A Road to Common Lisp

Posted on August 27, 2018.

I've gotten a bunch of emails asking for advice on how to learn Common Lisp in the present day. I decided to write down all the advice I've been giving through email and social media posts in the hopes that someone might find it useful.

One disclaimer up front: this is *a* road to Common Lisp, not *the* road to Common Lisp. It's what I followed (without some of the dead ends) and has a *lot* of my personal opinions baked in, but it is by no means the only way to learn the language.

Context

I think it's important to have a sense of where Common Lisp came from and what kind of a language it is before you start learning it. There are some things that will seem very strange if you're coming straight from modern languages, but will make more sense if you've got a bit of background context.

History

Common Lisp has a long, deep history. I'm not going to try to cover it all here — if you're interested you should check out some of the following (in roughly increasing order of detail):

- Wikipedia's History of Lisp and History of Common Lisp.
- The Where it Began section in Practical Common Lisp.
- The History: Where did Lisp come from? section of the comp.lang.lisp FAQ.
- Common Lisp: the Untold Story by Kent Pitman.
- The Evolution of Lisp by Guy Steele and Richard Gabriel.

I realize you probably won't want to read all of the links above immediately, so here's a whirlwind tour of sixty years of Lisp.

Lisp began in the late 1950's. It was invented by John McCarthy at MIT.

Over the next twenty or so years various versions and dialects of Lisp grew and flourished. Some of the more notable dialects were Maclisp, BBN Lisp/Interlisp, Franz Lisp, Spice Lisp, and Lisp Machine Lisp. There were others too. The point is that there were a *lot* of different implementations, all growing, changing, and trying out different things.

(Scheme also originated in this time frame, but took a very different route and diverged from the path we're looking at. I won't cover Scheme in this post.)

In the early 1980s people decided that having a whole slew of mutually-incompatible dialects of Lisp might be not be ideal. An effort was made to take these different languages that had grown organically and produce one common language that would satisfy the needs of everyone (or at least a reasonable subset of "everyone"). In 1984 the first edition of Guy Steele's Common Lisp: the Language was published.

If you do some math you'll see that at the time the book was published Lisp had around twenty-five years of real-world use, experimentation, experience, and history to draw upon. Even so, the book alone didn't quite satisfy everyone and in 1986 a committee (X3J13) was formed to produce an ANSI specification for Common Lisp.

While the committee worked on the standardization process, in 1990 the second edition of Common Lisp: the Language was published. This was more comprehensive and contained some of the things the committee was working on (see the comp.lang.lisp FAQ linked above for more on this). At this point the Lisp family of languages had over thirty years of experience and history to draw upon. For comparison: Python (a "modern" language many people think of as also being "kind of old") was released for the first time the following year.

In 1992 the X3J13 committee published the first draft of the new Common Lisp ANSI standard for public review (see Pitman's paper). The draft was approved in 1994 and the approved specification was finally published in 1995.

At this point Lisp was over thirty-five years old. The first version of Ruby was released in December of that year.

That's the end of the history lesson. There has not been another revision of the ANSI specification of Common Lisp. The version published in 1995 is the one that is still used today — if you see something calling itself "an implementation of Common Lisp" today, that is the specification it's referring to.

Consequences

I wanted to give you a quick overview of the history of Common Lisp because I want you to know what you're getting yourself into. I want you to realize that Common Lisp is a stable, large, practical, extensible, ugly language. Understanding these characteristics will make a lot of things make more sense as you learn the language, and I want to talk a little bit more about each of them before I start offering recommendations.

Escaping the Hamster Wheel of Backwards Incompatibility

If you're coming from other languages, you're probably used to things breaking when you "upgrade" your language implementation and/or libraries. If you want to run Ruby code you wrote ten years ago on the latest version of Ruby, it's probably going to take some effort to update it. My current day job is in Scala, and if a library's last activity is more than 2 or 3 years old on Github I just assume it won't work without a significant amount of screwing around on my part. The Hamster Wheel of Backwards Incompatibility we deal with every day is a fact of life in most modern languages, though some are certainly better than others.

If you learn Common Lisp, this is usually not the case. In the next section of this post I'll be recommending a book written in 1990. You can run its code, unchanged, in a Common Lisp implementation released last month. After years of jogging on the Hamster Wheel of Backwards Incompatibility I cannot tell you how much of a *relief* it is to be able to write code and reasonably expect it to still work in twenty years.

Of course, this is only the case for the language itself — if you depend on any libraries there's always the chance they might break when you update them. But I've found the stability of the core language is contagious, and overall the Common Lisp community seems fairly good about maintaining backwards compatibility.

I'll be honest though: there are exceptions. As you learn the language and start using libraries you'll start noticing some library authors who don't bother to document and preserve stable APIs for their libraries, and if staying off the Hamster Wheel is important to you you'll learn to avoid relying on code written by those people as much as possible.

Practicality Begets Purity

Another thing to understand about Common Lisp is that it's a large, practical language. The second edition of Common Lisp: the Language (usually abbreviated as "CLtL2" by Common Lisp programmers) is 971 pages long, not including the preface, references, or index. You can get a surprising amount done by writing pure Common Lisp without much extra support.

When programming applications in Common Lisp people will often depend on a small(ish) number of stable libraries, and library writers often try to minimize dependencies by utilizing as much of the core language as possible. I try to stick to fewer than ten or so dependencies for my applications and no more than two or three for my libraries (preferably zero, if possible), but I'm probably a bit more conservative than most folks. I *really* don't like the Hamster Wheel.

It's also worth noting that since Common Lisp has been around and stable for so long, it has *libraries* older and more stable than many programming languages. For example: Bordeaux Threads (the de-facto threading library for Common Lisp) was first proposed in 2004 and released soon after (2006 at the latest but possibly earlier, it's hard to tell because so many links are dead now), which makes it about fourteen years old. So yes, threading is handled by a library, but I'm not worried about it breaking my code in the next decade or two.

My advice is this: as you learn Common Lisp and look for libraries, try to suppress the voice in the back of your head that says "This project was last updated six years ago? That's probably abandoned and broken." The stability of Common Lisp means that sometimes libraries can just be *done*, not *abandoned*, so don't dismiss them out of hand.

Extensibility

Part of Common Lisp's practicality comes from its extensibility. No one has been clamoring for a new version of the

specification that adds features because Common Lisp's extensibility allows users to add new features to the language as plain old libraries, without having to alter the core language. Macros are what might come to mind when you hear "Lisp extensibility", and of course that's part of it. Macros allow users to write libraries that would need to be core language features in other languages.

Common Lisp doesn't include string interpolation. You want it? No problem, you don't have to wait for Scala 2.10 or Python 3.6, just use a library.

Want to try some nondeterministic programming without any boilerplate? Grab a library.

Pattern matching syntax can make for some really beautiful, readable code. Common Lisp doesn't include it, but of course there's a library.

Enjoying algebraic data types in Haskell or Scala? Here's your library.

All of these libraries rely on macros to make using them feel seamless. Of course you could *do* all of that without macros, but you've have to add a layer of boilerplate to manage evaluation. This:

```
(match foo
  '(list x y z) (lambda (x y z) (+ x y z))
  '(vector x y) (lambda (x y) (- x y)))
```

just doesn't flow off the fingers like:

```
(match foo
  ((list x y z) (+ x y z))
  ((vector x y) (- x y)))
```

No one's up in arms trying to get a new revision of the Common Lisp standard to add pattern matching because you can write it as a library and get 90% or more of what you've get if it were built in. The language gives you enough power to extend it in a way that feels like the extension was there from the beginning.

Having things that are core features in other languages be provided by libraries might seem at odds with the previous section about minimizing dependencies, and to some extent that's true. But I think there's a happy medium where you can write stable libraries in the core language and then depend on a small number of those libraries in your applications to add exactly the features you need for any particular problem.

Power

Macros are one of the things that make Lisp so extensible, because they let you transform arbitrary code into other arbitrary code. This is true for macros in languages like C too, but Common Lisp macros are different because they're *part of the language*.

In C you have a layer of macros on top, written in a preprocessor macro language. The macro layer and the language layer are separate from each other, with the macro layer providing one one extra level of abstractive power (which, don't get me wrong, is certainly useful).

In Common Lisp, you write macros *in Common Lisp itself*. You can then use those macros to write functions, and use those functions to write more macros. Instead of two stratified layers it's a *feedback loop* of abstractive power.

But macros aren't the only thing about Common Lisp that make it so practical and extensible. Something people often don't realize is that while Common Lisp is an extremely high-level language thanks to macros, it also has plenty of low-level facilities as part of the language. It's never going to be as low-level as something like C, Rust, or Forth, but you might be surprised at some of the things that the ANSI spec includes.

Want to see the assembly code a particular function compiles down to? `DISASSEMBLE` it!

Want to stack-allocate something to avoid some garbage collection? X3J13 thought of that.

Need arrays of unboxed floats to ship to a graphics card? The standard allows for that.

Think `GOTO` should be considered helpful, not harmful? Well, okay, we're all adults here. Good luck, try not to shoot your foot off.

Need to do unsigned 8-bit arithmetic in your Game Boy emulator, but would prefer it to compile down to just a

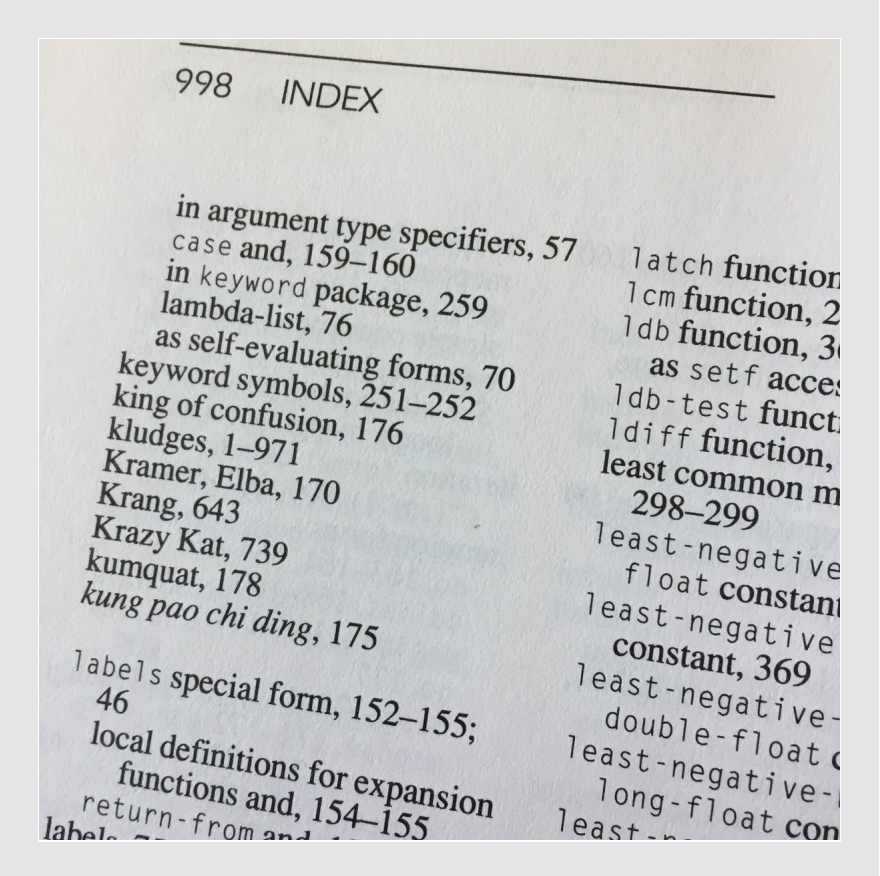
machine instruction or two? It's possible.

Not all Common Lisp implementations actually perform all these optimizations, but the designers of Common Lisp had the foresight to include the language features needed to support them. You can write vanilla Common Lisp as defined by the standard and trust that it will run everywhere, and implementations that *do* support these kinds of things will take advantage of the optimization opportunities.

This combination of supporting extremely high-level programming with macros and a reasonable amount of low-level optimization mean that even though the specification is over twenty years old, it's still a good solid base to build on today. The thirty years of experience and history the designers were drawing from allowed them to create a very practical language that has survived for decades.

Ugliness

It's also important to realize that while Common Lisp might be very practical, the need to accommodate existing users and dialects means that there are plenty of ugly parts. If you buy a paper copy of the second edition of Common Lisp: the Language and look up "kludges" in the index you'll find this:


998 INDEX

in argument type specifiers, 57
case and, 159–160
in keyword package, 259
lambda-list, 76
as self-evaluating forms, 70
keyword symbols, 251–252
king of confusion, 176
kludges, 1–971
Kramer, Elba, 170
Krang, 643
Krazy Kat, 739
kumquat, 178
kung pao chi ding, 175

labels special form, 152–155; 46
local definitions for expansion functions and, 154–155
return-from and

latch function
lcm function, 2
ldb function, 3
as setf acce
ldb-test funct
ldiff function,
least common m 298–299
least-negative float constan
least-negative constant, 369
least-negative- double-float
least-negative- long-float con

Common Lisp is not a beautiful crystal of programming language design. It's a scruffy workshop with a big pegboard wall of tools, a thin layer of sawdust on the floor, a filing cabinet in the office with a couple of drawers that open perpendicular to the rest, and there's a weird looking saw with `RPLACD` written on the side sitting off in a corner where no one's touched it for twenty years.

This historical baggage is a price paid to ensure Common Lisp had a future. It made it practical for people using the old dialects to actually adopt Common Lisp with a reasonable amount of effort. If the designers had tried to make it perfect and beautiful this could have made it too different to port implementations and code to and might have resulted in the language being ignored, instead of being adopted and embraced.

A Road to Learning Common Lisp

If all of this hasn't scared you away from the language, let's talk about how you can learn it in 2018.

If you search around on the internet for Common Lisp tutorials and guides, you're not going to find as much as you might expect. This is because a lot of Common Lisp reference material was created before or during the infancy

of the internet. There are a *lot* of books about Common Lisp out there. Some are better than others. I'll recommend the ones I think are the best, but don't hesitate to browse around and find others.

Get a Lisp

To get started with Common Lisp you'll need to install a Common Lisp implementation. Common Lisp is an ANSI specification, so there are multiple implementations of it, which gives you choices. There are a bunch of options, but I'll make it simple for you:

- If you're using MacOS and want a single GUI app you can download from the App Store, choose ClozureCL (often abbreviated "CCL").
- Otherwise, choose SBCL.

That's Clozure with a Z. Clojure is something entirely different that just happens to have a confusingly similar name.

You might also hear of something called CLISP, which sounds like it might be what you want. It's not. CLISP is just another implementation, but it hasn't had a release in eight years (even though development is still ongoing in its source repos!) and it's not as commonly used as CCL or SBCL, so it'll be harder to find help if you have questions about the installation, etc.

You might also hear about something called Roswell. Don't use Roswell, you don't need it (yet (or at all)).

Just install SBCL or CCL for now, you can explore the other options once you've got your bearings a bit better.

Pick an Editor

You might hear people tell you that you *must* learn Emacs before learning Common Lisp. They're wrong. You can get started learning the language just fine in whatever text editor you're comfortable in.

If you don't have a preference, CCL itself comes bundled with a text editor on MacOS. That one will work just fine to start.

Emacs, Vim, Sublime Text, Atom, whatever, for now it doesn't matter. As long as it can balance parentheses, highlight comments and strings, and autoindent Lisp code that's all you need to start. Worry about shaving the editor yak once you're more comfortable in the language.

Hello, Lisp

To check that you've got everything set up properly, make a `hello.lisp` file with the following contents:

```
(defun hello ()
  (write-line "What is your name?")
  (let ((name (read-line)))
    (format t "Hello, ~A.~%" name)))
```

Don't worry about what this means yet, it's just a check that everything's working properly.

Open an SBCL or CCL REPL and load the file by entering `(load "hello.lisp")`, then call the function and make sure it works. It should look something like this if you picked SBCL:

```
$ sbcl
* (load "hello.lisp")

T
* (hello)
What is your name?
Steve
Hello, Steve.
NIL
*
```

Or if you chose CCL but still want to use the command line, rather than the MacOS app (the command line program might be annoyingly named `ccl64` if you're on a 64-bit system):

```
$ ccl64
Clozure Common Lisp Version ...

? (load "hello.lisp")
#P"/home/sjl/Desktop/hello.lisp"
? (hello)
What is your name?
Steve
Hello, Steve.
NIL
?
```

If your arrow keys and backspace don't work in the REPL, use `rlwrap` to fix that. `rlwrap sbcl` will give you a non-miserable REPL. `rlwrap` is a handy tool to have in your toolbox anyway.

A Gentle Introduction

The best book I've found for getting started in Common Lisp is Common Lisp: A Gentle Introduction to Symbolic Computation. This book really does strive to be gentle. Even if you've programmed before I'd still recommend starting here because it eases you into the language.

The 1990 edition is available free from the site, and there's a 2013 reprint which fixes some minor errors in the 1990 version. If you can afford it I'd recommend buying the 2013 edition, but the 1990 version will also do fine.

Go through the book and *do all the exercises*. This will take a while, and is mainly meant to get you started overcoming some of the main obstacles to being comfortable in Common Lisp, such as:

- How am I ever going to remember all these weird function names?
- Why do people use strings so rarely?
- When do I need the god damn quotation mark?

If you find the book is moving too slow, just skim forward a bit. Skimming is a very useful skill to practice as a programmer. I think it's better for authors to err on the side of explaining too much when writing books and documentation — expert readers should be comfortable skimming if you explain too *much*, but new users will be stuck wallowing in confusion if you're too terse. Creating hours of newbie misery and confusion to save a few flicks of an expert's scroll wheel is a poor tradeoff to make.

You should also join the `#clschool` channel on the Freenode IRC network so you can ask questions if you get stuck. For the most part people there are friendly and helpful, though I'll warn you in advance that there's at least one person who can sometimes be abrasive. There's also a `#clnoobs` channel, but that was mostly abandoned during the latest wave of Freenode spam because no one had ops to help combat the spam.

If IRC isn't your thing there's also a Discord server that some of us hang out in. Join the `#common-lisp` channel there and we'll be happy to help you.

Getting Practical

Once you've finished that book the next one you should attack is Practical Common Lisp. You can get a paper copy if you want, but the full book is available on the site for free.

You can skip the editor/programming environment part because the environment it recommends (Lisp in a Box) is abandoned and no longer works. Just keep using the programming environment you're comfortable with for now.

Unfortunately the book doesn't include exercises. If you *really* want to get the most out of it you can type in all the code as you're reading it and poke at it, but if you've already done the exercises in the previous book it's probably safe to just sit down and read the book carefully. Don't read more than a chapter or two a day. It will take a while for your brain to digest all the information.

Make sure you understand everything as you go through the book. Don't be afraid to ask questions on IRC or Discord (or email me if you want, I don't mind) if something's not clear.

You should also begin to get comfortable looking up things in the Common Lisp language specification itself. It's the ultimate manual for Common Lisp. It can be pretty dense at points, but can answer many questions you might have if you read it slowly and carefully. You can either use the index page to find what you're looking for or just

search on Google for "clhs whatever" (CLHS stands for "Common Lisp HyperSpec", which is the hyperlinked, HTML version of the spec). If you already use the Dash app for MacOS, it has the Common Lisp spec available.

(Some people will tell you to learn the language by just reading the spec. That's ridiculous — it's like trying to learn French by reading a dictionary. It's a useful tool to have, but not the only one you'll need.)

Make Something

Once you've got those two books under your belt and some practice using the spec, it's time to make something without someone holding your hand. It doesn't have to be anything big or special, the goal is to just write some Lisp without having the answer on the next page.

If you need some ideas:

- Do some Project Euler problems.
- Do some Advent of Code exercises.
- Make a stupid Twitter bot.
- Make a personal calendar program that records your appointments, checks the weather forecast the day of, etc.
- Use Sketch to implement the stuff in some Coding Math videos.

It doesn't really matter what you make, just make *something* on your own.

Lisp as a System

At this point it's time to take your Common Lisp skills up a notch. Up until now I've told you to just use any text editor because it's more important to get you some experience with the language, but now it's time to dive into the deep end.

In most languages the development process looks something like this:

1. Edit some code in the project with an editor.
2. Compile the project (some languages skip this step).
3. Run the project (or the tests).
4. Observe the output (in the console, a browser, etc).
5. Go to 1.

This is not how most Common Lisp users interact with the language. In Common Lisp, the development cycle looks more like this:

1. Start a Lisp process.
2. Load the project.
3. Edit some code with your editor.
4. Tell the running process to compile *only the code you edited*.
5. Interact with the changed code in the process via the REPL, an HTTP request, etc.
6. Observe the output (in the console, a browser, etc).
7. Go to 3.

When you embrace the Lisp way of working you'll rarely recompile and reload an entire project. Usually you'll write a function (or a macro, or parameter, or whatever), compile *just that function*, maybe poke at it in the REPL a bit, and then move on to the next function. This has some advantages over the traditional compile-everything-then-run approach.

First: compiling a small chunk of code is fast. I just timed compiling a few of the larger functions in one of my projects and they took around 50-80 microseconds. You don't have to wait for the compiler, so your concentration/thought process never has time to wander.

Another advantage is that when you get back the results of your compilation (and running), any errors or warnings you receive are almost certainly related to the few lines of code you just compiled. If you compile a ten-line function, run it, and get a division by zero error you can immediately focus in on the ten lines you just compiled and think about what changed.

Because the Lisp process is always running, as soon as you compile a function it's ready to be used in the REPL. You can throw some arbitrary data at it and inspect the results to see how it behaves in isolation before you build more things on top of it. This cycle of making a function, compiling it, poking at it to make sure it's working as expected, and moving on happens *constantly*.

In contrast, when working in languages like Scala or Python I almost never find myself writing one single function and compiling or running the project immediately. Spinning up the compiler or running the unit tests takes at *least* a second or two (or sometimes *minutes* in Scala, unfortunately) so to avoid having a constant stream of gaps in my thought I end up writing a bunch of functions at once, and then I run the project or tests once I know they have a chance of working.

But then when I get back an error I have much more surface area to check, because I've added a lot of new code! So now I have to track down a problem that might be in something I wrote four minutes ago, whereas in Lisp I would only have to ever look at the code I wrote in the last few seconds.

I've started using IntelliJ with Scala to help make this a bit less painful. It does help with the compile times because it recompiles things on the fly, but it doesn't solve the rest of the problem. I can write a Scala function in IntelliJ and it will be compiled immediately, but I can't *interact* with it immediately like I can in Common Lisp.

When you work in this style with Common Lisp I think you'll really grow to love it. Writing in other languages will begin to feel like shipping your code off to the DMV and getting it back a week later with a page full of red ink somewhere in the hundred forms you filled out. Writing in Common Lisp feels like interacting with a living, breathing organism, or like teaching things to an eager assistant.

This philosophy of Lisp being not just a programming *language* but a living, breathing programming *system* goes beyond just the short feedback loop and interactive REPL, too.

As an example: imagine you're making a video game and have a bug somewhere in your damage calculation that will occasionally cause a division by zero. Now let's say you're working on the code for a particular quest. You'll start the game, load a save file at the beginning of the quest, and start going through the steps. All of a sudden, in the middle of killing the final monster for the quest, you hit the damage bug! In traditional languages, one of two things might happen:

1. The game crashes, and you get a stack trace and maybe a core dump.
2. You've wrapped a `try` block around the main game loop that logs a stack trace and ignores errors and allows the game to continue.

Case 1 is pretty bad. You've got to try to track down the bug from a snapshot of what things looked like at the time (the stack trace and core dump). And even if you manage to fix it, now you've got to redo all that playing to get back to testing your quest code that you were originally working on.

Case 2 is bad, in a different way. If you just ignore errors all the time, the game might now be in a weird state. You also might lose some critical context that's necessary to debug the problem, unless you're also saving a core dump (but I don't know of many people who save a core dump on every exception).

In Common Lisp you can certainly choose to panic on or ignore errors, but there's a better way to work. When an error is signaled in Common Lisp, it doesn't unwind the stack. The Lisp process will pause execution at that point and open a window in your editor showing you the stack trace. Your warrior's sword is hovering over the monster, waiting for you. At this point you can communicate with the running process at the REPL to see what's going on. You can examine variables in the stack, or even run any arbitrary code you want.

Once you figure out the problem ("Oh, I see, the `calculate-armor-percentage` function returns `0` if a shielding spell ran out during the same frame") you can fix the code, recompile the problematic function, and *restart the execution of that function (or any other one!) in the call stack*! Your warrior's sword lands, and you move back to what you were doing before.

You don't have to track down the bug from just a stack trace, like a detective trying to piece together what happened by the blood stains on the wall. You can examine the crime *as it's happening* and intervene to save the victim. It's like if you could run your code in a debugger with a breakpoint at every single line that only activates if something goes wrong!

Maybe you don't make video games, sure, but this process can be useful in all kinds of contexts. Maybe you're writing a web app that talks to an API somewhere, and are debugging a request that fails between two calls to the

API, e.g. between "create widget `foo`" and "add `foo` to widget list `bar`". Instead of just aborting the request, logging a stack trace, and now leaving things in a possibly weird state (`foo` having been created without being in the expected `bar` list), you can fix the problem and allow the request to finish properly.

Of course this won't always work. If you've got a big function that does some side effects and then crashes, restarting execution of the function would make the side effects happen again. But if you divide up your functions well (one function to a function!) this case is pretty rare. And even when it does happen, it just means you're back in the same situation you're in *by default* with other languages!

Support for this style of interactive development doesn't just come from some fancy editor plugins — it's baked into the bones of the language. For example: the standard specifies a method named `update-instance-for-redefined-class` that lets you customize what happens to objects when their class is redefined! This isn't something you'll use all the time, but something like Sketch (a Common Lisp equivalent of Java's Processing library) uses it to automatically update the running sketch when you redefine its class. Dynamically updating running code in a safe, consistent way doesn't require any dark magic in Common Lisp because it's the expected, usual way to work.

So how do you actually *get* this wonderful interactive experience? The bad news is that you're going to need to shave the editor yak. You really only have two choices here:

- Emacs with SLIME or Sly.
- Vim (or Neovim) with Vlime or Slimv.

I wish this weren't the case, but those are really only the realistic options today (aside from the editing environments for the (expensive) commercial Lisps).

If you're like me and already have Vim burned too deeply into your fingers to ever get it out, I'd recommend Vim with Vlime. It will give you 80% of the experience you'll get with Emacs.

Otherwise go with Emacs. You might want to look into Portacle, which bundles Emacs and SLIME and a bunch of other things together, or you might want to have a go at setting up Emacs and SLIME or Sly yourself. I can't really give you much advice on the Emacs side of things because I haven't had much experience with it, so you'll need to do a bit of research here.

Whatever you choose, spend some time setting up your editor and environment of choice. This will be a lot of fiddly metawork, but will pay off handsomely as you continue working in Lisp.

On a side note: if anyone is interested in making a Common Lisp LSP language server, I think it would be a hugely useful contribution to the community. Having an LSP server would mean you could get a much nicer programming experience in many editors out of the box, which would help new people quite a lot.

I think you could piggyback on top of Swank to do a lot of the language-side stuff, and it would mostly be a matter of implementing the LSP interface. If this sounds interesting to you, please let me know — I'd be willing to help. I've done some work at my day job making a Scala LSP language server that uses IntelliJ as a backend, so I have at least some idea of how that sausage gets made. I just don't have the time or motivation to do an entire LSP server for Common Lisp all by myself.

Learning Paradigms

At this point you should have a pretty good handle on the basics of Common Lisp, and have set up one of the more powerful development environments. Your next goals should be to learn how to write idiomatic Common Lisp and to get some practice using your fancy new environment.

I think the perfect book for both of these is Paradigms of Artificial Intelligence Programming, often abbreviated as PAIP. The book was recently made available for free as a PDF, or you can buy a used paper copy if you prefer.

This book was written in 1992 so it's not about the hyped up AI fields you've been hearing about in the news like machine learning — instead it's a tour of Good Old-Fashioned AI. Even if you're not particularly interested in this kind of AI, the book is a great example of how to write Common Lisp code.

One thing I really love about this book is that almost all the functions in it have docstrings. If you look at most other programming books they omit the documentation strings, presumably for space reasons and because they feel the surrounding text is documentation enough. But writing helpful docstrings is an art in and of itself, and I think books that omit them train readers that "good code omits docstrings", which is a bad habit to get into.

The book contains *plenty* of exercises, conveniently categorized by how difficult or involved they are:

- `S` for "seconds".
- `M` for "minutes".
- `H` for "hours".
- `D` for "days".

This is a very good idea which more books should steal. Do all of the `S` and `M` exercises, and try your hand at at least a few of the `H` ones. If a `D` sounds particularly interesting don't be afraid to spend some time on it — really digging into a problem is exactly what you need at this point in your Lisp journey.

Switch Things Up

Now that you're comfortable in Common Lisp and your programming environment, it's time to push yourself out of your comfort zone again. At the beginning I had you choose either SBCL or CCL. Now I want you to install whichever one you didn't originally choose and make sure all the code you've written so far runs in it.

This may seem a bit like running in place, but making sure your code runs in more than one implementation will keep you honest. It will force you to write portable code that doesn't rely on anything implementation-specific that might change in the next decade or two. And you might even discover that you like this other implementation better than the original — maybe CCL's super-fast compile times make you smile, or SBCL's strong type inference catches more of your bugs.

Go through all the code you've written so far and make sure it all runs in the new implementation. You might also want to take this opportunity to refactor or rewrite some of it — you've learned a lot since you first started, so your earliest Common Lisp code will probably look pretty rough to you now.

Recipes for Success

The final technical book I'll recommend to every aspiring Lisp programmer is Common Lisp Recipes, sometimes abbreviated as CLR. Unlike most of the other books I've recommended so far this one is relatively recent: it was published in 2015. It's not free, but I think it's well worth the money it costs.

The book is written by the author of several very heavily used Common Lisp libraries. It's a bit of a grab bag of topics (which is why I think you need a decent amount of Lisp under your belt before you tackle it) but it's a very well-written grab bag that will teach you a lot of things you won't find in other books.

Final Patterns

If you've gotten this far you're pretty invested in Common Lisp, and I want to recommend one not-strictly-technical book that I think you'll really enjoy: Patterns of Software by Richard Gabriel. It's available as a PDF on the author's site, and you can still find used print copies online if you prefer.

This is *not* the "Gang of Four"/"Design Patterns" book that you might have already read or heard about, but is a set of essays on a variety of loosely-related topics. It's the best book I've read so far this year. I don't want to spoil anything in it for you, so I'll just say that I think you'll find it well worth your time.

Where to Go From Here

If you made it through all the books and activities in the previous section: congratulations, you're off to a great start! Now that you've got a decent handle on the core language you can explore in many different directions, depending on your interests.

Macros

If you want to learn the secrets of macros, you'll probably want to read and work through On Lisp and Let Over Lambda (in that order).

I'll say that you should take both books (*especially* the latter) with a large grain of salt. A lot of Common Lisp users don't agree with all of the arguments and style in these books, but I think they can still provide plenty of value if you read them with a critical mind.

Object-Oriented Programming with CLOS

Common Lisp has some very sophisticated support for Object-Oriented Programming through CLOS. If you're like me and have bad memories of OOP from working in a Java cube farm, I'd urge you to give CLOS a fair chance to change your mind.

Start with Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS. It's a wonderfully-written, short and to-the-point book that will give you a good overview of how CLOS is intended to be used.

If you really want to bend your mind, try The Art of the Metaobject Protocol (usually abbreviated as AMOP). This book will probably take you a couple of tries to get through. Read it until you hit a mental wall, go work on other things for a couple of months, and come back and try again. Repeat that process as many times as necessary.

Low-Level Programming

Low-level programming can mean a lot of different things, so I'll just mention one possibility here.

If you're interested in writing emulators for old computers, I wrote a series of posts on making a CHIP-8 emulator in Common Lisp. cl-6502 is an emulator for the processor used in the NES (and lots of other things) and has a really nice literate programming version that's wonderful to read through.

Web Development

Unfortunately I don't have too many suggestions for web development in Common Lisp. I've made a conscious effort to avoid web development in the past five or so years, because it seems like the Hamster Wheel of Backwards Incompatibility has become more of a Hamster Centrifuge in that field.

There is a `#lispweb` channel on Freenode and a `#webdev` channel in the Lisp Discord, so if you have questions you could start by asking there. Those channels are a bit less populated than the other Lisp channels, so don't expect an answer immediately.

Game Development

Common Lisp has a small but enthusiastic community of people who like making games. There's a `#lispgames` channel on Freenode and a `#gamedev` channel on the Lisp Discord that you should join if you're interested.

Land of Lisp is a fun book to go through. The coding style in the book has some… "eccentricities", which is why I don't recommend it as a first book on Lisp (e.g. using `ash` instead of `truncate` or `floor` for integer division), but if you know the language and just want to get started making some simple games I think you'll enjoy working through it.

If you want an excuse to make a game in Lisp in a week, the Lisp Game Jam is something you can join. It's usually held once or twice each year, so you'll have to search around (or ask in `#lispgames`) to find out when the next one is.

Lisp doesn't have any engine as full-featured as Unity, but several people are currently working on making 3D game engines. Ask around to see what people are using these days. Unfortunately a 3D game engine will generally need to interface with the OS to render images and produce audio, and so can't be written in pure Common Lisp. This means that some running on the Hamster Wheel of Backwards Incompatibility will be necessary to keep up with OS changes (e.g. Apple deprecating OpenGL).

If you're interested in old-school ASCII/tile-based games, I've personally done some work with using ncurses and bearlibterminal in Common Lisp. There's something really fun about making a game people can play over telnet! Feel free to get in touch with me if you're interested in that kind of stuff and want to know more.

Window Management

If you're running Linux and like tinkering with your desktop environment, StumpWM is an X window manager written in Common Lisp. I've just recently switched back to Linux so I've only been using it for about two months, but it's really pleasant to be able to customize my working environment with Common Lisp.

StumpWM has a small but friendly community — if you're looking for a non-trivial open source Common Lisp project to contribute to, StumpWM would be a great choice.

Unit Testing

If you're coming from a modern language, especially one with a lot of test-driven development advocates, you might be surprised at the lack of an emphasis on unit testing in Common Lisp. I think one reason for this is that in some languages a unit test is the simplest way to actually *run* a function, but Lisp's interactive style of development gives you an even easier alternative: just *run the function* in the REPL!

Despite the lack of heavy unit testing in the community, there are almost as many unit testing *frameworks* as there are Common Lisp programmers! This is probably because making a unit testing framework is so easy with a few macros. I love 1am, but there are *plenty* more to choose from.

Whichever one you choose, please make sure to be a good citizen and create a separate ASDF system for your unit tests, so people can use your library without having to load Yet Another Testing Framework.

More Implementations

I had you use SBCL and CCL because those are the most popular free Common Lisp implementations today, but they aren't the only actively-developed ones out there. There's plenty of others you might want to explore:

- ABCL runs on the JVM.
- ECL can be embedded in a C program, and can translate Common Lisp code to C code.
- CLASP is still under development, but is an implementation designed to be easy to interoperate with C++.
- Lispworks and Allegro CL are commercial implementations with a lot of extra features and support, but are not free.

(I omitted CLISP because I'm mad at them for choosing a name that confuses the heck out of new people. Hey, I warned you this post would contain Opinions™.)

I tend to use SBCL for my own projects, but I make sure the units tests for all my libraries run in SBCL, CCL, ABCL, and ECL. This keeps me honest and gives me a reasonable degree of confidence that I'm writing portable code.

Modern Common Lisp

Common Lisp is old and stable, but that doesn't mean it's stagnant. The language gives you plenty of power to build on, and before I wrap this up I want go over a couple of recent developments in the Common Lisp world that the older books you've been learning from don't talk about. I also want to clarify some things that often trip up new people.

Structure

Common Lisp's terminology for various parts of projects is often confusing to new people because it's old and uses a lot of words that we use now (like "package") to mean subtly different things than people mean today. Things get easier once you internalize what Common Lisp means by the terms.

(Side note: I posted a quick-and-dirty version of this section as a comment on Lobste.rs while I was waiting for a plane — this section of the post is an expanded version of that comment.)

Packages

We often see questions in IRC and Discord that look something like: "How do I export a class from a package"? Questions worded like this are a sign of a very common misunderstanding about what packages in Common Lisp *actually are*.

A package in Common Lisp is a container for symbols. That's it. They're a way to group related names (symbols) together so you don't have to do the miserable prefixing of every name with `mylibrary-...` like you need to do in Emacs Lisp or C to avoid name clashes.

You don't export a class from a package, you export a *symbol*. You don't import a function, you import the *symbol* it's attached to. This sounds pedantic, but is important to keep clear in your head as you start using the package system. If you're not clear on what exactly a symbol *is*, I wrote a separate post just about symbols which you might find helpful.

Another major tripping point for new people is the relationship between packages and files. Or, rather: the completely *lack* of any relationship in Common Lisp.

In many languages like Python, Java, or Clojure, a file's package and its location on the hard drive are tied together. For example: when you say `import foo.bar.baz` in Python, Python will look for a `baz.py` file inside the `foo/bar/` directory (it's a little more complicated than this, but that doesn't matter for this example).

In Common Lisp, this is not the case. **Files and packages are completely unrelated in Common Lisp.** You can have many files that all work in the same package, or one file that switches between many packages, or even create or modify packages at runtime.

This gives you the flexibility to work however you want. For example: in my procedural art library Flax most of the packages are each used in one specific file, much like you would do in modern languages. But the `flax.drawing` package contains not only a drawing protocol but also several implementations of that protocol (PNG, SVG, etc), and so I split the code into a series of separate files, each one dealing with how to draw a single format (plus one for the protocol itself).

I could have created separate packages for each implementation and set up the imports/exports between them, but I didn't feel like the extra boilerplate was worth it. Common Lisp is flexible enough to let you make such choices.

So if files and packages aren't related, the next question is: how does Common Lisp know where to *find* anything on disk when it comes time to load the code?

Systems

A system in Common Lisp is a collection of serveral things:

- Some code.
- A description of how to load that code.
- A list of other systems this system depends on, which need to be loaded prior to loading this one.
- Some metadata like author, license, version, homepage, etc.

The Common Lisp language itself has no knowledge of systems. If you look at section 11.9 of CLtL2 you'll see that it was imagined that each author would write their own custom file to load their code. But since Common Lisp gives you the power to abstract almost anything, people eventually abstracted the process of loading Common Lisp code.

ASDF is a Common Lisp library bundled with most modern implementations which handles defining and loading systems. The name ASDF stands for "Another System Definition Facility", so as you might guess there have been several other such libraries. ASDF is the one everyone uses today.

ASDF standardizes the process of defining a system into something like this:

- The system definition(s) for a project called `foo` would be in a file named `foo.asd`.
- Each system is defined with a `(defsystem ...)` form inside this file.

We'll talk more about what a "project" is shortly. Note the extension of the file is `asd`, not `asdf`, which is a little confusing, but was probably chosen to work in environments with three-letter-extension limits.

The ASDF manual is the definitive resource for the syntax and semantics of `defsystem`, but can be a little heavy to read if you're just getting started. Another way to get started is to read some `.asd` files of some small-to-medium sized open source projects and see how they handle things.

Systems and packages are orthogonal in Common Lisp. Some systems (like small libraries) will define exactly one package. Some systems will define multiple packages. Rarely a system might not define any new packages, but will use or add to an existing one.

For example:

- My directed graph library cl-digraph contains a system called `cl-digraph`.
- That system has a description of how to load the code, which lives in the `cl-digraph.asd` file.
- One of the files specified for loading is `package.lisp`, which creates a package called `digraph`.

Even though ASDF standardizes some aspects of system definition, it still gives you plenty of flexibility. As you

read projects by different authors you'll encounter different ways of organizing systems — this can be a little overwhelming at first, but it means you can organize a system in the way that works *best for that system*, which is really nice once you've got some experience under your belt.

One example of this is how people define packages for their systems. There are a couple of common ways to do this you'll see in the wild:

- A single `package.lisp` file which contains all the definitions for all the packages in the project, and gets loaded before all other files. This is the strategy I usually prefer.
- Each file defines its package at the top of the file, much like you would in Clojure or other modern languages. Care is taken in the system definition to load the files in the correct order so that each package is defined before it is ever used.

To review: a system is a collection of code and a description of how to load it, a list of its dependencies, and some metadata. Now let's move up one level higher to the final layer of structure you need to know about.

Projects

A project in Common Lisp is not an official term defined anywhere that I know of, but is a word that's generally used to mean something like a library, a framework, an application, etc.

A project will usually define at least one system, because systems are where you describe how to load the code, and if a project didn't define a system how would you know how to load its code? My string-wrapping library Bobbin is a project that defines *two* systems:

- The `bobbin` system contains the actual data structure and API. It has no dependencies.
- The `bobbin/test` system contains the unit tests. It depends on the `bobbin` system (because that's the code it's going to test) and the `1am` system (a unit test framework). I made this a separate system because it allows users to load the main code without also having to load the unit testing framework if they're not going to be running the tests.

Both of these systems are defined in the `bobbin.asd` file. ASDF treats systems with a forward slash in their name specially and knows to look for them in the `asd` file named with the text before the slash.

We saw how Common Lisp has no concept of a system — that concept comes from ASDF. Similarly, ASDF has no concept of the internet or of reaching out to somewhere to download things. ASDF assumes you have somehow acquired the systems you want to load and stored them on your hard drive, perhaps by sending a check to an address and receiving a copy of the code on floppy disk, as many of my old Lisp books offer in their final pages.

Quicklisp is another library that works on top of ASDF to provide the "download projects from the internet automatically if necessary" functionality that people expect in the modern world. So when you say `(ql:quickload :bobbin)` you're asking Quicklisp to download Bobbin (and any dependencies) if necessary, and then hand it off to ASDF to actually load the code of the `bobbin` system.

Unlike ASDF, Quicklisp is relatively new in the Common Lisp world (it's only about eight years old) and so is not bundled with any modern Lisp implementations that I know of, which is why you need to install it separately.

Recap

Here's a quick recap of the different layers of project structure you'll encounter in Common Lisp. Jot these down on a post it note you can refer to as you're learning.

- **Files** are files on your hard drive.
- **Packages** are containers of symbols. They are orthogonal to files.
- **Systems** are collections of code, instructions on how to load that code, dependency lists, and metadata. They are orthogonal to packages.
- **Projects** are high-level collections of… "stuff" such as code, documentation, maybe some image assets, etc. They are (mostly) orthogonal to systems (are you seeing a trend here?).
- Common Lisp itself knows about files and packages.
- ASDF adds systems.
- Quicklisp adds the internet.

Common Libraries

Common Lisp doesn't have as *large* of a community as some newer languages, but it still has a lot of libraries because it's had a community for a longer time. The stability of the core language means that many libraries written in portable Common Lisp ten or fifteen years ago can still run just fine today.

In this final section I'll give you a quick overview of some of the more popular libraries you might run into as you learn the language. You don't have to use all of them, but it's helpful to have some idea of what's available.

Alexandria

Alexandria is one of the most popular Common Lisp libraries (the name is a pun on the Library of Alexandria), and it's a collection of all kinds of useful little utility functions like `read-file-into-byte-vector` and `map-permutations`.

There are a *lot* of utility libraries for Common Lisp around — one rite of passage is building up your own personal utility library over time — but Alexandria is the most popular one. Most projects with any dependencies at all will eventually end up with Alexandria in the dependency graph somewhere.

Bordeaux Threads

Bordeaux Threads was mentioned earlier. Threads aren't part of the Common Lisp standard, but most implementations provide their own custom interface for working with them. Bordeaux Threads wraps all these implementation-specific interfaces and provides an API so you can write threaded code that will work portably.

If you're looking for something like Java's `new Thread(() -> foo()).start()`, this is what you want.

CFFI

CFFI is a foreign-function interface library that lets you load C libraries (e.g. `foo.dylib` or `foo.so`) and call the functions in them. It works by wrapping implementation-specific interfaces, because this isn't part of the Common Lisp standard.

Unfortunately it has the same name as Python's FFI library, so if you're searching for documentation make sure you're looking at the right version.

CL-PPCRE

CL-PPCRE is an implementation of Perl-compatible regular expressions. If you're looking to use regular expressions in Common Lisp, this is what you want.

Drakma

Drakma is an HTTP client. If you need to make an HTTP request, this is what you want. There are other HTTP clients around, but Drakma is commonly used and is fine for almost anything you might need.

Iterate

Iterate is a replacement for the `loop` macro. It works similarly, but has a more Lispy syntax and a well-defined API for extending it with new iteration constructs. I really like it myself, but beware: if you get used to `iterate` going back to vanilla `loop` will feel painful.

local-time

local-time is a library for working with time and dates in Common Lisp. The standard has some basic support for times built in, but if you want to do much calculation with times (including timezones) this is probably what you want. If you're looking for something like Joda Time in Common Lisp, this is as close as you're going to get.

lparallel

lparallel is a library that builds on top of Bordeaux Threads to make common parallel processing operations much easier. Think of it as GNU Parallel for Lisp, with a few extra features (e.g. channels and tasks).

For example: if you've got a big vector you're mapping over with `(map 'vector #'work some-vector)` you can split it into chunks and run in multiple threads by changing it to `(lparallel:pmap 'vector #'work some-vector)`.

Named Readtables

Named readtables is a library that adds namespaces for readtables.

One painful part of the standard is that reader macros are added and removed to the global readtable on the fly, so if you load multiple systems that define the same reader macros things can get messy. Named readtables adds some much-needed hygiene to that process. If you're working with reader macros at all you absolutely want to use this.

Roswell

Roswell is a couple of things rolled into one. It's a C program that handles installing and running multiple different Common Lisp implementations (kind of like NVM or rvm), and it also provides a unified way to write small shell scripts in Common Lisp and compile them into binaries.

I used Roswell for a little over a year, but I eventually stopped and now I don't think it's worth the trouble, for a couple of reasons.

First: if you write portable code you generally don't need to worry running a particular version of an implementation, because Common Lisp is so stable. I usually just install the latest version of each implementation I use with a package manager or by building from source.

Second: after using it for a while I found that Roswell was always very brittle to upgrade, and whenever things broke it would spew an almost JVM-sized stack trace without a decent error message.

For me, the negatives outweighed the positives. I'd recommend simply using the latest version of the implementations you care about and writing portable code. For the compiling-into-binaries functionality I'd recommend using your implementation's built-in support for this, or using UIOP's wrapper around that, or using a separate library like Deploy.

Of course your mileage might vary. If you find yourself *really* needing to run specific versions of specific Common Lisp implementations in rapid succession, you should look into Roswell.

SERIES

SERIES was almost included in Common Lisp (it's in Appendix A of CLtL2), but didn't quite make it. It's a library for writing functional code that looks like the traditional `map` and `filter` and `reduce` operations but which compiles down to efficient loops.

If you're looking for Clojure's transducers in Common Lisp, this is what you want.

st-json

JSON support in Common Lisp is a god damn mess. There are an absurd number of JSON libraries and I don't really *like* any of them.

For me, the most important quality I need in a JSON library is an unambiguous, one-to-one mapping of types. For example: some libraries will deserialize JSON arrays as Lisp lists, and JSON `true`/`false` as `t`/`nil`. But this means `[]` and `false` both deserialize to `nil`, so you can't reliably round trip anything!

I've settled on using st-json and wrapping it up to be a little more ergonomic with some glue code. It's not the fastest solution out there, but it works for my needs. There are plenty of other options out there, so if you have different needs than me you should look into them.

usocket

usocket is a library for networking sockets. Sockets and networking aren't part of the Common Lisp standard, but most implementations provide a custom interface for working with them. usocket wraps the implementation-specific interfaces and provides an API so you can write networking code portably.

If you want to make Lisp listen on a port and read streams of bytes from clients, or want to connect to a port and send raw bytes to it, this is what you want.

Good Luck!

I hope this whirlwind tour was useful. Common Lisp is an old, deep language. It's not something you can learn in a month, but if you're willing to spend the time it will reward careful study.

Feel free to email me or pop into IRC or Discord if you have questions.

Good luck!