



Modelos de Linguagem de Programação

Unidade 9 – Programação Funcional com a Linguagem Clojure

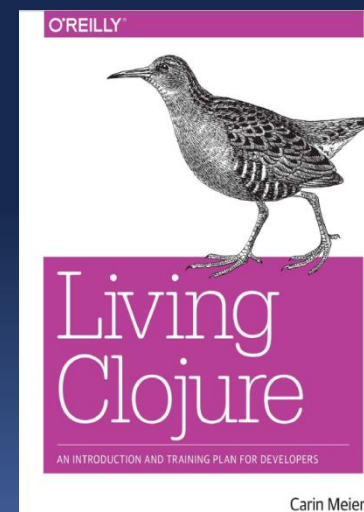
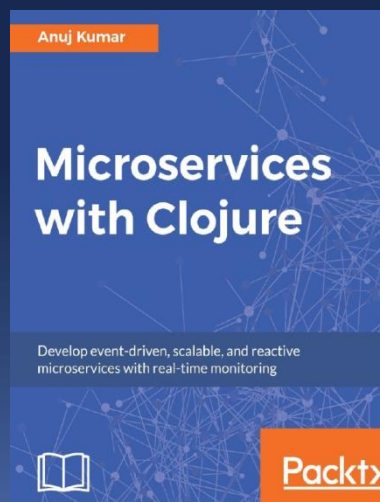
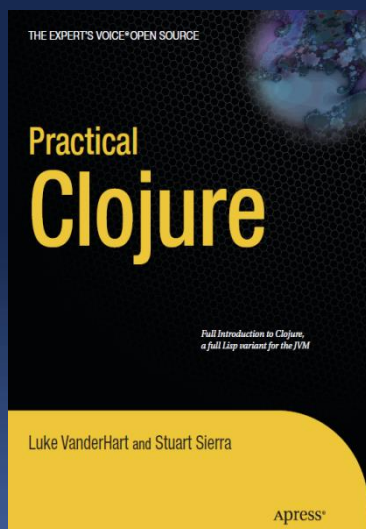
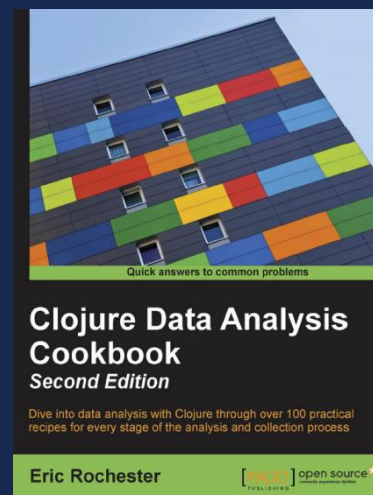
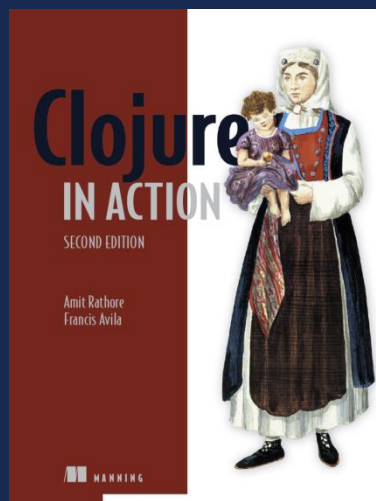


Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUVSP
aparecido.freitas@prof.uscs.edu.br
aparecidovfreitas@gmail.com





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 screenshot shows the Clojure.org website. At the top, there's a navigation bar with links: OVERVIEW, REFERENCE, API, RELEASES, GUIDES, COMMUNITY, DEV, NEWS, and a search icon. A green arrow points to the 'DEV' link. Below the navigation bar, the main header features the Clojure logo and a description: 'Clojure is a robust, practical, and fast programming language with a set of useful features that together form a simple, coherent, and powerful tool.' To the right of this description is a green 'Get Started!' button. The main content area is divided into two columns. The left column is titled 'The Clojure Programming Language' and contains three paragraphs of text about Clojure's features and philosophy, followed by a quote from Rich Hickey. The right column is titled 'Learn More' and contains links to 'Rationale', 'Getting Started', 'Reference', 'Guides', and 'Community'. At the bottom of the right column is a 'Clojure TV' logo. Below the main content area is a section titled 'Companies Succeeding with Clojure' which contains three quotes from users.

Clojure

OVERVIEW REFERENCE API RELEASES GUIDES COMMUNITY DEV NEWS

Clojure is a **robust, practical, and fast** programming language with a set of useful features that together form a **simple, coherent, and powerful** tool.

[Get Started!](#)

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

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!

 Clojure TV

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"





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:\clojure

```
Command Prompt
E:\>cd clojure
E:\clojure>_
```





Baixando clojure

```
Command Prompt
E:\>cd clojure
E:\Clojure>git clone https://github.com/clojure/clojure.git_
```





Baixando clojure

```
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

new folder

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
dojure.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`

```
Command Prompt - java -jar clojure.jar

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





Testando REPL

`java -jar clojure.jar`



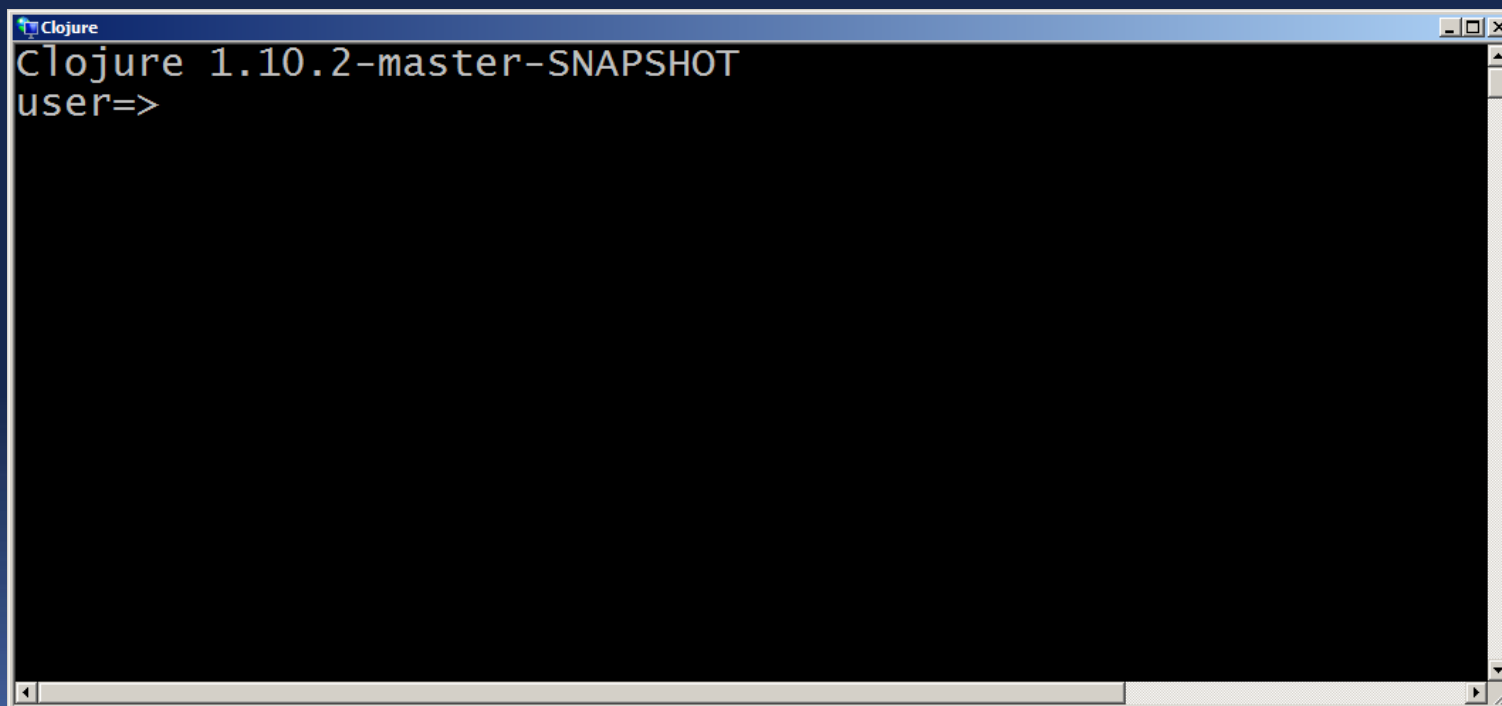
```
Command Prompt - java -jar clojure.jar
E:\clojure>java -jar clojure.jar
Clojure 1.10.2-master-SNAPSHOT
user=> (+ 2 3 )
5
user=>
```





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**;



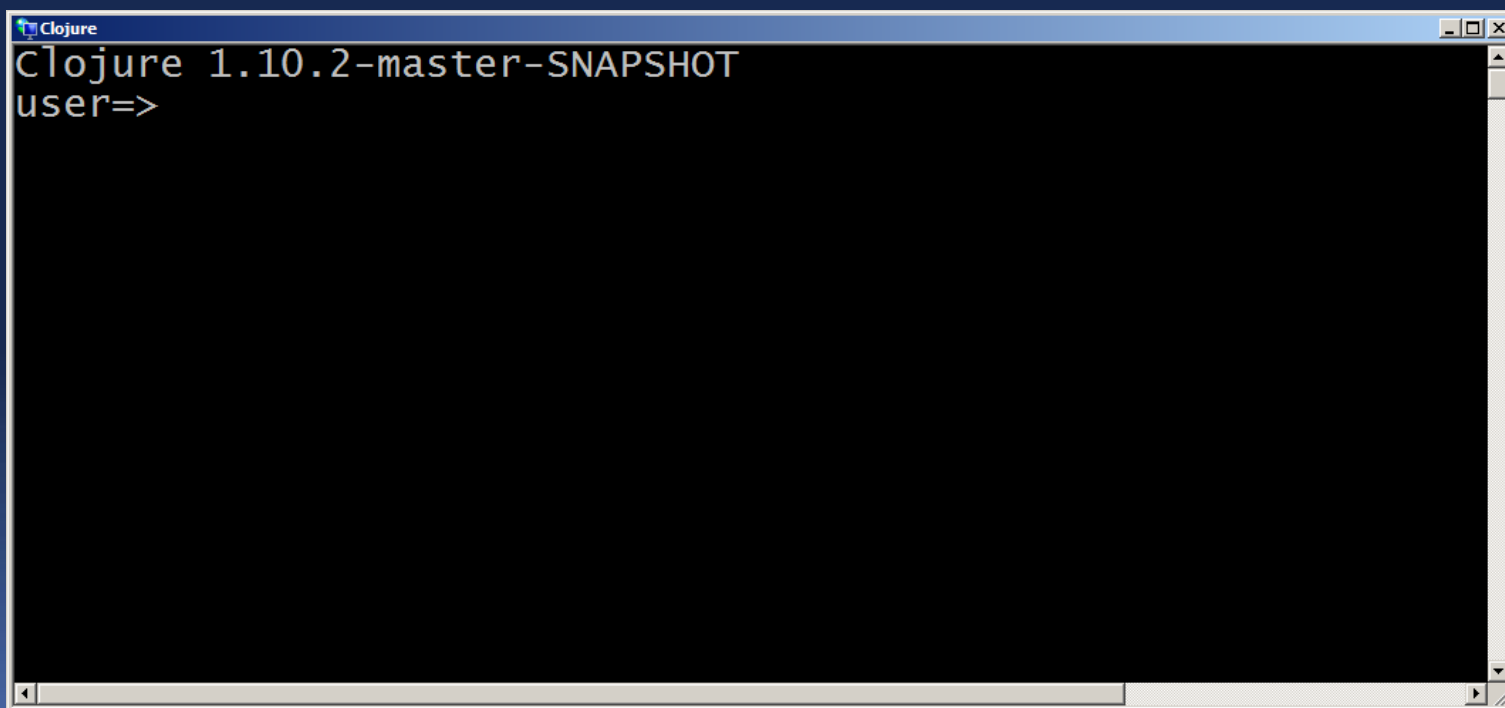
```
Clojure 1.10.2-master-SNAPSHOT
user=>
```





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**).



```
Clojure 1.10.2-master-SNAPSHOT
user=>
```





Avaliando expressões

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

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> "Hello USCS..."
"Hello USCS..."
user=>
```





Avaliando múltiplos strings

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> "Hello "      "USCS ...."
"Hello "
"USCS ...."
user=>
```

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





Avaliando expressões aritméticas

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> 5 + 8
5
#object[clojure.core$_PLUS_ 0x294a6b8e "clojure.core$_PLUS_@2
8
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

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





Avaliando expressões aritméticas

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





Avaliando expressões aritméticas

```
Clojure
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



```
Clojure 1.10.2-master-SNAPSHOT
user=> (println "Hello USCS....")
Hello USCS....
nil
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;



```
Clojure
Clojure 1.10.2-master-SNAPSHOT
user=> (print "Hello USCS")
Hello USCSnil
user=>
```





Funções nested

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





Exit REPL

(System/exit 0)

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





Operando com a última expressão avaliada => *1

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





Operando com a variável que contém o resultado da última

exceção => ***e**

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> (/ 1 0)
Execution error (ArithmeticException) at user/eval7 (REPL:1).
Divide by zero
user=>
user=>
user=>
user=>
user=> *e
#error {
  :cause "Divide by zero"
  :via
  [{:type java.lang.ArithmeticException
    :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$eval7 invokeStatic "NO_SOURCE_FILE" 1]
   [user$eval7 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=> (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)**

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
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")**

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (clojure.core/str "Hello")
"Hello"
user=>
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 **func-doc**;

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





lower-case e upper-case

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

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (clojure.string/upper-case "HELl11lo")
"HEL111LO"
user=>
user=>
user=> (clojure.string/lower-case "HELl11lo")
"hel111lo"
user=>
user=>
user=>
user=>
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.



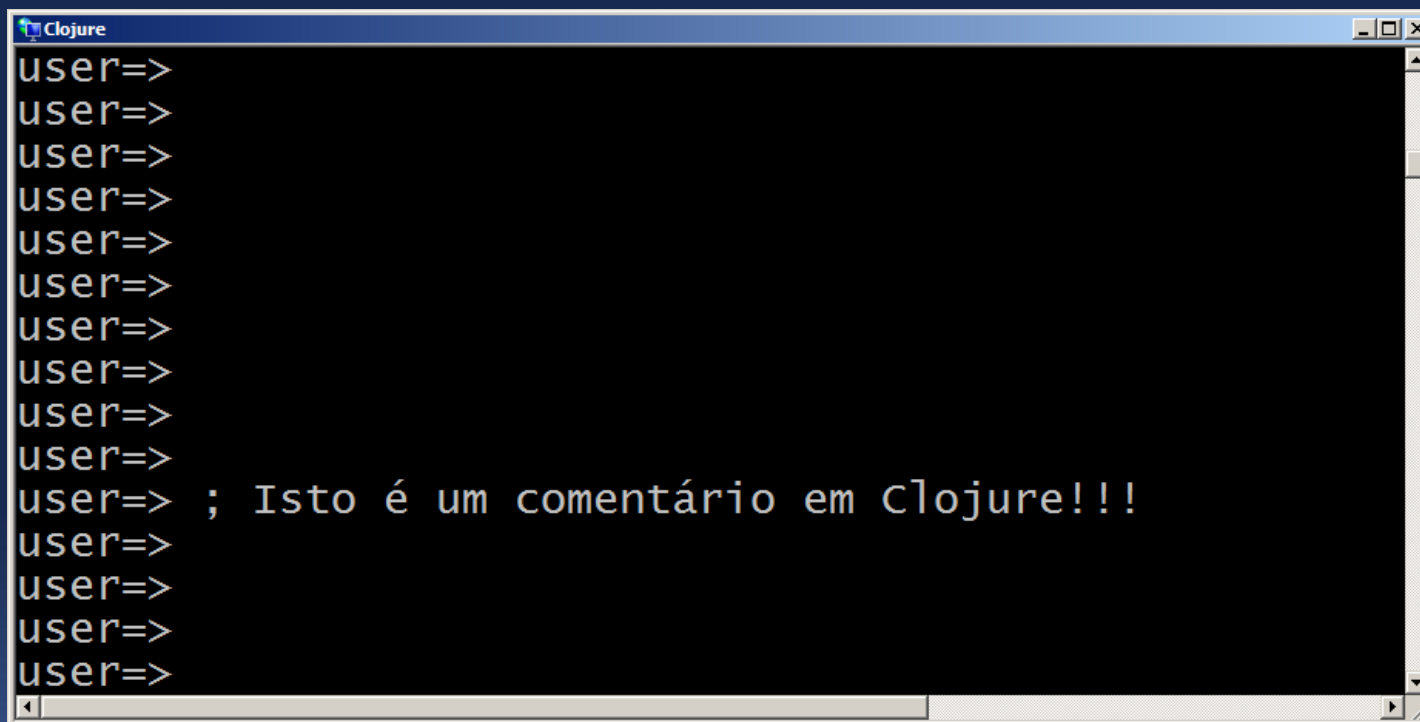
```
Clojure 1.10.2-master-SNAPSHOT
user=> "Hello"
"Hello"
user=>
user=>
user=>
user=> 1 2 3
1
2
3
user=> 1 2 "Hello"
1
2
"Hello"
user=> _
```





Comentários em Clojure

- ✓ Qualquer linha iniciada com ;



```
Clojure
user=>
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
```

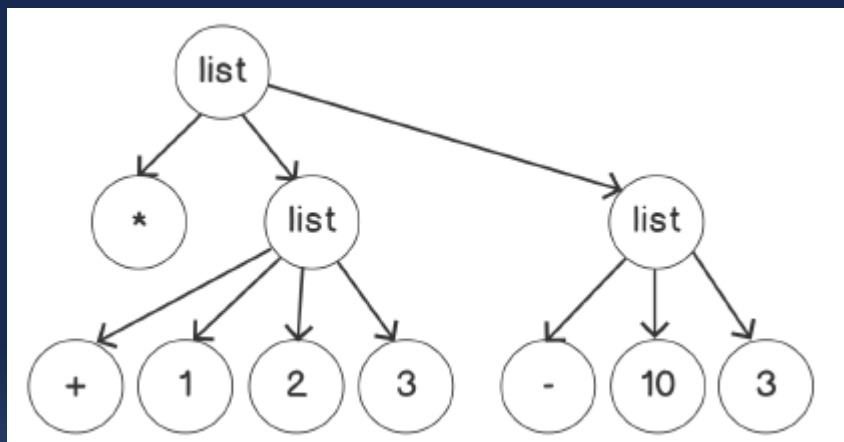




Expressões Simbólicas (s-expression)

- ✓ 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:

```
Clojure
user=>
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 (');

```
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** quisermos 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, dependendo 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 **fi**, mas não contém ramificação **else** e pode ser tratado como um **"do" implícito**.

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





Special Forms – **do**

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

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





Macro – **let**

```
Clojure
user=>
user=> (let [m "Maurício" a "Aparecido"] (println (str m " e " a) ) )
Maurício e Aparecido
nil
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
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=>
user=>
user=>
user=>
user=> (fn [] (println "Hello world...."))
#object[user$eval140$fn__141 0xaaafcffa "user$eval140$fn__141@aaafcffa"]
user=>
user=>
user=>
user=> (def hello-world-function (fn [] (println "Hello world")))
#'user/hello-world-function
user=>
user=>
user=>
user=>
user=> (hello-world-function)
Hello world
nil
user=>
user=>
user=>
```





Macro – fn

```
Clojure
user=>
user=>
user=>
user=>
user=> #( + 1 1 )
#object[user$eval155$fn__156 0x64a8c844 "user$eval155$fn__156@64a8c844"]
user=>
user=>
user=>
user=> (def um-mai-um-funcao #( + 1 1 ) )
#'user/um-mai-um-funcao
user=>
user=>
user=>
user=> (um-mais-um-funcao)
2
user=>
user=>
user=>
user=>
user=>
```





Macro – fn

```
Clojure
user=>
user=>
user=> #(+ 1 %)
#object[user$eval164$fn__165 0x2e9fda69 "user$eval164$fn__165@2e9fda69"]
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (def mais-um-funcao #(+ 1 %1) )
#'user/mais-um-funcao
user=>
user=>
user=>
user=> (mais-um-funcao 88)
89
user=>
user=>
user=>
```





Macro – **fn**

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> #( + %1 %2)
#object[user$eval177$fn__178 0x5a411614 "user$eval177$fn__178@5a411614"]
user=>
user=>
user=>
user=> (def funcao-soma #( + %1 %2) )
#'user/funcao-soma
user=>
user=>
user=>
user=> (funcao-soma 2 3)
5
user=>
user=>
user=>
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 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

```
Clojure
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

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> (if false (+ 10 5) (rand) )
0.6439786775978257
user=>
user=>
user=>
user=>
```





Atividade 4

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

Observação: A função **do** executa uma série de avaliações e sempre retorna a última





Atividade 4

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> (do (* 3 4 ) (/ 8 4) (+ 2 3 ) )
5
user=>
user=>
user=>
user=>
```





Observação – Side Effect

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (do (println "olá") (println "Tudo Bem?"))
olá
Tudo Bem?
nil
user=>
```





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

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (if true (do (println "Gerando numero randômico" (rand) ) ) )
Gerando numero randômico 0.290403568022474
nil
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

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> (when true (println "Gerando numero randômico" (rand) ) )
Gerando numero randômico 0.9317332462568976
nil
user=>
user=>
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).





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





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;

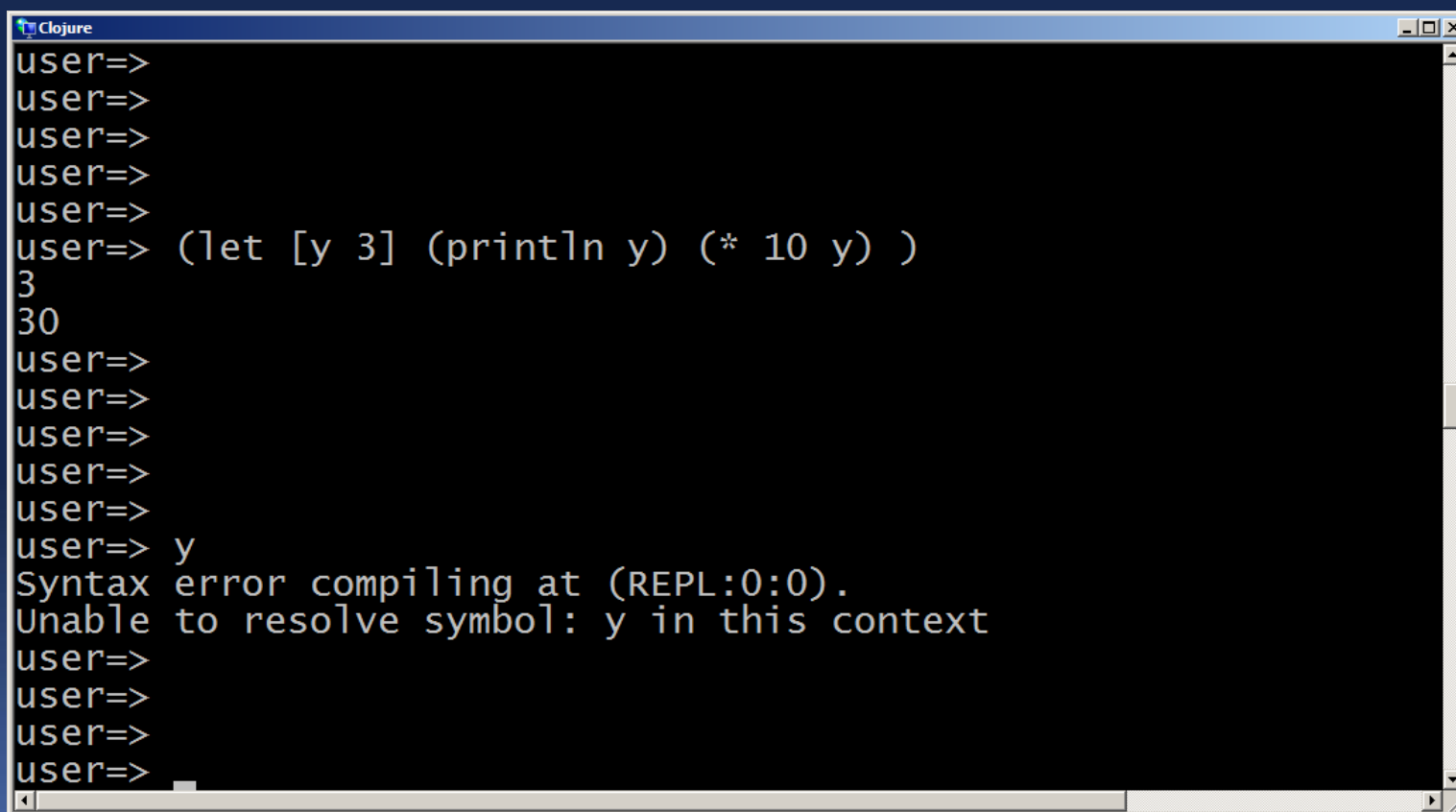
```
Clojure
user=>
user=>
user=>
user=> (def x 99)
#'user/x
user=>
user=> x
99
user=>
user=> (+ x 1)
100
user=>
user=>
user=> (inc x)
100
user=> (do (def x 42) )
#'user/x
user=> x
42
user=>
```





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**.



```
Clojure
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=>
```

y está fora
do escopo
definido
em let !!!





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]**





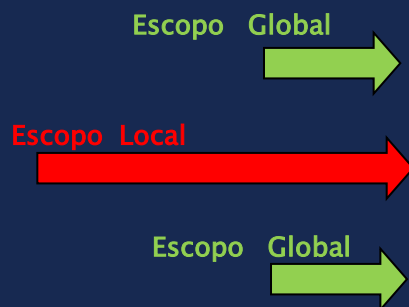
Definindo vetores

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





Escopos Global e Local



```
Clojure
user=>
user=>
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=>
```





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 [])

```
Clojure 1.10.2-master-SNAPSHOT
user=>
user=>
user=>
user=>
user=>
user=>
user=> (fn [] )
#object[user$eval136$fn__137 0x6b98a075 "user$eval136$fn__137@6b98a075"]
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 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

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> ( ( fn [par] (* par par) ) 5)
25
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
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 quisermos 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

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





Exemplo – Macro defn

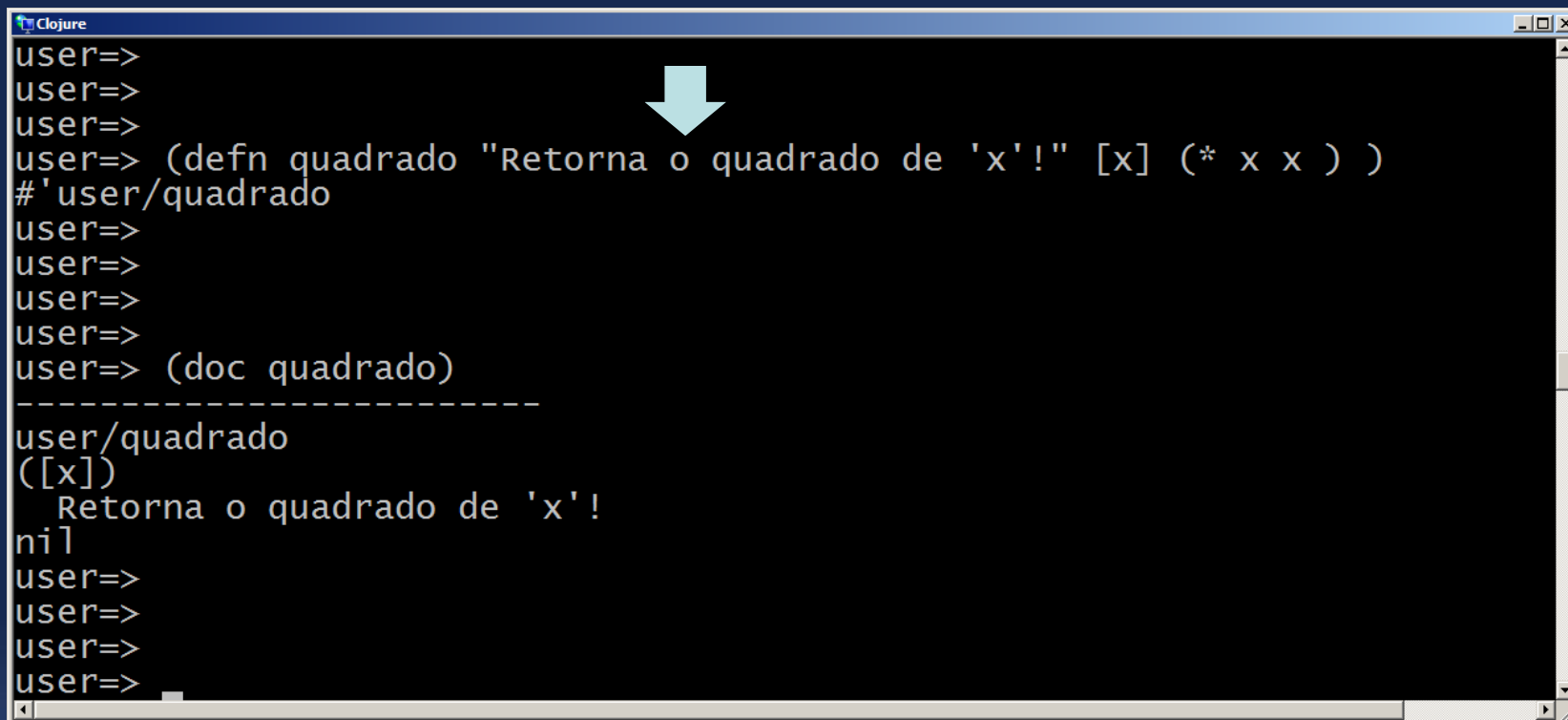
```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (defn funcao-hello [nome] (println (str "Hello, " nome) ) )
#'user/funcao-hello
user=>
user=>
user=>
user=>
user=>
user=> (funcao-hello "Maurício")
Hello, Maurício
nil
user=>
user=>
user=>
```





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 !



```
Clojure
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=>
```

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**.

```
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? "")
false
user=>
user=>
user=>
user=>
```



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 Igualdade

- ✓ 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 igualdade e retorna true se todos os argumentos forem iguais.
- ✓ Todas as outras operações de igualdade (`>` , `>=` , `<` , `<=`) são também implementações de funções.





Operações de Igualdade

```
Clojure
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 Igualdade

```
Clojure
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=>
user=>
```





Operações de Igualdade

```
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 Igualdade

```
Clojure
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=>
```





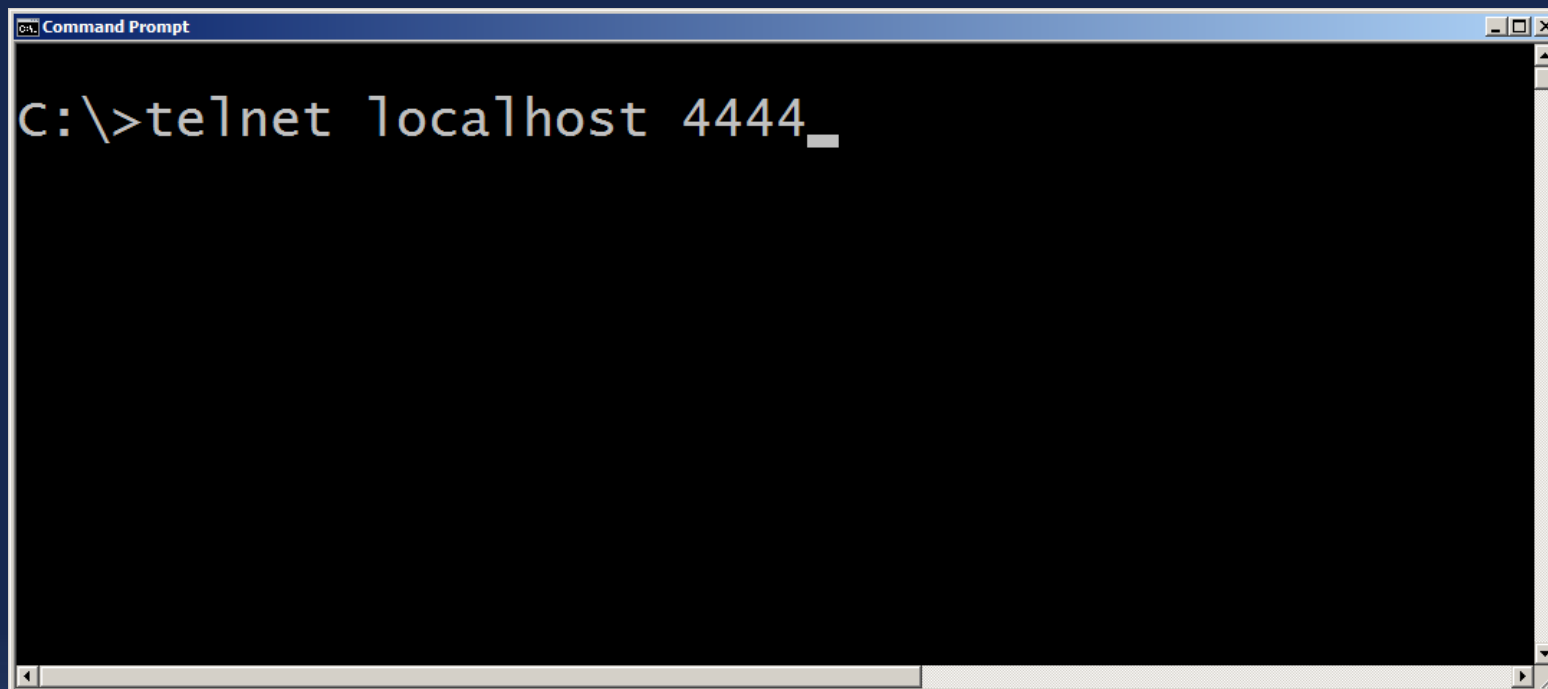
Ativando REPL Remoto

```
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=> _
```





Conectando com Telnet na porta 4444



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





Conectando com Telnet na porta 4444

```
Telnet localhost
c10j1re.core=> (println "Obrigado Maurício")
Obrigado Maurício
nil
c10j1re.core=> _
```

