Extracted from:

Programming Clojure, Third Edition

This PDF file contains pages extracted from *Programming Clojure, Third Edition*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2018 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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*.

Copyright © 2018 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

Functional programming (FP) is a big topic, not to be learned in 21 days[1] or in a single chapter of a book. Nevertheless, you can reach a first level of effectiveness using lazy and recursive techniques in Clojure fairly quickly, and that is what we'll accomplish this chapter.

We'll start with a quick overview of FP terms and concepts and an introduction to the guidelines of Clojure FP that we'll refer to throughout the chapter. Next we'll experience the power of lazy sequences by working through a series of implementations of the Fibonacci numbers. As cool as lazy sequences are, you rarely need to construct them yourself, and we'll see better ways to recast problems to solve them directly with the sequence library.

We'll close with some advanced techniques and see some scenarios where eager transformations have advantages over lazy sequences.

Functional Programming Concepts

Functional programming leads to code that is easier to write, read, test, and reuse. Here's how it works.

Pure Functions

Programs are built out of *pure functions*. A pure function has no *side effects*; that is, it doesn't depend on anything but its arguments, and its only influence on the outside world is through its return value.

Mathematical functions are pure functions. Two plus two is four, no matter where or when you ask. Also, asking doesn't *do* anything other than return the answer.

Program output is decidedly *impure*. For example, when you println, you change the outside world by pushing data onto an output stream. Also, the results of println depend on state outside the function: the standard output stream might be redirected, closed, or broken.

If you start writing pure functions, you'll quickly realize that pure functions and *immutable* data go hand in hand. Consider the following mystery function:

```
(defn mystery [input]
  (if input data-1 data-2))
```

If mystery is a pure function, then regardless of what it does, data-1 and data-2 have to be immutable! Otherwise, changes to the data would cause the function to return different values for the same input.

1. http://norvig.com/21-days.html

A single piece of mutable data can ruin the game, rendering an entire call chain of functions impure. So, once you make a commitment to writing pure functions, you end up using immutable data in large sections of your application.

Persistent Data Structures

Immutable data is critical to Clojure's approach to both FP and state. On the FP side, pure functions cannot have side effects, such as updating the state of a mutable object. On the state side, Clojure's reference types require immutable data structures to implement their concurrency guarantees.

The fly in the ointment is performance. When all data is immutable, "update" translates into "create a copy of the original data, plus my changes." This will use up memory quickly! Imagine that you have an address book that takes up 5 MB of memory. Then, you make five small updates. With a mutable address book, you are still consuming about 5 MB of memory. But if you have to copy the whole address book for each update, then an immutable version would balloon to 25 MB!

Clojure's data structures don't take this naive "copy everything" approach. Instead, all Clojure data structures are *persistent*. In this context, persistent means that the data structures preserve old copies of themselves by efficiently *sharing structure* between older and newer versions.

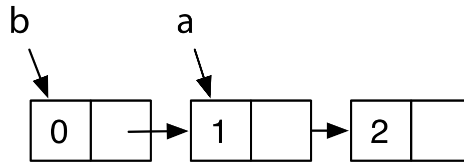
Structural sharing is easiest to visualize with a list. Consider list `a` with two elements:

```
(def a '(1 2))
-> #'user/a
```

Then from a, you can create a b with an additional element added:

```
(def b (cons 0 a))
-> #'user/b
```

b can reuse all of a's structure, rather than have its own private copy:



All of Clojure's data structures share structure where possible. For structures other than simple lists, the mechanics are more complex, of course. If you're interested in the details, check out the following articles:

- "Ideal Hash Trees"[2] by Phil Bagwell
- "Understanding Clojure's PersistentVector Implementation"[3] by Karl Krukow

Laziness and Recursion

Functional programs make heavy use of *recursion* and *laziness*. A recursion occurs when a function calls itself, either directly or indirectly. With laziness, an expression's evaluation is postponed until it's actually needed. Evaluating a lazy expression is called *realizing* the expression.

In Clojure, functions and expressions are not lazy. However, sequences *are* generally lazy. Because so much Clojure programming is sequence manipulation, you get many of the benefits of a fully lazy language. In particular, you can build complex expressions using lazy sequences and then "pay" only for the elements you actually need.

Lazy techniques imply pure functions. You never have to worry about when to call a pure function, since it always returns the same thing. Impure functions, on the other hand, do not play well with lazy techniques. As a programmer, you must explicitly control when an impure function is called, because if you call it at some other time, it may behave differently!

Referential Transparency

Laziness depends on the ability to replace a function call with its result at any time. Functions that have this ability are called *referentially transparent*,

2. http://lampwww.epfl.ch/papers/idealhashtrees.pdf
3. http://tinyurl.com/clojure-persistent-vector

• Click HERE to purchase this book now. discuss

because calls to such functions can be replaced without affecting the behavior of the program. In addition to laziness, referentially transparent functions can also benefit from the following:

- *Memoization*, automatic caching of results
- Automatic *parallelization*, moving function evaluation to another processor or machine

Pure functions are referentially transparent *by definition*. Most other functions are *not* referentially transparent, and those that are must be proven safe by code review.

Benefits of FP

Well, that is a lot of terminology, and we promised it would make your code easier to write, read, test, and compose. Here's how.

You'll find functional code easier to *write* because the relevant information is right in front of you, in a function's argument list. You don't have to worry about global scope, session scope, application scope, or thread scope. Functional code is easier to *read* for exactly the same reason.

Code that is easier to read and write is going to be easier to test, but functional code brings an additional benefit for testing. As projects get large, it often takes a lot of effort to set up the right environment to execute a test. This is much less of a problem with functional code, because there *is no relevant environment* beyond the function's arguments.

Functional code improves *reuse*. To reuse code, you must be able to do the following:

- Find and understand a piece of useful code.
- Compose the reusable code with other code.

The readability of functional code helps you find and understand the functions you need, but the benefit for *composing* code is even more compelling.

Composability is a hard problem. For years, programmers have used *encapsulation* to try to create composable code. Encapsulation creates a firewall, providing access to data only through a public API.

Encapsulation helps, but it's nowhere near enough. Even with encapsulated objects, there are far too many surprising interactions when you try to compose entire systems. The problem is those darn side effects. *Impure functions* violate encapsulation, because they let the outside world reach in (invisibly!) and change the behavior of your code. Pure functions, on the other hand, are

truly encapsulated and composable. Put them anywhere you want in a system, and they will always behave in the same way.

Guidelines for Use

Clojure takes a unique approach to FP that strikes a balance between academic purity and the reality of running well on the JVM. That means there's a lot to learn all at once. But fear not. If you're new to FP, the following guidelines will help on your initial steps toward FP mastery, Clojure-style:

1. Avoid direct recursion. The JVM can't optimize recursive calls, and Clojure programs that recurse will blow their stack.
2. Use `recur` when you're producing scalar values or small, fixed sequences. Clojure *will* optimize calls that use an explicit `recur`.
3. When producing large or variable-sized sequences, always be lazy. (Do *not* recur.) Then, your callers can consume just the part of the sequence they actually need.
4. Be careful not to realize more of a lazy sequence than you need.
5. Know the sequence library. You can often write code without using `recur` or the lazy APIs at all.
6. Subdivide. Divide even simple-seeming problems into smaller pieces, and you'll often find solutions in the sequence library that lead to more general, reusable code.

The last two guidelines are particularly important. If you're new to FP, you can translate those to: "Ignore this chapter and just use the techniques in Chapter 3, *Unifying Data with Sequences*, on page ? until you hit a wall."

Now, let's get started writing functional code.

How to Be Lazy

Before we get to laziness, we first need to delve into recursion as an approach to enumerating sequences of values.

Functional programs make great use of *recursive definitions*. A recursive definition consists of two parts:

- A *basis*, which explicitly enumerates some members of the sequence
- An *induction*, which provides rules for combining members of the sequence to produce additional members

Our challenge in this section is converting a recursive definition into working code. You might do this in several ways:

- A simple recursion, using a function that calls itself in some way to implement the induction step.
- A tail recursion, using a function calling itself only at the tail end of its execution. Tail recursion enables an important optimization.
- A lazy sequence that eliminates actual recursion and calculates a value later, when it's needed.

Choosing the right approach is important. Implementing a recursive definition poorly can lead to code that performs terribly, consumes all available stack and fails, consumes all available heap and fails, or does all of these. In Clojure, being lazy is often the right approach.

We'll explore all of these approaches by applying them to the Fibonacci numbers. Named for the Italian mathematician Leonardo (Fibonacci) of Pisa (c.1170–c.1250), the Fibonacci numbers were actually known to Indian mathematicians as far back as 200 BC. The Fibonacci numbers have many interesting properties, and they crop up again and again in algorithms, data structures, and even biology.[4] The Fibonaccis have a very simple recursive definition:

- Basis: F_0, the zeroth Fibonacci number, is zero. F_1, the first Fibonacci number, is one.
- Induction: For $n > 1$, F_n equals $F_{n-1} + F_{n-2}$.

Using this definition, the first 10 Fibonacci numbers are as follows:

```
(0 1 1 2 3 5 8 13 21 34)
```

Let's begin by implementing the Fibonaccis using a simple recursion. The following Clojure function will return the *nth* Fibonacci number:

src/examples/functional.clj

```
Line 1  ; bad idea
     2  (defn stack-consuming-fibo [n]
     3    (cond
     4     (= n 0) 0
     5     (= n 1) 1
     6     :else (+ (stack-consuming-fibo (- n 1))
     7              (stack-consuming-fibo (- n 2)))))
```

4. http://en.wikipedia.org/wiki/Fibonacci_number

Lines 4 and 5 define the basis, and line 6 defines the induction. The implementation is recursive because `stack-consuming-fibo` calls itself on lines 6 and 7.

Test that `stack-consuming-fibo` works correctly for small values of `n`:

```
(stack-consuming-fibo 9)
-> 34
```

Good so far, but there's a problem calculating larger Fibonacci numbers such as $F_{1000000}$:

```
(stack-consuming-fibo 1000000)
-> StackOverflowError clojure.lang.Numbers.minus (Numbers.java:1837)
```

Because of the recursion, each call to `stack-consuming-fibo` for $n > 1$ begets two more calls to `stack-consuming-fibo`. At the JVM level, these calls are translated into method calls, each of which allocates a data structure called a *stack frame*.[5]

The `stack-consuming-fibo` creates a depth of stack frames proportional to `n`, which quickly exhausts the JVM stack and causes the `StackOverflowError` shown earlier. (It also creates a total number of stack frames that's exponential in `n`, so its performance is terrible even when the stack does not overflow.)

Clojure function calls are designated as *stack-consuming* because they allocate stack frames that use up stack space. In Clojure, you should almost always avoid stack-consuming recursion as shown in `stack-consuming-fibo`.

5. http://tinyurl.com/jvm-spec-toc