1. Introduction: Why Lisp?

If you think the greatest pleasure in programming comes from getting a lot done with code that simply and clearly expresses your intention, then programming in Common Lisp is likely to be about the most fun you can have with a computer. You'll get more done, faster, using it than you would using pretty much any other language.

That's a bold claim. Can I justify it? Not in a just a few pages in this chapter--you're going to have to learn some Lisp and see for yourself--thus the rest of this book. For now, let me start with some anecdotal evidence, the story of my own road to Lisp. Then, in the next section, I'll explain the payoff I think you'll get from learning Common Lisp.

I'm one of what must be a fairly small number of second-generation Lisp hackers. My father got his start in computers writing an operating system in assembly for the machine he used to gather data for his doctoral dissertation in physics. After running computer systems at various physics labs, by the 1980s he had left physics altogether and was working at a large pharmaceutical company. That company had a project under way to develop software to model production processes in its chemical plants--if you increase the size of this vessel, how does it affect annual production? The original team, writing in FORTRAN, had burned through half the money and almost all the time allotted to the project with nothing to show for their efforts. This being the 1980s and the middle of the artificial intelligence (AI) boom, Lisp was in the air. So my dad--at that point not a Lisper--went to Carnegie Mellon University (CMU) to talk to some of the folks working on what was to become Common Lisp about whether Lisp might be a good language for this project.

The CMU folks showed him some demos of stuff they were working on, and he was convinced. He in turn convinced his bosses to let his team take over the failing project and do it in Lisp. A year later, and using only what was left of the original budget, his team delivered a working application with features that the original team had given up any hope of delivering. My dad credits his team's success to their decision to use Lisp.

Now, that's just one anecdote. And maybe my dad is wrong about why they succeeded. Or maybe Lisp was better only in comparison to other languages of the day. These days we have lots of fancy new languages, many of which have incorporated features from Lisp. Am I really saying Lisp can offer you the same benefits today as it offered my dad in the 1980s? Read on.

Despite my father's best efforts, I didn't learn any Lisp in high school. After a college career that didn't involve much programming in any language, I was seduced by the Web and back into

computers. I worked first in Perl, learning enough to be dangerous while building an online discussion forum for *Mother Jones* magazine's Web site and then moving to a Web shop, Organic Online, where I worked on big--for the time--Web sites such as the one Nike put up during the 1996 Olympics. Later I moved onto Java as an early developer at WebLogic, now part of BEA. After WebLogic, I joined another startup where I was the lead programmer on a team building a transactional messaging system in Java. Along the way, my general interest in programming languages led me to explore such mainstream languages as C, C++, and Python, as well as less well-known ones such as Smalltalk, Eiffel, and Beta.

So I knew two languages inside and out and was familiar with another half dozen. Eventually, however, I realized my interest in programming languages was really rooted in the idea planted by my father's tales of Lisp--that different languages really are different, and that, despite the formal Turing equivalence of all programming languages, you really can get more done more quickly in some languages than others and have more fun doing it. Yet, ironically, I had never spent that much time with Lisp itself. So, I started doing some Lisp hacking in my free time. And whenever I did, it was exhilarating how quickly I was able to go from idea to working code.

For example, one vacation, having a week or so to hack Lisp, I decided to try writing a version of a program--a system for breeding genetic algorithms to play the game of Go--that I had written early in my career as a Java programmer. Even handicapped by my then rudimentary knowledge of Common Lisp and having to look up even basic functions, it still felt more productive than it would have been to rewrite the same program in Java, even with several extra years of Java experience acquired since writing the first version.

A similar experiment led to the library I'll discuss in Chapter 24. Early in my time at WebLogic I had written a library, in Java, for taking apart Java class files. It worked, but the code was a bit of a mess and hard to modify or extend. I had tried several times, over the years, to rewrite that library, thinking that with my ever-improving Java chops I'd find some way to do it that didn't bog down in piles of duplicated code. I never found a way. But when I tried to do it in Common Lisp, it took me only two days, and I ended up not only with a Java class file parser but with a general-purpose library for taking apart any kind of binary file. You'll see how that library works in Chapter 24 and use it in Chapter 25 to write a parser for the ID3 tags embedded in MP3 files.

Why Lisp?

It's hard, in only a few pages of an introductory chapter, to explain why users of a language like it, and it's even harder to make the case for why you should invest your time in learning a certain language. Personal history only gets us so far. Perhaps I like Lisp because of some quirk in the way my brain is wired. It could even be genetic, since my dad has it too. So before you dive into learning Lisp, it's reasonable to want to know what the payoff is going to be.

For some languages, the payoff is relatively obvious. For instance, if you want to write low-level code on Unix, you should learn C. Or if you want to write certain kinds of cross-platform

applications, you should learn Java. And any of a number companies still use a lot of C++, so if you want to get a job at one of them, you should learn C++.

For most languages, however, the payoff isn't so easily categorized; it has to do with subjective criteria such as how it feels to use the language. Perl advocates like to say that Perl "makes easy things easy and hard things possible" and revel in the fact that, as the Perl motto has it, "There's more than one way to do it."[1] Python's fans, on the other hand, think Python is clean and simple and think Python code is easier to understand because, as *their* motto says, "There's only one way to do it."

So, why Common Lisp? There's no immediately obvious payoff for adopting Common Lisp the way there is for C, Java, and C++ (unless, of course, you happen to own a Lisp Machine). The benefits of using Lisp have much more to do with the experience of using it. I'll spend the rest of this book showing you the specific features of Common Lisp and how to use them so you can see for yourself what it's like. For now I'll try to give you a sense of Lisp's philosophy.

The nearest thing Common Lisp has to a motto is the koan-like description, "the programmable programming language." While cryptic, that description gets at the root of the biggest advantage Common Lisp still has over other languages. More than any other language, Common Lisp follows the philosophy that what's good for the language's designer is good for the language's users. Thus, when you're programming in Common Lisp, you almost never find yourself wishing the language supported some feature that would make your program easier to write, because, as you'll see throughout this book, you can just add the feature yourself.

Consequently, a Common Lisp program tends to provide a much clearer mapping between your ideas about how the program works and the code you actually write. Your ideas aren't obscured by boilerplate code and endlessly repeated idioms. This makes your code easier to maintain because you don't have to wade through reams of code every time you need to make a change. Even systemic changes to a program's behavior can often be achieved with relatively small changes to the actual code. This also means you'll develop code more quickly; there's less code to write, and you don't waste time thrashing around trying to find a clean way to express yourself within the limitations of the language.[2]

Common Lisp is also an excellent language for exploratory programming--if you don't know exactly how your program is going to work when you first sit down to write it, Common Lisp provides several features to help you develop your code incrementally and interactively.

For starters, the interactive read-eval-print loop, which I'll introduce in the next chapter, lets you continually interact with your program as you develop it. Write a new function. Test it. Change it. Try a different approach. You never have to stop for a lengthy compilation cycle.[3]

Other features that support a flowing, interactive programming style are Lisp's dynamic typing and the Common Lisp condition system. Because of the former, you spend less time convincing

the compiler you should be allowed to run your code and more time actually running it and working on it,[4] and the latter lets you develop even your error handling code interactively.

Another consequence of being "a programmable programming language" is that Common Lisp, in addition to incorporating small changes that make particular programs easier to write, can easily adopt big new ideas about how programming languages should work. For instance, the original implementation of the Common Lisp Object System (CLOS), Common Lisp's powerful object system, was as a library written in portable Common Lisp. This allowed Lisp programmers to gain actual experience with the facilities it provided before it was officially incorporated into the language.

Whatever new paradigm comes down the pike next, it's extremely likely that Common Lisp will be able to absorb it without requiring any changes to the core language. For example, a Lisper has recently written a library, AspectL, that adds support for aspect-oriented programming (AOP) to Common Lisp.[5] If AOP turns out to be the next big thing, Common Lisp will be able to support it without any changes to the base language and without extra preprocessors and extra compilers.[6]

Where It Began

Common Lisp is the modern descendant of the Lisp language first conceived by John McCarthy in 1956. Lisp circa 1956 was designed for "symbolic data processing"[7] and derived its name from one of the things it was quite good at: LISt Processing. We've come a long way since then: Common Lisp sports as fine an array of modern data types as you can ask for: a condition system that, as you'll see in Chapter 19, provides a whole level of flexibility missing from the exception systems of languages such as Java, Python, and C++; powerful facilities for doing object-oriented programming; and several language facilities that just don't exist in other programming languages. How is this possible? What on Earth would provoke the evolution of such a well-equipped language?

Well, McCarthy was (and still is) an artificial intelligence (AI) researcher, and many of the features he built into his initial version of the language made it an excellent language for AI programming. During the AI boom of the 1980s, Lisp remained a favorite tool for programmers writing software to solve hard problems such as automated theorem proving, planning and scheduling, and computer vision. These were problems that required a lot of hard-to-write software; to make a dent in them, AI programmers needed a powerful language, and they grew Lisp into the language they needed. And the Cold War helped--as the Pentagon poured money into the Defense Advanced Research Projects Agency (DARPA), a lot of it went to folks working on problems such as large-scale battlefield simulations, automated planning, and natural language interfaces. These folks also used Lisp and continued pushing it to do what they needed.

The same forces that drove Lisp's feature evolution also pushed the envelope along other dimensions--big AI problems eat up a lot of computing resources however you code them, and if you run Moore's law in reverse for 20 years, you can imagine how scarce computing resources were on circa-80s hardware. The Lisp guys had to find all kinds of ways to squeeze performance out of their implementations. Modern Common Lisp implementations are the heirs to those early efforts and often include quite sophisticated, native machine code-generating compilers. While today, thanks to Moore's law, it's possible to get usable performance from a purely interpreted language, that's no longer an issue for Common Lisp. As I'll show in Chapter 32, with proper (optional) declarations, a good Lisp compiler can generate machine code quite similar to what might be generated by a C compiler.

The 1980s were also the era of the Lisp Machines, with several companies, most famously Symbolics, producing computers that ran Lisp natively from the chips up. Thus, Lisp became a systems programming language, used for writing the operating system, editors, compilers, and pretty much everything else that ran on the Lisp Machines.

In fact, by the early 1980s, with various AI labs and the Lisp machine vendors all providing their own Lisp implementations, there was such a proliferation of Lisp systems and dialects that the folks at DARPA began to express concern about the Lisp community splintering. To address this concern, a grassroots group of Lisp hackers got together in 1981 and began the process of standardizing a new language called Common Lisp that combined the best features from the existing Lisp dialects. Their work was documented in the book *Common Lisp the Language* by Guy Steele (Digital Press, 1984)--*CLtL* to the Lisp-cognoscenti.

By 1986 the first Common Lisp implementations were available, and the writing was on the wall for the dialects it was intended to replace. In 1996, the American National Standards Institute (ANSI) released a standard for Common Lisp that built on and extended the language specified in CLtL, adding some major new features such as the CLOS and the condition system. And even that wasn't the last word: like CLtL before it, the ANSI standard intentionally leaves room for implementers to experiment with the best way to do things: a full Lisp implementation provides a rich runtime environment with access to GUI widgets, multiple threads of control, TCP/IP sockets, and more. These days Common Lisp is evolving much like other open-source languages--the folks who use it write the libraries they need and often make them available to others. In the last few years, in particular, there has been a spurt of activity in open-source Lisp libraries.

So, on one hand, Lisp is one of computer science's "classical" languages, based on ideas that have stood the test of time.[8] On the other, it's a thoroughly modern, general-purpose language whose design reflects a deeply pragmatic approach to solving real problems as efficiently and robustly as possible. The only downside of Lisp's "classical" heritage is that lots of folks are still walking around with ideas about Lisp based on some particular flavor of Lisp they were exposed to at some particular time in the nearly half century since McCarthy invented Lisp. If someone

tells you Lisp is only interpreted, that it's slow, or that you have to use recursion for everything, ask them what dialect of Lisp they're talking about and whether people were wearing bell-bottoms when they learned it.[9]

But I learned Lisp Once, And IT Wasn't Like what you're describing

If you've used Lisp in the past, you may have ideas about what "Lisp" is that have little to do with Common Lisp. While Common Lisp supplanted most of the dialects it's descended from, it isn't the only remaining Lisp dialect, and depending on where and when you were exposed to Lisp, you may very well have learned one of these other dialects.

Other than Common Lisp, the one general-purpose Lisp dialect that still has an active user community is Scheme. Common Lisp borrowed a few important features from Scheme but never intended to replace it.

Originally designed at M.I.T., where it was quickly put to use as a teaching language for undergraduate computer science courses, Scheme has always been aimed at a different language niche than Common Lisp. In particular, Scheme's designers have focused on keeping the core language as small and as simple as possible. This has obvious benefits for a teaching language and also for programming language researchers who like to be able to formally prove things about languages.

It also has the benefit of making it relatively easy to understand the whole language as specified in the standard. But, it does so at the cost of omitting many useful features that are standardized in Common Lisp. Individual Scheme implementations may provide these features in implementation-specific ways, but their omission from the standard makes it harder to write portable Scheme code than to write portable Common Lisp code.

Scheme also emphasizes a functional programming style and the use of recursion much more than Common Lisp does. If you studied Lisp in college and came away with the impression that it was only an academic language with no real-world application, chances are you learned Scheme. This isn't to say that's a particularly fair characterization of Scheme, but it's even less applicable to Common Lisp, which was expressly designed to be a real-world engineering language rather than a theoretically "pure" language.

If you've learned Scheme, you should also be aware that a number of subtle differences between Scheme and Common Lisp may trip you up. These differences are also the basis for several perennial religious wars between the hotheads in the Common Lisp and Scheme communities. I'll try to point out some of the more important differences as we go along.

Two other Lisp dialects still in widespread use are Elisp, the extension language for the Emacs editor, and Autolisp, the extension language for Autodesk's AutoCAD computer-aided design tool. Although it's possible more lines of Elisp and Autolisp have been written than of any other dialect of Lisp, neither can be used outside their host application, and both are quite old-fashioned Lisps compared to either Scheme or Common Lisp. If you've used one of these dialects, prepare to hop in the Lisp time machine and jump forward several decades.

Who This Book Is For

This book is for you if you're curious about Common Lisp, regardless of whether you're already convinced you want to use it or if you just want to know what all the fuss is about.

If you've learned some Lisp already but have had trouble making the leap from academic exercises to real programs, this book should get you on your way. On the other hand, you don't have to be already convinced that you want to use Lisp to get something out of this book.

If you're a hard-nosed pragmatist who wants to know what advantages Common Lisp has over languages such as Perl, Python, Java, C, or C#, this book should give you some ideas. Or maybe you don't even care about using Lisp--maybe you're already sure Lisp isn't really any better than other languages you know but are annoyed by some Lisper telling you that's because you just don't "get it." If so, this book will give you a straight-to-the-point introduction to Common Lisp. If, after reading this book, you still think Common Lisp is no better than your current favorite languages, you'll be in an excellent position to explain exactly why.

I cover not only the syntax and semantics of the language but also how you can use it to write software that does useful stuff. In the first part of the book, I'll cover the language itself, mixing in a few "practical" chapters, where I'll show you how to write real code. Then, after I've covered most of the language, including several parts that other books leave for you to figure out on your own, the remainder of the book consists of nine more practical chapters where I'll help you write several medium-sized programs that actually do things you might find useful: filter spam, parse binary files, catalog MP3s, stream MP3s over a network, and provide a Web interface for the MP3 catalog and server.

After you finish this book, you'll be familiar with all the most important features of the language and how they fit together, you'll have used Common Lisp to write several nontrivial programs, and you'll be well prepared to continue exploring the language on your own. While everyone's road to Lisp is different, I hope this book will help smooth the way for you. So, let's begin.

---

[1]Perl is also worth learning as "the duct tape of the Internet."

[2]Unfortunately, there's little actual research on the productivity of different languages. One report that shows Lisp coming out well compared to C++ and Java in the combination of programmer and program efficiency is discussed at `http://www.norvig.com/java-lisp.html`.

[3]Psychologists have identified a state of mind called *flow* in which we're capable of incredible concentration and productivity. The importance of flow to programming has been recognized for nearly two decades since it was discussed in the classic book about human factors in programming *Peopleware: Productive Projects and Teams* by Tom DeMarco and Timothy Lister (Dorset House, 1987). The two key facts about flow are that it takes around 15 minutes to get into a state of flow and that even brief interruptions can break you right out of it, requiring another 15-minute immersion to reenter. DeMarco and Lister, like most subsequent authors, concerned themselves mostly with flow-destroying interruptions such as ringing telephones and inopportune visits from the boss. Less frequently considered but probably just as important to programmers are the interruptions caused by our tools. Languages that require, for instance, a lengthy compilation before you can try your latest code can be just as inimical to flow as a noisy phone or a nosy boss. So, one way to look at Lisp is as a language designed to keep you in a state of flow.

[4]This point is bound to be somewhat controversial, at least with some folks. Static versus dynamic typing is one of the classic religious wars in programming. If you're coming from C++ and Java (or from statically typed functional languages such as Haskel and ML) and refuse to consider living without static type checks, you might as well put this book down now. However, before you do, you might first want to check out what self-described "statically typed bigot" Robert Martin (author of *Designing Object Oriented C++ Applications Using the Booch Method* [Prentice Hall, 1995]) and C++ and Java author Bruce Eckel (author of *Thinking in C++* [Prentice Hall, 1995] and *Thinking in Java* [Prentice Hall, 1998]) have had to say about dynamic typing on their weblogs (`http://www.artima.com/weblogs/viewpost.jsp?thread=4639` and `http://www.mindview.net/WebLog/log-0025`). On the other hand, folks coming from Smalltalk, Python, Perl, or Ruby should feel right at home with this aspect of Common Lisp.

[5]AspectL is an interesting project insofar as AspectJ, its Java-based predecessor, was written by Gregor Kiczales, one of the designers of Common Lisp's object and metaobject systems. To many Lispers, AspectJ seems like Kiczales's attempt to backport his ideas from Common Lisp into Java. However, Pascal Costanza, the author of AspectL, thinks there are interesting ideas in AOP that could be useful in Common Lisp. Of course, the reason he's able to implement AspectL as a library is because of the incredible flexibility of the Common Lisp Meta Object Protocol Kiczales designed. To implement AspectJ, Kiczales had to write what was essentially a separate compiler that compiles a new language into Java source code. The AspectL project page is at `http://common-lisp.net/ project/aspectl/`.

[6]Or to look at it another, more technically accurate, way, Common Lisp comes with a built-in facility for integrating compilers for embedded languages.

[7]*Lisp 1.5 Programmer's Manual* (M.I.T. Press, 1962)

[8]Ideas first introduced in Lisp include the if/then/else construct, recursive function calls, dynamic memory allocation, garbage collection, first-class functions, lexical closures, interactive programming, incremental compilation, and dynamic typing.

[9]One of the most commonly repeated myths about Lisp is that it's "dead." While it's true that Common Lisp isn't as widely used as, say, Visual Basic or Java, it seems strange to describe a language that continues to be used for new development and that continues to attract new users as "dead." Some recent Lisp success stories include Paul Graham's Viaweb, which became Yahoo Store when Yahoo bought his company; ITA Software's airfare pricing and shopping system, QPX, used by the online ticket seller Orbitz and others; Naughty Dog's game for the PlayStation 2, Jak and Daxter, which is largely written in a domain-specific Lisp dialect Naughty Dog invented called GOAL, whose compiler is itself written in Common Lisp; and the Roomba, the autonomous robotic vacuum cleaner, whose software is written in L, a downwardly compatible subset of Common Lisp. Perhaps even more telling is the growth of the Common-Lisp.net Web site, which hosts open-source Common Lisp projects, and the number of local Lisp user groups that have sprung up in the past couple of years.