Sunday, August 9, 2015

Best Clojure Books

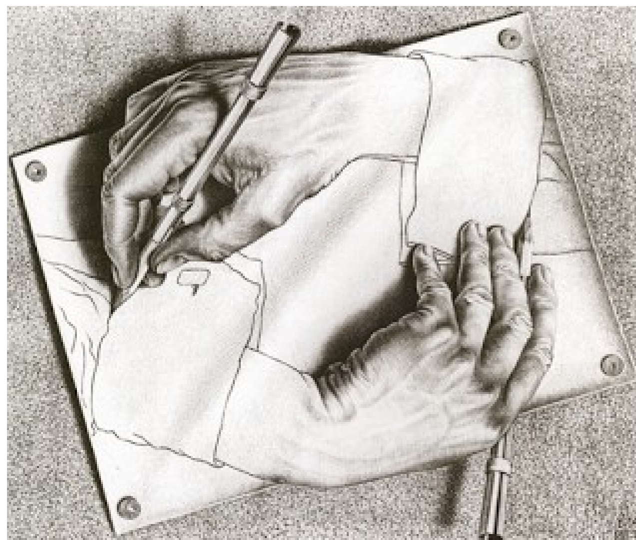
Are We There Yet?
~ Rich Hickey (creator of Clojure), title of his keynote at the JVM Languages Summit of 2009.

If you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases.
~ Guy Steele, as quoted in Michael Fogus' and Chris Houser's *The Joy of Clojure*

Every now and then a man's mind is stretched by a new idea or sensation, and never shrinks back to its former dimensions.
~ Oliver Wendell Holmes Sr., *Autocrat of the Breakfast Table*

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.
~ Eric Raymond, author of *The Art of Unix Programming*, *The Cathedral and the Bazaar*

Lisp has... assisted a number of our most gifted fellow humans in thinking previously impossible thoughts.
~ Edsger Dijkstra, *CACM, 15:10*



One of the ever-recursing marvels from the mind of Escher

If I were in a pinch to single out the bare minimum of *must-have* languages for the effective, daily practice of the software craft nowadays, I would point to Java and Scala as being absolutely indispensable. At the same time, it's undeniably helpful to have in one's toolbox—programming models, if you will—tools from *other* programming paradigms. I say so because I know full well the inevitability of the need to wield tools from whichever programming paradigm is up to snuff for slaying a particular, wicked problems. With that rather lofty assertion, I submit to you the Clojure programming language ;)

Scala enthusiasts should note that (1) this post is a direct analog of a recent post that has corresponding thoughts on the finest Scala books, and (2) this post on the best Clojure (think Lisp) books is also highly relevant to you. Allow me to elaborate very briefly: The programmer who has done justice to the notion expressed in the second of the two preceding points—in underscoring the eminent relevance of Lisp to Scala—is David Pollak. In his fine book entitled *Beginning Scala* (Apress), Pollak observes that

> After more than two years of writing and loving Scala, the only regrets that I have are that I didn't learn Lisp in college or take any programming language theory courses in grad school. Each morning that I sit down at the keyboard and start coding Scala, I get a feeling of calm, peace, and power. I know that I'm going to have another day of taking the ideas in my head and reducing them to a running computer program that will be fast and low in defects.

With that, let's foray into the land of Clojure...

And what a joy Rich Hickey has given to us programmers by creating Clojure—a functional programming language that runs on the Java Virtual Machine (JVM). It is a Lisp dialect and, as such, an inheritor of limitless expressive power. Though I can't claim to have advanced to the stage of a Lisp hacker, I've done *plenty* of dabbling in Clojure. In this post, I'll try to distill the essence of the finest resources in print that have helped me grok Clojure. I'll take an opinionated look at the following books, in turn

1. *The Joy of Clojure*, Second edition (Manning), by Michael Fogus and Chris Houser.
2. *Clojure Programming* (O'Reilly) by Chas Emerick, Brian Carper, and Christophe Grand.
3. *Functional Programming Patterns in Scala and Clojure* (The Pragmatic Bookshelf) by Michael Bevilacqua-Linn.
4. *Practical Common Lisp* (Apress) by Peter Seibel.
5. *Mastering Clojure Macros* (The Pragmatic Bookshelf), by Colin Jones.
6. *On Lisp: Advanced Techniques for Common Lisp* (Prentice Hall), by Paul Graham.
7. *Hackers & Painters: Big Ideas from the Computer Age* (O'Reilly), by Paul Graham.

The abstraction of *night and day*, as reified by Escher, abstractionist extraordinaire

On our way to my opinionated look at the preceding books, let's first meander through a preamble, which I hope will also be valuable for you—feel free to initially jump ahead to the reviews themselves, and return to this preamble afterwards!

But first, in the spirit of obviating redundancy—to avoid referring to the individual Lisp dialects out there as being *this* dialect or *that*—we will call each of them, simply, a *Lisp*. Now, of all the languages that run on the JVM, this Lisp is probably the most hard-core, functional programming language in existence, right up there with the likes of Common Lisp and Haskell. While Clojure is not a *pure* functional programming (FP) language like Haskell, it nonetheless is—unlike Scala, Groovy, JRuby—quite the radical departure from the programming model that most programmers are used to. Allow me to elaborate a bit...

Let's take Scala which, by the way, is an awesome language in its own right. Scala, though it clearly offers to developers the functional paradigm as part of its programming model, is a hybrid functional object-oriented language, and provides a *bridge* from object-orientation to FP. Thus, programmers can gradually embrace the functional aspects while operating within the familiar territory of object-oriented (OO) programming.

Enter Clojure, a language that does not provide the familiar OO model within whose comfortable confines most developers have been programming for years—to be sure, Clojure, being a Lisp, has *plenty* of arsenal to help you assemble any and all OO firepower imaginable, albeit in a *different* style than that which we associate with, say, the OO programming done in Java, or C# for that matter. Consider how Common Lisp, which is—along with Scheme and Clojure—one of the major Lisps that continues to flourish today, offers the highly sophisticated Common Lisp Object System (CLOS), whose wisdom has indisputably been captured peerlessly by Sonya Keene in her book Object-Oriented Programming in Common Lisp. But I digress...

Coming back to Clojure, you have in this stunningly powerful language an almost bewildering variety

of abstraction tools; naturally, this initially places extraordinary demands on a programmer venturing into this language for the first time. A bit like jumping into the deep-end of the swimming pool, getting started with Clojure can be a bit overwhelming.

But rest assured that all your efforts to tame the steep learning curve, which undeniably lies along the road to mastering the abstractions offered by Clojure, will be rewarded through a deepened understanding of the programming craft itself. Even if you don't end up doing your daily programming in this fine language, simply having access to its unparalleled strengths—and I haven't even mentioned the mind-expanding epiphany one has on first grokking Clojure *macros*—can be a source of inspiration. A short take on that is coming up next.

Clojure, being a Lisp, is the *ultimate programming language*—in the spirit of full disclosure, having made the preceding bold statement, I searched online to confirm the definition of a *smart aleck*, which Merriam Webster defines as "an obnoxiously conceited and self-assertive person with pretensions to smartness or cleverness", LOL ;)

Goodness, if your opinion of me gravitated (more like, um, lurched) toward the definition of a *smart aleck*, or a variation thereof—after reading the bold statement that a high quality Lisp will inevitably be the *ultimate programming language*—allow me to elaborate, albeit briefly, by quoting a Lisp luminary, Paul Graham. In his stellar manifesto on Lisp macros, which is entitled, simply, *On Lisp: Advanced Techniques for Common Lisp* (Prentice Hall), he observes that

> It’s difficult to convey the essence of a programming language in one sentence, but John Foderaro has come close: "Lisp is a programmable programming language".

Bear with me, if you will, and please suspend your judgment for a few moments: As you read through the following ideas and, in turn, the book reviews themselves that follow—the real meat of this post—my hope is that the rationale for my admittedly bold statement above will become clearer, especially by the time you reach the end.

Back now to Clojure, which of course is a functional programming language—but unlike a *pure* functional programming language such as Haskell, which has zero side-effects—Clojure *does* permit coding with side-effects, though it clearly *discourages* programmers from writing code which has side-effects (aka impure code).

Allow me to wedge in here the meme that, in Clojure—since it happens to be a Lisp—the experience of meta-programming feels far closer to the *regular* programming that you and I typically do in other languages. But I digress, again...

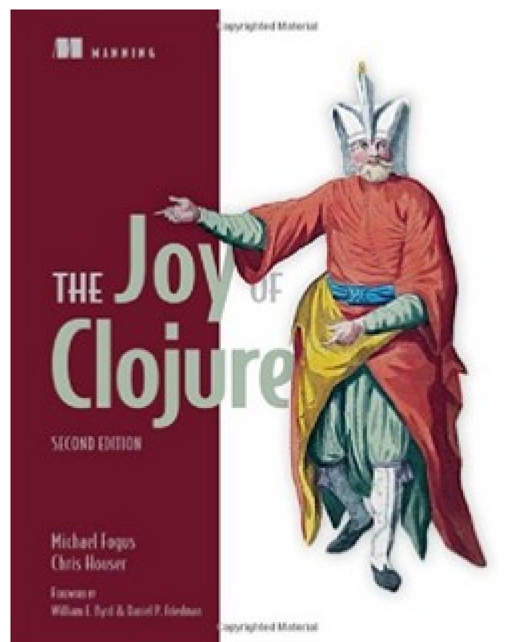
Importantly for the legions of Java programmers, Clojure is tightly integrated with Java and the JVM,

so all your legacy investment in Java codebases remains intact, while allowing seamless (bidirectional) interoperability between Java and Clojure. Like any other Lisp, your *code is data* in Clojure—see my comments on homo-iconicity, further along in this post. So programmers can wedge behavior in between reading and evaluating the code (macro-expansion). And *therein* lies the source of the immense power that Clojure programmers can wield to make programs bend to their will—yes, other languages also have macros, but all macros have not been created equal. But I digress, yet again...

OK, that was another digression, times three, to be precise—and given the proverbial, three strikes and you are out—I will cease and desist further ramblings ;)

So let's get right on with a rundown of the finest printed resources that I'm aware of, and which have helped me grok Clojure during the dabbling that I've done over the past several years. But first, I invite your comments—Once you've read my brief take each on the books below...

- Do you find that your experience of reading any of these books was different?
- Perhaps some qualities that I did not cover are the ones that you found the most helpful as you learned Clojure and its ecosystem.
- Did I leave out any of your favorite Clojure book(s)?
- I've covered only a *partial* list of the Clojure books that I've read, necessarily limited by the time available...



#1

The first spot is reserved for a phenomenal book, the likes of which I haven't seen in a great long

while: *The Joy of Clojure*, Second edition (Manning), by Michael Fogus and Chris Houser. It's significance makes me want to draw parallels to the Pulitzer prize-winning book *Gödel, Escher, Bach: An Eternal Golden Braid*, by Douglas R. Hofstadter. OK, so *The Joy of Clojure* is not going to win the Pulitzer any time soon—though there are passages peppered throughout, whose lyrical beauty is bound to grip your imagination—but what it has done for popularizing Clojure is right up there with what the illustrious *GEB* did for spreading the recursion meme ;)

In full candor, if you're new to Clojure, trying to follow the narratives in this book is going to be tough. Period. In fact, the authors unabashedly use the drink-from-the-proverbial-firehose approach, much as Steve Yegge noted in the *Foreword* to the first edition. But *The Joy of Clojure* can be a fabulous *third* or even *second* Clojure book. The reason for this can be traced to these words by its authors, in the *Preface*

> Specifically, this book is about how to write Clojure code *The Clojure Way*. Even more specifically, this book is about how experienced, successful Clojure programmers write Clojure code, and how the language itself influences the way they create software.

It's hard to do justice to capturing the elegance and clarity with which the authors share their deep understanding of exactly *what* makes Clojure tick. Trust me, their peeling back the onion layers to reveal the essence of this Lisp is highly entertaining, and anything but boring! Refreshingly free of hand-waving, their narratives—interleaved with plenty of succinct, high-quality Clojure code snippets—succeed spectacularly at pulling together all the strands of Clojure's abstractions into a well-knit fabric. The reader can reflect upon their newly garnered enlightenment, and confidently continue to deepen their grasp of *The Clojure way*.

To give you a flavor for the contents of *The Joy of Clojure*, the chapters in this book are nicely organized into the following six sections

> 1. Foundations:
> Chapter 1. Clojure philosophy
> Chapter 2. Drinking from the Clojure fire hose
> Chapter 3. Dipping your toes in the pool
>
> 2. Data types:
> Chapter 4. On scalars
> Chapter 5. Collection types
>
> 3. Functional programming techniques:
> Chapter 6. Being lazy and set in your ways
> Chapter 7. Functional programming
>
> 4. Large-scale design:
> Chapter 8. Macros

> Chapter 9. Combining data and code
> Chapter 10. Mutation and concurrency
> Chapter 11. Parallelism
>
> 5. Host symbiosis:
> Chapter 12. Java.next
> Chapter 13. Why ClojureScript?
>
> 6. Tangential considerations:
> Chapter 14. Data-oriented programming
> Chapter 15. Performance
> Chapter 16. Thinking programs
> Chapter 17. Clojure changes the way you think

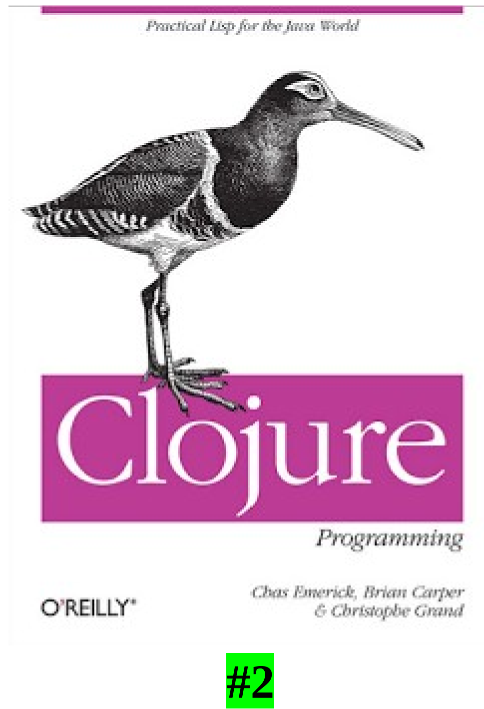
A *Resources* section toward the end contains just about the most fabulous set of eminently practical resources that I've seen anywhere. And it's not a dry list; witty remarks accompanying the entries in the *Resources* section make that section of the book enjoyable in its own right :)

Finally, here is Steve Yegge again, in the *Foreword* to the first edition of *The Joy of Clojure*, observing with his trademark perspicacity that

> In some sense, all this was inevitable, I think. Lisp—the notion of writing your code directly in tree form—is an idea that's discovered time and again. People have tried all sorts of crazy alternatives, writing code in XML or in opaque binary formats or using cumbersome code generators. But their artificial Byzantine empires always fall into disrepair or crush themselves into collapse, while Lisp, the road that wanders through time, remains simple, elegant, and pure. All we needed to get back on that road was a modern approach, and Rich Hickey has given it to us in Clojure.

By the way, even if you have read the first edition, you'll definitely want to get the second edition, which has been *significantly* revised to reflect, in the authors' own words, "...we felt that a second edition should include the lessons of our professional use of this amazing language. Nothing in this book is speculative. Instead, we've used every technique and library, from reducibles to core.logic to data-oriented design, to solve real systems problems".

Again, I can think of no other Clojure book that has brought as much joy to me as this one has; once you're reasonably comfortable in finding your way around Clojure code, you'll be ready to start partaking in the joy of Clojure as embodied in the pages of *The Joy of Clojure*!



#2

This book next up is an ideal first book. It is entitled, simply, Clojure Programming (O'Reilly) by Chas Emerick, Brian Carper, and Christophe Grand. Clearly a lot of thought went into putting together this eminently readable introduction to the Clojure programming language. There's a similarly-named book —entitled *Programming Clojure* (The Pragmatic Bookshelf), by Stuart Halloway and Aaron Bedra— which is a good book, so be sure to not confuse the one with the other ;)

For starters, *Clojure Programming* is replete with tons of high-quality Clojure code snippets which nicely illustrate the point each that those snippets seek to convey—I was able to load the snippets into the REPL in my IDE (IntelliJ IDEA Ultimate) and step through the code without any problem. What I especially liked about the snippets was the thoughtful annotations accompanying them.

I hasten to qualify with a caveat my earlier note about this being an introductory book—while it is definitely that, do note that it picks up pace rather quickly, so you'll need to stay alert and closely read (and re-read, as need be) the narratives. Adding that this is no watered-down treatment of Clojure by any stretch of the imagination. If you're looking for a serious, thought-provoking, first book as a companion and guide to your growing understanding of Clojure, this book is peerless.

So while there's not much hand-holding in this thoughtful book, the authors most emphatically do not throw the reader into the deep-end of the swimming pool either! Read along carefully—and do lots of experimentation at your Clojure REPL—and you'll do just fine. In fact, the authors pointedly remark in the *Preface* that

> Often the best way to learn is to dig straight into the nitty-gritty of how a language is used

> in the real world. If that sounds appealing, the hope is that you will find that at least a couple of the practicums resonate with what you do on a day-to-day basis, so that you can readily draw parallels between how you solve certain categories of problems in your current language(s) and how they may be solved using Clojure. You're going to bump into a lot of potentially foreign concepts and language constructs in those chapters—when you do, use that context within the domain in question as your entry point for understanding those concepts using the relevant instructional material in the first part of the book.

Far be it from me to pontificate, but I'll go ahead and throw this out there anyway: Learning Clojure programming won't be a cake-walk. Allow me to clarify... Unlike learning the typical programming language—each with its own unique syntax and whatnot—Clojure really doesn't have any syntax to speak of. Yes, I can hear you muttering under your breath, What is he talking about? Well, the deal is that with Clojure being a Lisp, you'll need to get used to writing your code directly in tree form, think abstract syntax tree-coding.

Hmm... That sounds dubious, you may well be thinking at this moment... Trust me, coding this way feels mightily unnatural at first ;)

This is probably best clarified by referring to what Michael Fogus and Chris Houser have to say on this exact point—in *The Joy of Clojure*—when they note that

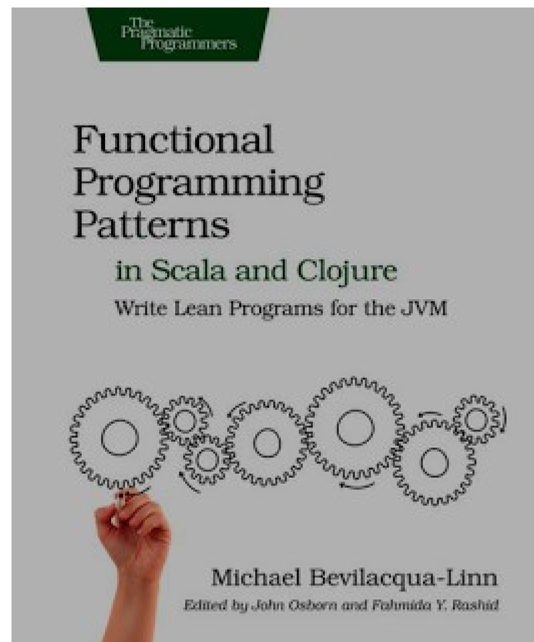
> Writing code is often a constant struggle against distraction, and every time a language requires you to think about syntax, operator precedence, or inheritance hierarchies, it exacerbates the problem. Clojure tries to stay out of your way by keeping things as simple as possible, not requiring you to go through a compile-and-run cycle to explore an idea, not requiring type declarations, and so on. It also gives you tools to mold the language itself so that the vocabulary and grammar available to you fit as well as possible to your problem domain. Clojure is expressive. It packs a punch, allowing you to perform highly complicated tasks succinctly without sacrificing comprehensibility.

Finally, you may be thinking, Why should I stretch myself by picking up Clojure when I've got so much already invested in the Java kingdom? That's an absolutely valid question, and which the authors nicely answer as follows in the *Preface* to this book, *Clojure Programming*

> There are millions of Java developers in the world, but some fewer number are working in demanding environments solving nontrivial, often domain-specific problems. If this describes you, you're probably always on the hunt for better tools, techniques, and practices that will boost your productivity and value to your team, organization, and community. In addition, you're probably at least somewhat frustrated with the constraints of Java compared to other languages, but you continue to find the JVM ecosystem compelling: its process maturity, massive third-party library selection, vendor support, and large skilled workforce is hard to walk away from, no matter how shiny and appealing alternative languages are.

If the preceding description fits you, then *Clojure Programming* is the book for you.

programming-digressions.blogspot.com/2015/08/best-clojure-books-are-we-there-yet.html



#3

This next book will be ideal for experienced OO programmers who *already* understand the basics of Clojure, and who now seek to acquire the functional programming style: Functional Programming Patterns in Scala and Clojure (The Pragmatic Bookshelf) by Michael Bevilacqua-Linn. Do note that it caters *equally* to the unique, functional programming style each offered by Clojure and Scala. In fact, the majority of the book revolves around a lengthy series of code examples, alternating between Clojure and Scala code.

In the *Preface*, the author convincingly makes the case for using *both* Clojure and Scala code to illustrate the essence of the functional programming style. Hence, he notes

> Both Scala and Clojure run on a Java virtual machine (JVM), so they interoperate well with existing Java libraries and have no issues being dropped into the JVM infrastructure. This makes them ideal to run alongside existing Java codebases. Finally, while both Scala and Clojure have functional features, they're quite different from each other. Learning to use both of them exposes us to a very broad range of functional programming paradigms.

Next, an excellent introductory chapter—which covers the whole notion of how patterns and functional programming are intertwined—kicks things off. It includes a superb, annotated glossary of all the patterns discussed in the book. Following that is a chapter containing an interesting, extended example (TinyWeb), which is in part a refresher of the basics of Scala and Clojure. The author then walks the reader through the steps to gradually transform TinyWeb from Java to Scala and Clojure, with notes on how to integrate Java code with Scala and Clojure.

programming-digressions.blogspot.com/2015/08/best-clojure-books-are-we-there-yet.html

What follow next are the two major sections that make up the remainder of *Functional Programming Patterns in Scala and Clojure*. The first section covers in detail the topic of replacing OO patterns by functional patterns, illustrated by lots of helpful code snippets—in both Scala and Clojure. The guidance on how to go about crafting functional replacements for OO patterns is thoughtful, and of high quality. The author elaborates on this approach by noting that

> By exploring both the similarities and the differences between Scala and Clojure, you should get a good feel for how each language approaches functional programming and how it differs from the traditional imperative style you may be used to.

The second section follows, delving into the details of patterns that are *native* to the functional paradigm. The rationale for the heavy reliance on immutability, the wisdom in making higher-order functions (HOFs) the primary unit of composition, and creating little languages to solve specific problems—think DSLs, which, by the way, Lisp introduced to the world—are all covered in this section.

For a flavor of this fine book's contents, this thoughtful books is organized like so

Table of Contents
Acknowledgments
Preface
How This Book Is Organized
Pattern Template
Why Scala and Clojure
How to Read This Book
Online Resources

1. Patterns and Functional Programming
1.1 What Is Functional Programming?
1.2 Pattern Glossary

2. TinyWeb: Patterns Working Together
2.1 Introducing TinyWeb
2.2 TinyWeb in Java
2.3 TinyWeb in Scala
2.4 TinyWeb in Clojure

3. Replacing Object-Oriented Patterns
3.1 Introduction
Pattern 1. Replacing Functional Interface
Pattern 2. Replacing State-Carrying Functional Interface
Pattern 3. Replacing Command
Pattern 4. Replacing Builder for Immutable Object
Pattern 5. Replacing Iterator

Pattern 6. Replacing Template Method
Pattern 7. Replacing Strategy
Pattern 8. Replacing Null Object
Pattern 9. Replacing Decorator
Pattern 10. Replacing Visitor
Pattern 11. Replacing Dependency Injection

4. Functional Patterns
4.1 Introduction
Pattern 12. Tail Recursion
Pattern 13. Mutual Recursion
Pattern 14. Filter-Map-Reduce
Pattern 15. Chain of Operations
Pattern 16. Function Builder
Pattern 17. Memoization
Pattern 18. Lazy Sequence
Pattern 19. Focused Mutability
Pattern 20. Customized Control Flow
Pattern 21. Domain-Specific Language

5. The End

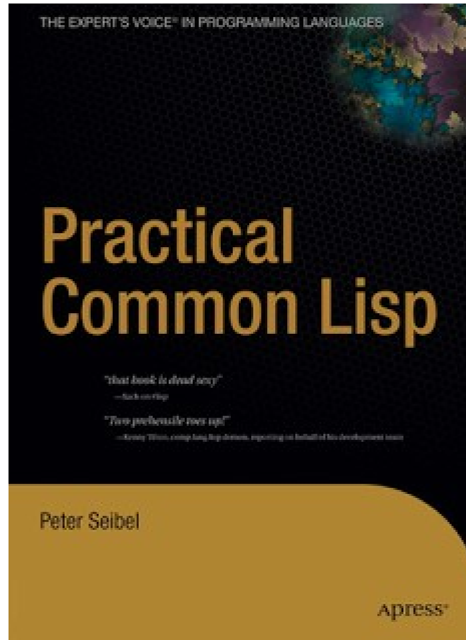
Bibliography

If any of the preceding topics excites you, or if you find yourself resonating with the author's viewpoint on the aims of *Functional Programming Patterns in Scala and Clojure*—which is as follows—then you'll find much of value in its pages

> Used together, these patterns let programmers solve problems faster and in a more concise, declarative style than with object-oriented programming alone. If you're using Java and want to see how functional programming can help you work more efficiently, or if you've started using Scala and Clojure and can't quite wrap your head around functional problem-solving, this is the book for you.

Aren't we lucky to have in Clojure a Lisp that runs on the stellar piece of software engineering that is the JVM? A blessing counted :)

programming-digressions.blogspot.com/2015/08/best-clojure-books-are-we-there-yet.html



#4

We have on our hands a powerful book in Practical Common Lisp (Apress) by Peter Seibel. Hmm... You may be wondering at this point, Isn't this post meant to be a *Clojure* books' who's who? Basically, what could you possibly learn about Clojure by reading about Common Lisp? A lot, as it turns out.

First, I've got a confession to make, since it directly ties in with this book: As much as I'm passionate about crafting quality software, I care *equally* about the craft of writing, seeking to serve the reader with unpretentiousness and humor. Stuffiness is out! As one classic Pink Floyd song memorably tells us, just the basic facts. Well, let's see now... So I jettisoned off my copies of *The Chicago Manual of Style* and *Roget's International Thesaurus*, ages ago—along with several hundred other books for reasons of expediency—when I pulled up stakes and moved from Minnesota to Texas. Oh, and I don't consult the dictionary much now when I write, plus I believe that all of us should have a good laugh at ourselves every now and then, because stuffiness just isn't for me ;)

OK, so here's what my preceding admission has to do with *Practical Common Lisp*: The writing is crisp and engaging, reminding me in several ways of the stellar writing with which Paul Graham has graced his essays and books—on the subject of Lisp programming, on acquiring the craft of programming, as well as on other diverse subjects. In this book, Peter Seibel regales us with hefty doses of eminently accessible Lisp wisdom, all served in a highly entertaining way. Yes, the Common Lisp language has a decidedly different vision than what we have in *The Clojure Way*. And then there is Scheme—which is the third of the three Lisps that continue to flourish—following yet another (decidedly minimalist) design philosophy that specifies a tiny standard core. But I digress...

The point is that all Lisps share a common heritage, which is cross-cutting across them all. So pretty much everything you learn from this book will help you out with understanding Clojure better.

Yes, while Clojure has introduced pleasant innovations—seamless interoperability with Java code, a rich set of literals for data and collections like vectors, maps, sets, and lists—it still retains its Lisp roots. It was really while reading and absorbing this wonderful book, *Practical Common Lisp*, that I really nailed down what makes Lisp macros tick.

This is an amazingly well-written book that you should not miss, even if you never end up writing even a single line of Common Lisp code.

Oh, and did I mention the delightful observation by the author in his intro that he "...is either a writer-turned-programmer or a programmer-turned-writer". That really sets the tone for the whole book. Granted, this book won't be a cake-walk, but then again, it's Lisp—a formidable language that has to be reckoned with on its own terms—that we're talking about. And with this power comes the inevitable complexity. But the fantastic news is that all the accompanying complexity is *essential* in nature, not *incidental*. Neal Ford has probably articulated this the best; here's his take on the tension between, and the dichotomy of, essential-accidental-versus-complexity in his fine book titled *The Productive Programmer* (O'Reilly)

> *Essential complexity* is the core of the problem we have to solve, and it consists of the parts of the software that are legitimately difficult problems. Most software problems contain some complexity. *Accidental complexity* is all the stuff that doesn't necessarily relate directly to the solution, but that we have to deal with anyway.

While the preceding quote from Neal Ford is in the context of the origins of service-oriented architecture (SOA)—an architectural style whose need derives from companies trying to mitigate the amount of accidental complexity that has built up over time—the notion of these two types of complexity can easily be generalized to the domain of programming languages. At this moment, you may well be thinking to some languages in which programming was a joy; in other languages, perhaps less so. And I didn't mention FORTRAN, or did I? Oops... Then again, let's try visualizing a book called *The Joy of FORTRAN*, shall we? ;)

So I think we agree that *all* languages inevitably carry some baggage—it's the relative degree of the conceptual burden that a programmer has to bear, when using the model of a given language, which sets a language apart from others. To put this in more concrete terms, allow me to share an anecdote based on my half-dozen years of programming in C++. With all due respect to those for whom the experience of programming in C++ is a pleasant and productive one—rather than the harrowing torment of Sisyphus, penitently toiling away at pushing the boulder uphill—given the *bewildering* variety of rules, exceptions-to-said-rules, which is the conceptual burden that a C++ programmer has to bear.

The setting for my half-dozen years of programming in C++ was the beautiful state of Minnesota—beautiful, but the harsh, frigid winter season, however, seemed to stretch interminably each year—so the more I would try to warm up to C++, the more that recalcitrant mule of a stubborn language would dig in its heels, responding with nothing but mirthless frostiness, ouch.

This rueful reminiscing notwithstanding, as a long-time subscriber to the magazine *MIT Technology Review*, I couldn't help but recall just how strongly I had resonated with what its then editor Jason Pontin had to say on this exact topic in the MIT TR editorial entitled "On Rules", back in January 1, 2007. Part of what he neatly articulated addresses precisely the same pain-point that I referred to a moment ago—the burden that some languages place on its practitioners. The editor went on to observe the

> ...generations of programmers who have struggled with C++'s quirks of syntax and *features overload*... A useful programming language should be what computer scientists call an *abstraction* of the underlying complexity of control flows and data structures. C++ preserves for programmers the maximum possible freedom of expression; but as "JLeslie" (another Slashdot commentator) admitted, "The cost was that it wasn't much of an abstraction."

And oh boy! I felt renewed vindication on further reading in the same *MIT Technology Review* editorial—right down to its use of the precise word, *bewildering*—the observation that

> C++ is notoriously hard to learn and use. Partly, this is owing to the difficulty of mastering the language's paradigmatic method... But mainly it is because software developers are free to express their ideas in C++ in a *bewildering* variety of forms.

Amen. But let's move on, shall we, onward and away from talk of purgatorial experiences?

Before we do, I feel compelled to reiterate, so please take note: The preceding impressions are *mine* alone. I would like to think, and in fact hope, that programming in C++ for some others has been *far* more pleasant than mine. I wish the C++ community—from which I parted many years ago—every success, and truly wish them well.

So when you program in Clojure—as you would with any Lisp—you get to define *your* own domain-specific languages (DSLs) to suit *your* needs. There's no conceptual burden whatsoever, because you become a language designer, making your own DSLs wherein you encode how your specific business use cases work. Without jumping ahead too much, I point you to a sublime quote from Paul Graham (in my review of his book *On Lisp: Advanced Techniques for Common Lisp*), which totally nails this idea down. That quote appears further along in this post, so stay tuned :)

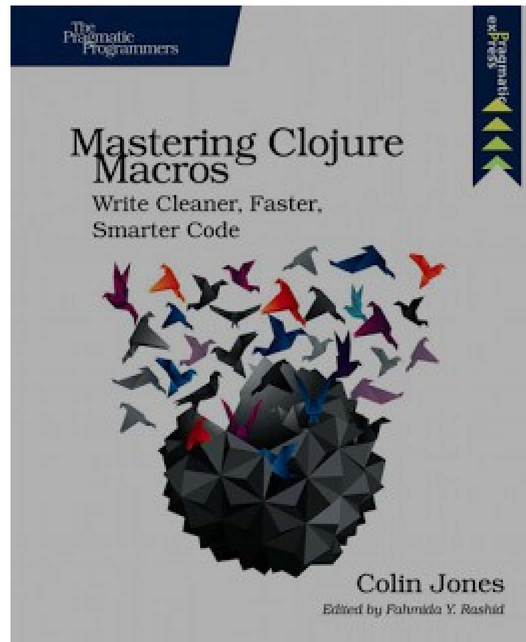
Ah, lest I forget, here's a brief rundown of the cool stuff which awaits the reader in the pages

of *Practical Common Lisp*

CHAPTER 1 Introduction: Why Lisp?
CHAPTER 2 Lather, Rinse, Repeat: A Tour of the REPL
CHAPTER 3 Practical: A Simple Database
CHAPTER 4 Syntax and Semantics
CHAPTER 5 Functions
CHAPTER 6 Variables
CHAPTER 7 Macros: Standard Control Constructs
CHAPTER 8 Macros: Defining Your Own
CHAPTER 9 Practical: Building a Unit Test Framework
CHAPTER 10 Numbers, Characters, and Strings
CHAPTER 11 Collections
CHAPTER 12 They Called It LISP for a Reason: List Processing
CHAPTER 13 Beyond Lists: Other Uses for Cons Cells
CHAPTER 14 Files and File I/O
CHAPTER 15 Practical: A Portable Pathname Library
CHAPTER 16 Object Reorientation: Generic Functions
CHAPTER 17 Object Reorientation: Classes
CHAPTER 18 A Few FORMAT Recipes
CHAPTER 19 Beyond Exception Handling: Conditions and Restarts
CHAPTER 20 The Special Operators
CHAPTER 21 Programming in the Large: Packages and Symbols
CHAPTER 22 LOOP for Black Belts
CHAPTER 23 Practical: A Spam Filter
CHAPTER 24 Practical: Parsing Binary Files
CHAPTER 25 Practical: An ID3 Parser
CHAPTER 26 Practical: Web Programming with AllegroServe
CHAPTER 27 Practical: An MP3 Database
CHAPTER 28 Practical: A Shoutcast Server
CHAPTER 29 Practical: An MP3 Browser
CHAPTER 30 Practical: An HTML Generation Library, The Interpreter
CHAPTER 31 Practical: An HTML Generation Library, the Compiler
CHAPTER 32 Conclusion: What's Next?
Index

For those venturing into Clojure land for the first time, this book will be an excellent companion to *Clojure Programming* (O'Reilly)—which was reviewed earlier in this post—and you'll find these two books complementing each other quite nicely. Good luck!

programming-digressions.blogspot.com/2015/08/best-clojure-books-are-we-there-yet.html



#5

I found a gem of a book in this title: Mastering Clojure Macros (The Pragmatic Bookshelf), by Colin Jones. I say petite because it's length is merely 100 pages. But appearances are deceptive, though... This happens to be a fast-moving, densely-packed introduction to Clojure macros, with plenty of mind-expanding ideas. In fact, this very paradox has got me waxing lyrical about its contents to the point where I'm taking poetic license and saying that

> While this book may well be petite and slender
> Trust me, it's in every way a true mind-bender

Hmm... Coming on the heels of the prior review, you're likely forming the impression of Clojure being a language unlike any other that you've encountered—assuming that you're a newcomer to the Lisp paradigm. In this regard, the following comments about this book, *Mastering Clojure Macros,* by legendary programmer, Robert "Uncle Bob" Martin will hopefully help clarify

> 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.

Much as its title heralds, this book is all about macros, up close and personal ;)

As a side-note, and if you want to stash away a data point on the topic of macros... When you really, really want to knock yourself out on macros—once you've got enough understanding of Clojure under your belt—look into tackling the meta mind-bender of a book entitled *Let Over Lambda: 50 years of Lisp*, by Doug Hoyte. This book by Hoyte will be a rich source of insights, of which the following is a

mere sample. In introducing the topic of "Closure-Oriented Programming", Hoyte quotes Guy Steele on the design of Scheme, when Guy Steele observed that

> One of the conclusions that we reached was that the "object" need not be a primitive notion in a programming language; one can build objects and their behavior from little more than assignable value cells and good old lambda expressions.

Hold on to that thought when you contemplate doing OO programming, and then some, in a Lisp such as Clojure...

OK, getting right back now to *Mastering Clojure Macros*! What I liked the most about this book is how the author thoughtfully presents considerable explanations to help the reader conceptualize exactly *what* happens behind the scenes to make macros possible in the first place. And as I mentioned in the preamble to this post: Yes, other languages *also* have macros, but all macros have *not* been created equal.

Basically, you can do meta-programming with Clojure macros that is simply impossible in other fine languages—including Ruby, C++, and even Scala, though there is an ongoing effort towards bringing compile-time meta-programming to Scala. Much as I love programming in Scala, I matter-of-factly have to confess that Lisps—because they are the only homo-iconic languages on this planet—provide the accruing raw programming power which you, as a Clojure programmer, can wield to bend programs to your will.

Very briefly, to help you assimilate my preceding remark on homo-iconicity, here's a remarkably compelling round-up of this concept, as described in the pages of *Clojure Programming* (O'Reilly) by by Chas Emerick et al, and which I also review in this post—the authors point out that

> The use of parentheses (as a textual representation of lists) is an outgrowth of Clojure being a homo-iconic language. We'll see what this means in (the side-notes for) *Homo-iconicity*, but the ramifications of it are manifold: homo-iconicity enables the development and use of meta-programming and domain-specific language constructs simply unavailable in any programming language that is not homo-iconic.

Circling back now to my earlier remark about how the author thoughtfully presents considerable explanations in *Mastering Clojure Macros* to help the reader conceptualize exactly *what* happens behind the scenes to make macros possible in the first place... Here are a couple of brief pointers from the book to give you a flavor of the intriguing discussions strewn across its pages, for example this one where the author tellingly notes that

> Macros in Clojure are an elegant meta-programming 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.

And a bit more nuts-and-bolts point—super helpful, of course—is when the author invites the reader to imagine like so

> ...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 meta-programming 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.

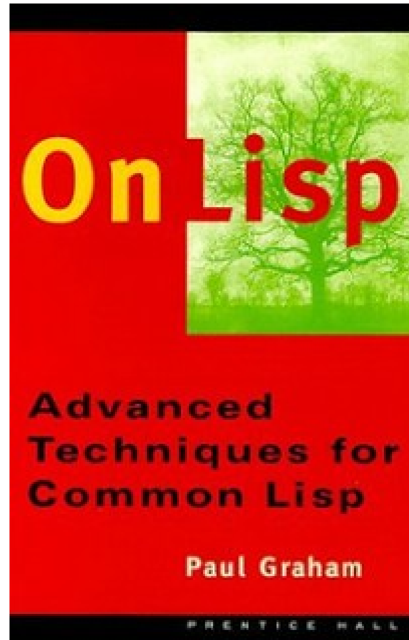
Be prepared to re-read *Mastering Clojure Macros* many times over; its brevity *utterly* belies the amount of knowledge that's packed in this slender book. Finally, to end this book's review, I leave you with an evocative quote, which could well be applied to undertaking the journey to grok Clojure macros

> There are subjects...that can be appreciated only if you make an effort proportional to their innate complexity. To hearken back to something like the language of courtly love in medieval France, there are certain objects of our affection that reveal their beauty and charm only when we make a chivalrous but determined assault on their defenses; they remain impregnable if we don't lay siege to the fortress of their inherent complexity.
>
> ~ Christian Queinnec, in ***L**isp **I**n **S**mall **P**ieces* (Cambridge University Press, 2003), p. xvi

In the end, here's a brief rundown of the contents in *Mastering Clojure Macros*

1. Build a Solid Foundation
2. Advance Your Macro Techniques
3. Use Your Powers Wisely
4. Evaluate Code in Context
5. Speed Up Your Systems
6. Build APIs That Say Just What They Mean
7. Bend Control Flow to Your Will
8. Implement New Language Features

You won't find a gentler guide to the profound subject of macros—Have fun with Clojure macros!



#6

In my mind—and it's a coincidence that we were talking about the Pulitzer prize earlier—here we have a Lisp hacker of the highest order, in the fine tradition of Timothy Hart and Guy Steele, besides being a writer extraordinaire whose writings could easily garner a Pulitzer. But oh well, founding the path-breaking Y Combinator is perhaps a decent-enough consolation prize ;)

We're of course talking about Paul Graham. I've written about Paul Graham and his stellar work in some of my earliest posts that you can find on this blog-site. And it's his book On Lisp: Advanced Techniques for Common Lisp (Prentice Hall) that I review here. This review is immediately followed by the review of another gem from him—I wish for Paul Graham to have the last word, so to say, because I can think of *no* other modern writer who brings as much wit and grace to the written page as he does with his works on Lisp and other diverse subjects.

With that, let's dive right into this tour de force :)

But first, allow me to fulfill a promise I had made earlier, while reviewing Peter Seibel's *Practical Common Lisp*, to be precise. I had noted then that a sublime quote from Paul Graham totally nails down the idea that Lisp programmers get to define *their* own domain-specific languages (DSLs) to suit *their* needs...

So I circled back to a post that I had published much earlier—you can read it in its entirety in an earlier post, entitled *On Paul Graham's Essays, and of Y Combinator*—I found the section, including the sublime quote that I wished to cite. Look in this quote, too, for the sublime metaphors that the

author uses in the second of the following two paragraphs with the cue sentence *Language and program evolve together*

> ...The traditional approach is called top-down design: you say "the purpose of the program is to do these seven things, so I divide it into seven major subroutines. The first subroutine has to do these four things, so it in turn will have four of its own subroutines," and so on. This process continues until the whole program has the right level of granularity—each part large enough to do something substantial, but small enough to be understood as a single unit.
>
> Experienced Lisp programmers divide up their programs differently. As well as top-down design, they follow a principle which could be called bottom-up design—changing the language to suit the problem. In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. As you're writing a program you may think "I wish Lisp had such-and-such an operator." So you go and write it. Afterward you realize that using the new operator would simplify the design of another part of the program, and so on. *Language and program evolve together* (italics are mine). Like the border between two warring states, the boundary between language and program is drawn and redrawn, until eventually it comes to rest along the mountains and rivers, the natural frontiers of your problem. In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small, and efficient.

I hasten to clarify that while I've read *numerous* portions of this utterly endearing book—many of them, *several* times over—but I don't in any way claim to have mastered its contents; mastering the art of macro-writing takes years of practice, and I've got merely a handful under my belt ;)

On Lisp is a highly literate, engaging, down-to-earth, and precise discussion of all things Lisp macros. With the exception of one other book, I'm not aware of any other books that cover this topic half as substantially; the other superb book I allude to is entitled *Let Over Lambda: 50 years of Lisp*, by Doug Hoyte. I would strongly suggest, though, that you *first* read the former, which will prepare you to tackle the latter.

Wrapping up that ever-so-brief segue into an allied gem as I did—Hoyte's book *Let Over Lambda*—here's an eloquent tribute from that book (to *On Lisp*), with its author's italicization kept intact

> *On Lisp* is one of those books that you either understand or you don't understand. You either adore it or you fear it. Starting with its very title, *On Lisp* is about creating programming abstractions which are layers *on top of lisp*. After we've created these abstractions we are free to create more programming abstractions which are successive layers on earlier abstractions.

And speaking of this book of Lisp lore—the magnum opus, *On Lisp*—pay close attention to how Paul Graham marvelously captures in the *Preface* the essence of what it means to program in Lisp

> When someone asked Louis Armstrong what jazz was, he replied “If you have to ask what jazz is, you’ll never know.” But he did answer the question in a way: he showed people what jazz was. That’s one way to explain the power of Lisp—to demonstrate techniques that would be difficult or impossible in other languages. Most books on programming—even books on Lisp programming—deal with the kinds of programs you could write in any language. *On Lisp* deals mostly with the kinds of programs you could only write in Lisp. Extensibility, bottom-up programming, interactive development, source code transformation, embedded languages—this is where Lisp shows to advantage.
>
> I hoped to do more than simply demonstrate the power of Lisp, though. I also wanted to explain *why* Lisp is different. This turns out to be a subtle question—too subtle to be answered with phrases like “symbolic computation.” What I have learned so far, I have tried to explain as clearly as I can.
>
> I hope reading this book will be fun. Of all the languages I know, I like Lisp the best, simply because it’s the most beautiful. This book is about Lisp at its lispiest. I had fun writing it, and I hope that comes through in the text.

I don't know about your reaction, but when I giddily read this passages, I sure fell for it, hook, line, and sinker—I'm a sucker for ravishing prose like that and, of course, for beautiful code.

Again, it's nearly impossible to convey a sense for the precision and elegance with which Paul Graham dissects, nay vivisects, the various topics he covers in the course of his wide-ranging discussions of all things Lisp macros. But I'll try some more and *paraphrase* what Jeremy Ashkenas—creator of CoffeeScript, Backbone.js, and Underscore.js—says about Clojure programmer extraordinaire Michael Fogus' book entitled *Functional JavaScript: Introducing Functional Programming with Underscore.js* (O’Reilly) in *that* book's *Foreword*

> This is a terribly exciting book... It breaks down... Lisp programming into its basic atoms, and builds it back up again into edifices of terrifying cleverness that will leave you wondering. It’s rare that a programming book can take you by surprise, but this one will.

Again, I paraphrased above the words of Jeremy Ashkenas to reflect my own, near-identical, sentiments toward *On Lisp*.

What's quite amazing, too, about *On Lisp* is the exposure to a plethora of Lisp technique you'll get by poring over its pages. In fact, as the author himself notes, "Just as a tour of New York could be a tour of most of the world’s cultures, a study of Lisp as the programmable programming language draws in most of Lisp technique".

Speaking of Lisp technique, when you're ready for exposure to additional mind-expanding technique, albeit with far less focus on Lisp macros—for which *On Lisp* and *Let Over Lambda* remain the champs—you simply can't go wrong with perusing the gem from Peter Norvig entitled *Artificial Intelligence*

Programming: Case Studies in Common Lisp, affectionately known to its adherents by the acronym PAIP.

At the moment, while I don't have room to delve into my opinions on the intriguing content of Peter Norvig's P**AI**P, I'll leave you with a couple of points that its author makes about Lisp, the programmable programming language, specifically about Common Lisp

> Lisp's flexibility allows it to adapt as programming styles change, but more importantly, Lisp can adapt to your particular programming problem. In other languages you fit your problem to the language; with Lisp you extend the *language* to fit your problem.
>
> Because of its flexibility, Lisp has been successful as a high-level language for rapid prototyping in areas such as **AI**, graphics, and user interfaces. Lisp has also been the dominant language for exploratory programming, where the problems are so complex that no clear solution is available at the start of the project. Much of **AI** falls under this heading.

With a blend of rigor and writing virtuosity permeating the pages of *On Lisp*—awash as this book is, in the Kool Aid of Lisp goodness—I wonder whether the superb light rock artist Steve Winwood, in his song *Don't You Know What The Night Can Do?*, perhaps had *Lisp* in mind when he sang mellifluously, intoning that ;)

> Now's the time that our dreams are finally coming true
> Feels so good, we're crying
>
> Now's the time when it's down to me and *you*
> Spread these wings, we'll be flying

OK, maybe my imagination ran away with me a bit, but you get the idea of how you get to be the language *designer* even as you wear your *programmer* hat? Yep, that's what Lisp buys you—Oh, and substitute the word 'Clojure' for wherever I used either 'Lisp' or 'Common Lisp' above, and you'll be good, for the most part :)

To round out this review, here then are the jewel-like chapters of *On Lisp*, each resplendent in its own shimmering lights, awaiting to irradiate the reader

1. The Extensible Language
2. Functions
3. Functional Programming
4. Utility Functions
5. Returning Functions
6. Functions as Representation
7. Macros
8. When to Use Macros
9. Variable Capture
10. Other Macro Pitfalls

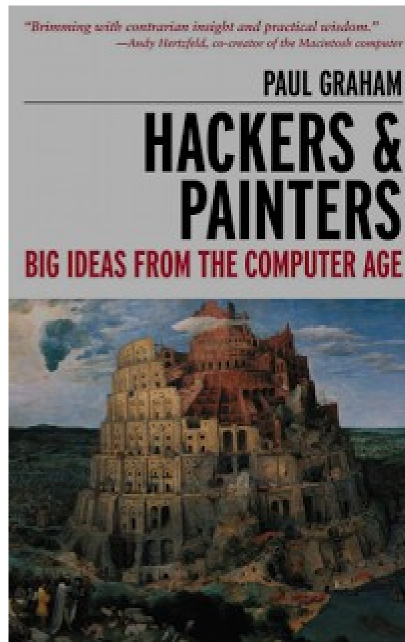
11. Classic Macros
12. Generalized Variables
13. Computation at Compile-Time
14. Anaphoric Macros
15. Macros Returning Functions
16. Macro-Defining Macros
17. Read-Macros
18. Destructuring
19. A Query Compiler
20. Continuations
21. Multiple Processes
22. Nondeterminism
23. Parsing with ATNs
24. Prolog
25. Object-Oriented Lisp

There you have it, a radiant gem of a book. And a first-class companion for *On Lisp*, should you wish to keep a reference handy while you study it, will be the following: So I also have a copy of Paul Graham's ANSI Common Lisp (Prentice Hall), which is another remarkably good book in its own right. What's really cool about the book *ANSI Common Lisp* is that it combines a top-notch—albeit, somewhat briefer, though not terse—introduction to Lisp programming, and a convenient, up-to-date reference manual for the standardized, ANSI Common Lisp language. Rounding out the second chapter of his book *ANSI Common Lisp* is, Paul Graham notes on p.28 that

> Richard Gabriel once half-jokingly described C as a language for writing Unix. We could likewise describe Lisp as a language for writing Lisp. But this is a different kind of statement... It opens up a new way of programming: as well as writing your program in the language, you can improve the language to suit your program. If you want to understand the essence of Lisp programming, this idea is a good place to start.

Enough said. There are a *ton* of substantial, extremely high-quality programming examples in *ANSI Common Lisp*, which bear the trademark virtuosity of its expert author-programmer-entrepreneur.

programming-digressions.blogspot.com/2015/08/best-clojure-books-are-we-there-yet.html



#7

Last, but certainly not the least, is this beguiling and highly entertaining book, Hackers & Painters: Big Ideas from the Computer Age (O'Reilly), by Paul Graham. In fact, if you recall my remark at the very outset of this post—where I had noted that "I wish for Paul Graham to have the last word, so to say"—I can think of *no* other modern writer who brings as much wit and grace to the written page as he does.

This charming book will be especially helpful to someone completely new to the Lisp family of languages, including Clojure. While there isn't much nuts-and-bolts programming advice at all in this book, there's definitely a hefty dose of eminently accessible material that'll help you appreciate and understand the Lisp philosophy. Two additional resource will also be worthy of your attention, so I mention them in the same breath

- Eric Raymond's wise post entitled *How To Become A Hacker*
- Peter Norvig's riotously entertaining and wise take on how to *Teach Yourself Programming in Ten Years*

The careful reader may have noted that one of the quotes that kicks things off in this post is from Eric Raymond's helpful post cited above. Let's now dive right into our final book review. But first let's disabuse ourselves of any vagueness that might surround the sense in which the overloaded word *hacker* is used here, both by the authors—in their respective books, which I'm reviewing here—and by myself. Here's how Paul Graham introduces this word in the *Preface* to *Hackers & Painters*

> Hackers? Aren't those the people who break into computers? Among outsiders, that's what the word means. But within the computer world, expert programmers refer to themselves as hackers. And since the purpose of this book is to explain how things really are in our world,

> I decided it was worth the risk to use the words we use.

With that definition to level-set our understanding of exactly who hackers are, I draw your attention to the fact that there's *so* much brilliantly endearing and accurate observations—including those on the role and impact of computing on the larger world, but more on that later—jam-packed into this book that you'll want to pore over every nook and cranny. And I literally mean, like, right down to the *Notes* section—which everyone, including myself, skips when reading a typical book—because *Hackers & Painters* is not your typical book. You don't want to miss the nuggets of wisdom tucked away in the recesses of this fantastic book. And I didn't even mention the top-notch *Glossary*, which appears before the *Index*.

To that end, here's the author's delightful observation on the topic of interruptions, which appears—see I told you to avoid skipping anything—in the *Notes* section ;)

> Different kinds of work have different time quanta. Someone proofreading a manuscript could probably be interrupted every fifteen minutes with little loss of productivity. But the time quantum for hacking is very long: it might take an hour just to load a problem into your head. So the cost of having someone from personnel call you about a form you forgot to fill out can be huge. This is why hackers give you such a baleful stare as they turn from their screen to answer your question. Inside their heads a giant house of cards is tottering.

Enough said.

What makes *Hackers & Painters* truly special is the exceptionally cultivated writing skill that the author wields in sharing insights, which of course are based on his *equally* exceptional programming skills. To get a sense for this virtuosity on display, in both spheres—the one where man communicates with fellow readers, the other where man communes with the computer—there's no substitute for reading the book itself. My impressions notwithstanding, and which I've tried to convey here, you really *have* to read this book and form your own assessment.

By the way, even as I wrote the second-half of the preceding point above—the sphere wherein man communes with the computer his codified logic—I was acutely aware of the following excellent advice shared by MIT professors Harold Abelson and Gerald Jay Sussman who, along with Julie Sussman, crafted the classic tome, *The Structure and Interpretation of Computer Programs*, from which to learn the fine art of programming (in the Lisp called Scheme). The authors note, in the *Preface* to the First Edition, that

> ...a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute.

So I've read *Hackers & Painters* at least six times since I bought it, way back when I lived in wintry

Minnesota. And unlike my experience with the assembly-language-on-steroids that is the C++ language, I positively warmed up to the Lisp philosophy enshrined in its pages.

Each time I re-read this delicious book, I seemed to find anew yet another insight, and this circling back continues as a virtuous cycle to this day, illuminating and informing me, deepening my appreciation for the role of computing—especially his stellar coverage of (Common) Lisp—in the digital world whose infrastructure is unerringly gravitating toward embracing the complexity-taming foundational work with which John McCarthy enlightened the functional world in 1958.

Here is Paul Graham's sparkling-with-wit take on this very topic

> What I mean is that Lisp was first discovered by John McCarthy in 1958, and popular programming languages are only now catching up with the ideas he developed then.
>
> Now, how could that be true? Isn't computer technology something that changes very rapidly? In1958, computers were refrigerator-sized behemoths with the processing power of a wristwatch. How could any technology that old even be relevant, let alone superior to the latest developments?
>
> I'll tell you how. It's because Lisp was not really designed to be a programming language, at least not in the sense we mean today... But in late1958, Steve Russell, one of McCarthy's grad students, looked at this definition of eval and realized that if he translated it into machine language, the result would be a Lisp interpreter.
>
> Suddenly, in a matter of weeks, McCarthy found his theoretical exercise transformed into an actual programming language—and a more powerful one than he had intended. So the short explanation of why this 1950s language is not obsolete is that it was not technology but math, and math doesn't get stale.

Hackers & Painters was published back in 2004, but it remains every bit as relevant and vibrant as it was when it was published. Here's a brief rundown of the topics which awaits the reader in the pages of this entertaining book

1. Why Nerds Are Unpopular
2. Hackers and Painters
3. What You Can't Say
4. Good Bad Attitude
5. The Other Road Ahead
6. How to Make Wealth
7. Mind the Gap
8. A Plan for Spam
9. Taste for Makers
10. Programming Languages Explained
11. The Hundred-Year Language
12. Beating the Averages

13. Revenge of the Nerds
14. The Dream Language
15. Design and Research

Notes
Glossary

While the entire book is uniformly excellent, the standout chapters are the final six, beginning with Chapter 10, which is entitled *Programming Languages Explained*. There's simply tons of good stuff in there that you don't want to miss :)

To drill down just a bit into one of those chapters, entitled *Revenge of the Nerds*, where Paul Graham—in leading up to a fabulous section called *What Made Lisp Different*—notes that, "So the short explanation of why this 1950s language is not obsolete is that it was not technology but math, and math doesn't get stale". The author then goes on to observe

> When it was first developed, Lisp embodied nine new ideas. Some of these we now take for granted, others are only seen in more advanced languages, and two are still unique to Lisp. The nine ideas are, in order of their adoption by the mainstream...

He then regales the reader with a highly accessible commentary on those nine ideas, which I simply cite here

1. Conditionals.
2. A function type.
3. Recursion.
4. Dynamic typing.
5. Garbage-collection.
6. Programs composed of expressions.
7. A symbol type.
8. A notation for code using trees of symbols and constants.
9. The whole language there all the time.

Paul Graham's take on the last idea in the preceding list—in his memorable words, "The whole language there all the time"—is in particular just plain ineffable, and worth lingering on!

> Many languages have something called a macro. But Lisp macros are unique. And believe it or not, what they do is related to the parentheses. The designers of Lisp didn't put all those parentheses in the language just to be different...
>
> ...You write programs in the parse trees that get generated within the compiler when other languages are parsed. But these parse trees are fully accessible to your programs. You can write programs that manipulate them. In Lisp, these programs are called macros. They are programs that write programs.

programming-digressions.blogspot.com/2015/08/best-clojure-books-are-we-there-yet.html

> Programs that write programs? When would you ever want to do that? Not very often, if you think in Cobol. All the time, if you think in Lisp.

So there, I've done my level best to give you a flavor of this marvelous book, *Hackers & Painters*, and a highly opinionated one at that, likely tinged by my subjective impressions. You are on to something if you think—well, I do so, anyway—that we have in this book an expert chiseling away at ideas plus some amazing conceptual artistry...

Lambda calculus (λ-calculus), the inspiration behind functional programming

Oh, goodness. I wasn't quite planning on this... So I dwell in a world powered by the duo dynamos of Java and Scala, along with their respective frameworks each. But my fondness for Clojure in particular, and the Lisp philosophy in general, never really left me, I realize. Functional programming, math, oh my!

In putting together my thoughts for these reviews—in particular while reviewing the last two books, where I let Paul Graham have the last word—I'm grippingly reminded, yet again, how some things just never go out of style. Ever.

We keep coming back to the basics. Functional programming is inspiring a renaissance in the way we craft software. MapReduce, Domain-Specific Languages, and Lambda Architecture, anyone?

This point is underscored by Michael Fogus and Chris Houser in their stellar book, *The Joy of Clojure*, where they invite the reader to

> Go to any open source project hosting site, and perform a search for the term Lisp interpreter. You'll likely get a cyclopean mountain of results from this seemingly innocuous term. The fact of the matter is that the history of computer science is littered with the abandoned husks of Lisp implementations. Well-intentioned Lisps have come and gone and been ridiculed along the way, and yet tomorrow, search results for hobby and academic Lisps will have grown almost without bounds.

And yes—Clojure, Common Lisp, Scheme—long live all the Lisps of the world!

The loveliness of virtuous feedback—Escher has done it, yet again!

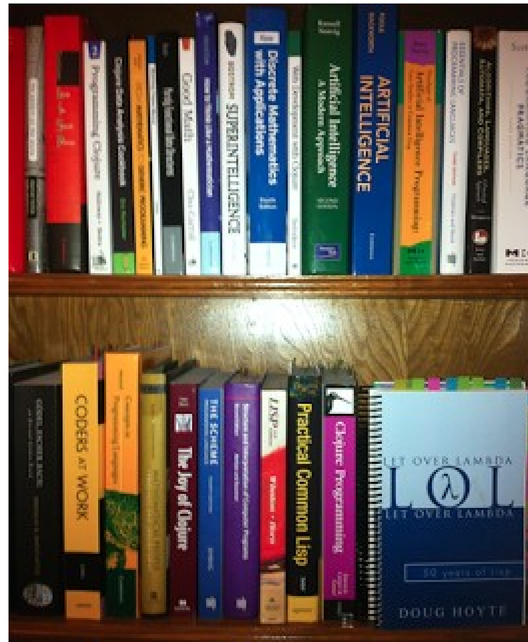
And so we come to the end. Much as I mentioned at the outset, I invite your comments—Having now read my brief take each on the preceding half-dozen books...

- Do you find that your experience of reading any of these books was different than mine?
- Perhaps some qualities that I didn't cover are the ones that you found the most helpful as you learned Clojure and its ecosystem.
- Did I leave out any of your favorite Clojure book(s)?
- If so, what could I also review, perhaps in a future blog post.
- I've covered only a *partial* list of the Clojure books that I've read, necessarily limited by the time available...

My hope is that these brief vignettes will help you in your journey to grokking Clojure—meandering out of necessity through its close cousin, Common Lisp—otherwise, you will miss out on some evergreen classics that inform how Clojure has been put together.

Bon voyage, and I leave you with a photo of a pseudo-random section of one of my bookshelves—heavens behold, I do confess this section se a tad biased toward Clojure material ;)

programming-digressions.blogspot.com/2015/08/best-clojure-books-are-we-there-yet.html



Posted by Akram Ahmad at

Email ThisBlogThis!Share to TwitterShare to FacebookShare to Pinterest

1 comment:

1.

   *Akram Ahmad*August 9, 2015 at 8:12 AM

   Adding an afterthought here, more akin to a post-script :)

   My hope, again, is that these brief vignettes will help you in your journey to grokking Clojure. But I enjoy writing—more on that in a moment—so that I would write even if I had an audience of only one :)

   But as I had commented in a recent post, **Best Scala Books**, I was pleasantly taken by surprise—I had glanced at the statistics of visitors to this post to learn that over 3,800 readers like you (and counting) have now made the time to come read what I had written. So thank you for your gracious time and interest!

programming-digressions.blogspot.com/2015/08/best-clojure-books-are-we-there-yet.html

As promised in my comment on that post, this post on Clojure is my way of saying, thank you!

Ah, lest I forget, to address my point above about how I derive joy from the craft of writing this blog on technical subjects that intrigue me—So a large part of my enjoyment is informed by the following mindset. Hearkening back to what someone has thoughtfully said, *I write code to better understand what I design*. Along with my resonating with the preceding metaphor during my daily work of design and coding, and closer to the current theme, *I write prose to better formulate what I think*.

Again, thank you. And I encourage you to share your thoughts and reactions to what you read on this blog, through your comments!