Extracted from:

Programming Clojure, Third Edition

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming Clojure

Third Edition

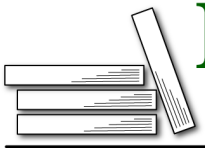
Alex Miller
with Stuart Halloway
and Aaron Bedra

edited by Jacquelyn Carter

Programming Clojure, Third Edition

Alex Miller
with Stuart Halloway
and Aaron Bedra

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Copy Editor: Paula Robertson
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-246-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2018

Clojure offers great power through functional style, concurrency support, and clean Java interop. But before you can appreciate all these features, you have to start with the language basics. Clojure is very expressive, and this chapter covers many concepts quite quickly. Don't worry if you don't understand every detail; we'll revisit these topics in more detail in later chapters. If possible, bring up a REPL and follow along with the examples as you read.

We'll start by looking at how to read and understand Clojure code, introducing the key parts of Clojure syntax. If your background is primarily in imperative languages, this tour may seem to be missing key language constructs, such as variables and for loops. Don't worry, you'll soon learn how to work in new ways that don't require them.

Reading Clojure

In this section, we'll cover many of the key parts of Clojure syntax and how they're used to form Clojure programs. In Clojure, there are no statements, only expressions that can be nested in mostly arbitrary ways. When evaluated, every expression returns a value that's used in the parent expression. This is a simple model, yet sufficient to cover everything in Clojure.

Numbers

To begin our exploration, let's consider a simple arithmetic expression as expressed in Clojure:

```
(+ 2 3)
-> 5
```

All Clojure code is made up of expressions, and every expression, when evaluated, returns a value. In Clojure, parentheses are used to indicate a list, in this example, a list containing the symbol + and the numbers 2 and 3.

Clojure's runtime evaluates lists as function calls. The first element (+ in this example) is always treated as the operation with the remaining elements treated as arguments. The style of placing the function first is called *prefix notation*, as opposed to the more familiar *infix notation*, 2 + 3.

A practical advantage of prefix notation is that you can easily extend it for arbitrary numbers of arguments:

```
(+ 1 2 3 4)
-> 10
```

Even the degenerate case of no arguments works as you'd expect, returning zero. This helps to eliminate special-case logic for boundary conditions:

```
(+)
-> 0
```

Many mathematical and comparison operators have the names and semantics that you'd expect from other programming languages. Addition, subtraction, multiplication, comparison, and equality all work as you would expect:

```
(- 10 5)
-> 5

(* 3 10 10)
-> 300

(> 5 2)
-> true

(>= 5 5)
-> true

(< 5 2)
-> false

(= 5 2)
-> false
```

Division may surprise you:

```
(/ 22 7)
-> 22/7
```

As you can see, Clojure has a built-in ratio type.

If what you actually want is decimal division, use a floating-point literal for the dividend:

```
(/ 22.0 7)
-> 3.142857142857143
```

If you want to stick to integers, you can get the integer quotient and remainder with `quot` and `rem`:

```
(quot 22 7)
-> 3

(rem 22 7)
-> 1
```

If you need to do arbitrary-precision, floating-point math, append `M` to a number to create a `BigDecimal` literal:

```
(+ 1 (/ 0.00001 1000000000000000000))
-> 1.0

(+ 1 (/ 0.00001M 1000000000000000000))
-> 1.000000000000000000000001M
```

For arbitrary-precision integers, you can append N to create a BigInt literal:

```
(* 1000N 1000 1000 1000 1000 1000 1000)
-> 1000000000000000000000N
```

Notice that only one BigInt literal is needed and is contagious to the entire calculation.

Next, let's see how we can group values using Clojure collections.

Symbols

Forms such as +, concat, and java.lang.String are called *symbols* and are used to name things. For example, + names the function that adds things together. Symbols name all sorts of things in Clojure:

- Functions like str and concat
- "Operators" like + and -, which are, after all, just functions
- Java classes like java.lang.String and java.util.Random
- Namespaces like clojure.core and Java packages like java.lang

Symbols cannot start with a number but can consist of alphanumeric characters, as well as +, -, *, /, !, ?, ., _, and '. The list of legal symbol characters is a minimum set that Clojure promises to support. You should stick to these characters in your own code, but do not assume the list is exhaustive. Clojure can use other, undocumented characters in symbols that it employs internally and may add more legal symbol characters in the future. See Clojure's online documentation[1] for updates to the list of legal symbol characters.

Clojure treats / and . specially in order to support namespaces; see *Namespaces, on page ?* for details.

Collections

Clojure provides four primary collection types—lists, vectors, sets, and maps. All Clojure collections are heterogeneous (can hold any type of data) and are compared for equality based on their contents. The four Clojure collection types are used in combination to create larger composite data structures.

First let's consider vectors, which are sequential, indexed collections. We can create a vector of the numbers 1, 2, and 3 using the following:

```
[1 2 3]
-> [1 2 3]
```

1. https://clojure.org/reference/reader

Lists are sequential collections stored as a linked list. A list is printed as (1 2 3), but we can't create a literal list at the REPL like we can with vectors. As we discussed in the previous section, Clojure function calls are represented as lists and evaluated by invoking the first element as the function. Thus (1 2 3) would be interpreted as invoking the function 1 with the arguments 2 and 3.

If we want a list to be read and interpreted as data (not evaluated like a function call), we can use the quote special form:

```
(quote (1 2 3))
-> (1 2 3)
```

Quoting also has a *reader macro* form (') understood by the reader. Reader macros are abbreviations of longer list forms and are used as shortcuts to improve readability. We'll see more of these as we explore. Here's how the quote looks in the shorter form:

```
'(1 2 3)
-> (1 2 3)
```

Sets are unordered collections that do not contain duplicates:

```
#{1 2 3 5}
-> #{1 3 2 5}
```

Because sets are unordered, you may see the elements printed in a different order than the original literal, and you should not expect any particular order. Sets are a good choice when you want fast addition and removal of elements and the ability to quickly check for whether a set contains a value.

Finally, Clojure maps are collections of key/value pairs. You can use a map literal to create a lookup table for the inventors of programming languages:

```
{"Lisp" "McCarthy" "Clojure" "Hickey"}
-> {"Lisp" "McCarthy", "Clojure" "Hickey"}
```

The key "Lisp" is associated with the value "McCarthy" and the key "Clojure" is associated with the value "Hickey". Like sets, maps are unordered, and the key/value pairs may be printed in an order different than the original map literal.

You may have noticed that the printed version lists a comma between the two key/value pairs. In Clojure, commas are whitespace and you're free to use them as an optional delimiter if you find it improves readability:

```
{"Lisp" "McCarthy", "Clojure" "Hickey"}
-> {"Lisp" "McCarthy", "Clojure" "Hickey"}
```

Any Clojure data structure can be a key in a map. However, the most common key type is the Clojure keyword.

A *keyword* is like a symbol, except that keywords begin with a colon (:). Keywords resolve to themselves:

```
:foo
-> :foo
```

The fact that keywords resolve to themselves makes keywords useful as keys. You could redefine the inventors map using keywords as keys: {:Lisp "McCarthy" :Clojure "Hickey"}

If several maps have keys in common, you can leverage this by creating a record with defrecord:

```
(defrecord name [arguments])
```

For example, consider using the defrecord to create a Book record:

```
(defrecord Book [title author])
-> user.Book
```

Then, you can instantiate that record with the ->Book constructor function:

```
(->Book "title" "author")
```

Once you instantiate a Book, it behaves almost like any other map. We will learn more about when and how to use records in Chapter 7, *Protocols and Datatypes*, on page ?.

Strings and Characters

Strings are another kind of form. They are delimited by double quotes and are allowed to span multiple lines. Clojure strings reuse the Java String implementation.

```
"This is a\nmultiline string"
-> "This is a\nmultiline string"

"This is also
a multiline string"
-> "This is also\na multiline string"
```

As you can see, the REPL always shows string literals with escaped newlines. If you actually print a multiline string, it will print on multiple lines:

```
(println "another\nmultiline\nstring")
| another
| multiline
| string
-> nil
```

Perhaps the most common string function you'll use is str, which takes any number of objects, converts them to strings, and concatenates the results into a single string. Any nils passed to str are ignored:

```
(str 1 2 nil 3)
-> "123"
```

Clojure characters are also Java characters. Their literal syntax is \{letter}, where letter can be a letter, or in a few special cases, the name of a character: backspace, formfeed, newline, return, space, or tab:

```
(str \h \e \y \space \y \o \u)
-> "hey you"
```

Booleans and nil

Clojure's rules for Booleans are easy to understand:

- true is true, and false is false.
- In addition to false, nil evaluates to false when used in a Boolean context.
- Other than false and nil, *everything else* evaluates to true in a Boolean context.

Note that true, false, and nil follow the rules for symbols but are read as other special values (either a Boolean or nil). These are the only special-case tokens like this in Clojure—anything else symbol-like is read as a symbol.

The empty list is not false in Clojure:

```
;      (if part)    (else part)
(if () "() is true" "() is false")
-> "() is true"
```

Zero is not false in Clojure, either:

```
;     (if part)      (else part)
(if 0 "Zero is true" "Zero is false")
-> "Zero is true"
```

A *predicate* is a function that returns either true or false. In Clojure, it's common to name predicates with a trailing question mark, for example, true?, false?, nil?, and zero?:

```
(true?  expr)
(false? expr)
(nil?   expr)
(zero?  expr)
```

true? tests whether a value is exactly the true value, *not* whether the value evaluates to true in a Boolean context. The only thing that's true? is true itself:

```
(true? true)
-> true
```

```
(true? "foo")
-> false
```

nil? and false? work the same way. Only nil is nil?, and only false is false?.

zero? works with any numeric type, returning true if it's zero:

```
(zero? 0.0)
-> true
```

```
(zero? (/ 22 7))
-> false
```

There are many more predicates in Clojure—go to the REPL and type:

```
(find-doc #"\?$")
```

The find-doc function is a REPL facility (included in the clojure.repl namespace) that searches all docstrings matching either a string or a regular expression. The syntax used here #"\?$" is a literal regular expression. Clojure uses Java's built-in regular expression library and is equivalent to a compiled Java Pattern. Clojure provides a set of functions designed for using regular expressions to find and/or replace matches in a string.