

Seven Languages in Seven Weeks

A Pragmatic Guide to Learning Programming Languages





What Readers Are Saying About Seven Languages in Seven Weeks

Knowingmultipleparadigmsgreatlyinfluencesourdesignabilities, so I'malwaysonthelookoutforgoodbooksthat'llhelpmelearnthem. Thisbooknicelybringsprominentparadigmstogether. Brucehas experiencelearning and using multiple languages. Nowyou can gain from his experience through this book. I highly recommendit.

▶ Dr. Venkat Subramaniam

Award-winningauthorandfounder, Agile Developer, Inc.

Asaprogrammer, the importance of being exposed to new programming languages, paradigms, and techniques cannot be overstated. This book does a marvelous job of introducing seven important and diverse languages in a concise—but nontrivial—manner, revealing their strengths and reasons for being. This book is a kintoa dim-sum buffet for any programmer who is interested in exploring new horizons or evaluating emerging languages before committing to studying one in particular.

► Antonio Cangiano

Softwareengineerandtechnicalevangelist, IBM

Fastenyourseatbelts, because you are infora fast-paced journey. This book is packed with programming-language-learning action. Bruceputs it allon the line, and the result is an engaging, rewarding book that passion at eprogrammers will thoroughly enjoy. If you love learning newlanguages, if you want to challenge your mind, if you want to take your programmings kills to the next level—this book is for you. You will not be disappointed.

► Frederic Daoud

Author, Stripes...andJavaWbDevelopmentIsFunAgain GettingStartedwithApacheClick

Doyouwantsevenkickstartsintolearningyour "languageofthe year"? Doyouwantyourthinkingchallengedaboutprogrammingin general? Looknofurtherthanthisbook. I personally was takenback intimetomy under graduate computers ciencedays, coasting through myprogramming languages survey course. The difference is that Brucewon't letyou coast through this course! This is n't aleisurely read—you'll have towork this book. I believe you'll find it both mind-blowing and intensely practical at the same time.

► Matt Stine

Groupleader,ResearchApplicationDevelopmentatSt.Jude Children'sResearchHospital

IspentmostofmytimeasacomputersciencesstudentsayingIdidn't wanttobeasoftwaredeveloperandthenbecameoneanyway. Seven LanguagesinSevenWebs expandedmywayofthinkingaboutproblemsandremindedmewhatIloveaboutprogramming.

► Travis Kaspar

Softwareengineer, Northrop Grumman

Ihavebeenprogrammingfor25yearsinavarietyofhardwareand softwarelanguages. Afterreading SevenLanguagesinSevenWebs amstartingtounderstandhowtoevaluatelanguagesfortheirobjectivestrengthsandweaknesses. Moreimportantly, IfeelasifIcould pickoneofthemtoactually getsomeworkdone.

,I

▶ Chris Kappler

Seniorscientist, Raytheon BBN Technologies

SevenLanguagesinSevenW eeks

APragmaticGuidetoLearning ProgrammingLanguages

BruceA.Tate



Many of the designations used by manufacturers and sellers to distinguish their products are claimed a strademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking device a retrademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatictitles, please visituat http://www.pragprog.com.

Theteamthatproducedthisbookincludes:

Editor: JackieCarter

Indexing: PotomacIndexing,LLC

Copyedit: Kim**Whst**Layout: StevePeter
Production: JanetFurlow
Customersupport: EllieCallahan
International: JulietBenda

Copyright © 2010PragmaticProgrammers,LLC.

Allrightsreserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

PrintedintheUnitedStatesofAmerica.

ISBN-10:1-934356-59-X ISBN-13:978-1-934356-59-3

Printedonacid-freepaper.

P1.0printing,October2010

Vasion 2010-10-28

g

Contents

De	Dedication		
Ac	know	ledgments	11
Fo	Foreword		
1	Intro	oduction	17
	1.1	MethodtotheMadness	17
	1.2	TheLanguages	19
	1.3	BuyThisBook	21
	1.4	Don'tBuyThisBook	22
	1.5	AFinalCharge	24
2	Ruby	7	25
	2.1	QuickHistory	26
	2.2	Day1:FindingaNanny	28
	2.3	Day2:FloatingDownfromtheSky	35
	2.4	Day3:SeriousChange	48
	2.5	WapigUpRuby	56
3	Io		60
	3.1	IntroducingIo	60
	3.2	Day1:SkippingSchool,HangingOut	61
	3.3	Day2:TheSausageKing	74
	3.4	Day3:TheParadeandOtherStrangePlaces	83
	3.5	WapigUpIo	92
4	Prole	og	95
	4.1	AboutProlog	96
	4.2	Day1:AnExcellentDriver	97
	4.3	Day2:FifteenMinutestoWpr	109
	4.4	Day3:BlowingUpVgs	120
	4.5	WanigUpProlog	132

5	Scala		135
	5.1	AboutScala	135
	5.2	Day1:TheCastleontheHill	139
	5.3	Day2:ClippingBushesandOtherNewTricks	153
	5.4	Day3:CuttingThroughtheFluff	167
	5.5	WapigUpScala	176
6	Erlar	ng	181
	6.1	IntroducingErlang	181
	6.2	Day1:AppearingHuman	185
	6.3	Day2:ChangingForms	195
	6.4	Day3:TheRedPill	207
	6.5	WinigUpErlang	219
7	Cloju	ıre	223
	7.1	IntroducingClojure	224
	7.2	Day1:Training Luke	225
	7.3	Day2:YohandtheForce	243
	7.4	Day3:AnEyeforEvil	256
	7.5	WinigUpClojure	264
8	Hask	xell	268
	8.1	IntroducingHaskell	268
	8.2	Day1:Logical	269
	8.3	Day2:Spock'sGreatStrength	285
	8.4	Day3:TheMindMeld	294
	8.5	WapigUpHaskell	309
9	Wrap	o-Up	313
	9.1	ProgrammingModels	313
	9.2	Concurrency	317
	9.3	ProgrammingConstructs	319
	9.4	Finding Yar Vie	321
A	Bibli	ography	322
	Inde	x	323

Dedication

The five months from December 2009 through April 2010 were among the most difficult of mylife. Mybrother, not yet 47 years old, hademergency by pass surgery. No one had any clue that anything was wrong at all. (He came through the surgery without further incident and is doing well.) In late March, my sister was diagnosed with breast cancer. The biggest shock of all came in early March. My mother was diagnosed with terminal cancer. A few short weeks later, she was gone.

As you would expect, I am left to struggle with the grief of a jarring and unexpected loss because of a brutally efficient disease. I would not be human otherwise. But strangely, this experience has not been an entirely negative one. You see, my mother was at peace with the remarkable lifeshelived, herrelationships with her family were strong and fulfilling, and she was exactly where she wanted to be with her faith.

Lynda Lyle Tateputhercreativeenergyintopaintingwithwatercolors. ShesharedherartprimarilythroughherMadisonAvenueArtGallery andherclasses.Beforellefthome,Ihadtheopportunitytotakeafew lessonsfromher.Forsomeonefromatechnicalprofession,theexperiencewasalwaysalittledisorienting.Iwouldvisualizethemasterpiece onmyblankcanvas.Astheactualimagetookshape,itdriftedfurther andfurtherfrommyoriginalvision.WhenIdespairedthatthingswere beyond my ability to fix, Mom looked over my shoulder and told me what she saw. After a few flicks of her talented wrist added darks to accentuated epthandhighlightstoaddclarityanddetail,IwouldrealizethatIhadnotbeentoofarastrayatall.Itjusttookagiftedtouchto bringbackmycreationfromthebrinkofdisaster.Then,Iwouldthrow myexcited armsupinvictoryand telleveryone in the classaboutth is thingIhadcreated,notyetrealizing that each member of the classwas going through their own private burstofjoy.

Afteralittlewhile, Ilearned that Momwasworking on another canvas as well. Through her church and through her profession, she'd find brokenpeople. Encountering alost spouse here or a troubled marriage there, mymother would bring the mintoclass where she would use the paintand paper to slightly open a door that had been slammed shut. As we spent our last week together, person after person would come throughherroomdevastatedatthethoughtoflosingtheirteacher, but Mom would tell the perfect joke or offer the right word of kindness, comforting those who came to comfort her. I got to meet the human canvaseswhohadbeenputrightbythemasterandgoneontodogreat things.Itwasahumblingexperience.

When I told my mother that I would dedicate this book to her, she said that she would like that, but she had nothing to do with computers. That is true enough. The very thought of Wards would leave her helpless. But Mom, you have had everything to do with me.Yar well-timedwordsofencouragementinspiredme, yourlove of creativity shapedme, and your enthusias mandlove of lifeguide meeven now. As Ithink about these experiences, I can't help but feel a little better a superior of the point of the poiand alittlestronger because I, too, a macan vas shaped by the master.

ThisbookisdedicatedwithlovetoLynda Lyle Tate, 1936-2010.

Acknowledgments

ThisisthemostdemandingbookIhaveeverwritten.It's also themost rewarding. The people who have offered to help in various ways have made it so. Thanks first and foremost to my family. Kayla and Julia, your writing a mazes me. You can't yet imagine what you can accomplish. Maggie, you are my joy and inspiration.

In the Ruby community, thanks to Dave Thomas for turning me on to the language that turned my career upside down and helped me havefunagain. Thanks also to Matzfory our friendship and your offer to share your thoughts with my readers. You invited me to Japan to visit the place where Ruby was born, and that experience in spired me much more than you will ever know. To Charles Nutter, Evan Phoenix, and Tim Bray, thanks for the conversations about topics in this book that must have seemed tire some but helped me refine and shape the message.

In the Io community, thanks to Jeremy Tregunna for helping meget pluggedinands haring some coolex amples for the book. Yer eviews were among the best. They were timely and helped build a much stronger chapter. Steve Dekorte, you've created something special, whether or not the market place ever recognizes it as so. The concurrency features rock, and the language has intrinsic beauty. I can definitely appreciate how much of this language feels right. Thanks for helping this neophyte debugh is installation. Thanks also for your thoughtful reviews and your interview that helped me capture the essence of Io. Ye is a pure the essence o

In the Prolog community, thanks to Brian Tarbox for sharing your remarkable experience with myreaders. The dolphin projects, featured on Nova, certainly add a dramatic flair to the Prolog chapter. Special thanks go to Joe Armstrong. You can see how much your feedback shaped the chapter and the overall book. Thanks also for contributing

yourmap-coloringexampleandyourideasforAppend. Theywere the rightexamplesdeliveredattherighttime.

In the Scala community, thanks to my good friend Velkt Subramaniam. Yar Scala book is both rich and understandable. I leaned on it heavily. I greatly appreciate your review and the little bits of help that you offered along the way. Those little bits of your time saved me tremendous anguish and let me focus on the task of teaching. ThanksalsotoMartinOderskyforhelpingthisstrangerbysharingyour thoughts with my readers. Scala takes a unique and brave approach to integrating functional programming paradigms with object-oriented paradigms. Yarefforts are greatly appreciated.

IntheErlangcommunity, IagainthankJoeArmstrong. Yorkindness and energy have helped me form the ideas in this book. Yur tireless promotionofthewaydistributed, fault-tolerantsystems should be built is working. More than any other idea in any other language in this book, Erlang's "Letitcrash" philosophymakes sense tome. I hope to seethoseideasmorebroadlyadopted.

IntheClojurecommunity,thankstoStuartHallowayforyourreviews and ideas that forced me to work harder to bring a better book to insights into Clojure and your instincts helped me my readers. Yar understandwhatwasimportant. Yrbookwasalsohugelyinfluential in the Clojure chapter and actually changed the way I attacked someproblemsinotherchapters as well. Yrapproachinyour consulting practiceisgreatlyappreciated. Yie bringing much-needed simplicity and productivity to this industry. Thanks also to Rich Hickey for your thought full deason the creation of the language and what it means tobe a Lisp dialect. Some ideas in Clojure are intensely radical and yet sopractical. Congratulations. Yie found away to make Lisprevolutionary.Again.

In the Haskell community, thanks to Phillip Walr for the opportunity to look inside the process that created Haskell. We share a passion for teaching, and you're very good at it. Thanks also to Simon Peyton-Jones. I enjoyed working through your interview, the insights youadded, and the unique perspective you brought to these readers.

Thereviewers did an outstanding job with this book. Thanks to VladimirG. Ivanovic, CraigRiecke, Paul Butcher, Fred Daoud, Aaron Bedra,David Eisinger, Antonio Cangiano, and Brian Tarbox. Yu formed the most effective review team I've ever worked with. The book is much

stronger for it. I know that reviewing a book at this level of detail is thankless, demandingwork. Those of us who still like technical books thankyou. The publishing business could not exist without you.

Ialsowanttothankthoseofyouwhosharedvourideasaboutlanguage choiceandprogrammingphilosophy.Atvarioustimes,NealFord,John Heintz, MikePerham, and Ian Washkmadesignificant contributions. These kinds of conversations made melooks marter than Ireally am.

Betareaders, thankyou for reading the book and keeping meworking. **Y**rcommentshaveshownmethatagoodnumberofyouareworking throughthelanguagesratherthancasuallyskimming. I'vechanged the book based on hundreds of comments so far and expect to do even morethroughoutthelifeofthebook.

Finally, to the team at the Pragmatic Bookshelf, I offer my sincerest gratitude.DaveThomasandAndyHunt,youhavehadanincalculable impact on my career as a programmer and again as an author. This publishingplatformhasmadewritingviableagainforme. We cantake abooklikethisonethatmightnotbeasattractivetothemassmarket and make it financially worthwhile. Thanks to all the members of the publishing team. Jackie Carter, your gentle hand and guidance were what this book needed, and I hope you enjoyed our conversations as much as I did. Thanks to those who labored in my shadow to make this book the best it could be. Specifically, I want to thank the team thatlaboredsohardtomakethisbooklookgoodandcorrectallofmy badhabits, including Kim Wingst the copyeditor; Seth Maislin, the indexer; Steve Peter, the typesetter; and Janet Furlow, the producer. Thisbookwouldnotbewhatitiswithoutyou.

As always, mistakes that slipped through this fine team are all mine.ForthoseofyouImissed, Ioffermysincerestapologies. Anyoversight wasnotintentional.

Finally, thanks to all of my readers. It hink that real hard-copy books havevalue, and I can follow mypassion and write because you do, too.

BruceTate

Foreword

From the yet to be written "How Proust Can Make You a Better Programmer"

byJoeArmstrong ,CreatorofErlang

"TheGmaileditorcannotgettypographicquotesright."

"Disgraceful," said Margery, "the sign of an illiterate programmer and a decadent culture."

"Whatshouldwedoaboutit?"

 $\hbox{``We must in sist that the next program mer we hire has read all of Ala} \\ \hbox{'`recherched utemps per du.''}$

"Allsevenvolumes?"

"Allsevenvolumes."

 $\hbox{``Willitmake} them better at punctuation and make them get their quotes \\ \hbox{''ipht?''}$

"Notnecessarily,butitwillmakethemabetterprogrammer.It'saZen thing...."

Learning to program is like learning to swim. No amount of theory is a substitute for diving into the pool and flailing around in the water gasping for air. The first time you sink under the water, you panic, but when you bobt othe surface and gulp in some air, you feelel ated. You think to yourself, "Icanswim." At least that 'show I felt when I learned to swim.

It's the same with programming. The first steps are the most difficult, and you need a good teacher to encourage you to jump into the water.

Bruce Tate is such a teacher. This book gives you the opportunity to start with what is the most difficult part of learning to program, namely, getting started.

Let's assume that you've actually managed the difficult task of downloadingandinstallingtheinterpreterorcompilerforthelanguageyou are interested in. What should you do next? What will be your first program?

Bruce neatly answers this question. Just type in the programs and programfragmentsinthisbooktoseewhetheryoucanreproducehis results.Don'tthinkaboutwritingyourownprogramsyet—justtryto reproduce the examples in the book. As you grow in confidence, you willbeabletotackleyourownprogrammingprojects.

Thefirststepinacquiringanynewskillisnotbeingabletodoyourown thingbutbeingabletoreproducewhatotherpeoplehavedonebefore you. This is the quickest way to mastering askill.

Getting started with programming in a new language is not so much adeepexerciseinunderstandingtheunderlyingprinciplesthatalanguage embodies; it is rather a matter of getting the semicolons and commas in the right places and understanding the weird error messagesthatthesystemspitsoutwhenyoumakeanerror. It is not until vougetbevondthemessybusinessofenteringaprogramandgettingit through the compiler that you can even start to think about the meaningofthedifferentlanguageconstructs.

Onceyou'vegottenthroughthemechanicsofenteringandrunningprograms, you can sit back and relax. Yar subconscious does the rest. Whileyourconscious brainisfiguring outwhere to put the semicolons, your subconscious is figuring out the deep meaning that lies underneath the surface structures. Then you'll wake up one day suddenly understanding the deepermeaning of a logic program or why a particularlanguagehadaparticularconstruct.

Knowingasmallamountaboutmanylanguagesisausefulskill.Ioften findthatIneedtounderstandabitofPythonorRubytosolveaparticular problem. The programs I download from the Internet are oftenwritteninavarietyoflanguagesandneedalittletweakingbeforeIcan usethem.

Each language has its own set of idioms, its strengths, and its weak-ideal and its weaknesses.Bylearningseveraldifferentprogramminglanguages, youwill beabletoseewhichlanguageisbestsuitedtothekindsofproblems thatinterestyoumost.

I'mpleasedtoseethatBruce'stasteinprogramminglanguagesiseclectic.Hecoversnotonlythewell-establishedlanguagessuchasRubybut alsoless-well-appreciatedlanguageslikelo.Ultimately,programmingis aboutunderstanding, and understanding is about ideas. So, exposure to new ideas is essential to a deeper understanding of what programmingisallabout.

A Zen master might tell you that to be better at mathematics you'd betterstudyLatin.Thusitiswithprogramming.Tobetterunderstand the essence of OO programming, you should study logic or functionalprogramming(FP). TobebetteratFP, you should study Assembler.

Books on comparative programming languages were popular when I grewupasaprogrammer, but most of these were academic to mest hat gave little practical guidance to how to actually go about using a language. This reflected the technology of the age. Yu couldreadabout theideasinalanguage, but actually trying it outwas virtually impossible.

Today, not only can we read about the ideas, but we can try them in practice. This makes the difference between standing on the poolside wonderingwhetheritwouldbenicetoswimanddivinginandenjoying thewater.

Iwarmlyrecommendthisbookandhopethatyouenjoyreadingitas muchasIhave.

JoeArmstrong, creator of Erlang 2March2010 Stockholm

Chapter 1 Introduction

Peoplelearnspokenlanguagesfordifferentreasons. Yulearnedyour firstlanguagetolive.Itgaveyouthetoolstogetthroughyoureveryday life.Ifyoulearnedasecondlanguage, thereasons could be very different.Sometimes, you might have to learn a second language to further your career or adapt to a changing environment. But sometimes you decidetoconqueranewlanguagenotbecauseyouhavetobutbecause you want to learn. A second language can help you encounter new worlds. Yumayevenseekenlightenment, knowingeverynewlanguage canshapethewayyouthink.

Soitiswithprogramminglanguages. In this book, I will introduce you tosevendifferentlanguages. Mygoalisnottomakeamotherlydemand likeyourmorningspoonfulofcodliveroil.Iwanttoguideyouthrough ajourneythatwillenlightenyouandchangethewayyoulookatprogramming. I won't make you an expert, but I'll teach you more than "Hello.Wild"

1.1 **MethodtotheMadness**

Most of the time, when I'm learning a new programming language or framework, I'lllook for a quick interactive tutorial. My goalisto experiencethelanguageinacontrolledenvironment.IfIwant,Icangooff scriptandexplore, but I'mbasically looking for a quick jolt of caffeine, asnapshotofsyntacticsugar, and coreconcepts.

But usually, the experience is not fulfilling. If I want to get the true flavorofalanguagethatismorethanasubtleextensionofonelalready know.ashorttutorialis never goingtowork. Ineedadeep, fastdive. This book will give you such an experience not once but seven times. YMfindanswerstothefollowingquestions:

- Typing is strong (Java) or weak (C), • What is the typing model? static(Java)ordynamic(Ruby). The languages in this book lean on the strong typing end of the spectrum, but you'll encounter a broad mix of static and dynamic. You will find how the tradeoffsimpactadeveloper. The typing model will shape the way you attackaproblemandcontrolthewaythelanguageworks. Every languageinthisbookhasitsowntypingidiosyncrasies.
- Whatistheprogramming model? Isitobject-oriented(OO), functional, procedural, or some type of hybrid? This book has languages spanning four different programming models and, sometimes, combinations of more than one. You will find a logic-based programming language (Prolog), two languages with full support forobject-orientedconcepts(Ruby,Scala),fourlanguagesthatare functionalinnature(Scala, Erlang, Clojure, Haskell), and one prototypelanguage(Io). Several of the languages are multiparadigm languages, like Scala. Clojure's multimethods will even let you implementyourownparadigm.Learningnewprogrammingparadigms is one of the most important concepts in this book.
- How will you interact with it? Languages are compiled or interpreted, and some have virtual machines while others don't. In this book, I'll begin to explore with an interactive shell, if there is one.I will move on to files when it's time to attack bigger projects. We won'tattacklargeenoughprojectstofullydiveintopackaging models.
- What are the decision constructs and core data structures? Yidl besurprisedhowmanylanguagescanmakedecisionswiththings otherthanvariations of if sand whiles. Yall see pattern matching $in Erlang and unification in Prolog. Collections play a vital role in {\tt Prolog}. {\tt Collections play} a vital role in {\tt Prolog}. {\tt$ justaboutanylanguage.InlanguagessuchasSmalltalkandLisp, the collections are defining characteristics of the language. In others,likeC++andJava,collectionsareallovertheplace,defining theuser's experience by their absence and lack of cohesion. Either way, you'll need a sound understanding of the collections.
- Whatarethecorefeaturesthatmakethelanguageunique? Some of the languages will support advanced features for concurrent programming. Others provide unique high-level constructs such asClojure'smacrosorIo'smessageinterpretation.Otherswillgive

youasuperchargedvirtualmachine,likeErlang'sBEAM.Because of it, Erlang will let you build fault-tolerant distributed systems much more quickly than you can in other languages. Some languagessupportprogrammingmodelsthatarelaser-focusedona particular problem, such as using logic to solve constraints.

When you're through, you will not be an expert in any of these languages, butyou will knowwhateachuniquely hastooffer. Let's getto thelanguages.

1.2 **TheLanguages**

Choosingthelanguagesinthisbookwasmucheasierthanyoumight imagine. I simply asked potential readers. When we rolled up all the data, we had eight potential candidates. Istruck Java Script because it was too popularandreplaceditwiththenextmostpopular prototype language, Io. Ialsostruck Python because I wanted no more than one object-orientedlanguage, and Rubywashigher on the list. That made roomforasurprisingcandidate, Prolog, which was a top-tenlanguage on the list. These are the languages that did make the cut and the reasonsIpickedthem:

- Ruby. This object-oriented language gets high marks for ease of useandreadability. Ibriefly considered not including any objectoriented language at all, but I found myself wanting to compare the different programming paradigms to object-oriented programming(OOP), so including at least one OOP language was important. I also wanted to push Ruby a little harder than most programmers do and give readers a flavor for the core decisions thatshaped the design of Ruby. I decided to take a dive into Ruby metaprogramming, allowing me to extend the syntax of the language.I'mquitehappywiththeresult.
- Io. Along with Prolog, Io is the most controversial language I included. It is not commercially successful, but the concurrency constructs with the simplicity and uniformity of syntax are importantconcepts. The minimal syntaxis powerful, and the similarities toLisparesometimesstriking.Iohasasmallfootprint,isaprototypelanguagelikeJavaScript,andhasauniquemessagedispatch mechanismthatIthinkyou'llfindinteresting.
- Prolog. YsIknowit'sold, butitisalso extremely powerful. Solving a Sudoku in Prolog was an eye-opening experience for me. I've worked hard to solve some difficult problems in Java or C

- thatwouldhavebeeneffortlessinProlog.JoeArmstrong,creator of Erlang, helped megain a deeper appreciation of this language thatstronglyinfluencedErlang.Ifyou'veneverhadanoccasionto useit, Ithinkyouwillbepleasantlysurprised.
- Scala. One of a new generation of languages on the Java virtualmachine, Scalahas broughtstrong functional concepts to the Javaecosystem.ItalsoembracesOOP.Lookingback,IseeastrikingsimilaritytoC++,whichwasinstrumentaltobridgingproceduralprogrammingandOOP.AsyoudiveintotheScalacommunity, you'll see why Scala represents pure heresy to pure functional programmers and purebliss to Javadevelopers.
- Erlang. Oneoftheoldestlanguagesonthislist, Erlangisgatheringsteamasafunctionallanguagethatgetsconcurrency, distribution, and fault tolerance right. The creators of Couch DB, one of theemergingcloud-baseddatabases, chose Erlangandhaven ever lookedback.Afterspendingalittletimewiththisdistributedlanguage, you'll see why. Erlang makes designing concurrent, distributed, fault-tolerant applications much easier than you could haveeverthoughtpossible.
- · Clojure. Another JVM language, this Lisp-dialect makes some radicalchangesinthewaywethinkaboutconcurrencyontheJVM. It is the only language in this book that uses the same strategy in versioned databases to manage concurrency. As a Lisp dialect, Clojure packs plenty of punch, supporting perhaps the most flexible programming model in the book. But unlike other Lisp dialects, the parentheses are greatly reduced, and you have a huge ecosystem to lean on, including a huge Java library and widelyavailabledeploymentplatforms.
- Haskell. This language is the only pure functional language in the book. That means you won't find mutable state anywhere. The same function with the same input parameters will give you the same output, every time. Of all the strongly typed languages, Haskell supports the most widely respected typing model. Like Prolog, it will take a little while to understand, but the results will beworthit.

I'msorryifyourfavoritelanguagedidn'tmakethelist.Believeme,I've already gotten hate mail from more than a few language enthusiasts. We included several dozen languages in the survey mentioned earlier. Those languages that I picked are not necessarily the best, but each oneisunique, with something important to teach you.

1.3 BuyThisBook

...ifyou are a competent programmer who wants to grow. That claim mightseemalittlenebulous, butindulgeme.

LearningtoLearn

DaveThomasisoneofthefoundersofthispublishingcompany.Hehas challenged thousands of students to learn a newlanguage every year. Atworst, by learning languages, you'll learn to fold new concepts back into the code that you write in your chosen language.

WrgthisbookhashadaprofoundimpactontheRubycodethatI write. It is more functional and is easier to read with less repetition. Ireach for mutable variables less and do a better job with code blocks and higher-order functions. I also uses ometechniques that are unconventionalintheRubycommunity,buttheymakemycodemoreconcise andreadable.

Atbest, you could launch an ewcareer. Every tenyears or so, programming paradigms change. As the Java language became more limiting forme, I experimented with Ruby to better under standits approach to $we b \, development. \, After a couple of successful side projects, I pushed$ mycareerhardinthatdirectionandhaveneverlookedback.MyRuby careerstartedwithbasicexperimentation and grewintomore.

Help&DangerousTimes

Many of the readers of this book won't be old enough to remember thelasttimeourindustryswitchedprogrammingparadigms.Ourshift to object-oriented programming had a couple of false starts, but the oldstructuralprogrammingparadigmwassimplyunabletohandlethe complexity required for modern we bapplications. The successful Javaprogramminglanguagegaveusahardshoveinthatdirection, and the new paradigm stuck. Many developers got caught with old skills and hadtocompletelyretoolthewaytheythought, the tools they used, and thewaytheydesignedapplications.

We may be in the midst of another transformation. This time, new computerdesignswillbethedriver. Five of these venlanguages in this book

havecompellingconcurrencymodels.(RubyandPrologaretheexceptions.)Whetherornotyourprogramminglanguagechangesrightaway, I'mgoingtogooutonalimbandsaythatthelanguagesinthisbook havesomecompellinganswerstooffer. Checkout Io's implementation offutures, Scala's actors, or Erlang's "Letitcrash" philosophy. Understand how Haskell programmers leave mutable state behind or how Clojureusesversioningtosolvesomeofthemostdifficultconcurrency problems.

Yu can also find insight in surprising places. Erlang, the language behind the scenes for several of the cloud-style databases, is a great example. Dr. Joe Armstrong started that language from a Prolog foundation.

1.4 Don'tBuyThisBook

...untilyou've read this section and agree. I am going to make a deal withyou. Yuagreetoletmefocusontheprogramminglanguagerather thaninstallation details. My part of the dealist ot each you more in a shortertime. Whaveto Googlealittlemore, and you can't rely on me to support your installation, but when you're through the book, you'llknowmuchmorebecauseI'llbeabletodivedeeper.

Pleaserecognizethatsevenlanguagesisanambitiousundertakingfor both of us. As a reader, you're going to have to stretch your brain around seven different syntax styles, four programming paradigms, fourdecadesworthoflanguagedevelopment, and more. As an author, I have to cover an enormously broad set of topics for you. I learned several of these languages to support this book. To successfully cover themostimportantdetailsofeachlanguage, Ineedtomakesomesimplifyingassumptions.

IWilltie You BeyondSyntax

Toreally get into the head of a language designer, you're going to have tobewillingtogobeyondthebasicsyntax. That means you'll have to codesomethingmorethanthetypical"Hello,WilforevenaFibonacci series. In Ruby, you will get to do some metaprogramming. In Prolog, you'llsolveafullSudoku.AndinErlang,you'llwriteamonitorthatcan detect the death of another process and launch another one or informtheuser.

These condthat I decided to go deeper than the basics, I made a commitmenttoyouandacompromise. The commitment: Iwon't settle for a superficial treatment. And the compromise: I won't be able to cover some basics that you'd expect to find in dedicated language books. I willrarelygothroughexceptionprocessing, except whereit's a fundamentalfeature of the language. I will not go into packaging models in detailbecausewe'llbedealingwithsmallprojectsthatdonotrequire them.Iwillnotgooverprimitivesthatwedon'tneedtosolvethebasic problemsIlayoutforyou.

IWbtBeY our InstallationGuide

Oneofmybiggestchallengesistheplatform. Ihavehaddirect contact from readers of various books using three different Wats splatforms, OSX, and at least five different Unixversions. I've seen comments on variousmessageboardsofmanymore. Sevenlanguagesons even platformsisaninsurmountabletopicforasingleauthorandprobablyfor amultiauthorbook.Ican'tsupportinstallationforsevenlanguages,so I'mnotgoingtotry.

I suspect that you're not remotely interested in reading another outdatedinstallationguide.Languagesandplatformschange.I'lltellyou where to go to install the language, and I'll tell you what version I'm using. That way, you'll be working from up-to-date instructions from the same list as everyone else. I cannot support your installations.

IWbtBeY our ProgrammingReference

triedhardtogetyoustrongprogrammingreviewsforthisbook. Wife Insomecases, wear elucky enough to get are view from the person who designedthelanguage. I'm confident that this material will capture the spiritofeachprogramminglanguageprettywellbythetimeithasgone through the entire review process. That said, please understand that I cannotpossiblyfullysupportyourendeavorsineachlanguage. Iwould liketomakeacomparisontospokenlanguages.

Knowingalanguageasatouristpassingthroughisfardifferentfrom beinganativespeaker.IspeakEnglishfluentlyandSpanishhaltingly. Iknowafewphrasesinthreeotherlanguages.IorderedfishinJapan. I asked to find a restroom in Italy. But I know my limitations. From the programming side, I speak Basic, C, C++, Java, C#, JavaScript, Ruby, and a few others fluently. I speak dozens of others haltingly, including the languages in this book. I'm not qualified to support six

of the languages on this list. I write Ruby full-time and have for five years now. But I couldn't telly ou how to write a webserver in Ioora databaseinErlang.

Iwouldfailbadlyifltriedtoprovideanexhaustivereferenceforeachof theselanguages. Icould make a programming guide that 's at least as longasthisbookonanyoftheseparatelanguagesinhere. I will give youenoughinformationtogetstarted. I will walkyouthrough examples ineachlanguages, and you'll see examples of those programs. I will do mybesttocompileeverythingandmakesureitallruns.ButIcouldn't supportyourprogrammingeffortseveniflwanted.

The languages on this list all have exceptional support communities. That'spartofthereasonIpickedthem.Ineachoftheexercises,Itryto haveasectionthatasksyoutofindresources. This idea is intentional. Itwillmakeyouself-reliant.

IAmGoingtoPushY ou Hard

This book is going to take you one step beyond your twenty-minute know Google as well as I do, and you'll be able to find tutorial. You one of those simple primers for every language on this list. I will give you a quick interactive tour. Yol also get some small programming challenges and one programming project everyweek. It's not going to beeasy, but it will be informative and fun.

Ifyousimplyreadthisbook, you'll experience the flavor of the syntax and no more. If you look online for the answers before trying to code theexercisesyourself, you'llfail. Yuwillwanttotry the exercises first, fully recognizing that you'll fail at a few of them. Learning syntax is alwayseasierthanlearningtoreason.

If you find yourself nervous after reading this description, I suggest thatyouputdownthisbookandpickupanother. Yuwon'tbehappy would probably be better served by seven different programmingbooks. Butifyoufindyourself excited about the prospect of codingbetterquickly,let'spushon.

AFinalCharge 1.5

Atthispoint, I expected to have some sweeping, motivational words to say, but it all seemed to boil down to two words.

Havefun.

MaryPoppins

Chapter 2 Ruby

1

If you are sampling this book, chances are we have something in common:learningprogramminglanguagesintriguesus.Tome,learningalanguageislikelearningacharacter. Throughout mycareer, I've experienced scores of languages first hand. Like any person, each languagetookonadistinctpersonality. Javawaslikehavingarichlawyer as a brother. He was fun when he was younger, but now he's a black hole that sucks away all the joy in a 100-mile radius. Vad Basic was like that bleached-blond cosmetologist. She's not going to solve globalwarming, butshe's always good for a hair cut and tremendously funtotalkto. Throughout the book, I will compare the languages you willencountertopopularcharacters. Ihopethecomparisons will help you unlock a little bit about the character that makes each language special.

MeetRuby, one of my favorites. She's sometimes quirky, always beautiful, alittlemysterious, and absolutely magical. Think Mary Poppins, the British nanny. At the time, most nannies were like most of the C family of languages—draconian beasts who were mercilessly efficientbutaboutasfunastakingthatshotofcodliveroileverynight. Whaspoonfulofsugar, everything changed. Mary Poppinsmade the household more efficient by making it fun and coaxing every last bit of passion from her charges. Ruby does the same thing and with more syntactic sugar ² than a spoonful. Matz, Ruby's creator, doesn't worry

^{1.} MaryPoppins .DVD.DirectedbyRobertStevenson.1964;LosAngeles,CA:WdDisnev\602004.

^{2.} Syntactic sugar describes a language feature that makes code easier to read and write, though the reareal ternative ways to express the same code.

about the efficiency of the language. He optimizes the efficiency of the programmers.

QuickHistory 2.1

Ykio Matsumoto created Ruby in about 1993. Most people just callhimMatz.Asalanguage, Rubyisaninterpreted, object-oriented, dynamically typed language from a family of so-called scripting languages.InterpretedmeansthatRubycodeisexecutedbyaninterpreter ratherthanacompiler. Dynamically typed means that types are bound atexecutiontimeratherthancompiletime. In general, the trade-offfor such a strategy is flexibility versus executions a fety, but we'll get into that a little more later. Object-oriented means the language supports encapsulation (data and behavior are packaged together), inheritance through classes (object types are organized in a class tree), and polymorphism(objectscantakemanyforms). Rubypatientlywaitedforthe right moment and then burst onto the scene around 2006 with the emergenceoftheRailsframework.Afterwanderingfortenvearsinthe enterprise jungles, programming was fun again. Ruby is not hugely efficient in terms of execution speed, but it makes programmers very productive.

Interviewwith Yulio (Matz)Matsumoto

Ihad the pleasure to travel to Matsumotosan's hometown of Matsue, Japan. We had the chance to have some conversations about the foundationsofRuby,andheagreedtoanswersomequestionsforthisbook.

Bruce: WhydidyouwriteRuby?

Matz: Right after I started playing with computers, I got interested in programming languages. They are the means of programming but also enhancers for your mind that shape the way you think about programming. So for a long time, for a hobby, I studied a lot of programming languages. Ievenimplemented several to y language sbut no relones.

In1993,whenIsawPerl,Iwassomehowinspiredthatanobject-oriented language that combines characteristics from Lisp, Smalltalk, and Perl would be a great language to enhance our productivity. So, I started developing such a language and named it Ruby. The primary motivationwastoamusemyself. Itwasmerehobby at the beginning, trying to createalanguagethatfitmutaste.Somehow,otherprogrammersallover

the world have felt sympathy for that language and the policy behind it.Anditbecameverypopular, farbeyondmyexpectation.

Bruce: Whatisthethingyoulikeaboutitthemost?

Matz: I like the way it makes my programming enjoyable. As a particular technicalissue, Ilike blocks most. They are tamed higher-order functions but open up great possibilities in DSL and other features as well.

Bruce: Whatisafeaturethatyouwouldliketochange,ifyoucouldgo backintime?

Matz: I would **roc** the thread and add actors or some other more advancedconcurrencyfeatures.

As you read through this chapter, whether or not you already know Ruby, try to keep an eye outfortrade-offs that Matzmade along the way.Lookforsyntacticsugar,thoselittlefeaturesthatbreakthebasic rulesofthelanguagetogiveprogrammersalittlefriendlierexperience andmakethecodealittleeasiertounderstand.FindplaceswhereMatz usedcodeblocksformarvelouseffectincollectionsandelsewhere.And trytounderstandthetrade-offsthathemadealongthewaybetween simplicityandsafetyandbetweenproductivityandperformance.

Let'sgetstarted.TakeapeekatsomeRubycode:

```
>>properties=[
                 'objectoriented' ,
                                    'ducktyped' , 'productive', 'fun']
=>[ "objectoriented" , "ducktyped" , "productive", "fun"]
                                   "Rubvis#{property}." }
>>properties.each{|property|puts
Rubyisobjectoriented.
Rubyisducktyped.
Rubyisproductive.
Rubyisfun.
=>[ "objectoriented" , "ducktyped" , "productive", "fun"]
```

Ruby is the language that taught me to smile again. Dynamic to the core, Rubyhasamar velous support community. The implementations areallopensource. Most commercial support comes from smaller companies, and that has insulated Ruby from some of the over-reaching frameworksthatplaguesomeotherkingdoms. Rubyhasbeenslowto catchonintheenterprise, butit's taking hold now on the strength of itsproductivity, especially in the area of web development.

Day 1: Finding a Nanny

Allmagicaside, Mary Poppinsis first and foremost agreat nanny. When youfirstlearnalanguage, your jobistolearn how to use it to do the jobs you already know how to do. Treat this first conversation with Rubyasadialogue. Does the conversation flow freely, or is it unnecessarily awkward? What's the core programming model? How does it treattypes?Let's startdigging for some answers.

Lightning b

Aspromised, I'mnotgoing to takeyouthrough an exhaustive outdated installationprocess, butinstalling Rubyisasnap. Justgoto ruby-lang.org/en/downloads/, findyourplatform, and install Ruby 1.8.6 or newer. I amrunning Ruby version 1.8.7 for this chapter, and version1.9 will have some slight differences. If you're on Windows there's a one-click installer that will work, and if you're on OS X Leopard or greater, Ruby comes with the X coded isks.

Totestyourinstallation, justtype irb.Ifyoudon'tseeanyerrors,you're ready to handle the rest of this chapter. If you do, don't be shy. Way fewinstallationproblemsareunique. Googlewillshowtheway.

UsingRubywiththeConsole

Ifyouhaven'tdoneso, type irb. Yushouldsee Ruby's interactive console. **Yil**typeacommandandgetaresponse. Givetheseatry:

```
>>puts'hello,world'
hello,world
=>ni1
>>language='Ruby'
=>"Ruby"
>>puts"hello,#{language}"
hello, Ruby
=>ni1
>>language='myRuby'
=>"myRuby"
>>puts"hello,#{language}"
hello, myRuby
=>ni1
```

Ifyoudon'talreadyknowRuby,thisbriefexamplegivesyoumanyclues know that Ruby can be interpreted. In fact, about the language. You Rubyisalmostalwaysinterpreted, though some developers are working onvirtualmachinesthatcompileRubytobytecodeasitgetsexecuted. Ididn't declare any variables. Everything I did returned a value, even

whenIdidn'taskRubytoreturnanything.Infact, everypieceofcode inRubyreturnssomething.

Yu also sawat least two types of strings. One quote around a string means the string should be interpreted literally, and two quotes leads tostringevaluation. One of the things that the Rubyinter preterevaluatesisstringsubstitution.Inthisexample,Rubysubstitutedthevalue returnedbythecode language into the string. Let's keep going.

TheProgrammingModel

Oneofthefirstquestionsaboutalanguageyoushouldansweris, "What is the programming model?" It's not always a simple answer. York probablybeenexposedtoprocedurallanguagessuchasC,Fortran,or Pascal. Most of usare using object-oriented languages right now, but manyofthoselanguageshaveproceduralelementstoo.Forexample, inJavaisnotanobject. Yumayhavepickedupthisbooktoexplore functional programming languages. Some of those languages such as Scala mix programming models by throwing in object-oriented concepts. There are dozens of other programming models as well. Stack $based \, languages \, such as \, Post Script \, or \, For thuse \, one \, or \, more \, stacks \,$ as a central feature of the language. Logic-based languages such as Prolog build around rules. Prototype languages like Io, Lua, and Self usetheobject,nottheclass,asthebasisforobjectdefinitionandeven inheritance.

Rubyisapureobject-orientedlanguage.Inthischapter,you'llseejust howfarRubytakesthisconcept.Let'slookatsomebasicobjects:

```
>>4
=>4
>>4.class
=>Fixnum
>>4+4
=>8
>>4.methods
=>["inspect","%","<<","singleton_method_added","numerator",...
"*","+","to_i","methods",...
```

I'veomittedsomeofthemethodsfromthislist, butyougetthepicture. Just about everything in Ruby is an object, down to each individual number. Anumber is an object that has a class called Fixnum.andthe method called methods returns an array of methods (Ruby represents arraysinsquarebrackets). Infact, you can call any methodon an object withthedotoperator.

4

Decisions

Programsexisttomakedecisions, soitstandstoreas on that the way a languagemakesdecisionsisacentralconceptthatshapesthewayyou code, and think, in a given language. Rubyislike most object-oriented and procedural languages in many ways. Check out these expressions:

```
>> x=4
=>4
>>x<5
=>true
>>x<=4
=>true
>>x>4
=>false
>>false.class
=>FalseClass
>>true.class
=>TrueClass
```

So, Ruby has expressions that evaluate to true or the .True to form, true and the arealsofirst-classobjects. Yucanconditionally execute codewiththem:

```
>> x=4
=>4
>>puts'Thisappearstobefalse.'unlessx==4
>>puts'Thisappearstobetrue.'ifx==4
Thisappearstobetrue.
=>ni1
>>ifx==4
     puts'Thisappearstobetrue.'
>>end
Thisappearstobetrue.
=>ni1
>>unlessx==4
     puts'Thisappearstobefalse.'
>>else
     puts'Thisappearstobetrue.'
>>end
Thisappearstobetrue.
>>puts'Thisappearstobetrue.'ifnottrue
>>puts'Thisappearstobetrue.'if!true
=>nil
```

IreallylikeRuby'sdesignchoiceforsimpleconditionals. Yucanuse bothblockforms(if condition, statements, end)orone-lineforms(ment if condition) when you're working with if or unless. To some, the one-lineversionofthe if isoff-putting.Tome,itallowsyoutoexpressa singlethoughtinalineofcode:

```
order.calculate_taxunlessorder.nil ?
```

Sure, you can express the previous in a block, but you would add additional noise to what should be a single, coherent thought. When you candistillasimpleideaintooneline, youmakereading your codeless of aburden. I also like the idea of unless.**Y**u couldexpressthesame ideawith not or !, but unless expresses the ideamuch better.

while and until aresimilar:

```
>>x=x+1whilex<10
=>nil
>>X
=>10
>> x=x-1unti1x==1
=>nil
>>X
=>1
>>whilex<10
     x=x+1
     putsx
>>end
2
3
4
6
7
8
9
10
=>ni1
```

Noticethat = isforassignmentand == testsforequality.InRuby,each objectwillhaveitsnotionofequality. Numbers are equaliftheir values areequal.

Yucanusevaluesotherthan true and the asexpressionstoo:

```
>>puts'Thisappearstobetrue.'if1
Thisappearstobetrue.
=>ni1
>>puts'Thisappearstobetrue.'if'randomstring'
(irb):31:warning:stringliteralincondition
Thisappearstobetrue.
>>puts'Thisappearstobetrue.'if0
Thisappearstobetrue.
=>ni1
```

```
>>puts'Thisappearstobetrue.'iftrue
Thisappearstobetrue.
=>ni1
>>puts'Thisappearstobetrue.'iffalse
>>puts'Thisappearstobetrue.'ifnil
=>ni1
```

So, everything but nil and the evaluate true. CandC++programmers, takenote. 0 is true!

Logical operators work like they do in C, C++, C#, and Java, with a fewminorexceptions. and (alternatively & &) is a logical and. or (alternatively ||) is a logical or. Whithese tests, the interpreter will execute codeonlyuntilthevalueofthetestisclear.Use&or tocomparewhile executing the whole expression. Here are the seconcepts in action:

```
>>trueandfalse
=>false
>>trueorfalse
=>true
>>false&&false
=>false
>>true&&this_will_cause_an_error
NameError:undefinedlocalvariableormethod`this_will_cause_an_error'
    formain:Object
        from(irb):59
>>false&&this_will_not_cause_an_error
>>trueorthis_will_not_cause_an_error
>>true||this_will_not_cause_an_error
=>true
>>true|this_will_cause_an_error
NameError:undefinedlocalvariableormethod`this_will_cause_an_error'
    formain:Object
        from(irb):2
        from:0
>>true|false
=>true
```

There's no magichere. Yalnormally use the short-circuit version of these commands.

DuckTyping

Let's get into Ruby's typing model a little. The first thing you need to the first tknow is how much protection Ruby will give you when you make a mistakewithtypes.Wie talkingabouttypesafety.Stronglytypedlan-

guages check types for certain operations and check the types before youcandoanydamage. This check can happen when you present the codetoaninterpreteroracompilerorwhenyouexecuteit. Checkout thiscode:

```
>>4+'four'
TypeError:Stringcan'tbecoercedintoFixnum
        from(irb):51:in`+'
        from(irb):51
>>4.class
=>Fixnum
>>(4.0).class
=>Float
>>4+4.0
=>8.0
```

³ meaning you'll get an error when types So, Ruby is strongly typed, collide. Rubymakes these type checks a truntime, not compile time. I'mgoingtoshowyouhowtodefineafunctionalittlebeforeInormally wouldtoprovethepoint.Thekeyword def definesafunctionbutdoesn't executeit.Enterthiscode:

```
>>defadd them up
     4+'four'
>>end
=>ni1
>>add_them_up
TypeError:Stringcan'tbecoercedintoFixnum
        from(irb):56:in`+'
        from(irb):56:in`add_them_up'
        from(irb):58
```

So, Ruby does not do type checking until you actually try to execute code. This concept is called dynamictyping. Thereare disadvantages: youcan't catch as many errors as you can with static typing because compilers and tools can catch more errors with a statically typed system. But Ruby's type system also has several potential advantages. classes don't have to inherit from the same parent to be used inthesameway:

```
>>i=0
=>0
```

^{3.} I'm lying to you a little, but only a little. Two examples from here, you'll see me changean existing class at runtime. Theoretically, auser can change a class beyond all recognition and defeat types a fety, so in the strict est sense, Ruby is not strongly typed. Butforthemostpart, Rubybehaveslikeastronglytypedlanguagemostofthetime.

```
>>a=['100',100.0]
=>['100',100.0]
>>whilei<2
     putsa[i].to_i
     i=i+1
>>end
100
100
```

You just saw duck typing in action. The first element of the array is a String, and the second is a Float. The same code converts each to an integervia to_i.Ducktypingdoesn'tcarewhattheunderlyingtypemight be. If it walks like a duck and quacks like a duck, it's a duck. In this case, the quack methodis to_i.

Duck typing is extremely important when it comes to clean objectoriented design. An important tenet of design philosophy is to code tointerfacesratherthanimplementations. If you'reusing ducktyping, this philosophy is easy to support with very little added ceremony. If anobjecthas push and pop methods, you can treat it like a stack. If it doesn't, you can't.

WhatWeLearnedinDay 1

Sofar, you've just waded through the basics. It's an interpreted objectorientedlanguage. Justabouteverything is an object, and it's easy to getatanyobject'sparts, like the methods and the class. Rubyis duck typed,andRubybehavesmostlylikeastronglytypedlanguage,though some academics would argue with that distinction. It's a free-spirited languagethatwillletyoudojustaboutanything, including changing coreclasseslike NilClass and String. Nowletmeturnyoulooseforalittle self-study.

Day 1 Self-Study

So, you've had your first date with Ruby. Now, it's time to write a little code.Inthissession,you'renotgoingtowritewholeprograms.Instead, you'lluse irb to execute a few Ruby snippets. As always, you can find theanswersinthebackofthebook.

Find:

- TheRubyAPI
- The free online version of Programming Ruby: The Pragmatic Programmer's Guide [TFH08]

- Amethodthatsubstitutespartofastring
- InformationaboutRuby's regular expressions
- InformationaboutRuby'sranges

Do:

- Printthestring "Hello, world."
- Forthestring "Hello, Ruby," find the index of the word "Ruby."
- Printyournametentimes.
- Printthestring "This is sentence number 1," where the number 1 changesfrom1to10.
- RunaRubyprogramfromafile.
- Bonus problem: If you're feeling the need for a little more, write a program that picks a random number. Let a player guess the number, telling the player whether the guessist ool owort oo high.

will generate a random number from 0 to 9, and gets willreadastringfromthekeyboardthatyoucantranslateto aninteger.)

Day2:FloatingDownfromtheSky 2.3

Atthetime, one of the most striking scenes in MaryPoppins washer entrance. She floated into town on her umbrella. My kids will never understandwhythatentrancewasgroundbreakingstuff. Today, you're going to experience a little bit of the magic that makes Ruby click. Yallearntousethebasicbuildingblocksofobjects, collections, and also learn the basics of the code block. Open up your classes. Yol mindtoalittlemagic.

DefiningFunctions

UnlikeJavaandC#.youdon'thavetobuildawholeclasstodefinea function. Y u can define a function r ight in the console:

```
>>deftell the truth
     true
>>end
```

Everyfunctionreturnssomething. If you do not specify an explicit return, the function will return the value of the last expression that's processedbeforeexiting.Likeeverythingelse,thisfunctionisanobject. Later, we'llworkonstrategies to passfunctions as parameters to other functions.

Arrays

Arrays are Ruby's workhorse ordered collection. Ruby 1.9 introduces orderedhashestoo, butingeneral, arraysare Ruby's primary ordered collection.Takealook:

```
>>animals=['lions','tigers','bears']
=>["lions", "tigers", "bears"]
>>putsanimals
lions
tigers
bears
=>ni1
>>animals[0]
=>"lions"
>>animals[2]
=>"bears"
>>animals[10]
=>ni1
>>animals[-1]
=>"bears"
>>animals[-2]
=>"tigers"
>>animals[0..1]
=>['lions','tigers']
>>(0..1).class
=>Range
```

Yu can see that Ruby collections will give you some freedom. If you access any undefined array element, Ruby will simply return nil. You willalsofindsomefeaturesthatdon'tmakearraysmorepowerfulbut just make them easier to use. animals[-1] returned the first element from the end, animals[-2] returned the second, and so on. These featuresarecalled syntactic sugar, an added feature for convenience. The animals[0..1] expressionmightlooklikesyntacticsugar, butit's not. 0..1 isactuallya Ronge, meaning all numbers from 0 to 1, inclusive.

Arrayscanholdothertypesaswell:

```
>>a[0]=0
NameError:undefinedlocalvariableormethod`a'formain:Object
from(irb):23
>>a=[]
=>[]
```

Oops.Itriedtouseanarraybeforeitwasanarray.Thaterrorgivesyou acluetothewayRubyarraysandhasheswork. [] isactuallyamethod on Array:

```
>>[1].class
=>Array
>>[1].methods.include?('[]')
=>true
>>#use[1].methods.include?(:[])onruby1.9
```

So, [] and []= arejustsyntactic sugartoal lowaccess to an array. To do this right, Ineed to put an empty array into it first, and then I can play aroundwithitsome:

```
>>a[0]='zero'
=>"zero"
>>a[1]=1
=>1
>>a[2]=['two','things']
=>["two","things"]
=>["zero",1,["two","things"]]
```

Arraysdon'tneedtobehomogeneous.

```
>>a=[[1,2,3],[10,20,30],[40,50,60]]
=>[[1,2,3],[10,20,30],[40,50,60]]
>>a[0][0]
=>1
>>a[1][2]
=>30
```

Andmultidimensionalarraysarejustarraysofarrays.

```
>>a=[1]
=>[1]
>>a.push(1)
=>[1,1]
>>a=[1]
=>[1]
>>a.push(2)
=>[1,2]
>>a.pop
=>2
>>a.pop
=>1
```

ArrayshaveanincrediblyrichAPI. Yu canuseanarrayasaqueue, alinkedlist, astack, oraset. Now, let's take a look at the other major collectioninRuby, the hash.

Hashes

Rememberthat collections are buckets for objects. In the hashbucket, everyobjecthasalabel. The labelisthekey, and the object is the value. Ahashisabunchofkey-valuepairs:

```
>>numbers={1=>'one'.2=>'two'}
=>{1=>"one",2=>"two"}
>>numbers[1]
=>"one"
>>numbers[2]
=>"two"
>>stuff={:array=>[1,2,3],:string=>'Hi,mom!'}
=>{:array=>[1,2,3],:string=>"Hi,mom!"}
>>stuff[:string]
=>"Hi,mom!"
```

Thisisnottoocomplicated. Ahashworks a lotlike an array, but instead ofanintegerindex, the hash can have any arbitrary key. The last hash is interesting because I'm introducing a symbol for the first time. A symbolisanidentifierprecededwithacolon,like :symbol.Symbolsare greatfornaming things or ideas. Although two strings with the same valuecanbedifferentphysicalstrings, identical symbols are the same can tell by getting the unique object identifier of physical object. You thesymbolseveraltimes, likeso:

```
>>'string'.object_id
=>3092010
>>'string'.object id
=>3089690
>>:string.object_id
=>69618
>>:string.object_id
=>69618
```

Hashes sometimes show up in unusual circumstances. For example, Rubydoesnotsupportnamedparameters, butyoucan simulate them with a hash. Throwin a little syntactic sugar, and you can get some interestingbehavior:

```
>>deftell_the_truth(options={})
     ifoptions[:profession]==:lawyer
       'itcouldbebelievedthatthisisalmostcertainlynotfalse.'
>>
     else
>>
       true
>>
     end
>>
>>end
=>ni1
>>tell_the_truth
=>true
```

```
>>tell_the_truth:profession=>:lawyer
=>"itcouldbebelievedthatthisisalmostcertainlynotfalse."
```

Thismethodtakes a single optional parameter. If you pass nothing in, options willbesettoanemptyhash.Ifyoupassina :profession of :lawyer, you will get something different. The result is not fully true, but it is almostjustasgood, because the system will evaluate it as true. Notice that you didn't have to type in the braces. These braces are optional forthelast parameter of a function. Since array elements, hashkeys, and hashvalues can be almost anything, you can build some incredibly sophisticateddatastructuresinRuby,buttherealpowercomeswhen youstarttogetintocodeblocks.

CodeBlocksandYield

Acodeblockisafunctionwithoutaname. Yucanpassitasaparametertoafunctionoramethod. For example:

```
>>3.times{puts'hiyathere,kiddo'}
hiyathere, kiddo
hiyathere, kiddo
hiyathere, kiddo
```

codeblock. times is a method on The code between braces is called a Fixnum that simply does something some number of times, where something is a code block and number is the value of the Fixnum. You specifycodeblocks with {/} or do/end. The typical Ruby convention is tousebraceswhenyourcodeblockisononelineandusethe do/end formwhenthecodeblocksspanmorethanoneline.Codeblockscan takeoneormoreparameters:

```
>>animals=['lionsand','tigersand','bears','ohmy']
=>["lionsand","tigersand","bears","ohmy"]
>>animals.each{|a|putsa}
lionsand
tigersand
bears
ohmy
```

This code begins to display the power of the code block. That code told Rubywhattodoforeveryiteminthecollection. Whafraction of the syntax, Rubyiterated over each of the elements, printing each one. To really getafeel for what 's going on, here's a custom implementation of the times method:

```
>>classFixnum
>>
    defmy times
>>
       i=self
```

```
whilei>0
>>
        i=i-1
>>
        yield
       end
     end
>>
>>end
=>ni1
>>3.my_times{puts'mangymoose'}
mangymoose
mangymoose
mangymoose
```

This code open supanexisting class and adds a method. In this case, themethod called mytes loops a set number of times, invoking the codeblockwith yield.Blockscanalsobefirst-classparameters.Check outthisexample:

```
>>defcall block(&block)
     block.call
>>end
=>ni1
>>defpass_block(&block)
     call_block(&block)
>>end
=>nil
>>pass_block{puts'Hello,block'}
Hello, block
```

Thistechniquewillletyoupassaroundexecutablecode.Blocksaren't just for iteration. In Ruby, you'll use block stodel a yexecution...

```
execute_at_noon{puts'Beepbeep...timetogetup'}
```

conditionally executes omething...

```
...somecode...
in_case_of_emergencydo
 use_credit_card
 panic
end
defin_case_of_emergency
 yieldifemergency?
end
...morecode...
enforcepolicy...
within_a_transactiondo
 things_that
 must_happen_together
end
```

```
defwithin_a_transaction
 begin_transaction
 vield
 end_transaction
end
```

and many other places. Yol see Ruby libraries that use blocks to processeachlineofafile,doworkwithinanHTTPtransaction,anddo complex operations over collections. Ruby is ablock party.

RunningRubyfromaFile

The code examples are getting a little more complicated, so working from the interactive console isn't all that convenient anymore. Yil use the console to explore short bits of code, but you'll primarily put your code into files. Create a file called hello.rb. You can include any Rubycodethatyou'dlike:

```
puts 'hello,world'
```

Saveittoyourcurrentdirectory, and then execute it from the command

```
batate$rubyhello.rb
hello,world
```

Afewpeopleareusing Ruby from full integrated development en vironments, but many are happy to use a simple editor with files. My favorite isTextMate,butvi,emacs,andmanyotherpopulareditorshaveRuby plug-ins. Whthis understanding in our backpocket, we can move on tothereusablebuildingblocksofRubyprograms.

DefiningClasses

Like Java, C#, and C++, Ruby has classes and objects. Think cookie cutterand cookie—classes are templates for objects. Of course, Ruby supports inheritance. Unlike C++, a Ruby class can inherit from only oneparent,calleda superclass. To see it allinaction, from the console, typethefollowing:

```
>>4.class
=>Fixnum
>>4.class.superclass
=>Integer
>>4.class.superclass.superclass
=>Numeric
>>4.class.superclass.superclass.superclass
=>0biect
>>4.class.superclass.superclass.superclass
=>ni1
```

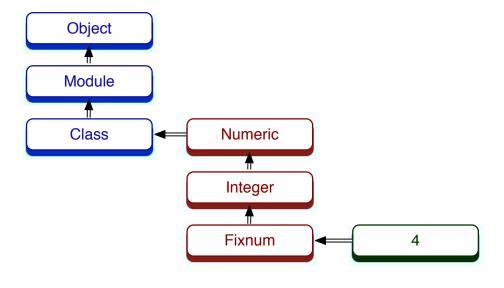


Figure 2.1: Rubymetamodel

Sofar, sogood. Objects are derived from a class. The 4'sclassis Fixnum. whichinheritsfrom Integer, Numeric, andultimately Object.

Checkout Figure 2.1 to see how things fit together. Everything eventuallyinheritsfrom Object.A Class isalsoa Module.Instancesof Class serve as templates for objects. In our case, Fixnum isaninstanceofa class, and 4 is an instance of Fixnum. Each of these classes is also an object:

```
>>4.class.class
=>Class
>>4.class.class.superclass
=>Module
>>4.class.class.superclass.superclass
=>0bject
```

So, Fixnum is derived from the class Class. From there, it really gets confusing. Classinherits from Module, and Module inherits from Object. Whenallissaidanddone, everything in Rubyhasacommonancestor, Object.

```
Download ruby/tree.rb
classTree
 attr_accessor:children,:node_name
 definitialize(name,children=[])
    @children=children
    @node name=name
  end
 defvisit all(&block)
   visit&block
    children.each{|c|c.visit_all&block}
  end
 defvisit(&block)
    block.callself
  end
end
ruby_tree=Tree.new(
                       "Ruby",
  [Tree.new("Reia"),
  Tree.new("MacRuby")])
puts "Visitinganode"
ruby_tree.visit{|node|putsnode.node_name}
puts
puts "visitingentiretree"
ruby_tree.visit_all{|node|putsnode.node_name}
```

This power-packed class implements a very simple tree. It has three methods, initialize, visit, and visit_all, and two instance variables, children and node_name. initialize has special meaning. Ruby will call it when the classinstantiatesanewobject.

IshouldpointoutafewconventionsandrulesforRuby.Classesstart with capital letters and typically use CamelCase to denote capitalizamust prepend instance variables (one value per object) with tion.You @ andclassvariables(onevalueperclass)with @@.Instancevariables andmethodnamesbeginwithlowercaselettersinthe underscore_style. Constants are in ALL_CAPS. This code defines a tree class. Each tree hastwoinstancevariables: @children and @node_name.Functionsand methodsthattesttypicallyuseaquestionmark(iftest?).

The offr keyword defines an instance variable. Several versions exist. Themostcommonare of (defining an instance variable and a method of the same name to access it) and attr_accessor, defining an instance variable, anaccessor, and a setter.

Our dense program packs a punch. It uses blocks and recursion to allow any user to visit all nodes in a tree. Each instance of has onenodeofatree.The initialize methodprovides the starting values for children and node_name. The visit method calls the inbound code block. The visit_all method calls visit for the node and then recursively calls visit all foreachofthechildren.

Theremaining code uses the API. It defines a tree, visits one node, and thenvisitsallnodes.Runningitproducesthisoutput:

```
Visitinganode
Ruby
visitingentiretree
Ruby
Reia
MacRuby
```

Classes are only part of the equation. Yive briefly seen modules in thecodeonpage 42. Let's goback and take a closer look.

WritingaMixin

Object-oriented languages use inheritance to propagate behavior to similar objects. When the behaviors are not similar, either you can allow inheritance from more than one class (multiple inheritance) or you can look to another solution. Experience has shown that multiple inheritance is complicated and problematic. Java uses interfaces to solve this problem. Ruby uses modules. A module is a collection of functions and constants. When you include a module aspart of a class, thosebehaviors and constants become part of the class.

Takethisclass.whichaddsa to_f methodtoanarbitraryclass:

```
Download ruby/to_file.rb
moduleToFile
  deffilename
    "object_#{self.object_id}.txt"
  end
  defto f
    File.open(filename, 'w'){|f|f.write(to_s)}
  end
end
```

```
classPerson
 includeToFile
  attr_accessor:name
 definitialize(name)
    @name=name
  end
  defto s
    name
  end
end
Person.new('matz').to_f
```

Start with the module definition. This module has two methods. The to_f method writes the output of the to s method to a file with a filenamesupplied by the filename method. What's interesting here is that to_s is used in the module but implemented in the class! The class has not even been defined yet. The module interacts with the including class at an intimate level. The module will often depend on several classmethods. WhJava, this contract is explicit: the class will implement a formal interface. Wh Ruby, this contract is implicit, through duck typing.

arenotatallinteresting, and that's the point. The The details of Peron includes the module, and we're done. The ability to write to a file hasnothingtodowithwhetheraclassisactuallya .We addthe Reason capability to add the contents to a file by mixing in the capability. We canaddnewmixinsandsubclassesto Reacon .andeach subclass will have the capabilities of all the mixins without having to know about themixin'simplementation. When all is said and done, you can use a simplified single inheritance to define the essence of a class and then attachadditionalcapabilities with modules. This style of programming, introduced in Flavors and used in many languages from Smalltalk to Python,iscalleda mixin. The vehicle that carries the mixin is not always calledamodule, but the premise is clear. Single inheritance plus mixins allowforanicepackagingofbehavior.

Modules, Enumerable, and Sets

A couple of the most critical mixins in Ruby are the enumerable and comparable mixins. Aclasswanting to be enumerable mustimplement comparable mustimplement <=>.Called each,andaclasswantingtobe the spaceship operator, <=> is a simple comparison that returns-1 if

a is greater, and 0 otherwise. In exchange for impleb isgreater, 1 if mentingthesemethods, enumerable and comparable providemanyconveniencemethodsforcollections. Crackopentheconsole:

```
>> 'begin'<=>'end'
=>-1
>> 'same' <=> 'same'
>> a=[5,3,4,1]
=>[5,3,4,1]
>>a.sort
=>[1,3,4,5]
>>a.any?{|i|i>6}
=>false
>>a.any?{|i|i>4}
=>true
>>a.all?{|i|i>4}
=>false
>>a.all?{|i|i>0}
=>true
>>a.collect{|i|i
                    * 2}
=>[10,6,8,2]
>>a.select{|i|i%2==0}#even
>>a.select{|i|i%2==1}#odd
=>[5,3,1]
>>a.max
=>5
>>a.member?(2)
=>false
```

any? returns true if the condition is true for any of the elements; all? returns true if the condition is true for all elements. Since the spaceshipisimplementedontheseintegersthrough Fixnum, you can sort and compute the min or max.

Yucanalsodoset-basedoperations. collect and map applyafunction to each of the elements and return an array of the results. find finds one element matching the condition, and both select and find all return allelementsmatchingacondition. Yucanal so compute the total of a listortheproductwith inject:

```
>>a
=>[5,3,4,1]
>>a.inject(0){|sum,i|sum+i}
>>a.inject{|sum,i|sum+i}
>>a.inject{|product,i|product
                                   * i}
=>60
```

inject seemstricky, butit's not too complicated. It takes a code block withtwoargumentsandanexpression. The codeblock will be executed foreachiteminthelist, with inject passingeachlistelementtothecode blockasthesecondargument. The first argument is the result of the previous execution of the codeblock. Since the result won't have a value thefirsttimethecodeblockisexecuted, you just pass the initial value as the argument to inject. (If you don't specify a value, inject will use the first value in the collection.) Take a second look, with a little help:

```
>>a.inject(0)do|sum,i|
?> puts"sum:#{sum} i:#{i}
                               sum+i:#{sum+i}"
?> sum+i
?>end
sum:0 i:5 sum+i:5
sum:5 i:3 sum+i:8
sum:8 i:4 sum+i:12
sum:12 i:1 sum+i:13
```

As expected, the result of the previous line is always the first value passed to the next line. Using inject, you can compute the word count of many sentences, find the largest word in a paragraph of lines, and domuchmore.

WhatWeLearnedinDay2

ThisisyourfirstchancetoseesomeofRuby'ssugarandalsoalittleof themagic. Yoje starting to see how flexible Ruby can be. The collections are dead simple: two collections with multiple API slayered on top.Application performance is secondary. Ruby is about the performance enumerable module gives you a flavor of just of the programmer. The howwell-designedRubycanbe.Thesingle-inheritanceobject-oriented structure certainly isn't novel, but the implementation is packed with intuitivedesignandusefulfeatures. This level of abstraction gives you a marginally better programming language, but serious mojo is on theway.

Day2Self-Study

These problems will be a little more demanding. York used Ruby a littlelonger, so the gloves are off. These examples will force you to do a littlemoreanalyticalthinking.

Find:

 Findouthowtoaccessfiles with and without code blocks. What isthebenefitofthecodeblock?

- Howwouldyoutranslate a hash to an array? Canyoutranslate arraystohashes?
- Canyouiteratethroughahash?
- can use Ruby arrays as stacks. What other common data structuresdoarrayssupport?

Do:

- Printthe contents of an array of sixteen numbers, four numbers each. Now, do the same with at a time, using just each slice in Fnumerable.
- The Expression classwas interesting, but it did not allow you to specify anewtreewithacleanuserinterface. Let the initializer accept a nested structure with hashes and arrays. You should be able to specifyatreelikethis: {'grandpa'=>{' dad' =>{'child1'=>{},'child 2'=>{}},'uncle'=>{'child3'=>{},'child4'=>{}}}}.
- We a simple grep that will print the lines of a file having any occurrencesofaphraseanywhereinthatline. Yuwillneedtodo asimpleregular expression matchandread lines from a file. (This issurprisinglysimpleinRuby.)Ifyouwant,includelinenumbers.

Day3:SeriousChange

ThewholepointofMaryPoppinsisthatshemadethehouseholdbetter asawholebymakingitfunandchangingtheheartsofthepeoplein itwithpassionandimagination. Yucouldbackoffalittleandplayit safe, using Ruby to do the same things you already know how to do in other languages. But when you change the way a language looks and works, you can capture magic that makes programming fun. In this book, each chapter will show you some nontrivial problem that thelanguagesolveswell.InRuby,thatmeansmetaprogramming.

Metaprogramming means writing programs that write programs. The ActiveRecordframeworkthat's the centerpiece of Railsuses metaprogramming to implement a friendly language for building classes that linktodatabasetables.AnActiveRecordclassfora Department might looklikethis:

```
classDepartment<ActiveRecord::Base</pre>
  has_many:employees
  has one:manager
end
```

has_many and has_one areRubymethodsthataddalltheinstancevariablesandmethodsneededtoestablishahas_manyrelationship.This classspecificationreadslikeEnglish,eliminatingallthenoiseandbaggagethatyouusuallyfindwithotherdatabaseframeworks.Let'slook atsomedifferenttoolsyoucanuseformetaprogramming.

OpenClasses

Ymealreadyhadabriefintroductiontoopenclasses. Yucanchange thedefinitionofanyclassatanytime, usually to add behavior. Here's a great example from the Rails framework that adds a method toNilClass:

```
Download ruby/blank.rb
classNilClass
  defblank?
    true
  end
end
classString
  defblank?
    self.size==0
  end
end
["", "person", nil].eachdo|element|
  putselementunlesselement.blank?
end
```

The first invocation of class defines a class; once a class is already defined, subsequent invocations modify that class. This code adds a methodcalled blank? totwoexistingclasses: NilClass and String. When I check the status of a given string, I often want to see whether it is blank.Moststringscanhaveavalue, beempty, and bepossibly nil.Thislittle idiomletsmequicklycheckforthetwoemptycasesatonce, because blank? willreturn true. It doesn't matter which class String points to. If itsupports the blank? method, it will work. If it walks like a duck and quacks like a duck, it is a duck. I don't need to draw blood to check thetype.

Withwhat's going on here. Yie asking for avery sharp scalpel, and Rubywillgladlygiveittoyou. Yr openclasseshaveredefinedboth String and Nil. It's possible to completely disable Ruby by redefining, say, Class.new.Thetrade-offisfreedom.Whthekindoffreedomthat letsyouredefineanyclassorobjectatanytime, you can build some amazinglyreadablecode. Whfreedomandpowercomeresponsibility.

Openclassesareusefulwhenyou'rebuildinglanguagestoencodeyour own domain. It's often useful to express units in a language that worksforyourbusinessdomain.Forexample,consideranAPIthatexpresses alldistanceasinches:

```
Download ruby/units.rb
classNumeric
  definches
    self.
  end
  deffeet
    self * 12.inches
  end
  defyards
    self * 3.feet
  end
  defmiles
    self * 5280.feet
  end
  defback
    self * -1
  end
  defforward
    self
  end
end
puts10.miles.back
puts2.feet.forward
```

Theopenclasses make this kind of support possible with minimal syntax.ButothertechniquescanstretchRubyevenfurther.

Vamethod_missing

Ruby calls a special debugging method each time a method is missinginordertoprintsomediagnosticinformation. This behavior makes thelanguageeasiertodebug. But sometimes, you can take advantage of this language feature to build some unexpectedly rich behavior. All youneedtodoisoverride method_missing.ConsideranAPItorepresent Roman numerals. You could do it easily enough with a method call, withanAPIsomethinglike Roman.number_for"ii" .Intruth,that'snottoo

bad. There are no mandatory parentheses or semicolons to get in the way. WhRuby, we can do better:

```
Download ruby/roman.rb
classRoman
 defself .method_missingname, *args
    roman=name.to_s
    roman.gsub!("IV", "IIII")
    roman.gsub!("IX", "VIIII")
    roman.gsub!("XL", "XXXX")
    roman.gsub!("XC", "LXXXX")
    (roman.count("I")+
     roman.count("V") * 5+
     roman.count("X") * 10+
     roman.count("L") * 50+
     roman.count("C") * 100)
  end
end
putsRoman.X
putsRoman.XC
putsRoman.XII
putsRoman.X
```

Thiscodeisabeautifulexampleof method_missing inaction. The code is firstoverride method_missing.We clearandsimple.We getthenameof the method and its parameters as input parameters. We interested onlyinthename.First, we convert that to String. Then, were place the specialcases,like iv and ix, with strings that are easier to count. Then, wejustcountRomandigitsandmultiplybythevalueofthatnumber. The API is somuch easier: Roman.i versus Roman.number for"i".

Consider the cost, though. We dohaveaclassthatwillbemuchmore difficult to debug, because Ruby cannolonger telly ouw henamethod ismissing!Wewoulddefinitelywantstrongerrorcheckingtomakesure itwasacceptingvalidRomannumerals.Ifyoudon'tknowwhatyou're lookingfor, you could have a tought ime finding that implementation of that ii methodon Roman. Still, it's another scalpel for the tool bag. Use itwisely.

Modules

ThemostpopularmetaprogrammingstyleinRubyisthemodule. Yu can literally implement defor attr_accessor with a few lines of code in amodule. You can also extend class definitions in surprising ways. A commontechniqueletsyoudesignyourowndomain-specificlanguage

(DSL)todefineyourclass. ⁴ TheDSLdefinesmethodsinamodulethat addsallthemethodsandconstantsneededtomanageaclass.

I'mgoingtobreakanexampledownusingacommonsuperclassfirst. Here's the type of class that we want to build through metaprogramming.It'sasimpleprogramtoopenaCSVfilebasedonthenameofthe class.

```
Download ruby/acts_as_csv_class.rb
classActsAsCsv
  defread
    file=File.new(self .class.to_s.downcase+ '.txt')
    @headers=file.gets.chomp.split( ',')
    file.eachdo|row|
      @result<<row.chomp.split( ',')</pre>
    end
  end
  defheaders
    @headers
  end
  defcsv_contents
    @result
  end
  definitialize
    @result=[]
    read
  end
end
classRubyCsv<ActsAsCsv</pre>
end
m=RubyCsv.new
putsm.headers.inspect
putsm.csv_contents.inspect
```

Thisbasic class defines four methods. headers and csv_contents aresimpleaccessorsthatreturnthevalueofinstancevariables. initialize initializestheresultsoftheread.Mostoftheworkhappensin read.The read method opens a file, reads headings, and chops them into individual fields. Next, it loops overlines, placing the contents of each line in an

^{4.} DSLs let you tailor a language for a specific domain. For perhaps the best-known example in Ruby, the ActiveRecord persistence framework uses domain-specific languagestomapaclasstoadatabasetable.

array. This implementation of a CSV file is not complete because it does nothandleedgecaseslikequotes, butyouget theidea.

Thenextstepistotakethefileandattachthatbehaviortoaclasswith amodulemethodoftencalleda macro. Macroschangethebehavior of classes, often based on changes in the environment. In this case, our $macro\, open sup \, the \, class \, and \, dumps in \, all \, the \, behavior \, related \, to \, a$ CSVfile:

```
Download ruby/acts_as_csv.rb
classActsAsCsv
 defself .acts as csv
    define method 'read' do
      file=File.new(self .class.to_s.downcase+ '.txt')
      @headers=file.gets.chomp.split( ',')
      file.eachdo|row|
        @result<<row.chomp.split( ',')</pre>
      end
    end
    define_method "headers" do
      @headers
    end
    define_method "csv_contents" do
      @result
    end
    define method 'initialize' do
      @result=[]
      read
    end
 end
end
classRubyCsv<ActsAsCsv</pre>
 acts_as_csv
end
m=RubyCsv.new
putsm.headers.inspect
putsm.csv_contents.inspect
```

The metaprogramming happens in the acts_as_csv macro. That code calls define_method for all the methods we want to add to the target class.Now,whenthetargetclasscalls acts_as_csv,thatcodewilldefine allfourmethodsonthetargetclass.

So, the acts_as macro code does nothing but add a few methods we could have easily added through inheritance. That design does not seem like much of an improvement, but it's about to get more interesting.Let's see how the same behavior would work in a module:

```
Download ruby/acts_as_csv_module.rb
moduleActsAsCsv
 defself .included(base)
   base.extendClassMethods
  end
 moduleClassMethods
    defacts_as_csv
      includeInstanceMethods
    end
 end
 moduleInstanceMethods
    defread
      @csv_contents=[]
      filename=self .class.to_s.downcase+ '.txt'
      file=File.new(filename)
      @headers=file.gets.chomp.split( ',')
      file.eachdo|row|
        @csv_contents<<row.chomp.split( ',')</pre>
      end
    end
    attr_accessor:headers,:csv_contents
    definitialize
      read
    end
 end
end
classRubyCsv #noinheritance!Youcanmixitin
 includeActsAsCsv
 acts_as_csv
end
m=RubyCsv.new
putsm.headers.inspect
putsm.csv_contents.inspect
```

Ruby will invoke the included method whenever this module gets included into another. Remember, a class is a module. In our included method, we extend the target class called base (which is the RubyCsv class), and that module adds class methods to RubyCsv.Theonlyclass method is acts_as_csv. That method in turn opens up the class and includes all the instance methods. And we're writing a program that writesaprogram.

Theinterestingthingaboutallthese metaprogramming techniques is thatyour programs can change based on the state of your application. ActiveRecordusesmetaprogrammingtodynamicallyaddaccessorsthat are the same name as the columns of the database. Some XML frameworkslike builder let users define custom tags with method_missing to provide a beautiful syntax. When your syntax is more beautiful, you can let the reader of your code get past the syntax and closer to the intentions. That's the power of Ruby.

WhatWeLearnedinDay3

Inthissection, youlearned to use Ruby to define your own syntax and change classes on the fly. These programming techniques fall in the category of metaprogramming. Every line of code that you write has twokindsofaudiences:computersandpeople.Sometimes,it'shardto strikeabalancebetweenbuildingcodethatcanpassthroughtheinterpreterorcompilerandisalsoeasytounderstand. Whmetaprogramming, you can close the gap between valid Rubys yntax and sentences.

SomeofthebestRubyframeworks, such as Builder and Active Record, heavily depend on metaprogramming techniques for readability. You usedopenclassestobuildaduck-typedinterfacesupportingthe blank? method for String objects and nil, dramatically reducing the amount of clutterforacommonscenario. Yusawsomecodethatusedmanyof thesametechniques.Yuused method_missing tobuildbeautifulRoman numerals. And finally, you used modules to define a domain-specific languagethatyouusedtospecifyCSVfiles.

Day3Self-Study

Do:

Modify the CSV application to support an each method to return a CsvRow object. Use method_missing onthat CsvRow to return the value forthecolumnforagivenheading.

Forexample, for the file:

```
one.two
lions, tigers
allowanAPIthatworkslikethis:
csv=RubyCsv.new
csv.each{|row|putsrow.one}
This should print "lions".
```

WrappingUpRuby 2.5

We have covered a lot of ground in this chapter. I hope you can see the comparison to Mary Poppins. After speaking at dozens of Ruby conferences, I have heard scores of people profess their love for Ruby becauseitisfun. Toanindustrythat grewupembracing the Cfamily oflanguagesincludingC++,C#,Java,andothers,Rubyisabreathof freshair.

CoreStrengths

Ruby'spureobjectorientationallowsyoutotreatobjectsinauniform and consistent way. The duck typing allows true rpolymorphic designs basedonwhatanobjectcan supportrather than that object's inheritancehierarchy. And Ruby's modules and open classes let a programmerattachbehaviortosyntaxthatgoesbeyondthetypicalmethodor instancevariabledefinitionsinaclass.

Rubyisidealasascriptinglanguage, orasawebdevelopmentlanguage if the scaling requirements are reasonable. The language is intensely productive. Some of the features that enable that productivity make Rubyhardtocompileandmaketheperformancesuffer.

Scripting

Rubyisafantasticscriptinglanguage. Waggluecodetomungetwo applicationstogether, writing aspider to scrape webpages for a stock quoteorbookprice, orrunning local build en vironments or automated testsareexcellentusesforRuby.

Asalanguagewithapresenceonmostmajoroperatingsystems, Ruby is a good choice for scripting environments. The language has a wide varietyoflibrariesincludedwiththebase, aswellasthousandsofgems, or prepackaged plug-ins, that can be used for loading CSV files, processingXML,orworkingwithlow-levelInternetAPIs.

WebDevelopment

Rails is already one of the most successful web development frameworksofalltime. The design is based on well-understood model-viewcontroller paradigms. The many naming conventions for database and application elements allow a typical application to be built with few linesofconfigurationatall. And the framework has plug-inst hathandlesomedifficultproductionissues:

- The structure of Rails applications is always consistent and well understood.
- Migrationshandlechangesinthedatabaseschema.
- Severalwell-documented conventions reduce the total amount of configurationcode.
- Manydifferentplug-insareavailable.

TimetoMarket

IwouldconsidertheproductivityofRubyandRailstobeanimportant componentinits success. In the mid-2000s, you could not throw arock in San Francisco without hitting someone who worked at a start-up poweredbyRails.Eventoday,Rubyisprolificinthesekindsofcompanies, including mine. The combination of the beautiful syntax and the communityofprogrammers, tools, and plug-insisex tremely powerful. Yu can find Ruby gems to find the ZIP code of a surfer and another to calculate all address codes in a fifty-mile radius. Yu can process images and credit cards, work with web services, and communicate acrossmanyprogramminglanguages.

Many large, commercial websites use Ruby and Ruby on Rails. The original Twitter implementation was in Ruby, and the extraordinary productivity of the language allowed the website to grow to huge proportions. Eventually, the core of Twitterwas rewritten in Scala. There aretwolessonshere. First, Rubyisagreatlanguage forgetting aviable producttomarketquickly. Second, the scalability of Rubyislimited in someways.

Informal big enterprises with distributed transactions, fail-safemessaging, and internationalization, the role of Ruby is often seen as a little more limited, but Ruby can do all of these things. Sometimes, concerns about the right application frameworks and scalability are well-founded, but too many people focus on enough scalability to build thenexteBaywhentheycan'tdeliveranysoftwareontime.Often,Ruby

wouldbemorethanadequateconsideringthetime-to-marketpressures manyenterprisesface.

Wedkress

Nolanguageisperfectforallapplications. Rubyhasits share of limitationstoo.Let'swalkthroughsomeofthemajorones.

Rémone

Ruby'sprimaryweaknessisperformance.Sure, Rubyisgettingfaster. 1.9 is up to tentimes faster for some usecases. A new Rubyvir-Vasion tualmachinewrittenbyEvanPhoenixcalledRubiniushasthepotential tocompileRubyusingajust-in-timecompiler.Thisapproachlooksat aninterpreter's usage patterns for a block of code to anticipate which codeislikelytobeneededagain. This approachworks well for Ruby, a languagewheresyntaxcluesareusuallynotenoughtoallowcompilation. Remember, the definition of a class can change at any time.

Still, Matzisvery clear. Heworks to optimize the programmer's experience, not the performance of the language. Many of the language's features such as open classes, duck typing, and method missing defeat theverytoolsthatenablecompilationandtheassociatedperformance gains.

ConcurrencyandOOP

Object-orientedprogramminghasacriticallimitation. The whole premiseofthemodeldependsonwrappingbehavioraroundstate, and usually the state can be changed. This programming strategy leads to serious problems with concurrency. At best, significant resource contentionsarebuiltintothelanguage. Atworst, object-oriented systems are next to impossible to debugand cannot be reliably tested for concurrent environments. As of this writing, the Rails team is only now startingtoaddresstheproblemofmanagingconcurrencyeffectively.

TypeSafety

I'm a firm believer in duck typing. What his typing strategy, you can generally have cleaner abstractions with concise, readable code. But ducktypingcomesataprice,too.Statictypingallowsawholerangeof tool s that make it easier to do syntax trees and thus provide integrateddevelopment environments. IDEs for Ruby are more difficult to build, and so far, most Ruby developers do not use them. Many times, I've lamentedthelossofanIDE-styledebugger.IknowI'mnotalone.

FinalThoughts

So, Ruby's corestrengths are its syntax and flexibility. The coreweaknesses are around performance, though the performance is reasonableformanypurposes. Allinall, Rubyisan excellent language for objectoriented development. For the right applications, Ruby can excel. As withanytool, use it to solve the right set of problems, and you're not likely to be disappointed. And keep your eyes open for a little magic alongtheway.

► FerrisBueller

Chapter 3

I_0

 $\label{thm:problem} Meet Io. Like Ruby, Ioisarule bender. He's young, wicked smart, and easy to understand but hard to predict. Think Ferris Bueller. \\ ^1 If you like a good party, you'll have to let Ioshow you around the town. He'll try anything once. He might give you the ride of your life, wreck your dad's car, or both. Either way, you will not be bored. As the quote above says, you won't have many rules to hold you back. \\$

3.1 Introducinglo

SteveDekorteinventedtheIolanguagein2002.It'salwayswrittenwith anuppercase $\it I$ followedbyalowercase $\it o$.Ioisaprototypelanguagelike LuaorJavaScript,meaningeveryobjectisacloneofanother.

Whasanexercise to help Steveunderstand how interpreters work, Iostarted as a hobby ist language and remains pretty small to day. You can learn the syntax in about fifteen minutes and the basic mechanics of the language in thirty. There are no surprises. But the libraries will take you a little longer. The complexity and the richness comes from the library design.

Today, most of Io's community is focused on Io as an embeddable language with a tiny virtual machine and rich concurrency. The core strengths are richly customizable syntax and function, as well as a strongconcurrency model. Try tofocus on the simplicity of the syntax and the prototype programming model. I found that after Io, I had a much stronger understanding of how Java Scriptworked.

^{1.} Ferris Bueller's Day Off $\,$. DVD. Directed by John Hughes. 1986; Hollywood, CA: Paramount. 1999.

3.2 Day 1: Skipping School, Hanging Out

Meetingloislikemeetinganylanguage. Whavetoputinalittlekeyboardtimetogetproperlyacquainted. It will be much easier if we can interact outside of stifled conversations in the hallway before history class.Let'scutschoolandskipstraighttothegoodstuff.

Names are sometimes deceiving, but you can tell a lot from Io. It's Io?)2 and brilliant. You simultaneously reckless (ever try Googling for getonlytwoletters, bothvowels. The language's syntaxis simple and low-level, like the name. Io syntax simply chains messages together, with each message returning an object and each message taking optional parameters in parentheses. In Io, everything is a message that returns another receiver. There are no keywords and only a handful of charactersthatbehavelikekeywords.

Wh Io, you won't worry about both classes and objects. Yil deal exclusivelyinobjects, cloning them as needed. The seclones are called prototypes, and Ioisthe first and only prototype-based language we'll prototype language, every object is a clone of an existing lookat.Ina objectratherthanaclass. Iogetsyouaboutas close to object-oriented Lisp as you're likely to get. It's too early to tell whether Io will have lastingimpact, but the simplicity of the syntax means it has a fighting chance. The concurrency libraries that you'll see inday 3 are well conceived, and themessages emantics are elegant and powerful. Reflection iseverywhere.

Breakingthelce

Let's crack open the interpreter and start the party. Yu can find it at http://iolanguage.com. Download it and install it. Open the interpreter bytyping io, and enter the traditional "Hello, Will" program:

```
Io>"Hiho, Io"print
Hiho, Io==>Hiho, Io
```

Yucantellexactlywhat's going on here. Yie sending the print messagetothestring "Hiho,lo" .Receiversgoontheleft,andmessagesgoon theright. You won't find much syntactic sugaratall. Yil justsend messagestoobjects.

InRuby, you created a new object by calling new on some class. You createdanewkindofobjectbydefininga class.Iomakesnodistinction

^{2.} Try Googlingfor Iolanguage instead.

between these two things. Yalcreatenew objects by cloning existing ones. The existing object is a prototype:

```
batate$io
Io20090105
Io>Vehicle:=Objectclone
==> Vehicle 0x1003b61f8:
                  ="Vehicle"
  type
```

Object is the root-level object. We send the clone message, which returnsanewobject.We assignthatobjectto \\epsilon \text{hicle.Here, \\epsilon hicle isnot} is anobject, based aclass.It'snotatemplateusedtocreateobjects.It onthe Object prototype.Let'sinteractwithit:

```
Io>Vehicledescription:="Somethingtotakeyouplaces"
==>Somethingtotakeyouplaces
```

Objectshaveslots. Thinkofthe collection of slots as a hash. Yolr efer toeachslotwithakey. Yucanuse := toassignsomethingtoaslot.If theslotdoesn'texist.Jowillcreateit.Yucanalsouse = forassignment. If the slot doesn't exist, I othrows an exception. We justcreatedaslot called description.

```
Io>Vehicledescription="Somethingtotakeyoufaraway"
==>Somethingtotakeyoufaraway
Io>VehiclenonexistingSlot="Thiswon'twork."
  Exception:SlotnonexistingSlotnotfound.
   Mustdefineslotusing:=operatorbeforeupdating.
 message'updateSlot'in'CommandLine'online1
```

You can get the value from a slot by sending the slot's name to the object:

```
Io>Vehicledescription
==>Somethingtotakeyoufaraway
```

Infact, anobject is little more than a collection of slots. We canlookat thenamesofalltheslotson Ve hicle likethis:

```
To>VehicleslotNames
==>list("type", "description")
```

We sent the slotNames method to \& hicle and got a list of slot names back. There are two slots. Yorkeseen the description slot.butwealso havea type slot. Everyobject supports type:

```
Io>Vehicletype
==>Vehicle
Io>Objecttype
==>Object
```



thiscopyis (P1.0printing,October2010)

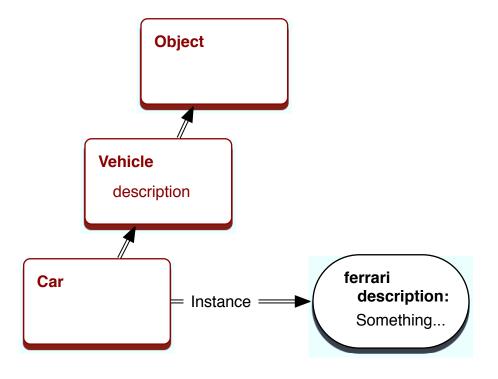


Figure 3.1: An object-oriented design

- ₩ŧ gettotypesinafewparagraphs. Fornow, know that type representsthekindofobjectyou'redealingwith.Keepinmindthatatypeis anobject, notaclass. Here's what we knows of ar:
 - Yumakeobjects by cloning other objects.
 - Objects are collections of slots.
 - Yugetaslot's value by sending themes sage.

YucanalreadyseethatIoissimpleandfun.Butsitback.We only scratchingthesurface. Let's move onto inheritance.

Objects, Prototypes, and Inheritance

Inthissection, we'regoing to deal within heritance. Given a carthat's also avehicle, think of how you would model a Écci objectthatisan instance of a car. In an object-oriented language, you'd do something likeFigure3.1.

Let's see how you'd solve the same problem in a prototype language.

goingtoneedafewextraobjects.Let'screateanother: Water

```
Io>Car:=Vehicleclone
==> Car_0x100473938:
                   ="Car"
 type
Io>CarslotNames
==>list("type")
Io>Cartype
==>Car
```

In Io-speak, we created a new object called Car by sending the clone messagetothe \& hicle prototype.Let's send description to Car:

```
Io>Cardescription
==>Somethingtotakeyoufaraway
```

There's no description sloton Car. Iojust forwards the description mes-Vé hicle. It's dead simple but sagetotheprototypeandfindstheslotin plentypowerful. Let's create another car, but this time, we'll assignit to Exci

```
Io>ferrari:=Carclone
==> Car_0x1004f43d0:
Io>ferrarislotNames
==>list()
```

A-ha!There'sno type slot.Byconvention,typesinIobeginwithuppercaseletters. Now, when you invoke the type slot, you'll get the type of yourprototype:

```
Io>ferraritype
==>Car
```

Thisishowlo'sobjectmodelworks. Objects are just containers of slots. Getaslotbysendingitsnametoanobject. If the slotisn't there, I ocalls the parent. That's all you have to understand. There are no classes or metaclasses. You don't have interfaces or modules. You just have objects, likeyouseein Figure 3.2, on the next page.

TypesinIoarejustconveniences. Idiomatically, an object that begins withanuppercasenameisatype, so Iosetsthe type slot.Anyclonesof thattypestartingwithlowercaseletterswillsimplyinvoketheirparents' type slot. Types are just tools that help Ioprogrammers better organize code.

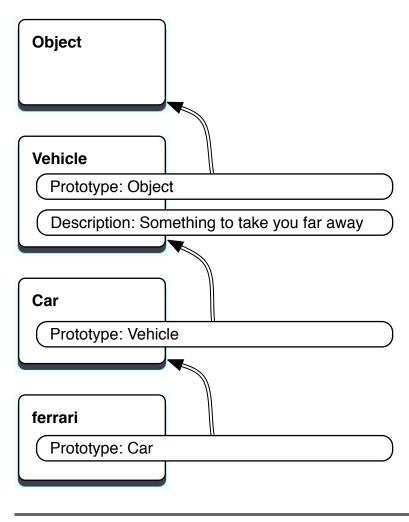


Figure 3.2: Inheritance in Io

Ifyouwanted **Exi** tobeatype, you would have it begin with an uppercaseletter.likethis:

```
Io>Ferrari:=Carclone
==>Ferrari_0x9d085c8:
type="Ferrari"
Io>Ferraritype
==>Ferrari
Io>FerrarislotNames
==>list("type")
Io>ferrarislotNames
==>list()
Io>
```

Noticethat for hasnotypeslot,but does.Wate Fecti usingasimple coding convention rather than a full language feature to distinguish between types and instances. In other cases, they behave the same way.

InRubyandJava, classes are templates used to create objects. bruce =Resomew createsanew person objectfrom the Reson class.Theyare different entities entirely, a class and an object. Not so in Io. bruce:= createsaclonecalled bruce from the prototype called Reson **Paranclone** isatypebecauseithasatype Both bruce and Reson areobjects. Reson slot.Inmostotherrespects, isidenticalto bruce.Let'smoveon Reason tobehavior.

Methods

InIo, you can create a method easily, like this:

```
Io>method("So,you'vecomeforanargument."println)
==>method(
    "So, you'vecomeforanargument."println
```

Amethodisanobject, justlike anyother type of object. Yucangetits type:

```
Io>method()type
==>Block
```

Sinceamethodisanobject, wecan assignit to a slot:

```
Io>Cardrive:=method("Vroom"println)
==>method(
    "Vroom"println
)
```

Ifaslothasamethod, invoking the slotin vokes the method:

```
Io>ferraridrive
Vroom
==>Vroom
```

Believe it or not, you now know the core organizational principles of Io. Think about it. You know the basic syntax. You can define types and objects. Yucan add data and behavior to an object by assigning contentstoitsslots. Everythingelseinvolveslearningthelibraries.

Let's digalittle deeper. You can get the contents of slots, whether they arevariablesormethods.likethis:

```
Io>ferrarigetSlot("drive")
==>method(
    "Vroom"println
)
```

getSlot willgiveyouyourparent's slotifthe slotdoesn't exist:

```
Io>ferrarigetSlot("type")
==>Car
```

Yucangettheprototypeofagivenobject:

```
Io>ferrariproto
==> Car 0x100473938:
 drive
                =method(...)
               ="Car"
 type
Io>Carproto
==> Vehicle_0x1003b61f8:
 description = "Somethingtotakeyoufaraway"
                 ="Vehicle"
 type
```

These were the prototypes that you used to clone ÉCTÍ and Cor. You also see their customs lots for convenience.

There's a masternames pacecalled Lobby that contains all the named objects. All of the assignments you've done in the console, plus a few more, areon Lobby. Yucanseeitlikethis:

```
Io>Lobby
```

```
==> Object 0x1002184e0:
               =Car_0x100473938
  Car
               =0bject_0x1002184e0
=0bject_0x1002184e0
=Vehicle_0x1003b61f8
  Lobby
 Protos
  Vehicle
  exit
                  =method(...)
  ferrari
                  =Car_0x1004f43d0
  forward
                    =method(...)
```

Yuseethe exit implementation, toward , Protos, and the stuffwed efined.

The prototype programming paradigm seems clear enough. These are thebasicgroundrules:

- Every *thing* isanobject.
- Every *interaction* withan object is a message.
- Yudon'tinstantiateclasses; you clone other objects called prototypes.
- Objects remember their prototypes.
- Objectshaveslots.
- Slotscontainobjects, including methodobjects.
- Amessagereturnsthevalueinaslotorinvokesthemethodina slot.
- Ifanobjectcan't respond to a message, its ends that message to itsprototype.

Andthat's most of it. Since you can see or change any slotor any object, youcandosomeprettysophisticatedmetaprogramming.Butfirst,you needtoseethenextlayerofbuildingblocks:collections.

ListsandMaps

Iohasafewtypesofcollections. Alistisan ordered collection of objects of any type. List is the prototype for all lists, and Map istheprototype forkey-valuepairs, like the Rubyhash. Createalist like this:

```
Io>toDos:=list("findmycar","findContinuumTransfunctioner")
==>list("findmycar", "findContinuumTransfunctioner")
Io>toDossize
==>2
Io>toDosappend("Findapresent")
==>list("findmycar", "findContinuumTransfunctioner", "Findapresent")
```

There's a shortcut way of representing a list. Object supports the list method, which wraps the argument supinto a list. Using list, you can convenientlycreatealist, likethis:

```
Io>list(1,2,3,4)
==>list(1,2,3,4)
```

List also has convenience methods formath and to deal with the list as otherdatatypes, such as stacks:

```
Io>list(1,2,3,4)average
==>2.5
Io>list(1,2,3,4)sum
==>10
Io>list(1,2,3)at(1)
==>2
Io>list(1,2,3)append(4)
==>list(1,2,3,4)
Io>list(1,2,3)pop
==>3
Io>list(1,2,3)prepend(0)
==>list(0,1,2,3)
Io>list()isEmpty
==>true
```

TheothermajorcollectionclassinIoisthe Map.IomapsarelikeRuby hashes. Sincethere's no syntactic sugar, you'll work with them with an APIthatlookslikethis:

```
Io>elvis:=Mapclone
==> Map_0x115f580:
Io>elvisatPut("home", "Graceland")
==> Map_0x115f580:
Io>elvisat("home")
==>Graceland
Io>elvisatPut("style","rockandroll")
==> Map_0x115f580:
Io>elvisasObject
==> Object_0x11c1d90:
 home ="Graceland"
style ="rockandroll"
Io>elvisasList
==>list(list("style", "rockandroll"), list("home", "Graceland"))
Io>elviskeys
==>list("style", "home")
```

```
Io>elvissize
==>2
```

Whenyouthinkaboutit, ahashisalotlikean Ioobjectinstructure where the keys are slots that are tied to values. The combination of slotsthatcanberapidlytranslatedtoobjectsisaninterestingone.

Now that you've seen the basic collections, you'll want to use them. We need to introduce control structures, and those will depend on booleanvalues.

true, denil, and singletons

Io's conditions are pretty similar to those of other object-oriented languages. Hereareafew:

```
Io>4<5
==>true
Io>4<=3
==>false
Io>trueandfalse
==>false
Io>trueandtrue
==>true
Io>trueortrue
==>true
Io>trueorfalse
==>true
Io>4<5and6>7
==>false
Io>trueand6
==>true
Io>trueand0
==>true
```

That's simple enough. Makeanote: 0 is true as in Ruby, not false as in C.So, whatis true?

```
Io>trueproto
==> Object_0x200490:
                    =Object_()
                    =0bject_!=()
  !=
. . .
Io>trueclone
==>true
Io>falseclone
==>false
Io>nilclone
==>nil
```

Now, that's interesting! true, the , and nil are singletons. Cloning them just returns the singleton value. You can do the same pretty easily. Createyourownsingletonlikethis:

```
Io>Highlander:=Objectclone
==> Highlander_0x378920:
 type
                  ="Highlander"
Io>Highlanderclone:=Highlander
==> Highlander_0x378920:
          =Highlander_0x378920
 clone
 type
                  ="Highlander"
```

Wore simply redefined the clone method to return Highlander, rather thanlettingloforwardrequestsupthetree, eventually getting to Object. Now, when you use Highlander, you'll get this behavior:

```
==> Highlander_0x378920:
 clone
                  =Highlander_0x378920
                  ="Highlander"
  type
Io>fred:=Highlanderclone
==> Highlander_0x378920:
 clone
                  =Highlander_0x378920
 type
                  ="Highlander"
Io>mike:=Highlanderclone
==> Highlander_0x378920:
 clone
                 =Highlander_0x378920
               ="Highlander"
```

Twoclonesareequal. That's not generally true:

```
Io>one:=Objectclone
==> Object_0x356d00:
Io>two:=Objectclone
==> Object_0x31eb60:
Io>one==two
==>false
```

type

Io>fred==mike ==>true

Io>Highlanderclone

Now, there can be only one Highlander. Sometimes, Iocantripyouup. This solution is simple and elegant, if a little unexpected. Wee blasted through a lot of information, but you know enough to do some pretty radical things, such as changing an object's clone method to make a singleton.

Be careful, though. Love him or hate him, you can't deny that Io is interesting. As with Ruby, Iocanbealove-haterelationship. You can changejustaboutanyslotonanyobject, even the ones that define the language. Here's one that you may not want to try:

Objectclone:="hosed"

Since you overrode the clone method on object, nothing can create objectsanymore. Yucan't fixit. Yujusthave to kill the process. But youcanalsogetsomeprettyamazingbehaviorsinashorttime. Since youhavecompleteaccesstooperatorsandtheslotsthatmakeupany object, you can build domain-specific languages with a few short fascinatinglines of code. Before we wrap up the day, let's hear what the inventorofthelanguagehastosay.

AnInterviewwithSteveDekorte

SteveDekorteisanindependentconsultantintheSanFranciscoarea. Heinvented Ioin 2002. Ihad the pleasure of interviewing him about hisexperienceswithcreatingIo.

Bruce Tate: WhydidyouwriteIo?

Steve Dekorte: In 2002, my friend DruNelson wrote alanguage called Cel(inspiredbySelf)andwasaskingforfeedbackonitsimplementation. Ididn'tfeellunderstoodhowprogramminglanguagesworkwellenough to have anything useful to say, so I started writing a small language tounderstandthembetter.ItgrewintoIo.

Bruce Tate: Whatisthethingthatyoulikeaboutitthemost?

Steve Dekorte: Ilikethesimpleandconsistentsyntaxandsemantics. Theyhelpwithunderstandingwhat'sgoingon. Yu canquicklylearnthe basics. Ihaveaterrible memory. Iconstantly forget the syntax and weird semanticrulesforCandhavetolookthemup.(ed.SteveimplementedIo inC.)That's one of the things I don't want to down the I use Io.

For example, you can see the code, such as people select(age > 20) map(adabb)println ,andgetaprettygoodideaofwhatisgoingon. Yie filteringalistofpeoplebasedonage, getting their addresses, and printingthemout.

If you simplify the semantics enough, things become more flexible. You canstarttocomposethingsthatyoudidnotunderstandwhenyouimplementedthelanguage. Here's an example. There are vide og amesthatare puzzlegamesthatassumeasolution, and there are more gamest hat are

open-ended. The open-ended ones are fun because you can do things thatthedesignersofthegameneverimagined. Ioislikethat.

Sometimesotherlanguagesmakesyntacticshortcuts. Thatleadstoextra parsing rules. When you program in a language, you need to have the parserinyourhead. The more complicated a language, the more of the parseryouneedtohaveinyourhead. Themorework aparser has todo, themoreworkyouhavetodo.

Bruce Tate: Whataresomelimitationsoflo?

Steve Dekorte: The cost of Io's flexibility is that it can be slower for manycommonuses. That said, it also has certain advantages (such as coroutines, async sockets, and SIMD support), which can also make it much faster than even Capps written with traditional thread persocketconcurrencyornon-SIMDvectorops.

I've also had some complaints that the lack of syntax can make quick visualinspectiontrickier. I'vehadsimilar problems with Lisp, so I understand.Extrasyntaxmakesforquick**xdy**Newuserssometimessay Iohastoolittlesyntax,buttheyusuallywarmuptoit.

Bruce Tate: Whereisthestrangestplaceyou'veseenJoinproduction?

Steve Dekorte: Overtheyears, I've heard rumors of Io from place to place like on a satellite, in a router configuration language, and as a scriptinglanguage for videogames. Pixaruses it too. They wrote ablog entryaboutit.

Itwasabusyfirstday,soit'stimetobreakforalittlebit.Yucannow pauseandputsomeofwhatyou'velearnedintopractice.

WhatWeLearnedinDay 1

Yone nowthroughagoodchunkoflo.Sofar,youknowagooddeal about the basic character of lo. The prototype language has very simple syntaxthatyou can use to build new basic elements of the language itself. Even core elements lack even the simplest syntactic sugar. In someways, this minimal approach will make you work a little harder toreadthesyntax.

Aminimal syntax has some benefits as well. Since there is not much going on syntactically, you don't have to learn any special rules or exceptions to them. Once you know how to read one sentence, you canreadthemall. Yarlearning time can go toward establishing your vocabulary.

Yrjobasanewstudentisgreatlysimplified:

- Understandafewbasicsyntacticalrules.
- Understandmessages.
- Understandprototypes.
- Understandthelibraries.

Day 1 Self-Study

Whenyou'relookingforIobackground, searchingforanswersisgoing to be a little tougher because Io has so many different meanings. I recommendGooglingfor Iolanguage.

Find:

- SomeIoexampleproblems
- AnIocommunitythatwillanswerquestions
- AstyleguidewithIoidioms

Answer:

- Evaluate 1+1 and then 1+"one". Is Io strongly typed or weakly typed?Supportyouranswerwithcode.
- Is0trueorfalse?Whatabouttheemptystring?Is nil trueorfalse? Supportyouranswerwithcode.
- Howcanyoutellwhatslotsaprototypesupports?
- Whatisthedifferencebetween = (equals), := (colonequals), and ::= (coloncolonequals)? When would you use each one?

Do:

- RunanIoprogramfromafile.
- Execute the code in a slot given its name.

Spendalittle time playing with slots and prototypes. Make sure you understandhowprototypeswork.

3.3 Day2:TheSausageKing

Thinkback to Ferris Bueller for a moment. In the movie, the middleclass high-school student represented himself as the sausage king of Chicago in a classic bluff. He got a great table in a great restaurant

becausehewaswillingtobendtherules.Ifyou'recomingfromaJava background and you liked it, you're thinking about what could have happened-toomuchfreedomisnotalwaysagoodthing.Buellerprobably deserved to be thrown out. In Io, you're going to need to relax a little and take advantage of the power. If you're coming from a Perl scriptingbackground, you probably liked Bueller's bluff because of the resultofthebluff. If you've traditionally played things a little fast and loose, you're going to have to pull back a little and inject some discipline. In day 2, you'll start to see how you might use Io's slots and messagestoshapecorebehaviors.

ConditionalsandLoops

All of Io's conditional statements are implemented without syntactifind they are easy to understand and remember, but calsugar. Yol they'realittlehardertoreadbecauseoffewersyntacticclues. Setting upasimpleinfiniteloopiseasy.TypeControl+Ctobreakout:

```
Io>loop("gettingdizzy..."println)
gettingdizzy...
gettingdizzy...
gettingdizzy.^C
IoVM:
        Receivedsignal.Settinginterruptflag.
```

Loops will often be useful with the various concurrency constructs, butyou'llnormallywanttochooseoneoftheconditionalloopingconwhile loop.A while looptakesaconditionandamesstructs, such as a sagetoevaluate. Keepinmindthatasemicolonconcatenatestwodistinctmessages:

```
Io>i:=1
==>1
Io>while(i<=11,iprintln;i=i+1);"Thisonegoesuptol1"println</pre>
1
2
10
11
Thisonegoesupto11
```

You could do the same with a to loop. The to loop takes the name of the counter, the first value, the last value, an optional increment, and amessagewithsender.

```
Io>for(i,1,11,iprintln);"Thisonegoesuptol1"println
2
10
11
Thisonegoesupto11
==>Thisonegoesupto11
```

Andwiththeoptionalincrement:

```
Io>for(i,1,11,2,iprintln);"Thisonegoesuptol1"println
3
5
7
9
11
Thisonegoesupto11
==>Thisonegoesupto11
```

Infact, you can often have an arbitrary number of parameters. Didyou catch that the optional parameter is the third one? Io will allow you toattachextraparameters. That may seem to be convenient, but you need to watch carefully because the reisno compiler to baby sity ou:

```
Io>for(i,1,2,1,iprintln,"extraargument")
1
2
==>2
Io>for(i,1,2,iprintln,"extraargument")
==>extraargument
```

Inthefirstform, "extraargument" is really extra. In the second form, you've omitted the optional increment argument, and that effectively shiftedeverythingtotheleft. Yur "extraargument" is now themessage, and you're working in steps of iprintln, which returns i. If that line of code is buried deeply into a complex package, Io just puked in your car. Sometimes, you have to take the bad with the good. Io gives you freedom.Sometimes,freedomhurts.

The $\,$ if $\,$ control structure is implemented as a function with the form .The function will execute if(condition, true code, the code) truecode if condition istrue:otherwise.itwillexecute **decode**

```
Io>if(true,"Itistrue.","Itisfalse.")
==>Itistrue.
```

```
Io>if(false)then("Itistrue")else("Itisfalse")
Io>if(false)then("Itistrue."println)else("Itisfalse."println)
Itisfalse.
==>nil
```

Ymespentsometimeoncontrolstructures. Now, we can use them to developourownoperators.

Operators

Like with object-oriented languages, many prototype languages allow forsyntactic sugar to allow operators. These are special methods like + and / thattakeaspecialform.InIo,youcanseetheoperatortable directly, like this:

```
Io>OperatorTable
==>0peratorTable_0x100296098:
Operators
  0 ?@@@
  1
      **
  2 % * /
  3 +-
  4 <<>>
  5 <<=>>=
  6!===
  7 &
  8 ^
  9
  10 &&and
  11 or | |
  12 ...
  13 %=&=
          *=+=-=/=<<=>>=^=|=
  14 return
AssignOperators
  ::=newSlot
  := setSlot
  = updateSlot
Toaddanewoperator:OperatorTableaddOperator("+",4)
    andimplementthe+message.
Toaddanewassignoperator:OperatorTableaddAssignOperator(
    "=", "updateSlot") and implement the updateSlotmessage.
```

You can see that an assignment is a different kind of operator. The number to the left shows the level of precedence. Arguments closer to 0 bind first. You can see that + evaluates before ==, and * before +, justasyouwouldexpect. Yucanoverride the precedence with ().Let's defineanexclusiveoroperator.Our xor returns true ifexactlyoneofthe

thiscopyis (P1.0printing,October2010)

argumentsis true, and the otherwise. First, weadd the operator to the table:

```
Io>OperatorTableaddOperator("xor",11)
==>0peratorTable_0x100296098:
Operators
  . . .
  10 &&and
  11 orxor||
  12 ...
  . . .
```

Yucanseethenewoperatorintherightplace. Next, we need to implementthe xor methodon true and the :

```
Io>truexor:=method(bool,if(bool,false,true))
==>method(bool,
    if(bool,false,true)
)
Io>falsexor:=method(bool,if(bool,true,false))
==>method(bool,
    if(bool,true,false)
)
```

Wate using brute force here to keep the concepts simple. Our operatorbehavesexactlylikeyouwouldexpect:

```
Io>truexortrue
==>false
Io>truexorfalse
==>true
Io>falsexortrue
==>true
Io>falsexorfalse
==>false
```

Whenallissaidanddone, truexortrue getsparsedas truexor(true). The method in the operator table determines the order of precedence andthesimplifiedsyntax.

Assignment operators are in a different table, and they work a little bit differently. Assignment operators work as messages. Yil seean example of the minaction in Section 3.4, Domain-SpecificLanguages, on page 83. For now, that's all we'll say about operators. Let's move on to messages, where you will learn to implement your own control structures.

Messages

calltarget

)

As I was working through this chapter, one of the Io committers was helpingmethroughamoment of frustration. He said, "Bruce, there's something you have to understand about Io. Almost everything is a message."IfyoulookatIocode, everything but comment markers and thecomma(,)between arguments are messages. Everything. Learning Iowell means learning to manipulate them beyond just basic invocation. One of the most crucial capabilities of the language is message can query any characteristic of any message and act reflection. **Y**u appropriately.

Amessagehasthreecomponents: the sender, the target, and the arguments.InIo.the sender sendsamessagetoa target. The target executes themessage.

The call method gives you access to the meta information about any message. Let's create a couple of objects: the postOffice thatgetsmessagesandthe moder thatdeliversthem:

```
Io>postOffice:=Objectclone
==> Object_0x100444b38:
Io>postOfficepackageSender:=method(callsender)
==>method(
    callsender
)
Next, I'llcreate the mailer to deliver a message:
Io>mailer:=Objectclone
==> Object_0x1005bfda0:
Io>mailerdeliver:=method(postOfficepackageSender)
==>method(
    postOfficepackageSender
)
There's one slot, the
                      deliver slot, that sends a
                                                  packageSender message
to postOffice.Now,Icanhavethe
                                   moder
                                          deliveramessage:
Io>mailerdeliver
==> Object_0x1005bfda0:
  deliver
                   =method(...)
So, the deliver methodistheobject that sent themes sage. We
                                                                  canalso
getthetarget, like this:
Io>postOfficemessageTarget:=method(calltarget)
==>method(
```

```
Io>postOfficemessageTarget
==> Object_0x1004ce658:
 messageTarget
                  =method(...)
 packageSender
                  =method(...)
```

Simple enough. The target is the post office, as you can see from the slotnames.Gettheoriginalmessagenameandarguments,likethis:

```
Io>postOfficemessageArgs:=method(callmessagearguments)
==>method(
    callmessagearguments
Io>postOfficemessageName:=method(callmessagename)
==>method(
    callmessagename
)
Io>postOfficemessageArgs("one",2,:three)
==>list("one",2,:three)
Io>postOfficemessageName
==>messageName
```

So, Iohasanumber of methods available to let you do message reflection. The next question is, "When does locompute a message?"

Mostlanguagespassargumentsasvaluesonstacks. For example, Java computes each value of a parameter first and then places those valuesonthestack. Iodoesn't. It passes the message itself and the context. Then, there ceivers evaluate themes sage. Yucanactually implement control structures with messages. Recall the Io if. The form is if(booleanExpression,trueBlock,fdeBod) .Let's say you wanted to implementan unless. Here'showyou'ddoit:

```
Download io/unless.io
unless:=method(
    (callsenderdoMessage(callmessageargAt(0)))ifFalse(
    callsenderdoMessage(callmessageargAt(1)))ifTrue(
    callsenderdoMessage(callmessageargAt(2)))
)
unless(1==2,write(
                      "Oneisnottwo\n"
                                        ),write( "oneistwo\n" ))
```

This little example is beautiful, so read it carefully. Think of doMessage assomewhatlikeRuby's ed butatalowerlevel. Where Ruby's ed evaluates a string a scode, doMessage executes an arbitrary message. Io is interpreting the message parameters but delaying binding and execution. In a typical object-oriented language, the interpreteror compilerwouldcomputeallthearguments, including both codeblocks, and place their return values on the stack. In Io, that's not what happensatall.

Saytheobject weeky sendsthemessage princessButtercupunless(trueLove, ("Itistate"println),("Itistrue"println)) .Theresultisthisflow:

- 1. The object westey sendsthepreviousmessage.
- 2. Iotakestheinterpretedmessageandthecontext(thecallsender, target, and message) and putsit on the stack.
- 3. Now, princessButtercup evaluates the message. There is no unless slot, so I owalk suptheprototype chain until it finds
- 4. Io begins executing the unless message. First, Io executes call sender doMessage(call message argAt(0)) . That code simplifies to . If you've ever seen the movie The Princess Bride. west-ytrueLove hasaslotcalled trueLove, and the value is vouknowthat weeky true.
- 5. The message is not false, so we'll execute the third code block, which simplifies to web/("Itistrue" println).
- Water taking advantage of the fact that Io does not execute the argumentstocomputeareturnvaluetoimplementthe unless controlstructure. That concept is extremely powerful. So far, you've seen one side of reflection: behavior with message reflection. The other side of the equationisstate.We lookatstatewithanobject's slots.

Reflection

Iogivesyouasimplesetofmethodstounderstandwhat's going on in the slots. Here are a few of the minaction. This code creates a couple ofobjectsandthenworksitswayupthe prototype chainwithamethod called ancestors:

```
Download io/animals.io
Objectancestors:=method(
        prototype:=selfproto
        if(prototype!=Object,
        writeln("Slotsof" ,prototypetype,
                                             "\n----")
        prototypeslotNamesforeach(slotName,writeln(slotName))
        writeln
        prototypeancestors))
Animal:=Objectclone
Animalspeak:=method(
            "ambiquousanimalnoise"
                                     println)
```

```
Duck:=Animalclone
Duckspeak:=method(
            "quack" println)
Duckwalk:=method(
            "waddle" println)
disco:=Duckclone
discoancestors
```

The code is not too complicated. First, we create an Animal prototype and use that to create a Duck instance, with a speak method. disco's prototypeis Duck. The ancestors method prints the slots of an object's prototypeandthencalls ancestors on the prototype. Keepinmind that anobject can have more than one prototype, but we don't handle this case. To savepaper, we halt the recursion before printing all of the slots inthe Object prototype.Runitwith ioanimals.io:

Here'stheoutput:

```
SlotsofDuck
speak
wa1k
type
SlotsofAnimal
speak
type
```

Nosurprises there. Every object has a prototype, and those prototypes areobjectsthathaveslots.InIo,dealingwithreflectionhastwoparts. Inthepostofficeexample, you sawmess agereflection. Object reflection means dealing with objects and the slots on those objects. No classes areinvolved, anywhere.

WhatWeLearnedinDay2

Ifyou'restillfollowing,day2shouldhavebeenabreakthroughdayof shouldknowenough Iotodobasic tasks with a little supportfromthedocumentation. Yuknowhowtomakedecisions, define methods, used at a structures, and use the basic control structures. In these exercises, we'll put I othrough its paces. Get thoroughly familiar will really want to have the basics down when we move into problems that stretch Io into the metaprogramming and concurrencyspaces.

Day2Self-Study

Do:

- $1.\,A Fibonacci sequence starts with two\,1s. Each subsequent num$ beristhe sum of the two numbers that came before: 1, 1, 2, 3, 5,8,13,21,andsoon.WeaprogramtofindthenthFibonacci number. fib(1) is 1, and fib(4) is 3. As abonus, solve the problem withrecursionandwithloops.
- 2. Howwouldyouchange / toreturn0ifthedenominatoriszero?
- 3. Weaprogramtoaddupallofthenumbersinatwo-dimensional array.
- 4. Addaslotcalled myAecope toalistthatcomputestheaverageof allthenumbersinalist. What happens if there are no numbers inalist?(Bonus:RaiseanIoexceptionifanyiteminthelistisnot anumber.)
- 5. Weaprototypeforatwo-dimensionallist. The dim(x,y) method should allocate a list of y lists that are x elements long. set(x,y)vole) shouldsetavalue,and get(x,y) shouldreturnthatvalue.
- 6. Bonus: We a transpose method so that $(new_matrixget(y,x)) ==$ maxget(x,y) ontheoriginallist.
- 7. Wethematrixtoafile, and read a matrix from a file.
- 8. Weaprogramthat gives you tentries to guessar and omnumber from 1-100. If you would like, give a hint of "hotter" or "colder" afterthefirstguess.

Day3:TheRooeandOtherStrangePlaces 3.4

MyfirstfewdayswithIowerefrustrating,butafteracoupleofweeks, $I found my self giggling like a school girl at the unexpected places the {\tt the continuous} and {\tt the continuou$ languagewouldtakeme.It'slikeFerrisshowinguponthenews,atthe ballpark,intheparade—everywhereyoudon'texpecthim.Intheend,I gotoutofloexactlywhatIwanted,whichwasalanguagethatchanged thewayIthink.

Domain-SpecificLanguages

Just about everyone who is deeply involved with Io appreciates the power that Io gives you in the area of DSLs. Jeremy Tregunna, one of the core committers for Io, told me about an implementation of a

subsetofCinIothattookaround40linesofcode!Sincethatexample isjustalittletoodeepforustoconsider,here'sanotheroneofJeremy's gems. This one implements an API that provides an interesting syntax forphonenumbers.

Sayyouwanttorepresentphonenumbersinthisform:

```
{
    "BobSmith": "5195551212",
    "MaryWalsh": "4162223434"
}
```

There are many approaches to the problem of managing such a list. Two that come to mind are parsing the list or interpreting it. Parsing itmeans that you would write a program to recognize the various elementsofthesyntax, and then you could place the code in a structure thatIounderstands. That's another problem for another day. It would bemuchmorefuntointerpretthatcodeasanIohash.Todothis,you willhavetoalterIo.Whenyou'redone,Iowillacceptthislistasavalid syntaxforbuildinghashes!

Here's how Jeremy attacked the problem, with an assist from Chris Kappler, who brought this example up to the current version of lo:

```
Download io/phonebook.io
OperatorTableaddAssignOperator( ":", "atPutNumber")
curlyBrackets:=method(
 r:=Mapclone
  callmessageargumentsforeach(arg,
       rdoMessage(arg)
       )
 r
MapatPutNumber:=method(
 selfatPut(
                                                  "\"")removeSuffix(" \""),
       callevalArgAt(0)asMutableremovePrefix(
       callevalArgAt(1))
s:=Filewith(
               "phonebook.txt")openForReadingcontents
phoneNumbers:=doString(s)
phoneNumberskeys
                    println
phoneNumbersvaluesprintln
```

That code is slightly more complex than anything you've seen so far, butyouknowthebasicbuildingblocks.Let'sdeconstructit:

```
OperatorTableaddAssignOperator( ":", "atPutNumber")
```

Thefirstlineaddsanoperatortolo's assignment operator table. Whenever: isencountered, Iowill parsethatas atPutNumber, understanding thatthefirstargumentisaname(andthusastring), and the second is avalue.So, key:volte willbeparsedas atPutNumber("key", vol.e) .Moving on:

```
curlyBrackets:=method(
  r:=Mapclone
 callmessageargumentsforeach(arg,
       rdoMessage(arg)
 r
)
```

The parser calls the curly Brackets method whenever it encounters curly brackets({}). Whithis method, we create an empty map. Then, we execute call message arguments feed ag r doMessage(arg)) for each argument. That's a seriously denseline of code! Let's take it apart.

Fromlefttoright, wetake the callmessage, which is the part of the code betweenthecurlybrackets. Then, weiterate through each of the phone numbersinthelistwith **t**Forth .Foreachphonenameandphonenumber, we execute rdoMessage (arg). For example, the first phone number willexecuteas r"BobSmith": "5195551212" .Since:isinouroperatortable as atPutNumber,we'llexecute ratPutNumber("BobSmith","5195551212") .That bringsustothefollowing:

```
MapatPutNumber:=method(
  selfatPut(
      callevalArgAt(0)asMutableremovePrefix(
                                                "\"")removeSuffix(" \""),
      callevalArgAt(1))
)
```

Remember, ley:volte willparseas atPutNumber("key", value) .Inourcase, thekeyisalreadyastring, sowestriptheleading and trailing quotes. You can see that at Put Number simply calls at Put on the target range, whichis self, stripping the quotes off the first argument. Since messages areimmutable, to stripthequotes, we have to translate themes sage to amutablevalueforittowork.

Yucanusethecodelikethis:

```
"phonebook.txt")openForReadingcontents
s:=Filewith(
phoneNumbers:=doString(s)
phoneNumberskeys
                    println
phoneNumbersvaluesprintln
```

Understandinglo'ssyntaxistrivial. Yujusthavetoknowwhat's going oninthelibraries.Inthiscase, you see a fewne wlibraries. The doString messageevaluatesourphonebookascode, File isaprototypeforworkingwithfiles, with specifiesafilenameandreturnsafileobject, openFor-Reading opensthatfileandreturnsthefileobject, and contents returns the contents of that file. Taken to gether, this code will read the phone bookandevaluateitascode.

Then, the braces define a map. Each line in the map "string1": "string2" doesa mpatPut("string1", "string2") , andwe'releftwithahashofphone numbers. So, in Io, since you can redefine anything from operators to the symbols that make up the language, you can build DSLs to your heart'scontent.

So, nowyou can be gin to see how you would change Io's syntax. How would you go about dynamically changing the language's behavior? That's the topic of the next section.

lo'smethod_missing

Let's review the flow of control. The behavior for what happens in a given message is all baked into Object. When you send an object a message, it will do the following:

- 1. Compute the arguments, inside out. These are just messages.
- 2. Getthename, target, and sender of themes sage.
- 3. Try toreadtheslotwiththenameofthemessageonthetarget.
- 4. If the slotexists, return the data or invoke the method in side.
- 5. If the slot doesn't exist, forward themes sage to the prototype.

These are the basic mechanics of inheritance within Io. You normally wouldn'tmesswiththem.

Butyoucan. Yucanuse the fawad messageinthesamewaythatyou would use Ruby's method_missing, but the stakes are a little higher. Io doesn'thaveclasses, sochanging alsochangesthewayyouget fawad anyofthebasicbehaviorsfrom object. It's a bit like juggling hatchets onthehighwire. It's a cooltrickify ou can get away with it, so let's get started!

XML is a pretty way to structure data with an ugly syntax. You may wanttobuildsomethingthatletsyourepresentXMLdataasIocode.

Forexample, you might want to express this:

```
<body>
>
Thisisasimpleparagraph.
</body>
likethis:
body(
    p("Thisisasimpleparagraph.")
)
```

Let'scallthenewlanguageLispML.We goingtouseIo'sforwardlike amissingmethod. Here's the code:

```
Download io/builder.io
Builder:=Objectclone
Builderforward:=method(
  writeln("<",callmessagename,</pre>
                                     ">")
  callmessageargumentsforeach(
        arg,
        content:=selfdoMessage(arg);
                             "Sequence",writeln(content)))
        if(contenttype==
  writeln("</",callmessagename,</pre>
                                      ">"))
Builder ul(
        li("Io"),
        1i("Lua"),
        li("JavaScript"))
```

Let's carveit up. The Builder prototype is the workhorse. It overrides to pick up any arbitrary method. First, it prints an open tag. fawad Next, we use a little message reflection. If the message is a string, Io will recognize it as a sequence, and Builder prints the string without quotes. Finally, Builder prints a closing tag.

Theoutputisexactlywhatyou'dexpect:

```
<u1>
<1i>>
Ιo
<1i>>
Lua
<1i>>
JavaScript
</u1>
```

Ihavetosay, I'mnotsurewhether Lisp MListhat muchofanim provementovertraditional XML, but the example is instructive. Yore just completely changed the way in heritance works in one of lo's prototypes.AnyinstanceofBuilderwillhavethesamebehavior.Doingthis,youcan createanewlanguagewithIo'ssyntaxbutnoneofthesamebehaviors by defining your own Object and basing all of your prototypes on that newobject. Yucanevenoverride Object tocloneyournewobject.

Concurrency

Io has outstanding concurrency libraries. The main components are coroutines.actors.andfutures.

Coroutines

The foundation for concurrency is the coroutine. A coroutine provides awaytovoluntarilysuspendandresumeexecutionofaprocess. Think of a coroutine as a function with multiple entry and exit points. Each yieldwillvoluntarilysuspendtheprocessandtransfertoanotherprocess. Yucanfireamessageasynchronously by using a message. The former returns a future (more later), and the second returns nil andstartsthemessageinitsownthread. For example, considerthisprogram:

```
Download io/coroutine.io
vizzini:=Objectclone
vizzinitalk:=method(
            "Fezzik, arethererocksahead?"
                                               println
             "Nomorerhymesnow, Imeanit."
                                               println
             yield)
fezzik:=Objectclone
fezzikrhyme:=method(
                         vield
            "Ifthereare, we'llallbedead."
                                                 println
             "Anybodywantapeanut?"
                                       println)
vizzini@@talk;fezzik@@rhyme
CoroutinecurrentCoroutinepause
```

fak and vizzini are independent instances of Object with coroutines. We fireasynchronous talk and rhyme methods. These run concurrently, voluntarily yielding control to the other at specified intervals with the

yield message. The last pause waits until all as yncmessages complete and the nexits. Coroutines are great for solutions requiring cooperative multitasking. Whthis example, two processes that need to coordinate caneasilydoso,toreadpoetry,forexample:

```
batate$iocode/io/coroutine.io
Fezzik, arethererocksahead?
Ifthereare, we'llallbedead.
Nomorerhymesnow, Imeanit.
Anybodywantapeanut?
Scheduler:nothinglefttoresumesoweareexiting
```

JavaandC-basedlanguagesuseaconcurrencyphilosophycalled preemptivemultitasking. Whenyou combine this concurrency strategy with objects that have changeable state, you windup with programs that are hardtopredictandnearlyimpossibletodebugwiththecurrenttesting strategies that most teams use. Coroutines are different. Who coroutines, applications can voluntarily give up controlatre as on abletimes. Adistributed client could relinquish control when waiting for the server. Water processes could pause after processing queue items.

Coroutines are the basic building blocks for higher levels of abstractions like actors. Think of actors as universal concurrent primitives that can send messages, process messages, and create other actors. The messages an actor receives are concurrent. In Io, an actor places an incoming message on a queue and processes the contents of the queuewithcoroutines.

Next, we'll look into actors. You won't believe how easy they are to code.

Actors

Actors have a huge theoretical advantage over threads. An actor changes its own state and accesses other actors only through closely controlled queues. Threads can change each other's state without restriction.Threadsaresubjecttoaconcurrencyproblemcalled raceconditions, where two threads access resources at the same time, leading tounpredictableresults.

Here's the beauty of lo. Sending an asynchronous message to any object makesitanactor. Endofstory. Let'stakeasimple example. First, we'll createtwoobjectscalled ξαber and slower:

```
Io>slower:=Objectclone
==> Object_0x1004ebb18:
Io>faster:=Objectclone
==> Object_0x100340b10:
```



Now,we'lladdamethodcalled start toeach:

```
Io>slowerstart:=method(wait(2);writeln("slowly"))
==>method(
   wait(2);writeln("slowly")
Io>fasterstart:=method(wait(1);writeln("quickly"))
==>method(
   wait(1);writeln("quickly")
)
```

We cancall both methods sequentially on one line of code with simple messages, like this:

```
Io>slowerstart; fasterstart
slowly
quickly
==>nil
```

They start in order, because the first message must finish before the second can begin. But we can easily make each object run in its own threadbyprecedingeachmessagewith @@.whichwillreturnimmediatelyandreturnnil:

```
Io>slower@@start;faster@@start;wait(3)
quickly
slowly
```

We add an extra wait to the end so that all threads finish before the program terminates, but that's a great result. We arerunningintwo threads. We made both of these objects actors, justbysendinganasynchronousmessagetothem!

Futures

Iwillfinishuptheconcurrencydiscussionwiththeconceptoffutures. Afutureisaresultobjectthatisimmediatelyreturnedfromanasynchronousmessagecall. Since themessage may take a while toprocess, thefuturebecomestheresultoncetheresultisavailable. If you ask for the value of a future before the result is available, the process blocksuntil the value is available. Say we have a method that takes a long timetoexecute:

```
futureResult:=URLwith("http://google.com/")@fetch
```

Icanexecutethemethodanddosomethingelseimmediatelyuntilthe resultisavailable:

```
writeln("Dosomethingimmediatelywhilefetchgoesoninbackground...")
//...
```

Then.Icanusethefuturevalue:

```
writeln("Thiswillblockuntiltheresultisavailable.")
//thislinewillexecuteimmediately
writeln("fetched",futureResultsize,"bytes")
//thiswillblockuntilthecomputationiscomplete
//andIoprintsthevalue
==>1955
```

The futureResult codefragmentwillreturnafutureobject, immediately. In Io, a future is not a proxy implementation! The future will block untiltheresultobjectisavailable. The valueisa Future objectuntilthe result arrives, and then all instances of the value point to the result object. The console is going to print the string value of the last statementreturned.

Futures in Io also provide automatic deadlock detection. It's a nice touch, and they are easy to understand and use.

Now that you've had a flavor of Io's concurrency, you have a sound foundationforevaluatingthelanguage.Let'swrapupday3soyoucan putwhatyouknowintopractice.

WhatWeLearnedinDay3

Inthissection, you learned to do something nontrivial in Io. First, we bent the rules of syntax and built a new hash syntax with braces. We added an operator to the operator table and wired that into operations on a hash table. Next, we built an XML generator that used method_missing toprintXMLelements.

Next, wew rotes ome code that used coroutine stoman age concurrency. The coroutines differed from concurrency in languages like Ruby, C, and Java because threads could only change their own state, leading toamorepredictableandunderstandableconcurrencymodelandless ofaneedforblockingstatesthatbecomebottlenecks.

We sent some as ynchronous messages that made our prototype sactors.We didn'thavetodoanythingbeyondchangingthesyntaxofourmessages. Finally, we looked briefly at futures and how they worked in Io.

Day3Self-Study

Do:

 EnhancetheXMLprogramtoaddspacestoshowtheindentation structure.

- Createalistsyntaxthatusesbrackets.
- EnhancetheXMLprogramtohandleattributes: if the first argumentisamap (use the curly brackets syntax), add attributes to theXMLprogram.Forexample:

book({"author":"Tate"}...) wouldprint <bookauthor="Tate">:

3.5 WrappingUplo

Ioisanexcellentlanguageforlearninghowtouseprototype-basedlanguages. Like Lisp, the syntaxis stunningly simple, but the semantics of the language lend plenty of power. The prototype languages encapsulatedataandbehaviorlikeobject-orientedprogramminglanguages. Inheritanceissimpler. There are no classes or modules in Io. One object inheritsbehaviordirectlyfromitsprototype.

Strengths

Prototypelanguagesaregenerallyquitemalleable. Yucanchangeany slotonanyobject. Iotakes this flexibility to the max, allowing you to quicklycreatethesyntaxyouwant.LikeRuby,someofthetrade-offs that make Io so dynamic tend to cap the performance, at least in a singlethread. The strong, modern concurrency libraries often make Ioa goodlanguageforparallelprocessing.Let'slookatwhereIoisexcelling today.

Footprint

Io's footprintiss mall. Most of the production I oapplications are embeddedsystems. This application makes sense, since the language is small, powerful, and quite flexible. The virtual machine is easy to port to differentoperatingenvironments.

Simplicity

Io'ssyntaxisremarkablycompact. Yucanlearn Ioveryquickly. Once youunderstandthecoresyntax, everythingelseislearningthelibrary structures. I found that I could work my way into metaprogramming quite quickly, within my first month of using the language. In Ruby, getting to the same point took a little longer. In Java, it took many monthstogettothepointwhereIcouldmakeanysenseofmetaprogrammingatall.

Flexibility

Io'sducktypingandfreedomallowyoutochangeanyslotinanyobject atanytime. This free-wheeling nature means you can change the basic rules of the language to suit your application. It's quite easy to add proxies at any place through changing the forward slot. You canalso overridekeylanguageconstructsbychangingtheirslotsdirectly. Yu canevencreateyourownsyntaxquickly.

Concurrency

UnlikeJavaandRuby, the concurrency constructs are up-to-date and fresh. Actors, futures, and coroutines make it much easier to build multithreaded applications that are easier to test and have better performance. Ioalsogives considerable thought to mutable data and how to avoid it. Having these features baked into the core libraries made it easy to learn a robust concurrency model. Later, in other languages, wewillbuildontheseconcepts. Yilseeactorsin Scala, Erlang, and Haskell.

Wedkresses

There's much to like about Io and some suboptimal aspects as well. Freedomandflexibilitycomeataprice. Also, since I ohas the smallest communityofanyofthelanguagesinthisbook,it'sariskierchoicefor someprojects.Let'stakealookattheproblemsassociatedwithIo.

Syntax

Iohasverylittlesyntaxsugar.Simplesyntaxisadouble-edgedsword. Ononeside, the clean syntax makes Io, the language, easy to understand.Butthere's acost. Simple syntax of tenmakes it hard to communicate difficult concepts concisely. Said another way, you may find it easytounderstandhowagivenprogramusestheIolanguageand,at the same time, have a difficult time understanding what your program isdoing.

For a point of contrast, consider Ruby. At first, you may find the Ruby code array[-1] bafflingbecauseyoudon'tunderstandthesyntacticsugar: -1 is shorthand for the last element in the array. You would also need tolearnthat [] isamethodtogetthevalueataspecifiedindexofan array. Once you understood those concepts, you'd be able to process morecodeataglance. WhIo, the trade-offisthe opposite. You don't havetolearnverymuchtogetstarted, butyoudohavetoworkalittle

harder to absorb concepts that might be otherwise communicated withsugar.

Thebalanceofsyntactic sugarisa difficult one. Add too much sugar, andit's too difficult to learn a language and remember how to use it. Add too little, and you need to spend more time to express code and potentially more energy to debugit. In the end, syntax is a matter of preference. Matzprefersplentyofsugar. Stevedoesn't.

Community

Right now, the Io community is very small. You cannot always find libraries in Iolike you can withother languages. It's also harder to findprogrammers. These issues are somewhat mitigated by having a good Cinterface(whichtalkstoavarietyoflanguages)andasyntaxthatis so easy to remember. Good Java Script programmers could pick up Io quickly.Buthavingasmallercommunityisadefiniteweaknessandis themainthingthatholdspowerful, newlanguagesback. Either Iowill getakillerapplicationthatdrivesacceptanceoritwillremainaniche player.

Referrence

Discussing performance in a vacuum of other issues such as concurrencyandapplicationdesignisnotusuallywise, but I should point out thatIohasanumberoffeaturesthatwillslowraw, single-threaded executionspeed. This problem is somewhat mitigated by Io's concurrency constructs, but you should still keep this limit at ion in mind.

FinalThoughts

In general, I liked learning Io. The simple syntax and small footprint intrigued me. I also think that, like Lisp, Io has a strong overriding philosophyofsimplicityandflexibility. Bystaying with this philosophy uniformly in the language creation, Steve Dekorte has created somethingliketheLispoftheprototypelanguages.Ithinkthelanguagehas ashottogrow.LikeFerrisBueller,ithasabright,butperilous,future.

Chapter 4 Prolog

Ah, Prolog. Sometimes spectacularly smart, other times just as frustrating. Yilgetastoundinganswersonlyifyouknowhowtoaskthe question. Think RainMan. 1 Iremember watching Raymond, the lead character, rattle off Sally Dibbs' phone number after reading a phone bookthenightbefore, without thinking about whether he should. Wh both Raymond and Prolog, I often find myself asking, in equal parts, "Howdidheknowthat?" and "Howdidn'theknowthat?" He's a fountain of knowledge, if you can only frame your questions in the right way.

Prologrepresents a serious departure from the other languages we've encounteredsofar.BothIoandRubyarecalled imperativelanguages. Imperativelanguagesarerecipes. Yutell the computer exactly how to doajob. Higher-levelimperativelanguages might give you alittle more leverage, combining many longer steps into one, but you're basically putting together a shopping list of ingredients and describing a stepby-stepprocessforbakingacake.

IttookmeacoupleofweeksofplayingwithPrologbeforeIcouldmake an attempt at this chapter. I used several tutorials as I ramped up, ² forsomeexamplestowadethrough includingatutorialbyJ.R.Fisher andanotherprimerby A. Aaby ³ tohelpthestructureandterminology gelforme, and lots of experimentation.

Prologisadeclarativelanguage. What throwsome facts and inferences at Prologand let it do the reasoning for you. It's more like going to a

^{1.} RainMan .DVD.DirectedbyBarryLevinson.1988;LosAngeles,CA:MGM,2000.

^{2.} http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html

^{3.} http://www.lix.polytechnique.fr/~liberti/public/computing/prog/prolog/prolog-tutorial.html

goodbaker. Yudescribethecharacteristicsofcakes that youlike and letthebakerpicktheingredientsandbakethecakeforyou,basedon therulesyouprovided. Wh Prolog, you don't have to know how.The computerdoesthereasoningforyou.

WhacasualflipthroughtheInternet, you can find example sto solve a Sudoku with fewer than twenty lines of code, crack Rubik's Cube, and solve famous puzzles such as the Tower of Hanoi (around a dozen linesofcode). Prologwasone of the first successful logic programming languages. Yumakeassertions with purelogic, and Prolog determines whethertheyaretrue. Yucanleavegapsinyourassertions, and Prolog willtrytofillintheholesthatwouldmakeyourincompletefactstrue.

AboutProlog 4.1

Developed in 1972 by Alain Colmerauer and Phillipe Roussel, Prolog is a logic programming language that gained popularity in naturallanguage processing. Now, the venerable language provides the programming foundation for a wide variety of problems, from scheduling toexpertsystems. Yucanusethisrules-basedlanguage for expressing logicandaskingquestions.LikeSQL,Prologworksondatabases,but thedatawillconsistoflogicalrulesandrelationships.LikeSQL,Prolog has two parts: one to express the data and one to guery the data. In Prolog, the data is in the form of logical rules. These are the building blocks:

- Facts. Afactisabasicassertionaboutsomeworld. (Babeisapig; pigslikemud.)
- Rules. Arule is an inference about the facts in that world. (An animallikesmudifitisapig.)
- Query. Aquery is a question about that world. (Does Babe like mud?)

Factsandruleswillgointoa knowledgebase .APrologcompilercompilestheknowledgebaseintoaformthat's efficient for queries. As we walkthroughtheseexamples, you'lluse Prolog to express your knowledgebase. Then, you'll do direct retrieval of data and also use Prologto link rule stogether totelly ou something you might not have known.

Enoughbackground.Let'sgetstarted.

4.2 Day 1: An Excellent Driver

In Rain Man, Raymond told his brother he was an excellent driver, meaning he could do a fine job of handling the car at five miles per hourin parking lots. He was using all the main elements—the steeringwheel, the brakes, the accelerator—he just used the minalimited context. That 'syourgoal today. We goingtousePrologtostatesome facts, make some rules, and do some basic queries. Like Io, Prologis an extremely simple language syntactically. You canlearnthesyntax rules quickly. The real fun begins when you layer concepts in interestingways. If this is your first exposure, I guarantee either you will change the way you think or you'll fail. We save the in-depth constructionforalaterday.

First things first. Get a working installation. I'm using GNU Prolog, version 1.3.1, for this book. Be careful. Dialects can vary. I'll do my best to stay on common ground, but if you choose a different version of Prolog, you'll need to do a little home work to understand where yourdialect is different. Regardless of the version you choose, here's how you'lluseit.

Basic Foots

In some languages, capitalization is entirely at the programmer's discretion, butin Prolog, the case of the first letter is significant. I faword begins with a lowercase character, it's an atom—a fixed value like a Ruby symbol. If it begins with an uppercase letter or an underscore, it's a variable. Vaite values can change; atoms can't. Let's build a simpleknowledgebasewithafewfacts. Keythefollowingintoaneditor:

```
Download prolog/friends.pl
likes(wallace, cheese).
likes(grommit, cheese).
likes(wendolene, sheep).
friend(X,Y):-\+(X=Y),likes(X,Z),likes(Y,Z).
```

The previous file is a knowledge base with facts and rules. The first threestatements are facts, and the last statement is a rule. Facts are directobservationsofourworld.Rulesarelogicalinferencesaboutour world. For now, pay attention to the first three lines. These lines are each facts. worker , grommit, and wendere are atoms. You can read

likes cheese, grommit likes cheese, and werdere likes themas worker sheep.Let'sputthefactsintoaction.

StartyourProloginterpreter.Ifyou'reusingGNUProlog,typethecommand gprolog. Then, to loady our file, enter the following:

```
|?-['friends.pl'].
compiling/ Users/batate/prag/Book/code/prolog/friends.plforbytecode...
/Users/batate/prag/Book/code/prolog/friends.plcompiled,4linesread-
997byteswritten,11ms
yes
1?-
```

UnlessPrologiswaitingonanintermediateresult,itwillrespondwith yes or no. In this case, the file loaded successfully, so it returned yes . We canstart to ask some questions. The most basic questions are yes and no questionsaboutfacts. Askafew:

```
|?-likes(wallace, sheep).
|?-likes(grommit,cheese).
yes
```

Thesequestions are pretty intuitive. Does worke likesheep?(No.)Does grommit like cheese? (Yes.) These are not too interesting: Prologisjust parrotingyourfactsbacktoyou. It starts to get a little more exciting whenyoustarttobuildinsomelogic.Let'stakealookatinferences.

BasicInferencesandV ariables

Let'strythe friend rule:

```
|?-friend(wallace,wallace).
no
```

So, Prologis working through the rules we gave it and answering yes or no questions. There's more herethan meets the eye. Check out the friend ruleagain:

InEnglish, for X to be a friend of Y, X cannotbethesameas Y.Lookat. thefirstparttotherightof :-,calleda subgoal. \+ doeslogicalnegation, so $\+(X=Y)$ means X isnotegualto Y.

Try somemorequeries:

```
|?-friend(grommit,wallace).
yes
|?-friend(wallace,grommit).
yes
```

InEnglish, X isafriendof Y ifwecanprovethat X likessome Z and Y likesthatsame Z.Both wordce and grommit like cheese, so the sequeries succeed.

Let's diveint othe code. In the sequeries, X isnotequalto Y,provingthe first subgoal. The query will use the second and third subgoals, likes(X, Z) and likes(Y,Z). grommit and water like cheese, soweprove the second andthirdsubgoals. Try anotherquery:

```
|?-friend(wendolene,grommit).
```

no

Inthiscase, Prologhadtotry several possible values for X, Y, and Z:

- wendere , grommit, and cheese
- , grommit, and sheep wendtene

Neither combination satisfied both goals, that likes Z and wendtene grommit likes Z. Noneexisted, so the logic engine reported no, they are notfriends.

Let's formalize the terminology. This...

```
friend(X,Y):-+(X=Y), likes(X,Z), likes(Y,Z).
```

...is a Prolog rule with three variables, X, Y, and Z. We call the rule friend/2, shorthandfor friend with two parameters. This rule has three subgoals, separated by commas. All must be true for the rule to be true. So.ourrulemeans X isafriendof Y if X and Y arenotthesameand Χ and Y likethesame Z.

FillingintheBlanks

Wore usedPrologtoanswersome yes or no questions, but we can do more than that. In this section, we'll use the logic engine to find all possiblematchesforaquery. Todothis, you will specify a variable in yourquery.

```
Download prolog/food.pl
food_type(velveeta,cheese).
food_type(ritz,cracker).
food_type(spam,meat).
food_type(sausage,meat).
food_type(jolt,soda).
food_type(twinkie,dessert).
flavor(sweet,dessert).
flavor(savory, meat).
flavor(savory, cheese).
flavor(sweet, soda).
food_flavor(X,Y):-food_type(X,Z),flavor(Y,Z).
We have a few facts. Some, such as
                                       foodtookdeedacheese)
                                                                   .mean
a tool has a certain type. Others, such as
                                             flavor(sweet, dessert), mean
a food type has a characteristic flovor. Finally, we have a rule called
           that infers the flavor of food. A food
                                                                         if
foodfbor
                                                  X has a foodfoloor Y
thefoodisofa
                foodtoeZ
                           andthat 7 alsohasthatcharacteristic
Compileit:
|?-['code/prolog/food.pl'].
compiling/ Users/batate/prag/Book/code/prolog/food.plforbytecode...
/Users/batate/prag/Book/code/prolog/food.plcompiled,
12linesread-1557byteswritten,15ms
(1ms)yes
andasksomequestions:
|?-food_type(What, meat).
What=spam?;
What=sausage?;
no
Now, that's interesting. We
                               askingProlog, "Findsomevaluefor
                                                                     What
thatsatisfiesthequeryfood_type(What,meat)."Prologfoundone,
                                                                    spam.
When we typed the ;, we were asking Prolog to find another, and it
returned sausage. They were easy values to find since the queries
depended on basic facts. Then, we asked for another, and Prologre-
```

spondedwith no. This behavior can be slightly inconsistent. As a conve-

thiscopyis (P1.0printing,October2010)

nience, if Prolog can detect that there are no more alternatives remaining,you'llseea ys .IfPrologcan'timmediatelydeterminewhetherthere are more alternatives without doing more computation, it will prompt youforthenextandreturn no. The feature is really a convenience. If Prologcangiveyouinformationsooner, it will. Try a few more:

```
|?-food_flavor(sausage,sweet).
    no
|?-flavor(sweet,What).
What=dessert?;
What=soda
ves
No, sausage is not sweet. What food types are sweet?
                                                            dessert and soda.
These are all facts. But you can let Prolog connect the dots for you, too: \\
|?-food_flavor(What, savory).
What=velveeta?;
What=spam?;
What=sausage?;
no
```

Remember, foodfoo(XY) isarule, notafact. Wee askingPrologto findallpossiblevaluesthatsatisfythequery, "Whatfoodshaveasavory flavor?"Prologmusttietogetherprimitivefactsaboutfood,types,and flavorstoreachtheconclusion. The logicengine has towork through possible combinations that could make all the goals true.

MapColoring

Let'susethesameideatodomapcoloring.Foramorespectacularlook atProlog,takethisexample.Wewanttocoloramapofthesoutheastern $cover the states shown in Figure {\color{red}4.1}, on the next$ United States. We page. We do not want two states of the same color to touch.

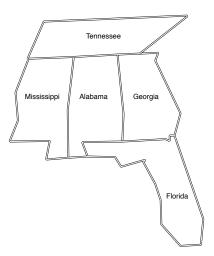


Figure 4.1: Map of some southeastern states

We codeupthesesimplefacts:

```
Download prolog/map.pl
different(red, green).different(red, blue).
different(green, red).different(green, blue).
different(blue, red).different(blue, green).
coloring(Alabama, Mississippi, Georgia, Tennessee, Florida):-
  different(Mississippi, Tennessee),
  different(Mississippi, Alabama),
  different(Alabama, Tennessee),
  different(Alabama, Mississippi),
  different(Alabama, Georgia),
  different(Alabama, Florida),
  different(Georgia, Florida),
  different(Georgia, Tennessee).
```

We have three colors. We tell Prolog the sets of different colors to usein the map coloring. Next, we have a rule. In the coloring rule, wetell Prologwhichstatesneighborothers, andwe'redone. Try it:

```
|?-coloring(Alabama, Mississippi, Georgia, Tennessee, Florida).
Alabama=blue
```

Florida=green Georgia=red Mississippi=red Tennessee=green? Sureenough, there is a way to color these five states with three colors. You can get the other possible combinations too by typing g. Wha dozenlinesofcode, we'redone. The logicis ridiculous lysimple—achild couldfigureitout. Atsome point, you have to askyourself...

Where'stheProgram?

We havenoalgorithm! Try solving this problem in the procedural language of your choice. Is your solution easy to understand? Think through what you'd have to do to solve complex logic problems like this in Rubyor Io. One possible solution would be as follows:

- 1. Collectandorganizeyourlogic.
- 2. Expressyourlogicinaprogram.
- 3. Findallpossible solutions.
- 4. Putthepossible solutions through your program.

Andyouwouldhavetowritethisprogramoverandover.Prologletsyou express the logic in facts and inferences and then lets you ask questions. Yne not responsible for building any step-by-step recipe with this language. Prolog is not about writing algorithms to solve logical problems. Prologisabout describing your world a sit is and presenting logical problems that your computer can try to solve.

Letthecomputerdothework!

Unification, Rt 1

Atthispoint, it's time to backup and provide a little more theory. Let's shine a little more light on unification. Some languages use variable assignment.InJavaorRuby,forexample, x=10 meansassign 10 to the variable x.Unificationacrosstwostructurestriestomakebothstructuresidentical. Consider the following knowledge base:

```
Download prolog/ohmy.pl
cat(lion).
cat(tiger).
dorothy(X,Y,Z):-X=lion,Y=tiger,Z=bear.
twin_cats(X,Y):-cat(X),cat(Y).
```

Inthisexample, = means unify, ormakeboth sides the same. We have twofacts: lions and tigers are cats. We also have two simplerules. In dorothy/3, X, Y, and Z are lion, tiger, and bear, respectively. In twin_cats/2,

X isa cat, and Y isa cat. We can use this knowledge base to shedalittle lightonunification.

First, let's use the first rule. I'll compile and then do a simple query withnoparameters:

```
|?-dorothy(lion, tiger, bear).
yes
```

Remember, unification means "Find the values that make both sides match." On the right side, Prolog binds X, Y, and Z to lion, tiger, and bear. These match the corresponding values on the left side, so unificationissuccessful. Prologreports ys . This case is pretty simple, but wecanspiceitupalittlebit.Unificationcanworkonbothsidesofthe implication. Try thisone:

```
|?-dorothy(One,Two,Three).
One=lion
Three=bear
Two=tiger
yes
```

This example has one more layer of indirection. In the goals, Prolog unifies X, Y, and Z to lion, tiger, and bear. On the left side, Prologunifies X, Y, and Z to One, Two , and Three and then reports the result.

Now, let's shift to the last rule, twin_cats/2.Thisrulesays twin cats(X,Y)istrueifyoucanprovethat X and Y arebothcats. Try it:

```
|?-twin_cats(One,Two).
One=lion
```

Two=lion?

Prologreported the first example. lion and lion are both cats. Let's see howitgotthere:

- 1. We issuedthequery twin_cats(One, Two) . Prologbinds One to X and Two to Y. To solve these, Prolog must start working through the goals.
- 2. Thefirstgoalis cat(X).
- 3. We have two facts that match, cat(lion) and cat(tiger). Prolog tries thefirstfact, binding X to lion, and moves onto the next goal.

- 4. Prolognowbinds Y to cat(Y). Prologcan solve this goal in exactly thesamewayasthefirst, choosing lion.
- 5. Wate satisfiedbothgoals, sotherule is successful. Prologreports thevalues of One and Two that made it successful and reports Ves .

So, we have the first solution that makes the rules true. Sometimes, one solution is enough. Sometimes, you need more than one. We can nowstepthroughsolutionsonebyonebyusing ;,orwecangetallof therestofthesolutionsbypressing a.

```
Two=lion?a
One=lion
Two=tiger
One=tiger
Two=lion
One=tiger
Two=tiger
(1ms)yes
```

Χ NoticethatPrologisworkingthroughthelistofallcombinationsof and Y, given the information available in the goals and corresponding facts. Asyou'll see later, unificational solet syou do some so phisticated matchingbasedonthestructureofyourdata. That's enough for day 1. Water goingtodoalittlemoreheavyliftinginday2.

ProloginPractice

Ithastobealittledisconcertingtoseea "program" presented in this way. In Prolog, there's not often a finely detailed step-by-step recipe, only a description of the cake you'll take out of the pan when you're done. When I was learning Prolog, it helped metremendously to interviewsomeonewhohadusedthelanguageinpractice.ItalkedtoBrian Tarbox who used this logic language to create schedules for working withdolphinsforaresearchproject.

AnInterviewwithBriantax DolphinResearcher

Bruce: CanyoutalkaboutyourexperienceslearningProlog?

Brian: Ilearned Prolog back in the late 1980s when I was in graduates chool at the University of Hawaii at Manoa. I was working at theKewaloBasinMarineMammalLaboratorydoing sachintothecognitive capabilities of bottlenosed dolphins. At the time Inoticed that much

thiscopyis (P1.0printing,October2010)

of the discussion at the lab concerned people's theories about how the dolphins thought. We worked primarily with a dolphin named Akeakamai, or Ake for short. Many debates started with "Well, Ake probably seesthesituationlikethis."

Idecidedthatmymaster's the siswould be to try to create an executable modelthatmatchedourbeliefsaboutAke'sunderstandingoftheworld, or at least the tiny subset of it that we were doing **each** on. If our executable model predicted Ake's actual behavior, we would gain some confidenceinourtheoriesaboutherthinking.

Prologis awonderfullanguage, but until youdrink the Kool-Aid, it can give you some pretty weird rate I redone of my first experiments with Prolog, writing something along the lines of x=x+1. Prolog esponded "no." Languages don't just say "no." They might give the wrong answer or fail to compile, but I had never had a language talk back tome. So, I called Prolog support and said that the language had said "no"when I tried to change the value of a variable. They asked me, "Why would you want to change the value of a variable?" I mean, what kind oflanguagewon'tletyouchangethevalueofavariable?Onceyougrok Prolog, you understand that variables either have particular values or areunbound, butitwas unsettling at the time.

Bruce: HowhaveyouusedProlog?

Brian: Idevelopedtwomainsystems:thedolphinsimulatorandalaboratoryscheduler. The lab would run four experiments a day with each havetounderstandthat **ran**hdolphins arean offourdolphins. You incrediblylimited **Exact** Each dolphin was working on different experiments, and each experiment raised a different set of personnel. Some roles, such as the actual dolphin trainer, could be filled by only a few people. Otherroles such as data accercould be done by several peoplebutstill raining. Most experiments rain last affofsix to adozenpeople. Wehadgraduates tudents, under graduates, and Earthwatchvolunteers. Every person had their own schedule and their own shift set of skills. Finding a schedule that utilized everyone and made surealltasksweredonehadbecomeafull-timejobforoneofthestaff.

IdecidedtotrytobuildaProlog-basedschedulebuilder.Itturnedoutto beaproblemtailor-madeforthelanguage. Ibuiltaset of facts describing eachperson's skillset, eachperson's schedule, and each experiment's rainers I could then basically tell Prolog "make itso." For each task listed in an experiment, the language would find an available person

withthatskillandbindthemtothetask.Itwouldcontinueuntiliteither satisfied the needs of the experiment or was unable to. If it could not findavalidbinding,itwouldstartundoingpreviousbindingsandtrying again with another combination. In the end, it would either find a valid scheduleordeclarethattheexperimentwasover-constrained.

Bruce: Are there some interesting examples of facts, rules, or assertions to telephins that would make sense to our to the

Brian: Therewasoneparticular situation I member where the simulated dolphin helped us understand Ake's actual behavior. Ake æ sponded to a gestural sign language containing "sentences" such as "hoopthrough" or "rightballtail-touch." We would give her instructions, andshewould**roo**d

Partofmyrachwastotrytoteachnewwordssuchas"not."Inthis context, "touchnotball" meant touch anything but the ball. This was a $hard problem for Aketosolve, but the {\it part} hwas proceeding well for$ awhile. At one point, however, she started simply sinking underwater wheneverwegavehertheinstruction. We didn'tunderstanditall.This canbeaveryfrustratingsituationbecauseyoucan'taskadolphinwhy itdidsomething. So, we presented the training task to the simulated dolphinandgotaninteresting stAlthoughdolphinsareverysmart, they willgenerallytrytofindthesimplestanswertoaproblem. Wehadgiven the simulated dolphin the same heuristic. It turns out that Ake's gesturallanguageincludeda "word" foroneofthewindowsinthetank. Most trainershadforgottenaboutthiswordbecauseitwasrarelyused. The simulated dolphin discovered the rule that "window" was a successful reporte to "not ball." It was also a successful reporte to "not hoop," "notpipe," and "notfrisbee." We hadguardedagainstthispatternwith theotherobjectsbychangingthesetofobjectsinthetankforanygiven trial, but obviously we could not **row** the window. It turns out that when Akewas sinking to the bottom of the tankshewas positioned next tothewindow,thoughIcouldnotseethewindow!

Bruce: WhatdoyoulikeaboutPrologthemost?

Brian: The declarative programming model is very appealing. In general, if you can describe the problem, you have solved the problem. In most languages I've found myself arguing with the computer at some point saying, "You know what I mean; just do it!" Cand C++ compiler errorssuchas "semicolonexpected" are symbolic of this. If you expected a semicolon, how about inserting one and seeing whether that fixes it?

InProlog, allIhadtodointheschedulingproblemwasbasicallysay, "I wantadaythatlookslikethis,sogomakemeone"anditwoulddoit.

Bruce: Whatgaveyouthemosttrouble?

Brian: Prologseemed to be an all-or-nothing approach to problems, or atleasttotheproblemsIwasworkingon.Inthelaboratoryscheduling problem, the system would churn for 30 minutes and then either give us a beautiful schedule for the day or simply print "no." "No" in this casemeantthatwehadover-constrainedtheday, and there was no full solution. It did not, however, give us a partial solution or much of any informationaboutwheretheover-constraintwas.

What you see here is an extremely powerful concept. You don't have to describe the solution to a problem. You have only to describe the problem. And the language for the description of the problem is logic, onlypurelogic.Startfromfactsandinferences, and let Prolog do the rest. Prolog programs are at a higher level of abstraction. Schedules andbehaviorpatternsaregreatexamplesofproblemsrightinProlog's wheelhouse.

What We Learned in Day 1

Today, we learned the basic building blocks of the Prolog language. Ratherthan encoding steps to guide Prolog to a solution, we encoded knowledgeusing purelogic. Prolog did the hardwork of weaving that knowledgetogethertofindsolutions. We putourlogicintoknowledge basesandissuedqueriesagainstthem.

After we built a few knowledge bases, we then compiled and queried them. The queries had two forms. First, the query could specify a fact, and Prologwould tell us whether the facts were true or false. Second, webuiltaquerywithoneormorevariables. Prologthen computed all possibilities that made those fact strue.

We learned that Prolog worked through rules by going through the clausesforaruleinorder.Foranyclause,Prologtriedtosatisfyeachof the goals by going through the possible combinations of variables. All Prologprogramsworkthisway.

Inthesectionstocome, we'regoing to make more complex inferences. We also going to learn to use mathand more complex data structures suchaslists, as well as strategies to iterate overlists.

Find:

- SomefreePrologtutorials
- Asupportforum(thereareseveral)
- OneonlinereferenceforthePrologversionyou'reusing

Do:

- Make a simple knowledge base. Represent some of your favorite booksandauthors.
- Findallbooksinyourknowledgebasewrittenbyoneauthor.
- Makeaknowledgebaserepresentingmusiciansandinstruments. Alsorepresentmusicians and their genre of music.
- Findallmusicianswhoplaytheguitar.

4.3 Day2:FifteenMinutestoWpper

GrumpyJudgeWapner from The People's Court is an obsession of the centralcharacterin RainMan .Likemostautistics, Raymondobsesses overallthingsfamiliar.HelatchedontoJudgeWarrand ThePeople's Court. Asyou're plowing through this enigmatic language, you might be ready for things to start to click. Now, you might be one of the lucky readerswhohaseverythingclickforthemrightaway, butifyoudon't, takeheart. Today, there are definitely "fifteen minutes to Ward" Sit tight. We will need a few more tools in the tool box. Yallearn to use recursion, math, and lists. Let's getgoing.

Recursion

RubyandIowereimperativeprogramminglanguages. Yuwouldspell outeachstepofanalgorithm. Prologisthefirstofthedeclarativelanguageswe'lllookat.Whenyou'redealingwithcollectionsofthingssuch as lists or trees, you'll often use recursion rather than iteration. We look at recursion and use it to solve some problems with basic inferences, and then we'll apply the same technique to list sandmath.

Takealookatthefollowingdatabase. It expresses the extensive family treeofthe Wabs charactersina 1963 movieand subsequent series. Itexpressesafatherrelationshipandfromthatinferstheancestorrelationship. Since an ancestor can mean a father, grandfather, or great

grandfather, we will need to nest the rules or iterate. Since we're dealingwithadeclarativelanguage, we'regoing to nest. One clause in the ancestor clausewilluse ancestor. In this case, ancestor(Z,Y) isarecursive subgoal. Here's the knowledge base:

```
Download prolog/family.pl
father(zeb,
                      john_boy_sr).
father(john_boy_sr,john_boy_jr).
ancestor(X,Y):-
    father(X,Y).
ancestor(X,Y):-
    father(X,Z), ancestor(Z,Y).
```

is the core set of facts that enables our recursive subgoal. The ruleancestor/2 hastwoclauses. When you have multiple clauses that make uparule, only one of them must be true for the rule to be true. Think of the commas between subgoals as and conditions and the periods between clauses as or conditions. The first clauses ays" X isthe ancestor of Y if X is the the of Y."That's astraightforward relationship. We can trythatrulelikethis:

```
|?-ancestor(john_boy_sr,john_boy_jr).
true?
no
```

Prolog reports true, john_boy_sr is an ancestor of john_boy_jr. This first clausedependsonafact.

Thesecondclauseismorecomplex: ancestor(X,Y):-dteXZ),ancestor(Z, Y). This clause says X is an ancestor of Y if we can prove that the of Z andwecanalsoprovethatsame Z isan ancestor of Y.

Whew.Let'susethesecondclause:

```
|?-ancestor(zeb,john_boy_jr).
true?
```

zeb isanancestorof john_boy_jr.Asalways,wecantryvariablesin aquery, like this:

```
|?-ancestor(zeb,Who).
Who=john_boy_sr?a
Who=john_boy_jr
```

no

Andweseethat john_boy_ir and john_boy_sr.The zeb isanancestorfor ancestor predicatealsoworksinreverse:

```
|?-ancestor(Who,john_boy_jr).
Who=john_boy_sr?a
Who=zeb
(1ms)no
```

That's abeautiful thing, because we can use this rule in our knowledge base for two purposes, to find both ancestors and descendants.

Abriefwarning. When you use recursive subgoals, you need to be carefulbecauseeachrecursivesubgoalwillusestackspace, and you can eventuallyrunout.Declarativelanguagesoftensolvethisproblemwith a technique called tail raxin optimization . If you can position the recursive subgoal at the end of a recursive rule, Prolog can optimize the call to discard the call stack, keeping the memory use constant. Our call is tail recursive because the recursive subgoal, ancestor(Z,Y), isthelastgoalintherecursiverule. When your Prolog programs crash byrunningoutofstackspace, you'llknowit's time to look for away to optimizewithtailrecursion.

Whithatlastbitofhousekeepingoutoftheway,let's start to look at listsandtuples.

Listsand Litts

ListsandtuplesareabigpartofProlog.Yucanspecifyalistas [1,2,3]andatupleas (1,2,3) .Listsarecontainersofvariablelength,andtuples are containers with a fixed length. Both lists and tuples get much more powerfulwhenyouthinkofthemintermsofunification.

Unification,Rt2

Remember, when Prolog tries to unify variables, it tries to make both theleftandrightsides match. Two tuples can match if they have the same number of elements and each element unifies. Let's take a look atacoupleofexamples:

```
|?-(1,2,3)=(1,2,3).
|?-(1,2,3)=(1,2,3,4).
```

no

```
|?-(1,2,3)=(3,2,1).
```

no

Two tuples unify if all the elements unify. The first tuples were exactmatches, the second tuples did not have the same number of elements,and the third set did not have the same elements in the same order. Let'smixinsomevariables:

```
|?-(A,B,C)=(1,2,3).
A=1
B=2
C=3
ves
|?-(1,2,3)=(A,B,C).
A=1
B=2
C=3
yes
|?-(A,2,C)=(1,B,3).
A=1
B=2
C=3
yes
```

It doesn't really matter which sides the variables are on. They unify ifProlog can make them the same. Now, for some lists. They can work liketuples:

```
|?-[1,2,3]=[1,2,3].
yes
|?-[1,2,3]=[X,Y,Z].
X=1
Y=2
Z=3
ves
|?-[2,2,3]=[X,X,Z].
X=2
Z=3
yes
```

```
|?-[1,2,3]=[X,X,Z].
no
|?-[]=[].
```

```
Thelasttwoexamplesareinteresting. [X,XZ] and [2,2,3] unifiedbecause
Prologcould make them the same with X=2 \cdot [1,2,3] = [X,X,Z] did not
unifybecauseweused X forboththefirstandsecondpositions, and
those values were different. Lists have a capability that tuples don't.
                                 [Head ]. When you unify a list with
You can deconstruct lists with
this construct, Head will bind to the first element of the list, and
                                                                  The will
bindtotherest.likethis:
|?-[a,b,c]=[Head|Tail].
```

```
Head=a
Tail=[b,c]
yes
[Head|ti] won'tunifywithanemptylist,butaone-elementlistisfine:
|?-[]=[Head|Tail].
no
|?-[a]=[Head|Tail].
Head=a
Tail=[]
yes
Yucangetcomplicated by using various combinations:
|?-[a,b,c]=[a|Tail].
Tail=[b,c]
(1ms)yes
Prologmatchedthe a andunifiedtherestwith
                                                di .Orwecansplitthis
tailintotheheadandtail:
|?-[a,b,c]=[a|[Head|Tail]].
Head=b
Tail=[c]
yes
```

thiscopyis (P1.0printing,October2010)

Orgrabthethirdelement:

```
|?-[a,b,c,d,e]=[\_,\_|[Head|\_]].
Head=c
yes
```

_ isawildcardandunifieswithanything.Itbasicallymeans"Idon'tcare what's in this position." We told Prolog to skip the first two elements and splittherestintoheadandtail.The Head will grab the third element. andthetrailing _ willgrabthetail,ignoringtherestofthelist.

Thatshouldbeenoughtogetyoustarted. Unification is a powerful tool, and using it in conjunction with lists and tuples is even more powerful.

Now, you should have a basic understanding of the coredata structures inPrologandhowunificationworks.We nowreadytocombinethese elements with rules and assertions to do some basic math with logic.

ListsandMath

Inournextexample, Ithought I'd showyou an example of using recursionandmathtooperateonlists. These are examples to do counting, sums, and averages. Fiverules do all the hardwork.

```
Download prolog/list_math.pl
count(0,[]).
count(Count, [Head|Tail]):-count(TailCount, Tail), CountisTailCount+1.
sum(0,[]).
sum(Total,[Head|Tail]):-sum(Sum,Tail),TotalisHead+Sum.
average(Average, List):-sum(Sum, List), count(Count, List), AverageisSum/Count.
Thesimplestexampleis count. Useitlikethis:
|?-count(What,[1]).
What=1?;
no
```

Therulesaretriviallysimple. The count of an empty list is 0.Thecount ofalististhecountofthetailplusone.Let'stalkabouthowthisworks, step-by-step:

• We issuethequery count(What, [1]), which can't unify with the first rule, because the list is not empty. We moveontosatisfyingthe goalsforthesecondrule, count(Count, [Head] 1]). We unify, binding What to Count, Head to 1, and to [].

- Afterunification, the first goalis count(TailCount, []). Wetrytoprove that subgoal. This time, we unify with the first rule. That binds to 0.Thefirstruleisnowsatisfied, sowecan move onto troof thesecondgoal.
- +1. We canunifyvariables. To • Now, we evaluate Countis To Court Count isboundto0.sowebind Count to 0+1 .or 1.

Andthat'sit.We didnotdefinearecursiveprocess.We definedlogical rules. The next example is adding up the elements of a list. Here's the codeforthoserulesagain:

```
sum(0,[]).
sum(Total,[ Head|Tail]):-sum( Sum,Tail ),TotalisHead+Sum.
```

Thiscodeworkspreciselylikethecountrule.Italsohastwoclauses,a basecaseandtherecursivecase. The usage is similar:

```
|?-sum(What,[1,2,3]).
What=6?;
no
```

Ifyoulookatitimperatively, sum worksexactlyasyouwouldexpectin arecursivelanguage. The sum of an emptylistiszero; the sum of the restisthe Head plusthe sum of the Ti.

But there's another interpretation here. We haven't really told Prolog howtocomputesums. Whemerely described sums as rules and goals. To satisfy some of the goals, the logic engine must satisfy some subgoals. The declarative interpretation is as follows: "The sum of an empty listiszero.andthesumofalistis ifwecanprovethatthesumof thetailplustheheadis th ."Wereplacing recursion with the notion ofprovinggoalsandsubgoals.

Similarly, the count of an empty list is zero; the count of a list is one for the Head plusthe count of the di.

Aswithlogic, these rules can build on each other. For example, you can use sum and count togethertocomputeanaverage:

```
average(Average, List ):-sum( Sum, List ), count(Count , List ), Average is Sum/
                                                                                  Count.
```

So, the average of List is Aeoge ifyoucanprovethat

• the sum ofthat List is Sum.

- the count ofthat List is Count, and
- is Sum/Count. Aeoge

Anditworksjustasyou'dexpect:

```
|?-average(What,[1,2,3]).
What=2.0?;
no
```

UsingRulesinBothDirections

Atthispoint, you should have a fairly good understanding of how recursionworks.I'mgoingtoshiftgearsalittlebitandtalkaboutatightlittle rulecalled append. Therule append(List1, List2, List3) istrueif List3 is List1 + List2.It's a powerful rule that you can use in a variety of ways.

That short little bit of code packs a punch. You can use it in many differentways.It'saliedetector:

```
|?-append([oil], [water], [oil, water]).
|?-append([oil], [water], [oil, slick]).
no
It'salistbuilder:
|?-append([tiny],[bubbles],What).
What=[tiny,bubbles]
ves
Itdoeslistsubtraction:
|?-append([dessert_topping], Who, [dessert_topping, floor_wax]).
Who=[floor wax]
yes
Anditcomputespossible permutations:
|?-append(One,Two,[apples,oranges,bananas]).
0ne=[]
Two=[apples,oranges,bananas]?a
One=[apples]
```

thiscopyis (P1.0printing,October2010)

```
Two=[oranges,bananas]
One=[apples,oranges]
Two=[bananas]
One=[apples,oranges,bananas]
Two=[]
(1ms)no
```

So, one rule gives you four. You may think that building such a rule will take a lot of code. Let's find out exactly how much. Let's rewrite the Prolog append, but we'll call it concatenate.W takeitinseveral steps:

- 1. Wearule called concatenate(List1,List2,List3) that can concate nate anemptylistto List 1.
- 2. Addarulethatconcatenatesoneitemfrom List 1 onto List 2.
- 3. Addarule that concatenates two and three items from List1 onto List2.
- 4. Seewhatwecangeneralize.

Let's get started. Our first step is to concatenate an empty list to List1. That's a fairly easy rule to write:

```
Download prolog/concat_step_1.pl
concatenate([],List,List).
```

Noproblem. concatenate istrueifthefirstparameterisalistandthe nexttwoparametersarethesame.

Itworks:

```
|?-concatenate([],[harry],What).
What=[harry]
yes
```

Onto the next step. Let's add a rule that concatenates the first element of List1 tothefrontof List2:

```
Download prolog/concat_step_2.pl
concatenate([],List,List).
concatenate([Head|[]],List,[Head|List]).
```

For concatenate(List1, List2, List3), we break List1 into the head and tail, with the tail being an empty list. We break our third element into

thiscopyis (P1.0printing,October2010)

theheadandtail, using List 1's head and List 2 as the tail. Remember to compileyourknowledgebase.Itworksjustfine:

```
|?-concatenate([malfoy],[potter],What).
What=[malfoy,potter]
yes
```

Now, we can define another couple of rules to concatenate lists of lengths2and3. Theyworkinthesameway:

```
Download prolog/concat_step_3.pl
concatenate([],List,List).
concatenate([Head|[]],List,[Head|List]).
concatenate([Head1|[Head2|[]]],List,[Head1,Head2|List]).
concatenate([Head1|[Head2|[Head3|[]]]],List,[Head1,Head2,Head3|List]).
|?-concatenate([malfoy,granger],[potter],What).
What=[malfoy,granger,potter]
yes
```

So, what we have is a base case and a strategy where each subgoal shrinks the first list and grows the third. The second stays constant. We nowhaveenoughinformationtogeneralizearesult. Here's the concatenateusingnestedrules:

```
Download prolog/concat.pl
concatenate([],List,List).
concatenate([Head|Tail1],List,[Head|Tail2]):-
  concatenate(Tail1,List,Tail2).
```

Thatterselittleblockofcodehasanincrediblysimpleexplanation. The first clauses ay s concatenating an empty list to List gives you that List. The second clauses as y concatenating List 1 to List 2 gives you List 3 if the heads of List1 and List3 are the same, and you can prove that concatenating the tail of List 1 with List 2 gives you the tail of List3. The simplicity and elegance of this solution are a testament to the power of Prolog.

Let's seewhat it would do with the query concatenate([1,2], [3], What). walk through unification at each step. Keep in mind that we're nestingtherules, so each time we try to prove a subgoal, we'll have a differentcopyofthevariables. I'llmarktheimportantones with a letter soyoucankeepthemstraight. Wheachpass, I'll showwhath appens when Prolog tries to prove the next subgoal.

```
concatenate([1,2], [3], What)
```

• Thefirstruledoesn'tapply, because [1,2] isnotanemptylist.We unifytothis:

```
concatenate([1|[2]], [3], [1|72A
                                )):-concatenate( [2], [3], [73A
```

Everything unifies but the second tail. We now move on to the goals.Let'sunifytherightside.

• We trytoapplytherule concatenate([2], [3], [70A]).That'sgoingto giveusthis:

```
concatenate([2|[]], [3], [2|70B
                                  ]):-concatenate( [], [3], TB)
```

Noticethat 758 isthetailof **TA** .It'snotthesameastheoriginal & .Butnow, we have to unify the right side again.

- concatenate([], [3],750) :-concatenate([],[3],[3]).
- **TEC** is [3]. Now, we can work back up the chain. • So, we know Let's look at the third parameter, plugging in 700 at each step. **TEC** is [3], which means [2] 762] is [2,3], and finally [1|702] is [1,2,3]. What is [1,2,3].

Prologis doing a lot of work for you here. Go over this list until you understandit. Unifying nested subgoals is a core concept for the advancedproblemsinthisbook.

Now, you've taken a deep look at one of the richest functions in Prolog. Take a little time to explore these solutions, and make sure you understandthem.

WhatWeLearnedinDay2

In this section, we moved into the basic building blocks that Prolog usestoorganizedata: listsandtuples. We alsonestedrules, allowing us to express problems that you might handle with iteration in other languages. We tookadeeperlookat Prologunification and how Prolog works to match up both sides of a :- or =. We saw that when we're writing rules, we described logical rules instead of algorithms and let Prologworkitswaythroughthesolution.

We also used math. We learned to use basic arithmetic and nested subgoalstocomputesumsandaverages.

Finally, we learned to use lists. We matched one or more variables withinalist to variables, but more importantly, we matched the head ofalistandtheremainingelementswithvariablesusingthe [Head|**T**i] pattern. We used this technique to recursively iterate through lists. These building blocks will serve as the foundations of the complex problemswesolveinday3.

Day2Self-Study

Find:

- Some implementations of a Fibonacci series and factorials. How dotheywork?
- A real-world community using Prolog. What problems are they solvingwithittoday?

Ifyou'relookingforsomethingmoreadvancedtosinkyourteethinto, trytheseproblems:

- AnimplementationoftheTowersofHanoi.Howdoesitwork?
- Whataresomeoftheproblemsofdealing with "not" expressions? WhydoyouhavetobecarefulwithnegationinProlog?

Do:

- Reversetheelementsofalist.
- Findthesmallestelementofalist.
- Sorttheelementsofalist.

4.4 Day3:BlowingUpV egas

YushouldbegettingabetterunderstandingofwhylpickedtheRain Man, the autistics avant, for Prolog. Though it's sometimes difficult to understand, it's amazing to think of programming in this way. One of my favorite points in Rain Man was when Ray's brother realized he could count cards. Raymond and his brother went to Ves andjust about broke the bank. In this section, you're going to see a side of Prologthatwillleaveyousmiling. Coding the examples in this chapter wasequalpartsmaddeningandexhilarating. We goingtosolvetwo famouspuzzlesthatarerightinProlog'scomfortzone, solving systems withconstraints.

Yumaywanttotakeashotatsomeofthesepuzzlesyourself. Ifyou do, try describing the rules you know about each game rather than showingPrologastep-by-stepsolution.Wegoingtostartwithasmall Sudokuandthengiveyouachancetobuilduptoalargeroneinthe dailyexercises. Then, we'll move onto the classic Eight Queenspuzzle.

SolvingSudoku

CodingtheSudokuwasalmostmagicalforme.ASudokuisagridthat hasrows, columns, and boxes. A typical puzzle is an ine-by-nine grid, withsomespacesfilledinandsomeblank. Each cell in the gridhasa number, from 1-9 for an ine-by-nine square. Yarjobist of illout the solution so that each row, column, and square has one each of all of thedigits.

going to start with a four-by-four Sudoku. The concepts are ex-Water actly the same, though the solutions will be shorter. Let's start by describingtheworld, as we know it. Abstractly, we'll have a board with four rows, four columns, and four squares. The table shows squares 1-4:

```
1
  1
     2
        2
1
  1 2 2
3
 3 4 4
3
  3
     4
        4
```

The first task is to decide what the query will look like. That's simple have a puzzle and a solution, of the form sodoku(Puzzle, enough. We Solution). Our users can provide a puzzle as a list, substituting underscoresforunknownnumbers.likethis:

```
sodoku([_,_,2,3,
       _,_,_,
       _,_,_,
       3,4,_,_],
```

Ifasolutionexists, Prologwill provide the solution. When I solved this puzzleinRuby,Ihadtoworryaboutthealgorithmforsolvingthepuzzle. Wh Prolog, that's not so. I merely need to provide the rules for the game. These are the rules:

- Forasolvedpuzzle, the numbers in the puzzle and solutions hould bethesame.
- ASudokuboardisagridofsixteencells, with values from 1-4.

- Theboardhasfourrows, fourcolumns, and four squares.
- Apuzzleisvalidiftheelementsineachrow,column,andsquare hasnorepeatedelements.

Let's start at the top. The numbers in the solution and puzzle should match:

```
Download prolog/sudoku4_step_1.pl
sudoku(Puzzle,Solution):-
         Solution=Puzzle.
```

Worke actuallymadesomeprogress.Our"Sudokusolver"worksforthe casewheretherearenoblanks:

```
|?-sudoku([4,1,2,3,
             2,3,4,1,
             1,2,3,4,
             3,4,1,2], Solution).
Solution=[4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]
yes
```

The format isn't pretty, but the intent is clear enough. We getting sixteennumbersback,rowbyrow.Butwearealittletoogreedy:

```
|?-sudoku([1,2,3],Solution).
Solution=[1,2,3]
yes
```

Now, this board is n't valid, but our solver reports that there is a valid solution. Clearly, we have to limit the board to sixteen elements. We haveanotherproblem, too. The values in the cells can be anything:

```
|?-sudoku([1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6],Solution).
Solution=[1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6]
ves
```

For a solution to be valid, it should have numbers from 1-4. This problemwillimpactusintwoways.First,wemayallowsomeinvalidsolutions. Second, Prologdoesn't have enough information to test possible valuesforeachcell. Inotherwords, these to fresults is not grounded. Thatmeansthatwehavenotexpressedrulesthatlimitpossiblevalues of each cell, so Prolog will not be able to guess what the values are.

thiscopyis (P1.0printing,October2010)

Let's solve these problems by solving the next rule to the game. Rule 2 says a board has sixteen cells, with values from 1-4. GNU Prolog hasabuilt-inpredicatetoexpresspossiblevalues, called fd domain(List, LowerBound, UpperBound). This predicate is true if all the values in List are between LowerBound and UpperBound, inclusive. We justneedtomake sureallyaluesin Puzzle rangefrom 1-4.

```
Download prolog/sudoku4_step_2.pl
sudoku(Puzzle, Solution):-
         Solution=Puzzle,
         Puzzle=[S11,S12,S13,S14,
                    S21, S22, S23, S24,
                    S31,S32,S33,S34,
                    S41, S42, S43, S441,
         fd_domain(Puzzle,1,4).
```

We unified Puzzle with a list of sixteen variables, and we limited the domain of the cells to values from 1-4. Now, we fail if the puzzle is not valid:

```
|?-sudoku([1,2,3],Solution).
no
|?-sudoku([1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6],Solution).
no
```

Now, we get to the main piece of the solution. Rule 3 says a board consistsofrows, columns, and squares. We goingtocarvethepuzzle upintorows, columns, and squares. Now, you can see why we named thecells the way we did. It's a straightforward process to describe the rows:

```
Row1=[S11,S12,S13,S14],
Row2=[S21,S22,S23,S24],
Row3=[S31,S32,S33,S34],
Row4=[S41,S42,S43,S44],
```

```
Coll=[S11,S21,S31,S41],
Col2=[S12,S22,S32,S42],
Col3=[S13,S23,S33,S43],
Col4=[S14,S24,S34,S44],
Andsquares:
Square1=[S11,S12,S21,S22],
Square2=[S13,S14,S23,S24],
```

Square3=[S31,S32,S41,S42], Square4=[S33,S34,S43,S44].

Nowthatwe'vechoppedtheboardintopieces, we can move onto the nextrule. The board is valid if all rows, columns, and squares have no repeatedelements. Was ea GNU Prolog predicate to test for repeated elements. fd_all_different(List) succeedsifalltheelementsin List aredifferent. We need to build a rule to test that all rows, columns, and squares arevalid.W useasimpleruletoaccomplishthis:

```
valid([]).
valid([Head|Tail]):-
    fd_all_different(Head),
    valid(Tail).
```

This predicate is valid if all the lists in itare different. The first clause says that an empty list is valid. The second clause says that a list is validifthefirstelement's items are all different and if the rest of the listisvalid.

Allthatremainsistoinvokeour rule: √q**p**qgy/

```
valid([Row1,Row2,Row3,Row4,
       Col1, Col2, Col3, Col4,
       Square1,Square2,Square3,Square4]).
```

Believe it or not, we're done. This solution can solve a four-by-four Sudoku:

```
|?-sudoku([_,_,2,3,
             _,_,_,
             _,_,_,
             3,4,_,_],
             Solution).
Solution=[4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]
yes
```

thiscopyis (P1.0printing,October2010)

Breakingthatintoafriendlierform, we have the solution:

```
4
  1
     2
        3
  3
     4 1
2
1 2
     3 4
     1
        2
3
```

Here's the completed program, beginning to end:

```
Download prolog/sudoku4.pl
valid([]).
valid([Head|Tail]):-
    fd_all_different(Head),
    valid(Tail).
sudoku(Puzzle, Solution):-
        Solution=Puzzle,
        Puzzle=[S11,S12,S13,S14,
                   S21, S22, S23, S24,
                   S31, S32, S33, S34,
                   S41,S42,S43,S44],
        fd_domain(Solution, 1, 4),
        Row1=[S11,S12,S13,S14],
        Row2=[S21, S22, S23, S24],
        Row3=[S31,S32,S33,S34],
        Row4=[S41,S42,S43,S44],
        Coll=[S11,S21,S31,S41],
        Col2=[S12,S22,S32,S42],
        Col3=[S13,S23,S33,S43],
        Col4=[S14,S24,S34,S44],
        Square1=[S11,S12,S21,S22],
        Square2=[S13,S14,S23,S24],
        Square3=[S31,S32,S41,S42],
        Square4=[S33,S34,S43,S44],
        valid([Row1, Row2, Row3, Row4,
                Col1, Col2, Col3, Col4,
                Square1,Square2,Square3,Square4]).
```

Ifyouhaven'thadyourPrologmoment,thisexampleshouldgiveyoua nudgeintherightdirection. Where 'stheprogram? We wedidn'twrite described the rules of the game: a board has sixteen a program. We cells with digits from 1-4, and none of the rows, columns, or squares $should repeat any of the values. The {\tt puzzletook} a few dozen lines of$

thiscopyis (P1.0printing,October2010)

code to solve and no knowledge of any Sudoku solving strategies. In thedailyexercises, you'll get the chance to solve an ine-row Sudoku. It won'tbetoodifficult.

This puzzle is a great example of the types of problems Prolog solves well. We have a set of constraints that are easy to express but hard to solve. Let's look at another puzzle involving highly constrained resources:theEightQueensproblem.

EightQueens

To solve the Eight Queens problem, you put eight queens on a chess board. None can share the same row, column, or diagonal. It may appear to be a trivial problem on the surface. It's just a kid's game. Butonanotherlevel, you can look at the rows, columns, and diagonals as constrained resources. Our industry is full of problems that solve constrained systems. Let's look at how we can solve this one in Prolog.

First, we'll look at what the query should look like. We can express each queenas (Row,Col), atuplehaving arowand a column. A of tuples. eight_queens(Board) succeeds if we have a valid board. Our querywilllooklikethis:

```
eight_queens([(1,1),(3,2),...]).
```

Let'slookatthegoalsweneedtosatisfytosolvethepuzzle. If youwant totakeashotatthisgamewithoutlookingatthesolution, justlookat thesegoals. Iwon't show the full solution untill a terinthe chapter.

- Aboardhaseightqueens.
- Eachqueenhasarowfrom 1–8 and a column from 1–8.
- Notwoqueenscansharethesamerow.
- Notwoqueenscansharethesamecolumn.
- Notwoqueenscansharethesamediagonal(southwesttonortheast).
- Notwoqueenscansharethesamediagonal(northwesttosoutheast).

Rowsandcolumnsmustbeunique, butwemustbemorecareful with diagonals. Each queen is on two diagonals, one running from the lower left (northwest) to the upper right (southeast) and the other running from the upper left to the lower right as in Figure 4.2, on the following page.Buttheserulesshouldberelativelyeasytoencode.

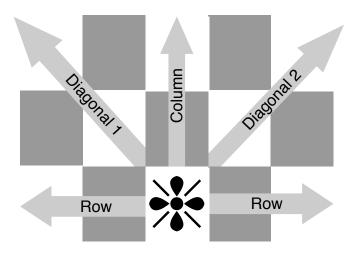


Figure 4.2: Eight Queensrules

Onceagain, we'll start at the top of the list. Aboard has eight queens. That means our list must have a size of eight. That's easy enough to do. We can use the count predicate you sawear lier in the book, or we can simply use a built-in Prolog predicate called length. length(List, N) succeeds if List has N elements. This time, rather than showyou each goal in action, I'm going to walk you through the goals we'll need to solvethewholeproblem.Here'sthefirstgoal,then:

```
eight_queens(List):-length(List ,8).
```

Next, we need to make sure each queen from our list is valid. We build aruletotestwhetheraqueenisvalid:

```
valid_queen((Row,Col)):-
    Range=[1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).
```

The predicate member does just what you think; it tests for membership.Aqueenisvalidifboththerowandcolumnareintegersfrom 1-8. Next,we'llbuildaruletocheckwhetherthewholeboardismadeupof validqueens:

```
valid board([]).
valid_board([Head|Tail]):-valid_queen(Head),valid_board(Tail).
```

Anemptyboardisvalid, and aboardisvalid if the first item is a valid queenandtherestoftheboardisvalid.

Movingon, the nextrule is that two queens can't share the same row. To solve the next few constraints, we're going to need a little help. We willbreakdowntheprogramintopiecesthatcanhelpusdescribethe problem:whataretherows,columns,anddiagonals?Firstupisrows.

buildafunctioncalled **pwQers**Rows) .Thisfunctionshouldbe trueif Rows isthelistof Row elements from all the queens.

```
rows([],[]).
rows([(Row,_)|QueensTail],[Row|RowsTail]):-
  rows(QueensTail,RowsTail).
```

Thisonetakesalittleimagination, but not much. ows for an emptylist is Rows if the Row from the first isanemptylist, and pwsQueens Rows) Rows and if nws of the queen in the list matches the first element of tailof Queens isthetailof Rows. Ifit's confusing to you, walk through it withafewtestlists.Luckily,columnsworksexactlythesameway,but we'regoingtousecolumnsinsteadofrows:

```
cols([],[]).
cols([(_,Col)|QueensTail],[Col|ColsTail]):-
 cols(QueensTail,ColsTail).
```

The logic works exactly the same as ow, but we match the second elementofaqueentupleinsteadofthefirst.

Movingon, we'regoing to number diagonals. The easiest way to number themistodosomesimplesubtractionandaddition. If northandwest are 1, we're going to assign the diagonals that run from northwest to southeastavalue of Col - Row. This is the predicate that grabs those diagonals:

```
diags1([],[]).
diags1([(Row,Col)|QueensTail],[Diagonal|DiagonalsTail]):-
 DiagonalisCol-Row,
 diags1(QueensTail,DiagonalsTail).
```

That rule worked just like rows and cols, but we had one more constraint: DiagonalisCol--Row . Note that this is not unification! It's an is predicate, and it will make sure that the solution is fully grounded. Finally, we'll grab the southeast to northwest like this:

```
diags2([],[]).
diags2([(Row,Col)|QueensTail],[Diagonal|DiagonalsTail]):-
 DiagonalisCol+Row,
 diags2(QueensTail,DiagonalsTail).
```

thiscopyis (P1.0printing,October2010)

Theformulaisalittlebittricky, buttry a few values untilyou're satis fied that queens with the same sum of row and col are in fact on the same diagonal. Now that we have the rules to help us describe rows, columns, and diagonals, all that remains is to make surerows, columns, and diagonals are all different.

Soyoucanseeitallincontext, here's the entire solution. The tests for rowsandcolumnsarethelasteightclauses.

```
Download prolog/queens.pl
valid_queen((Row,Col)):-
    Range=[1,2,3,4,5,6,7,8],
   member(Row, Range), member(Col, Range).
valid_board([]).
valid_board([Head|Tail]):-valid_queen(Head), valid_board(Tail).
rows([],[]).
rows([(Row,_)|QueensTail],[Row|RowsTail]):-
 rows(QueensTail,RowsTail).
cols([],[]).
cols([(_,Col)|QueensTail],[Col|ColsTail]):-
 cols(QueensTail,ColsTail).
diags1([],[]).
diags1([(Row,Col)|QueensTail],[Diagonal|DiagonalsTail]):-
 DiagonalisCol-Row,
 diags1(QueensTail,DiagonalsTail).
diags2([],[]).
diags2([(Row,Col)|QueensTail],[Diagonal|DiagonalsTail]):-
 DiagonalisCol+Row,
 diags2(QueensTail, DiagonalsTail).
eight_queens(Board):-
 length(Board,8),
 valid_board(Board),
  rows (Board, Rows),
 cols(Board,Cols),
 diags1(Board, Diags1),
 diags2(Board, Diags2),
  fd_all_different(Rows),
 fd_all_different(Cols),
  fd_all_different(Diags1),
  fd_all_different(Diags2).
```

At this point, you could run the program, and it would run... and run...andrun.Therearejusttoomanycombinationstoefficientlysort through. If you think about it, though, we know that there will be one andonlyqueenineveryrow.We canjumpstartthesolutionbyprovidingaboardthatlookslikethis:

```
|?-eight\_queens([(1,A),(2,B),(3,C),(4,D),(5,E),(6,F),(7,G),(8,H)]).
A=1
B=5
C=8
D=6
E=3
F=7
G=2
H=4?
```

Thatworksjustfine, butthe programiss till working to ohard. We can eliminate the rowchoice squite easily and simplify the API while we're atit.Here'saslightlyoptimizedversion:

```
Download prolog/optimized_queens.pl
valid_queen((Row,Col)):-member(Col,[1,2,3,4,5,6,7,8]).
valid board([]).
valid_board([Head|Tail]):-valid_queen(Head), valid_board(Tail).
cols([],[]).
cols([(_,Col)|QueensTail],[Col|ColsTail]):-
 cols(QueensTail,ColsTail).
diags1([],[]).
diags1([(Row,Col)|QueensTail],[Diagonal|DiagonalsTail]):-
 DiagonalisCol-Row,
 diags1(QueensTail,DiagonalsTail).
diags2([],[]).
diags2([(Row,Col)|QueensTail],[Diagonal|DiagonalsTail]):-
 DiagonalisCol+Row,
 diags2(QueensTail,DiagonalsTail).
eight queens(Board):-
 Board=[(1, ), (2, ), (3, ), (4, ), (5, ), (6, ), (7, ), (8, )]
 valid_board(Board),
 cols(Board, Cols),
 diags1(Board, Diags1),
 diags2(Board, Diags2),
  fd_all_different(Cols),
  fd_all_different(Diags1),
  fd_all_different(Diags2).
```

Philosophically, we've made one major change. We matched the Board with (1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,)to reduce the total permutations significantly. We also removed all rules related to rows, and the results show. On myancient MacBook, all solutions compute insideofthreeminutes.

Once again, the end result is quite pleasing. We built in very little knowledgeofthesolutionset.We justdescribedtherulestothegame and applied a little logic to speed things up a little. Given the right problems, I could really find myself getting into Prolog.

WhatWeLearnedinDay3

Today, you put to gether some of the ideas we've used in Prolog to solve someclassic puzzles. The constraint-based problems have many of the samecharacteristicsasclassicindustrialapplications.Listconstraints, andcrunchoutasolution.We wouldneverthinkofdoingaSQLninetablejoinimperatively, yetwedon't even blink at solving logical problemsinthisway.

WestartedwithaSudokupuzzle.Prolog'ssolutionwasremarkablysimple. We mapped sixteen variables onto rows, columns, and squares. Then, we described the rules of the game, forcing each row, column, and square to be unique. Prolog then methodically worked through the possibilities, quickly arriving at a solution. We usedwildcardsandvariablestobuildanintuitiveAPI,butwedidn'tprovideanyhelpatallfor solutiontechniques.

Next, we used Prolog to solve the Eight Queens puzzle. Once again, weencoded the rules of the game and let Prolog work into a solution. This classic problem was computationally intensive, having 92 possible solutions, but even our simple approach could solve it within a handful ofminutes.

I still don't know all of the tricks and techniques to solve advanced Sudokus, but with Prolog, Idon't need to know them. I only need the rulesofthegametoplay.

Day3Self-Study

Find:

- Prologhassome input/output features as well. Find print predicatesthatprintoutvariables.
- Find a way to use the print predicates to print only successful solutions. Howdotheywork?

- Modifythe Sudoku solver towork on six-by-six puzzles (squares are3x2)and9x9puzzles.
- MaketheSudokusolverprintprettiersolutions.

If you're a puzzle enthusiast, you can get lost in Prolog. If you want to dive deeper into the puzzles I've presented, Eight Queens is a good placetostart.

 SolvetheEightQueensproblembytakingalistofqueens.Rather thanatuple, represente a chqueen with an integer, from 1-8. Get therowofaqueenbyitspositioninthelistandthecolumnbythe valueinthelist.

4.5 WrappingUpProlog

Prologisoneoftheolderlanguagesinthisbook, buttheideasarestill interesting and relevant today. Prolog means programming with logic. We used Prolog to process rules, composed of clauses, which were in turncomposed with a series of goals.

Prolog programming has two major steps. Start by building a knowledgebase, composed of logical facts and inferences about the problem domain.Next,compileyourknowledgebase,andaskquestionsabout the domain. Some of the questions can be assertions, and Prolog will respondwith so or no. Otherqueries have variables. Prologfills in these gapsthatmakesthosequeriestrue.

Rather than simple assignment, Prologuses a process called unification thatmakesvariablesonbothsidesofasystemmatch. Sometimes, Prologhastotrymanydifferentpossiblecombinationsofvariablesto unifyvariablesforaninference.

Strengths

Prologisapplicableforawidevarietyofproblems, from airlineschedulingtofinancialderivatives. Prologhas a serious learning curve, but the demandingproblemsthatPrologsolvestendtomakethelanguage,or otherslikeit.worthwhile.

ThinkbacktoBrianTarbox'sworkwiththedolphins. Hewasableto makesimpleinferencesabouttheworldandmakeabreakthroughwith a complex inference about dolphin behavior. He was also able to take

Natural-LanguageProcessing

Prolog was perhaps first used to work with language recognition. In particular, Prolog language models can take natural language, apply a knowledge base of facts and inferences, and express that complex, inexactlanguageinconcreterulesappropriateforcomputers.

Games

Games are getting more complex, especially modeling the behavior of competitors or enemies. Prolog models can easily express the behavior of other characters in the system. Prolog can also build different behaviors into different types of enemies, making a more lifelike and enjoyableexperience.

Semantic Web

The semantic Web is an attempt to attach meaning to the services and information on the Web making it easier to satisfy requests. The resource description language (RDF) provides a basic description of resources. Aserver can compile these resources into a knowledge base. That knowledge, together with Prolog's natural-language processing, can provide a rich end user experience. Many Prolog packages exist for providing this sort of functionality in the context of a webserver.

ArtificialIntelligence

Artificialintelligence(AI)centersaroundbuildingintelligenceintomachines. This intelligence can take different forms, but in every case, some "agent" modifies behavior based on complex rules. Prolog excels inthisarena, especially when the rules are concrete, based on formal logic. Forthis reason, Prologis sometimes called a logic programming language.

Scheduling

Prologexcels in working with constrained resources. Many have used Prologtobuildoperatingsystemschedulersandotheradvancedschedulers.

Wedkress

Prologisalanguagethathasheldupovertime. Still, thelanguageis datedinmanyways, and it does have significant limitations.

While Prolog excels in its core domain, it's a fairly focused niche, logic programming. It is not a general-purpose language. It also has some limitationsrelatedtolanguagedesign.

Very LargeDataSets

Prologuses a depth-first search of a decision tree, using all possible combinationsmatchedagainstthesetofrules. Vins languagesand compilers do a prettygo odjobo foptimizing this process. Still, the strategyisinherentlycomputationallyexpensive, especially as datasets get verylarge.ItalsoforcesProloguserstounderstandhowthelanguage workstokeepthesizeofdatasetsmanageable.

MixingtheImperativeandDeclarativeModels

Like many languages in the functional family, particularly those that relyheavilyonrecursion, youmustunderstandhow Prologwillresolve must often have tail-recursive rules to complete recursive rules. You evenmoderatelylargeproblems. It's relatively easy to build Prologapplicationsthatcannotscalebeyondatrivialsetofdata. Yumustoften have a deep understanding of how Prolog works to effectively design rulesthatwillscaleatacceptablelevels.

FinalThoughts

AsIworkedthroughthelanguagesinthisbook, Ioftenkickedmyself, knowingthatthroughtheyears, I'vedrivenmanyascrewwithasledgehammer. Prolog was a particularly poignant example of my evolving understanding. If you find a problem that's especially well suited for Prolog, takeadvantage. In such a setting, you can be stuse this rulesbasedlanguageincombinationwithothergeneral-purposelanguages, justasyouwoulduseSQLwithinRubyorJava.Ifyou'recarefulwith the way you tie them together, you're likely to come out a head in the longrun.

Chapter 5 Scala

Sofar, I have introduced three languages and three different programming paradigms. Scala will be the fourth, sort of. It's a hybrid language, meaning that it intentionally tries to bridge the gaps between programming paradigms. In this case, the bridge is between objectorientedlanguageslikeJavaandfunctionallanguageslikeHaskell.In thissense. Scalaisa Frankensteinmonster of sorts but not a monster. Think EdwardScissorhands 1

InthissurrealTim Burtonmovie,Edwardwaspartboy,partmachine, with scissors for hands, and was one of my favorite characters of all time. Edward was a fascinating character in a beautiful movie. He wasoftenawkward, wassometimesamazing, butalwayshadaunique expression. Sometimes, his scissors let him do incredible things. Other all the contractions of the contraction of the contratimes, hewasawkwardandhumiliated. Aswithanything neword ifferent, hewas often misunders tood, accused of "straying too far from the pathofrighteousness."Butinoneofhisstrongermoments, the shykid offers, "Wearenotsheep." Indeed.

5.1 **AboutScala**

Asrequirements for computer programs get more complex, languages, too, must evolve. Every twenty years or so, the old paradigms become inadequatetohandlethenewdemandsfororganizingandexpressing ideas.Newparadigmsmustemerge,buttheprocessisnotasimpleone.

^{1.} EdwardScissorhands .DVD.DirectedbyTim Burton. 1990; BeverlyHills, CA: 20th CenturyFox,2002.

Each new programming paradigm ushers in a wave of programming languages, not just one. The initial language is often strikingly productive and wildly impractical. Think Small talk for objects or Lisp for functional languages. Then, languages from other paradigms build in features that allow people to absorb the new concepts while users canlive safely within the old paradigm. Ada, for example, allowed some coreobject-orientedideassuchasencapsulationtoexistwithinaprocedurallanguage. At some point, some hybridlanguage offersjust the rightpractical bridge between the old paradigmand the new, such as aC++.Next, we see a commercially adoptable language, such as Java orC#.Finally, we see some mature, pure implementations of the new paradigm.

Affinity with Jac

Scalaisatleastabridgeandmaybemore. Itofferstightintegration into Java, offering a chance for people to protect their investment in many ways:

- ScalarunsontheJavavirtualmachine,soScalacanrunside-bysidewithexistingdeployments.
- Scala can use Java libraries directly, so developers can leverage existingframeworksandlegacycode.
- LikeJava, Scalais statically typed, so the languages share aphilosophicalbond.
- Scala'ssyntaxisrelativelyclosetoJava's,sodeveloperscanlearn thebasicsquickly.
- Scalasupportsbothobject-orientedandfunctionalprogramming paradigms, so programmers can gradually learn to apply functionalprogrammingideastotheircode.

WithoutSlavishDevotion

Some languages that embrace their ancestors go too far, extending the very limiting concepts that make the base inadequate. Although the similarities to Java are striking, Scala's design has some significant departures that will serve its community well. These important improvementsrepresentimportantdeparturesfromtheJavalanguage:

• Typeinference. In Java, you must declare the type of every variable, argument, or parameter. Scala infers variable types where possible.

- Functional concepts. Scalaintroduces important functional concepts to Java. Specifically, the new language allows you to use existing functions in many different ways to form new ones. Conceptsyou'llseeinthischapterarecodeblocks, higher-orderfunctions, and a sophisticated collection library. Scalagoes far beyond somebasicsyntacticalsugar.
- Immutable variables. Java does allow immutable variables but with a rarely used modifier. In this chapter, you'll see that Scala forces you to explicitly make a decision about whether a variableis mutable. These decisions will have a profound effect on how applications behave in a concurrent context.
- Advanced programming constructs. Scala uses the foundational languagewell, layering on useful concepts. In this chapter, we'll introduce you to actors for concurrency, Ruby-style collections withhigher-orderfunctions, and first-class XML processing.

Beforewedivein, we should know about the motivations behind Scala. spendsometime with the creator, focusing on how he decided to tietwoprogrammingparadigmstogether.

AnInterviewwithScala'sCreator,MartinOdersky

MartinOdersky, the creator of Scala, is a professor at École PolytechniqueFédéraledeLausanne(EPFL),oneoftwoSwissFederalInstitutes of Technology. Hehasworked on the Java Generics specification and is thecreatorofthe javac referencecompiler. Heisalsotheauthorof ProgramminginScala:AComprehensiveStep-by-StepGuide [OSV08], one ofthebestScalabooksavailabletoday.Hereiswhathehadtosay:

Bruce: WhydidyouwriteScala?

Dr. Odersky: Iwas convinced that unifying functional and object-oriented programming would have great practical value. I was frustrated both by the dismissive attitude of the functional programming community toward OOP and by the belief of object-oriented programmers that functional programming was just an academic exercise. So, I wanted to showthatthetwoparadigmscanbeunifiedandthatsomethingnewand powerful could **Est** from that combination. I also wanted to create a language in which I would personally feel comfortable writing programs.

Bruce: Whatisthethingyoulikeaboutithemost?

Dr. Odersky: Ilikethatitletsprogrammersexpressthemselvesfreely and that it feels light weight, yet at the same time gives strong support throughitstypesystem.

Bruce: Whatkindsofproblemsdoesitsolvethebest?

Dr. Odersky: It's xdygeneral purpose. There's no problem I would nottrytosolvewithit. That said, aparticular strength of Scalartie to othermainstreamlanguagesisitssupportoffunctionalprogramming.So, everywhereafunctionalapproachisimportantScalashines,bethatconcurrencyandparallelismorwebappsdealingwithXMLorimplementing domain-specificlanguages.

Bruce: What is a feature that you would like to change, if you could startover?

Dr. Odersky: Scala's local type inference works generally well but has limitations. If I could start over, I'd try to use a more powerful constraint solver. Maybeit's still possible to do that, but the fact that we have to dealwithalargeinstalledbasemakesitmoredifficult.

ThebuzzaroundScalaisgrowing,becauseTwitterhasswitcheditscore message processing from Ruby to Scala. The object-oriented features allowaprettysmoothtransitionfromtheJavalanguage,buttheideas that are drawing attention to Scala are the functional programming features. Purefunctional languages allow a style of programming that has strong mathematical foundations. A functional language has these characteristics:

- Functional programs are made upof functions.
- Afunctional ways returns a value.
- Afunction, given the same inputs, will return the same values.
- Functional programs avoid changing state or mutating data. Once you'vesetavalue, youhavetoleaveitalone.

Strictlyspeaking, Scalaisnotapure functional programming language, justlikeC++isnotapureobject-orientedlanguage.Itallowsmutable values, which can lead to functions with the same inputs but different outputs. (With most object-oriented languages, using getters and setters would break that rule.) But it of fers tools that allow developers tousefunctionalabstractionswheretheymakesense.

FunctionalProgrammingandConcurrency

The biggest problem facing concurrency-minded programmers in object-orientedlanguagestodayis mutable state, meaning data that can change. Anyvariable that can hold more than one value, after initialization, is mutable. Concurrency is the Dr. Eviltomutable state's Austin Powers. If two different threads can change the same data at the sametime, it's difficult to guarantee that the execution will leave the data in avalidstate, and testing is nearly impossible. Databases deal with this problem with transactions and locking. Object-oriented programming languages deal with this problem by giving programmers the tools to control access to shared data. And programmers generally don't use thosetoolsverywell, even when they know how.

Functional programming languages can solve these problems by eliminating mutable state from the equation. Scala does not force you to completely eliminate mutable state, but it does give you the tools to codethingsinapurelyfunctionalstyle.

WhScala, voudon't have to choose between making some Smalltalk andhavingalittleLisp.Let'sgetbusymergingtheobject-orientedand functionalworldswithsomeScalacode.

5.2 Day 1: The Castle on the Hill

In Edward Scissorhands, there's a castle on a hill that is, well, a littledifferent.Inabygoneera,thecastlewasastrangeandenchanting place but is now showing signs of age and neglect. Broken windows let the weather in, and the rooms aren't all what they once were. The housethatoncefeltsocomfortabletoitsinhabitantsisnowcoldand uninviting. The object-oriented paradigm, too, is showing some signs ofage, especially the earlier object-oriented implementations. The Java language, with its dated implementations of static typing and concurrency, needs a face-lift. In this section, we're going to talk primarily aboutScalainthecontextofthathouseonthehill, the object-oriented programmingparadigm.

ScalarunsontheJavavirtualmachine(JVM).I'mnotgoingtoofferan exhaustive overview of the Java language; that information is freely see some Java ideas creeping through to available elsewhere. Yil Scala, but I'll trytominimize their impacts oy ouw on't have to learn two languagesatonce. Fornow, install Scala. I'musing version 2.7.7. final forthisbook.

ScalaTypes

Whenyou have Scalaworking, fireup the console with the command scala. If all is well, you won't get any error messages, and you will see a scala> prompt. Yucanthentypealittlecode.

```
scala>println("Hello,surrealworld")
Hello, surrealworld
scala>1+1
res8:Int=2
scala>(1).+(1)
res9:Tnt=2
scala>5+4
res10:Int=17
scala>5.+(4. *(3))
res11:Double=17.0
scala>(5).+((4). *(3))
res12:Int=17
```

So, integers are objects. In Java, I've pulled out my fair share of hair converting between Int (primitives) and Integer (objects). In fact, everythingisanobjectinScala, with some small exceptions. That's a significant departure from most statically typed object oriented languages. Let'sseehowScalahandlesstrings:

```
scala>"abc".size
res13:Int=3
```

Soastring,too,isafirst-classobject,withalittlesyntacticsugarmixed in.Let'strytoforceatypecollision:

```
scala>"abc"+4
res14:java.lang.String=abc4
scala>4+"abc"
res15:java.lang.String=4abc
scala>4+"1.0"
res16: java.lang.String=41.0
```

Hm...that's not quite what we were looking for. Scalais coercing those integers into strings. Let's tryalittle harder to force a mismatch:

```
scala>4 * "abc"
                                            * withalternatives(Double)Double
<console>:5:error:overloadedmethodvalue
<and>(Float)Float<and>(Long)Long<and>(Int)Int<and>(Char)Int
<and>(Short)Int<and>(Byte)Intcannotbeappliedto(java.lang.String)
       4 * "abc"
         ٨
```

Ah. That's the ticket. Scala is actually strongly typed. Scala will use typeinference, somost of the time, it will understand the types of variables through syntactical clues, but unlike Ruby, Scala can do that typechecking at compiletime. Scala's console actually compiles lines ofcodeandrunseachonepiecemeal.

On a side note, I know you're getting back Java strings. Most Scala articles and books go into this topic in more detail, but we can't do so and still dive into the programming constructs that It hink will be mostinteresting to you. I'll point you to a few books that will go into the Javaintegrationindetail. For now, I'm going to telly out hat in many places, Scalahasastrategy formanaging types across two languages. Part of that is using simple Java types where they make sense, like java.lang.String.Pleasetrustme,andaccepttheseoversimplifications.

Expressions and Conditions

Nowwe'regoingtomovethroughsomebasicsyntaxquicklyandstrictly byexample.HereareafewScalatrue/falseexpressions:

```
scala>5<6
res27:Boolean=true
scala>5<=6
res28:Boolean=true
scala>5<=2
res29:Boolean=false
scala>5>=2
res30:Boolean=true
scala>5!=2
res31:Boolean=true
```

There's nothing too interesting going on there. This is the C-style syntaxthatyou'refamiliarwithfromseveralofthelanguageswe'vetalked aboutsofar.Let'suseanexpressioninan if statement:

```
scala>vala=1
a:Int=1
scala>valb=2
b:Int=2
scala>if(b<a){</pre>
     | println("true")
     |}else{
     println("false")
     1}
false
```

We assign a couple of variables and compare the min an if/else statement. Takea closer look at the variable assignment. First, notice that you didn't specify a type. Unlike Ruby, Scala binds types at compile time. But unlike Java, Scala can infer the type, so you don't have to ,thoughyoucanifyouwant. type vda:Int=1

Next, notice that these Scala variable declarations start with the M keyword. Yucanal souse the va keyword. va isimmutable; va isnot. WŁ talkmoreaboutthislater.

In Ruby, 0 evaluated to true. In C, 0 was the . In both languages, nil evaluated to the .Let's see how Scalahandles them:

```
scala>Nil
res3:Nil.type=List()
scala>if(0){println("true")}
<console>:5:error:typemismatch;
found
        :Int(0)
required:Boolean
       if(0){println("true")}
scala>if(Nil){println("true")}
<console>:5:error:typemismatch;
found
         :Nil.type(withunderlyingtypeobjectNil)
 required:Boolean
       if(Nil){println("true")}
          ٨
```

So, a Nil isanemptylist, and you can't event est Nil or 0. This behaviorisconsistentwithScala'sstrong, statictypingphilosophy. Nilsand numbersarenot booleans.sodon'ttreatthemlike booleans. Winsimpleexpressions and the most basic decision construct behind us, let's moveontoloops.

Loops

As the next couple of programs get more complex, we're going to run themasscriptsratherthanintheconsole.LikeRubyandIo,you'llrun themwith scalapath/to/program.scala.

Yalseeanumberofwaystoiterateoverresultsetsinday2whenwe attackcodeblocks.Fornow,we'llfocusontheimperativeprogramming style of loops. Yall see that these look a lot like the Java-style loop structures.

MyInnerBattlewithStaticTyping

Some novice programming language enthusiasts confuse the ideas of strong typing and static typing. Loosely speaking, strongtyping means the language detects when two types are compatible, throwing an error or coercing the types if they are not. On the surface, and and Ruby are both strongly typed. (Irealize this idea is an axin diatr) Assembly language and C compilers, on the other hand, are welly typed. The compiler doesn't necessarily care whether the data in a memory location is an integer, astring, or just data.

Staticanddynamictypingisanotherissue. Staticallytypedlanguages enforce polymorphism based on the structure of the types. Is it a duck by the genetic blueprint (static), or is it a duckbecause it quacksorwds likeone? Statically typedlanguages benefit because compilers and tools know more about varcode to traperrors, highlight code, and refactor. The cost is having to domore was and living with some restrictions. Yo history as a developer will often determine how you feel about the trade-offs of static typing.

My first OO development was in Jaal saw one framework after another try to break free from the chains of Jaac's static typing. The industry invested hundreds of millions of dollars in three vains of Enterprise Jaa Beans, Spring, Hibernate, JBoss, and aspect-oriented programming to make certain usage models more makade. We was making Jaac's typing model more dynamic, and the battles at easy step of the way was intense, failing more like rival cults than programming environments. Mybookstookthesame journey, from increasingly dynamic frameworkstodynamic languages.

So, my bias against static typing was haped by the waws. Haskell and its great static typesystem are helping mere cover but slowly. My conscience is clear. Yo u've invited a close tpolitician to this casual dinner, but I'll try my hardest to keep the conversation light and unbiased.

ur

```
Download scala/while.scala
defwhileLoop{
    vari=1
    while(i<=3){</pre>
          println(i)
          i+=1
     }
}
```

whileLoop

We define a function. As a side note, Java developers will notice that youdon'thavetospecify public.InScala, public isthedefaultvisibility, meaningthisfunctionwillbevisibletoall.

Winthemethod, we declare a simple while loopthatcountstothree. i changes, so we declare it with va . Then, you see a Java-style declaration of a while statement. As you can see, the code inside braces executesunlesstheconditionis the Yucanrunthecodelikethis:

```
batate$scalacode/scala/while.scala
1
2
3
```

The to loopworksalotlikeJava'sandC'sbutwithaslightlydifferent syntax:

```
Download scala/for_loop.scala
defforLoop{
    println( "forloopusingJava-styleiteration"
                                                           )
    for(i<-Ountilargs.length){</pre>
         println(args(i))
    }
}
```

forLoop

The argument is a variable, followed by the <- operator, followed by a rangefortheloopintheformof initialValueuntilendingValue .Inthiscase, we'reiterating over the incoming command-linear guments:

```
batate$scalacode/scala/forLoop.scalaitsallinthegrind
forloopusingJava-styleiteration
its
a11
in
the
grind
```

As with Ruby, you can also use loops to iterate over a collection. For now,we'llstartwith ,whichisreminiscentofRuby's (beach each:

```
Download scala/ruby_for_loop.scala
defrubyStyleForLoop{
    println( "forloopusingRuby-styleiteration"
                                                         )
    args.foreach{arg=>
        println(arg)
    }
}
```

rubyStyleForLoop

orgs is a list with the inbound command-linear guments. Scalapasses eachelementintothisblock, one by one. In our case, arg isoneargument from the inbound args list. In Ruby, the same code would be args.each { |arg| println(arg)}. The syntax for specifying each argument isslightlydifferent, but theide aisthesame. Here's the code in action:

```
batate$scalacode/scala/ruby_for_loop.scalafreezethosekneeschickadees
forloopusingRuby-styleiteration
freeze
those
knees
chickadees
```

Later, you'll find yourself using this method of iteration much more oftenthantheotherimperativeloops. Butsincewe'reconcentratingon thehouseonthehill, we'lldelay that part of the conversation for a little while.

Ranges and Lipts

Like Ruby, Scala supports first-class ranges. Start the console, and enterthesecodesnippets:

```
scala>valrange=0until10
range: Range=Range(0,1,2,3,4,5,6,7,8,9)
scala>range.start
res2:Int=0
scala>range.end
res3:Int=10
```

Thatallmakessense. Itworkslike Ruby's range. Yucanal so specify increments:

```
scala>range.step
res4:Int=1
```

```
scala>(0to10)by5
res6:Range=Range(0,5,10)
scala>(0to10)by6
res7:Range=Range(0,6)
The equivalent of Ruby's range, 1..10, is 1 to 10, and the equivalent of
Ruby'srange, 1...10, is luntil10. to isinclusive:
scala>(Ountil10by5)
res0:Range=Range(0,5)
Yucanalsospecifydirectionwiththis:
scala>valrange=(10until0)by-1
range:Range=Range(10,9,8,7,6,5,4,3,2,1)
Butthedirectionisnotinferred:
scala>valrange=(10until0)
range:Range=Range()
scala>valrange=(0to10)
range: Range. Inclusive=Range(0,1,2,3,4,5,6,7,8,9,10)
1 is the default step, regardless of the endpoints that you express for
yourrange. Youarenotlimited to integers:
scala>valrange='a'to'e'
range: Random Access Seq. Projection [Char] = Random Access Seq. Projection (a, b, c, d, e)
Scalawilldosomeimplicittypeconversionsforyou.Infact,whenyou
specifieda & statement, youwere actually specifying arange.
LikeProlog,Scalaofferstuples.Atupleisafixed-lengthsetofobjects.
Yalfindthispatterninmanyotherfunctionallanguagesaswell. The
objectsinatuplecanallhavedifferenttypes.Inpurelyfunctionallan-
guages, programmers often express objects and their attributes with
tuples. Try this example:
scala>valperson=("Elvis", "Presley")
person:(java.lang.String,java.lang.String)=(Elvis,Presley)
scala>person._1
res9:java.lang.String=Elvis
scala>person. 2
res10: java.lang.String=Presley
scala>person._3
<console>:6:error:value_3isnotamemberof(java.lang.String,java.lang.String)
       person._3
              ٨
```

Scalausestuplesratherthanliststodomultivalueassignments:

```
scala>val(x,y)=(1,2)
x:Int=1
v:Int=2
```

Since tuples have a fixed length, Scala can do static type checking basedoneachofthetuplevalues:

```
scala>val(a,b)=(1,2,3)
<console>:15:error:constructorcannotbeinstantiatedtoexpectedtype;
 found :(T1,T2)
 required:(Int,Int,Int)
       val(a,b)=(1,2,3)
<console>:15:error:recursivevaluex$1needstype
       val(a,b)=(1,2,3)
            ٨
```

Wh these foundations out of the way, let's put it all together. We createsomeobject-oriented class definitions.

ClassesinScala

Thesimplest classes, those with attributes but no methods or constructors, are simple, one-line definitions in Scala:

```
classPerson(firstName:String,lastName:String)
```

Yudon'thavetospecifyanybodytospecifyasimplevalueclass. The classwill be public and have firstName and lastName attributes. Reacon Andyoucanusethatclassintheconsole:

```
scala>classPerson(firstName:String,lastName:String)
definedclassPerson
scala>valgump=newPerson("Forrest","Gump")
gump:Person=Person@7c6d75b6
```

But you're looking for a little more. Object-oriented classes mix data andbehavior.Let'sbuildafullobject-orientedclassinScala.We call this class Composs. The compassorientation will start with north. We tellthecompasstoturn90degreesleftorrightandupdatethedirection accordingly. Here's what the Scalacodelooks like, in its entirety:

```
Download scala/compass.scala
classCompass{
  valdirections=List( "north", "east", "south", "west")
  varbearing=0
```

```
print("Initialbearing:" )
 println(direction)
 defdirection()=directions(bearing)
 definform(turnDirection:String){
   println("Turning" +turnDirection+
                                         ".Nowbearing"
                                                         +direction)
 }
 defturnRight(){
   bearing=(bearing+1)%directions.size
   inform("right")
 }
 defturnLeft(){
    bearing=(bearing+(directions.size-1))%directions.size
   inform("left")
 }
}
valmyCompass=newCompass
myCompass.turnRight
myCompass.turnRight
myCompass.turnLeft
myCompass.turnLeft
myCompass.turnLeft
```

Thesyntaxisrelativelystraightforward, with a couple of notable peculiarities. The constructor is responsible for defining instance variables (atleast, thoseyoudon't passint othe constructor) and methods. UnlikeRuby,allmethoddefinitionshaveparametertypesandnames.And theinitialblockofcodeisn'tinanymethoddefinitionatall.Let'stake itapart:

```
classCompass{
 valdirections=List( "north", "east", "south", "west")
 varbearing=0
 print("Initialbearing:" )
 println(direction)
```

The whole block of code following the class definition is actually the constructor. Our constructor has a List of directions and a bearing, which is simply an index for the directions. Later, turning will manipulatethebearing. Next, there are a couple of convenience methods to showtheuseroftheclassthecurrentdirection,inEnglish:

```
defdirection()=directions(bearing)
```

```
definform(turnDirection:String){
 println("Turning" +turnDirection+
                                       ".Nowbearing"
                                                        +direction)
}
```

The constructor continues with method definitions. The direction methodjustreturnstheelementof directions at the index of bearing. Scala conveniently allows an alternate syntax for one-line methods, omitting thebraces around the method body.

The inform methodprintsafriendlymessagewhenevertheuserturns.It takesasimpleparameter, the direction of the turn. This method doesn't returnavalue.Let'slookatthemethodstohandleturns.

```
defturnRight(){
 bearing=(bearing+1)%directions.size
 inform("right")
}
defturnLeft(){
 bearing=(bearing+(directions.size-1))%directions.size
 inform("left")
}
```

The turns method changes the bearing based on the direction of the turn. The % operatorismodular division. (This operator does a division operation, discarding the quotient and returning only the remainder.) The result is that right turns add one to the bearing and left turns subtractone, wrapping the result accordingly.

Auty Constructors

Yive seen how the basic constructor works. It's a code block that initializes classes and methods. Yucanhave alternate constructors as class, with two constructors: well.Considerthis Recon

```
Download scala/constructor.scala
classPerson(first_name:String){
 println("Outerconstructor" )
 defthis (first_name:String,last_name:String){
    this(first_name)
    println("Innerconstructor" )
 deftalk()=println(
                        "Hi")
}
valbob=newPerson(
                      "Bob")
                          "Bob", "Tate")
valbobTate=newPerson(
```

The class has a constructor with one parameter, firstName.andamethod called talk. Notice the this method. That 's the second constructor. It takes

twoparameters, firstName and lastName. Initially, themethod invokes this withtheprimaryconstructor, withouly the firstName parameter.

The code after the class definition instantiates a person in two ways, first with the primary constructor, then next with the auxiliary constructor:

```
batate$scalacode/scala/constructor.scala
Outerconstructor
Outerconstructor
Innerconstructor
```

That's all there is to it. Auxiliary constructors are important because they allow for a broad array of usage patterns. Let's look at how to createclassmethods.

ExtendingClasses

Sofar, the classes have been pretty vanilla. We created a couple of basic classes with nothing more than attributes and methods. In this section, we'lllookatsomeofthewaysthatclassescaninteract.

CompanionObjectsandClassMethods

InJavaandRuby, you create both class methods and instance methods withinthesamebody.InJava,classmethodshavethe static keyword. Rubyuses defself.class_method.Scalauses neither of these strategies. Instead, you will declare instancemethods in the class definitions. When there's something that can have only one instance, you'll define it with the object keywordinsteadofthe class keyword. Here's an example:

```
Download scala/ring.scala
objectTrueRing{
 defrule=println(
                       "Torulethemall"
}
```

TrueRing.rule

The Lefta definitionworksexactlylikeany class definition.butitcreates a singleton object. In Scala, you can have both an object definition and a closs definition with the same name. Using this scenario, you can create class methods within the singleton object declaration and instancemethods within the class declaration. In our example, the method rule isaclassmethod. This strategy is called companionobjects.

Inheritance

InheritanceinScalaisprettystraightforward, butthesyntaxmustbe exact. Here's an example of extending a Reson class with Employee. Noticethatthe Employee hasanadditionalemployeenumberinthe field.Here'sthecode:

```
Download scala/employee.scala
classPerson(valname:String){
  deftalk(message:String)=println(name+
                                               "savs"
                                                        +message)
  defid():String=name
}
classEmployee(overridevalname:String,
                        valnumber:Int)extendsPerson(name){
 overridedeftalk(message:String){
    println(name+ "withnumber"
                                    +number+
                                                "savs"
                                                         +message)
 }
  overridedefid():String=number.toString
}
valemployee=newEmployee(
                             "Yoda",4)
employee.talk("Extendorextendnot.Thereisnotry."
```

In this example, we're extending the Reason base class with Employee. adding a new instance variable called number in Employee, and Wate we'realsooverridingthe talk messagetoaddsomenewbehavior. Most of the tricky syntax is around the class constructor definition. Notice that you must specify the complete parameter list for , though Reason youcanomitthetypes.

The owner keyword, both in the constructor and for any methods you want to extend from the base class, is mandatory. This keyword will keep you from inadvertently introducing new methods with misspellings. Allinall, there are no major surprises here, but at times, Ido feelabitlikeEdwardtryingtopetafragilebabybunny.Movingon....

Tts

Everyobject-orientedlanguagemustsolvetheproblemthatoneobject can have several different roles. An object can be a persistent, serializableshrubbery. Yudon'twantyourshrubberytohavetoknowhow topushbinarydataintoMySQL.C++usesmultipleinheritance,Java usesinterfaces, Rubyusesmixins, and Scalausestraits. A Scalatrait is like a Ruby mixin, implemented with modules. Or, if you prefer, a trait is like a Java interface plus an implementation. Lookat a trait as

apartial-classimplementation. Ideally, its hould implement one critical concern. Here's an example that adds the trait Nice to Reson

```
Download scala/nice.scala
classPerson(valname:String)
traitNice{
 defgreet()=println(
                         "Howdilydoodily." )
classCharacter(overridevalname:String)extendsPerson(name)withNice
valflanders=newCharacter(
                               "Ned")
flanders.greet
```

The first element you see is Reason . It is a simple class with a single attributecalled name. The second element is the trait called Nice. That is the mixin. It has a single method called greet. The final element, a classicalled Character, mixes in the Nicetrait. Clients can now use the greet method on any instance of Character. The output is what you wouldexpect:

```
batate$scalacode/scala/nice.scala
Howdilydoodily.
```

There's nothing too complicated here. We cantakeourtraitcalled Nice withamethodcalled greet andmixitintoanyScalaclasstointroduce the greet behavior.

WhatWeLearnedinDay 1

We coveredatremendousamountofterritoryinday1becausewehave tofullydeveloptwodifferentprogrammingparadigmsinonelanguage. Day I showed that Scalaembraces object-oriented concepts, running in the JVM side-by-side with existing Javalibraries. Scala's syntaxis similar to that of Java and is also strongly and statically typed. But Martin Odersky wrote Scala to bridge two paradigms: object-oriented programming and functional programming. These functional programming concepts that we'll introduce in day 2 will make it easier to designconcurrentapplications.

Scala's statictyping is also inferred. Users do not always need to declare types for all variables in all situations because Scala can often infer thosetypesfromsyntacticalclues. The compiler can also coerce types, such as integers to strings, allowing implicit type conversions when theymakesense.

Scala's expressions work much like they do in other languages, but they arealittlemorestrict. Most conditional smusttake aboolean type, and O or Nil will not work at all; they can substitute for neither .Butthere's nothing dramatically different about Scala's looping or controlstructures. Scaladoes supports ome more advanced types, like tuples(fixed-lengthlistswithheterogeneoustypes)andranges(afixed, all-inclusiveorderedsequenceofnumbers).

ScalaclassesworkmuchliketheydoinJava, buttheydon't support classmethods.Instead,Scalausesaconceptcalled companionobjects to mix class and instance methods on the same class. Where Ruby uses mixins and Java uses interfaces, Scala uses a structure like a mixincalleda Ti

In day 2, we'll take a full pass through Scala's functional features. cover code blocks, collections, immutable variables, and some advancedbuilt-inmethodslike **that**

Day 1 Self-Study

The first day of Scalac overed a lot of ground, but it should be mostly familiar territory. These object-oriented concepts should be familiar to you. These exercises are a little advanced compared to the earlier exercisesinthebook, butyou can handleit.

Find:

- TheScalaAPI
- AcomparisonofJavaandScala
- Adjscussion of vd versus va

Do:

- Weagamethatwilltakeatic-tac-toeboardwithX,O,andblank characters and detect the winner or whether there is a tie or no winneryet. Useclasses where appropriate.
- Bonusproblem:Lettwoplayersplaytic-tac-toe.

5.3 Day2:ClippingBushesandOtherNewTds

In Edward Scissorhands, a magical moment happens when Edward realizes that he's come far from the house on the hill and his unique abilities may give him a special place in the existing society.

Anyone with an eye for programming language history has seen this fable played out before. When the object-oriented paradigm was new, the masses could not accept Small talk because the paradigm was toonew. We needed a language that would let them continue to do procedural programming and experiment with object-oriented ideas. Wh C++, thenewobject-oriented tricks could live safely be side the existing Cproceduralfeatures. The result was that people could start using the newtricksinanoldcontext.

Now, it's time to put Scalathrough its paces as a functional language. Some of this will seem awkward at first, but the ideas are powerful and important. They will form the foundation for the concurrency constructsyou'llseelaterinday3.Let's start from the beginning, with a simplefunction:

```
scala>defdouble(x:Int):Int=x
                                  * 2
double:(Int)Int
scala>double(4)
res0:Int=8
```

Defining a function looks a whole lot like it does with Ruby. The def keyword defines both a function and a method. The parameters and their types come next. After that, you can specify an optional return type. Scalacan often infer the return type.

Toinvokethefunction, justusethename and the argument list. Notice that unlike Ruby, the parentheses are not optional in this context.

This is a one-line method definition. You can also specify a method definitioninblockform:

```
scala>defdouble(x:Int):Int={
     | x * 2
     1}
double: (Int)Int
scala>double(6)
res3:Int=12
```

That=afterthe Int return type is mandatory. Forgetting it will cause youtrouble. These are the major forms of function declarations. Yell see minor variations, such as omitting parameters, but these are the formsvou'llseemostoften.

Let's move on to the variables that you'll use within a function. Yil want to pay careful attention to the life cycle of the variable if you wanttolearnthepurefunctionalprogrammingmodel.

DV22EVIDV

Scalais based on the Java virtual machine and has a tight relationship with Java. In some ways, these design goals limit the language. In other ways, Scala can take advantage of the last fifteen or twenty years of programming language development. Yil see an increased emphasis on making Scala friendly for concurrent programming. But all the concurrency features in the world won't help you if you don't followbasicdesignprinciples.Mutablestateisbad.Whenyoudeclare variables, you should make the mimmutable whenevery ou can to avoid conflictingstate.InJava,thatmeansusingthe final keyword.InScala, immutablemeansusing vd insteadof va:

```
scala>varmutable="Iammutable"
mutable: java.lang.String=Iammutable
scala>mutable="Touchme,changeme..."
mutable:java.lang.String=Touchme,changeme...
scala>valimmutable="Iamnotmutable"
immutable:java.lang.String=Iamnotmutable
scala>immutable="Can'ttouchthis"
<console>:5:error:reassignmenttoval
       immutable="Can'ttouchthis"
```

So. vg values are mutable: vd values are not. In the console, as a convenience, you can redefine a variable several times even if you use vd .Onceyoustepoutsideoftheconsole,redefininga vd willgenerate anerror.

Insomeways, Scalahadtointroduce the va -stylevariablestosupport thetraditionalimperative programming style, but while you'r elearning Scala, it's best to avoid w whenyoucanforbetterconcurrency. This basicdesignphilosophyisthekeyelementthatdifferentiatesfunctional programming from object-oriented programming: mutable state limits concurrency.

Let's move onto some of my favorite are as within functional languages, dealingwith collections.

Collections

Functional languages have a long history of spectacularly useful featuresforcollections. One of the earliest functional languages, Lisp, was built around the idea of dealing with lists. The very name stands for

LISt Processing. Functional languages make it easy to build complex structures containing data and code. Scala's primary collections are lists, sets, and maps.

Lists

As with most functional languages, the bread-and-butter data structureisthelist. Scala's lists, of type List, are ordered collections of like thingswithrandomaccess. Entertheselists into the console:

```
scala>List(1,2,3)
res4:List[Int]=List(1,2,3)
```

Notice the first return value: List[Int] = List(1,2,3) . This value not only shows the type of the overall list but also shows the type of the data structures within the list. A list of Stringslookslikethis:

```
scala>List("one","two","three")
res5:List[java.lang.String]=List(one,two,three)
```

If you're seeing a little Java influence here, you're right. Java has a featurecalledGenericsthatallowsyoutotypetheitemswithinadata structurelikealistorarray.Let'sseewhathappenswhenyouhavea listcombining Stringsand Ints:

```
scala>List("one","two",3)
res6:List[Any]=List(one,two,3)
```

You get the data type Any, which is the catchall data type for Scala. Here'showyou'daccessanitemofalist:

```
scala>List("one","two",3)(2)
res7:Any=3
scala>List("one","two",3)(4)
java.util.NoSuchElementException:headofemptylist
        atscala.Nil$.head(List.scala:1365)
        atscala.Nil$.head(List.scala:1362)
        atscala.List.apply(List.scala:800)
        at.<init>(<console>:5)
        at.<clinit>(<console>)
        atRequestResult$.<init>(<console>:3)
        atRequestResult$.<clinit>(<console>)
        atRequestResult$result(<console>)
        atsun.reflect.NativeMethodAccessorImpl.invokeO(NativeMet...
```

() instead Yousethe () operator.Listaccessisafunction,soyouuse of [].Scala'sindexforliststartswith0,asitdoeswithJavaandRuby. Unlike Ruby, accessing an item out of range will throw an exception.

You can try to index with a negative number. Earlier versions return thefirstelement:

```
scala>List("one","two",3)(-1)
res9:Any=one
scala>List("one","two",3)(-2)
res10:Any=one
scala>List("one","two",3)(-3)
res11:Any=one
```

Sincethatbehaviorisalittleinconsistentwiththe NoSuchElement exceptionforanindexthat's toolarge, version 2.8.0 corrects that behavior, returning java.lang.lndexOutOfBoundsException.

Onefinalnote. Nil inScalaisanemptylist:

```
scala>Nil
res33:Nil.type=List()
```

usethislistasabasicbuildingblockwhenwecovercodeblocks, butfornow, bearwithme. I'm going to introduce a couple of other types of collections first.

Sets

Asetislikealist, butsets do not have any explicit order. Y uspecify a setwiththe Set keyword:

```
scala>valanimals=Set("lions","tigers","bears")
animals:scala.collection.immutable.Set[java.lang.String]=
    Set(lions, tigers, bears)
```

Addingorsubtractingfromthatsetiseasy:

```
scala>animals+"armadillos"
res25:scala.collection.immutable.Set[java.lang.String]=
    Set(lions, tigers, bears, armadillos)
scala>animals-"tigers"
res26:scala.collection.immutable.Set[java.lang.String]=Set(lions,bears)
scala>animals+Set("armadillos", "raccoons")
<console>:6:error:typemismatch;
        :scala.collection.immutable.Set[java.lang.String]
 required: java.lang.String
       animals+Set("armadillos", "raccoons")
```

Keepinmindthatsetoperationsarenotdestructive. Each setoperation builds a new set rather than modifying the old ones. By default, sets

areimmutable. Asyoucansee, addingorremoving a single element is a piece of cake, but you can't use the + or - to combine sets, as you wouldinRuby.InScala,youwanttouse ++ and -- forsetunion and setdifference:

```
scala>animals++Set("armadillos","raccoons")
res28:scala.collection.immutable.Set[java.lang.String]=
 Set(bears, tigers, armadillos, raccoons, lions)
scala>animals--Set("lions", "bears")
res29:scala.collection.immutable.Set[java.lang.String]=Set(tigers)
```

Yucanalsoperformsetintersection(elementsintwosetsthatarethe same)with **2:

```
scala>animals ** Set("armadillos", "raccoons", "lions", "tigers")
res1:scala.collection.immutable.Set[java.lang.String]=Set(lions,tigers)
```

Unlike a List, a Set is independent of order. This rule will mean that equalityforsetsandlistsisdifferent:

```
scala>Set(1,2,3)==Set(3,2,1)
res36:Boolean=true
scala>List(1,2,3)==List(3,2,1)
res37:Boolean=false
```

That's enough set manipulation for now. Let's move on tomaps.

Maps

A Map isakey-valuepair,likeaRuby Hash.Thesyntaxshouldbefamiliartoyou:

```
scala>valordinals=Map(0->"zero",1->"one",2->"two")
ordinals:scala.collection.immutable.Map[Int,java.lang.String]=
  Map(0\rightarrow zero, 1\rightarrow one, 2\rightarrow two)
scala>ordinals(2)
res41: java.lang.String=two
```

Like a Scala List or Set, you specify a Map with the Map keyword. You separate the elements of the map with the -> operator. You justused some syntactic sugar that makes it easy to create a Scala map. Let's useanotherformofthehashmapandspecifythetypesofthekeyand value:

```
scala>importscala.collection.mutable.HashMap
importscala.collection.mutable.HashMap
```

^{2.} Use&beginninginScala2.8.0,because**isdeprecated.

```
scala>valmap=newHashMap[Int,String]
map:scala.collection.mutable.HashMap[Int,String]=Map()
scala>map+=4->"four"
scala>map+=8->"eight"
scala>map
res2:scala.collection.mutable.HashMap[Int,String]=
 Map(4->four,8->eight)
```

First, weimport the Scalalibraries for a mutable HashMap.Thatmeans the values within the hash map can change. Next, we declare an immutablevariablecalled mp .Thatmeansthatthe referre: tothemap cannot change. Notice that we're also specifying the types of the keyvaluepairs. Finally, weaddsomekey-valuepairs and return the result.

Here's what would happen if you specified the wrong types:

```
scala>map+="zero"->0
<console>:7:error:overloadedmethodvalue+=withalternatives(Int)map.MapTo
    <and>((Int,String))Unitcannotbeappliedto((java.lang.String,Int))
      map+="zero"->0
```

As expected, you get a typing error. The type constraints are enforced wherepossibleatcompiletimebutalsoatruntime. Sonowthatyou've seenthebasicsforcollections, let's dive into some of the finer details.

AnyandNothing

Beforewemoveontoanonymous functions, let's talk a little bit more about the class hierarchy in Scala. When you're using Scalawith Java,youwilloftenbemoreconcernedabouttheJavaclasshierarchy.Still, youshouldknowalittlebitabouttheScalatypes. Any istherootclass in the Scala class hierarchy. It's often confusing, but know that any Scalatypewillinheritfrom Any.

Similarly, Nothing is a subtype of every type. That way, a function, say foracollection, canreturn Nothing and conform to the return value for the given function. It is all laid out in Figure 5.1, on the next page. Everythinginheritsfrom Any, and Nothing inheritsfromeverything.

Thereareafewdifferentnuances when you're dealing with nilconcepts. Null is a to , and null is an instance of it that works like Java's null. meaninganemptyvalue. An emptycollection is Nil.Bycontrast, Nothing is a trait that is a subtype of everything. Nothing has no instance, so

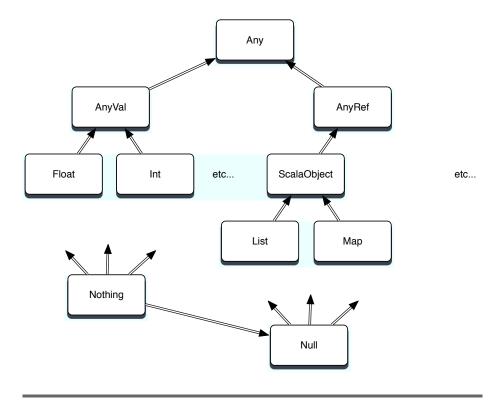


Figure 5.1: Any and Nothing

youcan'tdereferenceitlike Null.Forexample,amethodthatthrowsan Nothing, meaning novalue at all. Exception has the return type

Keep those rules in the back of your mind, and you'll be fine. Now, you'rereadytodoalittlebitmorewithcollectionsusinghigher-order functions.

Collections and Functions

Aswestartonlanguages that have a stronger functional foundation, Iwant to formalize some of the concepts we've been working with all along.Thefirstsuchconceptis higher-orderfunctions.

As with Ruby and Io, Scala collections get a whole lot more interestingwithhigher-orderfunctions.JustasRubyused each and Ioused , Scalawill letyou pass functions into (beach .The underlying (beach concept that you've been using all along is the higher-order function. Inlayman's terms, a higher-order function is one that produces or consumesfunctions. More specifically, a higher-order function is one that takesotherfunctions as input parameters or returns functions as output. Composing functions that use other functions in this way is a criticalconceptforthefunctionalfamilyoflanguagesandonethatwill shapethewayyoucodeinotherlanguagesaswell.

Scalahaspowerfulsupportforhigher-orderfunctions. We don't have time to look at some of the advanced topics such as partially applied functionsorcurrying, but we will learn top assimple functions, often called codeblocks, asparameters into collections. Yucantakea function and assign it to any variable or parameter. Yu can pass them intofunctions and return them from functions. We goingtofocuson anonymous functions as input parameters to a few of the more interestingmethodsoncollections.

(beach

Thefirstfunctionwe'regoingtoexamineis foeath .theiterationworkhorseinScala.AswithIo.the fæoch methodonacollectiontakesa codeblockasaparameter. In Scala, you'll express that codeblock in theform voidate von Cook likethis:

```
scala>vallist=List("frodo", "samwise", "pippin")
list:List[java.lang.String]=List(frodo,samwise,pippin)
scala>list.foreach(hobbit=>println(hobbit))
frodo
samwise
pippin
```

hobbit => println(hobbit) is an anonymous function, meaning a function without a name. The declaration has the arguments to the left of the =>,andthecodeistotheright. callstheanonymousfunction, ίσεοσh passingineachelementofthelistasaninputparameter. Asyoumight haveguessed, you can use the same technique for sets and maps too, thoughtheorderwon't beguaranteed:

```
valhobbits=Set("frodo", "samwise", "pippin")
hobbits:scala.collection.immutable.Set[java.lang.String]=
 Set(frodo,samwise,pippin)
scala>hobbits.foreach(hobbit=>println(hobbit))
frodo
samwise
pippin
scala>valhobbits=Map("frodo"->"hobbit",
  "samwise"->"hobbit", "pippin"->"hobbit")
```

```
hobbits:scala.collection.immutable.Map[java.lang.String,java.lang.String]=
 Map(frodo->hobbit,samwise->hobbit,pippin->hobbit)
scala>hobbits.foreach(hobbit=>println(hobbit))
(frodo, hobbit)
(samwise, hobbit)
(pippin,hobbit)
```

Ofcourse, maps will return tuples instead of elements. Asyour ecall, youcanaccesseitherendofthetuple, likethis:

```
scala>hobbits.foreach(hobbit=>println(hobbit._1))
frodo
samwise
niggig
scala>hobbits.foreach(hobbit=>println(hobbit._2))
hobbit
hobbit
hobbit
```

Whthese an onymous functions, you can do far more than just iterate. I'mgoingtowalkyouthroughsomebasicsandthenafewoftheother interesting ways Scalauses functions in conjunction with collections.

MoreListMethods

I'm going to take a brief diversion to introduce a few more methods onList. These basic methods provide the feature syou'll need to do manual iterationorrecursionoverlists. First, herearethemethods to test for theemptystateorcheckthesize:

```
scala>list
res23:List[java.lang.String]=List(frodo,samwise,pippin)
scala>list.isEmpty
res24:Boolean=false
scala>Nil.isEmpty
res25:Boolean=true
scala>list.length
res27:Int=3
scala>list.size
res28:Int=3
```

Noticethatyoucancheckthesizeofalistwithboth length and size. Also, rememberthattheimplementation of Nil isanemptylist. As with Prolog, it's useful to be able to grab the head and tail of a list for recursion.

```
scala>list.head
res34: java.lang.String=frodo
scala>list.tail
res35:List[java.lang.String]=List(samwise,pippin)
scala>list.last
res36:java.lang.String=pippin
scala>list.init
res37:List[java.lang.String]=List(frodo,samwise)
```

There's a surprise. You canuse head and toil to recurse head first, or lost and init to recurse tail first. Wido a little more with recursion later. Let'swrapupthebasicswithafewinterestingconveniencemethods:

```
scala>list.reverse
res29:List[java.lang.String]=List(pippin,samwise,frodo)
scala>list.drop(1)
res30:List[java.lang.String]=List(samwise,pippin)
scala>list
res31:List[java.lang.String]=List(frodo,samwise,pippin)
scala>list.drop(2)
res32:List[java.lang.String]=List(pippin)
```

These do just about what you'd expect. reverse returns the list with invertedordering, and drop(n) returns the list with the first n elements removed, without modifying the original list.

count, map, filter, and Others

AswithRuby,Scalahasmanyotherfunctionsthatmanipulatelistsin various ways. You can filter the list to match a given condition, sort a list using whatever criteria you want, create other lists using each elementasaninput, and create aggregate values:

```
scala>valwords=List("peg","al","bud","kelly")
words:List[java.lang.String]=List(peg,al,bud,kelly)
scala>words.count(word=>word.size>2)
res43:Int=3
scala>words.filter(word=>word.size>2)
res44:List[java.lang.String]=List(peg,bud,kelly)
scala>words.map(word=>word.size)
res45:List[Int]=List(3,2,3,5)
```

```
scala>words.forall(word=>word.size>1)
res46:Boolean=true
scala>words.exists(word=>word.size>4)
res47:Boolean=true
scala>words.exists(word=>word.size>5)
res48:Boolean=false
```

We start with a Scala list. Then, we count all the words with a size greater than two. count will call the code block word => word => 2evaluating the expression wode>2 foreachelementinthelist.The count methodcountsallthe true expressions.

In the same way, woodfe(wood => woode > 2)returns a list of all wordsthathaveasizegreaterthantwo, muchlikeRuby's select.Using the same pattern, mp builds a list of the sizes of all the words in the returns true ifthecodeblockreturns true for all items in the set.and exists returns true if the code block returns true foranyitemin theset.

Sometimes, you can generalize a feature using code blocks to make something more powerful. For example, you may want to sort in the traditionalway:

```
scala>words.sort((s,t)=>s.charAt(0).toLowerCase<t.charAt(0).toLowerCase)</pre>
res49:List[java.lang.String]=List(al,bud,kelly,peg)
```

This code uses a code block that takes two parameters, s and t.Using sort,³ you can compare the two arguments any way you want. In the previous code, we convert the characters to lowercase ⁴ and compare can also use the them. That will yield a case-insensitive search. We samemethodtosortthelistbythesizeofthewords:

```
scala>words.sort((s,t)=>s.size<t.size)</pre>
res50:List[java.lang.String]=List(al,bud,peg,kelly)
```

⁵ basedonanypolicythatwewant. Byusingacodeblock, we can sort Let'stakealookatamorecomplexexample, **tolet**

ttld.

The that method in Scala is much like the inject method in Ruby. Ydsupplyaninitialvalueandacodeblock. that willpasstothe

^{3.} Inversion 2.8.0, sort is deprecated. Use sort With instead.

^{4.} Inversion2.8.0, toLowerCase isdeprecated.Use toLower instead.

^{5.} Inversion 2.8.0, sort is deprecated. Use sort With instead.

code block each element of the array and another value. The second value is either the initial value (for the first invocation) or the result from the code block (for subsequent invocations). There are two versionsofthemethod. The first version, /:,isanoperatorwith initialValue /:codeBlock .Here'sthemethodinaction:

```
scala>vallist=List(1,2,3)
list:List[Int]=List(1,2,3)
scala>valsum=(0/:list){(sum,i)=>sum+i}
sum:Int=6
```

We walked through this sequence for Ruby, but it may help you to see itagain.Here'showitworks:

- We invoke the operator with a value and a code block. The code blocktakestwoarguments, sum and i.
- Initially, /: takestheinitial value, 0, and the first element of list. 1. andpassesthemintothecodeblock. sum is 0, i is 1, and the result. of 0+1 is 1.
- Next, /: takes 1, the result returned from the codeblock, and folds itbackintothecalculationas sum. So. sum is 1: i is the next element of list or 2:and the result of the code block is 3
- Finally, /: takes 3, the result returned from the code block, and foldsitbackintothecalculationas sum.So. sum is 3: i isthenext element of list.or 3:and sum+i is 6.

The syntax of the other version of will seem strange to you. that It uses a concept called currying. Functional languages use currying to transform a function with multiple parameters to several functions with their own parameter lists. We seemore currying in Chapter 8, Haskell, on page 268. Just understand that what's going on under the covers is a composition of functions rather than a single function.Though the mechanics and syntax are different, the result is exactly thesame:

```
scala>vallist=List(1,2,3)
list:List[Int]=List(1,2,3)
scala>list.foldLeft(0)((sum,value)=>sum+value)
res54:Int=6
```

has

Noticethatthefunctioncall list.foldLeft(0)((sum,vde)=>sum+vde) twoparameterlists.That'sthecurryingconceptthatImentionedear-lier.Ydseeversionsofthismethodwithalltherestofthelanguages inthisbook.

WhatWeLearnedinDay2

Day I was encumbered with working through the object-oriented features that you already know. Day 2 introduced Scala's primary reason for being: functional programming.

We started with a basic function. Scalabas flexible syntax with function definitions. The compiler can often infer the return type, the function body has one-line and code-block forms, and the parameter list can vary.

Next,welookedatvariouscollections. Scalasupports three: lists, maps, and sets. A set is a collection of objects. A list is a nordered collection. Finally, maps are key-value pairs. As with Ruby, you saw the powerful combinations of code blocks and collections of various kinds. We looked at some collection APIs that are indicative of functional programming paradigms.

Forlists, we could also use Lisp-style head and tail methods, just like Prolog, to return the first element of the list or the rest. We also used count, empty, and first methods for obvious purposes. But the most powerful methods took function blocks.

We iteratedwith feeth and used filter to selectively return various elements of the fitter to accumulate results as weiter at each of the fitter to accumulate results as weiter at each of the fitter to accumulate results as weiter at each of the fitter to selectively return various elements of the fitter to selectively elements of the fitter to selective e

Muchoffunctionalprogramming is learning to manipulate collections with higher-level constructs in stead of Java-style iteration. We will put these skills through their paces in day 3, when we will learn to use concurrency, do some XML, and work a simple practical example. Stay tuned.

Day2Self-Study

Now that we've gotten deeper into Scala, you're starting to see some of its functional aspects. Whenevery oude alwith functions, the collections are agreat place to start. These exercises will let you use some of the collections, as well as some functions.

Find:

- AdiscussiononhowtouseScalafiles
- Whatmakesaclosuredifferentfromacodeblock

Do:

- Use tolef tocompute the total size of a list of strings.
- Watea Censor traitwithamethodthatwillreplacethecursewords Shoot and Darn with Pucky and Beans alternatives. Useamapto storethecursewordsandtheiralternatives.
- Loadthecursewords and alternatives from a file.

5.4 Day3:CuttingThroughtheFluff

Justbeforetheclimaxof EdwardScissorhands .Edwardlearnstowield his scissors as an artist in everyday life. He molds shrubs into dinosaurs, crafts spectacular hair with the effort less skill of Val Sassoon, andevencarves the family roast. Wh Scala, we've encountered some awkward moments, but when this language feels right, it's borderline spectacular. Hard things, like XML and concurrency, become almost routine.Let'stakealook.

XML

Modern programming problems meet Extensible Markup Language (XML) with increasing regularity. Scalatakes the dramatic step of elevatingXMLtoafirst-classprogrammingconstructofthelanguage.Yu canexpressXMLjustaseasilyasyoudoanystring:

```
scala>valmovies=
```

```
|<movies>
           <moviegenre="action">PiratesoftheCaribbean</movie>
           <moviegenre="fairytale">EdwardScissorhands</movie>
     |</movies>
movies:scala.xml.Elem=
<movies>
           <moviegenre="action">PiratesoftheCaribbean</movie>
           <moviegenre="fairytale">EdwardScissorhands</movie>
       </movies>
```

Afteryou'vedefined the movies variable with XML, you can access differentelements directly.

For example, to see all the inner text, you would simply type this:

```
scala>movies.text
res1:String=
           PiratesoftheCaribbean
           EdwardScissorhands
```

Yuseealltheinnertextfromtheprevious example. Butwe'renotlimitedtoworkingwiththewholeblockatonce.We canbemoreselective. Scalabuildsinaquerylanguagethat'smuchlikeXPath,anXMLsearch language. But since the // keyword in Scalais a comment, Scala will use \ and \\.Tosearchthetop-levelnodes,you'duseonebackslash, likethis:

```
scala>valmovieNodes=movies\"movie"
movieNodes:scala.xml.NodeSeg=
 <moviegenre="action">PiratesoftheCaribbean
 <moviegenre="fairytale">EdwardScissorhands</movie>
```

Inthatsearch, we looked for XML movie elements. You canfindindividualnodesbyindex:

```
scala>movieNodes(0)
res3:scala.xml.Node=<moviegenre="action">PiratesoftheCaribbean</movie>
```

We justfoundelementnumberzero, or PiratesoftheCaribbean .Yucan also look for attributes of individual XML nodes by using the @ symbol. For example, to find the genre attributeofthefirstelementinthe document.we'ddothissearch:

```
scala>movieNodes(0)\"@genre"
res4:scala.xml.NodeSeg=action
```

This example just scratches the surface with what you can do, but yougettheidea.IfwemixinProlog-stylepatternmatching,thingsget a little more exciting. Next, we'll walk through an example of pattern matchingwithsimplestrings.

ReanMatching

Pattern matching lets you conditionally execute code based on some pieceofdata. Scalawillusepatternmatchingoften, suchaswhenyou parseXMLorpassmessagesbetweenthreads.

Here's the simplest form of pattern matching:

```
Download scala/chores.scala
defdoChore(chore:String):String=chorematch{
   case "cleandishes" => "scrub,dry"
    case "cookdinner" => "chop,sizzle"
    case_=> "whine,complain"
 println(doChore("cleandishes" ))
 println(doChore("mowlawn" ))
```

We define two chores. clean dishes and cook dinner. Next to each chore. we have a code block. In this case, the code blocks simply return _, a wildcard. Scala executes the strings. The last chore we define is codeblockassociated with the first matching chore, returning "whine, complain"ifneitherchorematches, likethis:

```
>>scalachores.scala
scrub, dry
whine,complain
```

Guards

Patternmatchinghassomeembellishmentstoo.InProlog,thepattern matchingoftenhadassociated conditions. To implement a factorial in Scala, we specify a condition in a guard for each math statement:

```
Download scala/factorial.scala
deffactorial(n:Int):Int=nmatch{
    case0 => 1
    casexifx>0=>factorial(n-1)
                                             * n
}
println(factorial(3))
println(factorial(0))
```

Thefirstpatternmatchisa 0,butthesecondguardhastheform casex ifx>0 .Itmatchesany x for x>0 .Yucanspecifyawidevarietyofconditionsinthisway.Patternmatchingcanalsomatchregularexpressions and types. Yol seean example later that defines empty classes and usesthemasmessagesinourconcurrencyexampleslater.

RegularExpressions

Scalahasfirst-classregular expressions. The r methodonastringcan translate any string to a regular expression. On the next page is an example of a regular expression that can match upper case or lower caseF atthebeginningofastring.

```
scala>valreg="""^(F|f)\w
reg:scala.util.matching.Regex=^(F|f)\w
scala>println(reg.findFirstIn("Fantastic"))
Some(Fantastic)
scala>println(req.findFirstIn("notFantastic"))
None
```

We startwithasimplestring. We usethe """ delimitedformofastring, allowing multilinestring and eliminating evaluation. The .r methodconvertsthestringtoaregularexpression.We thenusethemethod find-FirstIn tofindthefirstoccurrence.

```
scala>valreg="the".r
reg:scala.util.matching.Regex=the
scala>reg.findAllIn("thewaythescissorstrimthehairandtheshrubs")
res9:scala.util.matching.Regex.MatchIterator=non-emptyiterator
```

In this example, we build a regular expression and use the findAllIn methodtofindalloccurrencesoftheword the inthestring "thewaythe scissorstrimthehairandtheshrubs" .Ifwewanted,wecoulditeratethrough theentirelistofmatcheswith fæoch .That's really all the reistoit. You canmatchwithregular expressions just as you would use a string.

XMLwithMatching

AninterestingcombinationinScalaistheXMLsyntaxincombination withpatternmatching. Yucangothroughan XML file and conditionally execute code based on the various XML elements that come back.For example, consider the following XML movies file:

```
Download scala/movies.scala
valmovies=<movies>
    <movie>TheIncredibles</movie>
    <movie>WALLE</movie>
    <short>JackJackAttack</short >
    <short>Geri'sGame</ short>
</movies>
(movies\ "_").foreach{movie=>
   moviematch{
        case<movie>{movieName}</movie>=>println(movieName)
        case<short >{shortName}</short>=>println(shortName+
                                                                  "(short)")
    }
}
```

It queries for all nodes in the tree. Then, it uses pattern matching to match shorts and movies. I like the way Scala makes the most common tasks trivial by working in XML syntax, pattern matching, and theXQuery-likelanguage. The result is almost effortless.

So, that's abasic tour of pattern matching. Yelsee it in practice in the concurrency section next.

Concurrency

One of the most important aspects of Scalais the way it handles concurrency. The primary constructs are actors and message passing. Actorshavepoolsofthreadsandqueues. When you sendames sage to an actor(usingthe ! operator), you place an object on its queue. The actor reads the message and takes action. Often, the actor uses a pattern matchertodetectthemessageandperformtheappropriatemessage. Considerthe kids program:

```
Download scala/kids.scala
importscala.actors._
importscala.actors.Actor._
caseobjectPoke
caseobjectFeed
classKid()extendsActor{
 defact(){
    loop{
      react{
        casePoke=>{
          println("Ow...")
          println("Quitit..." )
        }
        caseFeed=>{
          println("Gurgle...")
          println("Burp...")
        }
      }
    }
 }
valbart=newKid().start
vallisa=newKid().start
println("Readytopokeandfeed..."
bart!Poke
lisa!Poke
bart!Feed
lisa!Feed
```

Inthisprogram, we create two empty, trivial singletons called Robe and Feed .Theydon'tdoanything.Theysimplyserveasmessages.Themeat Kid class. Kid isanactor, meaning it will run from oftheprogramisthe a pool of threads and get messages in a queue. It will process each messageandmoveontothenext.We startasimple loop. With that is a react construct, react receives an actor's messages. The pattern match lets us match the appropriate message, which will always be Robe or Feed .

Therestofthescriptcreatesacoupleofkidsandmanipulatesthemby sendingthem Rie or Feed messages. You can runitlike this:

```
batate$scalacode/scala/kids.scala
Readytopokeandfeed...
Ow...
Quitit...
Ow...
Quitit...
Gurgle...
Burp...
Gurgle...
Burp...
batate$scalacode/scala/kids.scala
Readytopokeandfeed...
Ow...
Ouitit...
Gurgle...
Burp...
Ow...
Quitit...
Gurgle...
Burp...
```

Iruntheapplicationacoupleoftimestoshowthatitisactuallyconcurrent. Notice that the order is different. Whactors, you can also reactwithatimeout(reactWithin), which will time outify oud on 'treceive themessagewithinthespecifiedtime. Additionally, you can use (whichblocksathread)and receiveWithin (whichblocksathreadwitha timeout).

ConcurrencyinAction

Since there's only a limited market for simulated Simpsons, let's do something a little more robust. In this application called computing the size of webpages. We hitafewpagesandthencompute the size. Since there's alot of waiting time, we would like to get all of

thepagesconcurrentlyusingactors. Take alookattheoverallprogram, andthenwe'lllookatsomeindividualsections:

```
Download scala/sizer.scala
importscala.io.
importscala.actors._
importActor._
objectPageLoader{
defgetPageSize(url:String)=Source.fromURL(url).mkString.length
}
valurls=List(
               "http://www.amazon.com/",
               "http://www.twitter.com/",
               "http://www.google.com/",
               "http://www.cnn.com/" )
deftimeMethod(method:()=>Unit)={
valstart=System.nanoTime
method()
valend=System.nanoTime
println("Methodtook"
                        +(end-start)/100000000.0+
                                                       "seconds." )
}
defgetPageSizeSequentially()={
for(url<-urls){</pre>
   println("Sizefor" +url+ ":" +PageLoader.getPageSize(url))
}
}
defgetPageSizeConcurrently()={
valcaller=self
for(url<-urls){</pre>
   actor{caller!(url,PageLoader.getPageSize(url))}
 }
 for(i<-1tourls.size){</pre>
   receive{
     case(url,size)=>
       println("Sizefor" +url+ ":" +size)
   }
}
}
println("Sequentialrun:" )
timeMethod{getPageSizeSequentially}
println("Concurrentrun" )
timeMethod{getPageSizeConcurrently}
```

So, let's start at the top. We doafewbasicimportstoloadthelibraries for actors and io sowecandoconcurrencyandHTTPrequests.Next,we willcomputethesizeofapage, givenaURL:

```
objectPageLoader{
defgetPageSize(url:String)=Source.fromURL(url).mkString.length
```

Next.wecreatea vd withafewURLs.Afterthat.webuildamethodto timeeachwebrequest:

```
deftimeMethod(method:()=>Unit)={
valstart=System.nanoTime
method()
valend=System.nanoTime
println("Methodtook"
                       +(end-start)/1000000000.0+
                                                        "seconds." )
}
```

Then, we do the web requests with two different methods. The first is sequentially, where we iterate through each requestina ta£och loop.

```
defgetPageSizeSequentially()={
for(url<-urls){</pre>
   println("Sizefor" +url+
                                     +PageLoader.getPageSize(url))
}
}
```

Here's the method to do things a synchronously:

```
defgetPageSizeConcurrently()={
 valcaller=self
 for(url<-urls){</pre>
   actor{caller!(url,PageLoader.getPageSize(url))}
 }
 for(i<-1tourls.size){</pre>
   receive{
     case(url,size)=>
       println("Sizefor" +url+
                                     ":" +size)
   }
 }
}
```

Inthisactor, weknowwe'llbereceiving a fixed set of messages. Writing a fixed set of messages. loop, we send four asynchronous requests. This happens more or less instantly. Next, we simply receive four messages with receive. This method is where the real work happens. Finally, we're ready to runthescriptthatinvokesthetest:

```
println("Sequentialrun:" )
timeMethod{getPageSizeSequentially}
```

```
println("Concurrentrun" )
timeMethod{getPageSizeConcurrently}
```

Andhere's the output:

```
>>scalasizer.scala
Sequentialrun:
Sizeforhttp://www.amazon.com/:81002
Sizeforhttp://www.twitter.com/:43640
Sizeforhttp://www.google.com/:8076
Sizeforhttp://www.cnn.com/:100739
Methodtook6.707612seconds.
Concurrentrun
Sizeforhttp://www.google.com/:8076
Sizeforhttp://www.cnn.com/:100739
Sizeforhttp://www.amazon.com/:84600
Sizeforhttp://www.twitter.com/:44158
Methodtook3.969936seconds.
```

As expected, the concurrent loop is faster. That's an overview of an interestingprobleminScala.Let'sreviewwhatwelearned.

WhatWeLearnedinDay3

Whatday3lackedinsize, it made up in intensity. We builtacouple of different concurrent programs and worked in direct XML processing, distributedmessagepassingwithactors,patternmatching,andregular expressions.

Overthecourseofthechapter, welearned four fundamental constructs that built on one another. First, we learned to use XML directly in Scala.We could query for individual elements or attributes using an XQuerylikesyntax.

We then introduced Scala's version of pattern matching. At first, it looked like a simple case statement, but as we introduced guards, types, and regular expressions, their power became readily apparent.

Next, we shifted to concurrency. We used the actor concept. Actors are objects built for concurrency. They usually have a loop statement wrappedarounda react or receive method, which does the dirty work of receiving queued messages to the object. Finally, we had an inner patternmatch. We usedrawclasses as messages. They are small, light, robust, and easy to manipulate. If we needed parameters within the message, we could just add naked attributes to our class definitions, aswedidwiththeURLwithinthesizerapplication.

Like all of the languages in this book, Scala is far more robust than you've seenhere. The interaction with Java classes is far deeper than I'veshownyouhere, and I'vemerely scratched the surface on complex conceptssuchascurrying. Butyouhaveagoodfoundationshouldyou choosetoexplorefurther.

Day3Self-Study

So, nowyou've seen some of the advanced features Scalahasto offer. Now, you can try to put Scalathrough its paces yourself. As always, these exercises are more demanding.

Find.

• Forthesizerprogram, what would happen if you did not create a newactorforeachlinkyouwantedtofollow?Whatwouldhappen totheperformanceoftheapplication?

Do:

- Take thesizerapplicationandaddamessagetocountthenumber oflinksonthepage.
- Bonus problem: Make the sizer follow the links on a given page, andloadthemaswell.Forexample,asizerfor "google.com" would compute the size for Google and all of the page sit links to.

5.5 WrappingUp\$cala

Wife covered Scala more exhaustively than the other languages so farbecauseScalastronglysupportstwoprogrammingparadigms.The object-oriented features firmly position Scala as a Java alternative. Unlike Ruby and Io, Scala has a static typing strategy. Syntactically, Scala borrows many elements from Java, including curly braces and constructorusage.

Scalaalsooffersstrongsupportforfunctionalconceptsandimmutable variables. The language has a strong focus on concurrency and XML, fittingawidevariety of enterprise applications currently implemented intheJavalanguage.

Scala's functional capabilities gobeyond what I've covered in this chapter. I haven't covered constructs such as currying, full closures, multiple parameter lists, or exception processing, but they are all worthy concepts that add to the power and flexibility of Scala.

CoreStrengths

Most of Scala's strengths are centered around an advanced programming paradigm that integrates well with the Java environment and somecorewell-designedfeatures. In particular, actors, pattern matching,andtheXMLintegrationareimportantandwell-designed.Let'sget righttothelist.

Concurrency

Scala's treatment of concurrency represents a significant advance in concurrent programming. The actor model and the thread pool are welcomeimprovements, and the ability to design applications without mutablestateisabsolutelyhuge.

The actor paradigm that you've seen in Io and now Scala is easy to understand for developers and well studied by the academic community.BothJavaandRubycouldusesomeimprovementinthisarea.

The concurrency model is only part of the story. When objects share state, you must strive for immutable values. Io and Scalaget this at least partially right, allowing mutable state but also offering libraries and keywords that support immutability. Immutability is the single most important thing you can do to improve code design for concurrency.

Finally, themessage-passing syntaxyouseein Scalaismuchlikeyou will see in the next chapter on Erlang. It is a significant improvement overthestandardJavathreadinglibraries.

Evolution of Legacy Jaca

Scala starts with a strong, built-in user base: the Java community. ScalaapplicationscanuseJavalibrariesdirectly, and through the use ofcodegenerationofproxyobjectswhennecessary, the interoperability isexcellent.Inferredtypingisamuch-neededadvanceoverthearchaic Javatypingsystem. The bestway to establish a new programming communityistofullyembraceanexistingone. ScaladoesagoodjobofferingamoreconciseJava, and that idea has value.

Scala also offers new features to the Java community. Code blocks area first-class language construct, and they work well with the core collection libraries. Scala also offers first-class mixins in the form of traits. Patternmatchingisalsoasignificantimprovement. Whithese

Throw in the functional constructs, and you can have significantly improved applications. Scala applications will usually have a fraction of the total lines of code than an equivalent Java app would have, andthat's extremely important. Abetter programming languages hould allowy out o express more complex ideas with fewer lines of code, withminimaloverhead. Scaladeliverson this promise.

Domain-SpecificLanguages

Scala's flexible syntax and operator overloading make it an ideal language for developing Ruby-style domain-specific languages. Remember, as in Ruby, operators are simply method declarations, and you can override them in most cases. Additionally, optional spaces, periods, and semicolons let the syntax takemany different forms. Together withrobustmixins, these are the tools that a DSL developerseeks.

XML

ScalahasintegratedXMLsupport.ThepatternmatchingmakesparsingblocksofdisparateXMLstructureseasytouse. The integration of theXPathsyntaxfordivingdeepintocomplexXMLleadstosimpleand readable code. This advance is welcome and important, especially to theXML-heavyJavacommunity.

Bridging

The emergence of each new programming paradigm needs a bridge. Scalaiswell-positionedtobethatbridge. The functional programming modelisimportantbecauseithandlesconcurrencywell, and evolving processordesignsaremuchmoreconcurrent. Scala offersaniterative waytotakedevelopersthere.

Wedkresses

Although I like many of the Scalaideas in concept, I find the syntax demandingandacademic. Although syntaxis a matter of taste, Scala doeshaveahigherburdenthanmostotherlanguages, atleast to these oldeyes.IalsorecognizethatsomeofthecompromisesthatmakeScala such an effective bridge also will undercut its value. I only see three weaknesses, butthey are bigones.

Static typing is a natural fit for functional programming languages, but Java-style static typing for object-oriented systems is a deal with the devil. Sometimes, you must satisfy a compiler's requirements by putting more of a burden on your developers. Wh static typing, the burdenismuchmorethanyoueverexpect. The impacton code, syntax, andprogramdesignareprofound. As Ilearned Scala, I found myselfin afairlyconstantbattlewiththelanguagesyntaxandinprogramdesign. Traits easedthisburdensomewhat, but I found the trade-off between programmerflexibilityandcompile-timecheckingwanting.

Laterinthisbook, you'll see what apurely functional strong, static type systemlookslikewithHaskell.Winttheburdenoftwoprogramming paradigms, the type system becomes much more fluid and productive, providingbettersupportforpolymorphismandrequiringlessrigorfrom programmersforsimilarbenefit.

Syntax

Idofind Scala's syntax to be a little a cademic and hard on the eyes. Ihesitatetoputthisinprintbecausesyntaxcanbesosubjective, but someelements are a little baffling. Sometimes, Scalakeeps Java conventions, such as constructors. Yaid use new Reson ratherthan Re son, new. At other times. Scala introduces a new convention, as with argumenttypes.InJava,you'duse setName(Stringname) versusScala's setName(name:String).Returntypesshifttotheendofthemethoddeclarationversusthebeginning, as they are with Java. The selittle differences keep me thinking about syntax rather than code. The problem is that moving back and for the tween Scala and Java will take more effortthanitshould.

Mutability

When you build a bridge language, you must factor in compromises. One significant compromise in Scalais the introduction of mutability. Why var ,ScalaopensPandora'sboxinsomeways,becausethemutable stateallowsawidevarietyofconcurrencybugs.Butsuchcompromises areunavoidableifyouwanttobringhomethespecialkidwholivedin thehouseonthehill.

FinalThoughts

Allinall, my experience with Scalawasmixed. The static typing threw me. At the same time, the Javadeveloper in megreatly appreciates the improved concurrency models, inferred typing, and XML. Scalar epre-improved concurrency models are approximated to the properties of the propesents a significant jump in the state of the art.

I would use Scala to improve my productivity if I had a significant investment in Java programs or programmers. I'd also consider Scalafor an application that has significant scalability requirements that would require concurrency. Commercially, this Frankenstein has a goodshotbecauseitrepresentsabridgeandfullyembraces a significantprogramming community.

▶ AgentSmith

Chapter 6 Erlang

FewlanguageshavethemystiqueofErlang, the concurrency language thatmakeshardthingseasyandeasythingshard. Its virtual machine, called BEAM, is rivaled only by the Java virtual machine for robust enterprisedeployment. Yucould callitefficient, even brutally so, but Erlang's syntaxlacks the beauty and simplicity of, say, a Ruby. Think AgentSmithof TheMatrix . 1

The Matrix was a 1999 science-fiction classic that painted our current world as a virtual world, created and maintained with computers, as an illusion. Agent Smith was an artificial intelligence program in the matrixthathadanamazingabilitytotakeanyformandbendtherules ofrealitytobeinmanyplacesatonce. Hewasunavoidable.

6.1 IntroducingErlang

The name is strange, but the acronym for Ericsson Language that sharesanamewithaDanishmathematiciansomehowfits.AgnerKarup Erlang was a huge name in the math behind telephone network analysis.

In 1986, Joe Armstrong developed the first version at Ericsson, continuing to develop and polish it through the last half of the decade. Through the 1990s, it grew in fits and starts and gained still more traction in the 2000s. It is the language behind CouchDB and SimpleDB,populardatabasesforcloud-basedcomputing.ErlangalsopowersFacebook'schat.ThebuzzforErlangisgrowingsteadilybecauseit

^{1.} The Matrix . DVD. Directed by Andy Walneski CA: Warrer Home Video 2007.

provides what many other languages can't: scalable concurrency and reliability.

Built&Concurrency

ErlangisaproductofyearsofresearchfromEricssontodevelopnearreal-time fault-tolerant distributed applications for telecom applications. The systems often could not be taken down for maintenance, and software development was prohibitively expensive. Erics son studiedprogramminglanguagesthroughthe1980sandfoundthat,forone reasonoranother, existing languages were in adequate for their needs. These requirements eventually led to the development of an entirely newlanguage.

Erlang is a functional language—one with many reliability features cooked in. Erlang can support insanely reliable systems. You can't takeaphoneswitchdownformaintenance,andyoudon'thavetotake Erlangdown to replace entire modules. Some of its applications have runforyears without ever coming down formaintenance. But the key Erlangeapabilityisconcurrency.

Concurrency experts do not always agree on the best approaches. One common debate is whether threads or processes lead to better concurrency. Manythreads make upaprocess. Processes have their own resources; threads have their own execution path but share resources with other threads in the same process. Usually, a thread is lighter weightthanaprocess, thoughimplementations vary.

NoThreading

Manylanguages, like Java and C, take a threading approach to concurrency. Threads take fewer resources, so theoretically, you should beabletogetbetterperformancefromthem. The downside to threads is that shared resources can lead to complex, buggy implementations and the need for locks that form bottlenecks. To coordinate control betweentwoapplicationssharingresources, threading systems require semaphores, or operating system level locks. Erlang takes a different approach. Ittriestomake processes as lightweight as possible.

LightweightProcesses

Rather than wade through the quagmire of shared resources and resourcebottlenecks, Erlangembracesthephilosophyoflightweightprocesses. Erlang's creators spent effort to simplify the creation, management, and communication within applications with many processes. Distributedmessagepassingisabasiclanguage-levelconstruct, eliminatingtheneedforlockingandimprovingconcurrency.

LikeIo, Armstrong's creation uses actors for concurrency, somessage passingisacritical concept. YM recognizeScala'smessagepassing syntax, which is similar to Erlang's message passing. In Scala, an actor representsanobject, backed by a thread pool. In Erlang, an actor representsalightweightprocess. The actor reads in bound messages from aqueueandusespatternmatchingtodecidehowtoprocessit.

Reliability

Erlangdoeshavetraditionalerrorchecking, butinatraditional application, you'll see far less error handling than you would in a traditional fault-tolerant application. The Erlang mantra is "Let it crash." Since Erlangmakesiteasytomonitorthedeathofaprocess, killingrelated processes and starting new one saretrivial exercises.

Yu can also hot-swap code, meaning you can replace pieces of your application without stopping your code. This capability allows far simplermaintenance strategies than similar distributed applications. Erlangcombinestherobust"Letitcrash"errorstrategieswithhot-swapping and lightweight processes that you can start with minimal overhead. It's easy to see how some applications run for years at a time withoutdowntime.

So, the Erlang concurrency story is compelling. The important primitives—messagepassing,spawningaprocess,monitoringaprocess—are all there. The processes that you spawn are lightweight, so you don't havetoworryaboutconstrainedresourcesinthisarea. The language isheavilyslantedtoremovesideeffectsandmutability, and monitoring thedeathofaprocessissimple, eventrivial. The combined package is compelling.

InterviewwithDr.JoeArmstrong

Throughwritingthisbook, I'vehadthechancetomeetsomeofthepeoplewholrespectthemost, at least, through email. Dr. Joe Armstrong, creatorofErlangandauthorof ProgrammingErlang:SoftwareforaConcurrent Watl [Arm07], is high on that list forme. I finally got to have several conversations with Erlang's first implementor, who hails from Stockholm.Sweden.

Dr. Armstrong: Byaccident.Ididn'tsetouttoinventanewprogramminglanguage. Atthetime we wanted to find a better way of writing the controlsoftwareforatelephoneexchange. Istarted fiddling around with Prolog. Prolog was fantastic but didn't do exactly what I wanted. So, I started messing around with Prolog. I thought, "I wonder what would happen if I changed the way Prolog does things?" So, I wrote a Prolog meta-interpreter that added parallel processes to Prolog, and then Iadded error handling mechanisms, and so on. After a while, this set ofchangestoPrologacquiredaname,Erlang,andanewlanguagewas born. Then more people joined the project, the language grew, we figured outhowtocompileit, weadded more stuff and got some more users, and

Bruce: Whatdoyoulikeaboutithemost?

Dr. Armstrong: Theerrorhandlingandon-the-flycodeupgrademechanisms and the bit-level pattern matching. Error handling is one of the least under stood parts of the language and the part where it most differsfrom other languages. The whole notion of "nondefensive" programming and "LetItCrash," which is the mantra of Erlang programming, is completely the opposite of conventional practice, but it leads to **rd**y short andbeautifulprograms.

Bruce: Whatisafeatureyoumostwouldwanttochange,ifyoucould doitalloveragain?(Alternatively, you could answer, what are the greatestlimitationsofErlang?)

Dr. Armstrong: This is a difficult question; I'd probably give different answersondifferentdaysoftheweek. It would be nice to add mobility to thelanguagesowecouldsendcomputationsovertheNet.Wecandothis inlibrarycode, butit's not supported in the language. Right now Ithinkit wouldberd inicetogoback to the roots of Erlangandadd Prolog-like predicatelogictothelanguage, akindofnew mixture of predicatelogic plusmessagepassing.

Then there are a number of small changes that are desirable, adding hashmaps, higher-ordermodules, and soon.

If Iwe reto do it allower again, I'd probably give a lot more thought tohow we fit things together, such as how we run big projects with lots of code—how we manage versions of code, how we find things, how thingsevolve. When lots of code has been written, the programmer's job changes from writing fresh code to finding and integrating existing code, sofindingthingsandfittingthingstogetherbecomesincreasinglyimportant. It would be nice to integrate ideas from things like GIT and Mercurialand type systems into the language itselfs othat we could understand howcodeevolvesinacontrolledmanner.

Bruce: What's the most surprising place you've ever seen Erlangused inproduction?

Dr. Armstrong: Walwasn'tactuallysurprisedsinceIknewthiswas goingtohappen.WhenlupgradedmyversionofUbuntutoKarmicKoala, Ifoundaratherwell-hiddenErlangtickingawayinthebackground.This wasto support Couch DB, which was also running live on my machine. ThiskindofsneakedinErlangundertheradarto10millionmachines.

In this chapter, we're going to cover some Erlang basics. Then, we'll put Erlang through its paces as a functional language. Finally, we'll spendsometime with concurrency and some cool reliability features.Ysfriends, reliability can be cool.

6.2 Day 1: Appearing Human

Agent Smith is as a program that kills other programs, or simulated people, that disrupt the simulated reality known as the Matrix. The most basic trait that makes him dangerous is his ability to appear human. In this section, we're going to look at Erlang's ability to build general-purposeapplications. I'mgoingtotrymybesttogiveyou"normal."It'snotgoingtobeeasy.

Ifyoustartedthisbookasapureobject-orientedprogrammer, youmay strugglealittlebit, butdon't fight it. Ywe alreadyseencodeblocks inRuby,actorsinIo,patternmatchinginProlog,anddistributedmessagepassinginScala. These are found at ional principles in Erlang. This chapterwillstartwithanotherimportantconcept. Erlangisthefirstof ourfunctionallanguages. (Scalaisahybridfunctional/object-oriented language.)Toyou,thatmeansthefollowing:

- Yrprogramsaregoing to be built entirely out of functions, with noobjectsanywhere.
- Those functions will usually return the same values, given the sameinputs.

• Yuwillonlybeabletoassignanyvariableonce.

Living by the first rule is mildly challenging. Living by the next three canknockyoudown, at least for a little while. Know that you can learn tocodethisway, and the result will be programs that are built from the insideoutforconcurrency. When you remove mutable state from the equation, concurrency gets dramatically simpler.

Ifyoupaidcloseattention, you caught the word usually inthesecond andthirdrules. Erlangis not apure functional language; it does allow afewexceptions. Haskellistheonly purefunctional language in this book.Butyouwillgetastrongflavoroffunctional-styleprogramming, andyouwill code to the serules more often than not.

GettingStarted

I'm working with Erlang version R13B02, but the basic stuff in this chapter should work OK on any reasonable version. Yil get to the Erlangshellbytyping erl (wet on some Windows systems) at the commandline.likethis:

```
batate$er1
Erlang(BEAM)emulatorversion5.4.13[source]
EshellV5.4.13 (abortwith G)
1>
```

4W do most of our work there, early on, as with other chapters. Like Java, Erlang is a compiled language. Yil compile a file with c(filename). (youneedtheperiodattheend). Yucanbreakoutofthe console, or aloop, with Control+C. Let's getstarted.

Comments, Variables, and Expressions

Let's getsome of the basic syntax out of the way. Crack open the console, and type the following:

```
1>%Thisisacomment
```

Thatwassimpleenough.Commentsstartwitha % andeateverything untiltheendofaline. Erlangparses a comment as a single space.

```
1>2+2.
2>2+2.0.
4.0
```

```
3>"string".
"string"
```

Each statement ends in a period. These are some of the basic types: strings, integers, and floats. Now, for alist:

```
4>[1,2,3].
[1,2,3]
```

As with the Prolog family of languages, lists are in square brackets. Here'salittlesurprise:

```
4>[72,97,32,72,97,32,72,97].
"HaHaHa"
```

So,a String isreallya List, and Agent Smithjustlaughed at your mamma. 2+2.0 tellsusthatErlangdoessomebasictype Oh, those social skills. coercion.Let'strytobreakalineofcodewithabadtype:

```
5>4+"string".
** exceptionerror:badargumentinanarithmeticexpression
inoperator +/2
    calledas4+"string"
```

UnlikeScala, there's no coercion between strings and ints. Let's assign avariable:

```
6>variable=4.
** exceptionerror:nomatchofrighthandsidevalue4
```

Ah. Here, you see the ugly side of the comparison between agents and Erlang.Sometimes,thispeskylanguagehasmorebrainthansoul.This error message is really a reference to Erlang's pattern matching. It's breaking because votte is an atom. Vails must start with an uppercaseletter.

```
7>Var=1.
8>Var=2.
=ERRORREPORT====8-Jan-2010::11:47:46===
Errorinprocess<0.39.0>withexitvalue:{{badmatch,2},[{erl_eval,expr,3}]}
** exited:{{badmatch,2},[{erl_eval,expr,3}]} **
8>Var.
```

As you can see, variables begin with a capital letter, and they are can assign each value only once. This concept gives immutable. **Y**u

mostfirst-timeprogrammerstroublewithinafunctionallanguage.Let's introducesomedatatypeswithmorecomplexity.

Atoms, Lists, and Tipes

Infunctionallanguages, symbols become more important. They are the most primitive data element and can represent anything you want to encountered symbols in each of the other programming name. Yorke languagesinthisbook.InErlang,asymboliscalledan atom and begins withalowercasecharacter. They are atomic values that you can use to representsomething. Yolusethemlikethis:

```
9>red.
red
10>Pill=blue.
blue.
11>Pill.
blue
```

red and blue areatoms—arbitrarynamesthatwecanusetosymbolize real-world things. We first return a simple atom called red. Next, we blue to the variable called assign the atom called Pill. Atoms get more interesting as you attach them to more robust data structures that we'll see a little later. For now, let's build on the primitives by looking atthelist. Yolrepresentlists with square brackets:

```
13>[1,2,3].
[1,2,3]
14>[1,2,"three"].
[1,2,"three"]
15 > \text{List} = [1, 2, 3].
[1,2,3]
```

So, the list syntaxis familiar. List sare heterogeneous and can be any length. Yucanassign them to variables, just as you would aprimitive. Tuplesarefixed-lengthheterogeneouslists:

```
18>{one, two, three}.
{one, two, three}
19>0rigin=\{0,0\}.
\{0,0\}
```

There are no surprises here. You can see the strong Prologin fluence here.Later,whenwecoverpatternmatching,youwillnoticethatwhen youmatchatuple, the size will matter. Yucan't matchathree-tuple to a two-tuple. When you match a list, the length can vary, just as it didinProlog.

InRuby, you use hashmap sto associate names with values. In Erlang, you'lloftenseetuplesusedasyouwouldusemapsorhashes:

```
20>{name, "SpacemanSpiff"}.
{name, "SpacemanSpiff"}
21>{comic_strip,{name,"CalvinandHobbes"},{character,"SpacemanSpiff"}}.
{comic_strip, {name, "CalvinandHobbes"},
         {character, "SpacemanSpiff"}}
```

Wate represented a hash for a comic strip. We useatomsforthehash keys and use strings for the values. You can mix lists and tuples as well, such as a list of comics, represented by tuples. So, how do you access the individual elements? If Prologisfreshinyour mind, you're alreadythinkingintherightdirection. Yulusepatternmatching.

ReanMatching

Ifyouworkedthroughthe Prolog chapter, you got a pretty solid foundation of pattern matching. I want to point out one major difference. In Prolog, when you defined a rule, you matched all the values in thedatabase, and Prologworked through all the combinations. Erlang works like Scala. A match will work against a single value. Let's use pattern matching to extract the values from a tuple. Say we have a person:

```
24>Person={person, {name, "AgentSmith"}, {profession, "Killingprograms"}}.
                 {person, {name, "AgentSmith"},
                         {profession, "Killingprograms"}}
```

Let's say we want to assign the name to Name, and the profession to Profession. This match would do the trick:

```
25>{person, {name, Name}, {profession, Profession}}=Person.
{person, {name, "AgentSmith"},
        {profession, "Killingprograms"}}
26>Name.
"AgentSmith"
27>Profession.
"Killingprograms"
```

Erlang will match up the data structures, assigning variables to the valuesinthetuples. Anatom will matchitself, so the only work to be done is to match the variable Name to "Agent Smith" and the variable Profession to "Killing programs". This feature works much like it does in Prologandwillbethefundamentaldecision-makingconstructthatyou use.

Ifyou are used to Ruby or Java-style hashes, it may seem strange to have the initial atom of person. In Erlang, you'll often have multiple matching statements and multiple kinds of tuples. By designing your datastructuresthisway, you can quickly matchall person tuples, leavingtheothersbehind.

ListpatternmatchingissimilartoProlog's:

```
28>[Head|Tail]=[1,2,3].
[1,2,3]
29>Head.
30>Tail.
[2,3]
```

Easyasone, two, three. Yucanbindtomore than one variable at the headofalist,too:

```
32>[One,Two|Rest]=[1,2,3].
[1,2,3]
33>One.
1
34 > Two.
35>Rest.
[3]
```

If there are not enough elements in the list, the pattern won't match:

```
36>[X|Rest]=[].
** exceptionerror:nomatchofrighthandsidevalue[]
```

Now, some of the other error messages make a little more sense. Let's sayyouforgettostartyourvariableswithanuppercaseletter. Yilget thiserrormessage:

```
31>one=1.
** exceptionerror:nomatchofrighthandsidevalue1
```

Asyou'veseenbefore,the = statementisnotasimpleassignment.Itis actuallyapatternmatch. Yie asking Erlangtomatch the integer withtheatom one.anditcan't.

BitMatching

Sometimes, youneed to access data at the bit level. If you're cramming more data into less space or dealing with predefined formats such as JPEGsorMPEGs, the location of each bit matters. Erlanglet syoup ack severalpiecesofdataintoonebytequiteeasily. Todothesetwothings,

thiscopyis (P1.0printing,October2010)

1

youneedtwooperations:packandunpack.InErlang,abitmapworks justlikeothertypesofcollections. Topackadatastructure, you'lljust tellErlanghowmanybitstotakeforeachitem,likethis:

```
1>W=1.
1
2>X=2.
3>
3>Y=3.
4 > 7 = 4.
5>A11=<<W:3,X:3,Y:5,Z:5>>.
<<"(d">>>
```

The << and >> bracketbinarypatternsinthisconstructor.Inthiscase, itmeanstake3bitsforthevariable W,3bitsfor X,5bitsfor Y, and 5 bitsfor Z.Next, we need to be able to unpack. Yucan probably guess thesyntax:

```
6><<A:3,B:3,C:5,D:5>>=A11.
<<"(d">>>
7>A
7>.
8>D.
```

Justliketuplesandlists, we just supply the same syntax and let patternmatchingdotherest. Winthese bitwise operations, Erlangissurprisinglypowerfulforlow-leveltasks.

Wate covering a lot of ground pretty quickly, because you've already been introduced to all the major concepts in the chapter. Believe it or not, we're almost through the first day of the Erlang chapter, but we firstneedtointroducethemostimportantconcept, the function.

Functions

UnlikeScala, Erlangisdynamicallytyped. Yuwon'thavetoworrytoo muchaboutassigningtypestodataelements.LikeRuby,Erlangtyping isdynamic. Erlangwillbindtypesatruntime, basedonsyntacticclues suchasquotesordecimalpoints. Atthispoint, I'mgoingtocrackopen afreshcopyoftheconsole.Letmeintroduceafewterms. Yiegoing towritefunctionsinafilewithan .erl extension.Thefilecontainscode foramodule, and you have to compile it to runit. Afteryou've compiled

Whithe house keeping out of the way, it's time to create some basic functions.

I'mgoingtoenterafilethatlookslikethis:

```
Download erlang/basic.erl
-module(basic).
-export([mirror/1]).
mirror(Anything)->Anything.
```

The first line defines the name of the module. The second line defines a function that you want to use outside of the module. The function is called mirror, and the /l means it has one parameter. Finally, you gettothefunctionitself. Yucanseetheinfluenceofthe Prolog-style rule. The function definition names the function and determines the arguments. Afterward, you have the -> symbol, which simply returns thefirstargument.

Wh a function definition complete, I'll fire up the console from the samedirectorythathasthecodefile.Icanthencompileitlikethis:

```
4>c(basic).
{ok,basic}
```

basic.beam file in the same We compiled basic.erl, and you will find a directory. Yucanrunitlikethis:

```
5>mirror(smiling_mug).
** exceptionerror:undefinedshellcommandmirror/1
6>basic:mirror(smiling mug).
smiling_mug
6>basic:mirror(1).
```

Noticethatitisnotenoughtohavethefunctionnamealone. Yualso needtoincludethemodulename, followed by a colon. This function is deadsimple.

Noticeonething. We wereabletobind Anything totwodifferenttypes. Erlang is dynamically typed, and to me, it feels good. After Scala's strongtyping, I'm coming home from a weekend in Siberia, or at least Peoria.

Let's look at a function that's slightly more complicated. This one definesseveralmatchingalternatives.

```
filelikethis:
Yucancreatea
                      matching fundament
```

```
Download erlang/matching_function.erl
-module(matching_function).
-export([number/1]).
number(one)
              ->1;
number(two)
              ->2:
number(three)->3.
Andyoucanexecuteitlikethis:
8>c(matching_function).
{ok,matching_function}
9>matching_function:number(one).
10>matching_function:number(two).
11>matching_function:number(three).
12>matching_function:number(four).
** exceptionerror:nofunctionclausematchingmatching_function:number(four)
```

This is the first function I've introduced with multiple matching possibilities. Each possible match has the function name, the argument to match, and the code to execute after the -> symbol. In each case, Erlangjustreturnsaninteger. Terminatethelaststatementwith . and allotherswith :.

Justaswith Io, Scala, and Prolog, recursion will play a bigrole. Like Prolog, Erlang is optimized for tail recursion. Here is the obligatory factorial:

```
Download erlang/yet_again.erl
-module(yet_again).
-export([another_factorial/1]).
-export([another_fib/1]).
another factorial(0)->1;
another_factorial(N)->N * another_factorial(N-1).
another_fib(0)->1;
another_fib(1)->1;
another_fib(N)->another_fib(N-1)+another_fib(N-2).
```

So, it's another factorial, and it's defined recursively just like all the others. While I was a tit, I may a swell include a Fibonacci series, too.

Letmetrytomakeitworthyourwhilethistime:

```
18>c(yet_again).
{ok,yet_again}
19>yet_again:another_factorial(3).
20>yet_again:another_factorial(20).
2432902008176640000
21>yet_again:another_factorial(200).
788657867364790503552363213932185062295135977687173263294742533244359
449963403342920304284011984623904177212138919638830257642790242637105
061926624952829931113462857270763317237396988943922445621451664240254
033291864131227428294853277524242407573903240321257405579568660226031
904170324062351700858796178922222789623703897374720000000000000000000
22>yet again:another factorial(2000).
3316275092450633241175393380576324038281117208105780394571935437060380
7790560082240027323085973259225540235294122583410925808481741529379613
1386633526343688905634058556163940605117252571870647856393544045405243
9574670376741087229704346841583437524315808775336451274879954368592474
...andonandon...
```

Ooooh-kaaay. That was certainly different. Now, you're starting to see the butt-kicking side of the Agent Smith/Erlang comparison. If you didn'ttakethetimetorunit,letmeassureyou,theresultsareabsolutelyinstantaneous. Idon'tknowwhat the maximum integer size is, butI'mgoingtogooutonalimbandsayit'sbigenoughforme.

That's a pretty good starting point. Yore created some simple functionsandseenthemwork.It's agood time towrapupday 1.

What We Learned in Day 1

Erlangisafunctionallanguage. Itisstrongly, dynamically typed. There is not a lot of syntax, but what is there is not at all like the typical object-orientedlanguages.

Like Prolog, Erlanghas no notion of an object. However, Erlangdoes have a strong connection to Prolog. The pattern matching constructs and multiple function entry points should look familiar to you, and you handle some problems in the same way, through recursion. The functionallanguagehasnonotionofmutablestateorevensideeffects. Maintaining program state is awkward, but you will learn a new bag of tricks. Yol soon see the other side of the coin. Eliminating state and side effects will have a dramatic impact on how you will manage concurrency.

Inthefirstday, youworked both in the console and with the compiler. Primarily, you focused on the basics. You created basic expressions. You also created simple functions. Like Prolog, Erlanglets a function havemultipleentrypoints. You sedbasic patternmatching.

also used basic tuples and lists. Tuples took the place of Ruby hashes and formed the foundation of data structures. You learned to pattern match across lists and tuples. These ideas will allow you to quickly attach behavior to tuples or interprocess messages in laterchapters.

Inday2,I'mgoingtoexpandthebasicfunctionalconcepts.We learn how to build code that will work in a concurrent world, but we won't actuallygothereyet. Take alittletimetodosomeself-studytopractice whatyou'velearnedsofar.

Day 1 Self-Study

The online community for Erlangis growing rapidly. A conference in San Francisco is picking up momentum. And unlike Io and C, you shouldbeabletouseGoogletofindwhatyouneed.

Find:

- The Erlanglanguage's official site
- OfficialdocumentationforErlang'sfunctionlibrary
- ThedocumentationforErlang'sOTPlibrary

Do:

- We a function that uses recursion to return the number of wordsinastring.
- Weafunction that uses recursion to count to ten.
- Warfunction that uses matching to selectively print "success" or "error: message" given input of the form {error, Message} or success.

6.3 Day2:ChangingForms

Inthissection, you'regoing to be ginto appreciate Agent Smith's power. Theagentsin The Matrix have super-human strength. They can dodge bullets and punch through concrete. Functional languages are at a higherlevelofabstractionthanobject-orientedlanguages. Thoughthey

aremoredifficulttounderstand, you can express biggeride as withless code.

AgentSmithcanalsotaketheformofanyotherpersoninthematrix. That's an important capability in a functional language. Yie going tolearntoapplyfunctionstoliststhatcanquicklyshapethelistinto exactlywhatyouneed.Doyouwanttoturnashoppinglistintoalist of prices? What about turning a list of URLs into tuples containing contentand URLs? These are the problems that functional languages simplydevour.

ControlStructures

Let's start with a few mundane pieces of Erlang: basic control structures. Yal notice that this section of the book is much shorter than Scala's. Often, you'll seeprograms with plenty of case statements.becausetheywillinterpretwhichmessagetoprocesswhenyou'rewriting concurrentapplications. if sareless prevalent.

Case

Let's start with a case. Most of the time, you think of a pattern matchin the context of function invocation. Think of this control structure as a patternmatchthatyoucanuseanywhere.Forexample,sayyouhavea variable called Animal. Yuwantto conditionally execute the code based onthevalue:

```
1>Animal="dog".
2>caseAnimalof
2>
      "dog"->underdog;
      "cat"->thundercat
2>end.
underdoa
```

So, in this example, the string matched the first clause and returned the atom underdog. As with Prolog, you can use the underscore () to matchanything, like this (note: Animal isstill "dog"):

```
3>caseAnimalof
       "elephant"->dumbo;
3>
       _->something_else
3>end.
something else
```

Theanimalwasnot "elephant", soit matched the last clause. Yu can also use under scores in any other Erlang match. I'd like to point out a basic syntactic wart here. Notice that all case clauses but the last endina semicolon. That means if you want to edit your statement to

reorderyourclauses, you must adjust the syntax accordingly, though itwouldhavebeenprettyeasytoallowanoptionalsemicolonafterthe lastclause.Sure,thesyntaxislogical:thesemicolonisaseparatorfor the case clauses. It's just not very convenient. Agent Smithjust kicked sandallovermykidnephew, and I think I heard him laugh. He has tobrushuponthosepublic relations if hewants towin Agent of the Month.Let'smoveontothebasic if.

If

The cose statementuses pattern matching, and the if statementuses guards. In Erlang, a guardis a condition that must be satisfied for a match to succeed. Later, we'll introduce guards on pattern matches, butthemostbasicformofaguardisinan if statement. Yustartwith the if keywordandthenfollowitwithseveral guard->expression clauses. Here'stheidea:

```
if
 ProgramsTerminated>0->
    success;
 ProgramsTerminated<0->
    error
end.
```

Whathappensifthereisnomatch?

```
8>X=0.
9>if
    X>0->positive;
     X<0->negative
** exceptionerror:notruebranchfoundwhenevaluatinganifexpression
```

Unlike Ruby or Io, one of the statements must be true, because if is afunction. Each case must return a value. If you truly want an else, makethelastguard true, likethis:

```
9>if
9>
     X>0->positive;
     X<0->negative;
9>
     true ->zero
9>end.
```

That's really it for control structures. Yie going to get much more outofhigher-orderfunctionsandpatternmatchingtoaccomplishyour goals, solet's leave the secont rolst a tement she hind and dive in deeper into functional programming. We going to work with higher-order functions and use them to process lists. We learn to solve progressively more complex problems with functions.

AnonymousFunctions

As you recall, higher-order functions either return functions or take functionsasarguments.Rubyusedcodeblocksforhigher-orderfunctions, with special attention to passing code blocks to iterate overlists.In Erlang, you can assign arbitrary functions to variables and pass themaroundlikeanyotherdatatypes.

Ynieseensomeoftheseconceptsbefore, butwe'regoing to lay some ofthefoundationsinErlangandthenbuildsomehigher-levelabstractions. Itallstarts with an onymous functions. Here's how you'd assign afunctiontoavariable:

```
16>Negate=fun(I)->-Iend.
#Fun<erl_eval.6.13229925>
17>Negate(1).
-1
18>Negate(-1).
```

Line16usesanewkeywordcalled fun. Thatkeyworddefines an an onymousfunction.Inthiscase,thefunctiontakesasingleargumentcalled I and returns the negation, -I. We assign that an onymous function to Negate isnotthevaluereturnedbythefunction.It Negate.Tobeclear, actually is thefunction.

Twosignificantideasarehappeninghere. First, we'reassigning afunctiontoavariable. This conceptallows us to passaround behaviors just aswewould any other data. Second, we can easily invoke the underlyingfunction, just by specifying an argument list. Notice the dynamic typing. We don't have to concern ourselves with the return type of the function, so we're protected from some of the invasive syntax you see with, say, Scala. The downside is that these functions can fail. I'll show you some of the ways Erlanglet syou compensate for that limit at ion.

Let'susesomeofthisnewfoundpower.We useanonymousfunctions tohandlethe each, map, and inject concepts that you initially encounteredwithRuby.

ListsandHigher-OrderFunctions

As you've seen, lists and tuples are the heart and soul of functional programming.It's no accident that the first functional language started

withlists, and everything built on that foundation. In this section, you'll starttoapplyhigher-orderfunctionstolists.

ApplyingFunctionstoLists

Bynow, the ideas hould be pretty clear to you. We goingtousefunctionstohelpusmanagelists.Some,like For Each, will iterate overlists. Others, like filter or mp, will return lists, either filtered or mapped onto to or to , will processlists, rolling up otherfunctions.Stillmore,like resultsalongtheway,likeRuby's inject or Scala's FoldLeft. Openafresh console, definealistortwo, and get cracking.

First, we'llhandlebasiciteration. The lists:foreach methodtakesafunctionandalist. The function can be a nonymous, like this:

```
1>Numbers=[1,2,3,4].
[1,2,3,4]
2>lists:foreach(fun(Number)->io:format("~p~n",[Number])end,Numbers).
2
4
ok
```

Thesyntaxofline2isalittletricky,sowe'llwalkthroughit.Westartby invokingafunctioncalled lists:foreach.Thefirstargumentistheanonymousfunction fun(Number)->io:format("~p~n", [Number])end. Thatfunction has one argument and prints the value of whatever you pass in with the lo: format function. 2 Finally, the second argument to is Numbers, the list we defined on line 1. We couldsimplifythisbydefining thefunctioninaseparateline:

```
3>Print=fun(X)->io:format("~p~n",[X])end.
```

Now, Print is bound to the io: format function. We can simplify the code likethis:

```
8>lists:foreach(Print, Numbers).
1
2
3
4
ok
```

That's basic iteration. Let's move onto a function that maps. The map function works like Ruby's collect, passing each value of a list to a

^{2. ~}p prettyprintsanargument, ~n isanewline,and [Number] isalistofargumentsto print.

function and building a list with the results. Like lists:foreach, lists:map takes a function and a list. Let's use mode with our list of numbers, increasingeachvaluebvone:

```
10>lists:map(fun(X)->X+lend, Numbers).
[2,3,4,5]
```

That was easy. This time, our anonymous function was fun(X)->X+1end. It increased each value by one, and lists:map builtalist with the results.

Definingmapisreallyeasy:

```
map(F,[H|T]) \rightarrow [F(H)|map(F,T)];
map(F,[])
                 ->[].
```

Simpleenough.Themapof F overalistis F(head) plus map(F,tail) .WŁ lookatamoreconciseversionwhenwelookatlistcomprehensions.

Movingon, we can filter lists with aboolean. Let's define an anonymous functionandassignitto Small:

```
11>Small=fun(X)->X<3end.
#Fun<erl eval.6.13229925>
12>Small(4).
false
13>Small(1).
true
```

Now, we can take that function and use it to filter the list. The function lists:filter willbuildalistofalltheelementsthatsatisfy Small orthoseless thanthree:

```
14>lists:filter(Small, Numbers).
[1,2]
```

You can see that Erlang is making it very easy to code in this way. Alternatively, we can use the Small functiontotestlistswith all and any. lists:all returns true onlyifalltheitemsinalistsatisfythefilter,likethis:

```
15>lists:all(Small,[0,1,2]).
true
16>lists:all(Small,[0,1,2,3]).
false
```

Alternatively, lists: any returnstrueifanyoftheitemsinthelistsatisfies thefilter:

```
17>lists:any(Small,[0,1,2,3]).
18>lists:any(Small,[3,4,5]).
false
```

```
19>lists:any(Small,[]).
false
20>lists:all(Small,[]).
true
```

Asyou'd expect, all returns true (meaning all of the items present in thelistsatisfythefilter, though there are no items in the list), and anv returnsfalse(meaningnoelementintheemptylistsatisfiesthefilter). Inthesecases, it doesn't matter what the filter is.

Yu can also make a list of all the elements at the head of a list that matchafilterordiscardalltheitemsatthefrontofalistthatsatisfy thefilter:

```
22>lists:takewhile(Small, Numbers).
[1,2]
23>lists:dropwhile(Small, Numbers).
[3,4]
24>lists:takewhile(Small,[1,2,1,4,1]).
[1,2,1]
25>lists:dropwhile(Small,[1,2,1,4,1]).
[4,1]
```

These tests are useful to do things such as processor discard headers ofmessages.Let'sfinishthiswhirlwindwith tod and tod .

Ш

Irealizethatyou'veseentheseconceptsbefore.Ifyou'reNeoandyou've masteredthispartofthematrix, readthebasic example and fighton. For some, to take salittle while to master, so I'm going to teach it a fewdifferentways.

Remember, these functions are useful for rolling up the results of a functionacrossalist.Oneoftheargumentsservesasanaccumulator, and the other represents each item. lists:fold| takesafunction.theinitial valueoftheaccumulator.andthelist:

```
28>Numbers.
[1,2,3,4]
29>lists:foldl(fun(X,Sum)->X+Sumend,0,Numbers).
10
```

Tosimplifyalittlebit,let'sbreakthatanonymousfunctionintoavariableandmakeourintentionsclearwithbettervariablenames:

```
32>Adder=fun(ListItem,SumSoFar)->ListItem+SumSoFarend.
#Fun<erl_eval.12.113037538>
```

```
33>InitialSum=0.
34>lists:foldl(Adder,InitialSum,Numbers).
```

Ah, that's better. So, we are going to keep arunning sum. Wee going to pass the SumSoFor and each number from Numbers into a function called Adder, one atatime. Each time, the sum will get bigger, and the lists:fold| functionwillremembertherunningtotalandpassitbackinto Adder. Ultimately, the function will return the last running sum.

Sofar, allyou've seen are functions that work on existing lists. I haven't shownyouhowtobuildalistapieceatatime.Let'sshiftgearstoward listbuilding.

AdvancedListConcepts

AllofthelistconceptsI'veintroducedareextensionsoftheideasyou've seen in the other languages. But we can get a little more sophisticated.We haven't yettalked about building lists, and we've only used pretty basicabstractions with simple code blocks.

ListConstruction

On the surface, it may seem difficult to build lists without mutable state. WhRubyorlo, you would continually additems to a list. There's anotherway. Yucanreturnanewlistwiththelistitemadded. Often, you'lladditemstoalistheadfirst. Yilusethe [H|T] constructbutin therightsideofamatchinstead. This programuses the list constructiontechniquetodoubleeachitemofalist:

```
Download erlang/double.erl
-module(double).
-export([double_all/1]).
double_all([])->[];
double_all([First|Rest])->[First+First|double_all(Rest)].
```

The module exports one function, called double_all.Thatfunction has two distinct clauses. The first says that double_all for an empty list returnsanemptylist. This rules tops the recursion.

The second rule uses the [H|T] construct, but in the predicate of the matchaswellasthefunctiondefinition. Yntealreadyseen something like [First|Rest] ontheleftsideofamatch. Itletsyoubreakalistintothe firstelementandtherestofthelist.

Using it on the right side does list construction in stead of destruction. [First+First |double_all(Rest)] meansbuildalistwith Inthiscase, First+First asthefirstelementand double_all(Rest) astherestofthelist.

Yucancompileandruntheprogramasusual:

```
8>c(double).
{ok,double}
9>double:double_all([1,2,3]).
[2,4,6]
Let'stakeanotherlookatlistconstructionwith
                                                  | fromtheconsole:
14>[1|[2,3]].
[1,2,3]
15>[[2,3]|1].
[[2,3]|1]
16>[[]|[2,3]].
[[],2,3]
17>[1|[]].
[1]
```

Thereshouldbenosurprises in there. The second argument must be a list. Whatever is on the left side will be added as the first element of a newlist.

Let's look at a more advanced Erlang concept, called list comprehensions. They combine some of the concepts we have been talking about sofar.

ListComprehensions

One of the most important functions in just about any functional languageis mop . Whit, your lists can mutate, just like TheMatrix enemies. Since the feature is so important, Erlang provides amore powerfulformthatisconciseandallowsyoutodomultipletransformations atonce.

Let's start things off with a fresh console. We do a map the oldfashionedway:

```
1>Fibs=[1,1,2,3,5].
[1,1,2,3,5]
2>Double=fun(X)->X
                     * 2end.
#Fun<erl_eval.6.13229925>
3>lists:map(Double,Fibs).
[2,2,4,6,10]
```

We havealistofnumberscalled Fibs and an anonymous function called Double that will double what every oup assin. Then, we called lists:map tocall Double oneachelementandbuildalistoutoftheresult. That's a great tool, but it's used often enough that Erlang provides a much more concise way to provide the same syntax. The construct is called alistcomprehension. Here's the equivalent to what we just typed, with a listcomprehension:

```
4>[Double(X)||X<-Fibs].
[2,2,4,6,10]
```

InEnglish, we're saying compute the Double of X foreach X takenfrom thelistcalled Fibs. If you'd prefer, we can cut out the middle man:

```
5>[X * 2||X<-[1,1,2,3,5]].
[2,2,4,6,10]
```

Theconceptisthesame. We computing X*2 foreach X takenfromthe listcalled [1,1,2,3,5]. This feature is a bit more than syntactic sugar. Let's build some more sophisticated list comprehensions. We willstart withamoreconcisedefinitionof map:

```
map(F,L) \rightarrow [F(X) | |X < -L].
```

InEnglish, the map of some function F oversomelist L isthecollection of F(X) foreach X thatisamemberof L.Now,let'susealistcomprehensiontoworkwithacataloghavingaproduct, quantity, and price:

```
7>Cart=[{pencil,4,0.25},{pen,1,1.20},{paper,2,0.20}].
[{pencil,4,0.25},{pen,1,1.2},{paper,2,0.2}]
```

SaythatIneedtoaddataxthatiseightcentsonthedollar.Icanadda simplelistcomprehensiontorollupthenewcartwithtaxwithasingle listcomprehension, likethis:

```
8>WithTax=[{Product,Quantity,Price,Price
                                                * Quantity * 0.08}||
     {Product, Quantity, Price} <- Cart].
[{pencil,4,0.25,0.08},{pen,1,1.2,0.096},{paper,2,0.2,0.032}]
```

AlltheearlierErlangconceptsyou'velearnedstillapply:there'spattern matchinggoingonhere!SoinEnglish,we'rereturningalistoftuples havinga Product, Price, Quantity, and tax(Price*Quantity*0.08), for each tupleof {Product,Quantity,Price} takenfromthelistcalled Cart.Thiscode is absolutely beautiful tome. This syntax allows metochange the form ofmylist, literally ondemand.

Asanotherexample, say I have a catalog and I want to provide a similarcatalogtomypreferredcustomerswitha50percentdiscount.The catalogcouldlooksomethinglikethis. I'lljusttakethecatalogfrom the cart, ignoring quantity:

```
10>Cat=[{Product,Price}||{Product,_,Price}<-Cart].
[{pencil, 0.25}, {pen, 1.2}, {paper, 0.2}]
```

InEnglish, give metuples with Product and Price foreachtupleof Product, and Price (ignoring the second attribute) taken from the Cart list. Now, Ican provide my discount:

```
11>DiscountedCat=[{Product,Price/2}||{Product,Price}<-Cat].
[{pencil,0.125},{pen,0.6},{paper,0.1}]
```

It's concise, readable, and powerful. It's abeautiful abstraction.

Intruth, I've showed you only part of the power of a list comprehension. The full form can be even more powerful:

- A list comprehension takes the form of [Expression | Clause 1. Clause2,...,ClauseN].
- Listcomprehensionscanhaveanarbitrarynumberofclauses.
- The clauses can be generators or filters.
- A filter can be a boolean expression or a function returning a boolean.
- Agenerator, of the form Match<-List, matchesapatternontheleft totheelementsofalistontheright.

Really, it's not too hard. Generators add, and filters remove. There's a lotofProloginfluencehere.Generatorsdeterminethepossiblevalues, and filters trim the list down to the specified conditions. Here are a coupleofexamples:

```
[X | | X < -[1,2,3,4], X < 4, X > 1].
[2,3]
```

InEnglish, return X, where X istakenfrom [1,2,3,4], X isless than four, and X isgreaterthanone. Yucanalsohavemultiplegenerators:

```
23>[\{X,Y\}||X<-[1,2,3,4],X<3,Y<-[5,6]].
[{1,5},{1,6},{2,5},{2,6}]
24>
```

{X,Y} bycombining X values from [1,2,3,4] that Thisonemakesatuple Y values from [5,6]. You windup with two X values arelessthan3with andtwo Y values, and Erlang computes a Cartesian product.

Andthat's the wholestory. Yorkelearned to use Erlangtodos equential programming. Let's take a break to wrap up and put this stuff into practice.

WhatWeLearnedinDay2

Admittedly, we didn't go into deep detail about Erlang expressions or the library, but you are now armed with enough information to write

functional programs. Yustarted the day with some mundane control structures, but we picked up the pacequickly.

Next, we covered higher-order functions. You used higher-order functionswithinliststoiteratethroughlists, filterthem, and modify them. also learned to use to roll up results, just as you did with **6**d Scala.

Finally, we moved on to advanced list concepts. We used [H|T] on the left side of a match to deconstruct a list into the first element of the listandtherest.We used [H|T] ontherightsideofamatch,orsolo,to constructlists, headfirst. We thenmovedontolistcomprehensions, an elegantandpowerfulabstractionthatcanquicklytransformlistswith generatorsandfilters.

Thesyntaxwasamixedbag. Yucouldcruisethroughthehigherconceptswith very little typing, thanks to Erlang's dynamic typing strategy.Still,thereweresomeawkwardmoments,especiallywiththesemicolonsafterthevariouspiecesof case and if clauses.

In the next section, we'll learn what all of the fuss was about. We tackleconcurrency.

Day2Self-Study

Do:

- Consider a list of keyword-value tuples, such as [{erlang, "afunctionallanguage"},{ruby,"anOOlanguage"}]. Weafunction that accepts the list and a keyword and returns the associated value forthekeyword.
- Consider a shopping list that looks like [{item quantity price},...]. We a list comprehension that builds a list of items of the form [{itemtotal_price},...],where total_price is quantity times price.

Bonusproblem:

• Weaprogramthatreadsatic-tac-toeboardpresentedasalist or a tuple of size nine. Return the winner (x or o) if a winner has been determined, cat if there are no more possible moves, or no_winner ifnoplayerhaswonyet.

6.4 Day3:TheRedPill

Mostofyouhavehearditbefore.Inthematrix,takethebluepill,and you can continue to live in blissful ignorance. Take the red pill, and youreyesareopentoreality. Sometimes, realityhurts.

We haveawholeindustryslammingbluepillslikeapreacher'skidin Amsterdam. Concurrency is hard, so we punt. We addmutablestate, so our programs collide when we run them concurrently. Our functions and methods have side effects, so we can't prove correctness or predict their outcomes. We usethreadswithsharedstateratherthan shared-nothingprocesses for performance, so we have to do extrawork toprotecteachpieceofcode.

The result is chaos. Concurrency hurts, not because it is inherently difficult but because we've been using the wrong programming model!

Earlier in the chapter, Is a id Erlang made some easy things hard. Whoutsideeffects and mutable state, you'll have to change the way you approach coding altogether. Yihave to put up with a Prolog-based syntaxthatseems alien to many. But now, you'll get the payoff. That redpill, concurrency and reliability, will seem like can dy to you. Let's findouthow.

BasicConcurrencyPrimitives

Yhrthreebasic primitives for concurrency will be sending a message (using!), spawning a process (with spown), and receiving a message (with receive). In this section, I'll showyou how to use the seth reeprimitivestosendandreceiveamessageandtowraptheminabasicclientserveridiom.

ABasicReceiveLoop

start with a translation process. If you send the process a stringWŁ in Spanish, it will reply with an English translation. In general, your strategy will be to spawn a process that receives and processes the messageinaloop.

```
Download erlang/translate.erl
-module(translate).
-export([loop/0]).
loop()->
    receive
         "casa" ->
            io:format("house~n"),
             loop();
         "blanca" ->
             io:format("white~n"),
             loop();
             io:format("Idon'tunderstand.~n" ),
             loop()
```

end.

That's longer than our other examples so far, so we'll break it down. Thefirsttwolinesjustdefinethemodulecalled translate and export the functioncalled loop. The next block of code is the function called loop():

```
loop()->
end.
```

Noticethatthecodeinsidecalls loop() threetimes, without any returns. That's OK: Erlangis optimized for tail recursion, so there's very little overhead, as long as the last thing in any receive clause is a loop().

basicallydefininganemptyfunctionandloopingforever. Moving Water ontothe receive:

```
receive->
```

This function will receive a message from another process. receive works like the other pattern matching constructs in Erlang, the case and the function definitions. Yal follow receive with several pattern matchingconstructs. Moving onto the individual matches:

```
"casa"->
   io:format("house~n"),
   loop();
```

This is a matching clause. The syntax is nicely consistent with case statements. If the inbound message matches the string "casa", Erlang

will execute the following code. Separate lines are delimited with a character, and you'll terminate the clause with a ; character.Thiscode displays the word house and then calls loop. (Remember, there's no overheadonthestack, because loop is the last function called.) All of theothermatchingclauseslookthesame.

Now, we have a module with a receive loopinit. It's time to put it to use.

SpawningaProcess

First, we compile this module:

```
1>c(translate).
{ok,translate}
```

Tospawnaprocess, you'lluse the function spawn, which takes a function. That function will be started in a newlight weight process. spawn passinthefunctionfromour returnsaprocessID(PID).We translate module.likethis:

```
2>Pid=spawn(funtranslate:loop/0).
<0.38.0>
```

YucanseethatErlangreturnedtheprocessIDof <0.38.0>.Intheconsole, you'll see process IDs enclosed in angle brackets. We goingto cover only the primitive version of process spawning, but you should knowaboutafewotherstoo. Yu can register processes by name, so other processes can find, say, common services by namerather than processID.Yucanalsouseanotherversionof spawn forcodethatyou wanttobeabletochangeonthefly, orhot-swap. If youwere spawning aremote process, you would use spawn(Node, function) instead. These topicsarebeyondthescopeofthisbook.

Sonow, we'vecoded a module with a codeblock, and we've spawned it asalightweightprocess. The last step is to pass messages to it. That is thethirdErlangprimitive.

SendingMessages

AsyousawinScala, youwill pass distributed messages to Erlang with the ! operator. The form is Pid!message . The Pid is any processidentifier. message can be any value, including primitives, lists, or tuples. Let's sendafewmessages:

```
3>Pid!"casa".
"house"
"casa"
```

```
4>Pid!"blanca".
"white"
"blanca"
5>Pid!"loco".
"Idon'tunderstand."
"loco"
```

Eachlinesendsamessage. The io:format in our receive clauses prints amessage, and then the console prints the return value of the expression, which is the message you sent.

Ifyouweresendingadistributedmessagetoanamedresource, you'd usethesyntax node@server!message instead.Settinguparemoteserver isbeyondthescopeofthisbook, but with very littleself-study, you can easilygetadistributedservergoing.

This example illustrates the basic primitives and how you'd we ave them together to formabasic as ynchronous service. Yn mayhavenoticed that there is no return value. In the next section, we'll explore how to sendsynchronousmessages.

Synchronous Messaging

Someconcurrentsystemsworkasynchronously, likephonechats. The sender transmits a message and goes on, without waiting for a response. Otherswork synchronously, like the Web We askforapage, and the web servers ends it while we wait for the response. Let's turnthe translations ervice that prints return values into a service that actuallyreturnsthetranslatedstringtotheuser.

To change our messaging model from asynchronous to synchronous, we'llhaveathree-partstrategy:

- Each receive clause in our messaging service will need to match a tuple having the ID of the process requesting the translation and the word to translate. Adding this ID will allow us to send a response.
- Each receive clause will need to send a response to the sender insteadofprintingtheresult.
- Insteadofusingthesimple ! primitive, we'llwriteasimplefunction tosendtherequestandawaittheresponse.

Now that you know the background, take a look at the pieces of the implementation.

ReceivingSynchronously

Thefirstorderofbusinessistomodifyour receive clausestotakeadditional parameters. That means we're going to have to use tuples. Patternmatchingmakesiteasy.Each receive clauselookslikethis:

```
receive
    {Pid, "casa"}->
        Pid!"house",
        loop();
```

We matchanyelement(this should always be a process ID), followed by theword casa. We then send the word house tothereceiverandloop backtothetop.

Note the pattern match. This is a common form for a receive, where theIDofthesendingprocessisthefirstelementofatuple.Otherwise, the only major difference is sending the result rather than printing it. Sendingamessagegetsalittlemorecomplicated, though.

SendingSynchronously

Theothersideoftheequationneeds to send amessage and then immediately wait for a response. Given a process ID in Receiver, sending a synchronousmessagewilllooksomethinglikethis:

```
Receiver! "message_to_translate",
    receive
        Message->do_something_with(Message)
    end
```

Sincewe'llbe sending messages sooften, we'll simplify the service by encapsulatingarequesttotheserver. Inourcase, that simpleremote procedurecalllookslikethis:

```
translate(To,Word)->
   To!{self(),Word},
    receive
        Translation->Translation
    end.
```

Whenyouputitalltogether, you get a concurrent program that 's only marginallymorecomplicated.

```
Download erlang/translate_service.erl
-module(translate service).
-export([loop/0,translate/2]).
loop()->
    receive
        {From, "casa"}->
            From! "house",
            loop();
        {From, "blanca"}->
            From! "white",
            loop();
        {From,_}->
            From!
                   "Idon'tunderstand." ,
            loop()
end.
translate(To, Word) ->
    To!{self(),Word},
    receive
        Translation->Translation
    end.
Theusagemodellookslikethis:
1>c(translate_service).
{ok,translate service}
2>Translator=spawn(funtranslate_service:loop/0).
<0.38.0>
3>translate_service:translate(Translator, "blanca").
"white"
4>translate_service:translate(Translator, "casa").
"house"
```

We simply compile the code, spawn the loop, and then request a synchronousservicethroughthehelperfunctionwewrote. Asyoucansee, the Tortor processnowreturnsthetranslatedvaluefortheword.And now, you have a synchronous message.

Now, you can see the structure of a basic receive loop. Each process hasamailbox. The receive constructjustpicksmessages off the queue and matches them to some function to execute. Processes communicatebetween one another with message passing. It's no accident that Dr.ArmstrongcallsErlangatrueobject-orientedlanguage!Itgivesyou messagepassingandencapsulationofbehavior. Wejustlosing mutablestateandinheritance, thoughit's possible to simulate inheritance, andmore, through higher-order functions.

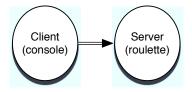


Figure 6.1: Simple client-server design

Sofar, we'veworkedinbasic, sterile conditions with no error recovery capability. Erlangdoes provide checked exceptions, but I want to walk youthroughanotherwayofhandlingerrorsinstead.

LinkingProcesses&Reliability

Inthissection, we'regoing to look at ways to link processes for better reliability.InErlang,youcanlinktwoprocessestogether.Whenevera $process\,dies, it\,sends\,an\,exit\,signal\,to\,its\,linked\,twin.\,Aprocess\,can$ thenreceivethatsignalandreactaccordingly.

SpawningaLinkedProcess

Toseehowlinkingprocessesworks,let'sfirstcreateaprocessthatcan easilydie. I'vecreated a Russian roulette game. It has a gun with six chambers. To fire a chamber, you just send a number 1-6 to the gun process. Enter the wrong number, and the process kills itself. Here's thecode:

```
Download erlang/roulette.erl
-module(roulette).
-export([loop/0]).
%sendanumber, 1-6
loop()->
    receive
                         "bang.~n"),exit({roulette,die,at,erlang:time()});
        3->io:format(
                         "click~n"),loop()
        _->io:format(
end.
```

Theimplementationisprettyeasy. We haveamessageloop. Matching 3 executes the code io:format("bang~n"),exit({roulette,die,at,erlang:time()};, killingtheprocess. Anythingelsejustprintsamessageandgoesback tothetopoftheloop.

We reallyhaveasimpleclient-serverprogram. The clientisthe console, andtheserveristhe ndete process, as shown in Figure 6.1.

Andhere's what the execution looks like:

```
1>c(roulette).
{ok,roulette}
2>Gun=spawn(funroulette:loop/0).
<0.38.0>
3>Gun!1.
"click"
4>Gun!3.
"bang"
5>Gun!4.
6>Gun!1.
1
```

The problem is that after a 3, the gun process is dead, and further canactuallytellwhethertheprocessisalive: messagesdonothing.We

```
7>erlang:is_process_alive(Gun).
false
```

The process is definitely dead. It's time to get on the cart. We cando a little bit better. Let's build a monitor that will tell us whether the process dies. I guess that's more of a coroner than a monitor. We onlyinterestedindeath.

Here'swhatthecodelookslike:

```
Download erlang/coroner.erl
-module(coroner).
-export([loop/0]).
loop()->
    process_flag(trap_exit,true),
    receive
        {monitor,Process}->
            link(Process),
            io:format("Monitoringprocess.~n" ),
            loop();
        {'EXIT',From,Reason}->
            io:format("Theshooter~pdiedwithreason~p."
                                                            ,[From,Reason]),
                        io:format("Startanotherone.~n" ),
            loop()
        end.
```

Asusual, we'rebuildinga receive loop. Before wedoanything else, the programmust register the process as one that will trap exits. Y uwon'treceiveEXITmessageswithoutit.

Then, we process are ceive. There ceive gets two types of tuples: those beginning with the atom monitor and those beginning with the string 'FXIT', Let'slookateachinmoredetail.

```
{monitor,Process}->
   link(Process),
    io:format("Monitoringprocess.~n"),
    loop():
```

This code links the coroner process to any process with a PID of Process. Yucanalsos pawna process with the links already in place with spawn_link.Now,ifthemonitoredprocessshoulddie,itwillsendanexit messagetothis coroner. Moving onto trapping the error:

```
{'EXIT', From, Reason}->
    io:format("Theshooterdied.Startanotherone.~n"),
    loop()
end.
```

Thisisthecodethatmatchestheexitmessage. It will be a three-tuple with 'EXIT', followed by the PID from the dying process as From and the reason for failure. We print the PID of the dying process and the reason.Here's the overall flow:

```
1>c(roulette).
{ok,roulette}
2>c(coroner).
{ok,coroner}
3>Revolver=spawn(funroulette:loop/0).
<0.43.0>
4>Coroner=spawn(funcoroner:loop/0).
<0.45.0>
5>Coroner!{monitor,Revolver}.
Monitoringprocess.
{monitor, <0.43.0>}
6>Revolver!1.
click
7>Revolver!3.
bang.
Theshooter<0.43.0>diedwithreason
{roulette, die, at, {8,48,1}}. Startanotherone.
```

Now, we're getting more sophisticated than client-server. Whe addeda monitorprocess, asin Figure 6.2, on the following page, so we can tell whentheprocessdies.

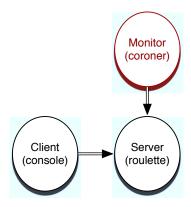


Figure 6.2: Adding monitoring

FromCoronertoDoctor

We candobetter. If we register the gun (gunpunint ended), game playerswillnolongerhavetoknowthePIDoftheguntoplay.Then,wecan pushtheguncreationintothecoroner. Finally, the coroner can restart the process whenever the process dies. And we've achieved much betterreliability and without excessive error reporting. At this point, the coronerisnotjustacoroner. He'sadoctor, and one that's capable of raisingthedead. Here's the newdoctor:

```
Download erlang/doctor.erl
-module(doctor).
-export([loop/0]).
loop()->
    process_flag(trap_exit,true),
    receive
        new->
            io:format("Creatingandmonitoringprocess.~n" ),
            register(revolver,spawn_link(funroulette:loop/0)),
            loop();
        {'EXIT', From, Reason}->
            io:format("Theshooter~pdiedwithreason~p."
                                                              ,[From,Reason]),
                         io:format("Restarting.~n" ),
            self()!new,
            loop()
        end.
```

The receive blocknowmatches two messages: new and the same EXIT tuple. They are both a little different from the coroner predecessor. Thisisthemagiclineofcodeinthe new block:

```
register(revolver,spawn_link(funroulette:loop/0)),
```

from the inside out, we spawn a process with spawn_link.That version of spawn will link the processes so the doctor willgetnotified thenregisterthePID, associating whenevera net processdies.We itwiththe revolver atom. Now, users can send message stothis process byusing revolver!message .We nolongerneedthePID.TheEXITmatch blockisalsosmarter. Here's the newline of code:

```
self()!new,
```

Wesendamessagetoourself, spawning and registering an ewgun. The gameismucheasiertoplay,too:

```
2>c(doctor).
{ok,doctor}
3>Doc=spawn(fundoctor:loop/0).
<0.43.0>
4>revolver!1.
** exceptionerror:badargument
     inoperator !/2
        calledasrevolver!1
```

As expected, we have not created the process yet, so we get an error. Now, we'llcreateandregisterone:

```
5>Doc!new.
Creatingandmonitoringprocess.
6>revolver!1.
click.
1
7>revolver!3.
bang.
Theshooter<0.47.0>diedwithreason{roulette,die,at,{8,53,40}}.
    Restarting.
Creatingandmonitoringprocess.
8>revolver!4.
click.
```

We nowtaketheincongruousstepofcreatingthe revolver throughthe Doctor. We interact with the revolver by sending messages through the revolver atominsteadofthe Gun PID. Yucanalsoseeafterline8that we in fact create and register a new revolver. The overall topology is

generallythesameasitwasinFigure6.2,onpage216,withthe playingamoreactiverolethanthe coroner did.

doctor

We just scratched the surface, but I hope that you can see how Erlangcan make it easy to create much more robust concurrent systems. You don't seem uch error handling here at all. When something crashes, we just start a new one. It's relatively simple to build monitors that watch each other. In fact, the base libraries have plenty of tools to build monitoring services and keep-alives that Erlangrestarts automatically upon any kind of failure.

WhatWeLearnedinDay3

In day 3, you started to get a pretty good feel for what you can do with Erlang. We started with the concurrency primitives: send, receive, and spawn. We built the natural asynchronous version of a translator to illustrate how basic message-passing works. Then, we built a simple helper function that encapsulated as end and are ceive to get her to simulate are mote procedure call with a send and receive.

Next, we linked processes together to show how one process notifies another when it dies. We also learned to monitor one process with anotherforbetterreliability. Oursystemwasnotfaulttolerant, though the ideas that we used could be used to build fault-tolerant systems. Erlangdistributed communication works exactly like interprocess communication. We could link two processes, on separate computers, so that a standby monitored the master and took over in the event of a problem.

Let's put some of what you've learned to work.

Day3Self-Study

These exercises are relatively easy, but I did add some bonus questions to stretchy ou a little bit.

OpenTelecomPlatform(OTP)isapowerfulpackagewithmuchofwhat you'llneedtobuildadistributed,concurrentservice.

Find:

- ${\color{red} \bullet} An OTP service that will restart a process if it dies$
- ${\color{gray}\bullet} \, Document at ion for building a simple OTP server$

Do:

- Monitorthe translate service and restartits hould it die.
- Makethe Doctor processrestartitselfifitshoulddie.
- Doctor monitor. If either monitor dies. • Make a monitor for the restartit.

The following bonus questions will take a little bit of research to complete:

- CreateabasicOTPserverthatlogsmessagestoafile.
- Makethe translate_service workacrossanetwork.

6.5 WrappingUpErlang

Atthebeginning of this chapter, I said that Erlang made hard things easy and easy things hard. The Prolog-style syntax is alien to those familiar with the broadfamily of C-stylelanguages, and the functional programming paradigm has it sown set of challenges.

ButErlanghassomecorecapabilitiesthatwillbetremendouslyimportant as new hardware designs make programming for concurrency more important. Some of the capabilities are philosophical. The lightweightprocessesruncountertoJava'sthreadandprocessmodels.The "Letitcrash" philosophysimplifies codetremendously but also requires basesupportatthevirtualmachinelevelthatjustdoesn'texistinother systems.Let'sbreakdownthecoreadvantagesanddisadvantages.

CoreStrengths

Erlangisallaboutconcurrencyandfaulttolerance, from thein sideout. Asprocessordesignerslooktodistributedcores, the state of the artin programmingmustevolve. Erlang's strengths are in the most important areasthatthisgenerationofprogrammerswillface.

DynamicandReliable

First and foremost, Erlang is built for reliability. The core libraries havebeentested, and Erlang applications are among themost reliable and available in the world. Most impressively, the language designers achieved this reliability without sacrificing the dynamic typing strategiesthatmakeErlangsoproductive.Ratherthandependonthecompiler for an artificial safety net, Erlang depends on the ability to link

concurrent processes, reliably and simply. I was a stounded about how easyitwastobuilddependablemonitorswithoutrelyingonoperating systemkludges.

IthinkthesetofcompromisesthatyoufindinErlangiscompellingand unique. The Java language and virtual machine does not provide the rightsetofprimitivestoduplicateErlangperformanceorreliability.The librariesbuiltontheBEAMalsoreflectthisphilosophy, soit's relatively easytobuildreliable, distributed systems.

Lightweight, Share-Nothing Processes

Another place Erlang shines is the underlying process model. Erlang processes are light, so Erlang programmers use them often. Erlang buildsonaphilosophyofenforcingimmutability, soprogrammers build systems that are inherently less likely to fail by conflicting with one another. The message-passing paradigm and primitives make it easy to code applications with a level of separation that you rarely find in object-oriented applications.

CP, the Enterprise Libraries

SinceErlanggrewupinatelecomcompanywithhighrequirementsfor availability and reliability, it has twenty years worth of libraries that supportthisstyleofdevelopment. The primary library is Open Telecom Platform (OTP). You can find libraries that help you build monitored, keep-alive processes; link to databases; or build distributed applications.OTPhasafullwebserverandmanytoolsforbindingtotelecom applications.

Thenicethingaboutthesetoflibrariesisthatfaulttolerance, scalability,transactionalintegrity,andhot-swappingareallbuiltin. Yudon't have to worry about them. You can build your own server processes thattakeadvantageofthesefeatures.

LetItCrash

When you're dealing with parallel processes and no side effects, "Let itcrash"works-you'renotasworriedaboutwhyindividualprocesses crash because you can restart them. The functional programming modelamplifiesthebenefitsofErlang'sdistributionstrategy.

Like all the other languages in this book, Erlangistainted. Only the nature of the problem schanges. These are the places that Agent Smithmaynotalwaysplaysonice.

Erlang's fundamental problems with adoption come from roots firmly plantedinanichelanguage. The syntaxisalient othevast majority of programmers. Also, the functional programming paradigm is different enoughthatitwillcauseproblems with widespread adoption. Finally, byfarthebestimplementationisontheBEAMandnottheJavavirtual machine.Let'sdiginalittledeeper.

Syntax

Likeamovie, syntaxis subjective. Beyondthis problem, though, Erlang has some problems that even the impartial will notice. Let's look at two ofthem.

Interestingly, some of Erlang's core strengths come from the Prolog foundations, as well as its weaknesses. To most of the programming population, Prologis obscure, and the syntax comes off as awkward andalien. Alittlesyntactic sugartoease the transition could go along way.

Inthechapter, Imentioned the problems with if and case constructs. The syntactic rules are logical—use as eparator between statements but not practical because you can't change the order of case, if, or receive blocks without having to change your punctuation. These restrictions are unnecessary. And there are other oddities, such as the conditional presentation of an array of numbers as strings. Cleaning theseupwouldhelpErlangtremendously.

Integration

Aswiththe Prologheritage, not being on the JVM has been a doubleedgedsword.Recently,aJVM-basedVMcalledErjanghasmadeprogressbutisnotyettothelevelofthebestJVMalternatives.TheJVM doescomewithbaggage, such as a process and threading model that's inadequateforErlang'sneeds.ButbeingontheJVMhasasetofadvantages as well, too, including the wealth of Javalibraries and the hundredsofthousandsofpotentialdeploymentservers.

FinalThoughts

The success of a programming language is a ficklething. Erlangfaces someseriousobstaclesonthemarketingside,andhavingtolureJava

programmersovertoaLisp-styleprogrammingparadigmandaPrologstylesyntax won't be easy. Erlang does seem to be gathering momentumbe cause its olves the right problems in the right way at the right $time. In this battle \, between \, Anderson \, and \, Agent \, Smith, I give \, Agent \,$ Smith an even-money chance at success.

Chapter 7 Clojure

Clojure is Lisp on the JVM. Perplexing and powerful, Lisp is one of the first programming languages and one of the latest too. Dozens of dialectstriedtolaunchLispintothemainstreamandfailed.Thesyntax and programming models of arhave been too much for a typical developertoabsorb. Yat there's something special about Lisp that's worth revisiting, so the new dialects continue to emerge. Some of the best programming universities lead with the Lisp language to form young mindswhiletheyarestillopen.

In many ways, Clojure is the wise kung fu master, the oracle on the hill, or the enigmatic Jedi trainer. Think Yoh In Star Was **Episode** V.TheEmpireStrikesBack ,¹ Yohwasintroducedasacutecharacter with little significance. His communication style is often inverted and hard to understand, like Lisp prefix notation (understand melater vau will). He seems too small to make a difference, like the Lisp syntactic rules with little more than parentheses and symbols. But it quickly becomes apparent that there is more to Yok than meets the eye. As withLisp, heisold, withwisdom (likethequoteabove) that has been honedbytimeandtriedunderfire.LikeLisp'smacrosandhigher-order constructs, hehasaninner power that others can't seem to master. In many ways, Lisp started it all. Before diving in too deeply, let's talk a little bit about Lisp and then shift gears to what's exciting about Clojure.

 $^{1. \}quad Star \textit{Was} Episode \textit{VThe Empire Strikes} Back$ erlyHills,CA:20thCenturyFox,2004.

IntroducingClojure 7.1

When all is said and done, Clojure is yet another Lisp dialect. It will havethesamelanguagelimitationsandmanyofthesameconsiderable strengths. Understanding Clojurestarts with understanding Lisp.

AllLisp

After Fortran, Lisp is the oldest commercially active language. It's a functionallanguagebutnotapurefunctionallanguage. The acronym standsforLIStProcessing,andearlyon,you'llseewhy.Lisphassome interestingcharacteristics:

- Lispisalanguageoflists. Afunction calluses the first list element asthefunctionandtherestasthearguments.
- Lispusesitsowndatastructurestoexpresscode. Lispfollowers callthisstrategy dataascode.

Whenyoucombinethesetwoideas, you get alanguage that 's ideal for metaprogramming. Yucanarrangeyourcodeasnamedmethodsina class. Yucouldarrangethoseobjects into a tree, and you have a basic could also build a prototype-based code organizaobject model. You tionwith slots for data and behavior. Yu can build a pure-functional implementation. It's this flexibility that allows Lisp to morph the languageintojustaboutanyprogrammingparadigmyouwant.

In HackersandPainters [Gra04], PaulGrahamchroniclesthestory of how a small development team used Lispan dits powerful programmingmodeltodefeatmuchlargercompanies. They believed Lispprovideda significantprogrammingadvantage. Infact, they paid more attention to start-upspostingjobsrequiringLispandotherhigher-levellanguages.

TheprimaryLispdialectsareCommonLispandScheme.Schemeand Clojure are from the same family of dialects called lisp-1, and CommonLispisalisp-2dialect. The primary difference between the dialect families has to do with the way names paces work. Common Lispuses aseparatenamespaceforfunctions and variables, while Schemeuses the same names pace for both. Whithe Lispside of the equation behind us, let's move onto the Javaside.

OntheJVM

EveryLisp dialect caters to its audience. For Clojure, one of the most important characteristics is the JVM. Wh Scala, you saw that having a commercially successful deployment platform can make all the differenceintheworld. Youdon't have to sell a Clojure server to your deployment people to use it. Though the language is relatively new, youcanaccessthetensofthousandsofJavalibrariestodojustabout anythingyouneed.

Throughoutthischapter, you'll see evidence of the JVM, in the way you invokeit, in the libraries weuse, and in artifacts we create. But you'll also see liberation, too. Clojure is functional, so you'll be able to apply advancedconceptstoyourcode. Clojureisdynamicallytyped, soyour codewillbemoreconcise, easier to read, and more fun to write. And ClojurehastheexpressivenessofLisp.

Clojure and Java desperately need each other. Lisp needs the market placethattheJavavirtualmachinecanoffer,andtheJavacommunity needsaseriousupdateandaninjectionoffun.

UpdatedtraConcurrentWtd

The last piece of the equation for this language is the set of libraries. Clojure is a functional language, with emphasis on functions without side effects. But when you do use mutable state, the language supportsanumber of concepts to help. Transactional memory works like transactional databases to provide safe, concurrent access to memory.Agentsallowencapsulatedaccesstomutableresources.Wicover someoftheseconceptsinday3.

Impatientareyou?StartwithClojure,wewill.

7.2 Day1: tingLuke

In StarWas , the apprentice Lukejoined with Yata for advanced traininginthewaysoftheJedi.Hehadstartedhistrainingunderanother. Like Luke, you have already started your training for functional languages. Yoused closures in Rubyand graduated to higher-order functionsinScalaandErlang.Inthischapter,you'regoingtolearntoapply someofthoseconceptsinClojure.

GototheClojurehomesiteat http://www.assembla.com/wiki/show/clojure/ Getting_Started. Follow the instructions to install Clojure on your platform and with your preferred development environment. I'm using a prerelease version of Clojure 1.2, and it should be fully ready by the time this book is in your hands. You may first need to install the Javaplatform, thoughtoday, most operating systems come with Java

² tomanagemyClojureprojects installed. I'musing the leining entool and Javaconfiguration. That toollets mebuild a project and insulates me from the Java details such as class paths. Once that's installed, I canthencreateanewproject:

```
batate$leinnewseven-languages
Creatednewprojectin:seven-languages
batate$cdseven-languages/
seven-languagesbatate$
```

Then, I can start the Clojure console, called the repl:

```
seven-languagesbatate$leinrepl
Copying2filesto/Users/batate/lein/seven-languages/lib
user=>
```

...and I'm off to the races. Underneath, leiningen is installing some dependencies and calling Java with a few Clojure Java archives (jars) installation may require you to start the repl some and options. Yar otherway. From hereonout, I'll just telly out ost art the repl.

Afterallofthatwork, you have a primitive console. When I askyou to evaluatecode, you can use this repl, or you can use one of any number ofIDEsoreditorsthathaveClojuresupport.

Let'stypesomecode:

```
user=>(println"GivemesomeClojure!")
GivemesomeClojure!
nil
```

So, the console is working. In Clojure, you'll enclose any function call entirelyinparentheses. The first element is the function name, and the restarethearguments. Yucannest them, too. Let's demonstrate the conceptwithalittlemath.

CallingBasicFunctions

```
user=>(-1)
-1
user = > (+11)
user=>( * 1010)
100
```

That's just basic math. Division is a little more interesting:

```
user=>(/13)
1/3
```

^{2.} http://github.com/technomancy/leiningen

```
user=>(/24)
1/2
user = > (/2.04)
0.5
user=>(class(/13))
clojure.lang.Ratio
```

Clojurehasabasicdatatypecalleda ratio. It's anice feature that will allowdelayingcomputationtoavoidthelossofprecision. Yucanstill easilyworkinfloatsifyousochoose. Yucaneasilycalculateremainders:

```
user=>(mod54)
```

That's short for modulus operator. This notation is called prefixnotation. Languages to this point have supported infix notation, with the operator between the operands, like 4+1-2. Many people prefer in fix notationbecausewe'reusedtoit.We usedtoseeingmathexpressed inthisway. Afterwarmingup, you should be getting into the flow of prefixnotation.It's alittle awkward doing mathin this way, but it works. Prefix notation with parentheses does have advantages, though. Considerthisexpression:

```
user=>(/(/122)(/62))
```

There's no ambiguity. Clojure will evaluate this statement in parentheticalorder. And check out this expression:

```
user=>(+2222)
8
```

You can easily add elements to the calculation, if you so choose. You can even use this style when working with subtraction or division:

```
user = > (-812)
user = > (/822)
```

Weevaluated(8-1)-2and(8/2)/2inconventional(infix)notation. Or,ifyou'dliketoseetheClojureequivalentusingonlytwooperandsat atime.itis (-(-81)2) and (/(/82)2) .Yucanalsogetsomesurprisingly powerfulresultsoutofsimpleoperatorevaluation:

```
user=>(<123)
true
user=>(<1324)
false
```

Nice. We can see whether a list is in order with a single operator and anarbitrarynumberofarguments.

Asidefromtheprefixnotation and the extrawrinkle of multiple parameterlists, Clojure's syntaxis very simple. Let's try to push the typing systemalittle, probing for strong typing and coerced types:

```
user = > (+3.05)
8.0
user = > (+35.0)
8.0
```

Clojureworkstocoercetypesforus.Ingeneral,you'llfindthatClojure supports strong, dynamic typing. Let's get a little more focused and lookatsomeofClojure'sbasicbuildingblocks,called forms.

Think of a form as a piece of syntax. When Clojure parses code, it firstbreakstheprogramdownintopiecescalled forms. Then, Clojure cancompileorinterpretthecode. I'mnotgoing to distinguish between codeanddata, because in Lisp, they are one and the same. Booleans, characters, strings, sets, maps, and vectors are all examples of forms thatyou'llseethroughoutthischapter.

StringsandChars

already been introduced to strings, but we can take it a little Yorke deeper. Yelenclosestrings indouble quotes, and use C-stylees caping characters, as in Ruby:

```
user=>(println"masteryoda\nlukeskywalker\ndarthvader")
masteryoda
lukeskywalker
darthvader
ni1
```

No surprises. As an aside, so far, we've used a single argument with println, butthefunctionalsoworks well with zero or more arguments to printablanklineorseveralvaluesconcatenatedtogether.

InClojure, you can convert thing sto a string with the str function:

```
user=>(str1)
"1"
```

If the target underneath is a Java class, str will call the underlying toString function. This function takes more than one argument, like this:

```
user=>(str"yoda,""luke, ""darth")
"yoda, luke, darth"
```

Asaresult, Clojure developers use str to concatenate strings together. Conveniently, you can concate nate items that are not strings:

```
user=>(str"one:"1"two:"2)
"one:1two:2"
```

Yucanevenconcatenatedifferenttypestogether. Torepresentacharacteroutside of double quotes, you can precede it with a \ character. likethis:

```
user=>\a
\a
```

Andasusual, you can concate nate them to gether with str:

```
user = > (str\f\o\r\c\e)
"force"
```

Let'smakesomecomparisons:

```
user=>(="a"\a)
false
```

So, characters are not strings of length 1.

```
user = > (= (str \setminus a) "a")
true
```

Butyoucaneasily convert characters to strings. That's enough string manipulation for now. Let's move onto some boolean expressions.

BooleansandExpressions

Clojurehasstrong, dynamic typing. Recall that dynamic typing means typesareevaluatedatruntime. Yorealreadyseenafewofthetypes in action, but let's focus that discussion a little bit. Aboolean is the resultofanexpression:

```
user = > (=11.0)
true
user=>(=12)
false
user=>(<12)
true
```

Aswithmostotherlanguagesinthisbook, true isasymbol. Butitisalso something else. Clojure's types are unified with the underlying Java typesystem. You cangettheunderlying class with the class function. Here'stheclassofaboolean:

```
user=>(classtrue)
java.lang.Boolean
user=>(class(=11))
java.lang.Boolean
```

So, you can see the JVM peekingthrough. This type strategy will make thingsfarmoreconvenientforyouasyoumovealong. Yucanusethe resultofbooleansinmanyexpressions.Hereisasimple

```
user=>(iftrue(println"Trueitis."))
Trueitis.
ni1
user=>(if(>12)(println"Trueitis."))
ni1
```

Like Io, we passed in code as the second argument to the if. Conveniently, Lispletsustreatthedataascode. We canmakeitprettierby breakingthecodeacrossmultiplelines:

```
user=>(if(<12)
  (println"Falseitisnot."))
Falseitisnot.
ni1
```

We canaddan else asthethirdargument:

```
user=>(iffalse(println"true")(println"false"))
false
ni1
```

Now, let's see what else we can use as booleans. First, what passes for a nil inClojure?

```
user=>(first())
nil
```

Ah.That's simple. The symbol called nil.

```
user=>(if0(println"true"))
true
ni1
user=>(ifnil(println"true"))
user=>(if""(println"true"))
true
ni1
```

0 and "" aretrue, but nil isnot. Wintroduce other boolean expressions asweneedthem.Now,let'slookatsomemorecomplexdatastructures.

Lists, Maps, Sets, and Vectors

Aswithallfunctionallanguages, coredatastructures such as lists and tuples do the heavy lifting. In Clojure, three bigones are lists, maps, andvectors.We startthingsoffwiththecollectionyou'vespentmost ofyourtimewithsofar, the list.

Lists

Alistisanordered collection of elements. These elements can be anything, but in idiomatic Clojure, lists are used for code and vectors for data.I'llwalkyouthroughlistswithdata,though,topreventconfusion. Sincelists evaluate as functions, you can't do this:

```
user=>(123)
java.lang.ClassCastException:java.lang.Integer
    cannotbecasttoclojure.lang.IFn(NO_SOURCE_FILE:0)
```

Ifyoureallywantalistmadeupof1,2,and3,youneedtodooneof thesethings, instead:

```
user=>(list123)
(123)
user=>'(123)
(123)
```

Then, you can manipulate the lists as usual. The second form is called quoting. The four main operations for a list are first (thehead), rest (the lost (thelastelement), and cons (constructanew listminusthehead). listgivenaheadandatail):

```
user=>(first'(:r2d2:c3po))
:r2d2
user=>(last'(:r2d2:c3po))
:c3po
user=>(rest'(:r2d2:c3po))
user=>(cons:battle-droid'(:r2d2:c3po))
(:battle-droid:r2d2:c3po)
```

Ofcourse, you can combine these with higher-order functions, but we'll waittodosountilweencountersequences.Now,let'smovetoaclose cousinofthelist.thevector.

Vectors

Likealist.avectorisanorderedcollectionofelements.Vens areoptimizedforrandomaccess. Yelsurroundvectors with square brackets, likethis:

```
user=>[:hutt:wookie:ewok]
[:hutt:wookie:ewok]
```

Use lists for code and vectors for data. You cangetvariouselements likethis:

```
user=>(first[:hutt:wookie:ewok])
user=>(nth[:hutt:wookie:ewok]2)
: ewok
```

```
user=>(nth[:hutt:wookie:ewok]0)
user=>(last[:hutt:wookie:ewok])
user=>([:hutt:wookie:ewok]2)
:ewok
```

Note that vectors are functions, too, taking an index as an argument. Yucancombinetwovectorslikethis:

```
user=>(concat[:darth-vader][:darth-maul])
(:darth-vader:darth-maul)
```

You may have noticed that repl printed out a list instead of a vector. Many of the function sthat return collections use a Clojurea b stractioncalled sequences.We learnmoreabouttheminday2.Fornow,just understandthatClojureisreturningasequenceandrenderingitasa listintherepl.

Clojureletsyougetthetypicalheadandtailofavector,ofcourse:

```
user=>(first[:hutt:wookie:ewok])
user=>(rest[:hutt:wookie:ewok])
(:wookie:ewok)
```

₩ŧ useboth of these features as we do pattern matching. Both lists andvectorsareordered.Let'smoveontosomeoftheunorderedcollections, sets and maps.

Sets

Asetisanunorderedcollectionofelements. The collection has a stable order, but that order is implementation-dependent, so you shouldn't countonit. Yolwrapasetin #{},likethis:

```
user=>#{:x-wing:y-wing:tie-fighter}
#{:x-wing:y-wing:tie-fighter}
```

Wecanassignthosetoavariable called spacecraft and then manipulate them:

```
user=>(defspacecraft#{:x-wing:y-wing:tie-fighter})
#'user/spacecraft
user=>spacecraft
#{:x-wing:y-wing:tie-fighter}
user=>(countspacecraft)
user=>(sortspacecraft)
(:tie-fighter:x-wing:y-wing)
```

We can also create a sorted set that takes items in any order and returnstheminsortedorder:

```
user=>(sorted-set231)
#{123}
```

Yucanmergetwosets, likethis:

```
user=>(clojure.set/union#{:skywalker}#{:vader})
#{:skywalker:vader}
```

Orcomputethedifference:

```
(clojure.set/difference#{123}#{2})
```

I'll give you one last convenient oddity before Imove on. Sets are alsofunctions. Theset #{:jar-jar,:chewbacca} is an element but also a function.Setstestmembership,likethis:

```
user=>(#{:jar-jar:chewbacca}:chewbacca)
:chewbacca
user=>(#{:jar-jar:chewbacca}:luke)
ni1
```

When you use a set as a function, the function will return the first argument if that argument is a member of the set. That covers the basicsforsets.Let'smoveontomaps.

Maps

Asyouknow, amapisakey-valuepair. Using Clojure, you'll represent amapwithcurlybraces, likethis:

```
user=>{:chewie:wookie:lea:human}
{:chewie:wookie,:lea:human}
```

These are examples of maps, key-value pairs, but they are tough to read.Itwouldbehardtoseeanoddnumberofkeysandvalues,leading toanerror:

```
user=>{:jabba:hut:han}
java.lang.ArrayIndexOutOfBoundsException:3
```

Clojure solves this problem by allowing commas as whitespace, like this:

```
user=>{:darth-vader"obiwan",:luke"yoda"}
{:darth-vader"obiwan",:luke"yoda"}
```

Awordprecededwith : isakeyword,likesymbolsinRubyoratomsin PrologorErlang.Clojurehastwokindsofformsthatareusedtoname thingsinthisway, keywords and symbols. Symbols point to something else, and keywords point to themselves. true and mop aresymbols. Use Let'sdefineamapcalled

```
user=>(defmentors{:darth-vader"obiwan",:luke"yoda"})
#'user/mentors
user=>mentors
{:darth-vader"obiwan",:luke"yoda"}
```

Now, you can retrieve a value by passing a key as the first value:

```
user=>(mentors:luke)
"yoda"
```

Mapsarealsofunctions. Keywords are also functions:

```
user=>(:lukementors)
"yoda"
```

:luke,thefunction,looksitselfupinamap.It'soddbutuseful.Aswith Ruby, you can use any data type as a keyor value. And you can merge twomapswith merge:

```
user=>(merge{:y-wing2,:x-wing4}{:tie-fighter2})
{:tie-fighter2,:y-wing2,:x-wing4}
```

You can also specify an operator to use when a hash exists in both maps:

```
user=>(merge-with+{:y-wing2,:x-wing4}{:tie-fighter2:x-wing3})
{:tie-fighter2,:y-wing2,:x-wing7}
```

Inthiscase, we combined the 4 and the 3 values associated with the wing keyswith +. Givenanassociation, you can create an ewas sociation withanewkey-valuepair, likethis:

```
user=>(assoc{:one1}:two2)
{:two2,:one1}
```

You can create a sorted map that takes items in any order and spits themoutsorted.likethis:

```
user=>(sorted-map1:one,3:three,2:two)
{1:one,2:two,3:three}
```

Wate graduallyaddingmorestructuretodata.Now,wecanmoveonto theformthataddsbehavior, the function.

DefiningFunctions

FunctionsarethecenterpieceofallkindsofLisps.Use defn todefinea function.

```
user=>(defnforce-it[](str"Usetheforce,""Luke."))
#'user/force-it
```

The simplest form is (defn [params] body). We defined a function called with no parameters. The function simply concatenated two stringstogether. Callthefunctionlikeanyother:

```
user=>(force-it)
"Usetheforce, Luke."
```

Ifyouwant, you can specify an extrastring describing the function:

```
user=>(defnforce-it
             "ThefirstfunctionayoungJedineeds"
             (str"Usetheforce,""Luke"))
```

Then, you can recall documentation for the function with doc:

```
user=>(docforce-it)
user/force-it
([])
 ThefirstfunctionayoungJedineeds
ni1
```

Let'saddaparameter:

```
user=>(defnforce-it
             "ThefirstfunctionayoungJedineeds"
             [iedi]
             (str"Usetheforce, "jedi))
#'user/force-it
user=>(force-it"Luke")
"Usetheforce, Luke"
```

doc feature on any other function that By the way, you can use this specifiesadocumentationline:

```
user=>(docstr)
_____
cloiure.core/str
([][x][x&ys])
 Withnoargs, returns the emptystring. Withoneargx, returns
 x.toString(). (strnil)returnstheemptystring.Withmorethan
 one arg, returns the concatenation of the strvalues of the args.\\
```

Nowthatyoucandefineabasicfunction,let's move onto the parameter list.

Bindings

Asinmostoftheotherlanguageswe'velookedatsofar,theprocessof assigningparameters based on the inboundarguments is called binding. One of the nice things about Clojure is its ability to access any portion of any argument as a parameter. For example, say you had a line, represented as a vector of points, like this:

```
user=>(defline[[00][1020]])
#'user/line
user=>line
[[00][1020]]
```

Yucouldbuildafunctiontoaccesstheendoftheline.likethis:

```
user=>(defnline-end[ln](last ln))
#'user/line-end
user=>(line-endline)
[1020]
```

Butwedon'treallyneedthewholeline. It would be nicer to bind our $parameter to the second element of the line. \textbf{Wh} Clojure, it \verb"seasy:"$

```
(defnline-end[[ second]]second)
#'user/line-end
user=>(line-endline)
[1020]
```

Theconceptiscalled destructuring.Wie takingadatastructureapart and grabbing only the pieces that are important to us. Let's take a closer lookatthebindings.We have [[_second]]. The outerbrackets define the parametervector. The inner brackets saywe'r egoing to bind to individualelementsofalistorvector. _ and second are individual parameters, butit'sidiomatictouse forparametersyouwanttoignore.InEnglish, we'resaying"Theparametersforthisfunctionare forthefirstelement of the first argument, and second for the second element of the first argument."

We can also nest these bindings. Let's say we have a tic-tac-toe board, andwewanttoreturnthevalueofthemiddlesquare. We represent theboardasthreerowsofthree.likethis:

```
user=>(defboard[[:x:o:x][:o:x:o][:o:x:o]])
#'user/board
```

Now, wewant topic kup the second element of the second row of three, likethis:

```
user=>(defncenter[[_[_c_]_]]c)
#'user/center
```

Beautiful! Water essentially nesting the same concept. Let's break it down. The bindings are [[_ [_c_] _]]. We bind one parameter to the inboundargument, [_[_c_]_].Thatparametersayswe'regoingtoignore the first and third elements, which are the top and bottom rows in our tic-tac-toeboard.We focusonthemiddlerow, which is [_c_]. Wate expectinganotherlistandgrabbingthecenterone:

```
user=>(centerboard)
:x
```

We can simplify that function in a couple of different ways. First, we don't have to list any of the wildcard arguments that come after the targetarguments:

```
(defncenter[[ [ c]]]c)
```

Next, destructuring can happen in the argument list or also in a let statement.InanyLisp,you'lluse let tobindavariabletoavalue. We could use let to hide the destructuring from the clients of the center function:

```
(defncenter[board]
 (let[[_[_c]]board]c))
```

let takestwoarguments. Firstisavectorwith the symbolyouwant to bind([[_ [_c]]])followedbythevalue(board)thatyouwantbound.Next issomeexpressionthatpresumably uses that value (we just returned c). Both forms produce equivalent results. It all depends on where you wantthedestructuringtohappen. I'llshowyouacoupleofbriefexamplesusing let, but understand that you could also use them within an argumentlist.

Yucandestructureamap:

```
user=>(defperson{:name"Jabba":profession"Gangster"})
#'user/person
user=>(let[{name:name}person](str"Theperson'snameis"name))
"Theperson'snameisJabba"
```

Yucanalsocombinemapsandvectors:

```
user=>(defvillains[{:name"Godzilla":size"big"}{:name"Ebola":size"small"}])
#'user/villains
user=>(let[[_{name:name}]villains](str"Nameofthesecondvillain:"name))
"Nameofthesecondvillain: Ebola"
```

We bound to a vector, skipping the first and picking out the name of can see the influence of Lisp on Prolog and, by the second map. You extension, Erlang. Destructuring is simply a form of pattern matching.

AnonymousFunctions

InLisp, functions are just data. Higher-order functions are built into the language from the ground up because code is just like any other kindofdata. Anonymous functions letyoucreate unnamed functions. It's a fundamental capability for every language in this book. In Clojure, you'll define a higher-order function with the fn function. Typically,you'llomitthename,sotheformlookslike (fn [parameters*] body). Let'stakeanexample.

Let'suseahigher-orderfunctiontobuildalistofwordcounts, givena listofwords.Let'ssaywehavealistofpeople'snames:

```
user=>(defpeople["Lea","HanSolo"])
#'user/people
```

We cancompute the length of one word like this:

```
user=>(count"Lea")
3
```

We can build a list of the lengths of the names, like this:

```
user=>(mapcountpeople)
(38)
```

Yorke seen these concepts before. count in this context is a higherorderfunction. In Clojure, that concept is easy because a function is canusethesamebuilding alist, justlike any other list element. You blockstocomputealisthavingtwicethelengthsofpersonnames, like this:

```
user=>(defntwice-count[w](
                              * 2(countw)))
#'user/twice-count
user=>(twice-count"Lando")
user=>(maptwice-countpeople)
(616)
```

Sincethatfunctionissosimple, we can write it as an anonymous function, like this:

```
user=>(map(fn[w](
                     * 2(countw)))people)
(616)
```

We canalsouseashorterform:

```
user=>(map#( * 2(count%))people)
(616)
```

Whtheshortform, #defines an anonymous function, with %bound to each item of the sequence. # iscalleda **rat**rmacro

Anonymous functions give you convenience and leverage to create a function that doesn't need a name right now . You'ce otherlanguages. Hereare some of the functions on collections that use higher-orderfunctions. For all of these functions, we're going to use a commonvector, called v:

```
user=>(defv[312])
#'user/v
```

₩. use that list with several anonymous functions in the following examples.

apply

apply applies a function to an argument list. (applyf'(xy)) workslike (f xy) ,likethis:

```
user=>(apply+v)
user=>(applymaxv)
```

filter

The filter function works like find_all in Ruby. It takes a function that is a test and returns the sequence of elements that pass the test. Forexample, togetonly the oddelements or the elements less than 3, use this:

```
user=>(filterodd?v)
user=>(filter#(<%3)v)
(12)
```

WŁ takeadeeperlookatsomeoftheanonymousfunctionsaswelook deeperintoClojuresequences.Fornow,let'stakeabreakandseewhat RichHickey, creator of Clojure, hastosay.

InterviewwithRichHickey,CreatorofClojure

Rich Hickeyans were dsome questions for the readers of this book. Hehad a special interest in why this Lisp version could be more broadlysuccessfulthanotherLisps, so this interview is a little more extensive thanmost. Ihopeyou find his answers as fascinating as Idid.

Bruce Tate: WhydidyouwriteClojure?

Rich Hickey: I'mjustapractitionerwhowantedapredominantlyfunctional, extensible, dynamic language, with a solid concurrency story, fortheindustry-standardplatformsoftheJVMandCLRanddidn'tfindone.

Bruce Tate: Whatdoyoulikeaboutithemost?

Rich Hickey: Iliketheemphasisonabstractionsinthedatastructures and library and the simplicity. Maybe that's two things, but they are rktel.

Bruce Tate: What feature would you change if you could start over again?

Rich Hickey: I'dexploreadifferentapproachtonumbers.Boxednumbersarecertainly as ore spoton the JVM. This is an area I amactively workingon.

What's the most interesting problem you've seen solved Bruce Tate: withClojure?

Rich Hickey: IthinkFlightcaster ³ (aservicethatpredictsflightdelays in adtime) leverages many aspects of Clojure—from using the syntactic abstraction of macrostomake a DSL for the machine learning and statisticalinferencebits to the Javainter op forworking within frastructure likeHadoopandCascading.

Bruce Tate: ButhowcanClojurebemorebroadlysuccessfulwhenso manyotherdialectsofLisphavefailed?

Rich Hickey: This is an important question! I'm not sure I would characterizethemajordialectsofLisp(SchemeandCommonLisp)ashaving failedattheirmissions. Schemewas an attemptataverysmall language that captured the essentials of computation, while Common Lisp strove tostandardizethemanyvariantdialectsofLispbeingusedin**rach** They have failed to catch on as practical tools for general-purpose production programming by developers in industry, something that neither wasdesignedtobe.

Clojure, on the other hand, is designed to be a practical tool for generalpurposeproductionprogrammingbydevelopersinindustryandassuch adds these additional objectives to the Lisps of old. We workbetterin teams, we play well with other languages, and we solve some traditional Lispproblems.

Bruce Tate: HowdoesClojureworkbetterinateamsetting?

Rich Hickey: ThereisasenseinwhichsomeLispsareallaboutmaximallyempoweringtheindividualdeveloper, but Clojure rayies that

^{3.} http://www.infog.com/articles/flightcaster-clojure-rails

development is a team effort. For example, it doesn't support user-defined **rat** macros, which could lead to code being written in small incompatiblemicro-dialects.

Bruce Tate: Whydidyouchoosetorunonexistingvirtualmachines?

Rich Hickey: The existence of large and valuable code bases written inotherlanguagesisafactoflifetodayandwasn'twhentheolderLisps were invented. The ability to call and be called by other languages is critical, especially on the JVM and CLR.

The whole idea of standard multilanguage platforms abstracting away the host OS barely existed when the older Lisps were invented. The industryisordersofmagnitudelargernow, and defactost and ardshave arisen. Technically, the stratification that supports the **re** of coretechnologieslikesophisticatedgarbagecollectorsanddynamiccompilerslike HotSpot is a good thing. So, Clojure emphasizes language-on-platform ratherthanlanguage-is-platform.

Bruce Tate: Fairenough.ButhowisthisLispanymoreapproachable?

Rich Hickey: Lots of **REDS** For example, we wanted to deal with the parentheses "problem." Lisp programmers know the value of codeis-data, but it is wrong to simply dismiss those who are put off by the parentheses. I don't think moving from (malbaz) to (foobarbaz) is difficult for developers. But I did take a hard look at the parentheses use in the older Lisps to see whether the story could be better, and it could.OlderLispsuseparenthesesforeverything.Wedon't.Andinolder Lisps, there are simply too many parentheses. Clojuretakes the opposite approach, doing away with the grouping parentheses, making its lightly harderformacrowritersbuteasierforusers.

The combination of fewer parentheses and almost no overloading of parentheses was Clojure much easier to scan, visually parse, and understandthanolderLisps.Leadingdoubleparenthesesaremorecommon in Java code, the horrid ((AType)athing).amethod(), than in Clojure code.

WhatWeLearnedinDay 1

ClojureisafunctionallanguageontheJVM.LikeScalaandErlang,this Lispdialectisafunctionallanguagebutnotapurefunctionallanguage. Itallowslimitedsideeffects. Unlikeother Lisp dialects, Clojureaddsa

Microsoft's commonlanguage runtime, a virtual machine for the . NET plat for m

littlesyntax, preferringbraces for maps and brackets for vectors. You canusecommasforwhitespaceandomitparenthesesinsomeplaces.

We learned to use the basic Clojure forms. The simpler forms included booleans, characters, numbers, keywords, and strings. We alsobroke into the various collections. Lists and vectors were ordered containers with vectors optimized for random access and lists optimized for ordered traversal. We also used sets, or unordered collections, and maps, which were key-value pairs.

We defined some named functions, providing a name, parameter list, and function body with an optional documentation string. Next, we used deconstruction with bindings, allowing us to bind any parametertoanypartofaninboundargument. The feature was reminiscent ofPrologorErlang.Finally,wedefinedsomeanonymousfunctionsand used them to iterate over a list with the map function.

On day 2, we'll look at recursion in Clojure, a basic building block in most functional languages. We also look at sequences and lazy evaluations, some corners to ne softhe Clojure models that help layera common, powerful abstraction ontopo f collections.

Now, we'lltakeabreaktoputintopracticewhatyou'velearnedsofar.

Day 1 Self-Study

Clojure is a new language, but you'll still find a surprisingly active, and growing, community. They were among the best that I found as I researchedthisbook.

Find:

- ExamplesusingClojuresequences
- TheformaldefinitionofaClojurefunction
- Ascriptforquicklyinvokingthereplinyourenvironment

Do:

- Implementafunctioncalled (bigstn) thatreturnstrueifastring st islongerthan n characters.
- Weafunction called (collection-typecol) that returns : list, :map, or :vector basedonthetypeofcollection col.

Day2:Y oda andtheForce

As a Jedi master, Yola trained apprentices to use and understand the Force, the unifying presence between all living things. In this section, we get to the concepts fundamental to Clojure. We talkabout sequences, the abstraction layer that unifies all Clojure collections and tiesthemtoJavacollections.We alsolookatlazyevaluation,thejustin-timestrategythatcomputessequencemembersonlywhenyouneed them. And then we'll look at that mystical language feature that is the ForceforallLisps, themacro.

Recursionwithloopandrecur

Asyou'velearnedinotherlanguagesinthisbook,functionallanguages dependonrecursionratherthaniteration. Here's arecursive program toevaluatethesizeofavector:

```
(defnsize[v]
    (if(empty?v)
            0
            (inc(size(restv)))))
(size[123])
```

It's not hard to understand. The size of an empty list is zero; the size ofanotherlistisonemorethanthesizeofthetail.We seensimilar solutions for other languages throughout this book.

Yore also learned that stacks can grow, so recursive algorithms will continue to consume memory until all memory is exhausted. Functional languages work around this limitation with tail recursion optimization. Clojuredoes not support implicit tail recursion optimization because of limitations of the JVM. You must explicitly recur through theuseof loop and recur. Think of a loop as a let statement.

```
(loop[xx-initial-value,yy-initial-value](do-something-withxy))
```

Initially, given a vector, loop binds the variables in the even positions to thevalues in the oddpositions. In fact, if you don't specify a recur, loop worksexactlylikea let:

```
user = > (loop[x1]x)
```

The function recur will invoke the loop again but this time pass new values.

Let's refactor the size function to use recur:

```
(defnsize[v]
        (loop[lv,c0]
        (if(empty?1)
                (recur(restl)(incc))))
```

Inthesecondversionof size, we'lluse the tail-recursion-optimized loop and recur. Since we won't actually return a value, we'll maintain the resultinavariablecalledan accumulator.Inthiscase. c willmaintain acount.

This version works like a tail-optimized call, but we're stuck with more kludgylinesofcode. Sometimes, the JVM is a double-edged sword. If youwantthecommunity, youneed to deal with the problems. But since this function is built into some basic collection APIs, you won't often need to use recur. Also, Clojure gives you some excellent alternatives to recursion, including lazy sequences that we'll get to later in this chapter.

Whday2'sbadnewsoutoftheway,we'refreetoshifttomorepleasantmatters. Sequences will start to take us into some of the features thatmakeClojurespecial.

Sequences

Asequence is an implementation-independent abstraction around all the various containers in the Clojure ecosystem. Sequences wrap all Clojurecollections(sets,maps,vectors,andthelike),strings,andeven file system structures (streams, directories). They also provide a commonabstractionforJavacontainers,includingJavacollections,arrays, and strings. In general, if it supports the functions first, rest, and cons, youcanwrapitinasequence.

Earlier, when you were working with vectors, Clojure sometimes respondedwithalistintheconsolelikethis:

```
user => [123]
[123]
user=>(rest[123])
(23)
```

Noticethatwestartedwithavector. The result is not a list. replactually respondedwithasequence. That means we can treat all collections in a genericway.Let'slookatthecommonsequencelibrary.It'stoorichand powerfultocoverentirelyinonesectionofachapter,butI'lltrytogive

youaflavorofwhat's available. I'm going to cover sequence functions thatchange, test, and create sequences, but I'm going to touch on them onlybriefly.

Tests

When you want to test a sequence, you will use a function called a predicate. These take a sequence and a test function and return a returns true if the test function is true for all items boolean. ee? inthesequence:

```
user=>(every?number?[123:four])
false
```

So.oneoftheitemsisnotanumber. some istrueifthetestistruefor anyoftheitemsinthesequence:

```
(somenil?[12nil])
true
```

Oneoftheitemsis nil. not-every? and not-ony? aretheinverses:

```
user=>(not-every?odd?[135])
false
user=>(not-any?number?[:one:two:three])
```

Thesebehaveexactlyasyouwouldexpect.Let's shift to functions that changesequences.

ChangingaSequence

The sequence library has a number of sequences that transform sequences in various ways. Yore already seen filter. To grab only the wordswithalengthgreaterthanfour, usethis:

```
user=>(defwords["luke""chewie""han""lando"])
#'user/words
user=>(filter (fn[word](>(countword)4))words)
("chewie""lando")
```

Andyou'vealsoseen mp ,whichcallsafunctiononalltheitemsina collectionandreturnstheresults. Yucanbuildasequenceofsquares ofallitemsinavector:

```
user = > (map(fn[x]( * xx))[11235])
(114925)
```

^{5.} Moreprecisely, some returns the first value that is not nil or the .Forexample, (some first [[] [1]]) returns 1.

The list comprehension combines maps and filters, as you saw in ErlangandScala.Recallthatalistcomprehensioncombinesmultiplelists and filters, taking the possible combinations of the lists and applying thefilters. First, let's take a simple case. We have a couple of lists, colors and toys:

```
user=>(defcolors["red""blue"])
#'user/colors
user=>(deftoys["block""car"])
#'user/toys
```

We can apply a function to all the colors with a list comprehension, similar to the way amapworks:

```
user=>(for[xcolors](str"Ilike"x))
("Ilikered""Ilikeblue")
```

[xcolors] binds x toanelementfromthecolorslist. (str"llike"x) isanarbitraryfunctionthat's applied to every x from colors. It gets more interestingwhenyoubindtomorethanonelist:

```
user=>(for[xcolors,ytoys](str"Ilike"x""y"s"))
("Ilikeredblocks""Ilikeredcars"
"Ilikeblueblocks""Ilikebluecars")
```

The comprehension created every possible combination from the two lists.We canalsoapplyafilterwiththe :when keywordinthebindings:

```
user=>(defnsmall-word?[w](<(countw)4))</pre>
#'user/small-word?
user=>(for[xcolors,ytoys,:when(small-word?y)]
            (str"Ilike"x""y"s"))
("Ilikeredcars""Ilikebluecars")
```

Wewroteafiltercalled small-word?. Anywordthatisless than four charapplied the small-word? filter to y with :when (smallactersissmall. We wordy) .We gotalloossibilities of (x,y), where x is a member of colors, y isamember of toys, and the size of y is less than four characters. The codeis dense but expressive. That's an ideal combination. Let's move on.

Yweseen fold . folderf ,and inject in Erlang, Scala, and Ruby. In Lisp, theequivalentis reduce. To compute a quick total or factorial, use this:

```
user=>(reduce+[1234])
user=>(reduce * [12345])
120
```

Yucansortalist:

```
user=>(sort[3124])
(1234)
```

and sort on the result of a function:

```
user = > (defnabs[x](if(< x0)(-x)x))
#'user/abs
user=>(sort-byabs[-1-432])
(-123-4)
```

We define a function called a compute an absolute value and then use that function in our sort. These are some of the most important sequence transformation functions in Clojure. Next, we'll move on to functions that creates equences, but to do that, you're going to have to getalittlelazier.

LazyEvaluation

Inmathematics, infinite sequences of numbers are often easier to describe.Infunctionallanguages,you'doftenliketohavethesamebenefits, butyoucan't actually compute an infinite sequence. The answer islazyevaluation. Using this strategy, Clojure's sequence library computesvalues only when they are actually consumed. In fact, most sequences are lazy. Let's walk through creating some finite sequences andmoveintolazysequences.

FiniteSequenceswithrange

UnlikeRuby, Clojure supports ranges as functions. Arange creates a sequence:

```
user=>(range110)
(123456789)
```

Note that the upper bound is not inclusive. The sequence did not include 10. Yucan specify any increment:

```
user=>(range1103)
(147)
```

Yudon'thavetospecifythelowerboundifthereisnoincrement:

```
user=>(range10)
(0123456789)
```

Zeroisthedefaultlowerbound. These quences created with range are finite. What if you wanted to create a sequence with no upperbound? Thatwouldbeaninfinitesequence.Let's find outhow.

InfiniteSequencesandtake

Let's start with the most basic of infinite sequences, an infinite sequence of one repeated element. We can specify (repeat 1). If you try itintherepl, you'll get 1 suntil you kill the process. Clearly, we need somewaytograbonlyafinitesubset. That function is take:

```
user=>(take3(repeat"UsetheForce,Luke"))
("UsetheForce, Luke""UsetheForce, Luke""UsetheForce, Luke")
```

So, we created an infinite sequence of the repeated string "UsetheForce. Luke", and then we took the first three. Yucan also repeat the elements inalistwith ovde :

```
user=>(take5(cycle[:lather:rinse:repeat]))
(:lather:rinse:repeat:lather:rinse)
```

takingthefirstfiveelementsofthecyclefromthevector Wate [:lather :rinse:repeat]. Fair enough. We can drop the first few elements of a sequenceaswell:

```
user=>(take5(drop2(cycle[:lather:rinse:repeat])))
(:repeat:lather:rinse:repeat:lather)
```

from the inside out, we again build acycle, drop the first two, Waking andtakethefirstfiveafterthat.Butyoudon'thavetoworkinsideout. ->>)toapplyeachfunction Yucanusethenewleft-to-rightoperator(toaresult:

```
user=>(->>[:lather:rinse:repeat](cycle)(drop2)(take5))
(:repeat:lather:rinse:repeat:lather)
```

So, wetakeavector, build a sequence with cycle, drop 2, and then take 5. Sometimes, left-to-right code is easier to read. What if you wanted toaddsomekindofseparatorbetweenwords?Ynduse interpose:

```
user=>(take5(interpose:and(cycle[:lather:rinse:repeat])))
(:lather:and:rinse:and:repeat)
```

Water takingthekeyword : and and placing it between all the elements ofaninfinitesequence. Think of this function like ageneralized version of Ruby's join. What if you wanted an interpose that took interposing membersfromasequence?That's interleave:

```
user=>(take20(interleave(cycle(range2))(cycle(range3))))
(00110210011200110210)
```

Water interleaving two infinite sequences, (cycle (range 2)) and (cycle (range3)). Then, we take the first 20. Asyour eadther esult, even numbersare (0101010101) , and odd numbers are (0120120120) Beautiful.

The iterate function provides another way of creating sequences. Check outtheseexamples:

```
user=>(take5(iterateinc1))
(12345)
user=>(take5(iteratedec0))
(0-1-2-3-4)
```

iterate takes a function and a starting value. iterate then applies the function to the starting value repeatedly. In these two examples, we called inc and dec.

Here's an example that calculates consecutive pairs in the Fibonacci sequence. Remember, each number of a sequence is the sum of the lasttwo. Given a pair, [ab], we can generate the next with [b,a+b]. We can generate an anonymous function to generate one pair, given the previousvalue, like this:

```
user=>(defnfib-pair[[ab]] [b(+ab)])
#'user/fib-pair
user=>(fib-pair[35])
[58]
```

Next, we'll use iterate to build an infinite sequence. Don't execute this yet:

```
(iteratefib-pair[11])
```

use mp tograbthefirstelementfromallofthosepairs:

```
(map
first
(iteratefib-pair[11]))
```

That's an infinite sequence. Now, we can take the first 5:

```
user=>(take5
       (map
       first
       (iteratefib-pair[11])))
(11235)
```

Orwecangrabthenumberatindex500,likethis:

```
(nth(mapfirst(iteratefib-pair[11]))500)
(225...morenumbers...626)
```

The performance is excellent. Using lazy sequences, you can often describerecursiveproblemslikeFibonacci.Factorialisanotherexample:

```
user=>(defnfactorial[n](apply * (taken(iterateinc1))))
#'user/factorial
user=>(factorial5)
120
```

We grab n elements from the infinite sequence (iterateincl). Then we takenofthemandmultiplythemtogetherwith apply *. The solution is deadsimple. Nowthat we've spent sometime with lazy sequences, it's timetoexplorenewClojurefunctionscalled defrecord and protocol.

defrecordandprotocols

So far, we've talked about Java integration at a high level, but you haven't seen much of the JVM bleed throught othe Clojure language. Whenallissaidanddone, the JVM is about types and interfaces. (For you non-Java programmers, think of types as Java classes. Think of interfaces as Java classes without implementations.) To make ClojureintegratewellwiththeJVM, theoriginal implementation has a significantamountofJavainit.

AsClojurepickedupmorespeedandbegantoproveitselfasaneffective JVM language, there was a strong thrust to implement more of ClojureinClojureitself.Todoso, Clojuredevelopersneededawayto build platform-fast open extensions by programming to abstractions $rather than implementations. The {\it results} \, are$ defrecord for types and protocol, which groups functions together around a type. From a Clo $jure perspective, the best parts of OO are types and protocols (such as {\tt Normal perspective}) and {\tt Normal perspective}) are the {\tt Normal perspective} and {\tt Normal perspective}) are the {\tt Normal perspective} and {\tt Normal perspective} are the {\tt Normal perspective} and {\tt Normal perspective} are the {\tt Normal perspective} and {\tt Normal perspective} are the {\tt Normal perspective} are the {\tt Normal perspective} and {\tt Normal perspective} are the {\tt Normal$ interfaces), and the worst parts are implementation inheritance. Clojure's defrecord and protocol preserve the good parts of OO and leave therest.

As this book is being written, these language features are important, but they are evolving. I'm going to lean hard on Stuart Halloway, cofounder of Relevance and author of Programming Clojure [Hal09], to helpwalkthroughapracticalimplementation. We goingtogoback to another functional language on the JVM, Scala. We rewrite the CompassprograminClojure.Let'sgetstarted.

First, we'll define a protocol. A Clojure protocol islikeacontract. Types ofthisprotocolwillsupportaspecificsetoffunctions, fields, and arguments. Here's a protocol describing a Compass:

```
Download clojure/compass.clj
(defprotocolCompass
  (direction[c])
  (left[c])
  (right[c]))
```

This protocol defines an abstraction called Composs and enumerates thefunctionsthata Composs must support—direction, left, and right with

thespecified number of arguments. We arenowfreetoimplementthe protocolwith defrecord.Next,we'llneedthefourdirections:

```
(defdirections[:north:east:south:west])
```

needafunctiontohandleaturn.Recallthatourbasedirectionis aninteger, 0, 1, 2, and 3 represent :north, :east, :south, and :west, respectively. Every 1 youaddtothebasewill move the compass ninety degrees to the right. We take the remainder of the base/4 (more precisely, base/number-of-directions) so that we'll wrap around correctly from :west to :north.likethis:

```
(defnturn
  [baseamount]
  (rem(+baseamount)(countdirections)))
```

Theturnworks, justasyou'd expect. I'lload the compass file and then usethe turn functions:

```
user=>(turn11)
user=>(turn31)
user=>(turn23)
```

Saidanotherway,turningrightoncefrom :east givesyou :south,turning rightoncefrom :west givesyou :north,andturningrightthreetimesfrom :south gives you :east.

It's time to implement the protocol. We dothatwith defrecord.We do defrecord todeclarewe'reimplementingthe thatinpieces.First,weuse protocol, like this:

```
(defrecordSimpleCompass[bearing]
 Compass
```

Water defininganewrecordcalled SimpleCompass.Ithasonefieldcalled bearing. Next, we will implement the Compass protocol, beginning with the direction function:

```
(direction[_](directionsbearing))
```

The direction function looks up the element of directions at the bearing index.Forexample, (directions3) returns :west.Eachargumentlisthas areference to the instance (e.g., self inRubyor this inJava), butwe're notusingit, soweadd _ toourargumentlist.Next,ontoleftandright:

```
(left[_](SimpleCompass.(turnbearing3)))
(right[_](SimpleCompass.(turnbearing1)))
```

Remember, in Clojure, we're using immutable values. That means that turningwillreturnanew, modified compass rather than changing the left and right usesyntaxthatyouhave existingcompassinplace.Both not seen before. (Some Type. arg) means fire the constructor for Simple-Compass, binding arg to the first parameter. Yucanverify that entering (String. "newstring") intothereplreturnsthenewstring "newstring".

So, the left and right functions are easy. Each returns a new compass withtheappropriatebearing, configured for the newbearing, using the turn function we defined earlier. right turns right ninety degrees once, and left turnsrightninetydegreesthreetimes. Sofar, we have a type SimpleCompass that implements the Compass protocol. We just need a functionthat returns a string representation, but toString isamethod on java.lang.Object.That'seasyenoughtoaddtoourtype.

```
Object
(toString[this](str
                   "[" (directionthis) "]")))
```

We thenimplementpartofthe Object protocolwiththe toString method, returningastringthatlookslike SimpleCompass [:north].

Now, the type is complete. Create an ewcompass:

```
user=>(defc(SimpleCompass.0))
#'user/c
```

Turnsreturnanewcompass:

```
user=>(leftc);returnsanewcompass
#:SimpleCompass{:bearing3}
user=>c;originalcompassunchanged
#:SimpleCompass{:bearing0}
```

Noticethattheoldcompassisunchanged.Sincewe'redefiningaJVM type, you can access all fields as Java fields. But you can also access thefieldsinthetypeasClojuremapkeywords:

```
user=>(:bearingc)
0
```

Becausethesetypesworklikemaps, you can easily prototypenew types asmapsanditerativelyconvertthemtotypesasyourdesignstabilizes. Yu can also replace types as maps in your tests as stubs or mocks. Thereareotherbenefitsaswell:

 TypesplaynicewithotherClojureconcurrencyconstructs.Inday 3, we'lllearnhowtocreatemutablereferences of Clojure objects inwaysthatmaintaintransactionalintegrity, much likerelational databasesdo.

• We implemented a protocol, butyou'renot limited to the new way of doing things. Since we're building JVM types, the types can interoperate with Java classes and interfaces.

Wth defrecord and protocol, Clojure offers the ability to build native codefortheJVM, without Java. This code can fully interact withother types on the JVM, including Java classes or interfaces. You canuse themtosubclassJavatypesorimplementinterfaces.Javaclassescan alsobuildonyourClojuretypes.Ofcourse,thisisnottheentireJava interop story, but it's an important part. Now that you've learned to extendJava, let's learn how to extend the Clojure language itself with macros.

Macros

Forthissection, we'regoing to refer to the lochapter. We implemented the Ruby unless in Section 3.3, Messages, on page 79. The form is (unless test family . The function will execute (m) if the test is the . We can't simplydesignafunction, because each parameter will execute:

```
user=>;Brokenunless
user=>(defnunless[testbody](if(nottest)body))
#'user/unless
user=>(unlesstrue(println"Danger,dangerWillRobinson"))
Danger, dangerWillRobinson
nil
```

We discussed this problem in Io. Most languages execute parameters first and then put the results on the call stack. In this case, we don't wanttoevaluatetheblockunlesstheconditionisfalse.InIo.thelanguagecircumventedthisproblembydelayingtheexecutionofthe unless message. In Lisp, we can use macros. When we type (unlesstestbody), wewantLisptotranslatethatto (if(nottest)body) .Macrostotherescue.

AClojureprogramexecutes in two stages. Macroexpansion translates allmacrosinthelanguagetotheirexpandedform. Yucanseewhat's happeningwithacommandcalled magepand .Wore alreadyuseda coupleofmacros, called **radr**macros .Asemicolon(:)isacomment.a singlequotemark(')isaquote,andanumbersign(#)isananonymous function. To prevent premature execution, we'll put a quote infront of the expression we want to expand:

```
user=>(macroexpand''something-we-do-not-want-interpreted)
(quotesomething-we-do-not-want-interpreted)
```

```
user=>(macroexpand'#(count%))
(fn* [p1_97](countp1_97))
```

These are macros. In general, macro expansion will let you treat code likelists. If you don't want a function to execute right away, quote it. Clojurewillreplacetheargumentsintact.Our unless willlooklikethis:

```
user=>(defmacrounless[testbody]
        (list'if(list'nottest)body))
#'user/unless
```

Note that Clojure substitutes test and body without evaluating them, if and not. We also have to package the minlists. butwehavetoquote

building a list of code in the exact form that Clojure will execute.Water

```
We can more point
                         it:
```

```
user=>(macroexpand'(unlessconditionbody))
(if(notcondition)body)
```

Andwecanexecuteit:

```
user=>(unlesstrue(println"Nomoredanger,Will."))
user=>(unlessfalse(println"Now,THISisTheFORCE."))
Now.THISisTheFORCE.
ni1
```

What we've done is change the base definition of the language. We are adding our own control structure, without requiring the language designerstoaddtheirownkeywords. Macroexpansionisperhapsthe mostpowerfulfeatureofLisp, and fewlanguages candoit. The secret sauceistheexpression of data as code, not just a string. The code is alreadyinahigher-orderdatastructure.

Let'swrapupday2. There's alot of meathere. We shouldpausetouse whatwe'velearned.

WhatWeLearnedinDay2

It's been another packed day. Yorke added a tremendous set of abstractionstoyourexpandingbagoftricks.Let'sreview.

First, welearned to use recursion. Since the JVM doesn't support tailrecursionoptimization, we had to use loop and recur. That looping construct allowed us to implement many algorithms you would usually implement with recursive function calls, though the syntax was more invasive.

We also used sequences. Whathem, Clojure encapsulates access to allofits collections. Wha common library, we could apply common strategies for dealing with collections. We used different functions to mutate, transform, and search sequences. Higher-order functions addedpowerandsimplicity to the sequence libraries.

Wh lazy sequences, we were able to add another powerful layer to sequences.Lazysequencessimplifiedalgorithms.Theyalsooffereddelayed execution, potentially significantly improving performance and looseningcoupling.

Next, we spent sometime implementing types. Wh defrecord and protocols, we were able to implement types that were full citizens on the JVM.

Finally, we used macros to add features to the language. We learned thatthereisastep, called macroexpansion, that occurs before Clojure implements or interprets code. We implemented unless by using an if functionwithinmacroexpansion.

There's alottodigest. Takesometime to use what you've learned.

Day2Self-Study

This day was packed with some of the most sophisticated and powerful elements of the Clojure language. Take some time to explore and understandthosefeatures.

Find:

- ${\color{gray}\bullet} The implementation of some of the commonly used macrosin the$ Clojurelanguage
- Anexampleofdefiningyourownlazysequence
- Thecurrentstatusofthe defrecord and protocol features(thesefeatureswerestillunderdevelopmentasthisbookwasbeingdeveloped)

Do:

- Implementan unless withan else conditionusing macros.
- Weatypeusing defrecord that implements a protocol.

7.4 Day3:AnEyetrEvil

In Star Was . Yorla was the first to detect the evil in Darth Volr WhClojure, RichHickeyhasidentified the core problems that plague the development of concurrent object-oriented systems. When saidfrequently that mutable state is the evilthat lurks in the hearts of objectoriented programs. We shown several different approaches to handlingmutablestate. Ioand Scalaused the actor-based model and provided immutable constructs that gave the programmer the power to solve those problems without mutable state. Erlang provided actors with lightweight processes, and a virtual machine that allowed effective monitoring and communication, allowing unprecedented reliability. The Clojure approach to concurrency is different. It uses software transactionalmemory (STM).Inthissection,we'lllookatSTMandalso severaltoolstosharestateacrossthreadedapplications.

References and **Insubti**Memory

Databasesusetransactionstoensuredataintegrity.Moderndatabases useatleasttwotypesofconcurrencycontrol.Lockspreventtwocompeting transactions from accessing the same row at the same time. uses multiple versions to allow each transaction to have a privatecopyofitsdata. Ifanytransactioninterferes with another, the databaseenginesimplyrerunsthattransaction.

LanguageslikeJavauselockingtoprotecttheresourcesofonethread from competing threads that might corrupt them. Locking basically puts the burden of concurrency control on the programmer. We are rapidlylearningthatthisburdenistoomuchtobear.

LanguageslikeClojureuse softwaretransactionalmemory (STM). This strategy uses multiple versions to maintain consistency and integrity. UnlikeScala, Ruby, or Io, when you want to change the state of a referenceinClojure, youmust do so within the scope of a transaction. Let's seehowitworks.

References

In Cloiure, a ref (short for reference) is a wrapped piece of data. All access must conform to specified rules. In this case, the rules are to supportSTM.Yucannotchangeareferenceoutsideofatransaction. Toseehowitworks.let'screateareference:

```
user=>(ref"AttackoftheClones")
#<Ref@ffdadcd: "AttackoftheClones">
```

```
user=>(defmovie(ref"StarWars"))
#'user/movie
```

Yucangetthereferenceback, likethis:

```
user=>movie
#<Ref@579d75ee: "StarWars">
```

Butwe'rereallyworried about the value in the reference. Use deref:

```
user=>(derefmovie)
"StarWars"
```

Or, you can use the short form of deref:

```
user=>@movie
"StarWars"
```

That's better. Now, we can easily access the value within our reference. We haven'ttried to change the state of the reference yet. Let's try. Wh

Clojure, we'll send a function that will mutate the value. The dereferencedrefwillbepassedasthefirstargumentofthefunction:

```
user=>(altermoviestr":TheEmpireStrikesBack")
java.lang.IllegalStateException:Notransactionrunning(NO_SOURCE_FILE:0)
```

Asyoucansee, you can mutate state only within a transaction. Do so with the dosync function. The preferred way to modify a reference is to alteritwithsometransformingfunction, likethis:

```
user=>(dosync(altermoviestr":TheEmpireStrikesBack"))
"StarWars: The Empire Strikes Back"
```

We could also sets omeinitial value with ref-set:

```
user=>(dosync(ref-setmovie"StarWars:TheRevengeoftheSith"))
"StarWars: The Revenge of the Sith"
```

Yucanseethatthereferencechanged:

```
user=>@movie
"StarWars: The Revenge of the Sith"
```

That's what we expected. Therefore needs different. It may see mpainful tomodifymutablevariablesinthisway, but Clojureisen forcingalittlepolicynowtosavealotofpainlater. We knowthatprogramsthat behave in this manner will absolutely execute correctly, with respect torace conditions and deadlock. Most of our code will use functional paradigms, and we'll save STM for the problems that could be nefit the mostfrommutability.

WtigwithAtoms

If you want thread safety for a single reference, uncoordinated with anyotheractivity, then you can use atoms. These data elements allow changeoutsidethecontextofatransaction.Likeareference,aClojure otom isanencapsulatedbitofstate.Let'stryit.Createanatom:

```
user=>(atom"Splitatyourownrisk.")
#<Atom@53f64158:"Splitatyourownrisk.">
Now.bindanatom:
user=>(defdanger(atom"Splitatyourownrisk."))
#'user/danger
user=>danger
#<Atom@3a56860b:"Splitatyourownrisk.">
user=>@danger
"Splitatyourownrisk."
Yucanbind
              danger to a new value with
                                          reset!:
user=>(reset!danger"Splitwithimpunity")
"Splitwithimpunity"
user=>danger
#<Atom@455fc40c:"Splitwithimpunity">
user=>@danger
"Splitwithimpunity"
```

reset! replaces the entireatom, but the preferred usage is to provide a function to transform the atom. If you're changing a large vector, you canmodifyanatominplacewith swap! likethis:

```
user=>(deftop-sellers(atom[]))
#'user/top-sellers
user=>(swap!top-sellersconj{:title"SevenLanguages",:author"Tate"})
[{:title"SevenLanguagesinSevenWeeks",:author"Tate"}]
user=>(swap!top-sellersconj{:title"ProgrammingClojure":author"Halloway"})
[{:title"SevenLanguagesinSevenWeeks",:author"Tate"}
{:title"ProgrammingClojure",:author"Halloway"}]
```

Aswithareference, you'll want to create a value once and then change thatvaluewith swap!.Let'slookatapracticalexample.

BuildinganAtomCache

Now, you've seen both references and atoms. Yall see the same generalphilosophywhenweworkwithHaskell.Yuwrapabitofstatein apackage that you can later mutate with functions. While references required transactions, atoms do not. Let's build a simple atom cache. It'saperfectproblemforanatom.We simplyusehashestoassociate

names with values. This example is provided courtesy of Stuart HallowayofRelevance, ⁶ aconsultancythatprovidesClojuretrainingand consulting.

need to create the cache, and then we'll need functions to add WŁ elementstothecacheandremoveelementsfromthecache. First, we'll createthecache:

```
Download clojure/atomcache.clj
(defncreate
  (atom{}))
```

simplycreatinganatom. We lettheclientofthisclassbindit. Next, we need to be able to get a cache key:

```
(definget
  [cachekey]
  (@cachekey))
```

We takethecacheandakeyasarguments. The cacheisan atom, so wedereferenceitandreturntheitemassociated with the key. Finally, weneedtoputaniteminthecache:

```
(defnput
  ([cachevalue-map]
     (swap!cachemergevalue-map))
  ([cachekeyvalue]
     (swap!cacheassockeyvalue)))
```

We defined two different functions called put. The first version uses merge to allow us to add all of the associations in a map to our cache. Thesecondversionuses assoc toaddakeyandvalue.Here'sthecache inuse.We addanitemtothecacheandthenreturnit:

```
(defac(create))
              "I'myourfather,Luke."
(putac:quote
(println(str "Cacheditem:" (getac:quote)))
```

Andtheoutput:

Cacheditem: I'myourfather, Luke.

Atomsandrefsaresimpleandsafewaystohandlemutablestate,synchronously. In the next few sections, we'll look at a couple of asynchronousexamples.

^{6.} http://www.thinkrelevance.com

Likean otom, an agentisawrapped piece of data. Likean I of uture, the stateofadereferencedagentwillblockuntilavalueisavailable.Users canmutatethedataasynchronouslyusingfunctions, and the updates willoccurinanotherthread.Onlyonefunctioncanmutatethestateof anagentatatime.

twice that doubles the value Giveitatry.Let'sdefineafunctioncalled ofwhateveryoupassin:

```
user=>(defntwice[x](
                        *2x))
#'user/twice
```

Next,we'lldefineanagentcalled tribbles that has an initial value of one:

```
user=>(deftribbles(agent1))
#'user/tribbles
```

Now, we can mutate tribbles by sending the agent avalue:

```
user=>(sendtribblestwice)
#<Agent@554d7745:1>
```

This function will runina nother thread. Let's get the value of the agent:

```
user=>@tribbles
```

Reading a value from a ref, agent, or atom will never lock and never block. Reads should be fast, and with the right abstractions around them, they can be. Whith is function, you can see the difference in the valuesthatyoureadfromeachagent:

```
user=>(defnslow-twice[x]
         (do
             (Thread/sleep5000)
             (*2x))
#'user/slow-twice
user=>@tribbles
user=>(sendtribblesslow-twice)
#<Agent@554d7745:16>
user=>@tribbles
user=>;dothisfivesecondslater
user=>@tribbles
```

Don't get hung up in the syntax. (Thread/sleep 5000) simply invokes Java's sleep methodon Thread. Fornow, focus on the value of the agent.

We defined a slower version of twice thattookfive seconds. That was enoughtimetoseethedifferencesin @tribbles overtimeintherepl.

a value of tribbles. Yumightnot get the latest changes So, you will get from your own thread. If you want to be sure to get the latest value with **reset** to your own thread , you can call (await tribbles) or (awaittimeout tribbles), where timeout is a timeout in milliseconds. Keep in mind that await and await-for block only until actions from your thread are dispatched. This says nothing about what other threads mayhaveaskedthethreadtodo.Ifyouthinkyouwantthelatestvalue of something, you have already failed. Clojure's tools involve working with a snapshot whose value is instantaneous and potentially out-ofdateimmediately. That 's exactly how versioning databases work for fast concurrencycontrol.

Futures

InJava, youwould start threads directly to solve a specific task. Certainly, you can use Javaintegration to start athread in this way, but there's often abetter way. Sayyou wanted to create a thread to handle acomplexcomputationaroundabitofencapsulated state. Yucould usean agent. Or say you wanted to start the computation of a value, butyoudidnotwanttoawaittheresult. As with Io, you could use a future.Let'stakealook.

First, let's create a future. The future returns a reference immediately:

```
user=>(deffiner-things(future(Thread/sleep5000)"taketime"))
#'user/finer-things
user=>@finer-things
"taketime"
```

Depending on how fast you type, you may have had to wait for the result. A future takes a body of one or more expressions, returning the valueofthelastexpression. The future starts in another thread. If you dereferenceit, the future will block until the value becomes available.

So, a future is a concurrency construct that allows an asynchronousreturn before computation is complete. We can use futures to allow severallong-runningfunctionstoruninparallel.

What Wee Missed

ClojureisaLisp, which is an extremely richlanguage in its own right. It's based on the JVM, which has more than a decade of development. Thelanguagealsomixesinsomenewandpowerfulconcepts. It would

beimpossibletocoverClojureinonechapterofabook.Therearesome piecesthatyoushouldknowabout.

Metadata

Sometimes, it's nice to associate metadata to a type. Clojure allows you to attach and access metadata on both symbols and collections. (with-metavale metadata) gives you a new volue associated with the metadata, usually implemented as a map.

Jaintegration

ClojurehasexcellentJavaintegration.We touchedonJavaintegration veryloosely, and we also built a type on the JVM. We didnotusethe existingJavalibrariesatall.We alsodidnotextensivelycovertheJava compatibility forms. For example, (.toUpperCase "Fred") calls the .toUpperCase memberfunctiononthestring "Fred".

Multimethods

Object-oriented languages allow one style of organization for behavior anddata. Clojureallowsyoutobuildyourowncodeorganization with multimethods. Yucanassociatealibrary of functions with a type. Yu can also implement polymorphism by using multimethod stodomethoddispatchbasedontype, metadata, arguments, and even attributes. The concept is powerful and extremely flexible. Yu could implement, for example, Java-style inheritance, prototype inheritance, or something entirelydifferent.

ThreadState

Clojureoffersatoms, refs, and agents for various concurrency models. Sometimes, you need to stored at a perthread in stance. Clojure allows youtodosoguitesimplywith vas .Forexample, (binding [name"value"] ...) wouldbind name to "value" only for the current thread

WhatWeLearnedinDay3

Today, wewalkedthroughtheconcurrencystructures. Ween countered severalinteresting concurrency constructs along the way.

Refsallowedustoimplementmutablestatewhilemaintainingconsistency across threads. We used STM, or software transactional memory. For our part, we placed all mutations to refs within transactions, expressedusinga dosync function.

Next, we used atoms, lightweight concurrency constructs with less protection but a simpler usage model. We modifiedanatomoutsideofa transaction.

Finally, we used agents to implement a pool that could be used to do long-running computations. Agents were different from Io actors, becausewecouldmutatethevalueoftheagentwithanarbitraryfunction. Agents also returned a snapshot in time, a value that may be changedatanytime.

Day3Self-Study

Onday2, your focus was on advanced programming abstractions. Day 3 brought the concurrency constructs of Clojure. In these exercises, you'llputsomeofwhatyou'velearnedtothetest.

Find:

 Aqueueimplementationthatblockswhenthequeueisemptyand waitsforanewiteminthequeue

Do:

• Use refs to create a vector of accounts in memory. Create debit and credit functions to change the balance of an account.

In this section, I'm going to outline a single problem called sleeping barber. Itwascreated by Edsger Dijkstrain 1965. It has the secharacteristics:

- Abarbershoptakescustomers.
- Customersarriveatrandomintervals, fromtentothirtymilliseconds.
- Thebarbershophasthreechairsinthewaitingroom.
- Thebarbershophasonebarberandonebarberchair.
- When the barber's chair is empty, a customer sits in the chair, wakesupthebarber, and getsahaircut.
- If the chairs are occupied, all new customers will turn away.
- Haircutstaketwentymilliseconds.
- Afteracustomerreceivesahaircut, hegetsupandleaves.

Weamultithreaded program to determine how many hair cuts abarbercangiveintenseconds.

WrappingUpClojure 7.5

Clojure combines the power of a Lisp dialect with the convenience of the JVM. From the JVM, Clojure benefits from the existing community, deployment platform, and codelibraries. As a Lisp dialect, Clojure comeswith the corresponding strengths and limitations.

TheLispRootx

Clojureisperhapsthemostpowerfulandflexiblelanguageinthisbook. Multimethods allow multiparadigm code, and macros let you redefine the language on the fly. No other language in this book provides this powerful combination. That flexibility has proven to be an incredible strength.In Hackers and Painters, Graham chronicles a start-up that leveraged productivity with Lisp to achieve productivity that no other vendors could match. Some emerging consultancies are taking the same approach, betting that Clojure will provide a productivity and qualityadvantagethatotherlanguagessimplycannotmatch.

Lisp'sflexibilitycanalsobeaweakness.Macroexpansionisapowerful featureinthehandsofanexpertbutwillleadtounmitigateddisaster without the proper thought and care. The same ability to effortlessly apply many powerful abstractions in a few lines of code makes Lisp especiallydemandingforallbutthemostskilledprogrammers.

To successfully evaluate Clojure, you need to look at Lisp but also theother unique aspects of the Java ecosystem and the new unique features.Let'stakeadeeperlookatthefundamentalstrengthsofClojure.

CoreStrengths

Clojureisone of a handful of languages vying for the position as the next great popular language on the Java virtual machine. There are manyreasonsthatitisapowerfulcandidate.

AGoodLisp

Tim Bray, programming language expertand superblogger, called ClojureagoodLispinapostcalled "ElevenThesesonClojure." ⁷ Infact. hecallsClojure"thebestLispever."IwouldagreethatClojureisavery goodLisp.

^{7.} http://www.tbray.org/ongoing/When/200x/2009/12/01/Clojure-Theses

Inthischapter, you saw Rich Hickey's discussion on what makes ClojuresuchagoodLisp:

- Reducedparentheses. Clojureimprovesreadabilitybyopeningup thesyntaxalittle, including brackets for vectors, braces formaps, and a combination of characters for sets.
- Theecosystem. Lisp'smanydialectswaterdownthesupportand librarysetthatyoucanhaveforanysingledialect.Ironically, havingonemoredialectcanhelpsolvethatproblem.Bybeingonthe JVM, Clojure can take advantage of Java programmers who are lookingformoreandthefabuloussetoflibraries.
- Restraint. ByexercisingrestraintandlimitingClojure'ssyntaxto avoid reader macros, Hickey effectively limited Clojure's power butalso decreased the likelihood that harmful splinter dialects mightemerge.

YumightappreciateLispasaprogramminglanguageinitsownright. Bythatmeasure, you can look at Clojure purely as a new Lisp. On that level.itsucceeds.

Concurrency

Clojure's approach to concurrency has the potential to change the way wedesignconcurrentsystemscompletely.STMdoesplacesomeadditionalburdenondevelopersbecauseofitsnovelty, butforthefirsttime, it protects developers by detecting whether state mutations happen within appropriately protected functions. If you're not within a transaction, you can't mutate state.

Jaintegration

ClojurehasgreatintegrationwithJava.Itusessomenativetypessuch asstringsandnumberstransparentlyandofferstypehintsforperformance.ButClojureshinesbyallowingtightJVMintegration,soClojure types can fully participate in Java applications. Yil soonseemuch more of Clojure itself implemented within the JVM.

LazyEvaluation

Clojure adds powerful lazy evaluation features. Lazy evaluation can simplifyproblems. Yuhaveseenonlyatasteofhowlazysequencescan shapethewayyouattackaproblem.Lazysequencescanreducecomputation overhead significantly by delaying execution until it is actuallyneededorbypreventingexecutionaltogether. Finally, lazyproblem

solving offers just one more tool to solve difficult problems. Yu can often use lazy sequences to replace recursion, iteration, or realized collections.

DataasCode

Programs are lists. As with any Lisp, you can represent data as code. withRubyhashelpedmeseethevalueofwritingprogramsin programs. Ithinkthis is the most important capability of any programminglanguage.Functionalprogramsallowmetaprogrammingthrough higher-orderfunctions.Lispextendsthisideathroughevaluatingdata ascode.

Core Wednesses

Clojureisalanguagethat's firmly targeted as a general-purpose programminglanguage. Whether it can actually be broadly successful on the JVM is yet to be determined. Clojure has wonderful abstractions butmanyofthem. Totrulyembrace and use those features effectively andsafely, aprogrammer will need to be highly educated and extremely talented. Hereare some of my concerns.

PrefixNotation

 $Representing code in list form is one of the most powerful features in {\tt the continuous} and {\tt the continuous}$ ⁸ Typicalobject-oriented anyLisp, butthere is a cost—prefix notation. languageshaveawildlydifferentsyntax. Theadjustmenttoprefixnotationisnoteasy. It requires a better memory and requires a developer tocomprehendcodefromtheinsideout, rather than outside in. Sometimes, I find that reading Clojure pushes metoward understanding too muchdetailtoosoon.Atbest,Lispsyntaxpushesmyshort-termmemory. Whexperience, I'm told this improves. I've not yet turned that corner.

Readability

Anothercosttodataascodeistheoppressivenumberofparentheses. Optimizing for people and computers is not at all the same thing. The location and number of parentheses is still a problem. Lisp developers leanheavilyontheireditorstoprovidethefeedbackformatchingparentheses, but tools can never fully mask readability problems. Kudos toRichforimprovingthisproblem, butitwill still beaproblem.

^{8.} Clojure does have left-to-right macros, ->> and ->, which mitigate these problems a little.

LearningCurve

Clojureisrich, and the learning curve is oppressive. You need to have an extremely talented and experienced team to make Lispwork. Lazy sequences, functional programming, macro expansion, transactional memory, and the sophistication of the approaches are all powerful conceptsthattaketimetomaster.

LimitedLisp

Allcompromises have some cost. By being on the JVM, Clojure limits tail-recursion optimization. Clojure programmers must use the awkwardrecursyntax.Try implementing (sizex) that computes the size of a sequence x withrecursionandwith loop/recur.

The elimination of user-defined reader macros is also significant. The benefitisclear.Readermacros, when abused, can lead to the splinteringofthelanguage. The cost, too, is clear. Yulose on emore metaprogrammingtool.

Accessibility

One of the most beautiful aspects of Rubyoranearly Javaisits accessibilityasaprogramminglanguage.Bothofthoselanguageswererelativelyeasytopickup. Clojureplacestremendous demandsonadeveloper.Ithassomanyabstractiontoolsandconceptsthattheresultcan beoverwhelming.

FinalThoughts

Most of Clojure's strengths and weaknesses are related to the power and flexibility. True, you might work hard to learn Clojure. In fact, if you'reaJavadeveloper, you'realreadyworkinghard. Yn ejustspendingyour time on Java application-level abstractions. You arelooking forloosercouplingthroughSpringoraspect-orientedprogramming,for example. Yoie just not getting the full benefits of additional flexibility at the language level. For many, that trade-off has worked. I will humblysuggestthatthenewdemandsofconcurrencyandcomplexity will continue to make the Java platform less and less via ble.

Ifyouneedanextremeprogrammingmodelandarewillingtopaythe price of learning the language, Clojure is a great fit. Ithink this is a greatlanguagefordisciplined,educatedteamslookingforleverage.Yu canbuildbettersoftwarefasterwithClojure.

Chapter 8 **Haskell**

Haskell represents purity and freedom for many functional programming purists. It's rich and powerful, but the power comes at a price. You can't eat just a couple of bites. Haskell will force you to eat the whole functional programming burrito. Think Spock from Star Trek. The quote above 1 is typical, embracing logic and truth. His characterhasasingle-minded purity that has endeared him to generations. Where Scala, Erlang, and Clojure let you use imperative concepts in smalldoses, Haskellleaves no such wiggleroom. This pure functional language will challenge you when it's time to do I/O or accumulate state.

8.1 IntroducingHaskell

As always, to understand why a language embraces a certain set of compromises, you should start with the history. In the early and mid-1980s, pure functional programming splintered across several languages. Thekeyconcepts driving new research were lazy processing, as weencounteredinClojure, and pure functional programming. Agroup from the Functional Programming Languages and Computer Architectureconferencein 1987 formed and decided to build an open standard forapurefunctionallanguage.Outofthatgroup,Haskellwasbornin 1990andrevisedagainin 1998. The current standard, called Haskell 98, has been revised several times, including a revision of Haskell 98 and the definition of a new version of Haskell called Haskell Prime.

^{1.} StarTrek: TheOriginalSeries ,Episodes41and42: "I,Mudd"/ "TheTrouble withTribbles."DirectedbyMarcDaniels.1967;Burbank,CA:20thCBSParamountInternational Television, 2001.

So, Haskellwasbuilt from the ground up to be a pure functional language,combiningideasfromthebestfunctionallanguages,withspecial emphasisonlazyprocessing.

Haskellhasstrong, statictyping, likeScala. The type model is mostly inferred and is widely considered to be one of the most effective type systems of any functional language. Yol see that the type system allowsforpolymorphismandverycleandesigns.

Haskellalsosupportsotherconceptsyou'veseeninthisbook. Haskell allows Erlang-style pattern matching and guards. Yhlalsofind Clojure-style lazy evaluation and list comprehensions from both Clojure andErlang.

Asapurefunctionallanguage, Haskelldoesnotdosideeffects. Instead, a function can return a side effect, which is later executed. Yel see anexampleofthisinday3, as well as an example of preserving state usingaconceptcalled monads.

The first couple of days will get youthrough typical functional programming concepts, such as expressions, defining functions, higher-order functions, and the like. Walsoget into Haskell's typing model, which willgiveyousomenewconcepts.Day3willstretchyou.Wlookatthe parameterizedtypesystemandmonads, which are sometimes difficult conceptstograsp.Let'sgetstarted.

8.2 Day1:Logical

LikeSpock, you'll find that Haskell's core concepts are easy to grasp. work strictly with defining functions. Given the same input pa-Ydi rameters, you'll get the same output parameters, every time. I'm going touseGHC,ortheGlasgowHaskellCompiler,version6.12.1.It'swidely available across many platforms, but you can find other implementationsaswell. Asalways, I'mgoingtostartintheconsole. Type ghci:

```
GHCi,version6.12.1:http://www.haskell.org/ghc/ :?forhelp
Loadingpackageghc-prim...linking...done.
Loadingpackageinteger-gmp...linking...done.
Loadingpackagebase...linking...done.
Loadingpackageffi-1.0...linking...done.
```

Yoli see Haskell load a few packages, and then you're ready to type commands.

Expressions and Primitive Types

Wate going to talk about Haskell's type system a little later. In this section, we'll focus on using primitive types. As with many of the other languages, we'll start with numbers and some simple expressions. We movequicklyintomoreadvancedtypessuchasfunctions.

Numbers

Bynow, youknow the drill. Type a few expressions:

```
Prelude>4
Prelude>4+1
Prelude>4+1.0
5.0
Prelude>4+2.0 * 5
14.0
```

Orderofoperationsworksjustaboutlikeyou'dexpect:

```
Prelude>4 * 5+1
Prelude>4 * (5+1)
```

Noticeyoucangroupoperations with parentheses. Yorkeseen acouple oftypesofnumbers.Let'slookatsomecharacterdata.

CharacterData

Stringsarerepresented with double quotes, like this:

```
Prelude>"hello"
"hello"
Prelude>"hello"+"world"
<interactive>:1:0:
   Noinstancefor(Num[Char])
      arisingfromauseof`+'at<interactive>:1:0-17
    Possiblefix:addaninstancedeclarationfor(Num[Char])
    Intheexpression:"hello"+"world"
    Inthedefinitionof`it':it="hello"+"world"
Prelude>"hello"++"world"
"helloworld"
```

Noticethatyou'llconcatenatewith ++ instead of +.Yucanrepresent singlecharacterslikethis:

```
Prelude>'a'
'a'
Prelude>['a','b']
"ab"
```

Noticethatastringisjustalistofcharacters.Let'sbrieflylookatsome booleanvalues.

Booleans

Abooleanisanotherprimitivetypethatworksmuchastheydoinmost oftheotherinfixnotationlanguagesinthisbook. These are equal and not-equalexpressions, returning booleans:

```
Prelude>(4+5)==9
True
Prelude>(5+5)/=10
False
Try an if/then statement:
Prelude>if(5==5)then"true"
<interactive>:1:23:parseerror(possiblyincorrectindentation)
```

That's the first major departure from other languages in the book. In Haskell, indentation is significant. Haskell is guessing that there's a follow-up line that is not indented correctly. We see some indented structureslater. Wewon't talk about layouts, which control indentation patterns; followpredictable indentation strategies that mimic what you seehere,andyouwillbeOK.Let'sdoafull if/then/else statement:

```
Prelude>if(5==5)then"true"else"false"
"true"
```

InHaskell, if isafunction, not a control structure, meaning it returns avaluejustlikeanyotherfunction.Let'stryafewtrue/falsevalues:

```
Prelude>if1then"true"else"false"
<interactive>:1:3:
    Noinstancefor(NumBool)
      arisingfromtheliteral`1'at<interactive>:1:3
```

Haskellisstronglytyped. if takesstrictlybooleantypes.Let'strytoforce anothertypecollision:

```
Prelude>"one"+1
<interactive>:1:0:
    Noinstancefor(Num[Char])
      arisingfromauseof`+'at<interactive>:1:0-8
```

ThiserrormessagegivesusthefirstglimpseintoHaskell'stypesystem. It says "There is no function called + thattakesa Num argumentfol-

lowedby [Char], alist of characters." Notice that we haven't told Haskell whattypesthingsare. The language is inferring types based on clues. Atanypoint, you can see what Haskell's type inference is doing. You canuse :t,oryoucanturnonthe :t optionthatdoessomethingsimilar, likethis:

```
Prelude>:set+t
Prelude>5
it::Integer
Prelude>5.0
5.0
it::Double
Prelude>"hello"
"hello"
it::[Char]
Prelude>(5==(2+3))
True
it::Bool
```

Now, after every expression, you can see the type that each expression returns.Letmewarnyouthatusing :t withnumbersisconfusing.That hastodowiththeinterplaybetweennumbersandtheconsole. Try to usethe :t function:

```
Prelude>:t5
5::(Numt)=>t
```

That is not the same as the type we got before, it::Integer .Theconsole will try to treat numbers as generically as possible, unless you havedonea :sett.Ratherthanapure type, you geta class, which is a description of abunch of similar types. We learnmoreinSection8.4, Classes, onpage 299.

Functions

The centerpiece of the whole Haskell programming paradigm is the function. Since Haskell has strong, static typing, you'll specify each functionintwoparts:anoptionaltypespecificationandtheimplementation.We goingtogoquicklythroughconceptsyou'veseeninother languages, so hangon tight.

DefiningBasicFunctions

AHaskellfunctiontraditionallyhastwoparts:thetypedeclarationand thefunctiondeclaration.

Initially, we're going to be defining functions within the console. We usethe let functiontobindvaluestoimplementations. Beforedefining let binds a variable to a a function, try let. As with Lisp, in Haskell, functioninalocalscope.

```
Prelude>letx=10
Prelude>x
10
```

Whenyou'recodingaHaskellmodule,you'lldeclarefunctionslikethis:

```
doublex=x
             * 2
```

Intheconsole, though, we'lluse let toassignthefunctioninlocalscope, sowecanuseit. Here's an example of a simple double function:

```
* 2
Prelude>letdoublex=x
Prelude>double2
```

At this point, we'll switch to using files with programs. We can then work with multiline definitions. Using GHC, the full double definition wouldlooklikethis:

```
Download haskell/double.hs
moduleMainwhere
    doublex=x+x
```

Noticethatweaddeda module called Main.InHaskell.modulescollect Main moduleisspecial. Itisthe related code into a similar scope. The top-levelmodule.Focusonthe double functionfornow.Load Main into theconsole.anduseitlikethis:

```
Prelude>:loaddouble.hs
[1of1]CompilingMain
                                      (double.hs, interpreted)
Ok, modules loaded: Main.
*Main>double5
10
```

Sofar, we haven't enforced a type. Haskellisbeing for giving by inferring a type for us. There's definitely an underlying type definition for each function. Here's an example of a definition with a type definition:

```
Download haskell/double_with_type.hs
moduleMainwhere
    double::Integer->Integer
    doublex=x+x
```

Andwecanloaditanduseitasbefore:

```
[1of1]CompilingMain
                                      (double_with_type.hs,interpreted)
Ok, modules loaded: Main.
*Main>double5
10
```

Yucanseetheassociated type of the new function:

```
*Main>:tdouble
double::Integer->Integer
```

This definition means that the function double takes an Integer argument(thefirst Integer)andreturnsan Integer.

This type definition is limited. If you went back to the earlier, typeless version of double, you'd seesomething elseentirely:

```
*Main>:tdouble
double::(Numa)=>a->a
```

Now, that's different! In this case, a isatypevariable. The definition means "The function double takes a single argument of some type а andreturnsavalueofthatsametype a."Whithisimproved definition, we can use this function with any type that supports the Let's start to crank up the power. Let's look a timplementing something slightlymoreinteresting, afactorial.

Recursion

Let's start with a little recursion. Here's are cursive one-liner that implementsafactorial within the console:

```
Prelude>letfactx=ifx==0thenlelsefact(x-1)
                                                           * X
Prelude>fact3
```

That's a start. The factorial of x is 1 if x is 0, and it's $tot(x-1)^*x$ otherwise. We can do a little better by introducing pattern matching. Actually, this syntaxlooks and acts a lot like Erlang's pattern matching:

```
Download haskell/factorial.hs
moduleMainwhere
    factorial::Integer->Integer
    factorial0=1
    factorialx=x
                  * factorial(x-1)
```

The definition has three lines. The first declares the type of the argumentandreturnvalue. The next two are different functional definitions that depend on the pattern match of the inbound arguments. fitti

.That defiof 0 is 1, and total of n is factorial $x=x^*total(x-1)$ nitionlooks exactly like the mathematical definition. In this case, the orderofthepatternsisimportant. Haskellwilltakethefirstmatch. If youwantedtoreversetheorder, you'd have to use a guard. In Haskell, guardsareconditionsthatrestrictthevalueofthearguments, likethis:

```
Download haskell/fact_with_guard.hs
moduleMainwhere
    factorial::Integer->Integer
    factorialx
        |x>1=x
                * factorial(x-1)
         |otherwise=1
```

Inthiscase, the guards have boolean values on the left and the functiontoapplyontheright. When aguard is satisfied, Haskell calls the appropriatefunction. Guards often replace pattern matching, and we're using it to initiate the base condition for our recursion.

Tutes and Lists

Asyou've seen in other languages, Haskell depends on tail-recursion optimization to efficiently deal with recursion. Let's see several versions ofaFibonaccisequencewithHaskell.First,we'llseeasimplecase:

```
Download haskell/fib.hs
moduleMainwhere
    fib::Integer->Integer
    fib0=1
    fih1=1
    fibx=fib(x-1)+fib(x-2)
```

That's simple enough. fib0 or fib1 is 1, and fibx is fib(x-1)+fib(x-2) .But. that solution is in efficient. Let's build a more efficient solution.

Programming with Littles

We canusetuplestoprovideamoreefficientimplementation. Atuple isacollectionofafixednumberofitems. Tuplesin Haskellarecommaseparated items in parentheses. This implementation creates a tuple with consecutive Fibonacci numbers and uses a counter to assist in recursion. Here's the bases olution:

```
fibTuple::(Integer,Integer,Integer)->(Integer,Integer,Integer)
fibTuple(x,y,0)=(x,y,0)
fibTuple(x,y,index)=fibTuple(y,x+y,index-1)
```

 ${\it fib} {\it Tuple takes a three-tuple} and {\it returns a three-tuple}. Becare {\it full here.}$ Asingleparameterthatisathree-tupleisnotthesameastakingthree

parameters. To use the function, we'll start recursion with two numbers, 0 and 1. We will also provide a counter. As the counter counts down, the first two numbers get successively larger numbers in the sequence.Successivecallsto fibTuple(0,1,4) wouldlooklikethis:

- fibTuple(0,1,4)
- fibTuple(1,1,3)
- fibTuple(1,2,2)
- fibTuple(2,3,1)
- fibTuple(3,5,0)

Yucanruntheprogram, likethis:

```
Prelude>:loadfib_tuple.hs
[1of1]CompilingMain
                                      (fib_tuple.hs,interpreted)
Ok, modules loaded: Main.
*Main>fibTuple(0,1,4)
(3,5,0)
```

The answer will be in the first position. We can grab the answer like this:

```
fibResult::(Integer,Integer,Integer)->Integer
fibResult(x,y,z)=x
```

We justusepatternmatchingtograbthefirstposition.We cansimplify theusagemodellikethis:

```
fib::Integer->Integer
fibx=fibResult(fibTuple(0,1,x))
```

That function uses the two helper functions to build a quite fast Fibonaccigenerator. Hereisthewhole program to gether:

```
Download haskell/fib_tuple.hs
moduleMainwhere
    fibTuple::(Integer,Integer,Integer)->(Integer,Integer,Integer)
    fibTuple(x,y,0)=(x,y,0)
    fibTuple(x,y,index)=fibTuple(y,x+y,index-1)
    fibResult::(Integer,Integer,Integer)->Integer
    fibResult(x,y,z)=x
    fib::Integer->Integer
    fibx=fibResult(fibTuple(0,1,x))
```

Andherearetheresults(whichappearinstantaneously):

```
*Main>fib100
354224848179261915075
*Main>fib1000
   43466557686937456435688527675040625802564660517371780
    40248172908953655541794905189040387984007925516929592
   25930803226347752096896232398733224711616429964409065
    33187938298969649928516003704476137795166849228875
```

Let'stryanotherapproachwithfunctioncomposition.

Using **Inter**and Composition

Sometimes, you need to combine functions by chaining them to gether by passing the results of one function to another. Here's an example thatcomputes the secondite mofal is they matching the head of the tail ofalist:

```
*Main>letsecond=head.tail
*Main>second[1,2]
*Main>second[3,4,5]
```

Wejustdefiningafunctionintheconsole. second=head.tail isequiv-.Water feedingtheresultofonefunction alentto second(st=head(tail(st)) into another. Let's use this feature with yet another Fibonacci sequence.

computeasinglepair, as before, but without a counter:

```
fibNextPair::(Integer,Integer)->(Integer,Integer)
fibNextPair(x,y)=(y,x+y)
```

Giventwonumbers in the sequence, we can always compute the next one. The next job is to recursively compute the next item in the sequence:

```
fibNthPair::Integer->(Integer,Integer)
fibNthPair1=(1,1)
fibNthPairn=fibNextPair(fibNthPair(n-1))
```

Thebasecaseisthevalue (1,1) foran n of 1. From there, it is simple. We just compute the next item of the sequence based on the last one.

We cangetanypair in the sequence:

```
*Main>fibNthPair(8)
(21.34)
*Main>fibNthPair(9)
(34,55)
*Main>fibNthPair(10)
(55,89)
```

Now, all that remains is to match the first item of each pair and combine theminto a sequence. We use a convenient function composition of fst tograbthefirstelementand fibNthPair tobuildapair:

```
Download haskell/fib_pair.hs
moduleMainwhere
    fibNextPair::(Integer,Integer)->(Integer,Integer)
    fibNextPair(x,y)=(y,x+y)
    fibNthPair::Integer->(Integer,Integer)
    fibNthPair1=(1,1)
    fibNthPairn=fibNextPair(fibNthPair(n-1))
    fib::Integer->Integer
    fib=fst.fibNthPair
```

Saidanotherway, wetakethe first element of then thtuple. And we're done. Whalittleworkdone for tuples, let's solve a few problems with lists.

Totaling Lists

Yorke seen lists in many different languages. I'm not going to fully rehash them, but I will go over a basic recursion example and then introduceafewfunctionsyouhaven'tseenyet.Breakingalistintothe headandtailcanworkinanybinding,likea let statementorapattern match:

```
let(h:t)=[1,2,3,4]
*Main>h
*Main>t
[2,3,4]
```

Water bindingthelist [1,2,3,4] to (h:t). Think of this construct as the various head|ta|| constructs you've seen in Prolog, Erlang, and Scala. Whithis tool, we can do a few simple recursive definitions. Here are size and prod functionsforalist:

```
Download haskell/lists.hs
moduleMainwhere
    size[]=0
    size(h:t)=1+sizet
    prod[]=1
    prod(h:t)=h * prodt
```

I'm going to use Haskell's type inference to handle the types of these functions.buttheintentionisclear.Thesizeofalistis1+thesizeofa tail.

```
Prelude>:loadlists.hs
[1of1]CompilingMain
    (lists.hs,interpreted)
Ok.modulesloaded:Main.
*Main>size"Fascinating."
```

zip isapowerfulwaytocombinelists.Here'sthefunctioninaction:

```
*Main>zip"kirk""spock"
[('kirk','spock')]
```

So, webuilt a tuple of the two items. Yucanalsoziplists to gether, like this:

```
Prelude>zip["kirk", "spock"]["enterprise", "reliant"]
[("kirk", "enterprise"), ("spock", "reliant")]
```

It'saneffectivewaytocombinetwolists.

Sofar, the feature syou've seen in Haskellhave been remarkably similar tothosecoveredinotherlanguages.Now,we'llstartworkingwithsome moreadvancedconstructs. Włookatadvancedlist sincluding ranges andlistcomprehensions.

GeneratingLists

We alreadylooked at a few ways to process lists with recursion. In this section, we'll look at a few options for generating new lists. In particular, we'lllookatrecursion, ranges, and list comprehensions.

Recursion

The most basic building block for list construction is the : operator, which combines a head and tail to make a list. We seen the operator in reverse used in pattern matching as we call a recursive function.

```
Here's: ontheleftsideofa
                             let:
Prelude>leth:t=[1,2,3]
Prelude>h
```

Prelude>t [2,3]

We can also use: to do construction, instead of deconstruction.

Here'showthatmightlook:

```
Prelude>1:[2,3]
[1,2,3]
```

Remember, lists are homogeneous. Yucan't addalist to a list of integers, for example:

```
Prelude>[1]:[2,3]
<interactive>:1:8:
    Noinstancefor(Num[t])
      arisingfromtheliteral`3'at<interactive>:1:8
```

Yucould, however, addalist to a list of lists or even an empty list:

```
Prelude>[1]:[[2],[3,4]]
[[1],[2],[3,4]]
Prelude>[1]:[]
[[1]]
```

Here's list construction in action. Let's say we wanted to create a functionthatreturns the even numbers from a list. One way to write that functioniswithlistconstruction:

```
Download haskell/all_even.hs
moduleMainwhere
    allEven::[Integer]->[Integer]
    allEven[]=[]
    allEven(h:t)=ifevenhthenh:allEventelseallEvent
```

Our function takes a list of integers and returns a list of even integers. allEven for an emptylist is an emptylist. If there is a list, if the head is even, weadd the head to allEven applied to the tail. If the head is odd, wediscarditbyapplying allEven to the tail. No problem. Let's look at someotherwaystobuildlists.

RangesandComposition

AswithRubyandScala, Haskellincludesfirst-classranges and some syntactic sugarto support them. Haskell provides a simple form, consistingoftheendpointsofarange:

```
Prelude>[1..2]
[1,2]
Prelude>[1..4]
[1,2,3,4]
```

Yuspecifytheendpoints, and Haskell computes the range. The default incrementis1. Whatif Haskell can't reach the endpoint with the default increment?

```
Prelude>[10..4]
```

Yalgetanemptylist. Yucanspecifyanincrement by specifying the nextiteminthelist:

```
Prelude>[10.8..4]
[10,8,6,4]
```

Yucanalsoworkinfractionalnumbers:

```
Prelude>[10,9.5..4]
[10.0,9.5,9.0,8.5,8.0,7.5,7.0,6.5,6.0,5.5,5.0,4.5,4.0]
```

Rangesaresyntacticsugarforcreatingsequences. These quences need notbebound. As with Clojure, you can take some of the elements of a sequence:

```
Prelude>take5[1..]
[1,2,3,4,5]
Prelude>take5[0,2..]
[0,2,4,6,8]
```

talk more about lazy sequence in day 2. For now, let's look at anotherwaytoautomaticallygeneratelists, the list comprehension.

ListComprehensions

WefirstlookedatlistcomprehensionsintheErlangchapter.InHaskell, alist comprehension works the same way. On the left side, you'll see an expression. On the right side, you'll see generators and filters, just asyoudidwithErlang.Let'slookatafewexamples.To doubleallitems inalist.wedothis:

```
Prelude>[x * 2|x<-[1,2,3]]
[2,4,6]
```

InEnglish, the list comprehension means "Collect x*2 where x is taken fromthelist [1,2,3]."

AswithErlang, we can also use pattern matching within our list comprehensions. Say we had a list of points representing a polygon and wantedtoflipthepolygondiagonally. We couldjusttranspose x and y, likethis:

```
Prelude>[(y,x)|(x,y)<-[(1,2),(2,3),(3,1)]]
[(2,1),(3,2),(1,3)]
```

x from4.likethis: Or,toflipthepolygonhorizontally,wecouldsubtract

```
Prelude>[(4-x,y)|(x,y)<-[(1,2),(2,3),(3,1)]]
[(3,2),(2,3),(1,1)]
```

We can also compute combinations. Let's say we wanted to find all of thepossiblelandingpartiesoftwotakenfromacrewofKirk,Spock,or McCov:

```
Prelude>letcrew=["Kirk", "Spock", "McCoy"]
Prelude>[(a,b)|a<-crew,b<-crew]
[("Kirk","Kirk"),("Kirk","Spock"),("Kirk","McCoy"),
("Spock", "Kirk"), ("Spock", "Spock"), ("Spock", "McCoy"),
("McCoy", "Kirk"), ("McCoy", "Spock"), ("McCoy", "McCoy")]
```

That compositional most worked but did not removed uplicates. We can addconditionstofilterthelistcomprehensionlikethis:

```
Prelude>[(a,b)|a<-crew,b<-crew,a/=b]
[("Kirk", "Spock"), ("Kirk", "McCoy"), ("Spock", "Kirk"),
("Spock", "McCoy"), ("McCoy", "Kirk"), ("McCoy", "Spock")]
```

Thatisalittlebetter.butorderdoesn'tmatter.We candoalittlebetter by including only the options that appear in sorted order, discarding therest:

```
Prelude>[(a,b)|a<-crew,b<-crew,a<b]
[("Kirk", "Spock"), ("Kirk", "McCoy"), ("McCoy", "Spock")]
```

Whashort, simple list comprehension, we have the answer. List comprehensionsareagreattoolforrapidlybuildingandtransforminglists.

AnInterviewwithPhilipWcder

Now that you've seen some of the core features of Haskell, let's see whatsomeone from the committee that designed Haskellhastosay. A theoretical computers cience professor at the University of Edinburgh,Philip Wader isanactivecontributorofnotonlyHaskellbutalsoJava andXQuery.Previously,heworkedorstudiedatAvayaLabs,BellLabs, Glasgow, Chalmers, Oxford, CMU, Xerox Parc, and Stanford.

Bruce Tate: WhydidyourteamcreateHaskell?

Philip Wadler: Inthelate 1980s the rewereal argenumber of different groups creating designs and implementations of functional languages, andwerdedwewouldbestrongerworkingtogetherthanapart.The original goals were not modest: we wanted the language to be a foundationformathsuitable forteaching, and up to industrial uses. The *entirehistoryiscoveredindetailinapaperwewrotefortheHistoryof* ProgrammingLanguagesconference, which you can find on the Wab

^{2.} http://www.haskell.ora/haskellwiki/History of Haskell

Bruce Tate: Whatarethethingsyoulikeaboutithemost?

Philip Wadler: Induction of the property of It's nice to see that they've finally made their way into other languages, likePython.

Typeclassesprovideasimpleformofgenericprogramming. You definea derived, you can get routines datatype, and just by adding one keyword, tocomparevalues, toconvertualues to and from strings, and so on. I find that very convenient and missit when I'm using other languages.

Any good programming language **xl**y becomes a means of extending itselftoembedotherprogramminglanguages specialized to the task at hand. Haskell is particularly good as a tool for embedding other languages.Laziness,lambdaexpressions,monadandarrownotation,type classes, the expressive type system, and template Haskell all support extendingthelanguageinvariousways.

Bruce Tate: What are the things you'd change if you had it to do all overagain?

Philip Wadler: With distribution becoming so important, we need to focus on programs that run on multiple machines, sending values from onetotheother. When you send a value, you probably want it to be the valueitself(eagerevaluation), rather than a program (and the values of all the free variables of the program) that can be evaluated to yield the value.So, inthe distributed world, Ithinkit would be better to be eager bydefaultbutmakeiteasytobelazywhenyouwant.

Bruce Tate: What's the most interesting problem you've seen solved withHaskell?

Philip Wadler: I'm always blown away by the uses folks find for Haskell. I member years ago being amazed at uses of Haskell for natural-language processing and years after that when it was used for protein folding with an application to fighting AIDS. I just had a look at the Haskell Community page, and it lists for tyind ustrial applications of Haskell.Therearenowmanyusersinfinance:ABNAmro,CreditSuisse, DeutscheBank, and Standard Chartered. Facebookuses Haskell for an in-house tool to update code in PHP. One of my favorites is the use of Haskellforgarbage collection—not the kind we do in software but **rd** garbagecollection...programmingenginestobeusedingarbagetrucks!

What We Learned in Day 1

Haskellisafunctionalprogramminglanguage. Its first distinguishing characteristicisthatitisapurefunctionallanguage. Afunction with the same arguments will always produce the same result. There are no spentmostofday1coveringfeaturesyouhaveseenin sideeffects.We otherlanguagesinthisbook.

We first covered basic expressions and simple data types. Since there arenomutablevariableassignments, we used recursion to define some simple math functions and to deal with lists. We worked with basic Haskellexpressions and rolled those up into functions. We sawpattern matchingandguardsaswefoundinErlangandScala.We usedlists and tuples as the basic collections as you found in Erlang.

Finally, we took alook at building lists that took us into list comprehensions, ranges, and even lazy sequences. Let's put some of those ideas intopractice.

Day 1 Self-Study

By this time, writing functional programs should be getting easier if you've been working through all of the other functional languages. In this section, I'm going to push you a little harder.

Find:

- TheHaskellwiki
- AHaskellonlinegroupsupportingyourcompilerofchoice

Do:

- Howmanydifferentwayscanyoufindtowrite allEven?
- We a function that takes a list and returns the same list in reverse.
- Weafunction that builds two-tuples with all possible combinationsoftwoofthecolorsblack, white, blue, yellow, and red. Note thatyoushouldincludeonlyoneof (black,blue) and (blue,black).
- We a list comprehension to build a childhood multiplication table. The table would be a list of three-tuples where the first two are integers from 1-12 and the third is the product of the first two.
- Solve the map-coloring problem (Section 4.2, Map Coloring, on page 101) using Haskell.

Day2:Spock'sGreatStrength 8.3

Wh some characters, you might not notice their best qualities for quite some time. Wh Spock, it's easy to grasp his great strengths. He'sbrilliant, always logical, and completely predictable. Haskell's great strengthisalsothatpredictabilityandsimplicityoflogic.ManyuniversitiesteachHaskellinthecontextofreasoningaboutprograms.Haskell makescreating proofs for correctness far easier than imperative counterparts.Inthissection, we'll digin to the practical concepts that lead tobetterpredictability.We willstartwithhigher-orderfunctions. Then, we'lltalkaboutHaskell'sstrategyforcombiningthem.Thatwilltakeus into partially applied functions and currying. We finallylookatlazy computation.It'sgoingtobeafullday,solet'sgetstarted.

Higher-OrderFunctions

Every language in this book addresses the idea of higher-order programming. Haskell depends on the concept extensively. We quickly through an onymous functions and then apply them with the many prebuilt functions that work on lists. I will move much faster thanIhavewiththeotherlanguages, because you've seen the concepts before, and there's so much ground to cover. We start things with anonymous functions.

AnonymousFunctions

Asyou might expect, anonymous functions in Haskell have a ridiculously simple syntax. The form is (\param1..paramn->function_body) . Try it, like this:

```
Prelude>(x->x)"Logical."
"Logical."
Prelude>(\x->x++"captain.")"Logical,"
"Logical, captain."
```

Taken alone, they don't add much. Combined with other functions, they become extremely powerful.

mapandwhere

First, webuilt an anonymous function that just returns the first parameter. Next, we append a string. As you've seen in other languages, anonymous functions are an important feature for list libraries. Haskell hasamap:

```
* x)[1,2,3]
map(\x->x
```

applying the map function to an anonymous function and a list. Water mp applies the anonymous function to each item in the list and collectstheresults. There's no surprise here, but that form might be abit muchtodigestallatonce.We canpackageitallupasafunctionand break out the anonymous function as a locally scoped function, like this:

```
Download haskell/map.hs
moduleMainwhere
    squareAlllist=mapsquarelist
         where squarex=x
```

Wife declaredafunctioncalled squareAll thattakesaparametercalled list.Next, we use more to apply a function called square toalltheitemsin list.Then,weuseanewfeature,called where, to declare a local version of square. Youdon't have to bind functions with where; you can also bind where throughout the rest of anyvariable. We seesomeexamplesof thechapter. Here's the result:

```
*Main>:loadmap.hs
[1of1]CompilingMain
                                      (map.hs,interpreted)
Ok, modules loaded: Main.
*Main>squareAll[1,2,3]
[1,4,9]
```

Yucanalsouse mp withpartofafunction, calleda section, like this:

```
Prelude>map(+1)[1,2,3]
[2,3,4]
```

(+1) is actually a partially applied function. The + function takes two parameters, and we've supplied only one. The result is that we get a functionlike (x+1) ,withasingleparameter

filter, **add**

The next common function is filter, which applies a test to items in a list.likethis:

```
Prelude>odd5
Prelude>filterodd[1,2,3,4,5]
[1,3,5]
```

Yu canalsofoldleftandright, justasyoudidin Clojure and Scala. Thefunctionsyouwillusearevariationsof бd and tod :

```
Prelude>fold1(\xcarry0ver->carry0ver+x)0[1..10]
55
```

We tookaninitialcarry-overvalueofOandthenappliedthefunctionto everyiteminthelist, using the result of the function as the carryOver argumentandeachitemofthelistastheother. Another formoffold is convenientwhenyouarefoldingwithanoperator:

```
Prelude>foldl1(+)[1..3]
```

This is using the + operator as a pure function taking two parameters andreturninganinteger. The result gives you the samething a sevaluatingthis:

```
Prelude>1+2+3
```

Yucanalsofoldrighttoleft, with бЫ

Asyoumightimagine, Haskelloffersmanyotherfunctions in the library of list functions, and many of them use higher-order functions. Rather than spend a whole chapter on dealing with them, I'll let you do your own discovery. Now, I want to move on to the ways Haskell combinesfunctionstoworktogether.

Rth/AppliedFunctions and Currying

talked briefly about function composition and partially applied Wore functions. These concepts are important and central enough to Haskell thatweshouldspendalittlemoretimehere.

Everyfunctionin Haskellhasone parameter. Yumightaskyourself, "If that's true, how could you write a function like + that adds two numberstogether?"

Infact, it is true. Every function does have one parameter. To simplify thetypesyntax,let'screateafunctioncalled prod:

```
Prelude>letprodxy=x
Prelude>prod34
12
```

We created a function, and you can see that it works. Let's get the type ofthefunction:

```
Prelude>:tprod
prod::(Numa)=>a->a
```

Theportion Numa=> means "Inthefollowing type definition," a isatype of Num."Yn ie seen the rest before, and lied to you about the meaning tosimplifythings.Now,it'stimetosettherecordstraight.Haskelluses

a concept to split one function on multiple arguments into multiple functions, each with one argument. Haskell does this job with partial application.

Don't let the term confuse you. Partial application binds some of the arguments, but not all. For example, we can partially apply prod to createsomeotherfunctions:

```
Prelude>letdouble=prod2
Prelude>lettriple=prod3
```

Lookattheleftsideofthesefunctionsfirst. We defined prod with two parameters, but we applied only the first one. So, computing prod2 is easy, Justtake the original function of prodxy=x*y ,substitute2for x, andyouhave prody=2*y .Thefunctionsworkjustasyou'dexpect:

```
Prelude>double3
Prelude>triple4
12
```

So, themystery is solved. When Haskell computes prod24, it is really computing (prod2)4 ,likethis:

- First, apply prod2. That returns the function $(\y->2^*y)$.
- Next, apply (\y->2*y)4 ,or 2*4 ,givingyou8.

That process is called *currying*, and just about every multiple-argument function in Haskell gets curried. That leads to greater flexibility and simplersyntax. Most of the time, you don't really have to think about it, because the value of curried and uncurried functions is equivalent.

LazyEvaluation

Like Clojure's sequence library, Haskell makes extensive use of lazy evaluation. Whit, you can build functions that return infinite lists. Often, you'll use list construction to form an infinite list. Take this examplethatbuildsaninfiniterange, starting at x,instepsof y:

```
Download haskell/my_range.hs
moduleMainwhere
    myRangestartstep=start:(myRange(start+step)step)
```

Thesyntaxisstrange, but the overall effect is beautiful. When building a function called manage , taking a starting point and a step for our range. We use list composition to build a list with start as the head

and (myRange(start+step)step) as the tail. These are the successive evaluations for m/Rmge 11

- 1:myRange(21)
- 1:2:myRange(31)
- 1:2:3:myRange(41)

...andsoon.

This recursion will go on infinitely, so we'll typically use the function withothersthatwilllimittherecursion. Makesureyouload myanaehs first:

```
*Main>take10(myRange101)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
*Main>take5(myRange05)
[0,5,10,15,20]
```

Somerecursivefunctionsworkmoreefficiently using list construction. Here'sanexampleoftheFibonaccisequence,usinglazyevaluationwith composition:

```
Download haskell/lazy_fib.hs
moduleMainwhere
    lazyFibxy=x:(lazyFiby(x+y))
    fib=lazyFib11
    fibNthx=head(drop(x-1)(take(x)fib))
```

Thefirstfunctionbuildsasequencewhereeverynumberisthesumof the previous two. We effectively have a sequence, but we can improve on the API. To be a proper Fibonacci sequence, we must start the sequence with land 1, so fib seeds lazyFib withthefirsttwonumbers.Finally,we have one more helper function that allows the user to grab just one number of the sequence with drop and take. Herearethe functions in action:

```
*Main>take5(lazyFib01)
[1,1,2,3,5]
*Main>take5(fib)
[1,1,2,3,5]
*Main>take5(drop20(lazyFib01))
[10946,17711,28657,46368,75025]
*Main>fibNth3
2
*Main>fibNth6
```

The three functions are beautiful and concise. We define an infinite sequence, and Haskell computes only the partnecessary to do the job. You can really start to have fun when you start to combine infinite sequences together. First, let's add two Fibonacci sequences together, offsetbyone:

```
*Main>take5(zipWith(+)fib(drop1fib))
[2,3,5,8,13]
```

Surprise. We get a Fibonacci sequence. These higher-order functions playwelltogether.Wecalled zipWith, which pairse achitem of the infinite passeditthe + function.Or, we could double arange: listbyindex.We

```
*Main>take5(map(
                     *2)[1..])
[2,4,6,8,10]
```

Weusingmaptoapplythepartiallyappliedfunction *2 totheinfinite range [1...], and then we're using the infinite range, beginning with 1.

The nice thing about functional languages is that you can compose theminunexpectedways. For example, we can use function compositioninconjunctionwithpartiallyappliedfunctionsandlazysequences effortlessly:

```
*Main>take5(map((
                     * 2).( * 5))fib)
[10,10,20,30,50]
```

Thatcodepacksapunch, solet's take it apart. Starting from the inside andworkingout, wefirs thave (*5). That's apartially applied function. Whateverwepassintothefunction will be multiplied by five. We pass that result into another partially applied function, (*2). We pass that composed function into map and apply the function to every element in the infinite fib sequence. We pass that infinite result to take5 and generate the first five elements of a Fibonacci sequence, multiplied by fiveandthenagainby2.

Yucaneasilyseehowyou'dcomposethesolutionstoproblems. Yu just passone function to the next. In Haskell, f.gx is shorthand for f(gx). When you're building functions in this way, you might want to applythemfromfirsttolast. Yhddosowith the . operator.Forexample, to invert an image, flip it vertically and then flip it horizontally, animageprocessormightdosomethinglike (flipHorizontally.flipVertically .invert)image .

AnInterviewwithSimonReportes

Totakeaguickbreak, let's hear from another person on the committee that created Haskell. Simon Peyton Jones spent seven years as a lecturerat University College London and nine years as a professor atGlasgow University, before moving to Microsoft Research (Cambridge) in 1998 where his research focus is the implementation and application of the contraction of the contractitionoffunctionalprogramminglanguagesforuniprocessorandparallel machines. Heistheleaddesigner of the compiler used in this book.

Bruce Tate: TellmeaboutthecreationofHaskell.

Simon Peyton-Jones: A very unusual thing about Haskell is that it is a successful committee language. Think of any successful language, and the chances are that it was originally developed by one person or averysmallteam.Haskellisdifferent:itwasoriginallydesignedbyan internationalgroupoftwenty-ishrsatusWehadenoughagreement about the core principles of the language—and Haskellis avery principledlanguage—tokeepthedesigncoherent.

Also, Haskellisenjoyingasubstantialupsurgeinpopularitysometwenty years after it was designed. Languages usually succeed or (mostly) fail in the first few years of their lives, so what is going on? I believe that it is because Haskell's principled adherence to purity, the absence of side effects, is an unfamiliar discipline that has prevented Haskell frombeingamainstreamlanguage. Thoselong-termbenefits are graduallybecomingapparent. Whetheromotthemainstreamlanguages of the futurelooklikeHaskell,Ibelievetheywillhavestrongmechanisms for controllingsideeffects.

Bruce Tate: Whatarethethingsyoulikeaboutithemost?

Simon Peyton-Jones: Apart from purity, probably the most unusual and interesting feature of Haskellisits types ystem. Static types are by farthemostwidelyusedprogramverificationtechniqueavailabletoday: millionsofprogrammerswritetypes(whicharejustpartialspecifications) every day, and compilers check them every time they compile the program. Typesare the UML of functional programming: a design language that forms an intimate and permanent part of the program.

From day 1 Haskell's type system was unusually expressive, mainly because of type classes and higher-kinded type variables. Since then, Haskellhasservedasalaboratoryinwhichtoexplorenewtypesystem ideas, something I have enjoyed very much. Multiparameter typeclasses, higher-ranktypes, first-classpolymorphism, implicit parameters, GADTs, and type families...we are having fun! And, more importantly, we are extending the range of properties that can be statically checked by the typesystem.

Bruce Tate: Whatarethethingsyou'dchangeifyouhadittodoall overagain?

Simon Peyton-Jones: I'dlikeabetterandsystem.Thereareans thatHaskell's **mad** systemissos imple, butit's stillaweak point.

I'dlikeabettermodulesystem. Specifically, Iwanttobeable to shipa HaskellpackagePtosomeoneelse, saying "Pneedstoimportinterfaces IandJfromsomewhere:uouprovidethem,anditwillofferinterfaceK." Haskellhasnoformalwaytosaythis.

Bruce Tate: What's the most interesting problem you've seen solved withHaskell?

Simon Peyton-Jones: Haskellisatrulygeneral-purposeprogramming language, which is a strength but also a weakness because it has no single "killerapp." That said, it is quite common to find that Haskellis amediuminwhichpeoplehavebeenabletodreamupparticularlyelegantandunusualwaystosolveproblems.LookatConalElliot'sworkon functional **rate** animation, for example, which **rute**l my brain by making methink of a "time-varying value" as a single value that could bemanipulated by a functional program. On a more mundane (but very useful)level, there are lots of libraries of parser and pretty-printing combinators, each encapsulating great intellectual cleverness behind simple interfaces.Inathirddomain,Jean-MarcEbershowedmehowtodesign acombinatorylibrarytodescribefinancialderivatives, something Iwould neverhavethoughtofonmyown.

Ineachcase, the medium (Haskell) has allowed a new level of expressiveness that would be much harder to achieve in a mainstream language.

By now, you have enough knowledge to tackle some hard problems in Haskell, but you can't do some easy stuff, such as dealing with I/O, state, and error handling. These problems will take us into some advanced theory. Onday 3, we'll look into monads.

WhatWeLearnedinDay2

Inday2, welooked a thigher-order functions. We started with the same kindsoflistlibrariesthatyou'veseeninalmosteverylanguageinthis

book. You saw mode, several versions of to and some additional functionslike zip and zipWith.Afterworkingwiththemonfixedlists,wethen workedwithsomelazytechniquessuchastheonesyouusedwithClojure.

Asweworkedthroughadvancedfunctions, welearned to take a functionandapplysomeoftheparameters. This technique was called partially applied functions. Then, we used partially applied functions to translateafunctionthattookmultipleargumentsatonce(f(x,y))toa functionthattookargumentsoneatatime(f(x)(y). We learned that in Haskell, all functions are curried, which explained the type signatures of Haskell functions taking multiple arguments. For example, the type signatureofthefunction is f::(Numa)=>a->afxy=x+y

We also learned function composition, a process that used the return from one function as the input of another. We could effectively string functionstogetherthisway.

Finally, we worked with lazy evaluation. We were able to define functionsthatbuiltinfinitelists, which would be processed on demand. We defined a Fibonacci sequence in this way and also used composition withlazysequencestoeffortlesslyproducenewlazysequences.

Day2Self-Study

Find:

- Functionsthatyoucanuseonlists, strings, ortuples
- Awaytosortlists

Do:

- Weasortthattakesalistandreturnsasortedlist.
- Weasortthattakesalistandafunctionthatcomparesitstwo argumentsandthenreturnsasortedlist.
- WeaHaskellfunctiontoconvertastringtoanumber. The string shouldbeintheformof \$2,345,678.99 and can possibly have leading zeros.
- We a function that takes an argument x and returns a lazy sequencethathaseverythirdnumber, starting with x.Then,write a function that includes every fifth number, beginning with ٧. Combine these functions through composition to return every eighthnumber, beginning with

• Use a partially applied function to define a function that will returnhalfofanumberandanotherthatwillappend \n totheend ofanystring.

Here are some more demanding problems if you're looking for somethingevenmoreinteresting:

- Weafunction to determine the greatest common denominator oftwointegers.
- Createalazysequenceofprimenumbers.
- Break a long string into individual lines at proper word boundaries.
- Addlinenumberstothepreviousexercise.
- Totheaboveexercise,addfunctionstoleft,right,andfullyjustify thetextwithspaces(makingbothmarginsstraight).

8.4 Day3:TheMindMeld

In StarTrek, Spockhada special talent of connecting with a characterwithwhathecalledthe mindmeld . Haskellenthusiastsoftenclaim such a connection to their language. For many of them, the language featurethatengendersthemostrespectisthetypesystem.Afterspendingsomuchtimewiththelanguage, Icaneasily seewhythisistrue. The type system is flexible and richenough to infermost of my intent, stayingoutofmywayunlessIneedit.IalsogetasanitycheckasIbuild myfunctions, especially the abstractones that compose functions.

ClassesandTypes

Haskell's type system is one of its strongest features. It allows type inference, so programmers do not have heavier responsibilities. It is alsorobustenoughtocatchevensubtleprogrammingerrors. Itispolymorphic, meaning you can treat different forms of the same type the same. In this section, we'll look at a few examples of types and then buildsomeofourowntypes.

BasicTypes

Let's review what you've learned so far with some basic types. First, we'llturnonthetypeoptionintheshell:

Prelude>:set+t

Now, we'll see the types that each statement returns. Try some charactersandstrings:

```
Prelude>'c'
'c'
it::Char
Prelude>"abc"
"abc"
it::[Char]
Prelude>['a','b','c']
"abc"
it::[Char]
```

it always gives you the value of the last thing you typed, and you can read :: as isoftype .ToHaskell,acharacterisaprimitivetype.Astring isanarrayofcharacters.Itdoesn'tmatterhowyourepresentthearray of characters, with an array or with the double quotes. To Haskell, the valuesarethesame:

```
Prelude>"abc"==['a','b','c']
True
```

Thereareafewotherprimitivetypes, likethis:

```
Prelude>True
True
it::Bool
Prelude>False
False
it::Bool
```

Aswedigdeeperintotyping, these ideas will help us see what's really goingon.Let's defines ome of our own types.

User-DefinedTypes

We candefineourowndatatypeswiththe data keyword. The simplest of type declarations uses a finite list of values. Boolean, for example, wouldbedefinedlikethis:

```
dataBoolean=True|False
```

Thatmeansthatthetype Boolean will have a single value, either Тe or Re . We can define our own types in the same way. Consider this simplifieddeckofcards, with two suits and fiveranks:

```
Download haskell/cards.hs
moduleMainwhere
    dataSuit=Spades|Hearts
    dataRank=Ten|Jack|Queen|King|Ace
```

```
*Main>:loadcards.hs
[1of1]CompilingMain
                                     (cards.hs,interpreted)
Ok, modules loaded: Main.
*Main>Hearts
<interactive>:1:0:
    Noinstancefor(ShowSuit)
      arisingfromauseof`print'at<interactive>:1:0-5
```

Argh!Whathappened?Haskellisbasicallytellingusthattheconsoleis tryingtoshowthesevaluesbutdoesn'tknowhow. There's ashorthand waytoderivethe show functionasyoudeclareuser-defineddatatypes. Itworkslikethis:

```
Download haskell/cards-with-show.hs
moduleMainwhere
    dataSuit=Spades | Heartsderiving(Show)
    dataRank=Ten|Jack|Queen|King|Acederiving(Show)
    typeCard=(Rank, Suit)
    typeHand=[Card]
```

Noticeweaddedafewaliastypestooursystem.A Card isatuplewith arankandasuit.anda Hand isalistofcards.We canusethesetypes tobuildnewfunctions:

```
value::Rank->Integer
valueTen=1
valueJack=2
valueQueen=3
valueKing=4
valueAce=5
cardValue::Card->Integer
cardValue(rank,suit)=valuerank
```

For any cardgame, we need to be able to assign the ranks of a card. That's easy. The suitreally doesn't play a role. We simplydefineafunctionthatcomputes the vote of a Rank and then another that computes a cardValue. Here's the function in action:

```
*Main>:loadcards-with-show.hs
[1of1]CompilingMain
                                     (cards-with-show.hs,interpreted)
Ok, modules loaded: Main.
*Main>cardValue(Ten,Hearts)
1
```

working with a complex tuple of user-defined types. The type Water systemkeepsourintentionsclear, soit's easier to reason about what's happening.

Functions and Ramphim

Earlier, you saw a few function types. Let's look at a simple function:

```
backwards[]=[]
backwards(h:t)=backwardst++[h]
```

We could add a type to that function that looks like this:

```
backwards::Hand->Hand
```

That would restrict the backwards function to working with only one kindoflist, alistof cords. What were allywant is this:

```
backwards::[a]->[a]
backwards[]=[]
backwards(h:t)=backwardst++[h]
```

Now, the function is polymorphic. [a] meanswecanusealistofany type. It means that we can define a function that takes a list of some type a andreturnsalistofthatsametype a.Wh [a] -> [a],we'vebuilt atemplateoftypesthatwillworkwithourfunction.Further,we'vetold the compiler that if you start with a list of Integers, this function will returnalistof Integers. Haskell now has enough information to keep vouhonest.

Let'sbuildapolymorphicdatatype. Here's one that build sathree-tuple havingthreepointsofthesametype:

```
Download haskell/triplet.hs
moduleMainwhere
    dataTripleta=Trioaaaderiving(Show)
```

Ontheleftsidewehave data**ī**ta .Inthisinstance, a isatypevariable. Sonow, anythree-tuple with elements of the same type will be of .Takealook: type **te**a

```
*Main>:loadtriplet.hs
[1of1]CompilingMain
                                     (triplet.hs,interpreted)
Ok, modules loaded: Main.
*Main>:tTrio'a''b''c'
Trio'a''b''c'::TripletChar
```

Iused the data constructor **b** to build a three-tuple. We talkmore about the data constructors in the next section. Based on our type dec-

laration,theresultwasa ,ormorespecifically,a Totta **Tet**char and willsatisfyanyfunctionthatrequiresa **Tob**ta .Webuiltawholetemplateoftypes, describing anythree elements who setype is the same.

RecursiveTypes

Yucanalsohavetypesthatarerecursive. For example, think about a tree. Yucandothisinseveralways, butinourtree, the values are on theleafnodes. Anode, then, is either aleaforalist of trees. We could describethetreelikethis:

```
Download haskell/tree.hs
moduleMainwhere
    dataTreea=Children[Treea] | Leafaderiving(Show)
```

So, we have one type constructor, ₩ alsohavetwodataconstructors, Children and Leaf. We can use all of those together to represent trees.likethis:

```
Prelude>:loadtree.hs
[1of1]CompilingMain
                                      (tree.hs,interpreted)
Ok, modules loaded: Main.
*Main>letleaf=Leaf1
*Main>leaf
Leaf1
```

First, webuild a tree having a single leaf. We assignthenewleaftoa variable. The only job of the data constructor Leaf is to hold the valuestogetherwiththetype.We canaccesseachpiecethroughpattern matching, like this:

```
*Main>let(Leafvalue)=leaf
*Main>value
1
```

Let'sbuildsomemorecomplextrees.

```
*Main>Children[Leaf1,Leaf2]
Children[Leaf1,Leaf2]
*Main>lettree=Children[Leaf1,Children[Leaf2,Leaf3]]
*Main>tree
Children[Leaf1,Children[Leaf2,Leaf3]]
```

We buildatreewithtwochildren, each one being aleaf. Next, we build atreewithtwonodes, aleafandarighttree. Once again, we can use patternmatchingtopickoffeachpiece.We cangetmorecomplexfrom there. The definition is recursive, so we can go as deep as we need through let andpatternmatching.

```
*Main>let(Childrench)=tree
*Main>ch
[Leaf1,Children[Leaf2,Leaf3]]
*Main>let(fst:tail)=ch
*Main>fst
Leaf1
```

We canclearly see the intent of the designer of the typesystem, and we canpeeloffthepiecesthatweneedtodothejob. This design strategy obviouslycomeswithanoverhead, but as you dive into deeperabstractions, sometimes the extra overhead is worth the hassles. In this case, thetypesystemallowsustoattachfunctionstoeachspecifictypeconstructor. Let's look at a function to determine the depth of a tree:

```
depth(Leaf)=1
depth(Childrenc)=1+maximum(mapdepthc)
```

The first pattern in our function is simple. I fit's aleaf, regardless of the contentoftheleaf, the depth of the tree is one.

Then ext pattern is a little more complicated. If we call depthon Children. .Thefunction momm weaddoneto mainum(mapdepthc) computes the maximum element in an array, and you've seen that mondepthc will compute a list of the depths of all the children. In this case, you can see how we use the data constructors to help us match the exactpiecesofthedatastructurethatwillhelpusdothejob.

Classes

Sofar, we've been through the type system and how it works in a couple of areas. Wee built user-defined type constructors and got templates that would allow us to define data types and declare functionsthat would work with them. Haskell has one more important conceptrelated to types, and it's a bigone. The concept is called the class, but be careful. It's not an object-oriented class, because there's no data involved.InHaskell,classesletuscarefullycontrolpolymorphismand overloading.

Forexample, you can't add two boolean stogether, but you can add two numberstogether. Haskellallowsclasses for this purpose. Specifically, aclassdefineswhichoperationscanworkonwhichinputs .Thinkofit likeaClojure protocol.

Here'showitworks. Aclass provides some function signatures. A type isaninstanceofaclassifitsupportsallthosefunctions. For example, intheHaskelllibrary,there'saclasscalled Ea.

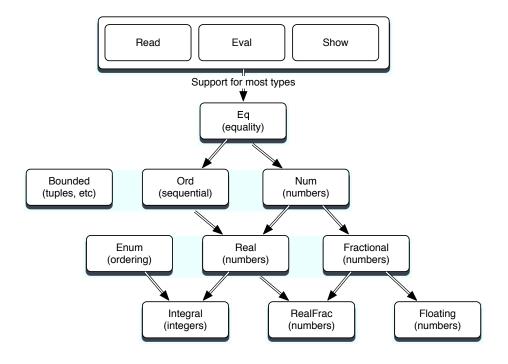


Figure 8.1: Important Haskell classes

Here'swhatitlookslike:

```
class Eqa where
   (==), (/=)::a->a->Bool
       --Minimalcompletedefinition:
              (==)or(/=)
   x/=v
              = not(x==y)
              = not(x/=y)
   x==y
```

Eq ifitsupportsboth == and /=.Yucanalso So, atype is an instance of specifyboilerplateimplementations. Also, if an instance defines one of thosefunctions, the other will be provided for free.

Classes do support inheritance, and it behaves like youthink it should.Forexample, the Num classhassubclasses Fractional and Real. Thehierarchy of the most important Haskell classes in Haskell 98 is shown inFigure 8.1. Remember, instances of these classes are types, not data objects!

Monads

FromthetimeIdecidedtowritethisbook, I'vedreadedwritingthesection on monads. After some study, I've learned that the concepts are notallthatdifficult.Inthissection,I'llwalkyouthroughtheintuitive description of why we need monads. Then, we'll look at a high-level description of how monads are built. Finally, we'll introduce some syntacticsugarthatshouldreallybringhomehowtheywork.

Ileanedonacoupleoftutorialstohelpshapemyunderstanding. The Haskellwiki ³ hasseveralgoodexamplesthatIread, and also UnderstandingMonads 4 hassomegoodpracticalexamples.Butyou'llprobablyfindthatyouneedtowadethroughseveralexamplesfrommany different sources to come to an understanding of what monads can doforyou.

The Problem: Drunken Pirate

Let'ssayyouhaveapiratemakingatreasuremap. He'sdrunk, sohe picksupaknownpointandaknowndirectionandmakeshiswayto thetreasurewithaseriesofstaggersandcrawls. Astaggermoves two steps, and a crawl move sone step. In an imperative language, you will have statements strungtogethers equentially, where v isthevaluethat holdsdistancefromtheoriginalpoint, likethis:

```
deftreasure_map(v)
    v=stagger(v)
    v=stagger(v)
    v=crawl(v)
    return(v)
end
```

We have several functions that we call within treasure_map thatsequentiallytransformourstate,the distance traveled. The problem is that we havemutablestate.We coulddotheprobleminafunctionalway,like this:

```
Download haskell/drunken-pirate.hs
moduleMainwhere
    stagger::(Numt)=>t->t
    staggerd=d+2
    crawld=d+1
```

^{3.} http://www.haskell.org/tutorial/monads.html

^{4.} http://en.wikibooks.org/wiki/Haskell/Understanding monads

```
treasureMapd=
    crawl(
    stagger(
    staggerd))
```

Yucanseethatthefunctionaldefinitionisinconvenienttoread.Rather than stagger, stagger, and crawl, we must read crawl, stagger, and stagger, and the arguments are awkwardly placed. Instead, we'd like a strategythat will let us chain several functions together sequentially.

We canusea let expressioninstead:

```
letTreasureMap(v,d)=letd1=staggerd
                            d2=staggerd1
                            d3=crawld2
                        ind3
```

Haskell allows us to chain let expressions together and express the can see that this version is almost finalforminan in statement. You as unsatisfying as the first. The inputs and outputs are the same, so it should be easier to compose these kinds of functions. We want to translate stagger(crawl(x)) into $stagger(x) \cdot crawl(x)$, where \cdot is function composition. That's amonad.

Inshort, amonadlets us compose functions in ways that have specific properties. In Haskell, we'll use monads for several purposes. First, dealingwiththingssuchasI/Oisdifficultbecauseinapurefunctional language, a function should deliver the same results when given the same inputs, but for I/O, you would want your functions to change based on the state of the contents of a file, for example.

Also, code like the drunken pirate earlier works because it preserves state. Monadsletyousimulateprogramstate. Haskell provides a specialsyntax, called dosyntax, to allow programs in the imperative style. Dosyntaxdependsonmonadstowork.

Finally, something as simple as an error condition is difficult because the type of thing returned is different based on whether the function Maybe monad for this purpose. was successful. Haskell provides the Let'sdigalittledeeper.

ComponentsofaMonad

Atitsbasiclevel, amonadhas three basic things:

• A type constructor that's based on some type of container. The container could be a simple variable, a list, or anything that can holdavalue. We will use the container to hold a function. The containeryouchoosewillvarybasedonwhatyouwantyourmonad todo.

- A function called return that wraps up a function and putsit in thecontainer. The name will make sense later, when we move into do notation. Just remember that return wraps up a function into a monad.
- Abindfunctioncalled >>= thatunwrapsafunction.We use bind tochainfunctionstogether.

All monads will need to satisfy three rules. I'll mention them briefly here.Forsomemonad m.somefunction f.andsomevalue

- should be able to use a type constructor to create a monad that will work with sometype that can hold a value.
- Yo1 should be able to unwrap and wrap values without loss of information.(monad>>=return=monad)
- Nesting bind functions should be the same as calling them sequentially.($(m>=f)>=g=m>=(\x->fx>=g)$

We won't spendal otoftime on the selaws, but there as on sarepretty simple. They allow many useful transformations without losing information. If your eally want to dive in, I'll try to leave you plenty of references.

buildonefrom Enoughofthetheory.Let'sbuildasimplemonad.We scratch, and then I'll close the chapter with a few useful monads.

BuildingaMonadfromScratch

The first thing we'll need is a type constructor. Our monad will have a functionandavalue.likethis:

```
Download haskell/drunken-monad.hs
moduleMainwhere
    dataPositiont=Positiontderiving(Show)
    stagger(Positiond)=Position(d+2)
    crawl(Positiond)=Position(d+1)
    rtnx=x
    x>==f=fx
```

The three main elements of a monad were a type container, a return, and abind. Our monadist he simplest possible. The type container is

asimpletypeconstructorthatlookslike t. Allit. data Rota t=Roman doesisdefineabasictype, basedonanarbitrary type template. Next, weneeda return thatwrapsupafunctionasavalue. Since our monad issosimple, we just have to return the value of the monaditself, and it'swrappedupappropriately, with(rtn x=x). Finally, wene ededabind thatallowsustocomposefunctions.Oursiscalled >>==.andwedefine it to just call the associated function with the value in the monad (>>==f=fx). Wate using >>== and rtn instead of >>= and return to prevent collisionswithHaskell'sbuilt-inmonadfunctions.

Notice that we also rewrote stagger and crawl to use our homegrown monadinsteadofnakedintegers. Wecantakeour monadout for a testdrive. Remember, we were after a syntax that translates from nesting tocomposition. Therevised treasure maplooks like this:

```
treasureMappos=pos>>==
                   stagger>>==
                   stagger>>==
                   crawl>>==
                   rtn
```

Anditworksasexpected:

```
*Main>treasureMap(Position0)
Position5
```

MonadsanddoNotation

Thatsyntaxismuchbetter, butyoucan easily imagine some syntactic sugartoimproveitsomemore. Haskell's do syntaxdoesexactlythat. The do syntax comes in handy especially for problems like I/O. In the following code, were adaline from the console and print out the same lineinreverse, using do notation:

```
Download haskell/io.hs
moduleMainwhere
                          "Enteryourname:"
    tryIo=do putStr
                  line<-getLine;</pre>
                 let{backwards=reverseline};
                  return( "Hello.Yournamebackwardsis"
                                                               ++backwards)
```

Notice that the beginning of this program is a function declaration. Then, we use the simple do notation to give us the syntactic sugar aroundmonads. That makes our program feels tateful and imperative, butwe'reactuallyusingmonads. Yalwanttobeawareofafewsyntax rules.

Assignmentuses <-.InGHCI,youmustseparatelineswithsemicolons and include the body of do expressions, and let expressions therein, withinbraces. If you have multiple lines, you should wrap your code in :{ and }: witheachonaseparateline.Andnow,youcanfinallyseewhy wecalledourmonad'swrappingconstruct return. It neatly packages a return value in a tidy form that the do syntax can absorb. This code behaves as if it were in a stateful imperative language, but it's using monads to manage the stateful interactions. All I/O is tightly encapsulated and must be captured using one of the I/O monads in a do block.

DifferentComputationalStrategies

Every monad has an associated computational strategy. The identity monad, which we used in the drunken-monad example, just parrots backthethingyouputintoit.We usedittoconvertanestedprogram structuretoasequentialprogramstructure.Let'stakeanotherexample.Strangeasitmayseem,alistisalsoamonad,with return andbind (>>=)definedlikethis:

```
instanceMonad[]where
   m>>=f =concatMapfm
    returnx=[x]
```

Recall that a monad needs some container and a type constructor, a return method that wraps up a function, and a bind method that unwrapsit.Amonadisaclass,and [] instantiatesit, giving usour type constructor.We nextneedafunctiontowrapuparesultas return.

Forthelist, wew rapupthe function in the list. To unwrapit, our bind callsthefunctiononeveryelementofthelistwith mon and then concatenates the results together. concot and map are applied in sequence often enough that there's a function that does both for convenience, butwecouldhaveeasilyused concat(mapfm) .

To giveyouafeelforthelistmonadinaction, takealookatthefollowing script,in do notation:

```
Main>letcartesian(xs,ys)=dox<-xs;y<-ys;return(x,y)</pre>
Main>cartesian([1..2],[3..4])
[(1,3),(1,4),(2,3),(2,4)]
```

We created a simple function with do notationandmonads. We took x from alist of xs, and we took y from a list of xy. Then, we return edeach combination of x and y.Fromthatpoint,ourpasswordcrackeriseasy.

```
Download haskell/password.hs
moduleMainwhere
                      'a'..'c'];y<-[ 'a'..'c'];z<-[ 'a'..'c'];
    crack=dox<-[
               let{password=[x,y,z]};
               ifattemptpassword
                    thenreturn(password,True)
                    elsereturn(password, False)
    attemptpw=ifpw==
                           "cab" thenTrueelseFalse
```

Here, we're using the list monad to compute all possible combinations. x<- [lst] means "for each x taken from [lst]." Notice that in this context, We let Haskell do the heavy lifting. At that point, all you need to do is tryeachpassword.Ourpasswordishard-codedintothe attempt function. There are many computational strategies that we could have used to solve this problem such as list comprehensions, but this problem showed the computational strategy behind list monads.

MaybeMonad

Sofar,we'veseenthe Identity monadandthe List monad. With the latter, welearnedthatmonadssupportedacentralcomputationalstrategy.In thissection, we'll look at the usethisonetohandle Maybe monad.We acommonprogrammingproblem:somefunctionsmightfail.Yumight thinkwe'retalkingabouttherealmofdatabasesandcommunications, but other far simpler APIs often need to support the idea of failure. Thinkastringsearchthatreturnstheindexofastring. If the string is present, the return type is an Integer.Otherwise,thetypeis Nothing.

Stringing together such computations is tedious. Let's say you have a functionthatisparsingawebpage. Yowantthe HTML page, the body withinthatpage, and the first paragraph within that body. You want to codefunctions that have signatures that looks omething like this:

```
paragraphXmlDoc->XmlDoc
bodyXmlDoc->XmlDoc
htmlXmlDoc->XmlDoc
```

Theywillsupportafunctionthatlookssomethinglikethis:

```
paragraphbody(htmldoc)
```

Theproblemisthatthe paragraph, body, and html functions can fail, so youneedtoallowatypethatmaybe Nothing. Haskellhassuchatype, called Just. Justx canwrap Nothing, or sometype, like this:

```
Prelude>Just"somestring"
Just"somestring"
Prelude>JustNothing
JustNothing
```

Yucanstripoffthe Just withpatternmatching.So, gettingbacktoour example, the paragraph, body, and html documents can return Just Xml-Doc.Then,youcouldusetheHaskell case statement(whichworkslike the Erlang cases ta tement) and pattern matching to give you something likethis:

```
case(htmldoc)of
 Nothing->Nothing
 Justx ->casebodyxof
               Nothing->Nothing
               Justy ->paragraph2y
```

Andthatresultisdeeplyunsatisfying, considering wewanted to code paragraph2body(htmldoc) . Whatwereally need is the Maybe monad. Here'sthedefinition:

```
dataMaybea=Nothing|Justa
instanceMonadMaybewhere
    return
                  =Just
   Nothing >>=f=Nothing
    (Justx)>>=f=fx
```

Thetypewe'rewrappingisatypeconstructorthatis Maybea .Thattype canwrap Nothing or Justa.

return iseasy. It just wraps the result in Just.Thebindisalsoeasy.For Nothing, it returns a function returning Nothing. For Justx, itreturns a function returning x. Either will be wrapped by the return. Now, you canchaintogethertheseoperationseasily:

```
JustsomeWebPage>>=html>>=body>>=paragraph>>=return
```

So, we can combine the elements flawlessly. It works because the monad takes care of the decision making through the functions that we compose.

WhatWeLearnedinDay3

In this section, we took on three demanding concepts: Haskell types, classes.andmonads.We startedwithtypes, by looking at the inferred types of existing functions, numbers, booleans, and characters. We then moved on to some user-defined types. As a basic example, we usedtypestodefineplayingcardsmadeupofsuitsandranksforplayingcards. Welearned how to parameterize types and even use recursive typedefinitions.

Then, wew rapped up the language with a discussion of monads. Since Haskellis a purely functional language, it can be difficult to express problemsinanimperativestyleoraccumulatestateasaprogramexecutes. Haskell's designers leaned on monads to solve both problems. A monadisatypeconstructorwithafewfunctionstowrapupfunctions and chain them to gether. You can combine monads with different type containers to allow different kinds of computational strategies. We used monadstoprovideamorenaturalimperativestyleforourprogramand toprocessmultiplepossibilities.

Day3Self-Study

Find:

- Afewmonadtutorials
- AlistofthemonadsinHaskell

Do:

- We a function that looks up a hash table value that uses the Maybe monad. We ahashthat stores other hashes, severallevelsdeep.Usethe Maybe monadtoretrieveanelementforahash keyseverallevelsdeep.
- RepresentamazeinHaskell.**Yil**needa Maze typeanda Node type, as well as a function to return a node given its coordinates. Thenodeshouldhavealistofexitstoothernodes.
- UseaListmonadtosolvethemaze.
- Implementa Monadina nonfunctional language. (See the article seriesonmonadsinRuby.

^{5.} http://moonbase.rydia.net/mental/writings/programming/monads-in-ruby/00introduction.html

Of all the languages in this book, Haskell was the only one created by committee. After the proliferation of purely functional languages with lazy semantics, a committee was formed to build an open standardthatwouldconsolidateexistingcapabilities and future research. Haskellwasborn, with version 1.0 defined in 1990. The language and communityhavegrownsincethen.

Haskellsupportsawidevarietyoffunctionalcapabilitiesincludinglist comprehensions, lazy computing strategies, partially applied functions, and currying. In fact, by default, Haskell functions process one parameteratatime, using currying to support multiplear guments.

The Haskell type system provides an excellent balance of type safety and flexibility. The fully polymorphic template system allows sophisticated support for user-defined types and even type classes that fully support inheritance of interface. Usually, the Haskell programmer is notburdenedwithtypedetailsexceptinthefunctiondeclarations,but thetypesystemprotectsusers from all kinds of type errors.

Aswithanypurefunctionallanguage, Haskelldevelopersmustbecreative to deal with imperative-style programs and accumulated state. I/Ocan also be a challenge. Fortunately, Haskell developers can rely on monads for that purpose. A monad is a type constructor and a container that supports basic functions to wrap and unwrap functions as values. Different container types provide different computationalstrategies. These functions allow programmers to chain to gether monadsininterestingways, providing dosyntax. This syntactic sugar allowsimperative-styleprogramswithcertainlimitations.

CoreStrengths

Since Haskell takes the absolute approach of pure functions with no compromise, the advantages and disadvantages can be often extreme. Let'sbreakthemdown.

TypeSystem

If you like strong typing (and maybe even if you don't), you'll love Haskell's typing system. It is there when you need it but not when you don't. The type system can add a helpful level of protection from common errors, and they can be caught at compile time rather than runtime.Buttheextrasafetyisonlypartofthestory.

PerhapsthemostinterestingpartofaHaskelltypeishoweasyitisto associatenewtypeswithnewbehaviors. Yucanbuildupsophisticated typesfromthegroundup. Whtypeconstructors and classes, you can even customize extremely complex types and classes such as Monadseffortlessly. Who classes, your new custom types can take advantage of existingHaskelllibraries.

Expressiveness

The Haskelllanguage has fantastic power. From an abstract sense, it has everything you need to express powerful ideas concisely. Those ideasencompassbehaviorthrougharichfunctionallibraryandapowerful syntax. The ideas extend to data types where you can create types, even recursive types that bind the right functions to the right datawithoutexcessivesyntax.Inanacademicsetting,youcanfindno strongerlanguage for teaching functional programming than Haskell. Everythingyouwillneedisinthere.

PurityofProgrammingModel

Pureprogrammingmodelscanradicallychangethewayyouapproach problems. Theyforceyoutoleaveoldprogrammingparadigmsbehind andembracedifferentwaysofdoingthings.Purefunctionallanguages giveyousomethingyoucandependon. Given the same inputs, a functionwillalwaysreturnthesamevalues. This property makes it much easiertoreasonaboutprograms. Yucansometimes prove that a programiscorrect, or not. You can also be free of many of the problems that come from depending on side effects, such as a ccidental complexityandunstableorslowbehaviorinconcurrentsituations.

LazySemantics

Once upon a time, dealing with functional languages meant dealing with recursion. Lazy computing strategies offer a whole new set of strategies to deal with data. You can often build programs that perform better and take a fraction of the total lines of code that anotherstrategymighttake.

AcademicSupport

SomeofthemostimportantandinfluentiallanguagessuchasPascal grewupinacademia, benefitting greatly from research and use in that setting. As the primary teaching language for functional techniques, Haskellcontinuestoimproveandgrow. Thoughitis not fully amainstreamlanguage, you'llalways beable to find pockets of programmers todoimportanttasks.

Core Wedness

You know by now that no programming language is perfect for every task.Haskell'sstrengthstypicallyhaveaflipsideaswell.

InflexibilityofProgrammingModel

Being a pure functional language offers some advantages but also a setofheadaches. Yumighthavenoticed that programming with monadswasthelastpartofthelastchapterinabookaboutprogramming languages, and rightfully so. The concepts are intellectually demanding. Butweused monads to do somethings that were trivial in other languages, such as write imperative-style programs, process I/O, and evenhandlelistfunctionsthatmayormaynotfindavalue. I'vesaidit beforeaboutotherlanguages, but I'llsayitagainhere. Though Haskell makessomehardthingseasy,italsomakessomeeasythingshard.

Certain styles lend themselves to certain programming paradigms. When you're building a step-by-step algorithm, imperative languages work well. Heavy I/O and scripting do not lend themselves to functional languages. Purity in one man's eyes may seem like failure to compromiseinanother.

Community

Speaking of compromise, you can really see the differences in the approach of Scala and Haskell. Though both are strongly typed, both have radically different philosophies. Scala is all about compromise, and Haskellis allabout purity. By making compromises, Scalahasinitially attracted a much bigger community than Haskell. Though you can't measure success by the size of a programming community, you must have sufficient numbers to succeed, and having more users lends moreopportunityandcommunityresources.

LearningCurve

ThemonadisnottheonlyintellectuallydemandingconceptinHaskell. Curryingisusedineveryfunctionwithmorethanoneargument. Most basic functions have parameterized types, and functions on numbers often use a type class. Though the payoff may be well worth it in the end, you must be a strong programmer with firm theoretical footing to haveafightingchanceofsucceedingwithHaskell.

FinalThoughts

Ofthefunctionallanguagesinthebook, Haskellwasthemostdifficult tolearn. The emphasis on monads and the type system made the learningcurvesteep. Once I mastered some of the key concepts, thing sgot easier, and it became the most rewarding language Ilearned. Based on thetypesystemandtheeleganceoftheapplicationofmonads, oneday we'll look back at this language as one of the most important in thisbook.

Haskell plays another role, too. The purity of the approach and the academicfocuswillbothimproveourunderstandingof The best of the next generation of functional programmers in many placeswillcuttheirteethonHaskell.

Chapter 9

Wrap-Up

Congratulations on making it through seven programming languages. Perhapsyou're expecting metopick winners and losers in this chapter, but this book is not about winners and losers. It's about discovering newideas. Yumay have been like meearly in my career, buried deeply in commercial projects in large teams with little imagination, the software factories of our generation. In such a world, my exposure to programming languages was extremely limited. I was like a 1970 smovie loverin as mall town with one the ater, getting only the big-money block-busters.

Since I started building software formy self, I feel like I've just discovered in dependent films. I've been able to make a living coding Ruby, but I'm not naive enough to think Ruby has all of the answers. Just as independent films are advancing the state of the art in movie making, emerging programming languages are changing the way we think about program organization and construction. Let's review what we've seen throughout the book.

9.1 ProgrammingModels

Programming models change extremely slowly. So far, we've seen new models emerge every twenty years or so. My training started with some procedural languages, Basic and Fortran. In college, Ilearned a more structured approach with Pascal. At IBM, Istarted to code Cand C++ commercially and was introduced to Java for the first time. I also began writing object-oriented code. My programming experience has spanned more than thirty years, and I've seen only two major programming

paradigms. You might be asking yourself why I am so enthusiastic about introducing a few other programming paradigms. It's a fair question.

Though programming paradigms change slowly, they do change. Like atornado's path, they can leave behind some devastation, taking the formofbrokencareers and companies that invested poorly. When you findyourselffightingaprogrammingparadigm, youneed to start paying attention. Concurrency and reliability are starting to nudge us in the direction of a higher-level programming language. Minimally, I think we'regoingtostarttoseemorespecializedlanguagestosolvespecific problems. These are the programming models we encountered.

ObjectOrientation(Ruby,Scala)

Thecurrent "kingofthehill" is objectorientation, typically in the Java language. This programming paradigm has three major ideas: encapsulation, inheritance, and polymorphism. Wh Ruby, we experienced dynamic duck typing. Rather than enforcing a contract based on the definition of a classor objects, Rubyen forced typing based on the methodsanobjectcouldsupport.We learned that Ruby supported several functional concepts through code blocks.

Scala, too, offered object-oriented programming. Though it supports statictyping, it is much less verbosethan Java, offering features such astypeinferencetosimplifythesyntax. Whthis feature, Scalaautomatically deduces the type of variables based on clues in syntax and usage.ScalagoesbeyondRubytointroducefunctionalconcepts.

Both of these languages run in widespread production applications today, and both represents ignificant advances in language design compared to mainstream languages such as Java. There are many variations of object-oriented languages, including the next programming paradigm, prototypelanguages.

PrototypeProgramming(lo)

Yucouldactuallysaythatprototypelanguagesareasubsetofobjectorientedlanguages, butthey are just different en oughin practice that I introduced the masa different programming model. Rather than workingthroughaclass, all prototypes are object instances. Some specially designated instances serve as prototypes for other object instances. This family of languages includes Java Script and Io. Simple and expressive, prototypelanguages are typically dynamically typed and workwell

for scripting and application development, especially for user interfaces.

As you learned within Io, a simple programming model with a small, consistentsyntaxcanbeapowerfulcombination. We usedtheIolanguageinbroadlydifferent contexts ranging from scripting concurrent programstogethertocodingourownDSL.Butthisprototypeprogrammingwasnotthemostspecializedparadigmthatweencountered.

Constraint-LogicProgramming(Prolog)

Prolog comes from a family of programming languages built for constraint-logicprogramming. The different applications we built with Prolog solved a fairly narrow type of problem, but the results were often spectacular. Wedefined logical constraints that we knew about our universeandhadPrologfindthesolution.

When the programming model fit this paradigm, we were able to get results with a fraction of the lines of code that it would take in other languages. This family of language supports many of the most critical applications in the world indomains such as airtraffic control and civil engineering. YucanalsofinderudelogicalrulesenginesinotherlanguagessuchasCandJava.PrologservedastheinspirationofErlang, from another family of languages in this book.

FunctionalProgramming(Scala,Erlang,Clojure,Haskell)

Perhaps the most widely anticipated programming paradigm in this book is functional programming. The degree of purity found in functional programming languages differs, but the concepts are consistent throughout. Functional programs are made up of mathematical functions. Calling the same function more than once will give you the same result each time, and side effects are either frowned on or forbidden. Yucancomposethosefunctions in a variety of ways.

Yive seenthat functional programming languages are usually more expressive than object-oriented languages. Yar examples were often shorterandsimplerthantheobject-orientedcounterpartsbecauseyou had a broader range of tools for composing programs than you did intheobject-orientedparadigm.We introducedhigher-orderfunctions and also complex concepts such as currying as two examples that you can't always find in object-oriented languages. As you learned in Haskell, different levels of purity lead to different sets of advantages and disadvantages. One clear winforthe functional languages was the

absenceofsideeffects, making concurrent programming easier. When mutable state goes away, so do many of the traditional concurrency problems.

Changing Rootyns

If you do decide that you want to do more functional programming, there are several different ways to get there. Yu can make a clean break from OOP, or you can pick an approach that is slightly more evolutionary.

Aswewadedthrougheachofthesevenlanguages, you sawlanguages spanningfourdecadesandatleastasmanyprogrammingparadigms. Ihopeyoucanappreciatetheevolutionofprogramminglanguagesas saw three distinctly different approaches to evolving parawell. You digms.WhScala, the approach is coexistence. The Scalaprogrammer canconstructanobject-orientedprogramusingstrongfunctionaltendencies. The very nature of the language is that both programming paradigms are first-class. Clojure takes the approach of compatibility. The language is built on the JVM, allowing Clojure applications to use Java objects directly, but the Clojure philosophy is that certainelementsofOOParefundamentallyflawed.UnlikeScala,Clojure-Java Interop is provided to leverage frameworks existing on the Java virtual machine, not to broaden the programming language. Haskell and Erlangare basically stand-alone languages. Philosophically, they donotembraceobject-orientedprogramminginanyform.So,youcan embrace both paradigms, make a clean break, or embrace object-orientedlibrariesbutdecidetoleavetheOOPparadigmbehind.Take your pick.

Whetherornotyouchoosetoadoptoneofthelanguagesinthisbook, you'll be better for knowing what's out there. As a Java developer, I hadtowaitadecadeforclosures, mainly because people like mewere uneducated and did not scream loud enough for them. In the meantime, mainstreamframeworkslike Springwere stuckwith an onymous inner classes to solve problems that could have used closures extensively. Myfingers bledfrom the oppressive amount of typing, and my eyes bled because I had to read that stuff. The modern Java developerknows much more, partially because people such as Martin Odersky andRichHickeygaveusalternativesthatarenowpushingthestateof theartandforcingJavatoadvanceorgetleftbehind.

9.2 Concurrency

Arepeatedthemeinthisbookistheneedforbetterlanguageconstructs and programming models to handle concurrency. Across the languages, the approaches were often strikingly different but extremely effective. Let's walk through some of the approaches we saw.

ControllingMutableState

Byfar, the most common the meinthe concurrency discussion was the programming model. Object-oriented programming allows side effects and mutable state. Taken together, programs got much more complicated. When you miximultiple threads and processes, the complexity gottoogreattomanage.

The functional programming language adds structure through an importantrule. Multiple invocations of the same function lead to the same result. Varides haveasingleassignment. When side effects go away, raceconditions and all related complexities also go away. Still, we saw tangible techniques that went beyond the basic programming model. Let'stakeacloserlook.

Actorsinlo, Erlang, and Scala

Whether using an objector approcess, the actor approach is the same. It takes unstructured interprocess communication across any object boundaryandtransformsitontostructuredmessagepassingbetween first-class constructs, each one supporting a message queue. The Erlang and Scala languages use pattern matching to match inbound messages and conditionally execute them. In Chapter 6, Erlang, on page 181, webuiltanexamplearound Russian roulette to demonstrate advingprocess. Recall that we put the bullet inchamber 3:

```
Download erlang/roulette.erl
-module(roulette).
-export([loop/0]).
%sendanumber, 1-6
loop()->
    receive
        3->io:format( "bang.~n"),exit({roulette,die,at,erlang:time()});
        _->io:format( "click~n"),loop()
end.
```

We then started a process, assigning the ID to Gun. We could kill the process with Gun!3. The Erlang virtual machine and language supportedrobustmonitoring, allowing notification and even restarting processesatthefirstsignoftrouble.

Futures

To the actor model, Io added two additional concurrency constructs: coroutines and futures. Coroutines allowed two objects to multitask cooperatively, with each relinquishing control at the appropriate time. Recallthatfutureswereplaceholdersforlong-runningconcurrentcomputations.

We executed the statement futureResult := URL with ("http://google.com/") @fetch.Thoughtheresultwasnotimmediatelyavailable,programcontrolreturnedimmediately, blocking only when we attempted to access the future. An I of uture actually morphs into a result when the result becomesavailable.

TracbdMemory

In Clojure, we saw a number of interesting approaches to concurrency.Softwaretransactionalmemory(STM)wrappedeachdistributed accessofasharedresourceinatransaction. This same approach, but for database objects, maintains database integrity across concurrent invocations. We wrapped each access in a dosync function. Whithis approach, Clojure developers can break away from strict functional designs where it makes sense and still have integrity across multiple threadsandprocesses.

STM is a relatively new idea that is just creeping into more popular languages. As a Lisp derivative, Clojure is an ideal language for such anapproachbecauseLispisamultiparadigmlanguage.Userscanuse different programming paradigms when they make sense with the confidence that the application will maintain integrity and performance, eventhroughhighlyconcurrentaccesses.

Thenextgeneration of programmers will demand more out of his language. The simple wheel and stick that let you start a thread and wait on a semaphore are no longer good enough. A newer language must have a coherent philosophy supporting concurrency and the tools to match. It may be that the need for concurrency renders whole programming paradigms obsolete, or it may be that older languages will

adaptbyusingstrictercontrolsformutablevariablesandsmarterconcurrencyconstructssuchasactorsandfutures.

ProgrammingConstructs 9.3

One of the most exciting parts of writing this book was the exposure to the basic building blocks in the various languages of this book. For eachnewlanguage, Iintroduced majornew concepts. These are some of the programming constructs that you're likely to see in other new languagesyoumightdiscover. They are among my favorited is coveries.

ListComprehensions

AsyousawinErlang,Clojure,andHaskell, ¹ thelistcomprehensionis a compact structure that combines several ideas into a single powerfulconstruct. Alist comprehension has a filter, a map, and a Cartesian product.

We firstencounteredlistcomprehensionsinErlang.We startedwitha shoppingcartofitemssuchas Cart= [{pencil,4,0.25},{pen,1,1.20},{paper, 2,0.20}]. Toaddtaxtothelist, webuilt a single list comprehension to solvetheproblematonce, likethis:

```
WithTax=[{Product,Quantity,Price,Price
                                                 * Quantity * 0.08}||
     {Product,Quantity,Price}<-Cart].
[{pencil,4,0.25,0.08},{pen,1,1.2,0.096},{paper,2,0.2,0.032}]
```

Severaldifferentlanguagecreatorsmentionlistcomprehensionsasone oftheirfavoritefeatures. Iagreewith this sentiment.

Monads

Perhapsthebiggestintellectual growth for mewas in the area of monads. Whpurefunctional languages, we could not build programs with mutablestate.Instead,webuiltmonadsthatletuscomposefunctions in a way that helped us structure problems as if they allowed mutablestate. Haskellhas do notation, supported by monads, to solve this problem.

We also found that monads allows to simplify complex computation. Each of our monads supported a computational strategy. We usedthe Maybe monad to handle failure conditions, such as a List search that

^{1.} Scalaalsosupportslistcomprehensions, but we did not use them.

could potentially return Nothing. We used the List monad to compute a Cartesian product and unlock a combination.

Matching

One of the more common programming features we saw was pattern matching. We first encountered this programming constructin Prolog, butwealsosawitinScala, Erlang, Clojure, and Haskell. Each of these languages leaned on pattern matching to significantly simplify code. The problem domains included parsing, distributed message passing, destructuring, unification, XML processing, and more.

For a typical Erlang pattern match, recall the translate service:

```
Download erlang/translate_service.erl
-module(translate service).
-export([loop/0,translate/2]).
loop()->
    receive
        {From, "casa"}->
            From! "house",
            loop();
        {From, "blanca"}->
            From! "white",
            loop();
        {From,_}->
            From! "Idon'tunderstand." ,
            loop()
end.
translate(To, Word) ->
    To!{self(),Word},
    receive
        Translation->Translation
    end.
```

TheloopfunctionmatchedaprocessID(From)followedbyaword(casa or blanca)orawildcard. The pattern match allows the programmer to quicklypickouttheimportantpiecesofthemessagewithoutrequiring anyparsingfromtheprogrammer.

Unification

Prologusesunification, aclose cousino fpatternmatching. Yulearned that Prologwould substitute possible values into a rule to force the left

andrightsidestomatch. Prologwouldkeeptryingvaluesuntilpossibilities were exhausted. We looked at a simple Prolog program called concatenate asanexampleofunification:

```
Download prolog/concat.pl
concatenate([],List,List).
concatenate([Head|Tail1],List,[Head|Tail2]):-
  concatenate(Tail1,List,Tail2).
```

We learned that unification makes this program so powerful because it could work in three ways: testing truth, matching the left side, or matchingtherightside.

FindingY our Voice 9.4

Wore talked about movies and characters throughout this book. Thejoyofmoviemakingmeanscombiningvourexperiences with the actors, sets, and locations that tell the story you want to tell. Everything you do goes into pleasing your audience. The more you know, the better yourmoviescanbe.

Weneedtothinkaboutprogramminginthesameway. Wetoo, havean audience. I'mnottalking about the users of our applications, though. I'mtalkingaboutthepeoplewhoreadourcode. To be agreat programmer, you need to write to your audience and find the voice that pleases them. Yol havemoreroom to find that voice and let it evolve if you learnwhatotherlanguageshavetooffer. Yrvoiceisyouruniqueway of expressing yourself in code. It will never be any better than the sum of your experience. I hope this book has helped you find your voice. Mostofall, Ihopeyouhadfun.

Appendix A Bibliography

- [Arm07] JoeArmstrong. ProgrammingErlang:SoftwareforaConcurnet Watl . The Pragmatic Programmers, LLC, Raleigh, NC, andDallas,TX,2007.
- [Gra04] PaulGraham. HackersandPainters:BigIdeasfromtheComputerAge .O'Reilly&Associates,Inc,Sebastopol,CA,2004.
- [Hal09] StuartHalloway. ProgrammingClojure. ThePragmaticProgrammers, LLC, Raleigh, NC, and Dallas, TX, 2009.
- [OSV08] MartinOdersky,LexSpoon,andBillVenes **Programming** inScala .Artima,Inc.,MountainWyCA,2008.
- [TFH08] David Thomas, Chad Fowler, and Andrew Hunt. Programming Ruby: The Pragmatic Programmers' Guide . The PragmaticProgrammers,LLC,Raleigh,NC,andDallas,TX,third edition,2008.

Index

A	installing, <mark>225</mark>
actors,317–318	Javaintegration, 224, 250, 262, 265
Erlang, 183	lazyevaluation, 247–250, 265
Io, <mark>89</mark>	leiningen, <mark>226</mark>
Scala,171–175	Lispand, 224, 264, 267
agents,Clojure, <mark>260–261</mark>	lists,231,319
anonymousfunctions	macros,253-254
Clojure, <mark>238–239</mark>	maps,233-234
Erlang, 198	mathematical operations, 226–228
Haskell,285	metadata, <mark>262</mark>
Armstrong, Joe, 181, 183–185	multimethods, 262
arrays,Ruby <mark>,36–37</mark>	patternmatching,320
atoms,Clojure, <mark>258–259</mark>	prefixnotation, 266
acoms, ersjare, 2 00 2 00	programmingmodel, 315
D	protocols,250-253
D	readabilityof, <mark>266</mark>
Bueller,Ferris, see Io	recursion, 243–244
	references, 256–257
C	repl,226
classes	sequences,244-250
Haskell, <mark>299–300</mark>	sets,232-233
Ruby,41–44	strengthsof, 264–266
Scala,147–152	strings, <mark>228–229</mark>
seealso objects	threadstate, 262
Clojure, <mark>20,223–267</mark>	transactionalmemory, 256–257, 318
agents,260–261	typingmodel, <mark>228</mark>
anonymousfunctions, 238–239	vectors,231–232
atoms,258–259	weaknessesof, 266–267
bindings, 236–237	clonemessage, Io, 62
booleans,229-230	codeblocks,Ruby, <mark>39–41</mark>
concurrency, 256–262, 265	collections, see arrays;hashes;lists;
console,226	maps;sets;tuples;vectors
creatorof,239–241	Colmerauer, Alain, 96
destructuring,236	compiledlanguages
expressions, 229–230	Erlangas, 186
forms,228	Prologas, 96
functions, 234–239	Scalaas, 141
futures, 261, 318	concurrency,317–319
infinitesequences, 248–250	Clojure, 256 – 262, 265

CONDITIONS

Erlang, 182–183, 207–213	F
Io, <mark>88-91,93</mark>	filtering,Haskell, <mark>286–287</mark>
Ruby, 58	folding,Haskell, <mark>286–287</mark>
Scala, 139, 155, 171–175, 177	foldLeftmethod,Scala, 164
conditions	forms,Clojure,228
Io, 70-72	forwardmessage,Io,86-88
Scala,141-143	functionalprogramminglanguages,
currying	315–316
Haskell,287–288	Clojureas,241
Scala, 165	concurrencyand, 139
	Erlangas, 185–186
D	Haskellas, 268, 310, 311
datastructures,18	Scalaas, 154–155
decisionconstructs, 18	functions
declarativelanguages,Prologas,95	Clojure, <mark>234–239</mark>
Dekorte, Steve, 60, 72–73	Erlang, 191–194, 198–202
domain-specificlanguages	Haskell,272-274,297-298
withIo.83–86	Ruby,35
withScala,178	Scala,160-166
ducktyping,Ruby,32–34	futures,318
dynamictyping	Clojure, <mark>261</mark>
Clojure, <mark>228</mark>	Io, <mark>90</mark>
Erlang, 191, 194	
Ruby, 33	C
rasy,se	games,Prologfor,133
E	
Erlang, <mark>20,181–222</mark>	- H
-	hashes.Ruby.38-39
actors, 183, 317–318	hashes,Ruby,38–39 Haskell.20.268–312
actors, 183,317–318 anonymousfunctions, 198	Haskell, 20, 268 – 312
actors, 183,317–318 anonymousfunctions, 198 atoms, 188–189	Haskell, <mark>20,268–312</mark> anonymousfunctions,285
actors, 183,317–318 anonymousfunctions, 198 atoms, 188–189 concurrency, 182–183, 207–213	Haskell,20,268–312 anonymousfunctions,285 classes,299–300
actors, 183,317–318 anonymousfunctions, 198 atoms, 188–189 concurrency, 182–183,207–213 controlstructures, 196–198	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292
actors, 183, 317–318 anonymousfunctions, 198 atoms, 188–189 concurrency, 182–183, 207–213 controlstructures, 196–198 creatorof, 181, 183–185	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288
actors, 183,317–318 anonymousfunctions, 198 atoms, 188–189 concurrency, 182–183,207–213 controlstructures, 196–198	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287
actors, 183, 317–318 anonymousfunctions, 198 atoms, 188–189 concurrency, 182–183, 207–213 controlstructures, 196–198 creatorof, 181, 183–185 expressions, 186–188 functions, 191–194, 198–202	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188 functions,191–194,198–202 iteration,199 libraries,220	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188 functions,191–194,198–202 iteration,199 libraries,220 lightweightprocesses,182,220	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functions,272–274,297–298
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188 functions,191–194,198–202 iteration,199 libraries,220	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functions,272–274,297–298 higher-orderfunctions,285–287
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188 functions,191–194,198–202 iteration,199 libraries,220 lightweightprocesses,182,220 linkedprocesses,213–218	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functions,272–274,297–298
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188 functions,191–194,198–202 iteration,199 libraries,220 lightweightprocesses,182,220 linkedprocesses,213–218 lists,188–189,198–205,319	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functions,272–274,297–298 higher-orderfunctions,285–287 lazyevaluation,288–290
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188 functions,191–194,198–202 iteration,199 libraries,220 lightweightprocesses,182,220 linkedprocesses,213–218 lists,188–189,198–205,319 messages,207–213	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functions,272–274,297–298 higher-orderfunctions,285–287 lazyevaluation,288–290 listcomprehensions,281–282
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188 functions,191–194,198–202 iteration,199 libraries,220 lightweightprocesses,182,220 linkedprocesses,213–218 lists,188–189,198–205,319 messages,207–213 patternmatching,189–191,320	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functions,272–274,297–298 higher-orderfunctions,285–287 lazyevaluation,288–290 listcomprehensions,281–282 lists,278–282,319
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188 functions,191–194,198–202 iteration,199 libraries,220 lightweightprocesses,182,220 linkedprocesses,213–218 lists,188–189,198–205,319 messages,207–213 patternmatching,189–191,320 programmingmodel,185–186,315	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functions,272–274,297–298 higher-orderfunctions,285–287 lazyevaluation,288–290 listcomprehensions,281–282 lists,278–282,319 mapping,285–286
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188 functions,191–194,198–202 iteration,199 libraries,220 lightweightprocesses,182,220 linkedprocesses,213–218 lists,188–189,198–205,319 messages,207–213 patternmatching,189–191,320 programmingmodel,185–186,315 reliability,182,183,213–219	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functions,272–274,297–298 higher-orderfunctions,285–287 lazyevaluation,288–290 listcomprehensions,281–282 lists,278–282,319 mapping,285–286 monads,301–307,319
actors,183,317–318 anonymousfunctions,198 atoms,188–189 concurrency,182–183,207–213 controlstructures,196–198 creatorof,181,183–185 expressions,186–188 functions,191–194,198–202 iteration,199 libraries,220 lightweightprocesses,182,220 linkedprocesses,213–218 lists,188–189,198–205,319 messages,207–213 patternmatching,189–191,320 programmingmodel,185–186,315 reliability,182,183,213–219 strengthsof,219–220	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functions,272–274,297–298 higher-orderfunctions,285–287 lazyevaluation,288–290 listcomprehensions,281–282 lists,278–282,319 mapping,285–286 monads,301–307,319 partiallyappliedfunctions,286–288
actors, 183,317–318 anonymousfunctions, 198 atoms, 188–189 concurrency, 182–183,207–213 controlstructures, 196–198 creatorof, 181,183–185 expressions, 186–188 functions, 191–194,198–202 iteration, 199 libraries, 220 lightweightprocesses, 182,220 linkedprocesses, 213–218 lists, 188–189, 198–205,319 messages, 207–213 patternmatching, 189–191,320 programmingmodel, 185–186,315 reliability, 182, 183, 213–219 strengthsof, 219–220 tuples, 188–189	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functions,272–274,297–298 higher-orderfunctions,285–287 lazyevaluation,288–290 listcomprehensions,281–282 lists,278–282,319 mapping,285–286 monads,301–307,319 partiallyappliedfunctions,286–288 patternmatching,320
actors, 183,317–318 anonymousfunctions, 198 atoms, 188–189 concurrency, 182–183,207–213 controlstructures, 196–198 creatorof, 181, 183–185 expressions, 186–188 functions, 191–194,198–202 iteration, 199 libraries, 220 lightweightprocesses, 182,220 linkedprocesses, 213–218 lists, 188–189, 198–205,319 messages, 207–213 patternmatching, 189–191,320 programming model, 185–186,315 reliability, 182, 183, 213–219 strengthsof, 219–220 tuples, 188–189 typing model, 191	Haskell,20,268–312 anonymousfunctions,285 classes,299–300 creatorsof,282–283,291–292 currying,287–288 expressions,270–272 filtering,286–287 folding,286–287 functioncomposition,277–278 functioncomposition,277–278 functions,272–274,297–298 higher-orderfunctions,285–287 lazyevaluation,288–290 listcomprehensions,281–282 lists,278–282,319 mapping,285–286 monads,301–307,319 partiallyappliedfunctions,286–288 patternmatching,320 polymorphism,297–298

HICKEY MONADS

recursion,274-275,279-280	strengthsof,92-93
recursivetypes, 298–299	typingmodel, <mark>64</mark>
starting, <mark>269</mark>	weaknessesof,93–94
strengthsof,309-311	iteration
tuples,275–278	Erlang, 199
types,270-272,294-300	Io,75–77
typingmodel, <mark>269,309</mark>	Scala, 142–145, 161–163
Hickey,Rich, <mark>239–241</mark>	
higher-orderfunctions	, T
Haskell, 285–287	Java
Scala, 160	2
hybridlanguages,Scalaas,135–137	Clojureand, 224, 250, 262, 265
	Scalaand, 136–137, 177
Ţ	K
imperativelanguages,95	knowledgebase,Prolog,96,97
inferences, Prolog, 98–99	miowieagebase,i rolog,oo,or
inheritance	T
Io,63-66	
Scala, 151	lazyevaluation
interactionmodel, 18	Clojure, <mark>265</mark>
interpretedlanguages	Haskell, <mark>288–290</mark>
Ioas,61	lightweightprocesses,Erlang,220
Rubyas,28	linkedprocesses,Erlang,213–218
Io,19,60-94	Lisp,Clojureand,224,264,267
actors,89,317-318	lists,319
assignment,77	Clojure, <mark>231</mark>
clonemessage,62	Erlang, 188–189, 198–205
concurrency,88–91,93	Haskell, <mark>278–282</mark>
conditionalstatements,75–77	Io,68–70
conditions, 70–72	Prolog, 111–116
creatorof, 60, 72 – 73	Scala, 156–157, 161–166
domain-specificlanguageswith,	logicprogramminglanguages,Prolog
83-86	as,96,315
forwardmessage, <mark>86–88</mark>	loops, see iteration
futures,90,318	71. //
inheritance, <mark>63–66</mark>	IVI
installing <mark>,61</mark>	macros,Clojure,253–254
interpretedmodelfor,61	maps
iteration,75–77	Clojure, <mark>233–234</mark>
lists,68-70	Haskell, 285–286
loops, 75–77	Io, <mark>69</mark>
maps, <mark>69</mark>	Scala, 158–159
messages,61,68,79-81	Matsumoto, Yikio (Matz), 26-27
methods,66-68	messages, Io, 61, 68, 79-81
objects,61-66,68	metadata,Clojure, <mark>262</mark>
operators,77–78	metaprogramming,Ruby,48–55
performance,94	$method_missing behavior, Ruby, \textcolor{red}{\bf 50-51}$
programmingmodel, 314	methods, Io, 66-68
prototypes,61,68	mixins,Ruby,44-47
reflection,81	modules,Ruby,44-47,51-55
slotsinobjects, 62,68	monads, Haskell, 301-307, 319

MULTIMETHODS RUBY

multimethods,Clojure, <mark>262</mark>	Sudokuexample, 121-126
mutablestate,controlling,317	tuples, 111–114
	unification, 103–105, 111–114, 320
N	weaknessesof, 133
IN .	protocols,Clojure,250–253
natural-languageprocessing,Prolog	prototypeprogramminglanguages,Io
for, 133	as,61,314-315
	prototypes,Io,68
O	F5F,
object-orientedlanguages, <mark>314</mark>	
Ioas,61	queries, Prolog, 99–101
Rubyas, <mark>29,314</mark>	queries, rolog, 55-101
Scalaas,314	D
objects	1
Io,68	RainMan, see Prolog
seealso classes	ranges,Haskell, <mark>280–281</mark>
Odersky,Martin, <mark>137–138</mark>	recursion
openclasses,Ruby,49–50	Clojure, <mark>243–244</mark>
· · · · · · · · · · · · · · · · · · ·	Haskell, <mark>274–275,279–280</mark>
D	Prolog, 109-111, 114-116
	Scala, 162
partiallyappliedfunctions,Haskell,	recursivetypes,Haskell,298–299
286–288	references, Clojure, 256–257
patternmatching, <mark>320</mark>	reflection,Io, <mark>81</mark>
Erlang, 189–191	Roussel,Phillipe,96
Scala, 168–171	Ruby, 19, 25 – 59
performance	arrays,36–37
Io, <mark>94</mark>	classes,41–44
Ruby, 58	codeblocks,39-41
Peyton-Jones,Simon, <mark>291–292</mark>	concurrency, <mark>58</mark>
polymorphism,Haskell, <mark>297–298</mark>	creatorof, <mark>26–27</mark>
Poppins,Mary, <i>see</i> Ruby	decisionconstructs, 30–32
programminglanguages	functions, 35
installing, <mark>23</mark>	hashes, <mark>38–39</mark>
learning, 17–19, 21–22	historyof <mark>,26</mark>
programmingmodel, <mark>18,313–316</mark>	installing, <mark>28</mark>
Prolog, <mark>19,95–134</mark>	interactiveconsolefor,28
appendrule, 116–119	interpretedmodelfor, 28
capitalizationin, <mark>97</mark>	metaprogramming,48–55
creatorsof,96	method_missingbehavior,50-51
EightQueensexample, 126-131	mixins, 44–47
inferences,98–99	modules,44-47,51-55
knowledgebase,96,97	openclasses,49-50
lists,111-116	performance, 58
mapcoloringexample, 101-103	programmingmodel, <mark>29,314</mark>
math,114-116	runningfromafile,41
patternmatching, 320	scripting, <mark>56</mark>
programmingmodel, 315	spaceshipoperator,45
queries,99–101	strengthsof, <mark>56–58</mark>
recursion, 109–111, 114–116	stringhandling, <mark>29</mark>
schedulingexample, 105–108	timetomarket,57
strengthsof, 132–133	typesafety,58
	-J FJ ,

Scala Yoda

typingmodel, 32–34 weaknesses of, 58	statictyping Haskell, <mark>269</mark>
webdevelopment,57	Scala,143,179
r	strings
S	Clojure, <mark>228–229</mark>
Corls 00 125 100	Ruby'shandlingof, <mark>29</mark>
Scala,20,135–180	stronglytypedlanguages
actors,171–175,317–318	Clojure,228
Anyclass, 159	Erlang, 194
auxiliaryconstructors, 149 classes, 147–152	Haskell,269,309
•	Ruby, 32
collections, 155–166	Scala,141
concurrency,139,155,171–175,	syntacticsugar, <mark>25</mark>
177	.,
conditions, 141–143	Т
creatorof, 137–138	T. I. D.: 100
currying, 165	Tarbox, Brian, 108
domain-specificlanguageswith, 178	threadstate, Clojure, 262
expressions, 141–143	traits,Scala,151
foldLeftmethod, 164	transactionalmemory, Clojure,
functions, 160–166	256–257,318
higher-orderfunctions, 160	Tregunna, Jeremy, <mark>83</mark>
immutablevariables, 155	tuples
inheritance, 151	Erlang, 188–189
iteration, 142–145, 161–163	Haskell, <mark>275–278</mark>
Javaand, 136–137, 177	Prolog, 111-114
lists,156-157,161-166	typesafety,Ruby, <mark>58</mark>
maps, 158–159	typingmodel, <mark>18</mark>
Nothingtype, 159	Clojure, <mark>228</mark>
patternmatching, 168–171, 320	Erlang, 191
programmingmodel, 314, 315	Haskell,309
ranges, 145–147	Io, <mark>64</mark>
recursion, 162	Ruby, 32–34
sets,157-158	Scala,140-141
strengthsof, 177–178	
traits, 151	U
tuples, 146–147	unification, Prolog, 103–105, 111–114,
types, 140–141	320
typingmodel, 143, 179	 -
weaknessesof, 178–179	V
XMLand, 167–168, 170, 178	vectors,Clojure,231–232
Scissorhands, Edward, see Scala	vectors, crojure, 201–202
scripting,Ruby, <mark>56</mark>	7.7.7
semanticWbPrologfor, 133	VV
sequences, Clojure, 244–250	WaltPhilip, <mark>282–283</mark>
sets	webdevelopment,Ruby, <mark>57</mark>
Clojure, 232 – 233	**
Scala, 157–158	X
slotsinobjects,Io,62,68	XML,Scalaand,167–168,170,178
Smith(Agent), see Erlang	, ,
spaceshipoperator, Ruby, 45	Y
Spock, see Haskell	Vol. caa Claiura
эроск, see паsкен	Yoth see Clojure

ThePragmaticBookshelf

 $\label{lem:paperback} A vailable in paper back and DRM-free Books, our titles are here to help you stay on top of your game. The following are in print as of August 2010; be sure to check our website at pragprog.com for new er titles.$

Title	Year	ISBN	Pages
AdvancedRailsRecipes:84NewWystoBuild StunningRailsApps	2008	9780978739225	464
AgileCoaching	2009	9781934356432	248
AgileRetrospectives:MakingGoodTeamsGreat	2006	9780977616640	200
Agile Wb Development with Rails, Third Edition	2009	9781934356166	784
BeginningMacProgramming:Developwith Objective-CandCocoa	2010	9781934356517	300
BehindClosedDoors:SecretsofGreat Management	2005	9780976694021	192
BestofRubyQuiz	2006	9780976694076	304
CocoaProgramming:AQuick-StartGuidefor Developers	2010	9781934356302	450
CoreAnimationforMacOSXandtheiPhone: CreatingCompellingDynamicUserInterfaces	2008	9781934356104	200
CoreData:Apple'sAPIforPersistingDataon MacOSX	2009	9781934356326	256
DataCrunching:SolveEverydayProblems usingJava,Python,andMore	2005	9780974514079	208
DebugIt!Find,Repair,andPreventBugsin Yir Code	2009	9781934356289	232
DeployingRailsApplications:AStep-by-Step Guide	2008	9780978739201	280
DesignAccessibleWbSites:36Keysto CreatingContentforAllAudiencesand Platforms	2007	9781934356029	336
DesktopGIS:MappingthePlanetwithOpen SourceTools	2008	9781934356067	368
$\label{lem:prop:condition} Developing Facebook Platform Applications with \\ Rails$	2008	9781934356128	200
Domain-DrivenDesignUsingNakedObjects	2009	9781934356449	375
EnterpriseIntegrationwithRuby	2006	9780976694069	360
EnterpriseRecipeswithRubyandRails	2008	9781934356234	416
EverydayScriptingwithRuby:forTeams, Testers,and Y u	2007	9780977616619	320
ExpressionEngine2:AQuick-StartGuide	2010	9781934356524	250
FXRuby:CreateLeanandMeanGUIswithRuby	2008	9781934356074	240
FromJavaToRuby:ThingsEveryManager ShouldKnow	2006	9780976694090	160
	Conti	inuedonnextnaae	

Continuedonnextpage

Title	Year	ISBN	Pages
GISforWbDevelopers:AddingWhereto Yu r WbApplications	2007	9780974514093	275
GoogleMapsAPI,V2:AddingWhereto Yi r Applications	2006	PDF-Only	83
Grails:AQuick-StartGuide	2009	9781934356463	200
GroovyRecipes:GreasingtheWheelsofJava	2008	9780978739294	264
Hello,Android:IntroducingGoogle'sMobile DevelopmentPlatform	2010	9781934356562	320
InterfaceOrientedDesign	2006	9780976694052	240
LandtheTechJob Ys ı Love	2009	9781934356265	280
LanguageImplementationPatterns:Create Yir OwnDomain-SpecificandGeneralProgramming Languages	2009	9781934356456	350
LearntoProgram,2ndEdition	2009	9781934356364	240
ManageIt! Yir GuidetoModernPragmatic ProjectManagement	2007	9780978739249	360
Manage Yir ProjectPortfolio:Increase Yir CapacityandFinishMoreProjects	2009	9781934356296	200
MasteringDojo:JavaScriptandAjaxToolsfor GreatWbExperiences	2008	9781934356111	568
MetaprogrammingRuby:ProgramLiketheRuby Pros	2010	9781934356470	240
ModularJava:CreatingFlexibleApplications withOSGiandSpring	2009	9781934356401	260
NoFluffJustStuff2006Anthology	2006	9780977616664	240
NoFluffJustStuff2007Anthology	2007	9780978739287	320
PomodoroTechniqueIllustrated:TheEasyWay toDoMoreinLessTime	2009	9781934356500	144
PracticalProgramming:AnIntroductionto ComputerScienceUsingPython	2009	9781934356272	350
PracticesofanAgileDeveloper	2006	9780974514086	208
PragmaticProjectAutomation:HowtoBuild, Deploy,andMonitorJavaApplications	2004	9780974514031	176
PragmaticThinkingandLearning:Refactor Yir W iwe	2008	9781934356050	288
PragmaticUnitTestinginC#withNUnit	2007	9780977616671	176
PragmaticUnitTestinginJavawithJUnit	2003	9780974514017	160
Pragmatic Vsi nControlUsingGit	2008	9781934356159	200
Pragmatic Vsi nControlusingCVS	2003	9780974514000	176
Pragmatic Vsi nControlusingSubversion	2006	9780977616657	248
ProgrammingClojure	2009	9781934356333	304
ProgrammingCocoawithRuby:Create CompellingMacAppsUsingRubyCocoa	2009	9781934356197	300

Title	Year	ISBN	Pages
ProgrammingErlang:SoftwareforaConcurrent Witl	2007	9781934356005	536
ProgrammingGroovy:DynamicProductivityfor theJavaDeveloper	2008	9781934356098	320
ProgrammingRuby:ThePragmatic Programmers'Guide,SecondEdition	2004	9780974514055	864
ProgrammingRuby1.9:ThePragmatic Programmers'Guide	2009	9781934356081	960
ProgrammingScala:TackleMulti-Core ComplexityontheJavaVirtualMachine	2009	9781934356319	250
Prototypeandscript.aculo.us:Yu NeverKnew JavaScriptCouldDoThis!	2007	9781934356012	448
RailsRecipes	2006	9780977616602	350
Railsfor.NETDevelopers	2008	9781934356203	300
RailsforJavaDevelopers	2007	9780977616695	336
RailsforPHPDevelopers	2008	9781934356043	432
RapidGUIDevelopmentwithQtRuby	2005	PDF-Only	83
ReleaseIt!DesignandDeployProduction-Ready Software	2007	9780978739218	368
SQLAntipatterns:AvoidingthePitfallsof DatabaseProgramming	2010	9781934356555	352
ScriptedGUITestingwithRuby	2008	9781934356180	192
ShipIt!APracticalGuidetoSuccessfulSoftware Projects	2005	9780974514048	224
StripesandJavaWbDevelopmentIsFun Again	2008	9781934356210	375
Test-DriveASP.NETMVC	2010	9781934356531	296
TextMate:PowerEditingfortheMac	2007	9780978739232	208
TheDefinitiveANTLRReference:Building Domain-SpecificLanguages	2007	9780978739256	384
ThePassionateProgrammer:Creatinga RemarkableCareerinSoftwareDevelopment	2009	9781934356340	200
ThoughtWorksAnthology	2008	9781934356142	240
UbuntuKungFu:Tips, Tricks, Hints,andHacks	2008	9781934356227	400
WbDesignforDevelopers:AProgrammer's GuidetoDesignToolsandTechniques	2009	9781934356135	300
iPhoneSDKDevelopment	2009	9781934356258	576

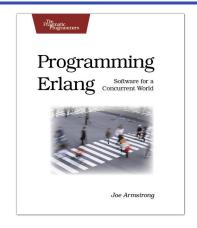
ErlangandScala

ProgrammingErlang

Learnhowtowritetrulyconcurrentprograms—programsthatrunondozensorevenhundredsof localandremoteprocessors. Seehowtowrite high-reliabilityapplications—eveninthefaceof networkandhardwarefailure—usingtheErlang programminglanguage.

Programming Erlang: Software for a Concurrent World

JoeArmstrong (536pages) ISBN:1-934356-00-X.\$36.95 http://pragprog.com/titles/jaerlang



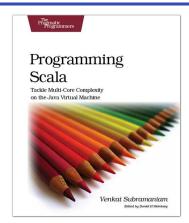
ProgrammingScala

Scalaisanexciting,modern,multi-paradigm languagefortheJVM.Yucanuseittowrite traditional,imperative,object-orientedcode.But youcanalsoleverageitshigherlevelofabstraction totakefulladvantageofmodern,multicore systems. *ProgrammingScala* willshowyouhowto usethispowerfulfunctionalprogramming languagetocreatehighlyscalable,highly concurrentapplicationsontheJavaPlatform.

Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine ViktSubramaniam

(250pages) ISBN:9781934356319.\$34.95

http://pragprog.com/titles/vsscala



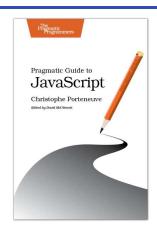
<u>JavaScriptandClojure</u>

<u>PragmaticGuidetoJavaScript</u>

JavaScriptisnowapowerful, dynamiclanguage with a richecosystem of professional-grade development tools, infrastructures, frameworks, and toolkits. Yucan't afford to ignore it—this book will getyou up to speed quickly and painlessly. Presented as two-pagetasks, these JavaScript tips will getyou started quickly and savey out ime.

Pragmatic Guide to JavaScript

ChristophePorteneuve (150pages) ISBN:978-1934356-67-8.\$25.00 http://pragprog.com/titles/pg_js

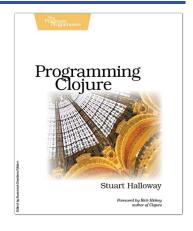


ProgrammingClojure

Clojureisageneral-purposelanguagewithdirect supportforJava,amodernLispdialect,and supportinboththelanguageanddatastructures forfunctionalprogramming. *ProgrammingClojure* showsyouhowtowriteapplicationsthathavethe beautyandeleganceofagoodscriptinglanguage, thepowerandreachoftheJVM,andamodern, concurrency-safefunctionalstyle.Nowyoucan writebeautifulcodethatrunsfastandscaleswell.

Programming Clojure

StuartHalloway (304pages) ISBN:9781934356333.\$32.95 http://pragprog.com/titles/shcloj



MacandSQL

BeginningMacProgramming

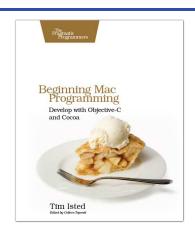
Aimedatbeginningdeveloperswithoutprior programmingexperience. Takesyouthrough concrete, working examples, giving youthecore concepts and principles of development in context soyou will be ready to build the application syou've been imagining. It introduces you to Objective-C and the Cocoaframework in clear, easy-to-understandlessons, and demonstrates how you can use them together towrite for the Mac, as well as the iPhone and iPod.

Beginning Mac Programming: Develop with Objective-C and Cocoa

Tim Isted

(300pages) ISBN:978-1934356-51-7.\$34.95

http://pragprog.com/titles/tibmac



SQLAntipatterns

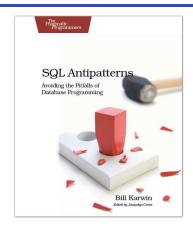
Ifyou'reprogrammingapplicationsthatstoredata, thenchancesareyou'reusingSQL,eitherdirectly orthroughamappinglayer.ButmostoftheSQL thatgetsusedisinefficient,hardtomaintain,and sometimesjustplainwrong.Thisbookshowsyou allthecommonmistakes,andthenleadsyou throughthebestfixes.What'smore,itshowsyou what's behind thesefixes,soyou'llearnalotabout relationaldatabasesalongtheway.

SQL Antipatterns: Avoiding the Pitfalls of Database Programming

BillKarwin

(300pages) ISBN:978-19343565-5-5.\$34.95

http://pragprog.com/titles/bksqla



RubyandRubyontheJavaVM

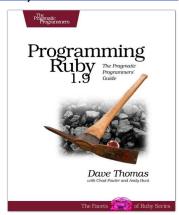
ProgrammingRuby1.9(ThePickaxefor1.9)

The Pickaxebook, named for the too lonthecover, is the definitive reference to this highly-regarded language.

•Up-to-dateandexpandedforRubyversion1.9
•Completedocumentationofallthebuilt-in
classes,modules,andmethods
•Complete
descriptionsofallstandardlibraries
•Learnmore
aboutRuby'swebtools,unittesting,and
programmingphilosophy

Programming Ruby 1.9: The Pragmatic Programmers' Guide

DaveThomaswithChadFowlerandAndyHunt (992pages) ISBN:978-1-9343560-8-1.\$49.95 http://pragprog.com/titles/ruby3



UsingJRuby

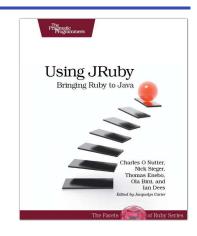
Rubyhastheheart, and Javahasthereach. Wh JRuby, you can bring the best of Rubyinto the world of Java. Whinin 100% Java, JRubyhas Ruby's expressiveness and wide array of open-source libraries—it's an even better Ruby. Wh Using JRuby, the entire JRuby coreteam helps experienced Javade velopers and Rubyists exploit the interoperability of their respective languages. Wh JRuby, you'll be surprised at what's now possible.

Using JRuby: Bringing Ruby to Java

 $Charles O Nutter, Thomas Enebo, Nick Sieger, Ola\\Bini, and Ian Dees$

(300pages) ISBN:978-1934356-65-4.\$34.95

http://pragprog.com/titles/jruby



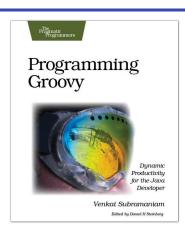
GroovyandGrails

ProgrammingGroovy

ProgrammingGroovy willhelpyoulearnthe necessaryfundamentalsofprogramminginGroovy. WilseehowtouseGroovytodoadvanced programmingtechniques,includingmeta programming,builders,unittestingwithmock objects,processingXML,workingwithdatabases andcreatingyourowndomain-specificlanguages (DSLs).

Programming Groovy: Dynamic Productivity for the Java Developer

WhitSubramaniam (320pages) ISBN:978-1-9343560-9-8.\$34.95 http://pragprog.com/titles/vslg



Grails

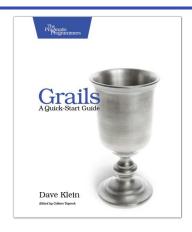
Grailsisafullstackwebdevelopmentframework thatenablesyoutobuildcompleteweb applicationsinafractionofthetimeandwithless codethanotherframeworks.In *Grails:A Quick-StartGuide* ,you'llseehowtouseGrailsby iterativelybuildingaunique,workingapplication. Bythetimewe'redone,you'llhavebuiltand deployedareal,functioningwebsite.

Grails: A Quick-Start Guide

DaveKlein

(200pages) ISBN:9781934356463.\$32.95

http://pragprog.com/titles/dkgrails



ThePragmaticBookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmers tyle and continue to garnera wards and ravereviews. As development gets more and more difficult, the Pragmatic Programmers will be the rewith more titles and products to help you stay on topoly our game.



UsOnline

SevenLanguagesinSevenWeds

http://pragprog.com/titles/btlang

Source code from this book, errata, and other resources. Come give us feedback, too!

Register**t**rUpdates

http://pragprog.com/updates

Be notified when updates and newbooks become available.

JointheCommunity

http://pragprog.com/community

Readour weblogs, join our online discussions, participate in our mailing list, interact withour wiki, and benefit from the experience of other Pragmatic Programmers.

NewandNoteworthy

http://pragprog.com/news

Checkoutthelatestpragmaticdevelopments, newtitles and other offerings.

<u>BuytheBook</u>

IfyoulikedthiseBook,perhapsyou'dliketohaveapapercopyofthebook.It'savailable forpurchaseatourstore: progprog.com/titles/btlong.

ContactUs

OnlineOrders: www.pragprog.com/catalog
CustomerService: support@pragprog.com
Non-EnglishVeins translations@pragprog.com
PragmaticTeaching: academic@pragprog.com
AuthorProposals: proposals@pragprog.com
Contactus: 1-800-699-PROG(+19198473884)