Common Lisp A Gentle Introduction To Symbolic Computation Review

Introduction

Common Lisp: A Gentle Introduction to Symbolic Computation is an introductory text on Common Lisp. It is designed for use as an introduction to programming in general, slowly easing readers into different concepts like functions and variables. It includes small exercises throughout each chapter, and beginning in chapter 3, it includes one final larger exercise at the end of each chapter that allows readers to apply all the concepts learned in one larger project.

It begins very slowly, with the first two chapters explaining the fundamentals of programming with Lisp without any code or coding assignments.

It doesn't provide any information on programming environments or text editors, relying on instructors in classes to provide students with the necessary resources. Due to the lack of code and information on coding environment instruction, it feels very slow to start and incomplete for use as a book for self-learners. If you want to use it as an introduction to Common Lisp, I recommend first learning Emacs. Given the lack of exercises or code examples, I recommend moving swiftly through the first 76 pages of the book and backtracking if you feel lost later.

The question is: Given the apparent weaknesss above, should you even bother reading Touretzky's book?

If you're learning programming on your own and new to Emacs, I'd say no. Learning Emacs is a rather large hurdle to overcome at the beginning of your journey. Even if the knowledge you gain in this book is good, the difficulty starting out will probably discourage you from continuing anyway. If you're determined, then I must emphasize that learning Emacs will be critical to learning Common Lisp. Start with downloading an Emacs distro like Doom or Spacemacs, watch some instructional videos on its features and how to use it, and then once you're comfortable using Emacs, continue learning Common Lisp.

If you know Emacs, should you study with this book?

The answer is: Yes! As a full-stack web developer, I'm familiar with both JavaScript and Python. While I like web development, I have worried about specializing too much as a frameworker or a UI builder, missing out on more fundamental knowledge of computers and programming. I enjoy programming and want to master it, not just get paid.

In an effort to expand my fundamental knowledge, I decided to learn Common Lisp, believing that it was a functional programming language ala Haskell or Ocaml. As I learned more about Lisp before getting too far in the book, I realized that Common Lisp really is paradigm agnostic, leaving developers to choose their preferred mode of programming. I was worried that I might not be exposed to enough new concepts while learning Common Lisp to make the exercise worth-while, but I was pleasantly surprised to find that wasn't the case.

So as a web developer, what did I learn?

Mapping, Filtering, and Recursion

One goal of mine was to learn "functional" programming techniques like mapping, filtering, and recursion. Recursion isn't exclusively a functional programming technique, but my impression is that recursion is used more in functional programming languages compared to languages like JS or Python. While mapping and filtering is possible in both JS and Python, especially in Python such techniques are considered "non-Pythonic" and generally discouraged. And since JS is really designed more for imperative programming, I figured learning Common Lisp would expose me to more "correct" ways of using mapping and filtering.

Touretzky did not disappoint. Iteration operators like `DO`, `DOLIST`, and `DOTIMES` aren't introduced until chapter 10. Instead, the first tools Touretzky teaches for working on sets of data are "applicative programming" tools, i.e. mapping and filtering. Perfect. In that chapter, the final exercise involves learning how to make what I would describe as a primitive querying API for querying a simple, list-based database.

In the next chapter, Touretzky teaches recursion. I have previously tried learning recursion via the venerable Structures and Interpretations of Computer Programs by Sussman and Abelson, and every time I felt overwhelmed by the explanations and exercises that require a bit more mathematical skill than I can muster. As a result, recursion has had a bit of a mysterious aura to me.

Fortunately, Touretzky slowly and gently teaches both how to use, and when to use recursion. Although I can't say that I am now an expert on recursion, I feel much more confident than I was prior to completing the chapter.

Overall, I was more than satisfied with Touretzky's greater emphasis on functional programming and recursion.

Data-Oriented Programming vs. Abstractions

Over the last year or so, I've been exposed to the "Data-Oriented Programming" paradigm championed by Jonathan Blow, Casey Muratori, and Mike Acton.

Simply put, those three horsemen of good data structures believe that object oriented programming techniques that rely on heavy abstractions are bad for building good software. The main reason is because the abstractions rely on modeling data in ways that lead to poor speed and reliability in software. Additionally, the heavy layering of abstractions makes the resulting code much more difficult to understand.

Lisp is notable for being a language that provides a lot of flexibility for making abstractions. While the linked-lists that provide the foundational data structure for Lisp are inefficient compared to other data types (a topic also covered in the book), they are excellent for prototyping. When beginning a project, it's recommended to use lists first and then "optimize" with other data structures when necessary.

But just because you **can** make clever abstractions, that doesn't mean you always should. Rather than relying on "clever" abstractions, is it possible to take a more "Data-Oriented" approach to programming in Lisp, while also using the flexibilty it provides for creating abstractions when that's necessary?

After reading this Gentle Introduction, my preliminary judgement is: Yes.

Data Structure Example: Tic-Tac-Toe

Tic-Tac-Toe is a classic toy project for learning Common Lisp. I actually watched some Youtube videos teaching how to do tic-tac-toe in Common Lisp. I found them rather difficult treatments of the problem, but I wasn't sure why.

The final project for the Assignment chapter happens to be tic-tac-toe. After learning Touretzky's method of making a tic-tac-toe game–complete with an AI computer player!–I am more convinced than ever that an emphasis on good, simple data structures can vastly improve the quality of code, and make solving certain problems far simpler than using more "obvious" but also more complicated data structures.

1. The Complicated Way

   ```lisp
   ;; Using a 2D array
   (defparameter *board*
     (make-array '(3 3) :initial-element '-))

   ;; Using nested lists
   (defparameter *board*
     (list (list nil nil nil)
           (list nil nil nil)
           (list nil nil nil)))
   ```

   The more complicated method of making a tic-tac-toe game models the game board as a 2D array or nested lists, with player icons modeled as string Xs and Os and empty spaces modeled as underscores or `NIL`. This is a more obvious and direct modeling of the game.

2. The Touretzky Method

   ```
   (defun make-board ()
     (list 'board 0 0 0 0 0 0 0 0 0))
   ```

   The Touretzky method models the game board as a simple flat list, player icons as ones and tens, and empty spaces as zeros. This model of the game is non-obvious and doesn't reflect what the game looks like in real life.

   That one difference–the choice of how to model the data–has a profound impact on the rest of the code. Lists are simpler to use and manipulate than arrays, and flat lists are especially easier than 2D arrays. Strings require extra manual work to model win-conditions, whereas simple math is all that is necessary to model win-conditions with the Touretzky method. The choice of flat lists of numbers, rather than 2D arrays of strings, completely changes the level of complexity of the rest of the code.

   While the tic-tac-toe exercise was the most instructional one for demonstrating the importance of choosing good data structures and modeling, several other exercises also demonstrate the power of simple data structures.

   Overall, Touretzky's treatment of data structures throughout the book was profoundly instructional and inspirational. It leaves me hope that there is a way to thread the needle between Sussman and Muratori using good, simple sets of abstractions to work on simple data structures.

Other stuff

Touretzky teaches all the other basics, like all of the important list processing operators, how to take user input, how to print data, how to use structures/arrays/hash tables, and the final chapter also teaches the basics of using macros.

The most conspicuously missing language construct missing from the book is the the Common Lisp Object System–CL's object oriented programming construct. Since my intention was to learn functional programming operations and recursion, I wasn't bothered at all by the lack of `DEFCLASS` or `DEFMETHOD` in the book.

Final Judgement

Overall, Common Lisp: A Gentle Introduction to Symbolic Computation is an excellent introduction to Common Lisp. While it starts slow, and in a modern context a working knowledge of Emacs is a prerequisite for learning Common Lisp and beginning the book, it has provided me both an excellent foundation for programming in Common Lisp and a greater understanding of programming methods and concepts that I can apply in any language I use in the future.

If you're interested in learning Common Lisp and wonder if you will get any value out of Touretzky's book, wonder no more.

Final Judgement: 8/10

While you can (and I did) purchase a new copy of the book, a PDF of the book is also available for free.