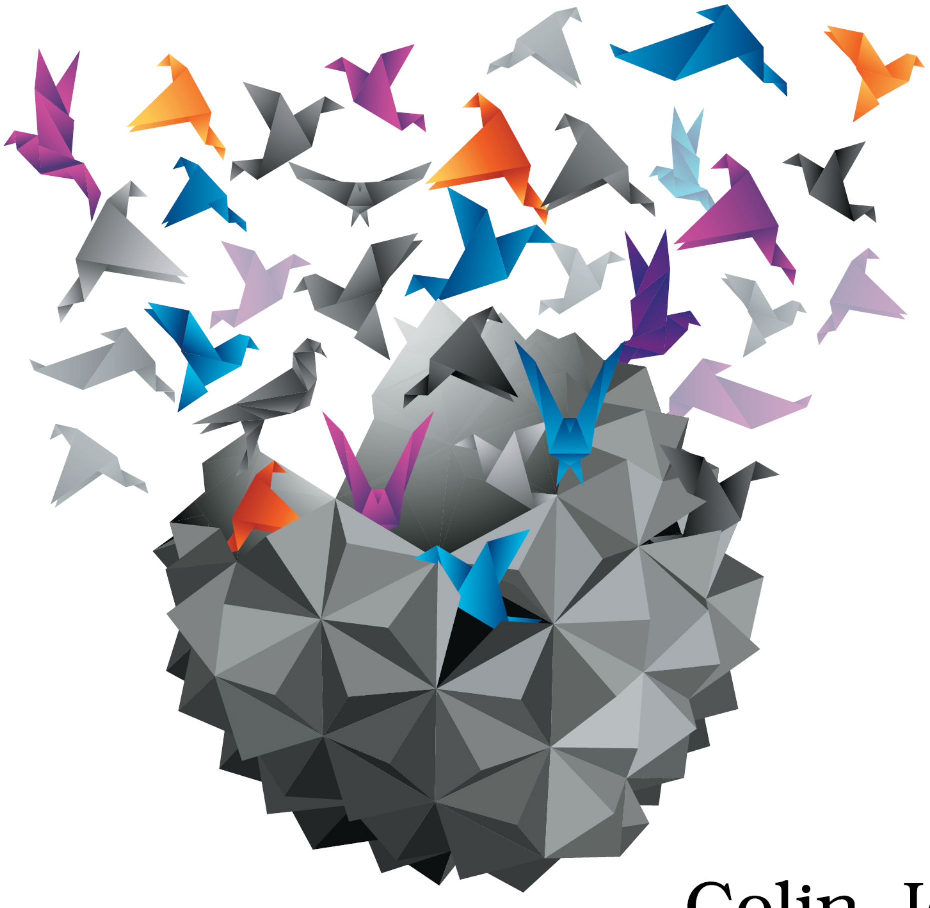
The Pragmatic Programmers

Pragmatic exPress

Mastering Clojure Macros

Write Cleaner, Faster, Smarter Code



Colin Jones

Edited by Fahmida Y. Rashid

Early Praise for *Mastering Clojure Macros*

So you thought you knew Clojure? This book changes everything. From its deceptively simple beginnings, to its very challenging conclusion, these pages will help you master the highest power tools of this already powerful language.

➤ **Robert C. “Uncle Bob” Martin**
Uncle Bob Consulting

This book provides a nice introduction to Clojure macros without being a general beginning Clojure book. Practical examples of how macros can help with performance enhancement and error detection are included. Many good books have been written on Clojure, but this is the first I have seen that treats Clojure macros in such great detail.

➤ **Charles Norton**
Manager of Software Development, Boston, MA

Clojure macros are a heavyweight topic, but Colin makes them light and fluffy without pulling any punches. He shares critical concepts and golden ideas that’ll level up your coding skills.

➤ **Micah Martin**
President, founder, 8th Light

I’d written thousands of lines of Clojure code but still shied away from macros. Colin Jones explains complicated concepts with simplicity and a sense of humor that few writers can match. This book’s engaging and humble prose finally helped me understand macros, enough to use them in my own code.

➤ **Eric Smith**
Director of Training Services

Mastering Clojure Macros

Write Cleaner, Faster, Smarter Code

Colin Jones

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Fahmida Rashid (editor)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-941222-22-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2014

Contents

Acknowledgments

I'm deeply indebted to Bob Martin for his suggestion that a book on Clojure macros was something the community could really use. If it hadn't been for that dinner conversation last June, I'd never have written this book.

My editor, Fahmida Y. Rashid, has been a constant source of insight into the famous Pragmatic style, and I've learned a lot about writing, and communication in general, from her.

I've been lucky to have a number of great tech reviewers along the way. It's been great learning about both macros and clarity in writing from Michael Baker, Alex Baranosky, Gary Fredericks, Bob Martin, Micah Martin, Carin Meier, Connor Mendenhall, Dave Moore, Charles Norton, Nhu Nguyen, Eric Smith, Jim Suchy, Zach Tellman, and Ben Trevor. I'd like to particularly thank Gary Fredericks, Bob Martin, Micah Martin, and Charles Norton for going above and beyond in the depth and substance of their feedback.

Of course, a book on Clojure macros wouldn't exist without Clojure's creator Rich Hickey, and the many other folks who have worked on and with Clojure macros in the open source world. Their work is both instructive and inspirational, and we're all lucky to have it.

Most importantly, thanks to my wonderful wife Kathy and my sweet son Owen for putting up with me working late nights and early mornings, and for being so supportive throughout the process.

And finally, thanks to you, dear reader. Yes, you, the one reading this sentence! I hope you enjoy the book!

Introduction

It probably won't surprise you to learn that this is a book about macros in Clojure. The title dashes away any hope of keeping that a secret, which is fine. I'm not big on surprises anyway.

Furthermore, this book is about *mastering* macros. Now, mastery is a pretty audacious goal: some folks say it takes 10,000 hours to master any skill, which is way more time than it'd take you to memorize this book forwards and backwards. There's always more to learn. But this is a path on the journey to mastery. So pack your things—it'll be a fun trip!

Why Clojure?

There are a number of reasons to learn Clojure. The discipline of staying immutable by default, the simplicity of functions as first-class entities, the practicality of the JVM, and… well, I could go on and on, but this book's goal isn't to sell you on Clojure. There are already several fantastic books out there for that. My first Clojure book was *Programming Clojure [Hal09]*, and my most recent intro-to-Clojure recommendation has been *Clojure Programming [ECG12]* (trust me, these are actually two different books, despite their naming similarities). *The Joy of Clojure [FH11]* is excellent as well, but it can be a bit advanced for Clojure newcomers. This book probably falls into the same category: you should already know a bit of Clojure in order to get the most out of this book.

So in the interest of focus and brevity, I'm going to be assuming throughout this book that you already know some Clojure. That said, if you haven't written a single line of Clojure before, never fear! You should set this book aside briefly and go through the Clojure Koans,[1] and I'll meet you back here in a couple of hours. We'll be tackling some advanced topics in this book, so if you haven't worked through the koans, a book, or a Clojure project, you

1. http://clojurekoans.com

may need to spend some extra time with some of the chapters in this book to get more Clojure under your belt.

You'll also want to make sure you have Leiningen[2] installed and know how to do things like launch a *REPL* (Read-Eval-Print Loop, `lein repl` from a command line), read the in-REPL docstrings, and peek here and there at the source code.

From this point forward, you should know your way around a Clojure REPL. I'll do my best to keep the progression steady through the tutorial part of the book, with each step building upon the one before. But my realistic, software-estimating side says I won't think of everything, and here and there I'll get too excited and skip a step. I hope that when that happens, you'll reach for your nearest REPL, recover quickly, and forgive my exuberance.

Why Macros?

Clojure stands on the shoulders of giants, with influences from several functional and object-oriented languages, database and distributed systems technologies, and of course the tremendous force of nature that is its creator, Rich Hickey. And despite Clojure's youth, we've already started to see some cross-pollination into other language communities. But one of the real killer features of Clojure is the macro system, which is similar in many ways to Common Lisp's but brings its own modern flair to the area.

Macros in Clojure are an elegant metaprogramming system, a means to accomplish goals that might seem impossible in other languages. How hard would it be to add pattern matching or a new control flow structure to your language as a library (rather than patching the core language)? In Clojure, people like you and me have the power to do these things ourselves.

It's true in a sense that all general-purpose languages are equally powerful, but we programmers know better. Our limitations and goals are not the same as those of a Turing machine. We want lean, clean code that expresses our intent clearly while allowing us to tell the machine what to do as succinctly as possible. Macros are a way to get there.

Metaprogramming in Non-Lisps

Lisp certainly isn't the only language family with metaprogramming facilities. C has a macro system that allows textual replacement as a preprocessing step at the start of compilation. C++ has a complex and powerful template

2. http://leiningen.org/

metaprogramming feature that's itself Turing-complete. Plenty of dynamic languages, like JavaScript, Python, and Ruby, have `eval` functions that acts on strings to produce a result. Ruby, in particular, has quite a few nice features, and Scala's macro system even allows you to manipulate the parse tree! But here's fair warning: once you've experienced Clojure's macros, you may not want to go back.

You may notice that in this book, sometimes I'll refer to Clojure, and sometimes to Lisp in general (since Clojure is part of the Lisp family of languages). This shouldn't be too confusing, but let's take a moment to clarify, just to make sure. When I talk about Lisp, the advice will apply across all the major Lisps (including Clojure, Scheme, and Common Lisp). Elixir is an interesting edge case: it's not quite a Lisp, but its macro system sure walks and talks like one. But just because I mention Clojure specifically doesn't mean that the advice *only* applies to Clojure. It may well apply in other Lisps too, but not necessarily.

Who Is This Book For?

This book assumes that you've programmed in Clojure a bit but that you're not yet a master of Clojure macros. Experienced Clojure programmers will no doubt recognize situations they've encountered before, whereas newer Clojure folks will learn to avoid some of the stumbles the rest of us have made when learning macros. Macros are typically one of the most difficult features, and this book will help anyone who wants to understand how they work and when to use them.

What's in This Book?

Chapter 1, *Build a Solid Foundation*, on page 1 introduces the basic building blocks that will allow you to understand how macros work and what kinds of things it's possible to do with them.

Chapter 2, *Advance Your Macro Techniques*, on page 15 shows how and why to use syntax-quoting, unquoting, and gensyms. These are some the trickiest concepts for new macro writers, but also some of the most useful.

Chapter 3, *Use Your Powers Wisely*, on page 27 takes a step back to dig into some of the problems macros can create, and how you can avoid those problems.

Chapter 4, *Evaluate Code in Context*, on page 35 starts the field guide portion of the book, and covers the first main use case you see in the wild: wrapping a block of code in a context for its execution.

Chapter 5, *Speed Up Your Systems*, on page 49 digs into how macros can help you write very fast Clojure code without sacrificing concision and cleanliness.

Chapter 6, *Build APIs That Say Just What They Mean*, on page 59 outlines some ways that macros allow you to provide easy-to-use APIs that let users write only the minimum amounts of code.

Chapter 7, *Bend Control Flow to Your Will*, on page 65 shows you how you can invent your own loops and other control flow mechanisms without relying on the language to provide you with new ones.

Chapter 8, *Implement New Language Features*, on page 77 goes a step further and shows you how you can steal some of the best features from other languages and introduce them to your Clojure code with macros.

How to Read This Book

There are lots of code examples in this book, and I suggest trying them out in your own REPL, creating a toy project for the longer examples. Leiningen[3] is the Clojure build tool of choice, and `lein new macrobook` will give you a new project to play around in. The examples in this book have been tested using Clojure 1.6.0, so you're safest using that version of Clojure when you try things out. But since the macro system changes so rarely, I anticipate that many earlier and future versions will work just fine as well.

This book is intended to be read front to back, but if you're the adventurous or time-constrained sort, here are a few ideas to get you the information you're looking for:

The first two chapters, Chapter 1, *Build a Solid Foundation*, on page 1 and Chapter 2, *Advance Your Macro Techniques*, on page 15, teach you all the building blocks you'll need to write your own macros. If you already have a pretty good working knowledge of macros and have written and debugged several of them yourself, you may wish to skip straight to Chapter 2, *Advance Your Macro Techniques*, on page 15.

Everyone should read Chapter 3, *Use Your Powers Wisely*, on page 27, where you'll see some of the problems that macros can cause and why you don't want to use them all the time.

In the remaining chapters, starting with Chapter 4, *Evaluate Code in Context*, on page 35, we'll look in depth at some use cases for macros, with examples

3. http://leiningen.org

from both the Clojure language itself and community libraries. In these chapters, you'll see reasons you'd use macros in practice, despite their imperfections, and sometimes you'll see ways to accomplish the same goals with normal functions.

It's fine to skip around once you've finished Chapter 3, *Use Your Powers Wisely*, on page 27. By that point you'll have all the basic tools you need to understand the rest of the book. The macros you see will generally get more complex as the chapters progress, but there are minimal dependencies among the later chapters.

Online Resources

Take a look at this book's official website,[4] where you can order copies of this book as gifts and download the book's source code. You can also submit error reports.[5]

Please send specific book questions or commentary directly to me via the forums on the official website, but there are also some wonderful community resources for more general questions, or ones outside the scope of this book. I highly recommend visiting the #clojure IRC channel on Freenode and the Clojure mailing list[6] for those questions that aren't specifically about this book.

This book is a brief one, but don't be fooled–there's a lot of material to cover. Don't hesitate to try things out at the REPL and reread sections that move a bit too quickly. A small time investment now could pay off in deeper understanding later on.

Now let's get started by digging into the details of how Clojure macros work.

4. http://pragprog.com/book/cjclojure/clojure-macros
5. http://pragprog.com/book/cjclojure/errata
6. http://groups.google.com/group/clojure

CHAPTER 1

Build a Solid Foundation

Over the course of this book you're going to learn how and when to write macros in Clojure, but first you need to make sure the fundamentals are in place. In this chapter, you'll see how reading, quoting, and macroexpansion work. Don't worry—you won't have to wait long before you get to write some simple macros. You'll also see some built-in tools that Clojure provides developers for investigation when macros misbehave.

You don't need a lot of setup to get started: just the REPL will do for now. If you don't have Leiningen[1] installed, I suggest you go ahead and do that now, since it's one of the easiest ways to use Clojure and comes with a nice REPL[2] built in. The examples in this book assume Clojure 1.6.0, so it's best to use that version when you type them into your REPL (though other versions are likely to work just fine as well).

If you've done a decent amount of Clojure, some of this chapter will be review, but there should be some new ways of thinking about things for all but the most erudite of macro scholars.

Code Is Data

Have you ever heard your friendly (but annoying) neighborhood Lisper loudly proclaim that "Code is data!" and wondered why in the heck anyone would store source code in a database? OK, of course you know that's not what she means, but it's not necessarily obvious what the expression *does* mean.

So let's drill in: she means that the code itself is constructed using only the data structures in the language. Non-Lisp languages, on the other hand, usually have code with some relatively large syntax, where the syntax for

1. http://leiningen.org
2. https://github.com/trptcolin/reply

their data structures is just a small subset of the full language syntax. This is much easier to visualize with an example, so let's take a look at the data structures contained in a bit of Clojure code from clojure.core:

basics/defn.clj
```
(defn mapcat
  "Returns the result of applying concat to the result of applying map
  to f and colls.  Thus function f should return a collection."
  {:added "1.0"
   :static true}
  [f & colls]
  (apply concat (apply map f colls)))
```

Try your best to ignore the details of what this code does—let's instead focus on its syntactic structure. The word defn here is a Clojure symbol, and the famous Lisp parentheses that surround it mean that everything from the opening to the closing parenthesis represents a list. Similarly, the parentheses around the two apply invocations also represent lists. The docstring ("Returns the result...") is just a literal string, and the :added and :static are keywords within a map of metadata. The argument list, [f & colls], is written as a vector of three symbols, f, &, and colls.

We have two ways to interpret this code. We can look at it the way we just did, as data structures, or we can look at how it evaluates. These dual interpretations are closely related to the way macros work. In the mapcat definition, the data structure representation of the code allows defn (which, as it turns out, is just a macro), to manipulate the source code passed to it.

Can you imagine why—aside from trivia and aesthetics—we actually care that the code can be viewed as normal data? The most compelling answer is that it makes it relatively straightforward to manipulate programs. When we do metaprogramming in Clojure, we can think at the expression level rather than at the textual level. It may not seem like a big deal right now, but as you'll see over the course of this book, this simple concept is the key that allows you to write macros.

Transforming Code

Now you have this sort of abstract idea that you can view code in two different ways: either as code or as data. But what does that mean concretely?

Let's put this in terms you already know by thinking about the first two phases of the REPL (Read-Eval-Print loop, remember?). The first phase, read, takes a character-based representation (typically an input stream), and turns it into Clojure data structures. So the output of the read phase is data, which is then interpreted by the second phase, eval, as code! Let's take a closer look so you'll understand how to transform the code-as-data from the reader into the data-as-code that will be evaluated.

read is a convenient way for us to build the expressions that macros will eventually act on. Let's look at a few inputs and outputs for read, make sure you follow why things work the way they do, and that'll give you enough to dive headfirst into your first macro. The actual read function in Clojure consumes streams, which can be a bit verbose to set up. So for the purposes of these examples, we'll use its sibling read-string, which consumes strings instead.

When we read this expression out of a string, we get back a list:

basics/read_string.clj

```
(read-string "(+ 1 2 3 4 5)")
;=> (+ 1 2 3 4 5)
(class (read-string "(+ 1 2 3 4 5)"))
;=> clojure.lang.PersistentList
```

This list is one of the pieces of data we've been talking about. It's a Clojure list, and yet it's also Clojure code. In the format that comes back from read-string, it's ready for evaluation. When we eval that list, we get what we'd expect as the value of that expression:

basics/eval_expression.clj

```
(eval (read-string "(+ 1 2 3 4 5)"))
;=> 15
(class (eval (read-string "(+ 1 2 3 4 5)")))
;=> java.lang.Long
(+ 1 2 3 4 5)
;=> 15
```

This is pretty much what happens when you write code in a Clojure REPL, or when you run a full-blown Clojure project: the code is read into data structures and then evaluated. Given these two distinct steps, it's not too hard to imagine wedging ourselves in there between the read and eval steps in order to change the code to be evaluated. For instance, we could replace addition with multiplication:

basics/replace_addition.clj

```
(let [expression (read-string "(+ 1 2 3 4 5)")]
  (cons (read-string "*")
        (rest expression)))
;=> (* 1 2 3 4 5)
(eval *1) ;; *1 holds the result of the previous REPL evaluation
;=> 120
```

And of course, if we eval that new list, we'd get an entirely different result than our original. The code here isn't bad, but let's work toward a version that constructs the expression from scratch rather than reading it in from a string, and clean it up a bit in the process. read-string works fine, but it would be much nicer to type them in as lists, without the fuss of going through a string.

But if we want to create the list (+ 1 2 3 4 5), we can't just type in (+ 1 2 3 4 5), because that would actually evaluate the expression.

basics/replace_addition_broken.clj

```
(let [expression (+ 1 2 3 4 5)] ;; expression is bound to 15
  (cons
    (read-string "*")    ;; *
    (rest expression)))  ;; (rest 15)
; IllegalArgumentException Don't know how to create ISeq from: java.lang.Long
;   clojure.lang.RT.seqFrom (RT.java:505)
```

So we want a different solution, one that suppresses execution somehow. Luckily for us, there's a verb in Clojure called quote that does exactly that:

basics/replace_addition_2.clj

```
(let [expression (quote (+ 1 2 3 4 5))]
  (cons (quote *)
        (rest expression)))
;=> (* 1 2 3 4 5)
```

But wait, didn't I just say that we couldn't type in (+ 1 2 3 4 5), yet here we are doing it, and seeing the right thing happen anyway! What gives? Note that I said *verb* instead of *function*. The quote verb is, in fact, a Clojure special form and not a function. We can think of verbs (not an official language term, just a convenient concept to think about) as the union of functions, macros, and special forms—the things that appear right after the opening parenthesis in the first position of a list. Lists can also appear in the verb position, but they have to reduce down to a function when evaluated. Only functions uniformly evaluate their arguments before passing control to the code that implements the verb. Verbs are summarized in Table 1, *Clojure Verbs*, on page 5.

Let's take a closer look and see how functions evaluate their arguments:

basics/function_argument_evaluation.clj

```
(defn print-with-asterisks [printable-argument]
  (print "*****")
  (print printable-argument)
  (println "*****"))

(print-with-asterisks "hi")
; *****hi*****
;=> nil
```

Verb	How do we define one?	When are the arguments evaluated?
Function	defn	Before executing the body
Macro	defmacro	Depends on macro; possibly multiple times or never.
Special form	We can't! They're only defined by the language.	Depends on special form; possibly multiple times or never.

Table 1—Clojure Verbs

When we just send something like a string into print-with-asterisks, we don't really need to think about the timing of the argument evaluation. But if we use an expression for the argument, we see that the expression is evaluated before any of the function body is evaluated.

basics/function_argument_evaluation-2.clj
```
(print-with-asterisks
  (do (println "in argument expression")
      "hi"))
; in argument expression
; *****hi*****
;=> nil
```

Macros and special forms, on the other hand, are not bound by this rule and may evaluate the arguments in whatever order they define (or do something entirely different!). This is a key difference, perhaps *the* key difference, between functions and macros, and we'll look at the consequences of this later on.

None of the code inside the quote expression, often bounded by parentheses or square or curly brackets, is evaluated. I say *often* instead of *always*, because you can also quote smaller tokens, things like numbers, strings, keywords, and symbols. We tend to use quote most often on symbols, lists, and (somewhat less often) vectors.

basics/quoting_tokens.clj
```
(quote 1)
;=> 1

(quote "hello")
;=> "hello"

(quote :kthx)
;=> :kthx

(quote kthx)
;=> kthx
```

There's a potentially confusing fact here, that things like numbers, strings, and keywords already appear to be *themselves* when being read. This simplifies things when we actually manipulate these expressions, and the fact is that there aren't many good reasons why you'd want different behavior here. Symbols and lists are the special ones: symbols are for representing locals, vars, classes, protocols, or other bindings; lists are for invoking verbs.

Believe it or not, `quote` gets even better, because there's a shorthand in Clojure called a *reader macro*, that converts a single quote character to wrap the following expression in the `quote` form. Reader macros aren't something you can create yourself—they're built into the language, and there are only a few of them. So I know it's hard, but don't get too excited here!

basics/quote_reader_macro.clj

```
'(+ 1 2 3 4 5)
;=> (+ 1 2 3 4 5)

'map
;=> map
```

The payoff of this exploration of the quoting mechanism is that we can rip out our previous monstrosity (using `read-string`) and express it much more succinctly with:

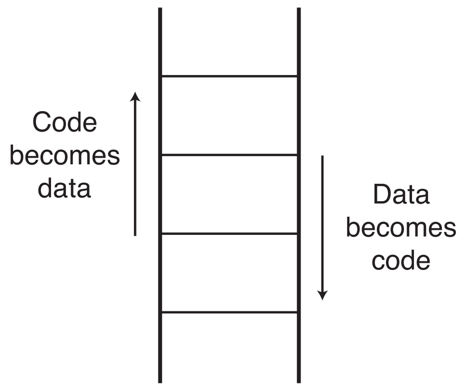
basics/replace_addition_3.clj

```
(let [expression '(+ 1 2 3 4 5)]
  (cons '* (rest expression)))
;=> (* 1 2 3 4 5)
```

Now you're starting to get a clearer idea of what people mean when they say that in Clojure, code is data, and data is code. We'll get to more complicated usages and variants of quoting soon enough, but this is plenty to get you started writing your first macro.

Evaluating Your First Macro

In order to think about how macros work, it's useful to picture a ladder like the one in the figure that follows this paragraph. When we take a step up this ladder, we're shifting from thinking in code to thinking in data, and when we step back down the ladder, we're shifting from data back to code. In this world, when we encounter a macro call in a piece of code, we'll think of it just like a function, but one that operates one level up the macro ladder, on unevaluated code rather than on evaluated data. When we need to expand a macro, we'll step up the ladder, and once we're done expanding it, we'll step back down (with the expanded expression in our pocket). This is a simplification, but it's a good enough way to think about macros for now.



Without further ado, let's look at the built-in `when` macro.

basics/when.clj
```
(defmacro when
  "Evaluates test. If logical true, evaluates body in an implicit do."
  {:added "1.0"}
  [test & body]
  (list 'if test (cons 'do body)))
```

Since we're thinking about macros as functions operating one step up the macro ladder, we can think of `when` as being a function that receives an argument `test`, and a variable number of arguments rolled up into a sequence as `body`. Now let's look at an actual call of `when` to see how this plays out in practice.

basics/when_call.clj
```
(when (= 2 (+ 1 1))
  (print "You got")
  (print " the touch!")
  (println))
```

Just like a normal function, `when` has values bound to `test` and `body` when it executes, but unlike a normal function, nothing is evaluated yet. `test` actually gets bound to the list `(= 2 (+ 1 1))`. And this list is just data. Sound familiar? No addition or comparison happens yet; this *is* that wedge between read and eval that we alluded to earlier. Moving on, `body` is bound to the list `((print "You got") (print " the touch!") (println))`: a list of lists, in fact! The extra set of parens come from rolling up three arguments due to the `& body` *varargs* (variable number of arguments) syntax.

So if we keep on reading `when` as a function, substituting the bindings for the names in the code, and then executing the macro body (not the resulting code), the result is another list.

basics/when_manual_expand.clj

```
;; with indentation and newlines added for clarity
(list 'if
      '(= 2 (+ 1 1))
      (cons 'do
            '((print "You got")
              (print " the touch!")
              (println))))
```

And indeed, this is the code that the `when` macro generates, but in our model of macros as functions acting on data, we're still thinking of this return value as a list, one step up the macro ladder from the bottom. So now we just need to descend the macro ladder by one rung, where we can evaluate this code that we've constructed. And of course, the result will be that we print out a lovely song lyric from a wonderful film (well, I choose to remember it as wonderful, anyway).

basics/when_result.clj

```
;; when evaluated at the macro level:
(if (= 2 (+ 1 1))
  (do (print "You got")
      (print " the touch!")
      (println)))

;; and when later evaluated as code:
;You got the touch!
;=> nil
```

This is pretty much how macros work. There is indeed an opportunity to wedge behavior in between reading and evaluating the code, and that wedge is named macroexpansion. Whenever Clojure encounters a macro in the verb position on an expression it's trying to execute, it will iterate this process we've just gone through. Macroexpansion is stepping up the macro ladder one rung: we treat the code of the macro's arguments as data. Those arguments are used in whatever way the macro says, and eventually we create a new expression by evaluating the code that makes up the body of the macro. We put that resulting expression into our pocket, descend the macro ladder one rung again, take the expression out of our pocket, and replace the original macro expression with the resulting expression for execution.

This may feel a bit like macros you've seen elsewhere, in C or C++, or even Scala. The difference is that with Lisp we write macros in terms of normal sequence manipulation rather than string or AST manipulation. We don't need to write our own parser or know how to generate the compiler's syntax tree nodes. In Lisp, metaprogramming feels more similar to regular programming.

Of course, things are often more complicated in practice. In a larger example, we might encounter another macro while in the process of expanding the first macro. Let's consider another macro defined in terms of `when`: `cond`.

basics/cond.clj

```
(defmacro cond
  "Takes a set of test/expr pairs. It evaluates each test one at a
  time.  If a test returns logical true, cond evaluates and returns
  the value of the corresponding expr and doesn't evaluate any of the
  other tests or exprs. (cond) returns nil."
  {:added "1.0"}
  [& clauses]
  (when clauses
    (list 'if (first clauses)
          (if (next clauses)
              (second clauses)
              (throw (IllegalArgumentException.
                       "cond requires an even number of forms")))
          (cons 'clojure.core/cond (next (next clauses))))))
```

This one will take a little more effort to expand, because we need to continue climbing the macro ladder until the expression we have is no longer a macro call. When we try to macroexpand the smallest possible invocation of `cond`, passing no arguments, we're already one macro-ladder rung up from the bottom, and so we might feel a little stuck when we find `when` (a macro) in the verb position.

basics/cond_expansion_1.clj

```
(cond)

;; expanding, up a ladder rung and treating `cond` as a function:

'(when clauses
  (list 'if (first clauses)
        (if (next clauses)
          (second clauses)
          (throw (IllegalArgumentException.
                   "cond requires an even number of forms")))
        (cons 'clojure.core/cond (next (next clauses)))))
```

As humans, we know that `clauses` will be `nil`, since no `clauses` were passed in, but we can't yet substitute `nil` in place of `clauses` in this `when` expression, since we're not at the bottom of the ladder. `when` is a macro, and macros act on *code*, evaluating if and when they choose. We need to climb the ladder another rung to see what lies in store for us, but we'll keep in mind that clauses will be bound to nil when we get back down to the first macro ladder rung.

basics/cond_expansion_2.clj
```
;; ascending another ladder rung, treating `when` as a function:

(list 'if 'clauses
  (cons 'do
        '((list 'if (first clauses)
                (if (next clauses)
                  (second clauses)
                  (throw (IllegalArgumentException.
                           "cond requires an even number of forms")))
                (cons 'clojure.core/cond (next (next clauses)))))))
```

Now since list is a function, we get to evaluate the arguments and move back down a rung to get closer to our eventual return value.

basics/cond_expansion_3.clj
```
;; descending a rung:
(if clauses
  (do (list 'if (first clauses)
            (if (next clauses)
              (second clauses)
              (throw (IllegalArgumentException.
                       "cond requires an even number of forms")))
            (cons 'clojure.core/cond (next (next clauses))))))
```

Now our result starts with a special form, if, and we're still one rung up from where we started, so we need to evaluate the form in the way the Clojure language defines if. Since if says we evaluate the test (the first argument) first to decide whether to evaluate the second or the third argument, we need to first evaluate clauses.

And because we're at level 1, and level 1 is where we have clauses bound to nil, we have that binding available for this evaluation. Since nil is a false-like value, we'd ordinarily go down the *else* branch (the third argument) of the if, but there is no third argument! This leaves us with nil as the result, because Clojure defaults the *else* branch to be nil.

So at long last, we've arrived at our goal: the result of macroexpanding the expression (cond). And that result is just nil. So we put that in our pocket, descend the last ladder rung back to the bottom, and insert that nil into the original context as code. Well, nil as code just evaluates to itself, so the result of evaluating (cond) is nil as well.

Sigh—that was pretty anticlimactic, and it took a lot of tedious and error-prone work to sort it out by hand. Trust me: macro writing is not always going to feel like this. It will become easier with each macro you write, and the macros you use will fade away into your tacit knowledge of your systems. And luckily, we do have tools to manage these sorts of complexities and to check our work as we write macros. Let's look at those, before we start feeling acrophobic with all this ladder talk.

Macroexpansion

Manually walking through a macroexpansion, as you saw in the previous section, is fairly tedious work. But as you might guess, you don't always have to do it manually. Clojure exposes to users the very same macroexpansion tools that it uses internally. This way, you can have a look at the expression a macro will generate, without having to go through that whole process you just saw.

`macroexpand-1` is the simplest of these tools, converting a macro expression to its resulting expression.

basics/macroexpand_1.clj

```
(macroexpand-1 '(when (= 1 2) (println "math is broken")))
;=> (if (= 1 2) (do (println "math is broken")))

(macroexpand-1 nil)
;=> nil

(macroexpand-1 '(+ 1 2))
;=> (+ 1 2)
```

Note the similarity here to what we did by hand in the last section with our first `when` expression. `macroexpand-1` is a big help when debugging macro issues, because it tells you precisely how a macro expression will be replaced during the macroexpansion step. Make sure you give `macroexpand-1` a quoted expression or something that generates one, or you'll be in for a surprise!

basics/macroexpand_1_2.clj

```
(macroexpand-1 (when (= 1 2) (println "math is broken")))
;=> nil
```

Omitting the quote here failed to do what we wanted, because `macroexpand-1` is a regular function. This means it will evaluate its arguments *before* passing control to the code that composes `macroexpand-1`. So the `when` expression actually executes, returns `nil`, and since `nil` isn't a macro expression, `macroexpand-1`'s job is done. Macroexpansion has no effect when a given expression is not a macro call.

It's worth noting that macros must return something that it makes sense to evaluate! The when macro works because it returns a list whose first element is the quoted symbol if. Let's see what would happen if we rewrote our own version of the when macro, but forgot that the result of the macro needed to be an eval-able form. We'll delete the 'if quoted symbol to demonstrate the point—removing the docstring and other metadata just makes it easier to focus.

basics/broken_macro_1.clj

```
(defmacro broken-when [test & body]
  (list test (cons 'do body)))

(broken-when (= 1 1) (println "Math works!"))
; ClassCastException java.lang.Boolean cannot be cast to clojure.lang.IFn
;   user/eval316 (NO_SOURCE_FILE:1)
```

When I'm learning something new, I sometimes find myself practicing EDD (exception-driven development). I try to evaluate some code, get an exception or error message, and then Google the error message to figure out what the heck happened. We could do that here, and we'd probably find the right answer, but since we now know about macroexpand-1, let's try that first.

basics/broken_macro_1_macroexpand.clj

```
(macroexpand-1
  '(broken-when (= 1 1) (println "Math works!")))
; ((= 1 1) (do (println "Math works!")))
```

Aha! The expression we generate here is a list, and its first element is another list, so in order to decide how to evaluate the top-level list, we'd need to evaluate the first element. (= 1 1) is true, so the first element of the top-level list will be true. But true is not a verb of any sort (function, macro, or special form)—it's a Boolean! This explains the error message entirely: we expected to have an IFn (a function), but we got a Boolean, and that doesn't make any sense. You've probably seen this kind of thing before in Clojure, when accidentally typing unquoted lists at the REPL. As long as we make sure the expressions we return from our macroexpansion are possible to evaluate, we'll avoid these kinds of errors.

We can go a step further with macroexpand, which does what macroexpand-1 does, but it actually continues along in the same way until the returned expression is either a non-list or a list whose first element is no longer a macro. So, for example, if we have a macro that expands to another macro call, macroexpand will do that second expansion for us, but macroexpand-1 will not. This is in contrast to the ladder idea, where a macro might *use* another macro instead of *expanding* to another macro.

basics/macroexpand.clj

```
(defmacro when-falsy [test & body]
  (list 'when (list 'not test)
        (cons 'do body)))

(macroexpand-1 '(when-falsy (= 1 2) (println "hi!")))
;=> (when (not (= 1 2)) (do (println "hi!")))

(macroexpand '(when-falsy (= 1 2) (println "hi!")))
;=> (if (not (= 1 2)) (do (do (println "hi!"))))
```

Both tools have their place, depending on whether you want to examine a single macroexpansion or look at the final product. Note that this applies only to macros at the start of the expression being expanded—expanding macros *within* the expression would need heavier machinery. It's tricky to correctly macroexpand everything, and the built-in `clojure.walk/macroexpand-all` works for some expressions but is too naïve to cover all cases. clojure.tools.macro[3] and Riddley[4] are two more ambitious code-walking projects intended to allow expanding *everything*, with different strategies based on those projects' goals.

A Confession, and Next Steps

So the macro ladder analogy is a bit of a stretch. It's a decent way to figure out which macro code will execute, but not necessarily *when*. When we write a macro that *uses* a macro to do its expansion, Clojure will perform the necessary expansion when it's compiling our macro. Whenever we go to use a macro, any macros it uses have already been expanded. In the case of `cond`, Clojure doesn't need to climb any of the ladder rungs above `when`, as they have already been traversed by the time we call `(cond)`. This means that there's usually only one ladder rung that we have to worry about at any given time.

But the ladder concept can still be useful when tracing code execution in a new macro, because unlike the Clojure compiler, when we do macroexpansion in our heads, we don't hold onto the expansions of every macro in memory! We humans usually have to look at each function or macro in isolation, and either remember know by heart what a given macro call does or find its definition and mentally expand it (or use tools like `macroexpand-1` as we explore).

At this point we have most of the tools we need to write macros. Next we'll look at some advanced techniques and syntax to avoid some of the verbosity that's required with what you know so far, and even some tricks that make crazy new things possible.

3. https://github.com/clojure/tools.macro
4. https://github.com/ztellman/riddley

CHAPTER 2

Advance Your Macro Techniques

Most of the macros you've seen so far have been small and straightforward. Wouldn't it be great if they could all be like that? Unfortunately, as you do more and more with macros, the syntax you know so far can get unwieldy.

What if you had to write an assert macro like the one that comes with Clojure? Given what you know at this point, you'd need to do something like this:

advanced_mechanics/assert_no_syntax_quote.clj

```
(defmacro assert [x]
  (when *assert* ;; check the dynamic var `clojure.core/*assert*` to make sure
                 ;;   assertions are enabled
    (list 'when-not x
       (list 'throw
             (list 'new 'AssertionError
                   (list 'str "Assert failed: "
                         (list 'pr-str (list 'quote x))))))))

user=> (assert (= 1 2))
;=> AssertionError Assert failed: (= 1 2)  user/eval214 (NO_SOURCE_FILE:1)

user=> (assert (= 1 1))
;=> nil
```

And this isn't even a complete solution! We've skipped the arity[1] that takes a failure message string, to keep things from getting too ridiculous. But there's a lot to read and learn here, right?

I don't know about you, but I find it very hard to parse all those nested lists to discover what's going to come out in the macroexpansion. Luckily we know about macroexpand from *Macroexpansion*, on page 11, so it doesn't have to stay a mystery for long:

1. http://en.wikipedia.org/wiki/Arity

advanced_mechanics/assert_no_syntax_quote_macroexpanded.clj
```
(macroexpand '(assert (= 1 2)))
;=> (if (= 1 2)
;     nil
;     (do (throw (new AssertionError
;                  (str "Assert failed: "
;                       (pr-str (quote (= 1 2))))))))
;;; [indentation for clarity]
```

How can we do this better? Maybe you already have some ideas about how we can make this code look much more like the code it generates. To do so, we'll use *syntax-quote* for the first time.

Syntax-Quoting and Unquoting

The big problem with this assert implementation is that it takes a pretty big structural leap to go from the macro implementation to the result of the macroexpansion. This is no problem for the compiler, and our human brains can work it out too, but there's an easier way. The syntax-quote gives us a way to structure a macro's code to look much more like its macroexpansion.

The syntax-quote lets us create lists similar to the way we create them with a normal quote, but it has the added benefit of letting us temporarily break out of the quoted list and interpolate values with an unquote. Think of it as a template, where we can punch holes and insert values wherever we like. For instance, if we had a list where we wanted to insert a value, our normal quote wouldn't fly:

advanced_mechanics/normal_quote_is_stubborn.clj
```
user=> (def a 4)
;=> #'user/a
user=> '(1 2 3 a 5)
;=> (1 2 3 a 5)

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

The fourth element in the first list we created is just the symbol a. If we want the value of a, we have to either use the more verbose list construction, or use a syntax-quote with an unquote:

advanced_mechanics/syntax_quote_1.clj
```
user=> (def a 4)
;=> #'user/a
user=> `(1 2 3 ~a 5)
;=> (1 2 3 4 5)
```

If you don't see the difference in these two quote characters at first, look a wee bit closer. The normal quote (') looks like it's on the straight and narrow, and the syntax-quote (`) is a little cockeyed and apparently ready to party. You might also know the syntax-quote as the backtick. The *unquote* (~), well, it unquotes, letting us insert evaluations into the syntax-quoted expression.

If we take a look at how assert is actually implemented, we see that syntax-quote and unquote completely solve the verbosity problem we saw earlier:

advanced_mechanics/assert_syntax_quote.clj
```
(defmacro assert [x]
  (when *assert*
    `(when-not ~x
       (throw (new AssertionError (str "Assert failed: " (pr-str '~x)))))))
```

Wow! That's a lot less code than the nested-list version, and it looks a lot closer to the macroexpansion. As a result, it's easier to understand and maintain. There's one potentially tricky bit left, though: what does it mean when we say '~x within a syntax-quoted expression?

The REPL is a great place to experiment whenever you see something you don't understand. Why don't we give it a try?

advanced_mechanics/syntax_quote_2.clj
```
user=> (def a 4)
;=> #'user/a
user=> `(1 2 3 '~a 5)
;=> (1 2 3 (quote 4) 5)
```

Aha! So this strange '~ dance gives us a way to quote the result of evaluation and plug *that* into a slot in the syntax-quote expression.

Recall from our introduction to quoting on page 6 that the normal quote is a reader macro expanding to (quote ...). Well, it turns out that the unquote ~ is another reader macro. So an expanded version of this would look like:

advanced_mechanics/syntax_quote_3.clj
```
user=> `(1 2 3 (quote (clojure.core/unquote a)) 5)
;=> (1 2 3 (quote 4) 5)
```

Internally, Clojure's reader has some special code to walk through the syntax-quote form looking for clojure.core/unquote occurrences and unquoting those things. I wouldn't try to use clojure.core/unquote outside the scope of a syntax-quote, though—it won't work unless you've written a macro that makes it work. Leiningen[2] (as of version 2.3.4) actually allows the unquote to be used in project.clj for evaluation, but it's now discouraged in favor of *read-eval*. That

2. https://leiningen.org

var (clojure.core/unquote) is unbound by default, and it's unclear whether it's strictly needed by the language, since everything using it looks for a *symbol*, not a var.

Joe asks:

What's Read-eval?

Read-eval is the name of the Clojure reader macro that allows you to evaluate code during a read. For instance, (read-string "#=(+ 1 2)") returns 3, not (+ 1 2) as you'd get without the #=. This has security implications for read and anything that depends on it, so you should always be careful not to read user input, preferring something like clojure.edn/read instead. Generally you should make it a goal to steer clear of read-eval whenever possible.

There are other strange ways we can mix and match these quotes and unquotes as well, but first let's see a special kind of unquote called *unquote-splicing*. Let's say we have a list in hand of an unknown length, and we want to insert all the elements from that list into another list. Using the unquote that we already know about won't fly here, but we can use the usual concat to get the result we want. Always remember that when you hit a wall in writing macros, you can fall back on all your normal Clojure-writing experience, since when you write macros you're manipulating normal data structures like lists.

advanced_mechanics/unquote_splicing_1.clj

```
user=> (def other-numbers '(4 5 6 7 8))
;=> #'user/other-numbers
user=> `(1 2 3 ~other-numbers 9 10)
;=> (1 2 3 (4 5 6 7 8) 9 10)
user=> (concat '(1 2 3) other-numbers '(9 10))
;=> (1 2 3 4 5 6 7 8 9 10)
```

This concat version is a little unsatisfying. It works fine for this use case, but it might not be so great if we needed to inject the values into a syntax-quoted expression. Luckily, the unquote-splicing reader macro, ~@, gives us a succinct and fully syntax-quote-compatible way of doing the same thing:

advanced_mechanics/unquote_splicing_2.clj

```
user=> (def other-numbers '(4 5 6 7 8))
;=> #'user/other-numbers
user=> `(1 2 3 ~@other-numbers 9 10)
;=> (1 2 3 4 5 6 7 8 9 10)
```

And there are hooks in Clojure's syntax-quote implementation in the reader that look for clojure.core/unquote-splicing occurrences, just like with the normal unquote, to make this behavior possible.

Syntax-quote As a Macro?

Believe it or not, it's also possible for syntax-quote to be written as a macro, and at least two people have created projects doing just that:

- https://github.com/brandonbloom/backtick
- https://github.com/hiredman/syntax-quote

So one day we might very well see the syntax-quote arise from the dark depths of the Reader into the light of macro-land.

Besides the ability to interpolate values when syntax-quoting, there's an additional implication for symbols that occur within the syntax-quoted form. Take a look at the difference between this normal-quoted expression and its syntax-quoted sibling:

advanced_mechanics/syntax_quote_4.clj
```
user=> '(a b c)
;=> (a b c)
user=> `(a b c)
;=> (user/a user/b user/c)
```

With the syntax-quoted version, the symbols all include the namespaces where the syntax-quote appears! We say that these symbols are *namespace-qualified*. At first this might seem strange, but there's a good reason for it. Imagine you had this macro (which, as you'll see in Chapter 3, *Use Your Powers Wisely*, on page 27, is a bad idea to begin with, but we'll use it here for clarity):

advanced_mechanics/syntax_quote_5.clj
```
user=> (defmacro squares [xs] (list 'map '#(* % %) xs))
;=> #'user/squares
user=> (squares (range 10))
;=> (0 1 4 9 16 25 36 49 64 81)
```

Easy enough—we just map over the input collection, squaring each element.

What do you think would happen if we were to use this macro from a namespace where `map` meant something different?

advanced_mechanics/syntax_quote_6.clj
```
user=> (ns foo (:refer-clojure :exclude [map]))
;=> nil
foo=> (def map {:a 1 :b 2})
;=> #'foo/map
foo=> (user/squares (range 10))
;=> (0 1 2 3 4 5 6 7 8 9)
foo=> (user/squares :a)
```

```
;=> :a
foo=> (first (macroexpand '(user/squares (range 10))))
;=> map
foo=> ({:a 1 :b 2} :nonexistent-key :default-value)
;=> :default-value
```

So in a situation like this, since the verb map is an unqualified symbol, the squares macro call in the namespace foo will cause foo/map to be used as a function. The squaring function gets passed in *as the value to look up*, and the not-found default gets returned as the result in these cases. Well, this clearly isn't what we wanted when we wrote that macro, and that's where syntax-quote's namespace qualification leaps to the rescue. If we instead define the squares macro using a syntax-quote (or if we're into making things cumbersome, namespace-qualifying the map symbol ourselves), we don't have this problem:

advanced_mechanics/syntax_quote_7.clj

```
user=> (defmacro squares [xs] `(map #(* % %) ~xs))
;=> #'user/squares
user=> (squares (range 10))
;=> (0 1 4 9 16 25 36 49 64 81)
user=> (ns foo (:refer-clojure :exclude [map]))
;=> nil
foo=> (def map {:a 1 :b 2})
;=> #'foo/map
foo=> (user/squares (range 10))
;=> (0 1 4 9 16 25 36 49 64 81)
foo=> (first (macroexpand '(user/squares (range 10))))
;=> clojure.core/map
```

This is a great example of how Clojure's macro system tries to help you avoid shooting yourself in the foot. Use and master the syntax-quote, and your macro-writing life will be much easier. But we do need to go just a bit further to get you all the syntax-quoting knowledge you need.

If we had chosen to write the squares macro in a slightly different way, using the fn special form instead of the shortcut syntax (#(* % %)), we would have seen an error:

advanced_mechanics/gensym_1.clj

```
user=> (defmacro squares [xs] `(map (fn [x] (* x x)) ~xs))
;=> #'user/squares
user=> (squares (range 10))
;=> CompilerException java.lang.RuntimeException:
;  Can't use qualified name as parameter: user/x, compiling: (NO_SOURCE_PATH:1:1)
```

Ack! The namespace expansion has bitten us in this case, rather than helping. We could roll back to the list approach without too much pain here, but we'd really like to take advantage of syntax-quote's benefits. Given what you know about syntax-quoting and unquoting, you can extrapolate to a solution that puts a non-namespaced symbol in the macroexpansion:

advanced_mechanics/unhygienic_1.clj

```
user=> `(* ~'x ~'x)
;=> (clojure.core/* x x)
```

This solution would work fine for our use case: we could slap ~' in front of all the xs in the definition of squares and have a working implementation in no time:

advanced_mechanics/unhygienic_2.clj

```
user=> (defmacro squares [xs] `(map (fn [~'x] (* ~'x ~'x)) ~xs))
;=> #'user/squares
user=> (squares (range 10))
;=> (0 1 4 9 16 25 36 49 64 81)
```

And because the scope of x is limited to that anonymous squaring function, we don't have the namespace-related problems we saw before. However, let's expand our worldview to include macros *other* than the now-worn-out squares:

advanced_mechanics/unhygienic_3.clj

```
user=> (defmacro make-adder [x] `(fn [~'y] (+ ~x ~'y)))
;=> #'user/make-adder
user=> (macroexpand-1 '(make-adder 10))
;=> (clojure.core/fn [y] (clojure.core/+ 10 y))
```

So make-adder expands into a function definition, and that function adds the macro's argument to the function's argument. This has a strange result when we try to use it:

advanced_mechanics/unhygienic_4.clj

```
user=> (defmacro make-adder [x] `(fn [~'y] (+ ~x ~'y)))
;=> #'user/make-adder
user=> (def y 100)
;=> #'user/y
user=> ((make-adder (+ y 3)) 5)
;=> 13
```

Despite the fact that we defined y to be 100, it looks like the value being used for y is 5! Why do you think this is? By macroexpanding the make-adder call in question, we can see that we're creating a function that takes one argument named y, which shadows the (def y 100) definition:

advanced_mechanics/unhygienic_5.clj

```
user=> (macroexpand-1 '(make-adder (+ y 3)))
;=> (clojure.core/fn [y] (clojure.core/+ (+ y 3) y))
```

If you were trying to use this version of make-adder without digging into the macro definition, you'd think this was horribly broken, wouldn't you? We've accidentally allowed what's called *symbol capture*, where the macro has internally shadowed, or captured, some symbols that users of this macro might expect to be available when their expression is evaluated. There is a solution here, though, and it's called the *gensym*.

Approaching Hygiene with the Gensym

In order to avoid symbol capture issues like the one we just saw, Clojure gives us a few tools, all related to the gensym function. gensym's job is simple: it produces a symbol with a unique name. The names will look funny because the name needs to be unique for the application, but that's OK because we never need to type them into code:

advanced_mechanics/gensym_2.clj

```
user=> (gensym)
;=> G__671
user=> (gensym)
;=> G__674
user=> (gensym "xyz")
;=> xyz677
user=> (gensym "xyz")
;=> xyz680
```

As you can see, any given invocation of gensym gives a unique value back—so if you want to refer to the same one twice, you'll need to hold onto the value with a let binding or something similar. These generated symbols (gensyms) are extremely useful for macros, but because they're normal data, you can use them anywhere you'd use a symbol. In our make-adder macro earlier, we can't have user/y as a function argument, and we just saw that we don't want plain old y as a function argument, but we *can* use a gensym as a function argument:

advanced_mechanics/gensym_3.clj

```
(defmacro make-adder [x]
  (let [y (gensym)]
    `(fn [~y] (+ ~x ~y))))

user=> y
100
user=> ((make-adder (+ y 3)) 5)
108
```

Now this version uses the value of y that we'd expect as users of this macro. Notice that the let and gensym here are *outside* of the syntax-quote. It's a bit unfortunate that this is so verbose—let's use the more concise and built-in version. We'll use a feature called the *auto-gensym*, which just looks like a normal symbol with a pound sign (#) at the end, like a reverse hashtag:

advanced_mechanics/gensym_4.clj
```
(defmacro make-adder [x]
  `(fn [y#] (+ ~x y#)))

user=> y
100
user=> ((make-adder (+ y 3)) 5)
108
```

This, and not any of the previous ways we've done it, should be the tool you reach for when you need to bind a name within a macro. There are several other very similar cases of binding symbols to values where we also need to use gensyms to construct macros safely:

advanced_mechanics/gensym_5.clj
```
(defmacro safe-math-expression? [expression]
  `(try ~expression
        true
        (catch ArithmeticException e# false)))

;; clojure.core/and
(defmacro and
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and# (and ~@next) and#))))
```

Bindings set up by special forms like let, letfn, and try's catch clause have the same requirement as function arguments, so you should typically use the auto-gensym for these situations, too.

A lot of care has been taken in Clojure to make macro construction less error-prone. These variable-capture problems, along with the ability to get gensyms explicitly, have been around for a long time in Common Lisp, but it takes a bit of voodoo (see Doug Hoyte's *Let Over Lambda [Hoy08]*) to get something like Clojure's auto-gensym feature. It's worth noting that if you wander into the dark forests of nesting syntax-quotes, you (a) may never return, and (b) may want to take a look at unify-gensyms from Zach Tellman's Potemkin.[3]

Of course, anyone with a Scheme background is probably howling at this point because they have a *hygienic* macro system that makes unintended

3. https://github.com/ztellman/potemkin

variable capture impossible. Allowing variable capture when we really, really want it makes Clojure's macro system technically more dangerous than hygienic systems. By getting us most of the way there, Clojure gives us more safety than Common Lisp's macro system, along with the power to do variable capture when it makes for an elegant solution.

Secret Macro Voodoo

In addition to all of the syntax-quoting tools that Clojure provides for both macros and normal programming, we have two special values, &form and &env, that are available only within macros. Both of these allow us to introspect a bit on the way a macro gets used. Let's see what information we have available when we use them:

advanced_mechanics/secret_macro_variables_1.clj

```
(defmacro info-about-caller []
  (pprint {:form &form :env &env})
  `(println "macro was called!"))

user=> (info-about-caller)
;{:form (info-about-caller), :env nil}
;macro was called!
;=> nil
user=> (let [foo "bar"] (info-about-caller))
;{:form (info-about-caller),
; :env {foo #<LocalBinding clojure.lang.Compiler$LocalBinding@23ef55fb>}}
;macro was called!
;=> nil
user=> (let [foo "bar" baz "quux"] (info-about-caller))
;{:form (info-about-caller),
; :env
; {baz #<LocalBinding clojure.lang.Compiler$LocalBinding@3f68eac0>,
;  foo #<LocalBinding clojure.lang.Compiler$LocalBinding@55ab9655>}}
;macro was called!
;=> nil
```

The &env value seems pretty magical: it's a map of local variables, where the keys are symbols and the values are instances of some class in the Clojure compiler internals. It turns out that if we *really* wanted to get crazy, this gives us access to all sorts of interesting data during macroexpansion—things like the Java types of the arguments and how the locals were initialized. People typically use &env to just look at the keys (which are symbols), and inject them into the expanded macro. If we want to get ahold of a map of local names to local values, we could do something like this:

advanced_mechanics/secret_macro_variables_2.clj

```
(defmacro inspect-caller-locals []
  (->> (keys &env)
       (map (fn [k] [`'~k k]))
       (into {})))

user=> (inspect-caller-locals)
{}
user=> (let [foo "bar" baz "quux"] (inspect-caller-locals))
{baz "quux", foo "bar"}
```

There's a little bit of hairy quoting to get through here: `` `'~k ``. If we think through this carefully, though, it makes complete sense. We want to produce a quoted symbol for each local variable in the macroexpansion, but we want those quoted symbols to be the same as the ones in the &env map. This is getting a bit too fancy, but you'll see this kind of quoting in the wild sometimes, so it's worth wrapping your head around. However, a couple of equivalent, possibly more readable ways to say `` `'~k `` are `` `(quote ~k) `` and `(list 'quote k)`:

advanced_mechanics/secret_macro_variables_3.clj

```
(defmacro inspect-caller-locals-1 []
  (->> (keys &env)
       (map (fn [k] [`(quote ~k) k]))
       (into {})))

(defmacro inspect-caller-locals-2 []
  (->> (keys &env)
       (map (fn [k] [(list 'quote k) k]))
       (into {})))

user=> (inspect-caller-locals-1)
{}
user=> (inspect-caller-locals-2)
{}
user=> (let [foo "bar" baz "quux"] (inspect-caller-locals-1))
{baz "quux", foo "bar"}
user=> (let [foo "bar" baz "quux"] (inspect-caller-locals-2))
{baz "quux", foo "bar"}
```

These all behave exactly the same, so it's a matter of preference which one you'd choose. Keep in mind that whenever you encounter a confusing-looking quoting composition in a macro, you can always isolate the quoting bits at the REPL to see how they behave.

The &form special variable is a bit more straightforward. It contains the expression that was used to call the macro, which will always be a list since lists are how you call things in Clojure. This doesn't really appear to buy you much, because as the macro author you already know what the name of the macro is, and you already have access to the argument expressions that are passed in:

advanced_mechanics/secret_macro_variables_4.clj
```
(defmacro inspect-called-form [& arguments]
  {:form (list 'quote (cons 'inspect-called-form arguments))})

user=> (inspect-called-form 1 2 3)
;=> {:form (inspect-called-form 1 2 3)}
```

There are several additional benefits of having &form available, though. First, there's duplication in the way we did things previously, so if you change your mind about the macro name, you have to change two names in the macro definition. This definition is a lot of code as well—it's just more convenient to use &form than to type that whole expression whenever we want to inspect the details of the macro call. One of the really big benefits that makes &form special is the fact that when we have the actual form available to us, we also have all of the metadata attached to it:

advanced_mechanics/secret_macro_variables_5.clj
```
(defmacro inspect-called-form [& arguments]
  {:form (list 'quote &form)})

user=> ^{:doc "this is good stuff"} (inspect-called-form 1 2 3)
;=> {:form (inspect-called-form 1 2 3)}
user=> (meta (:form *1))
;=> {:doc "this is good stuff", :line 1, :column 1}
```

This can be kind of a big deal! It means that we can improve error messaging in macros by including line and column number information—this is by far the most common use of &form. But I'm sure you can imagine some other fancy ways to use metadata on expressions to give information to the macros that consume them.

Whew!

In this chapter, we've looked at all of the most advanced bits of macro construction. There's much more to know, but not in terms of syntax! At this point you already have the building blocks that will allow you to construct any macro you want to. In the rest of this book, you'll see how and why to write macros in your day-to-day work. But first, you'll see in the next chapter why macros aren't the solution for every problem, and how things can go wrong if you treat them as though they are.

CHAPTER 3

Use Your Powers Wisely

Now that you have the knowledge you need about *how* to write macros, it's time to start building a great sense for *when* to write them. Believe it or not, macros do have downsides. In fact, there's a popular saying in the Clojure world that you should never use a macro when a function will do. In this chapter you'll see some problems that are inherent to macros, a few outright mistakes to avoid, and some design decisions that can make maintenance harder down the road.

Macros Aren't Values

The first and most important drawback with macros is that we can't treat them as values. As principled functional programmers, we're quite used to thinking about functions as values. For example, we can pass functions as arguments to higher-order functions like map and filter to decouple looping logic from transformation and selection logic. When we use a macro instead of a function, we lose that ability:

beware/not_values_1.clj

```
user=> (defn square [x] (* x x))
;=> #'user/square
user=> (map square (range 10))
;=> (0 1 4 9 16 25 36 49 64 81)
user=> (defmacro square [x] `(* ~x ~x))
;=> #'user/square
user=> (map square (range 10))
;CompilerException java.lang.RuntimeException:
;  Can't take value of a macro: #'user/square, compiling: (NO_SOURCE_PATH:1:1)
```

This might not seem like a big deal when you just consider the definitions of the two squaring verbs we've defined here. But imagine you're not the author of square, just a user—in a functional language, it can be really annoying to run into this limitation. Oh, and if you've spotted another problem with the macro version of square, congrats! If not, don't worry—we'll take a closer look in *Macros Can Be Tricky to Get Right, on page 31*.

In this specific case, you can work around the issue by wrapping the macro in a function call:

beware/not_values_2.clj
```
user=> (defmacro square [x] `(* ~x ~x))
;=> #'user/square
user=> (map (fn [n] (square n)) (range 10))
;=> (0 1 4 9 16 25 36 49 64 81)
```

This works because when the anonymous function (fn [n] (square n)) gets compiled, the square expression gets macroexpanded, to (fn [n] (clojure.core/* n n)). And this is a perfectly reasonable function, so we don't have any problems with the compiler. This can be a decent workaround whenever you're able to put the name of the macro in the verb position. Of course, many macros do more complicated things that make this function-wrapping technique impossible. Even a simple macro can prevent you from doing this if it does anything interesting with its input expressions:

beware/not_values_3.clj
```
(defmacro do-multiplication [expression]
  (cons `* (rest expression)))

user=> (do-multiplication (+ 3 4))
;=> 12
user=> (map (fn [x] (do-multiplication x)) ['(+ 3 4) '(- 2 3)])
; CompilerException java.lang.IllegalArgumentException:
;   Don't know how to create ISeq from: clojure.lang.Symbol,
;   compiling:(NO_SOURCE_PATH:1:14)
```

You might have noticed that this macro is pretty silly—it just ignores the verb in its input and replaces it with multiplication. It assumes that its input expression is something that we can use to create a sequence. So when Clojure tries to macroexpand (do-multiplication x), it can't do its job, because x is a symbol. And as the error message tells us, Clojure doesn't know how to turn a symbol into a sequence. At the risk of belaboring the point, macros take *code* as input–they don't (and *can't*) know what values will be substituted in place of the symbols in the code at runtime.

The only way to map across expressions to get what we want here is to convert do-multiplication to a function that outputs an expression, and then either write a little interpreter or use the dreaded eval to compute the results. Neither of these is anything resembling a macro-as-a-value. So while we can cheat and wrap some extremely simple macros in functions, in the general case we're left entirely without a way to treat them as values.

Macros Can Be Contagious

Next we'll look at an important consequence of macros not being values: macros that take a variable number of arguments can *infect* their callers, forcing the author to write more macros instead of functions. Let's take a look at an example and think about the implications for calling code:

beware/contagious_1.clj
```
(require '[clojure.string :as string])
(defmacro log [& args]
  `(println (str "[INFO] " (string/join " : " ~(vec args)))))

user=> (log "that went well")
;[INFO] that went well
;=> nil
user=> (log "item #1 created" "by user #42")
; [INFO] item #1 created : by user #42
;=> nil
```

If you recall what you learned in Chapter 2, *Advance Your Macro Techniques*, on page 15, you'll notice that we're converting args to a vector instead of just using ~args. Why? Because we want args to be a sequential thing in the macroexpanded code, but not an expression to be evaluated. If we didn't use a vector, Clojure would stick the args *sequence* (a list, for all intents and purposes) right into the macroexpanded code and treat the first sequence element as the verb at runtime. As you can imagine, that would go badly if we tried to use a string as the first argument. This macro works fine and is easy to call directly, but suppose we find ourselves holding onto some arbitrary collection of messages, perhaps via input to another function:

beware/contagious_2.clj
```
(defn send-email [user messages]
  (Thread/sleep 1000)) ;; this would send email in a real implementation

(def admin-user "kathy@example.com")
(def current-user "colin@example.com")

(defn notify-everyone [messages]
  (apply log messages)
  (send-email admin-user messages)
  (send-email current-user messages))

; CompilerException java.lang.RuntimeException:
;   Can't take value of a macro: #'user/log, compiling:(NO_SOURCE_PATH:2:3)
```

Macros Actually *Are* Functions!

As you saw in Chapter 1, *Build a Solid Foundation*, on page 1, you can think of macros as functions that transform one piece of code into another. And it turns out that with a bit of trickery, you can get ahold of those functions and use *them* as values. Whether you'll ever really need to do such a thing is questionable, but it gives you a clearer picture of what's really going on. Let's look at this trivial square macro again in a different way:

beware/macro_as_function_1.clj

```
user=> (defmacro square [x] `(* ~x ~x))
;=> #'user/square
user=> @#'square
;=> #<user$square user$square@2a717ef5>
user=> (fn? @#'square)
;=> true
```

So we can actually pull a function out of the var where we defined the macro by deref'ing the var! But what happens if we try to use that function?

beware/macro_as_function_2.clj

```
user=> (@#'square 9)
; ArityException Wrong number of args (1) passed to: user$square
;  clojure.lang.AFn.throwArity (AFn.java:437)
```

Clojure is complaining that we're passing one argument instead of the right number, but what is the right number of arguments for this function? It turns out this function takes two initial arguments, a form and an environment (remember &form and &env from secret macro voodoo on page 24?), as well as whatever arguments the macro defines (just one in our case). This macro doesn't happen to care about &form &env, so we can just pass dummy values:

beware/macro_as_function_3.clj

```
user=> (@#'square nil nil 9)
;=> (clojure.core/* 9 9)
```

Aha! This looks like macroexpansion, and in fact it turns out this is precisely the function that gets called by the Clojure compiler during macroexpansion! But as far as I'm concerned, it's an implementation detail that this is available to us at runtime. It's much clearer to use (macroexpand-1 '(square 9)). Hopefully this gives you a better picture of what a macro actually *is* under the hood.

We might have seen this one coming: anytime you see a macro name appear anywhere except the first position in a list, warning bells should go off in your head, since we can't treat macros as values. But how can we solve this issue? If we knew that there were exactly two messages, or three messages, etc., we might be able to pull specific elements from the input sequence to pass to the macro.

Take a few minutes to write another macro for this use case, assuming that you can't change the log macro or create a replacement.

Perhaps you came up with something like this:

beware/contagious_3.clj
```
(defmacro notify-everyone [messages]
  `(do
     (send-email admin-user ~messages)
     (send-email current-user ~messages)
     (log ~@messages)))

user=> (notify-everyone ["item #1 processed" "by worker #72"])
;[INFO] item #1 processed : by worker #72
;=> nil
```

Note that we need to wrap the three expressions in a do, since a macroexpansion can only return *one* expression. If we left that out, we'd lose those first two expressions in the macroexpanded code. This macro implementation does work. But of course now we can't use notify-everyone as a value, so we impose the same restrictions on users of our new code. Macros appear to be taking over our code!

In reality, the far better solution here is one where log isn't a macro at all. We don't *need* a macro here:

beware/contagious_4.clj
```
(require '[clojure.string :as string])
(defn log [& args]
  (println (str "[INFO] " (string/join " : " args))))

user=> (log "hi" "there")
;[INFO] hi : there
;=> nil

user=> (apply log ["hi" "there"])
;[INFO] hi : there
;=> nil
```

Of course, sometimes there are cases where we really do want a macro with varargs, or where it's not so obvious that there's an easy way to use a function instead. And in those cases, we really want to be sure that they're worth the cost of forcing clients to use macros as well.

Macros Can Be Tricky to Get Right

You've already seen plenty of subtleties with macros, such as the need to avoid symbol capture with gensyms, and there's more to come. While you're in the great position of knowing all the language's rules for macro construction, you may still be unclear about the *implications* of what you know. Let's look at a few more ways you might accidentally cause yourself grief by making the wrong assumptions about macro code.

Here's a re-implementation of the clojure.core/and macro that seems fairly straightforward.

beware/and_1.clj

```
(defmacro our-and
  ([] true)
  ([x] x)
  ([x & next]
   `(if ~x (our-and ~@next) ~x)))

user=> (our-and true true)
;=> true
user=> (our-and true false)
;=> false
user=> (our-and true true false)
;=> false
user=> (our-and true true nil)
;=> nil
user=> (our-and 1 2 3)
;=> 3
```

It appears to have the same behavior as clojure.core/and, returning its argument for the base case of the recursion, returning the first non-truthy value if it fails, and recursing when it doesn't fail. But what if the caller passes an expression to our-and?

beware/and_2.clj

```
user=> (our-and (do (println "hi there") (= 1 2)) (= 3 4))
;hi there
;hi there
;=> false
```

The expression we passed in is actually evaluated twice! Looking back at the macro implementation and a macroexpansion, it's clear why this happens: because the macroexpansion inserts the expression, in its entirety, in two places:

beware/and_3.clj

```
user=> (macroexpand-1 '(our-and (do (println "hi there") (= 1 2)) (= 3 4)))
;=> (if (do (println "hi there") (= 1 2))
;     (user/our-and (= 3 4))
;     (do (println "hi there") (= 1 2)))
```

In our examples, we still get the right answer, but the side effects are troublesome. Imagine what would happen if they were writing to the production database instead of printing log messages for programmers! Of course, ideally we wouldn't have them, but in practice Clojure programmers do use I/O, atoms, refs, and other side-effecting constructs from time to time.

So we have two choices here: we can either (a) tell clients of our-and to assume that their expression may be evaluated multiple times (via documentation or word of mouth), or (b) fix the macro to evaluate the expressions only once. The choice is clear: it's always preferable to make macros less surprising when we have the opportunity. Unless we're building a specialized macro expressly intended to evaluate zero, multiple, or an indeterminate number of times, we should evaluate arguments exactly once. So that's our default, but part of the power of macros is in being able to choose. The important thing to me as a macro user is that I know what the evaluation semantics are, and I tend to expect arguments to be evaluated once unless I have documentation that says otherwise.

In the case of our-and, we can easily fix it up to evaluate its arguments only once:

beware/and_4.clj
```
(defmacro our-and
  ([] true)
  ([x] x)
  ([x & next]
   `(if ~x (our-and ~@next) ~x)))

(defmacro our-and-fixed
  ([] true)
  ([x] x)
  ([x & next]
   `(let [arg# ~x]
      (if arg# (our-and-fixed ~@next) arg#))))

user=> (our-and-fixed (do (println "hi there") (= 1 2)) (= 3 4))
;hi there
;=> false
```

There's nothing magical here: extracting a local up to a let binding is the same thing we'd do if we had a duplicated expression inside a function to avoid evaluating it twice.

Proceed with Caution

My point here is only partly that we, as macro authors, should avoid executing input expressions multiple times unless we really, really mean to. But more importantly, it's that when we write macros, it's great when we can shield the people who use those macros from having to dig into the macro implementations to see why some surprising thing happens. Our macros are being invited to expand into users' namespaces, and we should appreciate and respect that invitation by making as little of a mess as possible.

This chapter hasn't been an exhaustive list of the ways that macros might trip you up. Macroexpand-time errors can sometimes generate confusing compiler errors without significant thought given to error cases. Errors in successfully macroexpanded code leave you with stack traces unaware of the line number in the macro definition that caused the error. Helper functions that need to be called by macroexpanded code must be public (no `defn-` or `^:private`) in order for macro calls to work from other namespaces. We've seen other potential stumbling blocks in the last couple of chapters, and we'll come across more. These are trade-offs we must accept on behalf of our team whenever we decide to use a macro.

But the thing is: sometimes a function *won't* do. Although the humble function is a pretty awesome and versatile tool, it isn't perfect for every job, and so sometimes a macro is just what the doctor ordered. In the rest of this book, we'll look at good reasons to use macros, despite the problems they can cause.

CHAPTER 4

Evaluate Code in Context

So far you've seen the building blocks for writing macros, and you've learned why they aren't the ideal solution for every problem. In the remainder of this book, we'll look at several broad categories of macros to learn how you can clarify your code's intentions, add your own control flow mechanisms, and take advantage of your other macro superpowers.

In this chapter, you'll see macros that wrap the code they're given into a new context for evaluating them. For example, you might want to evaluate a given expression with dynamic bindings, within a try/catch block, or where a resource is opened and then cleaned up. In this category of macros, you're not doing anything too fancy with the input expressions—they're simply wrapped with some logic that you want to abstract away from the caller. You'll see some ways to get rid of hard-to-remove structural duplication like the following example, where the small bit of code that varies ([INFO] vs. [ERROR]) is surrounded by a bunch of code that's duplicated from one function to the next:

context/structural_duplication.clj
```
(require '[clojure.java.io :as io])
(defn info-to-file [path text]
  (let [file (io/writer path :append true)]
    (try
      (binding [*out* file]
        (println "[INFO]" text))
      (finally
        (.close file)))))

(defn error-to-file [path text]
  (let [file (io/writer path :append true)]
    (try
      (binding [*out* file]
        (println "[ERROR]" text))
      (finally
        (.close file)))))
```

Does this kind of duplication make your programmer code-senses tingle? It sure does for me. We've got the things that change all mixed up with the things that stay the same, so it's easy to lose sight of what's important. In this chapter, we'll see how to strip away the boilerplate code and leave only what's necessary. And as we go, we'll build upon the work we started in the previous chapter on page 27 to develop a sense of how to accomplish some of these same goals with functions.

Dynamic Bindings

The Clojure community loves *lexical scoping*, where the value of a symbol in an expression is solely determined by its placement in the code, not by its position on the call stack. That is, we like functions to accept arguments for each piece of data that may vary from one invocation to the next.

context/circle_area_lexical.clj

```
(defn circle-area [radius]
  (* Math/PI (* radius radius)))

(circle-area 10.0)
;=> 314.1592653589793
```

Here radius is a lexically scoped local variable: its value is determined solely by the argument passed into the function. That makes it easy to see at a glance what the dependencies are. Function parameters, along with let and loop bindings, are common examples of lexical binding.

Can you imagine why many of us prefer lexical scoping over *dynamic scoping*, where the values that a function uses to complete its evaluation are injected from outside of the function definition?

context/circle_area_dynamic.clj

```
(declare ^:dynamic *radius*)
(defn circle-area []
  (* Math/PI (* *radius* *radius*)))

(binding [*radius* 10.0]
  (circle-area))
;=> 314.1592653589793
```

Note that the asterisks in *radius* are called *earmuffs* and are just a naming convention for dynamically scoped vars in Clojure—they're not required by the language.

Despite the conventional preference for lexical scoping, there are good reasons that Clojure has dynamic scope as an option. It gives us an escape hatch when it would be too cumbersome to pass values through a chain of functions that are otherwise oblivious to the value some lower-level function needs. Prime examples of this convenience are the I/O vars *out*, *in*, and *err*. In Unix, we're used to the idea of changing the source and destination of the stdout, stdin, and stderr streams by using pipes and command-line redirection. In scoping terms, rebinding these Clojure vars is very similar to how we redirect Unix input and output streams to wire programs together.

Whenever we use any of Clojure's printing functions, we're actually using dynamic bindings:

context/log.clj
```
(defn log [message]
  (let [timestamp (.format (java.text.SimpleDateFormat. "yyyy-MM-dd'T'HH:mmZ")
                           (java.util.Date.))]
    (println timestamp "[INFO]" message)))
```

It's nice for log to be able to call println, which uses *out*, without having to know what *out* is actually pointing to. This way, log only has to know about println, not println's dependency *out*. So if we want to vary where things are printed, we can just rebind *out* whenever we want to call code that uses log.

context/log_to_file.clj
```
(defn process-events [events]
  (doseq [event events]
    ;; do some meaningful work based on the event
    (log (format "Event %s has been processed" (:id event)))))

(let [file (java.io.File. (System/getProperty "user.home") "event-stream.log")]
  (with-open [file (clojure.java.io/writer file :append true)]
    (binding [*out* file]
      (process-events [{:id 88896} {:id 88898}]))))
```

Here we've decided to re-route *out* to a log file since we may want to take a look at it later. This works just fine, but it's kind of noisy, isn't it? If we wanted to do this in several places in code, we'd need to repeat all this wordy code, and that'd be a shame. Besides, the main point of this code is to process events, but that intent is buried inside these with-open and binding expressions that set up the output stream. With a couple minutes of investment, we can write a macro to abstract that pattern away:

context/with_out_file.clj
```
(defmacro with-out-file [file & body]
  `(with-open [writer# (clojure.java.io/writer ~file :append true)]
     (binding [*out* writer#]
       ~@body)))

(let [file (java.io.File. (System/getProperty "user.home") "event-stream.log")]
  (with-out-file file
    (process-events [{:id 88894} {:id 88895} {:id 88897}])
    (process-events [{:id 88896} {:id 88898}])))
```

This feels more aligned with the problem domain. We no longer have to specify the details of how the output stream is created and bound, just the file where we want to append output. Setting up bindings like those in with-out-file is a very common use case for macros in the wild in real-world scenarios. Clojure itself has several built-in macros that are similar to with-out-file. The *most* similar is probably with-out-str:

context/with_out_str.clj

```
(defmacro with-out-str
  "Evaluates exprs in a context in which *out* is bound to a fresh
  StringWriter.  Returns the string created by any nested printing
  calls."
  {:added "1.0"}
  [& body]
  `(let [s# (new java.io.StringWriter)]
     (binding [*out* s#]
       ~@body
       (str s#))))
```

Looks kind of familiar, right? with-out-str evaluates the body form with *out* rebound, in order to collect output and return it as a string. This is very useful for unit testing I/O produced by a function, along with its sibling with-in-str, which does something similar for input:

context/with_in_str.clj

```
(defmacro with-in-str
  "Evaluates body in a context in which *in* is bound to a fresh
  StringReader initialized with the string s."
  {:added "1.0"}
  [s & body]
  `(with-open [s# (-> (java.io.StringReader. ~s)
                      clojure.lang.LineNumberingPushbackReader.)]
     (binding [*in* s#]
       ~@body)))

(defn join-input-lines [separator]
  (print (clojure.string/replace (slurp *in*) "\n" ",")))

(let [result (with-in-str "hello there\nhi\nsup\nohai"
               (with-out-str
                 (join-input-lines ",")))]
  (assert (= "hello there,hi,sup,ohai" result)))
```

Even though the function join-input-lines acts directly on *in* and *out*, we're able to set up fake versions of those streams that allow us to inject the input we want and to look at the output to verify what happened. And this is all because those macros allowed us to evaluate join-input-lines in a context of our choosing.

Now's a great time to go back to our duplication-heavy code from the start of the chapter on page 35 and write a macro to remove the duplication.

A Non-Macro Approach

There are other ways besides macros to solve the problem of contextual evaluation, however. A version of with-out-file built only with functions is also a reasonable option if we're willing to modify the calling syntax slightly:

context/with_out_file_fn.clj
```
(defn with-out-file [file body-fn]
  (with-open [writer (clojure.java.io/writer file :append true)]
    (binding [*out* writer]
      (body-fn))))

(let [file (java.io.File. (System/getProperty "user.home") "event-stream.log")]
  (with-out-file file
    (fn []
      (process-events [{:id 88894} {:id 88895} {:id 88897}])
      (process-events [{:id 88896} {:id 88898}]))))
```

This is actually not bad at all. There aren't that many differences in this case, and it's not clear that one is inherently better than the other from a syntax perspective. From a code concision perspective, the macro version is nicer—we don't need the (fn [] ...) wrapping the input code—but there are always trade-offs. For example, if we wanted to pass with-out-file around as a value (a higher-order function), we wouldn't be able to use the macro version.

In general, a macro that consumes code-to-be-evaluated can usually be re-written as a function that takes a *thunk*—a function with no arguments that delays code evaluation. This macro-replacing function can choose to evaluate the thunk multiple times, or not at all, just like the macro could. The cost is a small one in code clarity on the caller's side, but I suspect it's why with-* macros seem to be so common in practice despite the fact that they could be replaced by functions. Using a macro allows your callers to avoid the (fn [] ...) boilerplate that a higher-order function would require.

Luckily, we can get the best of both worlds by wrapping a macro as a thin wrapper around the function version:

context/with_out_file_fn_wrapper.clj
```
(defn with-out-file-fn [file body-fn]
  (with-open [writer (clojure.java.io/writer file :append true)]
    (binding [*out* writer]
      (body-fn))))
```

```clojure
(defmacro with-out-file [file & body]
  `(with-out-file-fn ~file
     (fn [] ~@body)))

(let [file (java.io.File. (System/getProperty "user.home") "event-stream.log")]
  (with-out-file file
    (process-events [{:id 88894} {:id 88895} {:id 88897}])
    (process-events [{:id 88896} {:id 88898}])))
```

This way, users have the choice to use the more concise macro version *or* the more flexible function version. When this style of wrapping is possible, it's pretty much always a good idea. Giving your teammates and library users (and your future self!) the option to use a function when they want to is just a nice thing to do.

Evaluating (or Not) in Time and Place

One of the least complex (but most-used) macros in Clojure is comment, which entirely avoids evaluating the code contained in it:

context/comment.clj

```clojure
(defmacro comment
  "Ignores body, yields nil"
  {:added "1.0"}
  [& body])

(comment
  (println "wow")
  (println "this macro is incredible"))
;=> nil

(+ 1 2) ; this is another type of comment
(+ 1 2) #_(println "this is yet another")
```

There are a couple of other commenting mechanisms in Clojure, but comment is the only one that's a macro (the others are built into the Clojure reader). The return value of comment is always nil, and none of the code passed to it is ever evaluated. Since it's a macro, the code passed in has to be syntactically correct so that it is readable. One upside of using the comment macro is that you get syntax highlighting in your editor of choice. So it's fairly common in practice to use comment to show examples of how to use code in a particular namespace. I don't personally use it much, as I prefer to use unit tests that execute and don't go out of date instead. But it's nice to have it available when I need it. So comment gives us a degenerate case of evaluating code in some other context, in the same way that /dev/null gives us a degenerate place to send a stream of output.

A more interesting example of using macros to evaluate code in a different context is dosync, the entry point to Clojure's software transactional memory (STM) system:

context/dosync.clj

```
(defmacro dosync
  "Runs the exprs (in an implicit do) in a transaction that encompasses
  exprs and any nested calls.  Starts a transaction if none is already
  running on this thread. Any uncaught exception will abort the
  transaction and flow out of dosync. The exprs may be run more than
  once, but any effects on Refs will be atomic."
  {:added "1.0"}
  [& exprs]
  `(sync nil ~@exprs))

(defmacro sync
  "transaction-flags => TBD, pass nil for now

  Runs the exprs (in an implicit do) in a transaction that encompasses
  exprs and any nested calls.  Starts a transaction if none is already
  running on this thread. Any uncaught exception will abort the
  transaction and flow out of sync. The exprs may be run more than
  once, but any effects on Refs will be atomic."
  {:added "1.0"}
  [flags-ignored-for-now & body]
  `(. clojure.lang.LockingTransaction
      (runInTransaction (fn [] ~@body))))

(def ant-1 (ref {:id 1 :x 0 :y 0}))
(def ant-2 (ref {:id 2 :x 10 :y 10}))

(dosync
  (alter ant-1 update-in [:x] inc)
  (alter ant-1 update-in [:y] inc)
  (alter ant-2 update-in [:x] dec)
  (alter ant-2 update-in [:y] dec))
```

Any Clojure STM example you've ever looked at allows you to send code to be evaluated in a transaction, to be retried in case of conflicts, and the dosync macro (along with the underlying sync) bundles the argument expressions into a suitable form (a thunk). I find most folks don't need to use the STM system that often, so dosync is relatively rare outside of Clojure language introductions, but it's really handy to have it generate code for you. Otherwise you'd need to type out (or set up editor automation for) the LockingTransaction, anonymous function creation, and the rest.

Because we've got the power to say when and in what context we want to evaluate a bit of code, we can even do things like sending code to be evaluated in a future[1] (an instance of java.util.concurrent.Future, specifically). The future macro from clojure.core wraps its argument expressions up into a function and submits that function to an Executor (which works since Clojure functions implement java.util.concurrent.Callable):

context/future.clj
```
(defmacro future
  "Takes a body of expressions and yields a future object that will
  invoke the body in another thread, and will cache the result and
  return it on all subsequent calls to deref/@. If the computation has
  not yet finished, calls to deref/@ will block, unless the variant of
  deref with timeout is used. See also - realized?."
  {:added "1.1"}
  [& body] `(future-call (^{:once true} fn* [] ~@body)))

(def f (future (Thread/sleep 5000)
               (println "done!")
               (+ 41 1)))

@f
;=> 42 (after sleeping 5 seconds and then printing "done!")
```

There are a couple of interesting things going on here. The first is that the macro uses an underlying function, future-call, to do most of the work. In fact, future itself is just a thin wrapper around that function. Just like our eventual with-out-file implementation, this gives us the best of both worlds. We can use the underlying future-call function whenever we need it, but we still have the succinctness of the macro version for normal use. The second interesting thing is the {:once true} metadata on the fn*. This is a fairly low-level compiler feature to help us avoid accidental memory leaks. It makes sure that any closed-over locals in the function get cleared after the function is called. This way, Clojure doesn't have to hold onto those values indefinitely, wrongly thinking that you might call the function again. Anytime we create functions that we *know* will only be invoked once (or if we at least know we don't need closed-over locals in the function invocation), it's a good idea to use (^:once fn* [] ...) instead of the plain (fn [] ...), to avoid leaking memory. ^:keyword-here, by the way, is just a shorthand for the common ^{:keyword-here true} pattern for Clojure metadata.

context/once.clj
```
(let [x :a
      f (^:once fn* [] (println x))]
  (f)  ;; prints :a
  (f)) ;; prints nil
```

1. http://en.wikipedia.org/wiki/Futures_and_promises

```
(let [x :a
      f (fn [] (println x))]
  (f)  ;; prints :a
  (f)) ;; prints :a
```

In the ^:once-decorated version, after the first evaluation of f, the local x gets cleared, so the second evaluation has x bound to nil. Clearly this is useful only when the function will execute just once. Note that it's necessary to use fn* here, not fn, to get the benefit of this optimization. Of course, if the function will execute multiple times, or if you don't mind leaking the memory that your locals consume, you can always use the usual fn for your function definitions.

Rescuing Errors

Another common use case for macros is to limit the reach of any errors we encounter during evaluation. Unit testing libraries typically *have* to do this in order to capture failures while continuing to run other tests. In clojure.test, for instance, the is macro uses an internal macro try-expr to catch exceptions and report them to the test-running infrastructure.

context/try_expr.clj

```
(defmacro try-expr [msg form]
  `(try ~(assert-expr msg form)
        (catch Throwable t#
          (do-report {:type :error, :message ~msg,
                      :expected '~form, :actual t#}))))
(defmacro is
  ([form] `(is ~form nil))
  ([form msg] `(try-expr ~msg ~form)))
```

So when the form is actually evaluated within the assert-expr, if an unexpected exception propagates due to some failing code, we definitely want to (a) prevent that failure from propagating, so that tests can continue, and (b) capture information about the failure in order to report it to the user. Wrapping the evaluation in a try-catch block is really the only way to go. This allows us to write test cases like this:

context/test.clj

```
(require '[clojure.test :refer [is]])
(is (= 1 1))
;=> true

(is (= 1 (do (throw (Exception.)) 1)))
; ERROR in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:1)
; expected: (= 1 (throw (Exception.)))
;   actual: java.lang.Exception: null
;=> nil
```

Here we could *almost* avoid macros by requiring clients to pass functions rather than expressions. This would require users to pass a thunk instead of just an expression. But there's a problem with the following function version of try-expr. Can you spot it?

context/try_expr_fn.clj

```
(require '[clojure.test :as test])
(defn try-expr [msg f]
  (try (eval (test/assert-expr msg (f)))
    (catch Throwable t
      (test/do-report {:type :error, :message msg,
                       :expected f, :actual t}))))

(defn our-is
  ([f] (our-is (f) nil))
  ([f msg] (test/try-expr msg f)))

(our-is (fn [] (= 1 1)))
;=> true

(our-is (fn [] (= 1 2)))
; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:3)
; expected: f
;   actual: false
;=> false
```

Even assuming we could replace the ugly eval by updating assert-expr and all its dependencies with function versions to avoid the eval, we're left with the horrible error message you see here. When we stop and think about what the try-expr macro is actually doing for us when it fails, it's clear that nothing but a macro would do. When we report this failure to the test runner, we're giving it not only the message from the test and the exception we encountered, but also the complete expression that failed! And with a macro, we have access to that original expression, which we can evaluate (or, in this case, pass down to assert-expr to be evaluated), or use as an expression, depending on our needs. Ironically, while this example shows us a way macros can help us to provide excellent, context-specific error messages, in practice, macros are often a source for more confusing error messages.

Cleaning Up Resources

In many languages, including Java, safely cleaning up resources is a bit tricky if you haven't done it before. Take reading from a file as an example: first we need to open the file resource, then do our work with it, and then when we're finished, we need to close the resource. It's just a three-step process, but there's a lot that can go wrong. There might be an exception when reading from the file. There might have been an exception when *opening* the file. For this reason, it's not uncommon to see Java like the following code sample to read a file's contents into a string:

context/teardown_filestream.java
```
public String readFile(String filePath) throws IOException {
  FileInputStream fileStream = null;
  StringBuilder contents = new StringBuilder();
  byte[] buffer = new byte[4096];
  try {
    fileStream = new FileInputStream(filePath);
    while (fileStream.read(buffer) != -1) {
      contents.append(new String(buffer));
    }
  } finally {
    if (fileStream != null) {
      fileStream.close();
    }
  }
  return contents.toString();
}
```

Let's ignore for the purposes of this discussion the fact that reading a file into a string is an engineering mistake (because what if we're given a gigantic file?). Let's think about string appending as a special case of a general problem where we have a resource we need to clean up. Now, granted, Java 7 has finally (zing!) introduced a language feature to make this cleaner: try-with-resources.[2] But users of many other language (and Java users stuck on Java 6 or below) aren't so lucky. The core algorithm here seems particularly obscured by this try/finally pattern and the separate fileStream declaration to work around the block scoping. With macros to allow contextual evaluation, one nice solution is built into a macro, with-open:

context/with_open.clj
```
(defmacro with-open
  "bindings => [name init ...]

  Evaluates body in a try expression with names bound to the values
  of the inits, and a finally clause that calls (.close name) on each
  name in reverse order."
  {:added "1.0"}
  [bindings & body]
  (assert-args
     (vector? bindings) "a vector for its binding"
     (even? (count bindings)) "an even number of forms in binding vector")
```

2. http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

```
  (cond
    (= (count bindings) 0) `(do ~@body)
    (symbol? (bindings 0)) `(let ~(subvec bindings 0 2)
                             (try
                               (with-open ~(subvec bindings 2) ~@body)
                               (finally
                                 (. ~(bindings 0) close))))
    :else (throw (IllegalArgumentException.
                   "with-open only allows Symbols in bindings"))))
```

context/with_open_client.clj

```
(import 'java.io.FileInputStream)
(defn read-file [file-path]
  (let [buffer (byte-array 4096)
        contents (StringBuilder.)]
    (with-open [file-stream (FileInputStream. file-path)]
      (while (not= -1 (.read file-stream buffer))
        (.append contents (String. buffer))))
    (str contents)))
```

You've seen with-open in use a few times already, as part of the implementation of with-out-file and with-in-str. Its definition is a bit more involved than some of the macros you've seen, but it's nothing scary. It checks a few assumptions up front with assert-args, throwing an exception if the binding expression is not a symbol, and behaves recursively in case there are multiple resources to be cleaned up. Recursive macro definitions like this one can be very useful, though as with any other recursive algorithm it's important to ensure that the recursion will eventually terminate. Typically we want each recursive call to have something of a *smaller* value, moving toward the base case.

Notice that the details of the try/finally, along with the null check, have been abstracted away into the with-open macro. Otherwise, this is a pretty faithful translation of the Java, but all the ceremony around cleaning up the stream has just gone away. And it's not hard to imagine a version of with-open that would also wrap a try/catch around the .close call, in order to swallow or otherwise specially handle an exception when trying to *close* the resource.

Abstracting Away the Details

The main point here, and in the other examples in this chapter, is that macros allow you to eliminate the noisy details of cleaning up an open resource, or rescuing errors, or setting up dynamic bindings or other contexts for evaluation. By writing macros to abstract away these contextual details, you can clarify the core operations you're performing to make your code's purpose more obvious to the people who will read it in the future (including yourself!).

This is a core ability of macros, and you'll see lots of overlap between this category and ones we'll cover later.

Now that you've seen some of the ways you can use macros to abstract away the context in which your code will evaluate, we'll look at ways to use both abstraction and more macro-specific tools to improve the runtime performance of our Clojure code.

CHAPTER 5

Speed Up Your Systems

So there you are, deep in the midst of a task to analyze your website traffic logs. You were hoping to have finished it yesterday, so your heart leaps as you realize you finally have everything tested and working. Now it's time to kick off the task on the full dataset, and wait. And wait. And wait. How long will it take to finish? A day? A week? Until the heat death of the universe? After about an hour, things are looking pretty bleak.

Now, Clojure is fast. One of the language's guiding principles is that it should be useful wherever Java is useful, including use cases that require high performance, like your log analysis work. As with any other programming language, there are times when we'll need a bag of tricks to make our code as fast as possible. Sometimes, performance hacks like type hints and other Java interop artifacts result in uglier code. But with macros, we can hide the noise and maintain the beauty of our code. You'll see how in this chapter, and you'll also see how you can *entirely avoid* runtime costs in some situations with macros. But first you need to make sure you optimize the right things by benchmarking.

Benchmarking Your Code

No discussion about performance would be complete (or to my way of thinking, even worth beginning) without that famous quote from *Donald Knuth [Knu74]*:

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

If we apply our time-limited efforts randomly around the codebase, it may not improve any bottlenecks that exist. My experience tends to align with Knuth's advice here: it saves us time, and a lot of wasted effort, to first find problem areas in our programs. If our program doesn't need to be fast, *any* performance work is unnecessary, or at least, premature. If our web application spends the majority of its time on I/O tasks, such as transferring data from the database to the application itself and pushing it out to the user's web browser, any speed improvements we make will not noticeably impact users. When we make performance improvements, it's critical we have metrics showing that we made an impact.

Happily, there's a wonderful benchmarking tool that can help us make decisions about what parts of our code to optimize and how to optimize them. Hugo Duncan's Criterium[1] is the benchmarking library of choice for many Clojure developers, and with good reason. Getting started with Criterium is pretty easy, since it's just a matter of adding it to your Leiningen dependencies. Criterium runs your code many times in order to take JIT and garbage collection into account. It also collects statistics on the running times to provide more detailed information, such as standard deviation and outliers. (Don't worry about the cryptic :jvm-opts ^:replace [] line—it just enables some JVM JIT optimizations.)

As an aside, the optimizations result in a slightly longer startup time, but it's worth taking that extra time in performance testing and other real-world production scenarios.

performance/project.clj
```
(defproject foo "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.6.0"]]
  :jvm-opts ^:replace []
  :profiles {:dev {:dependencies [[criterium "0.4.2"]]}})
```

Criterium actually uses macros internally for some of the same reasons we discussed in Chapter 4, *Evaluate Code in Context*, on page 35, but here we just want to measure performance improvements. Let's pop open a REPL inside a project with project.clj and see how to use Criterium to measure the code's speed before optimizing:

performance/criterium_run.clj
```
user=> (use 'criterium.core)
user=> (defn length-1 [x] (.length x))
;=> #'user/length-1
user=> (bench (length-1 "hi there!"))
;Evaluation count : 26255400 in 60 samples of 437590 calls.
;              Execution time mean : 3.250388 µs
;     Execution time std-deviation : 850.690042 ns
;    Execution time lower quantile : 2.303419 µs ( 2.5%)
;    Execution time upper quantile : 5.038536 µs (97.5%)
```

1. https://github.com/hugoduncan/criterium

```
;                   Overhead used : 2.193109 ns
;
;Found 1 outliers in 60 samples (1.6667 %)
;        low-severe       1 (1.6667 %)
; Variance from outliers : 94.6569 % Variance is severely inflated by outliers
;=> nil
```

The bench macro executes the given expression several times, so it may take a while before we see the output. We can wrap it with (with-progress-reporting ...) to see more feedback while the benchmarking system is running. From here on out, we'll only show the mean and standard deviation times, but you can try them out in your REPL if you want statistics on outliers and other details. If you have some logs to analyze, why not run that through Criterium and see how it goes?

A few microseconds might not sound like a lot, but when you're in the middle of a graph search algorithm or some other CPU-intensive task, it may be a significant bottleneck. We can pretty easily knock off some time by avoiding reflection (a Java mechanism that determines dynamically how a method will dispatch at runtime). Let's see what happens when we add a type hint (^String) to the argument to length-1, or even just call .length on the literal string directly, since these cases don't require reflection:

performance/no_reflection.clj

```
user=> (defn length-2 [^String x] (.length x))
;=> #'user/length-2
user=> (bench (length-2 "hi there!"))
;              Execution time mean : 1.476211 ns
;    Execution time std-deviation : 0.295418 ns

user=> (bench (.length "hi there!"))
;              Execution time mean : 3.423909 ns
;    Execution time std-deviation : 0.202517 ns
```

Figuring out where to put type hints to avoid reflection can be a little tricky, and since avoiding reflection is necessary for high-performance Clojure code, it's nice to have a hook to warn us about it: (set! *warn-on-reflection* true). If we set that var to true, we'll get a helpful warning letting us know that the code we just wrote will need to use reflection:

performance/warn_on_reflection.clj

```
user=> (set! *warn-on-reflection* true)
;=> true
user=> (defn length-1 [x] (.length x))
;Reflection warning, NO_SOURCE_PATH:1:20 -
;   reference to field length can't be resolved.
;#'user/length-1
```

There are all kinds of other tools to see how your Clojure code is performing, from profilers like JVisualVM[2] (included in the Oracle JDK distribution) to disassemblers like no.disassemble.[3] These tools are there to help you pick the right problems to solve, so be sure to use them! You don't want to waste your valuable time optimizing code that's already fast enough.

Hiding Performance Optimizations

Now that we've proven a particular bit of code is really too slow, what can we do to make it faster? Shantanu Kumar's *Clojure High Performance Programming [Kum13]* is full of techniques to improve the performance of Clojure programs and we touch upon two tricks in this chapter. Often these techniques mean embracing the wonderful Java interop of Clojure, using primitives, arrays, and type hinting to match Java performance. However, a codebase packed with features like these can make Clojure *feel* like Java, which can be shocking for a functional programming devotee. Let's look at how we can speed things up without winding up with ugly, unreadable code.

Imagine you've been working on a feature for one of your back-office applications that computes a few statistics on the latest sales numbers. You've tracked the biggest problem to a tight loop that adds a vector of numbers together, using a function called sum that's written in a pretty typical Clojure style. Here's how you might speed it up:

performance/sum.clj
```
(defn sum [xs]
  (reduce + xs))
(def collection (range 100))
(bench (sum collection))
;              Execution time mean : 2.078925 µs
;    Execution time std-deviation : 378.988150 ns

(defn array-sum [^ints xs]
  (loop [index 0
         acc 0]
    (if (< index (alength xs))
      (recur (unchecked-inc index) (+ acc (aget xs index)))
      acc)))
(def array (into-array Integer/TYPE (range 100)))
(bench (array-sum array))
;              Execution time mean : 161.939607 ns
;    Execution time std-deviation : 5.566530 ns
```

2. http://jvisualvm.java.net
3. https://github.com/gtrak/no.disassemble

In array-sum, we've got a Java array, a type hint, and a low-level loop/recur instead of the much more elegant reduce version, but in exchange for that noise, we get about a 15x speedup. What do you think—is the speed worth all this code complexity? If you answered "It depends on the context," well done!

We won't try to cover every low-level speedup technique; instead you'll see how you can get the speed you want *and* the clarity you want. In this case, you can use a macro from clojure.core, areduce, to clean things up a bit:

performance/sum_with_areduce.clj
```
(defn array-sum [^ints xs]
  (areduce xs index ret 0 (+ ret (aget xs index))))

(bench (array-sum array))
;              Execution time mean : 170.214852 ns
;     Execution time std-deviation : 18.504698 ns
```

The clojure.core/areduce macro is implemented in the core language similarly to the previous version of array-sum, so it's no surprise that its performance is just as good.

performance/areduce_implementation.clj
```
(defmacro areduce
  "Reduces an expression across an array a, using an index named idx,
  and return value named ret, initialized to init, setting ret to the
  evaluation of expr at each step, returning ret."
  {:added "1.0"}
  [a idx ret init expr]
  `(let [a# ~a]
     (loop [~idx 0 ~ret ~init]
       (if (< ~idx  (alength a#))
         (recur (unchecked-inc ~idx) ~expr)
         ~ret))))
```

There's a library from Prismatic called hiphip[4] that goes a step further and eliminates the need for type hints.

After adding the Leiningen coordinates ([prismatic/hiphip "0.1.0"]) to your project.clj's :dependencies vector, you can try this out in your REPL as well (make sure to restart it after adding the dependencies):

performance/sum_with_hiphip.clj
```
(require 'hiphip.int)
(bench (hiphip.int/asum array))
;              Execution time mean : 144.535507 ns
;     Execution time std-deviation : 1.587751 ns
```

4. https://github.com/prismatic/hiphip

In hiphip's case, the big win in using a macro is that it can share nearly all of the implementation code across multiple data types (doubles, floats, ints, and longs). It even gets a little fancier, too. Because asum is a macro, it's also able to provide variants that allow some elegant bindings similar to the ones in for and doseq (see the hiphip.array docstring for details):

performance/sum_with_hiphip_fanciness.clj

```
(defn array-sum-of-squares [^ints xs]
  (areduce xs index ret 0 (+ ret (let [x (aget xs index)] (* x x)))))

(bench (array-sum-of-squares array))
;              Execution time mean : 1.419661 µs
;     Execution time std-deviation : 256.799353 ns

(bench (hiphip.int/asum [n array] (* n n)))
;             Execution time mean : 1.591465 µs
;     Execution time std-deviation : 232.503393 ns
```

The hiphip.int/asum version has pretty much identical performance to array-sum-of-squares. Meanwhile, it's a heck of a lot nicer to read and understand, right?

We don't have the space to go into all the details of hiphip's asum implementation, but here's the asum definition that's shared across the various data types:

performance/hiphip_asum.clj

```
;; from hiphip/type_impl.clj
(defmacro asum
  ([array]
     `(asum [a# ~array] a#))
  ([bindings form]
     `(areduce ~bindings sum# ~(impl/value-cast +type+ 0) (+ sum# ~form))))
```

The areduce here is hiphip's version that provides fancy bindings, not the areduce from clojure.core. impl/value-cast is what gives asum its ability to act on different data types based on the value of +type+ at compile time.

hiphip/type_impl.clj, where this macro is defined, is loaded directly from namespaces like hiphip.int via (load "type_impl"), instead of relying on the usual Clojure :require mechanism. Furthermore, hiphip/type_impl.clj doesn't actually define a namespace, so when it's loaded in hiphip.int, for example, it defines the asum macro in that namespace. This explains why +type+ can vary for the different types instead of just being a single unchanging value. This loading mechanism is a bit of a hack to avoid a macro-defining macro, which often means multiple levels of syntax-quoting. I can't fault anyone for wanting to avoid macro-defining macros—they're hard to write and even harder to understand later.

In Clojure's built-in areduce and hiphip's asum, we've seen macros enable great performance, comparable to loop/recur, while keeping the code concise and easy to understand. Next we'll look at a different approach to performance optimization that can allow us to sidestep problems entirely rather than improving them incrementally.

Moving Execution to Compile Time

If you were playing golf, where the lowest number of strokes wins, would you rather get a hole-in-one or skip the hole entirely? Assuming that's a legal thing to do (it's not) and that you don't enjoy playing the game in the first place (your mileage may vary), zero is clearly a better score than one.

When it comes to maintenance, the best code is no code at all.[5] The fastest code is code that doesn't need to execute at runtime. In our case, the lowest number of instructions wins! Remember that we can consider macros as not even *existing* at runtime; this is a clear opportunity for us to speed things up.

The caveat is that in order to move expensive computations to compile time, we need access to all of the computation's inputs at compile time. So computation that requires user input, even indirectly, would be a poor candidate for this type of optimization. The same goes for *any* computation where input values may vary: they wouldn't yet have their input values when they're asked to compute their values during macroexpansion.

For instance, a function that talks to a web service can't really be macro-ized, as the inputs aren't available until runtime, when the function is called:

performance/non_macroizable.clj
```
(defn calculate-estimate [project-id]
  (let [{:keys [optimistic realistic pessimistic]}
          (fetch-estimates-from-web-service project-id)
        weighted-mean (/ (+ optimistic (* realistic 4) pessimistic) 6)
        std-dev (/ (- pessimistic optimistic) 6)]
    (double (+ weighted-mean (* 2 std-dev)))))
```

But if we could reduce this function's responsibilities to have it handle only the calculation and not the web service call (leaving that job to some new function), that would give us both a better design and an idea about how we might benefit from macro-izing this call:

performance/macroizable_1.clj
```
(defn calculate-estimate [{:keys [optimistic realistic pessimistic]}]
  (let [weighted-mean (/ (+ optimistic (* realistic 4) pessimistic) 6)
        std-dev (/ (- pessimistic optimistic) 6)]
    (double (+ weighted-mean (* 2 std-dev)))))
```

5. http://www.codinghorror.com/blog/2007/05/the-best-code-is-no-code-at-all.html

```
user=> (calculate-estimate {:optimistic 3 :realistic 5 :pessimistic 8})
;=> 6.833333333333333

user=> (bench (calculate-estimate {:optimistic 3 :realistic 5 :pessimistic 8}))
;              Execution time mean : 1.974506 µs
;     Execution time std-deviation : 22.817749 ns

(defmacro calculate-estimate [estimates]
  `(let [estimates# ~estimates
         optimistic# (:optimistic estimates#)
         realistic# (:realistic estimates#)
         pessimistic# (:pessimistic estimates#)
         weighted-mean# (/ (+ optimistic# (* realistic# 4) pessimistic#) 6)
         std-dev# (/ (- pessimistic# optimistic#) 6)]
     (double (+ weighted-mean# (* 2 std-dev#)))))

user=> (calculate-estimate {:optimistic 3 :realistic 5 :pessimistic 8})
;=> 6.833333333333333

user=> (bench (calculate-estimate {:optimistic 3 :realistic 5 :pessimistic 8}))
;              Execution time mean : 2.208451 µs
;     Execution time std-deviation : 700.062170 ns
```

So on my machine, these are pretty comparable. This makes sense, because the only savings here is in avoiding a function call, and the JIT may even be inlining that.

If we were able to constrain the problem a bit and always call the macro with a literal map (this means it's available at compile time, not a value we get from a web service, database, or user input), we could do quite a bit better. Can you see how?

performance/macroizable_2.clj

```
(defmacro calculate-estimate [{:keys [optimistic realistic pessimistic]}]
  (let [weighted-mean (/ (+ optimistic (* realistic 4) pessimistic) 6)
        std-dev (/ (- pessimistic optimistic) 6)]
    (double (+ weighted-mean (* 2 std-dev)))))

user=> (calculate-estimate {:optimistic 3 :realistic 5 :pessimistic 8})
;=> 6.833333333333333

user=> (bench (calculate-estimate {:optimistic 3 :realistic 5 :pessimistic 8}))
;              Execution time mean : 4.814157 ns
;     Execution time std-deviation : 0.761096 ns
```

We've just moved all the work we were doing to macroexpansion time instead of runtime. Notice that we aren't quoting anything—this macro expands into a number. And it pays off: this version is on the order of *nanoseconds*. Units matter here. This isn't a few times slower, it's orders of magnitude faster, and it's because the macro has to expand only once!

Our example is a little unrealistic, since estimates come from a web service and normally won't be known at compile time. But there are certainly cases where we have all the information we need at compile time—the key criterion is having the input directly in the source code, like when you extract a function for clarity or to avoid duplication.

For example, the Hiccup[6] HTML templating library does as much work as it can in the `hiccup.core/html` macro at compile time. It converts its input data structures to HTML strings where possible, and when it's unable to do so, it defers the conversion *of only those parts* until runtime. Since completely eliminating runtime code costs is such a big win, it's good to keep track of what data we know and can take advantage of at compile time in performance-critical scenarios.

Using the JVM in ClojureScript Macros

The idea of shifting evaluation to compile time from runtime is particularly interesting when we apply it to ClojureScript, where macros are written and expanded in (JVM) Clojure but runtime evaluation happens on the JavaScript virtual machine. By moving expensive operations to compile time on the JVM, we can significantly reduce the amount of work we have to do at runtime.

On a project at work last year, our goal was to read in a data structure from a file to use in building up some HTML. There were a couple of barriers to achieving this goal:

- Reading from the filesystem is kind of an expensive operation, and our files were never going to change while the program was running.
- Our ClojureScript runs in the browser (as opposed to Node.js or something like that), so it doesn't even have access to the filesystem to be able to read the files.

Our initial solution to this problem was to use the ClojureScript macro system, which worked since the JVM-land macros have access to the filesystem. Because this felt a bit too clever and magical, we ended up moving away from that solution toward one that put the data structures into normal ClojureScript functions instead of their own files. But it's interesting that we were *able* to move a very expensive operation (and in fact one that seemed impossible to do at runtime!) forward to compile time to solve our problem. So, long story

6. https://github.com/weavejester/hiccup

short, if you're working in ClojureScript and you need to do things on the JVM that your JavaScript VM can't do, macros are a way forward.

The Need for Speed

In this chapter you've seen a few ways that macros can help you write fast systems while keeping your code concise, including hiding unsightly performance optimization hacks and shifting execution to compile time. You've seen how tools like Criterium can tell you what's slow and whether your subsequent changes have improved performance. Speeding up software can be really fun, but if it doesn't *need* to be fast, you don't need to waste your time working on it.

Now that you've seen how to speed up your code's runtime performance, we'll look at how you can speed up a more important bottleneck: your understanding of the code.

CHAPTER 6

Build APIs That Say Just What They Mean

When we read code that's more verbose than it needs to be, it slows us down. Ideally, we want code with only what's necessary to communicate its purpose. In this chapter we'll look at ways to provide beautiful APIs that allow developers to avoid including unnecessary boilerplate in their code. We're not talking about APIs here in the HTTP web service sense, just the interface that a library or piece of code exposes to let you use it.

This goal may seem a bit hazy, since "beauty" is subjective, and easy-to-use pure-function APIs are already prevalent. The good news is that providing a succinct macro API doesn't prevent you from offering an API with functions as well. I highly recommend this practice to help out your users who want to avoid the problems we talked about in Chapter 3, *Use Your Powers Wisely,* on page 27. The usual workflow is to write the functional API first, and then add a thin macro layer over the top where it's desirable. Of course, the order in which you do these things matters much less than the end result: offering both a flexible function API and a potentially more pleasant macro API.

In this chapter, we'll look at a couple of Clojure APIs that you've probably interacted with and see how their macros affect the interface that they present.

Compojure

Like most of James Reeves' libraries, Compojure[1] inspires us to strive for elegance in our own work. If you've done any web work with Clojure, at some point you've probably used Compojure, on top of Ring,[2] to handle your HTTP routing. A typical app using Compojure starts out something like this:

1. https://github.com/weavejester/compojure
2. https://github.com/ring-clojure/ring

apis/compojure_1.clj
```
(ns hello-world
  (:require [compojure.core :refer [defroutes GET]]
            [compojure.route :refer [not-found]]))

(defroutes app
  (GET "/" [] "Hello World")
  (not-found "Page not found"))
```

Before we look at the implementations of the Compojure bits that we're using, take a moment to consider how concise this code is. Beyond the opening ns form, we need only a few lines of code (plus a bit of wiring in the project.clj file) to launch a local web app with lein-ring.[3] We have a defroutes expression that defines a var that we can hand to the server infrastructure, a GET expression that defines a function to be executed upon a GET request to the root URL, and a not-found expression that defines the responder function for any request that didn't match the other routes.

Now let's see how Compojure makes things so nice:

apis/compojure_2.clj
```
(defmacro defroutes
  "Define a Ring handler function from a sequence of routes. The name may
  optionally be followed by a doc-string and metadata map."
  [name & routes]
  (let [[name routes] (name-with-attributes name routes)]
   `(def ~name (routes ~@routes))))
```

That's a pretty small macro, right? There's not much code at all, and a lot of it is using name-with-attributes from tools.macro,[4] a handy tool that lets your custom def-like macros act more like what people are used to, allowing doc-strings and metadata maps. And the only call that remains, routes (remember that the *invocation* of routes on the last line is resolved as compojure.core/routes due to the syntax-quote), is just a function. So defroutes is really just a nice layer of syntax to allow you to type this:

apis/compojure_3.clj
```
(defroutes app
  (GET "/" [] "Hello World")
  (not-found "Page not found"))
```

instead of the following:

3. https://github.com/weavejester/lein-ring
4. https://github.com/clojure/tools.macro

apis/compojure_4.clj
```
(def app
  (routes
    (GET "/" [] "Hello World")
    (not-found "Page not found")))
```

There's only a small difference between the two, that you'd give def a docstring in a different way than you would with defroutes. But the main benefit of defroutes here is that it eliminates an extra expression.

For many folks, this doesn't seem like a big win, and the great news for them is that because James has built this library the way he did, they don't *have* to use defroutes. So if you're among that crowd, you can use the def version instead, and of course you get the same code that defroutes macroexpands to. But for those who prefer the defroutes route, the more concise option is also available.

Of course, choosing to use the underlying function version of an API isn't always so easy. As it turns out, not-found is already just a function, but let's see what GET's implementation looks like:

apis/compojure_5.clj
```
(defn- compile-route
  "Compile a route in the form (method path & body) into a function."
  [method route bindings body]
  `(make-route
    ~method ~(prepare-route route)
    (fn [request#]
      (let-request [~bindings request#] ~@body))))

(defmacro GET "Generate a GET route."
  [path args & body]
  (compile-route :get path args body))
```

Here the case for a macro is more compelling, because without GET you'd need something like:

apis/compojure_6.clj
```
(ns hello-world.handler
  (:require [clout.core :refer [route-compile]]
            [compojure.core :refer [routes make-route]]
            [compojure.route :refer [not-found]]))

(def app
  (routes
    (make-route :get (route-compile "/")
                (fn [request] "Hello World"))
    (not-found "Page not found")))
```

Even this is selling GET (and its helper functions and macros) a bit short; prepare-route and let-request let you extract pieces of the request and bind them to symbols in the body of the request-servicing function. GET gives us a destructuring feature that we're not using at the moment. Nevertheless, two things are clear. First, the version of this web app using defroutes and GET is much more pleasant to look at and it's easier to quickly understand what's happening. And second, if we're so motivated, we can strip away the macro layer and use the underlying functions.

Clojure Koans

One helpful set of exercises for Clojure newcomers, the Clojure Koans,[5] has its code structure based on a macro called meditations that presents a boiler-plate-free interface. Perhaps you've already tried these out as I suggested in the Introduction on page ix and wondered how it worked under the hood. The koans ask you to fill in the blanks in order to make each test pass:

apis/koans_1.clj

```
(meditations
  "We shall contemplate truth by testing reality, via equality"
  (= __ true)

  "To understand reality, we must compare our expectations against reality"
  (= __ (+ 1 1)))
```

But in the interface that the meditations macro presents, there *are* no tests! We just see some (more or less helpful) text vaguely describing an expression that we want to make truthy. Under the hood, however, meditations creates assertions (using a helper function from another namespace called fancy-assert whose details we don't need to see) that allow users to see whether their input is correct:

apis/koans_2.clj

```
(ns koan-engine.core
  (:require [koan-engine.util :as u]))

(def __ :fill-in-the-blank)
(def ___ (fn [& args] __))

(defmacro meditations [& forms]
  (let [pairs (partition 2 forms)
        tests (map (fn [[doc# code#]]
                     `(u/fancy-assert ~code# ~doc#))
                   pairs)]
    `(do ~@tests)))
```

5. https://github.com/functional-koans/clojure-koans

By having a macro like this that maps over pairs of inputs and produces assertion expressions, we avoid littering the code with these fancy-assert bits. For this use case, eliminating the extra code allows the user to focus only on what the koans are trying to teach. The boilerplate we'd see without meditations isn't terribly noisy code, but it certainly isn't as tidy as the original version:

apis/koans_3.clj
```
(do
  (fancy-assert
    (= __ true)
    "We shall contemplate truth by testing reality, via equality")

  (fancy-assert
    (= __ (+ 1 1))
    "To understand reality, we must compare our expectations against reality"))
```

It's important to keep in mind here that macros are not the only way to remove duplicate code: delegating to single-purpose plain old functions also does that job nicely. However, in cases like these where we need full control of the execution of the expressions in order to provide error handling or other context, we need macros to provide that context (as we talked about in Chapter 4, *Evaluate Code in Context*, on page 35).

Decoupling Macros from Functions

When using macros for API purposes, remember that macros are icing, not the whole cake. If we choose to provide a macro API layer, providing access to the underlying machinery via functions is a great decoupling strategy. This makes our code more convenient for consumers who are likely to have varying requirements and aesthetic considerations, but also for us as we read, test, and maintain our code. All other things being equal, it's far easier to wrap our minds around functions and macros that do one thing than to understand ones that do many things.

Now that you've seen how you can make concise APIs to give users a nice interface on top of your domain-logic functions, we'll drill down to the lower levels of the language to see how you can build control flow mechanisms on top of lower-level constructs.

CHAPTER 7

Bend Control Flow to Your Will

Because functions evaluate their arguments before their bodies have a chance to execute, macros are the place to go when you want to create new control flow constructs. In this chapter, you'll see how you can build basic control flow mechanisms like while and do-while with macros. Then we'll dig into more advanced control flow features like delimited continuations and marvel at the fact that we didn't need to lobby any language committee to get these clarifying structures into our programs.

Loops and Loops and Loops and…

In many languages, within your first 24 hours with the language, you learn the few looping constructs that you will ever use. Often, those constructs include while, do-while, and for. Clojure does come with a number of built-in control flow tools like while and for (though Clojure's for is a list comprehension, much fancier than the for loop that imperative languages often use to iterate over indexed elements). More importantly, in Clojure's case, these constructs are almost always macros. We could have written them *ourselves* and used them in our programs. I don't know about you, but this gives me a tremendous feeling of power.

Let's see how easy it can be to build these ourselves, starting with while:

control_flow/while.clj
```
(defmacro while
  "Repeatedly executes body while test expression is true. Presumes
  some side-effect will cause test to become false/nil. Returns nil"
  {:added "1.0"}
  [test & body]
  `(loop []
     (when ~test
       ~@body
       (recur))))
```

This short macro on top of the loop special form gives us the familiar while loop that ships with Clojure! Here we take the potential danger of multiple evaluation that we saw in *Macros Can Be Tricky to Get Right*, on page 31 and use it to our advantage. Each time through the loop, we re-evaluate both the test and the body of expressions passed to while, until the test returns false or nil. As the docstring implies, we'd better have some mutable state changing at some point when using while—otherwise we'll have an infinite loop:

control_flow/while_example.clj
```
(def counter (atom 0))

(while (< @counter 3)
  (println @counter)
  (swap! counter inc))
; 0
; 1
; 2
;=> nil
```

If we were to write the analogous counter code using the raw loop/recur machinery, it wouldn't quite have the same clarity as the while version:

control_flow/while_as_loop_recur.clj
```
(def counter (atom 0))

(loop []
  (when (< @counter 3)
    (println @counter)
    (swap! counter inc)
    (recur)))
; 0
; 1
; 2
;=> nil
```

Many control flow macros build on top of loop/recur—while isn't just an isolated example. You can think of it as a low-level operation, almost like an assembly language for Clojure control flow. Clojure doesn't have a built-in do-while, but we could easily add one. This might be useful if we need something like while but that's guaranteed to execute the body of the expression at least once:

control_flow/do_while.clj
```
(defmacro do-while [test & body]
  `(loop []
     ~@body
     (when ~test (recur))))

(defn play-game [secret]
  (let [guess (atom nil)]
    (do-while (not= (str secret) (str @guess))
      (print "Guess the secret I'm thinking: ")
      (flush)
      (reset! guess (read-line)))
    (println "You got it!")))

(play-game "zebra")
; Guess the secret I'm thinking: lion
; Guess the secret I'm thinking: zebra
; You got it!
;=> nil
```

If we were only playing this game once, a normal while loop would be fine, since the guess starts as nil. But if we played with a crafty secret of "" (and with while instead of do-while), the player would win without having to even enter an answer. So the do-while loop solves that problem. I won't go further and try to convince you that you should look for opportunities to *use* a do-while loop. The important thing here is that you can create this control flow feature yourself with just a few short lines of code.

Turning Expressions Inside Out with Threading Macros

Some of Clojure's most fun built-in macros are the ones that allow you to rewrite deeply nested expressions in a way that more closely reflects their execution order. In this section, we'll take a look at how we can use these *threading macros* to simplify our code and how those macros work their magic. And to avoid confusion: *threading* in this context has nothing to do with JVM/OS threads; the two concepts just have an unfortunate naming collision.

The most common of these macros is ->, which threads the result of each expression into the next one as the first argument (converting symbols to lists where the next expression is not a list). Perhaps you've seen this macro before, when composing Ring[1] middleware:

control_flow/threading_ring_middleware.clj

```
(def app-1
  (wrap-head
    (wrap-file-info
      (wrap-resource (wrap-session
                       (wrap-flash
                         (wrap-params app-handler)))
                     "public"))))
```

1. https://github.com/ring-clojure/ring

```
;; The ,,, placeholders represent the result from the previous line
(def app-2
  (-> app-handler                   ;; app-handler
      wrap-params                   ;; (wrap-params ,,,)
      wrap-flash                    ;; (wrap-flash ,,,)
      wrap-session                  ;; (wrap-session ,,,)
      (wrap-resource "public")      ;; (wrap-resource ,,, "public")
      wrap-file-info                ;; (wrap-file-info ,,,)
      wrap-head))                   ;; (wrap-head ,,,)
```

With its much flatter look, the app-2 version is what most Clojure folks prefer for these kinds of expressions. It's not that we hate parentheses—it's just that we prefer the clarity of this pipelined approach. Now, I said that -> was nice because it clarifies execution order, but can you see why the Ring approach might be confusing for newcomers?

If you've used Ring before, you know that the wrap-head middleware has the first opportunity to respond to a web request, and app-handler is actually *last* in line to handle the request, though it's first in the threading macro. I've found that for newcomers (and for old-timers too!), this can be tricky to get your head around. It works this way for Ring middleware because we're threading *functions* through the expressions, and those functions don't actually get evaluated until later, when a web request comes in.

Now let's take a look at the implementation of ->, which turns expressions inside out:

control_flow/threading_implementation.clj

```
(defmacro ->
  "Threads the expr through the forms. Inserts x as the
  second item in the first form, making a list of it if it is not a
  list already. If there are more forms, inserts the first form as the
  second item in second form, etc."
  {:added "1.0"}
  [x & forms]
  (loop [x x, forms forms]
    (if forms
      (let [form (first forms)
            threaded (if (seq? form)
                       (with-meta `(~(first form) ~x ~@(next form)) (meta form))
                       (list form x))]
        (recur threaded (next forms)))
      x)))
```

Again, we see loop/recur used to iterate through the given forms, preserving metadata and transforming symbols to lists, and building up the result as x, the eventual return value from the macro (and thus, the expression to be evaluated). Take a few minutes to macroexpand a few sample expressions in your head, like these (don't hesitate to use macroexpand-1 if you get stuck!):

control_flow/threading_examples.clj

```
(-> "hi")
;=> ???

(-> 4
    (+ 3)
    (* 2))
;=> ???

(-> 10
    ^clojure.lang.LazySeq range
    .iterator
    (doto .next .next)
    .next)
;=> ???
```

Now that you've seen how -> does its magic, let's take a moment to think about what it would take to do this kind of rewriting without macros. As usual, we can get reasonably close with functions:

control_flow/threading_as_functions.clj

```
(-> 1
    (+ 2)
    (* 3)
    (+ 4)
    (* 5))
;=> 65

(defn thread-first-fn [x & fns]
  (reduce (fn [acc f] (f acc))
          x
          fns))

(thread-first-fn 1
                 #(+ % 2)
                 #(* % 3)
                 #(+ % 4)
                 #(* % 5))
;=> 65

;; or even:
(defn thread-first-fn' [x & fns]
  ((apply comp (reverse fns)) x))
```

Once again, this isn't bad at all. It looks even better for things like middleware composition, where most of the threaded elements are named functions already. It does take a bit more syntax for cases like the previous one, since we need to deal in functions, not just expressions. One thing we can do with -> that we *can't* do with either thread-first-fn variant is to insert a macro into the pipeline using only its name. If we tried that with thread-first-fn, we'd run into our old friend, CompilerException java.lang.RuntimeException: Can't take value of a macro.

There are plenty of other useful threading macros that you can dig into more: ->> has been in clojure.core for years, and clojure.core/some->, clojure.core/as->. Swiss Arrows[2] really goes to town and provides a whole tool belt of arrow macros similar to these. But this idea of rewriting the input expressions (as opposed to just wrapping the expression up to be evaluated or not, as we see fit) is a huge step. We're starting to get into the territory of macros that do things that functions can't.

Delimited Continuations

If you've programmed in an imperative language, chances are you've used a guard clause to return early from a method or function in certain cases, like the following Ruby code that approximates following someone on Twitter:

control_flow/guard_clause.rb
```ruby
def follow_user(user, user_to_follow)
  if user_to_follow.blocked?(user)
    puts "#{user_to_follow.name} has blocked #{user.name}"
    return
  end
  user.following << user_to_follow
  user_to_follow.followers << user
end
```

Early returns can be confusing if we use them all over the place, but they can be useful to describe requirements for the main method body to execute. Normally in Clojure, we'd need to write something like this:

control_flow/guard_clause_1.clj
```clojure
(defn follow-user [user user-to-follow]
  (if (contains? @(:blocked user-to-follow) (:name user))
    (println (:name user-to-follow) "has blocked" (:name user))
    (do
      (println "Adding follow relationship...")
      (swap! (:following user) conj (:name user-to-follow))
      (swap! (:followers user-to-follow) conj (:name user)))))
```

This kind of nesting isn't out of the ordinary in Clojure, but it's not really an early return. In the previous example, detecting whether a user is blocked or

2. https://github.com/rplevy/swiss-arrows

not doesn't really look like a precondition for the function—it looks much more like a peer to the rest of the function body.

It turns out that we *can* simulate early returns in Clojure by using *delimited continuations*. A good way to think about continuations is that they let you pause the execution of your code, capture its environment, and save a reference to that environment to use later. Again, they let you capture the *environment*: things like local variables and even the current state of the call stack!

If you haven't seen continuations before, this probably sounds insane. It still kind of does sound that way to me. Oleg Kiselyov, who's done some staggering research[3] in both quantity and quality, has some good arguments against *undelimited* continuations like Scheme's call/cc,[4] but the delimited variety are less problematic.

The delimc[5] library provides delimited continuations as macros, so if you add [delimc "0.1.0"] to your project.clj, you can try this out in your sample project:

control_flow/guard_clause_2.clj
```
(use 'delimc.core) ;; gives us `shift` and `reset`

(defn make-user [name]
  {:name name
   :blocked (atom #{})
   :following (atom #{})
   :followers (atom #{})})

(def colin (make-user "Colin"))
(def owen (make-user "Owen"))

(defn follow-user [user user-to-follow]
  (reset
    (shift k
           (if (contains? @(:blocked user-to-follow) (:name user))
             (println (:name user-to-follow) "has blocked" (:name user))
             (k :ok)))
    (println "Adding follow relationship...")
    (swap! (:following user) conj (:name user-to-follow))
    (swap! (:followers user-to-follow) conj (:name user))))

(swap! (:blocked owen) conj (:name colin))
(follow-user colin owen)
; Owen has blocked Colin
;=> nil
```

3. http://okmij.org/ftp/
4. http://okmij.org/ftp/continuations/against-callcc.html
5. https://github.com/swannodette/delimc

shift captures the current *continuation*, delimited by the containing reset. This continuation (which the shift binds to the name k) can be called just like a function, and when we do that, execution resumes from the point where the shift occurred in the code. So when we call (k :ok), we're saying: insert the value :ok where the shift occurred in the code. And if we *don't* invoke k, we don't insert *any* value in place of the shift, and in fact the reset is complete at that point!

If this seems a bit confusing, you're not alone: continuations are a tricky concept to wrap your mind around. The purpose of these examples is to show what's possible, not to master the shift/reset programming model.

We've needed to add another level of nesting in the delimc version, but now the guard condition stands on its own, independently of the rest of the function body. We can go a step further and extract the guard clause to clean things up:

control_flow/guard_clause_3.clj

```clojure
(defmacro require-user-not-blocked [user user-to-follow]
  `(shift k#
          (if (contains? @(:blocked ~user-to-follow) (:name ~user))
            (println (:name ~user-to-follow) "has blocked" (:name ~user))
            (k# :ok))))

(defn follow-user [user user-to-follow]
  (reset
    (require-user-not-blocked user user-to-follow)
    (println "Adding follow relationship...")
    (swap! (:following user) conj (:name user-to-follow))
    (swap! (:followers user-to-follow) conj (:name user))))

(swap! (:blocked owen) conj (:name colin))
(follow-user colin owen)
; Owen has blocked Colin
;=> nil
```

So within the context of the reset, every shift expression is able to decide whether or not to continue executing the body of the reset. What's more, the shift could even decide to continue the execution multiple times! We can use this trick to implement even more interesting control flow operations, like the *nondeterministic* choice operator. Here *nondeterminism* doesn't imply anything about randomness: it's a term for a programming paradigm where an expression may have more than one possible value. We'll see more of the implications of this idea in Chapter 8, *Implement New Language Features*, on page 77, when we look at logic programming. But for now, here's what nondeterministic programming looks like with delimited continuations:

control_flow/choice.clj

```clojure
(defmacro choice [xs]
  `(shift k# (mapcat k# ~xs)))

(def return list)

(defmacro insist [p]
  `(when-not ~p (choice [])))

(let [numbers (range 1 20)
      square (fn [x] (* x x))]
  (reset
    (return
      (let [a (choice numbers)
            b (choice numbers)
            c (choice numbers)]
        (insist (< a b c))
        (insist (= (square c) (+ (square a) (square b))))
        [a b c]))))
;=> ([3 4 5] [5 12 13] [6 8 10] [8 15 17] [9 12 15])
```

Notice that with choice, we're actually calling the continuation *multiple* times, instead of just choosing between 0 and 1 time as we did with the early return example. And we prune the search space with insist, which does basically the same job as our guard clause from earlier, but only stops one branch at a time of the search, rather than the whole execution.

So how does all this shift/reset magic work? Clearly no function could warp the flow of control so deeply. And as you'd expect, there's a macro under the hood allowing these things to happen. delimc uses a technique called *continuation-passing style* (CPS) to rewrite reset's input expressions into ones that implement these semantics we've been seeing.

We don't have nearly enough room to dig into all the details, but suffice to say that the reset macro is our entry point, kicking things off by setting up some context and calling a function that transforms its expression sequence into continuation-passing style:

control_flow/shift_reset_implementation.clj

```clojure
(ns delimc.core)

«snip»
(defmulti transform (fn [[op & forms] k-expr] (keyword op)))
«snip»

(defn shift* [cc]
  (throw (Exception.
          "Please ensure shift is called from within the reset macro.")))
```

```
(defmacro shift [k & body]
  `(shift* (fn [~k] ~@body)))

(defmethod transform :shift* [cons k-expr]
  (when-not (= (count cons) 2)
    (throw (Exception. "Please ensure shift has one argument.")))
  `(~(first (rest cons)) ~k-expr))

«snip»

(defmacro reset [& body]
  (binding [*ctx* (Context. nil)]
    (expr-sequence->cps body identity)))
```

There's a decent amount of complexity omitted here, but as you can see, shift is just a thin wrapper around shift*, which will always throw an exception. What gives? shift* is purely a marker for the expr-sequence->cps transformations to use, and transform is one of the key ones. We'll look at other examples of code walking like this in more detail in the last chapter, but first let's see some examples of shift/reset macroexpansion:

control_flow/shift_reset_expansion_1.clj

```
(macroexpand '(reset 1))
;=> (#<core$identity clojure.core$identity@750a6c68> 1)

(reset 1)
;=> 1

(macroexpand '(reset 1 2))
;=> ((clojure.core/fn [r__1247__auto__ & rest-args__1248__auto__]
;      (#<core$identity clojure.core$identity@750a6c68> 2))
;    1)

(reset 1 2)
;=> 2

(macroexpand '(reset (shift k (k 1))))
;=> ((clojure.core/fn [k] (k 1))
;    #<core$identity clojure.core$identity@750a6c68>)

(reset (shift k (k 1)))
;=> 1
```

In the first example, (reset 1), we call the identity function with 1 as an argument. Not bad at all. (reset 1 2) gets more complicated: we construct an anonymous function that calls the identity function with 2 as an argument (ignoring its argument, which is 1). In the third example, (reset (shift k (k 1))), we capture the continuation k inside an anonymous function and call it with 1 as an argument. In this small case, the continuation k is just the identity function.

You probably noticed that the clojure.core/identity function is injected directly into the macroexpansion rather than being quoted. This trick might be surprising at first, because it's not obvious that this is a legal thing to do. It turns out the Clojure compiler handles this case for us, but in general it's nicer to refer to names in macros than to emit functions directly into them. This is one of the least mind-bending parts of the code, though. These transformations become much more complicated as more complex expressions are passed into them, especially as soon as we give shift something to actually capture:

control_flow/shift_reset_expansion_2.clj

```
(macroexpand '(reset (+ 1 (shift k (k 1)))))
;=> ((clojure.core/fn [G__2268 & rest-args__1225__auto__]
;      ((clojure.core/fn [G__2269 & rest-args__1225__auto__]
;         ((clojure.core/fn [k] (k 1))
;          (clojure.core/fn [G__2270 & rest-args__1225__auto__]
;            (delimc.core/funcall-cc
;              G__2268
;              #<core$identity clojure.core$identity@750a6c68>
;              G__2269
;              G__2270))))
;       1))
;    (delimc.core/function +))

(reset (+ 1 (shift k (k 1))))
;=> 2
```

As you can see, things get complicated very quickly once the CPS transformations need to track the pending state of the computation. This is a bad way to compute (+ 1 1), but as you saw earlier, all this complexity pays off when you decide to call the continuation 0 times, or many times.

What's perhaps most interesting about this library is that all the code required for this syntax transformation is only a couple hundred lines long. The functions underneath the reset macro, the ones that expr-sequence->cps uses, have a big job to do, and they don't handle every scenario. reset is what's known as a *code-walking macro*, because it walks its input expressions, transforming them into a form that jives with delimc's worldview. The second thing to know about code-walking macros (after what they can do) is that they are hard to get right. In order to do its job as users might expect, delimc would have to be able not only to transform simple expressions like those you've seen, but also to handle local variable bindings, anonymous functions, conditionals, and who knows what else! It works very well with many inputs, but as of this writing, some input expressions can't yet be handled, such as loop.

If you're interested in digging more into delimited continuations, the delimc README[6] lists a number of papers that go into much more detail about this programming model and its implementation details.

Control Flow: A Special Case of Language Features

In this chapter, you've seen some amazing control flow mechanisms that macros give you the power to create. Now that you've seen one specific way of extending the language in ways you can't do in other languages, we'll look at the more general case. In the next chapter, you'll learn how to implement other language features as macros.

6. https://github.com/swannodette/delimc

CHAPTER 8

Implement New Language Features

In every programming language I've ever used, I occasionally yearn for a feature from some other language. When I'm in Java I want first-class functions like the ones in Scheme. In Ruby I want explicit dependencies like the ones in Java. No matter how good your language is, there's enough great work going on in other languages that chances are you'll eventually see a feature from another language that intrigues you. Thankfully, in Clojure, we can implement many of these missing features ourselves.

Implementing Pattern Matching

If you've ever programmed in a functional style outside of Clojure, perhaps using Haskell, Erlang, or Scala, you've almost certainly seen *pattern matching*. You can think of pattern matching as similar to Clojure's *destructuring* feature. It allows you to pull apart input expressions and bind values to names wherever you want. But pattern matching actually goes a bit further than destructuring as it lets you provide several potential patterns to match, along with clauses that will be executed if there's a match.

So if you're a pattern matching fan coming to Clojure, what do you do? If you immediately thought, "I could write a macro!," you're more optimistic about your macro skills than I was when I learned about pattern matching—great! And you're absolutely right, too! Whether or not you're a little hazy on what pattern matching looks like, let's decide what we might like to see in our pattern matcher. We'll start with something simple just to make sure that we can control the flow of execution, similar to what we did in the last chapter on page 65:

language_features/pattern_matching_1.clj
```clojure
;; desired API (using vectors so we can match multiple things later on):
(match [x]
  [0] :zero
  [1] :one
  :else :foo)

;; desired macroexpansion:
(cond (= [x] [0]) :zero
      (= [x] [1]) :one
      :else :foo)

;; expected outcomes
(describe "pattern matching"
  (it "matches an int"
    ;; will only compile once we've written `match`
    (let [match-simple-int-input (fn [n]
                                   (match [n]
                                     [0] :zero
                                     [1] :one
                                     [2] :two
                                     :else :other))]
      (should= :zero (match-simple-int-input 0))
      (should= :one (match-simple-int-input 1))
      (should= :two (match-simple-int-input 2))
      (should= :other (match-simple-int-input 3)))))
```

The expected outcomes here are encoded using a unit testing library called Speclj,[1] whose Leiningen coordinates are [speclj 3.0.2].

How would you write match as a macro? One way is to build on top of cond:

language_features/pattern_matching_1a.clj
```clojure
(defmacro match [input & more]
  (let [clauses (partition 2 more)]
    `(cond
       ~@(mapcat (fn [[match-expression result]]
                   (if (= :else match-expression)
                     [:else result]
                     [`(= ~input ~match-expression)
                      result]))
                 clauses))))
```

This version, small as it is, is enough to pass our first set of tests. Take some time to make sure you see how it expands into the cond expression, because things are going to get trickier. We've said before that where possible, macros should stay small and delegate most of their work to helper functions, so let's take our own advice and extract this function we passed to mapcat:

1. https://github.com/slagyr/speclj

language_features/pattern_matching_1b.clj

```
(defn match-clause [input [match-expression result]]
  (if (= :else match-expression)
    [:else result]
    [`(= ~input ~match-expression)
     result]))

(defmacro match [input & more]
  (let [clauses (partition 2 more)]
    `(cond ~@(mapcat (partial match-clause input)
                     clauses))))
```

Ah, much clearer. What's next? Well, one of the main features of pattern matching is that it allows you to bind values to names that we can use in the result expressions, so let's make sure we can do that. Here's what we expect to be able to do:

language_features/pattern_matching_2.clj

```
(it "matches and binds"
  (let [match-and-bind (fn [[a b]]
                         (match [a b]
                           [0 y] {:axis "Y" :y y}
                           [x 0] {:axis "X" :x x}
                           [x y] {:x x :y y}))]
    (should= {:axis "Y" :y 5} (match-and-bind [0 5]))
    (should= {:axis "Y" :y 3} (match-and-bind [0 3]))
    (should= {:axis "X" :x 1} (match-and-bind [1 0]))
    (should= {:axis "X" :x 2} (match-and-bind [2 0]))
    (should= {:x 1 :y 2} (match-and-bind [1 2]))
    (should= {:x 2 :y 1} (match-and-bind [2 1]))))
```

Notice that in some cases, like the `[0 y]` clause, we want to check that the first element, `a`, is equal to `0`, and if so, bind `y` to whatever `b` is. This isn't such a straightforward problem, because we have to use some combination of checking and binding: there's no straightforward function or macro in Clojure for us to use as a target for our macroexpansion.

Nevertheless, it's not an insurmountable task: I spent an hour or two working on a solution to pass this test case, and here's what came out of it:

language_features/pattern_matching_2a.clj

```
;; runtime helper
(defn match-item? [matchable-item input]
  (if (symbol? matchable-item)
    true
    (= input matchable-item)))

;; macroexpansion helpers
(defn create-test-expression [input match-expression]
  `(and (= (count ~input) ~(count match-expression))
        (every? identity
                (map match-item? '~match-expression ~input))))

(defn create-bindings-map [input match-expression]
  (let [pairs (map vector match-expression input)]
    (into {} (filter (comp symbol? first) pairs))))

(defn create-result-with-bindings [input match-expression result]
  (let [bindings-map (create-bindings-map input match-expression)]
    `(let [~@(mapcat identity bindings-map)]
       ~result)))

(defn match-clause [input [match-expression result]]
  (if (= :else match-expression)
    [:else result]
    [(create-test-expression input match-expression)
     (create-result-with-bindings input match-expression result)]))

(defmacro match [input & more]
  (let [clauses (partition 2 more)]
    `(cond ~@(mapcat (partial match-clause input)
                     clauses))))
```

Now, we'll go into some of the details, but it's important to note here that this solution didn't spring from my fingers fully formed. When I write macros, I tend to hit compile errors from time to time, at which point I take a deep breath, realize the compile errors are trying their best to help me, and move forward. Usually it's helpful to check out what code is being generated, either directly from `macroexpand-1` or by calling the helper functions with the intermediate values I expect to be passing around. We're doing some pretty complex metaprogramming here—the fact that there's not much code doesn't mean it doesn't take any time to get it all loaded up into your head.

The `match` macro stayed the same for this iteration, and the `match-clause` helper function kept its basic structure, but the matching logic needed to get much smarter. Since now the test expression needs to accommodate either a symbol binding or some literal value, `create-test-expression` makes sure there are the same number of elements in the input and match expressions. Did you notice that we're being a little sloppy with how we dump the input expression into the macroexpansion? Anytime you unquote the same value multiple times, be sure there's no potential for multiple evaluation like we saw in *Macros Can Be Tricky to Get Right,* on page 31—or at least be sure you're aware of that potential and willing to accept it. For the purpose of this example, I am going to accept it.

There's a bit of an annoying issue to deal with, though. We're macroexpanding into let expressions, but we're not taking advantage of destructuring to allow pattern matching in nested structures. Let's capture that in a test:

language_features/pattern_matching_3.clj
```
(it "handles vector destructuring"
  (let [match-and-bind (fn [[a b]]
                         (match [a b]
                                [0 [y & more]] {:axis "Y" :y y :more more}
                                [[x & more] 0] {:axis "X" :x x :more more}
                                [x y] {:x x :y y}))]
    (should= {:axis "Y" :y 5 :more [6 7]} (match-and-bind [0 [5 6 7]]))
    (should= {:axis "X" :x 1 :more [2 3]} (match-and-bind [[1 2 3] 0]))
    (should= {:x 1 :y 2} (match-and-bind [1 2]))))
```

This issue just needs a few lines tweaked:

language_features/pattern_matching_3a.clj
```
(defn match-item? [matchable-item input]
  (cond (symbol? matchable-item) true
        (vector? matchable-item)
          (and (sequential? input)
               (every? identity (map match-item? matchable-item input)))
        :else (= input matchable-item)))

(defn create-bindings-map [input match-expression]
  (let [pairs (map vector match-expression (concat input (repeat nil)))]
    (into {} (filter (fn [[k v]]
                       (not (or (keyword? k)
                                (number? k)
                                (nil? k))))
                     pairs))))
```

There's some duplication that could be cleaned up, but this gives us a much better pattern matcher. We can use it to implement a concise and fast-enough merge function, to be used as part of merge sort, where we have two sorted halves of a collection from recursive calls, and need to merge them together:

language_features/pattern_matching_4.clj
```
(defn merge [xs ys]
  (loop [acc [] xs xs ys ys]
    (match [(seq xs) (seq ys)]
      [nil b] (concat acc b)
      [a nil] (concat acc a)
      [[x & x-rest] [y & y-rest]]
        (if (< x y)
          (recur (conj acc x) x-rest ys)
          (recur (conj acc y) xs y-rest)))))
```

```
(it "implements merge (from merge-sort)"
  (should= [1 2 3] (merge [1 2 3] []))
  (should= [1 2 3] (merge [1 2 3] nil))
  (should= [1 2 3 4] (merge [1 2 3] [4]))
  (should= [1 2 3] (merge [] [1 2 3]))
  (should= [1 2 3] (merge nil [1 2 3]))
  (should= [1 2 3 4 5 6 7] (merge [1 3 4 7] [2 5 6])))
```

We'll give this pattern matcher the tongue-in-cheek name snore.match—as a community, can do a lot better than this. Consider core.match,[2] a pattern matching library written by David Nolen, which is also built using macros, but with much more thought and care put into it. It's built for speed, based on an algorithm by Luc Maranget to eliminate branches of the search tree more quickly (but keeping the sequential cond-like ordering of the match clauses). In addition to providing many more types of pattern matches than the simple ones we've done here, it even allows extending the pattern matching language itself! Take a look at the core.match wiki[3] for details.

Keep in mind that pattern matching is considered to be a crucial differentiator for a number of programming languages, and a mark of pride. And despite the fact that Clojure itself doesn't ship with a pattern matcher, we're able to create that feature ourselves, as a *library*, so that others can use it without having to change the core language.

Error Handling in Macros

Most macros, like our pattern matching example in the previous section, have very specific sets of input that they accept, and will throw strange-looking errors when those input expectations aren't satisfied. For instance, if we didn't realize that snore.match only handled vectors as input, we might try to match against a keyword, which would give us this error:

language_features/error_handling_1.clj

```
(match :foo
  :oh "hi"
  :foo "bar"
  :else "else")

UnsupportedOperationException count not supported on this type: Keyword
                               clojure.lang.RT.countFrom  (RT.java:556)
java.lang.UnsupportedOperationException: count not supported on this type: Keyword
                 RT.java:556 clojure.lang.RT.countFrom
                 RT.java:530 clojure.lang.RT.count
                match.clj:13 snore.match/create-test-expression
```

2. https://github.com/clojure/core.match
3. https://github.com/clojure/core.match/wiki

```
          match.clj:33 snore.match/match-clause
          AFn.java:156 clojure.lang.AFn.applyToHelper
          AFn.java:144 clojure.lang.AFn.applyTo
          core.clj:626 clojure.core/apply
         core.clj:2468 clojure.core/partial [fn]
       RestFn.java:408 clojure.lang.RestFn.invoke
         core.clj:2559 clojure.core/map [fn]
       LazySeq.java:40 clojure.lang.LazySeq.sval
                 :                :
                 :                :
    Compiler.java:6666 clojure.lang.Compiler.eval
         core.clj:2927 clojure.core/eval
          main.clj:239 clojure.main/repl [fn]
          main.clj:239 clojure.main/repl [fn]
          main.clj:257 clojure.main/repl [fn]
          main.clj:257 clojure.main/repl
```

This stack trace gives us all the information we need to trace the code path from the call all the way to the place the exception was thrown, but it doesn't tell the user of the macro that they're just using the macro with unexpected values. And if you're not used to reading compiler stack traces or remembering to try macroexpanding the expression, that long stack trace might be a bit overwhelming. There are a few ways to deal with this issue, some more common than others, and each with pros and cons:

Do nothing

Pro:
- Takes no extra work for the macro author

Con:
- Users need to read source code or ask others to help debug

Written documentation

Pro:
- Full power of the English language

Con:
- Goes out of date easily
- Little context/slow feedback loop when something goes wrong

Make the macro smarter

Pro:
- Friendlier to newcomers

Con:
- Hard to know what kinds of mistakes users can make
- Hard to figure out what users *mean* when they make a mistake
- Can add significant code complexity
- Not possible in full generality

Manually construct exception messages

Pro:	• Easy to explain the problem (when we know what it is)
Con:	• Requires manual decisions: easy to miss cases • Not always easy to figure out at what level a user went wrong

Have exception messages constructed automatically

Pro:	• No manual work; covers all cases that a grammar can cover • Very good error messages (though potentially not as good as bespoke ones where the problem is known)
Con:	• May require significant change in macro-writing practice • Quite rare in practice

The first two options, doing nothing and writing documentation, are quite easy to understand. Writing good documentation is difficult, but it's something we're used to seeing, and hopefully doing, as programmers. The other options, on the other hand, could use some discussion.

Making the macro smarter, to accept more kinds of input, seems like a great goal at first blush. It's basically Postel's Law[4] applied to macros. This can be a great hack for simple and obvious errors. But the problems space of "errors a user might make" is vast, approaching infinite, and with many errors, it's hard to even reason through why they were made. Unfortunately, being able to detect and fix any user error is not a realistic goal. So even if we're able to give the user some help and we don't mind a bit of added code complexity, it's not a complete solution.

If we wanted to improve the error messages for our mistaken pattern match call on page 82, we could start by adding an assertion that the match input is a vector, either with explicit assert calls or with Clojure preconditions:

language_features/error_handling_2.clj

```
;; Assertions, using custom error messages
(defmacro match [input & more]
  (assert (vector? input) "Match input must be a vector")
  (let [clauses (partition 2 more)]
    `(cond ~@(mapcat (partial match-clause input)
                     clauses))))
```

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

```
;; Clojure preconditions
(defmacro match [input & more]
  {:pre [(vector? input)]}
  (let [clauses (partition 2 more)]
    `(cond ~@(mapcat (partial match-clause input)
                     clauses))))
```

But how do we know what assertions to add? We might have happened upon that annoying error message because a coworker was blocked for half an hour trying to figure it out, but what happens next time? Have we provided good error messages for all the things someone could do wrong? What if someone assumes that our match is like case instead of cond, in that its last clause doesn't need an :else prefix? As things stand, they might be in for a surprise:

language_features/error_handling_3.clj

```
(match [1 2 3]
  [0 y z] :yx-plane
  [x 0 z] :xz-plane
  [x y 0] :xy-plane
  :other)

;=> nil
```

So to provide a good error message for this mistake, we need an additional precondition that checks that there are an even number of expressions after the match input:

language_features/error_handling_4.clj

```
(defmacro match [input & more]
  {:pre [(vector? input)
         (even? (count more))]}
  (let [clauses (partition 2 more)]
    `(cond ~@(mapcat (partial match-clause input)
                     clauses))))
```

Are we finished now? Perhaps so. match (or at least our version of it) is a pretty simple macro—we haven't provided many places for behavior to diverge. But the question of completion is fuzzier for macros with more complex input constraints, like this one that acts like defn but provides a default docstring where none is given:

language_features/default_docstring.clj

```
;; lein try [org.clojure/tools.macro "0.1.5"]
(require '[clojure.tools.macro :as m])

(defn- default-docstring [name]
  (str "The author carelessly forgot to provide a docstring for `"
       name
       "`. Sorry!"))
```

```
(defmacro my-defn [name & body]
  (let [[name args] (m/name-with-attributes name body)
        name (if (:doc (meta name))
               name
               (vary-meta name assoc :doc (default-docstring name)))]
    `(defn ~name ~@args)))

(my-defn foo [])
;=> #'user/foo
(doc foo)
; -------------------------
; user/foo
; ([])
;   The author carelessly forgot to provide a docstring for `foo`. Sorry!
;=> nil
```

Enumerating all of the possible things that can go wrong for a user here would be time-consuming. There's a combinatorial number of things the user could do wrong, with the function name, docstring, function arguments, pre- and post-conditions, and metadata map. You may remember making some confusing mistakes around what the `defn` or `ns` macros allowed when you were first learning Clojure. In situations like these, the idea of *illuminated macros* is particularly compelling, and we'll look at that next.

Illuminated Macros

The idea of illuminated macros, as far as I can tell, was introduced in a talk of the same name at Clojure/Conj 2013[5] by Jonathan Claggett and Chris Houser, where they discussed the pain of learning to use complex and under-documented macros, and a potential solution, based in part on research in Scheme by Culpepper & Felleison.[6] They introduced a library called seqex[7] that aims to help *automatically* create both good documentation and good error messages for macros that are written using its tooling.

Do you remember when you were still learning the syntax for `defn`, and where the docstring goes in relation to the argument list and any preconditions? The built-in `defn` macro does a pretty good job of providing helpful error messages, but it's had many hours of thought put into it, and it's not perfect:

5. https://www.youtube.com/watch?v=o75g9ZRoLaw
6. http://www.ccs.neu.edu/racket/pubs/icfp10-cf.pdf
7. https://github.com/jclaggett/seqex

tools.macro

In addition to the name-with-attributes helper function, the Clojure contrib library tools.macro provides a number of powerful tools that you may be interested in as you continue to advance your macro-fu:

- Locally scoped macros with macrolet, with a similar syntax to letfn.

language_features/macrolet.clj
```
(macrolet [(do-twice [form] `(do ~form ~form))]
  (do-twice (println "hi")))
; hi
; hi
;=> nil
```

- Symbol macros, which allow replacing specific symbols with expressions:

language_features/symbol_macros.clj
```
(let [counter (atom 0)]
  (symbol-macrolet [bump! (swap! counter inc)]
    bump!
    bump!
    bump!
    @counter))
;=> 3
```

- Analogues macroexpand-1 and macroexpand that expand regular macros and symbol macros (but not locally scoped ones, since they aren't available at runtime)

- A more robust analogue of clojure.walk/macroexpand-all

It's well worth checking out, even if you don't intend to use these advanced features—the code is a fun read!

language_features/defn_errors.clj
```
(defn "Squares a number" square [x] (* x x))
; IllegalArgumentException First argument to defn must be a symbol
;                                   clojure.core/defn (core.clj:277)
; java.lang.IllegalArgumentException: First argument to defn must be a symbol
;                       core.clj:277 clojure.core/defn

;; Not bad, huh? Let's try to remember how destructuring works:

(defn square-pair [(x y)]
  (list (* x x) (* y y)))
; CompilerException java.lang.Exception: Unsupported binding form: (x y),
;                                                       compiling: [...]

;; Still pretty darn good, but doesn't tell us what binding forms *are* valid.
```

A seqex-based implementation of defn goes even a bit further and provides both additional context around errors and generated grammar documentation:

language_features/seqex_defn_usage.clj
```
;; lein try org.clojars.trptcolin/seqex "2.0.1.1"

(require '[n01se.syntax.repl :as syntax])

(syntax/defn "Squares a number" square [x] (* x x))
; Bad value: "Squares a number"
; Expected var-name
;=> nil

(syntax/syndoc syntax/defn)
;          defn => (defn var-name doc-string? attr-map? (sig-body | (sig-body)+))
;      sig-body => binding-vec prepost-map? form*
;   binding-vec => [binding-form* (& symbol)? (:as symbol)?]
;  binding-form => symbol | binding-vec | binding-map
;   binding-map => {((binding-form form) | (:as symbol) | keys | defaults)*}
;          keys => (:keys | :strs | :syms) [symbol*]
;      defaults => :or {(symbol form)*}
;=> nil

;; The above is even better in the terminal, with its ANSI color output.
```

The syndoc syntax documentation for this macro looks just like a BNF[8] grammar, and because it's automatically generated from the same code that provides the error messages, it's bound to stay in sync even if the allowable macro inputs change!

This is a pretty big win for anyone who's struggled with poor macro usability. It's not without cost, though: syntax/defn takes work to implement, even though it happens to be built on top of clojure.core/defn. seqex, as you may have guessed based on its name or problem domain, uses tools that the authors have dubbed "sequence expressions," which are for sequences what regular expressions are for strings. Ignoring the most complex parts of syntax/defn, the destructuring logic, there's still a lot of thought that's been put into structuring the sequence expressions:

language_features/seqex_defn_implementation.clj
```
(ns n01se.syntax.repl
  (:require [n01se.seqex :refer [cap recap]]
            [n01se.syntax :refer [defterminal defrule defsyntax
                                  cat opt alt rep* rep+
                                  vec-form, map-form, map-pair, list-form
                                  rule sym form string]])
  (:refer-clojure :exclude [let defn]))
(alias 'clj 'clojure.core)
```

8. http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form

```
(defterminal prepost-map map?)
(defterminal attr-map map?)
(defterminal doc-string string?)

(declare binding-form)

(defrule binding-vec
  (vec-form (cat (rep* (delay binding-form))
                 (opt (cat '& sym))
                 (opt (cat :as sym)))))

(defrule binding-map
  ;; [complex logic mostly because of destructuring]
)

(defrule binding-form
  (alt sym binding-vec binding-map))

(defrule sig-body
  (cat binding-vec (opt prepost-map) (rep* form)))

(defterminal var-name symbol?)

(defsyntax defn
  (cap (cat var-name
            (opt doc-string)
            (opt attr-map)
            (alt sig-body
                 (rep+ (list-form sig-body))))
       (fn [forms] `(clj/defn ~@forms))))
```

This style of macro-writing hasn't yet taken the Clojure community by storm, but that may change because of the documentation and error-generating benefits. Quite recently, another entrant has appeared on the sequence expression scene: Christophe Grand's elegant and powerful seqexp.[9] At version 0.2.1, it's more of a single-purpose tool, currently only handling the sequence expression part of the problem, and not the generation of error messages or documentation. I wouldn't be surprised to see those kinds of features built on top of seqexp in the future, though. Because most macros tend to be fairly focused and because the cost of switching over to an illuminated approach is high, we're likely to keep writing macros in the normal way for the foreseeable future. But as a user of complex macros, I'd sure like to have the benefits conferred by the illuminated macro approach.

9. https://github.com/cgrand/seqexp

Code-Walking Macros

Let's say you're a former Ruby developer, and you really miss the feature of Ruby that allows any method, class, or module definition to act as an implicit begin (Ruby's version of try). In other words, you'd like to be able to write this code in Clojure:

language_features/implicit_try_1.clj
```
(defn delete-file [path]
  (clojure.java.io/delete-file path)
  (catch java.io.IOException e false))
```

This code doesn't work in Clojure, of course—you'll get a compiler exception if you try to type that in at your REPL. You'd need to wrap a try expression around everything following the argument list to get it to work as you want. Take a moment and think about what it would take to have this code work. One option would be to file an enhancement request for the language to allow this, but major language changes like this are unlikely to be accepted without lots of thought, and certainly not anytime soon. But if we wrap our code in a macro call, we can definitely write the macro that allows this code to work.

As we have only a single example, we might first try a naïve approach:

language_features/implicit_try_1a.clj
```
(defmacro with-implicit-try [& defns]
  `(do
     ~@(map
         (fn [defn-expression]
           (let [initial-args (take 3 defn-expression)
                 body (drop 3 defn-expression)]
             `(~@initial-args (try ~@body))))
         defns)))

(with-implicit-try
  (defn delete-file [path]
    (clojure.java.io/delete-file path)
    (catch java.io.IOException e false)))
```

And in fact this version will work great for this kind of input defn. It's fragile, though: if we do something as simple as adding a docstring, suddenly our macro is broken because the number of initial-args needs to change. We could use clojure.tools.macro/name-with-attributes to solve the docstring issue, but as we saw in the last section, there are plenty of other variations in the input that defn allows. Besides, we'd really like to be able to have our with-implicit-try work for other things besides defn as well: do, fn, let, loop, when, and letfn all feel like places it'd make sense to have our implicit try in place.

In order to cover all these cases correctly, we need to look not only at each top-level expression, but also at all of the internal expressions. And what about macros like when that end up expanding to other expressions on our hit list? In order to handle those kinds of things, we'd need a macro that actually walks through the code given to it, macroexpanding at each step and generating the code we want to end up with.

There are a wide range of code walking tools out there. For the very simplest of tasks, we could use the built-in clojure.walk, but its macroexpansion facilities aren't aware of bindings, a shortcoming that Riddley[10] overcomes:

language_features/clojure_walk.clj

```
(require '[clojure.walk :as cw])
(cw/macroexpand-all '(let [when :now] (when {:now "Go!"})))
;=> (let* [when :now] (if {:now "Go!"} (do)))

;; lein try org.clojars.trptcolin/riddley "0.1.7.1"

(require '[riddley.walk :as rw])
(rw/macroexpand-all '(let [when :now] (when {:now "Go!"})))
;=> (let* [when :now] (when {:now "Go!"}))
```

Riddley also provides access to &env and takes care of expanding any :inline function definitions. Besides macroexpand-all, and getting back to our purposes, Riddley exposes a handy function, riddley.walk/walk-exprs, that allows us to replace input expressions with expressions of our choosing.

language_features/riddley_basics.clj

```
(require '[riddley.walk :as walk])

(defn malkovichify [expression]
  (walk/walk-exprs
    symbol?                 ;; predicate: should we run the handler on this?
    (constantly 'malkovich) ;; handler: does any desired replacements
    expression))

(malkovichify '(println a b))
;=> (malkovich malkovich malkovich)
```

This is exactly the sort of thing we need for our implicit try. Handling all the cases correctly in a code-walking macro can often be similar to thinking recursively, because of macroexpansion. What are the base cases? Well, the base cases of macroexpansion are precisely the special forms, so they're a good place to start:

10. https://github.com/ztellman/riddley

language_features/special_forms.clj
```
(source special-symbol?)
; (defn special-symbol?
;   "Returns true if s names a special form"
;   {:added "1.0"
;    :static true}
;   [s]
;     (contains? (. clojure.lang.Compiler specials) s))

user=> (sort (keys clojure.lang.Compiler/specials))
;=> (& . case* catch def deftype* do finally fn* if let* letfn* loop*
;     monitor-enter monitor-exit new quote recur reify* set! throw try var
;     clojure.core/import*)
```

Continuing along our recursive-programming train of mind, if we correctly handle the base cases (special forms like let*, fn*, etc.), and we correctly handle macroexpansions, that means that by induction we'll be able to correctly handle code that contains macros.

A first cut at with-implicit-try takes only around 60 lines of code, but it did take a lot of trial and error, with plenty of test cases:

language_features/trymplicit_1.clj
```
(ns trptcolin.trymplicit
  (:require [riddley.walk :as walk]))

(declare add-try)

(defn should-transform? [x]
  (and (seq? x)
       (#{'fn* 'do 'loop* 'let* 'letfn* 'reify*} (first x))))

(defn- wrap-fn-body [[bindings & body]]
  (list bindings (cons 'try body)))

(defn- wrap-bindings [bindings]
  (->> bindings
       (partition-all 2)
       (mapcat
         (fn [[k v]]
           (let [[k v] [k (add-try v)]]
             [k v])))
       vec))

(defn- wrap-fn-decl [clauses]
  (let [[name? args? fn-bodies]
          (if (symbol? (first clauses))
            (if (vector? (second clauses))
              [(first clauses) (second clauses) (drop 2 clauses)]
              [(first clauses) nil (doall (map wrap-fn-body (rest clauses)))])
           [nil nil (doall (map wrap-fn-body clauses))])]
    (cond->> fn-bodies
      (and name? args?) (#(list (cons 'try %)))
      (not (and name? args?)) (map add-try)
      args? (cons args?)
      name? (cons name?))))

(defn- wrap-let-like [expression]
  (let [[verb bindings & body] expression]
    `(~verb ~(wrap-bindings bindings) (try ~@(add-try body)))))

(defn transform [x]
  (condp = (first x)
    'do (let [[_ & body] x]
          (cons 'try (add-try body)))

    'loop* (wrap-let-like x)

    'let* (wrap-let-like x)

    'letfn* (wrap-let-like x)

    'fn* (let [[verb & fn-decl] x]
           `(fn* ~@(wrap-fn-decl fn-decl)))

    'reify* (let [[verb options & fn-decls] x]
              `(~verb ~options ~@(map wrap-fn-decl fn-decls)))
    x))

(defn add-try [expression]
  (walk/walk-exprs should-transform? transform expression))

(defmacro with-implicit-try [& body]
  (cons 'try (map add-try body)))
```

There's clearly some complexity here, particularly around pulling out the various styles of fn* expressions. It would have been nice if there was only one base-level form, but because of backward compatibility, we'll probably be stuck with this sort of thing for a while.

Much more significantly, however, there's one critical flaw in this approach to wrapping try inside each of these special forms: it's not possible to recur across a try. This means that any special form that can act as a recur target can't be changed in such a cavalier way. So we need a way to avoid our wrapping of fn*, loop*, and reify*, at least where they're being used for recur. Would it be better to simply always skip those expressions? Perhaps, but it's not nearly as interesting, and besides, wrapping a function definition was really our main use case anyway!

What Are :inline Functions, Anyway?

We said that one nice feature of Riddley is that it expands :inline functions, but what are those? If you've read through the source code for the clojure.core namespace, you've probably noticed math-related functions like > that appear to have multiple implementations of the same code:

language_features/less_than.clj
```
(defn >
  "Returns non-nil if nums are in monotonically decreasing order,
  otherwise false."
  {:inline (fn [x y] `(. clojure.lang.Numbers (gt ~x ~y)))
   :inline-arities #{2}
   :added "1.0"}
  ([x] true)
  ([x y] (. clojure.lang.Numbers (gt x y)))
  ([x y & more]
   (if (> x y)
     (if (next more)
       (recur y (first more) (next more))
       (> y (first more)))
     false)))
```

You may be able to guess by the name of the feature what :inline does: it tells the Clojure compiler that wherever it finds a call to >, the compiler can go ahead and inline the definition using the expansion function in the metadata. These calls need to be in the verb position, the first thing after the opening parenthesis, in order to be inlined. Seem familiar? Inline functions are really quite similar to macros! But there's a crucial difference: inline functions can also be treated as values, in which case the non-inline implementation gets used. This is why we can say (sort > [1 2 3 4 5]) and get (5 4 3 2 1) back.

So when should you use :inline functions? Basically only when you need to eke out a bit of performance. It would be confusing for a function to behave differently depending on whether it's passed around as a value or not, you don't want to confuse your users. You may notice that Clojure's with-redefs can't replace :inline versions of functions, since they've already been expanded by the time with-redefs gets ahold of them:

language_features/inline_with_redefs.clj
```
(with-redefs [< +] (< 1 2))
;=> true

(with-redefs [< +] (apply < [1 2]))
;=> 3
```

But if you want to eliminate the runtime overhead of function dispatch while keeping the flexibility of functions, :inline is not a bad way to go.

To make things concrete, how can we avoid inserting a try for fn* when it has a recur? Since we're code-walking anyway, one interesting strategy is to extend the existing code walker with a case for recur, and have fn* keep track of whether a recur was found while its contents were being walked:

language_features/trymplicit_finding_recur.clj
```
(def recur-found (atom false))

(defn should-transform? [x]
  (and (seq? x)
       ;; NOTE: we've added 'recur - easy to forget
       (#{'recur 'fn* 'do 'loop* 'let* 'letfn* 'reify*} (first x))))

(defn transform [x]
  (condp = (first x)
    'recur (let [[verb & args] x]
             (reset! recur-found true)
             x)
    ;; ...
    'fn* (let [[verb & fn-decl] x
               _ (reset! recur-found false)
               result `(fn* ~@(doall (wrap-fn-decl fn-decl)))]
           (if @recur-found
             x
             result))
    ;; ...
))
```

There is a major problem here: we're looking for *any* recur expression inside the function, not just the ones that could be affected. For instance, the recur in this useless expression, (fn* [] (loop* [] (recur))), affects only the loop* expression, not the outer fn*. It's as though we need to push a new context onto a stack for every potential recur target we walk and pop it off when we're done walking it. Good news—Clojure's dynamic bindings work perfectly for that, and are the key to solving this dilemma in a reasonable way.

There are lots of other details here: for instance, we've added doall in a number of places to realize any lazy seqs, since we're relying on side effects while code-walking. And there's also still room for improvement, both in terms of code quality and functionality. How would we handle each of the remaining special forms, for instance?

language_features/trymplicit/src/trptcolin/trymplicit.clj
```
(ns trptcolin.trymplicit
  (:require [riddley.walk :as walk]))

(def ^:dynamic *recur-search-tracker*
  (atom false))
```

```
(declare add-try)

(defn should-transform? [x]
  (and (seq? x)
       (#{'fn* 'do 'loop* 'let* 'letfn* 'reify* 'recur} (first x))))

(defn- wrap-fn-body [wrapper-fn [bindings & body]]
  (if (nil? wrapper-fn)
    (cons bindings body)
    (list bindings (wrapper-fn body))))

(defn- wrap-bindings [bindings]
  (->> bindings
       (partition-all 2)
       (mapcat
         (fn [[k v]]
           (let [[k v] [k (add-try v)]]
             [k v])))
       vec))

(defn- wrap-fn-decl [wrapper-fn clauses]
  (let [[name? args? fn-bodies]
          (cond (symbol? (first clauses))
                  (if (vector? (second clauses))
                    [(first clauses) (second clauses) (drop 2 clauses)]
                    [(first clauses) nil
                     (doall (map (partial wrap-fn-body wrapper-fn)
                                 (rest clauses)))])
                (vector? (first clauses))
                  [nil (first clauses) (rest clauses)]
                :else
                  [nil nil (doall (map (partial wrap-fn-body wrapper-fn)
                                       clauses))])]
    (cond->> fn-bodies
      (and name? args?) (#(if (nil? wrapper-fn)
                             (list `(do ~@(doall (map add-try %))))
                             (list (wrapper-fn (doall (map add-try %))))))
      (not (and name? args?)) (#(let [not-both-result (map add-try %)]
                                   not-both-result))
      args? (cons args?)
      name? (cons name?))))

(defn- wrap-let-like [expression]
  (let [[verb bindings & body] expression
        result `(~verb ~(wrap-bindings bindings) (try ~@(doall (add-try body))))]
    (if @*recur-search-tracker*
      `(~verb ~(wrap-bindings bindings) ~@(add-try body))
      result)))
```

```
(defn transform [x]
  (condp = (first x)

    'recur (let [[verb & args] x]
             (reset! *recur-search-tracker* true)
             x)

    'do (let [[_ & body] x
              result (cons 'try (add-try body))]
          (if @*recur-search-tracker*
            (cons 'do (add-try body))
            (cons 'try (add-try body))))

    'loop* (binding [*recur-search-tracker* (atom false)]
             (wrap-let-like x))

    'let* (wrap-let-like x)

    'letfn* (wrap-let-like x)

    'fn* (binding [*recur-search-tracker* (atom false)]
           (let [[verb & fn-decl] x
                 result `(fn* ~@(doall (wrap-fn-decl #(cons 'try %) fn-decl)))]
             (if @*recur-search-tracker*
               `(fn* ~@(doall (wrap-fn-decl nil fn-decl)))
               result)))

    'reify* (let [[verb options & fn-decls] x
                  wrap-reify-fn (fn [expression]
                                  (binding [*recur-search-tracker* (atom false)]
                                    (let [result (doall (wrap-fn-decl #(cons 'try %)
                                                                      expression))]
                                      (if @*recur-search-tracker*
                                        (wrap-fn-decl nil expression)
                                        result))))]
              `(~verb ~options ~@(doall (map wrap-reify-fn fn-decls))))

    x))

(defn add-try [expression]
  (walk/walk-exprs should-transform? transform expression))

(defmacro with-implicit-try [& body]
  (cons 'try (map #(binding [*recur-search-tracker* (atom false)] (add-try %))
                  body)))
```

I'm trying to make the case here that code-walking macros are not easy: they take a lot of effort to do well. There aren't very many examples of robust code-walking macros out there, but those that do exist are quite interesting:

- Proteus[11] creates local mutable variables… OK, I realize that reads like a joke in a Clojure book, but it's probably fine, right? Proteus, like our trymplicit, uses Riddley to do its code-walking.
- Clojure-TCO[12] rewrites Clojure expressions to provide tail-call optimization, a feature for which the JVM lacks full support but that we can get via macros. Clojure-TCO does its own custom code-walking and, like delimc, doesn't handle all of Clojure.
- core.async[13] generates a state machine from imperative-looking code that allows the library to schedule the code's execution asynchronously, whether on the JVM using threads or on the JavaScript VM, and using Go-style channels (Communicating Sequential Processes). It uses tools.analyzer.jvm[14] for the Clojure implementation's code-walking, and the built-in ClojureScript analyzer for the ClojureScript implementation.

Every nontrivial code-walking macro I've written or studied has some caveats where you need to know a bit about how the underlying machinery works. But this is also true of every interesting language feature I've studied. Abstractions, at some point, break down. These libraries may all seem to do magical-looking things, but as a user of a code-walking macro it's great for you to be aware of where the abstractions can leak. Typically these kinds of macros support a *subset* of Clojure, and it's more important than ever for library authors to document that subset, along with any changes in language semantics.

As Douglas Hoyte points out in *Let Over Lambda [Hoy08]*, writing a robust code walker is tough. If we can avoid it, we're usually better off not trying to do it ourselves, which is what makes libraries like riddley and tools.analyzer.jvm so useful. Even writing a code-walking macro like the one we just wrote, that simply *uses* a code walker, is itself a serious undertaking. It's quite easy to continually find yourself 90% of the way done, with 90% still left to go. But if you want to do the kinds of deep transformations that core.async, Clojure-TCO, Proteus, and our own Trymplicit library are able to do, it's worth rolling up your sleeves and doing the hard work.

11. https://github.com/ztellman/proteus
12. https://github.com/cjfrisz/clojure-tco
13. https://github.com/clojure/core.async
14. https://github.com/clojure/tools.analyzer.jvm

Macros Are [Not] Magic

In this book, we've covered a lot of ground in a short time. You've seen how you can construct expressions using normal Clojure programming techniques, how to get what you want from the various quoting mechanisms, and how to find out what's going wrong with misbehaving macros. You've learned about a few potential downsides to macros and seen some ways you can use plain old functions to do some of the jobs that macros can do. And we've explored a number of the most common use cases for macros, from performance to new language features.

Your journey to Clojure macro mastery doesn't end here, but you're well on your way. Now it's time to see where your new skills take you. Dig into the code for the macros in your favorite libraries. See how they're put together and why they do the things they do. Macros may seem full of magic to some, but as you know, they're just programming like everything else. And most importantly, have fun!

Bibliography

[ECG12] Chas Emerick, Brian Carper, and Christophe Grand. *Clojure Programming*. O’Reilly & Associates, Inc., Sebastopol, CA, 2012.

[FH11] Michael Fogus and Chris Houser. *The Joy of Clojure*. Manning Publications Co., Greenwich, CT, 2011.

[Hal09] Stuart Halloway. *Programming Clojure*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.

[Hoy08] Doug Hoyte. *Let Over Lambda: 50 Years of LISP*. Doug Hoyte/HCSW, http://www.hcsw.org/, 2008.

[Knu74] Donald E. Knuth. Structured Programming with go to Statements. *ACM Comput. Surv.* 6[4]:261–301, 1974.

[Kum13] Shantanu Kumar. *Clojure High Performance Programming*. Packt Publishing, Birmingham, UK, 2013.