(Clojure)

Revelado

Clojure Revelado

Felipe Gonçalves Marques

Revision 1
2016-07-25

Table of Contents

Chapter 1. Disclaimer

This is a non-professional translation of ClojureScript Unraveled[1] to both Portuguese and *Clojure*, done in a spike of energy and motivation over a week, which means that we may not be very careful with the translation. However, we tried not to change the content or the opinions of the authors of the *ClojureScript Unraveled* book. I really appreciate their book and the fact that it is written under Creative Commons.

[1] https://funcool.github.io/clojurescript-unraveled/

Chapter 2. Sobre esse livro

Esse livro é uma dupla tradução do livro ClojureScript Unraveled[1]. Dupla tradução pois ele foi traduzido do inglês para o português e de *ClojureScript* para *Clojure*. O objetivo é obter uma guia extenso das funcionalidades do *Clojure* em português, democratizando o acesso a esse conhecimento nos países lusofonos, principalmente no Brasil.

Ele não é um livro introdutório em programação, pois ele assume que o leitor tenha experiência em ao menos uma linguagem de programação. Contudo, ele não assume nenhuma experiência prévia com *Clojure* ou programação funcional. Por ser uma tradução, muitos dos links ainda referenciam textos em inglês, mas onde foi possível, substituí por equivalentes em português.

Por ser uma linguagem não tão difundida como Python, Ruby ou Java, não existe tantos recursos de *Clojure* em português, optamos por traduzir esse livro, pois ele cobre a maior parte das funcionalidades. Talvez ele não seja tão concreto por não fornecer exemplos reais como mini-projetos, ou tão profundo, entrando nas discussões de como a língua foi construída. Porém, acho que ele é um bom começo para fornecer material gratuito sobre *Clojure*.

Por enquanto, somente o capítulo "A Linguagem (o básico)" foi traduzido. Em breve, pretendo traduzir os outros capítulos do livro que são: "Ferramentas" e "A Linguagem (avançado)".

[1] https://funcool.github.io/clojurescript-unraveled/

Chapter 3. Introdução

```
_Clojure_ é uma linguagem funcional e dinâmicamente tipada, e direcionada
 por dados
(data-driven). Ela foca em ser pratica, por isso foi implementada para a
 _Java Virtual Machine(JVM)_.
Clojure é um dialeto de _LISP_ e por tanto oferece todo o poder de macros
 e da idea de código é dados.
```

Antes de começarmos, vamos resumir alguma das ideias mais importantes que ClojureScript traz. Não se preocupe se não entendê-las de primeira, elas vão ficar claras ao longo desse livro.

- Clojure favore o paradigma de programção funcional através das suas decisões de design e idiomas. Apesar de ser assertiva em relação ao paradigma funcional, ela é uma linguagem pragmática: ao invés de buscar uma pureza em relação a programação funcional como Haskell, por exemplo, ela foca em ser prática.
- Encoraja programação utilizando estruturas imutáveis, oferencendo implementações de estruturas de dados altamente performáticas.
- Ela faz uma distinção clara entre identidade e o seu estado, com ferramentas explícitas para o gerenciamento de mudanças vinculadas a uma identidade, que é mantida como uma série de valores imutáveis ao longo do tempo.
- Ela possui um polimorfismo baseado em tipos e valores, que resolve o problema de expressar o domínio de forma elegante.
- Ela é um dialeto de Lisp, onde programas são escritos utilizando as mesmas estruturas de dados presentes na língua, uma propriedade conhecida como *homoiconicity* que torna a meta-programação (programas que escrevem programas) mais simples e palpável.

Essas ideias influenciam a maneira como desenhamos e implementamos nossos programas, mesmo quando não estamos usando Clojure. Programação funciona, desacoplamento dos dados das operação que os transformam e ferramentas explícitas para gerenciamento de mudanças trazem uma grande simplicidade para os sistemas que escrevemos.

> Podemos fazer os mesmos softwares que fazemos hoje com ferramentas drasticamente mais simples - linguagems, ferramentas, técnicas e abordagens muito mais simples.
>
> — Rich Hickey

```
É de longe uma das linguages que mais gostamos de utilizar e nos sintimos
 extremamente produtivos
quando a usamos. Espero que você também se sinta assim quando estiver
 trabalhando com Clojure :).
```

Chapter 4. A Linguagem (o básico)

Esse capítulo é uma pequena introdução ao Clojure sem assumir nenhum conhecimento prévio da linguagem, ele provê um tour rápido sobre as principais coisas que você precisa saber sobre Clojure para entender o resto do livro.

Você pode rodar os trechos de código na REPL online interativa em: https://repl.it/languages/clojure

/* todo: adicionar tutorial de como rodar clojure */

4.1. Primeiros passos com a sintaxe Lisp

Inventada por John Mccarthy em 1958, Lisp é uma das linguagens de progrmação mais antigas que ainda são utilizadas. Ela possui várias outras derivações chamadas de dialeto, sendo Clojure uma delas. É uma linguagem de programação escrita utilizando suas próprias estrutura de dados - originalmente uma lista rodeada de parênteses - mas Clojure evoluiu essa sintaxe para possuir mais estruturas de dados e ser mais agradável de escrever.

Uma lista com uma função na primeira posição é utilizada para chamar uma função em Clojure. No exemplo abaixo, nos aplicamos a função de adição a três argumentos. Observe que diferente de outras linguagens, `+` não é um operado, mas uma função. Lisp não possui operadores; somente funções.

```
(+ 1 2 3)
;; => 6
```

No exemplo acima, estamos aplicando a função de adição `+` aos argumentos `1`, `2` e `3`. Clojure permite vários caracteres não muito comuns como `?` e `-` em nome de símbolos, o que a torna a leitura mais fácil:

```
(zero? 0)
;; => true
```

Para diferencial chamadas de funções de uma lista de itens, podemos usar uma aspa o que evite que a lista seja calculada/avaliada, como no primeiro exemplo.

Uma lista com aspa no começo (*quoted list*) será tratada como dado ao invés de uma chamada de função:

```
'(+ 1 2 3)
;; => (+ 1 2 3)
```

Clojure não usa somente listas para sua sintaxe. Cobriremos isso com mais detalhes mais tarde, mas aqui está um exemplo de utilização de um vetor (rodeado por cochetes) para definir uma relação entre nome e valor (*bidinng*) local:

```
(let [x 1
      y 2
      z 3]
  (+ x y z))
;; => 6
```

Essa é a sintaxe mínima que voc6e precisa saber para utilizar não só Clojure, mas qualquer Lisp. Sendo escrita nas suas próprias estruturas de dados (aquela coisa de *homoiconicity*) é uma propriedade muito poderosa, pois a sintaxe é uniforme e simples, além disso, a geração de código através de macros é mais simples que outras linguagens, nos dando poder de ampliar a linguagem para se adequar às nossas necessidades.

4.2. Os tipos básicos

Clojure possui uma conjunto de tipos primitivos como maioria das linguagens. Ele prove escalares que serão muito familiar para você como números, strings e números de ponto flutuantes (*floats*). Além desses, ele provê alguns outros tipos não muito comuns como símbolos, chaves, expressões regulares (*regex*), *vars* e *atoms*.

Clojure utiliza a linguagem hospedeira (*Java*), e quando possível, utiliza os tipos fornecidos por ela. Por exemplo, números e strings são usados como providos pelo linguagem e se comportam do mesmo jeito que se comportam em Java.

4.2.1. Números

Em *Clojure*, números incluem tanto inteiros quanto floating points (*float*). Tenha em mente, que como uma linguagem hospedada na *JVM*, os tipos numéricos em *Clojure* são os tipos nativos do Java como Integer e Long por baixo dos panos, mas também provendo alguns outros tipos como BigInt.

Como em outras linguagens, números em *Clojure* são representados da seguinte maneira:

```
23
+23
-100
1.7
-2
33e8
12e-14
3.2e-4
```

4.2.2. Chaves (*Keywords*)

Chaves em *Clojure* são objetos que sempre são avalidados a eles mesmos. Eles são usualmente usado em estruturas de dados do tipo mapa para representar de maneira eficiente as chaves do mapa (ou dicionário).

```
:foobar
:2
:?
```

Como voc6e pode ver, as chaves são todas prefixadas com `:`, mas esse caracter é apenas parte da sintaxe literal e não parte do nome do objeto.

Você também pode criar uma chave chamando a função **`keyword`**. Não se preocupe se você não entende ou nada está claro no próximo exemplo, vamos discutir funções em uma seção adiante.

```
(keyword "foo")
```

```
;; => :foo
```

Chaves qualificadas (ou com *Namespace*)

Quando prefixamos uma chave com `::`, essa chave passa a ser prefixada com o *namespace* atual. Note que chaves qualificadas afetam comparações de igualdade.

```
---
::foo
;; => :user/foo
```

(= ::foo :foo) ;; $\Rightarrow$ false ---

Outra alternativa é incluir o *namespace* na sintaxe literal da chava, isso é útil para criamos chaves qualificadas para outro *namespace*:

```
---
:clojure.unraveled/foo
;; => :clojure.unraveled/foo
---
```

A função **`keyword`** também funciona como uma função que recebe 2 argumentos (*arity-2*) onde você especifica o *namespace* como primeiro argumento:

```
---
(keyword "clojure.unraveled" "foo")
;; => :clojure.unraveled/foo
---
```

4.2.3. Símbolos

Símbolos em *Clojure* são muito parecidos com chaves (que agora você conhece0. Mas ao invés de serem avaliados (*evaluated*) a eles mesmos, os símbolos são avaliados para outra coisa a qual eles referem, que pode ser funções, variáveis, etc.

Símbolos começam sembre com um caractere não numério e pode conter caracteres alfa-numéricos assim como *, +, !, -, -, ', e ? como:

```
sample-symbol
othersymbol
f1
my-special-swap!
```

Não se preocupe se você não entender isso de imediato, símbolos são usados praticamente em todos os nossos outros exemplosl, o que te dá a oportunidade de aprender mais a medida que avançamos.

4.2.4. Strings

Strings em *Clojure* não são tão diferente de outras linguagens, então você já deve saber suficiente sobre elas. Um único ponto interessante de mencionar é que elas são imutáveis.

```
"An example of a string"
```

Um aspecto peculiar de strings em *Clojure* devido a sintaxe de Lisp é que seja strings de uma linha ou de várias linhas, elas possuem a mesma sintaxe.

```
"This is a multiline
      string in ClojureScript."
```

4.2.5. Caracteres

Clojure também permite você escrever um único caractere usando a seguinte sintaxe literal:

```
\a        ; "a" minúsculo
\newline  ; Caractere que indica uma nova linha
```

4.2.6. Coleções

O próximo grande passo em explicar a linguagem é explicar suas coleções e as abstrações de coleções. *Clojure* não é uma exceção a essa regra.

Clojure vem com vários tipos de coleções. A principal diferenção do *Clojure* para outras linguagens é que suas coleções são persistentes e imutáveis.

Antes de entrarmos nesses conceitos (provavelmente) desconhecidos, vamos dar uma visão geral das coleções existentes em *Clojure*.

Listas

Essa é uma coleção clássica em qualquer linguagem derivada do Lisp. Listas são o tipo de coleção mais simples em *Clojure*. Elas podem conter itens de qualquer tipo, incluindo outras coleções.

Listas em *Clojure* são representadas por items envolvidos por parênteses:

```
'(1 2 3 4 5)
'(:foo :bar 2)
```

Como você pode ver, todos os exemplos de listas são prefixados com o caractere `'`. Isso porque em linguagems derivadas do Lisp, listas são usadas para expressar chamadas de funções (ou de macros). Nesse caso, o primeiro item deveria ser um símbolo que é avaliado a alguma coisa que é chamável (ex: uma função), e o resto dos elementos da lista serão argumentos da função. porém, nos exemplos anteriores, nos não queremos que o primeiro item da lista seja um símbolo, nos queremos que seja apenas uma lista de items.

O exemplo seguinte mostra a diferença entre listas com e sem uma aspa no começo:

```
(inc 1)
;; => 2

'(inc 1)
;; => (inc 1)
```

Como você pode ver, se evaluarmos **`(inc 1)`** sem prefixar com uma aspa `'`, ele será transformado na função **`inc`** (de incrementar) e irá executar essa função com 1 como seu primeiro argumento, retornando o valor de 2.

Você pode também contruir uma lista com a função **`list`**:

```
(list 1 2 3 4 5)
;; => (1 2 3 4 5)

(list :foo :bar 2)
;; => (:foo :bar 2)
```

Listas tem a peculiaridade de ser muito eficiente se você acessar elas sequencialmente ou os primeiros elementos, porém elas não são uma boa opção se você precisa acessar elementos utilizando a posição (index) desses elementos.

Vetores

Como listas, vetores armazenam uma série de valores, mas nesse caso, de forma mais eficiente para acesso através do index desses elementos. Não se preocupe, nas seções seguintes vamos entrar nos detalhes, mas por hora, essa explicação é mais que suficiente.

Vetores usam cochetes como sintaxe literal, vamos ver alguns exemplos:

```
[:foo :bar]
[3 4 5 nil]
```

Como listas, vetores podem contem objetos de qualquer tipo, como mostrado no exemplo anterior.

Você pode também explicitamente criar um vetor com a função **`vector`**, mas esse não é o jeito mais comum de fazê-lo em *Clojure*.

```
(vector 1 2 3)
;; => [1 2 3]

(vector "blah" 3.5 nil)
;; => ["blah" 3.5 nil]
```

Mapas (ou dicionários)

Mapas são coleções de abstrações que permite você armazenar pares de chave e valor. Em outras linguages, esse tipo de estrutura são comumente conhecidas

como has-map ou dicionários. Mapas são literais em *Clojure* e são escritos como pares entre chaves.

```
{:foo "bar", :baz 2}
{:alphabet [:a :b :c]}
```

NOTA: Podemos usar vírgulas para separar pares, mas elas são opcionais. No geral, a formatação dos arquivos já favorece a leitura. Em *Clojure*, vírgulas são como espaços.

Como vetores, cada item em um mapa literal é evaluado antes que seu resultado seja armazenado no map, mas a ordem de resolução não é garantida.

Conjuntos

E finalmente, **sets** (ou conjuntos).

Conjuntos armazenam zero ou mais itens únicos de forma não ordenada. Como mapas, eles possuem chaves como sintaxe litera, com a diferença de serem prefixados com `#`. Você também pode usar a função `set` para converter uma coleção em um set:

```
#{1 2 3 :foo :bar}
;; => #{1 :bar 3 :foo 2}
(set [1 2 1 3 1 4 1 5])
;; => #{1 2 3 4 5}
```

Nas seções seguintes, vamos explorar a fundo conjuntos e outras coleções que vimos nessa seção.

4.3. Vars

Clojure é uma linguagem funcional que foca principalmente em imutabilidade. Por cause disso, ela não tem o conceito de variáveis como estamos acostumados em outras linguagens. A analogia mais próxima de vars são as variáveis que definimos na álgebra; quando dizemos `x = 6` na matemática, estamos dizendo que queremos que o símbolo `x` tenha, ou represente, o número seis.

Em *Clojure*, vars são representadas por símbolos e armazenam um único valor junto com alguns meta-dados.

Você pode definir uma var utilizando a forma especial **`def`**:

```
(def x 22)
(def y [1 2 3])
```

Vars são tempre top level em um *namespace*(which we will explain later). Se você usar **`dev`** em uma chamada de função, aquela var será definida no nível do namespace e poderá ser usada em outros lugares (diferente de variáveis locais que algumas linguagens possuem), mas não recomendamos isso - ao invés, você devereria utilizar um bloco **`let`** para definir variáveis dentro de uma função.

4.4. Funções

4.4.1. O primeiro contato

É hora de fazer as coisas acontecerem. *Clojure* possui o que conhecemos como *first class functions*. Funções se comportam como qualquer outro tipo; você pode passá-las como argumentos e retorná-las como valores, sempre respeitando o escopo léxico. *Clojure* também possui algumas funcionalidades devido ao escopo dinâmica, mas vamos ver isso em outra seção.

Se você quer saber mais sobre escopos, esse Artigo da Wikipedia[1] é bem completo e explica os diversos tipos de escopo.

Como *Clojure* é um dialeto do Lisp, ela utiliza a notação prefixada para chamar funções:

```
(inc 1)
;; => 2
```

No exemplo acima, **`inc`** é uma função e é parte da *runtime* do *Clojure*, e **`1`** é o primeiro argumento para a função **`inc`**.

[1] https://pt.wikipedia.org/wiki/Escopo_(computa%C3%A7%C3%A3o))

```
(+ 1 2 3)
;; => 6
```

O símbolo de **+** representa a função **add** (de adição). Ela permite múltiplos parâmetros, enquanto em linguagens que seguem o estilo *ALGOL*, **+** é um operador e permite somente dois parâmetros.

A notação prefixada possui algumas vantagens, que não são óbvias. *Clojure* não faz distinção entre função e operador; tudo é uma função. A vantagem imediata é que a notação prefixada permite um número arbitrário de argumentos por "operador". Isso remove completamente o problema de precedência de operadores.

4.4.2. Definindo suas próprias funções

Você pode definir funções sem nome (anônimas) com a forma especial **fn**. Esse é um tipo de definição de função; no exemplo seguinte, a função recebe dois argumentos e retorna a média deles.

```
(fn [param1 param2]
  (/ (+ param1 param2) 2.0))
```

Você pode definir a função e chamá-la ao mesmo tempo (em uma única expressão):

```
((fn [x] (* x x)) 5)
;; => 25
```

Agora vamos criar algumas funções com nomes. Mas o que uma *função com nome* significa? Em *Clojure* é bem simples, funções são *first-class* e se comportam como qualquer outro valor. Então, nomear uma função é feita simplesmente vinculando a função a um símbolo.

```
(def square (fn [x] (* x x)))

(square 12)
```

```
;; => 144
```

Clojure também oferece uma macro **`defn`** que permite fazer a mesma coisa de uma maneira mais idiomática:

```
(defn square
  "Return the square of a given number."
  [x]
  (* x x))
```

A string que vem entre o nome da função e o vetor de parâmetros é chamada de *docstring* (documentation string); existem programas que geram documentação a partir do código fonte que utilizam essas *docstrings*.

4.4.3. Funções com múltiplas aridades (ou diferente números de argumentos)

Clojure também vem com a habilidade de definir funções com um número arbitrário de argumentos. (O termo *aridade* signifca o número de argumentos que um função aceita.) A sintaxe é praticamente idêntica a sintaxe de definição de uma função comum, com a pequana diferença que ela contém mais de um *body*(ou corpo de expressões).

Vejamos um exemplo, o qual vamos explicar melhor:

```
(defn myinc
  "Self defined version of parameterized `inc`."
  ([x] (myinc x 1))
  ([x increment]
   (+ x increment)))
```

Essa linha: **`([x] (myinc x 1))`** diz que se a função for chamada somente com um argumento, chame a função **`myinc`** com aquele argumento e o número **`1`**`como segundo argumento. O outro corpo de expressões `**`([x increment] (+ x increment))`** dis que se houver dois argumentos, retorna-se o resultado da adição deles.

Aqui estão mais alguns exemplos de como usar a função de múltipla aridade definida anteriormente. Observe que se você chama uma função com o número errado de argumentos, o compilador irá produzir uma mensagem de erro.

```
(myinc 1)
;; => 2

(myinc 1 3)
;; => 4

(myinc 1 3 3)
;; Compiler error
```

> Explicar o conceito de "aridade" está fora do escopo desse livro, entretanto você pode ler mais sobre isso nesse artigo da Wikipedia[2].

4.4.4. Funções de aridade variável

Outro jeito de aceitar múltiplo número de argumentos é definir uma função aridade variável. Essas funções podem aceitar um número arbitrário de argumentos:

```
(defn my-variadic-set
  [& params]
  (set params))

(my-variadic-set 1 2 3 1)
;; => #{1 2 3}

(my-variadic-set 1 2)
;; => #{1 2}
```

A função acima aceita de 0 a quantos argumentos quisermos, podendo receber 1, 2, 3 ou até mais. O jeito de definir uma função com aridade variável é utilizando o símbolo `&` como prefixo no seu vetor de argumentos.

[2] https://pt.wikipedia.org/wiki/Aridade

4.4.5. Sintaxe mais simmples para funções anônimas

Clojure provê uma sintaxe mais compacta para definir funções anônimas usando a macro de leitura **`#()`** (usualmente consistindo de somente uma linha). Macros de leitura são expressões especiais que serão transformadas para algo compatível com a língua em tempo de compilação; nesse caso, para uma expressão que usa a forma especial **`fn`**.

```
(def average #(/ (+ %1 %2) 2))

(average 3 4)
;; => 3.5
```

A definição precedente é uma jeito mais curto de escrever:

```
(def average-longer (fn [a b] (/ (+ a b) 2)))

(average-longer 7 8)
;; => 7.5
```

`%1`, **`%2`**... **`%N`** são marcadores simples da posição dos argumentos no vetor de argumentos que são implicitamente declarados quando a macro de leitura vai interpretar e converter essa forma em uma expressão **`fn`**.

Se a função aceitar somente um argumento, você pode omitir o número depois do `%`, por exemplo, uma função que eleva um número ao quadrado pode ser escrita tanto assim **`#(* %1 %1)`** quanto assim **`#(* % %)`**.

Além disso, a sintaxe também suporta a forma de aridade variável com o símbolo **`%&`**:

```
(def my-variadic-set #(set %&))

(my-variadic-set 1 2 2)
;; => #{1 2}
```

4.5. Lógica de controle

Clojure tem uma abordagem diferente a lógica de controle (if, else, for) do que outras linguagens como JavaScript, C, etc.

4.5.1. Executando coisas diferentes com `if`

Vamos começar com o simples **`if`**. Em *Clojure*, o **`if`** é uma expressão e não uma declaração, e ele recebe três parâmetros: o primeiro é a expressão de condição (que define qual expressão vai ser executada), o segundo é a expresão que será executada se a expressão de condição for evaluada para algo "verdadeiro", e a terceira expressão vai ser executada caso contrário.

```
(defn discount
  "Você pode obter 5% de disconto se pedir 100 ou mais itens"
  [quantity]
  (if (>= quantity 100)
    0.05
    0))

(discount 30)
;; => 0

(discount 130)
;; => 0.05
```

O bloco de execução **`do`** pode ser usado para termos múltiplas expressoão em um **`if`**. **`do`** is explained in the next section.

4.5.2. Executando coisas diferentes com `cond`

Algumas vezes, a expressão **`if`** pode ser um pouco limitante porque ela não possui o bloco "else if" para adicionarmos mais de uma condição. A macro **`cond`** resolve isso.

Com a expressão **`cond`**, podemos definir múltiplas expressões de condição:

```
(defn mypos?
  [x]
```

```
  (cond
    (> x 0) "positive"
    (< x 0) "negative"
    :else "zero"))

(mypos? 0)
;; => "zero"

(mypos? -2)
;; => "negative"

(mypos? 1)
;; => "positive"
```

Também, **cond** tem outra forma, chamada **condp**, que trabalha de forma muito similar com que o **cond** trabalha, porém é um pouco mais simples quando a condição (também chamada de predicado0 é a mesma para todas as condições:

```
(defn translate-lang-code
  [code]
  (condp = (keyword code)
    :es "Spanish"
    :en "English"
    "Unknown"))

(translate-lang-code "en")
;; => "English"

(translate-lang-code "fr")
;; => "Unknown"
```

A linha **condp = (keyword code)** significa que, em cada uma das linhas seguintes, *Clojure* vai aplicar a função **=** ao resultado de **(keyword code)** e o argumento seguinte. Por exemplo para **:es**, será executado o seguinte: **(= :es (keyword code)**.

4.5.3. Case

O operador de controle **case** tem um uso similar ao exemplo do **condp**. A principal diferença é que no **case** o predicado é sempre **=** e seus valores de controle (os

:es e **:en** nos exemplos anteriores) são evaluados em tempo de compilação. Isso resulta em algo mais performático que **cond** e **condp**, com a desvantagem de os valores serem estáticos.

Aqui está o exemplo anterior, reescrito usando **case**:

```
(defn translate-lang-code
  [code]
  (case code
    "es" "Spanish"
    "en" "English"
    "Unknown"))

(translate-lang-code "en")
;; => "English"

(translate-lang-code "fr")
;; => "Unknown"
```

4.6. O que é "Verdade" em *Clojure*?

Esse é um aspecto onde cada língua possui sua própria semântica. Maioria das línguas consideram coleções vazias, o valor 0 e outras coisas como sendo "logicamente falso", isso é levam para o **else** em um **if**. Em *Clojure* diferente de outras línguas, somente duas coisas são consideradas "logicamente falsas": **nil** e **false**. Todo o resto é tratado como "logicamente verdadeiro" (**true**).

Junto com a abilidade de implementar o protocolo "chamável" (*callable*, o **IFN** explicado com mais detalhes mais tarde), estruturas de dados como sets podem ser usados como predicados, sem a necessidade de adicionar funções:

```
(def valid? #{1 2 3})

(valid? 2)
;; => true

(valid? 4)
;; => nil
```

```
(filter valid? (range 1 10))
;; => (1 2 3)
```

Isso funciona porque um set retorna ou o valor do elemento se ele contiver esse elemento ou **`nil`**:

```
(valid? 1)
;; => 1

(valid? 4)
;; => nil
```

4.7. Locals, Blocos, e Loops

4.7.1. Locals

Clojure não possui o conceito de variáveis como linguagens similares ao **`ALGOL`** possuem, mas ele possui o conceito de locals. Locals, como sempre, são imutáveis, e se você tentar mudá-los, o compilador vai lançar uma exceção.

Locals são definidos usando a expressão **`let`**. Essa expressão começa com um vetor de vínculos e por um número arbitrário de expressões (que chamámos de let-body ou corpo do **`let`**). O vetor de vínculos deve contar um número arbitrário de pares, onde o primeiro item do par é normalmente um símbolo e o segundo item, o valor daquele símbolo, que será usado no corpo do **`let`**.

```
(let [x (inc 1)
      y (+ x 1)]
  (println "Uma simples mensagem do corpo do  let")
  (* x y))
;; Uma simples mensagem do corpo do let
;; => 6
```

No exemplo precedente, o símbolo **`x`** está vinculado ao valor **`(inc 1)`**, que é evaluado como **`2`**, e o símbolo **`y`** está vinculado a soma de **`x`** e **`1`**, que passa a ser

3. Dado esses vínculos, as expressões **`(println "Uma simples mensagem do corpo do let")`** e **`(* x y)`** são evaluadas.

4.7.2. Blocos

Blocos de expressões que devem estar juntas, são criados usando a expressão **`do`** em *Clojure* e normalmente são usado para "efeitos" (side effects), como imprimir algo no console ou logar algo.

Um side effect é alguma coisa que não precisa necessariamente retornar um valor.

A expressão **`do`** aceita como argumento um número arbitrário de outras expressões, mas retorna somente o valor da última expressão:

```
(do
  (println "hello world")
  (println "hola mundo")
  (* 3 5) ;; this value will not be returned; it is thrown away
  (+ 1 2))

;; hello world
;; hola mundo
;; => 3
```

O corpo de uma expressão **`let`**, explicado anteriormente, é muito parecido com a expressão **`do`** no sentido que aceita múltiplas expressões. Na verade, o bloco **`let`** possui um bloco **`do`** implícito.

4.7.3. Loops

A abordagem funcional de *Clojure* significa que ela não possui o tradicional loop **`for`** de outras línguas como *C*. Os loops em *Clojure* funcionam através de recursão. Recursão algumas vezes precisa de um esforço adicional de como modelar um problema um pouco diferente de como se faz em linguagens imperativas.

Muitas dos usos comuns para qual **`for`** é utilizado em outras línguas são obtidos com funções de **`high-order`** - isso é, funções que aceitam outras funções como parâmetros.

Looping com loop/recur

Vamos dar uma olhada em como expressar loops usando recursão com as formas **`loop`** e **`recur`**. **`loop`** define uma lista de vínculos (observe a simetria com **`let`**) e **`recur`** retorna a execução de volta para o loop com novos valores para esses vínculos.

Vamos ver um exemplo:

```
(loop [x 0]
  (println "Looping com " x)
  (if (= x 2)
    (println "Terminei o loop!")
    (recur (inc x))))
;; Looping com 0
;; Looping com 1
;; Looping com 2
;; Terminei o loop!
;; => nil
```

No código acima, nos vinculamos o nome **`x`** ao valor **`0`** e executamos o corpo do loop. Como a condição não é verdadeira na primeira vez, ela é roda novamente com o **`recur`**, com o novo vínculo de **`x`** sendo **`1`**, resultado do **`(inc x)`**. Fazemos isso mais uma vez até que a condição é satisfeita, e não ocorram outras chamadas do **`recur`** e saímos do loop.

Observe que não estamos restritos a usar o **`recur`** somente dentro do **`loop`**. Podemos usá-lo também na execução do corpo de uma função recursiva:

```
(defn recursive-function
  [x]
  (println "Looping com" x)
  (if (= x 2)
    (println "Done looping!")
    (recur (inc x))))

(recursive-function 0)
;; Looping com 0
;; Looping com 1
```

```
;; Looping com 2
;; Terminei o loop!
;; => nil
```

Substituindo loops por funções de *high-order*

Em linguagens de programação imperativas, é comum o uso de **`for`** loops para iterar dados e transformá-los, usualmente com algum dos objetivos abaixo:

- Transformar cada valor na coleção retornando uma nova coleção
- Filtrar alguns elementos na coleção baseado em algum critério
- Converter uma coleção em um valor onde cada iteração depende do resultado da iteração anterior
- Rodar algum tipo de rotina para cada valor na coleção

As ações acimas são expressas em funções de *high-oder* e em construções sintáticas do *Clojure*, vamos ver um exemplo para as três primeiras.

Para transformar cada valor em uma coleção, nos usamos a função **`map`**, que recebe uma função e uma sequência e aplica essa função em cada elemento:

```
(map inc [0 1 2])
;; => (1 2 3)
```

O primeiro argumento do **`map`** pode ser qualquer função que receba **um argumento** e retorne um valor. Por exemplo, se você tiver uma aplicação gráfica e quiser desenhar o gráfico da equação **`y = 3x + 5`** para algum conjunto de valores de **`x`**, você poderia obter os valores de **`y`** assim:

```
(defn y-value [x] (+ (* 3 x) 5))

(map y-value [1 2 3 4 5])
;; => (8 11 14 17 20)
```

Se a função de mapeamento é pequena, você pode usar uma função anônima, seja com a forma normal ou com a sintaxe **`#()`**:

```
(map (fn [x] (+ (* 3 x) 5)) [1 2 3 4 5])
;; => (8 11 14 17 20)

(map #(+ (* 3 %) 5) [1 2 3 4 5])
;; => (8 11 14 17 20)
```

Para filtrar valores em uma coleção, nos usamos a função **`filter`**, que recebe um predicado e uma sequência e retorna uma nova sequência somente com os elementos que retornaram algum valor "logicamente true" para o predicado fornecido:

```
(filter odd? [1 2 3 4])
;; => (1 3)
```

Novamente, você pode usar qualquer função que retorne **`true`** ou **`false`** como o primeiro argumento do **`filter`**. Aqui está um exemplo que mantém somente as palavras com menos de 5 letras. (A função **`count`** retorna o comprimento da coleção passada como argumento - uma string é uma coleção de caracteres.)

```
(filter (fn [word] (< (count word) 5))
 ["ant" "baboon" "crab" "duck" "echidna" "fox"])
;; => ("ant" "crab" "duck" "fox")
```

Convertendo uma coleção para um único valor, acumulando o resultado intermediário a cada passo da iteraçào pode ser obtido usando a função **`reduce`**, que recebe uma função para acumular os valores, um valor inicial opcional e uma coleção:

```
(reduce + 0 [1 2 3 4])
;; => 10

(reduce + [1 2 3 4])
;; => 10
```

Uma outra vez, podemos usar nossa própria função como argumento do **`reduce`**, mas ela deve receber **dois** argumentos. O primeiro é o resultado intermediário e

o segundo é o item da coleção sendo processado. A função retorna o valore que se torna o novo resultado intermediário para ser usado junto com o próximo item na lista. Por exemplo, aqui está o que você obtém a soma dos quadrados de um conjunto de números.

```
(defn sum-squares
  [accumulator item]
  (+ accumulator (* item item)))

(reduce sum-squares 0 [3 4 5])
;; => 50
```

Agora com uma função anônima:

```
(reduce (fn [acc item] (+ acc (* item item))) 0 [3 4 5])
;; => 50
```

Aqui um **`reduce`** que encontra o total de número de caracteres de um conjunto de palavras:

```
(reduce (fn [acc word] (+ acc (count word))) 0
 ["ant" "bee" "crab" "duck"])
;; => 14
```

Aque não usamos a sintaxe **`#()`**, porque apesar de reduzirmos o tamanho do código, ficaria menos legível.

Lembre-se de escolher o valor inicial do seu acumulador com atenção. Se você quiser usar o **`reduce`** para encontrar a multiplicação de uma série de números, você teria que começar com 1 ao invés de 0, se não, estaríamos multiplicando os números por zero!

```
;; valor inicial errado
(reduce * 0 [3 4 5])
;; => 0

;; valor inicial correto
```

```
(reduce * 1 [3 4 5])
;; => 60
```

sequências for

Em *Clojure*, o **for** não é usado para iteração, mas para gerar uma sequência, uma operação também conhecida como "sequence comprehension". Em esta seção, você vai aprender como ela funciona e como usá-la para construir sequências declarativas.

for recebe um vetor de vínculos e uma expressão e gera uma sequência com o resultado de evaluar cada expressão. Vamos ver um exemplo:

```
(for [x [1 2 3]]
  [x (* x x)])
;; => ([1 1] [2 4] [3 9])
```

Nesse exemplo, **x** seria vinculado a cada item do vetor **[1 2 3]** por vez, e retornaria uma nova sequência onde cada item é um vetor de dois itens com o item original e o quadrado dele.

for suporta vários vínculos, o que vai fazer a coleção ser iterada em uma maneira aninhada, muito parecido quando colocamos **for** dentro de **for** em uma linguagem imperativa. O vínculo mais interno itera "mais rápido".

```
(for [x [1 2 3]
      y [4 5]]
  [x y])

;; => ([1 4] [1 5] [2 4] [2 5] [3 4] [3 5])
```

Podemos também colocar depois dos vínculos, três modificadores: **:let** para criar vínculos locais, **:while** para parar a geração da sequência, e `:when`para filtrar valores.

Aqui está um exemplo de vínculos locais utilizando o modificador **:let**, note que os vínculos definitos estarão disponíveis na expressão:

```
(for [x [1 2 3]
      y [4 5]
      :let [z (+ x y)]]
  z)
;; => (5 6 6 7 7 8)
```

Podemos utilizar o modificador **`:while`** para expressar a condição que uma vez que deixar de ser verdade, vamos parar a geraçào da sequência. Aqui está um exemplo:

```
(for [x [1 2 3]
      y [4 5]
      :while (= y 4)]
  [x y])

;; => ([1 4] [2 4] [3 4])
```

Para fitrar os valores gerados, podemos usar o modificador **`:when`** como no exemplo a seguir:

```
(for [x [1 2 3]
      y [4 5]
      :when (= (+ x y) 6)]
  [x y])

;; => ([1 5] [2 4])
```

Podemos combinar os modificados acima para expressar geração de sequências mais complexas ou expressar a intenção do nosso **`for`** loop de maneira mais clara:

```
(for [x [1 2 3]
      y [4 5]
      :let [z (+ x y)]
      :when (= z 6)]
  [x y])

;; => ([1 5] [2 4])
```

Quando listamos os usos mais comuns do **`for`** loop em linguagens imperativas, nós mencionamos que algumas vezes queremos rodar alguma rotina pra cada valor na sequência, sem nos importarmos com o resultado. Normalmente, fazemos isso para realizar algum efeito (*side-effect*) com os valores da sequência.

Clojure provê o construtor **`doseq`**, que é análogo ao **`for`**, mas executa a expressão, descarta o resultado, e retorna **`nil`**. Como o **`for`**, ele aceita os mesmos modificadores **`:let`**, **`:when`** e **`:while`**.

```
(doseq [x [1 2 3]
        y [4 5]
       :let [z (+ x y)]]
  (println x "+" y "=" z))

;; 1 + 4 = 5
;; 1 + 5 = 6
;; 2 + 4 = 6
;; 2 + 5 = 7
;; 3 + 4 = 7
;; 3 + 5 = 8
;; => nil
```

Se você quer simplesmente iterar e aplicar alguma rotina com efeito (*side-effect*) como **`println`** para cada item na coleção, você usar a função especializada **`run!`** que internamente usa uma "redução" mais rápida.

```
(run! println [1 2 3])
;; 1
;; 2
;; 3
;; => nil
```

Essa função explicitamente retorna **`nil`**.

4.8. Tipos de coleções

4.8.1. Imutáveis e persistentes

Como mencionamos antes, as coleções no *Clojure* são persistentes e imutáveis, mas nós não explicamos o que isso significa.

Uma estrutura de dado imutável, como o nome sugere, são estrutura de dados que não podem ser mudadas. Alterações in-loco não são permitidas em estruturas imutáveis.

Vamos ilustrar isso com um exemplo: adicionando valores a um vetor usando **`conj`**(conjoin).

```
(let [xs [1 2 3]
      ys (conj xs 4)]
  (println "xs:" xs)
  (println "ys:" ys))

;; xs: [1 2 3]
;; ys: [1 2 3 4]
;; => nil
```

Como você pode ver, nós derivamos uma versão do vetor **`xs`** ao adicionar um elmento a ele, e obtemos um novo vetor **`ys`** com esse elemento adicionado. Entretanto, o vetor **`xs`** se mantém inalterado, porque ele é imutável.

Uma estrutura persistente é uma estrutura de dados que retorna uma nova versão dela mesmo enquanto a transforma, deixando o original não modificado. *Clojure* faz isso ser eficiente em termos de memória e tempo usando uma técnica de implementação chamada *structural sharing* (compartilhamento de estrutura), onde a maioria dos dados são compartilhados entre as duas versões e não duplicada e as transformações copiam o mínimo possível de dados.

Se você quer saber mais como esse compartilhamento funciona, continue lendo. Se não está interessado em saber mais dos detalhes, sinta-se livre para pular para próxima seção.

Para ilustrar melhor esse *structural sharing* nas estruturas do *Clojure*, vamos comparar se algumas partes da velha e nova versão da estrutura de dados são na verdade o mesmo objeto usando a função **`identical?`**. Nós vamos usar uma lista para isso:

```
(let [xs (list 1 2 3)
      ys (cons 0 xs)]
  (println "xs:" xs)
  (println "ys:" ys)
  (println "(rest ys):" (rest ys))
  (identical? xs (rest ys)))

;; xs: (1 2 3)
;; ys: (0 1 2 3)
;; (rest ys): (1 2 3)
;; => true
```

Como você pode ver no exemplo, nós usamos **`cons`** (construct0 para prefixar um valor a lista **`xs`** e obtemos uma nova lista **`ys`** com o elemento adicionado. O resto de **`ys`** obtidos usando a função **`rest`**, é o mesmo objeto em memória que a lista **`xs`**, e então **`xs`** e **`ys`** compartilham a mesma estrutura.

4.8.2. A Abstração de Sequências

Uma das abstrações centrais do *Clojure* é a *sequence* que pode ser pensando como uma lista e pode ser derivada de qualquer tipo de coleção. É uma coleção persistente e imutável como todos os tipos de coleção, e muitas funções centrais do *Clojure* retornam sequências.

Os tipos que podem ser usados para gerar uma sequência são chamados de "seqables"; nos podemos chamar **`seq`** com eles como argumento e obter uma sequência de volta. Sequências suportam duas operações básicas: **`first`** e **`rest`**. Ambas chamam **`seq`** no argumento fornecidos a eles:

```
(first [1 2 3])
;; => 1

(rest [1 2 3])
```

```
;; => (2 3)
```

Chamando **seq** em um *seqable*, podemos obter resultados diferentes se o *seqable* está vazio ou não. Ele irá retornar **nil** quando a coleção está vazia ou se não, uma sequence:

```
(seq [])
;; => nil

(seq [1 2 3])
;; => (1 2 3)
```

next é similar a operação **rest**, exceto que ela retorna **nil** quando chamada com uma sequência com um ou zero elementos. note que, quando chamamos **rest** com uma sequência vazia, ela retornará um valor "logicamente true"(**()**), enquanto **next** irá retornar um valor "logicalmente false" (**nil**). (revise a seção sobre "O que é verdade em Clojure" caso tenha alguma dúvida sobre isso).

```
(rest [])
;; => ()

(next [])
;; => nil

(rest [1 2 3])
;; => (2 3)

(next [1 2 3])
;; => (2 3)
```

nil-punning

Como **seq** retorna **nil** quando a coleção está valia, e **nil** é considerado um valor "logicalmente false", você pode checar se uma coleção está vazia usando a função **seq**. O termo técnico para isso é nil-punning.

```
(defn print-coll
  [coll]
```

```
  (when (seq coll)
    (println "Saw " (first coll))
    (recur (rest coll))))

(print-coll [1 2 3])
;; Vi 1
;; Vi 2
;; Vi 3
;; => nil

(print-coll #{1 2 3})
;; Vi 1
;; Vi 3
;; Vi 2
;; => nil
```

Apesar de **`nil`** não ser nem um *seqable* nem uma sequência, ele é suportado por todas as funções que vimos até agora:

```
(seq nil)
;; => nil

(first nil)
;; => nil

(rest nil)
;; => ()
```

Funções que trabalham com sequências

As funções centrais do *Clojure* para transformar coleções criam sequências a partir dos seus argumentos e são implementadas em termos das operações genéricas que aprendemos na seção precedente. Isso faz com que elas sejam super genéricas porque podemos usar em qualquer tipo de dado que seja um *seqable*. vamos ver como **`map`** funciona nos diversos tipos de coleções:

```
(map inc [1 2 3])
;; => (2 3 4)
```

```
(map inc #{1 2 3})
;; => (2 4 3)

(map count {:a 41 :b 40})
;; => (2 2)

(map inc '(1 2 3))
;; => (2 3 4)
```

> Quando usamos **`map`** em uma coleção do tipo map, sua função de mapeamento irá receber como argumento um vetor com dois items, contendo chave e valor contidos no mapa. O exemplo abaixo usa destructuring para acessar a chave e o valor de maneira mais simples.

```
(map (fn [[key value]] (* value value))
     {:ten 10 :seven 7 :four 4})
;; => (100 49 16)
```

Obviamente a mesma operação pode ser feita de forma mais idiomática obtendo somente uma *seq* dos valores do mapa:

```
(map (fn [value] (* value value))
     (vals {:ten 10 :seven 7 :four 4}))
;; => (100 49 16)
```

Como você pode notar, funções que operam em sequências são seguras para serem usadas com coleções vazias ou até mesmo **`nil`** já que elas não precisam fazer nada a não ser retornar um sequência vazia quando encontram tais valores.

```
(map inc [])
;; => ()

(map inc #{})
;; => ()

(map inc nil)
;; => ()
```

Nós já vimos alguns exemplos com as funções como **`map`**, **`filter`**, e **`reduce`**, mas *Clojure* fornece uma variedade de funções genéricas que operam em sequências no seu *core namespace*. Note qua muitas das operações que aprendemos até agora funcionam com *seqables* e são extensíveis a tipos definidos pelo usuário.

Podemos verificar se um valor é uma coleção através da função predicado **`coll?`**:

```
(coll? nil)
;; => false

(coll? [1 2 3])
;; => true

(coll? {:language "ClojureScript" :file-extension "cljs"})
;; => true

(coll? "ClojureScript")
;; => false
```

Existem funções predicados similares para checarem se um valor é uma sequência (**`seq?`**) ou um *seqable* (**`seqable?`**):

```
(seq? nil)
;; => false
(seqable? nil)
;; => false

(seq? [])
;; => false
(seqable? [])
;; => true

(seq? #{1 2 3})
;; => false
(seqable? #{1 2 3})
;; => true

(seq? "ClojureScript")
;; => false
(seqable? "ClojureScript")
```

```
;; => false

(seq? '(1 2 3))
;; => true
(seqable? '(1 2 3))
;; => true
```

Para coleções que podem ser contadas em tempo constante, podemos utilizar a função **`count`**. Essa operação inclusive funciona com strings, apesar de elas não serem uma coleção, sequência, ou seqable.

```
(count nil)
;; => 0

(count [1 2 3])
;; => 3

(count {:language "ClojureScript" :file-extension "cljs"})
;; => 2

(count "ClojureScript")
;; => 13
```

Nós também podemos obterm uma variante vazia de uma certa coleção através da função **`empty`**:

```
(empty nil)
;; => nil

(empty [1 2 3])
;; => []

(empty #{1 2 3})
;; => #{}
```

A função predicado **`empty?`** retorna **`true`** se uma certa coleção está vazia:

```
(empty? nil)
;; => true
```

```
(empty? [])
;; => true

(empty? #{1 2 3})
;; => false
```

A função **conj**(conjoin) adiciona um elemento a um coleção e pode adicionar ela em diferentes "lugares" dependendo do tipo da coleção. O elemento é adicionado onde é mais performático, mas note que nem toda coleção possui uma ordem definida.

Podemos passar muitos elementos quanto quisermos para **conj**; vamos ver alguns exemplos:

```
(conj nil 42)
;; => (42)

(conj [1 2] 3)
;; => [1 2 3]

(conj [1 2] 3 4 5)
;; => [1 2 3 4 5]

(conj '(1 2) 0)
;; => (0 1 2)

(conj #{1 2 3} 4)
;; => #{1 3 2 4}

(conj {:language "ClojureScript"} [:file-extension "cljs"])
;; => {:language "ClojureScript", :file-extension "cljs"}
```

Laziness/Preguiça

Maioria das funções que retornam sequência no *Clojure* retornam uma sequência "preguiçosa" ao invés de calcular todos os elementos da nova sequênica. Sequências *Lazy* geram seu conteúdo a medida que são solicitadas a fazê-lo, normalmente quando estamos iterando sob elas. Laziness/Preguiça garante que

não estamos fazendo mais trabalho do que precisamos e nos dá a possibilidade de tratar sequências potencialmente infinitas como sequências comuns.

Considere a função **`rage`**, que gera uma sequência de inteiros:

```
(range 5)
;; => (0 1 2 3 4)
(range 1 10)
;; => (1 2 3 4 5 6 7 8 9)
(range 10 100 15)
;; (10 25 40 55 70 85)
```

Se você dizer apenas **`(range)`**, você irá obter uma sequência de todos os inteiros. Não tente isso dentro de uma REPL, pois ela tentará evaluar a expressão e todos os elementos da sequência.

Aqui está um exemplo controlado. Supondo que você esteja escrevendo um programa gráfico e queira desenhar o gráfico da equação $y = 2x^2 + 5$, e queira somente aqueles valores de **`x`** para quais **`y`** é menor que 100. Você pode geral todos os números entre 0 e 100, que certamente serão suficientes, e então usar **`take-while`** com a condição **`y # 100`**:

```
(take-while (fn [x] (< (+ (* 2 x x) 5) 100))
            (range 0 100))
;; => (0 1 2 3 4 5 6)
```

4.8.3. Entendendo Coleções a fundo

Agora que conhecemos a abstração de serquências de Clojure e algum das funções para manipulá-las, é hora de conhecermos alguns tipos de coleções concretas e as operações que elas suportam.

Listas

Em *Clojure*, listas são uma estrutura de dados usada principalmente par agrupar símbolos juntos para criar programas. Diferente de outras Lisps, muitas construções sintáticas do *Clojure* usam estruturas sintáticas diferentes da lista

(como vetores e mapas). Isso torna o código menos uniforme, porém aumenta a facilidade de leitura.

Você pode pensar nas listas do *Clojure* como listas ligadas (mas não duplamente), onde cada nó contém um valor e um ponteiro para o resto da lista. Isso faz com que seja natural (e rápido) adicionar items ao começo da lista, já que adicionar ao fim iria criar a necessidade de percorrer toda a lista. Essa adição é feita utilizando a função **`cons`**.

```
(cons 0 (cons 1 (cons 2 ())))
;; => (0 1 2)
```

Nós usamos a sintaxe literal **`()`** para representar uma lista vazia. Já que ela não contém nenhum símbolo, não é tratada como uma chamada de função. porém, quando usando a sintaxe literal de listas que contenham elementos, precisamos prefixá-la com aspa **`'`** para previnir que *Clojure* evalue ela como uma chamada de função:

```
(cons 0 '(1 2))
;; => (0 1 2)
```

Como adicionar ao começo (ou cabeça, ou head) leva tempo constante para ser feito, a função **`conj`** operando em listas adiciona items ao começo.

```
(conj '(1 2) 0)
;; => (0 1 2)
```

Listas e outras estruturas de dados do *Clojure* podem ser usadas como pilhas usando as funções **`peek`**, **`pop`** e **`conj`**. Note que o topo da pilha será o lugar onde **`conj`** vai adicionar elementos, fazendo **`conj`** equivalente a função **`push`** de uma pilha. No caso de listas, **`conj`** adiciona elementos no começo da lista, e **`peek`** retorna o primeiro elemento da lista, e **`pop`** retorna a lista com todos os elementos exceto o primeiro.

Note que as duas operações que retornam a pilha (**`conj`** e **`pop`**) não mudam o tipo da coleção usada pela pilha.

```
(def list-stack '(0 1 2))

(peek list-stack)
;; => 0

(pop list-stack)
;; => (1 2)

(type (pop list-stack))
;; => cljs.core/List

(conj list-stack -1)
;; => (-1 0 1 2)

(type (conj list-stack -1))
;; => cljs.core/List
```

Uma coisa que listas não particularmente goas é acesso arbitrário de elementos através de um index. Como elas são armazenadas como listas ligadas em memória, para acesso arbitrário a um certo index, é necessário percorrer a lista em ordem para obter o elemento desejado, ou lançar uma exceção de index não presente na lista, caso o index maior que a quantidade de elementos na lista. Outras coleções que não possuem index também sofrem dessa limitação como *lazy sequences* .

Vetores

Vetores são uma das estruturas de dados mais comum em *Clojure*. Elas são usadas como estrutura sintática em vários lugares em que Lisps mais tradicionais utilizam lista, como por exemplo em declaração de argumentos de funções e em blocos de vínculos **`let`**.

Vetores em *Clojure* são delimitados por cochetes **`[]`** como sintaxe literal. Eles também podem ser criados com a função **`vector`** ou a partir de outra coleção utilizando a função **`vec`**:

```
(vector? [0 1 2])
;; => true

(vector 0 1 2)
```

```
;; => [0 1 2]

(vec '(0 1 2))
;; => [0 1 2]
```

Vetores são, como listas, coleções ordenadas de valores heterogêneos. Diferente de listas, vetores crescem naturalmente a partir do fim deles (ou cauda), então a função **`conj`** adiciona itens ao final do vetor. Adições ao fim do vetor são feitas em tempo constante:

```
(conj [0 1] 2)
;; => [0 1 2]
```

Outra coisa que diferencia listas de vetores é que vetores são coleções indexadas e suportam acesso eficiente a items através de índices e atualizações nào destrutivas. Nós usamos a função **`nth`** para obter valores de um certo index:

```
(nth [0 1 2] 0)
;; => 0

(nth [0 1 2] 2)
;; => 2
```

Como vetores associam chaves sequênciais númericas (indexes) aos valroes, podemos tratar eles como uma estrutura associativa. *Clojure* provê a função **`assoc`** que dado uma estrutura de dados associativa, um conjunto de pares de chave e valores retorna uma nova estrutura de dados com as chaves fornecidas modificadas. O index começa em 0 referindo-se ao primeiro elemento do vetor.

```
(assoc ["cero" "uno" "two"] 2 "dos")
;; => ["cero" "uno" "dos"]
```

Note que podemos somente associar uma chave que já está no vetor ou é a última posição do vetor (fazendo ele crescer):

```
(assoc ["cero" "uno" "dos"] 3 "tres")
```

```
;; => ["cero" "uno" "dos" "tres"]

(assoc ["cero" "uno" "dos"] 4 "cuatro")
;; Error: Index 4 out of bounds [0,3]
```

Algo supreendente é que estruturas de dados associativas podem ser usadas como funções. Elas sào funções das suas chaves para os valores associado a elas. No caso dos vetores, se uma certa chave não está presente uma exceção é lançada:

```
(["cero" "uno" "dos"] 0)
;; => "cero"

(["cero" "uno" "dos"] 2)
;; => "dos"

(["cero" "uno" "dos"] 3)
;; Error: Not item 3 in vector of length 3
```

Como listas, vetores podem também ser usadas como pilhas com as funções **`peek`**, **`pop`** e **`conj`**. É importante notar que vetores crescem na direção contrária das listas:

```
(def vector-stack [0 1 2])

(peek vector-stack)
;; => 2

(pop vector-stack)
;; => [0 1]

(type (pop vector-stack))
;; => cljs.core/PersistentVector

(conj vector-stack 3)
;; => [0 1 2 3]

(type (conj vector-stack 3))
;; => cljs.core/PersistentVector
```

As funções **`map`** e **`filter`** são operações que retornam uma *lazy sequence*, mas é comum precisarmos de sequências onde todos os valores já foram computados. Por isso, existem as funções **`mapv`** e **`filterv`** que funcionam igual **`map`** e **`filter`** porém retornam vetores. Elas tem a vantagem de serem mais rápidas do que construir um vetor a partir de uma *lazy sequence* e fazendo a intenção do código mais explícita:

```
(map inc [0 1 2])
;; => (1 2 3)

(type (map inc [0 1 2]))
;; => cljs.core/LazySeq

(mapv inc [0 1 2])
;; => [1 2 3]

(type (mapv inc [0 1 2]))
;; => cljs.core/PersistentVector
```

Mapas

Mapas são onipresente em *Clojure*. Como vetores, eles também são usados como parte da sintaxe da linguagem, principalmente para adição de metadadaos a uma var. Qualquer estrutura de dados em *Clojure* pode ser usada como uma chave em um mapa, apesar de ser comum usarmos *keywords* pois elas também podem ser chamadas como funções.

Mapas em *Clojure* são escritos literalmente como pares chave-valor envoltos por chaves **`{}`**. Alternativamente, eles também pode ser criados usando a função **`hash-map`**:

```
(map? {:name "Cirilla"})
;; => true

(hash-map :name "Cirilla")
;; => {:name "Cirilla"}

(hash-map :name "Cirilla" :surname "Fiona")
```

```
;; => {:name "Cirilla" :surname "Fiona"}
```

Como mapas não possuem uma ordem específica, a função **conj** apenas adiciona um ou mais pares de chave-valor ao mapa. **conj** para mapas espera um ou mais sequência de pares chave-valor como seus últimos argumentos:

```
(def ciri {:name "Cirilla"})

(conj ciri [:surname "Fiona"])
;; => {:name "Cirilla", :surname "Fiona"}

(conj ciri [:surname "Fiona"] [:occupation "Wizard"])
;; => {:name "Cirilla", :surname "Fiona", :occupation "Wizard"}
```

No exemplo anterior, por causalidade, a ordem foi preservada, mas para muitas chaves, você verá que a ordem não é preservada.

Mapas associam chaves a valores e são portanto uma estrutura de dados associativa. Eles suportam a adição de novas associações usando a função **assoc** e, diferente de vetores, a remoção usando **dissoc**. **assoc** também pode atualizar valores de uma chave existente. Vamos ver como essas funções funcionam:

```
(assoc {:name "Cirilla"} :surname "Fiona")
;; => {:name "Cirilla", :surname "Fiona"}
(assoc {:name "Cirilla"} :name "Alfonso")
;; => {:name "Alfonso"}
(dissoc {:name "Cirilla"} :name)
;; => {}
```

Mapas também são funções das suas chaves, retornando os valores relacionados com a chave passada como argumentol. Diferente de vetores, eles retornam **nil** caso a chave não esteja presente no mapa:

```
({:name "Cirilla"} :name)
;; => "Cirilla"

({:name "Cirilla"} :surname)
```

```
;; => nil
```

Clojure também fornece hash maps ordenados que se comportam como sua versão não ordenada mas preservam a ordem quando iteramos sob seus elementos. Podemos criar um mapa ordenado com um ordenamento padrão usando a função **`sorted-map`**:

```
(def sm (sorted-map :c 2 :b 1 :a 0))
;; => {:a 0, :b 1, :c 2}

(keys sm)
;; => (:a :b :c)
```

Se precisarmos de um ordenamento diferente, podemos prover uma função comparadora a função **`sorted-map-by`**, vamos ver um exemplo em que invertemos a ordem retornada pela função **`compare`**. Uma função comparadora recebe dois items para comparar e retorna -1 (se o primeiro item é menor que o segundo), 0 (se eles são iguais) e 1 (se o primeiro item é maior que o segundo).

```
(defn reverse-compare [a b] (compare b a))

(def sm (sorted-map-by reverse-compare :a 0 :b 1 :c 2))
;; => {:c 2, :b 1, :a 0}

(keys sm)
;; => (:c :b :a)
```

Conjuntos (ou Sets)

Sets em *Clojure* também possuem uma sintaxe literal que é **`#{}`** e também podem ser criados utilizando a função **`set`**. Eles são coleções não ordenadas de valores sem duplicações.

```
(set? #{\a \e \i \o \u})
;; => true

(set [1 1 2 3])
```

```
;; => #{1 2 3}
```

A sintaxe literal não permite duplicações. Se você escrever um set literal com duplicações um erro será lançado:

```
#{1 1 2 3}
;; clojure.lang.ExceptionInfo: Duplicate key: 1
```

Existem várias operações que podem ser realizadas com sets, porem elas estão localizadas no *namespace* **`clojure.set`** e portanto precisam ser importadas. Você vai aprender os detalhes sobre namespaces mais tarde, por hora, você só precisa saber que estamos carregando um *namespace* chamado **`clojure.set`** e vinculá-lo ao símbolo **`s`**:

```
(require '[clojure.set :as s])

(def danish-vowels #{\a \e \i \o \u \æ \ø \å})
;; => #{"a" "e" "å" "æ" "i" "o" "u" "ø"}

(def spanish-vowels #{\a \e \i \o \u})
;; => #{"a" "e" "i" "o" "u"}

(s/difference danish-vowels spanish-vowels)
;; => #{"å" "æ" "ø"}

(s/union danish-vowels spanish-vowels)
;; => #{"a" "e" "å" "æ" "i" "o" "u" "ø"}

(s/intersection danish-vowels spanish-vowels)
;; => #{"a" "e" "i" "o" "u"}
```

Uma propriedade interessante dos sets é que eles podem ser aninhados. Linguagens que possuem sets mutáveis podem acabar contendo valores duplicados, mas isso não pode acontecer no *Clojure*. Todas as estruturas de dado do *Clojure* podém ser aninhadas de forma arbitrária devido a imutabilidade.

Sets também suportam a operação genérica **`conj`** como todas as outras coleções suportam.

```
(def spanish-vowels #{\a \e \i \o \u})
;; => #{"a" "e" "i" "o" "u"}

(def danish-vowels (conj spanish-vowels \æ \ø \å))
;; => #{"a" "e" "i" "o" "u" "æ" "ø" "å"}

(conj #{1 2 3} 1)
;; => #{1 3 2}
```

Sets funcionam como uma estrutura associativa que associam os valores que ele contém a eles mesmos. E como qualquer valore exceto **`nil`** e **`false`** são "logicamente verdade" em *Clojure*, podemos usar os sets como funções predicados:

```
(def vowels #{\a \e \i \o \u})
;; => #{"a" "e" "i" "o" "u"}

(get vowels \b)
;; => nil

(contains? vowels \b)
;; => false

(vowels \a)
;; => "a"

(vowels \z)
;; => nil

(filter vowels "Hound dog")
;; => ("o" "u" "o")
```

Sets também possuem uma variante ordenada como mapas que é criada utilizando as funções **`sorted-set`** e **`sorted-set-by`** que são análogas as funções **`sorted-map`** e **`sorted-map-by`**.

```
(def unordered-set #{[0] [1] [2]})
;; => #{[0] [2] [1]}
```

```
(seq unordered-set)
;; => ([0] [2] [1])

(def ordered-set (sorted-set [0] [1] [2]))
;; =>## {[0] [1] [2]}

(seq ordered-set)
;; => ([0] [1] [2])
```

Filas

Clojure também provê uma fila persistente e imutável. Filas não são usadas tanto quanto outros tipos de coleções. Elas podem ser criadas utilizando a sintaxe literal **`#queue []`**, mas não existe uma função para criá-las. ClojureScript also provides a persistent and immutable queue. Queues are not used as pervasively as other collection types. They can be created by simply getting the empty queue: **`(clojure.lang.PersistentQueue/EMPTY)`**. Porém não existe nenhuma função construtora para criá-las.

```
(def pq (conj (clojure.lang.PersistentQueue/EMPTY) 1 2 3))
;; => #object[clojure.lang.PersistentQueue 0x28cb9120
 "clojure.lang.PersistentQueue@7861"]
```

Usar **`conj`** para adiconar valores a files, os adiciona ao fim dela:

```
(def pq (conj (clojure.lang.PersistentQueue/EMPTY) 1 2 3))
;; => #object[clojure.lang.PersistentQueue 0x28cb9120
 "clojure.lang.PersistentQueue@7861"]

(last (conj pq 4 5))
;; => 5
```

Infelizmente, as filas não são impressas de forma muito legível na Repl. É importante lembrar que as operações que usamos para pilha funcionam de maneira diferente para filas. **`pop`** retira valores do começo da fila, e **`conj`** coloca valores no fim da fila.

```
(def pq (conj (clojure.lang.PersistentQueue/EMPTY) 1 2 3))
```

```
;; => #object[clojure.lang.PersistentQueue 0x28cb9120
 "clojure.lang.PersistentQueue@7861"]

(peek pq)
;; => 1

(println (mapv identity (pop pq)))
;; => [2 3]

(println (mapv identity (conj pq 4)))
;; => [1 2 3 4]
```

Filas não são usadas com muita frequência como lista ou vetores, mas é bom saber que elas estão disponíveis em *Clojure*, e podem eventualmente serem úteis.

4.9. Desestruturando (*Destructuring*)

Destructuring, como o nome sugere, é um modo de quebrar uma estrutura de dados como coleções e focar em partes individuais dela. *Clojure* oferece uma sintaxe concisa para desestruturar seja sequência indexadas ou estruturas associativas que pode ser usada onde criamos vínculos entre símbolos e valores.

Vamos ver um exemplo de *destructuring* que é útil pare entendermos o parágrafo anterior. Imagine que você tem uma sequência e esteja interessado somente no primeiro e terceiro item. Você pode obter uma referência a eles usando a função **`nth`**:

```
(let [v [0 1 2]
      fst (nth v 0)
      thrd (nth v 2)]
  [thrd fst])
;; => [2 0]
```

Porém, o código anterior é um pouco verboso. *Destructuring* pode nos ajudar a extrair valores de uma sequência indexada de maneira mais sucinta se usarmos um vetor no lado esquerdo do vínculo:

```
(let [[fst _ thrd] [0 1 2]]
```

```
  [thrd fst])
;; => [2 0]
```

No exemplo acima, **`[fst _ thrd]`** é uma expressão de *destructuring*. Ela é representada como um vetor e é usada para vincular valores indexados aos símbolos **`fst`** e **`thrd`**, correspondendo aos valores com index **`0`** e **`2`** respectivamente. O símbolo **`_`** é utilizado como um placeholder para valores que não estamos interessados, nesse caso **`1`**.

Note que *destructuring* não está limitado aos vínculos criados dentro de um **`let`**; ele funciona em qualquer lugar que criámos um vínculo entre valores e símbolos como nas formas especiais **`for`** e **`doseq`** ou em argumentos de funções. Podemos escrever uma função que receba um pair e troque as posições desse par de maneira muito sucinta utilizando a sintaxe do *destructuring* como argumento da função:

```
(defn swap-pair [[fst snd]]
  [snd fst])

(swap-pair [1 2])
;; => [2 1]

(swap-pair '(3 4))
;; => [4 3]
```

Destructuring posicional com vetores é bastante útil para tirarmos valores indexados de uma sequência, mas algumas vezes não queremos descartar o resto dos elementos na sequência. Similar com ao **`&`** usado em funções de aridade variável, o **`&`** pode ser usado dentro do *destructuring* de um vetor para agrupar o resto dos elementos de uma sequência:

```
(let [[fst snd & more] (range 10)]
  {:first fst
   :snd snd
   :rest more})
;; => {:first 0, :snd 1, :rest (2 3 4 5 6 7 8 9)}
```

Note que o valor na posição **0** foi vinculado a **fst**, o valor na posição **1** foi vinculado a **snd**, e a sequência de elementos a partir da posição **2** foi vinculada ao símbolo **more**.

Nós podemos ainda estar interessados na estrutura de dados como um todo, mesmo quando estamos fazendo um *destructuring*. Podemos manter a referência a estrutura usando a *keyword* **:as**. Se usada dentro de um *destructuring*, a estrutura de dados original fica vinculada ao símbolo que segue a *keyword* **:as**:

```
(let [[fst snd & more :as original] (range 10)]
  {:first fst
   :snd snd
   :rest more
   :original original})
;; => {:first 0, :snd 1, :rest (2 3 4 5 6 7 8 9), :original (0 1 2 3 4 5 6
 7 8 9)}
```

Não só podemos usar *destructuring* com sequências, mas estruturas associativas ainda podem ser desestruturadas. Nesse caso, o *destructuring* é representado utilizando um map ao invés de um vetor. Nesse mapa, as chaves são símbolos que queremos vincular aos valores e os valores são as chaves que queremos usar para obter os valores dentro da estrutura associativa que queremos desestruturar. Vamos ver um exemplo:

```
(let [{language :language} {:language "ClojureScript"}]
  language)
;; => "ClojureScript"
```

No exemplo acima, estamos extraindo o valor associado com a chave **:language** e vinculando ele com o símbolo **language**. Quando procuramos o map por uma chave que não está presente, o símbolo é vinculado a **nil**:

```
(let [{name :name} {:language "ClojureScript"}]
  name)
;; => nil
```

Destructuring associative permite o fornecimento de valores *default* caso a chave não esteja presente no mapa. Um mapa seguido da chave `:or` é usada para valores *default* como o exemplo seguinte mostra:

```
(let [{name :name :or {name "Anonymous"}} {:language "ClojureScript"}]
  name)
;; => "Anonymous"

(let [{name :name :or {name "Anonymous"}} {:name "Cirilla"}]
  name)
;; => "Cirilla"
```

Destructuing associativo também suporta vincular a estrutura de dados original a um símbolo colocado depois da keyword `:as`:

```
(let [{name :name :as person} {:name "Cirilla" :age 49}]
  [name person])
;; => ["Cirilla" {:name "Cirilla" :age 49}]
```

Keyword não são as únicas coisas que podem ser chaves em uma estrutura associative. Números, strings, símbolos e muitas outras estruturas de dados podem ser usadas como chave, então também podemos usá-las no *destruturing*. Mas note que para símbolos, precisamos prefixá-los com aspa `'` para evitar que sejam evaluados.

```
(let [{one 1} {0 "zero" 1 "one"}]
  one)
;; => "one"

(let [{name "name"} {"name" "Cirilla"}]
  name)
;; => "Cirilla"

(let [{lang 'language} {'language "ClojureScript"}]
  lang)
;; => "ClojureScript"
```

Como normalmente os valores correspondente as chaves são usualmente vinculado a símbolos com o mesmo nome (Ex: **`:language`** e **`language`**) e chaves normalmente são keywords, strings ou símbolos, *Clojure* oferece uma sintaxe mais simples para esses casos.

Vamos ver alguns exemplos desses casos, começando pelas *keywords* usando **`:keys`**:

```
(let [{:keys [name surname]} {:name "Cirilla" :surname "Fiona"}]
  [name surname])
;; => ["Cirilla" "Fiona"]
```

Como você pode ver, utilizamos a *keyword* **`:keys`** e associamos ela a um vetor de símbolos, os valores correspondentes as versões chaves do símbolo são vinculado a eles. A expressão **`{:keys [name surname]}`** é equivalente a **`{name :name surname :surname}`**, porém mais sucinta.

A versão string e simbólica dessa sintaxe funciona exatamente igual, só que nesses casos usamos **`:strs`** e **`:syms`** respectivamente:

```
(let [{:strs [name surname]} {"name" "Cirilla" "surname" "Fiona"}]
  [name surname])
;; => ["Cirilla" "Fiona"]

(let [{:syms [name surname]} {'name "Cirilla" 'surname "Fiona"}]
  [name surname])
;; => ["Cirilla" "Fiona"]
```

Uma propriedade interessante do *destruturing* é que podemos ter um dentro do outro de maneira arbitrária, o que torna o código para acesso de estruturas aninhadas muito fácil de compreender, uma vez que ele faz mímica da estrutura da coleção:

```
(let [{[fst snd] :languages} {:languages ["ClojureScript" "Clojure"]}]
  [snd fst])
;; => ["Clojure" "ClojureScript"]
```

4.10. *Threading Macros* (ou ordenando chamadas consecutivas de funções)

Threading macros, também conhecido como funções flecha, permite que escrevamos código mais fácil de ler quando possuímos várias chamadas de funções aninhadas.

Imagine que você tenha `(f (g (h x)))` onde a função `f` recebe como seu primeiro argumento o resultado da chamada da função `g` repetida e assim por diante. Com a *threading macro* mais básica → podemos converter essa expressão em `(-> x (h) (g) (f))` que é muito mais fácil de ler.

O resulta é uma sintaxe mais simples, porque a função flecha é definida como macro e não impacta na performance do código. A forma `(-> x (h) (g) (f))` é convertida automaticamente para `(f (g (h x)))` durante a compilação

Observe que os parenteses em `h`, `g` e `f` são opcionais e podem ser omitidos: `(f (g (h x)))` é o mesmo que `(-> x h g f)`.

4.10.1. *Thread-first macro* (`->`)

Isso é chamado de *thread first*, ou thread primeiro, poque o primeiro argumento das diferentes funções é o resultado da expressão anterior.

Usando um exemplo mais concreto, assim é como seria o código sem utilizarmos *threading macro*:

```
(def book {:name "Lady of the Lake"
           :readers 0})

(update (assoc book :age 1999) :readers inc)
;; => {:name "Lady of the lake" :age 1999 :readers 1}
```

Podemos reescrevê-lo com a _threading macro `->`:

```
(-> book
    (assoc :age 1999)
```

```
    (update :readers inc))
;; => {:name "Lady of the lake" :age 1999 :readers 1}
```

Esse *threading macro* é especialmente útil para transformarmos estrutura de dados, porque em *Clojure*, funções que transforma estruturas de dados recebem como primeiro argumento a estrutura de dados.

4.10.2. Thread-last macro, ou thread por último (`->>`)

A principal diferença entre *thread-last* e *thread-first* é que ao invés do retorno ser utilizado como primeiro argumento das funções, ele é utilizado como último.

Vejamos um exemplo:

```
(def numbers [1 2 3 4 5 6 7 8 9 0])

(take 2 (filter odd? (map inc numbers)))
;; => (3 5)
```

O mesmo código reescrito com `->>`:

```
(->> numbers
     (map inc)
     (filter odd?)
     (take 2))
;; => (3 5)
```

Esse thread macro é especialmente útil para transformarmos sequências ou coleções com `map`, `filter`, `reduce` e outras funções, pois no *Clojure* essas funções recebem como último argumento a sequência.

4.10.3. Thread-as macro ou thread-como (`as->`)

Finalmente, tem os casos que nem `->` nem `->>` são aplicáveis. Nesses casos, tudo que você precisa utilizar é `as->`, a alternativa mais flexível, que permite aplicar o retorno em qualquer posição da chamada de função, e não somente a primeira ou a última.

Essa forma espera dois argumentos fixos e um número arbitrário de expressões. Assim como **`->`**, o primeiro argumento é o valor a ser utilizado nas formas seguintes. O segundo arbumento é um símbolo ao qual esse valor estará vinculado. Em cada forma subsequente, podemos utilizar esse símbolo para nos referirmos ao valor retornado na forma anterior.

Vejamos um exemplo:

```
(as-> numbers $
  (map inc $)
  (filter odd? $)
  (first $)
  (hash-map :result $ :id 1))
;; => {:result 3 :id 1}
```

4.10.4. Thread-some macros ou thread se algo (**`some->`** and **`some->>`**)

Duas outras *threading macros* mais especializadas que o *Clojure* possui são as *thread-some*. Elas funcionam de forma semelhante a **`->`** e **`->>`** exceto que elas encerram a execução das formas uma vez que alguma expressão retorne **`nil`**.

Vejamos um exemplo:

```
(some-> (rand-nth [1 nil])
        (inc))
;; => 2

(some-> (rand-nth [1 nil])
        (inc))
;; => nil
```

Esse é um modo fácil de evitar uma null pointer exception.

4.10.5. Thread-cond macros ou thread se (`cond->` and `cond->>`)

As macros `cond->` and `cond->>` são análogas a `->` e `->>` e oferecem a abilidade de pularmos alguns itens na pipeline. Vejamos um exemplo:

```
(defn describe-number
  [n]
  (cond-> []
    (odd? n) (conj "odd")
    (even? n) (conj "even")
    (zero? n) (conj "zero")
    (pos? n) (conj "positive")))

(describe-number 3)
;; => ["odd" "positive"]

(describe-number 4)
;; => ["even" "positive"]
```

A expressão seguinte é executada somente quando a condição é "logicamente true".

4.10.6. Leituras adicionais (em inglês)

- http://www.spacjer.com/blog/2015/11/09/lesser-known-clojure-variants-of-threading-macro/
- http://clojure.org/guides/threading_macros

4.11. Namespaces

4.11.1. Definindo um *namespace*

Um *namespace* é um peça fundamental de modularizar o código. *Namespaces* são análagos aos pacotes em Java ou módulos em Ruby ou Python e podem ser definidos com a macro `ns`. Se você já deu uma olhada algum código em *Clojure*, você terá notado que alguns arquivoc começam com:

```
(ns myapp.core
  "Alguma string de documentação.")

(def x "hello")
```

Namespaces são dinâmicos, significando que você pode criá-los a qualquer momento. Entretanto a convenção é possuir um *namespace* por arquivo. Naturalmente, a definição de um *namespace* está usualmente no começo de um arquivo, seguida de um docstring opcional.

Antes, explicamos vars e símbolos. Cada var que definimos será associada com um namespace. Se você não define um *namespace*, então o default, "user" será usado:

```
(def x "hello")
;; => #'user/x
```

4.11.2. Carregando outros *namespaces*

Definir um *namespace* e variáveis dentro dele, mas não é muito útil se não podemos utilizar símbolos de outros namespaces. Para isso, a macro **ns** oferece um jeito simpels de carregar outros *namespaces*:

Observe o exemplo seguinte:

```
(ns myapp.main
  (:require myapp.core
            clojure.string))

(clojure.string/upper-case myapp.core/x)
;; => "HELLO"
```

Como você pode observar, estamos usando nomes qualificados (*namespace* + nome da var) para acessar vars e funções de um outros *namespaces*.

Isso permite acessarmos outros *namespaces*, mas também é repetitivo e extremamente verbose. E será bem mais verbose se o nome do namespace for muito grande. Para resolver isso podemos utilizar a diretiva **:as** para criar um "apelido" (*alias*) para o *namespace*.

```
(ns myapp.main
  (:require [myapp.core :as core]
            [clojure.string :as str]))

(str/upper-case core/x)
;; => "HELLO"
```

Adicionalmente, *Clojure* oferece um jeito simples de se refer a uma var ou função de um *namespace* usando a diretiva **`:refer`**, seguido pela sequência de símbolos que vão referir-se as vars daquele *namespace*. Efetivamente, é como se essas vars e funções fizessem parte do seu *namespace*, e não precisamos qualificá-las.

```
(ns myapp.main
  (:require [clojure.string :refer [upper-case]]))
(upper-case x)
;; => "HELLO"
```

E finalmente, é importante saber que tudo localizado no *namespace* **`clojure.core`** é automaticamente carregado e não deveria ser feito o **`require`** de forma explícita. Algumas vezes, você pode querer definir vars com nomes que conflituam com aqueles definidos no *namespace* **`clojure.core`**. Para fazer isso, a macro **`ns`** oferece outra diretiva que permite excluir alguns símbolos específicos e prevenir que eles sejam carregados.

Vejamos um exemplo:

```
(ns myapp.main
  (:refer-clojure :exclude [min]))

(defn min
  [x y]
  (if (> x y)
    y
    x))
```

A macro **`ns`** tem outras diretivas para carregar classes da linguagem hospeira , Java, com **`:import`**, mas isso é explicado em outra seção.

4.11.3. *Namespaces* e nome de arquivos

Quando você possui um *namespace* como **`myapp.core`**, o código deve estar em um arquivo chamado **`core.clj`** dentro do diretório **`myapp`**. Então, no exemplo precedente com os *namespaces* **`myapp.core`** e **`myapp.main`** seriam encontrados em um projeto com uma estrutura de arquivos assim:

```
myapp
└── src
    └── myapp
        ├── core.cljs
        └── main.cljs
```

4.12. Abstrações e Polimorfismos

Tenho certeza que em mais de uma vez você se encontrou em uma situação somo essa: você definiu uma ótima abstração (utilizando interfaces ou algo similar) para sua "lógica de negócio", e você precisa lidar com outro módulo sob o qual você não tem nenhum controle, e você provavelmente estava pensando em criar adapters, proxies, e outras abordagens que implicam em um grande volume de complexidade adicional.

Algumas linguagens dinâmicas permitem "mokey-patching": linguagems onde as classes são abertas e qualquer método pode ser definido e redefinido a qualquer momento. Também sabemos que essa é uma má prática.

Nós nào podemos confiar em uma linguagem que permite que você sobreescreva métodos que você está usando quando importarmos uma biblioteca de um terceiro. Não podemos esperar um comportamento consistente quando isso acontece.

Esses sintomas são comumente chamados de "problema de expressão" veja http://en.wikipedia.org/wiki/Expression_problem para mais detalhes.

4.12.1. Protocolos

O mecanismo em *Clojure* para definir "interfaces" é chamado protocolo (*protocol*). Um protocolo consiste de um nome e um conjunto de assinaturas de funções.

Todas as funções tem ao menos um argumento correspondendo ao **`this`** em Javascript ou **`self`** em Python.

Protocolos provém um polimorfismo baseado em tipos, e eles escolhem qual função executar baseado no tipo do primeiro argumento.

Um protocolo parece o seguinte:

```
(ns myapp.testproto)

(defprotocol IProtocolName
  "A docstring describing the protocol."
  (sample-method [this] "A doc string associated with this function."))
```

> o prefixo "I" é comumente usado para criar uma separação entre protocolos e tipos. Na comunidade *Clojure*, existe opiniões muitas diferentes sobre como o prefixo "I" deveria ser usado. Na nossa opinião, é uma solução aceitável para evitar conflito de nomes e possíveis confusões. Porém, nào utilizar esse prefixo não é considerado uma prática ruim.

Da perspectiva do usuário, funções de um protocolo são simpels funções definidas no *namespace* onde o protocolo foi definido. Isso permite uma abordagem fácil e simples para evitar conflitos entre diferentes protocolos implementados para o mesmo tipo que contém nomes de funções iguais.

Aqui está um exemplo. Vamos criar um protocolo chamado **`IInvertible`** para os dados que podem ser "invertidos". Ele irá conter um único método chamado **`invert`**.

```
(defprotocol IInvertible
  "Esse protocolo é para tipos de dados que são 'invertíveis'"
  (invert [this] "Invert the given item."))
```

Extendendo tipos já existentes

Uma dos pontos positivos de protocolos é a habilidade de extender tipos existentes e talvez tipos providos por bibliotecas de terceiros. Essa operação pode ser feito de jeitos diferentes.

A maioria das vezes você irá usar as macros **extend-protocol** ou **extend-type**. A sintaxe do **`extend-type`** é assim:

```
(extend-type TypeA
  ProtocolA
  (function-from-protocol-a [this]
    ;; implementação aqui
    )

  ProtocolB
  (function-from-protocol-b-1 [this parameter1]
    ;; implementação aqui
    )
  (function-from-protocol-b-2 [this parameter1 parameter2]
    ;; implementação aqui
    ))
```

Como você pode observar, que com **extend-type** você pode extender um só tipo com diferentes protocolos em uma única expressão.

Vamos brincar com o nosso protocolo **`IInvertible`** definido anteriormente:

```
(extend-type String
  IInvertible
  (invert [this] (apply str (reverse this))))

(extend-type clojure.lang.PersistentList
  IInvertible
  (invert [this] (reverse this)))

(extend-type clojure.lang.PersistentVector
  IInvertible
  (invert [this] (into [] (reverse this))))
```

Agora, é hora de testarmos a implementação do nosso protocolo:

```
(invert "abc")
;; => "cba"

(invert 0)
;; => 0

(invert '(1 2 3))
;; => (3 2 1)

(invert [1 2 3])
;; => [3 2 1]
```

Em comparação, **extend-protocol** faz o inverso; dado um protocolo, ele adiciona uma implementação para vários tipos. A sintaxe é assim:

```
(extend-protocol ProtocolA
  TypeA
  (function-from-protocol-a [this]
    ;; implementação aqui
    )

  TypeB
  (function-from-protocol-a [this]
    ;; implementação aqui
    ))
```

Então, o exemplo anterior poderia ser escrito também dessa maneira:

```
(extend-protocol IInvertible
  String
  (invert [this] (apply str (reverse this)))

  clojure.lang.PersistentList
  (invert [this] (reverse this))

  clojure.lang.PersistentVector
  (invert [this] (into [] (reverse this))))
```

Introspeção usando Protocolos

Clojure vem com algumas funções úteis que permitem verificar se algo implementa um protocolo usando a função **`satisfies?`**. O propósito dessa função é determinar em runtime se algum objeto satisfaz um protocolo.

Se definirmos um protocolo Baz e um record Foo que implementa esse protocolo, podemos verificar isso com a função **`satisfies?`**:

```
(defprotocol Baz
             (bar [this]))

(defrecord Foo [some-name]
           Baz
           (bar [this] (str "Hello, " some-name)))


(satisfies? Baz (->Foo "Alguém"))
;; => true
```

4.12.2. Multimethods

Protocolos resolvem um problema muito comum de polimorfismo: despachar uma função por tipo. Mas em algumas circunstâncias, essa abordagem pode ser limitante. E são nesses casos que usamos *multimethods*

Esses *multimethods* não estão limitados apenas por tipos; ao invés, eles oferecem a possibilidade de chamar funções de acordo com tipos de múltiplos argumentos ou por valor. Eles também permitem criarmos hierarquias. Além disso, como protocolos, *multimethods* são um sistema aberto de extensão que pode ser usado por outras bibliotecas.

As funções básicas dos *multimethods* são **`defmulti`** e **`defmethod`**. O **`defmulti`** é utilizado para criar uma função *dispatch* inicial.

```
(defmulti say-hello
  "Uma função polimórfica que retorna uma mensagem de
  olá dependendo da chave :locale, sendo o padrão `:en`"
```

```
  (fn [param] (:locale param))
  :default :en)
```

A função anônima definida dentro do **`defmulti`** é chamada de função *dispatch*. Ela é chamada toda vez que a função **`say-hello`** é chamada e deveria retornar algum tipo de valor que será usado para selecionar a implementação correta. No nosso exemplo, ela retorna o conteúdo da chave **`:locale`** do primeiro argumento.

E finalmente, devemos adicionar algumas implementações. Isso é feito através do **`defmethod`**:

```
(defmethod say-hello :en
  [person]
  (str "Hello " (:name person "Anonymous")))

(defmethod say-hello :es
  [person]
  (str "Hola " (:name person "Anónimo")))
```

Então, se executamos essa função com um mapa contendo a chave **`:local`** e opcionalmente a chave **`:name`**, o *multimethod* vai primeiro chamar a função **`dispatch`** para determinar o valor de **`dispatch`**, e então irá procurar pela implementação daquele valor. Se uma implementação é encontrada, a respectiva função será executada. Se não, o **`dispatch`** vai procurar pela implementação padrão (se houver) e executará ela.

```
(say-hello {:locale :es})
;; => "Hola Anónimo"

(say-hello {:locale :en :name "Ciri"})
;; => "Hello Ciri"

(say-hello {:locale :fr})
;; => "Hello Anonymous"
```

Se uma implementação padrão não for especificada, uma exceção será gerada, notificando que o valor não possui uma implementação para aquele **`multimethod`**.

4.12.3. Hierarquias

Hierarquias são o modo que o *Clojure* fornece para construir relações que seu domínio possa precisar. Hierarquias são definidas em termos de relaçãoes entre objetos nomeados como símbolos, *keywords* ou tipos.

Hierarquias podem ser definidas globalmente ou localmente, de acordo com suas necessidades. Você pode extender uma hierarquia em qualquer *namespace*, não apenas onde ela é definida.

O *namespace* global é mais limitado, por boas razões. *Keywords* e símbolos que não são qualificados com um *namespace* não podem ser usados em uma hierarquia global. Esse comportamento previne situações onde duas ou mais bibliotecas tentam utilizar o mesmo símbolo com semânticas diferentes.

Definindo uma hierarquia

Uma relação dentro de uma hierarquia deveria ser definida usando a função **`derive`**:

```
(derive ::circle ::shape)
(derive ::box ::shape)
```

Nos apenas definimos um conjunto de relações entre *keywords* qualificadas. Nesse caso, o **`::circle`** é filho de **`::shape`** e **`::box`** também é filho de **`::shape`**.

> A sintaxe **`::circle`** é um "apelido" para **`:current.ns/circle`**. Então, se você estiver executando isso na REPL, **`::circle`** será transformado em **`user/circle`**.

Hierarquias e introspecção

Clojure vem com uma série de funções que permite verificar relações relacionadas a hierarquias em runtime. Essas funções são: **`isa?`**, **`ancestors`** e **`descendants`**.

Vejamos um exemplo de como podemos usá-las com a hierarquia definida no exemplo anterior:

```
(ancestors ::box)
```

```
;; => #{:cljs.user/shape}

(descendants ::shape)
;; => #{:cljs.user/circle :cljs.user/box}

(isa? ::box ::shape)
;; => true

(isa? ::rect ::shape)
;; => false
```

Hierarquias definidas localmente

Como mencionamos anteriormente, em *Clojure* voc6e pode definir uma hierarquia local. Isso pode ser feito com a função **`make-hierarchy`**. Aqui está o exemplo anterior com uma hierarquia local.

```
(def h (-> (make-hierarchy)
           (derive :box :shape)
           (derive :circle :shape)))
```

Agora podemos fazer a mesma instrospecção com a hierarquia local:

```
(isa? h :box :shape)
;; => true

(isa? :box :shape)
;; => false
```

Como você pode observar, em uma hierarquia local podemos usar *keywords* sem *namespaces*, e se executamos **`isa?`** sem passar uma hierarquia local, false é retornado, como esperado.

Hierarquias em *multimethods*

Uma das grandes vantagens de hierarquias é que elas funcionam junto com *multimethods*. Isso é porque os *multimethods* utilizam *by default* a função **`isa?`** como último passo do *dispatch*.

Vejamos um exemplo para compreendermos isso. Primeiro vamos definir um *multimethod*:

```
(defmulti stringify-shape
  "A function that prints a human readable representation
  of a shape keyword."
  identity
  :hierarchy #'h)
```

Com o parâmetro nomeado **`:hierarchy`**, nos indicamos ao *multimethod* que queremos usar essa hierarquia; se não for especificado, a hierarquia global será usada.

Em seguida, definimos a implementação de nosso *mulitmethod*:

```
(defmethod stringify-shape :box
  [_]
  "A box shape")

(defmethod stringify-shape :shape
  [_]
  "A generic shape")

(defmethod stringify-shape :default
  [_]
  "Unexpected object")
```

Agora, vejamos o que acontece se executarmos essa função com **`:box`**:

```
(stringify-shape :box)
;; => "A box shape"
```

Tudo funciona como esperado; o *multimethod* executa uma *matching* direto na implementação do parâmetro dado. Em seguida, vejamos o que acontece se executarmos a mesma função com o parâmetro **`:circle`** que não possui uma implementação "direta":

```
(stringify-shape :circle)
```

```
;; => "A generic shape"
```

O **multimethod** automaticamente resolve qual implementação usar, utilizando a hierarquia fornecidade. Como **:circle** é um descendente de **:shape**, a implementação do *multimethod* para **:shape** é executada.

Finalmente, se você fornecer uma *keyword* que não faz parte da hierarquia, você obtém a implementação default:

```
(stringify-shape :triangle)
;; => "Unexpected object"
```

4.13. Tipos de dados

Até agora, nós usamos mapas, **sets**, listas e vetores para representar nossos dados. E na maioria dos casos, essa é um abordagem muito boa. Mas algumas vezes precisamos definir nossos próprios tipos, e nesse livro vamos chamá-los de *data types*.

Um *data type* fornece o seguinte:

- Um tipo único fornecido pelo host, seja nomeado ou anônimo.
- A habilidade de implementar protocolos.
- Estrutura explicitamente declarada utilizando campos e *closures*.
- Comportamento similar a mapas (via *records*, que veremos em seguida)

4.13.1. Deftype

A opção mais baixo nível para criar seus próprios tipos em *Clojure* é a macro **deftype**. Como demonstração, vamos definir um tipo chamado **User**:

```
(deftype User [firstname lastname])
```

Uma vez que o tipo foi definido, podemos criar uma instância do nosso **User**. No exemplo a seguir, o **.** depois do **User** indica que estamos chamando um construtor.

```
(def person (User. "Triss" "Merigold"))
```

Os campos do **User** podem ser acessados usando a notação prefixada com **.**:

```
(.-firstname person)
;; => "Triss"
```

Tipos definidos com **deftype** (e **defrecord**, que veremos depois0 criam uma classe na linguagem host associada com o *namespace* atual. Por conveniência, *Clojure* também define uma função construtora chamada **→User** que pode ser importada utilizando a diretiva **:require**.

Nós, pessoalmente, não gostamos desse tipo de função, e prefirimos definir nossos próprios construtores com nomes mais idiomáticos.

```
(defn make-user
  [firstname lastname]
  (User. firstname lastname))
```

Usamos essa função no nosso código ao invés de **→User**.

4.13.2. Defrecord

Um *record* é uma abstração um pouco mais alto nível para definirmos tipos em *Clojure* e deveria ser favorecido em relação a **deftype**.

Como sabemos, *Clojure* tende a usar tipos de dados como **map**, mas na maioria dos casos precisamos tipos vinculados a um nome para representar entidades na nossa aplicação. Para isso usamos os *records*:

Um *record* é um tipo de dado que implementa a interface do mapa e portanto pode ser utilizado como qualquer outro mapa. E como *records* são tipos, eles suportam polimorfismo baseado em tipos através de protocolos.

Em resumo: com *records* temos o melhor dos dois mundos, mapas que podem participar em diferentes abstrações.

Vamos começar definindo o tipo **`User`** mas utilizando *records*:

```
(defrecord User [firstname lastname])
```

Parece muito com a sintaxe de **`deftype`**, e na verdade, **`defrecord`** utililiza o **`deftype`** por baixo dos panos como um função de baixo nível para definir tipos.

Agora, perceba a diferença para acessar campos:

```
(def person (User. "Yennefer" "of Vengerberg"))

(:firstname person)
;; => "Yennefer"

(get person :firstname)
;; => "Yennefer"
```

Como mencionamos, *records* são mapas e agem como tais:

```
(map? person)
;; => true
```

E como mapas, eles suportam campos extras que não definimos inicialmente:

```
(def person2 (assoc person :age 92))

(:age person2)
;; => 92
```

Como podemos ver, a função **`assoc`** funciona como esperado e retorna uma instância do mesmo tipo, porém com um novo par chave-valor. Porém tome cuidado com o **`dissoc`**! Seu comportamento com **`records`** é um pouco diferente do que com mapas; ele vai retornar um novo *record* se o campo desassociado for opcional, se não ele vai retornar um mapa normal.

Outra diferença com mapas é que os *records* não agem como funções:

```
(def plain-person {:firstname "Yennefer", :lastname "of Vengerberg"})

(plain-person :firstname)
;; => "Yennefer"

(person :firstname)
;; => class user.User cannot be cast to class clojure.lang.IFn
```

Por conveniência, a macro **defrecord**, como **deftype**, expõe uma função **→User**, assim como uma função construtora adicional **map→User**. Mas mantemos nossa opinião de criar nossos próprios construtores ao invés de usar os outros. Mas como eles existem, vamos dar uma olhada em como usá-los:

```
(def cirilla (->User "Cirilla" "Fiona"))
(def yen (map->User {:firstname "Yennefer"
                     :lastname "of Vengerberg"}))
```

4.13.3. Implementando *protocols*

Tanto **deftype** quanto **defrecord** permitem a implementação de protocolos "inline" como fizemos com *extend-type* e *extend-protocol*. Vamos definir um protocolo como exemplo

```
(defprotocol IUser
  "A common abstraction for working with user types."
  (full-name [_] "Get the full name of the user."))
```

Agora, podemos definir um tipo que implementando nossa abstração **IUser**:

```
(defrecord User [firstname lastname]
  IUser
  (full-name [_]
    (str firstname " " lastname)))

;; Create an instance.
(def user (User. "Yennefer" "of Vengerberg"))
```

```
(full-name user)
;; => "Yennefer of Vengerberg"
```

4.13.4. Reify (Materializar)

A macro **`reify`** é um construtor ad-hoc que podemos usar para criar objectos que implementam uma interface sem definir um tipo. Porém, com **`reify`** não possuímos campos acessíveis como em **`deftype`** e **`defrecord`**.

No exemplo abaixo, podemos criar um objeto que implementa a abstração **`IUser`**:

```
(defn user
  [firstname lastname]
  (reify
    IUser
    (full-name [_]
      (str firstname " " lastname))))

(def yen (user "Yennefer" "of Vengerberg"))
(full-name yen)
;; => "Yennefer of Vengerberg"
```

4.14. Interoperabilidade com a linguagem hospedeira

Clojure foi feita para ser uma linguagem "convidada". Isso significa que o seu design foi pensando para trabalhar bem em cima do eco-sistema já existente da JVM.

4.14.1. Os tipos

Clojure tira vantagens do tipos fornecidos pela plataforma. Essa é uma lista (provavelmente incompleta) de tipos que o *Clojure* utiliza do java. *ClojureScript*, unlike what you might expect, tries to take advantage of every type that the platform provides. This is a (perhaps incomplete) list of things that *ClojureScript* inherits and reuses from the underlying platform:

- Strings em *Clojure* são **Strings** do Java.
- *Clojure* utiliza os tipos numéricos primitivos do Java além de **`BigInteger`** e **`BigDecimal`**.

- *Clojure* **`nil`** tem o mesmo valor que o **`null`** em Java..
- As expressões regulares em *Clojure* são instâncias da class **`java.util.regex.Pattern`**.
- *Clojure* não é interpretada, é sempre compilada para Java Bytecode.
- *Clojure* permite chamada das APIs da plataforma usando a mesma semântica já existente.

Em cima disso, *Clojure* constroí suas próprias abstrações e tipos que não existem na plataforma, como Vectors, Maps, Sets, e outros que foram explicados nas seções anteriores.

4.14.2. Interagindo com tipos da plataforma

Clojure vem com um conjunto de formas especiais que nos permite interagir com tipos da plataformas como por exemplo chamar métodos de objetos, criar novas instâncias, e acessar propriedades de objetos.

Acesso a plataforma

Clojure já possui vários símbolos importados do Java que podem ser usados, como por exemplo **`Integer`**, **`Long`**, **`String`**, etc:

```
(Integer. 23)
;; => 23
(Integer/parseInt "23")
;; => 23
```

Criando novas instâncias

Clojure fornece dois modos de criar uma nova instância de uma classe Java:

Usando a forma special **`new`**:

```
(import 'java.io.File)
(new File "some_file.txt")
```

Usando a forma especial `.`: Using the `.` special form

```
(import 'java.io.File)
(File. "some_file.txt")
```

A última opção é o modo recomendado de criar instâncias. Não sabemos de nenhuma diferença real entre as duas formas, porém na comunidade *Clojure*, a última é mais utilizada.

Chamando métodos dos objetos

Para chamar métodos de algum objeto, ao invés de fazermos como fazemos em Java, como **`obj.method()`**, o método vem primeiro prefixado com `.` e o objeto como o primeiro argumento, seguido pelos argumentos do método. To invoke methods of some object instance, as opposed to how it is done in JavaScript (e.g., **`obj.method()`**, the method name comes first like any other standard function in Lisp languages but with a little variation: the function name starts with special form `..`

Let's see how we can call the **`.test()`** method of a regexp instance:

```
(def java-map (java.util.HashMap.))
(.put java-map "a 1)
```

Acessar e modificar propriedades de um objeto

Acesso a uma propriedade de um objeto é muito similar a chamar um método. A unica diferença é que ao invés de ser prefixado com `.` prefixamos o nome da propriedade com **`.-`**.

```
(def point (java.awt.Point. 1 2))
(.-y point)
;; => 2
```

Para modificar uma propriedade utilizamos a forma especial **`set!`** passando um um lista que realiza acesso a propriedade e o novo valor:

```
(def point (java.awt.Point. 1 2))
(set! (.-y point) 3)
;; => 3

(.-y point)
;; => 3
```

Arrays

Algumas funções do Java aceitam somente arrays. Para isso *Clojure* fornece algumas funções que permitem que criarmos arrays do Java a partir de coleções do *Clojure*.

```
(def path ["username" "dev" "clojure"])
(java.nio.file.Paths/get "/Users" path)
;; => class clojure.lang.PersistentVector cannot be cast to class
 [Ljava.lang.String;

(java.nio.file.Paths/get "/Users" (into-array
 ["username" "dev" "clojure"]))
;; #object[sun.nio.fs.UnixPath 0x1f193686 "/Users/username/dev/clojure"]
```

Em *Clojure*, arrays também funcionam bem com a abstração de sequências, então podemos usar funções como:

```
(count (into-array ["username" "dev" "clojure"]))
;; => 10
```

4.15. Gerenciamento de estado

Nós aprendemos que uma das ideias fundamentais do *Clojure* é a imutabilidade. Tanto valores escalres quanto coleções são imutáveis em *Clojure*.

Imutabilidade possui muitas propriedades interessantes, mas algumas vezes precisamos modelar valores que mudam ao longo do tempo. Como podemos fazer isso se não podemos mutar as estruturas de dado?

4.15.1. Vars

Vars podem ser redefinidas como desejarmos dentro de um *namespace*, mas não existe jeito de saber **quando** elas mudam. A inabilidade de redefinir *vars* de outros *namespaces* é um pouco limitante; além disso, se estamos modificando estado, provavelmente estamos interessados em saber quando isso ocorre.

4.15.2. Atoms

Clojure nos da o tipo **`Atom`**, que é um objeto contendo um valor que pode ser alterado quando quisermos. Além de alterar seu valor, eles também suportam serem observados através de funções *watcher*(observadores) que podem ser vinculados e desvinculados do átomo e validações para garantir que o valor associado é sempre válido.

Se queremos modelar uma identidade correspondende a um pessoa chamada Ciri, podemos colocar uma estrutura imutável dentro de um átomo. Note que podemos obter o valor do átomo com a função **`deref`** ou usando a forma especial mais curta `@`:

```
(def ciri (atom {:name "Cirilla" :lastname "Fiona" :age 20}))
;; #<Atom: {:name "Cirilla", :lastname "Fiona", :age 20}>

(deref ciri)
;; {:name "Cirilla", :lastname "Fiona", :age 20}

@ciri
;; {:name "Cirilla", :lastname "Fiona", :age 20}
```

Podemos utilizar a função **`swap!`** para alterar seu valor com uma função, como na função update. Como o aniversário da Ciri é hoje, vamos incrementar sua idade:

```
(swap! ciri update :age inc)
;; {:name "Cirilla", :lastname "Fiona", :age 21}

@ciri
;; {:name "Cirilla", :lastname "Fiona", :age 21}
```

A função **`reset!`** substitui o valor presenter no átomo por um novo:

```
(reset! ciri {:name "Cirilla", :lastname "Fiona", :age 22})
;; {:name "Cirilla", :lastname "Fiona", :age 22}

@ciri
;; {:name "Cirilla", :lastname "Fiona", :age 22}
```

Observabilidade

Nós podemos adicionar e remover funções *watchers*(observadoras) de átomos. Quando um átomo é alterado através de **`swap!`** ou **`reset!`**, todos os *watchers* de um átomo são chamados. *Watchers* podem ser adicionados através da função **`add-watch`**. Note que cada *watcher* tem uma chave associada com ele (**`:logger`** por exemplo) que usada para remover o *watch* do átomo.

```
(def a (atom))

(add-watch a :logger (fn [key the-atom old-value new-value]
                       (println "Key:" key "Old:" old-value "New:" new-
value)))

(reset! a 42)
;; Key: :logger Old: nil New: 42
;; => 42

(swap! a inc)
;; Key: :logger Old: 42 New: 43
;; => 43

(remove-watch a :logger)
```

4.15.3. Volatiles

Volatiles, como átomos, são objetos que possuem um valor que pode ser alterado. Contudo, eles não provem a possibilidade de validação observação que átomos possuem. Isso faz com que eles sejam mais performáticos e mais adequados para serem usados dentro de funções que não precisam de observação ou validação.

API deles é muito parecida com a dos átomos. Eles podem ser *dereferenced* para obtermos o valor deles e suportam mudanças através de **`vswap!`** e **`vreset!`**:

```
(def ciri (volatile! {:name "Cirilla" :lastname "Fiona" :age 20}))
;; #<Volatile: {:name "Cirilla", :lastname "Fiona", :age 20}>

(volatile? ciri)
;; => true

(deref ciri)
;; {:name "Cirilla", :lastname "Fiona", :age 20}

(vswap! ciri update :age inc)
;; {:name "Cirilla", :lastname "Fiona", :age 21}

(vreset! ciri {:name "Cirilla", :lastname "Fiona", :age 22})
;; {:name "Cirilla", :lastname "Fiona", :age 22}
```

Note que outra diferença com átomos é o construtor que é **`volatile!`**.

Chapter 5. Agradecimentos

Agradecimentos especiais aos autores originais do livro que fizeram o grande trabalho de escrevê-lo e disponibilizá-lo sob um licensa que permita sua tradução e adaptação.

Special thanks to:

- Andrey Antukh niwi@niwi.nz[1]
- Alejandro Gómez alejandro@dialelo.com[2]

Who wrote the original book and to all others involved in the elaboration of that book. - Andrey Antukh niwi@niwi.nz[3] - Alejandro Gómez alejandro@dialelo.com[4]

1 mailto:niwi@niwi.nz
2 mailto:alejandro@dialelo.com
3 mailto:niwi@niwi.nz
4 mailto:alejandro@dialelo.com

Chapter 6. Recursos adicionais

Aqui está uma lista de mais recursos sobre *Clojure*. A medida que acharmos ou produzirmos mais conteúdo em português, iremos adicionar aqui.

- Clojure Cheatsheet[1]: uma referência em inglês da linguagem *Clojure*.
- Clojure Community-docs [2]: documentação sobre a linguagem produzida pela comunidade.

[1] https://clojure.org/api/cheatsheet/
[2] https://clojuredocs.org/