

Programação Funcional

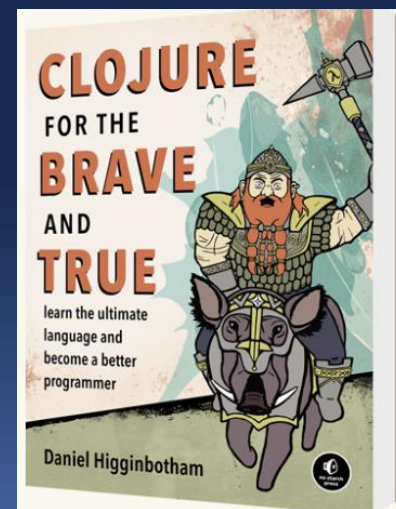
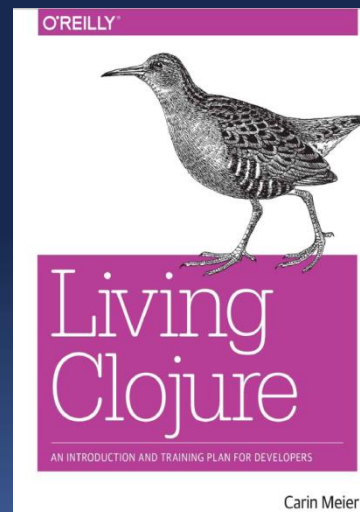
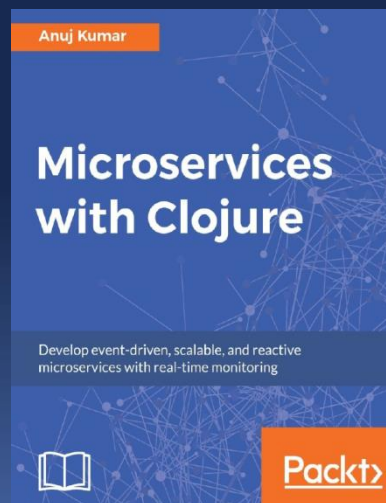
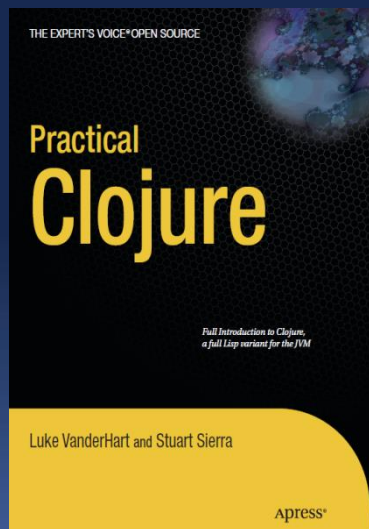
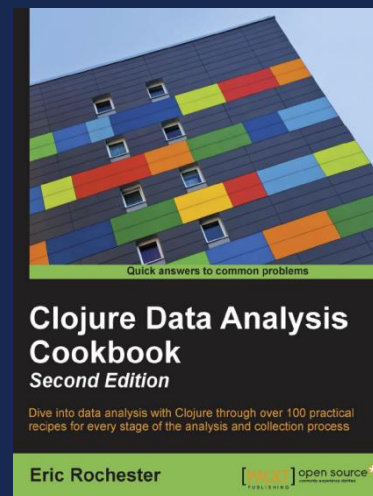
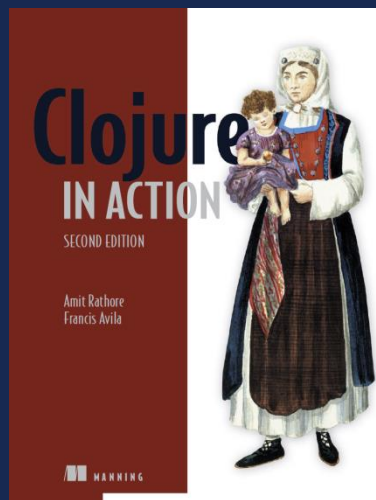
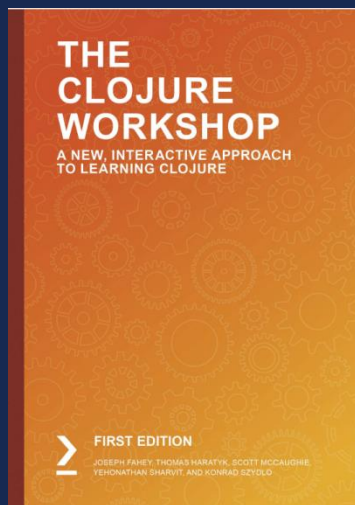
Unidade 5– Tipos simples de dados e Imutabilidade em Clojure



Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecidovfreitas@gmail.com



Bibliografia



Introdução

- ✓ Na linguagem **Clojure** trabalha-se quase que exclusivamente com tipos de dados imutáveis;
- ✓ Esses tipos de dados trazem **legibilidade ao código fonte** e evitam conflitos decorrentes de compartilhamento de dados;
- ✓ **Clojure** trabalha dessa forma, com estruturas de dados persistentes. Ou seja, novos dados são produzidos a partir dos dados existentes. Versões prévias dos dados são mantidas;
- ✓ Clojure trabalha com os seguintes tipos básicos de dados: **Strings**, **Números**, **Booleans**, **Keywords** e **nil** (todos **imutáveis**).

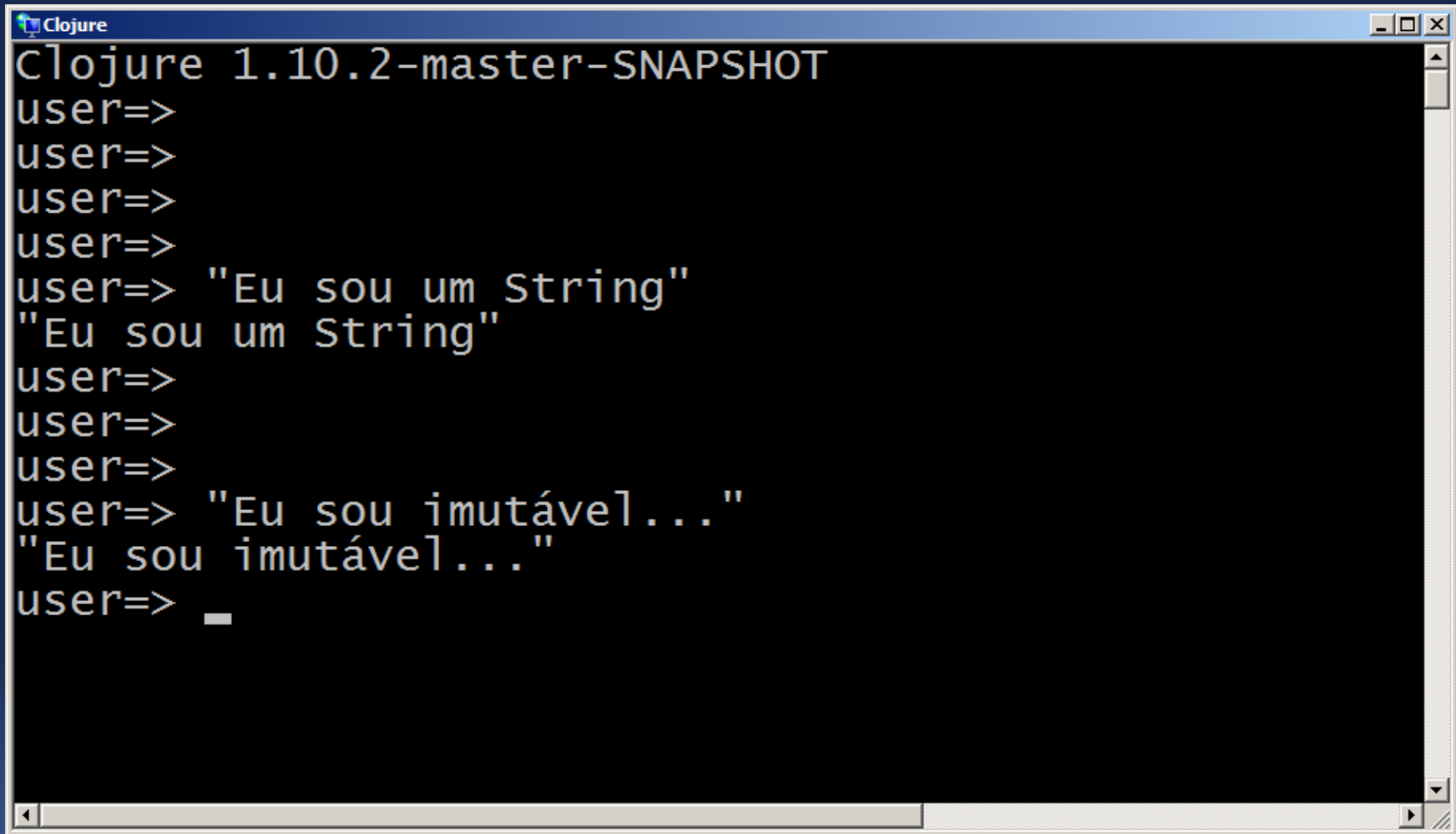


Strings

- ✓ **Strings** são sequências de caracteres representando texto;
- ✓ Sempre criados com aspas duplas ("");
- ✓ Não podem ser alterados; São imutáveis;
- ✓ Qualquer função que os usa, transforma-os num outro valor.



Strings



```
Clojure 1.10.2-master-SNAPSHOT
user=>
user=>
user=>
user=>
user=> "Eu sou um String"
"Eu sou um String"
user=>
user=>
user=>
user=> "Eu sou imutável..."
"Eu sou imutável..."
user=> _
```




Strings são imutáveis


- ✓ No exemplo a seguir , a função `clojure.string/replace` retorna um novo String;
- ✓ Mesmo após a aplicação desta função, o valor original do símbolo `texto-string` será mantido.



Strings são imutáveis



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (def texto-string "Eu sou imutável")
#'user/texto-string
user=>
user=>
user=> texto-string
"Eu sou imutável"
user=>
user=> (clojure.string/replace texto-string "Eu sou" "Você é")
"Você é imutável"
user=>
user=>
user=>
user=> texto-string
"Eu sou imutável"
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

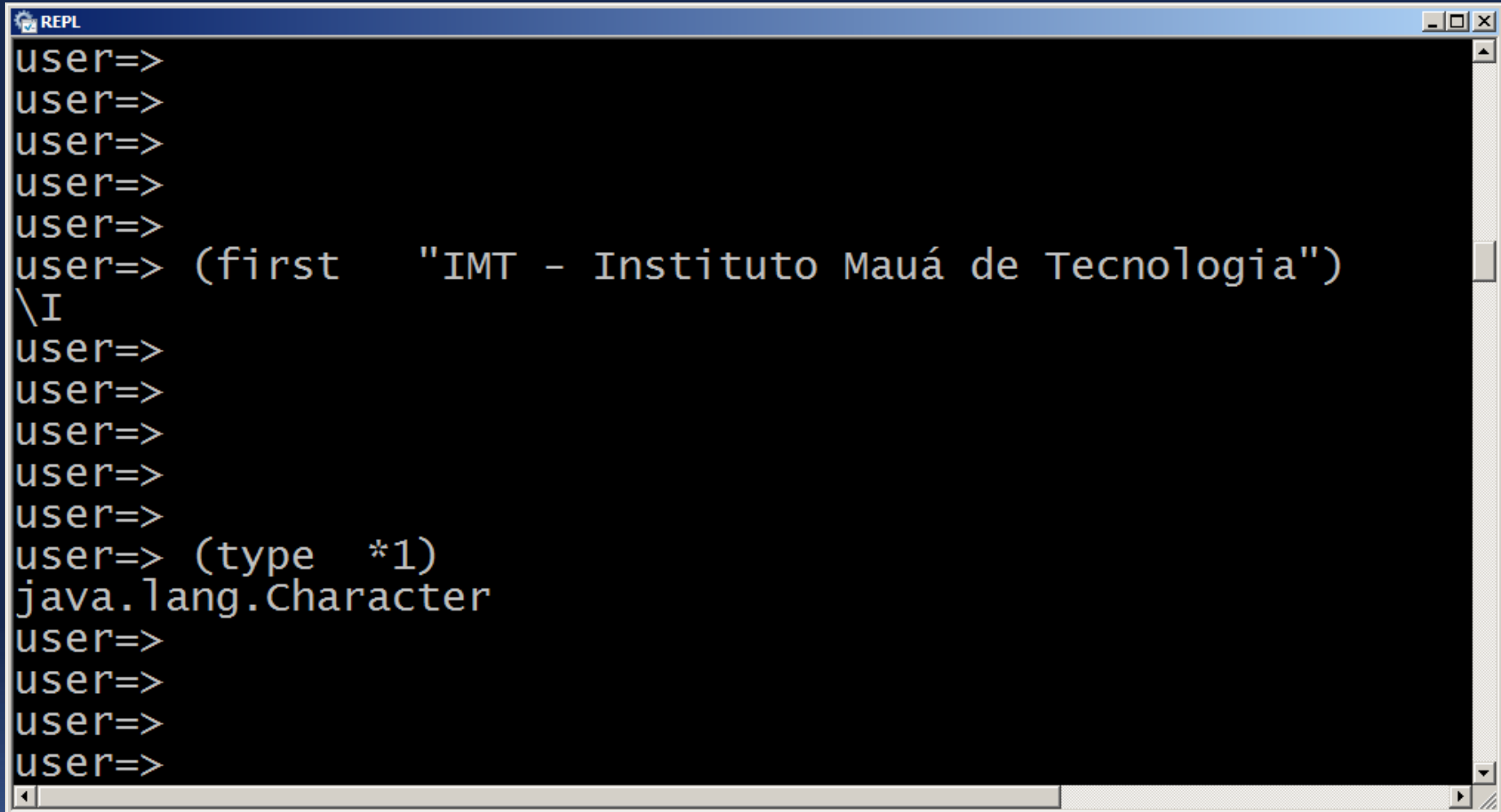


Strings são imutáveis

- ✓ Embora string seja uma unidade de representação de texto, na verdade é composto por um conjunto de caracteres;
- ✓ Na implementação **Java JVM** um string é do tipo `java.lang.String` e é implementado por coleções de caracteres de `java.lang.Character`;
- ✓ **first** retornou o primeiro elemento da coleção, o qual é um caractere;
- ✓ Literais caracteres são representados por `\`;
- ✓ Em REPL, ***1** representa o último valor retornado;



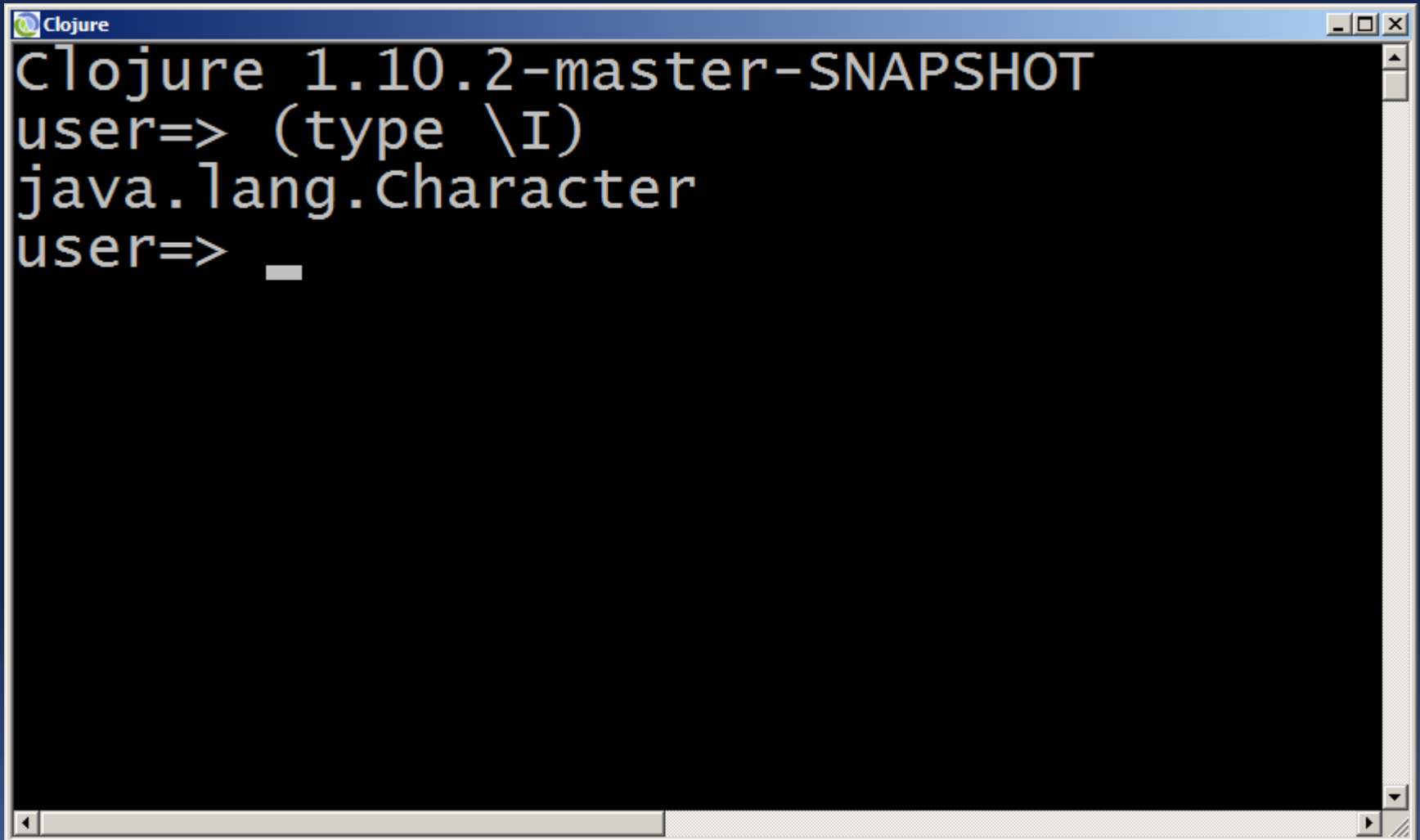
Strings são imutáveis



```
REPL
user=>
user=>
user=>
user=>
user=>
user=> (first "IMT - Instituto Mauá de Tecnologia")
\I
user=>
user=>
user=>
user=>
user=>
user=> (type *1)
java.lang.Character
user=>
user=>
user=>
```



Strings são imutáveis



```
Clojure 1.10.2-master-SNAPSHOT
user=> (type \I)
java.lang.Character
user=> _
```

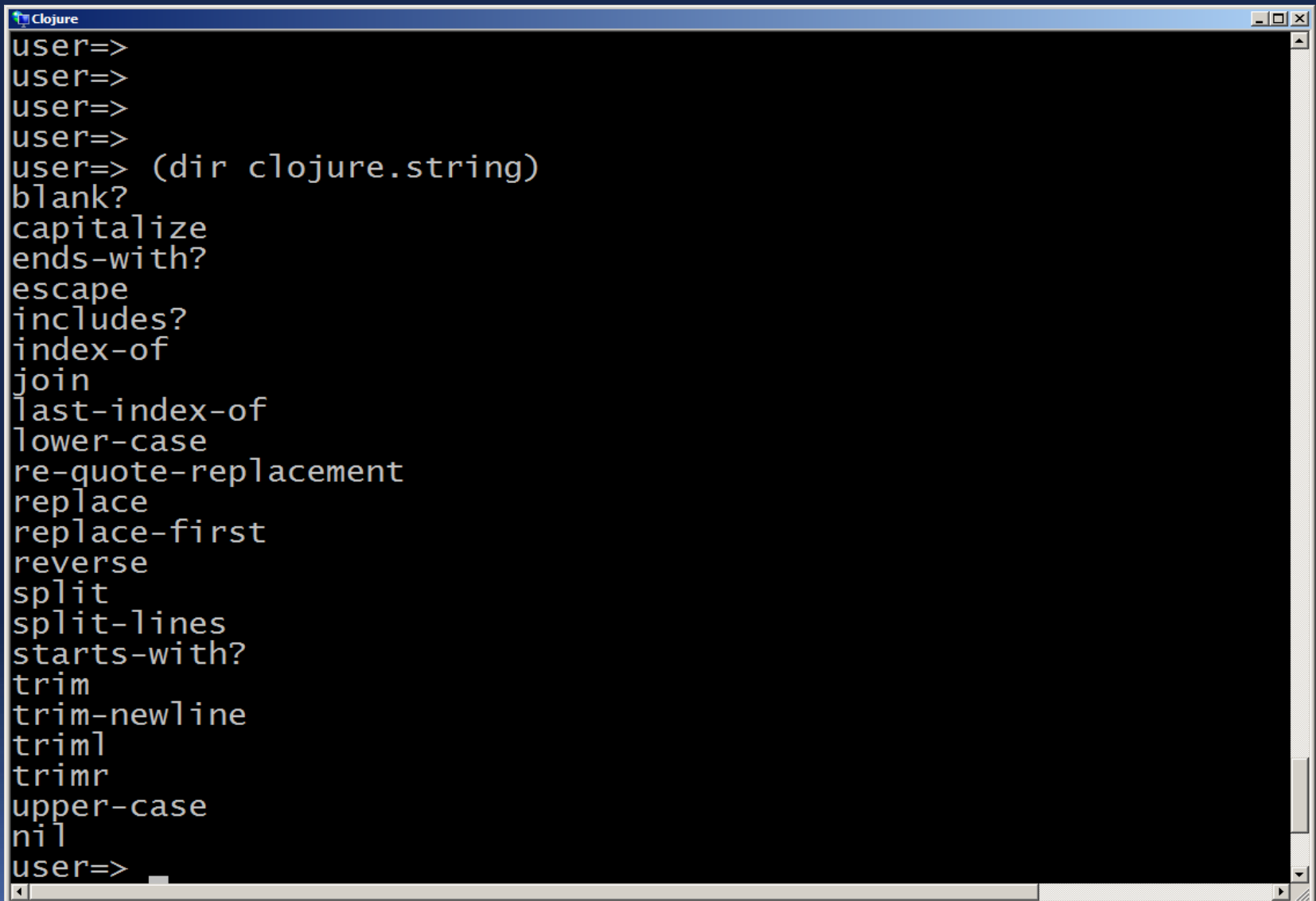


Strings

- ✓ No **namespace core** há diversas funções para manuseio de strings;
- ✓ Mais funções para tratamento de strings podem ser encontradas no **namespace clojure.string**.



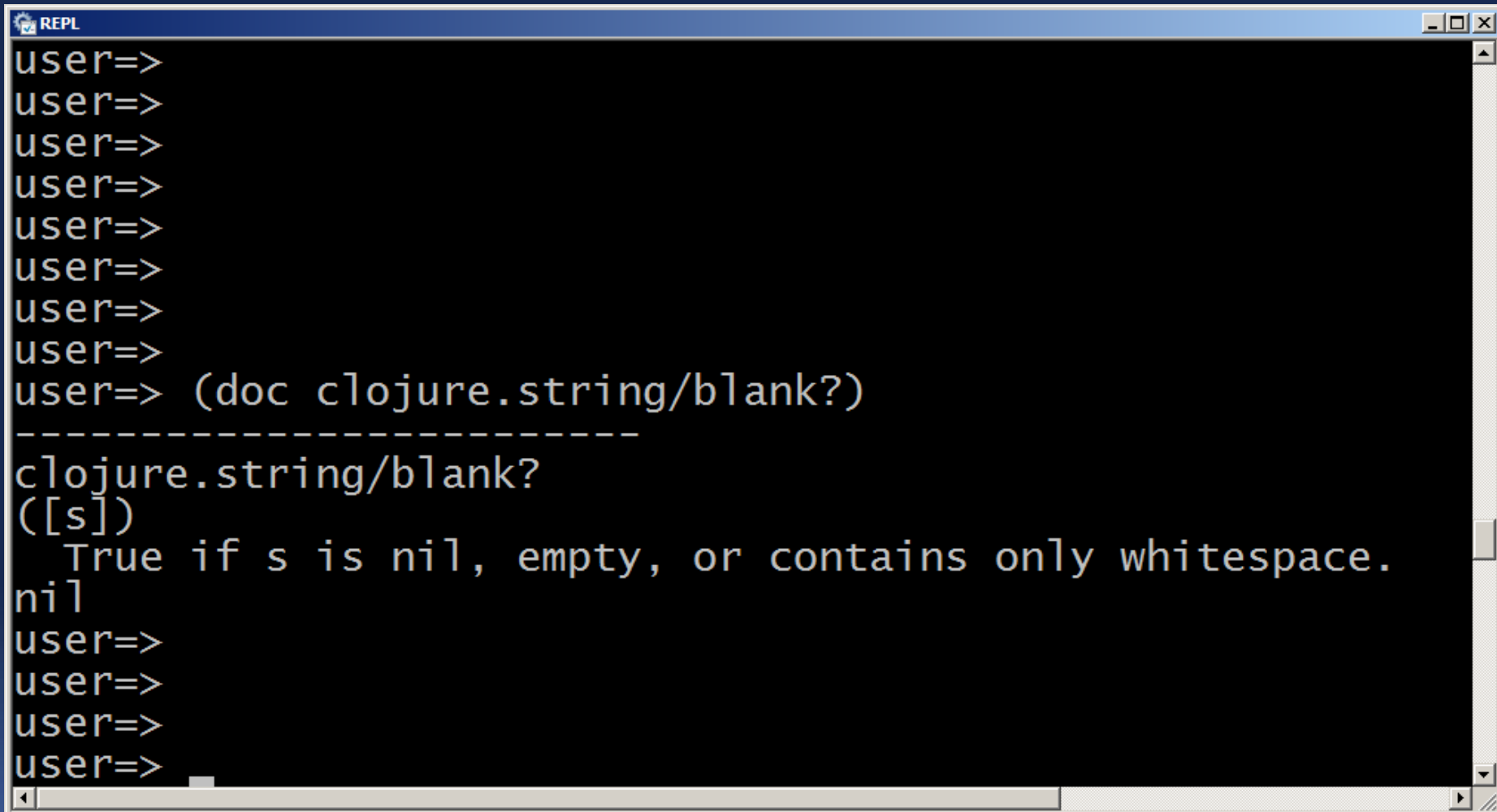
Strings



```
Clojure
user=>
user=>
user=>
user=>
user=> (dir clojure.string)
blank?
capitalize
ends-with?
escape
includes?
index-of
join
last-index-of
lower-case
re-quote-replacement
replace
replace-first
reverse
split
split-lines
starts-with?
trim
trim-newline
triml
trimr
upper-case
nil
user=>
```

Apenas lembrando....

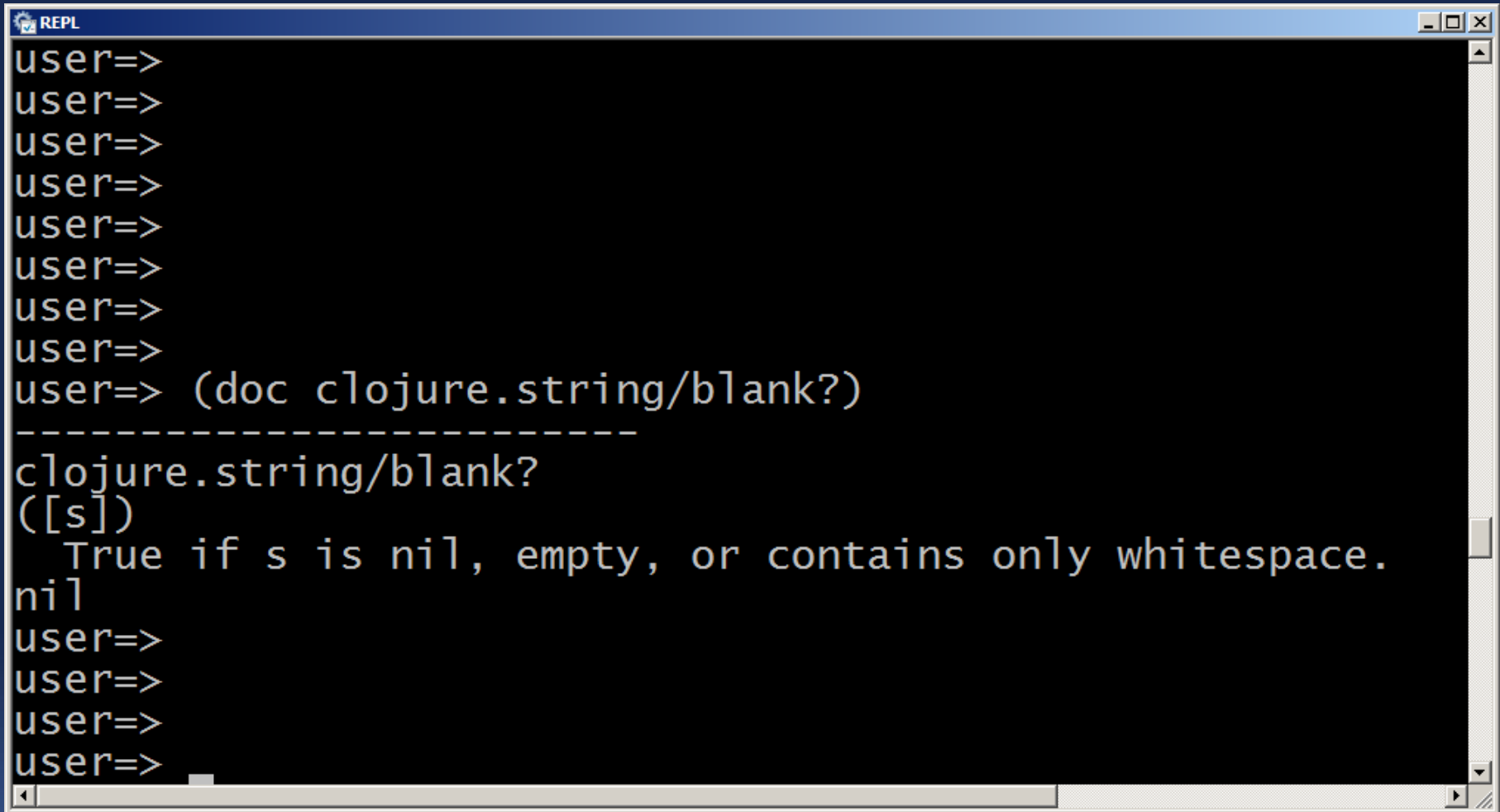
- ✓ O exemplo abaixo mostra como usamos uma função de um **namespace** específico:



```
REPL
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (doc clojure.string/blank?)
-----
clojure.string/blank?
([s])
  True if s is nil, empty, or contains only whitespace.
nil
user=>
user=>
user=>
user=>
```



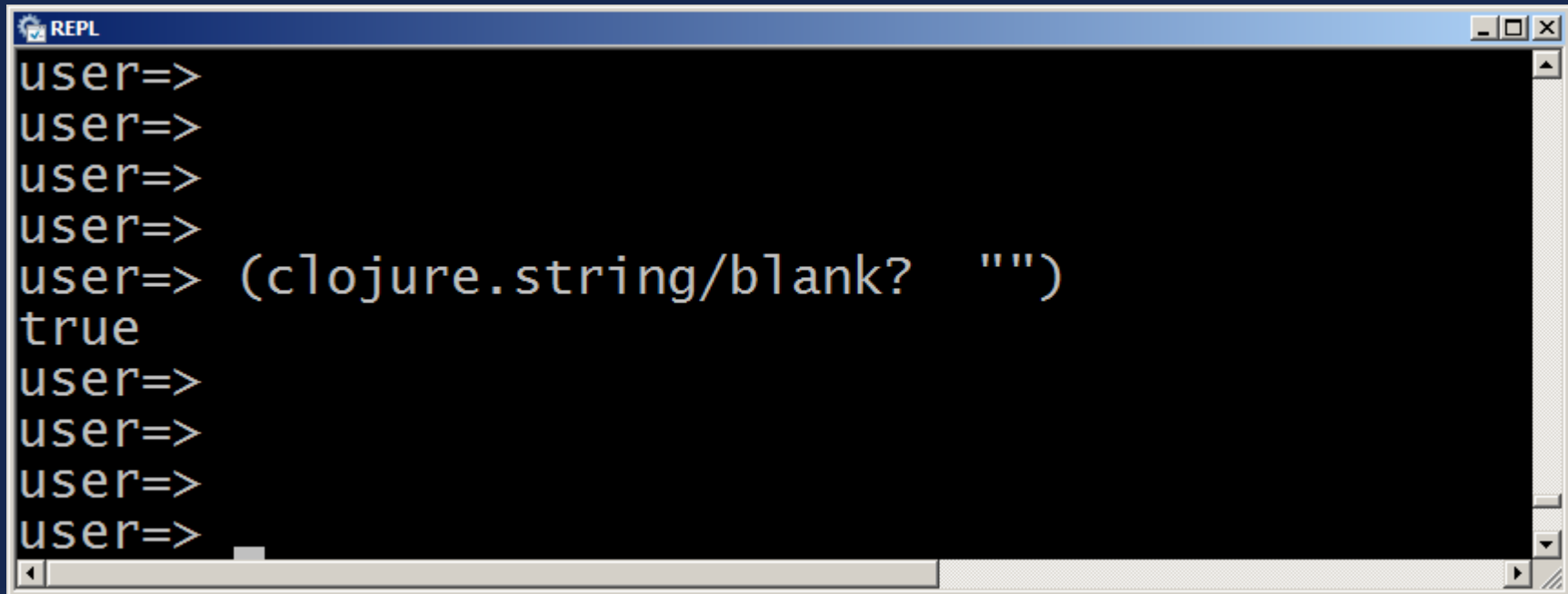
Strings



```
REPL
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (doc clojure.string/blank?)
-----
clojure.string/blank?
([s])
  True if s is nil, empty, or contains only whitespace.
nil
user=>
user=>
user=>
user=>
```

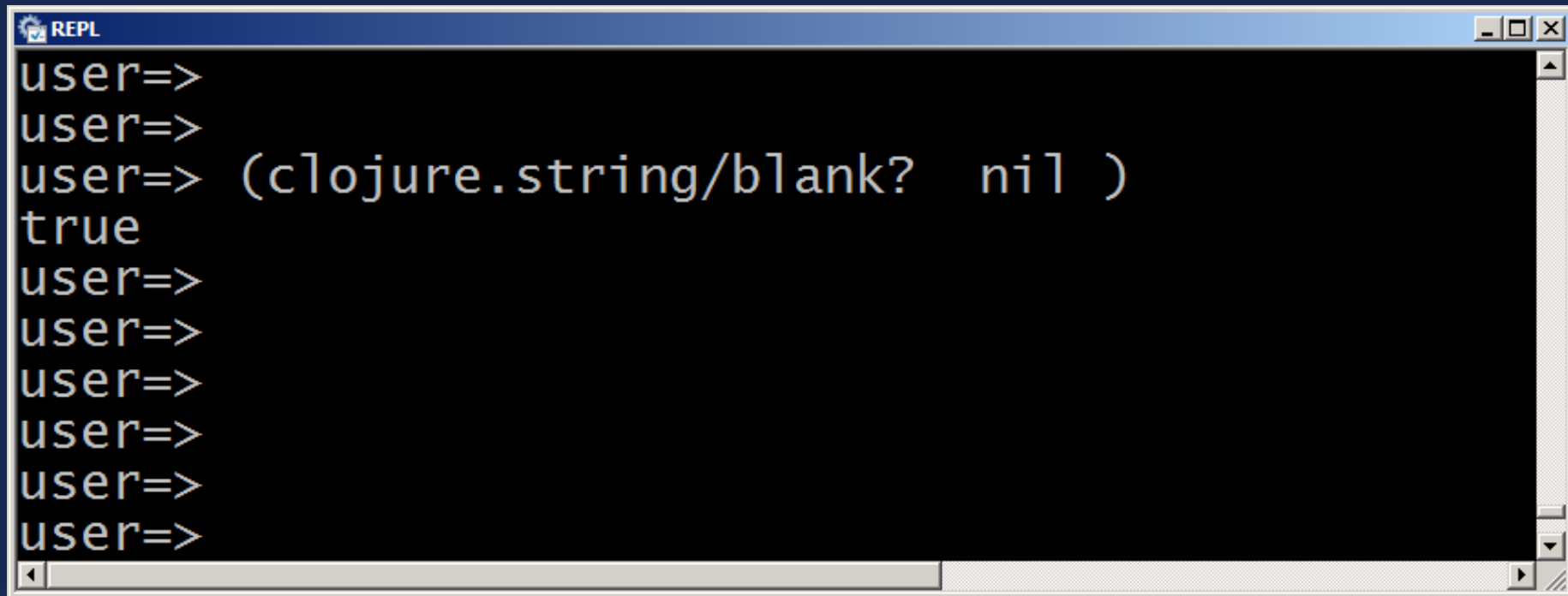


Strings



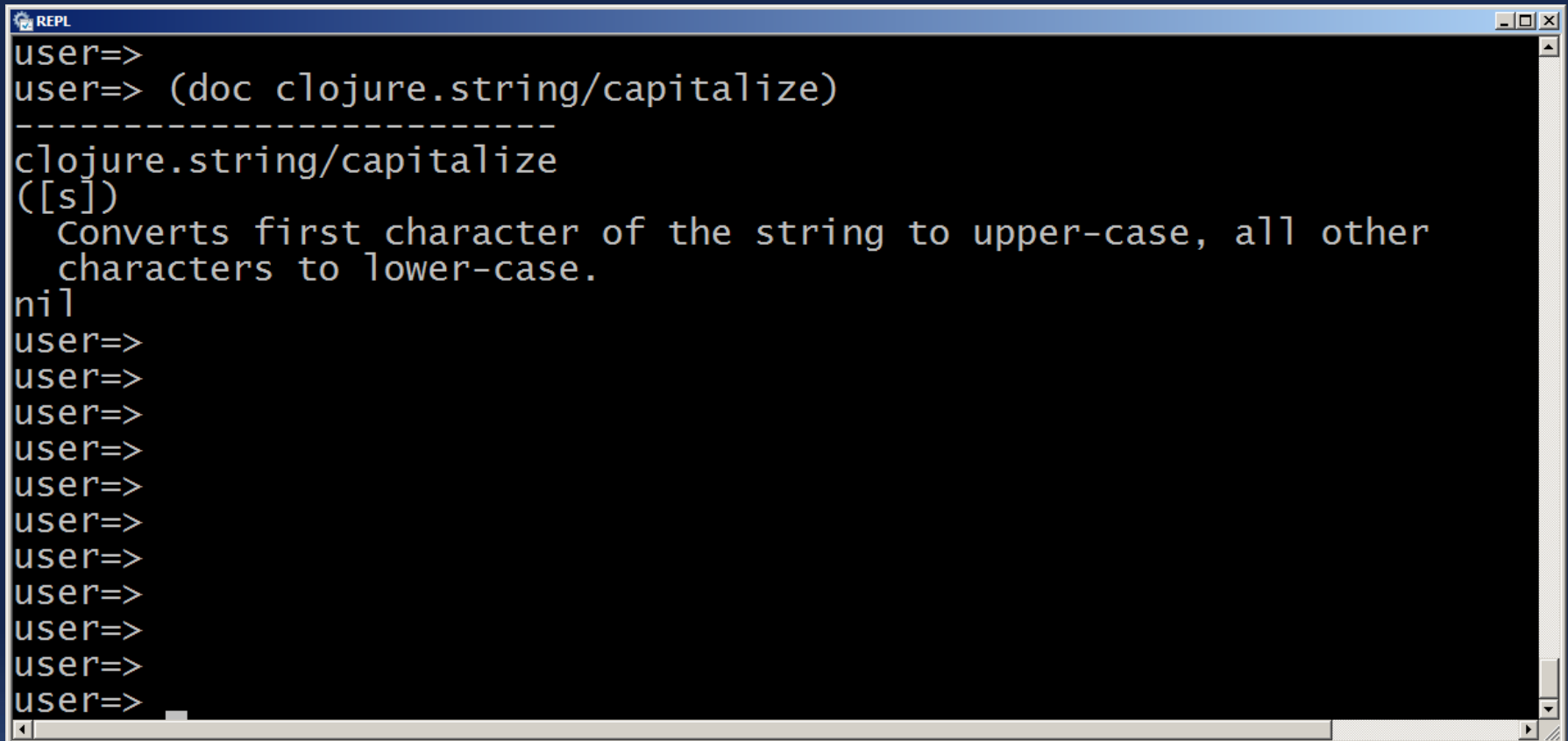
```
REPL
user=>
user=>
user=>
user=>
user=> (clojure.string/blank? "")
true
user=>
user=>
user=>
user=>
```


Strings

A screenshot of a REPL (Read-Eval-Print Loop) window. The window has a title bar with the text 'REPL' and standard window controls (minimize, maximize, close). The main area is black with white text. The text shows a series of prompts 'user=>' followed by the expression '(clojure.string/blank? nil)' and the result 'true'. There are several more 'user=>' prompts below, indicating further input. The window has a scrollbar on the right side.

```
REPL
user=>
user=>
user=> (clojure.string/blank? nil )
true
user=>
user=>
user=>
user=>
user=>
user=>
```

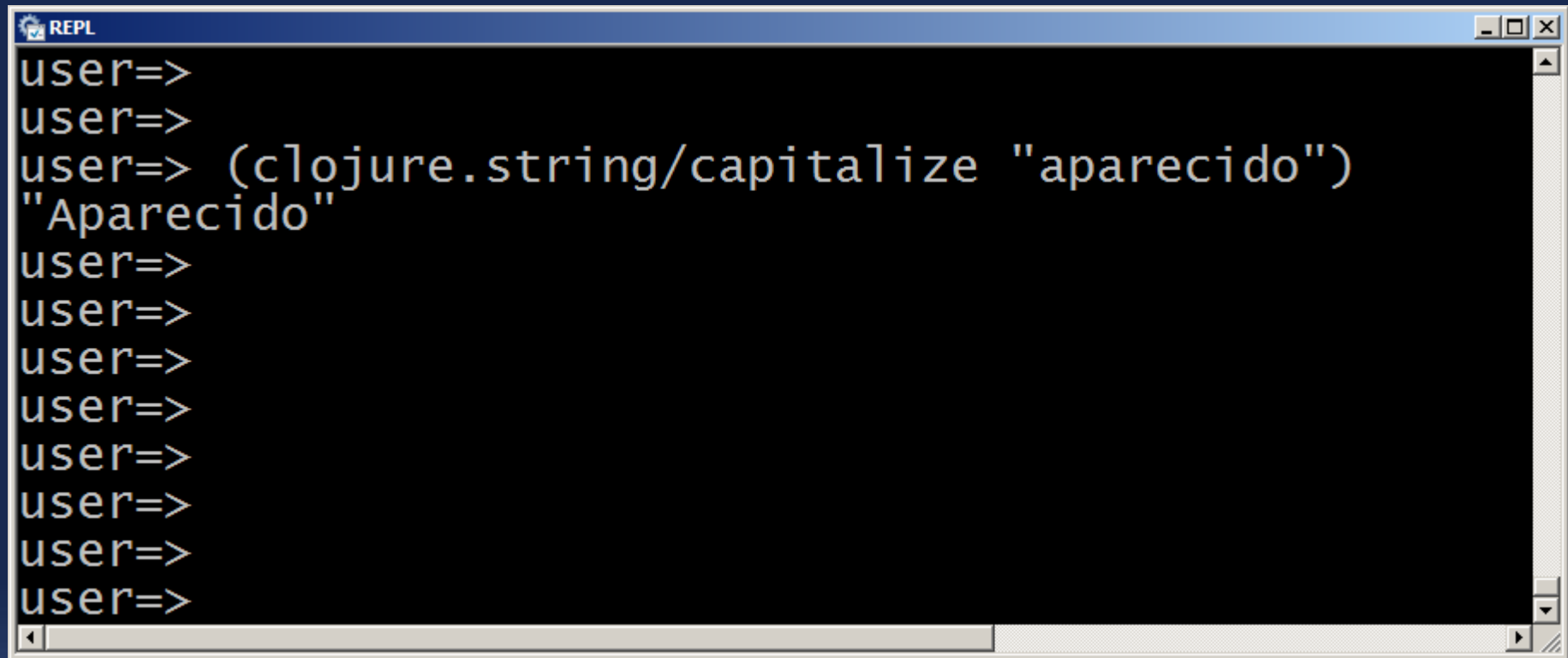
Strings



```
REPL
user=>
user=> (doc clojure.string/capitalize)
-----
clojure.string/capitalize
([s])
  Converts first character of the string to upper-case, all other
  characters to lower-case.
nil
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

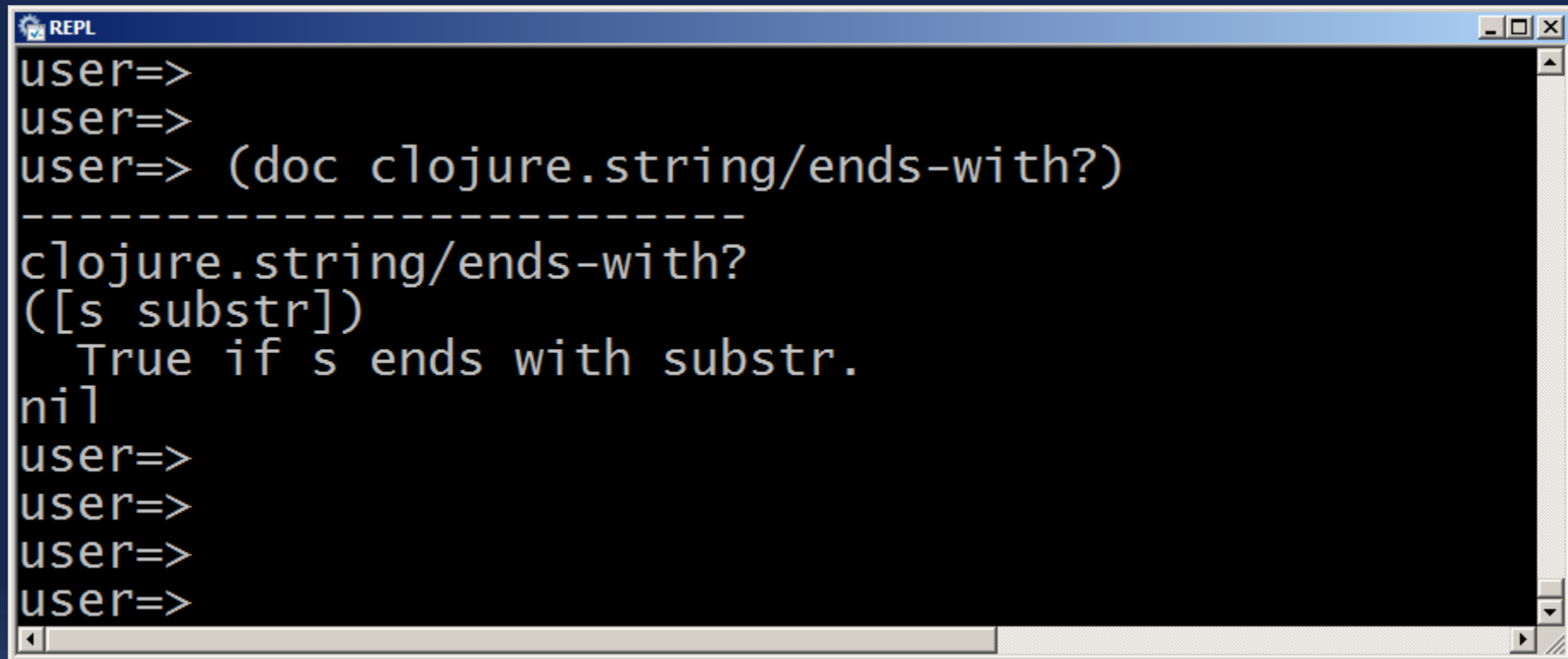


Strings

A screenshot of a REPL (Read-Eval-Print Loop) window titled "REPL". The window has a blue title bar with standard window controls (minimize, maximize, close) on the right. The main area is black with white text. The text shows a series of prompts "user=>" followed by the expression `(clojure.string/capitalize "aparecido")` and its result `"Aparecido"`. There are several empty prompts below the result. A scrollbar is visible on the right side of the text area.

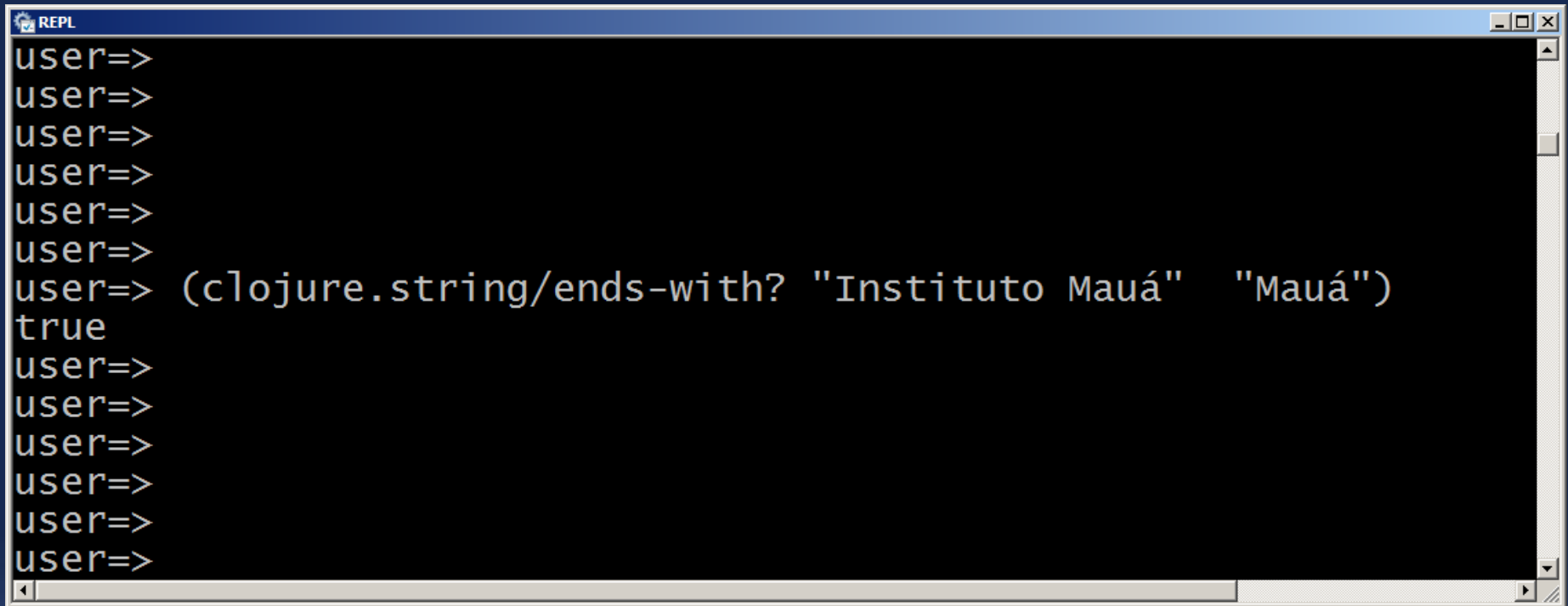
```
user=>
user=>
user=> (clojure.string/capitalize "aparecido")
"Aparecido"
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

Strings



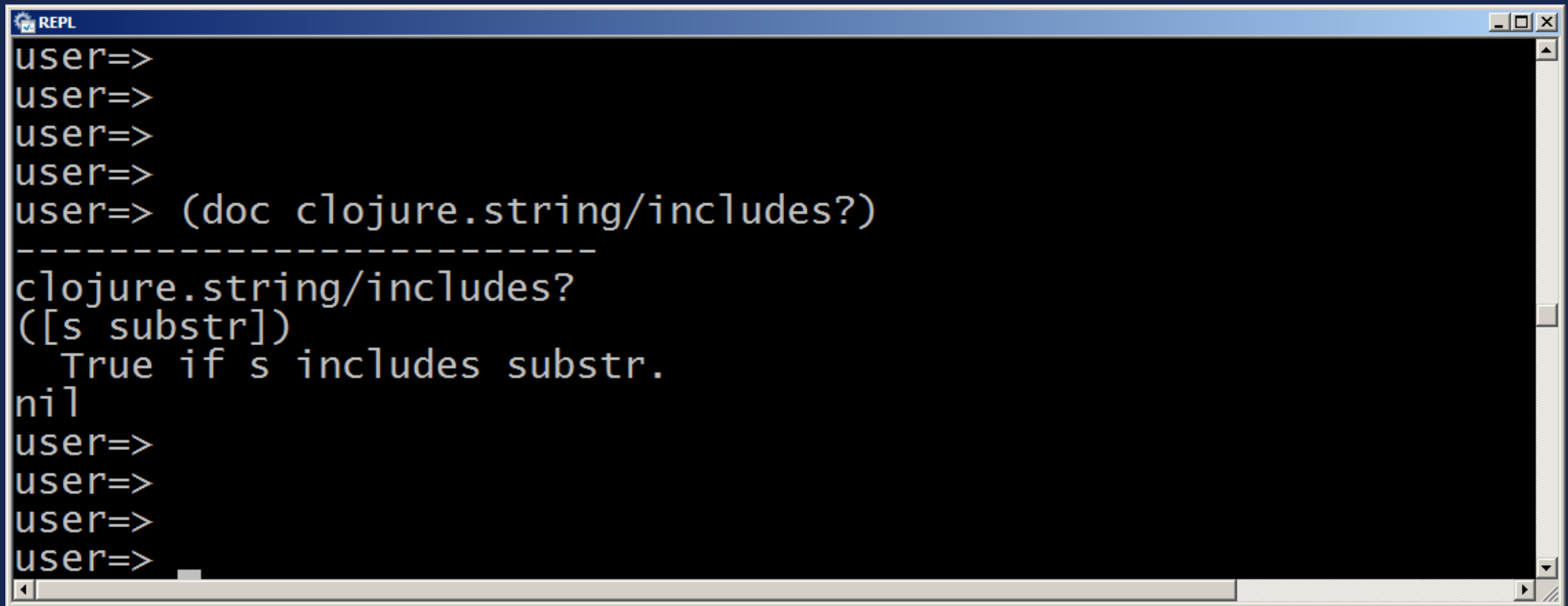
```
REPL
user=>
user=>
user=> (doc clojure.string/ends-with?)
-----
clojure.string/ends-with?
([s substr])
  True if s ends with substr.
nil
user=>
user=>
user=>
user=>
```

Strings

A screenshot of a REPL (Read-Eval-Print Loop) window. The window has a title bar with the text 'REPL' and standard window controls (minimize, maximize, close). The background is black, and the text is white. The text inside the window shows a series of prompts 'user=>' followed by several empty lines, then a Clojure expression '(clojure.string/ends-with? "Instituto Mauá" "Mauá")', and finally the output 'true'.

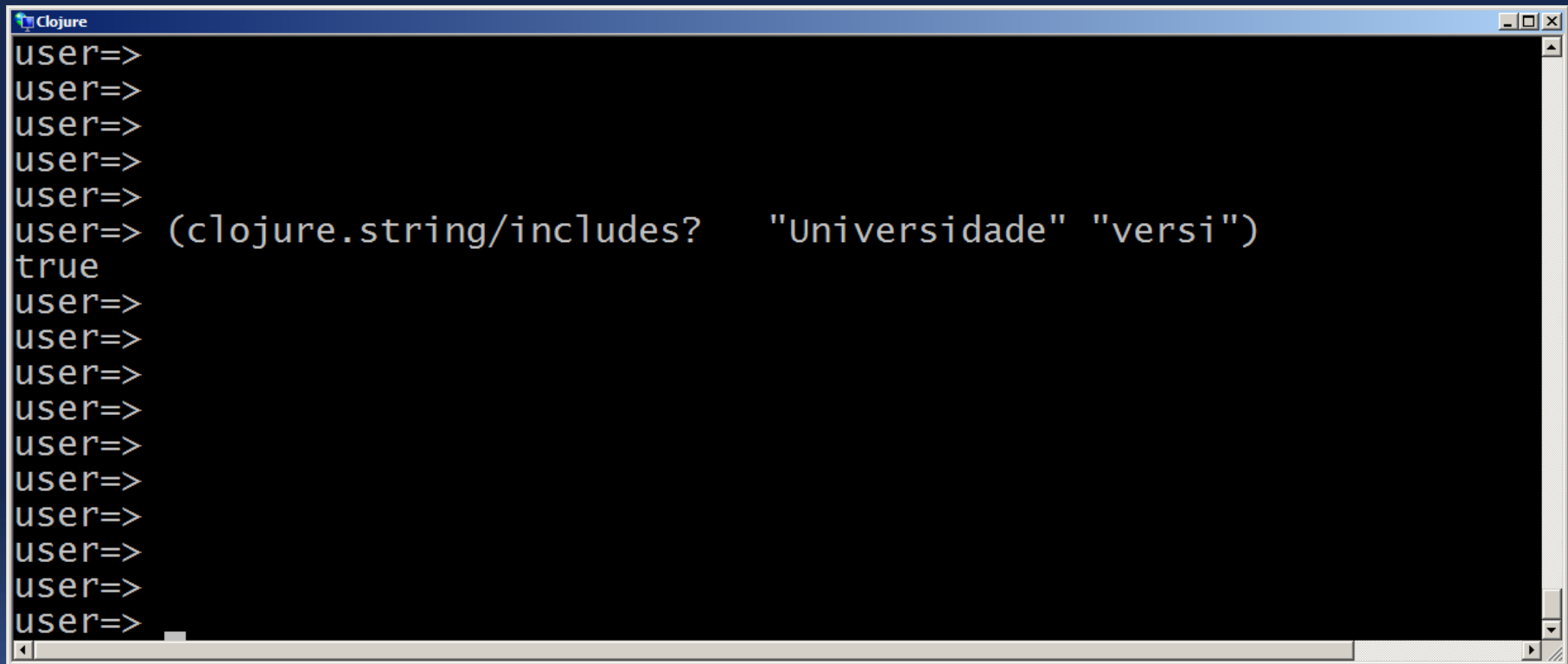
```
REPL
user=>
user=>
user=>
user=>
user=>
user=>
user=> (clojure.string/ends-with? "Instituto Mauá" "Mauá")
true
user=>
user=>
user=>
user=>
user=>
user=>
```

Strings



```
REPL
user=>
user=>
user=>
user=>
user=> (doc clojure.string/includes?)
-----
clojure.string/includes?
([s substr])
  True if s includes substr.
nil
user=>
user=>
user=>
user=>
```

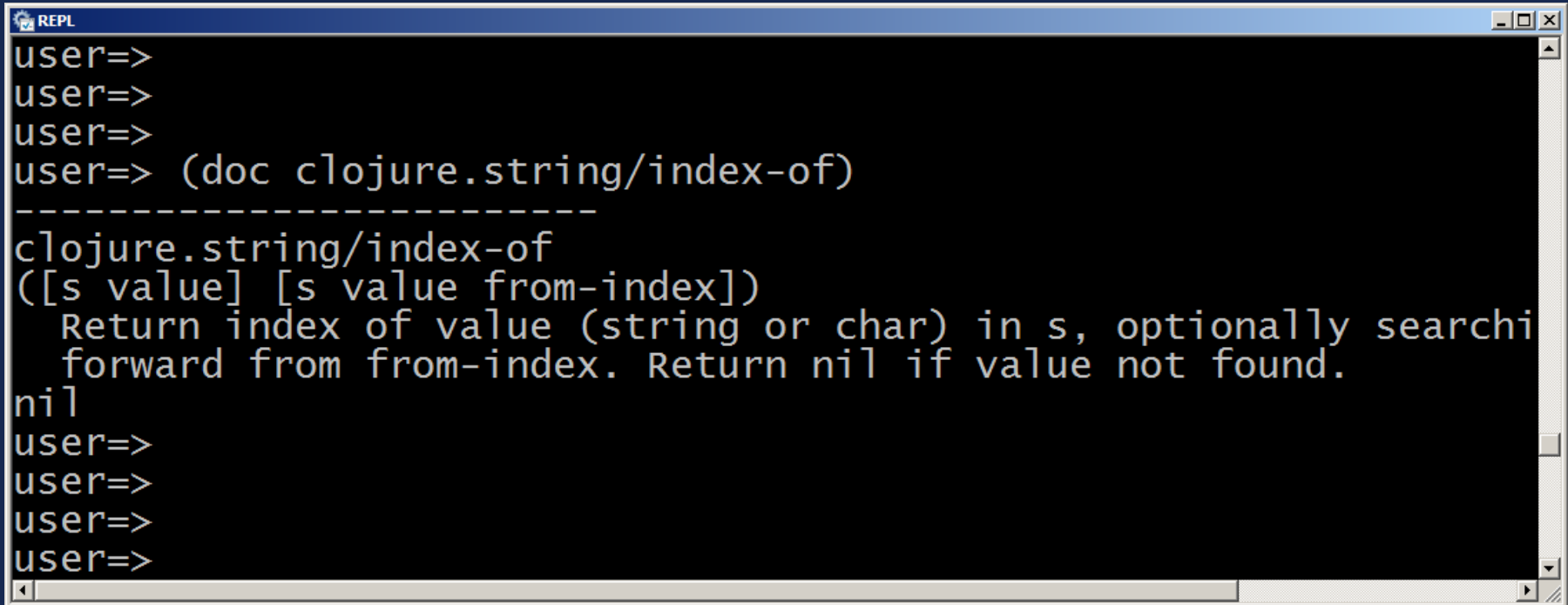
Strings



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (clojure.string/includes? "Universidade" "versi")
true
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```



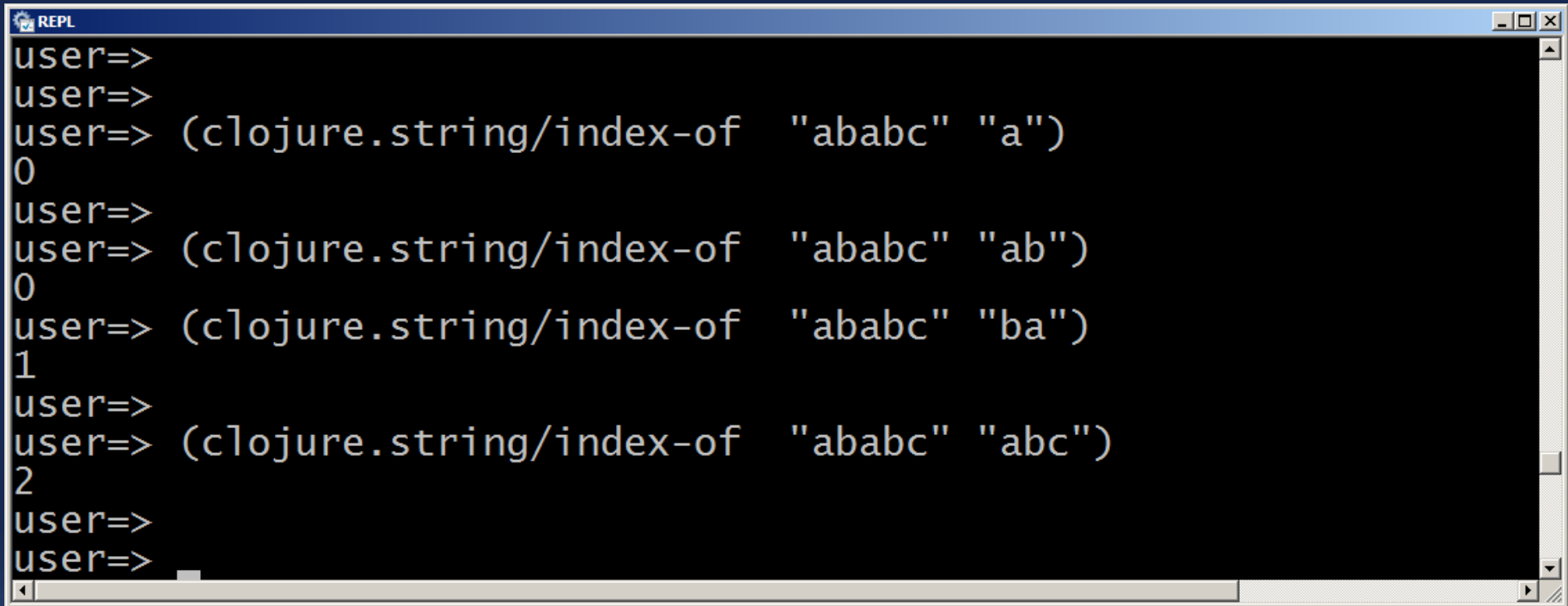
Strings



```
REPL
user=>
user=>
user=>
user=> (doc clojure.string/index-of)
-----
clojure.string/index-of
([s value] [s value from-index])
  Return index of value (string or char) in s, optionally searching
  forward from from-index. Return nil if value not found.
nil
user=>
user=>
user=>
user=>
```



Strings



```
REPL
user=>
user=>
user=> (clojure.string/index-of "ababc" "a")
0
user=>
user=> (clojure.string/index-of "ababc" "ab")
0
user=> (clojure.string/index-of "ababc" "ba")
1
user=>
user=> (clojure.string/index-of "ababc" "abc")
2
user=>
user=>
```



Números

- ✓ Em **Clojure**, por default, números naturais são implementados pelo tipo `java.lang.Long`;
- ✓ Para números muito grandes, números naturais são do tipo `clojure.lang.BigInt`;

A screenshot of a Clojure REPL window titled "Clojure". The window has a black background with white text. It shows a series of user prompts "user=>" followed by various inputs and outputs. The inputs include "(type 1)", "java.lang.Long", "(type 10000000000000000000000000000000)", and "clojure.lang.BigInt". The output for the last input is "clojure.lang.BigInt".

```
Clojure
user=>
user=>
user=>
user=> (type 1)
java.lang.Long
user=>
user=>
user=>
user=> (type 10000000000000000000000000000000)
clojure.lang.BigInt
user=>
user=>
user=>
```



Números Racionais

- ✓ Em **Clojure**, números expressos por razões inexatas, são do tipo **"Ratio"**

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> 5/4
5/4
user=>
user=>
user=> (/ 4 5)
4/5
user=>
user=>
user=>
user=> (/ 4 4 )
1
user=>
user=>
user=>
user=> 4/4
1
user=>
user=>
user=>
```



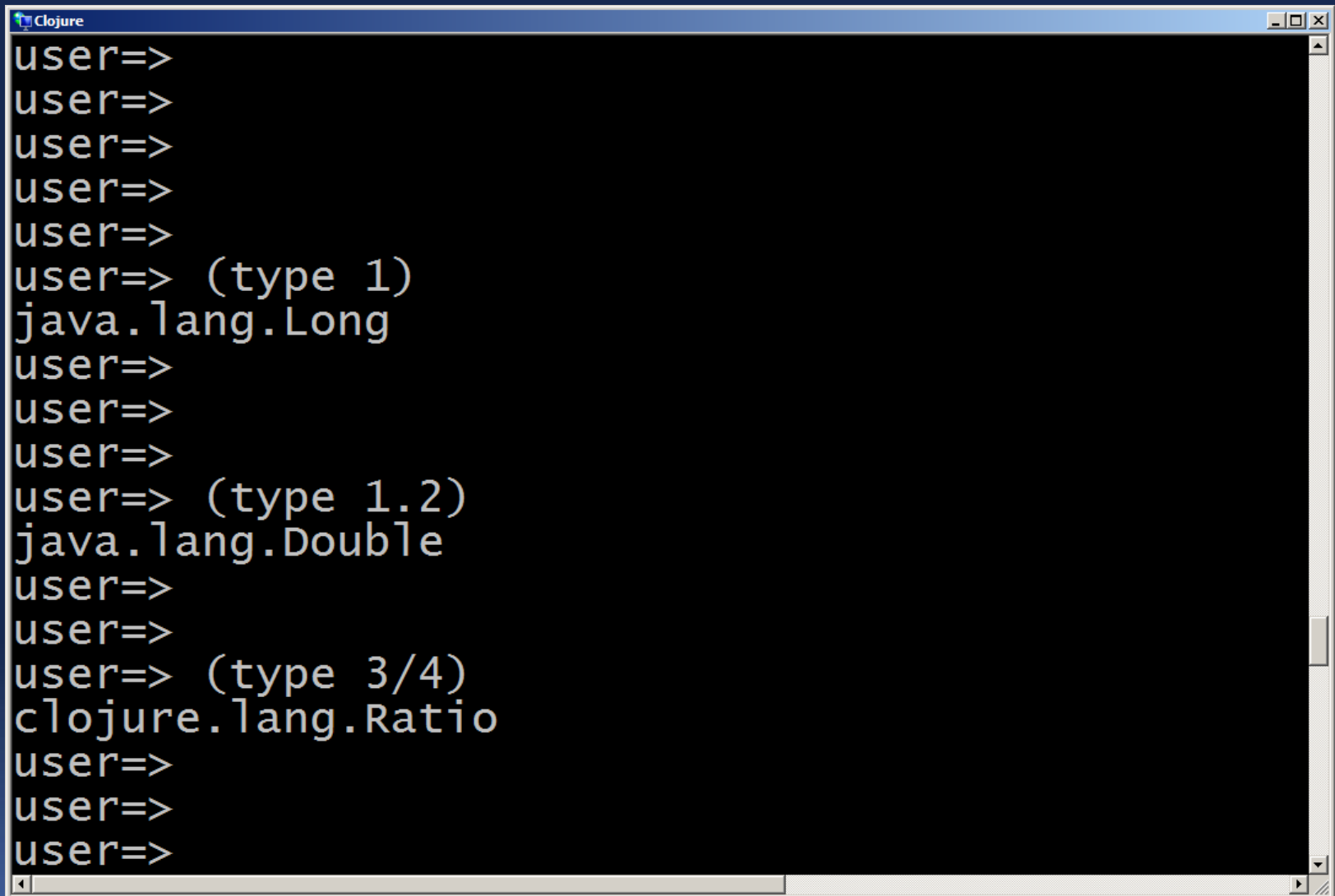
Números Decimais

- ✓ Em **Clojure**, números decimais são números de precisão dupla ("double").

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> 1.2
1.2
user=>
user=>
user=>
user=> (/ 3 4.0)
0.75
user=>
user=>
user=> (* 1.0 2)
2.0
user=>
user=>
user=>
user=>
user=>
```



Função type



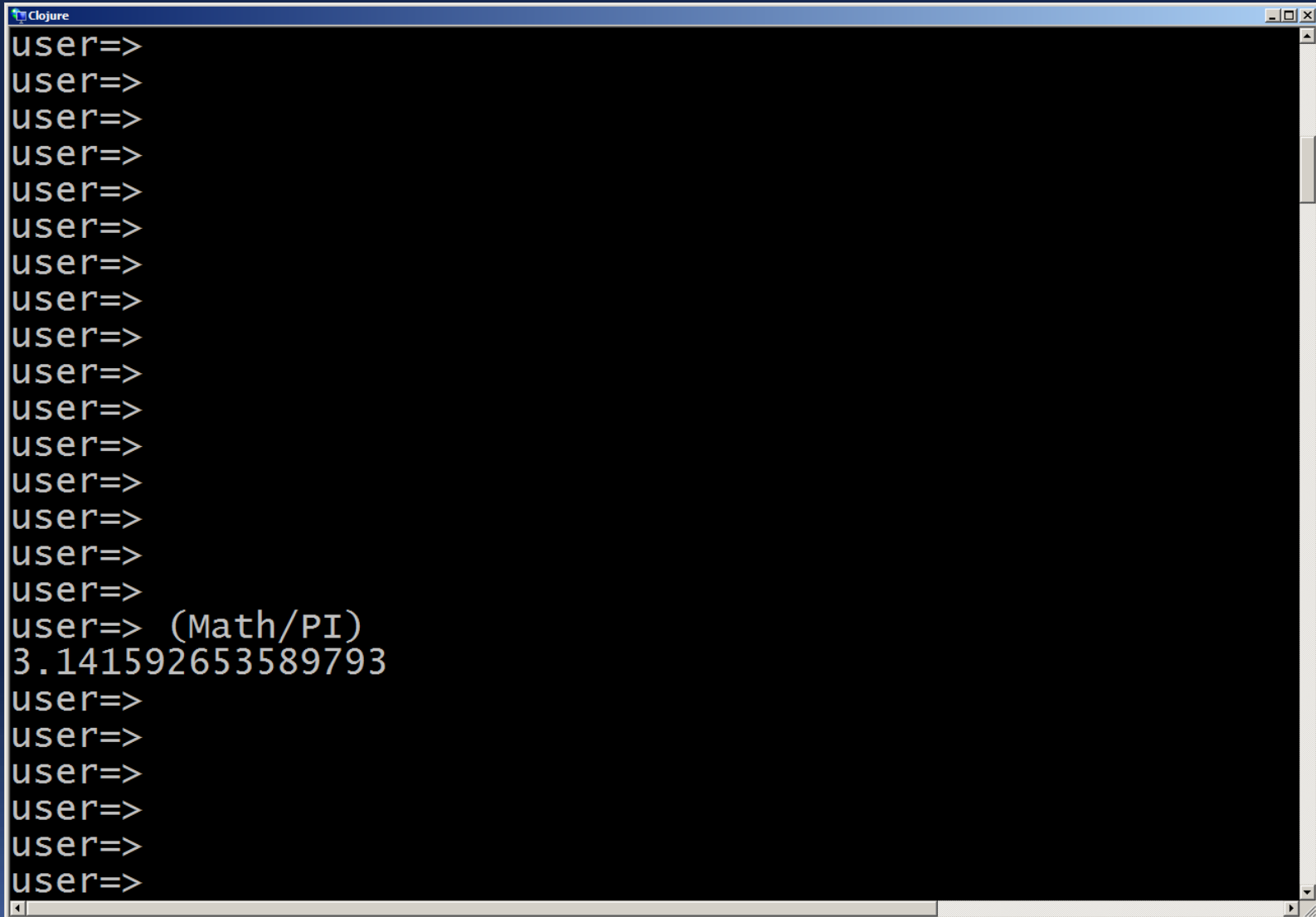
```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (type 1)
java.lang.Long
user=>
user=>
user=>
user=> (type 1.2)
java.lang.Double
user=>
user=>
user=> (type 3/4)
clojure.lang.Ratio
user=>
user=>
user=>
```



Função type

```
Clojure 1.10.2-master-SNAPSHOT
user=>
user=>
user=>
user=> (type "Hello World...")
java.lang.String
user=>
user=> (type 1.2)
java.lang.Double
user=>
user=> (type nil)
nil
user=>
user=> (type true)
java.lang.Boolean
user=>
user=> (type 4/4)
java.lang.Long
user=>
user=> (type 3/4)
clojure.lang.Ratio
user=>
user=> _
```

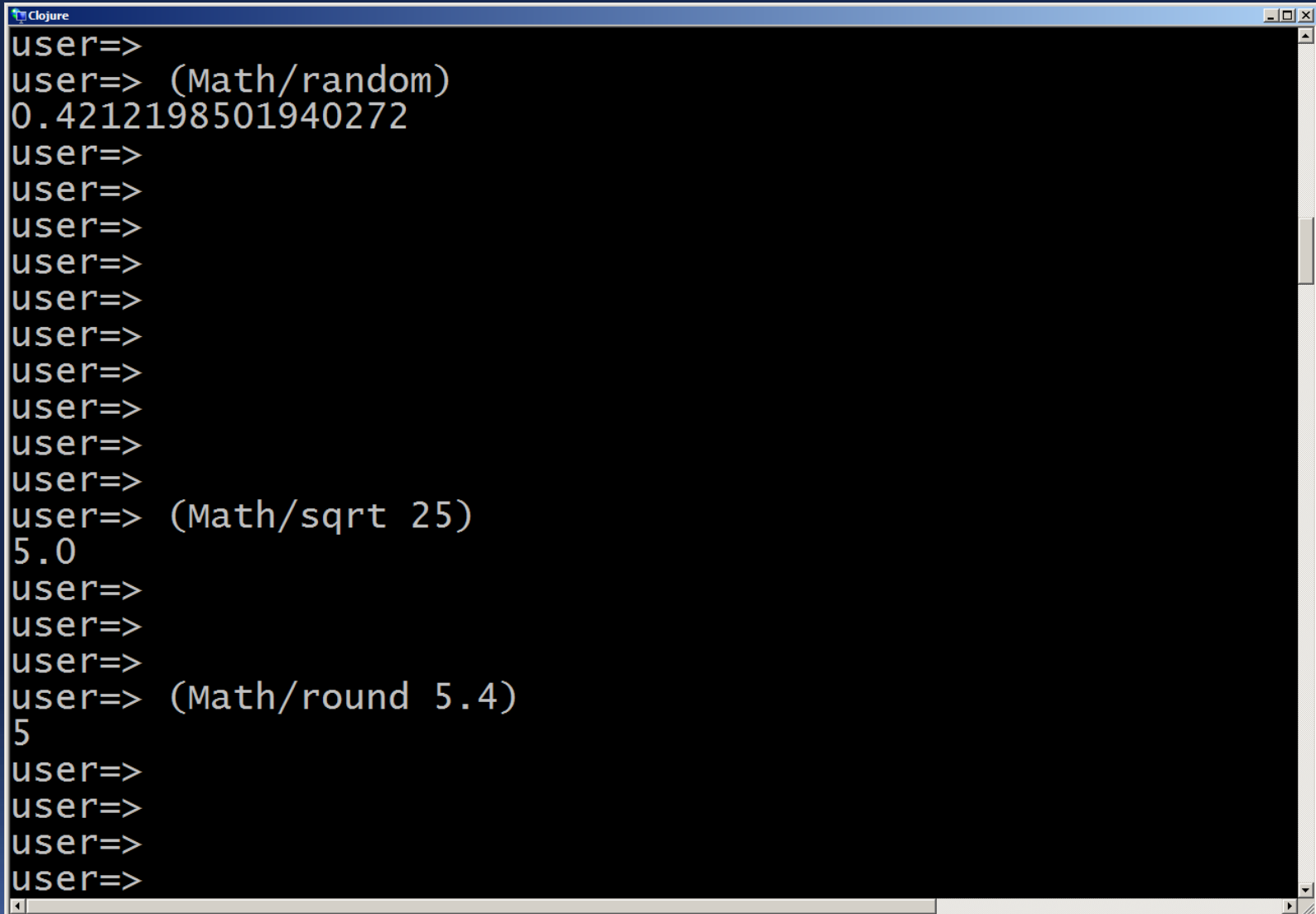

Lendo um valor constante da library Math



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (Math/PI)
3.141592653589793
user=>
user=>
user=>
user=>
user=>
user=>
```



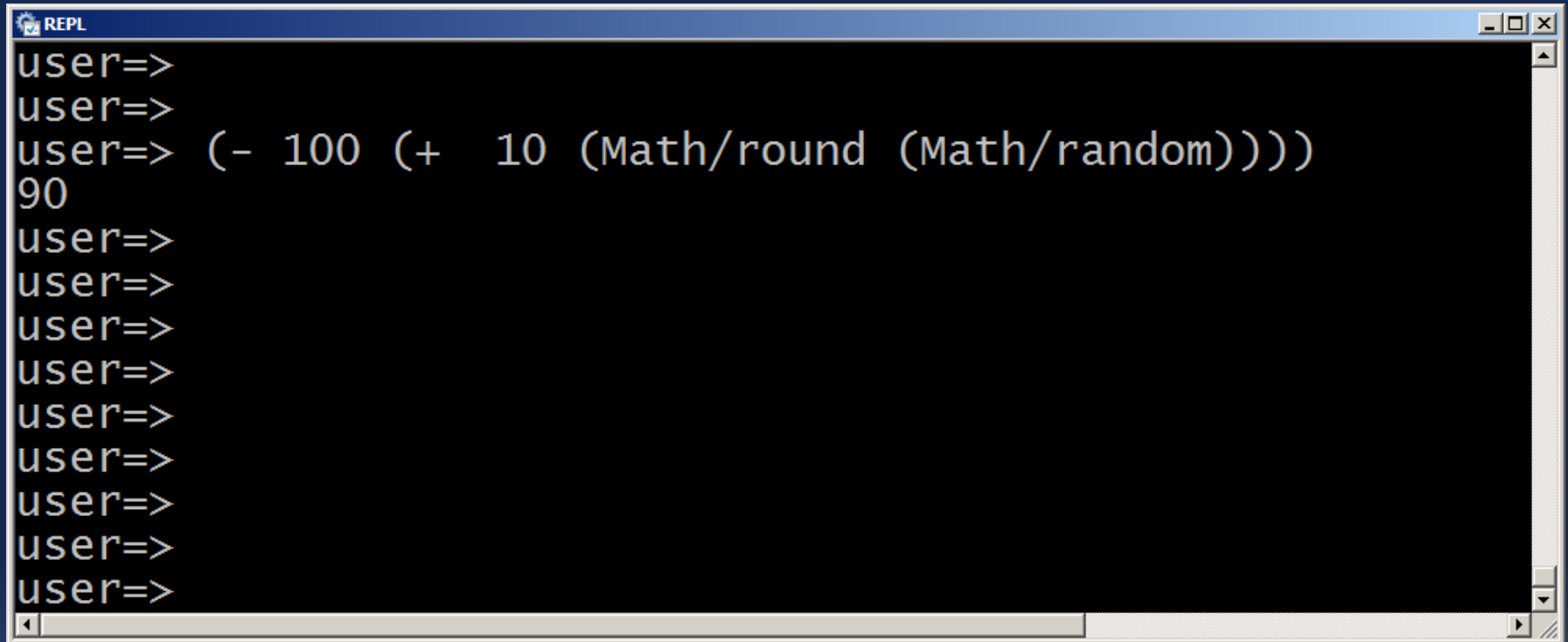
Chamando uma função da library Math



```
Clojure
user=>
user=> (Math/random)
0.4212198501940272
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (Math/sqrt 25)
5.0
user=>
user=>
user=>
user=> (Math/round 5.4)
5
user=>
user=>
user=>
user=>
```



Chamando uma função da library Math



```
REPL
user=>
user=>
user=> (- 100 (+ 10 (Math/round (Math/random))))
90
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

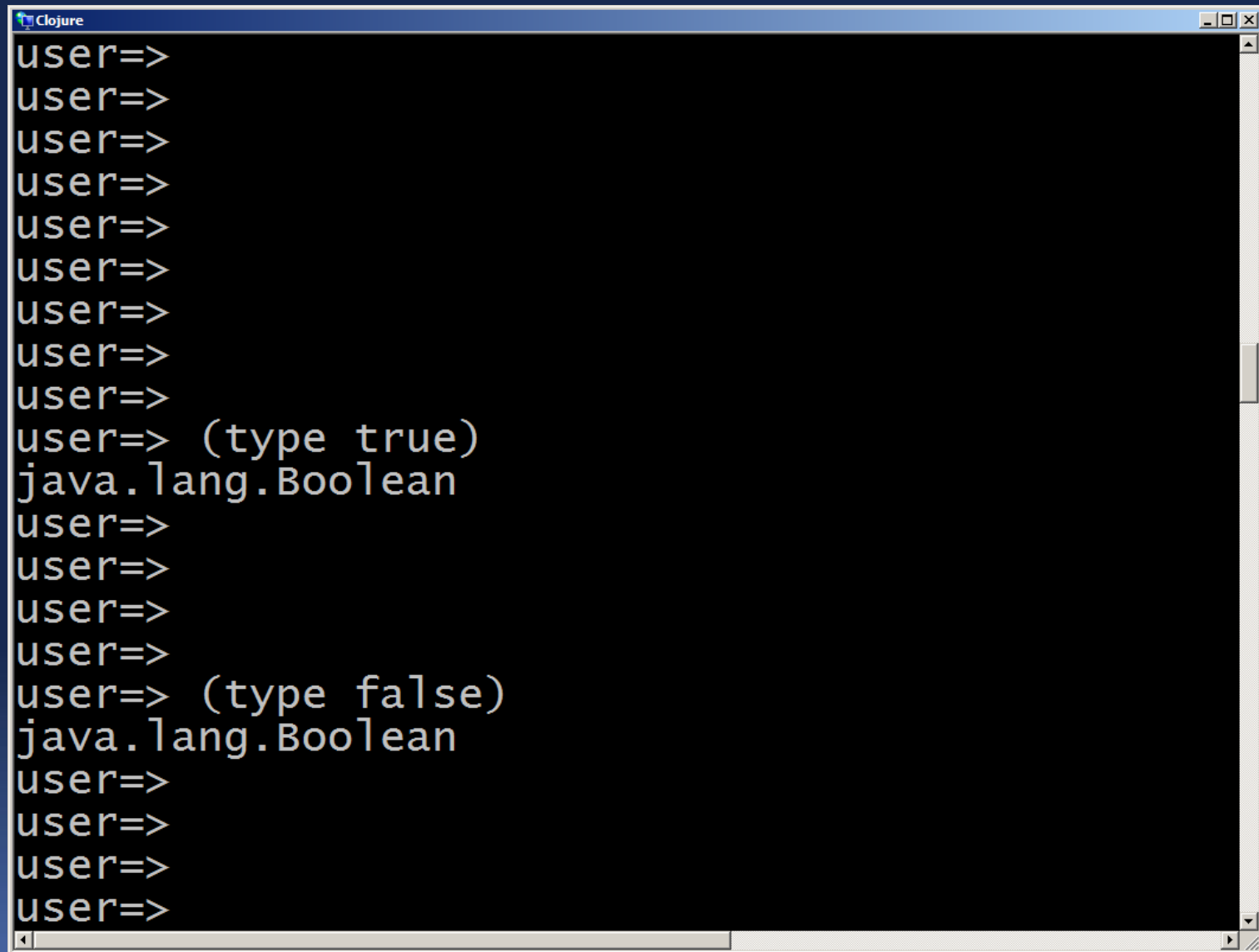


Valores Booleanos

- ✓ Em **Clojure**, valores boolean são implementados com o tipo **java.lang.Boolean**;
- ✓ Esse tipo de dados pode assumir os valores **true** ou **false** e suas representações literais são **true** e **false** (caracteres minúsculos).



Valores Booleanos



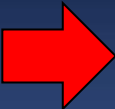
```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (type true)
java.lang.Boolean
user=>
user=>
user=>
user=> (type false)
java.lang.Boolean
user=>
user=>
user=>
user=>
```

Símbolos

- ✓ Em **Clojure**, símbolos são identificadores que se referem à alguma "coisa".
- ✓ Já os utilizamos na criação de bindings (ligações) e funções.
- ✓ Quando usamos **def**, o primeiro argumento é um símbolo o qual estará associado a algum valor.



Símbolos



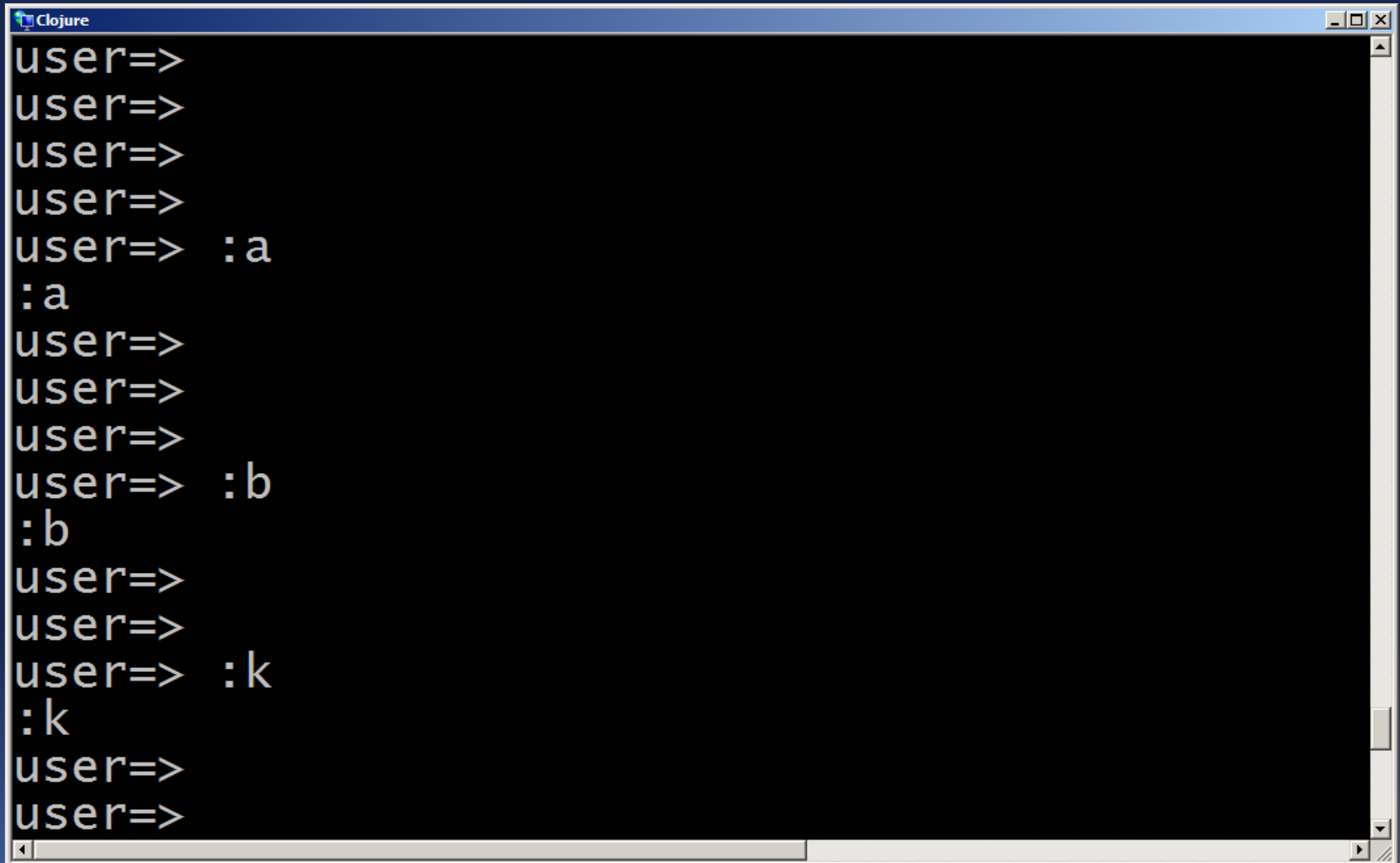
```
Clojure
user=>
user=>
user=>
user=> (def foo "bar")
#'user/foo
user=>
user=>
user=>
user=> foo
"bar"
user=>
user=>
user=> (defn add-2 [x] (+ x 2) )
#'user/add-2
user=>
user=>
user=> add-2
#object[user$add_2 0x640f11a1 "user$add_2@640f11a1"]
user=>
user=>
user=>
```


Keywords

- ✓ Em Clojure, **keywords** correspondem à constantes especiais string;
- ✓ São definidas simplesmente anexando-se **:** no início da palavra que será tratada como keyword;
- ✓ São tipicamente usadas como **chaves** em um mapeamento **associativo** do tipo **chave-valor**.



Keywords



```
Clojure
user=>
user=>
user=>
user=>
user=> :a
:a
user=>
user=>
user=>
user=> :b
:b
user=>
user=>
user=> :k
:k
user=>
user=>
```



Coleções

- ✓ Na programação funcional, e mais especificamente em Clojure, trabalha-se com poucos tipos de dados;
- ✓ **Collections** são tipos de dados que podem conter outros tipos de dados e descrevem a forma pela qual esses itens de dados se relacionam mutuamente;
- ✓ As quatro principais estruturas de dados para Collections são: **Maps**, **Sets**, **Vectors** e **Lists**.



Coleções

- ✓ Na programação funcional, e mais especificamente em Clojure, trabalha-se com poucos tipos de dados;
- ✓ **Collections** são tipos de dados que podem conter outros tipos de dados e descrevem a forma pela qual esses itens de dados se relacionam mutuamente;
- ✓ As quatro principais estruturas de dados para Collections são: **Maps**, **Sets**, **Vectors** e **Lists**.

"Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming." – Rob Pike's Rule #5 of programming.

“Os dados dominam. Se você definiu a estrutura de dados correta e as organizou convenientemente, os algoritmos serão quase sempre evidentes e naturais. Regra #5 de Programação de Rob Pike”.



Maps



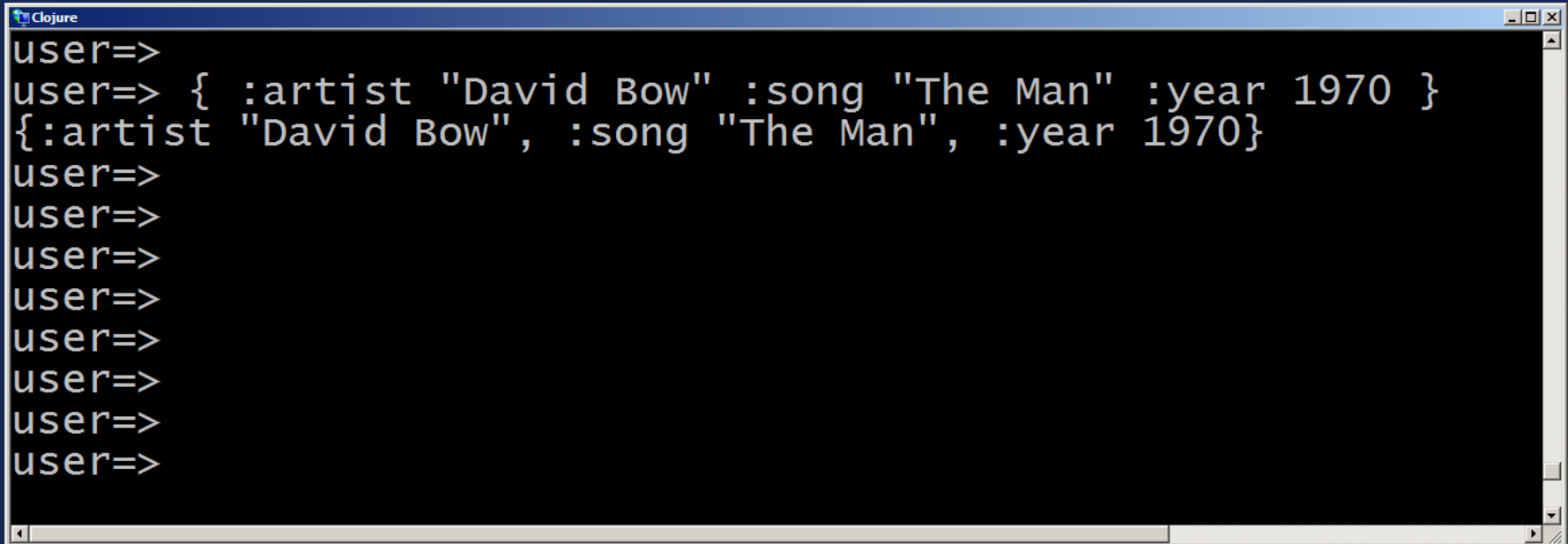
Maps

- ✓ Um **Map** é uma coleção de pares **key-value**;
- ✓ Clojure provê - de forma persistente e imutável - o usual **HashMap** mas também um **SortedMap**;
- ✓ **HashMaps** são chamados "**Hash**" porque eles criam um hash da chave e a mapeia para um dado valor;
- ✓ Consultas, bem como outras operações comuns (**insert** e **delete**) são eficientes;
- ✓ Pode-se criar um **HashMap** com a notação literal **{ }**.



Maps

- ✓ Segue um exemplo de um Map com três pares **key-value**:



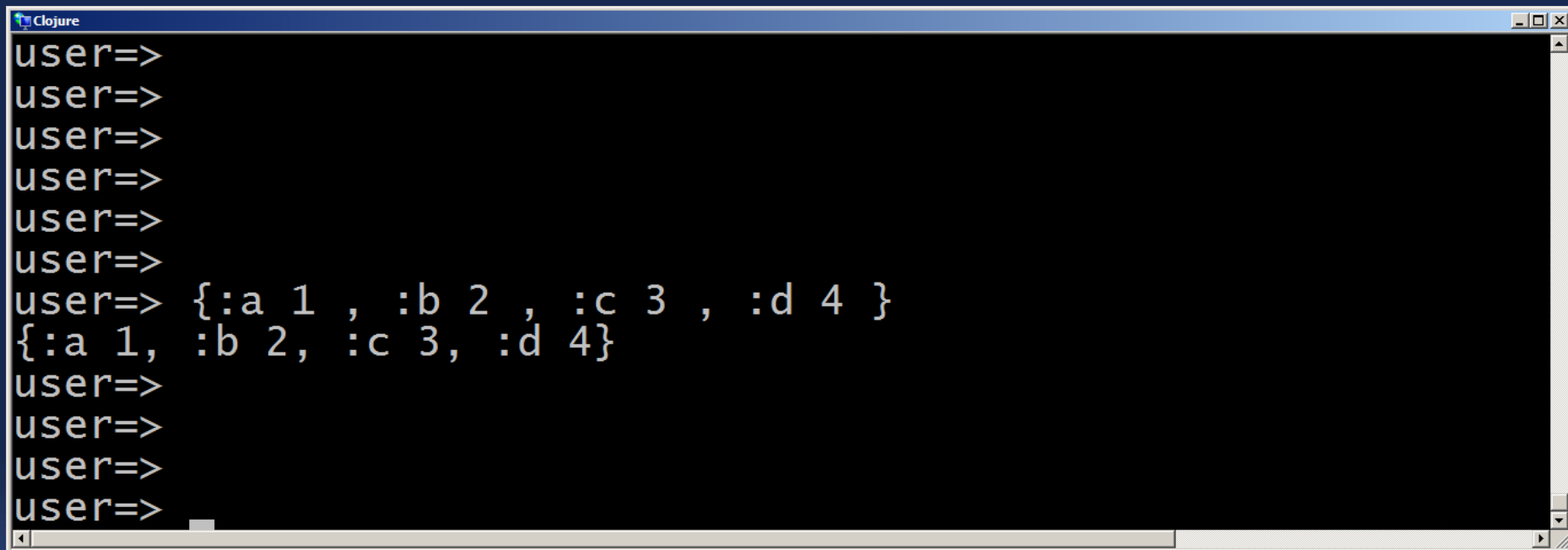
```
Clojure
user=>
user=> { :artist "David Bow" :song "The Man" :year 1970 }
{:artist "David Bow", :song "The Man", :year 1970}
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

- ✓ A definição de um **Map** é feita com espaços entre um **key-value** e outro;
- ✓ Porém **Clojure** avalia o **Map** retornando os pares **separados** por **vírgula**.



Maps

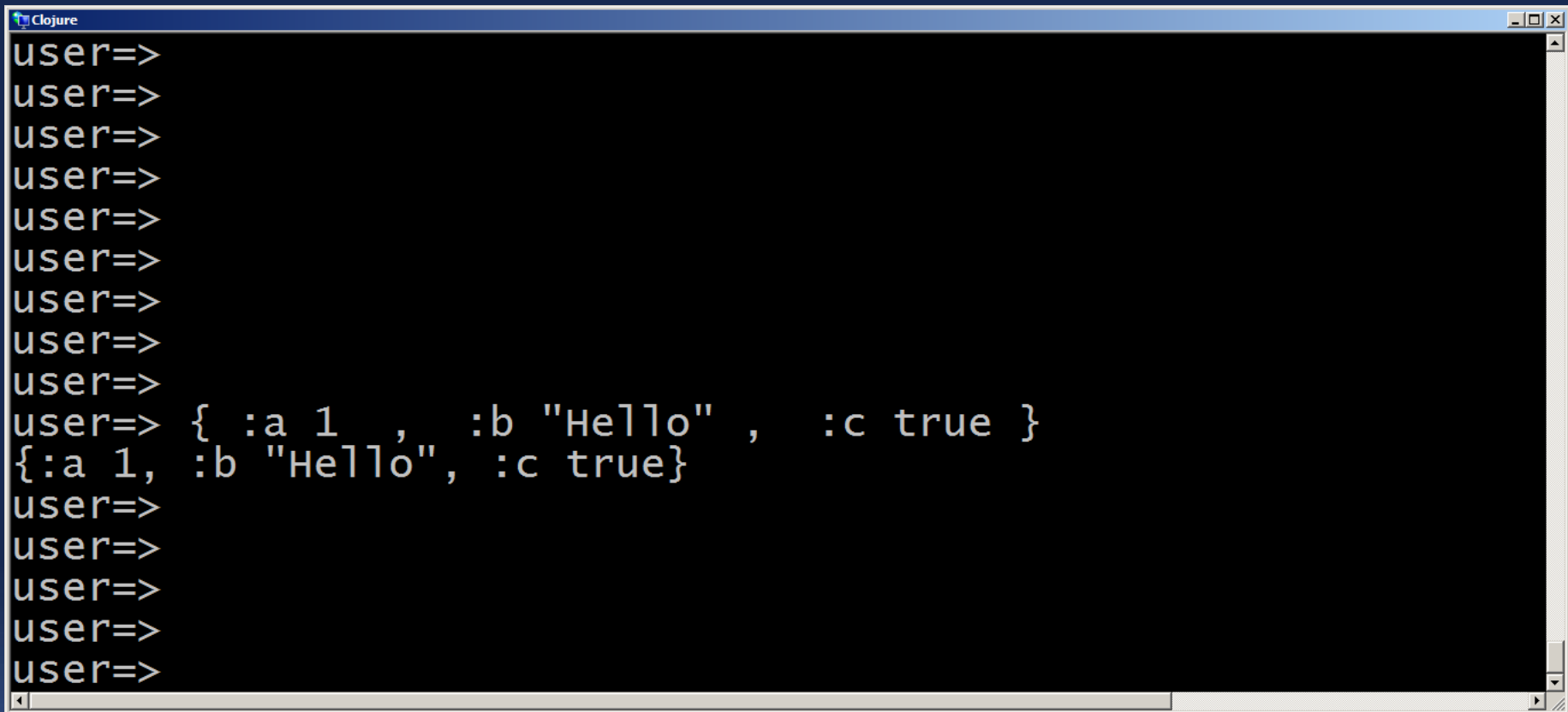
- ✓ Porém, pode-se definir um **Map** separando-se também os pares entre vírgulas.



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> {:a 1 , :b 2 , :c 3 , :d 4 }
{:a 1, :b 2, :c 3, :d 4}
user=>
user=>
user=>
user=>
```


Maps

✓ Os pares **key-value** podem ter tipos diferentes;

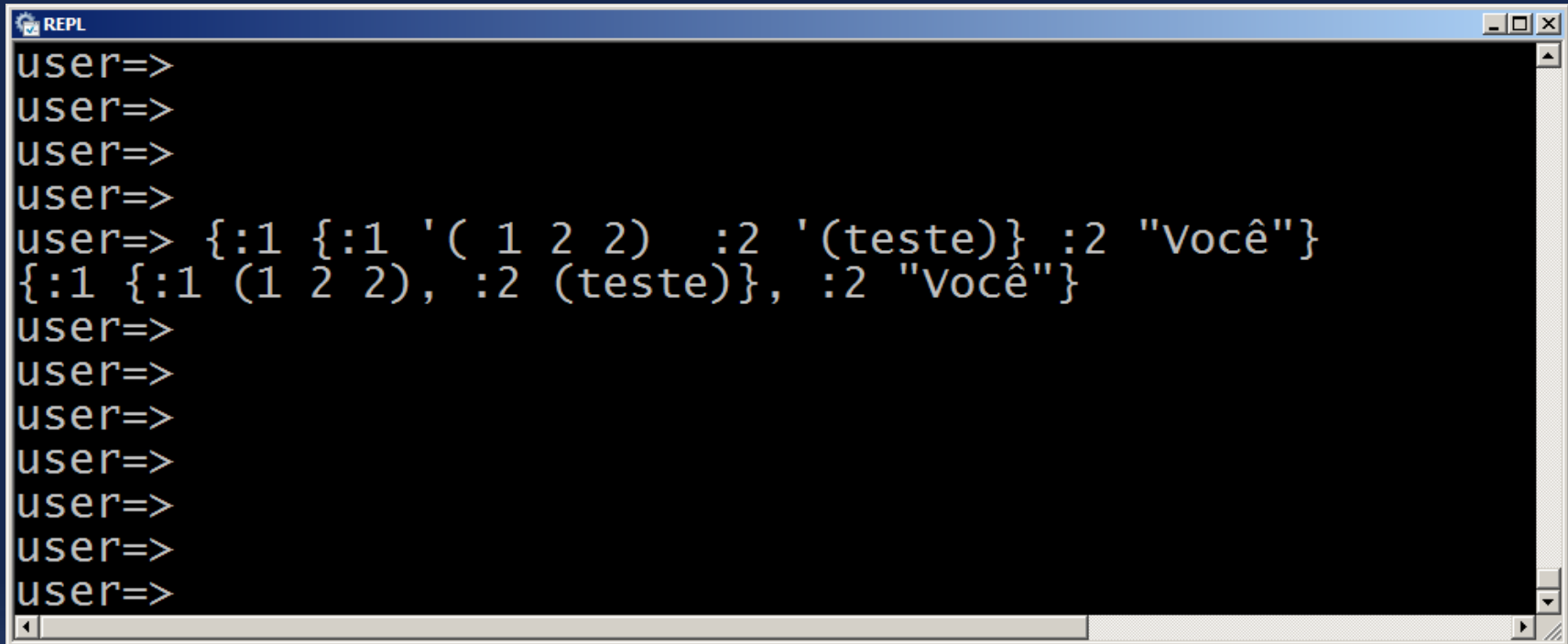


```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> { :a 1 , :b "Hello" , :c true }
{:a 1, :b "Hello", :c true}
user=>
user=>
user=>
user=>
user=>
```



Maps

✓ **Maps** podem ser definidos de forma **aninhada**;

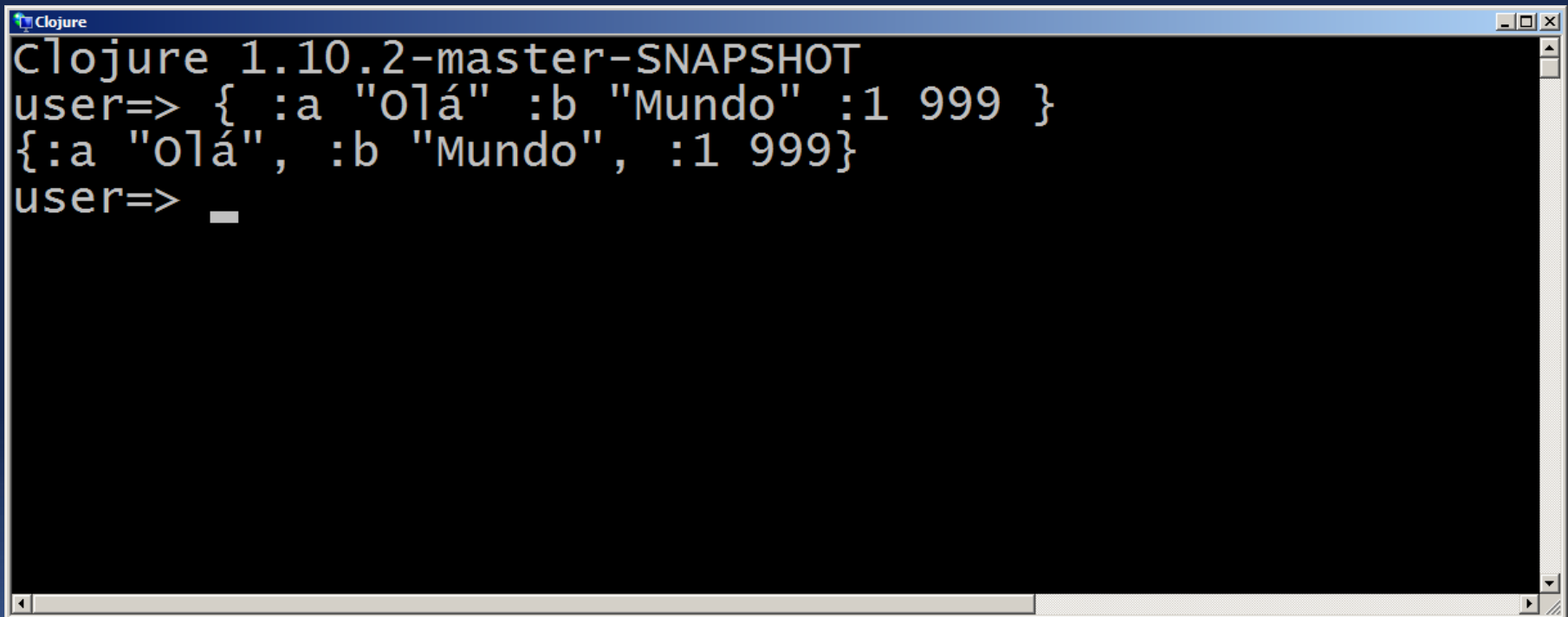


```
REPL
user=>
user=>
user=>
user=>
user=> {:1 {:1 '( 1 2 2) :2 '(teste)} :2 "Você"}
{:1 {:1 (1 2 2), :2 (teste)}, :2 "Você"}
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```



Maps

- ✓ Chaves (**keys**) podem também ser de diferentes tipos (strings, números ...), porém, geralmente, utilizam-se **keywords**.

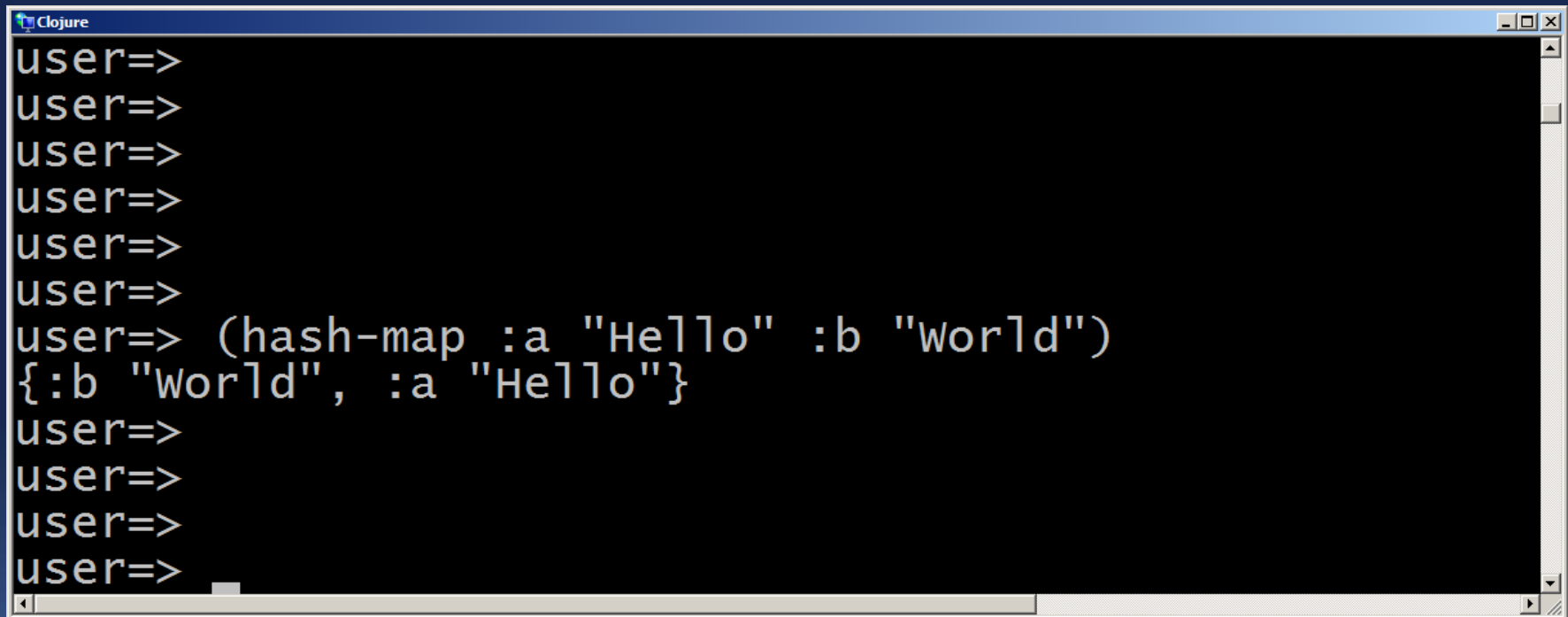


```
Clojure 1.10.2-master-SNAPSHOT
user=> { :a "Olá" :b "Mundo" :1 999 }
{:a "Olá", :b "Mundo", :1 999}
user=> _
```



Maps

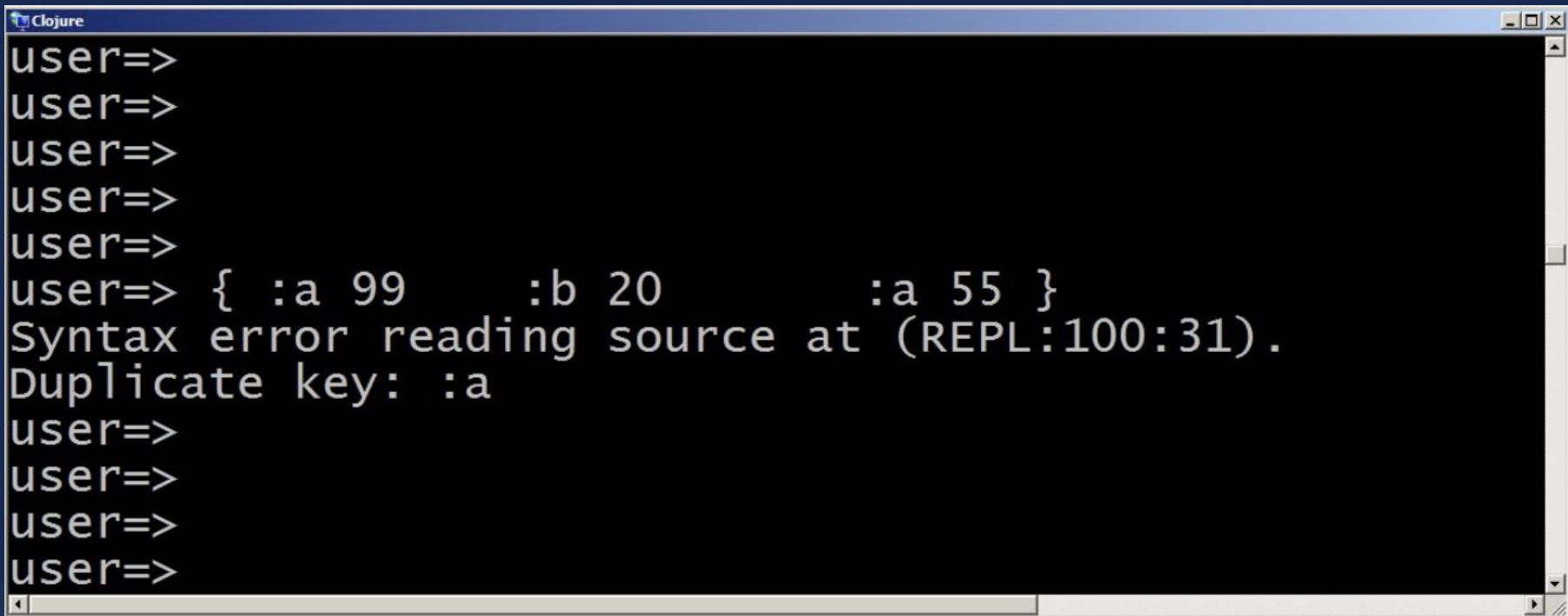
- ✓ Pode-se também criar maps com o uso da função **hash-map**, passando-se os pares **key-value** como **parâmetros**;



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> (hash-map :a "Hello" :b "World")
{:b "World", :a "Hello"}
user=>
user=>
user=>
user=>
```

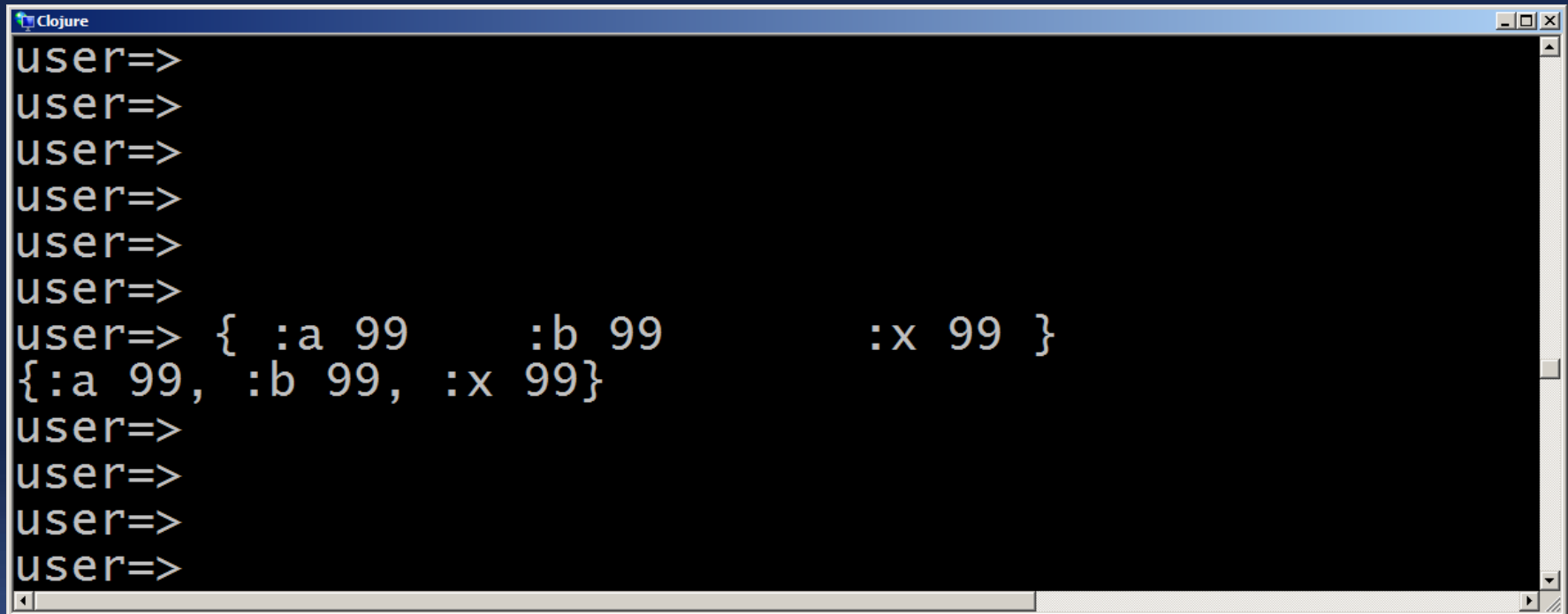


Keys em Maps são únicas



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> { :a 99      :b 20      :a 55 }
Syntax error reading source at (REPL:100:31).
Duplicate key: :a
user=>
user=>
user=>
user=>
```

Entretanto, diferentes keys podem ter o mesmo valor



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> { :a 99      :b 99      :x 99 }
{:a 99, :b 99, :x 99}
user=>
user=>
user=>
user=>
```



Exercícios com maps



1. Sob **REPL**, crie um **map** chamado **carro-favorito** com marca, modelo, marca, cor e capac-bagagem.

Marca => “Toyota”

Modelo => “Corolla”

Cor => “Preto”

Capac-bagagem => 470



1. Sob **REPL**, crie um **map** chamado **carro-favorito** com marca, modelo, cor e capac-bagagem.

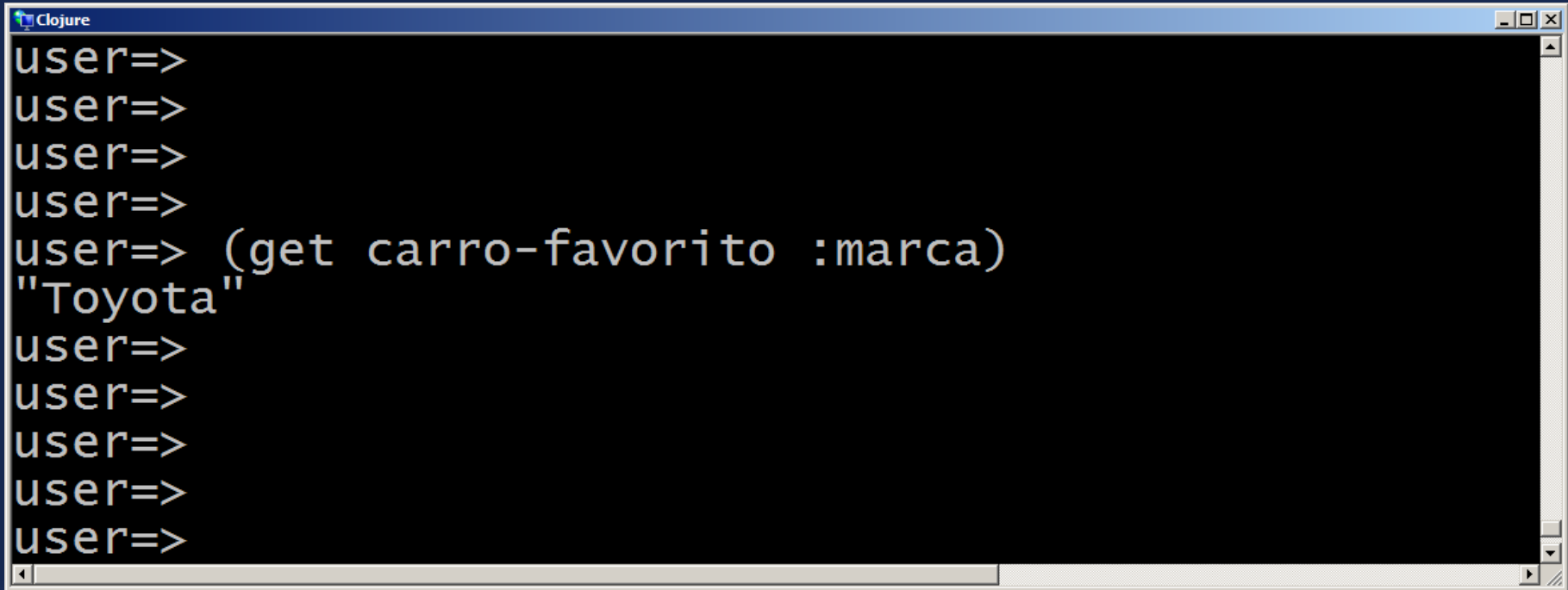
```
REPL
user=>
user=>
user=> (def carro-favorito { :marca "Toyota" :modelo "Corolla" :cor "Preto" :capac-bagagem 470 } )
#'user/carro-favorito
user=>
user=>
user=>
user=>
user=> carro-favorito
{:marca "Toyota", :modelo "Corolla", :cor "Preto", :capac-bagagem 470}
user=>
```



2. Use a função **get** para obter uma entrada do map.



2. Use a função **get** para uma entrada do map.

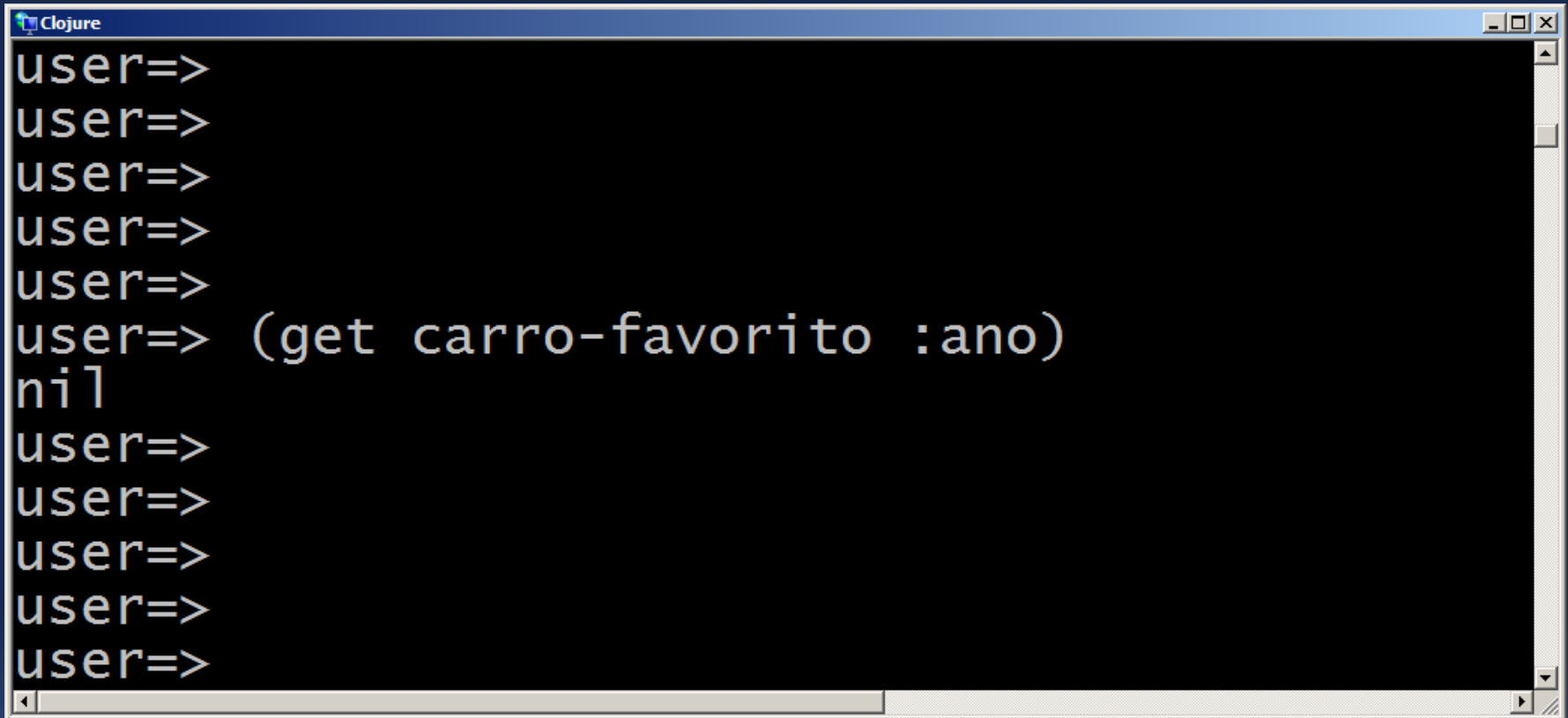


```
Clojure
user=>
user=>
user=>
user=>
user=> (get carro-favorito :marca)
"Toyota"
user=>
user=>
user=>
user=>
user=>
```

3. Se uma dada **key** não existir **get** retorna **nil**.



3. Se uma dada **key** não existir **get** retorna **nil**.

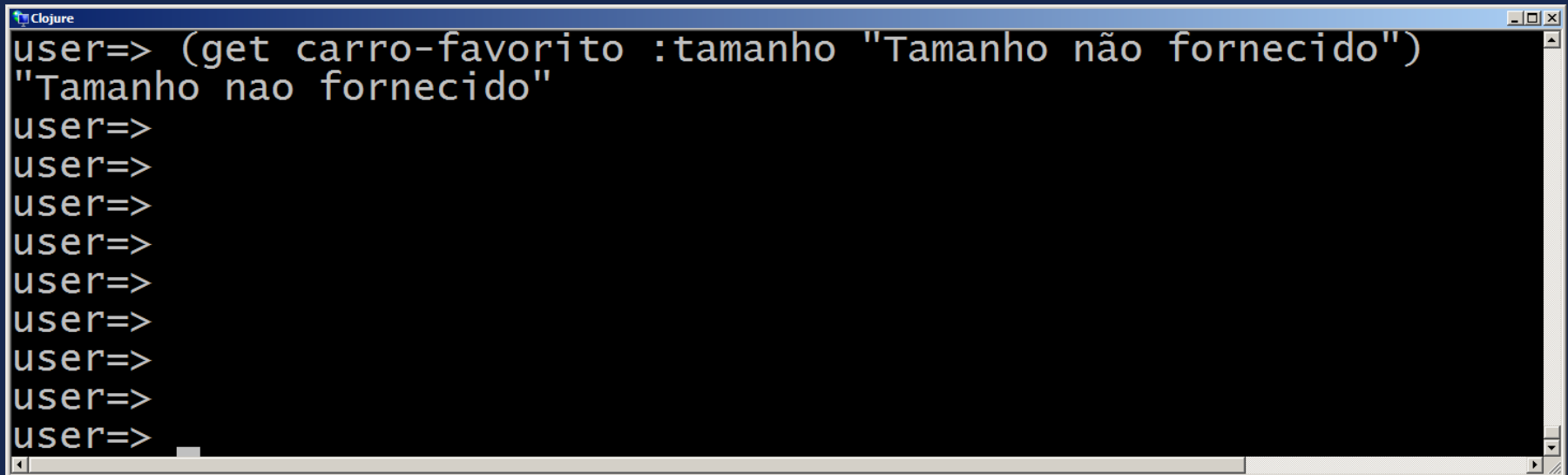


```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (get carro-favorito :ano)
nil
user=>
user=>
user=>
user=>
user=>
```

4. Se a chave não existir, pode-se acrescentar um terceiro argumento ao **get**, o qual representa um valor **fallback**.



4. Se a chave não existir, pode-se acrescentar um terceiro argumento ao **get**, o qual representa um valor **fallback**.



```
Clojure
user=> (get carro-favorito :tamanho "Tamanho não fornecido")
"Tamanho nao fornecido"
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

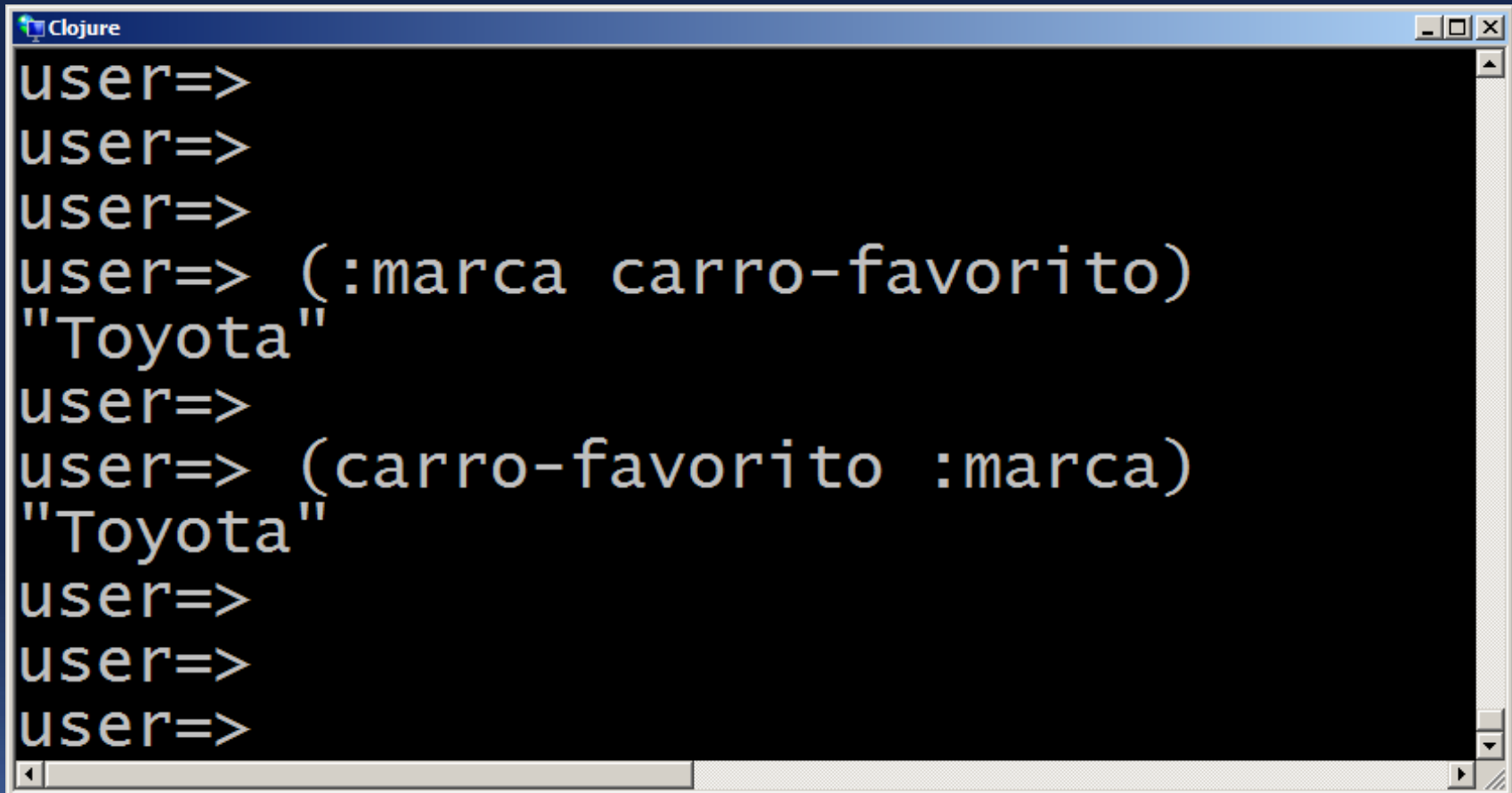


5. **Maps** e **keywords** têm a habilidade especial de serem usados como **funções**.

Quando colocados como primeiro item da lista (“operator position”), eles são invocados como uma função que pode ser usada para a pesquisa (**look up**).



5. **Maps** e **keywords** têm a habilidade especial de serem usados como **funções**. Quando colocados como primeiro item da lista (“operator position”), eles são invocados como uma função que pode ser usada para a pesquisa (**look up**).



```
Clojure
user=>
user=>
user=>
user=> (:marca carro-favorito)
"Toyota"
user=>
user=> (carro-favorito :marca)
"Toyota"
user=>
user=>
user=>
```

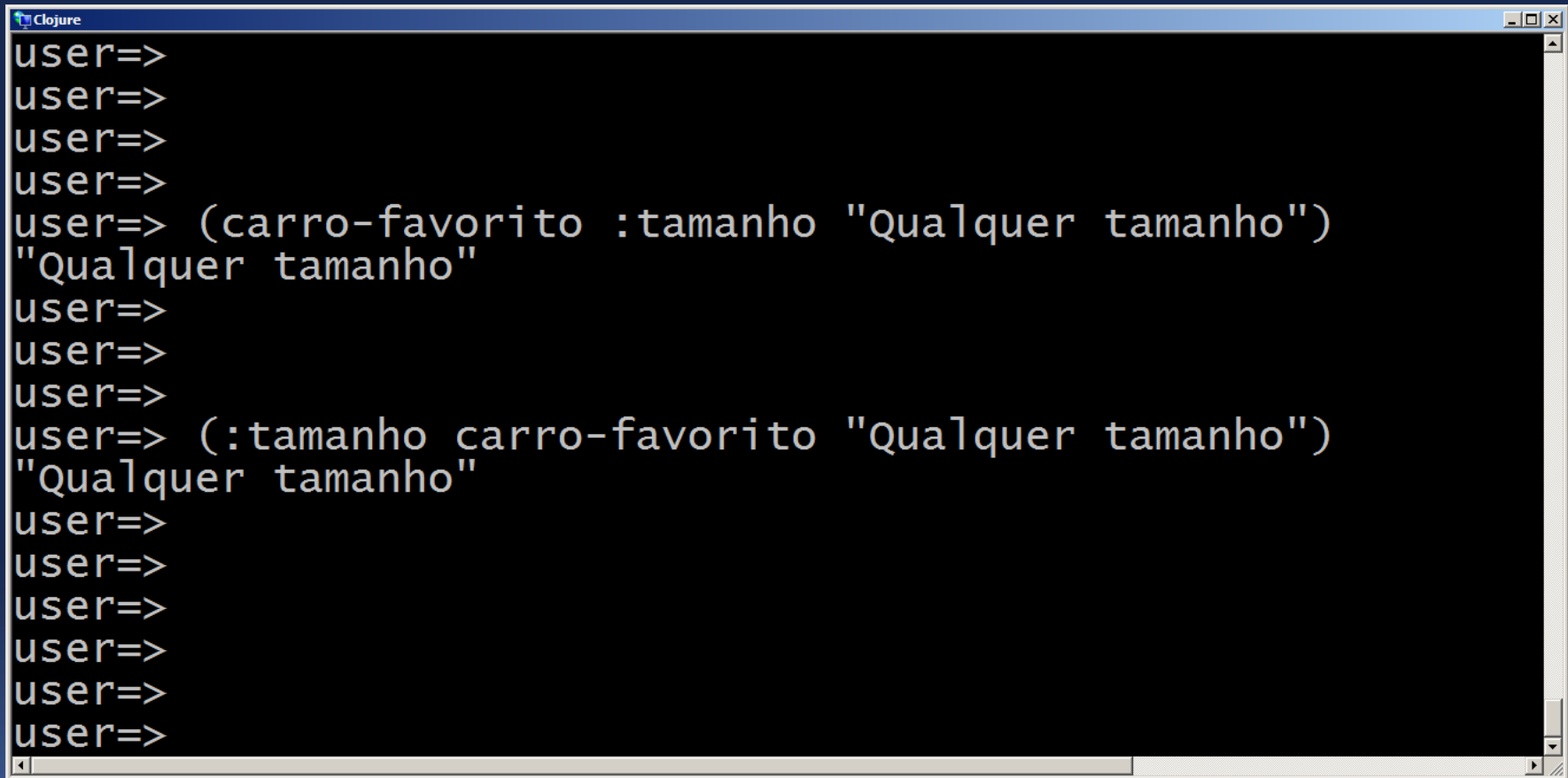


6. **Maps** e **keywords** têm a habilidade especial de serem usados como **funções**.

Da mesma forma como a função **get**, um valor de **callback** pode também ser definido.



6. **Maps** e **keywords** têm a habilidade especial de serem usados como **funções**. Da mesma forma como a função `get`, um valor de `callback` pode também ser definido.



```
Clojure
user=>
user=>
user=>
user=>
user=> (carro-favorito :tamanho "Qualquer tamanho")
"Qualquer tamanho"
user=>
user=>
user=>
user=> (:tamanho carro-favorito "Qualquer tamanho")
"Qualquer tamanho"
user=>
user=>
user=>
user=>
user=>
user=>
```



7. Para se armazenar um novo par **key-value** em um **map**, pode-se usar a função **assoc**.



7. Para se armazenar um novo par key-value em um **map**, pode-se usar a função **assoc**.

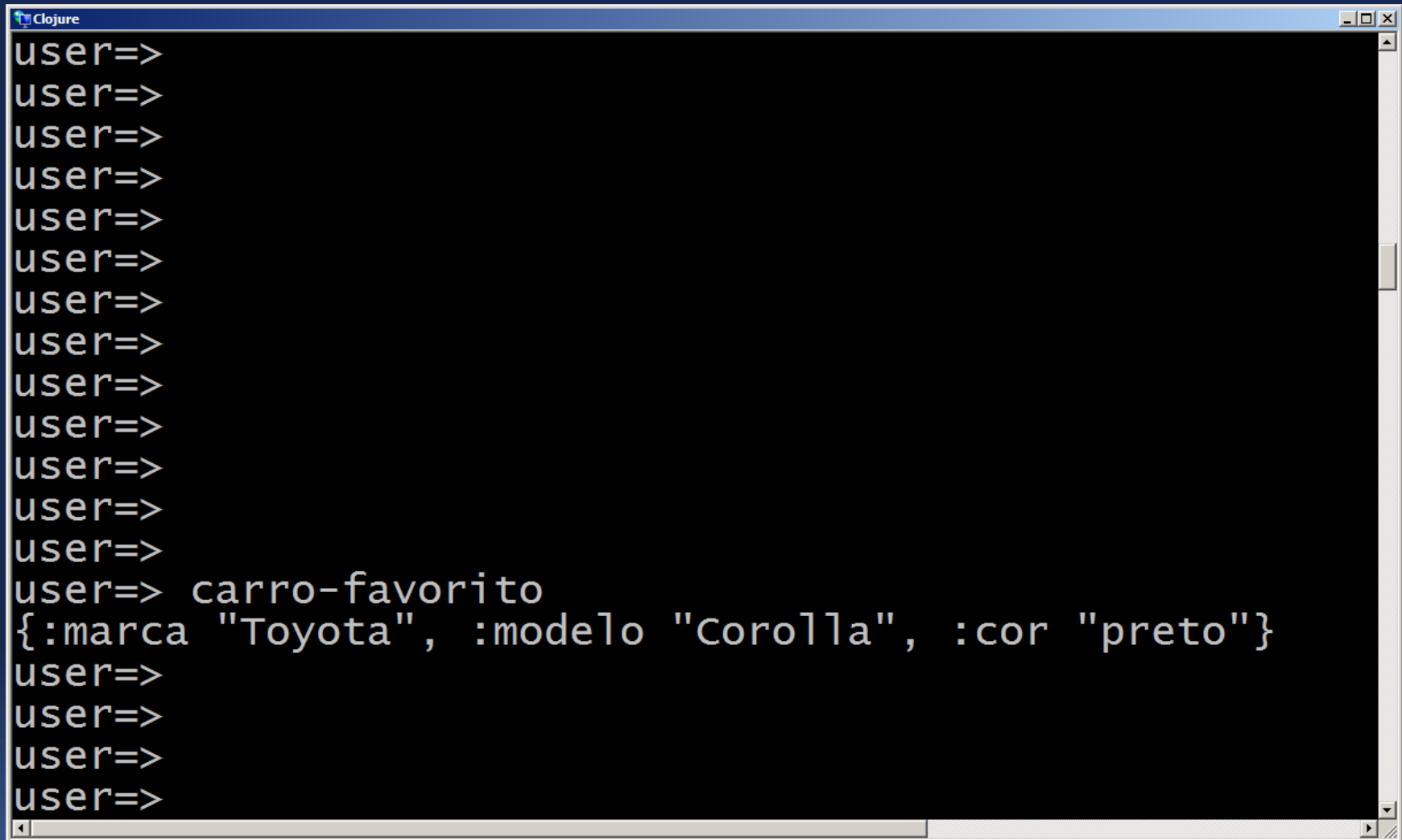
```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (assoc carro-favorito :veloc-max 200)
{:marca "Toyota", :modelo "Corolla", :cor "preto", :capac-bagagem 470, :veloc-max 200}
user=>
user=>
user=>
user=>
```



8. Avalie novamente o **map** carro-favorito !



8. Avalie novamente o **map** carro-favorito !



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> carro-favorito
{:marca "Toyota", :modelo "Corolla", :cor "preto"}
user=>
user=>
user=>
user=>
```



Vé ... Não entendi !!!

O novo valor não foi
acrescentado ao map ?



maps são imutáveis !

- ✓ Pois é, **maps** são **imutáveis**;
- ✓ O map original permanece **imutável**;
- ✓ A função **assoc** na verdade, retorna um novo **map** contendo o map original bem como o novo par de valores acrescentado ao map;
- ✓ Ou seja, a função map criou uma **nova versão** do map.



9. Modifique no **map** carro-favorito o valor da cor para “vermelho” com o uso da função **map**.



9. Modifique no **map** carro-favorito o valor da cor para “vermelho” com o uso da função **map**.

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (assoc carro-favorito :cor "vermelho")
{:marca "Toyota", :modelo "Corolla", :cor "vermelho", :capac-bagagem 470}
user=>
user=>
user=>
user=>
user=>
```



10. Inclua no **map** carro-favorito os modelos do carro.

Modelo a => “GLI”

Modelo b => “xEi”



10. Inclua no **map** carro-favorito os modelos do carro.

```
Clojure
user=>
user=>
user=>
user=>
user=> (assoc carro-favorito :modelos { :A "GLI" :B "xEi" } )
{:marca "Toyota", :modelo "Corolla", :cor "preto", :capac-bagagem 470, :modelos
{:A "GLI", :B "xEi"}}
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

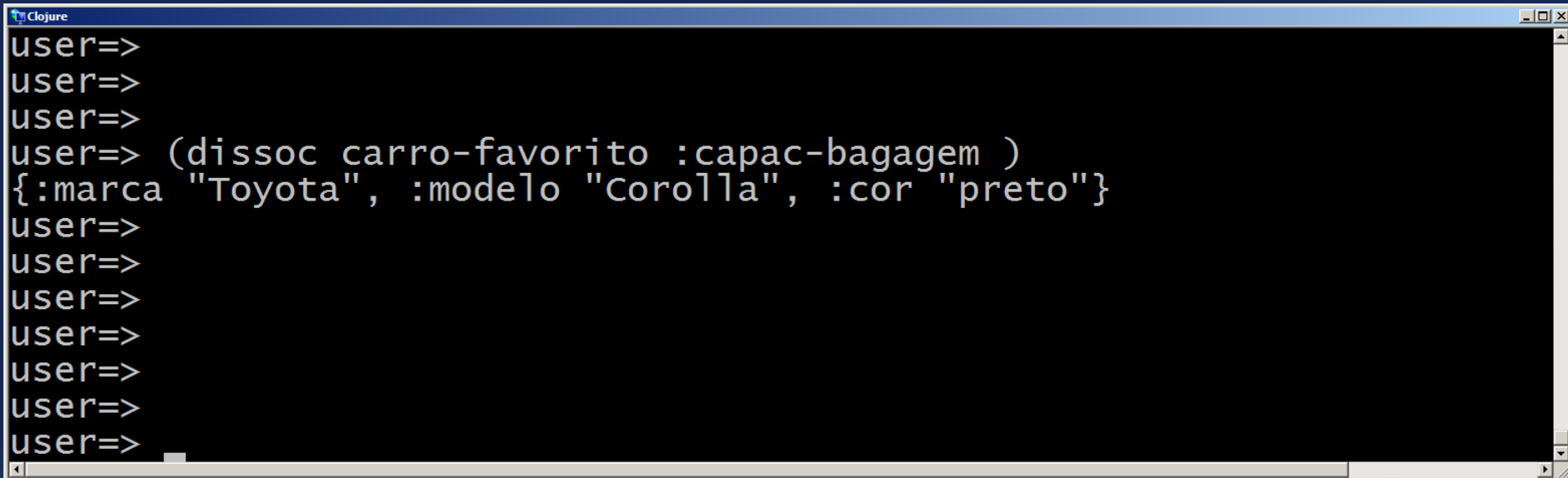


11. Para remover um elemento (key-value) de um map, pode-se usar a função **dissoc**.

Por exemplo, vamos remover o elemento capac-bagagem do map carro-favorito.



11. Para remover um elemento (key-value) de um map, pode-se usar a função **dissoc**. Por exemplo, vamos remover o elemento capac-bagagem do map carro-favorito.

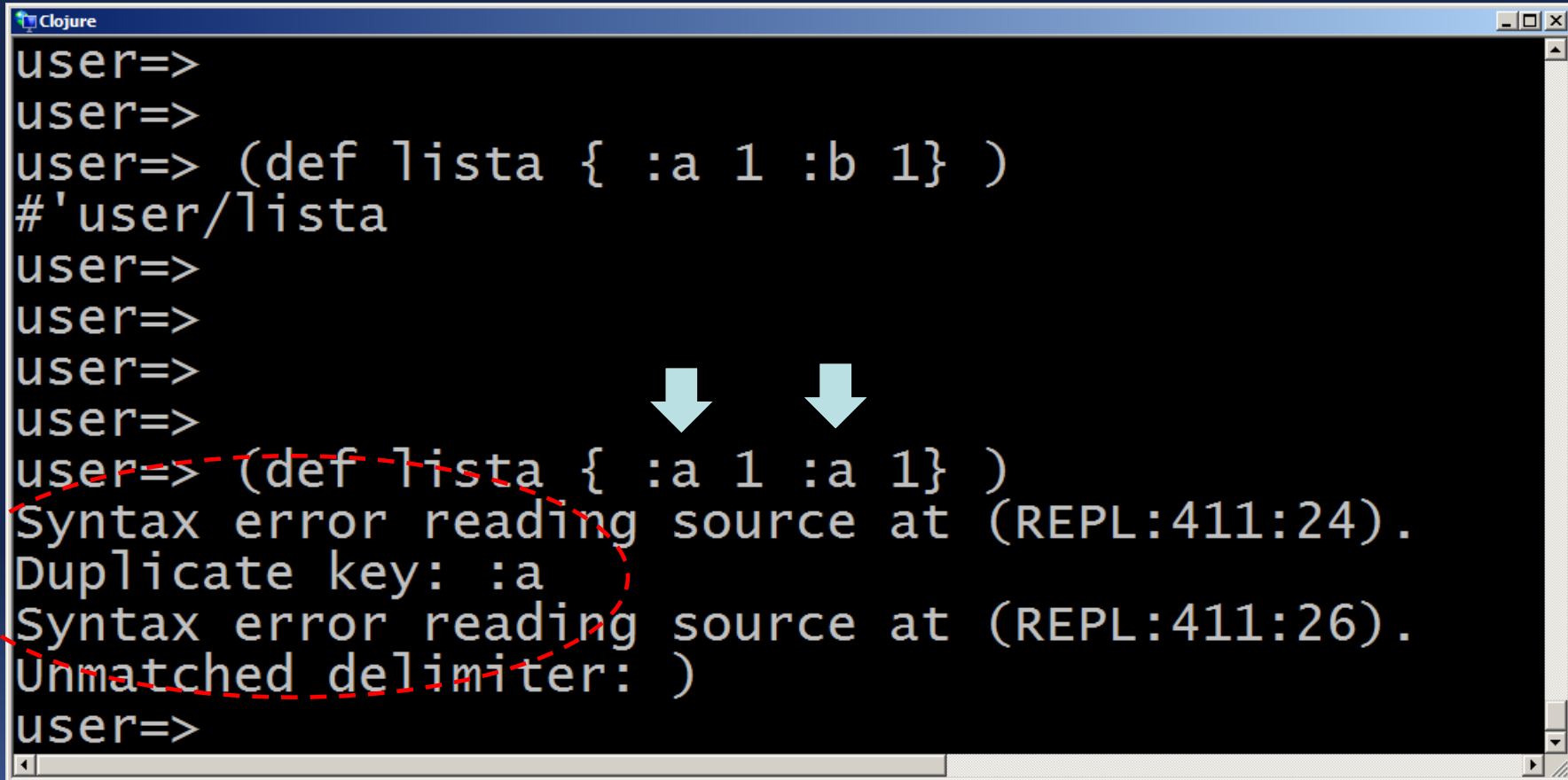


```
Clojure
user=>
user=>
user=>
user=> (dissoc carro-favorito :capac-bagagem )
{:marca "Toyota", :modelo "Corolla", :cor "preto"}
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```



keys e values em maps

- ✓ Vimos que maps **podem** ter valores **iguais** associados à **chaves diferentes**;
- ✓ **Mas**, maps **não** permitem **chaves iguais**.



```
Clojure
user=>
user=>
user=> (def lista { :a 1 :b 1 } )
#'user/lista
user=>
user=>
user=>
user=>
user=> (def lista { :a 1 :a 1 } )
Syntax error reading source at (REPL:411:24).
Duplicate key: :a
Syntax error reading source at (REPL:411:26).
Unmatched delimiter: )
user=>
```

Two blue arrows point from the first two empty prompts to the second definition line, illustrating the contrast between valid and invalid map syntax.





Sets

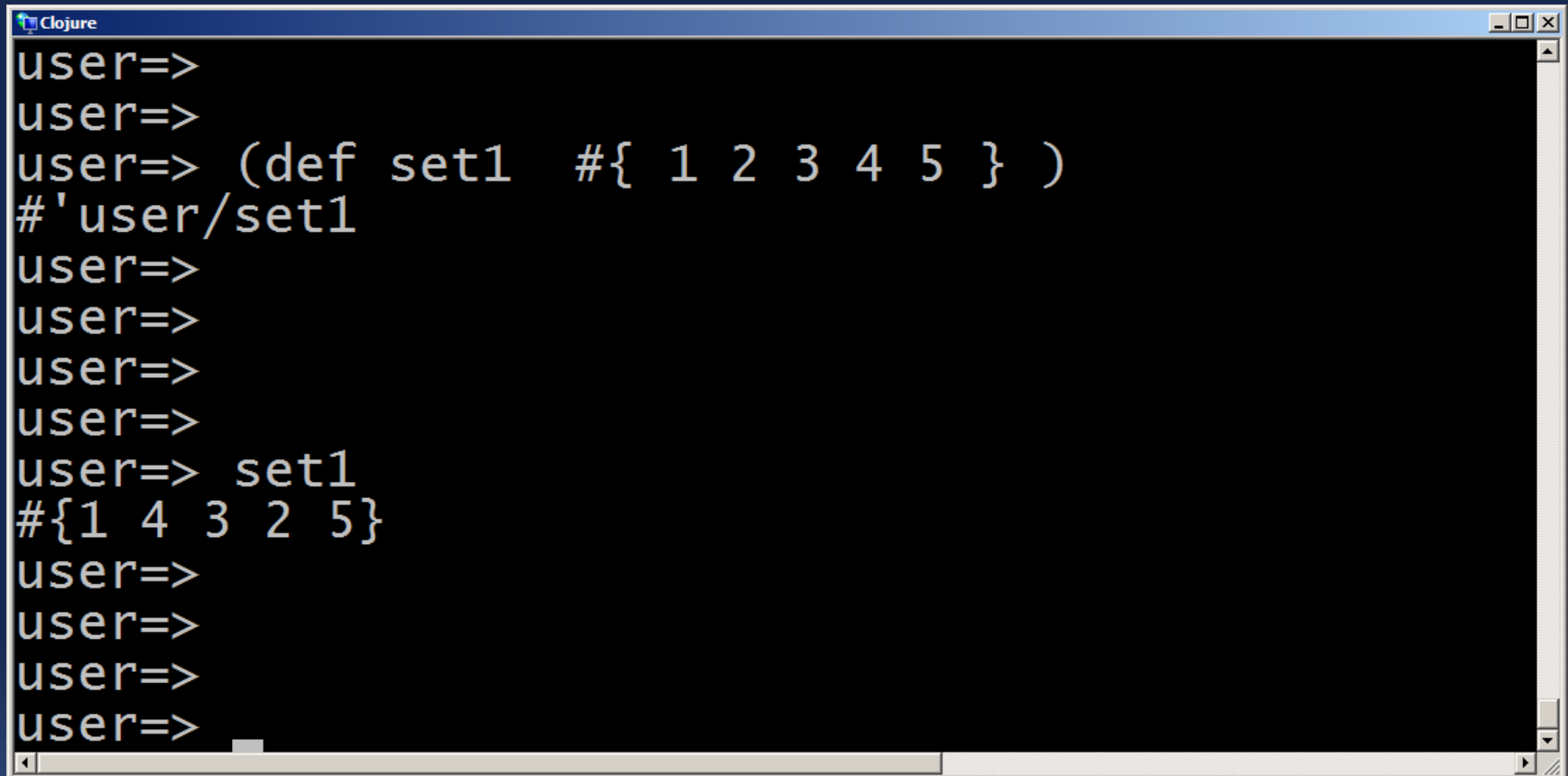


Sets

- ✓ Um **set** (conjunto) é uma coleção de **valores** únicos;
- ✓ **Clojure** provê **HashSet** e **SortedSet**;
- ✓ Hash Sets são implementados como **Hash Maps**, com **chave** e **valor** idênticos ;
- ✓ Hash Sets são muito usados em Clojure e têm uma notação literal definida por: **#{ }**



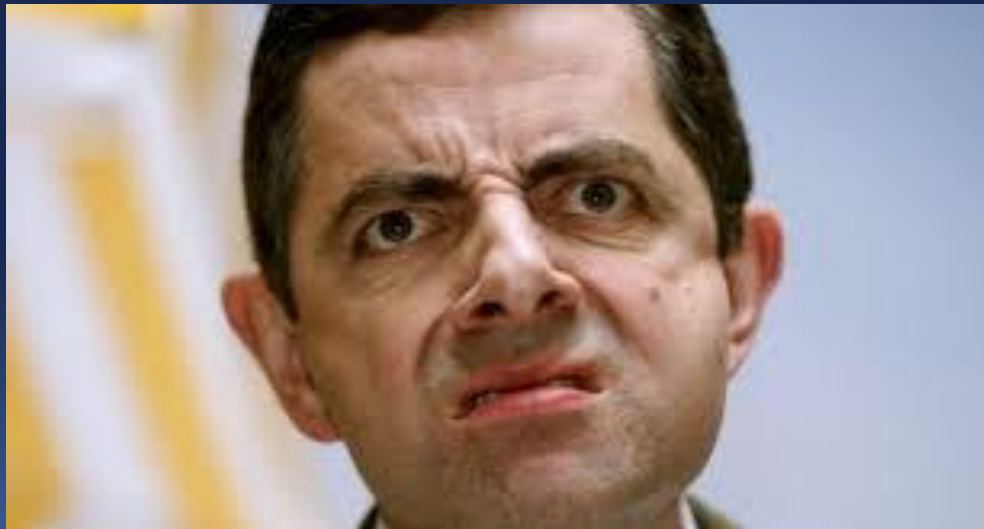
Sets – Exemplo



```
Clojure
user=>
user=>
user=> (def set1 #{ 1 2 3 4 5 } )
#'user/set1
user=>
user=>
user=>
user=>
user=> set1
#{1 4 3 2 5}
user=>
user=>
user=>
user=>
```

Vê ... Não entendi !!!

Por que quando um set é avaliado, os elementos **não** são retornados na ordem em que foram definidos ?



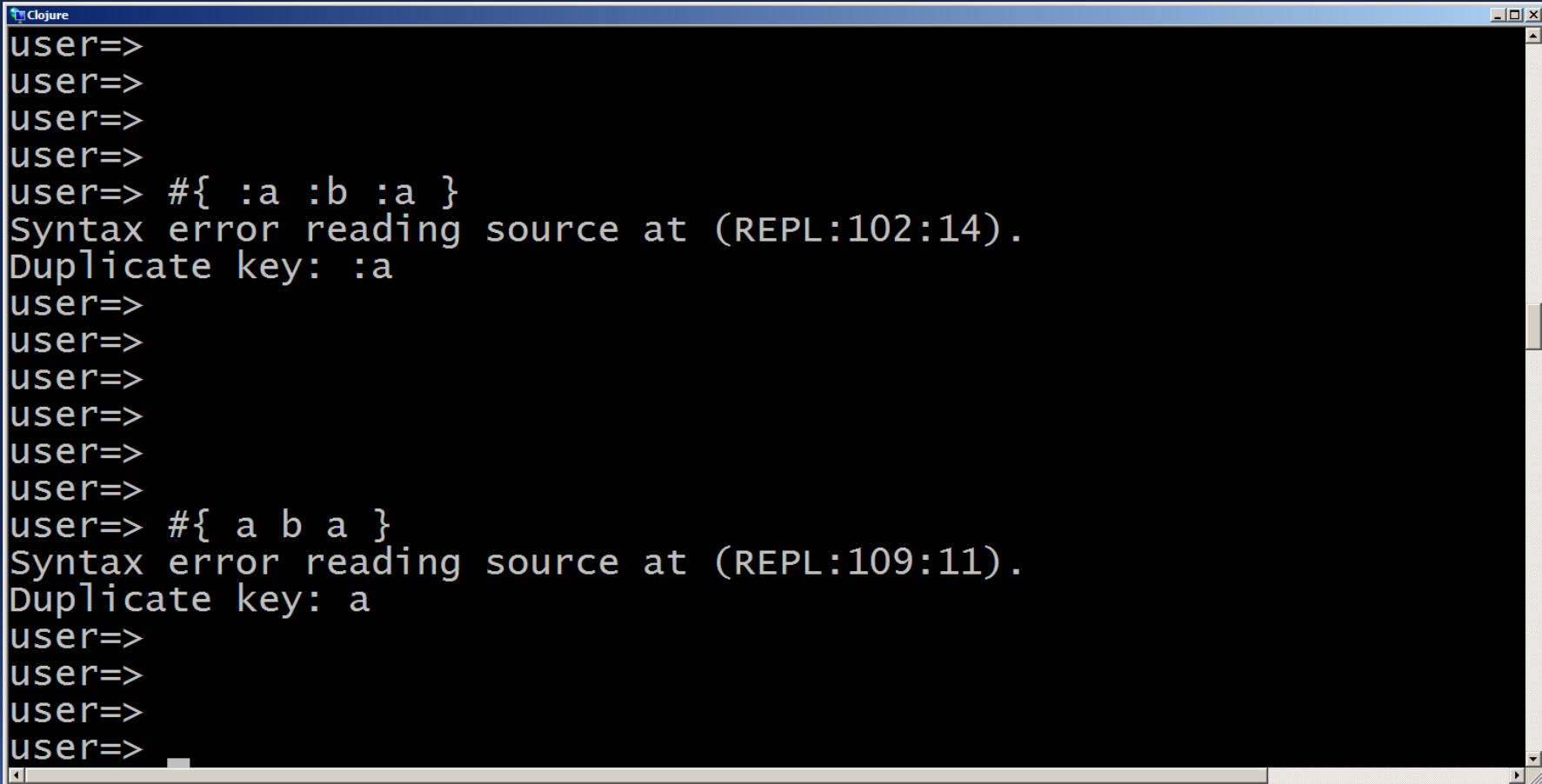
Sets – Observação

- ✓ Quando um **set** é avaliado, **não** são retornados os elementos do set na ordem em que foram definidos na expressão literal.
- ✓ Isso ocorre por causa da **estrutura interna do HashSet**.
- ✓ O valor de cada elemento é transformado em um **hash único**, o qual permite acesso rápido, mas não se mantém a ordem de inserção.
- ✓ Caso se queira manter a ordem na avaliação dos elementos de um set, deve-se usar uma diferente estrutura de dados, por exemplo, um vector.



Sets – Observação

- ✓ Como dito anteriormente, sets **não** podem ter valores duplicados.

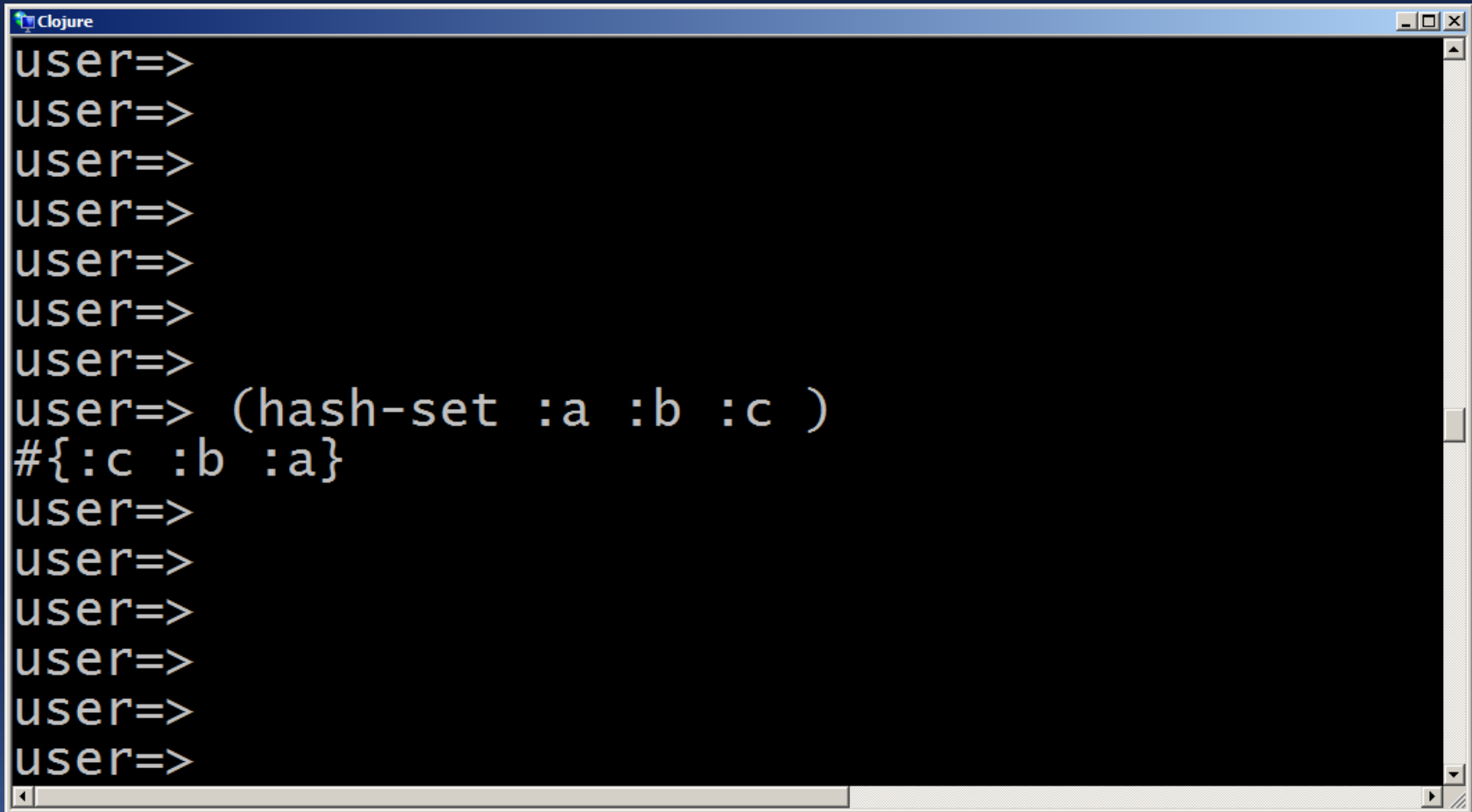


```
Clojure
user=>
user=>
user=>
user=>
user=> #{ :a :b :a }
Syntax error reading source at (REPL:102:14).
Duplicate key: :a
user=>
user=>
user=>
user=>
user=>
user=>
user=> #{ a b a }
Syntax error reading source at (REPL:109:11).
Duplicate key: a
user=>
user=>
user=>
user=>
```



Hash Sets

- ✓ Podem também ser criados a partir da função **hash-set**.

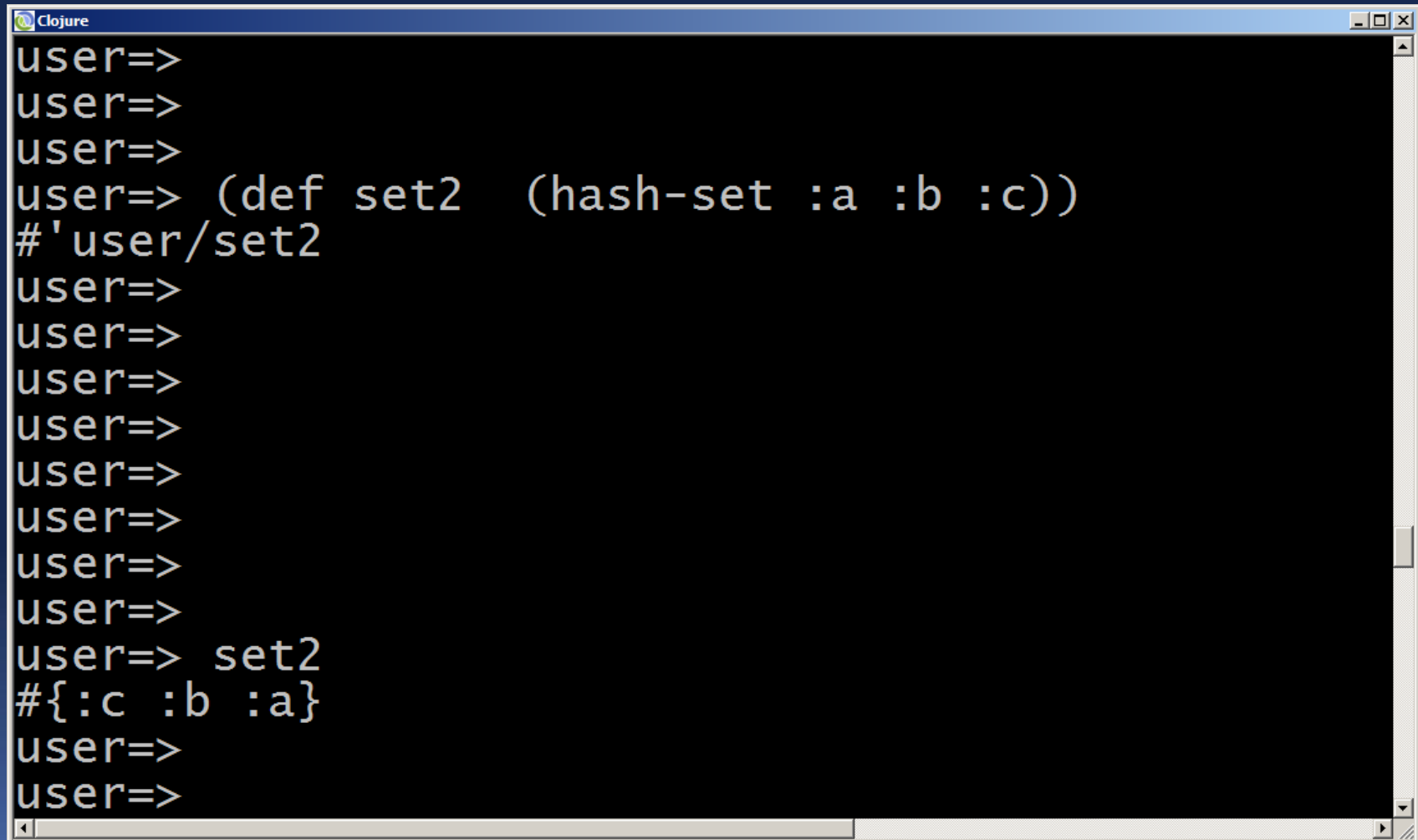


```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (hash-set :a :b :c )
#{:c :b :a}
user=>
user=>
user=>
user=>
user=>
user=>
```



Hash Sets

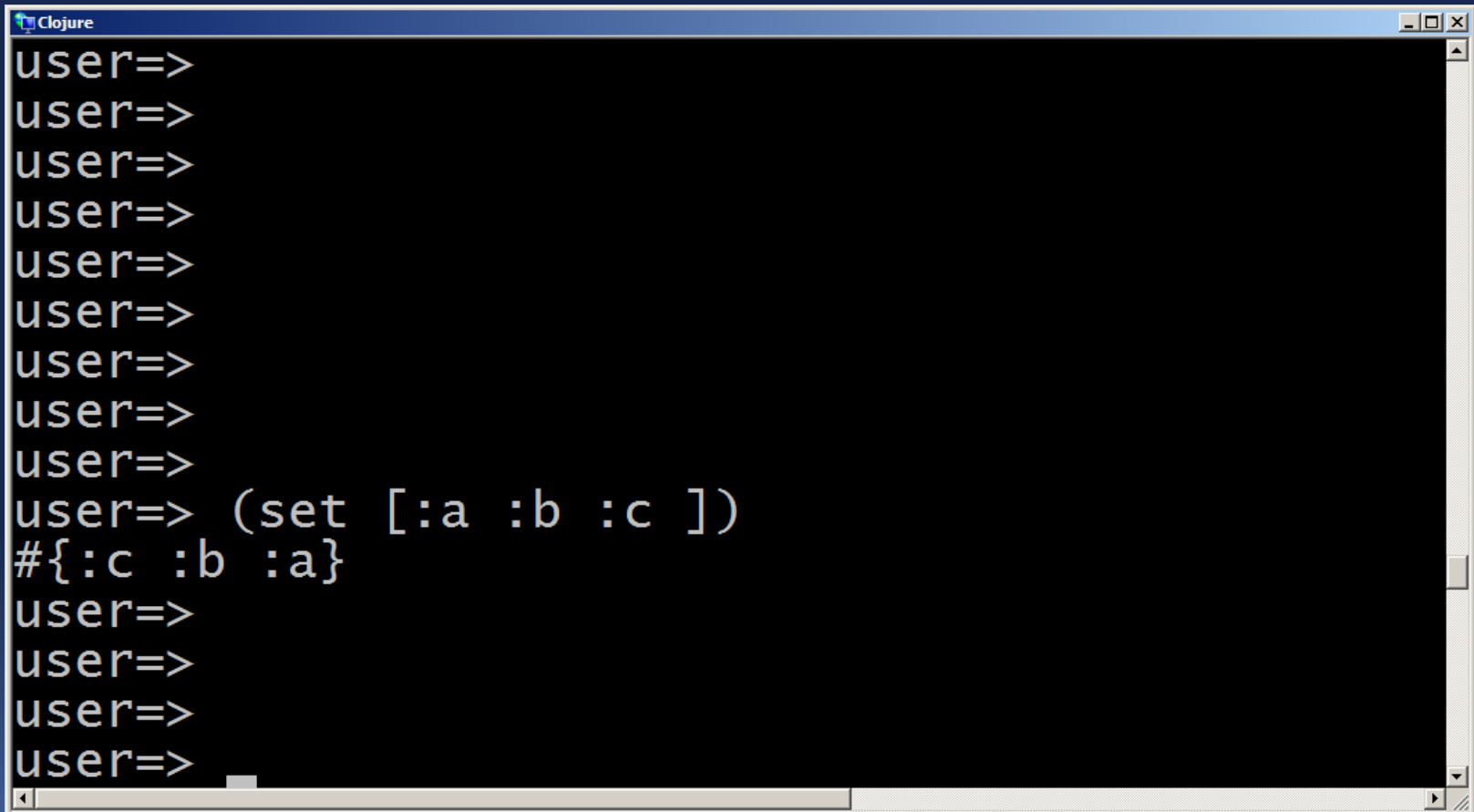
- ✓ Podem também ser criados a partir da função **hash-set**.



```
Clojure
user=>
user=>
user=>
user=> (def set2 (hash-set :a :b :c))
#'user/set2
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> set2
#{:c :b :a}
user=>
user=>
```

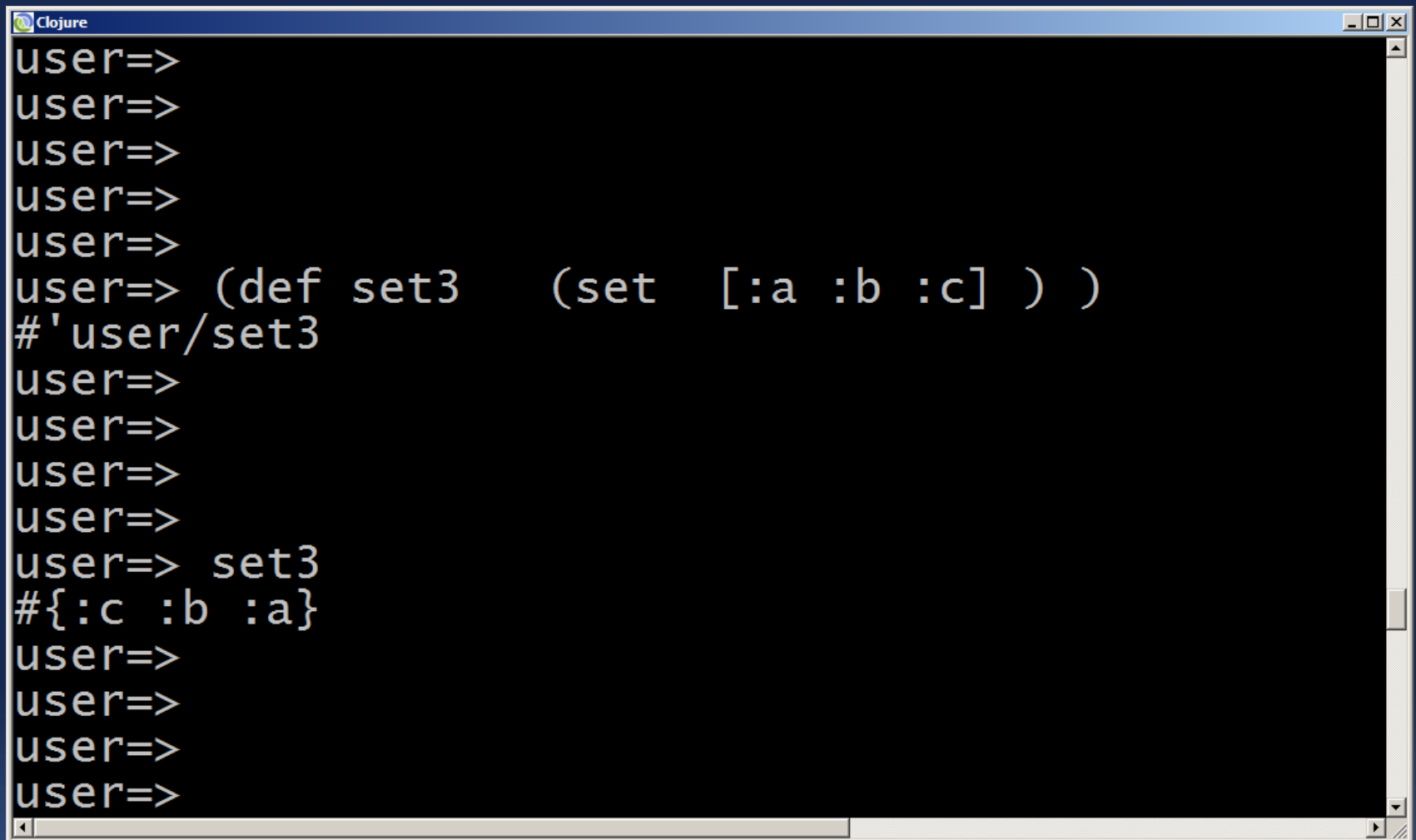

Hash Sets

- ✓ Podem ser criados a partir de outra coleção, por meio da função **set**.
- ✓ O exemplo mostra a criação de um set a partir de um vector.

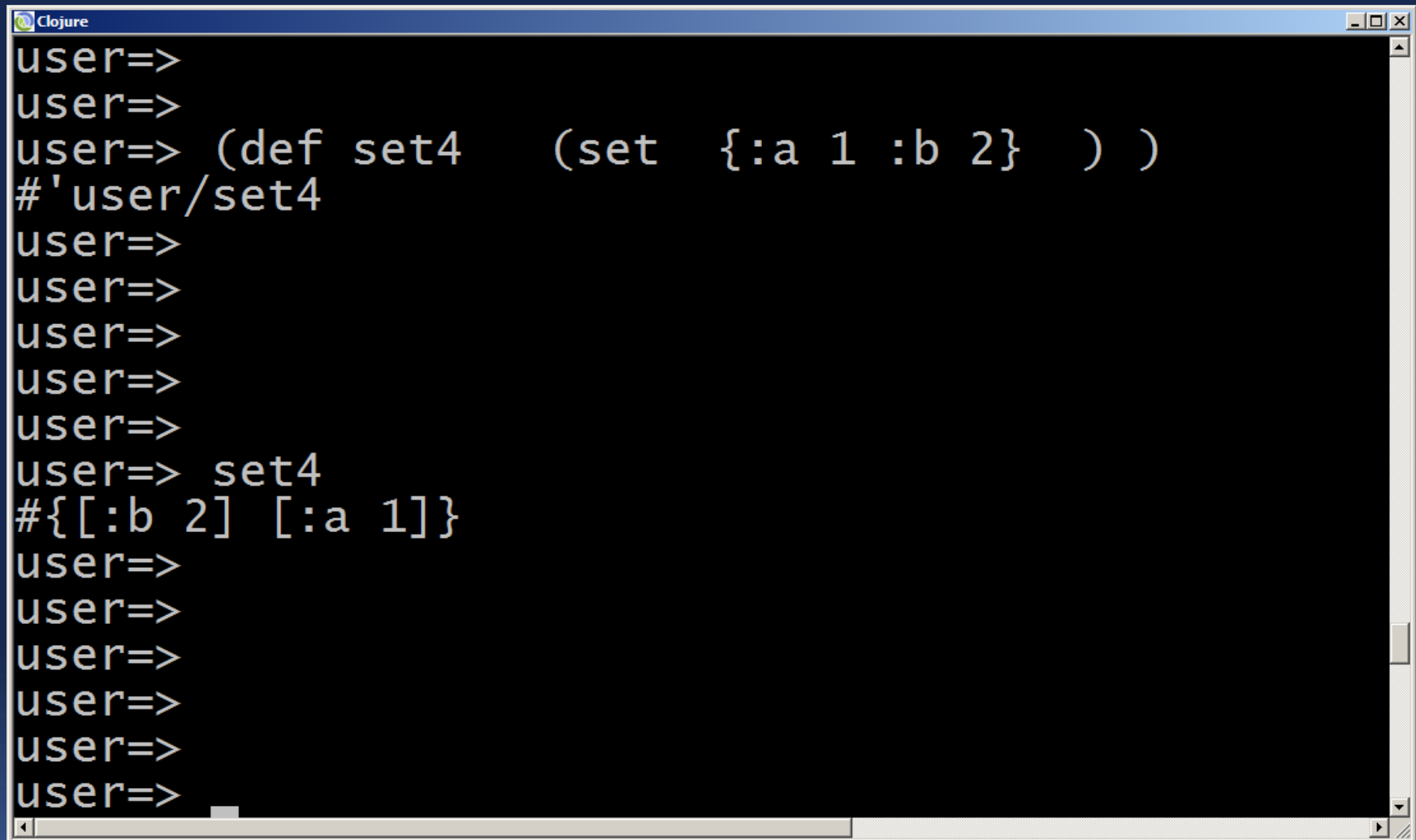


```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (set [:a :b :c ])
#{:c :b :a}
user=>
user=>
user=>
user=>
```

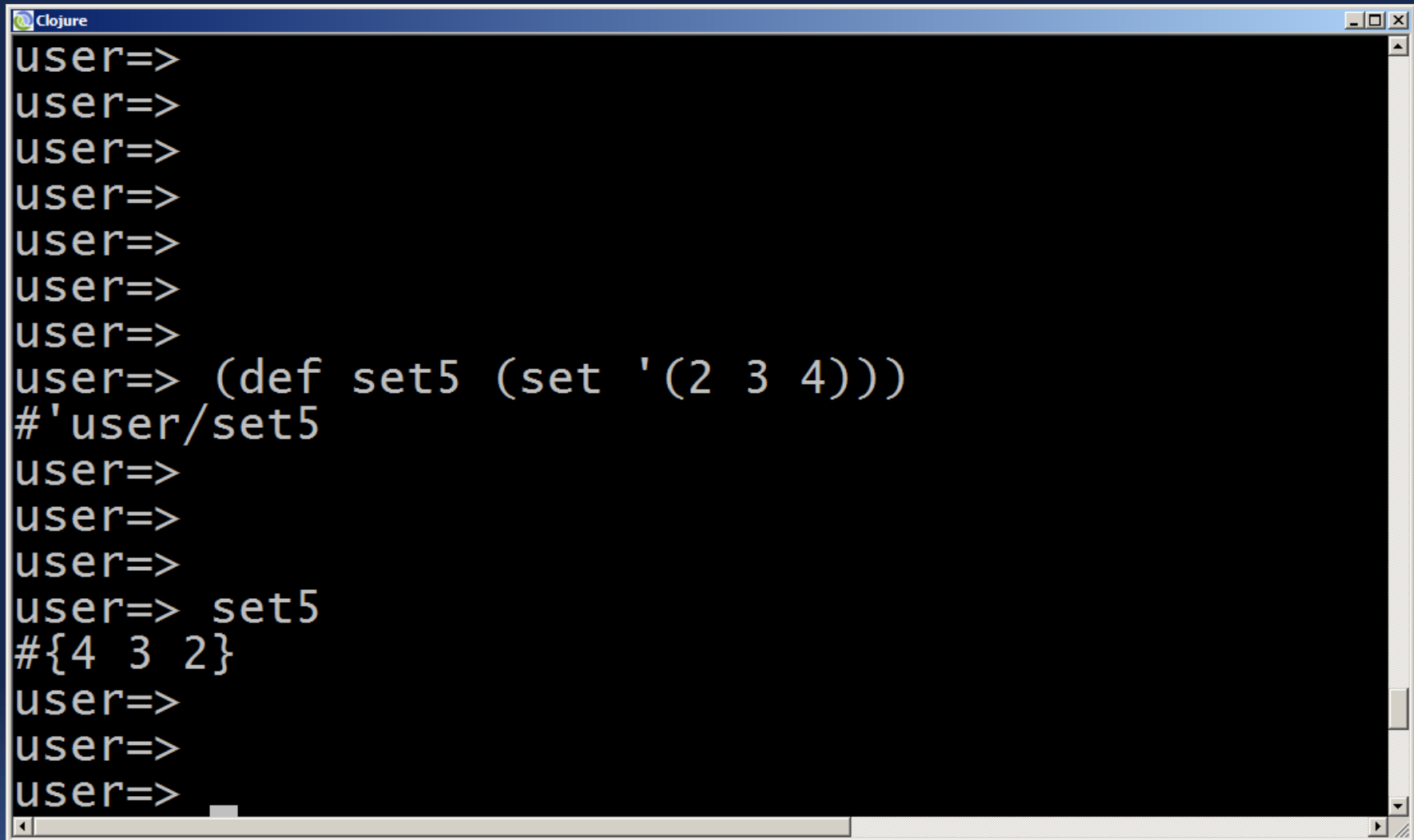




```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (def set3 (set [:a :b :c] ) )
#'user/set3
user=>
user=>
user=>
user=>
user=> set3
#{:c :b :a}
user=>
user=>
user=>
user=>
```

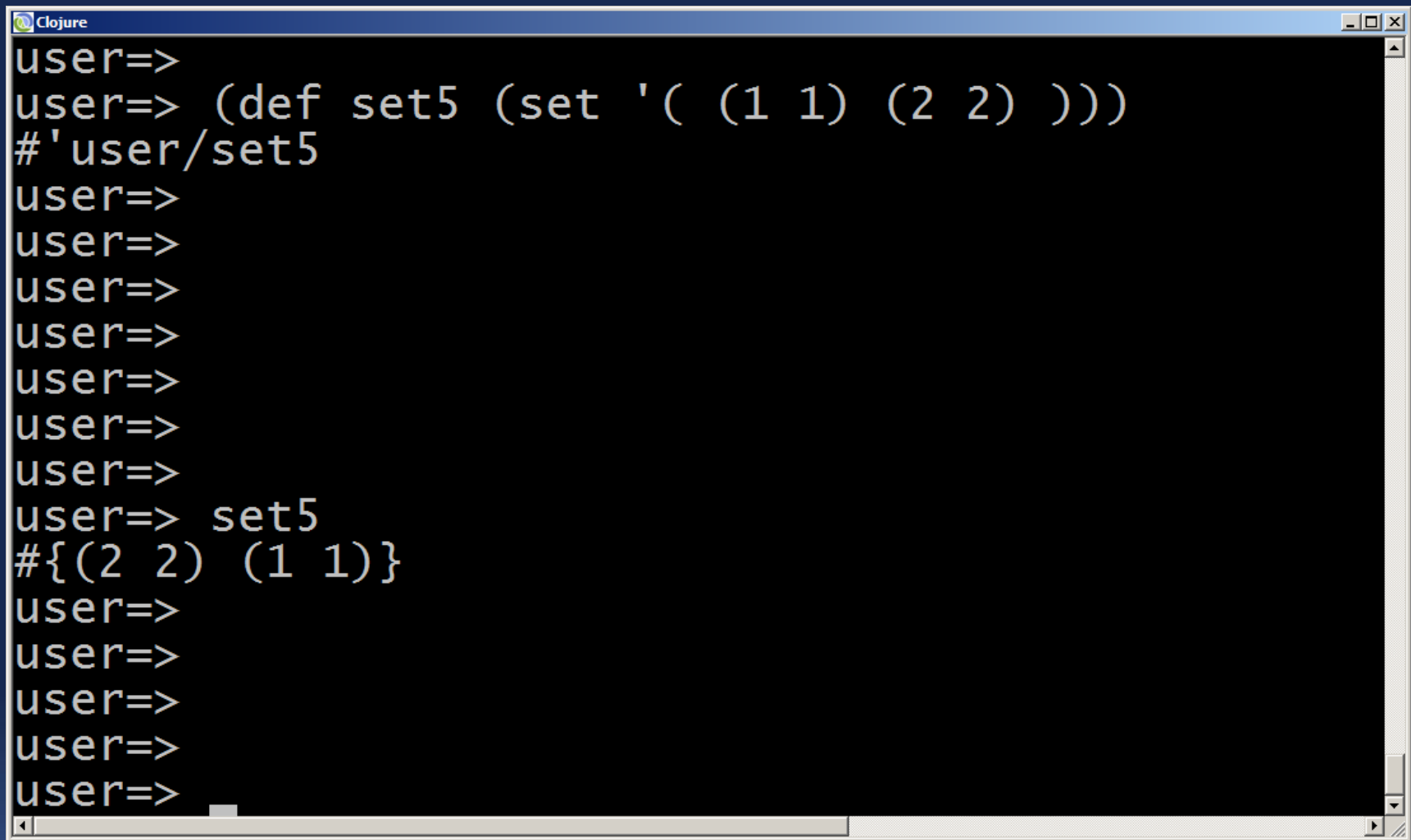


```
Clojure
user=>
user=>
user=> (def set4 (set {:a 1 :b 2} ))
#'user/set4
user=>
user=>
user=>
user=>
user=>
user=> set4
#{[:b 2] [:a 1]}
user=>
user=>
user=>
user=>
user=>
user=>
```



A screenshot of a Clojure REPL window. The window has a title bar with the Clojure logo and the text 'Clojure'. The background is black with white text. The text shows a series of 'user=>' prompts. The first seven prompts are followed by no output. The eighth prompt is followed by the output '(def set5 (set '(2 3 4)))'. The ninth prompt is followed by the output '#'user/set5'. The tenth prompt is followed by no output. The eleventh prompt is followed by the output 'set5'. The twelfth prompt is followed by the output '#{4 3 2}'. The thirteenth and fourteenth prompts are followed by no output. The window has a scrollbar on the right and a status bar at the bottom.

```
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (def set5 (set '(2 3 4)))
#'user/set5
user=>
user=>
user=>
user=> set5
#{4 3 2}
user=>
user=>
user=>
```



```
Clojure
user=>
user=> (def set5 (set '( (1 1) (2 2) )))
#'user/set5
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> set5
#{(2 2) (1 1)}
user=>
user=>
user=>
user=>
user=>
```

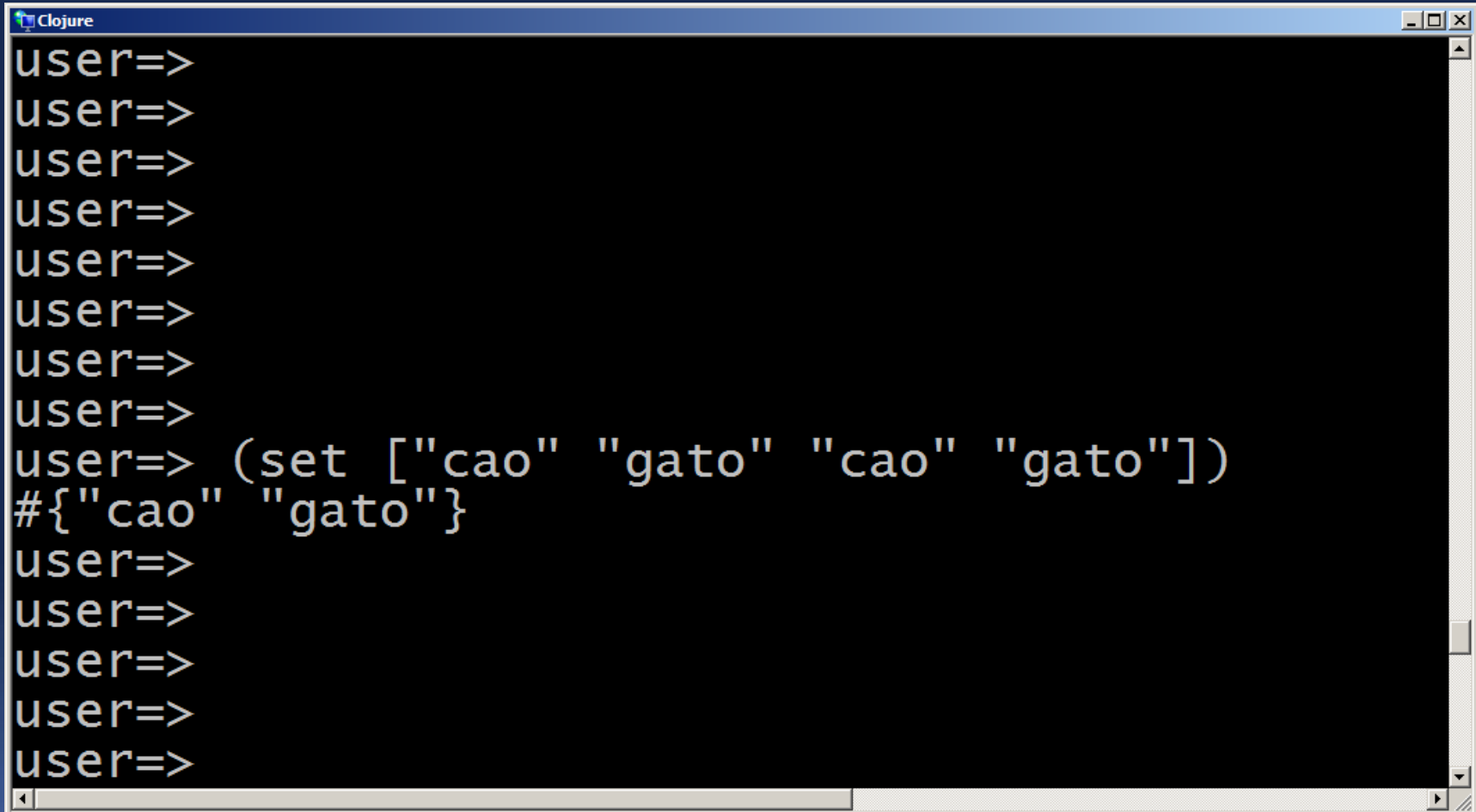
Função set – observação

- ✓ A função **set** **não** retorna erro em operações de **deduplicação** de valores.



Função set – observação

- ✓ A função **set** **não** retorna erro em operações de **deduplicação** de valores.



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (set ["cao" "gato" "cao" "gato"])
#{ "cao" "gato" }
user=>
user=>
user=>
user=>
user=>
```



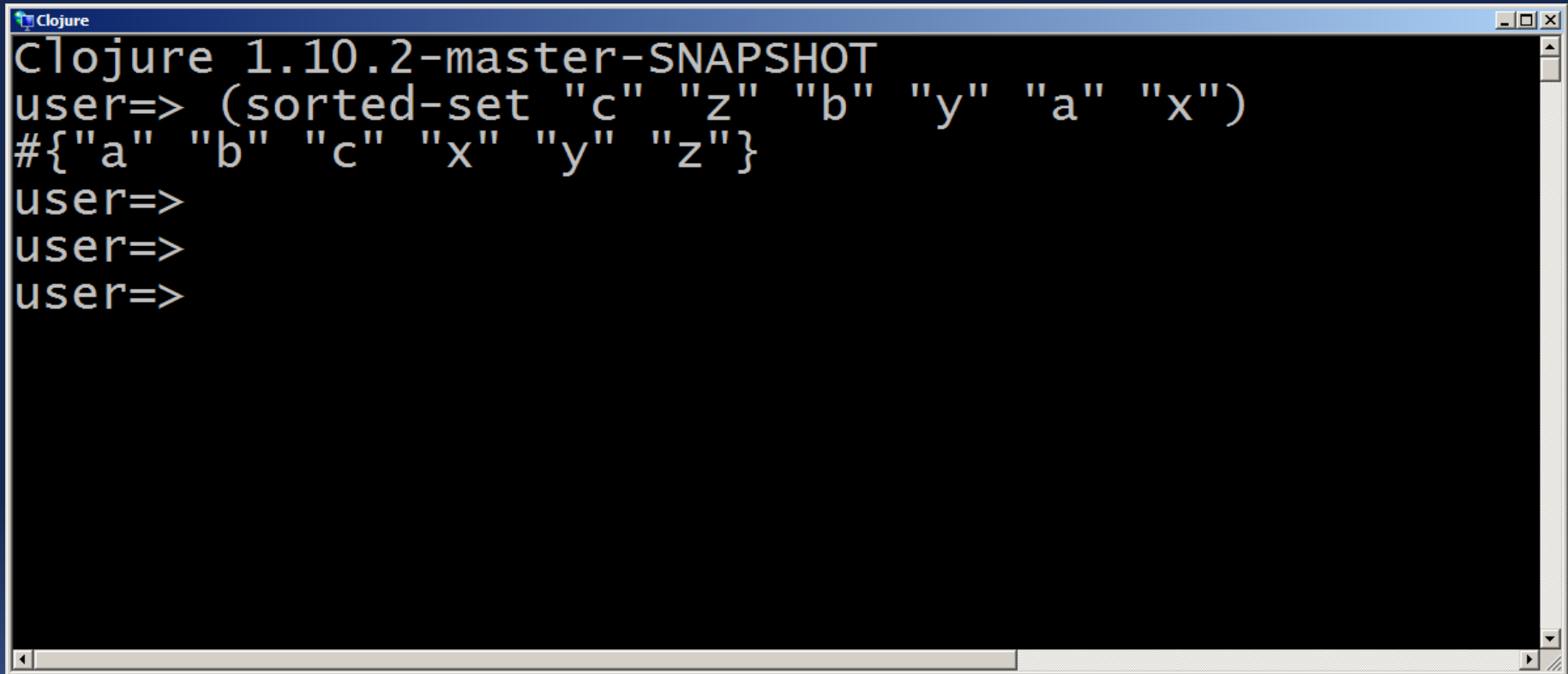
Função sorted-set

- ✓ Um Sorted Set pode ser criado pela função **sorted-set** e **não** possui notação literal para sua definição, como Hash Sets têm.



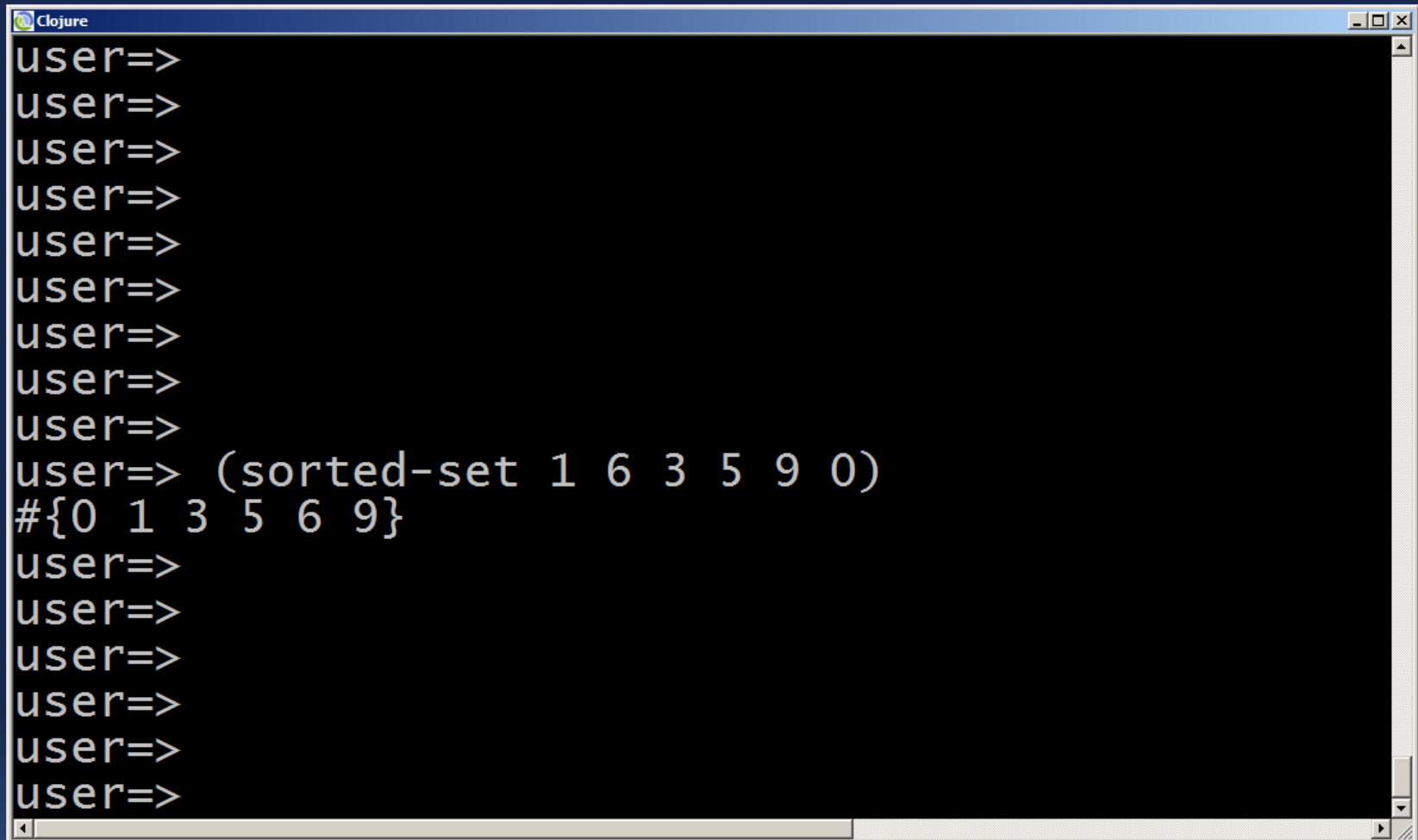
Função sorted-set

- ✓ Um Sorted Set pode ser criado pela função **sorted-set** e **não** possui notação literal para sua definição, como Hash Sets têm.



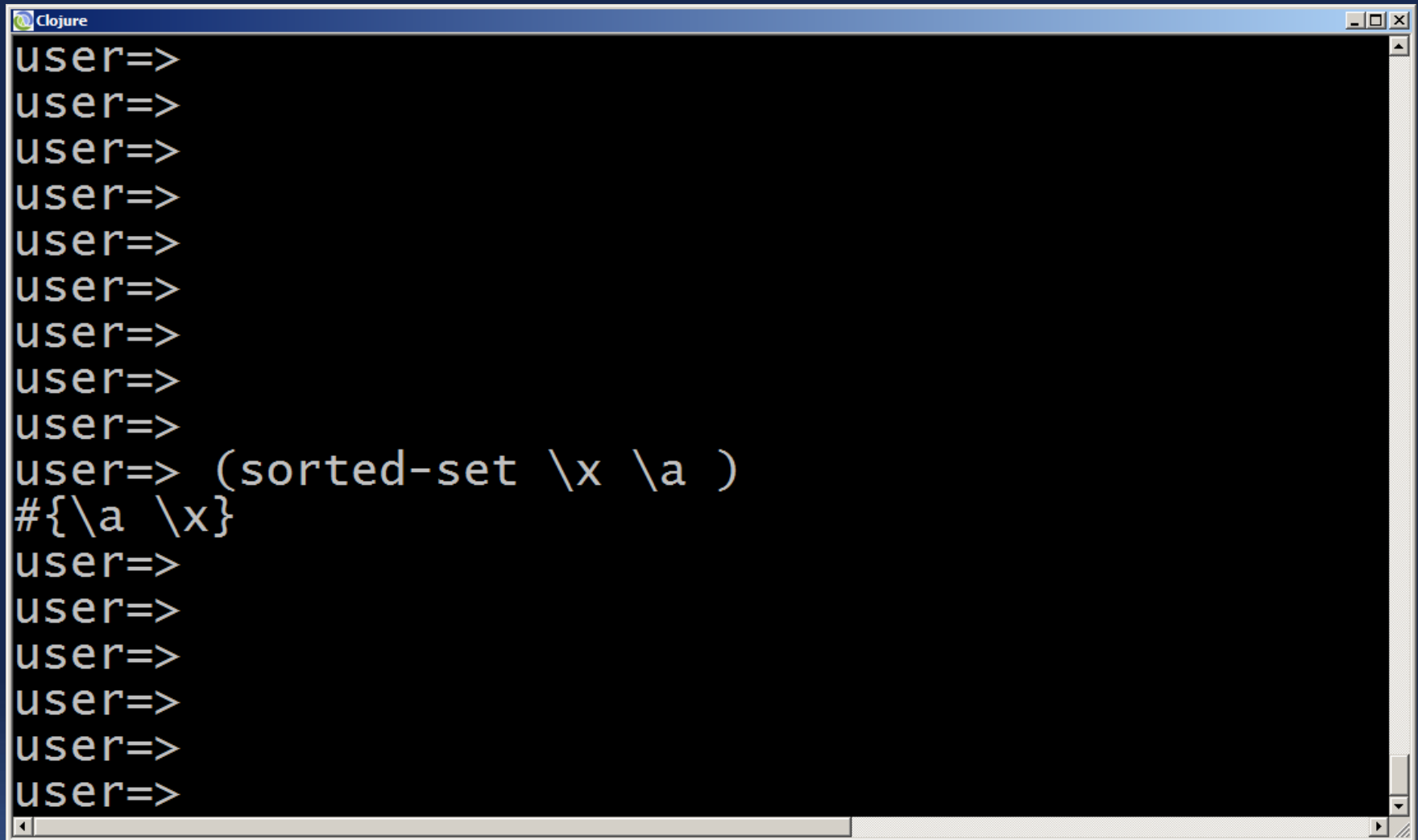
```
Clojure 1.10.2-master-SNAPSHOT
user=> (sorted-set "c" "z" "b" "y" "a" "x")
#{"a" "b" "c" "x" "y" "z"}
user=>
user=>
user=>
```





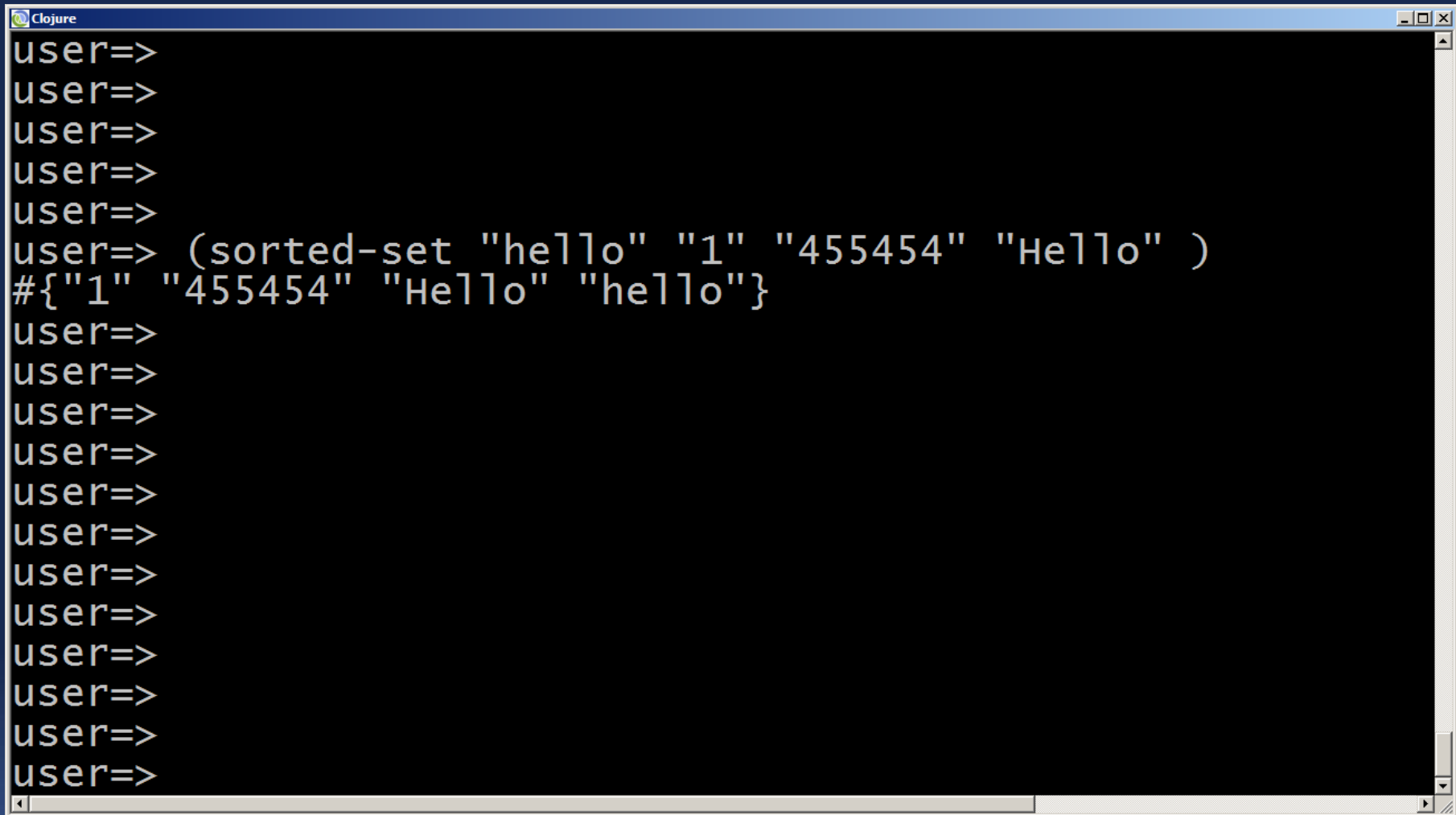
A screenshot of a Clojure REPL window. The window has a title bar with the Clojure logo and the word "Clojure". The background is black with white text. The text shows a series of "user=>" prompts. The 11th prompt is followed by the expression "(sorted-set 1 6 3 5 9 0)", and the 12th prompt shows the result "#{0 1 3 5 6 9}". The rest of the prompts are empty.

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (sorted-set 1 6 3 5 9 0)
#{0 1 3 5 6 9}
user=>
user=>
user=>
user=>
user=>
user=>
```



A screenshot of a Clojure REPL window. The window has a title bar with the Clojure logo and the word "Clojure". The background is black, and the text is white. The REPL shows a series of prompts "user=>" followed by the command "(sorted-set \x \a)" and its output "#{\a \x}". There are several empty prompts below the output.

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (sorted-set \x \a )
#{\a \x}
user=>
user=>
user=>
user=>
user=>
user=>
```



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (sorted-set "hello" "1" "455454" "Hello" )
#{ "1" "455454" "Hello" "hello" }
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

Exercícios com sets

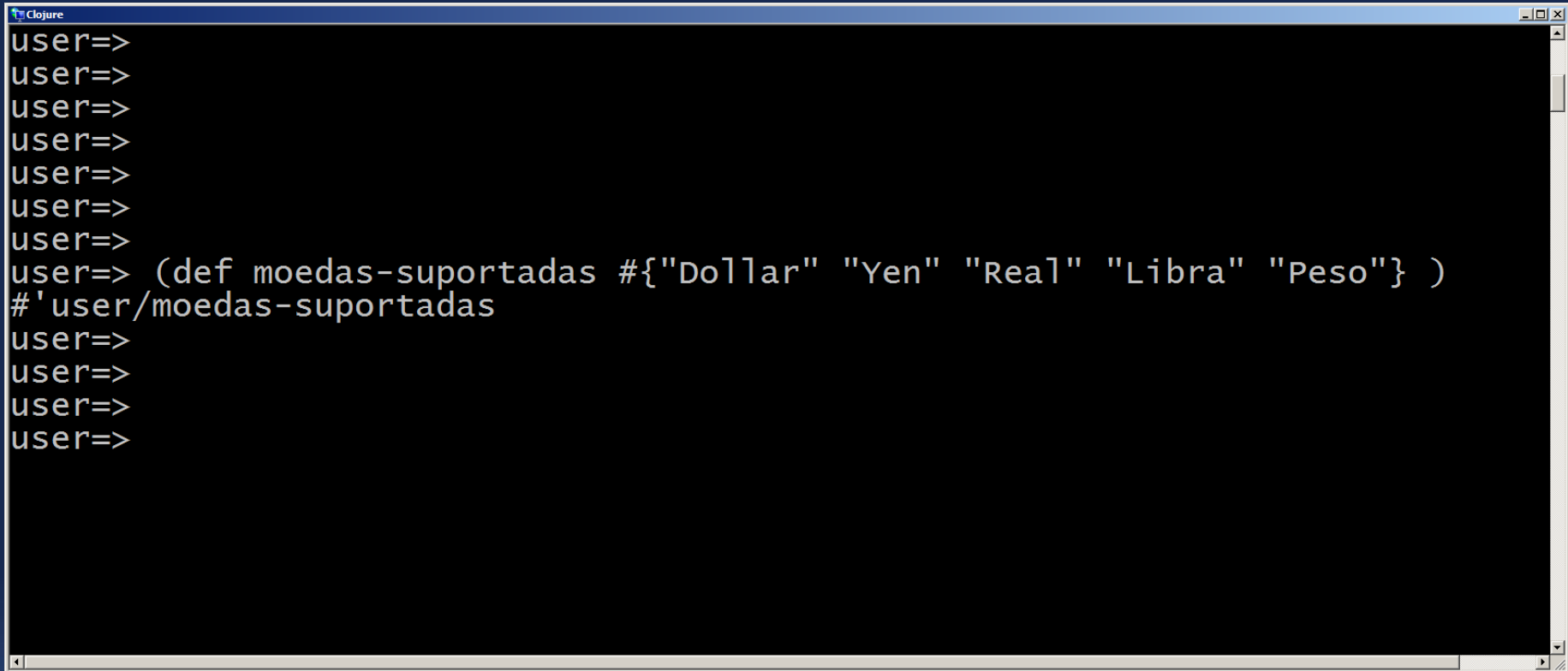


1. Sob **REPL**, crie um **set** e o ligue (**bind**) ao símbolo **moedas-suportadas**:

Definir no set os valores: “Dollar” , “Yen”, “Real” , “Peso”



1. Sob **REPL**, crie um **set** e o ligue (**bind**) ao símbolo **moedas-suportadas**:



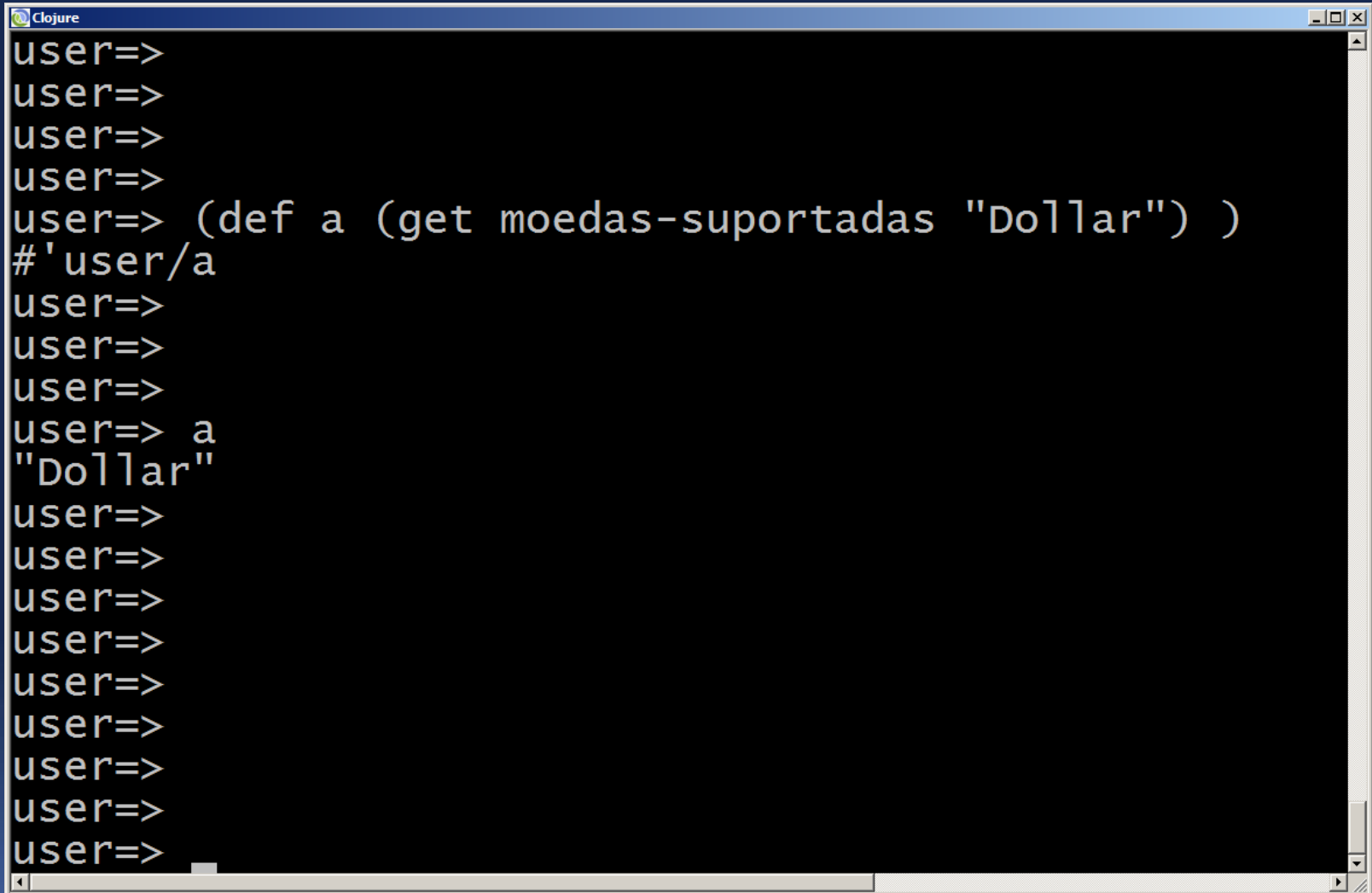
```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (def moedas-suportadas #{"Dollar" "Yen" "Real" "Libra" "Peso"})
#'user/moedas-suportadas
user=>
user=>
user=>
user=>
```



2. Assim como **maps**, pode-se usar a função **get** para se recuperar uma entrada de um **set**.



2. Assim como **maps**, pode-se usar a função **get** para se recuperar uma entrada de um **set**.



```
Clojure
user=>
user=>
user=>
user=>
user=> (def a (get moedas-suportadas "Dollar") )
#'user/a
user=>
user=>
user=>
user=> a
"Dollar"
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

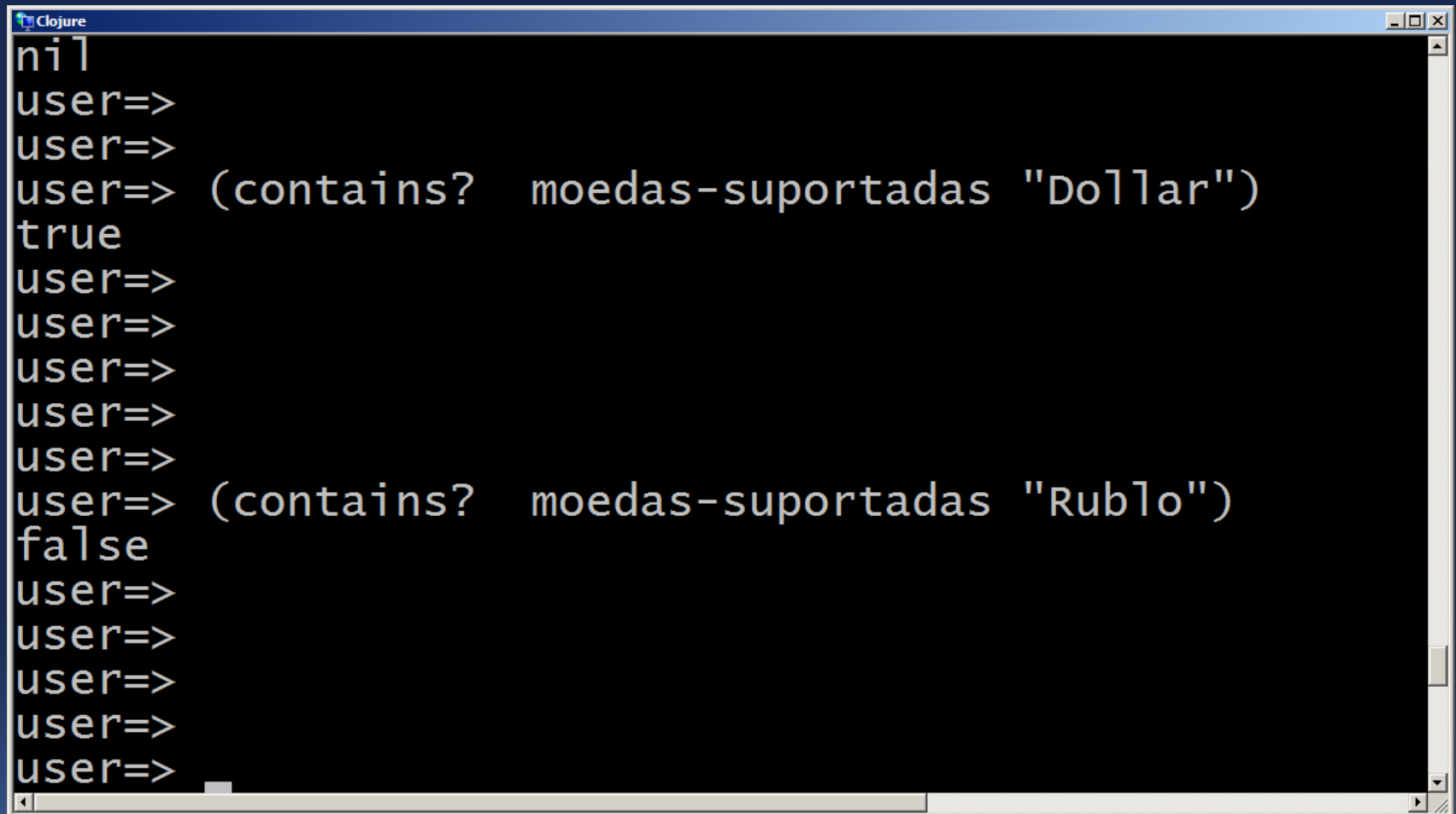


3. Para checar se um elemento pertence ao set, pode-se usar a função **contains?**.

Essa função retorna um valor Boolean.



3. Para checar se um elemento pertence ao set, pode-se usar a função `contains?`. Essa função retorna um valor Boolean.



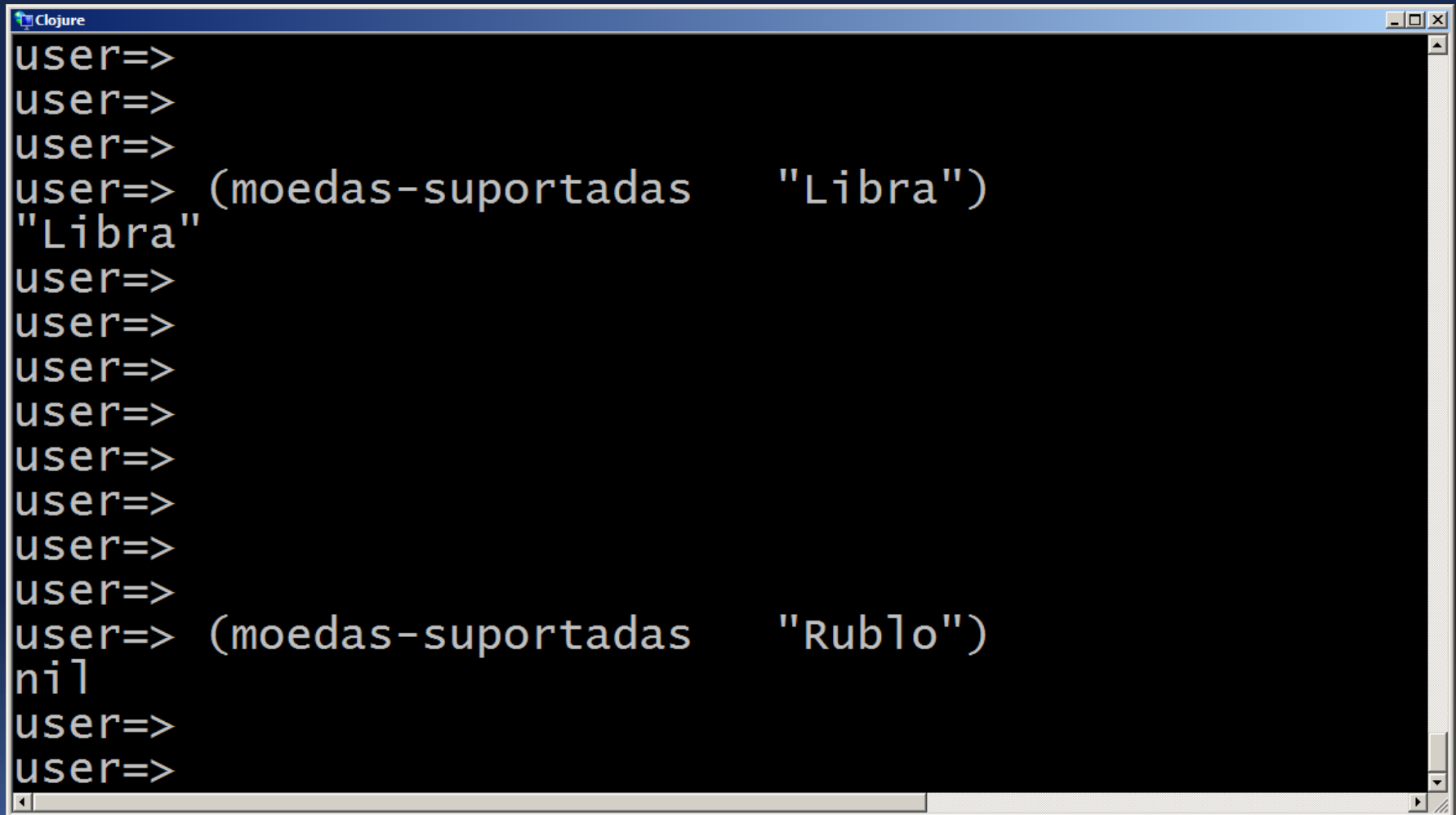
```
Clojure
nil
user=>
user=>
user=> (contains? moedas-suportadas "Dollar")
true
user=>
user=>
user=>
user=>
user=> (contains? moedas-suportadas "Rublo")
false
user=>
user=>
user=>
user=>
user=>
```



4. Da mesma forma como maps, **sets** e **keywords** podem ser usados como funções.



4. Da mesma forma como maps, **sets** e **keywords** podem ser usados como funções.



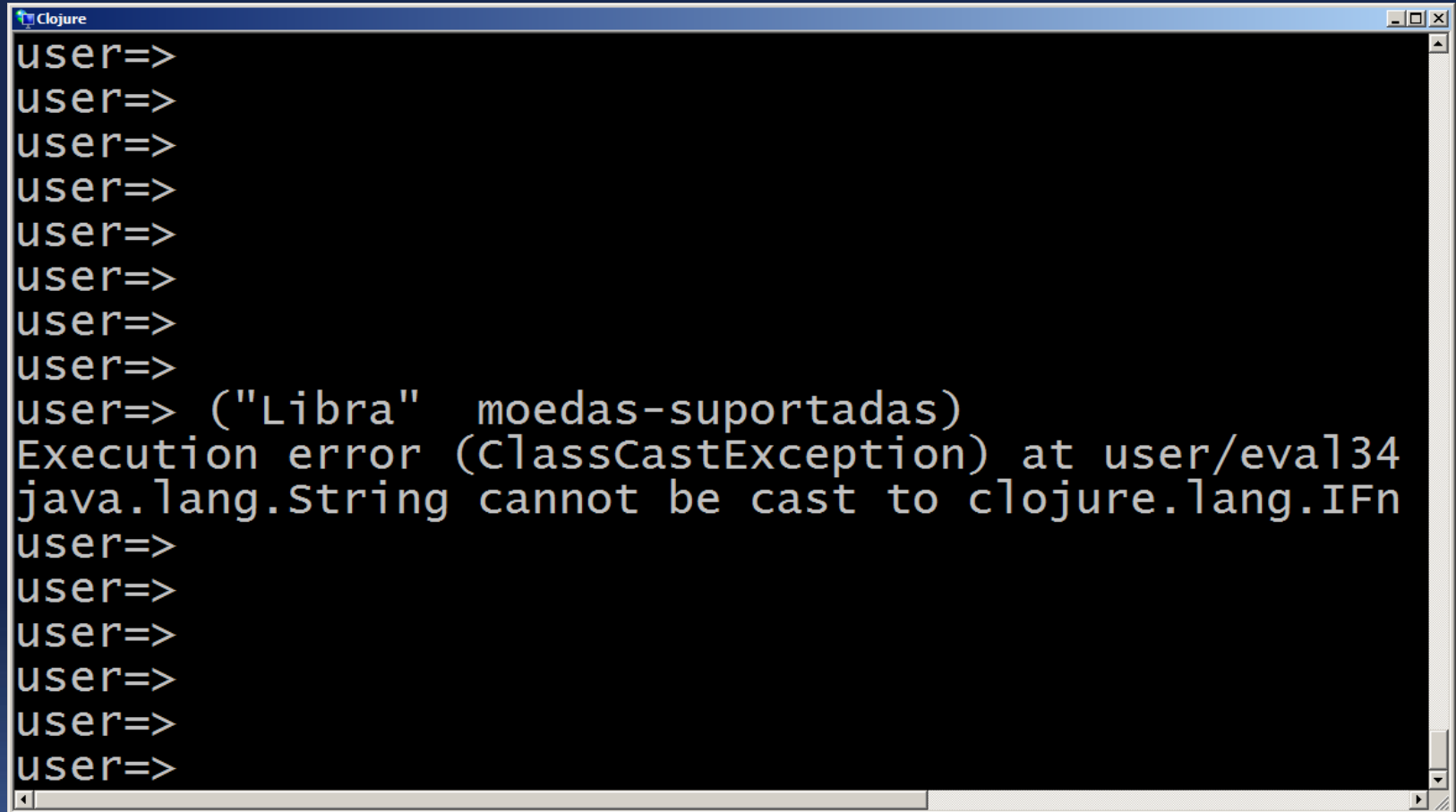
```
Clojure
user=>
user=>
user=>
user=> (moedas-suportadas "Libra")
"Libra"
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (moedas-suportadas "Rublo")
nil
user=>
user=>
```



5. No entanto, **não** se pode usar strings como funções para se pesquisar um valor em um set ou em um map.



5. No entanto, **não** se pode usar strings como funções para se pesquisar um valor em um set ou em um map.



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> ("Libra" moedas-suportadas)
Execution error (ClassCastException) at user/eval34:
java.lang.String cannot be cast to clojure.lang.IFn
user=>
user=>
user=>
user=>
user=>
user=>
```



6. Para se adicionar uma entrada em um **set**, pode-se usar a função **conj**.

Para a função **conj**, pode-se passar mais de um argumento.



6. Para se adicionar uma entrada em um **set**, pode-se usar a função **conj**. Para a função **conj**, pode-se passar mais de um argumento.

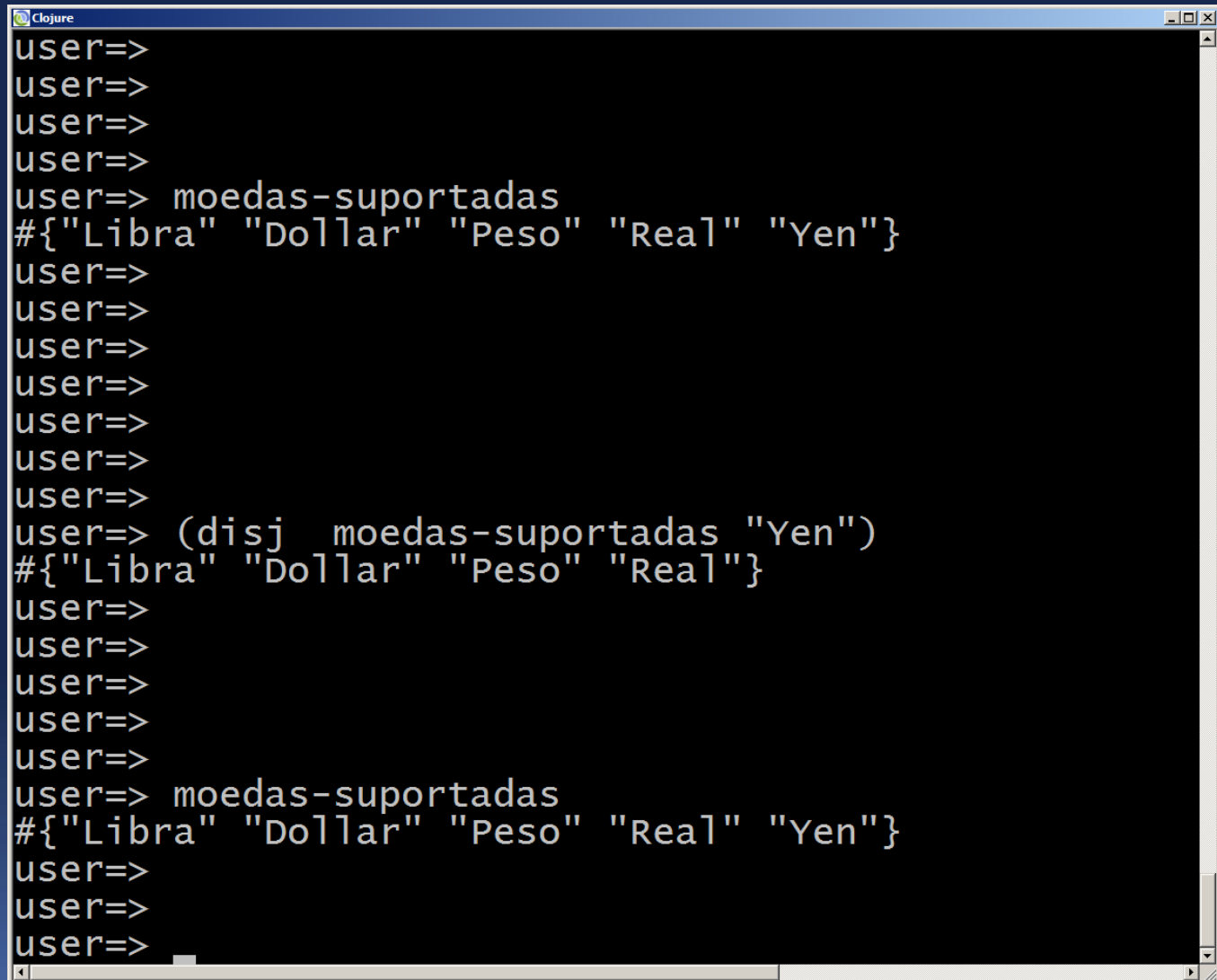
```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (conj moedas-suportadas "Rublo" )
#{"Libra" "Dollar" "Peso" "Rublo" "Real" "Yen"}
user=>
user=>
user=> (conj moedas-suportadas "Euro" "Franco Suíço")
#{"Libra" "Euro" "Dollar" "Franco Suíço" "Peso" "Real" "Yen"}
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```



7. Finalmente, pode-se **remove** um ou mais itens de um set com a função **disj**.



7. Finalmente, pode-se **remover** um ou mais itens de um set com a função **disj**.



```
Clojure
user=>
user=>
user=>
user=>
user=> moedas-suportadas
#{ "Libra" "Dollar" "Peso" "Real" "Yen" }
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (disj moedas-suportadas "Yen")
#{ "Libra" "Dollar" "Peso" "Real" }
user=>
user=>
user=>
user=>
user=> moedas-suportadas
#{ "Libra" "Dollar" "Peso" "Real" "Yen" }
user=>
user=>
user=>
```



Vectors



Vectors

- ✓ Um **vector** é um outro tipo de **collection** que é largamente empregado na Linguagem Clojure;
- ✓ Pode-se imaginar um vector como sendo um **poderoso array imutável**;
- ✓ **Vectors** são coleções de valores acessíveis de forma eficiente por inteiros (**índices**) que começam com **0**;
- ✓ A **ordem** da inserção dos itens de dados é mantida, bem como permite-se valores **duplicados**.
- ✓ Usados quando se necessita armazenar e recuperar elementos seguindo um critério de ordem;
- ✓ **Vectors** têm uma notação literal definida por **[]**.



Vectors

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> [ 1 2 3 ]
[1 2 3]
user=>
user=>
user=>
user=> [ 5 3 8 1 ]
[5 3 8 1]
user=>
user=>
user=>
user=>
```



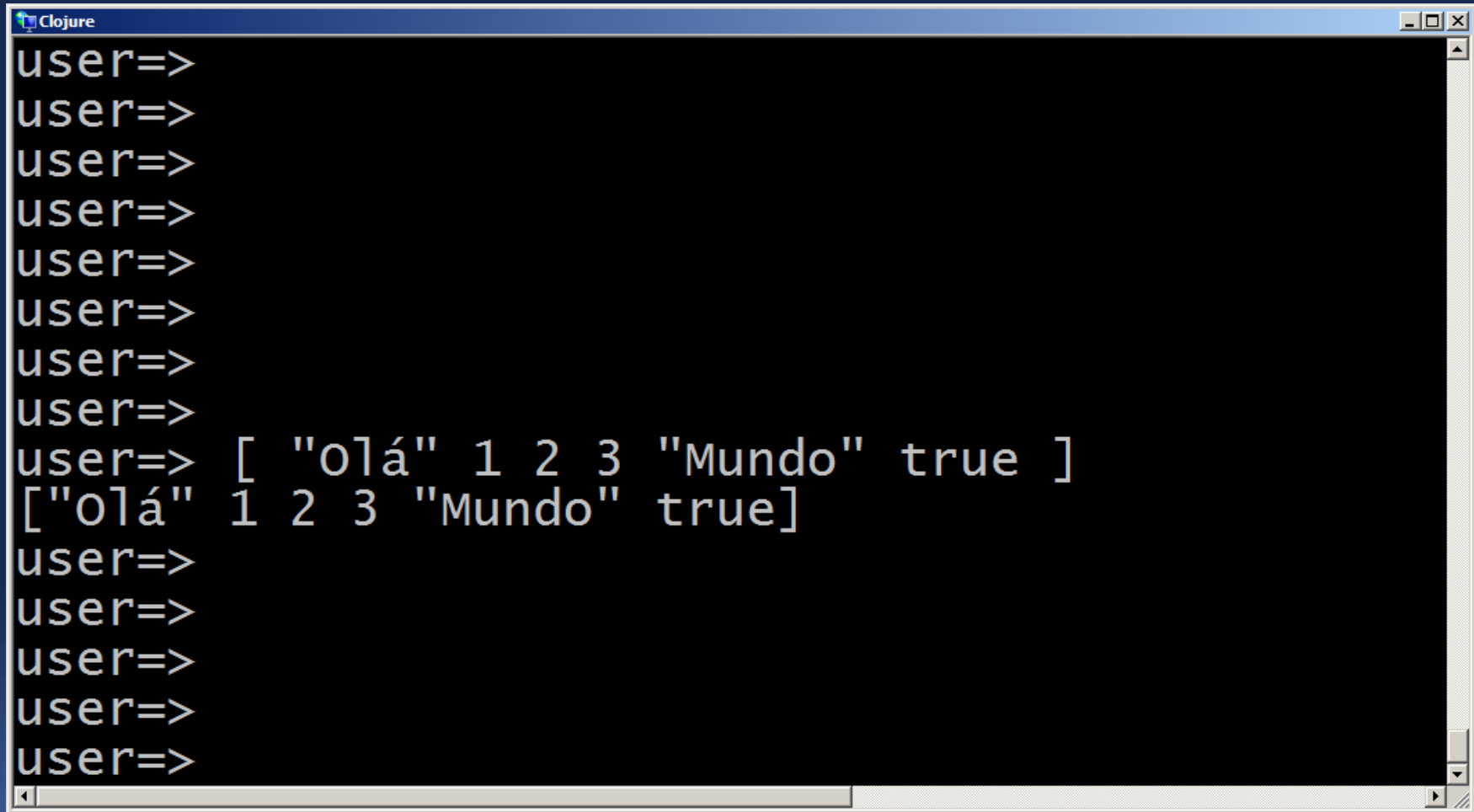
Vectors

- ✓ Assim como em outras coleções, um vector pode conter diferentes **tipos** de valores:



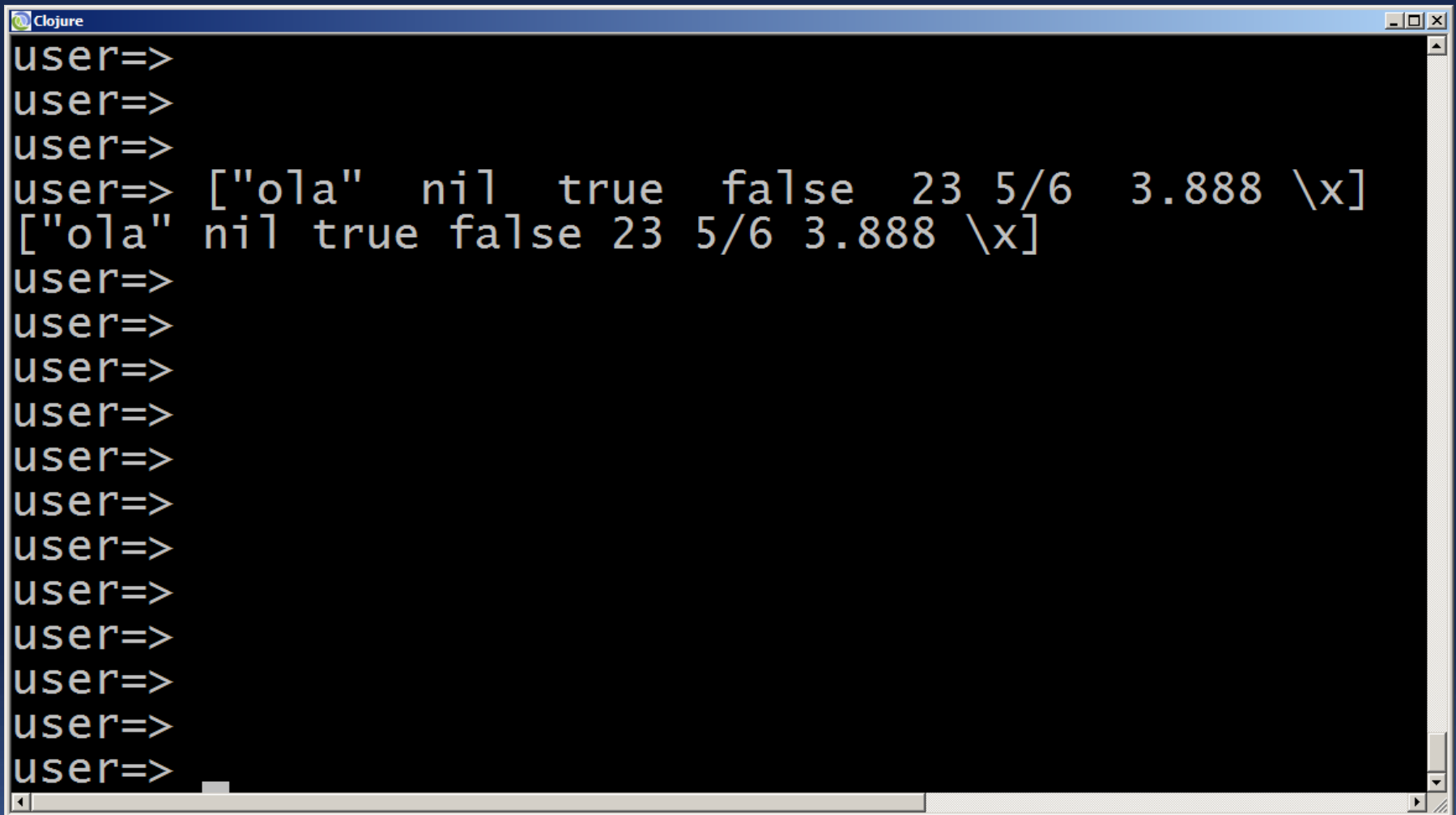
Vectors

- ✓ Assim como em outras coleções, um vector pode conter diferentes tipos de valores:



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> [ "olá" 1 2 3 "Mundo" true ]
["olá" 1 2 3 "Mundo" true]
user=>
user=>
user=>
user=>
user=>
```





A screenshot of a Clojure REPL window. The window has a title bar with the Clojure logo and the text "Clojure". The background is black, and the text is white. The REPL shows a series of prompts "user=>" and their corresponding outputs. The first output is a vector containing the elements: "ola", nil, true, false, 23, 5/6, 3.888, and \x. The second output is the same vector as the first. The remaining prompts are followed by no output, indicating the end of the session or a new line.

```
Clojure
user=>
user=>
user=>
user=> ["ola" nil true false 23 5/6 3.888 \x]
["ola" nil true false 23 5/6 3.888 \x]
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

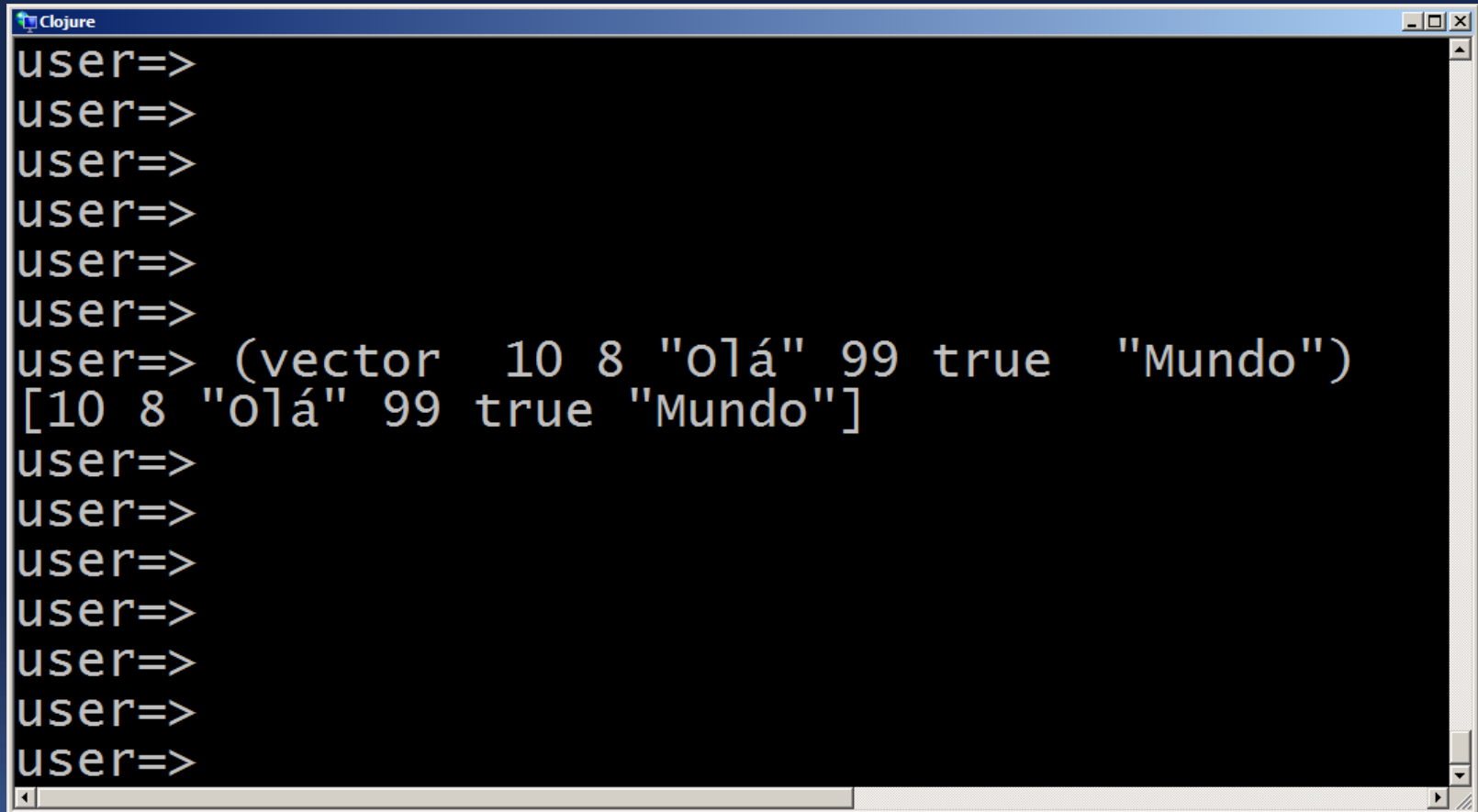
Vectors

- ✓ Um vector pode também ser criado com a função **vector**:



Vectors

- ✓ Um vector pode também ser criado com a função **vector**:

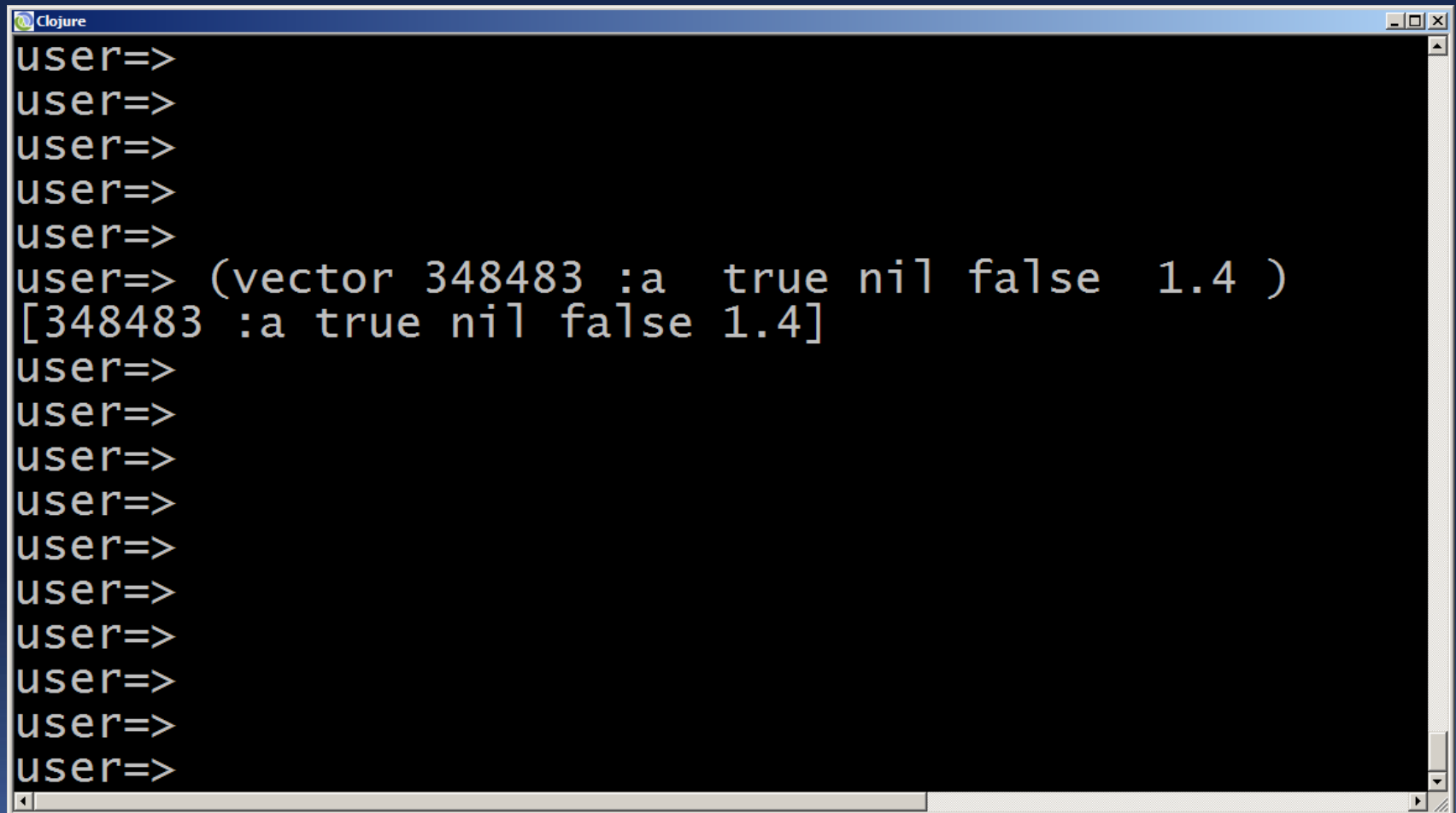


```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> (vector 10 8 "olá" 99 true "Mundo")
[10 8 "olá" 99 true "Mundo"]
user=>
user=>
user=>
user=>
user=>
user=>
```



Vectors

- ✓ Um vector pode também ser criado com a função **vector**:



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (vector 348483 :a true nil false 1.4 )
[348483 :a true nil false 1.4]
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```



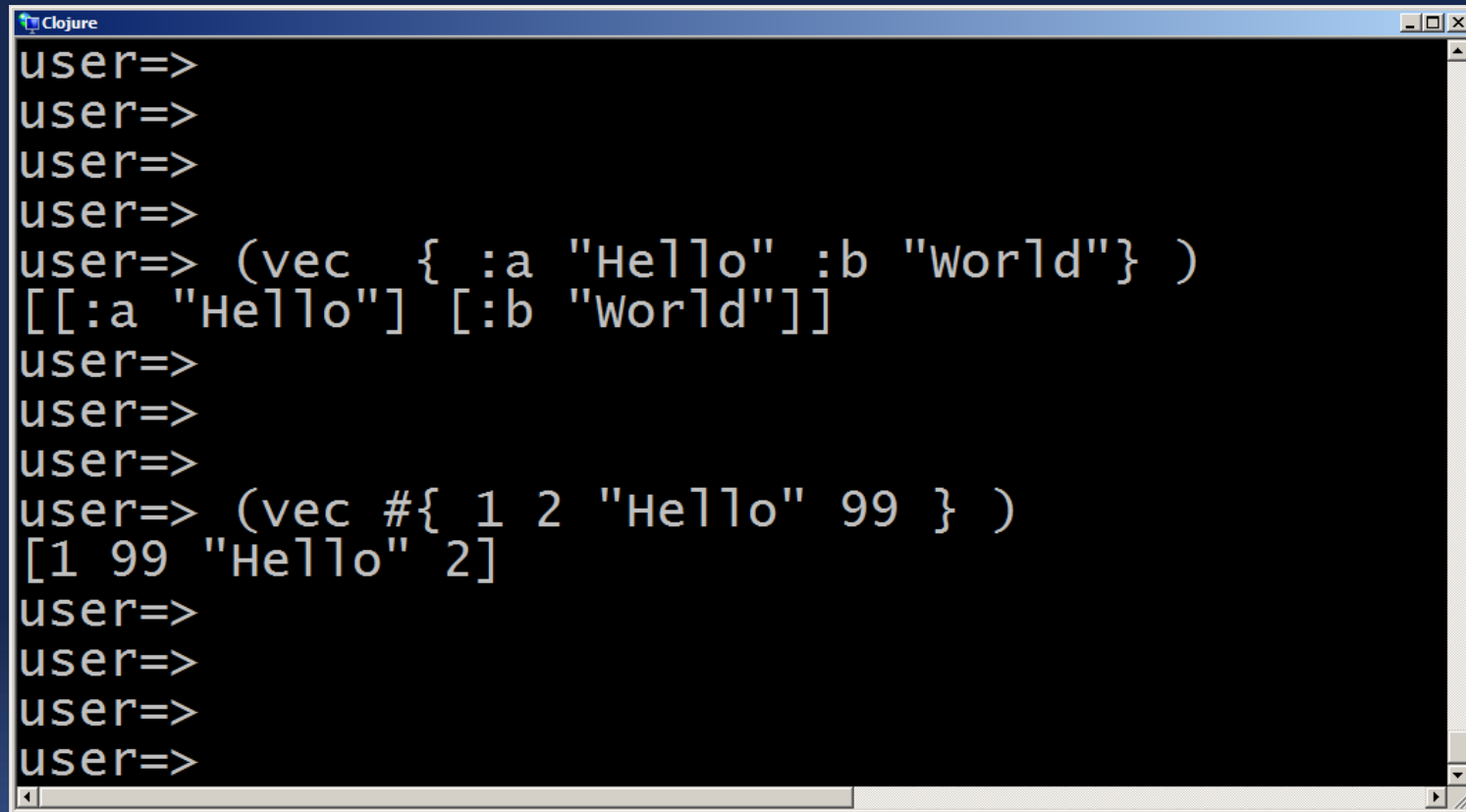
Vectors

- ✓ Com a função **vec**, pode-se criar um vector a partir de outras collections:



Vectors

- ✓ Com a função **vec**, pode-se criar um vector a partir de outras collections:

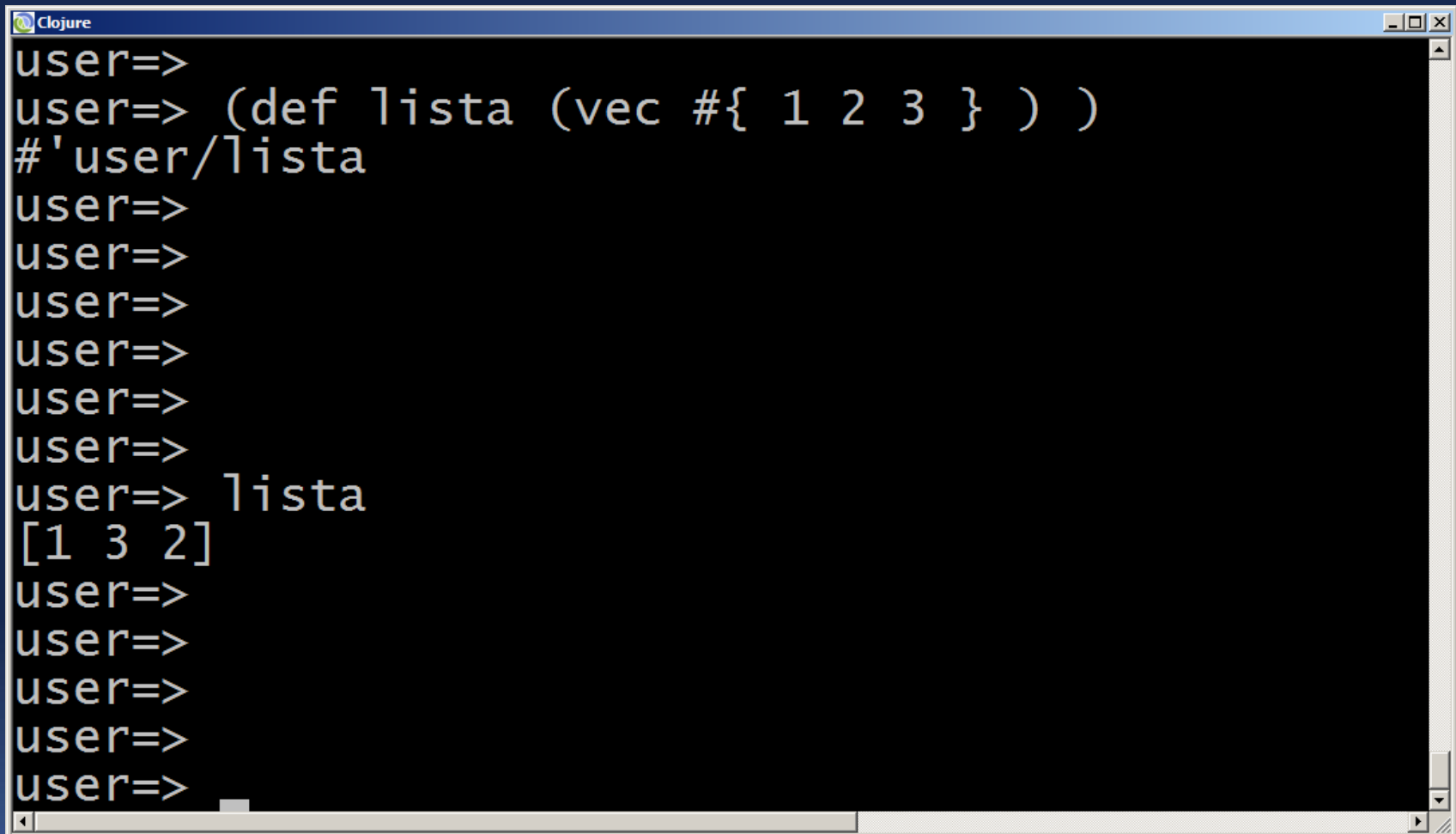


```
Clojure
user=>
user=>
user=>
user=>
user=> (vec { :a "Hello" :b "World" } )
[[:a "Hello"] [:b "World"]]
user=>
user=>
user=>
user=> (vec #{ 1 2 "Hello" 99 } )
[1 99 "Hello" 2]
user=>
user=>
user=>
user=>
```



Vectors

- ✓ Com a função **vec**, pode-se criar um vector a partir de outras collections:

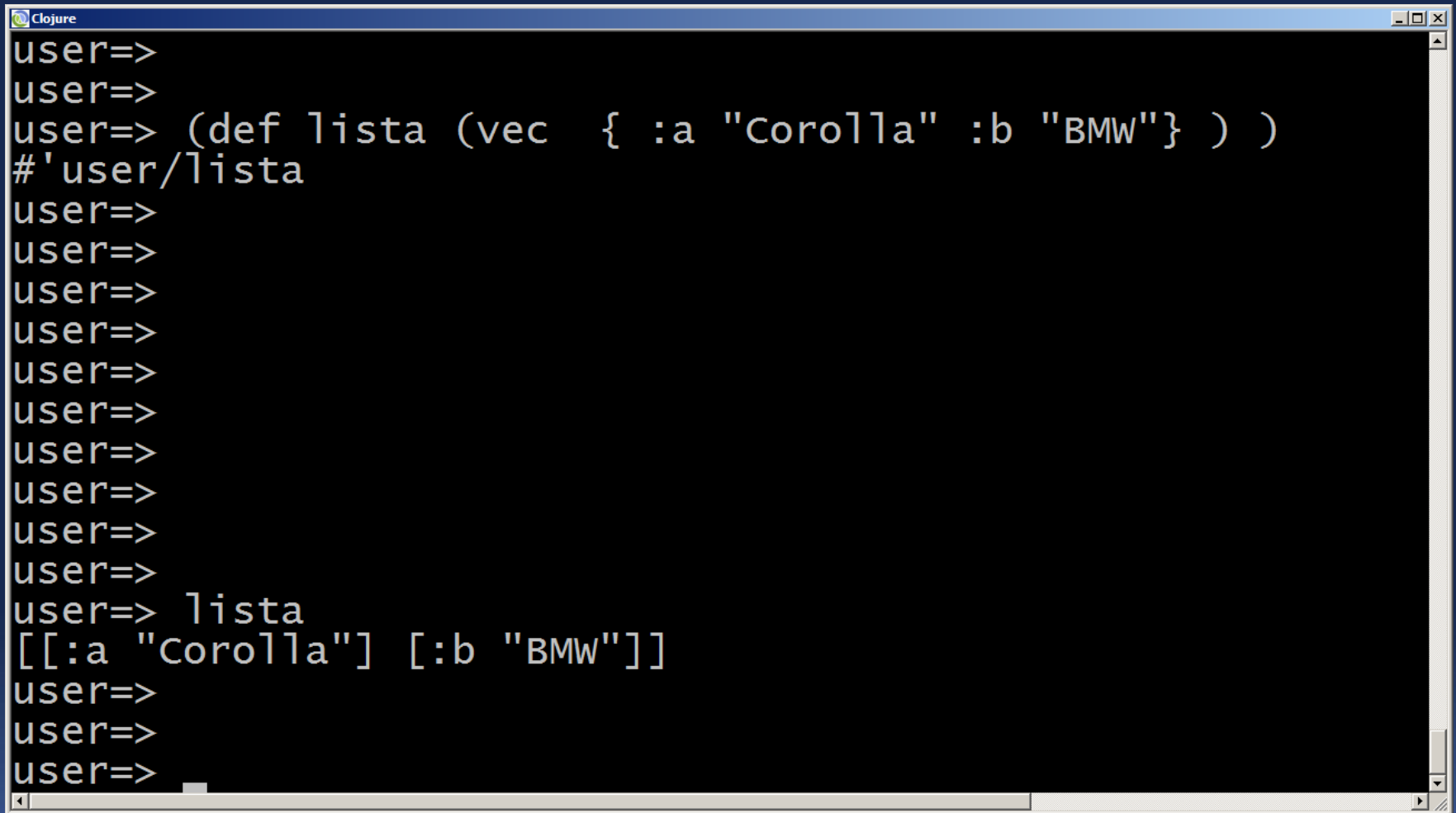


```
Clojure
user=>
user=> (def lista (vec #{ 1 2 3 } ) )
#'user/lista
user=>
user=>
user=>
user=>
user=>
user=>
user=> lista
[1 3 2]
user=>
user=>
user=>
user=>
user=>
```



Vectors

- ✓ Com a função **vec**, pode-se criar um vector a partir de outras collections:

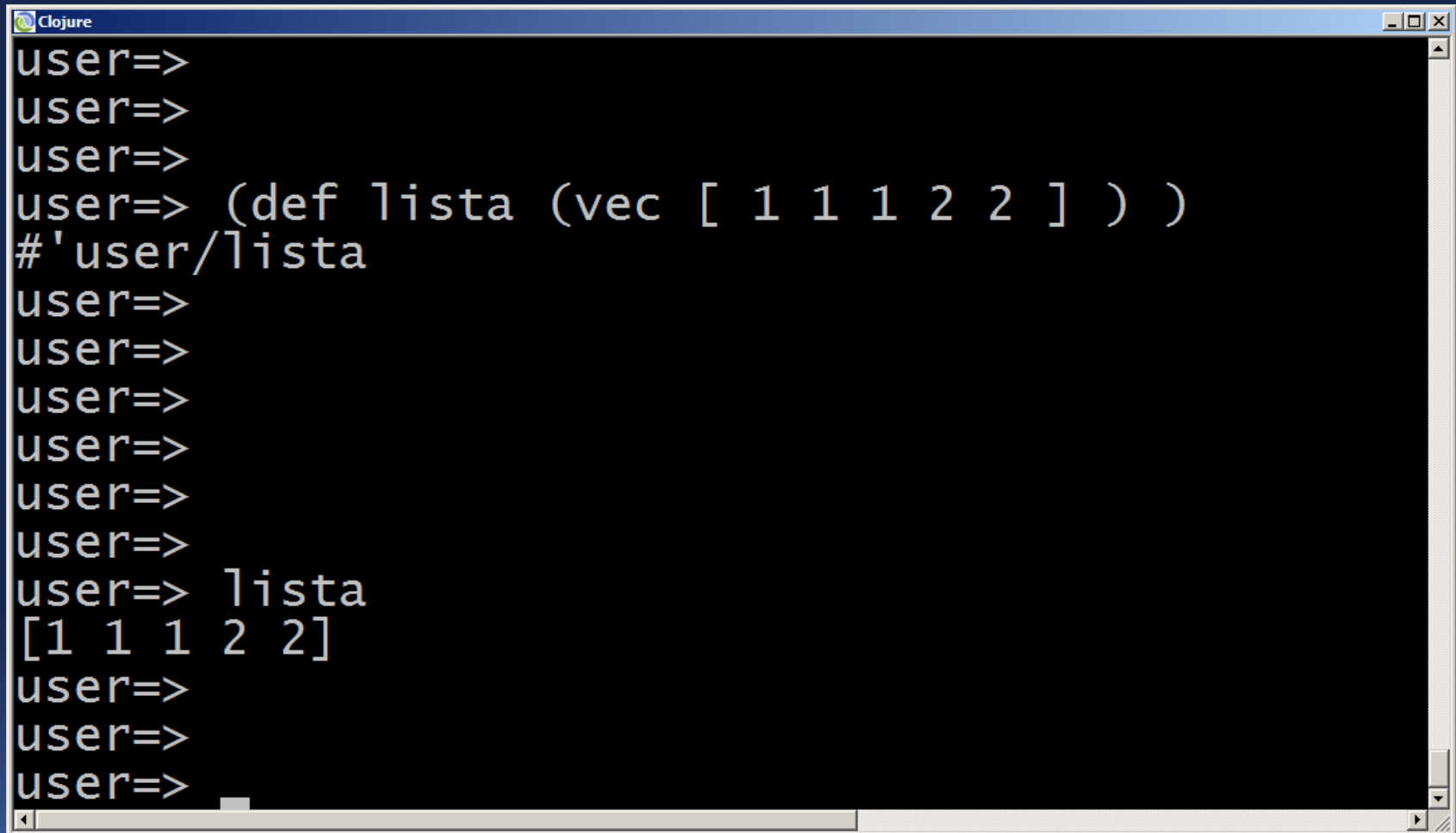


```
Clojure
user=>
user=>
user=> (def lista (vec { :a "Corolla" :b "BMW" } ) )
#'user/lista
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> lista
[[:a "Corolla"] [:b "BMW"]]
user=>
user=>
user=>
```



Vectors

- ✓ Com a função **vec**, pode-se criar um vector a partir de outras collections:

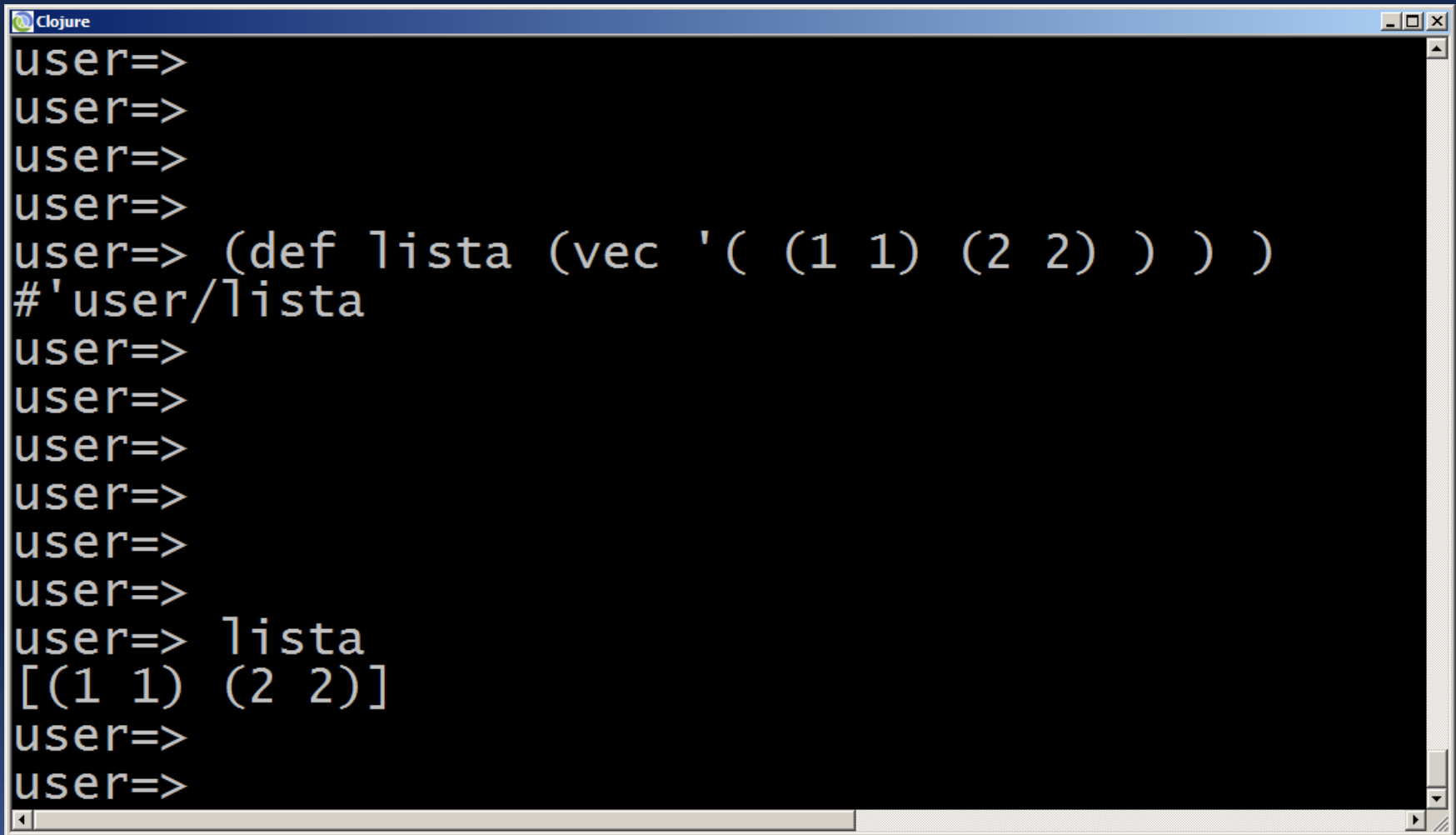


```
Clojure
user=>
user=>
user=>
user=> (def lista (vec [ 1 1 1 2 2 ] ) )
#'user/lista
user=>
user=>
user=>
user=>
user=>
user=>
user=> lista
[1 1 1 2 2]
user=>
user=>
user=>
```



Vectors

- ✓ Com a função **vec**, pode-se criar um vector a partir de outras collections:



```
Clojure
user=>
user=>
user=>
user=>
user=> (def lista (vec '( (1 1) (2 2) ) ) )
#'user/lista
user=>
user=>
user=>
user=>
user=>
user=>
user=> lista
[(1 1) (2 2)]
user=>
user=>
```



Exercícios com vectors

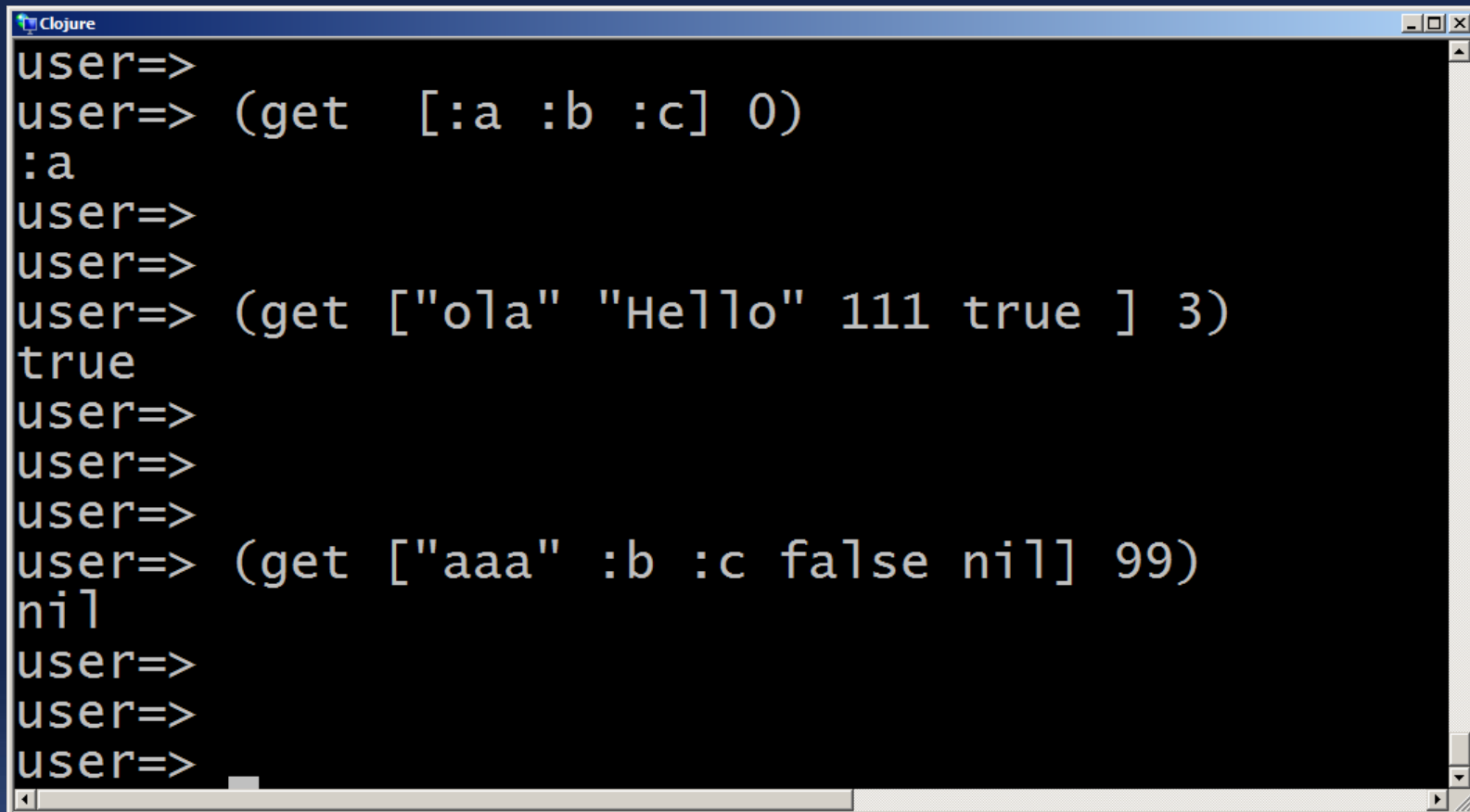


1. Inicie **REPL**. Pode-se pesquisar valores em um **vector** por meio da função **get**.

O **índice** é passado como argumento.



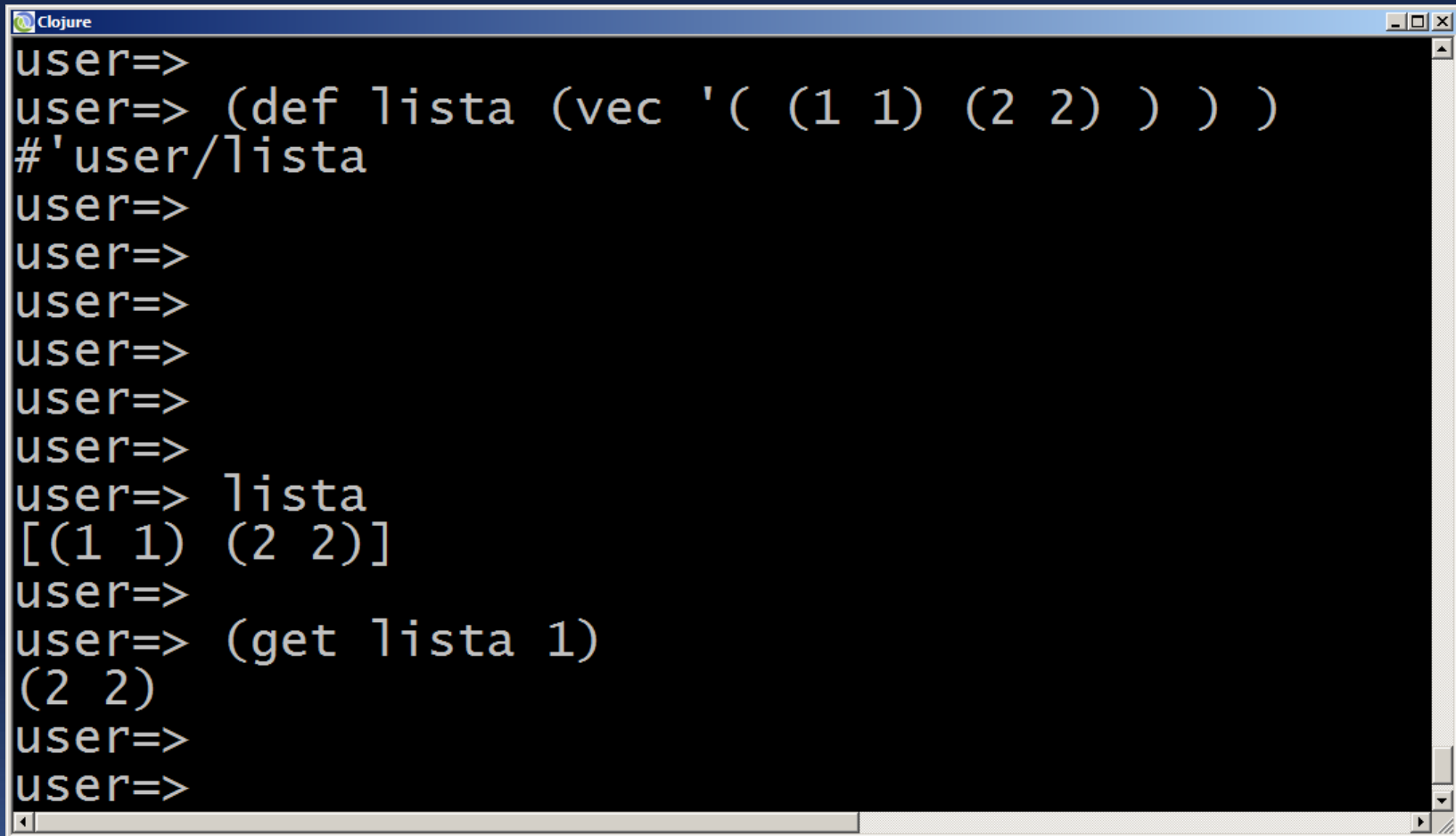
1. Inicie **REPL**. Pode-se pesquisar valores em um **vector** por meio da função **get**. O índice é passado como argumento.



```
Clojure
user=>
user=> (get [:a :b :c] 0)
:a
user=>
user=>
user=> (get ["ola" "Hello" 111 true ] 3)
true
user=>
user=>
user=>
user=> (get ["aaa" :b :c false nil] 99)
nil
user=>
user=>
user=>
```



1. Inicie **REPL**. Pode-se pesquisar valores em um **vector** por meio da função **get**. O índice é passado como argumento.



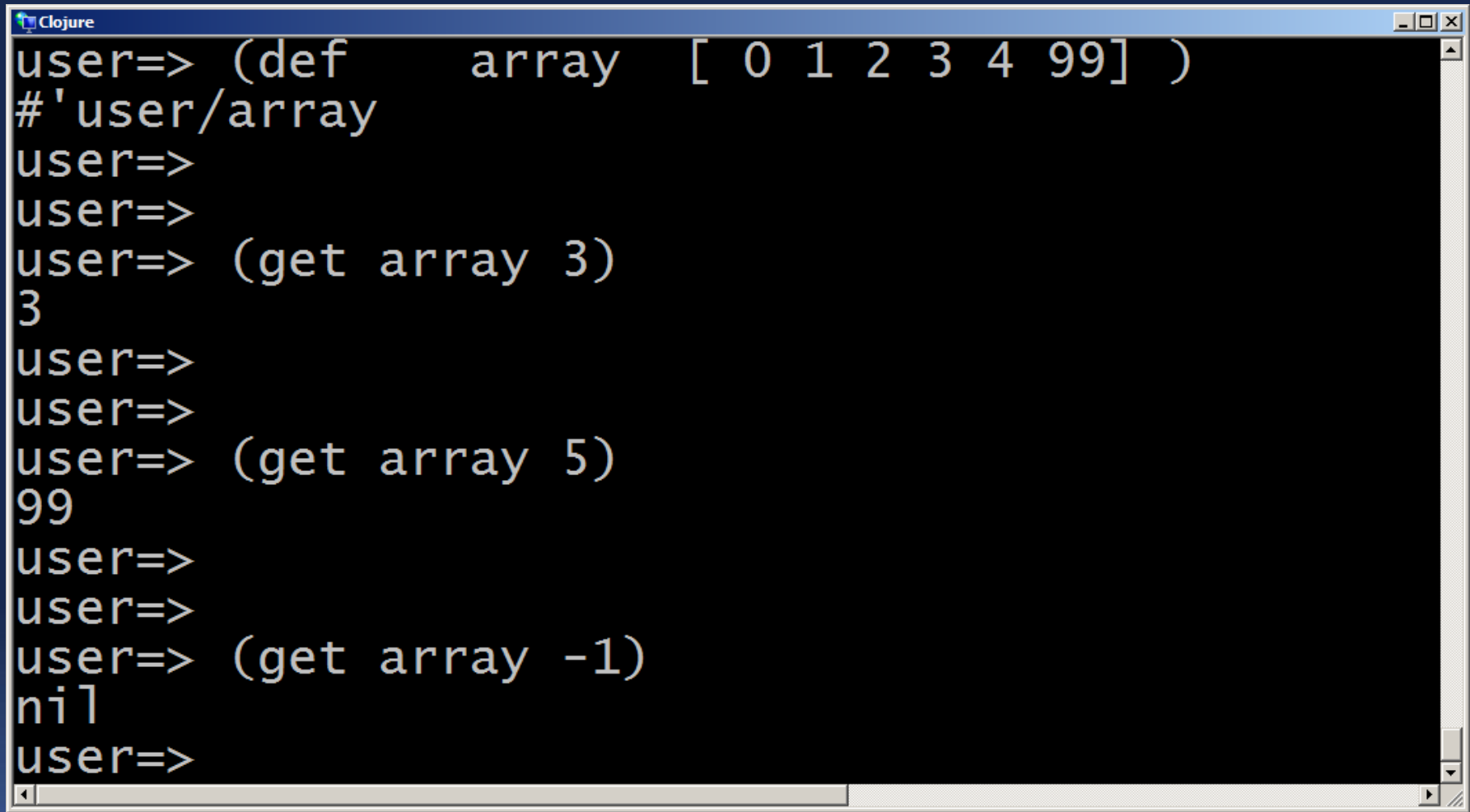
```
Clojure
user=>
user=> (def lista (vec '( (1 1) (2 2) ) ) )
#'user/lista
user=>
user=>
user=>
user=>
user=>
user=>
user=> lista
[(1 1) (2 2)]
user=>
user=> (get lista 1)
(2 2)
user=>
user=>
```



2. Binding de um **vector** a um símbolo tornam a escrita mais prática:



2. **Binding** de um **vector** a um símbolo tornam a escrita mais prática:



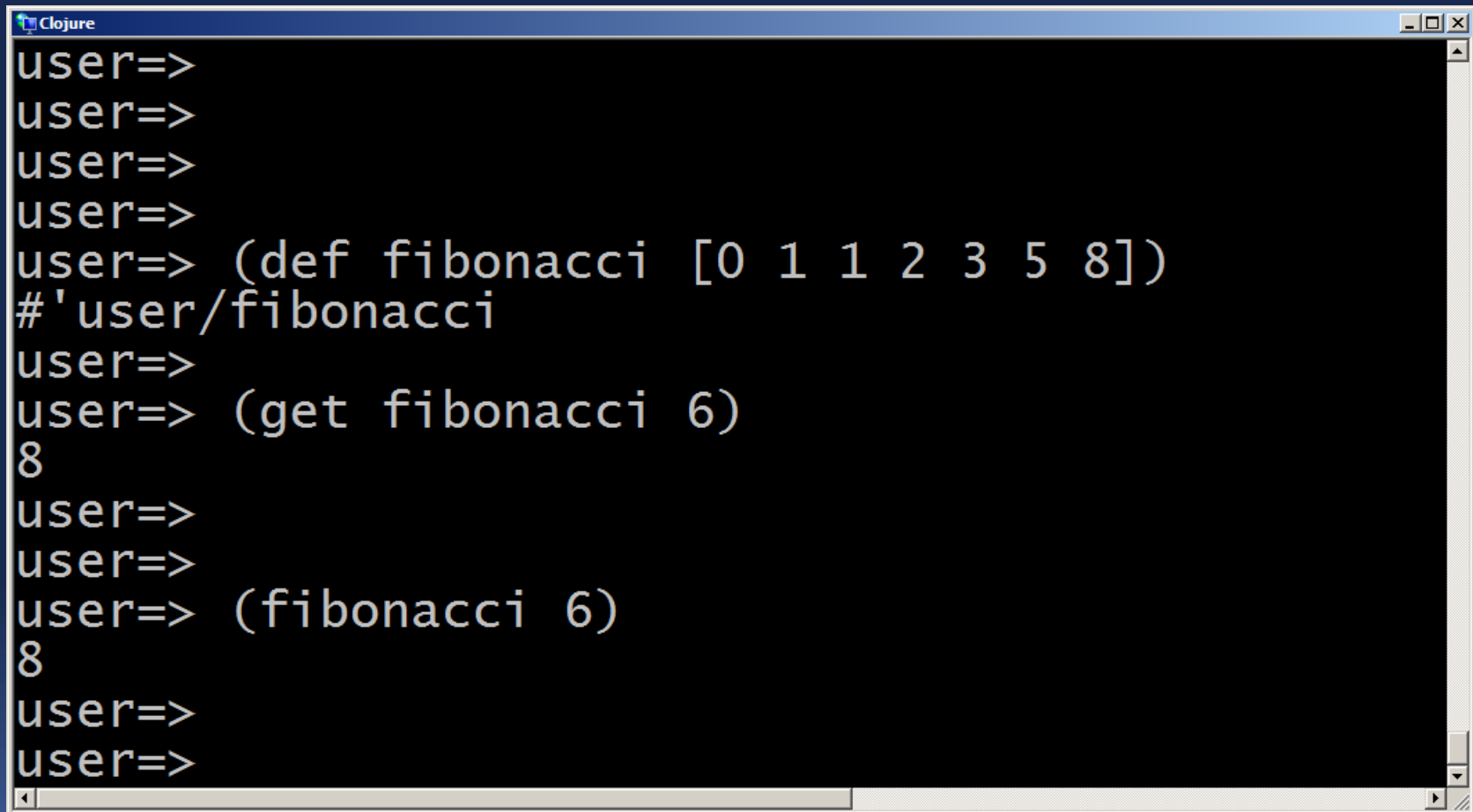
```
Clojure
user=> (def array [0 1 2 3 4 99])
#'user/array
user=>
user=>
user=> (get array 3)
3
user=>
user=>
user=> (get array 5)
99
user=>
user=>
user=> (get array -1)
nil
user=>
```



3. Da mesma forma que com **maps** e **sets**, pode-se usar o **vector** como uma função para se pesquisar itens, mas para **vectors**, o parâmetro é o **índice** do valor no **vector**.



3. Da mesma forma que com **maps** e **sets**, pode-se usar o **vector** como uma função para se pesquisar itens, mas para **vectors**, o parâmetro é o **índice** do valor no **vector**.



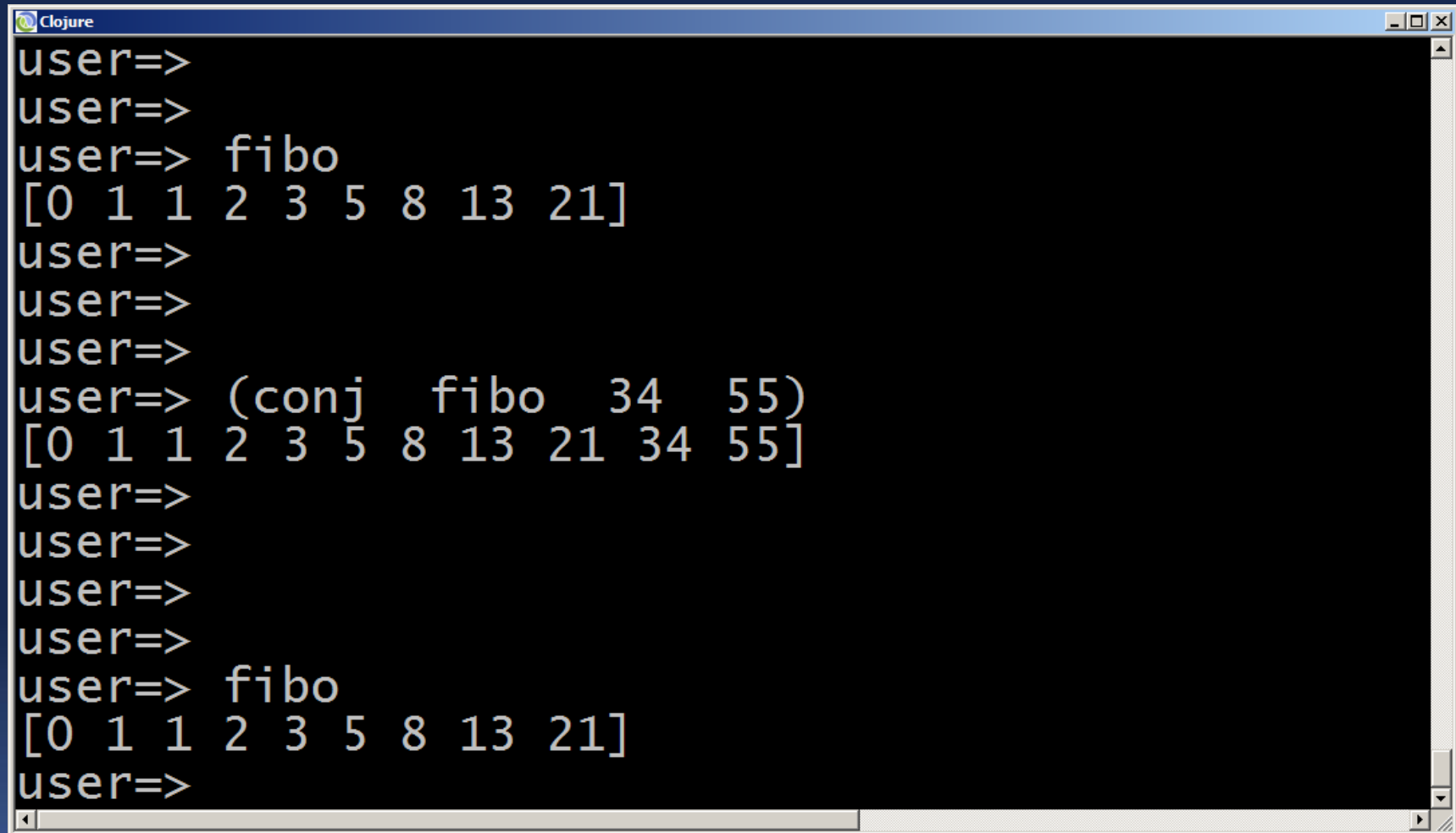
```
Clojure
user=>
user=>
user=>
user=>
user=> (def fibonacci [0 1 1 2 3 5 8])
#'user/fibonacci
user=>
user=> (get fibonacci 6)
8
user=>
user=>
user=> (fibonacci 6)
8
user=>
user=>
```



4. Para se adicionar valores em um vector, usa-se a função **conj**:



4. Para se adicionar valores em um vector, usa-se a função **conj**:



```
Clojure
user=>
user=>
user=> fibo
[0 1 1 2 3 5 8 13 21]
user=>
user=>
user=>
user=> (conj fibo 34 55)
[0 1 1 2 3 5 8 13 21 34 55]
user=>
user=>
user=>
user=>
user=> fibo
[0 1 1 2 3 5 8 13 21]
user=>
```





Lists



Lists

- ✓ **Lists** são coleções sequenciais, similares à **vectors**;
- ✓ Porém, itens em **lists** são adicionados na **frente** (no **início** da lista);
- ✓ **Lists** são criadas pela literal string **()**, mas para diferenciar listas que representam código das listas que representam dados, usamos **' '**.



Lists

```
Clojure
user=>
user=>
user=> ( 1 2 3 )
Execution error (ClassCastException) at user/eval81 (REPL:1).
java.lang.Long cannot be cast to clojure.lang.IFn
user=>
user=>
user=>
user=> '(1 2 3)
(1 2 3)
user=>
user=>
user=> (+ 1 2 3)
6
user=>
user=> ' (+ 1 2 3)
(+ 1 2 3)
user=>
user=>
user=>
```



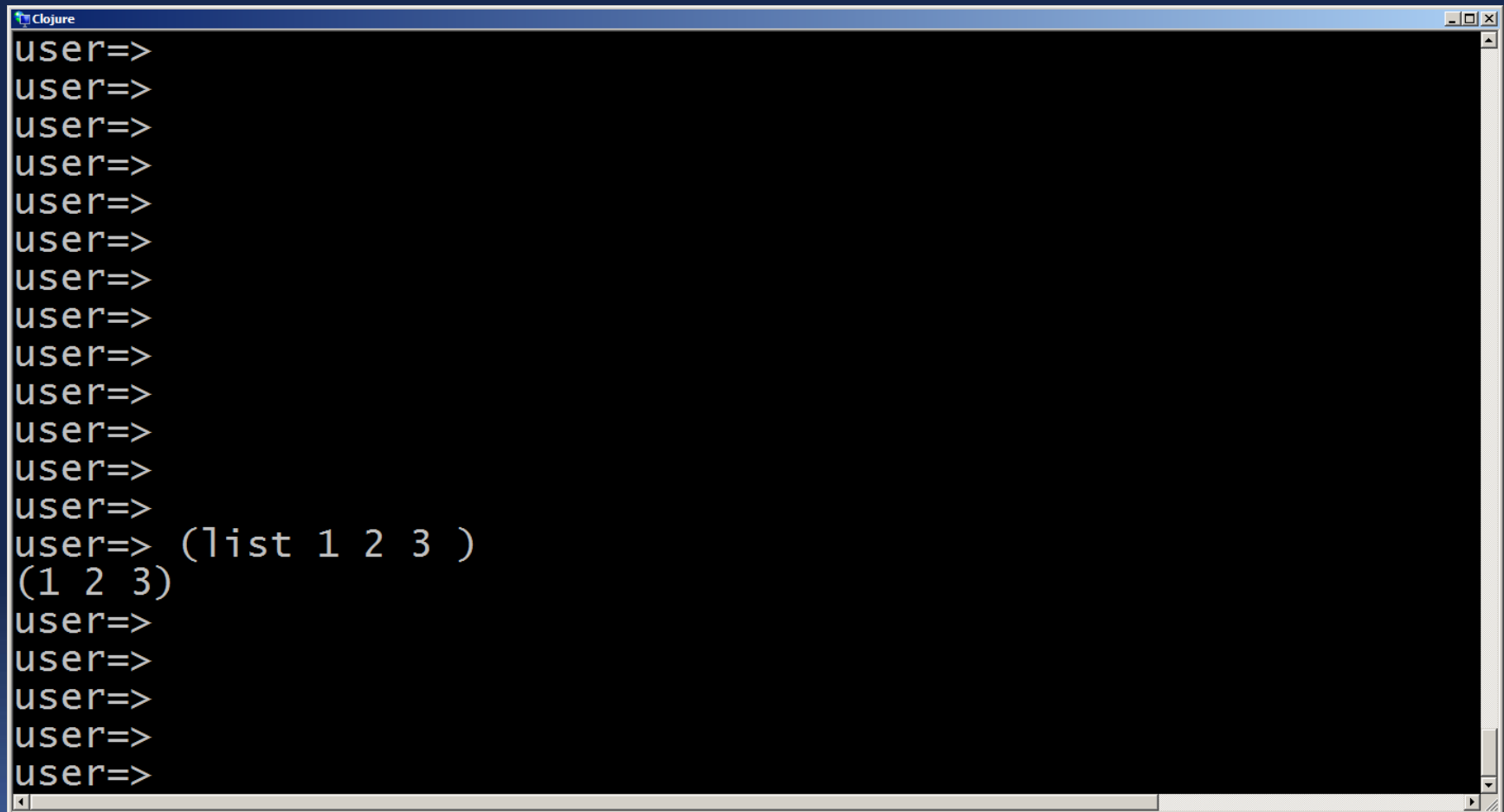
Lists

- ✓ Também podem ser criadas pela função **list**:



Lists

✓ Também podem ser criadas pela função **list**:



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (list 1 2 3 )
(1 2 3)
user=>
user=>
user=>
user=>
user=>
```



Lists

- ✓ A função **first** recupera o primeiro elemento da list;
- ✓ A função **rest** retorna a lista sem o primeiro elemento;
- ✓ A função **nth** recupera um elemento genérico da lista



Lists

- ✓ A função **first** recupera o primeiro elemento da list;
- ✓ A função **rest** retorna a lista sem o primeiro elemento;
- ✓ A função **nth** recupera um elemento genérico da lista

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (first '(1 2 3 ) )
1
user=>
user=>
user=> (rest '(1 2 3 ) )
(2 3)
user=>
user=>
user=> (nth '(:a :b :c ) 2 )
:c
user=>
user=>
user=>
```



Exercícios com lists



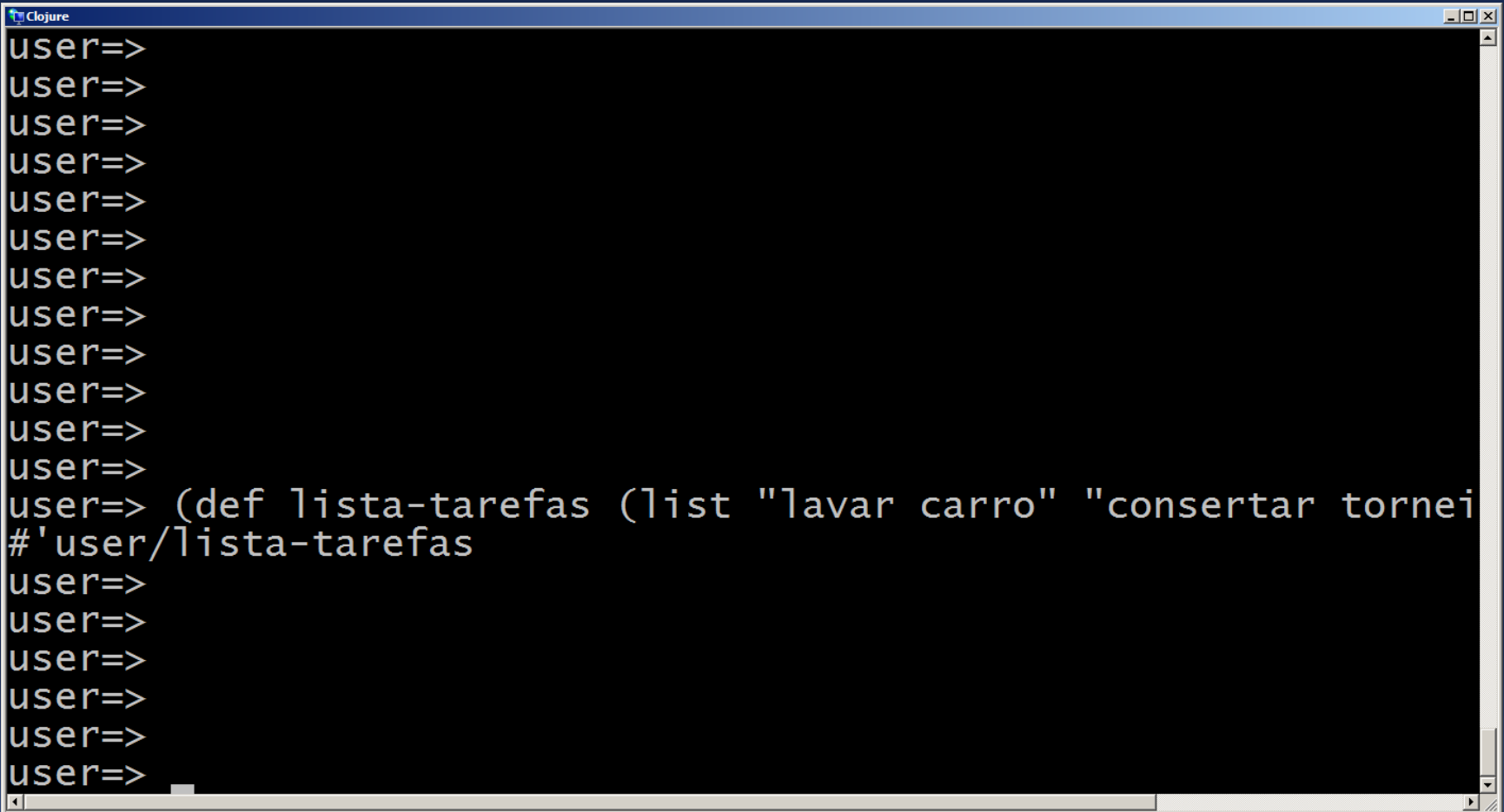
1. Inicie **REPL**.

Crie uma lista chamada lista-tarefas com ações que necessitam ser feitas:

Tarefas “lavar carro” “consertar torneira”



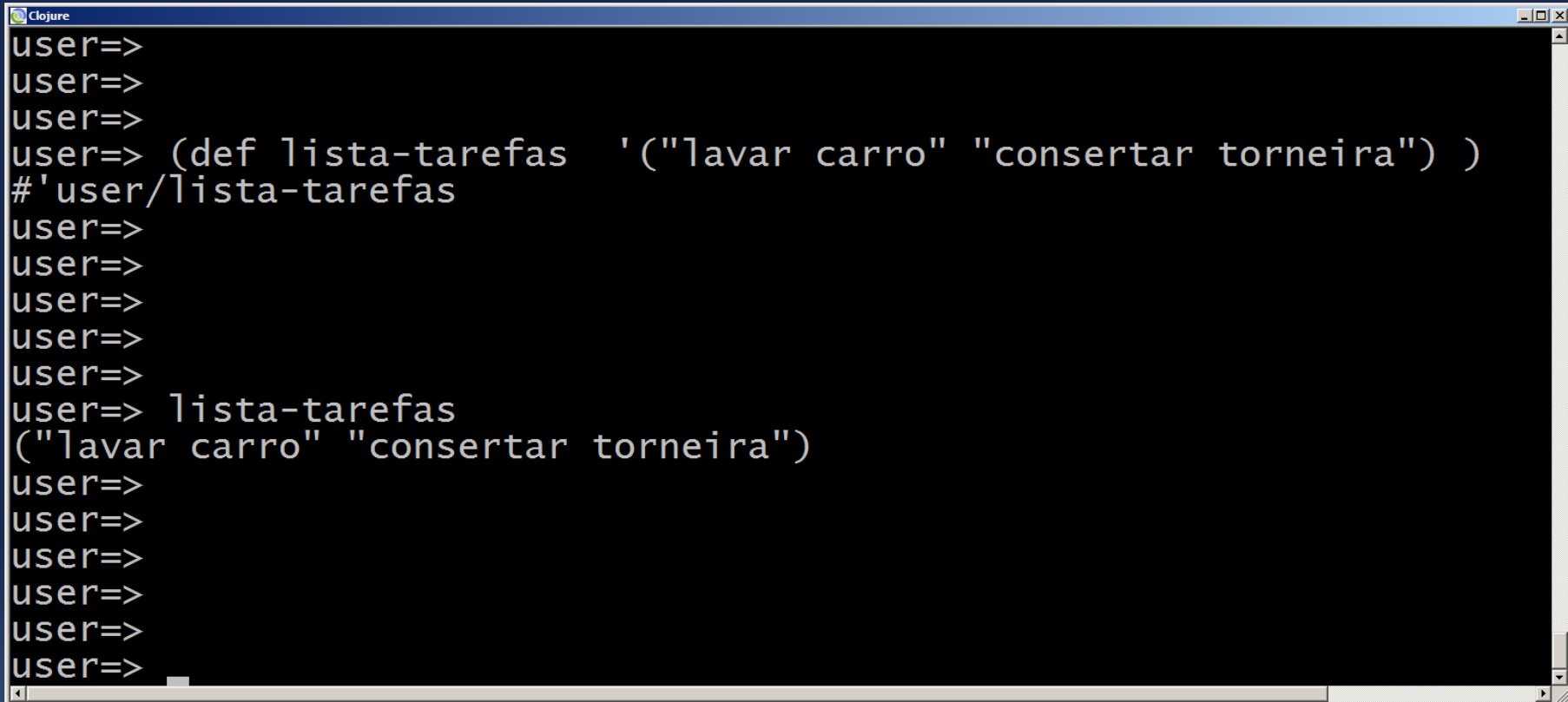
1. Inicie **REPL**. Crie uma lista chamada lista-tarefas com ações que necessitam ser feitas:



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (def lista-tarefas (list "lavar carro" "consertar tornei
#'user/lista-tarefas
user=>
user=>
user=>
user=>
user=>
user=>
```



1. Inicie **REPL**. Crie uma lista chamada lista-tarefas com ações que necessitam ser feitas:



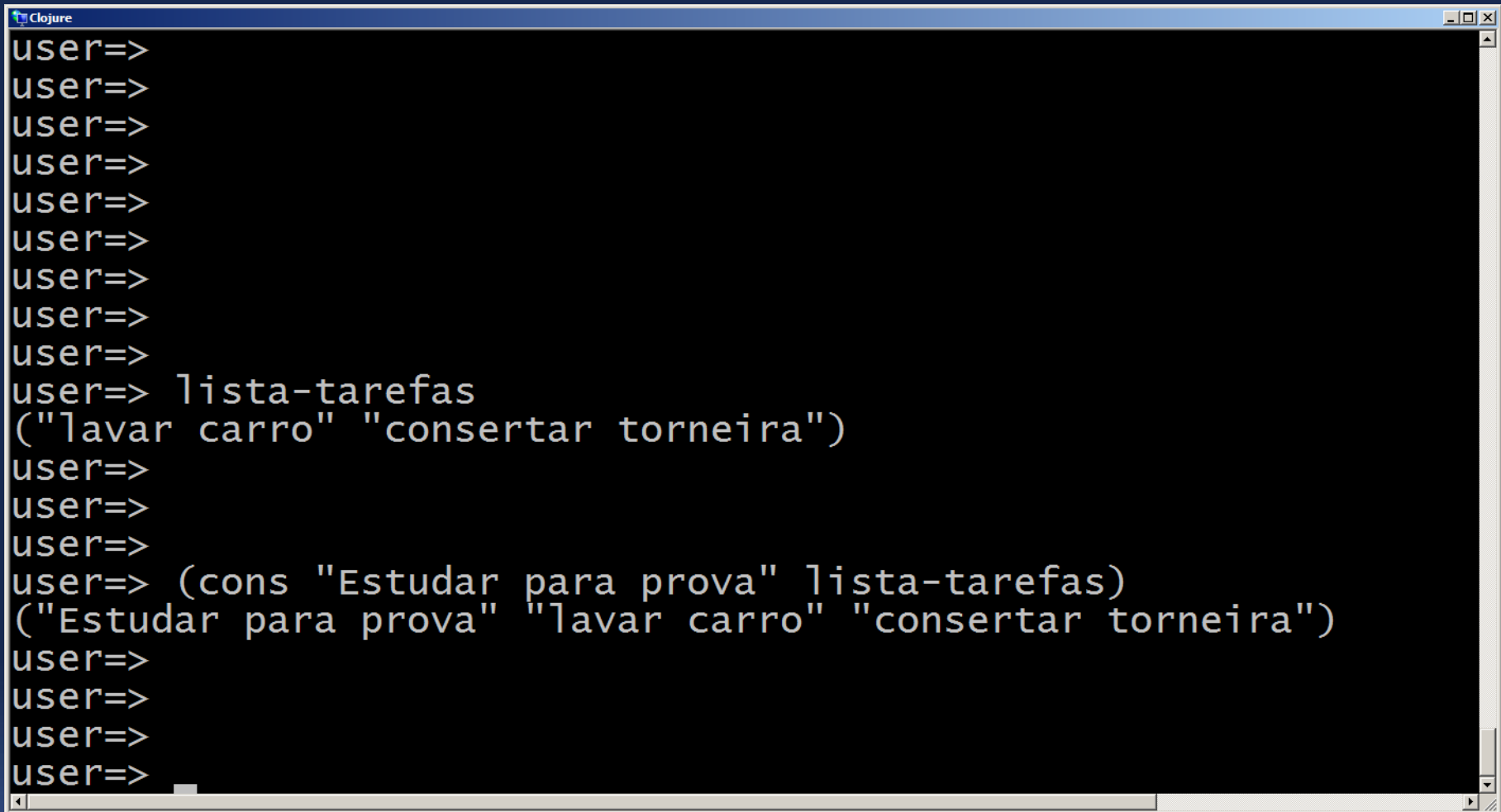
```
Clojure
user=>
user=>
user=>
user=> (def lista-tarefas '("lavar carro" "consertar torneira") )
#'user/lista-tarefas
user=>
user=>
user=>
user=>
user=> lista-tarefas
("lavar carro" "consertar torneira")
user=>
user=>
user=>
user=>
user=>
```



2. Pode-se adicionar itens à lista, por meio da função **cons**:



2. Pode-se adicionar itens à lista, por meio da função **cons**:

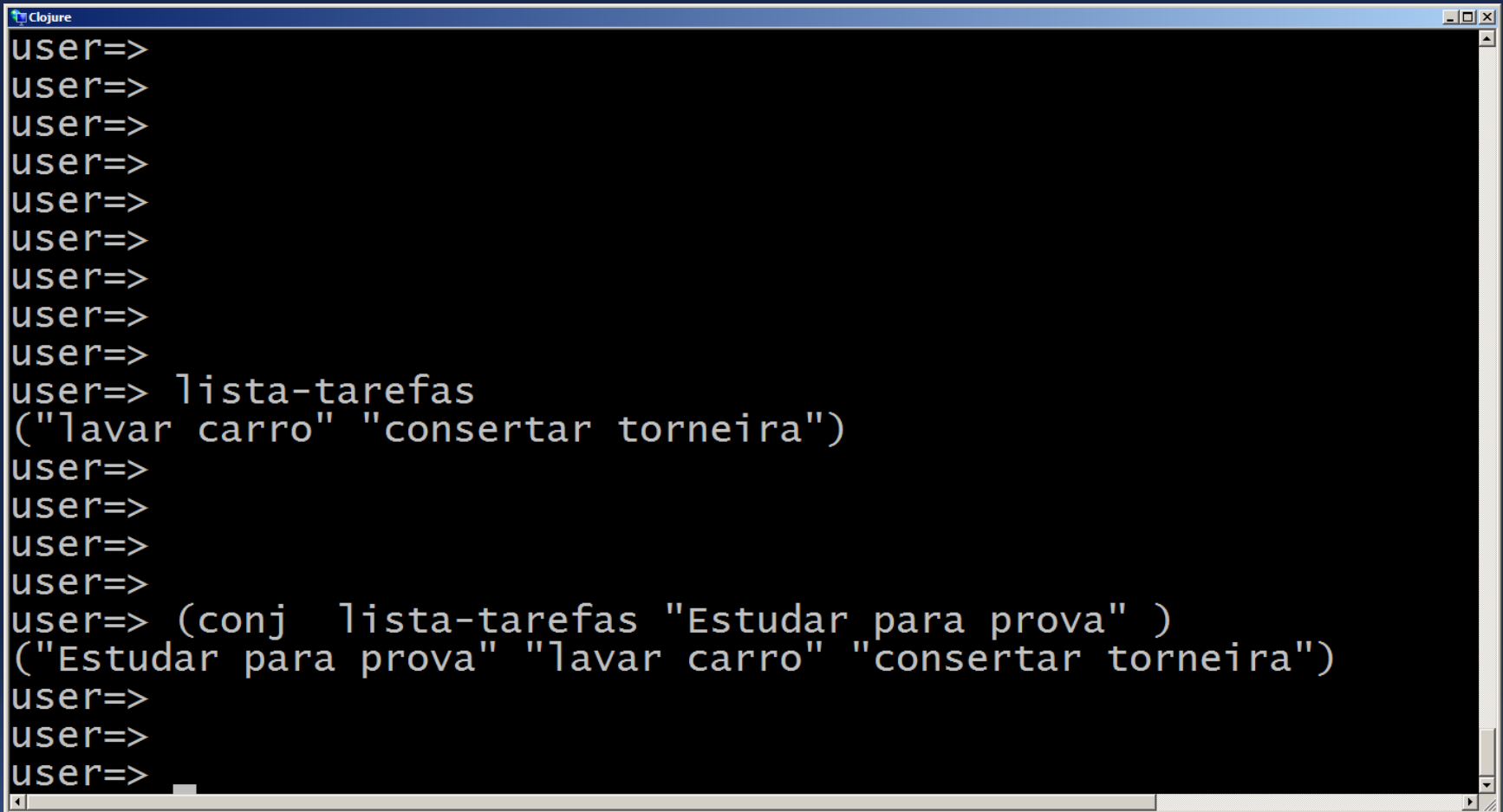


```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> lista-tarefas
("lavar carro" "consertar torneira")
user=>
user=>
user=> (cons "Estudar para prova" lista-tarefas)
("Estudar para prova" "lavar carro" "consertar torneira")
user=>
user=>
user=>
user=>
```

3. Similarmente, pode-se também usar a função **conj**, pois uma lista é uma collection.



3. Similarmente, pode-se também usar a função **conj**, pois uma lista é uma collection.



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> lista-tarefas
("lavar carro" "consertar torneira")
user=>
user=>
user=>
user=> (conj lista-tarefas "Estudar para prova" )
("Estudar para prova" "lavar carro" "consertar torneira")
user=>
user=>
user=>
```



3. A função **conj** permite adicionar mais de um elemento à list.

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> lista-tarefas
("lavar carro" "consertar torneira")
user=>
user=>
user=>
user=>
user=> (conj lista-tarefas "Estudar para prova" "descansar" )
("descansar" "Estudar para prova" "lavar carro" "consertar torneira")
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```



4. A função **first** recupera o primeiro elemento da lista.

A função **rest** recuperar a lista sem o primeiro elemento.

A função **nth** recupera um elemento genérico da lista.



4. A função **first** recupera o primeiro elemento da lista. A função **rest** recuperar a lista sem o primeiro elemento. A função **nth** recupera um elemento genérico da lista.

```
user=>
user=>
user=>
user=>
user=>
user=> (def lista-times (list "SP" "Palmeiras" "Santos" "Flamengo" ) )
#'user/lista-times
user=>
user=>
user=> (first lista-times)
"SP"
user=>
user=> (rest lista-times)
("Palmeiras" "Santos" "Flamengo")
user=>
user=>
user=> (nth lista-times 2)
"Santos"
user=>
user=>
user=>
user=>
user=>
```



Observações sobre Coleções



Observações

- ✓ Uma sequência corresponde à uma coleção de elementos em uma ordem particular, no qual cada item de dado é seguido pelo próximo.
- ✓ **Maps**, **sets**, **vectors** e **lists** são todos coleções de dados;
- ✓ Mas, somente **vectors** e **lists** são **sequências**, embora possamos facilmente obter uma sequência a partir de um **map** ou de um **set**;



Função count

- ✓ Usada para se obter o número de elementos de uma collection.



Função count

- ✓ Usada para se obter o número de elementos de uma collection.

```
Clojure
user=>
user=>
user=> (def lista-times (list "SP" "Palmeiras" "Santos" "Flamengo" ) )
#'user/lista-times
user=>
user=>
user=>
user=>
user=>
user=> (count lista-times)
4
user=>
user=>
user=> (count #{ } )
0
user=>
user=> (count '( ) )
0
user=>
user=>
user=> (count [ ] )
0
user=>
user=> (count { } )
0
user=>
user=>
```



Função empty?

- ✓ Usada para checar se uma collection é vazia.



Função empty?

- ✓ Usada para checar se uma collection é vazia.

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> (def lista-times (list "SP" "Palmeiras" "Santos" "Flamengo" ) )
#'user/lista-times
user=>
user=> (empty? lista-times)
false
user=>
user=> (empty? { } )
true
user=>
user=> (empty? [ ] )
true
user=>
user=> (empty? { } )
true
user=>
user=> (empty? #{ } )
true
user=>
user=>
user=>
user=>
```



Função seq

- ✓ Um map **não** é sequencial pois **não** há ordenação lógica em seus elementos.
- ✓ Entretanto, pode-se converter um map para uma sequência por meio da função **seq**.



Função seq

- ✓ Um map não é sequencial pois não há ordenação lógica em seus elementos. Entretanto, pode-se converter um map para uma sequência por meio da função **seq**:

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (def language {:name "Clojure" :creator "Rick Hickey" :platforms ["Java"
"JavaScript" ".NET"]})
#'user/language
user=>
user=>
user=>
user=> language
{:name "Clojure", :creator "Rick Hickey", :platforms ["Java" "JavaScript" ".NET"
]}
user=>
user=> (seq language)
([:name "Clojure"] [:creator "Rick Hickey"] [:platforms ["Java" "JavaScript" ".NET"
]])
user=>
user=>
user=> (nth (seq language) 2)
[:platforms ["Java" "JavaScript" ".NET"]]
user=>
user=>
user=>
user=>
user=>
```



Função into

- ✓ Essa função possibilita que se coloque elementos de uma coleção em outra.



Função into

- ✓ Essa função possibilita que se coloque elementos de uma coleção em outra.
- ✓ No exemplo abaixo cada elemento de `#{ 5 6 7 8 }` é adicionado ao vector `[1 2 3 4]`
- ✓ No entanto, o vector resultante **não** é ordenado pois Hash Sets **não** são ordenados.

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (into [1 2 3 4] #{ 5 6 7 8 })
[1 2 3 4 7 6 5 8]
user=>
user=>
user=>
user=>
```





Função into

- ✓ No exemplo abaixo o vector [5 6 7 8] foi adicionado ao set #{ 1 2 3 4 }. Novamente, Hash Sets não mantém ordem de inserção e assim, o set obtido é simplesmente uma coleção lógica de valores únicos.

```
user=>
user=>
user=> (into [1 2 3 4 ]      #{ 5 6 7 8 } )
[1 2 3 4 7 6 5 8]
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (into #{ 5 6 7 8 } [1 2 3 4 5 ] )
#{7 1 4 6 3 2 5 8}
user=>
user=>
user=>
user=>
```





Função into

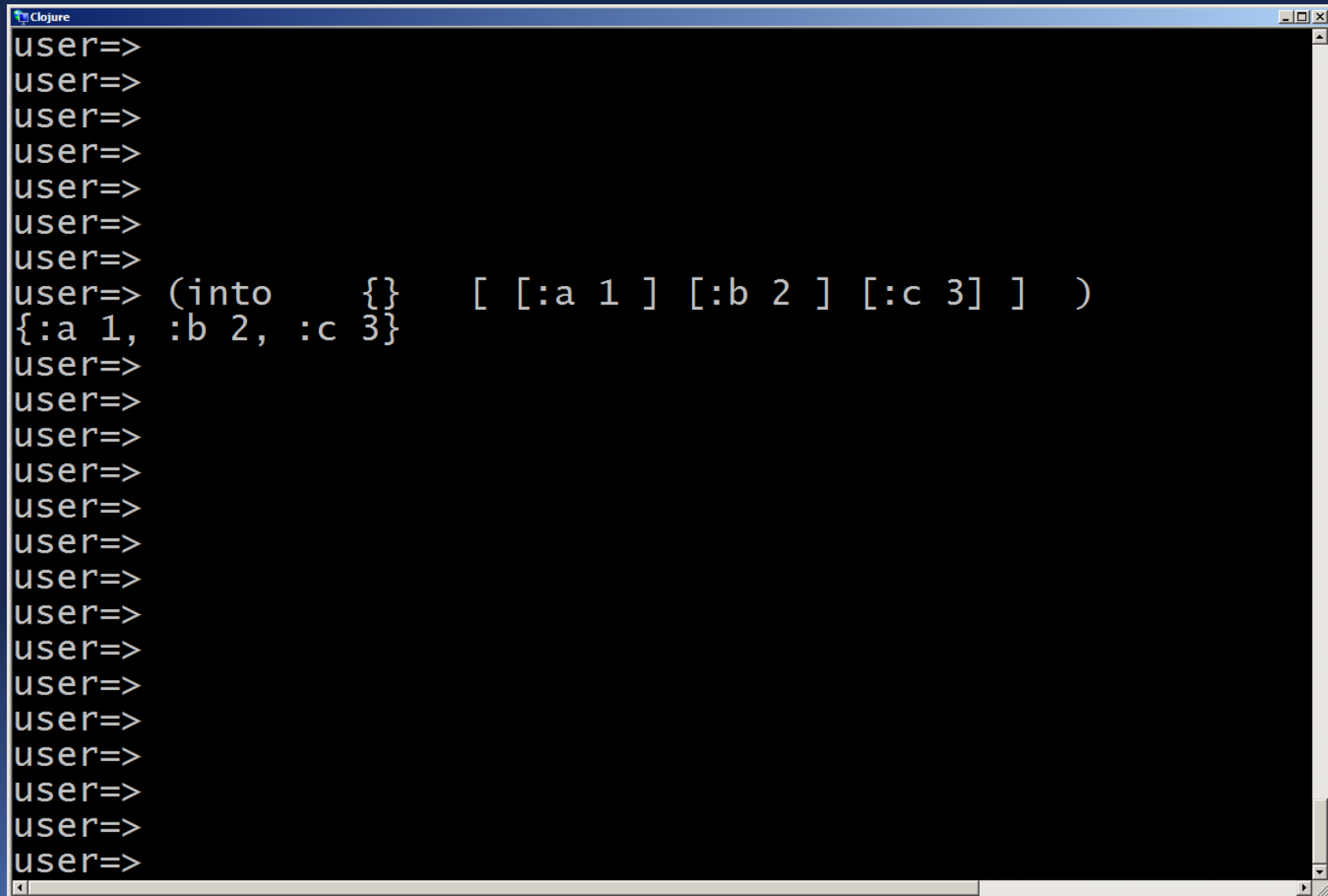
- ✓ O exemplo abaixo apresenta um set obtido de um vector (deduplicação).

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (into #{ } [ 1 2 2 2 2 2 2 3 3 3 3 3 3])
#{1 3 2}
user=>
user=>
user=>
user=>
user=>
user=>
```



Função into

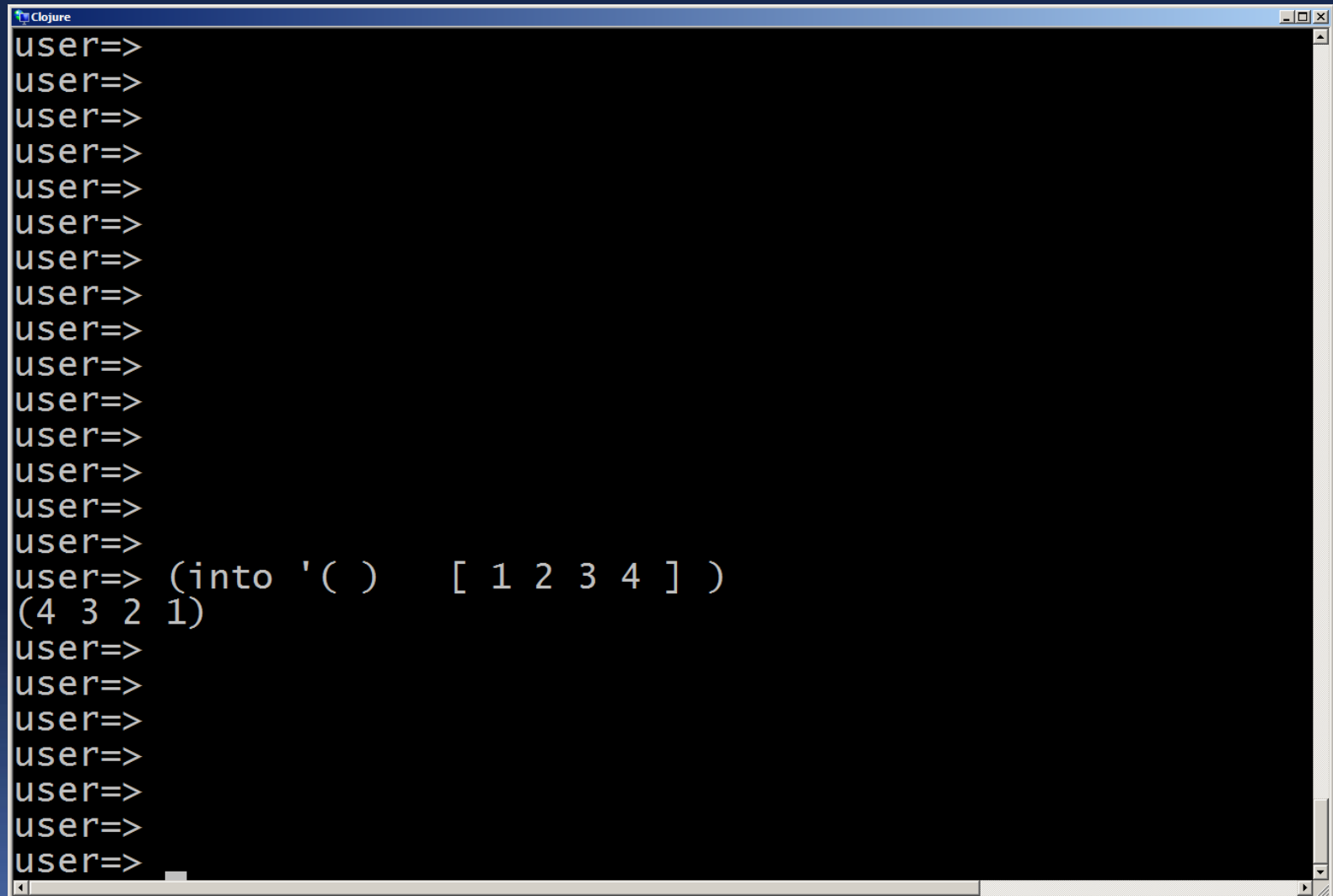
- ✓ Para inserir itens em um map, necessita-se passar uma coleção de tuplas representando pares key-values.



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (into {} [[:a 1] [:b 2] [:c 3]])
{:a 1, :b 2, :c 3}
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

Função into

- ✓ Elementos são adicionados à uma lista a partir do início (front)



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (into '( ) [ 1 2 3 4 ] )
(4 3 2 1)
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```



Função concat

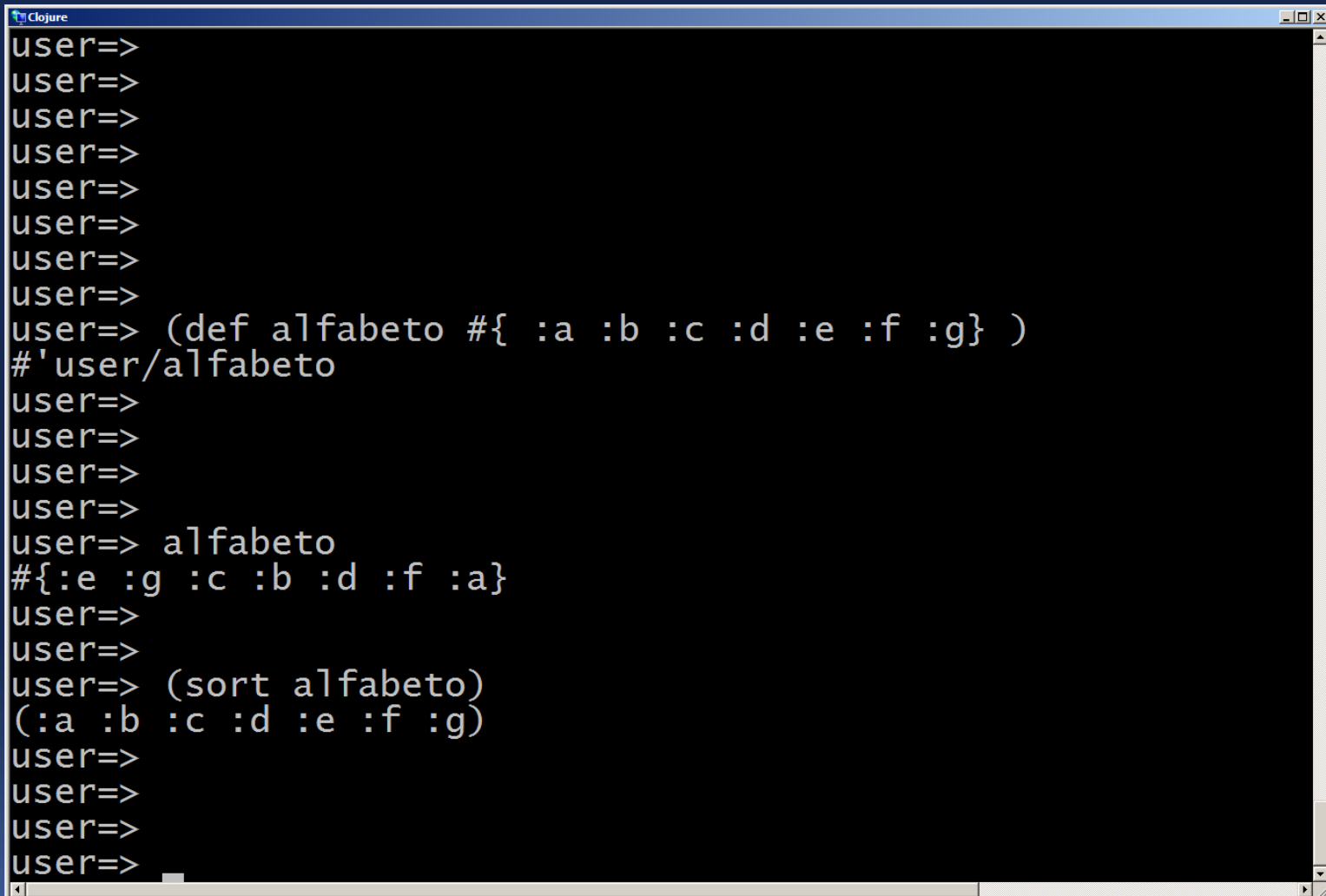
- ✓ Usada para a concatenação de coleções.

```
Clojure
user=>
user=>
user=> (concat '(1 2) '(3 4) )
(1 2 3 4)
user=>
user=>
user=> (concat '(1 2) '(5 6) )
(1 2 5 6)
user=>
user=>
user=>
user=>
user=> (concat [1 2] [3 4] )
(1 2 3 4)
user=>
user=>
user=> (concat #{ 1 2 3} ["Hello"] )
(1 3 2 "Hello")
user=>
user=>
user=> (concat {:a 1} ["Hello"] )
([:a 1] "Hello")
user=>
user=>
```



Função sort

- ✓ Usada para ordenar os elementos de uma coleção.



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (def alfabeto #{ :a :b :c :d :e :f :g } )
#'user/alfabeto
user=>
user=>
user=>
user=>
user=> alfabeto
#{:e :g :c :b :d :f :a}
user=>
user=>
user=> (sort alfabeto)
(:a :b :c :d :e :f :g)
user=>
user=>
user=>
```

