# Reading the CS Canon

Harry R. Schwartz

July 3, 2014

Last year, a friend of mine who was learning computer science asked me for some reading advice. I wrote him the following big list of suggestions. If you're looking for reading suggestions, they might be useful for you, too!

It's biased toward classic books over papers, since it's meant to be approachable. It's also (unsurprisingly) biased towards things I think are especially interesting, so it thoroughly covers distributed systems, theory, AI, and algorithms while totally ignoring (for example) security, graphics, and databases.

I've included links to PDFs of papers when I could easily find them.

## Learning the environment

**The UNIX Programming Environment**, *Kernighan and Pike.* Learn to use the terminal.

**Learning the bash Shell**, *Newham and Rosenblatt.* You're gonna have to learn shell scripting, so just make your peace with it.

Once you've gotten pretty familiar with this stuff, and you've put in a few dozen hours of programming, it might be worth it to invest some time to learn a few handy tools. Specifically:

- A serious editor. Emacs and vi are both great (though one is better than the other, *cough*)
- Version control. git seems to have won, so start with that.
- Regular expressions.

## Writing good code

**Test-Driven Development By Example**, *Beck.* TDD is pretty darn important, and Kent Beck is its prophet.

**Design Patterns: Elements of Reusable Object-Oriented Software** (the "Gang of Four" book), *Gamma, Helm, Johnson, and Vlissides.* A supremely useful book if you spend a lot of time in a static language like Java. Otherwise this next one is way better:

**Design Patterns in Ruby**, *Olson.* Infinitely more useful than Gang of Four if you're writing in a dynamic language. And a more enjoyable read, in my opinion.

**Clean Code**, *Martin.* Learn about code smells!

And what the hell, a couple classic papers. They're mostly interesting *historically*, but that's only because we've absorbed the ideas in them so completely:

- Dijkstra, *Go To Statement Considered Harmful*
- Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*

## Being a good programmer

**The Pragmatic Programmer**, *Hunt & Thomas.* Seriously, so good. Learn why there's a rubber ducky on my desk!

**The Mythical Man-month**, *Brooks.* Why adding people to a late project makes it later, and stories about IBM in the 60s.

## Ruby (+ Rails, why not!)

**Eloquent Ruby**, *Olson.* The best book for learning Ruby. Russ Olson is one of my favorite technical writers.

**Practical Object-Oriented Design in Ruby**, *Metz.* What it says on the tin: learn to use good OOP style in Ruby. Sandi Metz is terrific.

**Metaprogramming Ruby**, *Perrotta.* Learn to sin against the Ruby object system. Punch some ducks.

**Agile Web Development with Rails**, *Ruby, Thomas, Hansson.* The book I first read to learn Rails.

## JavaScript

**JavaScript: The Good Parts**, *Crockford.* I'm not much of a JS programmer, but this is the standard book about how to write good Javascript. Focused on the language and good style, but less interested in, say, building interactive websites.

**Eloquent JavaScript**, *Haverbeke.* I haven't read this one, but I've heard good things. It's a bit more introductory than Crockford, and eventually it gets in to DOM manipulation, so it might be relevant if you're looking to build dynamic sites.

## Algorithms

**Algorithms Unlocked**, *Cormen.* A relatively gentle introduction to the world of algorithmic analysis.

**Introduction to Algorithms** ("CLRS"), *Cormen, Leiserson, Rivest, & Stein.* The standard tome.

**Algorithms**, *Dasgupta, Papadimitriou, & Vazirani.* Covers some stuff (like FFTs and quantum algorithms) better than CLRS. And alternate explanations are always nice.

**Introduction to Automata Theory, Languages, and Computation** (the "Cinderella Book"), *Hopcroft, Motwani, & Ullman.* Haven't read it, but my officemate in school was a fan.

**The Art of Computer Programming** ("TAOCP"), *Knuth.* No one has ever completely read these books, but they look damn good on a shelf. Every now and then you'll come across a problem worthy of cracking these books open and you'll feel like a *badass.*

**How to Solve It: Modern Heuristics**, *Michalewicz & Fogel.* I love this book! Specifically, I love dealing with the problems that this book will help you solve.

**An Introduction to Genetic Algorithms**, *Mitchell.* The only good book on genetic algorithms that I'm aware of.

**Programming Collective Intelligence**, *Segaran.* Neat stuff. Might be good for getting your feet wet. Uses Python. The code is a little atrocious, but it demonstrates the concepts.

## Compilers

**Compilers: Principles, Techniques, and Tools** ("the dragon book"), *Aho, Sethi, & Ullman.* Still the definitive book on the subject, but (IMO) a terrible way to learn to actually build a compiler.

The following papers are a better intro:

- Ghuloum, *An Incremental Approach to Compiler Construction*
- Crenshaw, *Let's Build a Compiler*
- Sarkar, Waddell, & Dybvig, *A Nanopass Framework for Compiler Education*
- Schorre, *META II: A Syntax-Oriented Compiler Writing Language*

## AI & Machine Learning

**Artifical Intelligence: A Modern Approach**, *Russell & Norvig.* The standard undergrad text. Pretty darn good.

**Paradigms of Artificial Intelligence Programming** ("PAIP"), *Norvig.* A guide to good Lisp programming disguised as a collection of AI techniques.

**Pattern Recognition and Machine Learning**, *Bishop.* The standard intro to statistical machine learning.

**Neural Networks: A Systematic Introduction**, *Rojas.* Not too famous, but it's my favorite book on artificial neural networks.

**Neural Networks for Pattern Recognition**, *Bishop.* Just what it sounds like.

## Networking

**Computer Networking: A Top-Down Approach**, *Kurose.* The standard undergrad text. Covers most of the stuff you'd need to know.

**TCP/IP Illustrated, Volume I**, *Stevens.* Covers TCP (a key networking protocol) in painstaking depth.

## C, Systems, and OSes

**The C Programming Language** ("K&R"), *Kernighan & Ritchie.* Ya gots to learn C to write OSes, son. This is the classic intro, though modern C development practices have moved on significantly.

**Operating System Concepts** ("the dinosaur book"), *Silberschatz.* My undergrad textbook. No complaints.

**Modern Operating Systems**, *Tanenbaum.* I really want to read this. Apparently it's great: Linus Torvalds used it to write Linux.

**The Design of the UNIX Operating System**, *Bach.* A great case study of how an OS works.

## Theory of Computation

**Introduction to the Theory of Computation**, *Sipser.* The standard undergraduate textbook on the subject.

**An Introduction to Formal Languages and Automata**, *Linz.* A slightly less rigorous alternative.

**Computers and Intractability**, *Garey & Johnson.* Heavy stuff, but if you find that you're writing a lot of NP-completeness proofs it's basically the best.

## Distributed Systems

First, N.B. that there are (at least) two alternate definitions of what a *distributed system* is:

1. A symmetric system of approximately-identical nodes running the same code. This is generally what academics mean when they talk about "distributed systems." Think flight control systems, highly redundant systems to manage nuclear reactors, etc.—stuff you could prove theorems about.

2. A large system of heterogeneous machines running different services (maybe some databases, some queues, some exposed APIs, etc). This is usually what working engineers mean by "distributed system." Twitter's ingestion system would be an example of this kind of distributed system.

### Definition #1

**Distributed Algorithms: An Intuitive Approach**, *Fokkink.* The only distributed systems textbook that I've liked so far.

The Fokkink book should really be supplemented with papers. The following papers cover most of the high points, and Lamport and Dijkstra in particular are excellent writers. These are ordered in a semi-sensible way.

- Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*
- Suzuki and Kasami, *A Distributed Mutual Exclusion Algorithm*
- Dijkstra, *Self-stabilizing Systems in Spite of Distributed Control*
- Hoare, *Communicating Sequential Processes*
- Schlichting, *Using Message Passing for Distributed Programming: Proof Rules and Disciplines*
- Lamport, Shostak, and Pease, *The Byzantine Generals Problem*
- Lai and Yang, *On Distributed Snapshots*
- Dijkstra, Feijen, and van Gasteren, *Derivation of a Termination Detection Algorithm for Distributed Computations*
- Lamport, *Paxos Made Simple*

Christopher Meiklejohn also maintains a nice list of canonical distributed systems papers.

### Definition #2

Definition #2 doesn't really have a canon that I'm aware of; it's mostly composed of inherited folklore and ever-changing best practices. One of the hot terms is "Service-Oriented Architecture," so that might get you started in the blogs.

On the subject of "programming language theory applied to building distributed systems," consider plowing through Joe Armstrong's PhD thesis, *Making Reliable Distributed Systems in the Presence of Software Errors.*

## Lisp & Scheme

**Structure and Interpretation of Computer Programs** ("the wizard book"), *Abelson and Sussman.* The undergrad text at MIT for a bunch of years. Still considered a rite of passage for learning to program. There are also an excellent set of accompanying lectures on OCW.

**On Lisp**, *Graham.* Macros! They're great. Being able to rewrite how your language works has some consequences.

**The Art of the Metaobject Protocol**, *Kiczales, des Rivieres, and Bobrow.* Lisp can orient the *hell* out of some objects.

If you've read through those other Lisp/Scheme books, you should probably consider ploughing your way through the "Lambda Papers."

## Programming Language Theory

**Concepts of Programming Languages**, *Sebesta.* My undergrad textbook. It's fine.

**Types and Programming Languages**, *Pierce.* I still haven't read this one, but I'd love to. Supposedly it's awesome.

If you find yourself drawn deeper into functional programming, consider:

- Okasaki, *Purely Functional Data Structures*
- Pierce, *Basic Category Theory for Computer Scientists*

A couple fun papers on the topic include:

- Wadler, *Propositions as Types*
- Hughes, *Why Functional Programming Matters*
- Meijer, Fokkinga, and Paterson, *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*