Practical Lisp for the Java World



# Programming

Chas Emerick, Brian Carper & Christophe Grand



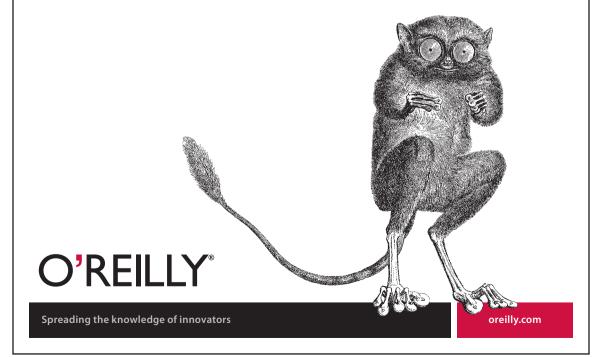
# Want to read more?

# You can <u>buy this book</u> at **oreilly.com** in print and ebook format.

## Buy 2 books, get the 3rd FREE!

Use discount code: OPC10 All orders over \$29.95 qualify for **free shipping** within the US.

It's also available at your favorite book retailer, including the iBookstore, the <u>Android Marketplace</u>, and <u>Amazon.com</u>.



#### **Clojure Programming**

by Chas Emerick, Brian Carper, and Christophe Grand

Copyright © 2012 Chas Emerick, Brian Carper, and Christophe Grand. All rights reserved. Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Julie Steele Production Editor: Teresa Elsey Copyeditor: Nancy Reinhardt Proofreader: Linley Dolby Indexer: Fred Brown Cover Designer: Karen Montgomery Interior Designer: David Futato Illustrator: Robert Romano

April 2012: First Edition.

#### **Revision History for the First Edition:**

2012-03-28 First release See *http://oreilly.com/catalog/errata.csp?isbn=*9781449394707 for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Clojure Programming*, the image of a painted snipe, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-39470-7 [LSI] 1332955528

# **Table of Contents**

Prefa	Preface xi		
1.	Down the Rabbit Hole		
	Why Clojure?	1	
	Obtaining Clojure	3	
	The Clojure REPL	3	
	No, Parentheses Actually Won't Make You Go Blind	6	
	Expressions, Operators, Syntax, and Precedence	7	
	Homoiconicity	9	
	The Reader	12	
	Scalar Literals	13	
	Comments	18	
	Whitespace and Commas	19	
	Collection Literals	19	
	Miscellaneous Reader Sugar	20	
	Namespaces	20	
	Symbol Evaluation	23	
	Special Forms	23	
	Suppressing Evaluation: quote	24	
	Code Blocks: do	25	
	Defining Vars: def	26	
	Local Bindings: let	27	
	Destructuring (let, Part 2)	28	
	Creating Functions: fn	36	
	Conditionals: if	42	
	Looping: loop and recur	43	
	Referring to Vars: var	44	
	Java Interop: . and new	44	
	Exception Handling: try and throw	45	
	Specialized Mutation: set!	45	
	Primitive Locking: monitor-enter and monitor-exit	45	
	Putting It All Together	46	

eval	46
This Is Just the Beginning	48

# Part I. Functional Programming and Concurrency

2.	Functional Programming	51
	What Does Functional Programming Mean?	52
	On the Importance of Values	52
	About Values	53
	Comparing Values to Mutable Objects	54
	A Critical Choice	58
	First-Class and Higher-Order Functions	59
	Applying Ourselves Partially	65
	Composition of Function(ality)	68
	Writing Higher-Order Functions	71
	Building a Primitive Logging System with Composable Higher-Order	
	Functions	72
	Pure Functions	76
	Why Are Pure Functions Interesting?	78
	Functional Programming in the Real World	81
3.	Collections and Data Structures	83
	Abstractions over Implementations	84
	Collection	87
	Sequences	89
	Associative	99
	Indexed	103
	Stack	104
	Set	105
	Sorted	106
	Concise Collection Access	111
	Idiomatic Usage	112
	Collections and Keys and Higher-Order Functions	113
	Data Structure Types	114
	Lists	114
	Vectors	115
	Sets	117
	Maps	117
	Immutability and Persistence	122
	Persistence and Structural Sharing	123
	Transients	130
	Metadata	134

	Putting Clojure's Collections to Work Identifiers and Cycles Thinking Different: From Imperative to Functional	136 137 138
	Navigation, Update, and Zippers	150
	In Summary	157
4.	Concurrency and Parallelism	159
	Shifting Computation Through Time and Space	160
	Delays	160
	Futures	162
	Promises	163
	Parallelism on the Cheap	166
	State and Identity	168
	Clojure Reference Types	170
	Classifying Concurrent Operations	172
	Atoms	174
	Notifications and Constraints	176
	Watches	176
	Validators	178
	Refs	180
	Software Transactional Memory	180
	The Mechanics of Ref Change	181
	The Sharp Corners of Software Transactional Memory	191
	Vars	198
	Defining Vars	198
	Dynamic Scope	201
	Vars Are Not Variables	206
	Forward Declarations	208
	Agents	209
	Dealing with Errors in Agent Actions	212
	I/O, Transactions, and Nested Sends	214
	Using Java's Concurrency Primitives	224
	Locking	225
	Final Thoughts	226

# Part II. Building Abstractions

5.	Macros	229
	What Is a Macro?	229
	What Macros Are Not	231
	What Can Macros Do that Functions Cannot?	232
	Macros Versus Ruby eval	234

		225
	Writing Your First Macro	235
	Debugging Macros	237
	Macroexpansion	237
	Syntax	239
	quote Versus syntax-quote	240
	unquote and unquote-splicing	241
	When to Use Macros	243
	Hygiene	244
	Gensyms to the Rescue	246
	Letting the User Pick Names	248
	Double Evaluation	249
	Common Macro Idioms and Patterns	250
	The Implicit Arguments: &env and &form	251
	&env	252
	&form	254
	Testing Contextual Macros	258
	In Detail: -> and ->>	259
	Final Thoughts	262
	That Thoughts	202
6.	Datatypes and Protocols	. 263
0.	Protocols	264
	Extending to Existing Types	266
	Defining Your Own Types	200
	Records	270
	Types	277
	Implementing Protocols	280
	Inline Implementation	281
	Reusing Implementations	285
	Protocol Introspection	289
	Protocol Dispatch Edge Cases	290
	Participating in Clojure's Collection Abstractions	292
	Final Thoughts	299
7	Multimathada	201
7.	Multimethods	. 301
	Multimethods Basics	301
	Toward Hierarchies	304
	Hierarchies	306
	Independent Hierarchies	308
	Making It Really Multiple!	311
	A Few More Things	313
	Multiple Inheritance	313
	Introspecting Multimethods	314
	type Versus class; or, the Revenge of the Map	314

The Range of Dispatch Functions Is Unlimited	316
Final Thoughts	317

# Part III. Tools, Platform, and Projects

8.	Organizing and Building Clojure Projects	321
	Project Geography	321
	Defining and Using Namespaces	322
	Location, Location	332
	The Functional Organization of Clojure Codebases	334
	Build	336
	Ahead-of-Time Compilation	337
	Dependency Management	339
	The Maven Dependency Management Model	339
	Build Tools and Configuration Patterns	344
	Final Thoughts	353
9.	Java and JVM Interoperability	355
	The JVM Is Clojure's Foundation	356
	Using Java Classes, Methods, and Fields	357
	Handy Interop Utilities	360
	Exceptions and Error Handling	362
	Escaping Checked Exceptions	364
	with-open, finally's Lament	364
	Type Hinting for Performance	366
	Arrays	370
	Defining Classes and Implementing Interfaces	371
	Instances of Anonymous Classes: proxy	372
	Defining Named Classes	374
	Annotations	381
	Using Clojure from Java	385
	Using deftype and defrecord Classes	388
	Implementing Protocol Interfaces	390
	Collaborating Partners	392
10.	REPL-Oriented Programming	. 393
	Interactive Development	393
	The Persistent, Evolving Environment	397
	Tooling	398
	The Bare REPL	399
	Eclipse	403
	Emacs	405

Debugging, Monitoring, and Patching Production in the REPL	411
Special Considerations for "Deployed" REPLs	414
Limitations to Redefining Constructs	415
In Summary	417

#### Part IV. Practicums

11.	Numerics and Mathematics	421
	Clojure Numerics	421
	Clojure Prefers 64-bit (or Larger) Representations	422
	Clojure Has a Mixed Numerics Model	422
	Rationals	424
	The Rules of Numeric Contagion	425
	Clojure Mathematics	427
	Bounded Versus Arbitrary Precision	428
	Unchecked Ops	430
	Scale and Rounding Modes for Arbitrary-Precision Decimals Ops	432
	Equality and Equivalence	433
	Object Identity (identical?)	433
	Reference Equality (=)	434
	Numeric Equivalence (==)	435
	Optimizing Numeric Performance	436
	Declare Functions to Take and Return Primitives	438
	Use Primitive Arrays Judiciously	442
	Visualizing the Mandelbrot Set in Clojure	449
12.	Design Patterns	457
	Dependency Injection	459
	Strategy Pattern	462
	Chain of Responsibility	463
	Aspect-Oriented Programming	466
	Final Thoughts	470
13.	Testing	471
	Immutable Values and Pure Functions	471
	Mocking	472
	clojure.test	473
	Defining Tests	474
	Test "Suites"	477
	Fixtures	479
	Growing an HTML DSL	481
	Relying upon Assertions	486

Preconditions and Postconditions

14.	Using Relational Databases	491
	clojure.java.jdbc	491
	with-query-results Explained	494
	Transactions	496
	Connection Pooling	496
	Korma	498
	Prelude	498
	Queries	499
	Why Bother with a DSL?	500
	Hibernate	503
	Setup	503
	Persisting Data	506
	Running Queries	506
	Removing Boilerplate	507
	Final Thoughts	509
15.	Using Nonrelational Databases	511
	Getting Set Up with CouchDB and Clutch	512
	Basic CRUD Operations	512
	Views	514
	A Simple (JavaScript) View	514
	Views in Clojure	516
	_changes: Abusing CouchDB as a Message Queue	520
	À la Carte Message Queues	522
	Final Thoughts	525
16.	Clojure and the Web	527
	The "Clojure Stack"	527
	The Foundation: Ring	529
	Requests and Responses	529
	Adapters	531
	Handlers	532
	Middleware	534
	Routing Requests with Compojure	535
	Templating	545
	Enlive: Selector-Based HTML Transformation	546
	Final Thoughts	554
17.	Deploying Clojure Web Applications	557
	Java and Clojure Web Architecture	557
	Web Application Packaging	560

Running Web Apps Locally	565
Web Application Deployment	566
Deploying Clojure Apps to Amazon's Elastic Beanstalk	567
Going Beyond Simple Web Application Deployment	570

#### Part V. Miscellanea

18.	Choosing Clojure Type Definition Forms Wisely	573
19.	Introducing Clojure into Your Workplace	577
	Just the Facts	577
	Emphasize Productivity	579
	Emphasize Community	580
	Be Prudent	582
20.	What's Next?	583
	(dissoc Clojure 'JVM)	583
	ClojureCLR	583
	ClojureScript	584
	4Clojure	584
	Overtone	585
	core.logic	585
	Pallet	586
	Avout	587
	Clojure on Heroku	587
Index		589

# CHAPTER 1 Down the Rabbit Hole

If you're reading this book, you are presumably open to learning new programming languages. On the other hand, we assume that you expect reciprocity for the time and effort you'll expend to learn a new language, some tangible benefits that can make you more productive, your team more effective, and your organization more flexible.

We believe that you will find this virtuous cycle in effect as you learn, apply, and leverage Clojure. As we are fond of saying, *Clojure demands that you raise your game, and pays you back for doing so*.

As software developers, we often build up a complex and sometimes very personal relationship with our tools and languages. Deciding which raw materials to use is sometimes dominated by pragmatic and legacy concerns. However, all other things being equal, programmers prefer using whatever maximally enhances their productivity and hopefully enables us to fulfill our potential to build useful, elegant systems. As the old saying goes, we want whatever makes the easy stuff easy, and the hard stuff possible.

# Why Clojure?

Clojure is a programming language that lives up to that standard. Forged of a unique blend of the best features of a number of different programming languages—including various Lisp implementations, Ruby, Python, Java, Haskell, and others—Clojure provides a set of capabilities suited to address many of the most frustrating problems programmers struggle with today and those we can see barreling toward us over the horizon. And, far from requiring a sea-change to a new or unfamiliar architecture and runtime (typical of many otherwise promising languages over the years), Clojure is hosted on the Java Virtual Machine, a fact that puts to bed many of the most pressing pragmatic and legacy concerns raised when a new language is considered.

To whet your appetite, let's enumerate some of Clojure's marquee features and characteristics:

#### Clojure is hosted on the JVM

Clojure code can use any Java library, Clojure libraries can in turn be used from Java, and Clojure applications can be packaged just like any Java application and deployed anywhere other Java applications can be deployed: to web application servers; to desktops with Swing, SWT, or command-line interfaces; and so on. This also means that Clojure's runtime is Java's runtime, one of the most efficient and operationally reliable in the world.

#### Clojure is a Lisp

Unlike Java, Python, Ruby, C++, and other members of the Algol family of programming languages, Clojure is part of the Lisp family. However, forget everything you know (or might have heard rumored) about Lisps: Clojure retains the best of Lisp heritage, but is unburdened by the shortcomings and sometimes anachronistic aspects of many other Lisp implementations. Also, being a Lisp, Clojure has *macros*, an approach to metaprogramming and syntactic extension that has been the benchmark against which other such systems have been measured for decades.

Clojure is a functional programming language

Clojure encourages the use of first-class and higher-order functions with values and comes with its own set of efficient immutable data structures. The focus on a strong flavor of functional programming encourages the elimination of common bugs and faults due to the use of unconstrained mutable state and enables Clojure's solutions for concurrency and parallelization.

Clojure offers innovative solutions to the challenges inherent in concurrency and parallelization

The realities of multicore, multi-CPU, and distributed computing demand that we use languages and libraries that have been designed with these contexts in mind. Clojure's *reference types* enforce a clean separation of *state* and *identity*, providing defined concurrency semantics that are to manual locking and threading strategies what garbage collection is to manual memory management.

Clojure is a dynamic programming language

Clojure is dynamically and strongly typed (and therefore similar to Python and Ruby), yet function calls are compiled down to (fast!) Java method invocations. Clojure is also dynamic in the sense that it deeply supports updating and loading new code at runtime, either locally or remotely. This is particularly useful for enabling interactive development and debugging or even instrumenting and patching remote applications without downtime.

Of course, we don't expect you to understand all of that, but we do hope the gestalt sounds compelling. If so, press on. By the end of this chapter, you'll be able to write simple programs in Clojure, and be well on your way to understanding and leveraging it to help realize your potential.

# **Obtaining Clojure**

You'll need two things to work with the code in this chapter and otherwise explore Clojure on your own:

- 1. The Java runtime. You can download the Oracle JVM for free for Windows and Linux (*http://java.com/en/download/*); it is bundled with or automatically installed by all versions of Mac OS X. Clojure requires Java v1.5 or higher; the latest releases of v1.6 or v1.7 are preferable.
- 2. Clojure itself, available from clojure.org (*http://clojure.org/downloads*). All of the code in this book requires v1.3.0 or higher, and has been tested against v1.4.0 as well.<sup>1</sup> Within the zip file you download, you'll find a file named something like *clojure-1.4.0.jar*; this is all you'll need to get started.



There are a number of different Clojure plug-ins for popular development environments like Eclipse and Emacs; see "Tooling" on page 398 for an overview of Clojure tooling. While Clojure's command-line REPL is sufficient for your first few steps in understanding Clojure, we encourage you to use your favorite text editor or IDE if it has quality Clojure support, or to pick up one that does.

If you don't yet want to commit to a particular editor or IDE for Clojure development, you should at least use Leiningen, the most popular project management tool for Clojure. It will download Clojure for you, give you a better REPL than Clojure's default, and you'll likely be using it on a daily basis for your own projects in short order anyway. See "Leiningen" on page 347 for an introduction to it.

If you want to avoid downloading anything right now, you can run many of the samples in this book in the online, in-browser Clojure implementation available at *http://tryclj.com*.

# The Clojure REPL

Many languages have REPLs, often also referred to as interpreters: Ruby has *irb*; Python has its command-line interpreter; Groovy has its console; even Java has something akin to a REPL in BeanShell. The "REPL" acronym is derived from a simple description of what it does:

- 1. Read: code is read as text from some input (often stdin, but this varies if you're using a REPL in an IDE or other nonconsole environment).
- 2. Eval: the code is evaluated, yielding some value.
- 1. Given Clojure's history with regard to backwards compatibility, the code and concepts in this book should remain applicable to future versions of Clojure as well.

- 3. Print: the value is printed to some output device (often stdout, sometimes preceded by other output if the code in question happened to print content itself).
- 4. Loop: control returns to the *read* step.

Clojure has a REPL too, but it differs from many other languages' REPLs in that it is not an interpreter or otherwise using a limited or lightweight subset of Clojure: all code entered into a Clojure REPL is *compiled* to JVM bytecode as part of its evaluation, with the same result as when code is loaded from a Clojure source file. In these two scenarios, compilation is performed entirely at runtime, and requires no separate "compile" step.<sup>2</sup> In fact, Clojure is *never* interpreted. This has a couple of implications:

- 1. Operations performed in the REPL run at "full speed"; that is to say, there is no runtime penalty or difference in semantics associated with running code in the REPL versus running the same code as part of a "proper" application.
- 2. Once you understand how Clojure's REPL works (in particular, its read and eval phases), you'll understand how Clojure itself works at the most fundamental level.

With this second point in mind, let's dig into the Clojure REPL and see if we can find bedrock.



The optimal workflow for programming in Clojure makes much more use of the REPL than is typical in other languages to make the development process as interactive as possible. Taking advantage of this is a significant source of the enhanced productivity—and really, fun!—that Clojure enables. We talk about this extensively in Chapter 10.

Example 1-1. Starting a Clojure REPL on the command line

% java -cp clojure-1.4.0.jar clojure.main Clojure 1.4.0 user=>

This incantation starts a new JVM process, with a *classpath* that includes the *clojure.jar* file in the current directory, running the *clojure.main* class as its main entry point.<sup>3</sup> See "A classpath primer" on page 331 if you don't yet know what the classpath is; for now, you can just think of the classpath as the JVM's analogue to Python's PYTHONPATH, Ruby's \$:, and your shell's PATH, the set of files and directories from which the JVM will load classes and resources.

When you see the user=> prompt, the REPL is ready for you to enter some Clojure code. The portion of the Clojure REPL prompt preceding => is the name of the *current* 

- 2. If necessary, you can ahead-of-time compile Clojure to Java class files. See "Ahead-of-Time Compilation" on page 337 for details.
- 3. Alternatively, you can use java -jar clojure.jar, but the -cp flag and the clojure.main entry point are both important to know about; we talk about both in Chapter 8.

*namespace*. Namespaces are like modules or packages; we discuss them extensively later in this chapter in "Namespaces" on page 20. Clojure REPL sessions always start in the default user namespace.

Let's look at some real code, a function that calculates the average of some numbers in Java, Ruby, and Python:

Example 1-2. Averaging numbers in Java, Ruby, and Python

```
public static double average (double[] numbers) {
   double sum = 0;
   for (int i = 0; i < numbers.length; i++) {
      sum += numbers[i];
   }
   return sum / numbers.length;
}
def average (numbers)
   numbers.inject(:+) / numbers.length
end
def average (numbers):
     return sum(numbers) / len(numbers)</pre>
```

Here is the Clojure equivalent:

(defn average
 [numbers]
 (/ (apply + numbers) (count numbers))) ③

• defn defines a new function named average in the current namespace.

② The average function takes one argument, referred to within its body as numbers. Note that there is no type declaration; this function will work equally well when provided with any collection or array of numbers of any type.

The body of the average function, which sums the provided numbers with (apply + numbers),<sup>4</sup> divides that sum by the number of numbers provided—obtained with (count numbers)—and returns the result of that division operation.

We can enter that defn expression at the REPL, and then call our function with a vector of numbers, which yields the expected result:

4. Note that + here is not a special language operator, as in most other languages. It is a regular function, no different in type than the one we're defining. apply is also a function, which applies a function it is provided with to a collection of arguments (numbers here); so, (apply + [a b c]) will yield the same value as (+ a b c).

#### A Word about REPL Interaction Styles

From here on, we will adopt a simple convention for listings that show REPL interactions so you can identify the different types of REPL output. The return value of evaluated expressions will be printed with a ;= prefix:

```
(average [60 80 100 400])
;= 160
```

And content that is written to **stdout** by an expression—aside from what the REPL prints for the expression's return value—will be shown with a single semicolon prefix:

```
(println (average [60 80 100 400]))
; 160
;= nil
```

There are the two differently prefixed lines of REPL output because println returns nil after printing the provided value(s) to stdout.

Lines prefixed with semicolons are comments in Clojure, so you can copy and paste these interactions into your REPL with relative abandon. We'll not include the name space=> prompt in listings, as they are not valid Clojure code and will cause an error if they are accidentally pasted into a REPL.

# No, Parentheses Actually Won't Make You Go Blind

Many programmers who don't already use a Lisp or secretly harbor fond memories of their last usage of Lisp from university blanch at the sight of Lisp syntax. Typical reasons offered for this reaction include:

- 1. The particular usage of parentheses to delimit scope, rather than the more familiar braces {...} or do ... end blocks
- 2. The use of prefix notation indicating the operation being performed; e.g., (+ 1
  2) rather than the familiar infix 1 + 2

These objections are born first out of simple unfamiliarity. The braces that Java (and C and C++ and C# and PHP and...) uses for delimiting scope seem perfectly fine why bother with what appears to be an ill-conceived animal? Similarly, we've all known and used infix notation for mathematics since early childhood—why work to use an unusual notation when what we've been using seems to have been so reliable? We are creatures of habit, and outside of building an understanding of why any particular difference may be significant, we understandably prefer the familiar and reliable.

In both cases, the answer is that Clojure did not import its syntactic foundations from other Lisp implementations on a whim; their adoption carries powerful benefits that are worth a minor shift in perspective:

- Prefixed operations used uniformly simplify the language's syntax significantly and eliminate potential ambiguity from nontrivial expressions.
- The use of parentheses (as a textual representation of lists) is an outgrowth of Clojure being a *homoiconic* language. We'll see what this means in "Homoiconicity" on page 9, but the ramifications of it are manifold: homoiconicity enables the development and use of metaprogramming and domain-specific language constructs simply unavailable in any programming language that is not homoiconic.

After getting through an initial period of unfamiliarity, you will very likely find that Clojure's syntax reduces the cognitive load necessary to read and write code. Quick: is << (bit-shift left) in Java executed before or after & (bitwise and) in order of operations? Every time a programmer has to pause and think about this (or look it up in a manual), every time a programmer has to go back and add grouping parentheses "just in case," a mental page fault has occurred. And, every time a programmer forgets to think about this, a potential error has entered his code. Imagine a language with no order of operations to worry about at all; Clojure is that language.

You might be saying, "But there are so many parentheses!" Actually, there aren't.

In places where it makes sense, Clojure has borrowed a lot of syntax from other languages—like Ruby—for its data literals. Where other Lisps you might have seen use parenthesized lists *everywhere*, Clojure provides a rich set of literals for data and collections like vectors, maps, sets, and lists, as well as things like records (roughly, Clojure's corollary to structs).

If you count and compare the number of delimiting characters and tokens of all kinds ((), [], {}, Ruby's || and end, and so on) in Clojure, Java, Ruby, and Python codebases of similar sizes, you will find that the Clojure code won't have appreciably more than the others—and will often have many fewer thanks to its concision.

# Expressions, Operators, Syntax, and Precedence

All Clojure code is made up of expressions, each of which evaluates to a single value. This is in contrast to many languages that rely upon valueless statements—such as if, for, and continue—to control program flow imperatively. Clojure's corollaries to these statements are all expressions that evaluate to a value.

You've already seen a few examples of expressions in Clojure:

- 60
- [60 80 100 400]
- (average [60 80 100 400])
- (+ 1 2)

These expressions all evaluate to a single value. The rules for that evaluation are extraordinarily simple compared to other languages:

- 1. Lists (denoted by parentheses) are calls, where the first value in the list is the operator and the rest of the values are parameters. The first element in a list is often referred to as being in *function position* (as that's where one provides the function or symbol naming the function to be called). Call expressions evaluate to the value returned by the call.
- 2. Symbols (such as **average** or +) evaluate to the named value in the current scope which can be a function, a named local like **numbers** in our **average** function, a Java class, a macro, or a special form. We'll learn about macros and special forms in a little bit; for now, just think of them as functions.
- 3. All other expressions evaluate to the literal values they describe.



Lists in Lisps are often called *s-expressions* or *sexprs*—short for *symbolic expressions* due to the significance of symbols in identifying the values to be used in calls denoted by such lists. Generally, valid s-expressions that can be successfully evaluated are often referred to as *forms*: e.g., (if condition then else) is an if form, [60 80 100 400] is a vector form. Not all s-expressions are forms: (1 2 3) is a valid s-expression— a list of three integers—but evaluating it will produce an error because the first value in the list is an integer, which is not callable.

The second and third points are roughly equivalent to most other languages (although Clojure's literals are more expressive, as we'll see shortly). However, an examination of how calls work in other languages quickly reveals the complexity of their syntax.

•			•
Clojure expression	Java equivalent	Python equivalent	Ruby equivalent
(not k)	!k	not k	not kor! k
(inc a)	a++,++a,a += 1,a + 1 <sup>a</sup>	a += 1,a + 1	a += 1
(/ (+ x y) 2)	(x + y) / 2	(x + y) / 2	(x + y) / 2
(instance? java.util.List al)	al instanceof java.util.List	isinstance(al, list)	al.is_a? Array
(if (not a) (inc b) (dec b)) <sup>b</sup>	!a ? b + 1 : b - 1	b + 1 if not a else b - 1	!a ? b + 1 : b - 1
(Math/pow 2 10) <sup>c</sup>	Math.pow(2, 10)	pow(2, 10)	2 ** 10
(.someMethod some Obj "foo" (.otherMe thod otherObj 0))	someObj.someMe thod("foo", other Obj.otherMethod(O))	someObj.someMe thod("foo", other Obj.otherMethod(0))	someObj.someMe thod("foo", other Obj.otherMethod(O))
(Math/pow 2 10) <sup>c</sup> (.someMethod some Obj "foo" (.otherMe	<pre>someObj.someMe thod("foo", other</pre>	pow(2, 10) someObj.someMe thod("foo", other	someObj.someMe thod("foo", other

Table 1-1. Comparison of call syntax between Clojure, Java, Python, and Ruby

<sup>a</sup> In-place increment and decrement operations have no direct corollary in Clojure, because unfettered mutability isn't available. See Chapter 2, particularly "On the Importance of Values" on page 52 for a complete discussion of why this is a good thing.

<sup>b</sup> Remember, even forms that influence control flow in Clojure evaluate to values just like any other expression, including if and when. Here, the value of the if expression will be either (inc b) or (dec b), depending on the value of (not a).

<sup>c</sup> Here's your first taste of what it looks like to call Java libraries from Clojure. For details, see Chapter 9.

Notice that call syntax is all over the map (we're picking on Java here the most, but Python and Ruby aren't so different):

- Infix operators are available (e.g., a + 1, al instanceof List), but any nontrivial code ends up having to use often-significant numbers of parentheses to override default precedence rules and make evaluation order explicit.
- Unary operators are seemingly arbitrary in regard to whether they use prefix (e.g., !k and ++a) or postfix position (e.g., a++).
- Static method calls have prefix position, such as Math.pow(2, 10), but...
- Instance method calls use an unusual variety of infix positions, where the target of the method (which will be assigned to **this** within the body of the method being called) is specified first, with the formal parameters to the method coming after the method name.<sup>5</sup>

In contrast, Clojure call expressions follow one simple rule: the first value in a list is the operator, the remainder are parameters to that operator. There are no call expressions that use infix or postfix position, and there are no difficult-to-remember precedence rules. This simplification helps make Clojure's syntax very easy to learn and internalize, and helps make Clojure code very easy to read.

# Homoiconicity

Clojure code is composed of literal representations of its own data structures and atomic values; this characteristic is formally called *homoiconicity*, or more casually, *code-as-data*.<sup>6</sup> This is a significant simplification compared to most other languages, which also happens to enable metaprogramming facilities to a much greater degree than languages that are not homoiconic. To understand why, we'll need to talk some about languages in general and how their code relates to their internal representations.

Recall that a REPL's first stage is to *read* code provided to it by you. Every language has to provide a way to transform that textual representation of code into something that can be compiled and/or evaluated. Most languages do this by parsing that text into an *abstract syntax tree* (AST). This sounds more complicated than it is: an AST is simply a data structure that represents formally what is manifested concretely in text. For example, Figure 1-1 shows some examples of textual language and possible transformations to their corresponding syntax trees.<sup>7</sup>

- 5. Python uses the same sort of infix position for its instance methods, but varies from Algol-family brethren by requiring that methods explicitly name their first parameter, usually self.
- 6. Clojure is by no means the only homoiconic language, nor is homoiconicity a new concept. Other homoiconic languages include all other Lisps, all sorts of machine language (and therefore arguably Assembly language as well), Postscript, XSLT and XQuery, Prolog, R, Factor, Io, and more.
- 7. The natural language parse tree was mostly lifted from *http://en.wikipedia.org/wiki/Parse\_tree*.

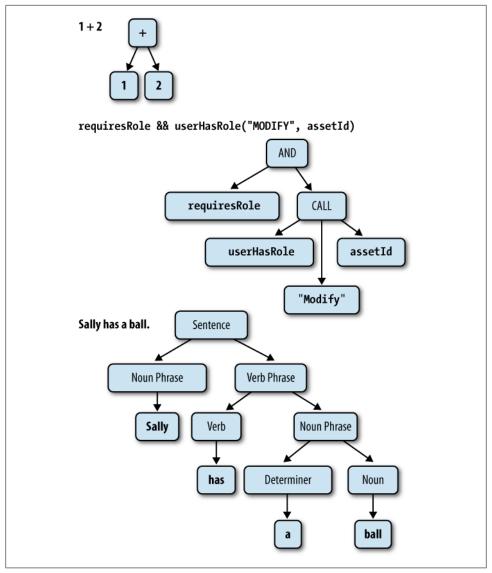


Figure 1-1. Sample transformations from textual language to formal models

These transformations from a textual manifestation of language to an AST are at the heart of how languages are defined, how expressive they are, and how well-suited they are to the purpose of relating to the world within which they are designed to be used. Much of the appeal of domain-specific languages springs from exactly this point: if you have a language that is purpose-built for a given field of use, those that have expertise in that field will find it far easier to define and express what they wish in that language compared to a general-purpose language.

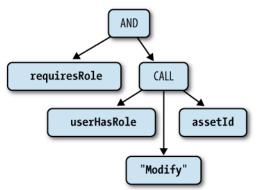
The downside of this approach is that most languages do not provide any way to control their ASTs; the correspondence between their textual syntax and their ASTs is defined solely by the language implementers. This prompts clever programmers to conjure up clever workarounds in order to maximize the expressivity and utility of the textual syntax that they have to work with:

- Code generation
- Textual macros and preprocessors (used to legendary effect by C and C++ programmers for decades now)
- Compiler plug-ins (as in Scala, Project Lombok for Java, Groovy's AST transformations, and Template Haskell)

That's a lot of incidental complexity—complexity introduced solely because language designers often view textual syntax as primary, leaving formal models of it to be implementation-specific (when they're exposed at all).

Clojure (like all Lisps) takes a different path: rather than defining a syntax that will be transformed into an AST, Clojure programs are written using Clojure data structures that represent that AST directly. Consider the **requiresRole...** example from Figure 1-1, and see how a Clojure transliteration of the example *is* an AST for it (recalling the call semantics of function position in Clojure lists).

#### (and requiresRole (userHasRole "MODIFY" assetId))



The fact that *Clojure programs are represented as data* means that Clojure programs can be used to write and transform other Clojure programs, trivially so. This is the basis for macros—Clojure's metaprogramming facility—a far different beast than the gloriously painful hack that are C-style macros and other textual preprocessors, and the ultimate escape hatch when expressivity or domain-specific notation is paramount. We explore Clojure macros in Chapter 5.

In practical terms, the direct correspondence between code and data means that the Clojure code you write in the REPL or in a text source file isn't text at all: you are

programming using Clojure data structure literals. Recall the simple averaging function from Example 1-2:

```
(defn average
[numbers]
 (/ (apply + numbers) (count numbers)))
```

This isn't just a bunch of text that is somehow transformed into a function definition through the operation of a black box; this is a list data structure that contains four values: the symbol defn, the symbol average, a vector data structure containing the symbol numbers, and another list that comprises the function's body. Evaluating that list data structure is what defines the function.

# The Reader

Although Clojure's compilation and evaluation machinery operates exclusively on Clojure data structures, the practice of programming has not yet progressed beyond storing code as plain text. Thus, a way is needed to produce those data structures from textual code. This task falls to the Clojure *reader*.

The operation of the reader is completely defined by a single function, **read**, which reads text content from a character stream<sup>8</sup> and returns the next data structure encoded in the stream's content. This is what the Clojure REPL uses to read text input; each complete data structure read from that input source is then passed on to be evaluated by the Clojure runtime.

More convenient for exploration's sake is **read-string**, a function that does the same thing as **read** but uses a string argument as its content source:

```
(read-string "42")
;= 42
(read-string "(+ 1 2)")
;= (+ 1 2)
```

The operation of the reader is fundamentally one of deserialization. Clojure data structures and other literals have a particular textual representation, which the reader deserializes to the corresponding values and data structures.

You may have noticed that values printed by the Clojure REPL have the same textual representation they do when entered into the REPL: numbers and other atomic literals are printed as you'd expect, lists are delimited by parentheses, vectors by square brackets, and so on. This is because there are duals to the reader's read and read-string functions: pr and pr-str, which prints to \*out\*<sup>9</sup> and returns as a string the readable textual representation of Clojure values, respectively. Thus, Clojure data structures and

<sup>8.</sup> Technically, read requires a java.io.PushbackReader as an implementation detail.

<sup>9. \*</sup>out\* defaults to stdout, but can be redirected easily. See "Building a Primitive Logging System with Composable Higher-Order Functions" on page 72 for an example.

values are trivially serialized and deserialized in a way that is both human- and readerreadable:

```
(pr-str [1 2 3])
;= "[1 2 3]"
(read-string "[1 2 3]")
;= [1 2 3]
```



It is common for Clojure applications to use the reader as a generalpurpose serialization mechanism where you might otherwise choose XML or java.io.Serializable serialization or pickling or marshaling, especially in cases where human-readable serializations are desirable.

# Scalar Literals

*Scalar literals* are reader syntax for noncollection values. Many of these are bread-andbutter types that you already know intimately from Java or very similar analogues in Ruby, Python, and other languages; others are specific to Clojure and carry new semantics.

#### Strings

Clojure strings are Java Strings (that is, instances of java.lang.String), and are represented in exactly the same way, delimited by double quotes:

```
"hello there"
;= "hello there"
```

Clojure's strings are naturally multiline-capable, without any special syntax (as in, for example, Python):

```
"multiline strings
are very handy"
;= "multiline strings\nare very handy"
```

#### Booleans

The tokens true and false are used to denote literal Boolean values in Clojure, just as in Java, Ruby, and Python (modulo the latter's capitalization).

#### nil

nil in Clojure corresponds to null in Java, nil in Ruby, and None in Python. nil is also logically false in Clojure conditionals, as it is in Ruby and Python.

#### Characters

Character literals are denoted by a backslash:

(class \c)
;= java.lang.Character

Both Unicode and octal representations of characters may be used with corresponding prefixes:

```
\u00ff
;= \ÿ
\o41
;= \!
```

Additionally, there are a number of special named character literals for cases where the character in question is commonly used but prints as whitespace:

- \space
- \newline
- \formfeed
- \return
- \backspace
- \tab

#### Keywords

Keywords evaluate to themselves, and are often used as accessors for the values they name in Clojure collections and types, such as hash maps and records:

Here we create a hashmap with two slots, :name and :city, and then look up the value of :city in that map. This works because keywords are functions that look themselves up in collections passed to them.

Syntactically, keywords are always prefixed with a colon, and can otherwise consist of any nonwhitespace character. A slash character (/) denotes a *namespaced keyword*, while a keyword prefixed with two colons (::) is expanded by the reader to a namespaced keyword in the current namespace—or another namespace if the keyword started by a namespace alias, ::alias/kw for example. These have similar usage and motivation as namespaced entities in XML; that is, being able to use the same name for values with different semantics or roles:<sup>10</sup>

```
(def pizza {:name "Ramunto's"
                                :location "Claremont, NH"
                          ::location "43.3734,-72.3365"})
;= #'user/pizza
pizza
;= {:name "Ramunto's", :location "Claremont, NH", :user/location "43.3734,-72.3365"}
```

10. Namespaced keywords are also used prominently with multimethods and isa? hierarchies, discussed in depth in Chapter 7.

```
(:user/location pizza)
;= "43.3734,-72.3365"
```

This allows different modules in the same application and disparate groups within the same organization to safely lay claim to particular names, without complex domain modeling or conventions like underscored prefixes for conflicting names.

Keywords are one type of "named" values, so called because they have an intrinsic name that is accessible using the name function and an optional namespace accessible using namespace:

```
(name :user/location)
;= "location"
(namespace :user/location)
;= "user"
(namespace :location)
;= nil
```

The other named type of value is the symbol.

#### Symbols

Like keywords, symbols are identifiers, but they evaluate to values in the Clojure runtime they name. These values include those held by vars (which are named storage locations used to hold functions and other values), Java classes, local references, and so on. Thinking back to our original example in Example 1-2:

```
(average [60 80 100 400])
;= 160
```

average here is a symbol, referring to the function held in the var named average.

Symbols must begin with a non-numeric character, and can contain \*, +, !, -, \_, and ? in addition to any alphanumeric characters. Symbols that contain a slash (/) denote a *namespaced symbol* and will evaluate to the named value in the specified namespace. The evaluation of symbols to the entity they name depends upon their context and the namespaces available within that context. We talk about the semantics of namespaces and symbol evaluation extensively in "Namespaces" on page 20.

#### Numbers

Clojure provides a plethora of numeric literals (see Table 1-2). Many of them are pedestrian, but others are rare to find in a general-purpose programming language and can simplify the implementation of certain algorithms—especially in cases where the algorithms are defined in terms of particular numeric representations (octal, binary, rational numbers, and scientific notation).



While the Java runtime defines a particular range of numeric primitives, and Clojure supports interoperability with those primitives, Clojure has a bias toward longs and doubles at the expense of other widths, including bytes, shorts, ints, and floats. This means that these smaller primitives will be produced as needed from literals or runtime values for interop operations (such as calling Java methods), but pure-Clojure operations will default to using the wider numeric representations.

For the vast majority of programming domains, you don't need to worry about this. If you are doing work where mathematical precision and other related topics is important, please refer to Chapter 11 for a comprehensive discussion of Clojure's treatment of operations on primitives and other math topics.

Table 1-2. Clojure numeric literals

Literal syntax	Numeric type		
42,0xff,2r111,040	long (64-bit signed integer)		
3.14,6.0221415e23	double (64-bit IEEE floating point decimal)		
42N	clojure.lang.BigInt (arbitrary-precision integer <sup>a</sup> )		
0.01M	java.math.BigDecimal (arbitrary-precision signed floating point decimal)		
22/7	clojure.lang.Ratio		

<sup>a</sup> clojure.lang.BigInt is automatically coerced to java.math.BigInteger when needed. Again, please see Chapter 11 for the in-depth details of Clojure's treatment of numerics.

Any numeric literal can be negated by prefixing it with a dash (-).

Let's take a quick look at the more interesting numeric literals:

#### Hexadecimal notation

Just as in most languages, Clojure supports typical hexadecimal notation for integer values; 0xff is 255, 0xd055 is 53333, and so on.

#### Octal notation

Literals starting with a zero are interpreted as octal numbers. For example, the octal **040** is **32** in the usual base-10 notation.

#### Flexible numeral bases

You can specify the base of an integer in a prefix BrN, where N is the digits that represent the desired number, and B is the base or radix by which N should be interpreted. So we can use a prefix of 2r for binary integers (2r111 is 7), 16r for hexadecimal (16rff is 255), and so on. This is supported up to base 36.<sup>11</sup>

<sup>11.</sup> The implementation limit of java.math.BigInteger's radix support. Note that even though BigInteger is used for parsing these literals, the concrete type of the number as emitted by the reader is consistent with other Clojure integer literals: either a long or a big integer if the number specified requires arbitrary precision to represent.

Arbitrary-precision numbers

Any numeric literal (except for rational numbers) can be specified as arbitraryprecision by suffixing it appropriately; decimals with an M, integers with an N. Please see "Bounded Versus Arbitrary Precision" on page 428 for a full exploration of why and when this is relevant.

Rational numbers

Clojure directly supports rational numbers, also called *ratios*, as literals in the reader as well as throughout its numeric operators. Rational number literals must always be two integers separated by a slash (/).

For a full discussion of rational numbers in Clojure and how they interact with the rest of Clojure's numerical model, please see "Rationals" on page 424.

#### **Regular expressions**

The Clojure reader treats strings prefixed with a hash character as regular expression (regex) literals:

```
(class #"(p|h)ail")
;= java.util.regex.Pattern
```

This is exactly equivalent to Ruby's /.../ regex syntax, with a minor difference of pattern delimiters. In fact, Ruby and Clojure are *very* similar in their handling of regular expressions:

```
# Ruby
>> "foo bar".match(/(...) (...)/).to_a
["foo bar", "foo", "bar"]
;; Clojure
(re-seq #"(...) (...)" "foo bar")
;= (["foo bar" "foo" "bar"])
```

Clojure's regex syntax does not require escaping of backslashes as required in Java:

(re-seq #"(\d+)-(\d+)" "1-3") ;; would be "(\\d+)-(\\d+)" in Java ;= (["1-3" "1" "3"])

The instances of java.util.regex.Pattern that Clojure regex literals yield are entirely equivalent to those you might create within Java, and therefore use the generally excellent java.util.regex regular expression implementation.<sup>12</sup> Thus, you can use those Pattern instances directly via Clojure's Java interop if you like, though you will likely find Clojure's related utility functions (such as re-seq, re-find, re-matches, and others in the clojure.string namespace) simpler and more pleasant to use.

<sup>12.</sup> See the java.util.regex.Pattern javadoc for a full specification of what forms the Java regular expression implementation supports: *http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html*.

### Comments

There are two comment types that are defined by the reader:

- Single-line comments are indicated by prefixing the comment with a semicolon (;); all content following a semicolon is ignored entirely. These are equivalent to // in Java and JavaScript, and # in Ruby and Python.
- *Form-level* are available using the **#**\_reader macro. This cues the reader to elide the next Clojure *form* following the macro:

(read-string "(+ 1 2 #\_(\* 2 2) 8)") ;= (+ 1 2 8)

What would have been a list with four numbers—(+ 1 2 4 8)—yields a list of only three numbers because the entire multiplication form was ignored due to the #\_ prefix.

Because Clojure code is defined using data structure literals, this comment form can be far more useful in certain cases than purely textual comments that affect lines or character offsets (such as the /\* \*/ multiline comments in Java and JavaScript). For example, consider the time-tested debugging technique of printing to stdout:

```
(defn some-function
 [...arguments...]
 ...code...
 (if ...debug-conditional...
  (println ...debug-info...)
  (println ...more-debug-info...))
 ...code...)
```

Making those println forms functionally disappear is as easy as prefixing the if form with the #\_reader macro and reloading the function definition; whether the form spans one or a hundred lines is irrelevant.



There is only one other way to comment code in Clojure, the **comment** macro:

```
(when true
  (comment (println "hello")))
;= nil
```

**comment** forms can contain any amount of ignored code, but they are not elided from the reader's output in the way that **#**\_ impacts the forms following it. Thus, **comment** forms always evaluate to **nil**. This often is not a problem; but, sometimes it can be inconvenient. Consider a reformulation of our first **#**\_ example:

```
(+ 1 2 (comment (* 2 2)) 8)
;= #<NullPointerException java.lang.NullPointerException>
```

That fails because  ${\tt comment}$  returns  ${\tt nil}, {\tt which}$  is not a valid argument to +.

#### Whitespace and Commas

You may have noticed that there have been no commas between forms, parameters to function calls, elements in data structure literals, and so on:

```
(defn silly-adder
[x y]
(+ x y))
```

This is because whitespace is sufficient to separate values and forms provided to the reader. In addition, *commas are considered whitespace by the reader*. For example, this is functionally equivalent to the snippet above:

```
(defn silly-adder
[x, y]
(+, x, y))
```

And to be slightly pedantic about it:

```
(= [1 2 3] [1, 2, 3])
;= true
```

Whether you use commas or not is entirely a question of personal style and preference. That said, they are generally used only when doing so enhances the *human* readability of the code in question. This is most common in cases where pairs of values are listed, but more than one pair appears per line:<sup>13</sup>

```
(create-user {:name new-username, :email email})
```

### **Collection Literals**

The reader provides syntax for the most commonplace Clojure data structures:

'(a b :name 12.5)	;; list
['a 'b :name 12.5]	;; vector
<pre>{:name "Chas" :age 31}</pre>	;; map
#{1 2 3}	;; set

Since lists are used to denote calls in Clojure, you need to quote (') the list literal in order to prevent the evaluation of the list as a call.

The specifics of these data structures are explored in detail in Chapter 3.

<sup>13.</sup> Questions of style are notoriously difficult to answer in absolutes, but it would be very rare to see more than two or three pairs of values on the same line of text in any map literal, set of keyword arguments, and so on. Further, some forms that expect pairs of values (such as bindings in let) are essentially *always* delimited by linebreaks rather than being situated on the same line.

### **Miscellaneous Reader Sugar**

The reader provides for some additional syntax in certain cases to improve concision or regularity with other aspects of Clojure:

- Evaluation can be suppressed by prefixing a form with a quote character ('); see "Suppressing Evaluation: quote" on page 24.
- Anonymous function literals can be defined very concisely using the #() notation; see "Function literals" on page 40.
- While symbols evaluate to the values held by vars, vars themselves can be referred to by prefixing a symbol with #'; see "Referring to Vars: var" on page 44.
- Instances of reference types can be dereferenced (yielding the value contained within the reference object) by prefixing @ to a symbol naming the instance; see "Clojure Reference Types" on page 170.
- The reader provides three bits of special syntax for macros: `, ~, and ~@. Macros are explored in Chapter 5.
- While there are technically only two Java interop forms, the reader provides some sugar for interop that expands into those two special forms; see "Java Interop: . and new" on page 44.
- All of Clojure's data structures and reference types support *metadata*—small bits of information that can be associated with a value or reference that do not affect things like equality comparisons. While your applications can use metadata for many purposes, metadata is used in Clojure itself where you might otherwise use keywords in other languages (e.g., to indicate that a function is namespace-private, or to indicate the type of a value or return type of a function). The reader allows you to attach metadata to literal values being read using the ^ notation; see "Metadata" on page 134.

# Namespaces

At this point, we should understand much of how the nontrivial parts of the Clojure REPL (and therefore Clojure itself) work:

- Read: the Clojure reader reads the textual representation of code, producing the data structures (e.g., lists, vectors, and so on) and atomic values (e.g., symbols, numbers, strings, etc.) indicated in that code.
- Evaluate: many of the values emitted by the reader evaluate to themselves (including most data structures and scalars like strings and keywords). We explored earlier in "Expressions, Operators, Syntax, and Precedence" on page 7 how lists evaluate to calls to the operator in *function position*.

The only thing left to understand about evaluation now is how symbols are evaluated. So far, we've used them to both name and refer to functions, locals, and so on. Outside

of identifying locals, the semantics of symbol evaluation are tied up with namespaces, Clojure's fundamental unit of code modularity.

All Clojure code is defined and evaluated within a namespace. Namespaces are roughly analogous to modules in Ruby or Python, or packages in Java.<sup>14</sup> Fundamentally, they are dynamic mappings between *symbols* and either *vars* or imported Java classes.

One of Clojure's *reference types*,<sup>15</sup> vars are mutable storage locations that can hold any value. Within the namespace where they are defined, vars are associated with a symbol that other code can use to look up the var, and therefore the value it holds.

Vars are defined in Clojure using the def special form, which only ever acts within the current namespace.<sup>16</sup> Let's define a var now in the user namespace, named x; the name of the var is the symbol that it is keyed under within the current namespace:

```
(def x 1)
;= #'user/x
```

We can access the var's value using that symbol:

```
x
;= 1
```

The symbol x here is *unqualified*, so is resolved within the current namespace. We can also redefine vars; this is critical for supporting interactive development at the REPL:

```
(def x "hello")
;= #'user/x
x
;= "hello"
```



#### Vars are not variables

Vars should only ever be defined in an interactive context—such as a REPL—or within a Clojure source file as a way of defining named functions, other constant values, and the like. In particular, top-level vars (that is, globally accessible vars mapped within namespaces, as defined by def and its variants) should only ever be defined by top-level expressions, never in the bodies of functions in the normal course of operation of a Clojure program.

See "Vars Are Not Variables" on page 206 for further elaboration.

- 14. In fact, namespaces correspond precisely with Java packages when types defined in Clojure are compiled down to Java classes. For example, a Person type defined in the Clojure namespace app.entities will produce a Java class named app.entities.Person. See more about defining types and records in Clojure in Chapter 6.
- 15. See "Clojure Reference Types" on page 170 for a full discussion of Clojure's reference types, all of which contribute different capabilities to its concurrency toolbox.
- 16. Remember that the Clojure REPL session always starts in the default user namespace.

Symbols may also be namespace-qualified, in which case they are resolved within the specified namespace instead of the current one:

```
*ns* 1
;= #<Namespace user>
(ns foo)
;= nil
*ns*
;= #<Namespace foo>
user/x
;= "hello"
x
;= #<CompilerException java.lang.RuntimeException:
;= Unable to resolve symbol: x in this context, compiling:(NO_SOURCE_PATH:0)>
```

• The current namespace is always bound to \*ns\*.

Here we created a new namespace using the ns macro (which has the side effect of switching us to that new namespace in our REPL), and then referred to the value of x in the user namespace by using the namespace-qualified symbol user/x. Since we only just created this new namespace foo, it doesn't have a mapping for the x symbol, so attempting to resolve it fails.



You need to know how to create, define, organize, and manipulate namespaces in order to use Clojure effectively. There is a whole suite of functions for this; please refer to "Defining and Using Namespaces" on page 322 for our guidelines in their use.

We mentioned earlier that namespaces also map between symbols and imported Java classes. All classes in the java.lang package are imported by default into each Clojure namespace, and so can be referred to without package qualification; to refer to unimported classes, a package-qualified symbol must be used. Any symbol that names a class evaluates to that class:

```
String
;= java.lang.String
Integer
;= java.lang.Integer
java.util.List
;= java.util.List
java.net.Socket
;= java.net.Socket
```

In addition, namespaces by default alias all of the vars defined in the primary namespace of Clojure's standard library, clojure.core. For example, there is a filter function defined in clojure.core, which we can access without namespace-qualifying our reference to it:

```
filter
;= #<core$filter clojure.core$filter@7444f787>
```

These are just the barest basics of how Clojure namespaces work; learn more about them and how they should be used to help you structure your projects in "Defining and Using Namespaces" on page 322.

# Symbol Evaluation

With a basic understanding of namespaces under our belt, we can turn again to the example **average** function from Example 1-2 and have a more concrete idea of how it is evaluated:

```
(defn average
  [numbers]
  (/ (apply + numbers) (count numbers)))
```

As we learned in "Homoiconicity" on page 9, this is just a canonical textual representation of a Clojure data structure that itself contains other data. Within the body of this function, there are many symbols, each of which refers to either a var in scope in the current namespace or a local value:

- /, apply, +, and count all evaluate to functions held in vars defined and so named in the clojure.core namespace
- numbers either defines the sole argument to the function (when provided in the argument vector [numbers]),<sup>17</sup> or is used to refer to that argument's value in the body of the function (when used in the (apply + numbers) and (count numbers) expressions).

With this information, and recalling the semantics of lists as calls with the operator in function position, you should have a nearly complete understanding of how calls to this function are evaluated:

(average [60 80 100 400]) ;= 160

The symbol **average** refers here to the value of **#'average**, the var in the current namespace that holds the function we defined. That function is called with a vector of numbers, which is locally bound as **numbers** within the body of the **average** function. The result of the operations in that body produce a value—**160**—which is then returned to the caller: in this case, the REPL, which prints it to **stdout**.

# **Special Forms**

Ignoring Java interoperability for a moment, symbols in function position can evaluate to only two things:

<sup>17.</sup> We'll get into all the details of how to define functions and therefore their arguments in "Creating Functions: fn" on page 36.

- 1. The value of a named var or local, as we've already seen.
- 2. A Clojure special form.<sup>18</sup>

Special forms are Clojure's primitive building blocks of computation, on top of which all the rest of Clojure is built. This foundation shares a lineage with the earliest Lisps, which also defined a limited set of primitives that define the fundamental operations of the runtime, and are taken as sufficient to describe any possible computation.<sup>19</sup> Further, special forms have their own syntax (e.g., many do not take arguments per se) and evaluation semantics.

As you've seen, things that are often described as primitive operations or statements in most languages—including control forms like when and operators like addition and negation—are not primitives in Clojure. Rather, everything that isn't a special form is implemented in Clojure itself by bootstrapping from that limited set of primitive operations.<sup>20</sup> The practical effect of this is that, if Clojure doesn't provide a language construct that you want or need, you can likely build it yourself.<sup>21</sup>

Though all of Clojure is built on top of its special forms, you need to understand what each one does—as you'll use many of them constantly. Let's now discuss each one in turn.

# Suppressing Evaluation: quote

**quote** suppresses evaluation of a Clojure expression. The most obvious impact of this relates to symbols, which, if they name a var, evaluate to that var's value. With **quote**, evaluation is suppressed, so symbols evaluate to themselves (just like strings, numbers, and so on):

```
(quote x)
;= x
(symbol? (quote x))
;= true
```

There is reader syntax for quote; prefixing any form with a quote character (') will expand into a usage of quote:

- 18. Special forms are always given precedence when resolving symbols in function position. For example, you can have a var or local named def, but you will not be able to refer to the value of that var or local in function position—though you can refer to that value anywhere else.
- 19. Paul Graham's *The Roots of Lisp (http://www.paulgraham.com/rootsoflisp.html*) is a brief yet approachable precis of the fundamental operations of computation, as originally discovered and enumerated by John McCarthy. Though that characterization of computation was made more than 50 years ago, you can see it thriving in Clojure today.
- 20. If you were to open the *core.clj* file from Clojure's source repository, you will see this bootstrapping in action: everything from when and or to defn and = is defined in Clojure itself. Indeed, if you were so motivated, you could implement Clojure (or another language of your choosing) from scratch, on your own, on top of Clojure's special forms.
- 21. This sort of syntactic extension generally requires macros, which are treated in detail in Chapter 5.

'x ;= x

Any Clojure form can be quoted, including data structures. Doing so returns the data structure in question, with evaluation recursively suppressed for all of its elements:

```
'(+ x x)
;= (+ x x)
(list? '(+ x x))
;= true
```

While lists are usually evaluated as calls, quoting a list suppresses that evaluation, yielding the list itself; in this case, a list of three symbols: '+, 'x, and 'x. Note that this is exactly what we get if we "manually" construct the list without using a list literal:

```
(list '+ 'x 'x)
;= (+ x x)
```



You can usually have a peek at what the reader produces by quoting a form. Let's go meta for a moment and try it first on quote itself:

```
;= (quote x)
```

It's informative to use this trick on other reader sugars:

```
'@x
;= (clojure.core/deref x)
'#(+ % %)
;= (fn* [p1_3162792#] (+ p1_3162792# p1_3162792#))
'(a b ~c)
;= (seq (concat (list (quote user/a))
;= (list (quote user/b))
;= (list c)))
```

**1 clojure.core** namespace-prefixes elided for legibility.

## Code Blocks: do

**do** evaluates all of the expressions provided to it in order and yields the last expression's value as its value. For example:

```
(do
(println "hi")
(apply * [4 5 6]))
; hi
;= 120
```

The values of all but the last expression are discarded, although their side effects do occur (such as printing to standard out as we're doing here, or manipulations of a stateful object available in the current scope).

Note that many other forms (including fn, let, loop, and try—and any derivative of these, such as defn) wrap their bodies in an implicit do expression, so that multiple inner expressions can be evaluated. For example, let expressions—like this one that defines two locals—provide an implicit do context to their bodies:

```
(let [a (inc (rand-int 6))
      b (inc (rand-int 6))]
(println (format "You rolled a %s and a %s" a b))
(+ a b))
```

This allows any number of expressions to be evaluated within the context of the let form, with only the final one determining its ultimate result. If let didn't wrap its body with a **do** form, you would have to add it explicitly:<sup>22</sup>

```
(let [a (inc (rand-int 6))
      b (inc (rand-int 6))]
(do
    (println (format "You rolled a %s and a %s" a b))
    (+ a b)))
```

## **Defining Vars: def**

We've already seen def in action;<sup>23</sup> it defines (or redefines) a var (with an optional value) within the current namespace:

```
(def p "foo")
;= #'user/p
p
;= "foo"
```

Many other forms implicitly create or redefine vars, and therefore use def internally. It is customary for such forms to be prefixed with "def," such as defn, defn-, defproto col, defonce, defmacro, and so on.



Although forms that create or redefine vars have names that start with "def," unfortunately not all forms that start with "def" create or redefine vars. Examples of the latter include deftype, defrecord, and defmethod.

- 22. The other alternative would be for let (and all other forms that utilize do) to (re?) implement its own semantics of "do several things and return the value of the last expression": hardly a reasonable thing to do.
- 23. See "Namespaces" on page 20 for a discussion of the typical usage of vars as stable references to values in namespaces; see "Vars" on page 198 for more a more comprehensive treatment of them, including esoteric usages related to dynamic scope and thread-local references.

## Local Bindings: let

**let** allows you to define named references that are lexically scoped to the extent of the **let** expression. Said another way, **let** defines locals. For example, this rudimentary static method in Java:

```
public static double hypot (double x, double y) {
   final double x2 = x * x;
   final double y2 = y * y;
   return Math.sqrt(x2 + y2);
}
```

is equivalent to this Clojure function:

```
(defn hypot
[x y]
(let [x2 (* x x)
y2 (* y y)]
(Math/sqrt (+ x2 y2))))
```

The x2 and y2 locals in the respective function/method bodies serve the same purpose: to establish a named, scoped reference to an intermediate value.



There are many terms used to talk about named references established by let in Clojure parlance:

- locals
- local bindings
- particular values are said to be *let-bound*

*Bindings* and *bound* used in connection with let are entirely distinct from the **binding** macro, which controls scoped thread-local variables; see "Dynamic Scope" on page 201 for more about the latter.

Note that let is implicitly used anywhere locals are required. In particular, fn (and therefore all other function-creation and function-definition forms like defn) uses let to bind function parameters as locals within the scope of the function being defined. For example, x and y in the hypot function above are let-bound by defn. So, the vector that defines the set of bindings for a let scope obeys the same semantics whether it is used to define function parameters or an auxiliary local binding scope.



Occasionally, you will want evaluate an expression in the binding vector provided to let, but have no need to refer to its result within the context of the let's body. In these cases, it is customary to use an underscore as the bound name for such values, so that readers of the code will know that results of such expressions are going unused intentionally.

This is only ever relevant when the expression in question is sideeffecting; a common example would be printing some intermediate value:

Here we're retrieving our current latitude and longitude using a hypothetical API, and we'd like to print that out before converting the location data to a human-recognizable city name. We might want to rebind the same name a couple of times in the course of the let's binding vector, paving over those intermediate values. To print out that intermediate value, we add it to the binding vector prior to rebinding its name, but we indicate that we are intentionally ignoring the return value of that expression by naming it \_.

**let** has two particular semantic wrinkles that are very different from locals you may be used to in other languages:

- 1. All locals are *immutable*. You can *override* a local binding within a nested let form or a later binding of the same name within the same binding vector, but there is no way to bash out a bound name and change its value within the scope of a single let form. This eliminates a source of common errors and bugs without sacrificing capability:
  - The loop and recur special forms provide for looping cases where values need to change on each cycle of a loop; see "Looping: loop and recur" on page 43.
  - If you really need a "mutable" local binding, Clojure provides a raft of reference types that enforce specific mutation semantics; see "Clojure Reference Types" on page 170.
- 2. **let**'s binding vector is interpreted at compile time to provide optional *destructuring* of common collection types. Destructuring can aid substantially in eliminating certain types of verbose (and frankly, dull) code often associated with working with collections provided as arguments to functions.

### Destructuring (let, Part 2)

A lot of Clojure programming involves working with various implementations of data structure abstractions, *sequential* and *map* collections being two of those key

abstractions. Many Clojure functions accept and return seqs and maps generallyrather than specific implementations—and most Clojure libraries and applications are built up relying upon these abstractions instead of particular concrete structures, classes, and so on. This allows functions and libraries to be trivially composed around the data being handled with a minimum of integration, "glue code," and other incidental complexity.

One challenge when working with abstract collections is being able to concisely access multiple values in those collections. For example, here's a collection, a Clojure vector:

```
(def v [42 "foo" 99.2 [5 12]])
;= #'user/v
```

Consider a couple of approaches for accessing the values in our sample vector:

```
0
(first v)
;= 42
(second v)
;= "foo"
(last v)
;= [5 12]
             0
(nth v 2)
;= 99.2
(v 2)
             Ø
;= 99.2
             4
(.get v 2)
;= 99.2
```

• Clojure provides convenience functions for accessing the first, second, and last values from a sequential collection.

**2** The nth function allows you pluck any value from a sequential collection using an index into that collection.

**3** Vectors are functions of their indices.

• All of Clojure's sequential collections implement the java.util.List interface, so you can use that interface's .get method to access their contents.

All of these are perfectly fine ways to access a single "top-level" value in a vector, but things start getting more complex if we need to access multiple values to perform some operation:

```
(+ (first v) (v 2))
;= 141.2
```

Or if we need to access values in nested collections:

(+ (first v) (first (last v))) ;= 47

Clojure destructuring provides a concise syntax for declaratively pulling apart collections and binding values contained therein as named locals within a let form. And, because destructuring is a facility provided by let, it can be used in any expression that implicitly uses let (like fn, defn, loop, and so on).

There are two flavors of destructuring: one that operates over sequential collections, and another that works with maps.

#### Sequential destructuring

Sequential destructuring works with any sequential collection, including:

- Clojure lists, vectors, and seqs
- Any collection that implements java.util.List (like ArrayLists and LinkedLists)
- Java arrays
- Strings, which are destructured into their characters

Here's a basic example, where we are destructuring the same value  $\boldsymbol{v}$  discussed above:

Example 1-3. Basic sequential destructuring

```
(def v [42 "foo" 99.2 [5 12]])
;= #'user/v
(let [[x y z] v]
  (+ x z))
;= 141.2
```

In its simplest form, the vector provided to let contains pairs of names and values, but here we're providing a vector of symbols— $[x \ y \ z]$ —instead of a scalar symbol name. What this does is cause the value v to be destructured sequentially, with the first value bound to x within the body of the let form, the second value bound to y, and the third value bound to z. We can then use those destructured locals like any other locals. This is equivalent to:

```
(let [x (nth v 0)
        y (nth v 1)
        z (nth v 2)]
   (+ x z))
;= 141.2
```



Python has something similar to Clojure's sequential destructuring, called *unpacking*. The equivalent to the preceding code snippet in Python would be something like:

```
>>> v = [42, "foo", 99.2, [5, 12]]
>>> x, y, z, a = v
>>> x + z
141.1999999999999
```

The same goes for Ruby:

>> x, y, z, a = [42, "foo", 99.2, [5, 12]] [42, "foo", 99.2, [5, 12]] >> x + z 141.2

Clojure, Python, and Ruby all seem pretty similar on their face; but, as you'll see as we go along, Clojure goes quite a long ways beyond what Python and Ruby offer.

Destructuring forms are intended to mirror the structure of the collection that is being bound.<sup>24</sup> So, we can line up our destructuring form with the collection being destructured and get a very accurate notion of which values are going to be bound to which names:<sup>25</sup>

[x y z] [42 "foo" 99.2 [5 12]]

Destructuring forms can be composed as well, so we can dig into the nested vector in  ${\bf v}$  with ease:  $^{26}$ 

```
(let [[x _ [y z]] v]
(+ x y z))
;= 59
```

If we visually line up our destructuring form and the source vector again, the work being done by that form should again be very clear:

```
[x [y z ]]
[42 "foo" 99.2 [5 12]]
```



If our nested vector had a vector inside of it, we could destructure it as well. The destructuring mechanism has no limit to how far it can descend into a deeply nested data structure, but there are limits to good taste. If you're using destructuring to pull values out of a collection four or more levels down, chances are your destructuring form will be difficult to interpret for the next person to see that code—even if that next person is you!

There are two additional features of sequential destructuring forms you should know about:

Gathering extra-positional sequential values

You can use & to gather values that lay beyond the positions you've named in your destructuring form into a sequence; this is similar to the mechanism underlying varargs in Java methods and is the basis of **rest** arguments in Clojure functions:

- 24. Thus the term: destructuring is undoing (*de-*) the creation of the data *structure*.
- 25. Values in the source collection that have no corresponding bound name are simply not bound within the context of the let form; you do not need to fully match the structure of the source collection, but sequential destructuring forms do need to be "anchored" at the beginning of the source.
- 26. Again, note the use of underscores (\_) in this destructuring form to indicate an ignored binding, similar to the idiom discussed in the note earlier in this chapter.

(let [[x & rest] v] rest) ;= ("foo" 99.2 [5 12])

This is particularly useful when processing items from a sequence, either via recursive function calls or in conjunction with a **loop** form. Notice that the value of **rest** here is a sequence, and *not* a vector, even though we provided a vector to the destructuring form.

Retaining the destructured value

You can establish a local binding for the original collection being destructured by specifying the name it should have via the **:as** option within the destructuring form:

```
(let [[x _ z :as original-vector] v]
  (conj original-vector (+ x z)))
;= [42 "foo" 99.2 [5 12] 141.2]
```

Here, **original-vector** is bound to the unchanged value of v. This comes in handy when you are destructuring a collection that is the result of a function call, but you need to retain a reference to that unaltered result in addition to having the benefit of destructuring it. Without this feature, doing so would require something like this:

```
(let [some-collection (some-function ...)
     [x y z [a b]] some-collection]
    ...do something with some-collection and its values...)
```

#### Map destructuring

Map destructuring is conceptually identical to sequential destructuring—we aim to mirror the structure of the collection being bound. It works with:

- Clojure hash-maps, array-maps, and records<sup>27</sup>
- Any collection that implements java.util.Map
- Any value that is supported by the get function can be map-destructured, using indices as keys:
  - -Clojure vectors

—Strings

—Arrays

Let's start with a Clojure map and a basic destructuring of it:

```
(def m {:a 5 :b 6
        :c [7 8 9]
        :d {:e 10 :f 11}
        "foo" 88
        42 false})
;= #'user/m
```

27. See "Records" on page 272 to learn more about records.

```
(let [{a :a b :b} m]
  (+ a b))
;= 11
```

Here we're binding the value for :a in the map to a, and the value for :b in the map to b. Going back to our visual alignment of the destructuring form with the (in this case, partial) collection being destructured, we can again see the structural correspondence:

```
{a :a b :b}
{:a 5 :b 6}
```

Note that there is no requirement that the keys used for map lookups in destructuring be keywords; any type of value may be used for lookup:

```
(let [{f "foo"} m]
  (+ f 12))
;= 100
(let [{v 42} m]
  (if v 1 0))
;= 0
```

Indices into vectors, strings, and arrays can be used as keys in a map destructuring form.<sup>28</sup> One place where this can be helpful is if you are representing matrices by using vectors, but only need a couple of values from one. Using map destructuring to pull out two or three values from a 3×3 matrix can be much easier than using a potentially nine-element sequential destructuring form:

```
(let [{x 3 y 8} [12 0 0 -18 44 6 0 0 1]]
(+ x y))
;= -17
```

Just as sequential destructuring forms could be composed, so can the map variety:

```
(let [{{e :e} :d} m]
(* 2 e))
;= 20
```

The outer map destructuring—{ $\{e : e\} : d\}$ —is acting upon the top-level source collection m to pull out the value mapped to :d. The inner map destructuring—{ $e : e\}$ —is acting on the value mapped to :d to pull out its value for :e.

The *coup de grâce* is the composition of both map and sequential destructuring, however they are needed to effectively extract the values you need from the collections at hand:

```
(let [{[x _ y] :c} m]
  (+ x y))
;= 16
(def map-in-vector ["James" {:birthday (java.util.Date. 73 1 6)}])
;= #'user/map-in-vector
(let [[name {bd :birthday}] map-in-vector]
```

28. This is due to the polymorphic behavior of get, which looks up values in a collection given a key into that collection; in the case of these indexable sequential values, get uses indices as keys. For more about get, see "Associative" on page 99.

```
(str name " was born on " bd))
;= "James was born on Thu Feb 06 00:00:00 EST 1973"
```

Map destructuring also has some additional features.

**Retaining the destructured value.** Just like sequential destructuring, adding an **:as** pair to the destructuring form to hold a reference to the source collection, which you can use like any other let-bound value:

```
(let [{r1 :x r2 :y :as randoms}
      (zipmap [:x :y :z] (repeatedly (partial rand-int 10)))]
      (assoc randoms :sum (+ r1 r2)))
;= {:sum 17, :z 3, :y 8, :x 9}
```

**Default values.** You can use an **:or** pair to provide a defaults map; if a key specified in the destructuring form is not available in the source collection, then the defaults map will be consulted:

```
(let [{k :unknown x :a
            :or {k 50}} m]
    (+ k x))
;= 55
```

This allows you to avoid either merging the source map into a defaults map ahead of its destructuring, or manually setting defaults on destructured bindings that have nil values in the source collection, which would get *very* tiresome beyond one or two bindings with desired default values:

Furthermore, and unlike the code in the above example, **:or** knows the difference between no value and a false (**nil** or **false**) value:

```
(let [{opt1 :option} {:option false}
        opt1 (or opt1 true)
        {opt2 :option :or {opt2 true}} {:opti opt1 :opt2 opt2})
;= {:opt1 true, :opt2 false}
```

**Binding values to their keys' names.** There are often stable names for various values in maps, and it's often desirable to bind those values by using the same names in the scope of the let form as they are mapped to in the source map. However, doing this using "vanilla" map destructuring can get very repetitive:

```
(def chas {:name "Chas" :age 31 :location "Massachusetts"})
;= #'user/chas
(let [{name :name age :age location :location} chas]
  (format "%s is %s years old and lives in %s." name age location))
;= "Chas is 31 years old and lives in Massachusetts."
```

Having to type the content of each key twice is decidedly contrary to the spirit of destructuring's concision. In such cases, you can use the **:keys**, **:strs**, and **:syms** options to specify keyword, string, and symbol keys (respectively) into the source map and the names the corresponding values should be bound to in the **let** form without repetition. Our sample map uses keywords for keys, so we'll use :keys for it:

```
(let [{:keys [name age location]} chas]
  (format "%s is %s years old and lives in %s." name age location))
;= "Chas is 31 years old and lives in Massachusetts."
```

...and switch to using :strs or :syms when we know that the source collection is using strings or symbols for keys:

```
(def brian {"name" "Brian" "age" 31 "location" "British Columbia"})
;= #'user/brian
(let [{:strs [name age location]} brian]
  (format "%s is %s years old and lives in %s." name age location))
;= "Brian is 31 years old and lives in British Columbia."
(def christophe {'name "Christophe" 'age 33 'location "Rhône-Alpes"})
;= #'user/christophe
(let [{:syms [name age location]} christophe]
  (format "%s is %s years old and lives in %s." name age location))
;= "Christophe is 31 years old and lives in Rhône-Alpes."
```

You will likely find yourself using :keys more than :strs or :syms; keyword keys are by far the most common key type in Clojure maps and keyword arguments, and are the general-purpose accessor by dint of their usage in conjunction with records.

**Destructuring rest sequences as map key/value pairs.** We've already seen how extra-positional values in sequential destructuring forms can be gathered into a "rest" seq, and map and sequential destructuring can be composed as needed to drill into any given data structure. Here's a simple case of a vector that contains some positional values, followed by a set of key/value pairs:

```
(def user-info ["robert8990" 2011 :name "Bob" :city "Boston"])
;= #'user/user-info
```

Data like this isn't uncommon, and handling it is rarely elegant. The "manual" approach in Clojure is tolerable as these things go:

```
(let [[username account-year & extra-info] user-info
                                                        0
     {:keys [name city]} (apply hash-map extra-info)]
                                                        2
  (format "%s is in %s" name city))
:= "Bob is in Boston"
```



• We can destructure the original vector into its positional elements, gathering the remainder into a rest seq.

**2** That rest seq, consisting of alternating keys and values, can be used as the basis for creating a new hashmap, which we can then destructure as we wish.

However, "tolerable" isn't a very high bar given the prevalence of sequences of key/ value pairs in programming. A better alternative is a special variety of the compositional behavior offered by let's destructuring forms: map destructuring of rest seqs. If a rest

seq has an even number of values—semantically, key/value pairs—then it can be destructured as a map of those key/value pairs instead of sequentially:

```
(let [[username account-year & {:keys [name city]}] user-info]
  (format "%s is in %s" name city))
;= "Bob is in Boston"
```

That is a far cleaner notation for doing exactly the same work as us manually building a hash-map out of the rest seq and destructuring that map, and is the basis of Clojure functions' optional keyword arguments described in "Keyword arguments" (page 39).

# **Creating Functions: fn**

Functions are first-class values in Clojure; creating them falls to the fn special form, which also folds in the semantics of let and do.

Here is a simple function that adds **10** to the number provided as an argument:

(fn [x] 0 (+ 10 x)) **2** 



**1** fn accepts a let-style binding vector that defines the names and numbers of arguments accepted by the function; the same optional destructuring forms discussed in "Destructuring (let, Part 2)" on page 28 can be applied to each argument here.

**2** The forms following the binding vector constitute the *body* of the function. This body is placed in an implicit do form, so each function's body may contain any number of forms; as with do, the last form in the body supplies the result of the function call that is returned to the caller.

The arguments to a function are matched to each name or destructuring form based on their positions in the calling form. So in this call:

```
((fn [x] (+ 10 x)) 8)
;= 18
```

8 is the sole argument to the function, and it is bound to the name x within the body of the function. This makes the function call the equivalent of this **let** form:

```
(let [x 8]
 (+ 10 x))
```

You can define functions that accept multiple arguments:

((fn [x y z] (+ x y z)) 3 4 12) ;= 19

In this case, the function call is the equivalent of this let form:

(let [x 3 y 4 z 12] (+ x y z)) Functions with *multiple arities* can be created as well; here, we'll put the function in a **var** so we can call it multiple times by only referring to the **var**'s name:

When defining a function with multiple arities, each arity's binding vector and implementation body must be enclosed within a pair of parentheses. Function calls dispatch based on argument count; the proper arity is selected based on the number of arguments that we provide in our call.

In this last example, notice the optional name that we've given to the function, adderself-reference. This optional first argument to fn can be used within the function's bodies to refer to itself—in this case, so that the single-argument arity can call the twoargument arity with a default second argument without referring to or requiring any containing var.



#### Mutually recursive functions with letfn

Named fns (like the above adder-self-reference) allow you to easily create self-recursive functions. What is more tricky is to create *mutually* recursive functions.

For such rare cases, there is the letfn special form, which allows you to define several named functions at once, and all these functions will know each other. Consider these naive reimplementations of odd? and even?:

• The vector consists of several regular fn bodies, only the fn symbol is missing.

**defn builds on fn.** We've already seen defn used before, and the example above should look familiar; defn is a macro that encapsulates the functionality of def and fn so that you can concisely define functions that are named and registered in the current namespace with a given name. For example, these two definitions are equivalent:

```
([x y] (+ x y))))
(defn strange-adder
  ([x] (strange-adder x 1))
  ([x y] (+ x y))))
```

and single-arity functions can be defined, with the additional parentheses eliminated as well; these two definitions are also equivalent:

We'll largely use defn forms to illustrate fn forms for the rest of this section, simply because calling functions bound to named vars is easier to read than continually defining the functions to be called inline.

#### **Destructuring function arguments**

**defn** supports the destructuring of function arguments thanks to it reusing **let** for binding function arguments for the scope of a function's body. You should refer to the prior comprehensive discussion of destructuring to remind yourself of the full range of options available; here, we'll discuss just a couple of destructuring idioms that are particularly common in conjunction with functions.

**Variadic functions.** Functions can optionally gather all additional arguments used in calls to it into a seq; this uses the same mechanism as sequential destructuring does when gathering additional values into a seq. Such functions are called *variadic*, with the gathered arguments usually called *rest arguments* or *varargs*. Here's a function that accepts one named positional argument, but gathers all additional arguments into a remainder seq:

```
(defn concat-rest
  [x & rest]
  (apply str (butlast rest)))
;= #'user/concat-rest
(concat-rest 0 1 2 3 4)
;= "123"
```

The seq formed for the rest arguments can be destructured just like any other sequence; here we're destructuring rest arguments to make a function behave as if it had an explicitly defined zero-arg arity:

```
(defn make-user
  [& [user-id]]
  {:user-id (or user-id
                     (str (java.util.UUID/randomUUID)))})
;= #'user/make-user
(make-user)
```

```
;= {:user-id "ef165515-6d6f-49d6-bd32-25eeb024d0b4"}
(make-user "Bobby")
;= {:user-id "Bobby"}
```

**Keyword arguments.** It is often the case that you would like to define a function that can accept many arguments, some of which might be optional and some of which might have defaults. Further, you would often like to avoid forcing a particular argument ordering upon callers.<sup>29</sup>

fn (and therefore defn) provides support for such use cases through *keyword arguments*, which is an idiom built on top of the map destructuring of rest sequences that let provides. Keyword arguments are pairs of keywords and values appended to any strictly positional arguments in a function call, and if the function was defined to accept keyword arguments, those keyword/value pairs will be gathered into a map and destructured by the function's map destructuring form that is placed in the same position as the rest arguments seq:

```
(defn make-user
  [username & {:keys [email join-date]
                                                      a
                                                      2
               :or {join-date (java.util.Date.)}}]
  {:username username
   :join-date join-date
   :email email
   ;; 2.592e9 -> one month in ms
   :exp-date (java.util.Date. (long (+ 2.592e9 (.getTime join-date))))})
;= #'user/make-user
(make-user "Bobby")
                                                      0
;= {:username "Bobby", :join-date #<Date Mon Jan 09 16:56:16 EST 2012>,
;= :email nil, :exp-date #<Date Wed Feb 08 16:56:16 EST 2012>}
(make-user "Bobby"
                                                      4
  :join-date (java.util.Date. 111 0 1)
  :email "bobby@example.com")
;= {:username "Bobby", :join-date #<Date Sun Jan 01 00:00:00 EST 2011>,
;= :email "bobby@example.com", :exp-date #<Date Tue Jan 31 00:00:00 EST 2011>}
```

• The make-user function strictly requires only one argument, a username. The rest of the arguments are assumed to be keyword/value pairs, gathered into a map, and then destructured using the map destructuring form following &.

② In the map destructuring form, we define a default of "now" for the join-date value.

Calling make-user with a single argument returns the user map, populated with defaulted join- and expiration-date values and a nil email value since none was provided in the keyword arguments.

• Additional arguments provided to make-user are interpreted by the keyword destructuring map, without consideration of their order.

<sup>29.</sup> Python is a language that supports this usage pervasively, where every argument may be named and provided in any order in a function call, and argument defaults can be provided when a function is defined.



Because keyword arguments are built using let's map destructuring, there's nothing stopping you from destructuring the rest argument map using types of key values besides keywords (such as strings or numbers or even collections). For example:

```
(defn foo
  [& {k ["m" 9]}]
  (inc k))
;= #'user/foo
(foo ["m" 9] 19)
;= 20
```

["m" 9] is being treated here as the name of a "keyword" argument.

That said, we've never actually seen non-keyword key types used in named function arguments. Keywords are overwhelmingly the most common argument key type used, thus the use of *keyword arguments* to describe the idiom.

**Pre- and postconditions.** fn provides support for pre- and postconditions for performing assertions with function arguments and return values. They are valuable features when testing and for generally enforcing function invariants; we discuss them in "Preconditions and Postconditions" on page 487.

#### **Function literals**

We mentioned function literals briefly in "Miscellaneous Reader Sugar" on page 20. Equivalent to blocks in Ruby and lambdas in Python, Clojure function literals' role is straightforward: when you need to define an anonymous function—especially a very simple function—they provide the most concise syntax for doing so.

For example, these anonymous function expressions are equivalent:

```
(fn [x y] (Math/pow x y))
#(Math/pow %1 %2)
```

The latter is simply some reader sugar that is expanded into the former; we can clearly see this by checking the result of reading the textual code:<sup>30</sup>

```
(read-string "#(Math/pow %1 %2)")
;= (fn* [p1_285# p2_286#] (Math/pow p1_285# p2_286#))
```

The differences between the fn form and the shorter function literal are:

**No implicit do form.** "Regular" fn forms (and all of their derivatives) wrap their function bodies in an implicit do form, as we discussed in "Creating Functions: fn" on page 36. This allows you to do things like:

<sup>30.</sup> Since the name of the arguments to the function is irrelevant, the function literal generates a unique symbol for each argument to refer to them; in this case, p1\_285# and p2\_286#.

```
(fn [x y]
 (println (str x \^ y))
 (Math/pow x y))
```

The equivalent function literal requires an explicit **do** form:

Arity and arguments specified using unnamed positional symbols. The fn examples above use the named symbols x and y to specify both the arity of the function being defined, as well as the names of the arguments passed to the function at runtime. In contrast, the literal uses unnamed positional % symbols, where %1 is the first argument, %2 is the second argument, and so on. In addition, the highest positional symbol defines the arity of the function, so if we wanted to define a function that accepted four arguments, we need only to refer to %4 within the function literal's body.

There are two additional wrinkles to defining arguments in function literals:

- 1. Function literals that accept a single argument are so common that you can refer to the first argument to the function by just using %. So, #(Math/pow % %2) is equivalent to #(Math/pow %1 %2). You should prefer the shorter notation in general.
- 2. You can define a variadic function<sup>31</sup> and refer to that function's rest arguments using the **%8** symbol. These functions are therefore equivalent:

```
(fn [x & rest]
  (- x (apply + rest)))
#(- % (apply + %&))
```

Function literals cannot be nested. So, while this is perfectly legal:

```
(fn [x]
(fn [y]
(+ x y)))
```

This is not:

```
#(#(+ % %))
;= #<IllegalStateException java.lang.IllegalStateException:
;= Nested #()s are not allowed>
```

Aside from the fact that the bodies of function literals are intended to be terse, simple expressions, making the prospect of nested function literals a readability and comprehension nightmare, there's simply no way to disambiguate which function's first argument % is referring to.

31. See "Variadic functions" (page 38).

# Conditionals: if

if is Clojure's sole primitive conditional operator. Its syntax is simple: if the value of the first expression in an if form is *logically* true, then the result of the if form is the value of the second expression. Otherwise, the result of the if form is the value of the third expression, if provided. The second and third expressions are only evaluated as necessary.

Clojure conditionals determine logical truth to be anything other than nil or false:

```
(if "hi" \t)
;= \t
(if 42 \t)
;= \t
(if nil "unevaluated" \f)
;= \f
(if false "unevaluated" \f)
;= \f
(if (not true) \t)
;= nil
```

Note that if a conditional expression is logically false, and no **else** expression is provided, the result of an **if** expression is nil.<sup>32</sup>

Many refinements are built on top of if, including:

- when, best used when nil should be returned (or no action should be taken) if a condition is false.
- cond—similar to the else if construction in Java and Ruby, and elif in Python allows you to concisely provide multiple conditions to check, along with multiple then expressions if a given conditional is true.
- if-let and when-let, which are compositions of let with if and when, respectively: if the value of the test expression is logically true, it is bound to a local for the extent of the then expression.



Clojure provides true? and false? predicates, but these are unrelated to if conditionals. For example:

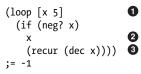
```
(true? "string")
;= false
(if "string" \t \f)
;= \t
```

true? and false? check for the Boolean values true and false, not the logical truth condition used by if, which is equivalent to (or (not (nil? x)) (true? x)) for any value x.

 $\ensuremath{\mathsf{32.}}$  when is far more appropriate for such scenarios.

### Looping: loop and recur

Clojure provides a number of useful imperative looping constructs, including doseq and dotimes, all of which are built upon recur. recur transfers control to the local-most *loop head* without consuming stack space, which is defined either by **loop** or a function. Let's take a look at a very simple countdown loop:





• loop establishes bindings via an implicit let form, so it takes a vector of binding names and initial values.

**2** If the final expression within a **loop** form consists of a value, that is taken as the value of the form itself. Here, when x is negative, the loop form returns the value of x.

• A recur form will transfer control to the local-most loop head, in this case the loop form, resetting the local bindings to the values provided as arguments to recur. In this case, control jumps to the beginning of the loop form, with x bound to the value (dec x).

Loop heads are also established by functions, in which case recur rebinds the function's parameters using the values provided as arguments to recur:

```
(defn countdown
  [x]
  (if (zero? x)
    :blastoff!
    (do (println x)
        (recur (dec x)))))
;= #'user/countdown
(countdown 5)
; 5
; 4
; 3
; 2
; 1
;= :blastoff!
```

**Appropriate use of recur.** recur is a very low-level looping and recursion operation that is usually not necessary:

- When they can do the job, use the higher-level looping and iteration forms found in Clojure's core library, doseq and dotimes.
- When "iterating" over a collection or sequence, functional operations like map, reduce, for, and so on are almost always preferable.

Because **recur** does not consume stack space (thereby avoiding stack overflow errors), recur is critical when implementing certain recursive algorithms. In addition, because it allows you to work with numerics without the overhead of boxed representations, **recur** is very useful in the implementation of many mathematical and data-oriented operations. See "Visualizing the Mandelbrot Set in Clojure" on page 449 for a live example of **recur** within such circumstances.

Finally, there are scenarios where the accumulation or consumption of a collection or set of collections is complicated enough that orchestrating things with a series of purely functional operations using map, reduce, and so on is either difficult or inefficient. In these cases, the use of recur (and sometimes loop in order to set up intermediate loop heads) can provide an important escape hatch.

## **Referring to Vars: var**

Symbols that name a var evaluate to that var's value:

```
(def x 5)
;= #'user/x
x
;= 5
```

However, there are occasions when you'd like to have a reference to the var itself, rather than the value it holds. The **var** special form does this:

```
(var x)
;= #'user/x
```

You've seen a number of times now how vars are printed in the REPL: #', followed by a symbol. This is reader syntax that expands to a call to **var**:

```
#'x
;= #'user/x
```

You'll learn a lot more about vars in "Vars" on page 198.

## Java Interop: . and new

All Java interoperability—instantiation, static and instance method invocation, and field access—flows through the new and . special forms. That said, the Clojure reader provides some syntactic sugar on top of these primitive interop forms that makes Java interop more concise in general and more syntactically consistent with Clojure's notion of function position for method calls and instantiation. Thus, it's rare to see . and new used directly, but you will nevertheless come across them out in the wild at some point:

Operation	Java code	Sugared interop form	Equivalent special form usage
Object instantiation	new java.util.Array	(java.util.ArrayList.	(new java.util.Array
	List(100)	100)	List 100)
Static method invocation	Math.pow(2, 10)	(Math/pow 2 10)	(. Math pow 2 10)
Instance method invocation	"hello".sub	(.substring "hello" 1	(. "hello" substring 1
	string(1, 3)	3)	3)
Static field access	Integer.MAX_VALUE	Integer/MAX_VALUE	(. Integer MAX_VALUE)
Instance field access	someObject.some	(.someField some-	(. some-object some-
	Field	object)	field)

Table 1-3. Sugared Java interop forms and their fully expanded equivalents

The sugared syntax shown in Table 1-3 is idiomatic and should be preferred in every case over direct usage of the . and new special forms. Java interop is discussed in depth in Chapter 9.

### **Exception Handling: try and throw**

These special forms allow you to participate in and use the exception-handling and -throwing mechanisms in Java from Clojure. They are explained in "Exceptions and Error Handling" on page 362.

#### **Specialized Mutation: set!**

While Clojure emphasizes the use of immutable data structures and values, there are contexts where you need to effect an in-place mutation of state. The most common settings for this involve the use of setter and other stateful methods on Java objects you are using in an interop setting; for the remaining cases, Clojure provides set!, which can be used to:

- Set the thread-local value of vars that have a non-root binding, discussed in "Dynamic Scope" on page 201
- Set the value of a Java field, demonstrated in "Accessing object fields" (page 359)
- Set the value of mutable fields defined by deftype; see "Types" on page 277 for details of that usage

#### Primitive Locking: monitor-enter and monitor-exit

These are lock primitives that allow Clojure to synchronize on the monitor associated with every Java object. You should never need to use these special forms, as there's a macro, **locking**, that ensures proper acquisition and release of an object's monitor. See "Locking" on page 225 for details.

# **Putting It All Together**

We've continued to pick at the running example from Example 1-2 throughout our first explorations of Clojure:

```
(defn average
  [numbers]
  (/ (apply + numbers) (count numbers)))
```

We learned how this expression is simply a canonical representation of Clojure data structures in "Homoiconicity" on page 9. In the beginning, in "Expressions, Operators, Syntax, and Precedence" on page 7, we established that lists are evaluated as calls, with the value in function position as the operator. After exploring namespaces, we saw in "Symbol Evaluation" on page 23 how the symbols in that data structure are evaluated at runtime in the course of a call. Now, after we've learned about special forms—in particular, def and fn—we have the final pieces in hand to comprehensively understand what happens when you evaluate this expression (whether at the REPL or as part of loading a Clojure source file from disk in a production application).

defn is simply a shorthand for:

So, fn creates the average function (recall from "Creating Functions: fn" on page 36 that the first argument to fn here, average, is a self-reference, so the function can be called recursively if necessary without looking up the value of the corresponding var again), and def registers it as the value of the average var in the current namespace.

#### eval

All of the evaluation semantics we've been discussing are encapsulated within eval, a function that evaluates a single argument form. We can see very clearly that, for example, scalars and other literals evaluate to the values they describe:

```
(eval :foo)
;= :foo
(eval [1 2 3])
;= [1 2 3]
(eval "text")
;= "text"
```

...and a list will evaluate to the return value of the call it describes:

```
(eval '(average [60 80 100 400]))
;= 160
```



While eval's semantics underly all of Clojure, it is itself very rarely used within Clojure programs. It provides the ultimate in flexibility—allowing you to evaluate any data that represents a valid Clojure expression—that you simply don't need most of the time. In general, if you're using eval in application code, it's likely that you're working with far more rope than you need, and might end up hanging yourself in the process.

Most problems where eval is applicable are better solved through judicious application of macros, which we explore in Chapter 5.

Knowing everything we do now, we can *reimplement* the Clojure REPL quite easily. Remember that **read** (or **read-string**) is used to produce Clojure values from their textual representations:

```
(eval (read-string "(average [60 80 100 400])"))
;= 160
```

...and we can construct a control loop using a **recur** within a function (a **loop** form would work as well). Just a sprinkling of I/O-related functions for printing results and the REPL prompt, and we have a functioning REPL:

Example 1-4. A naive reimplementation of Clojure's REPL

```
(defn embedded-repl
  "A naive Clojure REPL implementation. Enter `:quit`
  to exit."
  []
  (print (str (ns-name *ns*) ">>> "))
  (flush)
  (let [expr (read)
        value (eval expr)]
    (when (not= :quit value)
      (println value)
      (recur))))
(embedded-repl)
; user>>> (defn average2
            [numbers]
            (/ (apply + numbers) (count numbers)))
; #'user/average2
; user>>> (average2 [3 7 5])
; 5
; user>>> :quit
;= nil
```

This REPL implementation is ill-behaved in a variety of ways—for example, any thrown error leaks out of the loop in embedded-repl—but it's a start.<sup>33</sup>

<sup>33.</sup> Clojure's actual REPL is also implemented in Clojure, in the clojure.main namespace, and is waiting for you if you are interested in seeing how the REPL you'll use every day is built.

# **O'Reilly Ebooks**—Your bookshelf on your devices!



When you buy an ebook through <u>oreilly.com</u> you get lifetime access to the book, and whenever possible we provide it to you in five, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, Android .apk, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

### Learn more at ebooks.oreilly.com

You can also purchase O'Reilly ebooks through the iBookstore, the <u>Android Marketplace</u>, and <u>Amazon.com</u>.



Spreading the knowledge of innovators

# This Is Just the Beginning

What we've explored here is the bedrock of Clojure: the fundamental operations of computation (special forms), the interchangeability of code and data, and the tip of the iceberg that is interactive development. On top of this foundation, and in conjunction with the facilities of its JVM host, Clojure provides immutable data structures; concurrency primitives with defined, tractable semantics; macros; and much, much more.

We'll help you understand much of it throughout the rest of the book, and hopefully tie Clojure into your day-to-day life as a programmer with the practicums in Part IV.

There are some key resources you'll may want to keep close at hand along the way:

- The core API documentation, available at *http://clojure.github.com/clojure*
- The main Clojure mailing list, available at *http://groups.google.com/group/clojure*, and the **#clojure** IRC channel on Freenode,<sup>34</sup> both friendly places to get quality help with Clojure, no matter your skill or experience level
- The companion site for this book, *http://clojurebook.com*, which will be maintained over time with additional resources to help you along in learning and using Clojure effectively

Are you ready to take the next step?

<sup>34.</sup> You can use *http://webchat.freenode.net/?channels=#clojure* if you aren't on IRC regularly enough to maintain a desktop client.