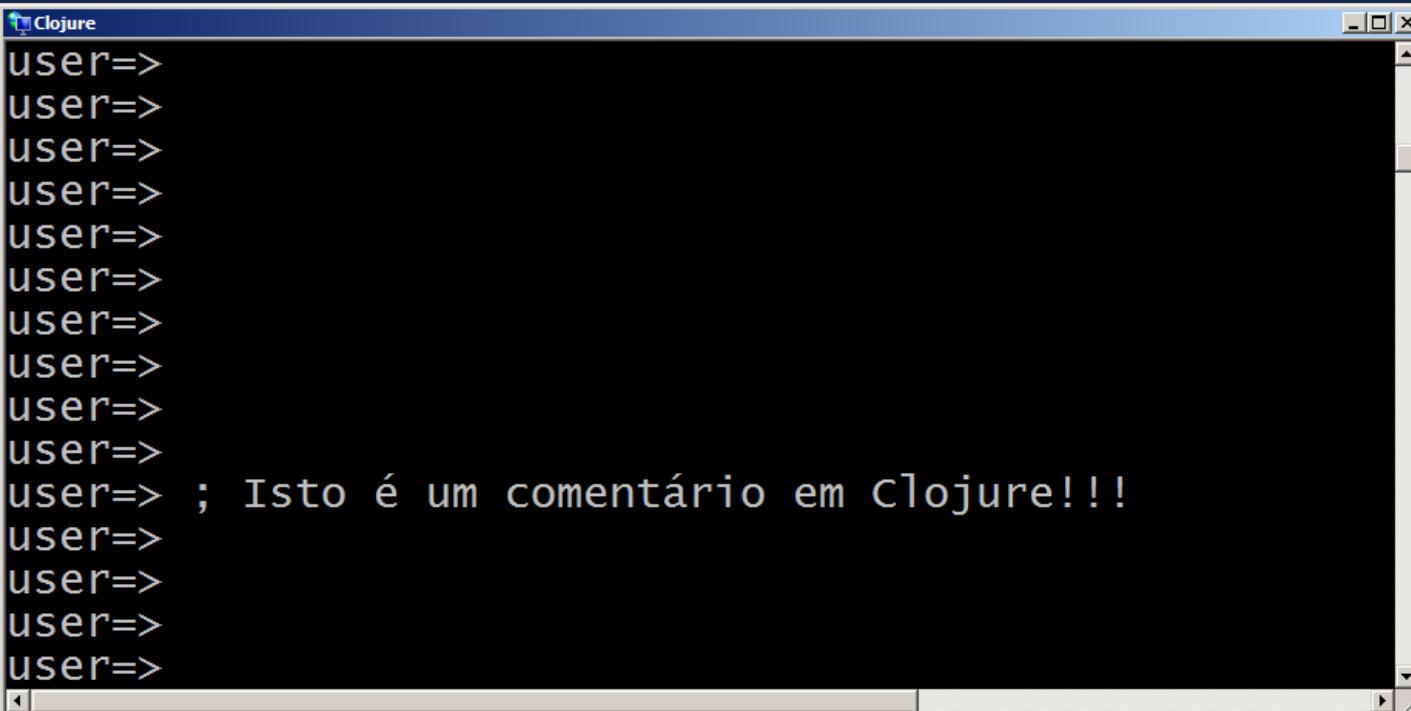
The screenshot shows a terminal window titled "Clojure" with the version "Clojure 1.10.2-master-SNAPSHOT". The window displays the following Clojure code and its evaluated results:

```
Clojure 1.10.2-master-SNAPSHOT
user=> "Hello"
"Hello"
user=>
user=>
user=>
user=> 1 2 3
1
2
3
user=> 1 2 "Hello"
1
2
"Hello"
user=> _
```

Three light blue arrows point from the left towards the first three lines of the code, highlighting them as examples of literals being evaluated.

Comentários em Clojure

- ✓ Qualquer linha iniciada com ;



A screenshot of a Windows-style application window titled "Clojure". The window contains a black text area with white text. It shows multiple lines of Clojure code starting with "user=>". At the bottom, there is a single line of text: "user=> ; Isto é um comentário em Clojure!!!". The window has standard window controls (minimize, maximize, close) at the top right and scroll bars on the right side.

```
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> ; Isto é um comentário em Clojure!!!
user=>
user=>
user=>
user=>
```

Funções em Clojure

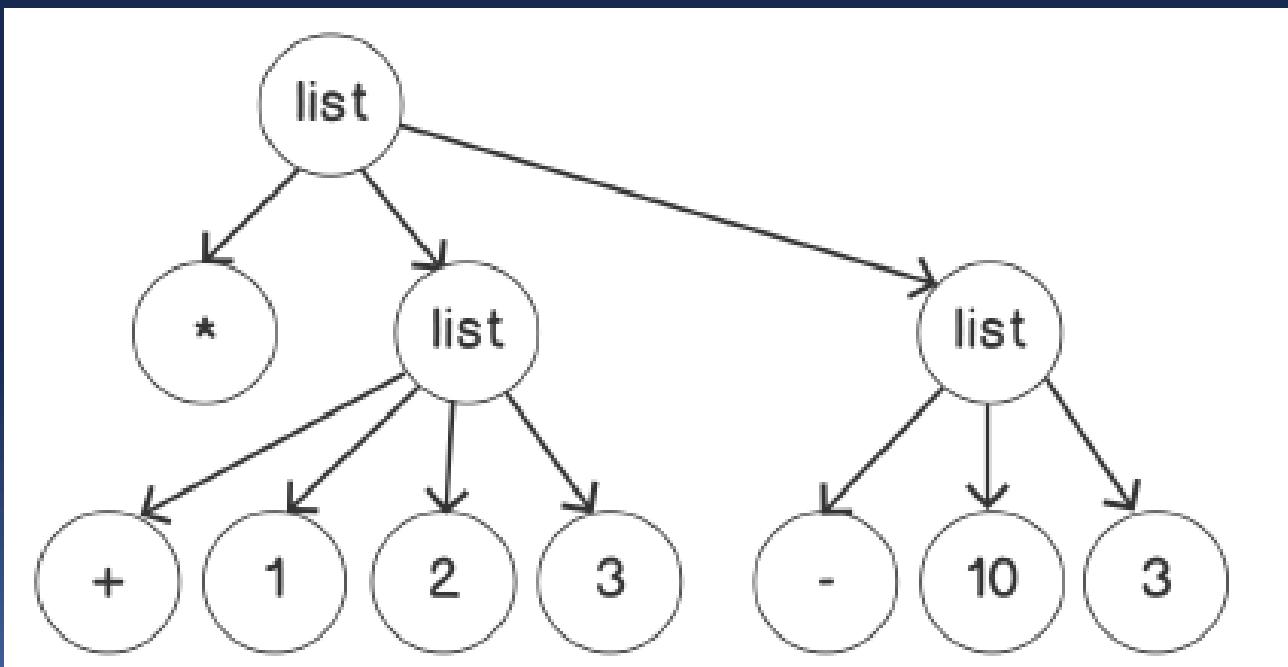
- ✓ São invocadas com a seguinte estrutura:

```
; (operator operand-1 operand-2 operand-3 ...)
; for example:
user=> (* 2 3 4)
24
```

Expressões Simbólicas (s-expression)

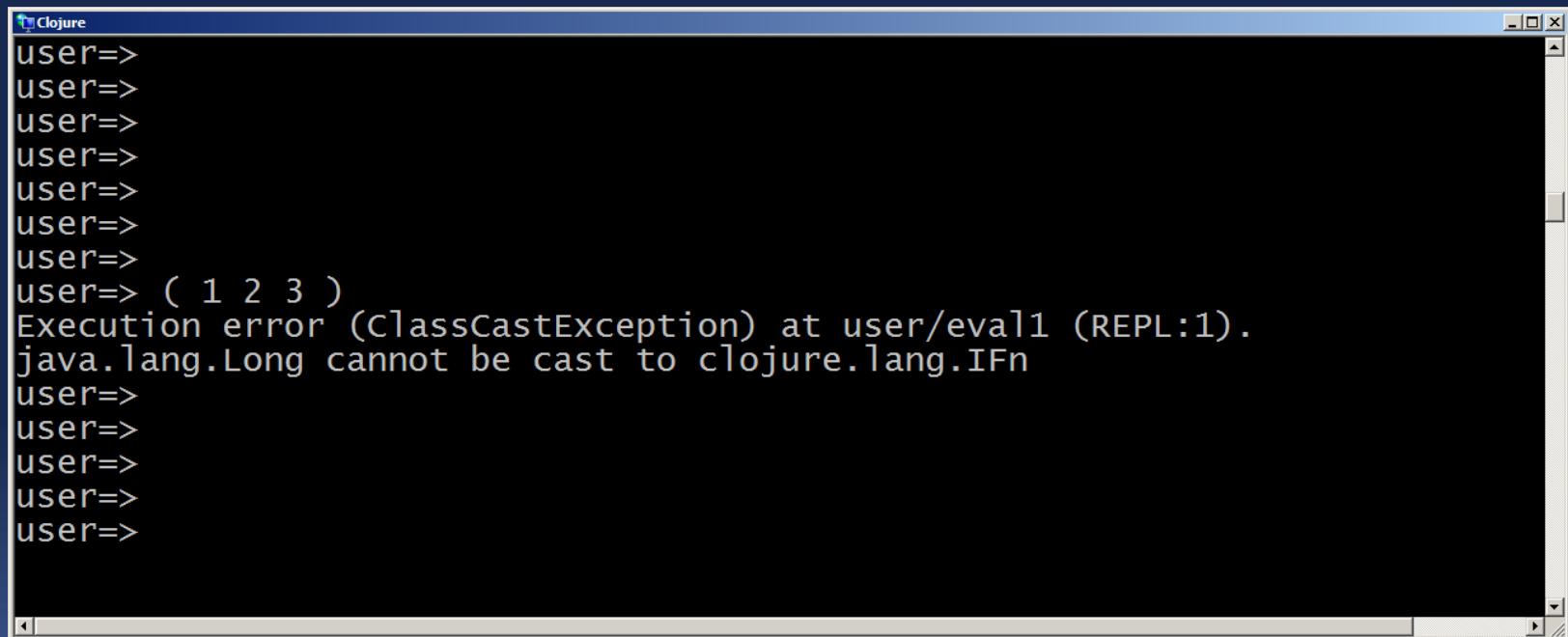
- ✓ Podem ser visualizadas em uma árvore

```
(* (+ 1 2 3) (- 10 3))
```



Listas

- ✓ São usadas para representar funções;
- ✓ Ao usarmos listas para representar **dados**, Clojure retorna erro:

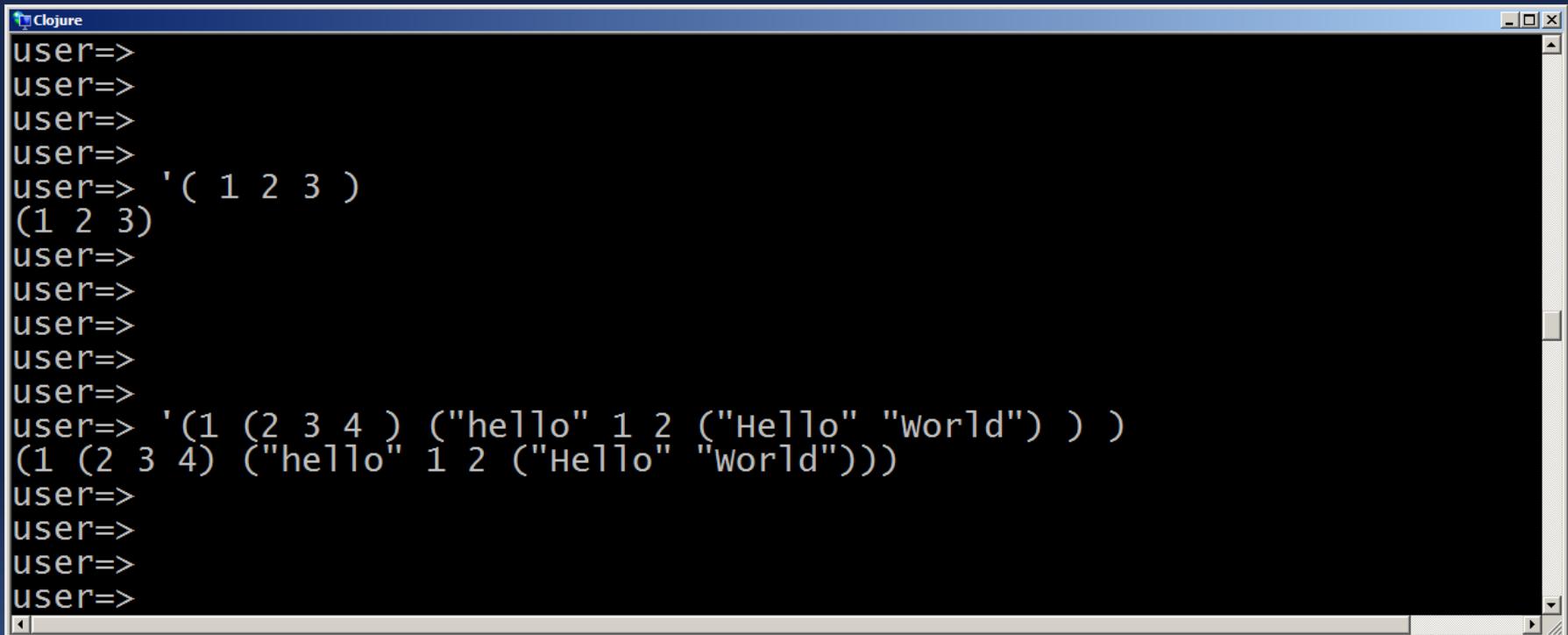


The screenshot shows a Windows-style application window titled "Clojure". Inside, the REPL history is displayed. The user has entered several "user=>" prompts, followed by "(1 2 3)". This triggers an execution error, resulting in a "ClassCastException" at "user/eval1 (REPL:1)". The error message specifies that "java.lang.Long cannot be cast to clojure.lang.IFn". The window has standard window controls (minimize, maximize, close) and scroll bars.

```
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> ( 1 2 3 )
Execution error (ClassCastException) at user/eval1 (REPL:1).
java.lang.Long cannot be cast to clojure.lang.IFn
user=>
user=>
user=>
user=>
user=>
```

Listas como dados

- ✓ Para que listas não sejam interpretadas como funções, mas simplesmente como dados, devemos circundá-las por apóstrofe ('');



The screenshot shows a Windows-style application window titled "Clojure". Inside, the Clojure REPL is running. The user has entered several list literals, each preceded by a red "user=>". The first two lists are simple numerical sequences: '(1 2 3)' and '(1 2 3)'. The third and fourth lists are more complex, containing strings and numbers: '(1 (2 3 4) ("hello" 1 2 ("Hello" "world")))' and '(1 (2 3 4) ("hello" 1 2 ("Hello" "world")))'.

```
Clojure
user=>
user=>
user=>
user=>
user=> '( 1 2 3 )
(1 2 3)
user=>
user=>
user=>
user=>
user=>
user=> '(1 (2 3 4) ("hello" 1 2 ("Hello" "world") ) )
(1 (2 3 4) ("hello" 1 2 ("Hello" "world")))
user=>
user=>
user=>
user=>
```

Código Clojure

- ✓ Código **Clojure** é constituído por estruturas de dados;
- ✓ Nossos programas podem gerar essas estruturas de dados;
- ✓ Assim, num jargão **Lisp**, programas em tempo de execução podem gerar código;
- ✓ Esse conceito é conhecido por **Meta-Programação**.

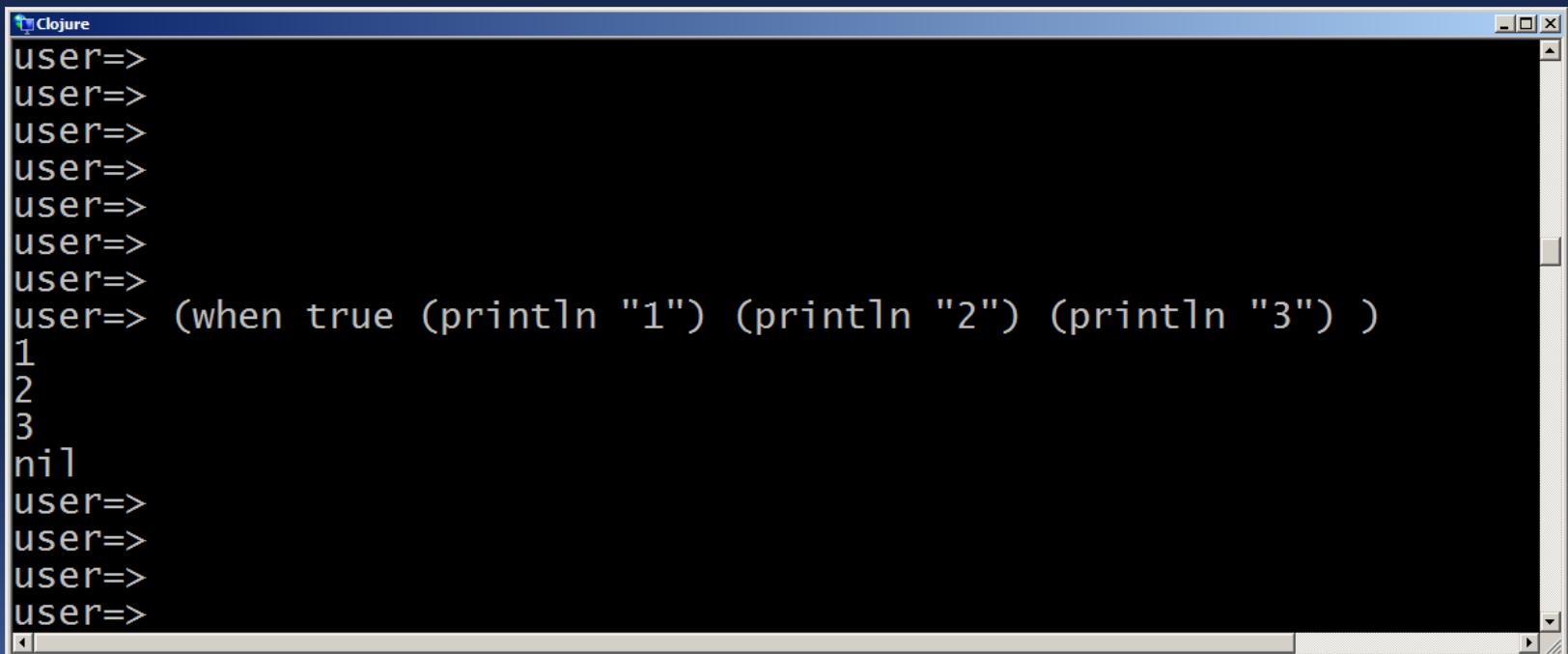


Special Forms

- ✓ Vimos até agora códigos nos quais **Clojure** emprega regras simples de avaliação incorporadas em listas;
- ✓ **Mas**, existem alguns comportamentos que **não** são tratados como nas avaliações usuais;
- ✓ Por exemplo, argumentos passados para uma função sempre são avaliados, mas e se **não** quizermos que todos os argumentos sejam avaliados ?
- ✓ Essas regras especiais de avaliação ocorrem nos “**Special Forms**”.
- ✓ Por exemplo, a “**special form**” **if** pode **não** avaliar um de seus argumentos, dependendendo do resultado da avaliação do primeiro argumento;

Macro - when

- ✓ Usado quando somente estamos interessados no caso em que a condição lógica for verdadeira;
- ✓ É semelhante a um **if**, mas não contém ramificação **else** e pode ser tratado como um “**do**” **implícito**.



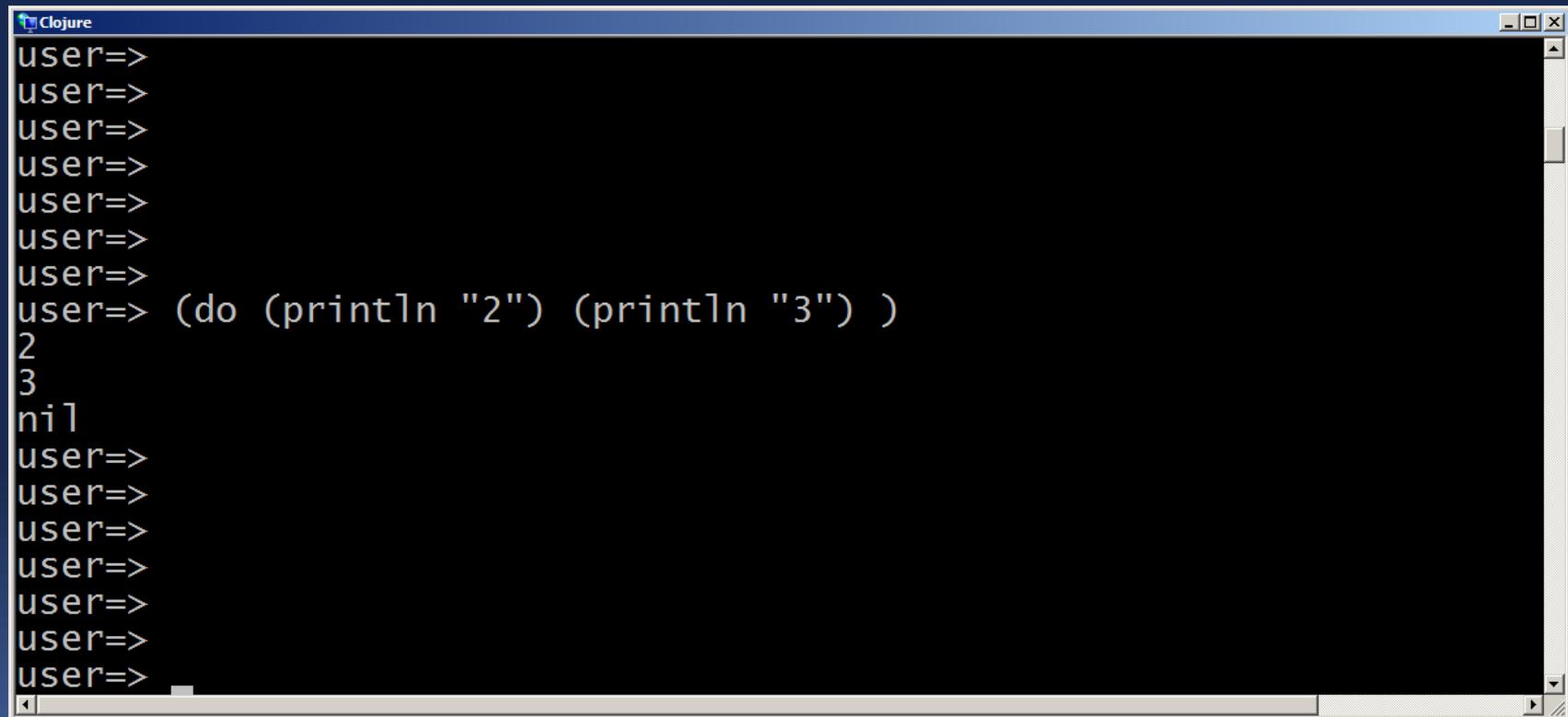
The screenshot shows a Windows-style application window titled "Clojure". Inside, the Clojure REPL is running. The user has typed:

```
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (when true (println "1") (println "2") (println "3"))
1
2
3
nil
user=>
user=>
user=>
user=>
```

The output shows the numbers 1, 2, and 3 printed sequentially, followed by nil, indicating that the when macro evaluated its condition as true and executed the body code.

Special Forms - do

- ✓ Usado para executar uma série de expressões e retornar o valor da última expressão.



The screenshot shows a terminal window titled "Clojure". The user has entered the following Clojure code:

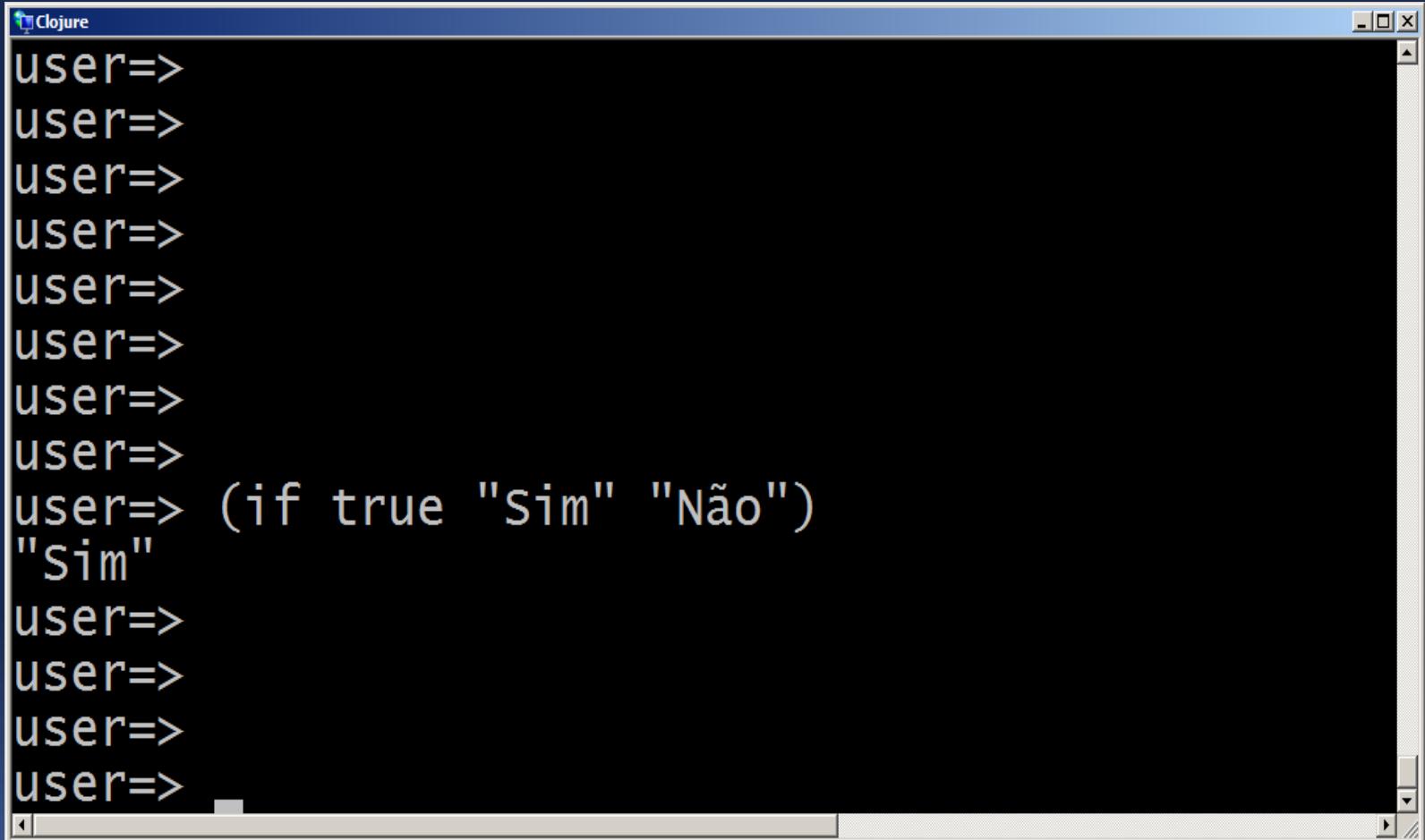
```
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (do (println "2") (println "3") )
2
3
nil
user=>
user=>
user=>
user=>
user=>
user=>
```

The output shows the numbers 2 and 3 printed to the console, followed by the nil value, which is the return value of the do expression.

Atividade 2

- ✓ Abra o **REPL**;
- ✓ Escreva a função `if` que retorna “Sim” se a condição for verdadeira e “Não” caso contrário.
- ✓ Encerre o **REPL**.

Atividade 2



The screenshot shows a terminal window titled "Clojure" with a blue header bar. The window contains the following text:

```
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (if true "Sim" "Não")
"Sim"
user=>
user=>
user=>
user=>
```

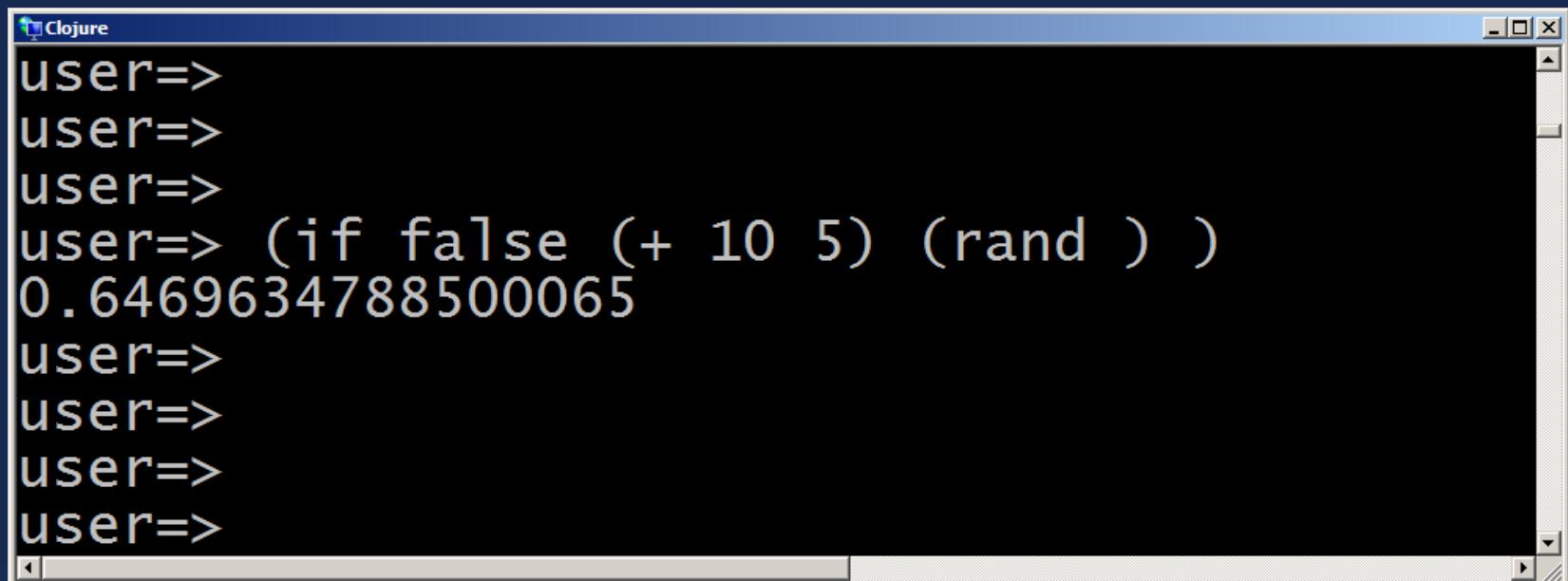
Atividade 3

- ✓ Abra o **REPL**;
- ✓ Escreva a função `if` que retorna a soma de 10 com 5 se a condição for falsa. Caso contrário retorna um número randômico.
- ✓ Encerre o **REPL**.

Dica. A função `rand` retorna um número `rand`.



Atividade 3



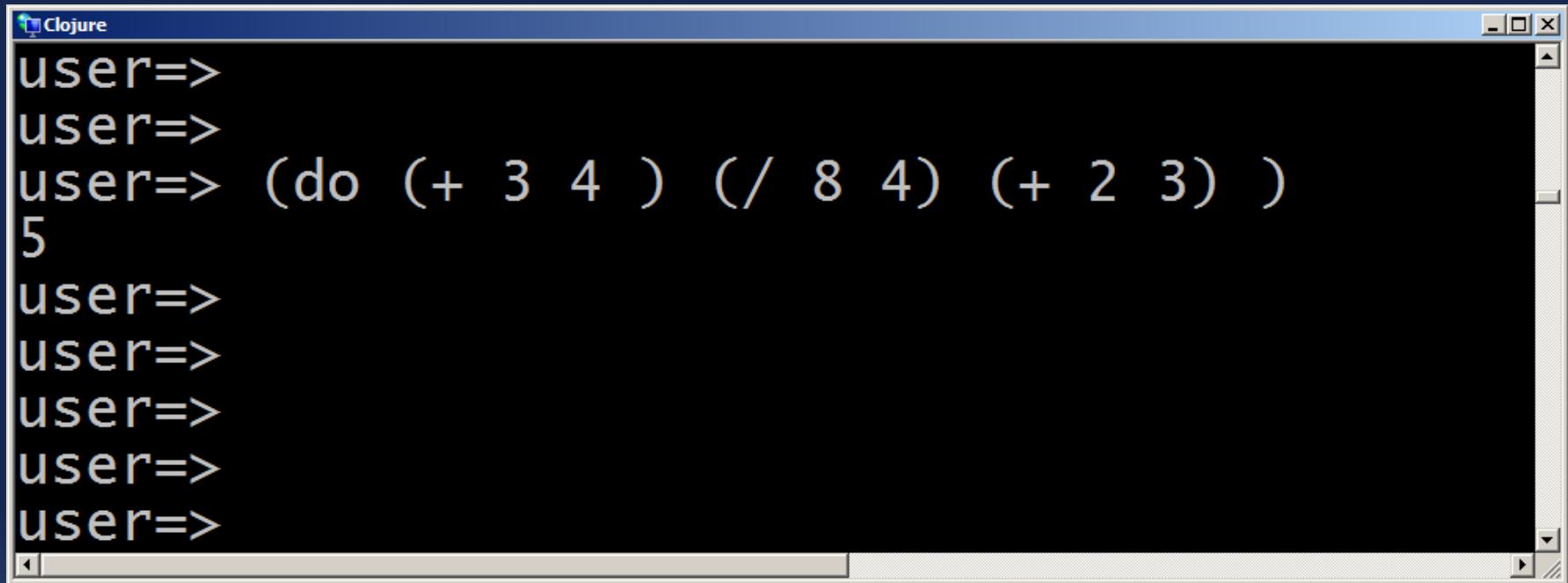
The screenshot shows a Windows-style application window titled "Clojure". Inside, a terminal-like interface displays the following interaction:

```
user=>
user=>
user=>
user=> (if false (+ 10 5) (rand) )
0.6469634788500065
user=>
user=>
user=>
user=>
```

Atividade 4

- ✓ Abra o **REPL**;
- ✓ Escreva a função “**do**” que executa sequencialmente as seguintes avaliações:
 - ✓ Multiplica 3 por 4;
 - ✓ Divide 8 por 4;
 - ✓ Soma 2 com 3.
- ✓ Encerre o **REPL**.

Atividade 4



The screenshot shows a Windows-style window titled "Clojure". Inside, a Clojure REPL session is running. The user has entered several expressions:

- "user=>" (repeated twice)
- "user=> (do (+ 3 4) (/ 8 4) (+ 2 3))"
- "5"
- "user=>" (repeated four times)

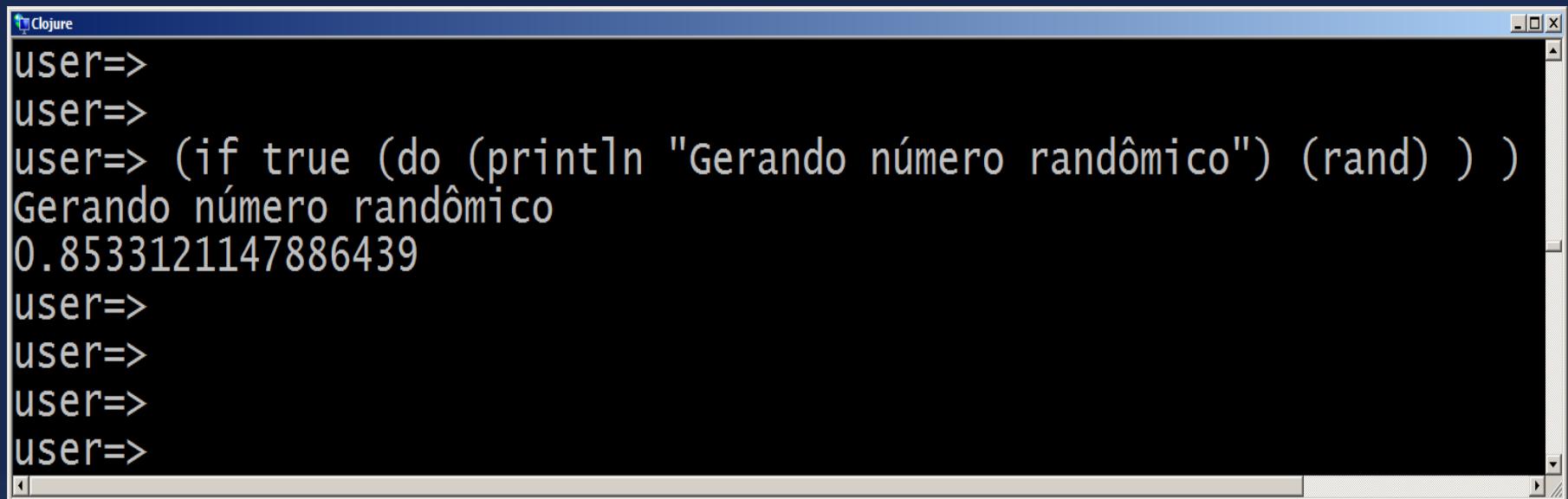
Atividade 5

- ✓ Abra o **REPL**;

- ✓ Escreva a função “**if**” que imprime a mensagem “**Gerando um número randômico**” e exibe o número gerado, caso a condição seja verdadeira;

- ✓ Encerre o **REPL**.

Atividade 5



A screenshot of a Windows-style application window titled "Clojure". The window contains a black terminal-like interface. The user has entered several Clojure expressions. The first two expressions are empty lines starting with "user=>". The third expression is a conditional statement that prints a message and generates a random number. The message "Gerando número randômico" is printed, followed by the generated number "0.8533121147886439". Subsequent empty lines starting with "user=>" indicate further interactions.

```
Clojure
user=>
user=>
user=> (if true (do (println "Gerando número randômico") (rand) ) )
Gerando número randômico
0.8533121147886439
user=>
user=>
user=>
user=>
```

Atividade 6

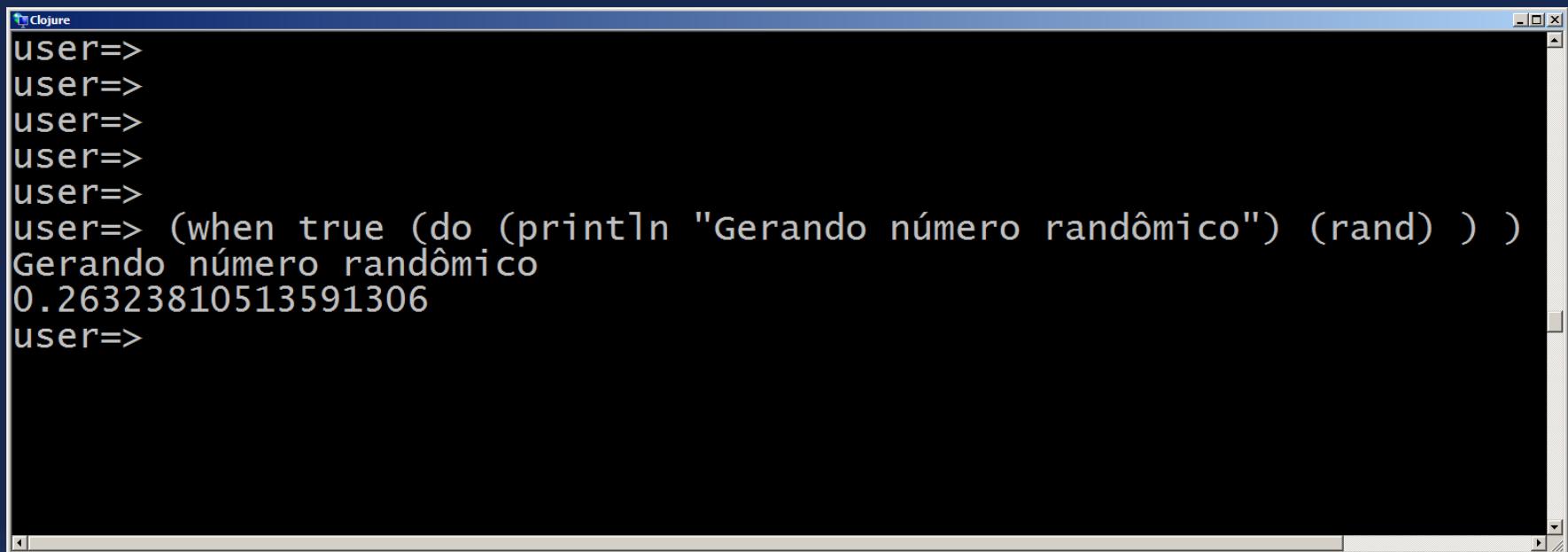
- ✓ Abra o **REPL**;

- ✓ Escreva a função “**when**” que imprime a mensagem “Gerando um número randômico” e exibe o número gerado, caso a condição seja verdadeira;

- ✓ Encerre o **REPL**.



Atividade 6



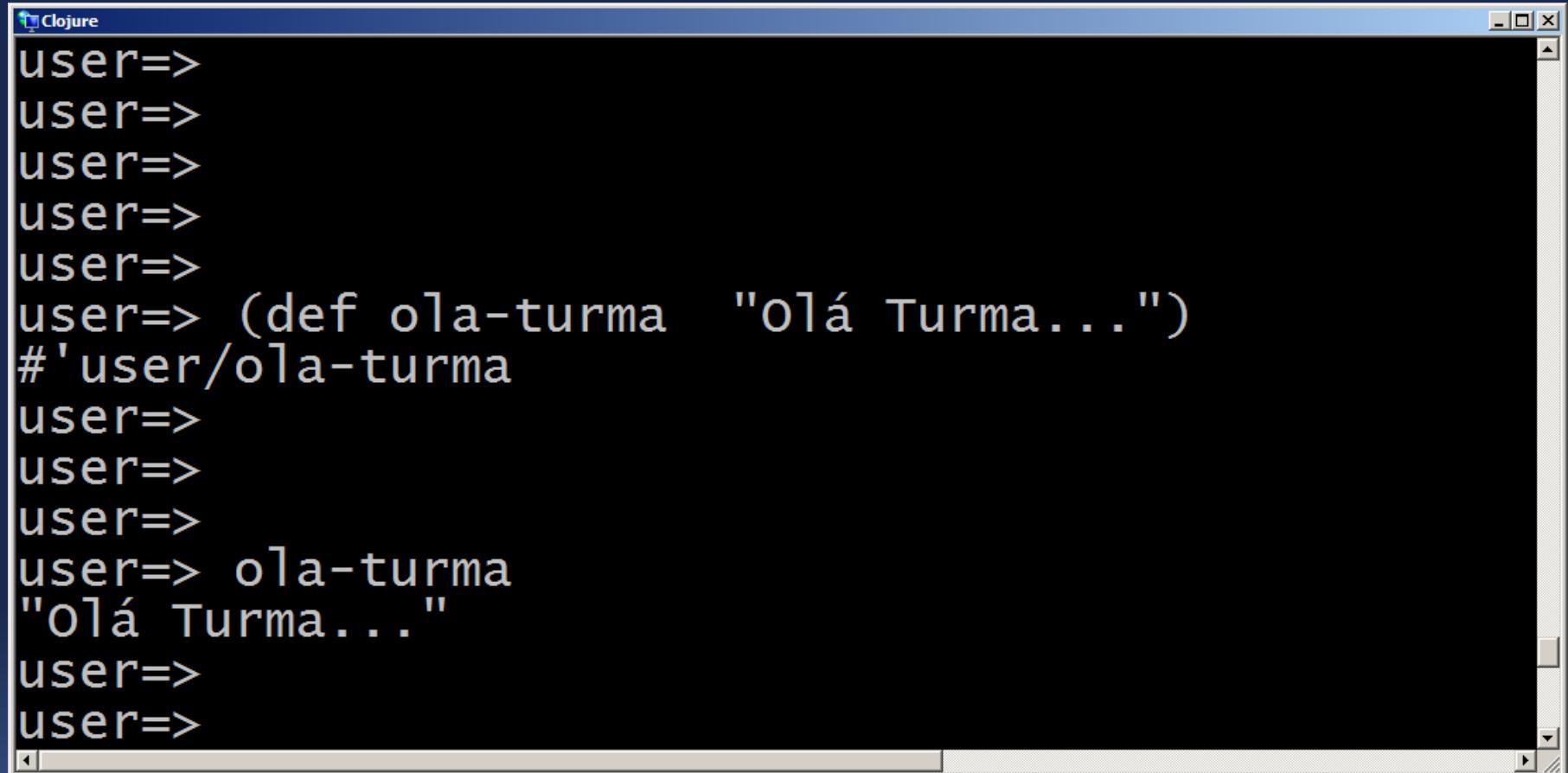
A screenshot of a Clojure REPL window titled "Clojure". The window shows the following interaction:

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (when true (do (println "Gerando número randômico") (rand) ) )
Gerando número randômico
0.26323810513591306
user=>
```

Bindings

- ✓ Em **Clojure**, **Binding** significa **ligação** de símbolos à valores;
- ✓ O termo **Binding** é preferível ao invés de atribuição, pois essas ligações quase sempre são feitas uma única vez;
- ✓ Diferentemente de atribuição em variáveis que, na grande maioria das linguagens, ocorrem com frequência justificando assim o uso no nome de variáveis;
- ✓ Em **Clojure**, o nome **símbolo** é preferível ao nome **variável**;
- ✓ Bindings podem ser feitos em **Clojure** de forma local (função **let**) ou de forma global (função **def**).

Macro - def



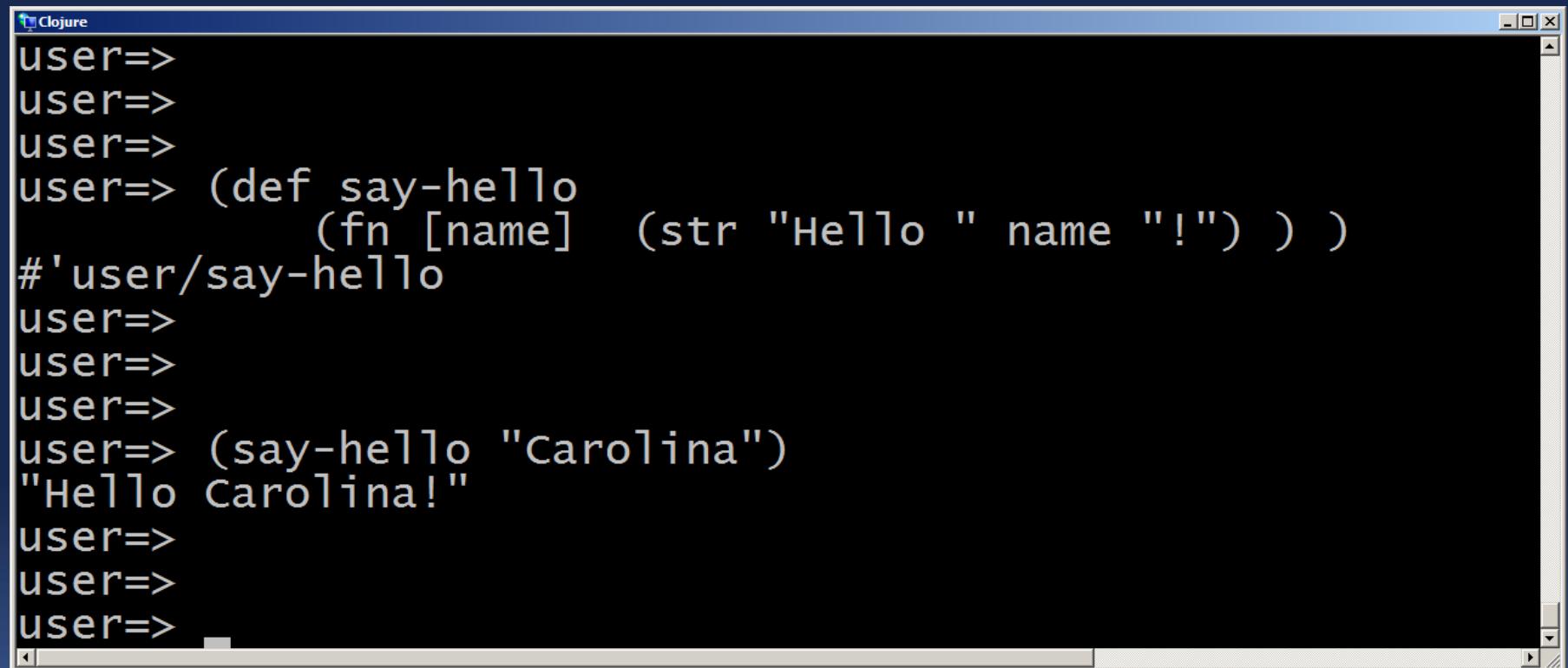
The screenshot shows a Windows-style application window titled "Clojure". Inside, a Clojure REPL session is running. The user has typed the following code:

```
user=>
user=>
user=>
user=>
user=>
user=> (def ola-turma "olá Turma...")
#'user/ola-turma
user=>
user=>
user=>
user=> ola-turma
"olá Turma..."
user=>
user=>
```

The final line, "olá Turma...", is the result of the macro expansion of the `def` form. The REPL prompt "user=>" appears at the end of each line of input.

Macro - def

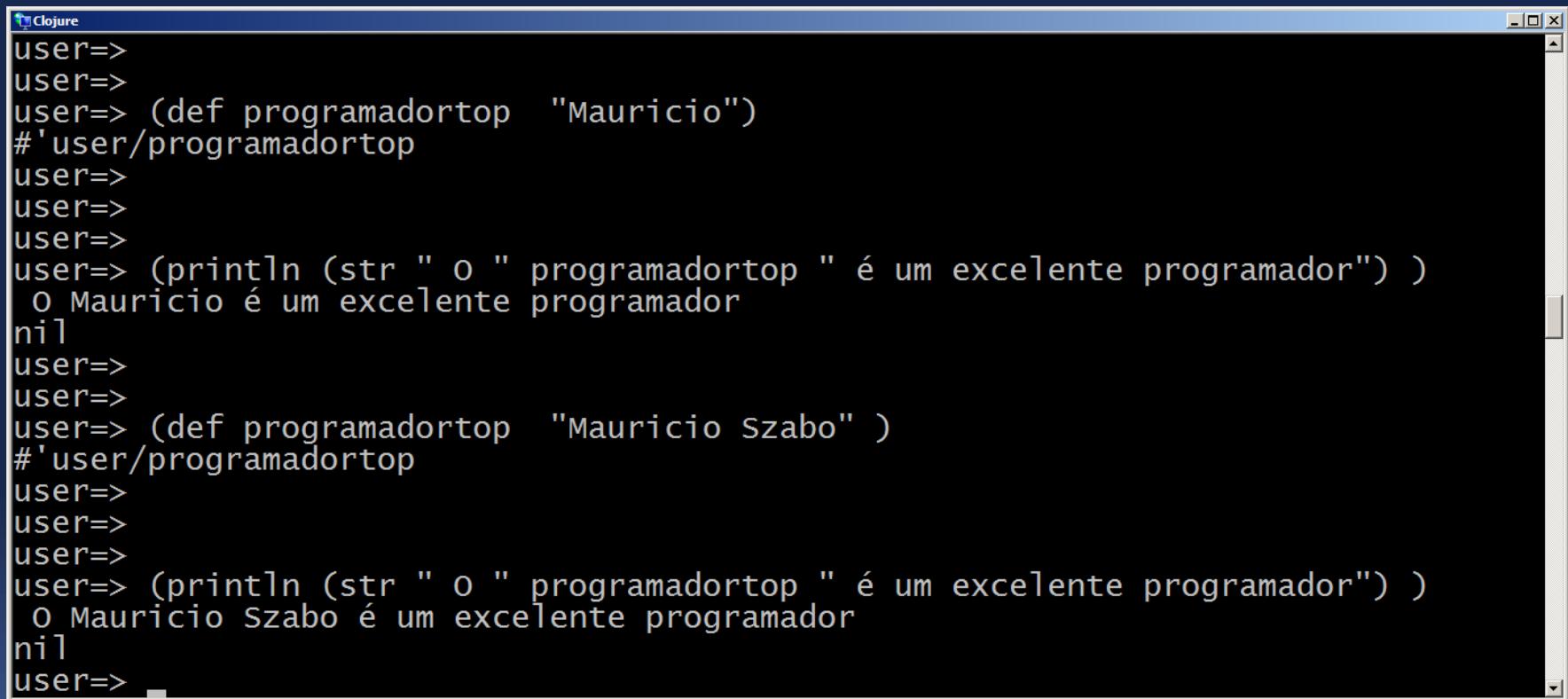
- ✓ Pode ser usada para se associar um **nome** à uma função:



```
Clojure
user=>
user=>
user=>
user=> (def say-hello
          (fn [name]  (str "Hello " name "!") ) )
#'user/say-hello
user=>
user=>
user=>
user=> (say-hello "Carolina")
"Hello Carolina!"
user=>
user=>
user=>
```

Macro - def

- ✓ Usado para se ligar (**binding**) símbolos a determinados valores;



```
Clojure
user=>
user=>
user=> (def programadortop "Mauricio")
#'user/programadortop
user=>
user=>
user=>
user=> (println (str " o " programadortop " é um excelente programador") )
O Mauricio é um excelente programador
nil
user=>
user=>
user=> (def programadortop "Mauricio Szabo" )
#'user/programadortop
user=>
user=>
user=>
user=> (println (str " o " programadortop " é um excelente programador") )
O Mauricio Szabo é um excelente programador
nil
user=>
```

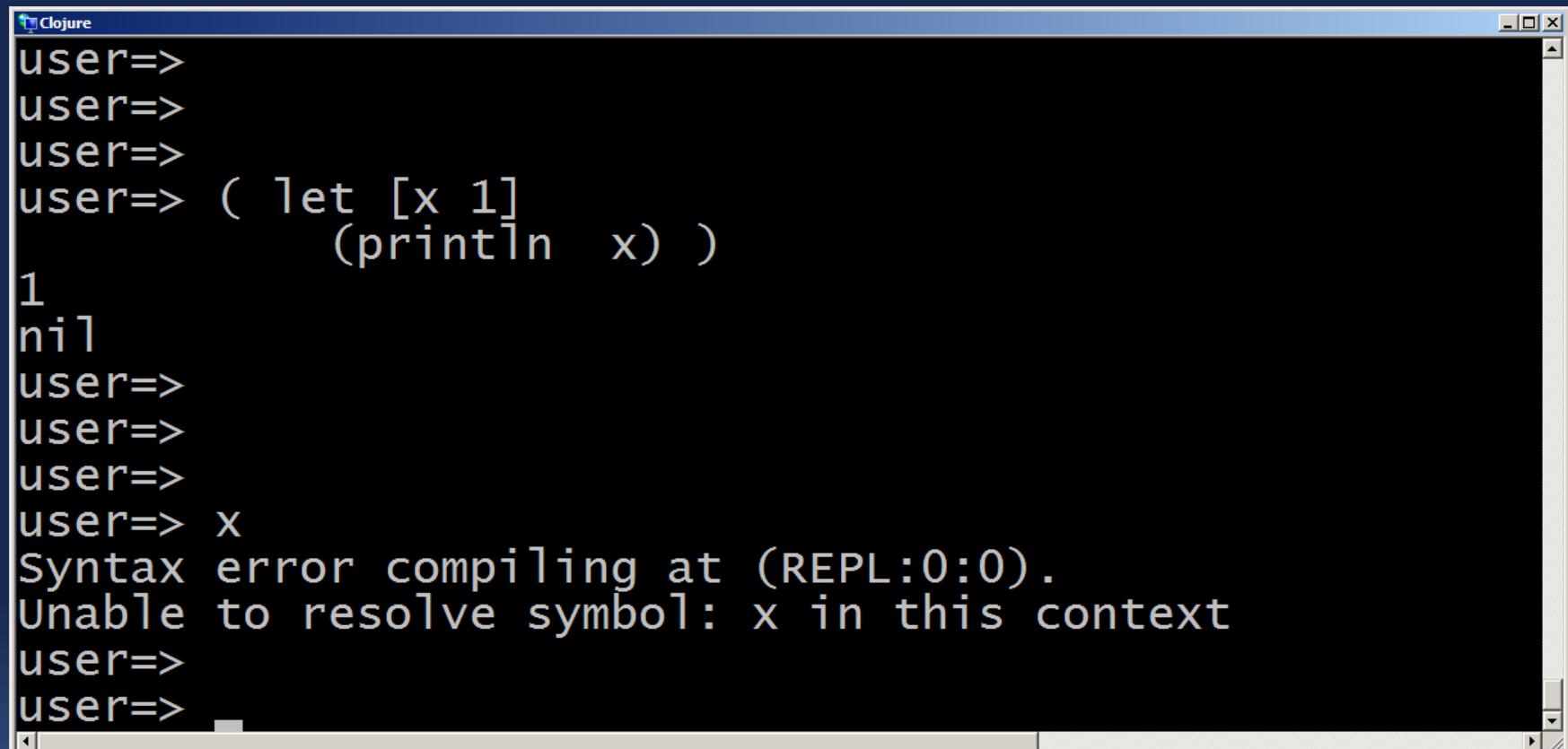
Macro - let

- ✓ Usado para se ligar (**binding**) símbolos a determinados valores;
- ✓ Opera com um **vetor** o qual associa um **símbolo** especificado no primeiro elemento a um **valor** especificado no segundo elemento;
- ✓ Os símbolos definidos no **let** **não** podem ser usados fora do let. Este comportamento é semelhante à variáveis **private** em outras linguagens;
- ✓ Avalia a expressão em um contexto léxico.

Macro - let

```
Clojure
user=>
user=>
user=> ( let [x 1] )
nil
user=>
user=>
user=> x
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: x in this context
user=>
user=>
user=>
user=>
```

Macro - let

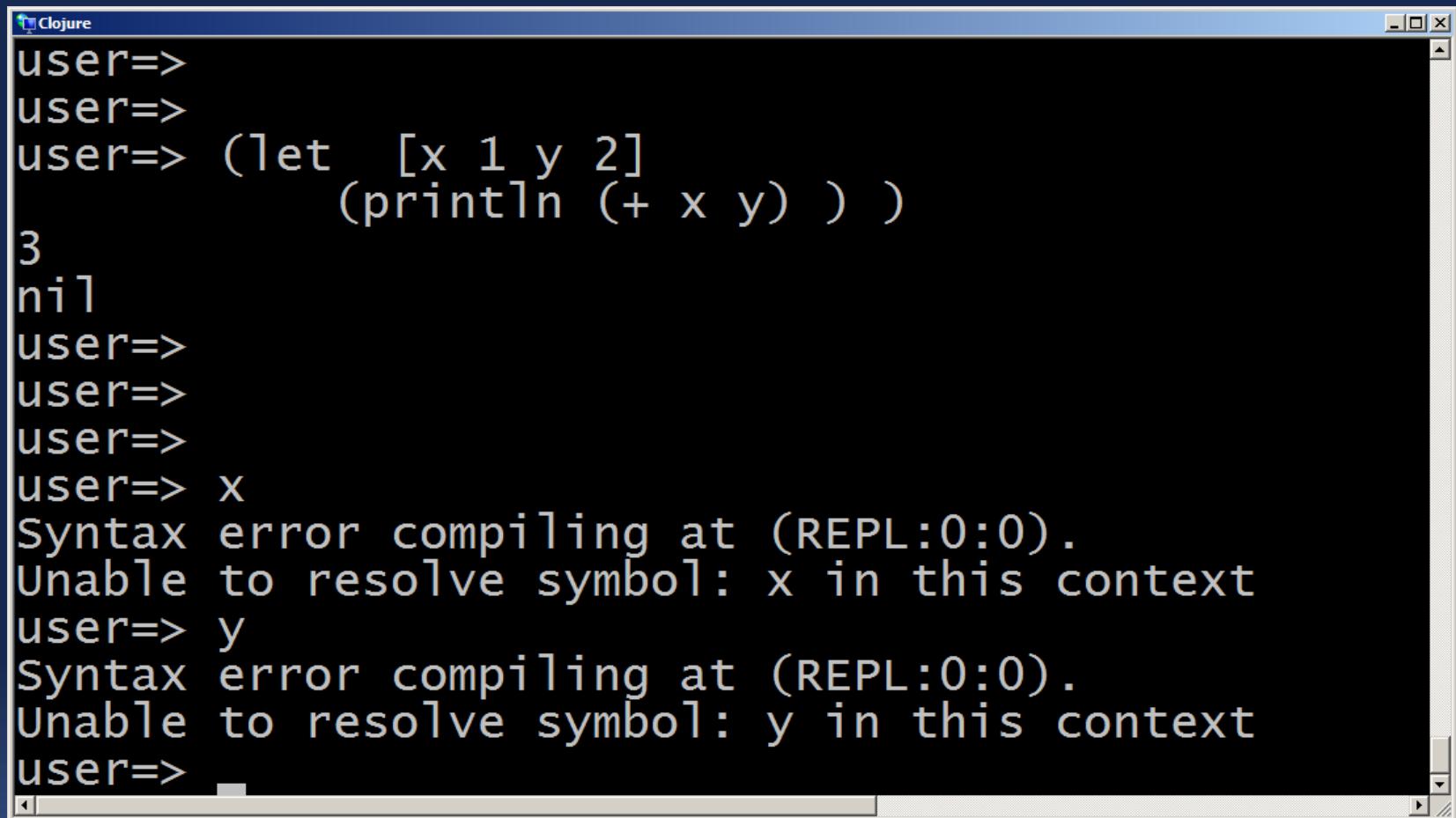


The screenshot shows a Clojure REPL window with the title 'Clojure'. The user has entered the following code:

```
user=>
user=>
user=>
user=> ( let [x 1]
           (println x) )
1
nil
user=>
user=>
user=>
user=> x
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: x in this context
user=>
user=>
```

The final line 'Syntax error compiling at (REPL:0:0). Unable to resolve symbol: x in this context' indicates that the symbol 'x' is not defined in the current lexical environment when the macro expansion is attempted.

Macro - let



The screenshot shows a Windows-style window titled "Clojure". Inside, a Clojure REPL session is running. The user types a let macro:

```
user=>
user=>
user=> (let [x 1 y 2]
          (println (+ x y)) )
```

The REPL responds with the value 3 and nil, then asks for another input:

```
3
nil
user=>
```

When the user types "x", the REPL returns an error message:

```
user=> x
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: x in this context
```

When the user types "y", the REPL returns another error message:

```
user=> y
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: y in this context
```

Finally, the user types a carriage return, which is shown as a small white square in the REPL window.

Macro - let

```
Clojure
user=>
user=> (let [m "Maurício" a "Aparecido"] (println (str m " e " a) ) )
Maurício e Aparecido
nil
user=>
user=> m
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: m in this context
user=>
user=>
user=> a
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: a in this context
user=>
```

Macro - **fn**

- ✓ Funções são **objetos de primeira ordem em Clojure**; Isso significa que em **Clojure** pode-se fazer todas as operações básicas com funções;
- ✓ Ou seja, pode-se passar funções à outras funções, retornar funções de outras funções e ligar (**binding**) funções à símbolos (variáveis);
- ✓ A macro **fn** é usada para se criar **funções anônimas**;
- ✓ **#()** é o shortcut para **fn**;
- ✓ **%** será substituído pelos argumentos passados para a função. Quando houver múltiplos argumentos, **%1** para o primeiro argumento, **%2** para o segundo argumento e assim por diante.

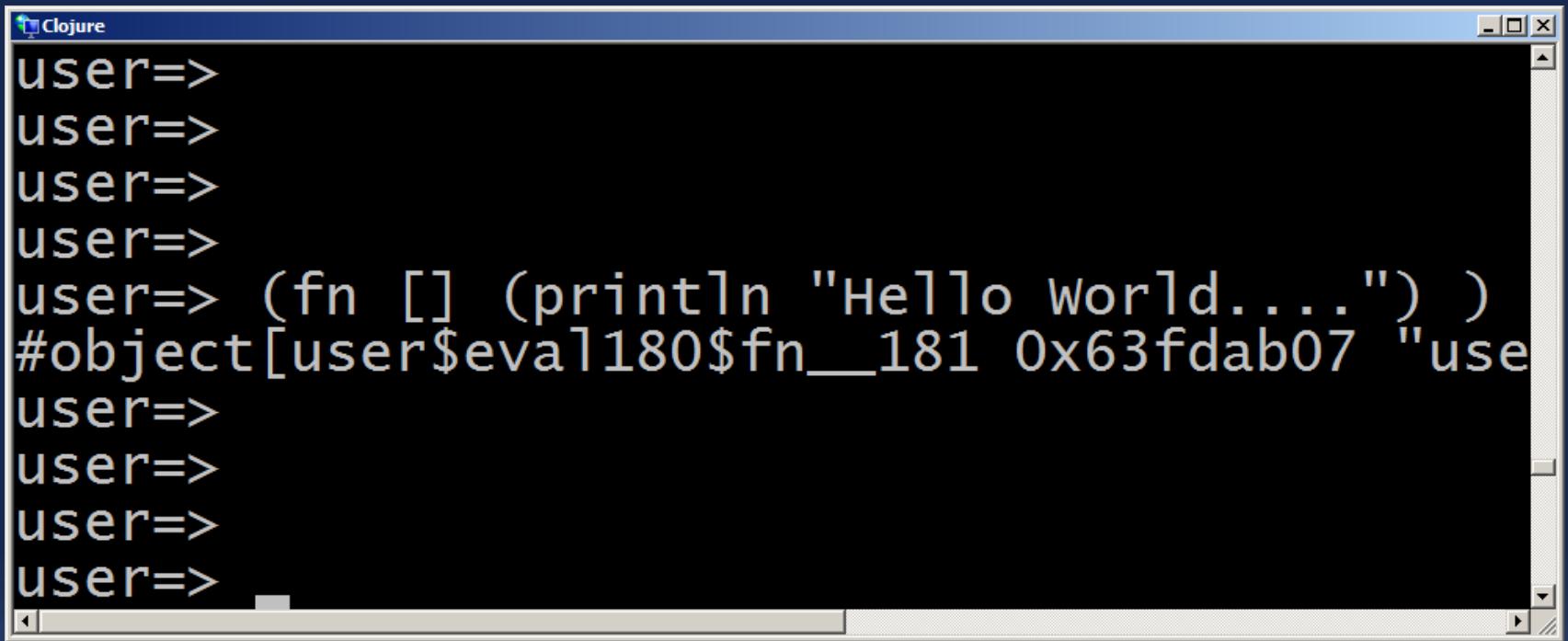
Macro - fn

```
Clojure
user=>
user=> (fn [x] (println x) )
#object[user$eval176$fn__177 0x1755e85b "user$eval176$fn__177@1755e85b"]
user=>
user=>
user=>
user=>
```

Macro - fn

```
Clojure
user=>
user=>
user=>
user=>
user=> (fn [x] (println x) )
#object[user$eval168$fn__169 0x18c5069b "user$eval168$fn__169@18c5069b"]
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> ( (fn [x] (println x) ) "Olá Turma do Quinto Ano...")
Olá Turma do Quinto Ano...
nil
user=>
user=>
user=>
user=>
```

Macro - fn



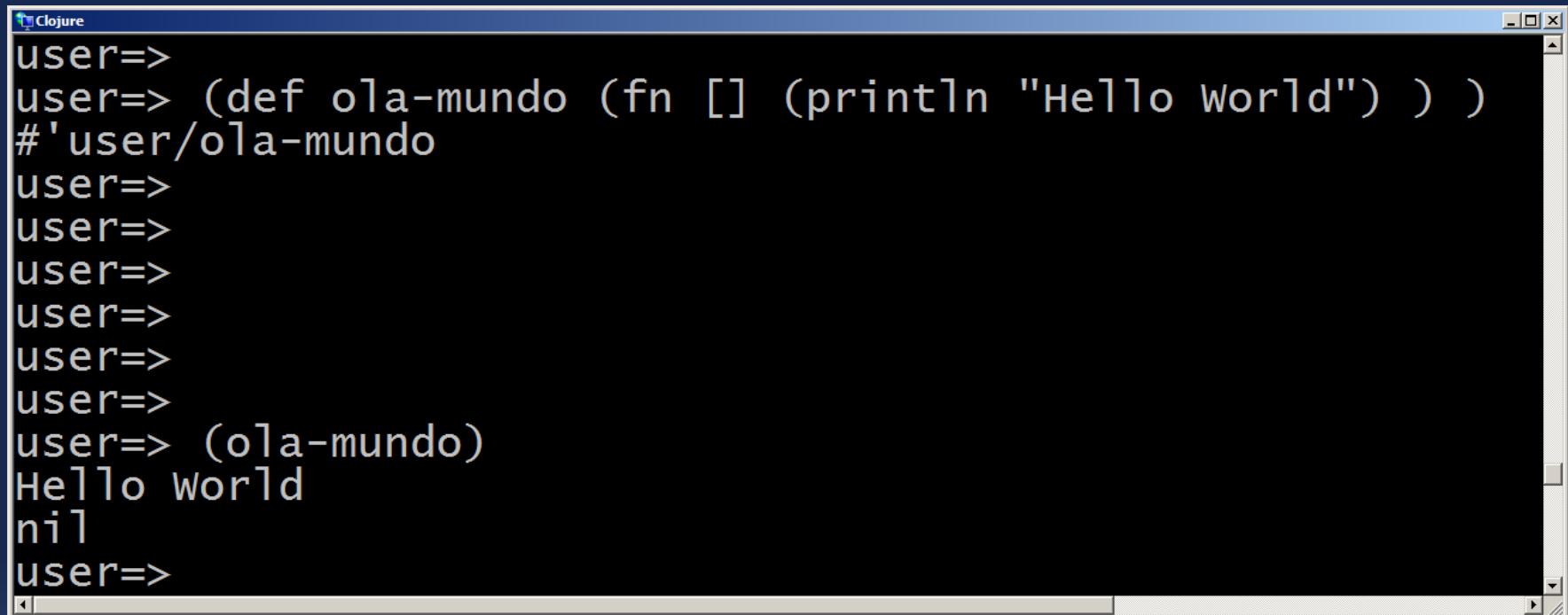
The screenshot shows a Windows-style window titled "Clojure". Inside, the Clojure REPL is running. The user has entered several forms, starting with "user=>". The final form is "(fn [] (println "Hello World...."))". Instead of evaluating this as a function, the REPL shows its internal representation as "#object[user\$eval180\$fn__181 0x63fdab07 "use". This demonstrates how Clojure's macro system expands the "fn" form into a more complex object structure.

```
user=>
user=>
user=>
user=>
user=> (fn [] (println "Hello World...."))
#object[user$eval180$fn__181 0x63fdab07 "use"
user=>
user=>
user=>
user=>
```

Macro - **fn**

```
Clojure
user=>
user=>
user=> (fn [] (println "Hello World....") )
#object[user$eval180$fn__181 0x63fdab07 "user$eval180$fn_"
user=>
user=>
user=>
user=> (def ola-mundo (fn [] (println "Hello world") ) )
#'user/ola-mundo
user=>
user=>
```

Macro - **fn**



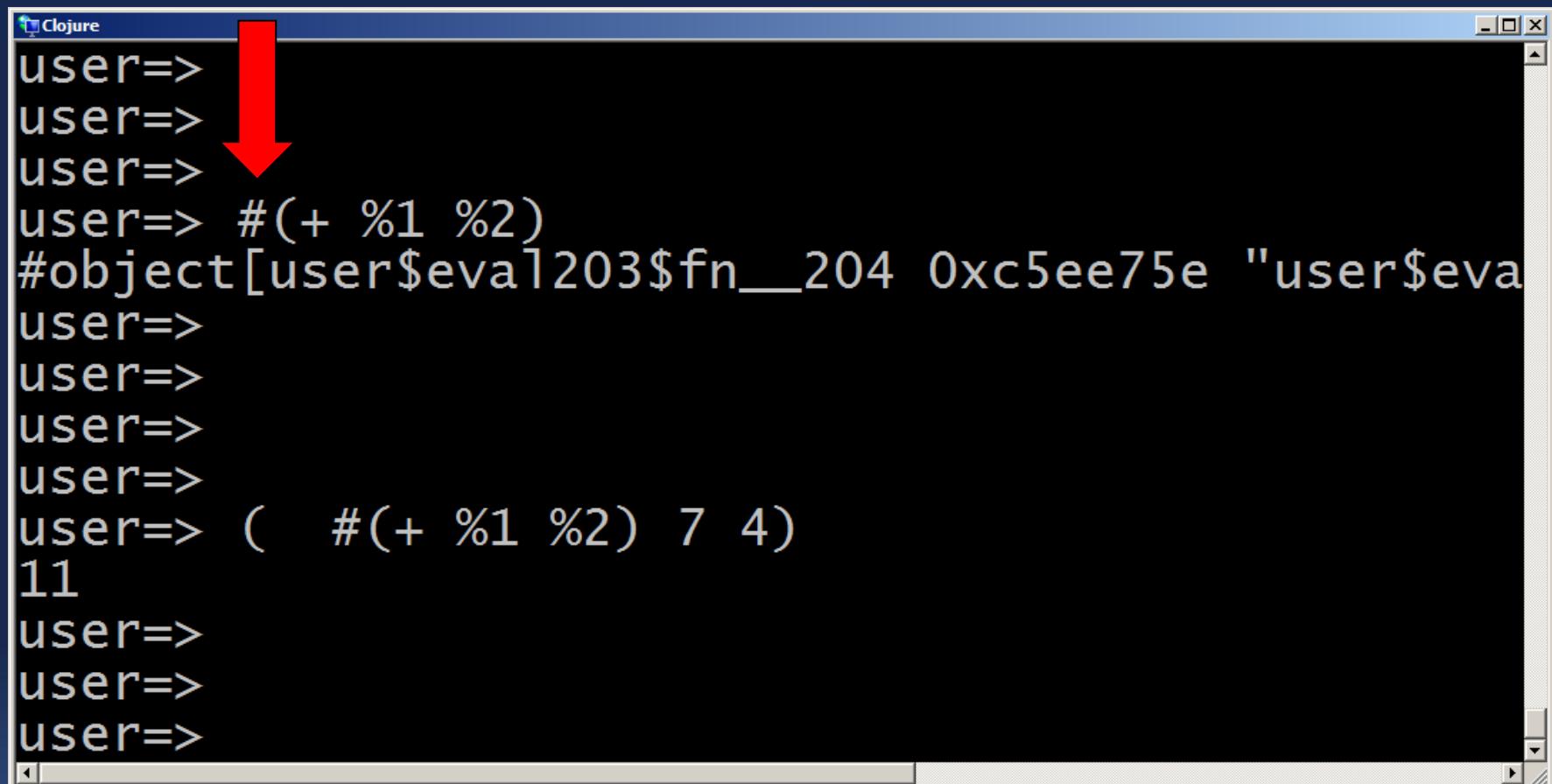
```
Clojure
user=>
user=> (def ola-mundo (fn [] (println "Hello World") ) )
#'user/ola-mundo
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (ola-mundo)
Hello World
nil
user=>
```

Macro - fn



```
Clojure
user=>
user=>
user=> #(println %)
#object[user$eval189$fn__190 0x1506f20f "user$eval189$fn__190"]
user=>
user=>
user=>
user=> ( #(println %) "OLá quinto ano...")
OLá quinto ano...
nil
user=>
user=>
user=>
```

Macro - fn

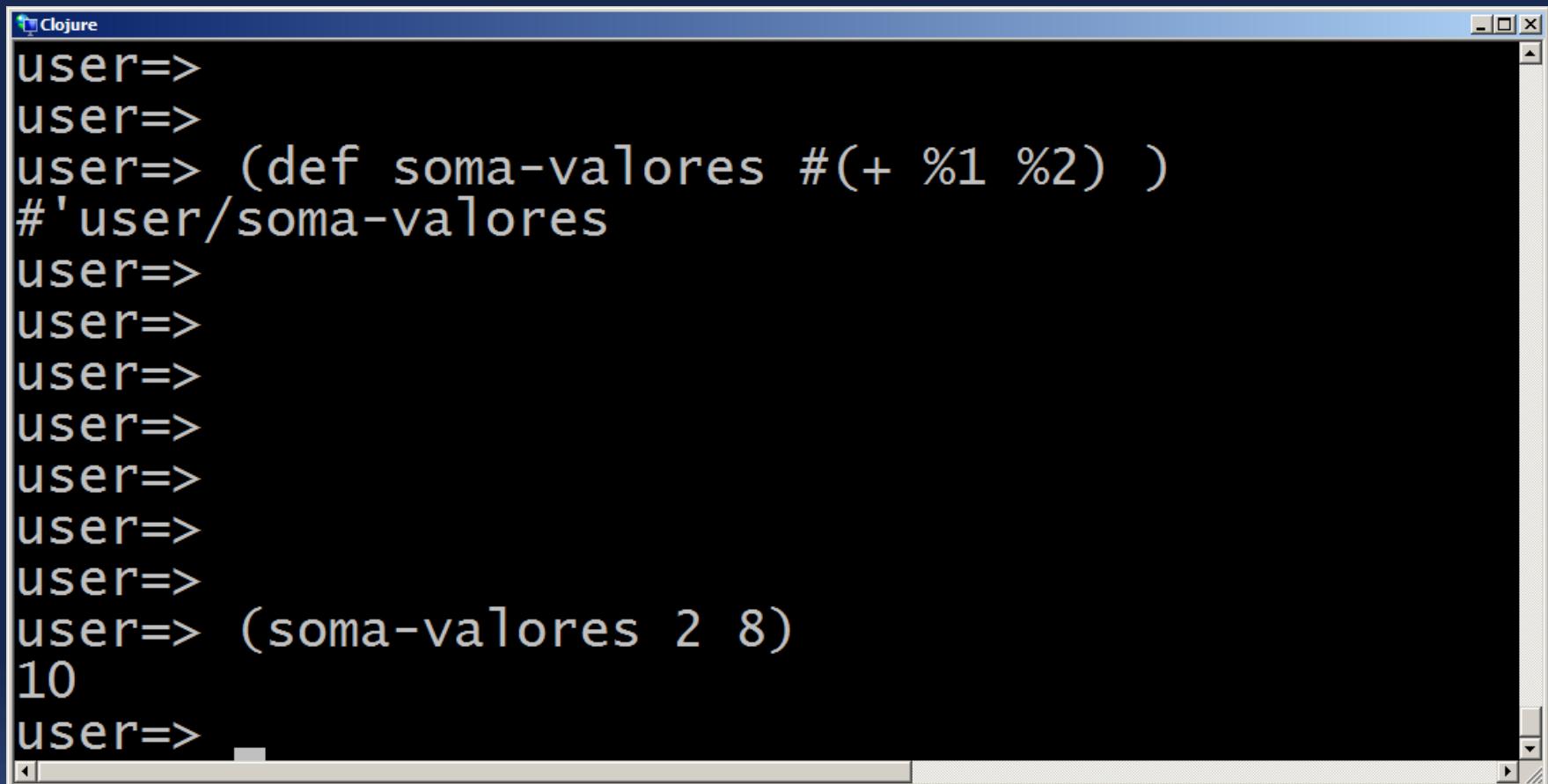


A screenshot of a Clojure REPL window titled "Clojure". The window shows the following interaction:

```
user=>
user=>
user=>
user=> #(+ %1 %2)
#object[user$eval1203$fn__204 0xc5ee75e "user$eval1203$fn__204"]
user=>
user=>
user=>
user=>
user=> (  #(+ %1 %2) 7 4)
11
user=>
user=>
user=>
```

A large red arrow points vertically downwards from the top of the first line of text ("user=>") to the second line of text ("user=> #(+ %1 %2)"). This visual cue highlights the point at which the macro expansion begins.

Macro - **fn**



The screenshot shows a Windows-style window titled "Clojure". Inside, a Clojure REPL session is running. The user has defined a macro named "soma-valores" which takes two arguments and adds them together. When the macro is called with arguments 2 and 8, it expands into the addition expression (+ 2 8), which then evaluates to 10.

```
Clojure
user=>
user=>
user=> (def soma-valores #(+ %1 %2) )
#'user/soma-valores
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (soma-valores 2 8)
10
user=>
```

Macro - fn

Passando funções à funções

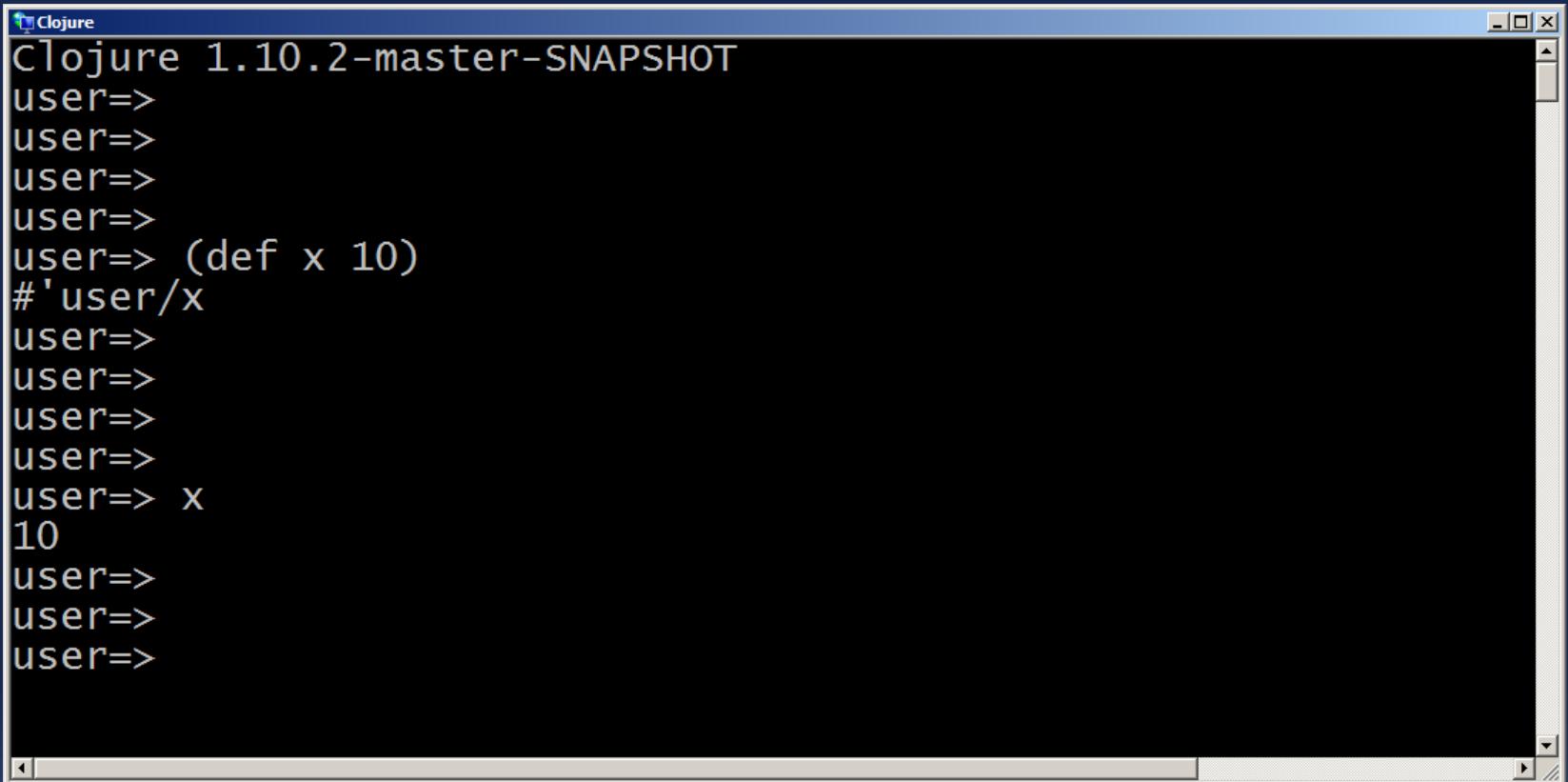
```
Clojure
user=>
user=>
user=> (def funcao-hello (fn [nome] (println (str "Hello, " nome) ) ) )
#'user/funcao-hello
user=>
user=>
user=>
user=> (def funcao-tchau (fn [nome] (println (str "Tchau, " nome) ) ) )
#'user/funcao-tchau
user=>
user=>
user=> (def saudacoes (fn [funcao nome] (funcao nome) ) )
#'user/saudacoes
user=>
user=>
user=> (saudacoes funcao-hello "Maurício")
Hello, Maurício
nil
user=>
user=> (saudacoes funcao-tchau "Maurício")
Tchau, Maurício
nil
user=>
user=>
```



Atividade 7

- ✓ Abra o **REPL**;
- ✓ Com o emprego da special form **def**, escreva o código que efetua a ligação do símbolo **x** ao valor 10;
- ✓ Em seguida, avalie o valor do símbolo **x**;
- ✓ Encerre o **REPL**.

Atividade 7



```
Clojure 1.10.2-master-SNAPSHOT
user=>
user=>
user=>
user=>
user=> (def x 10)
#'user/x
user=>
user=>
user=>
user=>
user=> x
10
user=>
user=>
user=>
```

- ✓ Observação: Quando **REPL** retorna **'#user/x**, ele está retornando uma referência ao símbolo **x** (variável) criado. A parte **user** indica o **namespace** onde o símbolo **x** foi definido

Macro - fn

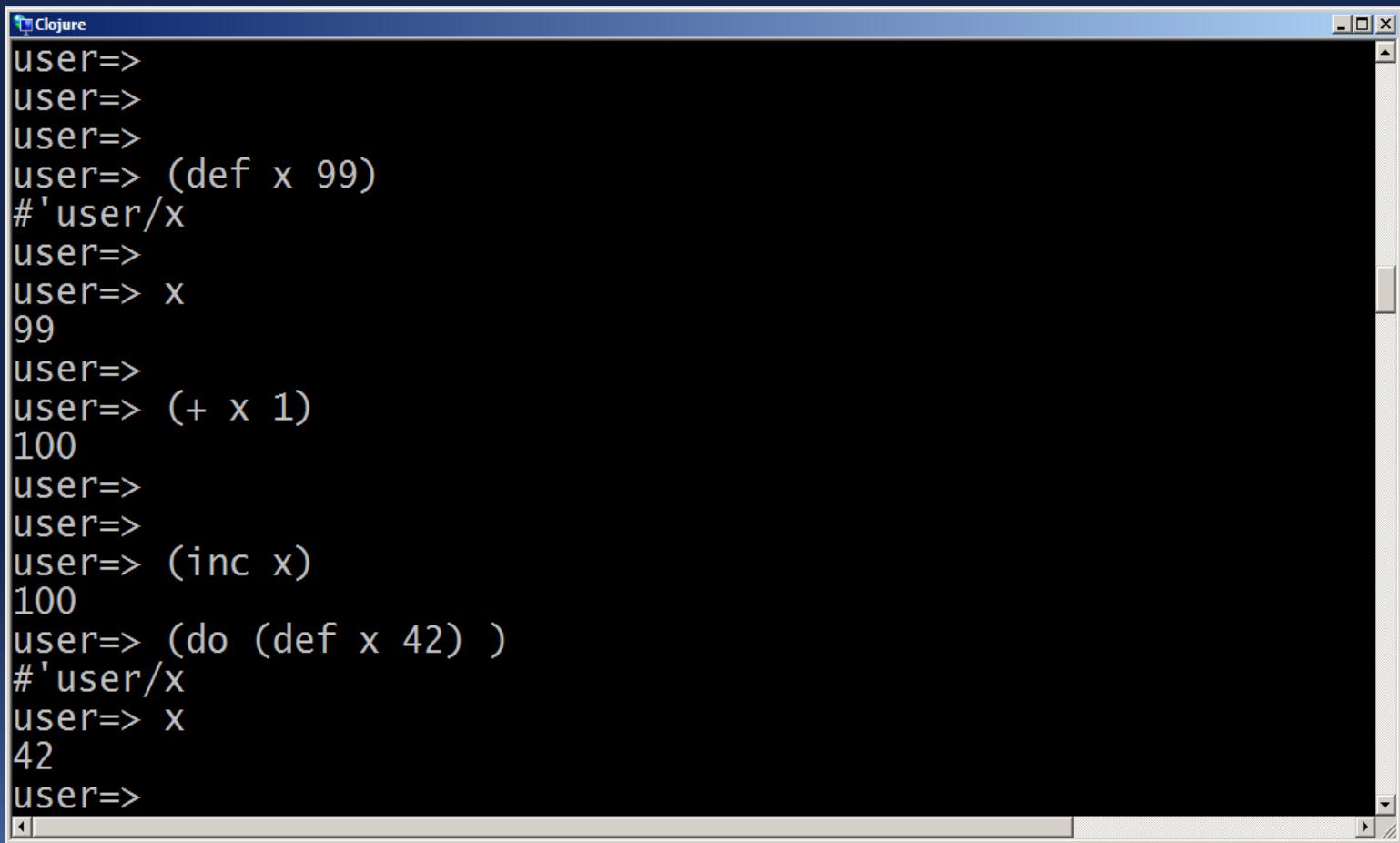


Binding

- ✓ É possível redefinir um símbolo criado previamente ?
- ✓ Sim, é possível modificar-se o binding por meio de: **(def x 20)**
- ✓ **Mas, não** se recomenda redefinir-se símbolos em nossos programas uma vez que tal procedimento dificulta a leitura e a manutenção do código;
- ✓ Assim, em **Clojure** uma boa prática é considerar a operação de binding como constante.

Binding

- ✓ Uma vez definido um símbolo por meio de **binding**, pode-se utilizá-lo em avaliação de expressões e redefiní-lo ainda por **binding**;



The screenshot shows a Windows-style application window titled "Clojure". Inside, the Clojure REPL is running. The session starts with several "user=>" prompts. The user then defines a variable "x" with the value 99, and subsequently evaluates the expression (+ x 1), resulting in 100. Later, the user increments "x" by one more, resulting in 100 again. Finally, the user redefines "x" to 42 and checks its value, which is now 42.

```
Clojure
user=>
user=>
user=>
user=> (def x 99)
#'user/x
user=>
user=> x
99
user=>
user=> (+ x 1)
100
user=>
user=>
user=> (inc x)
100
user=> (do (def x 42) )
#'user/x
user=> x
42
user=>
```

Binding e Escopo

- ✓ Bindings criados pela special form `def` têm escopo dinâmico e podem ser considerados como “globais”. Eles são, dessa forma, automaticamente namespaced, o qual é útil para evitar conflitos com nomes existentes;
- ✓ Para se definir bindings com escopo local (ou escopo léxico), deve-se usar a special form `let`.

y está
fora do
escopo
definido
em let !!!

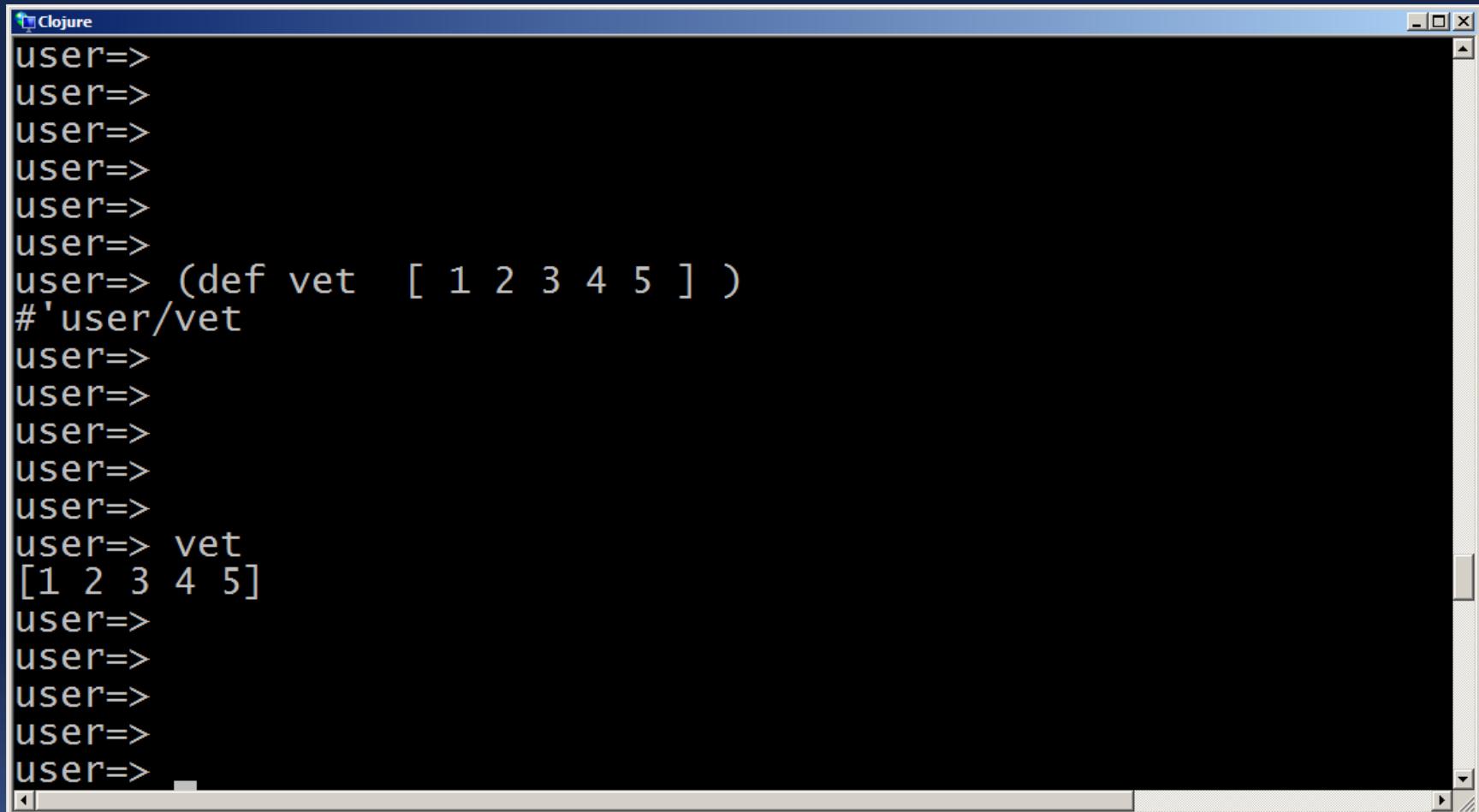


```
user=>
user=>
user=>
user=>
user=>
user=> (let [y 3] (println y) (* 10 y) )
3
30
user=>
user=>
user=>
user=>
user=>
user=> y
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: y in this context
user=>
user=>
user=>
user=>
```

Observação let

- ✓ A special form **let** emprega um vetor como um parâmetro para criar bindings locais e uma série de expressões que serão avaliadas como se estivessem em um bloco;
- ✓ Um **vetor** é similar a uma **lista**, no sentido em que ambos correspondem a uma coleção sequencial de valores;
- ✓ Na próxima unidade veremos com mais detalhes os vetores;
- ✓ Por ora, apenas é importante saber que **vetores** podem ser criados por **colchetes**;
- ✓ Por exemplo: [1 2 3 4 5]

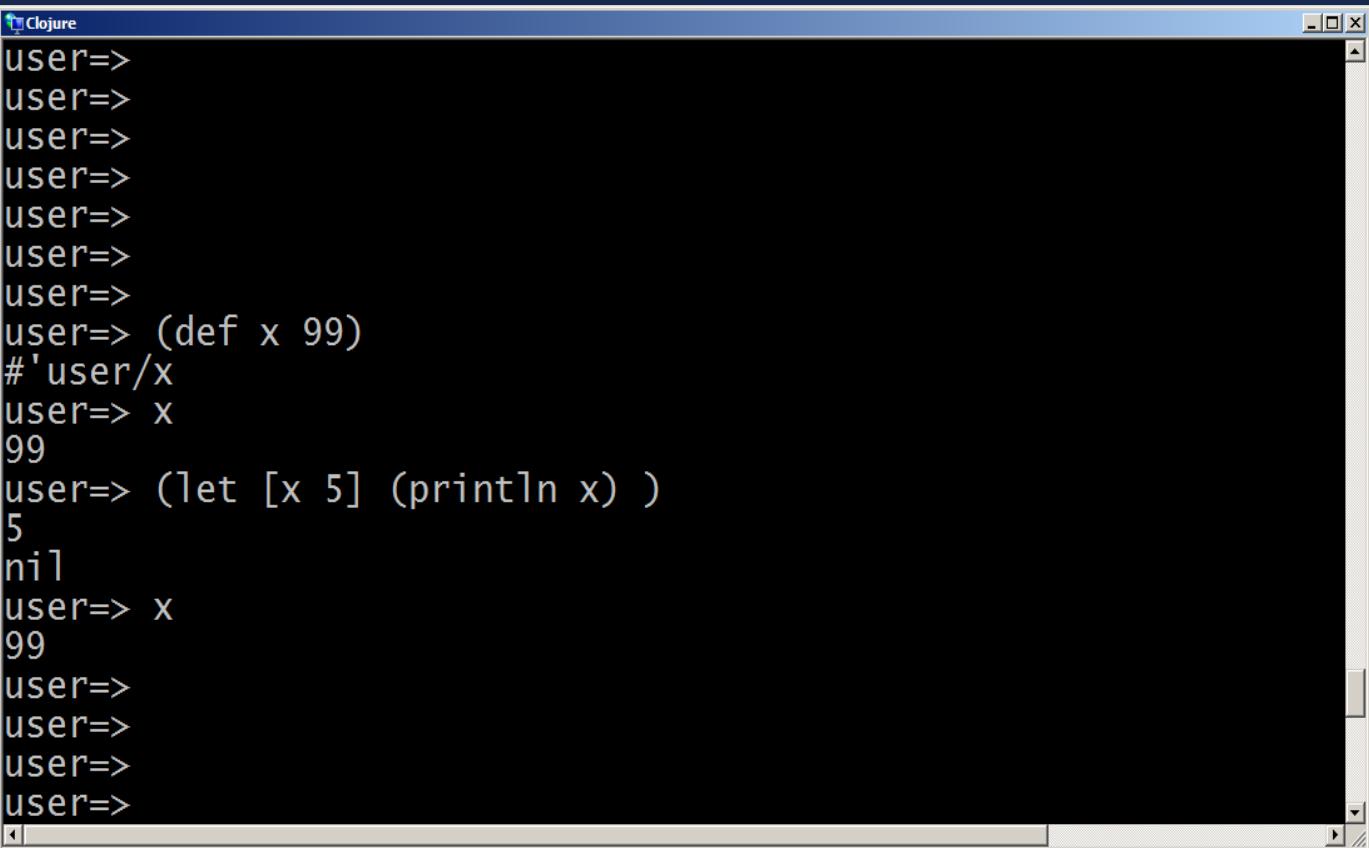
Definindo vetores



The screenshot shows a Clojure REPL window with the title "Clojure". The user has defined a vector named "vet" containing the integers 1 through 5. When the user then types "vet", the REPL returns the vector [1 2 3 4 5].

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> (def vet  [ 1 2 3 4 5 ] )
#'user/vet
user=>
user=>
user=>
user=>
user=>
user=>
user=> vet
[1 2 3 4 5]
user=>
user=>
user=>
user=>
user=>
```

Escopos Global e Local



The screenshot shows a Clojure REPL window with the following interaction:

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> (def x 99)
#'user/x
user=> x
99
user=> (let [x 5] (println x) )
5
nil
user=> x
99
user=>
user=>
user=>
user=>
```

Annotations with arrows point to specific parts of the code:

- A green arrow labeled "Escopo Global" points to the first two occurrences of "user=>".
- A red arrow labeled "Escopo Local" points to the line "#'user/x".
- A green arrow labeled "Escopo Global" points to the final four occurrences of "user=>".

Macro let – Escopo local

Escopo Local



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (let [x 10 y 99]  (str "x=" x " y=" y) )
"x=10 y=99"
user=>
user=>
user=>
user=>
user=> x
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: x in this context
user=>
user=>
user=>
user=> y
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: y in this context
user=>
user=>
```

Macro let – Escopo local

```
Clojure
user=>
user=>
user=>
user=>
user=> (def mensagem "Olá a todos")
#'user/mensagem
user=>
user=>
user=>
user=>
user=> (let [x (* 3 10) y 20 z 4] (println mensagem) (+ x y z))
Olá a todos
54
user=>
user=>
user=>
user=>
user=> x
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: x in this context
user=> y
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: y in this context
user=> z
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: z in this context
user=> -
```

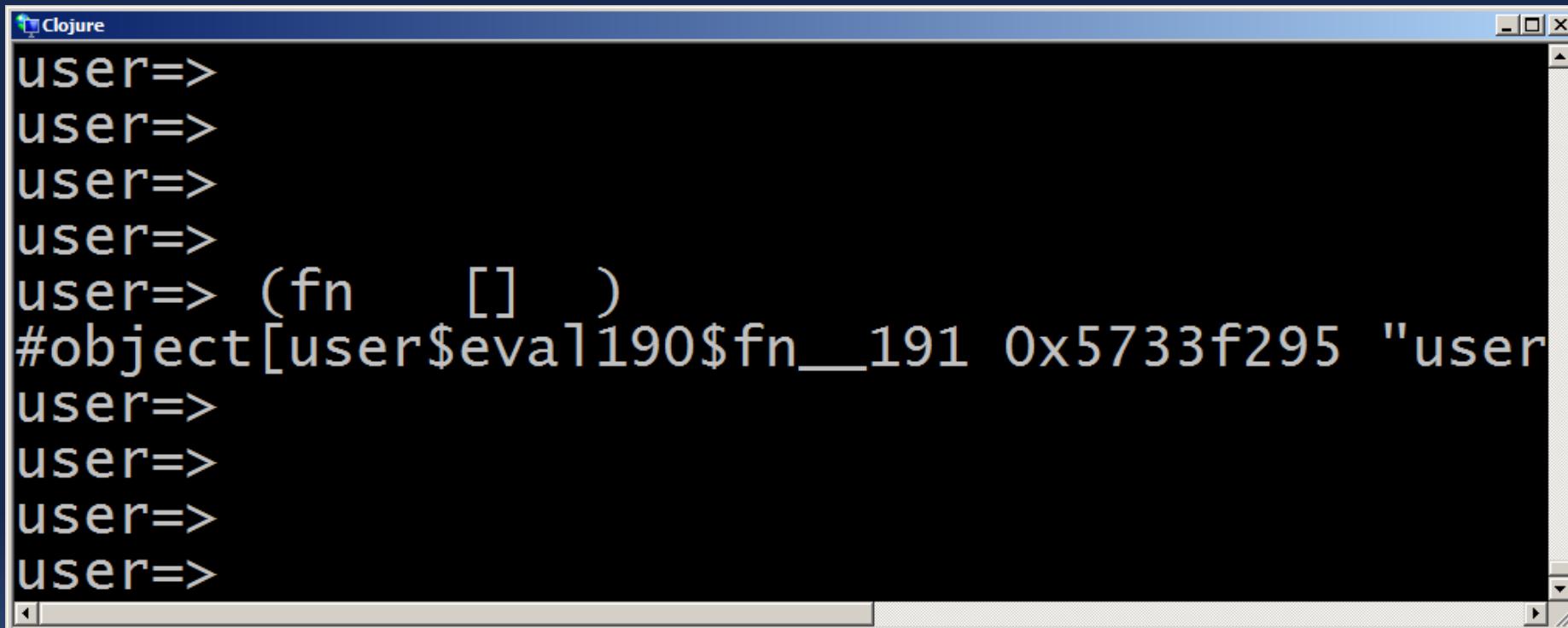
Atividade 8

- ✓ Abra o **REPL**;
- ✓ Com a **macro fn** defina uma função que **não** recebe argumentos e que também **nada** faz;
- ✓ Encerre o **REPL**.



Atividade 8

(fn [])



A screenshot of a Clojure REPL window titled "Clojure". The window shows the following text:
user=>
user=>
user=>
user=>
user=> (fn [])
#object[user\$eval1190\$fn__191 0x5733f295 "user
user=>
user=>
user=>
user=>

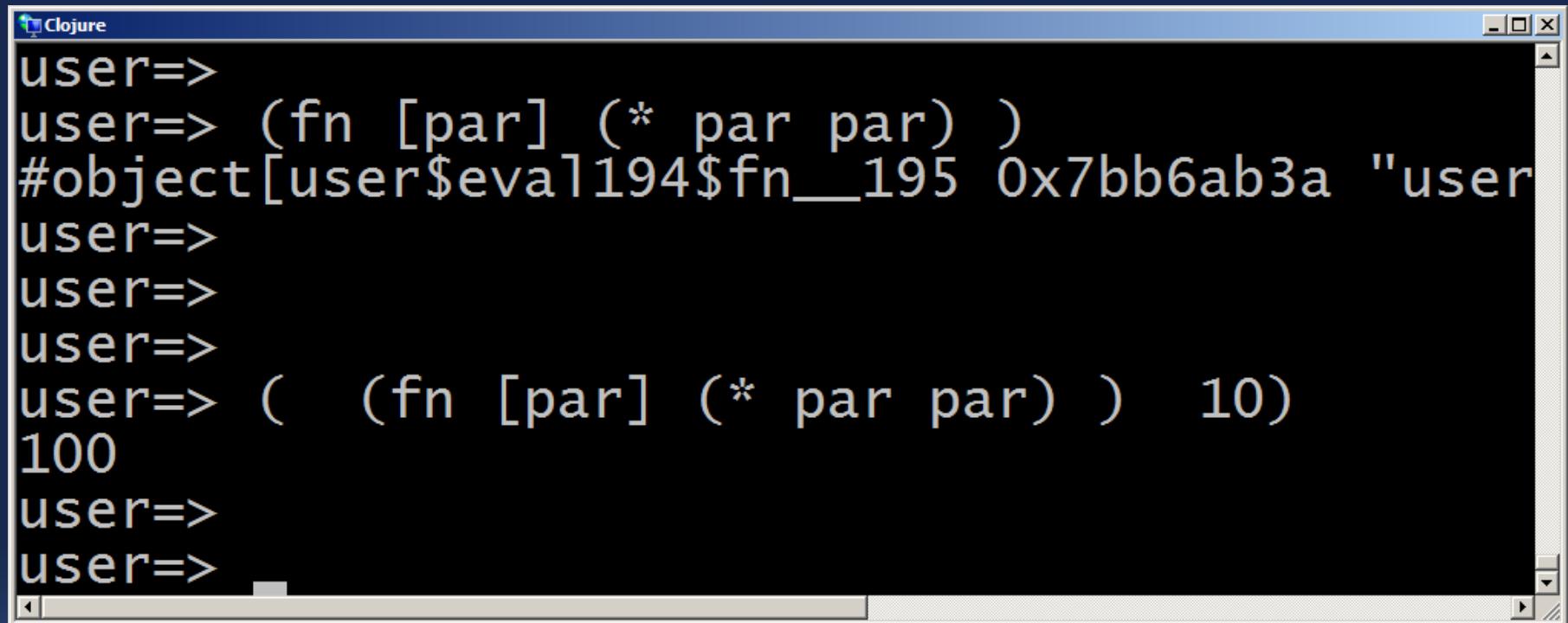
Atividade 9

- ✓ Abra o **REPL**;

- ✓ Com a **macro fn** defina uma função que recebe um parâmetro chamado **par** e retorna seu valor ao **quadrado** (**par** multiplicado por ele mesmo)

- ✓ Encerre o **REPL**.

Atividade 9



The screenshot shows a Clojure REPL window with the title "Clojure". The user has defined a function that takes a parameter and multiplies it by itself. This function is then used to calculate the square of 10.

```
user=>
user=> (fn [par] (* par par) )
#object[user$eval194$fn__195 0x7bb6ab3a "user"
user=>
user=>
user=>
user=> ( (fn [par] (* par par) ) 10)
100
user=>
user=>
```

Atividade 10

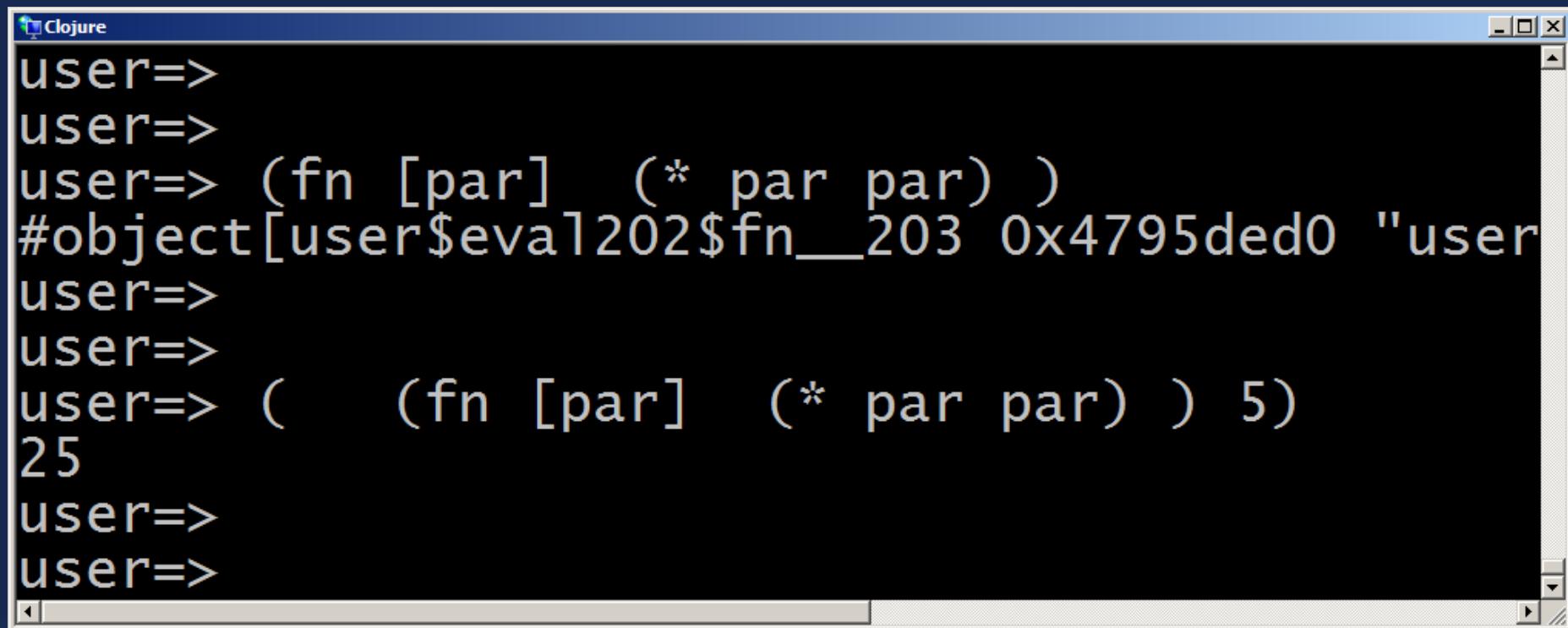
- ✓ Abra o **REPL**;

- ✓ Com a macro **fn** defina uma função que recebe um parâmetro chamado **par** e retorna seu valor ao quadrado (**par** multiplicado por ele mesmo).

- ✓ Após a definição da função anônima (com **fn**) efetuar a aplicação da função passando a ela um argumento com o valor 5.

- ✓ Encerre o **REPL**.

Atividade 10



The screenshot shows a Clojure REPL window with the title bar "Clojure". The console output is as follows:

```
user=>
user=>
user=> (fn [par] (* par par) )
#object[user$eval1202$fn__203 0x4795ded0 "user"
user=>
user=>
user=> ( (fn [par] (* par par) ) 5)
25
user=>
user=>
```

Observação: Nessa atividade, definimos uma função anônima e junto com a definição fizemos uma aplicação da função (execução) passando a ela um argumento com o valor 5. A função anônima retornou o quadrado de 5 que é 25.

Atividade 11

- ✓ Abra o **REPL**;

- ✓ Com a macro **fn** desenvolvida na atividade anterior, foi possível definirmos uma função de forma anônima e executá-la por meio da aplicação da função com o argumento passado, no caso 5.

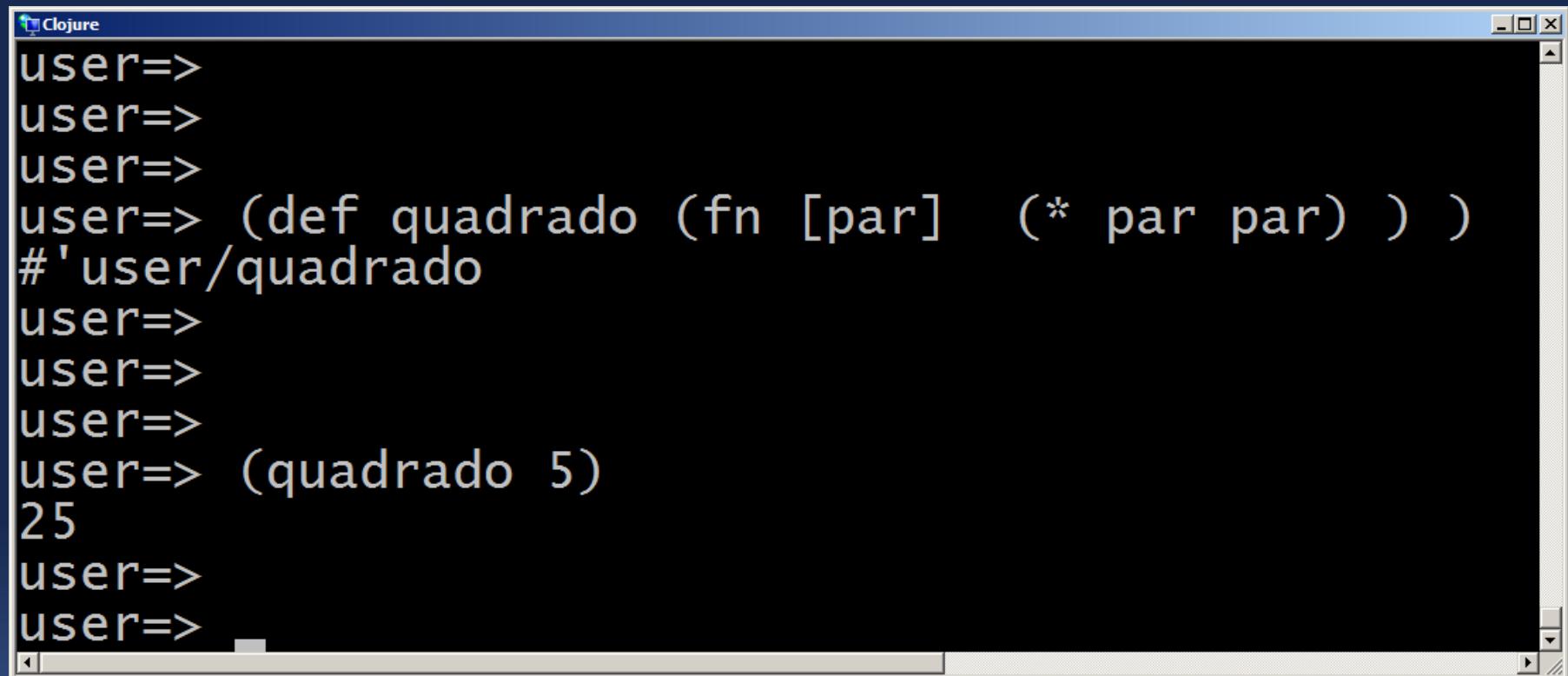
- ✓ Mas, e se quizermos reaproveitar essa função ? Ou seja, reusá-la em outra ocasião ! Para esse caso, precisaríamos associar à essa função um nome que permitisse chamá-la em outra ocasião. Repita a atividade de forma que a função deixe de ser anônima e passe assim a ter um nome.

- ✓ Encerre o **REPL**.

Dica: Nessa atividade, para definirmos função com nome devemos usar a macro **def**.



Atividade 11



A screenshot of a Clojure REPL window titled "Clojure". The window shows the following interaction:

```
user=>
user=>
user=>
user=> (def quadrado (fn [par] (* par par) ) )
#'user/quadrado
user=>
user=>
user=>
user=> (quadrado 5)
25
user=>
user=>
```

Macro defn

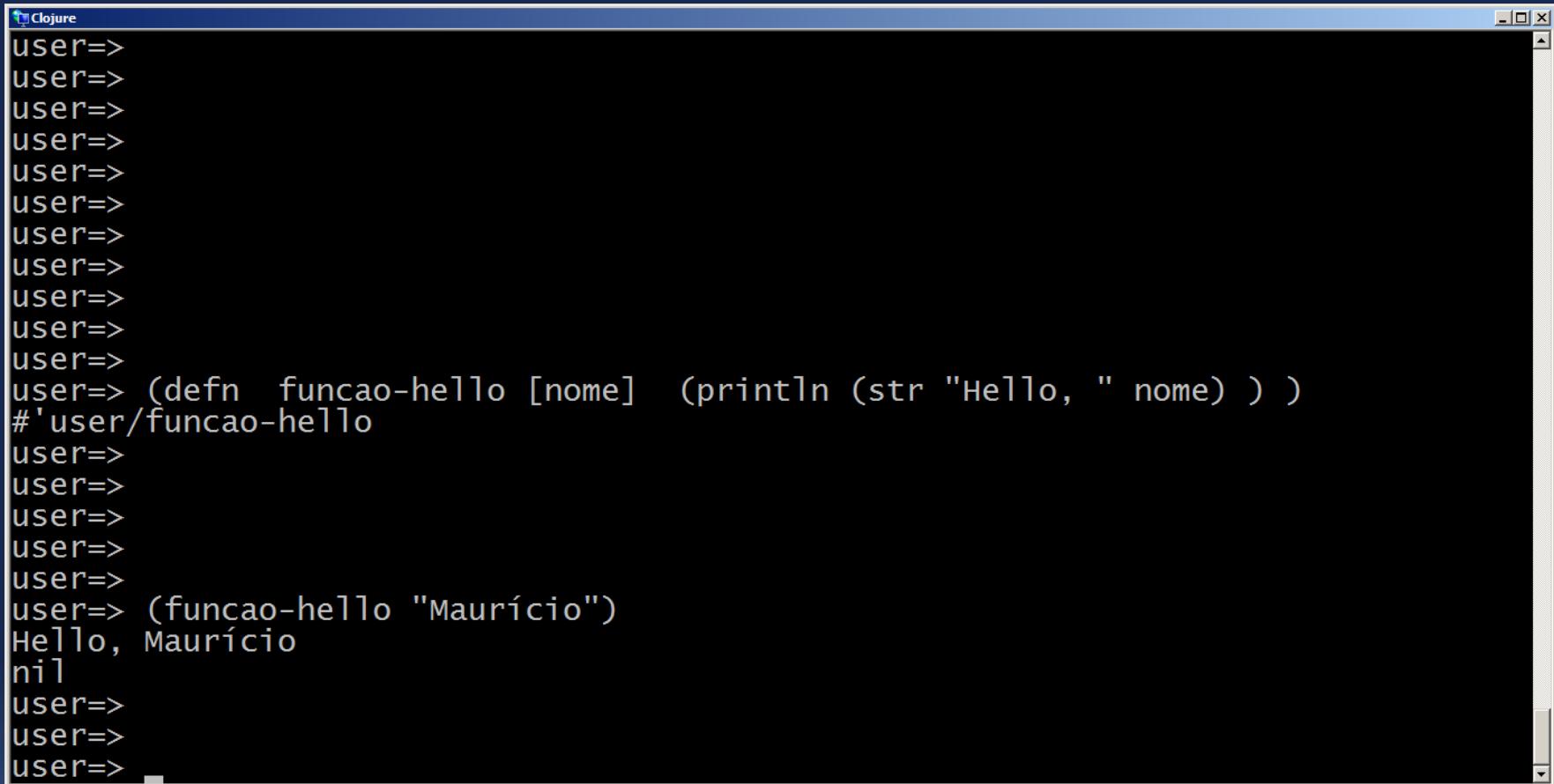
- ✓ Vimos nas atividades anteriores que com a special form **fn** podemos criar funções anônimas.
- ✓ Vimos também que podemos retirar da função essa característica anônima, ou seja deixá-la reusável por meio da special form **def**.
- ✓ Essa combinação de **def** com **fn** é portanto bem usual;
- ✓ Em função disso, **Clojure** disponibiliza uma macro **built-in** para essa finalidade.
- ✓ Trata-se da macro **defn**.



Exemplo - Macro defn

```
Clojure
user=>
user=> (defn quadrado [x] (* x x) )
#'user/quadrado
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (quadrado 5)
25
user=>
user=>
```

Exemplo - Macro defn

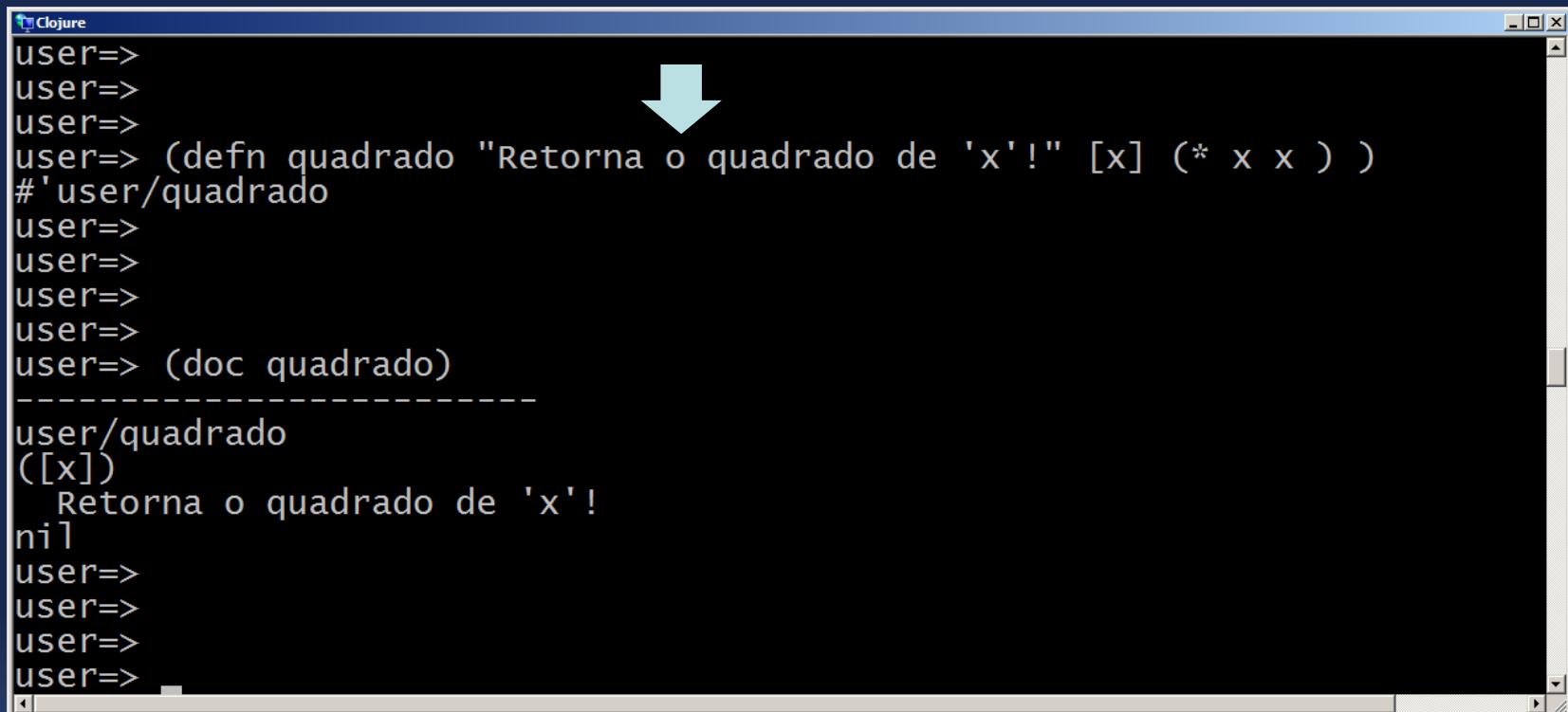


The screenshot shows a Windows-style window titled "Clojure". Inside, a Clojure REPL session is running. The user has defined a macro named `funcao-hello` which prints a greeting message. When the macro is called with the argument "Maurício", it correctly expands to print "Hello, Maurício". The REPL also shows the standard `nil` return value for macros.

```
Clojure
user=>
user=> (defn funcao-hello [nome] (println (str "Hello, " nome) ) )
#'user/funcao-hello
user=>
user=>
user=>
user=>
user=>
user=>
user=> (funcao-hello "Maurício")
Hello, Maurício
nil
user=>
user=>
user=>
```

Adicionando documentação à função

- ✓ Para adicionar alguma documentação em uma função definida com a macro **defn**, basta acrescentar um **doc-string** antes dos argumentos da função !



The screenshot shows a terminal window titled "Clojure" with the following text:

```
user=>
user=>
user=>
user=> (defn quadrado "Retorna o quadrado de 'x'!" [x] (* x x) )
#'user/quadrado
user=>
user=>
user=>
user=>
user=> (doc quadrado)
-----
user/quadrado
([x])
  Retorna o quadrado de 'x'!
nil
user=>
user=>
user=>
user=>
```

A large blue arrow points from the text "Para adicionar alguma documentação em uma função definida com a macro **defn**, basta acrescentar um **doc-string** antes dos argumentos da função !" up towards the "(defn quadrado ...)" line.

Observação: O doc-string deve vir antes da definição dos parâmetros da função. Se for escrito após, será avaliado sequencialmente como parte do corpo da função, resultando naturalmente em erro!

Veracidade, Falsidade e nil

- ✓ Na linguagem **Clojure**, **nil** representa ausência de valor; Corresponde à **NULL** em outras linguagens de programação;
- ✓ Representar ausência de valor é útil, pois permitir saber que “algo” está faltando;
- ✓ Em **Clojure**, **nil** se comporta como “**false**” quando for avaliado em uma expressão booleana;
- ✓ Em **Clojure**, **false** e **nil** são os **únicos** valores que são tratados como “**falsidade**”. Tudo o mais é verdadeiro. Essa simples regra torna código **Clojure** mais robusto e confiável.

Veracidade, Falsidade e nil - Exemplos

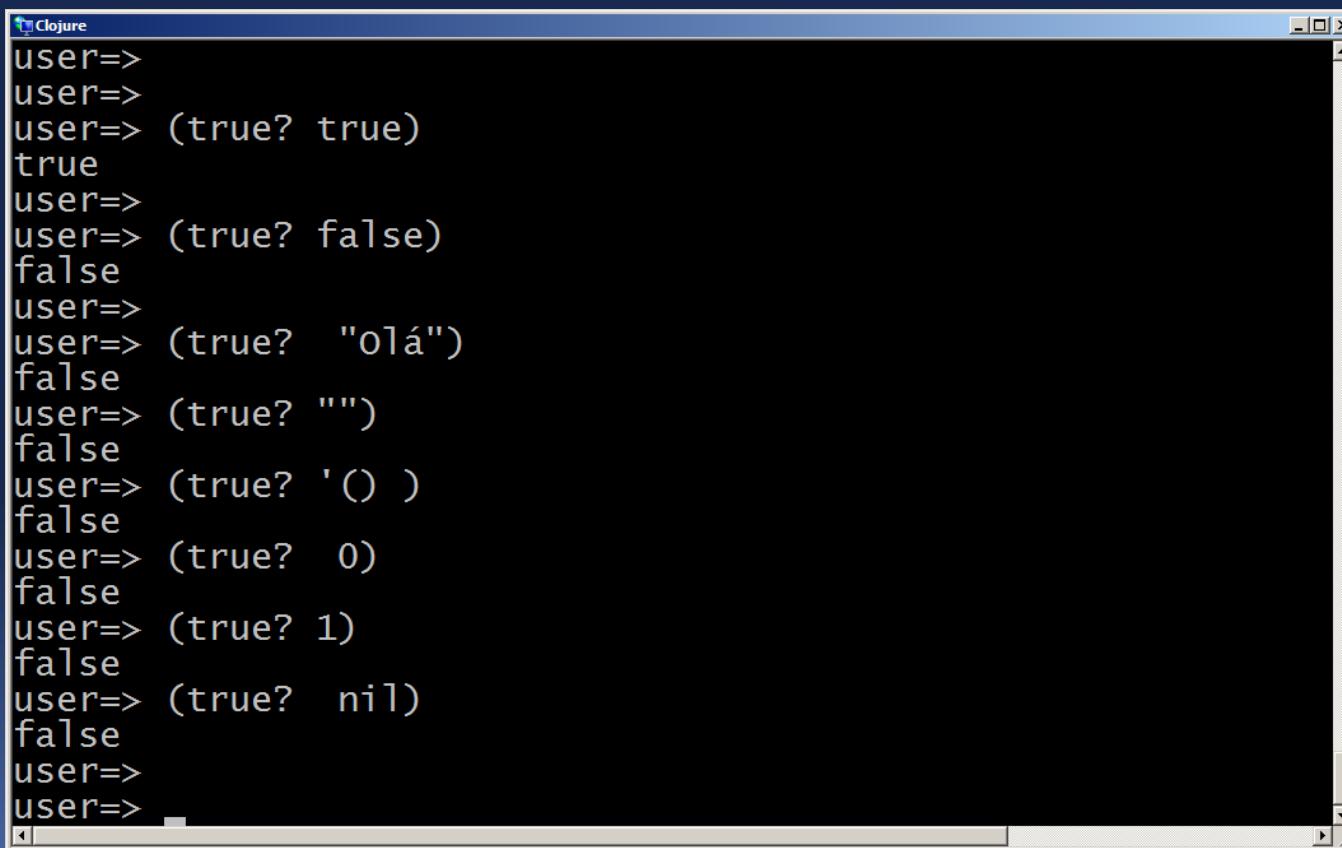
```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (if nil "Veracidade" "Falsidade")
"Falsidade"
user=>
user=> (if false "Veracidade" "Falsidade")
"Falsidade"
user=>
user=>
user=>
user=>
user=>
user=>
```

Veracidade, Falsidade e nil - Exemplos

```
Clojure
user=>
user=>
user=> (if 0 "verdade" "Falso")
"Verdade"
user=>
user=> (if -1 "verdade" "Falso")
"Verdade"
user=>
user=> (if '() "verdade" "Falso")
"Verdade"
user=>
user=>
user=> (if [] "verdade" "Falso")
"Verdade"
user=>
user=> (if "false" "Veracidade" "Falsidade")
"Veracidade"
user=>
user=> (if "" "Veracidade" "Falsidade")
"Veracidade"
user=>
user=>
```

Função true?

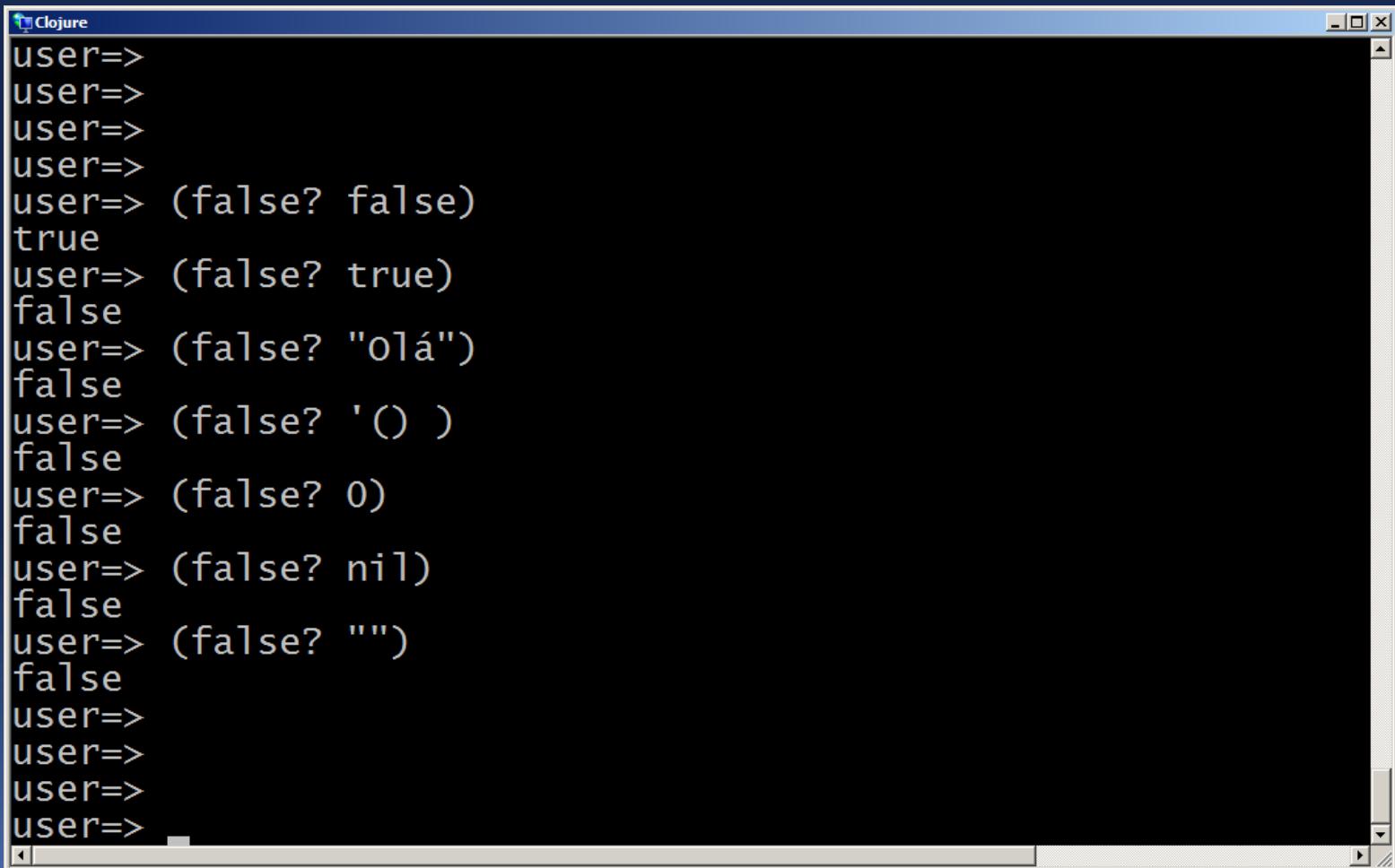
- ✓ A função **true?** retorna um valor booleano **true** ou **false** e permite checar se o parâmetro é **exatamente true** e não simplesmente tratado como **veracidade** ou **falsidade**.
- ✓ Assim, **true?** retorna **true** apenas quando o parâmetro for **verdadeiramente true**; Do contrário, sempre retornará **false**.



```
Clojure
user=>
user=>
user=> (true? true)
true
user=>
user=> (true? false)
false
user=>
user=> (true? "Olá")
false
user=> (true? "")
false
user=> (true? '())
false
user=> (true? 0)
false
user=> (true? 1)
false
user=> (true? nil)
false
user=>
user=>
```

Função false?

- ✓ A função **false?** retorna um valor booleano **true** ou **false** e permite checar se o parâmetro é **exatamente false** e não simplesmente tratado como **veracidade** ou **falsidade**.
- ✓ Assim, **false?** retorna true apenas quando o parâmetro for verdadeiramente **false**; Do contrário, sempre retornará **false**.

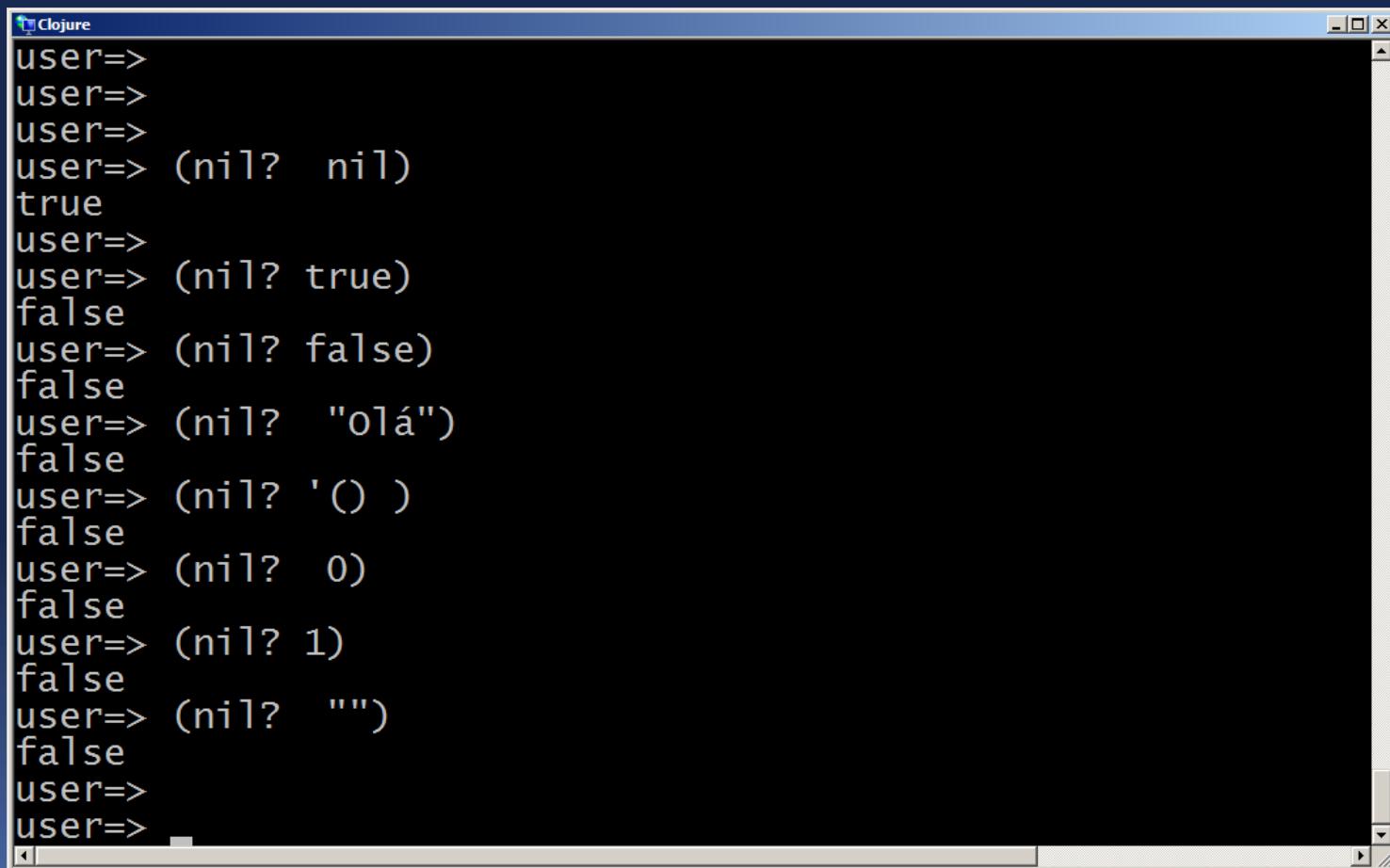


The screenshot shows a terminal window titled "Clojure" running on a Mac OS X desktop. The window contains a series of Clojure code evaluations. Each line starts with "user=>" followed by a call to the "false?" function with a different argument. The arguments include various values: false, true, a string ("olá"), an empty list '()', the number 0, nil, and an empty string ''. The "false?" function returns true for false, true, and nil, and false for the other three arguments.

```
Clojure
user=>
user=>
user=>
user=>
user=> (false? false)
true
user=> (false? true)
false
user=> (false? "olá")
false
user=> (false? '() )
false
user=> (false? 0)
false
user=> (false? nil)
false
user=> (false? "")
```

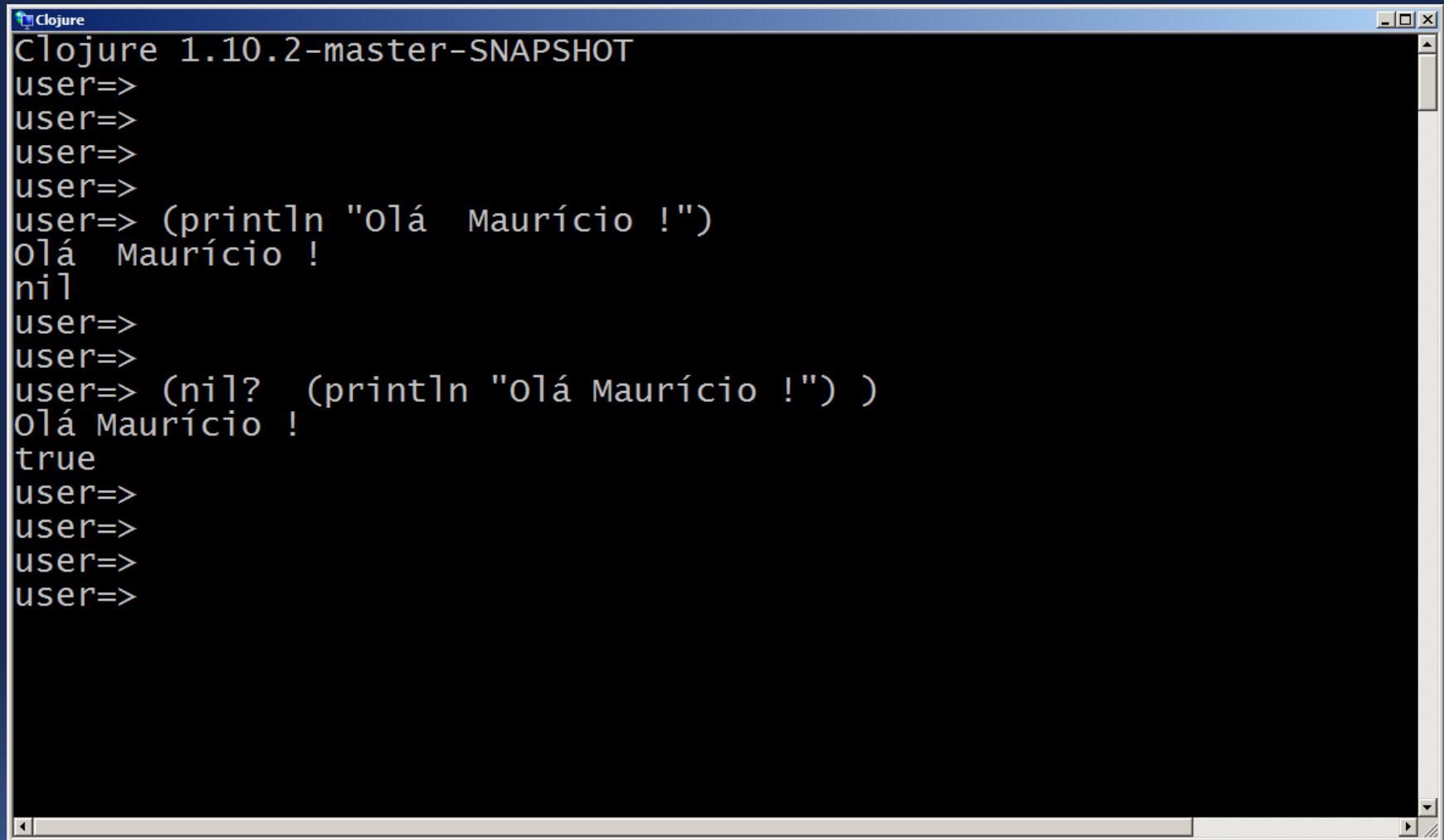
Função nil?

- ✓ A função **nil?** retorna um valor booleano **true** ou **false** e permite checar se o parâmetro é **exatamente nil** e não simplesmente tratado como **veracidade** ou **falsidade**.
- ✓ Assim, **nil?** retorna true apenas quando o parâmetro for verdadeiramente **nil**; Do contrário, sempre retornará **false**.



```
Clojure
user=>
user=>
user=>
user=> (nil? nil)
true
user=>
user=> (nil? true)
false
user=> (nil? false)
false
user=> (nil? "olá")
false
user=> (nil? '())
false
user=> (nil? 0)
false
user=> (nil? 1)
false
user=> (nil? "")
false
user=>
user=>
```

Exemplo interessante

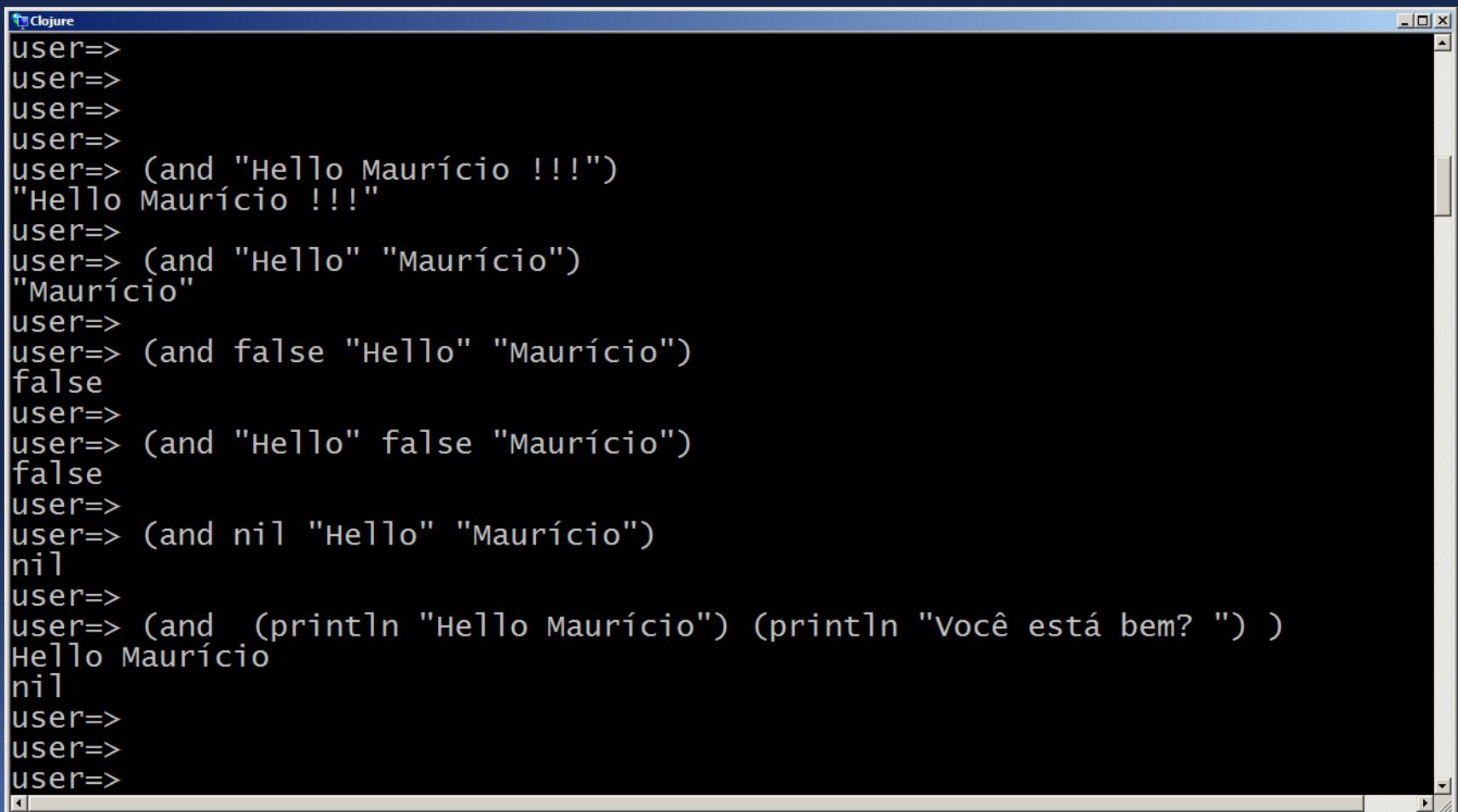


Clojure 1.10.2-master-SNAPSHOT

```
user=>
user=>
user=>
user=>
user=> (println "Olá Maurício !")
Olá Maurício !
nil
user=>
user=>
user=> (nil? (println "Olá Maurício !"))
Olá Maurício !
true
user=>
user=>
user=>
user=>
```

Macro and

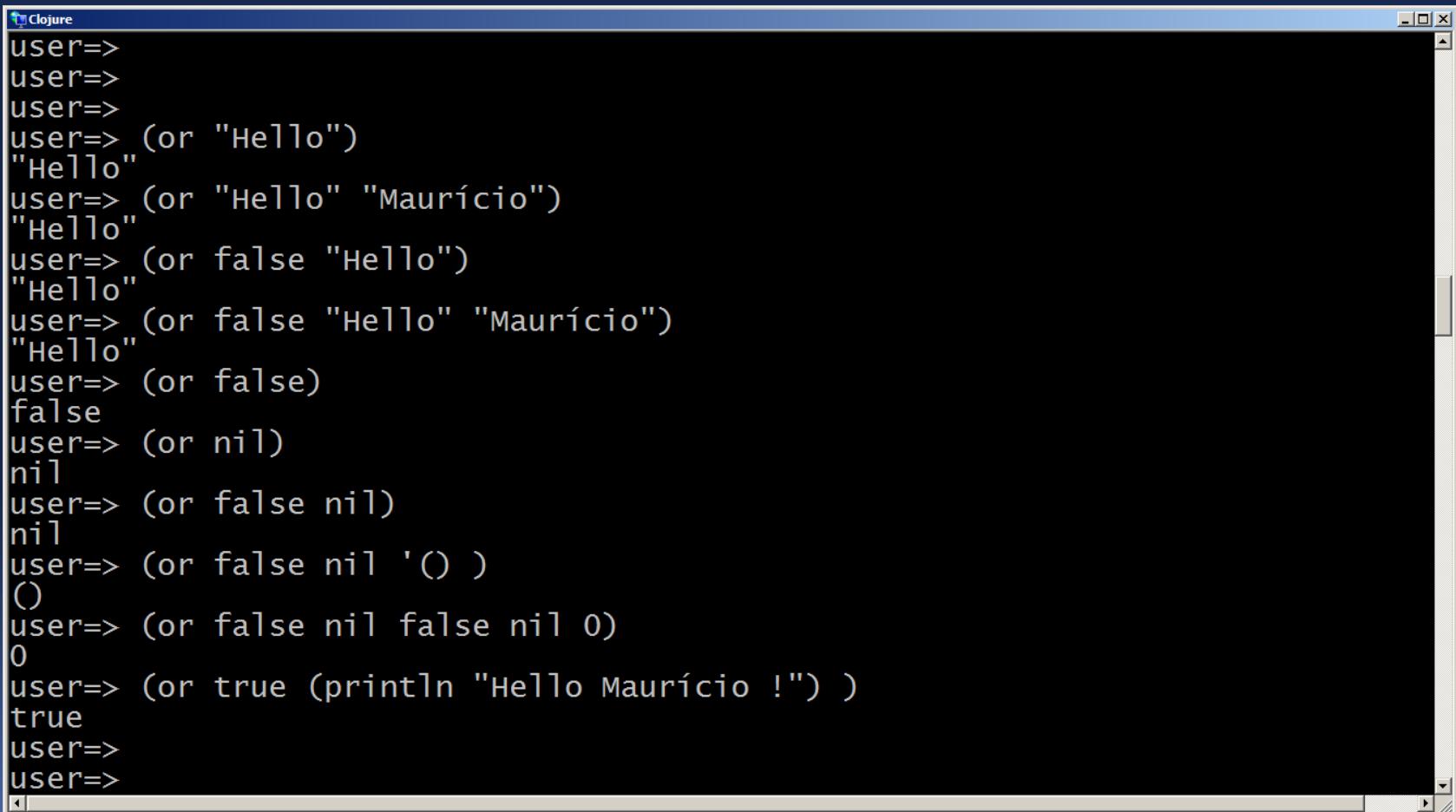
- ✓ A macro **and** retorna a primeira **falsidade** que encontrar (da esquerda para a direita) e **não** avaliará o restante da expressão quando for o caso;
- ✓ Quando todos os valores passados para a macro **and** resultarem em "veracidade", a macro **and** retorna o valor da última expressão.



```
Clojure
user=>
user=>
user=>
user=>
user=> (and "Hello Maurício !!!")
"Hello Maurício !!!"
user=>
user=> (and "Hello" "Maurício")
"Maurício"
user=>
user=> (and false "Hello" "Maurício")
false
user=>
user=> (and "Hello" false "Maurício")
false
user=>
user=> (and nil "Hello" "Maurício")
nil
user=>
user=> (and (println "Hello Maurício") (println "Você está bem? ") )
Hello Maurício
nil
user=>
user=>
user=>
```

Macro or

- ✓ A macro **or** retorna a primeira **veracidade** que encontrar (da esquerda para a direita) e **não** avaliará o restante da expressão quando for o caso;
- ✓ Quando todos os valores passados para a macro **or** resultarem em "falsidade", a macro **or** retorna false ou nil, dependendo do caso.

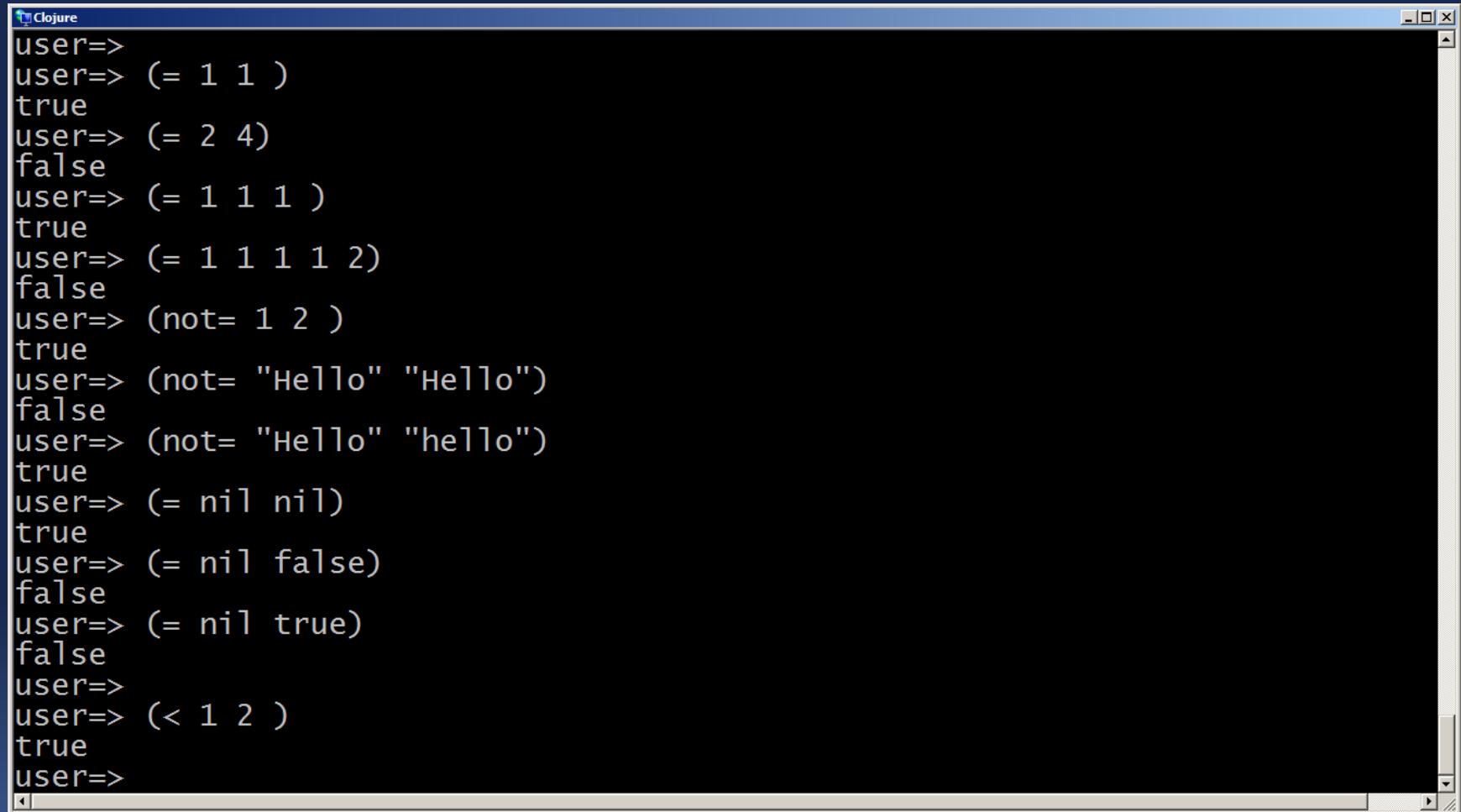


```
Clojure
user=>
user=>
user=>
user=> (or "Hello")
"Hello"
user=> (or "Hello" "Maurício")
"Hello"
user=> (or false "Hello")
"Hello"
user=> (or false "Hello" "Maurício")
"Hello"
user=> (or false)
false
user=> (or nil)
nil
user=> (or false nil)
nil
user=> (or false nil '())
()
user=> (or false nil false nil 0)
0
user=> (or true (println "Hello Maurício !"))
true
user=>
user=>
```

Operações de Equalidade

- ✓ Na maioria das linguagens de programação o símbolo `=` é usado para operações de atribuição. Porém em **Clojure**, bindings de símbolos à valores são feitos pelas special forms **def** e **defn**;
- ✓ Em **Clojure**, o símbolo `=` está associado à uma função para equalidade e retorna **true** se todos os argumentos forem **iguais**.
- ✓ Todos as outras operações de equalidade (`>` , `>=` , `<` , `<=`) são também implementações de funções.

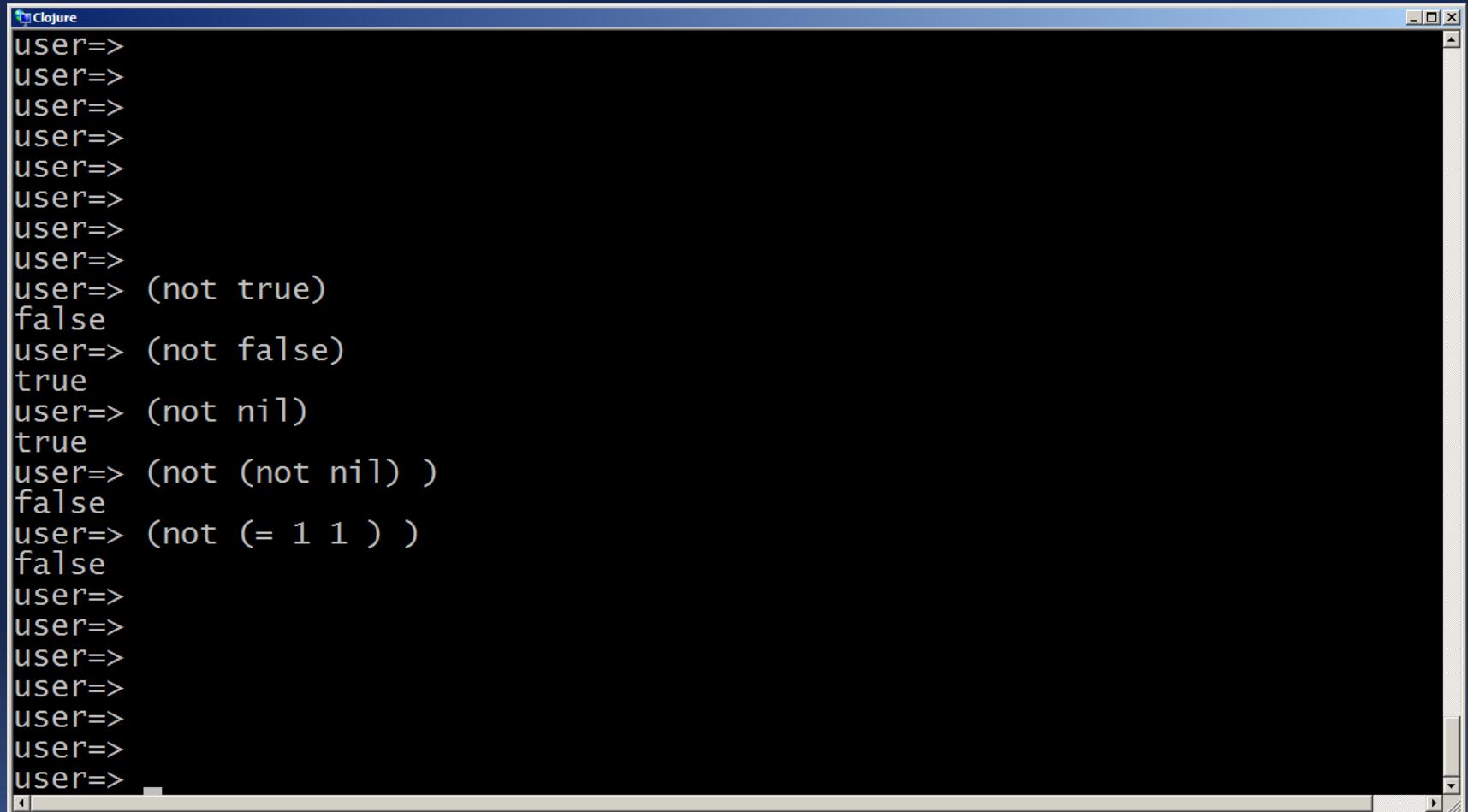
Operações de Equalidade



The screenshot shows a Clojure REPL window with the title bar "Clojure". The window contains the following text:

```
user=>
user=> (= 1 1 )
true
user=> (= 2 4)
false
user=> (= 1 1 1 )
true
user=> (= 1 1 1 1 2)
false
user=> (not= 1 2 )
true
user=> (not= "Hello" "Hello")
false
user=> (not= "Hello" "hello")
true
user=> (= nil nil)
true
user=> (= nil false)
false
user=> (= nil true)
false
user=>
user=> (< 1 2 )
true
user=>
```

Operações de Equalidade



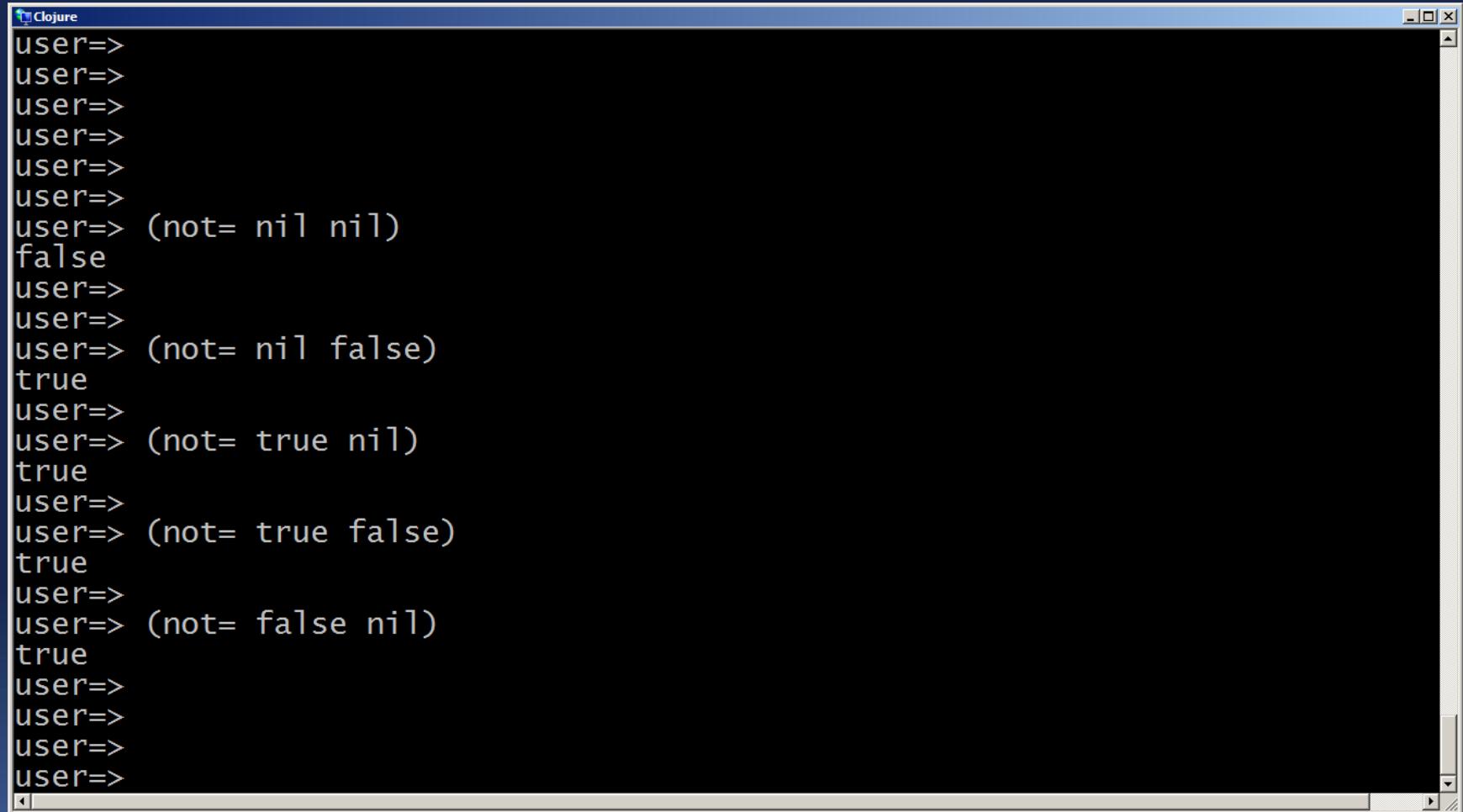
The screenshot shows a Clojure REPL window with the title "Clojure". The console output displays a series of equality checks between various Clojure values:

```
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (not true)
false
user=> (not false)
true
user=> (not nil)
true
user=> (not (not nil) )
false
user=> (not (= 1 1 ) )
false
user=>
user=>
user=>
user=>
user=>
user=>
```

Operações de Equalidade

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (not= 1 2)
true
user=>
user=>
user=> (not= "Hello" "hello")
true
user=>
user=>
user=> (not= (> 1 2) (not= 1 1) )
false
user=>
user=>
user=> (not= (> 1 2) (not= 3 6) )
true
user=>
user=>
user=>
user=>
```

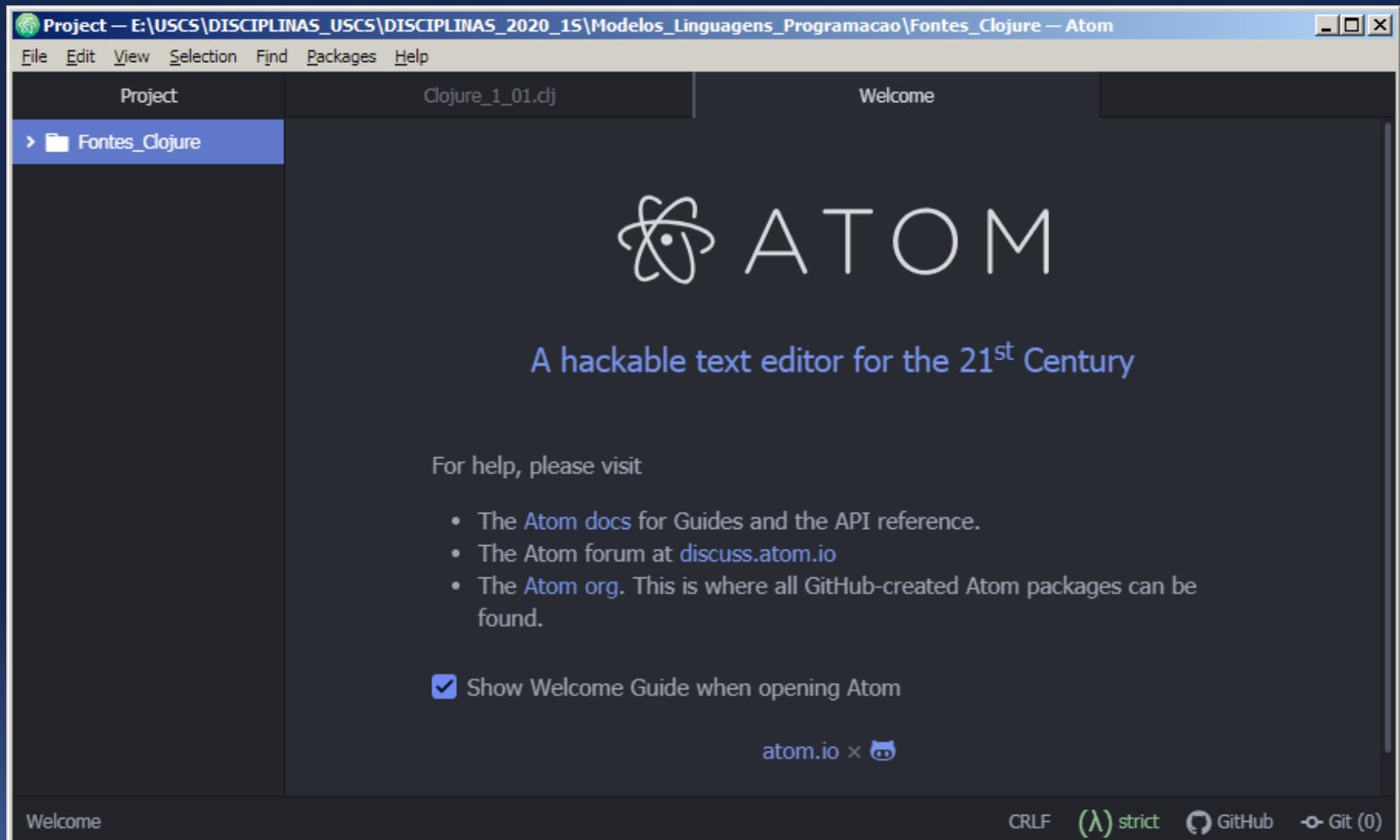
Operações de Equalidade



The screenshot shows a terminal window titled "Clojure" with the following text:

```
user=>
user=>
user=>
user=>
user=>
user=>
user=> (not= nil nil)
false
user=>
user=>
user=> (not= nil false)
true
user=>
user=> (not= true nil)
true
user=>
user=> (not= true false)
true
user=>
user=> (not= false nil)
true
user=>
user=>
user=>
user=>
```

Execução de Código Clojure no Atom Package Chlorine



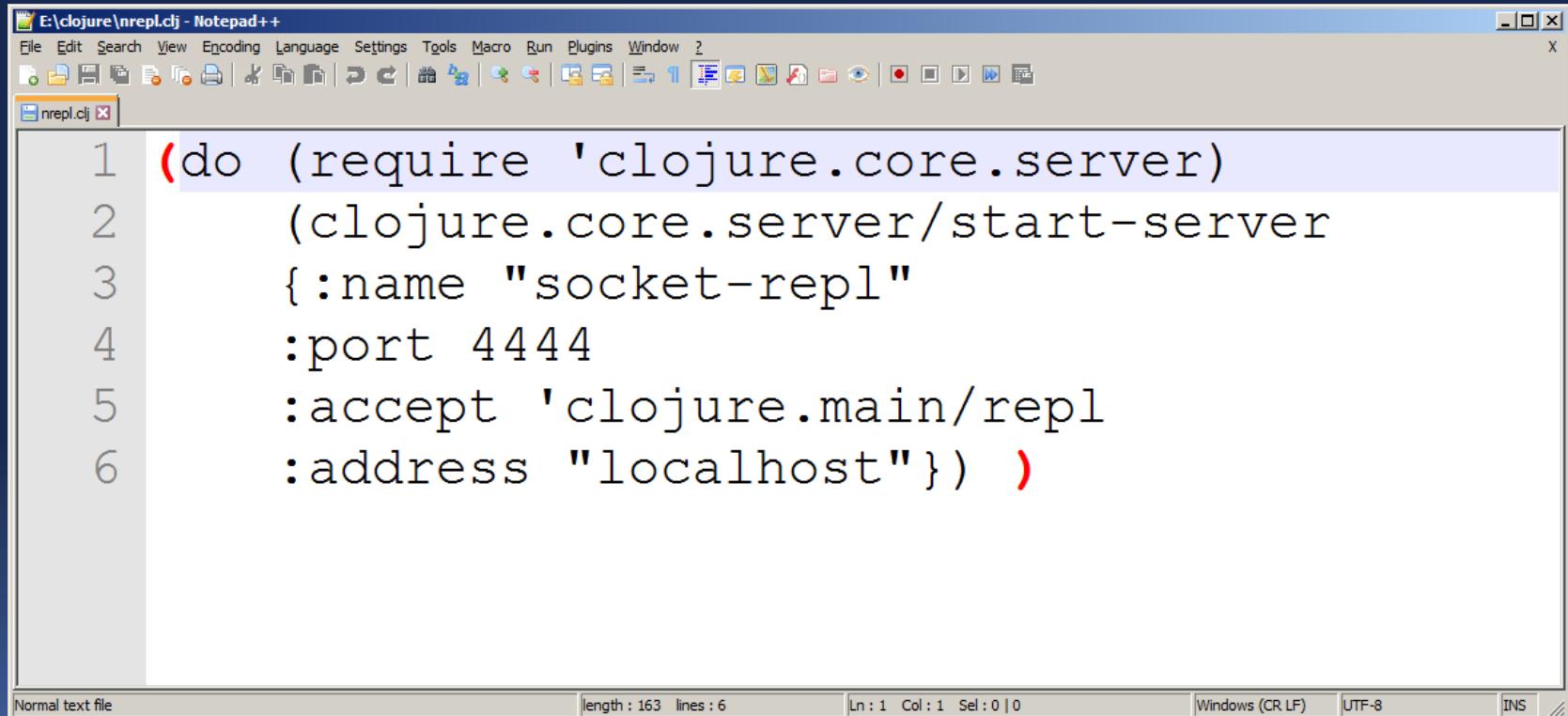
Ativando REPL Remoto

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



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 code in the editor is:

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

The code is a Clojure script that requires the 'clojure.core.server' namespace and starts a server named "socket-repl" on port 4444, accepting connections from 'clojure.main/repl' at the address "localhost". The line numbers 1 through 6 are visible on the left.



Ativando REPL Remoto

Digitando-se o código na console

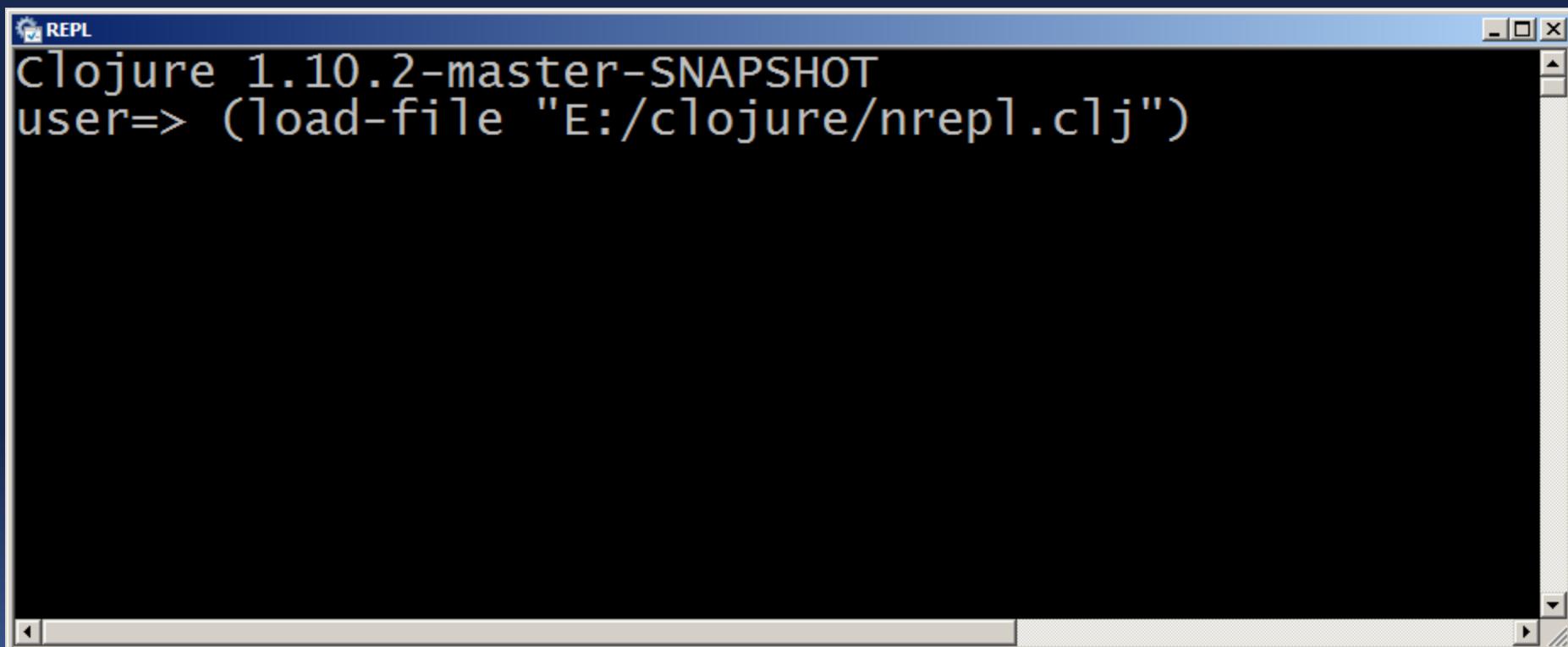
```
Clojure 1.10.2-master-SNAPSHOT
user=> (do (require 'clojure.core.server)
(clojure.core.server/start-server
{:name "socket-repl"
:port 4444
:accept 'clojure.main/repl
:address "localhost"})
#object[java.net.ServerSocket 0x43f82e78 "ServerSocket[addr=localhost/127.0.0.1,
localport=4444]"]
user=> -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



Ativando REPL Remoto

Carregando-se arquivo digitado anteriormente

> (load-file “E:/clojure/nrepl.clj”)



The screenshot shows a Windows-style application window titled "REPL". The title bar also displays "Clojure 1.10.2-master-SNAPSHOT". The main window contains the Clojure user namespace prompt "user=>" followed by the command "(load-file "E:/clojure/nrepl.clj")". The window has standard window controls (minimize, maximize, close) and scroll bars on the right side.

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

Ativando REPL Remoto

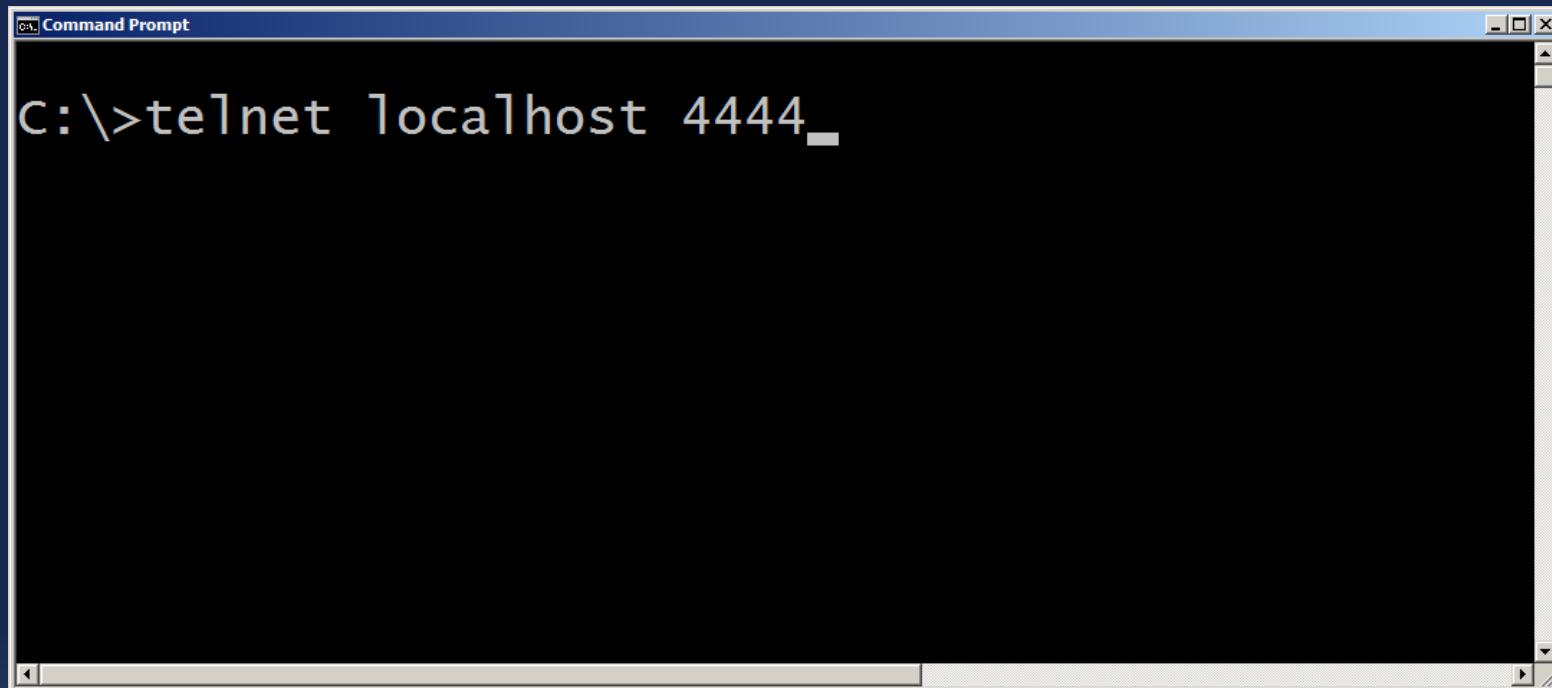
Carregando-se arquivo digitado anteriormente

> (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
localport=4444"]
user=>
```

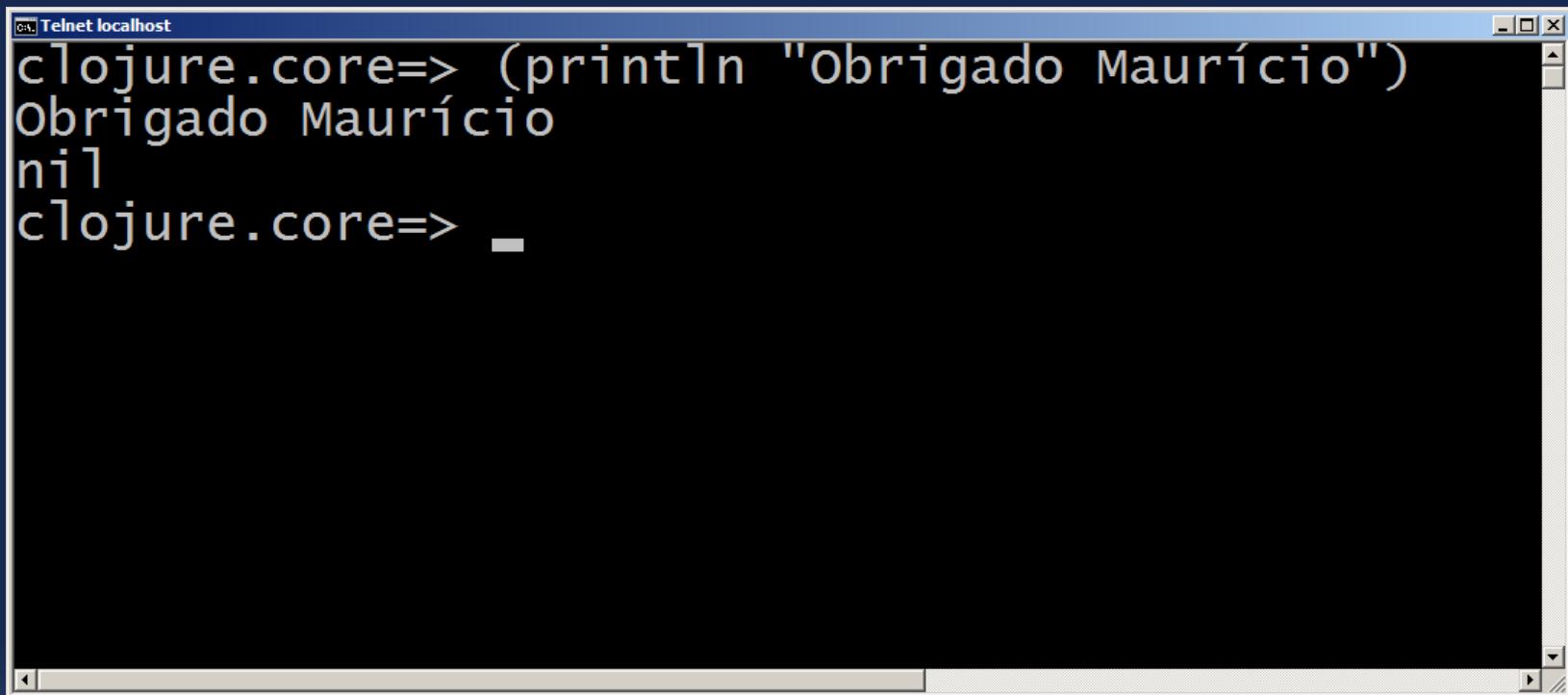


Conectando com Telnet na porta 4444



A screenshot of a Windows Command Prompt window. The title bar reads "Command Prompt". The main area shows the command "c:\>telnet localhost 4444" followed by a cursor. The window has standard Windows-style scroll bars on the right and bottom.

Conectando com Telnet na porta 4444



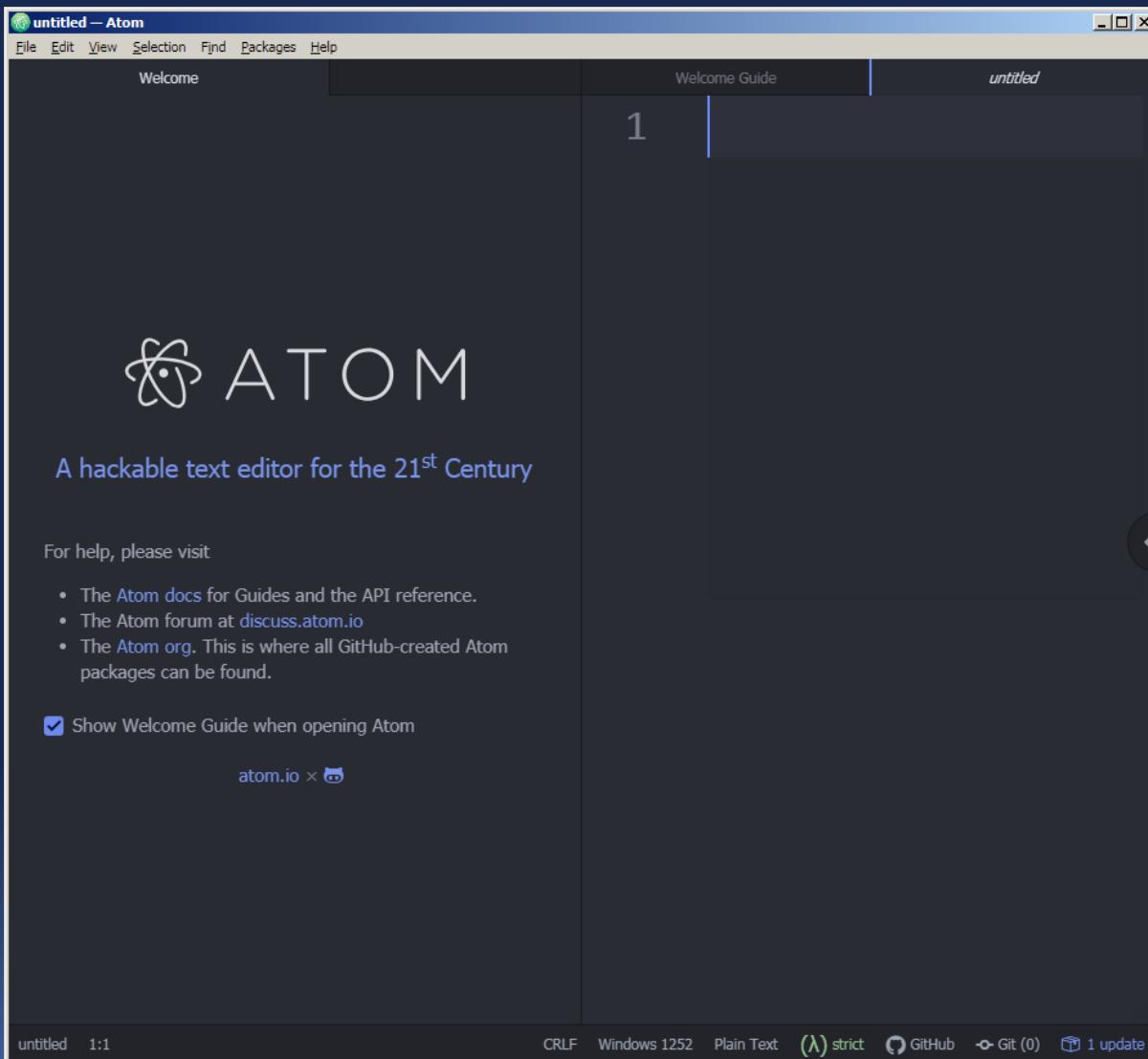
The screenshot shows a Windows Telnet window titled "Telnet localhost". The window contains the following Clojure code:

```
clojure.core=> (println "Obrigado Maurício")
Obrigado Maurício
nil
clojure.core=> _
```

Desenvolvimento Clojure com a IDE Atom

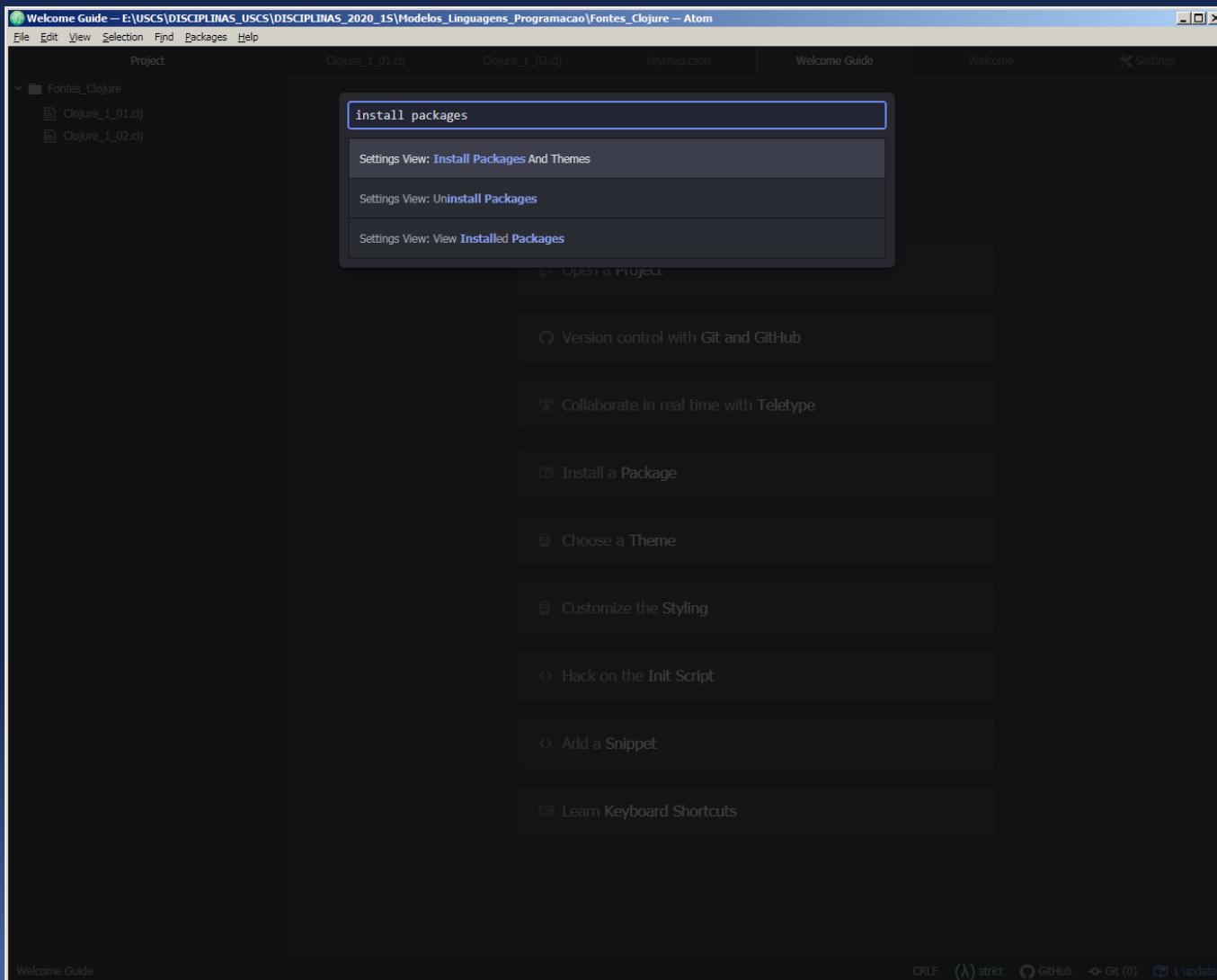


Código Clojure no Atom



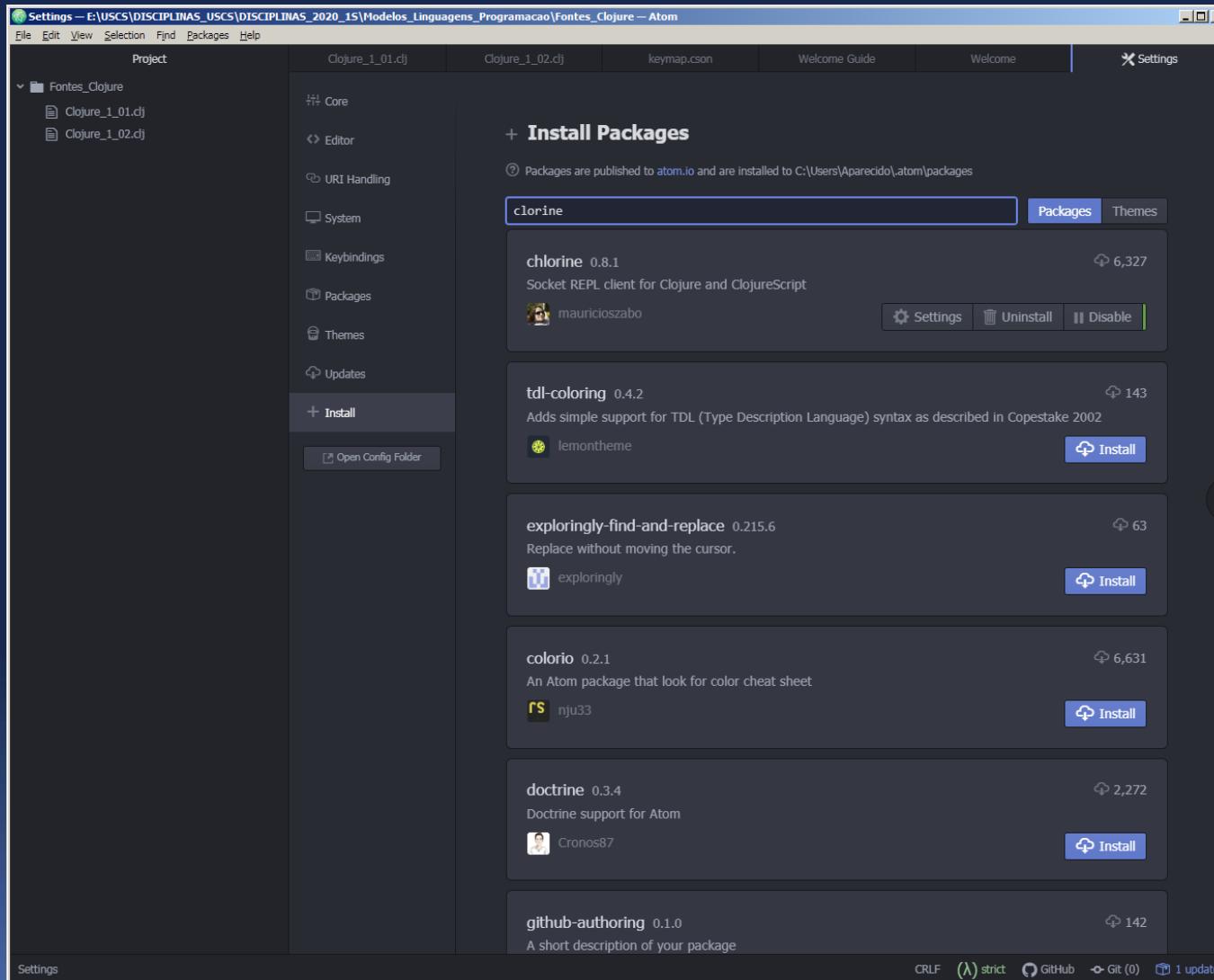
Atom – Package Chlorine

✓ **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:

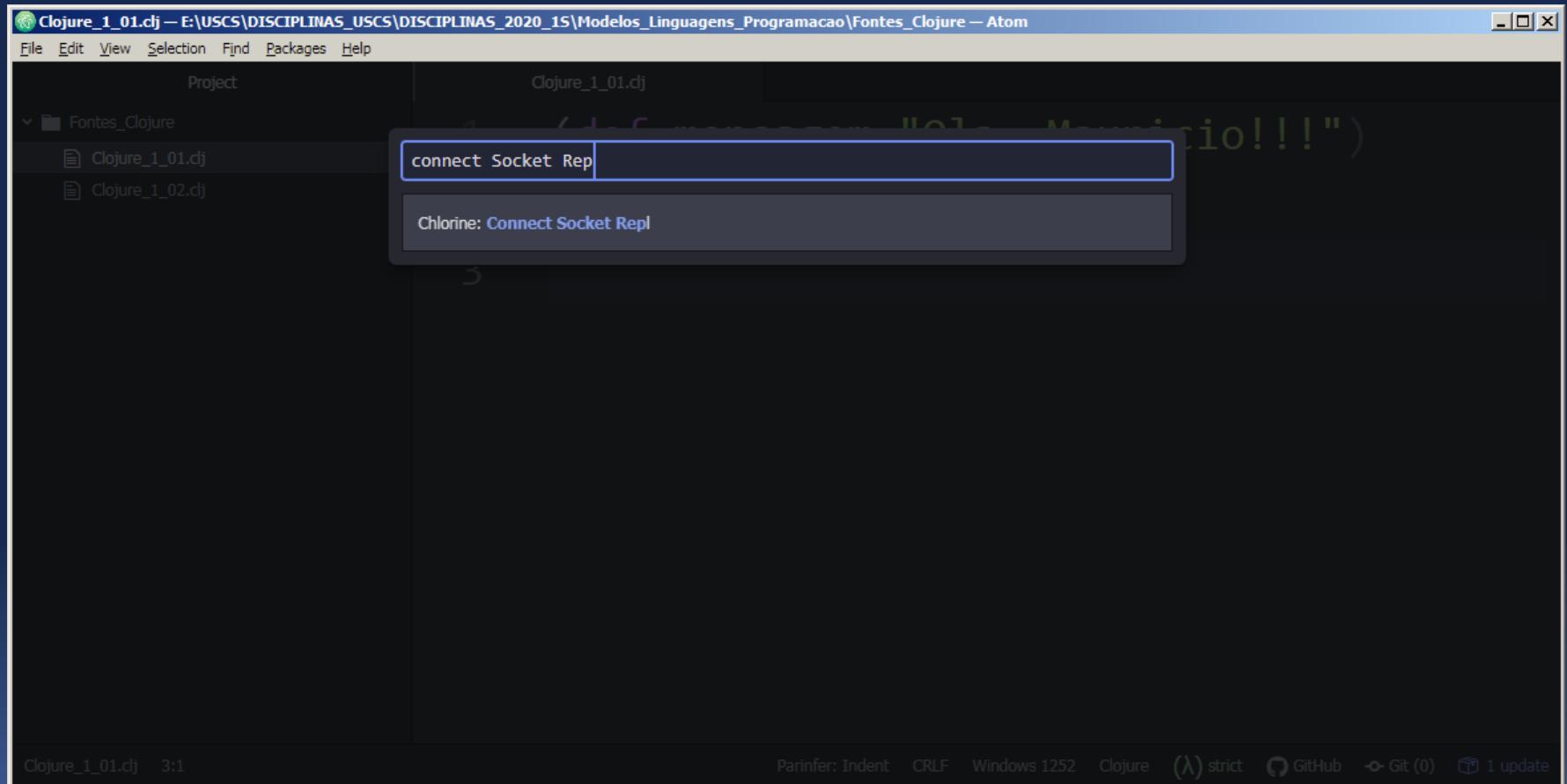
```
'atom-text-editor':  
  'ctrl-k':  
  'ctrl-shift-I':  
  'ctrl-shift-l':  
  'ctrl-shift-enter':  
  'ctrl-enter':  
  'ctrl-c':  
  'ctrl-d':  
    'chlorine:clear-console'  
    'chlorine:load-file'  
    'chlorine:clear-inline-results'  
    'chlorine:evaluate-block'  
    'chlorine:evaluate-top-block'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

Chlorine – Configuração de Hot Keys

```
# Atom Flight Manual:  
# http://flight-manual.atom.io/using-atom/sections/basic-customization/  
  
'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'
```

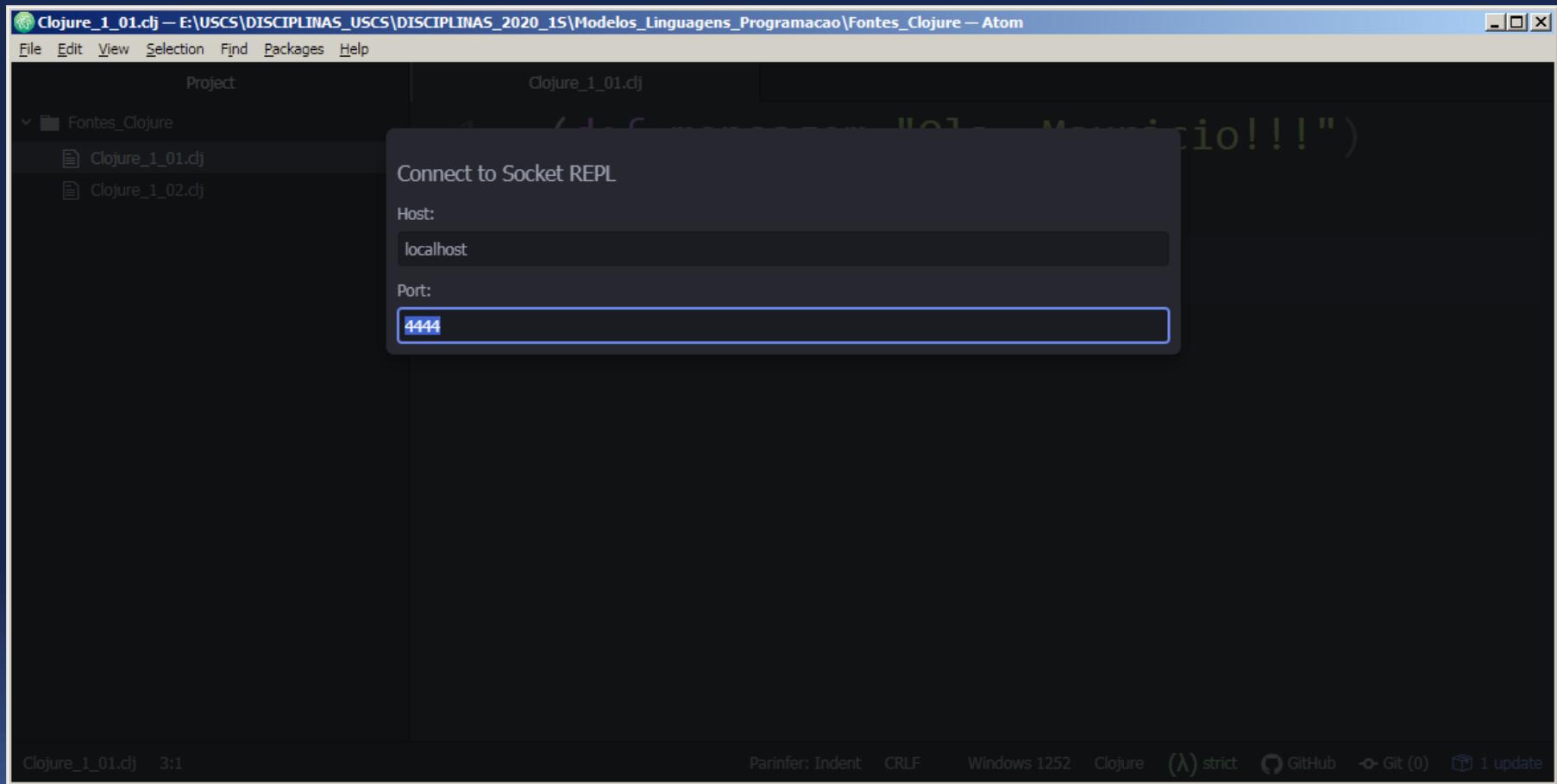
Atom – Package Chlorine

- ✓ 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 !!!

The screenshot shows the Atom code editor interface. On the left is the Project sidebar with a folder named 'Fontes_Clojure' containing files 'Clojure_1_01.clj' and 'Clojure_1_02.clj'. The main editor area displays the following Clojure code:

```
1 (def mensagem "Olá")
2 (println mensagem)
3
```

The status bar at the bottom indicates the file is 'Clojure_1_01.clj' at line 3:1. The bottom right corner shows status icons for Parinfer, CLJ (simple), Windows 1252, Clojure, strict mode, GitHub, Git (0), and 1 update.

A yellow warning dialog box titled 'Chlorine REPL' is open in the center-right of the screen, displaying the message: 'REPL already connected' with a link 'REPL is already connected. Please, disconnect the current REPL if you want to connect to another.'

Atom – Package Chlorine

✓ Escrever código *Clojure* (extensão .clj)



The screenshot shows the Atom code editor interface. The title bar reads "Clojure_1_01.clj – E:\USCS\DISCIPLINAS_USCS\DISCIPLINAS_2020_1S\Modelos_Linguagens_Programacao\Fontes_Clojure – Atom". The menu bar includes File, Edit, View, Selection, Find, Packages, and Help. The left sidebar shows a "Project" tree with "Fontes_Clojure" expanded, containing "Clojure_1_01.clj" and "Clojure_1_02.clj". The main editor area displays the following Clojure code:

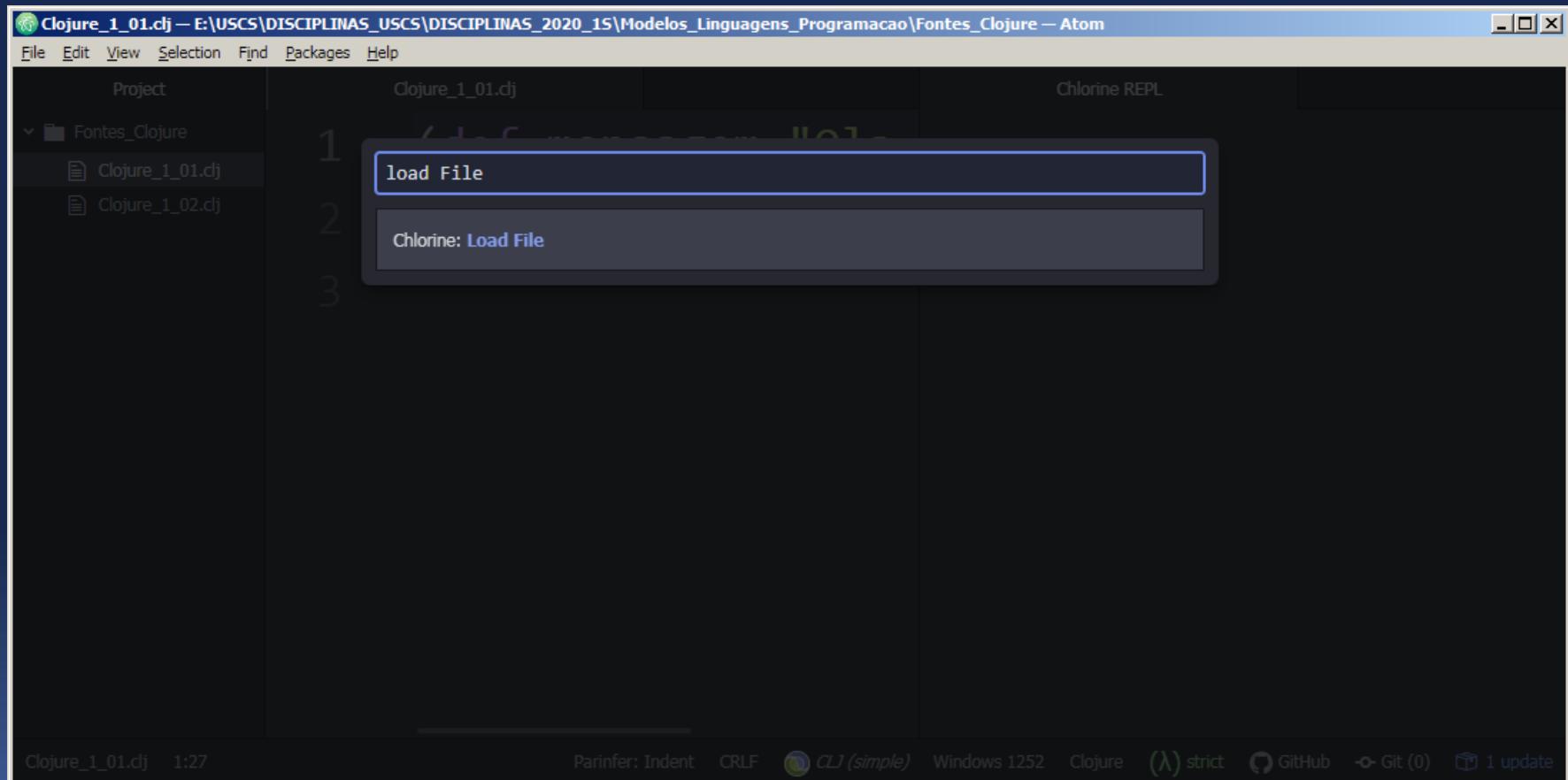
```
1 (def mensagem "Olá Mundo !!!")
2 (println mensagem)
3
```

The status bar at the bottom shows "Clojure_1_01.clj 1:27", "Parinfer: Indent CRLF", "CLJ (simple)", "Windows 1252", "Clojure", "(λ) strict", "GitHub", "Git (0)", and "1 update".



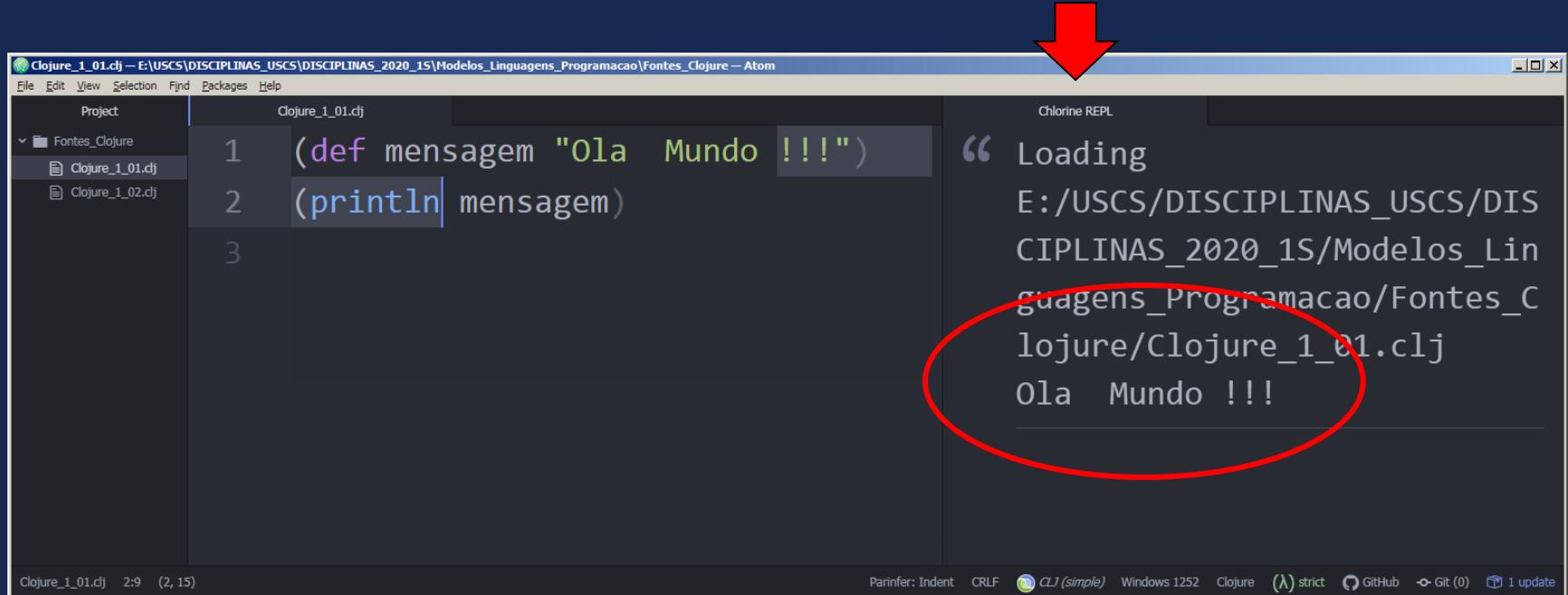
Atom – Package Chlorine

- ✓ Carregar código para REPL remoto
- ✓ **Ctrl + Shift + p > load File**



Atom – Package Chlorine

✓ Carga do código Clojure para o REPL



The screenshot shows the Atom text editor interface. On the left, the project structure shows 'Fontes_Clojure' expanded, with 'Clojure_1_01.clj' selected. The main editor area contains the following Clojure code:

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

To the right, the 'Chlorine REPL' panel displays the output of the code execution:

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

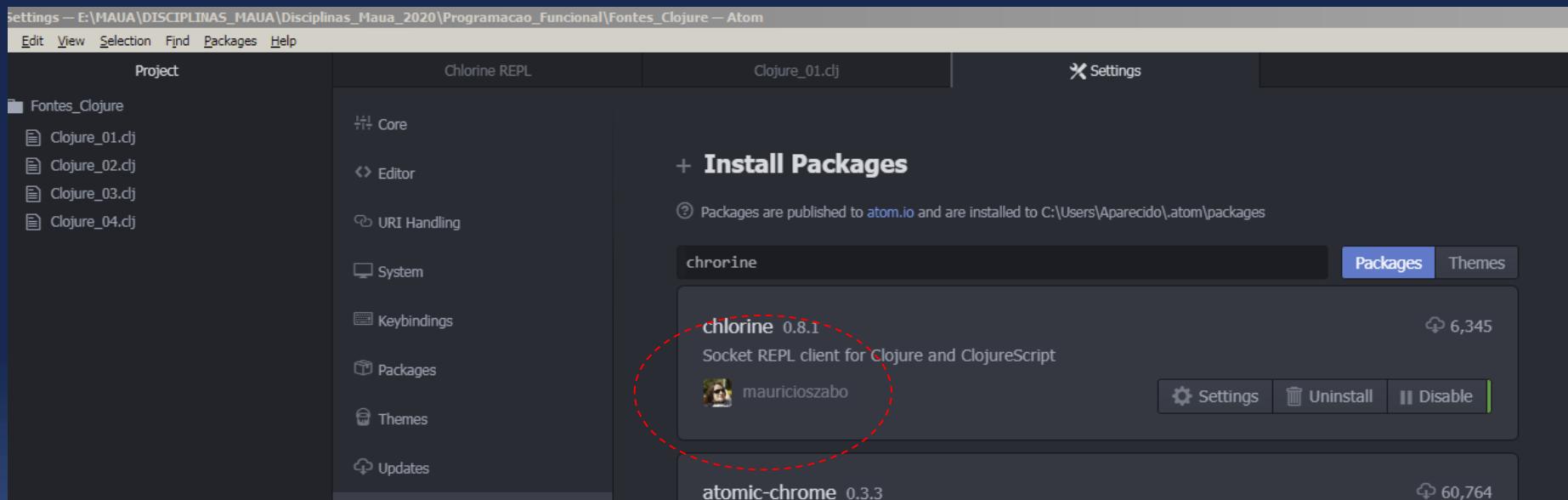
A red arrow points from the top right towards the status bar at the bottom right of the window. The status bar includes the file name 'Clojure_1_01.clj', line numbers '2:9 (2, 15)', and various status indicators like 'Parinfer: Indent', 'CRLF', 'CLJ (simple)', 'Windows 1252', 'Closure', '(λ) strict', 'GitHub', 'Git (0)', and '1 update'.

Atom – Package Chlorine + nREPL (Leiningen)



Configuração do Package Chlorine no Atom

- ✓ Instalação do Package *Chroline* no Atom
- ✓ No menu do Atom: Packages > Settings View > Install Packages / Themes



Configuração do Package Chlorine no Atom

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

My Atom editor configuration for working with Clojure/Script, revisited

Published by Maurício Szabo on 2019-09-29

Sometime ago, I did a post on how I work with Atom to develop Clojure and ClojureScript projects. It is in Portuguese, so I'm gonna re-visit the subject and also update with my current workflow.

There are two packages that I *have to install* to work with Clojure: `lisp-paredit` and `chlorine`. Without `lisp-paredit`, when I start a newline, the indentation gets all sorts of problematic. I use it on "strict mode" and use the tools to slurp/barf forward only. As for `chlorine`, it is needed to have autocomplete, evaluation, show documentation, goto var definition and so on. Last, I use also `parinfer` so I can remove whole lines of text and `parinfer` will infer the correct closing of parenthesis for me (most of the time at least).

Categories

Select Category

Tags

ActiveRecord advent of parens

agile android arte atom Banco de Dados



Configuração do Package Chlorine no Atom

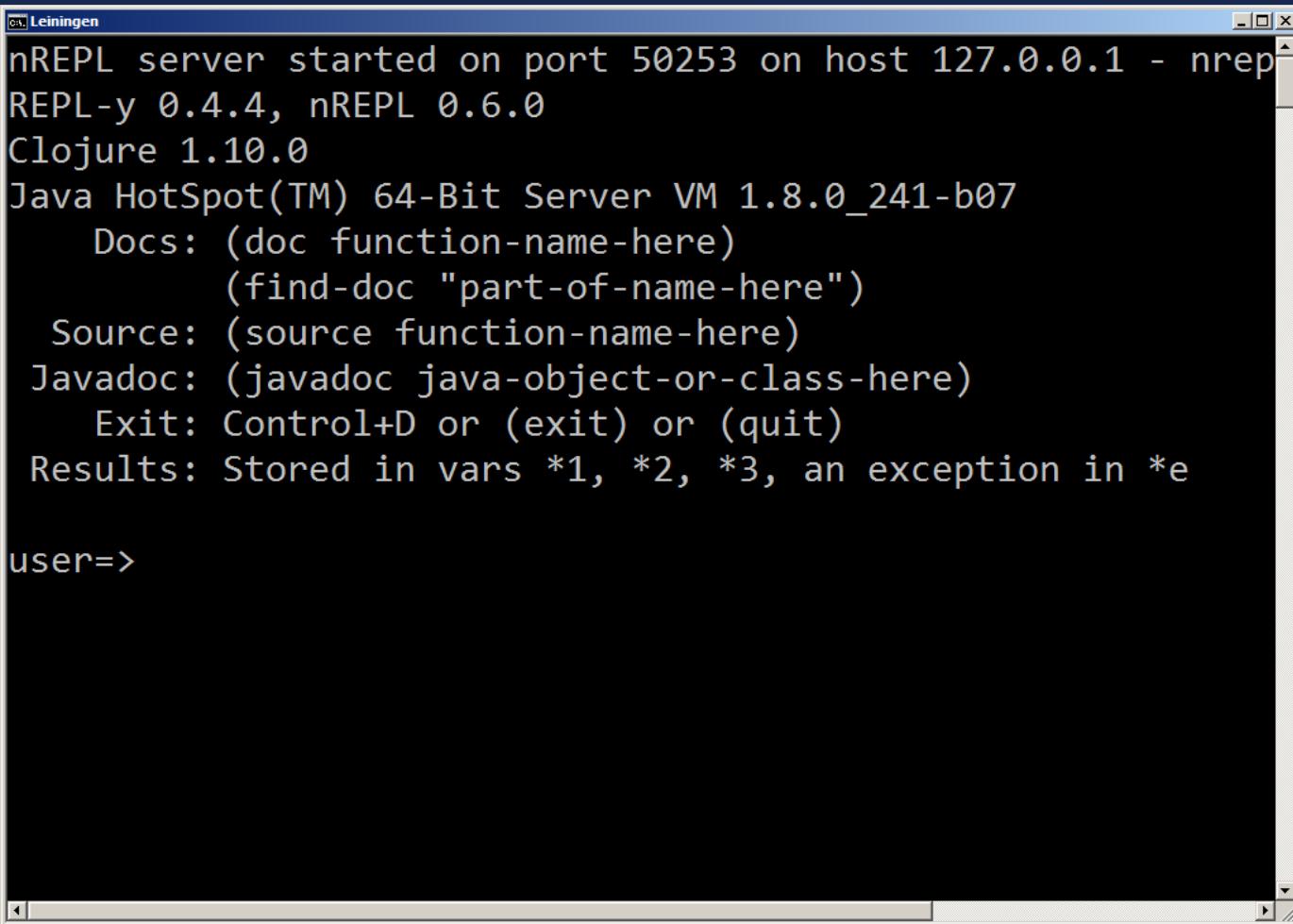
- ✓ Sob Atom, teclar **Crtl + Shift + p** e procurar “**Open your keymap**”;
- ✓ Após abertura do arquivo para customizar **keys**, incluir:

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



Ativar nREPL - Leiningen

- ✓ Anotar a porta de comunicação, por exemplo: **50253**

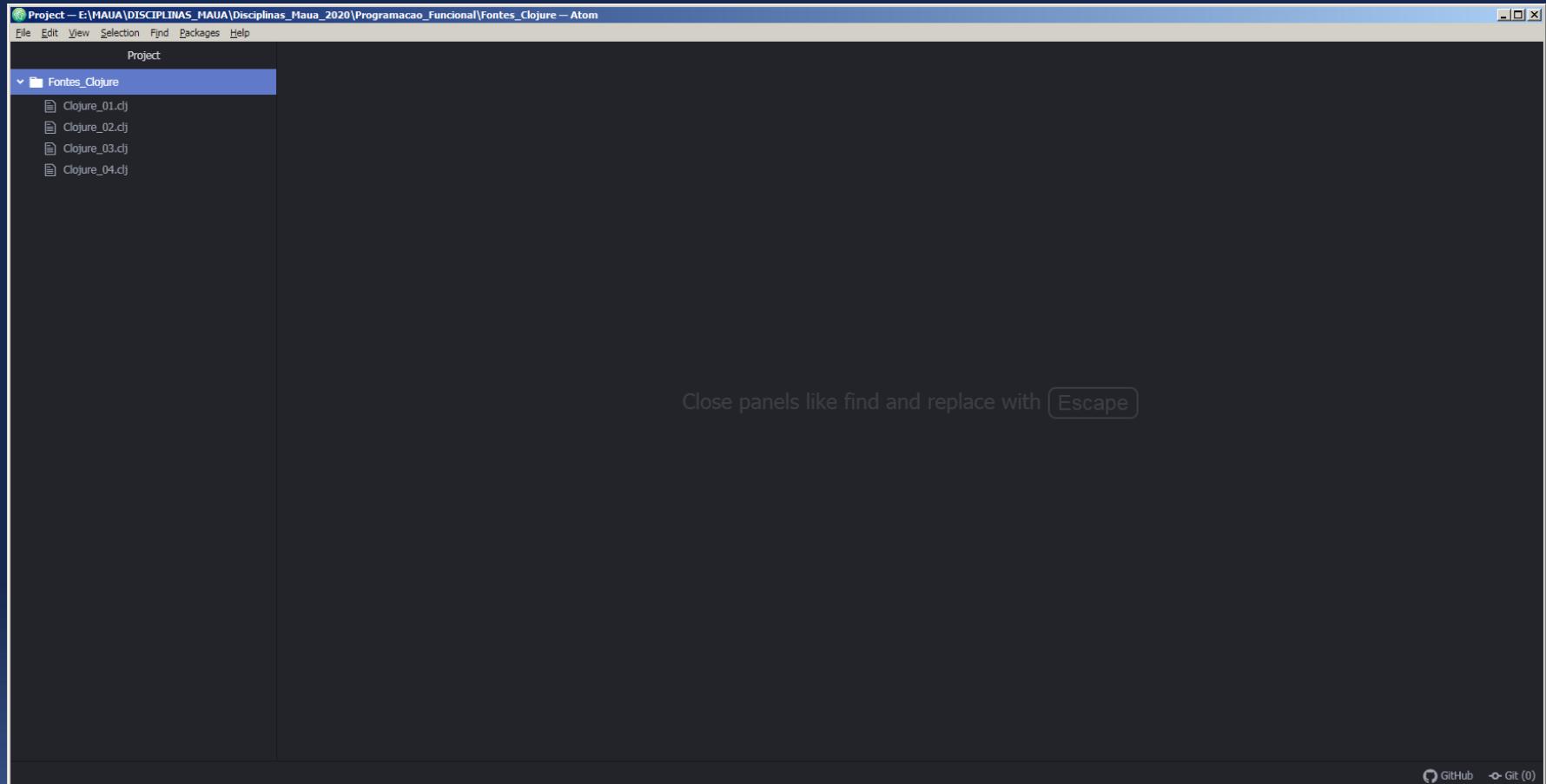


```
nREPL server started on port 50253 on host 127.0.0.1 - nrepl
REPL-y 0.4.4, nREPL 0.6.0
Clojure 1.10.0
Java HotSpot(TM) 64-Bit Server VM 1.8.0_241-b07
Docs: (doc function-name-here)
      (find-doc "part-of-name-here")
Source: (source function-name-here)
Javadoc: (javadoc java-object-or-class-here)
Exit: Control+D or (exit) or (quit)
Results: Stored in vars *1, *2, *3, an exception in *e

user=>
```

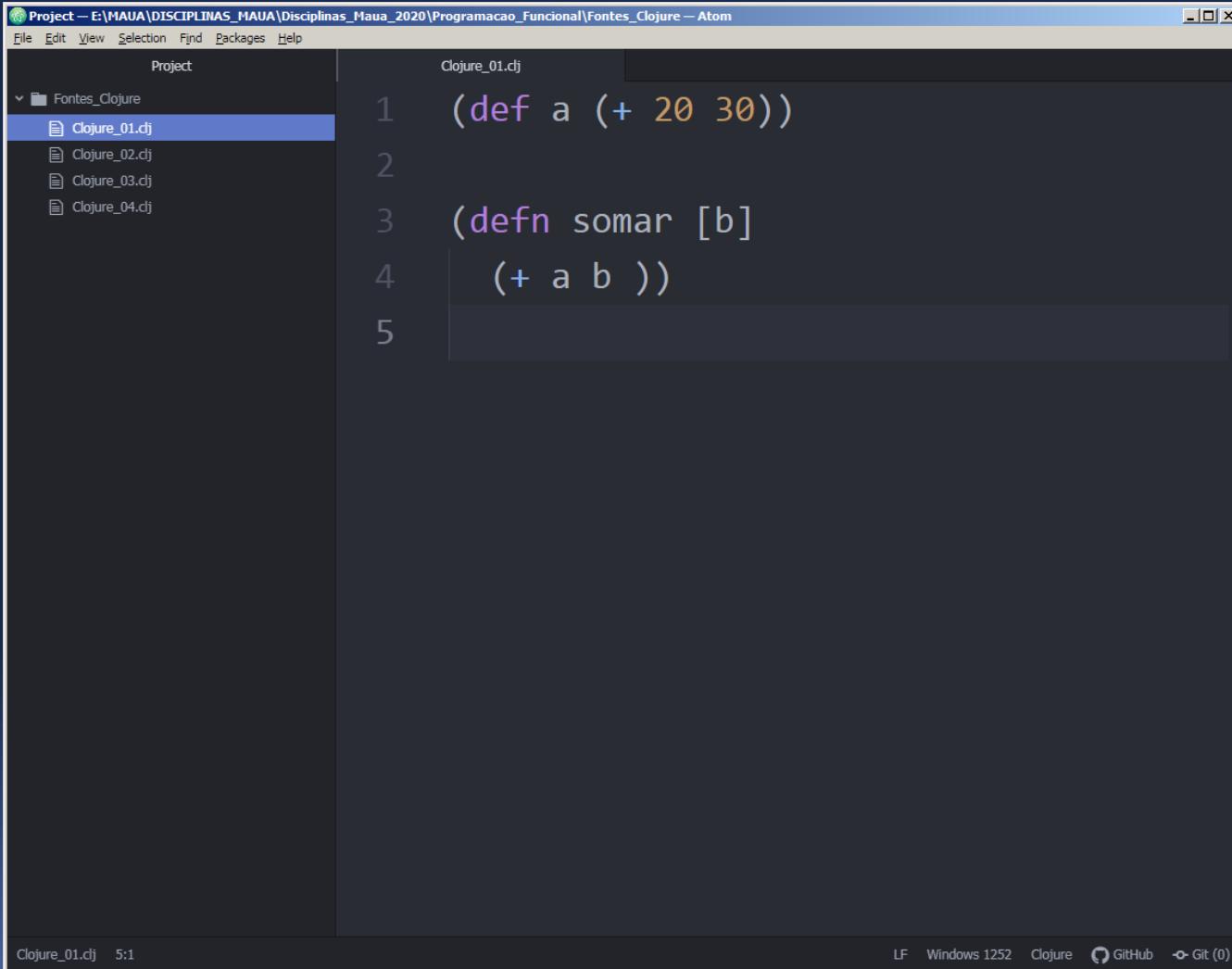
Ativar Atom

✓ File > Open Folder



Ativar Atom

✓ Escrevendo código Clojure no Atom



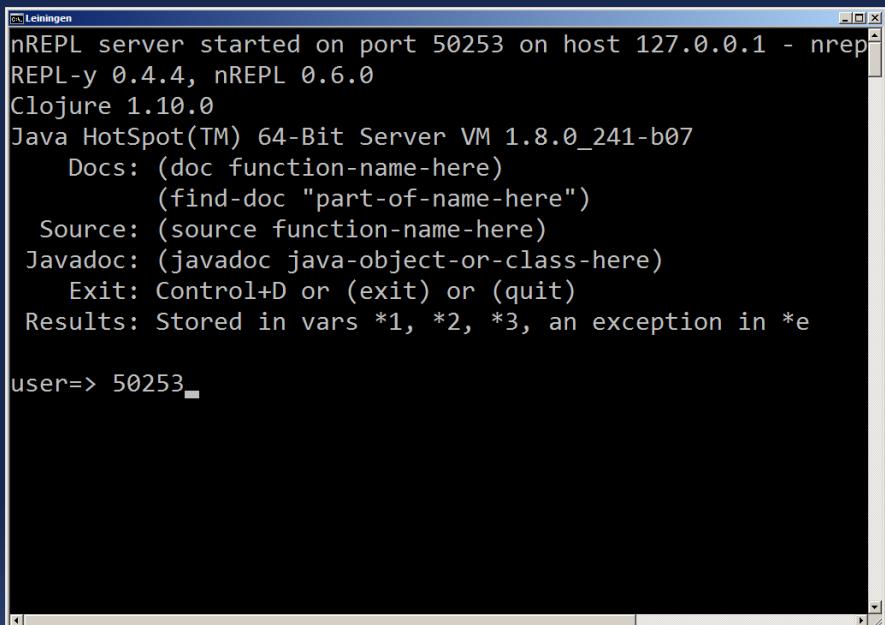
The screenshot shows the Atom code editor interface. The title bar reads "Project - E:\MAUA\DISCIPLINAS_MAU\Disciplinas_Maua_2020\Programacao_Funcional\Fontes_Clojure - Atom". The menu bar includes File, Edit, View, Selection, Find, Packages, and Help. The left sidebar is titled "Project" and shows a folder structure under "Fontes_Clojure": "Clojure_01.clj" (selected), "Clojure_02.clj", "Clojure_03.clj", and "Clojure_04.clj". The main editor area displays the following Clojure code:

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

The status bar at the bottom shows "Clojure_01.clj 5:1" and icons for LF, Windows 1252, Clojure, GitHub, and Git (0).

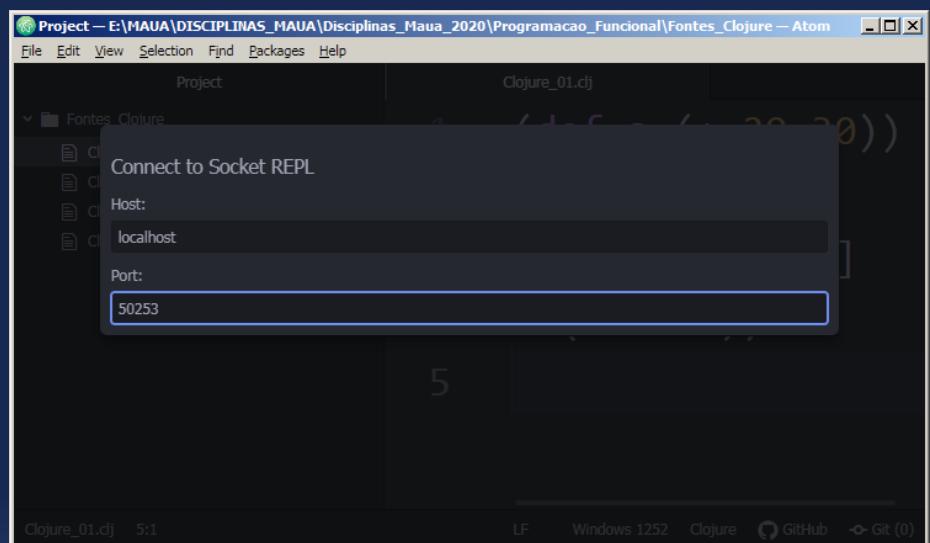
Ativar Atom

- ✓ Ativando **nREPL** (Leiningen)
- ✓ **Ctrl + Shift + p** > Connect to Socket REPL
 - > Host: **localhost**
 - > Port: aquela fornecida pela nREPL (Leiningen), por exemplo: **50253**



```
Leiningen
nREPL server started on port 50253 on host 127.0.0.1 - nrepl
REPL-y 0.4.4, nREPL 0.6.0
Clojure 1.10.0
Java HotSpot(TM) 64-Bit Server VM 1.8.0_241-b07
  Docs: (doc function-name-here)
         (find-doc "part-of-name-here")
  Source: (source function-name-here)
Javadoc: (javadoc java-object-or-class-here)
  Exit: Control+D or (exit) or (quit)
Results: Stored in vars *1, *2, *3, an exception in *e

user=> 50253
```



Clojure nREPL Connected

The screenshot shows a Clojure development environment with the title bar "Clojure_01.clj — E:\MAUA\DISCIPLINAS_MAU\Disciplinas_Maua_2020\Programacao_Funcional\Fontes_Clojure — A...".

The interface has three main panes:

- Project:** Shows a file tree under "Fontes_Clojure" with files: Clojure_01.clj, Clojure_02.clj, Clojure_03.clj, and Clojure_04.clj. "Clojure_01.clj" is selected.
- Clojure_01.clj:** Displays the code:

```
2
3 (defn
4   (+ a
5
```

A tooltip "Clojure nREPL Connected" is visible above the code area.
- Chlorine REPL:** Displays the output of the nREPL session:

```
2
3 (defn
4   (+ a
5
```

At the bottom, the status bar shows "Clojure_01.clj 5:1", "LF", "CLJ (simple)", "Windows 1252", "Clojure", "GitHub", and "Git (0)".

Carregando Código para nREPL

✓ Ctrl + Shift + p > Chroline: Load File



The screenshot shows the Atom code editor interface. On the left, the project structure includes files like Clojure_01.clj, Clojure_02.clj, Clojure_03.clj, and Clojure_04.clj. The Clojure_01.clj file is open, displaying the following code:

```
(def a (+ 20 30))
(defn somar [b]
  (+ a b))
```

To the right of the code editor is the nREPL (Clojure REPL) window. It shows the message "Loading E:/MAUA/DISCIPLINAS_MAUΑ/Disciplinas_Maua_2020/Programacao_Funcional/Fontes_Clojure/Clojure_01.clj". A tooltip "Loaded file" appears over the nREPL window, indicating the successful loading of the file. The status bar at the bottom of the screen shows various system and application icons.

Evaluate "top-blocks"

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

```
(def a (+ 20 30))
(defn somar [b]
  (+ a b))
(somar 5)
```

Evaluate "top-blocks"

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

The screenshot shows the Atom code editor interface. On the left, the project sidebar lists files: Clojure_01.clj, Clojure_02.clj, Clojure_03.clj, and Clojure_04.clj. The main editor pane displays Clojure code:

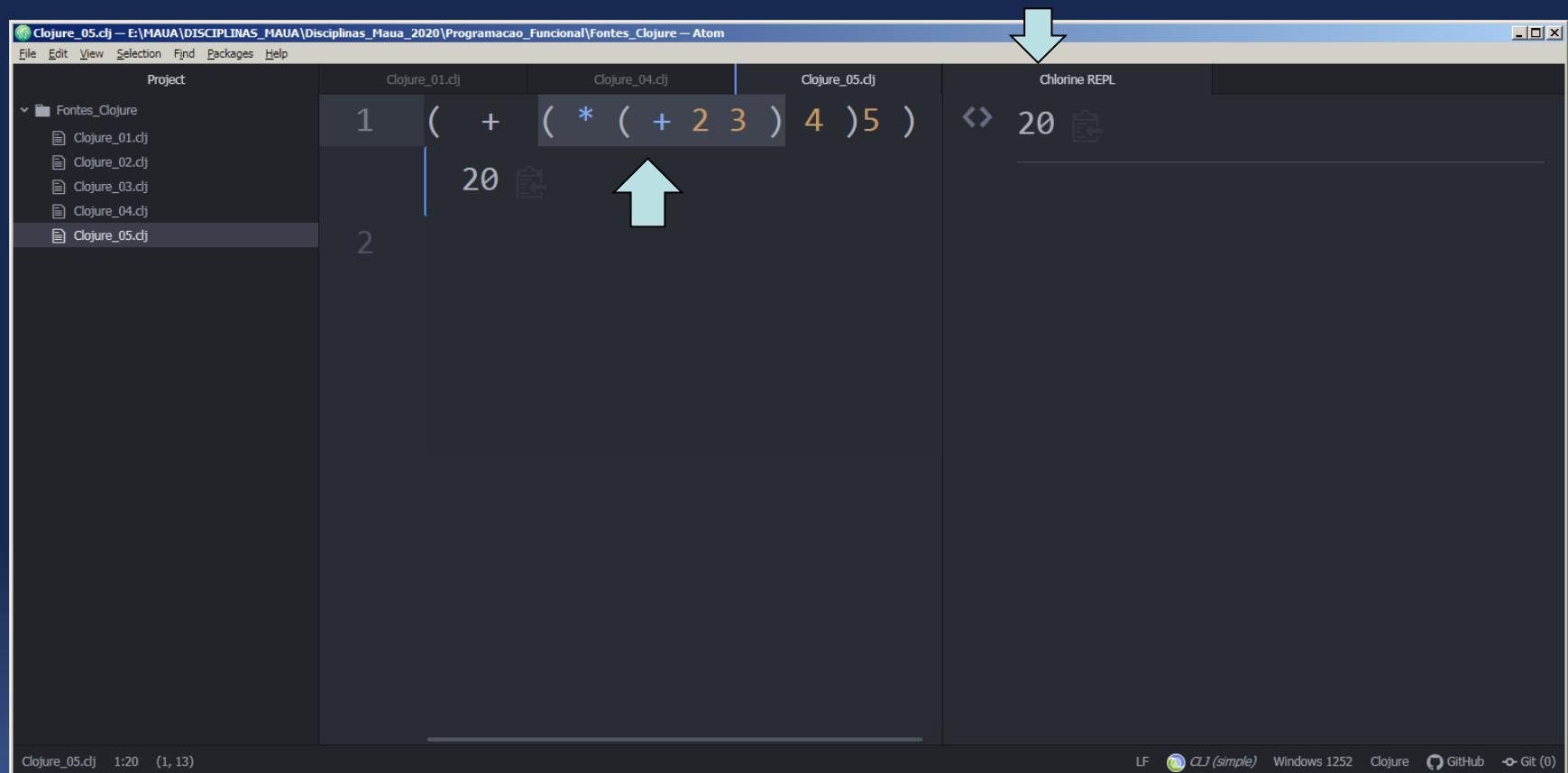
```
1 (def a (+ 20 30))
2
3 (defn somar [b]
4   (+ a b ))
5
6 (somar 5)
7
```

A light blue arrow points from the line '(somar 5)' in the editor to the same line in the 'Chlorine REPL' window on the right. The REPL window shows the result '55'.

Atom status bar at the bottom: Clojure_01.clj 7:1 (1, 10) LF CLJ (simple) Windows 1252 Clojure GitHub Git (0)

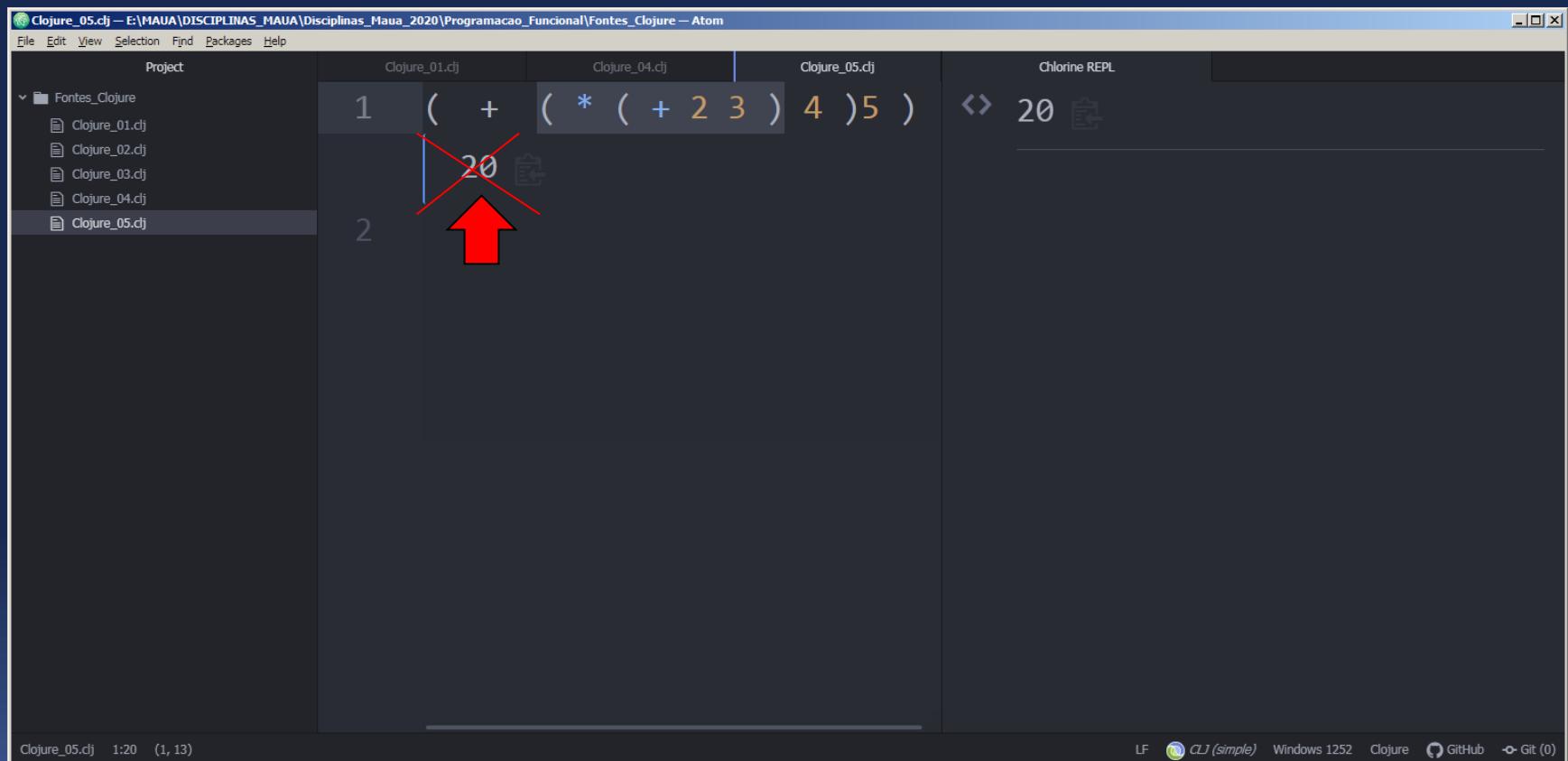
Evaluate "blocks"

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



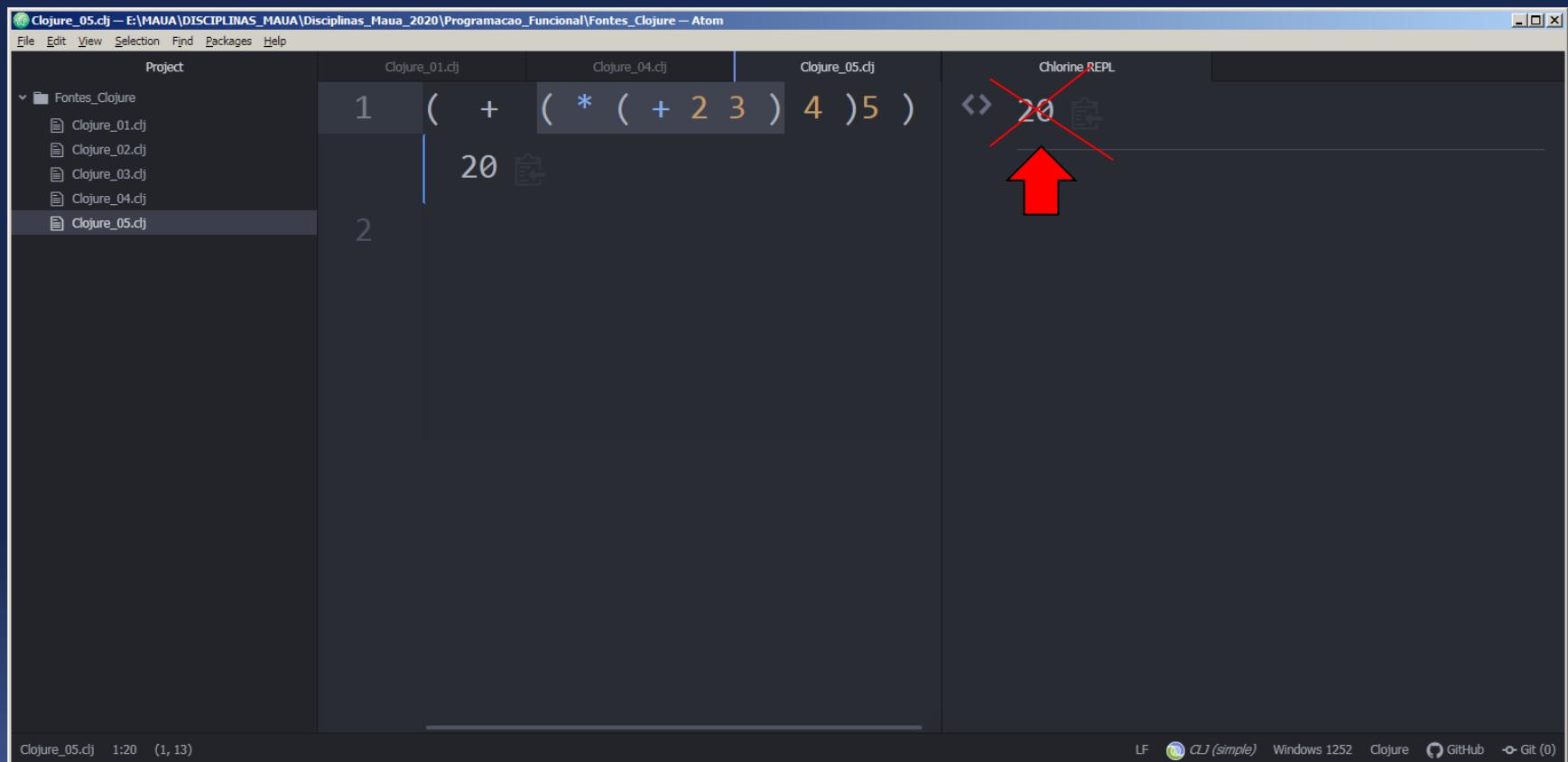
Clear Inline Results

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



Clear Console

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



The screenshot shows the Atom code editor interface. On the left, the project structure for 'Fontes_Clojure' is visible, with 'Clojure_05.clj' selected. The main editor area displays the following Clojure code:

```
1 ( + (* (+ 2 3) 4) 5 )  
2 20
```

To the right of the editor is the 'Chlorine REPL' terminal window. The terminal shows the output of the code execution. A large red arrow points to the clear console icon (an upward-pointing arrow) located in the bottom right corner of the terminal window.

Clear console

- ✓ $\text{Ctrl} + \text{Shift} + \text{p} > \text{Chroline: Clear Inline Results}$

