SAMPLE CHAPTER

Elegant applications on the JVM

Clojure IN ACTION

Amit Rathore

MANNING

Clojure in Action

by Amit Rathore

Chapter 1

brief contents

1 Introduction to Clojure

This chapter covers

- Clojure basics
- What makes Clojure special
- Clojure as a Lisp
- Clojure as a functional programming language
- Clojure as a JVM-based language

The greatest single programming language ever designed.

Alan Kay on Lisp

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never use Lisp itself a lot.

Eric Raymond

Any sufficiently complicated C or Fortran program contains an ad hoc, informally specified, bug-ridden, slow implementation of half of Common Lisp.

Philip Greenspun

1.1 What is Clojure?

Clojure is a new programming language, designed as a fresh take on Lisp, one of the oldest programming languages still in active use (it's the second oldest; only Fortran is older). Why would anyone want to learn something associated with such old technology? It turns out that although the answer is obvious to someone who already knows Lisp, it can't be explained without some background. This chapter attempts to provide that background.

We'll begin with the motivation for the language, in order to gain an understanding of why Lisp was created. After that, we'll address the seemingly strange syntax of the language. This unfamiliarity with Lisp syntax often causes people to be turned off the language; hence, it's important to understand the reasoning behind its choice. Once we get that out of the way, we'll address three main topics. The first will deal with what makes Lisp special and how Clojure benefits from being a Lisp. The second will explain what it means for Clojure to be a functional programming language. Finally, we'll discuss the advantage of Clojure being hosted on the Java Virtual Machine (JVM).

By the end of this chapter, you should have an understanding of what's possible with a language like Clojure. There is some code in this chapter that serves as examples of the topics being discussed. Because Clojure code looks so different from other languages you might be working with, you can choose to gloss over the code samples. Rest assured that the next few chapters take a more detailed look at each concept. The aim of this chapter is to arm you for what lies ahead in the remainder of the book: a deep dive into an incredible language that's both new and old.

1.1.1 Clojure—the reincarnation of Lisp

When someone says that Lisp is the world's most powerful programming language, many folks agree (even if they refer to the speaker as a smug Lisp weenie.) What other programming language can lay claim to something similar and get away with it? C++? Java? Ruby?

Many people think of Lisp as a dead language, one that no one uses anymore. At the same time, people hear of Lisp being used for some cutting-edge software systems in various domains: NASA's Pathfinder mission-planning software, algorithmic trading of hedge funds, airline reservations, data mining, natural language processing, expert systems, bio-informatics, robotics, electronic design automation, and so on.

Lisp has the reputation of being a dark art; indeed, it has been referred to as a secret weapon by several successful startups. All this is for good reason, and this chapter attempts to explain this mysticism by talking about a new Lisp called Clojure. This new computer programming language is not only a practical Lisp, but it has added to its effectiveness by embracing the functional paradigm, by incorporating concurrency semantics into its core, and by being hosted on the Java Virtual Machine.

At the end of this discussion, you won't be surprised to learn that Clojure is being used in an equally wide set of domains to solve an equally challenging set of problems:

in large-scale text-archiving and data-mining systems, software implementing semantic web technologies to understand the deep web, statistics and modeling packages, AI-driven price-optimization systems for e-commerce, flight-delay prediction, weather-based event forecasting for insurance purposes, robotics, drive worlds in the Second-Life environment, and so on.

We'll now explore each strength of the Clojure programming language in some depth.

1.1.2 How we got here

LISt Processing (Lisp) is a programming language, originally designed in 1958 by John McCarthy, who lists the Turing award among his many achievements. Its design arose from requirements in the field of artificial intelligence, specifically from a need to operate on symbolic expressions (instead of numbers) and to represent these expressions as lists of symbols. Lisp was also designed with functional abstraction as a means of managing complexity, which means that functions are first-class citizens of the language and can be passed around like values and can be composed of each other. This is different from languages like Java and Ruby. Finally, Lisp was created with a "code as data" ideology, which meant that code could be manipulated as easily as regular data. This combination of features, as you shall see, results in a tremendous level of expressiveness.

But over the next few years, circumstances conspired against Lisp. A lot of money was invested in AI research during the 70s and 80s, but AI ultimately delivered little of all it had promised. Because Lisp had become associated with the field of artificial intelligence, when the AI boom ended, so did the popularity of the language. Many blamed the failure of AI on Lisp, and the stigma has been difficult to lose.

Many Lisps have been born since those early days, and many Lisps have passed into oblivion. Some are still being used today, especially certain Common Lisps like SBCL, CMUCL, and Allegro CL. Several computer science schools use Scheme as a teaching language, a role that it's admirably suited for.

There have been several attempts at a Lisp hosted on the JVM: JScheme, Kawa, ABCL, and others. For a variety of reasons, these never became particularly popular. Clojure is the latest attempt at reviving Lisp on the JVM, and Rich Hickey, its creator, has done an incredible job. Clojure, finally, could be the Lisp that survives. Indeed, it could be the future of Lisp and of dynamic programming languages.

We'll wrap up this section with an overview of the approach this book will take in order to teach Clojure. We'll then dive right into the first thing most people talk about when starting to learn a Lisp-like language—the syntax.

1.1.3 How this book teaches Clojure

The philosophy of this book rests on two main pillars: emphasizing Clojure's first principles and taking a hands-on approach to understanding those principles. You'll see plenty of code examples that illustrate these concepts. Programming in Clojure

requires a new way of thinking, one that's probably different from what you might be used to, depending on your programming background. We'll address this by focusing on the basics of Lisp, on functional programming, and on understanding the new facilities provided by Clojure.

As you work through the book, we'll write a lot of code that will be applicable to a variety of systems developed today. Once we get past the features of the language itself, we'll address real-world topics such as test-driven development, data storage, web services, and more. We'll look at scaling our Clojure applications through the use of messaging systems, and we'll even create a little distributed computing framework. We'll address using Clojure to process big data by leveraging technologies such as Hadoop and HBase. We'll also look at creating domain-specific languages in Clojure.

With this background, we're ready to explore Clojure. Our first stop is going to address a question nearly everyone asks the first time they see code written in the language: why does Clojure code look the way it does?

1.2 Understanding Clojure syntax

When most people think of learning a new programming language, they first think of syntax. Syntax is what makes languages look different from each other; indeed, it's often a reason why some people like (or dislike!) certain languages.

Syntax, however, is only skin-deep. Concrete syntax, which is the rules that the language imposes on the programmer in terms of what each construct looks like, isn't that important. Compilers generate a data structure called an abstract syntax tree after parsing the code written in the concrete syntax of the language. The source code of the program is discarded once the AST (abstract syntax tree) is generated. For all other phases of compilation (for example, semantic analysis), only the AST is required. You might say, therefore, that concrete syntax is fundamentally for humans. That's not to say that human convenience isn't important, but syntax shouldn't get in the way of what's possible with a computer language. (We'll revisit this issue in section 1.3.1.)

Clojure is an extremely simple language to learn; from a syntax point of view, there's nearly nothing to it. Take a look at the following line of Clojure code:

```
(add 3 5 7)
```

This expression is a call to the `add` function with three numbers as arguments. A function is always called this way, with the function name appearing first followed by any number of arguments. Let's examine the general rules of Clojure syntax.

A Clojure expression is either a symbol or a sequence. For example, in the one-liner just shown, the expression is a list containing four symbols (add, 3, 5, and 7). An example of an expression that contains only a symbol is `13` or `+`.

If an expression is a sequence, it's either empty or contains other symbols or sequences. A symbol is anything that appears in the program text. That's all there is to it! Note the recursive nature of these rules. By allowing a Clojure list to contain other lists (in essence allowing expressions to be nested), arbitrarily complex expressions

can be represented. Further, because of this regularity, all Clojure expressions look the same. The same evaluation rule applies to nearly all Clojure expressions, with only a few special cases. The evaluation rule states that the first symbol in a list represents a function and is evaluated by treating the remaining expressions in the list as parameters. You saw this in the call to the `add` function previously.

This simplicity is also its strength and is what makes Lisp's famous macro system possible. Macros are tiny, inline code generators that any programmer can use to modify program code or even generate arbitrary code on the fly. This language-level construct that allows code generation and transformation at runtime (the compile phase of the runtime, to be specific) is one of the reasons why Clojure is so different from other languages and why it's so powerful. You'll learn a lot about macros in this book.

The remainder of this section talks about Clojure's syntax. The comparison with XML will help you realize that the unfamiliar parentheses have familiar cousins. By the time you're finished reading this section, you'll at least understand the reasoning behind all those parentheses. You'll certainly be more comfortable reading Clojure code. You may even begin to see the possibilities that representing code this way provides.

1.2.1 XML and parentheses

Many people get turned off at the sight of Clojure code; they complain about too many parentheses. Let's see why they're no big deal, and in fact, they're a large source of Clojure's capabilities. Few other programming languages have the sort of metaprogramming capabilities that Clojure has, and this is in large part due to the way code is represented in the language. First, however, you'll see that the unfamiliar syntax is similar to something most programmers are already familiar with.

XML has been used for many purposes, including as a programming language. XSLT is one such example. As a thought experiment, let's use XML to create our own programming language. It might look something like the following listing.

Listing 1.1 A small program written in our fictitious XML-based language

```
<program>
  <function name=addToStock>
    <param name=counter></param>
    <callFunction name=increment>
      <argument value=counter></argument>
    </callFunction>
  </function>
  <function name=removeFromStock>
    <param name=item></param>
    <callFunction name=decrementFromStockFile>
      <argument value=item></argument>
    </callFunction>
  </function>
</program>
```

This program should be understandable by anyone who has used any kind of XML-based programming tool. For example, Ant build files look similar in some respects. With a little imagination, you could envision a system that can read this file and execute the code described within it.

These days, however, everyone likes to criticize XML for being kludgy and verbose. So let's try to clean it up by removing nonessential things from this example. First, if we assume that these are source code files, we don't need the root `program` tag. Also, let's get rid of all closing tags, because we can still write a parser that understands the program structure without named closing tags. The resulting code might look like the next listing.

Listing 1.2 A slightly less verbose version of the same program

```
<function name=addToStock>
    <param name=counter></>
    <callFunction name=increment>
      <argument value=counter></>
    </>
  </>

  <function name=removeFromStock>
    <param name=item></>
    <callFunction name=decrementFromStockFile>
      <argument value=item></>
    </>
  </>
```

Let's make another couple of improvements: let's use starting angle brackets to denote the start of statements (no closing angle brackets until the end of the statement), and let's use a simple closing angle bracket to denote the ends. The code might now look like the following:

```
<function name=addToStock
    <param name=counter >
    <callFunction name=increment
      <argument value=counter >
    >
  >

  <function name=removeFromStock
    <param name=item >
    <callFunction name=decrementFromStockFile
      <argument value=item >
    >
  >
```

For our final set of changes, let's cook up a couple of rules. When defining a function, we'll use the special `define` construct, which is always followed by a set of symbols between brackets. These symbols begin with the name of the function being defined and are followed by its parameters. The `define` construct doesn't need an extra closing bracket because it ends with the body of the function definition.

Also, the first symbol in any non-special statement is always the name of a function, so we don't need to call out that fact. The following symbols are always parameters, so we don't need to call out that fact either. The resulting code looks like the following:

```
<define < addToStock counter >
    <increment counter > >

  <define < removeFromStock item >
    <decrementFromStockFile item > >
```

This is still quite readable and still follows rules that a parser can use to decode. For a bit of flourish, let's switch to using the nicer, curvier parentheses instead of the pointy angle brackets. Take a look:

```
(define (addToStock counter)
    (increment counter))

  (define (removeFromStock item)
    (decrementFromStockFile item))
```

Believe it or not, this is Scheme (a Lisp dialect) syntax. Most Lisps look similar, with minor differences about how many parentheses are used and where they go.

The reason the parentheses exist is the same reason tags exist in XML, and that's to define a tree structure, which contains (in this case) the source code of the program. Just as XML can be manipulated (or generated) with ease, so can Lisp programs. This is because they get converted to simple data structures themselves. As mentioned earlier, this code manipulation and generation is done using the macro system. We'll explore the macro system a little bit in section 1.3.1 and then again in chapter 7. Thanks to this language-level code generation facility, some expressive abstractions can be created rather easily in Clojure.

Before completing this section, we'll address one more aspect of Clojure syntax. Clojure has syntactic sugar that makes it easy to work with all its core data structures, namely lists, vectors, and hashes. This convenience makes Clojure code more readable as well.

1.2.2 Lists, vectors, and hashes

Okay, so I lied a little bit. Clojure does have a little more syntax than other Lisps, but happily, this extra syntax improves readability.

In order to make it easier to read and write Clojure code, it uses two other types of brackets, square brackets and braces, in addition to the parentheses. As usual, simple lists are denoted using parentheses, and most of your Clojure code will use lists. Here's a typical example:

```
(do
    (process-element (first all-the-xs))
    (process-element (last all-the-ys))
    (find-max all-the-numbers))
```

Vectors are denoted using square brackets. Vectors are like lists and can be used like them, except that they can be indexed by integers. Clojure uses vectors to denote function arguments or binding forms.

```
(defn do-something [an-argument another-argument]
  (do-something-with-both an-argument another-argument)
  (return-answer an-argument another-argument))
```

Hash maps are denoted using braces. Hash maps behave as you'd expect; they're collections of key-value pairs where values can be indexed by their keys. Here's what that looks like:

```
(def a-map
  {:first-key "first-value"
   :second-key "second-value"
   "third-key" :third-value
   "fourth-key" 4})
```

The combination of these three notations—parentheses, square brackets, and braces—makes Clojure easy to read when compared to other Common Lisp or Scheme.

You should now have an idea of what Clojure code looks like and even why it looks like it does. With the question of the strange syntax out of the way, we can start talking about the other features that Clojure offers.

1.3 The sources of Clojure's power

Computers understand only one language, binary code. All other programming languages, including assembly (symbolic representation of the computer's instruction set), are at a higher level than native machine code. The reason these other languages exist is so that programmers don't have to deal with cryptic sequences of ones and zeroes. High-level languages increase the programmer's productivity.

Programming languages vary greatly in expressiveness. Low-level languages, such as assembly language and the C programming language, are useful for certain tasks, such as writing operating systems and device drivers. For most other programming tasks, software engineers favor high-level languages, such as C++, Java, Ruby, or Python. Each programming language has a different philosophy, for instance, static or dynamic, strongly or weakly typed. It's these differences in design that result in different levels of programmer productivity.

Clojure has three broad sources of power: the ideology of Lisp, the functional paradigm, and the JVM. The rest of this section explores each of these topics in some detail. The rest of this book illustrates these as well and makes the reasoning behind these choices quite apparent.

1.3.1 Clojure and Lisp

Lisp is different from most other programming languages. This is immediately apparent to anyone who looks at a fragment of Lisp code. In section 1.2, we talked about why the syntax looks as it does and how it relates to being able to generate code on the fly. Languages that represent code using their own data structures are said to follow the code-as-data philosophy.

This idea of code as data is why the Lisp family of languages can offer the ultimate in programmer productivity for such a large class of problems. Specifically, it allows

program code to be manipulated or generated at runtime (at compile time to be specific), through a facility called the macro system. In section 1.3.4, we'll explore in more detail what it means for a language to be a Lisp.

1.3.2 Clojure and functional programming

A computer language is said to be functional if it treats computation as the application of mathematical functions. As in mathematics, functional programs have no state that arbitrarily mutates. In this sense, functional programming languages are different from other popular languages such as Ruby and Java. In essence, the difference is that these other languages are imperative in nature, which emphasizes modifying state as a means of representing computations.

Programming in functional languages is different from programming in imperative languages, and it can take a little getting used to. It's different from procedural languages (such as C), despite the superficial similarity of procedures and functions. The prime difference is that procedures aren't first class, which means that procedures are subroutines, whereas functions are real objects that can be passed around and created dynamically.

Being a functional language, however, Clojure is able to provide some rather unique features. These include immutability, higher-order functions, laziness, and the excellent concurrency semantics that allow Clojure programs to use all available CPU cores in a thread-safe manner. A combination of these features packs quite a punch. When combined with the fact that Clojure is a Lisp, magical things can be made to happen.

Code written in Clojure is a great deal shorter and less error prone when compared to that written in other languages. We'll examine why this is so next.

1.3.3 Clojure and the JVM

For a programming language to be productive, it needs to come packaged with a large set of libraries. Indeed, libraries often define a language as much as its syntax. Think of C++ with its standard library and the STL. As a corollary, the lack of a comprehensive set of libraries can cause a language to be neglected by the developer community. Indeed, in many ways, Lisp itself is an example of this problem.

Clojure sidesteps this problem neatly. By being hosted on the JVM, programmers have instant access to thousands of libraries and frameworks that serve a wide variety of purposes. The Clojure runtime benefits from the high-performance characteristics of the HotSpot VM, which can optimize programs at runtime to make them run faster. You'll see how to exploit this synergy with Java in your programs.

1.3.4 Clojure as a Lisp

As described earlier, having been designed back in 1958, Lisp is one of the oldest computer languages. It had originally been created as a notation to aid in the advancement of the theory of computation. Specifically, it was being used to develop a model of computation based on lambda calculus. Further, because it was born during a

period of intense interest in artificial intelligence, there was a specific need to represent symbolic expressions as data. This combination of the functional basis and the idea of code as data made it suitable for the complex applications being developed at the time. In fact, the original Lisp spawned several dialects, and they all shared the same basic tenets. These languages formed the Lisp family of languages. Clojure shares these advantages because it too belongs to this family.

Over the past few pages, we've talked a little about the relationship of some of the features of Clojure to its syntax. The parentheses serve an important purpose (similar to tags in an XML document) by marking the beginning and the end of each unit of code. In order to understand what being a Lisp means for Clojure, you need to first understand a couple of terms: *s-expressions* and *forms*.

A line of Clojure code can be made up of a symbol that evaluates to something such as `account-balance` (a variable) or `calculate-total` (a function). It can be a literal such as the string `"Enter password"` or the number `124.95`. It can be a list (denoted by a pair of parentheses), which could in turn contain symbols, vectors, hash maps, sets, or other nested lists. When the program runs, these lines of code are first converted into Clojure objects called s-expressions. An example of an s-expression is `println`. Another is `(+ 1 2)`. Here's another example of a larger s-expression:

```
(defn print-all-things [a-list-of-things]
  (let [total (count a-list-of-things)]
     (println "Total things: " total)
     (dorun (map println a-list-of-things))))
```

Not all s-expressions are valid. An example of an invalid s-expression is `("me" "and" "you")`,because the first element of a Clojure s-expression must be a function, a macro, or a special form. You'll learn what these are soon. For now, suffice it to say that the string `"me"` is not one of these. A valid s-expression is one that follows all the rules of Clojure and is called a *form*. Valid Clojure programs are composed of one or more Clojure forms.

It's useful to have a basic understanding of how computer languages work. Figure 1.1 illustrates the three phases of processing before a file containing source code can execute as a program.

The lexical analyzer tokenizes the incoming character stream of the source code into symbols and words. The parser parses the tokens into an AST (abstract syntax tree, as was mentioned earlier). This AST is a syntax-free representation of the source code; it represents each line of code in a tree structure, as XML does. A simple example is shown in figure 1.2. From this point onward, the source code is no longer needed. The AST is checked for validity. If it follows the rules of the language, it's

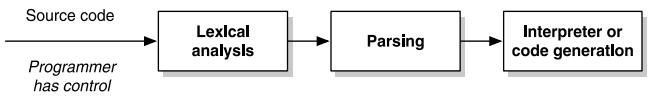

Figure 1.1 Shows the stages of a typical language processor and where the developer has programmatic control. Clojure doesn't follow this model, as you'll see.

deemed ready for the final stage. This third phase depends on whether the language is interpreted or compiled. If it's an interpreted language, each form is evaluated and executed in sequence. If it's compiled, machine code is generated, which can then be executed later.

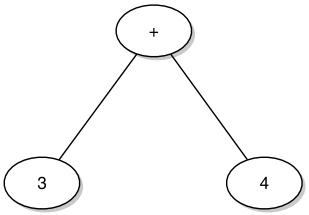

Figure 1.2 An example AST. This shows a tree structure representation of code that might be written as `1 + 2`, in a language such as Java or Ruby. There's no notion of concrete syntax here.

With this background out of the way, we can talk about more interesting things. In computer languages that work as described previously, as far as the programmer is concerned, the source code is what gets executed (minus runtime optimizations). The whole AST thing is an internal detail well below the required level of abstraction when writing a program. In fact, the programmer has no control of what happens during the various stages of transformation and processing of the code. You may well wonder why such a thing would even be desirable. What possible use could there be of being able to programmatically manipulate an AST of a program? It turns out that having such control allows for great flexibility, and it comes in the form of the macro system. We'll explore macros in the next section.

The Clojure language runtime, on the other hand, works differently from other languages. It features a just in time (JIT) compiler, and thanks to the JVM, it compiles and executes code extremely fast. Internally though, it can be thought of as having two separate phases, as illustrated in figure 1.3.

The first phase, as the name implies, concerns itself with reading the source code and converting it into s-expressions. Invalid s-expressions cause read errors, and the program will abort with an appropriate error message. The second phase is to evaluate valid s-expressions (forms) to produce a return value. If code needs to be compiled, it happens here and is then executed. As alluded to, the power of Lisp kicks in here, because the programmer has full control between the read and evaluate phases.

We're now going to examine the first phase in a little more detail. The Clojure reader is quite a marvel of innovation. Along with the syntax, It's what makes this whole macro thing possible.

THE READER

To understand this better, it's useful to know what the Clojure reader is and what it does. The reader is the entity that reads source code, from a program source file, for instance, and converts it into s-expressions. These s-expressions are composed of ordinary

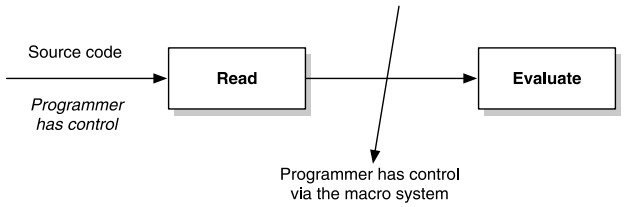

Figure 1.3 Shows the stages of the Clojure runtime. Unlike typical languages, the developer has programmatic control in between the read and evaluate stages, via the macro system.

Clojure data structures such as symbols, lists, vectors, sets, and hash maps. These data structures are then evaluated by the evaluator to produce a result. To make this more concrete, let's consider a contrived example. Here's a function definition in Clojure:

```
(defn my-own-function [an-argument another-argument]
  (println "The arguments were:" an-argument another-argument)
  (process-these an-argument another-argument))
```

Think of this as a list. The first symbol is `defn`, and it's followed by a symbol that's the name to which the function will be bound. Next, is a vector of two symbols, `an-argument` and `another-argument`. They're followed by yet another list that begins with `println`, and so on. This form represents a function definition, and when it's evaluated, the symbol `my-own-function` gets associated with the newly created function object.

To summarize, the reader converts source code into an AST implemented using Clojure's own data structures. The reader does this by using parentheses (and other brackets) as delimiters of each s-expression. In essence, the combination of the brackets and the reader allows code to be written in the AST directly. This is the reason why it's sometimes said that Lisp has no syntax. It's probably more accurate to say that Lisp has no concrete syntax. After all, as described earlier, language processors discard concrete syntax once the AST is created.

Programming languages that exhibit this property are called homoiconic (*homo* means same, *iconic* means representation). Code is written in the language's own data structures. This homoiconicity is also what makes Clojure's macro system possible, and you'll see this in the next section. Further, the reader invalidates the need to write language parsers because the reader does that already, and along with `eval` everything needed to write an internal DSL is already present. This is why most DSLs in Lisp look structurally similar to Lisp (as opposed to the English-like syntax favored by programmers of other languages). We'll explore this in some depth in chapter 15, and we'll write several DSLs that illustrate this point throughout the second part of this book. Meanwhile, let's get back to our discussion of the macro system.

THE MACRO SYSTEM

Having seen the s-expression for the definition of `my-own-function`, let's consider another data structure. This s-expression is structurally similar to the form that defined `my-own-function` previously:

```
(1 [2 3] (4 5 6 7) (8 9 10))
```

This is a nested list containing symbols that represent numbers. It contains a leading number, `1`, a vector, `[2 3]`, and two lists, `(4 5 6 7)` and `(8 9 10)`. It's easy to imagine writing code that transformed this data structure into something else. The s-expressions representing the definition of `my-own-function` can also be manipulated the same way. For instance, let's add logic to log the fact that it was called. We'd like the new definition to be

```
(defn my-own-function [an-argument another-argument]
  (log-function-call "my-own-function")
  (println "The arguments were:" an-argument another-argument)
  (process-these an-argument another-argument))
```

Doing this programmatically is as simple as inserting another list into the original s-expression. The list that should be inserted is `(log-function-call "my-own-function")`. This list has `log-function-call` as its first symbol, followed by a string containing the name of the function being logged. This ability to programmatically manipulate code is what it means to have access to the AST as a simple data structure. It can be manipulated and transformed as desired. To make this useful, however, there needs to be a way for the transformed data structure to be evaluated instead of the original form. To be specific, it would need a hook between the read and evaluate phases of the Clojure runtime.

Clojure's macro system is exactly that hook. Macros are Clojure functions, but they accept s-expressions as their arguments. Because s-expressions are data structures, they can be transformed and returned, and the return values are used in place of the original forms. These code-transforming macros are used to create mini-languages on top of Clojure, or domain-specific languages (DSLs) as they're called these days.

Let's now look at how macros can be used to write code that manipulates code. Specifically, you'll see how the macro system can be used to eliminate duplication and to increase the expressiveness of the language by adding domain-specific constructs.

Metaprogramming with macros

The introduction of Clojure's macro system poses a question: "What could such programmatic manipulation of code be used for?" The answer is "a lot" and has to do with metaprogramming, which is the idea of programs generating or manipulating other programs (or themselves). Such metaprogramming is used for several purposes, for instance, to reduce boilerplate code or to build syntactic abstractions (DSLs) on top of the core language. Clojure's macro system takes such metaprogramming to a whole new level when compared to metaprogramming facilities provided by languages such as Python or Ruby.

Another reason metaprogramming is easier in Clojure is because of its strange syntax. All Clojure forms have the same structure, which makes code generation easy compared to languages that have non-regular syntax. To generate Clojure code, you create Clojure data structures containing symbols and other data structures! This absence of formal syntax and the existence of the macro system make Clojure well suited to creating DSLs. In Clojure, creating mini-languages on top of the core language is a common approach to programming. It's the whole reason why Lisp is considered a *programmable* programming language.

To wrap up this section, let's look at a couple of examples of using macros.

Example of a macro—removing boilerplate

Certain kinds of boilerplate code can't be eliminated in languages like Java. Consider the following Java methods.

Listing 1.3 Typical duplication in Java code

```
public List getExpenses(long userId, Date start_date, Date end_date) {
    AuditLogger.logLookup(userId, AuditCode.GET_EXPENSES);
    ExpensesConnection connection = new ExpensesConnection(userId);
    List expenses = connection.findAllBetween(start_date, end_date);
    connection.close();
    connection.flush();
    return expenses;
}

public void addExpense(long userId, Date date, BigDecimal amount) {
   AuditLogger.logLookup(userId, AuditCode.ADD_EXPENSES);
   ExpensesConnection connection = new ExpensesConnection(userId);
   connection.saveNewExpense(date, amount);
   connection.flush();
   connection.close();
}
```

In both these methods, the first thing that happens is that an audit log entry is created to record the fact that the method was called. Then, an `ExpensesConnection` object is created, which is used for different purposes in each method. The first two lines of code in each method are repeated. There's little we can do in Java to eliminate this duplication. We could try using the template method pattern, but it would still not be elegant enough, without gaining much in return.

Let's examine the same situation in Clojure. First, equivalent code in Clojure might look like that in the following listing.

Listing 1.4 The same code in Clojure, written as is

```
(defn get-expenses [user-id start-date end-date]
  (create-audit-log user-id GET-EXPENSES)
  (let [connection (connect-to-expenses-db user-id)
        expenses (find-all-between connection start-date end-date)]
    (close-connection connection)
    (flush-connection connection)
    expenses))

(defn add-expense [user-id date amount]
  (create-audit-log user-id ADD-EXPENSE)
  (let [connection (connect-to-expenses-db user-id)]
    (save-new-expense connection date amount)
    (flush-connection connection)
    (close-connection connection)))
```

In Clojure, we could write a macro that we might name `with-audited-connection` that would handle this duplication. The resulting code would look like the following listing.

Listing 1.5 Removing duplication via a domain-specific macro

```
(defn get-expenses [user-id start-date end-date]
  (with-audited-connection [user-id connection]
    (find-all-between connection start-date end-date)))
```

```
(defn add-expense [user-id date amount]
  (with-audited-connection [user-id connection]
    (save-new-expense connection date amount)))
```

The implementation of the `with-audited-connection` isn't shown here because chapter 7 focuses exclusively on macros, but the two functions in listing 1.5 are much cleaner than what would be possible with most languages. The domain-specific `with-audited-connection` macro is now available to use anywhere you need to connect to the expenses data store in an audited way. It also takes care of flushing and closing the connection, so you never have to remember to do this. This is another happy advantage, and it would be difficult to implement the same way in a language like Java. In fact, `with-audited-connection` can set things up so that any function called within its scope will be audited appropriately, without having to be explicit about it.

If this were a more dynamic language such as Ruby, you could get rid of some duplication by creating a third method that did the audit logging and accepted a block that did the remaining work. To see what macros can do that blocks and functions can't, let's look at another example that illustrates an even more powerful feature of macros—manipulating raw source code.

EXAMPLE OF A MACRO—SYNTACTIC ABSTRACTION

We discussed the advantages that a homoiconic language has in being able to make a macro system possible. Prefix notation goes along with it, because it adds uniformity to all function calls; the function object appears first, followed by arguments. This works fine for most functions, but it causes mathematical operators to look a bit strange, until you get used to them. Let's write a somewhat frivolous macro to support in-fix notation for math operators. First, a quick recap of what happens if we try to evaluate `(1 + 2)` at the REPL:

```
user=> (1 + 2)
; Evaluation aborted.
java.lang.Integer cannot be cast to clojure.lang.IFn
  [Thrown class java.lang.ClassCastException]
```

Our `infix` macro will allow us to write addition in this manner:

```
user=> (infix (1 + 2))
3
```

Experienced Lispers will question the advantage of creating such a macro, because prefix notation has significant advantages (that we'll explore shortly). The point is that it's easy to write such a macro in Clojure, and you'll see the implementation of this macro in chapter 7. As a hint, it literally manipulates the s-expression `(1 + 2)` into `(+ 1 2)`. This is a trivial example of macros, but it would be rather hard to do this in most other languages.

Clojure macros can do much more. For now, it's worth noting that they are one of the crucial features of Clojure that distinguishes it from most other languages. Macros are the ultimate form of metaprogramming.

1.3.5 More advantages of Clojure

As you saw, the macro system is powerful indeed. There are, still more advantages that come from Clojure being a Lisp. We'll discuss a few of them now.

DYNAMIC LANGUAGE

Clojure, like Lisp, is a dynamic language. This means that many more things are determined at a program's runtime rather than when a compiler looks at the source code. This allows programs to be written in ways that wouldn't be possible in static languages. For example, many static languages require type declarations when using variables or defining functions, whereas Clojure doesn't.

As another example, Clojure has an `eval` function that allows code to be loaded up and executed at runtime. This feature is absent in nearly all static languages.

THE REPL

Clojure includes an interactive environment that allows code to be typed in and executed. It gives quick feedback and is helpful in incrementally writing code to solve the problem at hand. This interactive environment is called the REPL, which stands for read-eval-print loop.

Many Clojure editors integrate with the REPL. This allows programmers to use the IDEs' text-editing features to write the code the way they're used to. The integration provides a convenient way to evaluate code written in the editor inside of the REPL. Because the REPL is a long-running process, the edit-evaluate-test-edit cycle can keep going as long as the programmer wants. The workflow becomes an uninterrupted flow of feedback-driven editing and REPL interaction. In fact, in chapter 8, we'll illustrate this by using the REPL along with creating unit tests for a few functions, in a step-by-step manner. The ability to work this way makes REPL an important contributor to the increased productivity of working in Clojure.

This section talked about the advantages that Clojure gains from belonging to the Lisp family. There's another reason why Clojure code is often shorter than code in other languages, and it has to do with its primary means of representing computation: function evaluation. We'll explain why this is so in the next section.

1.3.6 Clojure as a functional language

Clojure deviates from many Lisps in that it exhibits far more functional purity. An example of this is in its choice of treating all Clojure data structures as immutable. Along with this aspect of functional programming, Clojure encourages the use of higher-order functions such as `map` and `reduce`. Core data structures are lazy, which means things get executed only as needed, thereby enabling some rather efficient programming constructs. An example of the use of laziness is the ability to define and use infinite sequences. Finally, Clojure takes advantage of its immutability and provides language-level support for safe, lock-free concurrency (using a software transactional memory model). We'll examine these features in this section.

HIGHER-ORDER FUNCTIONS

Clojure functions are first class, which means that functions can be passed as parameters to other functions, can be created dynamically, and can be returned from functions. They can be stored inside other data structures like regular data. Clojure also provides lexical closures, which is a powerful construct that allows for expressive code. You'll learn about closures, along with other aspects of functional programming, in the upcoming chapters of this book.

This functional aspect of Clojure makes it easy to write code using higher-level constructs like `map`, `reduce`, and `filter`, which apply arbitrary functions to lists of data. Given that all core data structures are immutable, this results in tight code that has fewer bugs than equivalent code in imperative languages such as Java or C++, because the code is far more declarative in nature. After all, lesser code leaves less scope for programming error. Look at the following listing for a slightly larger example.

Listing 1.6 Typical usage of higher-order functions

```
(def post-headers [{:title "first one ever" :length 430}
                   {:title "second baby step" :length 650}
                   {:title "three is company" :length 720}
                   {:title "fourth for the road" :length 190}
                   {:title "five again" :length 280}])

(defn long-post-headers [threshold-length headers]
  (let [is-long? (fn [header]
                   (> (header :length) threshold-length))]
    (filter is-long? headers)))

(defn long-post-titles [threshold-length headers]
  (map :title (long-post-headers threshold-length headers)))

(long-post-titles 300 post-headers)
```

The output of the last function call is

```
("first one ever" "second baby step" "three is company")
```

This is how a lot of functional code looks: mapping over sequences to collect things, filtering from sequences using predicate functions, and so on. Notice how you can define local functions like `is-long?` and how it functions as a closure around the `threshold-length` parameter.

All this leads to shorter code with fewer defects. Given how fundamental this is to Clojure, you'll be seeing a lot more about higher-order functions. For now, we'll look at another important aspect of functional languages: the absence of mutating state.

IMMUTABILITY

Clojure supports another aspect of functional programming, that of immutability. What this means is that all of Clojure's core data structures are immutable—once created they can't change. When something needs to change, a new object is created that includes the change and is returned. (Clojure's implementation of immutability performs extremely well and doesn't slow down programs). What this means in practice is

that an entire class of bugs related to mutating state is eliminated from your code. To imagine how this might be so, consider an imperative `for` loop with a loop variable `counter`. If the programmer inadvertently modifies the value of `counter` in the body of the `for` loop, the code won't work properly. Similar bugs can happen with any mutable variable.

In languages that support immutability, the only way a function can do something is to return a new data structure and in this way each function is independent and can be written, debugged, and tested independently.

```
(def expenses [{:amount 12.99 :merchant "amazon"}])
(def updated-expenses (conj expenses {:amount 199.95 :merchant "frys"}))
```

That last function call returns a *new* sequence with the two expenses in it and is assigned to `updated-expenses`. The old sequence `expenses` remains unchanged. As you'll see, this immutability greatly aids Clojure's support of concurrency-safe multi-threaded programs.

By the way, in case you're wondering how you can possibly write real-world code with everything being immutable, relax! Clojure has fantastic support for explicit state management, and you'll see this soon.

LAZY AND INFINITE SEQUENCES

Most programming languages like Ruby and Java are eager. This means that when a method or constructor is called, it's executed immediately, and the result is returned. But sometimes it's desirable to defer such evaluation. An example where such a behavior is useful is to avoid unnecessary computation (say an expensive calculation returns a value that's never used).

Clojure solves this problem inside its core data structures. Clojure sequences can be *lazy*; this means that elements are not computed until the values are needed for something else. Most core functions produce lazy data structures that aren't realized until needed. Examples of such commonly used functions are `map`, `reduce`, and `filter`. Further, this implies that a chain of such function calls returns a value that's also lazy! This laziness is a source of great convenience, and you'll see more of it in action later.

As an example of laziness, Clojure allows the programmer to create infinite lists of data. These infinite data structures are called *streams* and are a different way to model the world. For instance, objects can be modeled as a series of events, as opposed to a snapshot of the world.

Here's a classic example of an infinite sequence; it's trivial but simple to understand. We'll create a sequence of all Fibonacci numbers. Yes, *all*, and because this sequence is lazy, elements will be calculated only as needed:

```
(defn next-terms [term-1 term-2]
  (let [term-3 (+ term-1 term-2)]
    (lazy-seq
      (cons term-3
            (next-terms term-2 term-3)))))
```

```
(defn fibonacci [t1 t2]
  (concat [t1 t2]
          (next-terms t1 t2)))

(take 15 (fibonacci 0 1))
```

This last function call is what realizes the lazy sequence of Fibonacci numbers. The result is

```
(0 1 1 2 3 5 8 13 21 34 55 89 144 233 377)
```

First, let's talk about the magic ingredient here, `lazy-seq`. It's a macro that doesn't evaluate its body immediately but returns a sequence-like object. This object will evaluate the body only when needed (and will also cache the result for subsequent uses, saving CPU cycles).

The important thing to realize about this code is that without `lazy-seq`, there'd be no useful way of using `next-terms`. You can see this by directly calling `next-terms` at the REPL; it will run in an infinite loop until the program runs out of memory. This happens because the REPL tries to print the sequence, and being infinite, the realization fails. By using certain functions like `take`, you can intelligently produce the required number of elements from such infinite sequences.

Languages like Java and Ruby need lots of boilerplate code that implements something like the lazy-load pattern to do something similar. Much of Clojure is already lazy, making code written in it automatically lazy. Such code then transparently benefits from this behavior. We'll visit this in more detail in the next chapter and in part 2 of the book, where we'll create an infinite sequence of messages read off a messaging system.

Now, you'll see another incredible benefit that results from a language being functional and immutable: the possibility of safe concurrency.

Concurrency and the multicore future

Moore's law states that the number of transistors that can be placed on a single integrated circuit doubles every 18–24 months. This is what has been happening to CPUs over the past two to three decades, and this gave us an exponential increase in processor speed. Advances in CPU speed can't proceed in this manner forever. Indeed Moore's law has pretty much already reached its limit in providing performance enhancements.

Over the past few years, instead of increasing the speed of individual processor cores, companies like Intel have started to increase the number of processor cores that go into a single CPU. This has caused software performance to become tied to Amdahl's law, which relates the possible performance of a running computation to the number of parallel processing units available. As the number of cores available on a CPU grows, software will need to make use of them; this will soon become a crucial way to make programs run faster.

Multithreaded programming means trouble, as anyone who has written multithreaded code knows. At least two things are related to this issue: getting multithreaded programs to behave correctly and using all available cores to speed up the program. Clojure helps with both.

SOFTWARE TRANSACTIONAL MEMORY

Clojure's state management system addresses the issue of correctness of multithreaded programs. Not only does the language provide simple ways to handle mutation, but it also provides constructs that allow and enforce safe mutation.

Clojure implements a multiversion concurrency control (MVCC)–based software transactional memory (STM) model. What this means in simpler terms is that mutating the value of an object can only be done inside a transaction (think database transaction).

This has two advantages. The first is that code becomes self-documenting. When the value of something needs to change in a thread-safe manner, the programmer must be explicit and use a special Clojure construct for it: the `ref`. The other advantage is that if you attempt to modify the value inside a `ref` without an STM transaction, the Clojure runtime will throw an exception. This is how Clojure enforces the use of the transaction semantics and helps keep code thread-safe.

When a transaction needs to commit, and another thread has already committed a change to a shared ref, the later transaction is rolled back. Clojure's STM system even retries the failed transaction several times, and as far as the programmer is concerned, all this happens transparently (this behavior depends on the function used to effect the mutation; more on this in chapter 6). Here's an example of this in action.

Listing 1.7 Clojure's STM in action

```
(def total-expenditure (ref 0))

;; The following will throw a "No transaction running"
;; IllegalStateException exception
 (defn add-amount [amount]
  (ref-set total-expenditure (+ amount @total-expenditure)))

;; The following will work fine because it will do the update inside a
;; transaction

(defn add-amount [amount]
  (dosync
    (ref-set total-expenditure (+ amount @total-expenditure))))
```

The `@total-expenditure` is a reader macro in action. It expands to `(deref total-expenditure)` and it gets the value out of the object that the ref is pointing to. Clojure provides several reader macros to make certain things convenient, and we'll visit them in the next chapter.

This language-level support for concurrency-safe state management is what makes Clojure extremely well suited for multithreaded applications. This is true whether code runs on a single core or on a multicore CPU. By using a ref and by making sure that updates are always performed inside a `dosync` block, access to the variable protected by the ref becomes thread safe.

Over the past couple of sections, you've seen how Clojure benefits from being a Lisp and from being a functional language. The macro system helps in creating powerful abstractions such as domain-specific languages. The functional features allow for

Writing Clojure-like code in an other languages

There's nothing to stop someone from writing code that treats variables as immutable (for example, by declaring them `final` in a language like Java, or through convention and discipline). Indeed, a large part of Clojure is written in Java, and you could imagine a program written with great care and careful consideration of all we've seen so far: immutability, an STM system for safe concurrency, and so on.

There's a vast difference between such an approach and writing a similar program in Clojure. The first is that the Java code (say) would look alien to even fluent Java programmers, because it would be far from idiomatic. More importantly, the difference is that although the Java code would rely on convention and discipline, the Clojure runtime would enforce it. That means the Clojure program wouldn't work (and would complain loudly when an attempt was made to run it) if, for example, some part of the code violated either immutability or STM transactions.

This support from the language makes programs written in Clojure less error prone and forms part of the reason why functional languages are a big deal.

higher-quality code thanks to higher-order functions and immutability. Finally, thanks to Clojure's built-in support for concurrency, programs can take advantage of multiple CPU cores without any effort from the programmer. These things all make Clojure a capable language. But it doesn't end there; by running on top of the Java runtime (the JVM), Clojure manages to solve a critical piece of the puzzle for any new language: the availability of libraries and interoperability with existing systems.

1.3.7 Clojure as a JVM-based language

Clojure is hosted on the Java Virtual Machine. This means that it's ultimately a Java program and runs as JVM byte code. The design of Clojure embraces interoperability with other Java libraries as one of its central goals. Indeed, when Rich Hickey designed Clojure, one of the goals was extreme practicality. In today's world, a practical programming language almost demands interoperability with Java. Clojure achieves this goal remarkably well, and in practice, this means several things.

The obvious advantage is that the Clojure programmer has instant access to the thousands of existing Java libraries. This substantially boosts productivity when compared with other Lisps (and indeed other programming languages) because there's probably a Java library for most systems and frameworks.

Another advantage that comes from running on the JVM is that Clojure can be embedded into Java programs, thereby providing an incredibly flexible scripting capability to the end user.

Often a combination of these two ideas leads to a system design where parts of the code are written in Java and other parts are written in Clojure. Clojure itself can be used as a glue language to bring the whole system together. This may be especially useful when legacy applications are involved.

One final advantage of being a JVM language is that of the JVM itself. Sun Microsystems (now Oracle) and the open-source community have together invested thousands of person-years into improving the JVM, and today it's one of the most efficient virtual machines. The HotSpot optimizer does amazing things to speed up code running on top of it, and Clojure benefits from all this innovation. Despite being incredibly dynamic, Clojure code gets compiled to Java byte code and runs as fast or nearly as fast as Java code itself. This gives programmers all of Clojure's benefits without any major performance costs.

CALLING JAVA FROM CLOJURE

It's trivial to use Java libraries from Clojure. For instance, here's an example of using a method on the string class to do some simple text manipulation:

```
user=> (.toUpperCase "clojure")
"CLOJURE"
```

This is especially convenient, because Clojure strings are Java strings. Further, once you import required classes using the `import` form, you can then use them like other Clojure code. For instance, Selenium is an open-source functional testing automation tool for web applications. You can find it at http://seleniumhq.org, and you'll need the Java Selenium client driver JAR on your classpath to try this example of using it to drive a browser:

```
(import '(com.thoughtworks.selenium DefaultSelenium))

(defn start-new-selenium []
  (let [s (DefaultSelenium. "localhost" 4444 "chrome*" "http://
     localhost:3000")]
    (.start s)
    s))
```

Clojure provides a few syntactic conveniences to help use Java classes. The `Default-Selenium.` (notice the period at the end) invokes the constructor of the class. Similarly, `.start` calls the instance method `start` on the newly constructed Selenium object.

There's syntactic sugar for creating new instances of Java classes, accessing static members of Java classes, and calling methods on Java objects. There are also helper macros that make using Java classes and objects from within Clojure easy. You can even implement interfaces using pure Clojure code. You'll see all this and more in chapter 5, which is dedicated to exploring Java-interop facilities of the Clojure language.

CALLING CLOJURE FROM JAVA

Calling a Clojure script from inside a Java program is easy too. Clojure is itself a Java program, so it's as simple as using any other Java library and knowing the API.

Here's an example:

```
import clojure.lang.RT;
import clojure.lang.Var;

// some code here
```

```
RT.loadResourceScript("clojure/file/from/classpath/clojure_script.clj");
Var aClojureFunction = RT.var("a-clojure-name-space",
                              "a-clojure-function");
aClojureFunction.invoke("an argument" "another one");
```

There's a lot more you can do here, including compiling all your Clojure code into Java byte code and then using it like any other library. This is called AOT (ahead of time) compilation, and you'll see all this and more in a future chapter.

TYPE HINTS

Clojure is an extremely dynamic language, but it strives to be so without the typical cost associated with doing everything at runtime. You can help run Clojure programs faster by giving the evaluator type hints about any Java class you're using. That way, Clojure will know how to call methods on objects of such classes without reflection.

Using type hints is easy; here's an example:

```
(import '(com.thoughtworks.selenium DefaultSelenium))

;; blah blah blah

(defn start-this-selenium [^DefaultSelenium selenium]
  (.start s)
  s)
```

The type hint is the ^ followed by the name of the class. When the start method is called, Clojure will do so without any reflection. This is much faster than without the type hint.

Typical programming workflow is to write the Clojure code, test it, and debug until you're satisfied. Then, if performance is a concern, add type hints to your code. It's easy and quick.

EXTENSIBILITY

It's worth mentioning one final point about Clojure being hosted on the JVM. Thanks to the internal design of the language, Clojure is easily extensible. This means that programmers can add to the core of Clojure in a natural manner. As an example, consider that Clojure's internal data structures like lists, vectors, and hash maps all behave in the same way: as sequences. Operations like `map`, `reduce`, and `filter` all work, no matter what kind of sequence is passed to them. Internally, they all do this because they all implement the `ISeq` interface.

The great thing about such a clean design is that anyone can add new data structure that implements the same `ISeq` interface. If your problem domain involves dealing with a stream of financial charges, for instance, you can imagine creating a new kind of sequence that would then work seamlessly with the core Clojure functions. Or, as another example, the Clojure-based `Incanter` project provides matrices that implement the `ISeq` interface. A matrix consists of a sequence of rows, and each row is a one-dimensional row matrix.

This extensibility pays off when existing (or even new) Java code needs to work with Clojure. By implementing the `ISeq` interface, the new data structure becomes native to Clojure and behaves the same as core Clojure data structures. This makes for

easy extensibility of Clojure and allows programmers to use all core Clojure functions for free with their custom data structures.

So far, you've seen where Clojure derives its power—and this comes primarily from it being a JVM-based functional Lisp. Thinking in the functional way often takes some time, especially for those coming from imperative backgrounds (such as Java and Ruby). One of the first questions people ask is whether Clojure is object oriented. Although OO has been all the rage over the past couple of decades, it isn't the perfect way of solving all problems. In fact, it's only one of the many approaches that might apply in any situation. Clojure, as a language, doesn't limit the programmer by imposing a specific paradigm. Instead, it allows programmers to move beyond objects.

1.4 Clojure—beyond object orientation

People new to Lisp and Clojure often ask where the object-oriented (OO) paradigm fits into the picture. They're disappointed to find that such languages have no obvious support for objects. The truth is more nuanced. Clojure is more general than any specific paradigm. First, let's examine the landscape of programming language paradigms. Figure 1.4 shows that both functional and OO paradigms are only a couple from the wide array that exist in the field today.

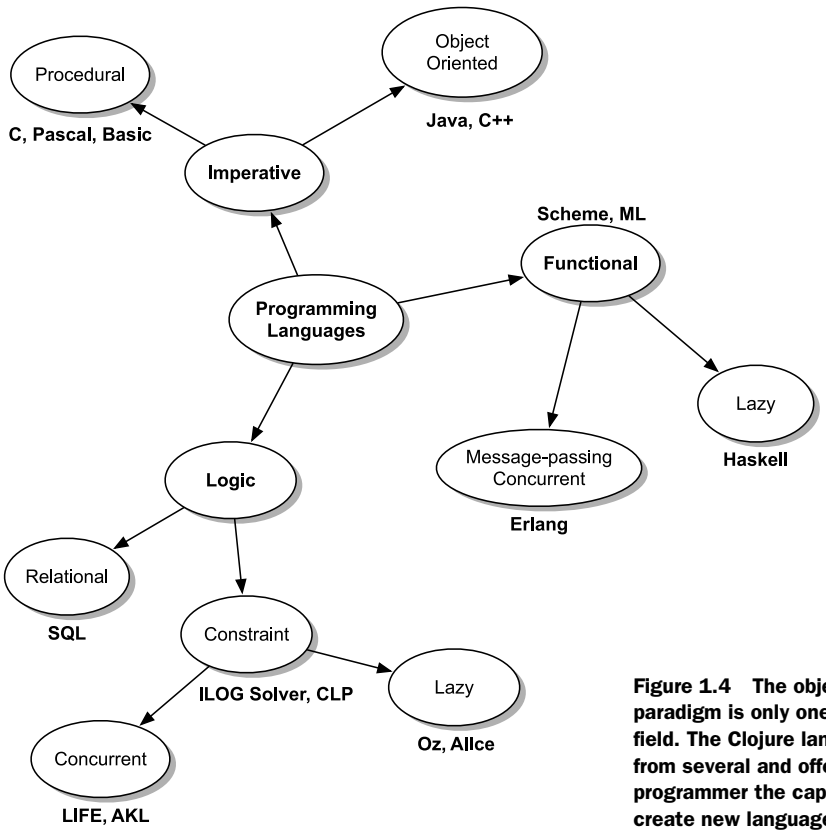

Figure 1.4 The object-oriented paradigm is only one of many in the field. The Clojure language borrows from several and offers the programmer the capability to quickly create new languages on top of it.

Because most concerns in this area are about Clojure's lack of object-oriented constructs, we'll talk about that a little. Consider, for example, that Clojure provides something called generic functions. In OO languages like Java or Ruby, method polymorphism is achieved by dispatching calls on the type of object the method is called on, and this is decided at runtime. Generic functions, also called multimethods, let the programmer decide how to dispatch functions at runtime. This is a far more capable approach to polymorphism, and you'll learn about this in a chapter on multimethods.

As you've seen, Clojure already sports various bits of different paradigms: functional, aspects of laziness, various options for concurrent programming, and so on. But as a language, it's much broader in its support for all kinds of other paradigms. You can write procedural code if you want, or you can write your program in a functional style. If you desire, Clojure programs can be written with an OO approach; indeed, you can build your own object-oriented language on top of the core language. You could build an Erlang-style process-oriented language on top of Clojure if you wanted to. The idea of building custom languages on top of a Lisp foundation isn't new. Before the term *domain-specific languages* (DSL) became popular, people were using Lisp to create such little languages. Clojure is well suited for this style of programming. The macro system built into Clojure plays a large role in this.

Clojure is also well suited for bottom-up design. Bottom-up design is an approach where higher-level components are built up from a collection of lower-level ones. A problem domain is analyzed, and several low-level components are created where each represents a single concept. These pieces are then combined to create the higher-level components as demanded by the problem at hand. Bottom-up design is often used to solve a whole class of problems in a specific domain rather than a specific instance of a problem. Systems built this way are more flexible and are more resilient to changing requirements.

Thanks to Clojure's functional paradigm (higher-order functions and closures) and the associated possibility of using function combination, it's a great language for designing systems in a bottom-up manner. In fact, Clojure programs are often "grown up" from the ground, by using this bottom-up decomposition of the domain, and are then combined with one or more little languages built on top of it. It's common to see Clojure programs that first create a mini-language that allows concepts in the problem domain to be expressed at a high level and then the problem solved in this language instead. This is a powerful style of programming that results in systems that are more flexible and code that's more readable and ultimately more maintainable. It's why Lisp is called a programmable programming language and is why Clojure benefits from being a Lisp.

The bottom line is that Clojure doesn't limit you by imposing a particular paradigm. It's general enough to support any paradigm that might suit the problem domain. Perhaps more important, it's capable of evolving into a language that works well for that paradigm and problem domain.

1.5 Summary

As you saw, Lisp is a special language, and Clojure is a special Lisp. It's homoiconic with almost no syntax and has a full macro system. It's a functional language with first-class functions and immutable data structures, and it has concurrency semantics built into its core. It's hosted on the extremely mature and performant Java VM, which allows it to offer seamless interoperability with Java code. Being as fast, or nearly as fast as Java, adds another gratifying advantage to Clojure: raw speed.

This combination packs a tremendous productivity punch. The dynamic programming style that Clojure makes possible is well suited to bottom-up design and results in programs that can do a lot more with a lot fewer lines of code. Indeed, when compared with equivalent lines of Ruby code, Clojure code can be two to three times smaller; when compared with Java, it can be nearly five to ten times smaller. Less code means fewer bugs. Clojure code comes close to satisfying the claim of being better, faster, smaller, and cheaper.

When is Clojure not applicable?

Although Clojure brings together the best ideas from a variety of programming languages, it's no panacea. There can be times when it isn't quite the language of choice for a project. The first issue is that despite the fact that the syntax is a key part of its strength, it can be alien and confusing to some programmers. If your team is unable to look past the unfamiliarity of the syntax, it can be difficult to embrace the language.

Further, the macro system can be used for good or bad. We've explored some of its power in this chapter, and there's more throughout this book. But it's important to realize that macros aren't functions and that specifically, they don't compose well. It takes experience to know when a macro is the right tool for the job, and indeed, if you can do without a macro, you should. The misuse of macros results in suboptimal code, and a team without an experienced developer or two may suffer from this problem.

The functional programming paradigm is also oftentimes a major shift for most developers. Because OOP is the most commonly used approach to writing software today, functional programming languages can leave developers at a loss as to how to design systems. It can be expeditious to have a developer or two on hand who has experience with functional programming. Without these, it can be difficult to get started.

Having said that, every tool has its pros and cons. If you can mitigate these issues, then Clojure is often the best choice, especially for new projects.

Compared to several other dynamic languages, Clojure supports better rapid prototyping and incremental development of code. By combining the dynamism of the language and the hyperproductivity of the REPL, you can develop code quickly and in a more exploratory fashion. When a bottom-up approach to code is combined with a functional approach to domain modeling and the macro system, a powerful design pattern emerges—that of creating little languages on top of Clojure. This kind of

metalinguistic programming is extremely expressive, and you'll learn a lot more about this throughout the rest of this book.

Further, the functional paradigm, along with immutable data structures, removes a whole class of bugs associated with imperative, state-based code design. And the concurrency support built into Clojure makes the complicated task of writing multithreaded programs that work correctly downright easy. All this means that Clojure programs more often work right the first time and have fewer problems during their life spans.

Overall, using Clojure on a project can mean higher-quality software that runs faster and that can use multicore CPUs efficiently and correctly. The resulting code base will be smaller, which also means it will be cheaper to develop and maintain. It's possible that Clojure (or a variant of it) will be the Lisp that survives, indeed, as the language of the future.

The remaining chapters of the first part of the book will give you a thorough understanding of Clojure the language. They'll address practical issues of using it and advanced topics that show Clojure use in the real world. To get the most of the rest of this part, however, we'll need to get some more fundamentals out of the way. The next chapter is a quick breeze through most of the basic features of the language. With that background, the details in the following chapters will be easier to understand.

PROGRAMMING

Clojure IN ACTION

Amit Rathore

Clojure is a modern Lisp for the JVM, and it has the strengths you'd expect: first-class functions, macros, support for functional programming, and a Lisp-like, clean programming style. But it's not enough to learn a language, you also need to know what to do with it.

Clojure in Action is a practical guide focused on applying Clojure to practical programming challenges. You'll start with a language tutorial written for readers who already know OOP. Then, you'll dive into the use cases where Clojure really shines: state management, safe concurrency and multicore programming, first-class code generation, and Java interop. In each chapter, you'll first explore the unique characteristics of a problem area and then discover how to tackle them using Clojure. Along the way, you'll explore practical matters like architecture, unit testing, and setup as you build a scalable web application that includes custom DSLs, Hadoop, HBase, and RabbitMQ.

What's Inside

- A fast-paced Clojure tutorial
- Creating web services with Clojure
- Scaling through messaging
- Creating DSLs with Clojure's macro system
- Test-driven development with Clojure
- Distributed programming with Clojure, and more

This book assumes you're familiar with an OO language like Java, C#, or C++ but requires no background in Lisp or Clojure itself.

Amit Rathore is a VP of Engineering with a decade of experience building highly performant data-heavy web applications.

For access to the book's forum and a free eBook for owners of this book, go to manning.com/ClojureinAction

Free eBook
SEE INSERT

"An easy to read book and a great way to get up-to-speed on Clojure."
—Craig Smith, Suncorp

"Down-to-earth and thorough."
—Stuart Caborn, BNP Paribas

"Explains functional programming with Java."
—Doug Warren
Java Web Services

"An intriguing mix of emerging tech."
—Andrew Oswald
Chariot Solutions

"Teaches you how to put Clojure into action at warp speed!"
—Baishampayan Ghose (BG)
Qotd, Inc.

ISBN 13: 978-1-935182-59-7
ISBN 10: 1-935182-59-5
54999
9 781935 182597

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]